



Synthèse automatique de circuits asynchrones QDI

Anh Vu Dinh Duc

► To cite this version:

Anh Vu Dinh Duc. Synthèse automatique de circuits asynchrones QDI. Autre [cs.OH]. Institut National Polytechnique de Grenoble - INPG, 2003. Français. NNT: . tel-00002937

HAL Id: tel-00002937

<https://theses.hal.science/tel-00002937>

Submitted on 3 Jun 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

[illegible]

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de
l'Ecole Doctorale « **ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE,
TELECOMMUNICATIONS, SIGNAL** »

présentée et soutenue publiquement

par

Anh-Vu DINH-DUC

le 14 mars 2003

Titre :

SYNTHESE AUTOMATIQUE DE CIRCUITS ASYNCHRONES QDI

Directeur de thèse : Marc RENAUDIN

Codirecteur : Laurent FESQUET

JURY

Mme. Anne Mignotte
M. Bruno Rouzeyre
M. Patrice Quinton
M. Marc Renaudin
M. Laurent Fesquet
M. Jean-Pierre Schoellkopf

- , Présidente
- , Rapporteur
- , Rapporteur
- , Directeur
- , Codirecteur
- , Examineur

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

[illegible]

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de
l'Ecole Doctorale « **ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE,
TELECOMMUNICATIONS, SIGNAL** »

présentée et soutenue publiquement

par

Anh-Vu DINH-DUC

le 14 mars 2003

Titre :

SYNTHESE AUTOMATIQUE DE CIRCUITS ASYNCHRONES QDI

Directeur de thèse : Marc RENAUDIN

Codirecteur : Laurent FESQUET

JURY

Mme. Anne Mignotte
M. Bruno Rouzeyre
M. Patrice Quinton
M. Marc Renaudin
M. Laurent Fesquet
M. Jean-Pierre Schoellkopf

- , Présidente
- , Rapporteur
- , Rapporteur
- , Directeur
- , Codirecteur
- , Examineur

Résumé

Contrairement aux circuits synchrones, les circuits asynchrones fonctionnent avec un mécanisme de synchronisation local (sans signal d'horloge). Ils ont montré depuis de nombreuses années leur pertinence vis-à-vis des circuits synchrones grâce à leurs propriétés de robustesse, de faible consommation, de faible bruit et de modularité. Cependant, le manque actuel de méthodes et d'outils de conception est un frein à leur développement. Ce travail de thèse porte sur la définition d'une méthodologie de conception de circuits intégrés asynchrones quasi-insensibles aux délais (QDI). Les circuits QDI font partie de la classe des circuits asynchrones les plus robustes, propriété avantageuse pour les technologies à venir.

La méthode de conception proposée permet d'une part la modélisation dans un langage de haut niveau, et d'autre part la génération de circuits en portes logiques élémentaires et en portes de Muller. Cette méthode a été prototypée par le développement d'un outil de conception automatique de circuits asynchrones TAST (« TIMA Asynchronous Synthesis Tools »). C'est un environnement de conception principalement composé d'un compilateur et d'un synthétiseur offrant la possibilité de générer des circuits asynchrones QDI avec différents modèles de circuits cibles (séquentiel, WCHB, PCHB et PCFB) en partant de descriptions de haut niveau décrites en langage CHP. Le résultat produit par le synthétiseur est une description VHDL de niveau porte qui peut cibler soit une technologie spécifique pour l'asynchrone, soit une bibliothèque de cellules standard (circuits précaractérisés ou FPGAs).

Mots-clés : Circuits asynchrones, méthodologies de conception, synthèse de circuits, circuits quasi-insensibles aux délais, processus concurrents communicants, protocoles de communications, langage CHP, réseaux de Pétri, équations de dépendances, cellules standard, CAO, outils de synthèse.

Abstract

Contrary to the synchronous circuits, the asynchronous circuits operate with a mechanism of local synchronization (without clock signal). Since many years, they showed their relevance with respect to the synchronous circuits thanks to their properties of robustness, low power, low noise and modularity. However, the current lack of design methods and associated tools prevents them from being widely spread. This dissertation deals with a new design methodology for quasi-delay insensitive integrated asynchronous circuits (QDI). The QDI circuits are the most robust class of the practical asynchronous circuits, which is an advantageous property for upcoming technologies.

The suggested design method allows on the one hand to model circuits in a high-level language, and on the other hand to generate circuits using only elementary logical gates and Muller gates. This method was prototyped by the development of an automatic design tool for asynchronous circuits, namely TAST (“TIMA Asynchronous Synthesis Tools”). It is a design environment mainly made up of a compiler and a synthesizer targeting QDI asynchronous circuits with various handshaking protocols (sequential, WCHB, PCHB and PCFB) from high level descriptions (language based on concurrent communicating processes). The result produced by the synthesizer is a VHDL gate netlist which can target either an asynchronous specific technology, or a standard cell library (precharacterized circuits or FPGAs).

Keywords : Asynchronous circuits, design methodologies, circuit synthesis, quasi-delay insensitive circuits, concurrent communicating processes, communication protocols, CHP language, Petri nets, dependency equations, standard cells, CAD, synthesis tools.

A ma femme et à mes parents

Remerciements

Les travaux de cette thèse ont été effectués au sein du Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs (TIMA), sur le site Viallet de l'Institut National Polytechnique de Grenoble (INPG). Je remercie Monsieur Bernard COURTOIS, Directeur du laboratoire, pour m'avoir accueilli dans son laboratoire.

Je voudrais remercier vivement Marc RENAUDIN, Professeur à l'INPG et chef du groupe de recherche CIS au laboratoire TIMA, pour m'avoir accueilli dans son groupe et pour avoir guidé en permanence mes travaux de recherche qui ont abouti à cette thèse. Qu'il trouve toute l'expression de ma sincère gratitude.

Je tiens à remercier profondément mes rapporteurs : Monsieur Bruno ROUZEYRE, Professeur à l'Université de Montpellier II et Monsieur Patrice QUINTON, Professeur à l'IRISA, Université de Rennes I, pour avoir consacré une partie de leur temps afin de juger mon travail.

Je remercie Madame Anne MIGNOTTE, Professeur à l'INSA Lyon et Directrice du laboratoire L3i, de m'avoir fait l'honneur de présider mon jury de thèse.

J'adresse aussi mes remerciements à Monsieur Jean-Pierre SCHOELLKOPF, chef du département ADT à STMicroelectronics Crolles, pour avoir accepté de faire partie du jury.

J'exprime également mes remerciements à Laurent FESQUET, maître de conférence à l'INPG, pour la liberté qu'il m'a donnée dans mon travail et surtout pour l'aide qu'il m'a donnée dans ces derniers temps.

Je tiens aussi à avoir une attention toute particulière pour Monsieur Pierre GENTIL, Directeur de l'école doctorale EEATS qui a accepté que je poursuive mes études à Grenoble au sein du DEA de Microélectronique.

Je remercie chaleureusement tous les membres du groupe CIS pour leur collaboration, leur attitude sympathique et pour l'ambiance amicale de travail. J'exprime particulièrement mes reconnaissances à

- Gilles SICARD pour ses bons conseils et ses encouragements
- Jean-Baptiste RIGAUD, mon voisin de bureau et mon compagnon, qui a répondu à toutes mes questions pendant plus de trois ans
- Dhanistha PANYASAK, ma « compatriote », pour sa gentillesse et son dévouement remarquable qu'elle m'a réservés
- Emmanuel ALLIER (« Manu ») pour son aide dans la correction de ce manuscrit de thèse
- Mohamed ESSALIENNE, Jérôme QUARTANA, Fraïdy BOUESSE, Amine REZZAG, Kamel SLIMANI, João FRAGOSO pour leur bonne humeur permanente, leurs critiques et leurs encouragements.

Je remercie également tous les membres du laboratoire TIMA – actuels et ceux que j'ai rencontrés tout au long de ces années – pour leurs encouragements et pour l'ambiance agréable de

travail, et plus particulièrement, Damien LYONNARD du « bureau d'informations » pour l'aide qu'il m'a apportée et l'équipe administrative du TIMA pour leur gentillesse et leur disponibilité.

Je remercie tout particulièrement mes professeurs Son T. NGUYEN et Viet V. LE de m'avoir donné les facilités pour faire mes débuts dans la recherche et surtout pour avoir cru en moi.

J'aimerais également remercier tous mes amis pour leurs encouragements et leur aide.

Je n'arriverai jamais à remercier assez mes parents, ma sœur, mes frères et mes proches de m'avoir permis de finaliser mon éducation et de m'avoir laissés suivre mon chemin.

Un immense MERCI à ma petite « toto » Trâm pour son soutien inépuisable tout au long de ce travail de longue haleine et pour son support toutes ces années durant lesquelles je me suis exilé loin d'elle. Enfin, une pensée très tendre pour ma petite « Mina » Anh-Thu.

Table des matières

Résumé	iii
Abstract	iv
Remerciements	vii
Table des matières	ix
Liste des figures	xv
Introduction	1
Chapitre 1 Etat de l'art sur la conception de circuits asynchrones	7
1.1 Pourquoi les circuits asynchrones	7
1.1.1 Problèmes des circuits synchrones	7
1.1.2 Principes des circuits asynchrones	8
1.1.2.1 Mode de fonctionnement asynchrone	8
1.1.2.2 Problèmes	9
1.2 Concepts de base	10
1.2.1 Canaux de communication	10
1.2.2 Représentation des données	11
1.2.2.1 Codage « données groupées »	11
1.2.2.2 Codage insensible aux délais	11
1.2.3 Protocoles de communication	13
1.2.3.1 Protocoles 2 phases	14
1.2.3.2 Protocoles 4 phases	14
1.2.4 Implémentation du protocole : utilisation de la porte de Müller	14
1.2.5 Aléas	15
1.2.5.1 Aléas combinatoires fonctionnels	16
1.2.5.2 Aléas combinatoires logiques	16
1.2.5.3 Aléas séquentiels	19
1.2.5.4 Conclusion	19
1.2.6 Modèles de délais, de circuits et d'environnement	19
1.2.6.1 Modèles de délais	19
1.2.6.2 Modes de fonctionnement	20
1.2.7 Catégorie de circuits asynchrones	20

1.2.7.1	Circuits insensibles aux délais (DI, « Delay Insensitive »).....	21
1.2.7.2	Circuits quasi-insensibles aux délais (QDI, « Quasi-Delay Insensitive »)	22
1.2.7.3	Circuits indépendants de la vitesse (SI, « Speed Independence »)	22
1.2.7.4	Circuits micropipelines	23
1.2.7.5	Circuits de Huffman.....	24
1.2.7.6	Conclusion	24
1.3	Méthodologies de conception des circuits asynchrones	25
1.3.1	Méthodologies de spécification	25
1.3.1.1	Méthodes basée sur un langage de description	26
1.3.1.2	Méthodes basée sur un graphe	26
1.3.2	Méthodologies de synthèse	27
1.3.2.1	Synthèse dirigée par la syntaxe.....	27
1.3.2.2	Synthèse logique	28
1.3.2.3	Synthèse par compilation.....	29
1.3.3	Méthodologies et outils de synthèse actuelles des circuits asynchrones	29
1.3.3.1	Méthode Tangram.....	29
1.3.3.2	Méthode Balsa.....	30
1.3.3.3	Méthode de Caltech	31
1.3.3.4	Méthode NCL	33
1.3.3.5	Méthode basée sur les STG (Petrify)	35
1.3.3.6	Méthode basée sur ASM (Minimalist).....	37
1.4	Conclusion	39
Chapitre 2	Spécification synthétisable de circuits asynchrones	41
2.1	Langage CHP	42
2.2	Représentation intermédiaire d'un circuit asynchrone.....	42
2.2.1	Réseau de Petri (« Petri Net »).....	42
2.2.2	Graphe de flot de données (« Data Flow Graph »)	43
2.2.3	Combinaison de PN-DFG	44
2.3	DTL : spécification synthétisable de circuits asynchrones	45
2.3.1	Analyse des formes synthétisables.....	45
2.3.1.1	Représentation d'un circuit asynchrone.....	46
2.3.1.2	Mémoire causée par une variable locale	47
2.3.1.3	Mémoire causée par l'opérateur séquentiel	47
2.3.1.4	Mémoire causée par l'opérateur parallèle.....	47
2.3.1.5	Initialisation d'un système	48
2.3.2	Règles DTL	48

2.3.3	Transformation d'un programme en DTL.....	49
2.3.3.1	Extraction des variables de mémorisation.....	49
2.3.3.2	Extraction des opérateurs séquentiels	52
2.3.4	Conclusion sur la spécification DTL.....	55
2.4	Conclusion.....	56
Chapitre 3	Modèle des circuits cibles	57
3.1	Pipeline asynchrone.....	57
3.1.1	Définitions.....	58
3.1.1.1	Mécanisme de pipeline.....	58
3.1.1.2	Performance du pipeline.....	59
3.1.2	Pipeline linéaire vs pipeline non-linéaire	59
3.1.2.1	Pipeline linéaire.....	59
3.1.2.2	Pipeline non-linéaire : fourche et convergence	59
3.1.2.3	Problèmes associés au traitement des fourches et convergences	61
3.2	Travaux antérieurs.....	63
3.3	Modèle des circuits cibles	64
3.3.1	Types de pipeline linéaire de base.....	65
3.3.1.1	Séquentiel	65
3.3.1.2	WCHB (« Weak Condition Half-Buffer »)	65
3.3.1.3	PCHB (« PreCharge Half-Buffer »).....	66
3.3.1.4	PCFB (« PreCharge Full-Buffer »)	67
3.3.2	Structure du circuit cible	68
3.3.2.1	Structures implémentées en séquentiel	69
3.3.2.2	Structures implémentées en WCHB.....	71
3.3.2.3	Structures implémentées en PCHB	72
3.3.2.4	Structures implémentées en PCFB.....	73
3.4	Conclusion.....	74
Chapitre 4	Génération d'une forme indépendante des protocoles et de la technologie	75
4.1	Flot de synthèse de circuits QDI	76
4.2	Du CHP aux réseau de Petri et graphes de flot de données	77
4.2.1	Expressions.....	78
4.2.2	Gardes.....	78
4.2.3	Instructions	78
4.2.4	Opérateur de séquence	79
4.2.5	Opérateur de parallélisme.....	79
4.2.6	Opérateur de sélection.....	79

4.2.7	Opérateur de répétition.....	80
4.3	Vérification DTL du réseau de Petri	80
4.3.1	Vérification de l'initialisation	80
4.3.2	Vérification de l'accès séquentiel	80
4.3.3	Vérification de l'accès parallèle.....	81
4.3.4	Vérification du contexte.....	81
4.3.5	Conclusion sur la vérification DTL	81
4.4	Transformation de réseaux de Petri DTL.....	82
4.4.1	Expansion des structures de choix	82
4.4.2	Expansion des types de données	83
4.5	Génération des équations de dépendances	83
4.5.1	Définition	83
4.5.1.1	Equations de dépendances au niveau des canaux	85
4.5.1.2	Equations de dépendances au niveau des signaux	86
4.5.2	Equations de dépendances des structures de base du réseau de Petri	87
4.5.2.1	Equations de dépendances de la structure place/transition/place.....	88
4.5.2.2	Equations de dépendances de la structure de divergence en « ET ».....	88
4.5.2.3	Equations de dépendances de la structure de convergence en « ET »	89
4.5.3	Génération des équations de dépendances d'un réseau de Petri quelconque.....	90
4.5.4	Génération des équations de dépendances : une solution alternative	94
4.5.5	Optimisation des équations de dépendances	96
4.5.5.1	Règle 1	97
4.5.5.2	Règle 2	97
4.5.5.3	Exemple	97
4.5.6	Conclusion sur les équations de dépendances.....	98
4.6	Conclusion	98
Chapitre 5	Synthèse de circuits QDI.....	101
5.1	Génération des circuits QDI en utilisant des cellules génériques	104
5.1.1	Développement des équations de dépendances en introduisant le protocole de communication.....	104
5.1.1.1	Protocole séquentiel	104
5.1.1.2	Protocole WCHB	106
5.1.2	Génération de la netlist de portes.....	108
5.1.2.1	Test de validité.....	109
5.1.2.2	Initialisation de canaux	110
5.1.2.3	Implémentation QDI des gardes	110

5.1.2.4	Implémentation QDI des fonctions de calcul	115
5.2	Un exemple de synthèse	117
5.3	Optimisation et projection technologique	119
5.3.1	Contraintes QDI	120
5.3.1.1	Aléa	120
5.3.1.2	Fourche isochrone	121
5.3.2	Optimisation logique	121
5.3.3	Décomposition : situation du problème	122
5.3.4	Projection technologique : situation du problème	123
5.4	Conclusion	124
Chapitre 6	Synthétiseur de circuits QDI : TAST	127
6.1	Introduction	127
6.2	Interface utilisateur	129
6.3	Module parseur syntaxique et analyseur contextuel du langage CHP	131
6.4	Module translateur en réseau de Pétri	132
6.5	Module vérificateur de règles DTL	134
6.5.1	Vérification des accès séquentiels et parallèles	134
6.5.2	Vérification des dépendances et de l'initialisation	134
6.6	Module générateur des équations de dépendances et logiques	135
6.7	Module optimiseur	135
6.8	Module gestionnaire du réseau de portes (<i>netlist</i>)	136
6.9	Module générateur de rapports	136
6.10	Une session typique de travail	136
6.11	Conclusion	141
Conclusion		143
Bibliographie		147
Annexe		159
A)	Système de numération	159
1)	Nombre non signé	159
2)	Nombre signé	159
B)	Syntaxe et sémantique du langage CHP	160
1)	Types de données	160
a)	Type de données non signé	160
b)	Type de données signé	160
2)	Conversion de type de données	161
3)	Opérateurs	161

4)	Structure de contrôle	161
5)	Modélisation structurelle de programme	162
6)	Exemple	163

Liste des figures

Figure 1	Flot de conception de l’outil TAST.....	2
Figure 2	Notion de canal de communication.....	11
Figure 3	Codages les plus courants	12
Figure 4	Codage « 1-parmi-N ».....	12
Figure 5	Protocole 4 phases « données groupées »	13
Figure 6	Protocole 2 phases.....	14
Figure 7	Protocole 4 phases.....	14
Figure 8	Symbole et spécifications de la porte de Muller à deux entrées	15
Figure 9	Réalisations de la porte de Muller.....	15
Figure 10	Exemple de porte de Muller dissymétrique [RIGA02]	15
Figure 11	Aléas statiques causés par des portes logiques « AND » « OR »	17
Figure 12	Exemple d’aléas logiques statiques : SIC	17
Figure 13	Exemple d’aléas logiques statiques : MIC	17
Figure 14	Aléas dynamiques	18
Figure 15	Exemple d’aléas dynamiques logiques : MIC.....	18
Figure 16	Différentes classes de circuits asynchrones.	21
Figure 17	Equivalence entre les modèles QDI et SI.....	22
Figure 18	Structure de base des circuits micro pipelines.	23
Figure 19	Structure micro pipeline avec traitement et registre.....	23
Figure 20	Méthode de Tangram	30
Figure 21	Méthode Balsa.....	31
Figure 22	Méthode de Caltech.....	32
Figure 23	Fonctionnement d’une porte à seuil ($M \leq N$).....	33
Figure 24	Motif de base de circuits généré par la méthode NCL	33
Figure 25	Synthèse dans la méthode NCL	34
Figure 26	Flot de conception de la méthode STG	35
Figure 27	Porte de Muller et son environnement	36
Figure 28	Implémentation en utilisant un C élément.....	36
Figure 29	Exemple de la spécification à mode rafale.....	37
Figure 30	Schéma de la machine auto-synchronisée	38

Figure 31	Exemple de réseau de Petri	43
Figure 32	Graphe de flot de données : $x := a + b * c * d + 1$	44
Figure 33	Représentation intermédiaire de circuits asynchrones	44
Figure 34	Modèle FSM d'un circuit asynchrone/synchrone	46
Figure 35	Modélisation d'un élément mémoire comme un registre.....	50
Figure 36	Modélisation d'un élément mémoire comme une variable tournante.....	50
Figure 37	Exemple d'une mémoire	51
Figure 38	Extraction de la mémoire : utilisation de registre	52
Figure 39	Extraction de la mémoire : utilisation de processus tournant	52
Figure 40	Exemple de la décomposition de processus.....	54
Figure 41	Autre solution pour la décomposition de processus	55
Figure 42	Pipeline synchrone	58
Figure 43	Pipeline asynchrone	58
Figure 44	Pipeline non-linéaire : fourche.....	60
Figure 45	Pipeline non-linéaire : convergence.....	60
Figure 46	Pipeline non-linéaires : démultiplexage.....	60
Figure 47	Pipeline non-linéaires : multiplexage.....	61
Figure 48	Pipeline de fourche.....	62
Figure 49	Pipeline de convergence.....	63
Figure 50	Protocole séquentiel : schéma et protocole de communication en STG.....	65
Figure 51	WCHB : schéma et son protocole de communication en STG	65
Figure 52	Implémentation du « demi-buffer » double-rails en protocole WCHB	66
Figure 53	PCHB : modèle schématique et son protocole de communication en STG.....	67
Figure 54	Implémentation du « demi-buffer » double-rails en protocole PCHB.....	67
Figure 55	PCFB : modèle schématique et protocole de communication en STG.....	68
Figure 56	Implémentation du « buffer » double rails avec le protocole PCFB.....	68
Figure 57	Implémentation de circuits asynchrones QDI avec a) buffers à la sortie et b) buffers en entrée	69
Figure 58	Pipeline de fourche en protocole séquentiel	70
Figure 59	Pipeline de fourche en protocole séquentiel : avec les blocs de calcul.....	70
Figure 60	Pipeline de convergence en protocole séquentiel	70
Figure 61	Pipeline de démultiplexage en protocole séquentiel	71
Figure 62	Pipeline de multiplexage en protocole séquentiel.....	71
Figure 63	Pipeline de fourche et pipeline de convergence en protocole WCHB.....	72
Figure 64	Pipeline de démultiplexage et pipeline de multiplexage en protocole WCHB.....	72
Figure 65	Pipeline de fourche et pipeline de convergence en protocole PCHB	73

Figure 66	Pipeline de démultiplexage et pipeline de multiplexage en protocole PCHB.....	73
Figure 67	Pipeline de fourche et pipeline de convergence implémentées en protocole PCFB .	73
Figure 68	Pipeline de démultiplexage et pipeline de multiplexage en protocole PCFB	74
Figure 69	Flot détaillé de la synthèse de circuits asynchrones QDI.....	76
Figure 70	Exemple d'un graphe de flot de données $x := a + b * c * d + 1$	78
Figure 71	Représentation en PN-DFG d'une instruction $S !x+y$	79
Figure 72	Représentation en réseau de Petri de l'opérateur de séquence.....	79
Figure 73	Représentation en réseau de Petri de l'opérateur de parallélisme.....	79
Figure 74	Représentation en réseau de Petri de l'opérateur de sélection	79
Figure 75	Représentation en réseau de Petri de l'opérateur de répétition	80
Figure 76	Expansion des structures de choix du réseau de Petri.....	82
Figure 77	Expansion le réseau de Petri sous forme de digits	83
Figure 78	Exemple de multi-rails en langage CHP et son réseau de Petri	86
Figure 79	Réseau de Petri de la structure séquentielle	88
Figure 80	Réseau de Petri de la structure de divergence en « ET »	89
Figure 81	Réseau de Petri de la structure de convergence en « ET ».....	90
Figure 82	Réseaux de Petri du multiplexeur.....	92
Figure 83	Synthèse bas niveau	101
Figure 84	Première solution d'optimisation et de projection technologique.....	103
Figure 85	Deuxième solution d'optimisation et de projection technologique.....	103
Figure 86	Implémentation de circuits avec le protocole séquentiel	105
Figure 87	Implémentation du multiplexeur en protocole séquentiel (optimisée).....	106
Figure 88	Implémentation de circuits en protocole WCHB	107
Figure 89	Implémentation du multiplexeur en protocole WCHB	108
Figure 90	Algorithme à 2-niveaux pour l'opérateur d'égalité.....	109
Figure 91	Test de validité d'un canal de communication.....	110
Figure 92	Circuit d'initialisation pour des buffers WCHB	110
Figure 93	Implémentation QDI pour le test d'égalité.....	111
Figure 94	Implémentation non-QDI du test d'inégalité	111
Figure 95	Test d'inégalité entre une variable et une constante	112
Figure 96	Implémentation QDI pour le test d'inégalité.....	112
Figure 97	Exemple avec l'opérateur de sonde.....	113
Figure 98	Implémentation de l'opérateur booléen « NOT »	113
Figure 99	Exemple d'opérateurs booléens dans des gardes	114
Figure 100	Problème d'acquiescement des signaux intermédiaires.....	114
Figure 101	Renforcement des gardes imbriquées.....	115

Figure 102	Implémentation QDI d'un exemple de conversion de type	115
Figure 103	Implémentation QDI de l'opérateur « NOT ».....	116
Figure 104	Opérateur de calcul « AND ».....	116
Figure 105	Opérateur de calcul « OR ».....	116
Figure 106	Opérateur de calcul « XOR ».....	117
Figure 107	Exemple de la génération de la netlist de portes génériques.....	117
Figure 108	Implémentation en protocole WCHB de l'exemple de la Figure 107.....	119
Figure 109	Exemple d'optimisation logique	121
Figure 110	Problème de décomposition des circuits QDI.....	123
Figure 111	Principe général de la projection technologique	124
Figure 112	Flot de synthèse implémenté dans TAST	128
Figure 113	Interface graphique de l'utilisateur	130
Figure 114	Interaction dans le module parseur du langage CHP	132
Figure 115	Construction des réseaux de Pétri	133
Figure 116	Exemple d'un sélecteur.....	137
Figure 117	Netlist VHDL de portes	140
Figure 118	Schéma de portes du circuit Sélecteur	141
Figure 119	Exemple de modélisation structurelle : code CHP et schéma correspondant.....	163

Introduction

Motivation et contexte du travail

L'évolution de l'industrie des circuits intégrés durant la dernière décennie a été tellement rapide, qu'il est maintenant possible d'intégrer plusieurs systèmes complexes sur une seule puce. Cette évolution vers des niveaux d'intégration de plus en plus élevés est motivée par le besoin de systèmes plus performants, légers, compacts et consommant un minimum de puissance. Dans de telles circonstances, c'est sans doute l'arrivée d'obstacles majeurs à la logique synchrone (distribution d'horloge, consommation, bruit, modularité) qui a permis à la logique asynchrone de gagner en popularité ces dernières années. Solutions naturelles et viables aux limitations liées à la présence de synchronisation globale, les circuits asynchrones ont ouvert la voie vers une nouvelle approche de conception de circuits complexes à grande échelle. Les circuits asynchrones ont montré leurs potentiels intéressants dans plusieurs domaines d'application : les conceptions de microprocesseurs, de cartes à puce, de circuits à faible consommation et de circuits à faible émission électromagnétique.

Malgré les avantages et le potentiel des circuits asynchrones, de nombreux facteurs expliquent pourquoi les flots de conception asynchrone ne brillent que par leur absence – ou presque. Tout d'abord, le fait que très peu d'applications industrielles en asynchrone aient vu le jour jusqu'à présent souligne bien le fait que la motivation industrielle pour l'asynchrone n'en est qu'à ses débuts. Or le développement d'un outil complet de synthèse asynchrone est une entreprise conséquente. Elle nécessite un apport important de la part de l'industrie afin de passer des méthodologies complexes inhérentes à la recherche, à un flot complet exploitable à grande échelle. Par ailleurs, la complexité des calculs mis en jeu dans un tel outil constitue elle aussi un obstacle majeur, d'autant plus que les compétences réutilisables depuis le domaine du synchrone sont limitées.

D'autre part, la conception VLSI numérique synchrone a atteint un niveau de maturité et d'automatisation tel, que le secteur n'est pas prêt à un retour en arrière en termes de méthodologie (outils) et de productivité. La conception des circuits asynchrones reste encore le domaine de spécialistes utilisant des outils faiblement automatisés. Sa diffusion dans le secteur industriel ne sera possible tant que les outils des circuits asynchrones n'auront pas atteint le même niveau d'automatisation que ceux des circuits synchrones, ni tant que des ingénieurs n'auront pas été formés pour leur utilisation.

Dans ce contexte, le travail de thèse s'est focalisé dans deux directions : un travail au niveau méthodologique d'une part et le développement d'un outil de CAO d'autre part. Ces deux études ont été évidemment réalisées conjointement. La première concerne l'étude d'une méthode de synthèse permettant de générer des circuits asynchrones sans aléa de type quasiment insensible aux délais. La seconde est un prototype de cette méthode : il s'agit de développer un environnement logiciel pour la conception des circuits asynchrones.

Une approche de synthèse des circuits QDI

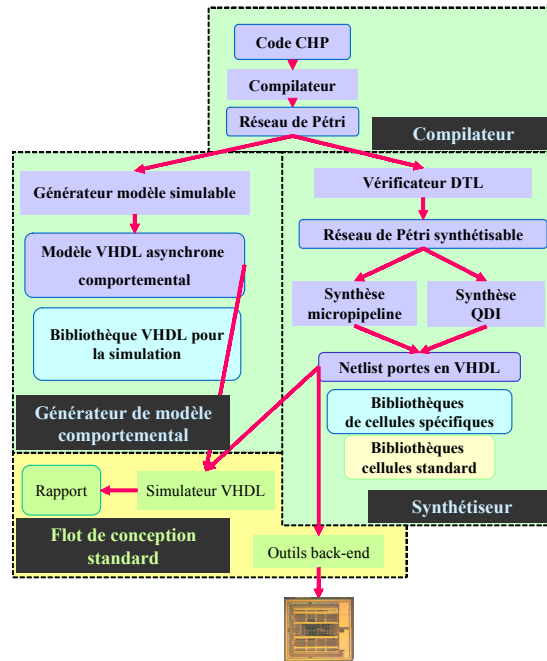


Figure 1 Flot de conception de l'outil TAST

Le projet TAST, acronyme de « TIMA Asynchronous Synthesis Tools », est un environnement de conception dédié à la synthèse de circuits asynchrones. Il est développé au sein du groupe de recherche CIS (« Concurrent Integrated Systems ») du Professeur Marc Renaudin au laboratoire TIMA. Pour fournir des outils faciles à utiliser pour la conception de circuits intégrés logiques asynchrones, TAST dont le flot est représenté Figure 1 peut être considéré comme un compilateur multi source (perspective), multi cible, organisé autour d'un format intermédiaire basé sur les réseaux de Petri et des graphes de flot de données. Il prend en entrée des descriptions de haut niveau de circuits asynchrones en langage CHP (« Communicating Hardware Processes ») évolutif. Pour la sortie, TAST offre la possibilité de cibler plusieurs formats de sorties (description comportementale en VHDL, langage C, description matérielle au niveau porte en VHDL).

Le *front-end* de l'outil TAST est en fait un compilateur permettant la compilation et la transformation du code source CHP des systèmes asynchrones en un ensemble de réseaux de Petri et de graphes de flot de données. Le langage CHP utilisé est une version enrichie de celle initialement développée par Alain Martin à Caltech. Il permet de décrire des circuits à base de processus concurrents communicants. Le réseau de Petri offre une structure de contrôle et de choix mixte : séquentielle et parallèle – qui tient lieu de sémantique opérationnelle de la spécification, alors que le graphe de flot de données permet d'exposer les dépendances de données.

Le *back-end* de l'outil TAST s'appuie sur le format intermédiaire pour générer plusieurs modèles de sorties. Tout d'abord, un modèle comportemental équivalent en VHDL est généré afin de pouvoir vérifier par simulation la correction fonctionnelle du circuit à l'aide d'outils industriels standard. Parallèlement, un modèle équivalent en langage C est aussi généré pour une validation de la description CHP par la simulation comportementale accélérée sous Unix/Linux. La synthèse de circuits asynchrones est basée sur la spécification DTL (« Data Transfer Level ») qui est l'ensemble des règles garantissant que les circuits décrits en langage CHP sont synthétisables. Deux classes de circuits asynchrones sont ciblées : la classe de circuits quasiment insensibles aux délais (QDI) et micropipeline. Pour les circuits asynchrones de type QDI, une implémentation logique au niveau porte sous forme d'une *netlist* en VHDL est générée pour la partie contrôle ainsi que pour la partie

flot de données. Pour ce faire, les canaux de communication utilisent un codage insensible aux délais et un protocole de communication 4 phases. Quant à eux, les circuits de type micropipeline sont générés en séparant les deux parties : la partie flot de données implémentée au niveau VHDL RTL et la partie contrôle implémentée par un circuit asynchrone logique au niveau porte. L'interconnexion entre les deux parties suit l'implémentation définie par Sutherland.

Le logiciel TAST a fait l'objet d'une démonstration à DATE 2002 et de « travaux pratiques » à la conférence ASYNC 2002 et à l'école d'été ACiD 2002.

Le travail de cette thèse constitue le flot principal du projet TAST : synthétiser des circuits asynchrones de type QDI à partir de leur spécification décrite en langage de haut niveau CHP ([DINH02][DINH02b]). Pour ce faire, il faut d'abord étudier et développer les formes intermédiaires adéquates pour représenter les circuits asynchrones à concevoir. En vue de séparer le *front-end* et le *back-end* de l'outil TAST, cette forme intermédiaire doit non seulement être capable de représenter exactement toutes les structures de contrôle et toutes les dépendances de données du programme initial, mais en plus permettre à l'avenir une validation par vérification formelle. Cette étape constitue un premier pas dans le flot de synthèse : le compilateur du langage CHP.

La deuxième contribution de cette thèse est la méthode de synthèse de circuits asynchrones QDI ([DINH02c][DINH02d]). Cette dernière est basée sur les règles DTL – conditions synthétisables de circuits asynchrones. Le modèle des circuits cibles développés est basé sur les modèles les plus robustes initiés par Andrew Lines. Dans un premier temps, les équations de dépendances des sorties du circuit sont générées : elles explicitent les dépendances des sorties en fonctions des entrées. Ces équations sont indépendantes du protocole de communication et de la technologie cible. Ces équations de dépendances sont ensuite développées avec les options de synthèses afin de donner les équations logiques. Ces dernières peuvent être vues comme une image matérielle sans aléa du circuit généré sous forme d'une *netlist* de portes génériques. Pour compléter la synthèse de circuits asynchrones QDI, une bibliothèque spécifique à la technologie est prise en compte : il faut optimiser et « projeter » la *netlist* de portes. Le principal inconvénient de la conception de circuits asynchrones est qu'elle requière une maîtrise complète des aléas. Comme on peut faire apparaître des aléas dans n'importe quelle phase de la conception, il faut apporter une attention particulière à ce problème à toutes les étapes, de la spécification à l'implémentation logique. Il faut donc s'assurer que les techniques de réalisation choisies sont exemptes d'aléas. La réalisation de cette méthode de synthèse est prototypée par un outil final dans le flot de TAST : le synthétiseur de circuits QDI.

Plan du manuscrit

Le manuscrit est organisé comme suit.

Le premier chapitre présente tout d'abord le mode de fonctionnement asynchrone et ses avantages aussi bien que ses inconvénients par rapport au mode synchrone. Les concepts de base des circuits asynchrones sont ensuite définis. Le transfert d'informations entre les blocs du circuit est réalisé à l'aide des canaux de communication. La communication sur les canaux est localement synchronisée grâce au codage des données et au protocole de type « poignée de main » choisis. Pour implémenter un tel protocole de communication, la notion de porte de Muller est présentée. Le mécanisme de communication asynchrone a la propriété principale de rendre le système indépendant du temps de calcul des opérations de chaque bloc. Cela signifie que les blocs du circuit asynchrone sont « pilotés » par les événements sur leurs canaux. Par conséquent, les circuits asynchrones doivent être exemptes d'aléas. Les hypothèses simplificatrices pour la conception de circuits asynchrones sans aléa sont faites sur le modèle de délais appliqués aux portes et aux interconnexions, ainsi qu'au mode

de fonctionnement (comportement du circuit vis-à-vis de son environnement). Ces hypothèses caractérisent les circuits asynchrones et permettent d'en différencier les principales classes. Enfin, une multitude de méthodologies de conception de circuits asynchrones et les outils correspondants sont représentés. Principalement, deux méthodes de spécification de circuits asynchrones sont étudiées : la première basée sur un langage de description de haut niveau et la seconde basée sur un graphe. Les méthodes de synthèse de circuits asynchrones se scindent en trois groupes : la synthèse dirigée par la syntaxe, la synthèse logique et la synthèse par la compilation. Le chapitre se termine avec les discussions sur les différentes méthodologies effectivement dominantes.

Le deuxième chapitre introduit ensuite la méthode de spécification de circuits asynchrones vis-à-vis de la synthèse et de la simulation. Premièrement, le langage CHP est présenté : c'est un langage de description de haut niveau qui permet de décrire des circuits asynchrones basés sur des processus concurrents communicants. La syntaxe complète de ce langage est donnée en annexe. Nous proposons ensuite une forme de représentation intermédiaire de circuits asynchrones. Elle est basée sur le réseau de Petri et le graphe de flot de données. La spécification synthétisable de circuits asynchrones est proposée sous forme de règles DTL en étudiant l'implémentation des éléments de mémorisation des circuits ([DINH02c][DINH02d]). Un circuit asynchrone est synthétisé si sa description en langage CHP est conforme aux règles DTL. Sinon, la spécification doit être transformée en DTL : différentes transformations sont discutées.

Le modèle de circuits cibles sera exposé dans le troisième chapitre. Les techniques de pipeline sont étudiées. Ces techniques – les pipelines linéaires et les pipelines non linéaires – permettent une implémentation plus performante en vitesse des circuits asynchrones complexes. L'intérêt s'est naturellement porté sur les techniques de pipeline permettant de réaliser les circuits les plus robustes : les pipelines QDI. Suite au choix des pipelines, les différentes structures du circuit cible allant du niveau simple au niveau complexe sont développées : linéaire, fourche, convergence, multiplexage et démultiplexage.

Le quatrième chapitre propose une méthode de synthèse de circuits asynchrones QDI basée sur les réseaux de Petri conforme à la spécification DTL [DINH02c][DINH02d]. La compilation de la description de circuits en langage CHP vers les réseaux de Petri et les graphes de flot de données est représentée. L'ensemble de ces réseaux de Petri est ensuite vérifié à la spécification DTL. Les équations de dépendances du circuit sont générées à partir des réseaux de Petri. Elles sont indépendantes du protocole de communication et de la technologie cible. Ces équations de dépendances permettent d'exprimer les dépendances des sorties du circuit en fonctions des entrées. Le chapitre se termine avec les optimisations proposées sur ces équations.

Grâce aux équations de dépendances générées (voir chapitre 4), le cinquième chapitre présente ensuite la synthèse de bas niveau avec prise en compte du protocole de communication et de la technologie cible. Les équations de dépendances sont développées dans un premier temps avec le protocole de communication et le modèle de circuit choisi. L'implémentation QDI de plusieurs opérateurs est également présentée. Ces implémentations permettent de générer un réseau de portes (*netlist*) du circuit avec une bibliothèque de cellules génériques. Cette *netlist* est optimisée et finalement « projetée » grâce à une bibliothèque spécifique à la technologie adoptée. Pour garantir la correction de fonctionnement du circuit contre les aléas, les techniques d'optimisation, de décomposition et de projection technologique doivent non seulement être exemptes d'aléas, mais encore assurer le respect des fourches isochrones.

Le dernier chapitre est consacré au développement d'un outil de CAO TAST, un prototype de la méthode proposée ([DINH02][DINH02b]). Cet outil de CAO offre un environnement de

travail facile d'utilisation, permettant au concepteur de concevoir des circuits asynchrones QDI avec les différentes options de synthèse et de comparer leurs résultats. Ce chapitre présente également les algorithmes de différents modules du logiciel TAST. Pour terminer, une session typique de travail est illustrée avec un exemple de circuit.

Enfin, la partie « Conclusion » effectue un bilan du travail effectué et propose des perspectives de travail.

Chapitre 1

ETAT DE L'ART SUR LA CONCEPTION DE CIRCUITS ASYNCHRONES

1.1 Pourquoi les circuits asynchrones

La conception de la plupart des circuits intégrés logiques est facilitée par deux hypothèses fondamentales : les signaux manipulés sont binaires et le temps est discrétisé. La binarisation des signaux permet une implémentation électrique simple et offre un cadre de conception maîtrisé grâce à l'algèbre de Boole. La discrétisation du temps permet quant à elle de s'affranchir des problèmes de rétroactions et/ou boucles combinatoires, ainsi que des fluctuations électriques transitoires. Cependant, un système fonctionnant sans ces hypothèses peut obtenir de meilleurs résultats. Les circuits asynchrones conservent un codage discret des signaux mais ne fait pas l'hypothèse que le temps est discrétisé. Ils définissent ainsi une classe de circuits beaucoup plus large car leur contrôle peut être assuré par tout autre moyen alternatif à l'horloge unique des circuits synchrones.

1.1.1 Problèmes des circuits synchrones

La plupart des systèmes numériques modernes sont synchrones. Ils sont organisés autour d'un signal d'horloge global, et les activités du système sont séquencées par ce signal. Tous les éléments du système évoluent ensemble lors de l'occurrence d'un événement d'horloge. 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 sont donc synchronisés dans le temps.

Grâce à la simplicité de la centralisation du contrôle de cette approche, les systèmes synchrones ont dominé l'industrie microélectronique depuis des années. Cependant, quand les composants deviennent plus petits et plus rapides et quand les systèmes deviennent plus complexes et concurrents, l'approche synchrone devient de plus en plus contraignante. Il existe de nombreuses difficultés pour la conception des systèmes synchrones.

- Distribution d'horloge : Si le signal d'horloge n'est pas uniformément distribué, les éléments du circuit peuvent recevoir l'horloge à des instants différents (« skew ») et le circuit peut ne pas fonctionner. Cependant, le problème de distribution d'horloge peut actuellement être éliminé de

deux façons. La première est de diminuer la fréquence d'horloge pour assurer la condition de bon fonctionnement. Cependant, le prix à payer pour cette approche est la perte de performance. Une alternative consiste à minimiser le problème en équilibrant soigneusement les chemins d'horloge. Le prix à payer est une augmentation de la surface du circuit.

- **Ordre d'occurrence des entrées :** Les entrées qui peuvent arriver à des instants arbitraires peuvent poser un problème d'exactitude. De telles entrées peuvent rendre les éléments de mémorisation du circuit dans un état indéfini : métastabilité ([CHAN73]). Aucune solution n'existe pour éliminer ce problème. L'utilisation d'une paire d'éléments de mémorisation pour re-synchroniser une entrée asynchrone avec l'horloge ([CLUS86]) peut diminuer la performance du circuit.
- **Performance pire cas :** Si un composant du système peut procéder rapidement au traitement d'une certaine entrée, sa performance est toujours limitée par la fréquence d'horloge.
- **Consommation :** Quand le concepteur s'intéresse à des applications à faible consommation, la distribution d'horloge dans le circuit synchrone est une source importante de consommation. De plus, celle-ci croît rapidement quand la fréquence d'horloge augmente.
- **Modularité :** Dans un système synchrone, un composant ne peut pas être remplacé sans aucune implication globale. Si le nouveau composant est lent, le système ne peut pas fonctionner correctement sauf dans le cas où la fréquence d'horloge globale est réduite. Ceci souligne le principal problème des systèmes modernes orientés objets : il est difficile de combiner les sous-systèmes synchrones fonctionnant à différentes fréquences. La modularité est un aspect important dans la conception de circuits mais elle ne peut pas être mise en place dans le domaine synchrone.
- **Emissions électromagnétiques :** Dans un circuit synchrone, le traitement des actions est régi par une horloge globale. Tous les signaux commutent à la fréquence de l'horloge, ce qui provoque des variations brusques du courant donc de forts pics d'émission aux harmoniques de la fréquence d'horloge.

C'est sans doute l'arrivée d'obstacles majeurs à la logique synchrone qui a permis à la logique asynchrone de gagner en popularité ces dernières années. Solution naturelle aux limitations liées à la présence de synchronisation globale, la méthodologie asynchrone a ouvert la voie vers une nouvelle approche de conception. Au lieu de fonctionner avec une horloge globale, les systèmes asynchrones fonctionnent avec un contrôle localement distribué, ce qui permet d'éviter les problèmes liés à l'horloge globale.

1.1.2 Principes des circuits asynchrones

1.1.2.1 Mode de fonctionnement asynchrone

Contrairement aux systèmes synchrones où le signal d'horloge joue le rôle d'un actionneur global, les systèmes asynchrones sont des circuits dont le contrôle, ou le séquençage, est assuré par tout autre méthode que le recours à un signal périodique globalement distribué. 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 du système peut ainsi être indépendante de la durée d'exécution des composants du système.

Les circuits asynchrones fonctionnent donc avec la seule connaissance de l'occurrence des événements, sans connaissance de l'ordre. Le fonctionnement est similaire à celui des systèmes « flot de données ». On peut ainsi spécifier l'enchaînement des événements sous forme d'un graphe de

dépendances (réseau de Petri par exemple). L'analogie avec le modèle des processus séquentiels communicants ([HOAR78]) est très forte. Dans ce modèle, les processus se synchronisent par passage des messages via des canaux de communication. Pour échanger des informations, deux composants doivent se synchroniser. Ils échangent leurs informations à travers des canaux de communication, puis peuvent continuer leur flot d'exécution de manière indépendante. Un protocole de communication est alors nécessaire pour assurer le séquençement des transferts.

De manière générale, la structure distribuée du contrôle utilisé dans les composants permet la construction de systèmes corrects, sans limite de concurrence, par association de composants en utilisant des règles élémentaires d'interconnexions et d'ordre. Ceci est dû au fait que les spécifications fonctionnelles d'un composant et de son interface avec le reste du système, sont exprimées sans référence explicite au temps. L'architecture obtenue rend compte de façon explicite de la fonction : elle n'est pas cachée dans un séquenceur centralisé. Cette architecture permet d'étudier la conception d'une fonction sans contraindre sa spécification par des considérations liées à la réalisation du séquençement ou du contrôle.

Du point de vue de la conception, les systèmes asynchrones permettent de s'affranchir des problèmes de distribution des horloges et des alimentations. Sans utilisation d'horloge, les systèmes asynchrones peuvent fonctionner à des vitesses croissantes. De plus, ils ne possèdent pas de problèmes d'interfaçage délicat entre composants fonctionnant à des vitesses différentes. L'extension d'un système asynchrone est facile à réaliser. Ainsi, en raison de leur propriété de synchronisation locale, les systèmes asynchrones sont fonctionnels quelle que soit la réalisation des composants qui le constituent, pourvu que le protocole de communication soit respecté. Au niveau technologique, il est possible de modifier l'implémentation des composants, ou de remplacer des composants par leurs équivalents, sans en modifier la fonction. La migration technologique peut donc être progressive.

1.1.2.2 Problèmes

Malgré les avantages et les gains potentiels apportés, il existe des problèmes lors de la conception des circuits asynchrones. Le problème le plus important est lié au phénomène d'aléas [UNGE69]. En logique synchrone, comme le temps est discret, on peut se permettre d'ignorer les aléas liés aux phénomènes transitoires, alors qu'en asynchrone, il n'y a pas d'horloge globale : un circuit peut répondre aux transitions des entrées à n'importe quel instant. Toute transition d'un signal peut être interprétée comme un événement porteur d'information. En conséquence, tous les types d'aléas sont susceptibles de causer un mauvais fonctionnement du circuit. En raison de la sensibilité des circuits asynchrones aux aléas, les problèmes sont les suivants.

- Correction de fonctionnement : les méthodes de conception de circuits asynchrones doivent garantir une implémentation sans aléa.
- Flexibilité : plusieurs restrictions sévères sont imposées sur le circuit pour assurer un fonctionnement correct. Une restriction typique est le mode de fonctionnement fondamental : un seul changement des entrées à la fois. Cette restriction aide à la conception de circuits corrects car les techniques éliminant les aléas pour le mode fondamental sont bien connues et plus simple que celles pour le mode dans lequel plusieurs changements sont permis [UNGE69]. Cependant, les circuits générés sont peu utilisés en pratique.
- Compatibilité : plusieurs méthodes de conception de circuits asynchrones ne sont pas compatibles avec les interfaces existantes, comme celles des circuits synchrones. Elles requièrent l'utilisation de protocole de communication particulier, par exemple le protocole de type

« poignée de main » à 4 phases. Cette contrainte limite les applications de circuits asynchrones avec les interfaces existantes.

- Performance : en pratique certains types de circuits asynchrones ont des performances faibles. Les aléas sont souvent évités en ajoutant des retards aux circuits. Cette politique garantit la correction de fonctionnement mais enlève la performance potentielle du circuit asynchrone. De plus, la gestion de la communication localement dans chaque bloc fonctionnel du circuit nécessite des contrôleurs locaux, ce qui augmente la surface.
- Outil de conception : finalement, en raison de la présence d'aléas, la conception de circuits asynchrones, notamment de circuits complexes, est trop difficile à effectuer manuellement. Malheureusement, la conception des circuits asynchrones ne peut pas être généralement supportée par les outils de CAO existants et les implémentations synchrones alternatives. Les outils de CAO existants (placement, routage, partitionnement, synthèse logique, *etc.*) requièrent des modifications pour s'appliquer à la conception de circuits asynchrones ou ne sont pas applicables du tout.

De telles difficultés ont freiné l'adoption et l'utilisation des circuits asynchrones pendant des années. Cependant, avec les progrès considérables récents dans la communauté asynchrone, les circuits asynchrones, avec leurs avantages et leurs potentiels, promettent un changement de position à l'avenir, dans la conception de circuits. Avant d'apporter des solutions au problème de conception de circuits asynchrones dans les chapitres suivants, nous allons nous attacher à décrire brièvement les concepts de base permettant de bien cerner les enjeux par la suite. Nous aborderons en second lieu les principales méthodologies utilisées actuellement pour concevoir des circuits asynchrones, leurs points forts et leurs principales limitations.

1.2 Concepts de base

Au cours de ces dernières années, plusieurs formalismes et théories ont été proposés pour la conception des circuits asynchrones. De multiples approches se basent sur la combinaison de différents formalismes de spécification, d'hypothèses sur l'interaction entre le circuit et son environnement et également d'hypothèses sur le modèle de délai pour les éléments de circuits (portes, fils, ...). Dans ce chapitre, nous ne présenterons pas en détail tous ces aspects, mais tenterons de préciser certaines définitions, hypothèses et caractéristiques essentielles de la conception des circuits asynchrones.

1.2.1 Canaux de communication

Les circuits asynchrones sont souvent décomposés en blocs fonctionnels entre lesquels communiquent des données (dits jetons) via des canaux de communication. (Cette décomposition facilite la réutilisation des blocs asynchrones et simplifie donc la conception de systèmes complexes). 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 a) des signaux de contrôle nécessaires pour accomplir la communication et b) des signaux de données qui porte les données. Le protocole de communication est nécessaire pour gouverner la communication afin de maintenir le fonctionnement correct à l'égard de la spécification souhaitée. La Figure 2 illustre un canal de communication.

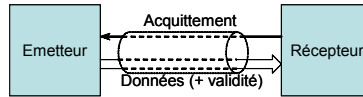


Figure 2 Notion de canal de communication

Les échanges de données à travers un canal sont directionnelles : 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. Pour un émetteur actif, il 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 provenant du récepteur. Pour éviter toute confusion, les canaux de communication abordés tout au long de cette thèse – sauf dans des cas bien précisés – sont composés d'émetteurs actifs et de récepteurs passifs.

Un canal de communication sert aussi à synchroniser les différents modules entre eux. Ce type de canal, que l'on présente plus loin, ne contient aucune donnée. Il est également possible de définir des canaux de communication où les données sont transférées dans les deux sens ([BERK96], [FERR02]). Ce type de canal, qui n'est pas abordé dans ce travail de thèse, est souvent utilisé pour accomplir l'interface avec des mémoires de type ROM, RAM, *etc.*

1.2.2 Représentation des données

Les circuits asynchrones sont activés en fonction de la présence de données sur leurs entrées et de leur disponibilité. Afin de toujours assurer la synchronisation entre les différents chemins de données dans la logique asynchrone, l'information de la validité est associée avec chaque bus de données. Ce principe implique nécessairement une nouvelle représentation des informations dans le circuit, *i.e.* un nouveau codage. On en distingue aujourd'hui principalement 2 types ([RENA00], [JERR02]).

1.2.2.1 Codage « données groupées »

La façon la plus évidente de caractériser la validité des données consiste simplement à ajouter un fil aux bus de données afin de spécifier si les données y sont valides (cf. Figure 3a). Les bus de données utilisent le traditionnel schéma de la logique synchrone : un fil par bit de données, ce qui s'appelle parfois *mono-rail* (« single rail ») dans la littérature ([JOSE99]). Le fil spécifiant la validité des données, dit signal de requête, est typiquement implémenté avec le retard adéquat ([FURB99], [KESS99], [NIEL99], [TERA99]). Ce retard est conçu égal ou supérieur au temps du calcul dans le pire cas.

Efficace en terme de surface (en terme de nombre de fils et donc, nombre de portes conduisant ces fils), ce type de codage permet une bonne réalisation des circuits asynchrones. Cependant, l'inconvénient de ce codage dans certains circuits est qu'en utilisant le retard assorti, le fonctionnement est fixé au pire cas : le fonctionnement ne dépend donc pas de la propagation réelle des données à l'intérieur d'un étage.

1.2.2.2 Codage insensible aux délais

Contrairement au codage « données groupées » où les données et leur validité sont totalement séparées, une autre approche plus complexe consiste à intégrer l'information de la validité dans les données. Cette approche possède la propriété suivante : les données sont détectées à l'arrivée sans que cela repose sur aucune hypothèse temporelle. Ceci implique donc robustesse, portabilité et facilité lors de la conception.

1.2.2.2.1 Codage 4 états

Dans ce codage (cf. Figure 3b), chaque bit de données est représenté par deux fils. Parmi les 4 états possibles pour ces 2 fils, la moitié est réservée à la valeur « 0 », l'autre à la valeur « 1 ». L'émission d'une nouvelle donnée se traduit par le changement d'un seul fil : celui de gauche pour exprimer la même valeur que précédemment, celui à droite pour exprimer la valeur opposée. La validité des données est donc assurée par le changement de parité du couple de fils.

1.2.2.2.2 Codage 3 états

De même que pour le codage 4 états, chaque bit est ici représenté par deux fils. En revanche, les valeurs représentées ne sont pas dupliquées : une seule combinaison représente chaque valeur, tandis que la troisième indique l'état invalide et la quatrième reste inutilisée. Ainsi, comme on le voit sur le schéma (cf. Figure 3c), le passage d'une valeur valide à l'autre se traduit nécessairement par le passage par l'état invalide.

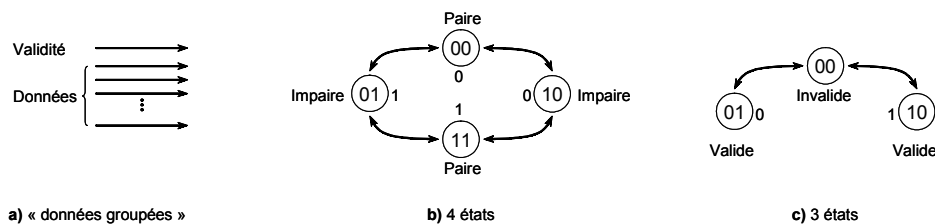


Figure 3 Codages les plus courants

Parmi tous ces codages, le codage 3 états – qui peut sembler par ailleurs le moins naturel – est aujourd'hui de loin le plus utilisé pour des raisons d'implémentation et de sécurité. En effet, les deux autres posent des problèmes en termes de logique additionnelle : dans le codage « données groupées », il faut re-combiner les données avec le signal de validité et dans le codage 4 états, un étage de logique supplémentaire est nécessaire pour tester la parité du couple de fils.

Le codage 3 états, appelé souvent codage *double-rails* – en raison de l'utilisation de 2 fils par bit de données – est un cas particulier du codage « one hot ». Pour étendre cette représentation et représenter un digit en base N, N fils sont alors utilisés : chaque fil affirmé représente une valeur, et l'état invalide est déterminé par la remise à zéro de tous les fils. Les combinaisons où au moins deux fils sont à « 1 » sont interdites. Ce codage (représenté Figure 4) est appelé le codage « 1-parmi-N » ou codage *multi-rails* dans ce manuscrit. Il est également possible d'étendre ce codage au codage « *M-parmi-N* ».

Rail / Valeur	Invalide	N-1	N-2	...	1	0	Interdite
Rail(N-1)	0	1	0		0	0	1
Rail(N-2)	0	0	1		0	0	x
...	0	0	0		0	0	x
Rail(1)	0	0	0		1	0	1
Rail(0)	0	0	0		0	1	x

Figure 4 Codage « 1-parmi-N »

Un cas dégénéré du codage 3 états est celui de deux états : un état valide et un état invalide. Servant à exprimer simplement l'information de la validité ou de l'invalidité, un canal utilisant un tel codage sert exclusivement à synchroniser l'activité entre différents modules asynchrones. Dans ce cas, un seul fil est nécessaire pour représenter l'information : la mise à « 1 » du fil représente la validité, tandis que la remise à « 0 » indique l'état invalide. Le codage dans ce cas est dit *mono-rail* (« single-rail ») pour être cohérent avec le codage multi-rails, ce qui peut entraîner une confusion de

nom avec le codage « données groupées » (cf. §1.2.2.1). Pour être clair, le nom mono-rail est réservé pour le codage possédant un seul fil dans ce manuscrit de thèse.

1.2.3 Protocoles de communication

La description comportementale de chaque bloc asynchrone repose avant tout sur le choix du protocole utilisé pour communiquer avec ses homologues. Le protocole de synchronisation (ou protocole de type « poignée de main ») est l'ensemble des règles qui gouverne la communication au sein du canal entre plusieurs blocs asynchrones.

Plusieurs protocoles ont été proposés pour respecter les contraintes des hypothèses posées. Afin de simplifier au mieux le protocole, la méthode la plus utilisée consiste à développer l'interaction (séquencement du transfert de l'information) entre 2 blocs fonctionnels, selon un cycle élémentaire simplifié au mieux (le transfert d'un jeton). Avec cette convention, on retient principalement deux types de protocoles : le protocole à 2 phases et le protocole à 4 phases.

Un protocole de communication entre 2 blocs fonctionnels via un canal est caractérisé par a) la séquence de transfert des informations, b) le codage adopté pour les données et c) l'activité des ports. Le nombre de combinaison est le produit cartésien de ces options : $\{2 \text{ phases}, 4 \text{ phases}\} \times \{\text{« données groupées », double-rails, multi-rails, ...}\} \times \{\text{actif, passif}\}$

La Figure 5 représente le protocole 4 phases « données groupées » à titre d'exemple. Ce protocole est souvent utilisé dans la conception des circuits asynchrones de type « micropipeline » [SUTH89].

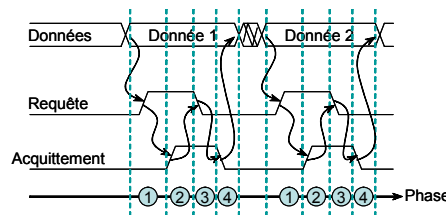


Figure 5 Protocole 4 phases « données groupées »

De même que pour le codage, l'implémentation d'un protocole de synchronisation est coûteuse en terme de logique : c'est le prix à payer pour combiner données, validité et coordination entre les blocs fonctionnels asynchrones. Elle est également coûteuse en terme de rapidité : là où précédemment il suffisait de propager un jeton (« flot de données ») l'un après l'autre, il faut désormais propager l'établissement de nouveau jeton, l'information de validité, réaliser le « rendez-vous » (synchronisation des communications) et enfin désactiver la procédure. Ces multiples pénalités temporelles constituent un facteur critique dont l'étude est primordiale pour permettre aux circuits asynchrones de rivaliser en vitesse avec les circuits synchrones.

Le choix du protocole affecte les caractéristiques de l'implémentation du circuit (surface, vitesse, consommation, robustesse, *etc.*). De manière générale, le protocole 4 phases nécessite un matériel moins important que le protocole 2 phases et les techniques d'optimisation du pipeline présentées plus tard sont bien adéquates pour le protocole 4 phases. Le protocole 2 phases est efficacement utilisé lors de la conception des circuits dans lesquels les signaux doivent traverser des éléments possédant une latence plus élevée ([RENA98]).

1.2.3.1 Protocoles 2 phases

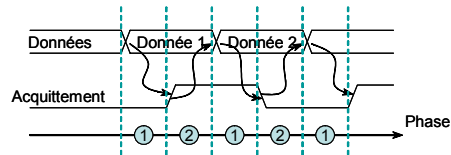


Figure 6 Protocole 2 phases

Le protocole à deux phases (appelé aussi NRZ comme *Non Retour à Zéro*) est représenté Figure 6. Il constitue la séquence d'échange d'informations minimale nécessaire à une communication : les données et les acquittements sont représentées par des fronts (montant ou descendant). Les deux phases sont les suivantes :

- Phase 1 : c'est la phase active du récepteur qui détecte la présence d'une donnée, 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 une nouvelle donnée si elle est disponible

Ce protocole, malgré son efficacité apparente, n'est que peu utilisé en raison des difficultés rencontrées dans la logique nécessaire à son implémentation. L'asymétrie entre deux phases successives (montée ou descente de l'acquittement) se révèle un obstacle majeur à la réalisation du protocole.

1.2.3.2 Protocoles 4 phases

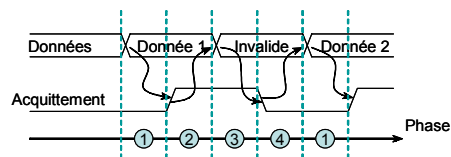


Figure 7 Protocole 4 phases

Les 4 phases de ce protocole (également nommé RZ comme *Retour à Zéro*) sont les suivantes :

- Phase 1 : le récepteur détecte une nouvelle donnée, effectue le traitement et génère le signal d'acquittement.
- Phase 2 : l'émetteur détecte le signal d'acquittement et invalide les données.
- Phase 3 : le récepteur détecte l'état invalide et désactive le signal d'acquittement.
- Phase 4 : l'émetteur détecte l'état invalide de l'acquittement, émet une nouvelle donnée si elle est disponible.

A l'opposé du protocole 2 phases, le protocole 4 phases est le protocole le plus utilisé en raison de la symétrie de ses phases. En doublant leur nombre, l'état du système est invariablement le même au début et à la fin d'une communication (Figure 7).

1.2.4 Implémentation du protocole : utilisation de la porte de Müller

Pour implémenter de tels protocoles, les portes logiques élémentaires ne suffisent pas. Ce paragraphe présente rapidement le fonctionnement des portes de Muller (ou *C élément*), proposées par Muller dans [MILL65]. Elles sont essentielles pour l'implémentation de circuits asynchrones : elles réalisent le rendez-vous entre plusieurs signaux. La sortie copie la valeur de ses entrées lorsque celles-ci sont identiques, sinon elle mémorise la dernière valeur obtenue. La Figure 8 représente le symbole d'une porte de Muller à deux entrées et sa spécification en différentes formes.

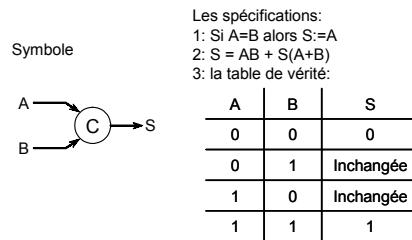


Figure 8 Symbole et spécifications de la porte de Muller à deux entrées

A partir de la spécification de la porte de Muller, la Figure 9 décrit plusieurs réalisations possibles au niveau logique et au niveau transistor.

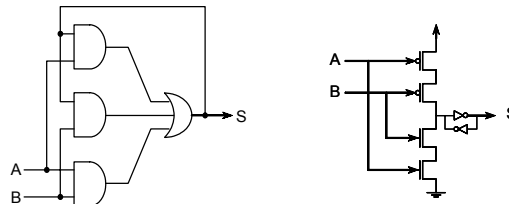


Figure 9 Réalisations de la porte de Muller

Il existe aussi des variantes de porte de Muller. La porte de Muller généralisée (de l'anglais « generalized C element ») [MART86] est une porte de Muller dans laquelle les signaux qui font monter la sortie à « 1 » (les signaux_1) ne sont pas toujours les mêmes que ceux qui la font descendre à « 0 » (les signaux_0). Dans ce cas, la porte de Muller est également dite dissymétrique. Le fonctionnement d'une telle cellule est le suivant :

- Lorsque l'état des signaux_1 est identique et égal à « 1 », la sortie passe à « 1 »
- Lorsque l'état des signaux_0 est identique et égal à « 0 », la sortie passe à « 0 »
- Si aucune des conditions ci-dessus n'est vraie, la sortie est mémorisée.

A titre d'exemple, la Figure 10 illustre une porte de Muller généralisée (le symbole, la spécification et la réalisation au niveau transistor). Dans cet exemple, les entrées B et C font commuter la sortie à « 1 » quand elles valent « 1 ». Les entrées A et B font commuter la sortie à « 0 » quand elles valent « 0 ». La sortie reste mémorisée sinon.

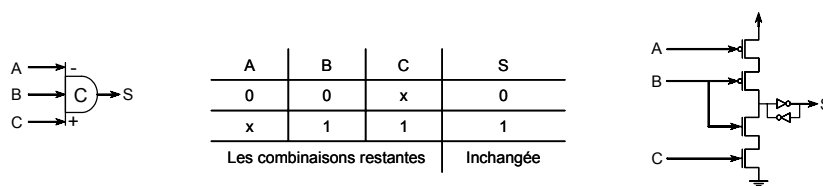


Figure 10 Exemple de porte de Muller dissymétrique [RIGA02]

1.2.5 Aléas

Dans le sens le plus général, un aléa est une activité non désirée (« glitch ») en réponse à un changement sur certaines entrées. Un aléa peut se produire en raison de la différence de délais entre portes. La présence d'aléas dans un circuit – notamment pour un circuit asynchrone qui est très sensible à ces transitions – entraîne des fautes de fonctionnement. Elle 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 problème est réglé, il peut encore apparaître des aléas au niveau de l'implémentation. Cela signifie que la conception d'un circuit ne s'arrête pas après la spécification. Il faut encore s'assurer que les techniques de réalisation choisies sont exemptes d'aléas.

Dans le cadre de la conception de circuits asynchrones, il est à remarquer que les circuits asynchrones requièrent une conception attentive et fine de toutes les parties (combinatoires et séquentielles) du circuit, de la spécification jusqu'à l'implémentation matérielle. Pour la partie combinatoire du circuit, il existe deux classes de base pour les aléas combinatoires : aléas logiques et aléas fonctionnels. Chaque classe d'aléas comprend les aléas statiques et les aléas dynamiques. Les paragraphes suivants présentent les différents types d'aléas dont tout circuit asynchrone ne faisant aucune hypothèse temporelle doit se prémunir.

1.2.5.1 Aléas combinatoires fonctionnels

Aléa fonctionnel statique ([BRED72]) : f est une fonction logique booléenne possédant un aléa fonctionnel statique pour une transition des entrées du monôme A au monôme C ($A, C \in \{0, 1\}^n$) si et seulement si :

- $f(A) = f(C)$
- il existe un état des entrées (monôme) $B \in [A, C]$ tel que $f(A) \neq f(B)$

Aléa fonctionnel dynamique ([BRED72]) : f est une fonction logique booléenne possédant un aléa fonctionnel dynamique pour une transition des entrées du monôme A au monôme D si et seulement si :

- $f(A) \neq f(D)$
- il existe une paire d'états des entrées B et C ($A \neq B, C \neq D$) tel que « $B \in [A, D]$ et $C \in [B, D]$ et $f(A) = f(C)$ et $f(B) = f(D)$ »

Ainsi, les aléas combinatoires fonctionnels sont la propriété de la fonction logique. Ils peuvent être uniquement détectés et supprimés à un niveau plus élevé de la conception en étudiant et en modifiant la spécification logique de la fonction. Ces aléas ne dépendent pas de l'implémentation et de la distribution des délais. Il faut une spécification exempte d'aléa fonctionnel pour obtenir une réalisation sans aléa si aucune hypothèse n'est faite sur les délais des éléments.

Il est bien connu que si une transition possède un aléa fonctionnel, aucune implémentation de la fonction n'est assurée pour éviter les aléas dans cette transition, quelque soit le modèle de délais des portes et des fils ([EICH65][BRED72]). Par conséquent, dans le reste de cette thèse, les transitions des entrées sont supposées exemptes d'aléa fonctionnel.

1.2.5.2 Aléas combinatoires logiques

Les aléas combinatoires logiques sont la propriété de l'implémentation de la fonction logique. Leur apparition dépend de la distribution des délais dans la logique. [BRED72] indique que si l'implémentation d'une fonction logique contient des aléas fonctionnels pour une transition des entrées donnée, alors elle ne peut pas avoir d'aléa logique pour la même transition. Cependant, si l'implémentation est exempte d'aléa fonctionnel, elle peut contenir des aléas logiques.

Dans cette classe d'aléas, il existe deux principaux types d'aléas ([UNGE69][UNGE71]) :

- **Aléas de type SIC** (de l'anglais « Single Input Change ») : Comme indiqué par son nom, ce type d'aléas se produit au cours de la transition d'une seule entrée.
- **Aléas de type MIC** (« Multi-Input Change ») : c'est les aléas logiques rencontrés lorsque plusieurs entrées changent.

Comme les aléas fonctionnels, les aléas logiques comprennent également les aléas statiques et les aléas dynamiques.

1.2.5.2.1 Aléa logique statique

Définition ([BRED72]) : une implémentation combinatoire de la fonction logique f contient un aléa logique statique pour une transition des entrées du monôme A au monôme B si et seulement si :

- $f(A) = f(B)$
- pour un certain modèle de délais, la sortie du circuit ne reste pas monotone au cours de la transition

Cet aléa est aussi appelé « spike » dans le jargon du concepteur. Par exemple, toute porte « AND » et « OR » est susceptible de générer ce type d'aléas si deux entrées changent simultanément (cf. Figure 11).

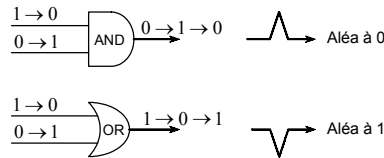


Figure 11 Aléas statiques causés par des portes logiques « AND » et « OR »

Les aléas statiques apparaissent quand une transition n'est pas complètement couverte par une porte, c'est-à-dire, chaque fois que l'implémentation ne contient pas une porte qui maintient la valeur de la sortie lors de la transition. A titre d'exemple, dans la Figure 12, la transition de A (xyz) à B (xyz) n'est pas couverte par une seule porte dans l'implémentation. Il est alors possible, avec un certain modèle de délais, que les deux portes « AND » descendent momentanément, ce qui fait passer la sortie f à « 0 » (aléa statique « 1 »). Cet aléa peut être éliminé en complétant la fonction f par une porte « AND » avec ses entrées xy . Cette porte maintient la sortie à « 1 » pendant la transition.

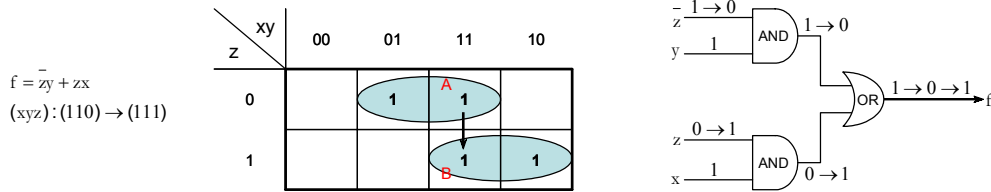


Figure 12 Exemple d'aléas logiques statiques : SIC

La Figure 13 illustre un aléa logique statique de type MIC. Lors de la transition de A à B, il n'y a aucune porte qui maintient la sortie au niveau « 1 ». Ainsi, si les portes wx et yz sont suffisamment rapides et si les portes xz et wy sont suffisamment lentes, alors la sortie peut momentanément prendre la valeur « 0 » pendant la transition (aléa statique « 1 »).

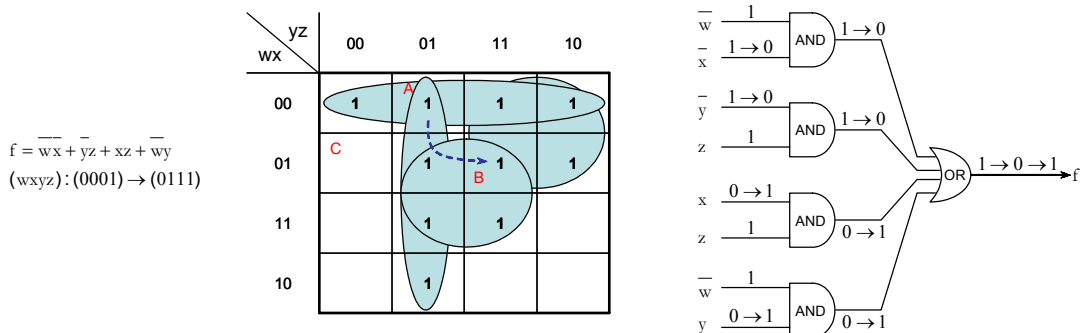


Figure 13 Exemple d'aléas logiques statiques : MIC

En conclusion, il a été montré [WAKE94] que pour une implémentation quelconque sous forme SOP (« Sum Of Product » ou somme de produits), aucun aléa ne peut survenir lors des

transitions $0 \rightarrow 0$, $0 \rightarrow 1$ et $1 \rightarrow 0$ 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 deux niveaux a été démontrée dans [EICH65]. Pour les implémentations SOP multi-niveaux, les conditions sont plus complexes et sont discutées dans [BRED75].

1.2.5.2.2 Aléa logique dynamique

Définition ([BRED72]) : une implémentation combinatoire de la fonction logique f contient un aléa logique dynamique pour une transition des entrées du monôme A au monôme B si et seulement si :

- $f(A) \neq f(B)$
- pour un certain modèle de délais, la sortie du circuit ne reste pas monotone au cours de la transition

La Figure 14 illustre un signal passant à « 1 » (respectivement à « 0 »), qui subit un aléa dynamique à la montée (respectivement à la descente).

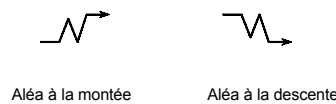


Figure 14 Aléas dynamiques

Les aléas logiques dynamiques s'appliquent aux deux types SIC et MIC. Dans le cas SIC, cet aléa correspond à la situation où un littéral et son complément se déploient dans plusieurs chemins de données. Dans le cas MIC, un aléa dynamique apparaît quand une porte commute momentanément pendant une transition. A titre d'exemple, un aléa logique dynamique de type MIC est représenté Figure 15. Dans cet exemple, la transition de C à A peut provoquer un aléa logique dynamique.

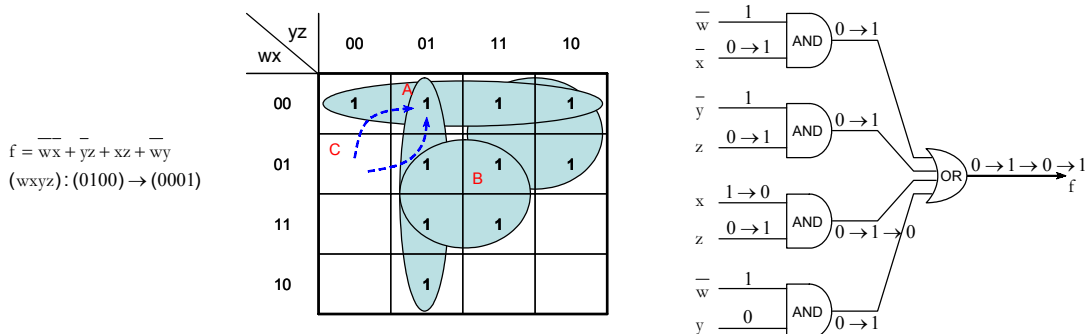


Figure 15 Exemple d'aléas dynamiques logiques : MIC

[SIEG93] présente comment détecter les aléas logiques dynamiques.

Pour conclure, l'implémentation est donc de toute première importance en asynchrone et nécessite la conception d'outils qui intègrent la notion d'aléas. En particulier, l'étape de projection technologique 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.

La suppression d'aléas est donc un problème de couverture du tableau de Karnaugh :

- Les transitions $1 \rightarrow 1$ doivent être couvertes.
- Pour les transitions $1 \rightarrow 0$ et $0 \rightarrow 1$, un produit qui intersecte avec la transition doit contenir le monôme de départ ou le monôme d'arrivée.

- Les transitions $0 \rightarrow 0$ ne génèrent pas d'aléas dans les réalisations SOP.

Ces conditions suffisent pour éliminer n'importe quel aléa de type MIC. Si 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 état) l'existence d'une couverture du tableau de Karnaugh conduisant à une réalisation SOP sans aléa n'est pas assurée.

1.2.5.3 Aléas séquentiels

Ces aléas trouvent leur origine dans les signaux bouclés. Ces aléas sont détectables lors de la spécification d'un problème, par exemple lors de l'étude d'un tableau de flots (« flow table » [UNGE69]). On cite souvent les aléas séquentiels qui peuvent être détectés en faisant changer la même entrée une fois puis trois fois. Il y a aléa séquentiel si l'état final n'est pas le même dans les deux cas.

1.2.5.4 Conclusion

Il existe plusieurs classes et plusieurs types d'aléas dans la conception de circuits. En particulier, les circuits asynchrones sont très sensibles aux aléas et demandent donc une conception très soignée et attentive de toutes les parties (combinatoires et séquentielles).

En résumé, les aléas sont générés dans un certain modèle de délais. Pour une transition donnée entre un monôme A et un monôme B, les aléas statiques s'appliquent aux transitions où $f(A) = f(B)$ tandis que les aléas dynamiques s'appliquent aux cas où $f(A) \neq f(B)$. Les aléas combinatoires fonctionnels sont difficiles à détecter et à corriger, puisqu'ils dépendent uniquement de la spécification du circuit. Nous ne les aborderons pas dans la suite. Par contre, les aléas logiques sont générés au cours de la réalisation des circuits et il faut avoir des méthodes d'implémentation sans aléa pour que l'implémentation finale du circuit fonctionne correctement dans le mode asynchrone.

Par la suite, il est vu 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.

1.2.6 Modèles de délais, de circuits et d'environnement

Les modèles qui régissent le comportement des délais, des circuits et des environnements déterminent les différentes implémentations de circuits. Les principaux modèles couramment utilisés pour la conception de circuits asynchrones sont présentés dans ce paragraphe.

1.2.6.1 Modèles de délais

La notion de délai est très importante lors de l'implémentation des circuits. Les délais sont inhérents aux circuits physiques. Ils peuvent être ajoutés de façon explicite à un circuit en utilisant des éléments de délais. Il existe deux modèles courants de délais : le modèle « *pur* » et le modèle « *inertiel* » ([UNGE69][UNGE71]). Le « *délai pur* » ne change pas la forme d'onde des signaux mais les retarde temporellement, tandis que le « *délai inertiel* » peut changer la forme d'onde des signaux. En particulier, le délai inertiel possède une période de seuil D : les impulsions de durée inférieure à D sont éliminées.

Les délais peuvent aussi être caractérisés par leurs modèles temporels. Dans le modèle de délais dit « *non borné* », le délai peut prendre une valeur finie arbitraire. Dans le modèle « *borné* », le délai possède une valeur située dans un intervalle connu. Dans le modèle de délais « *fixe* », le délai est fixé à une valeur connue déterminée.

Les délais servent à caractériser le comportement des fils et des portes dans un circuit. Typiquement, un délai est associé à chaque fil du circuit. Dans le modèle dit « *porte simple* » (ou « *niveau porte* »), un délai est associé à chaque porte du circuit. Dans le modèle de « *porte complexe* », un seul délai est associé à un sous-ensemble de portes simples. Ainsi, dans ce modèle, le réseau de portes est traité comme une porte individuelle.

Un modèle de circuit est défini par les modèles de délais adoptés pour ses fils et ses portes.

1.2.6.2 Modes de fonctionnement

Il est également important de formaliser l'interaction entre le circuit et l'environnement dans lequel il s'insère. Le circuit et son environnement forment un système fermé, dit « *circuit complet* » ([MILL65]). Selon les terminologies utilisées par [BRZO89], les modèles d'environnement sont classifiés de la manière suivante.

- **Mode fondamental** : il existe des contraintes temporelles pour que l'environnement et le circuit interagissent. En particulier, l'environnement ne peut pas répondre avant que le circuit ne soit dans un état stable (toutes les entrées, les sorties et les signaux internes sont stables). Dans cet état, l'environnement est autorisé à changer une seule entrée. L'environnement ne peut changer une entrée à nouveau que lorsque le circuit est revenu dans un état stable après avoir produit une sortie. Puisque l'environnement ne connaît pas l'état stable du circuit, il doit respecter le délai le plus long nécessaire pour stabiliser le circuit. Ceci implique de borner les délais des portes et des fils du circuit. Cette limitation sur l'environnement est aussi formalisée comme une contrainte de synchronisation absolue.
- **Mode rafale** : c'est une extension du mode fondamental. Quand le circuit est dans l'état stable, l'environnement peut possiblement changer plusieurs entrées (l'ordre de changements n'est pas important). Il n'est pas autorisé à changer de nouveau les entrées avant que le circuit produise les sorties et revienne à l'état stable.
- **Mode entrée-sortie** : l'environnement répond au circuit sans contrainte de délais. Il est possible pour l'environnement de changer les entrées avant que le circuit soit stabilisé en réponse aux changements précédents. Les seules restrictions sur l'environnement sont les relations de causalité entre les transitions sur les entrées et les sorties.

1.2.7 Catégorie de circuits asynchrones

Les circuits asynchrones sont communément classés suivant le modèle de circuit et d'environnement adopté. La Figure 16 présente la terminologie habituellement utilisée pour qualifier les circuits asynchrones. Plus le fonctionnement du circuit respecte fondamentalement la notion d'asynchronisme, plus le circuit est robuste et plus il est complexe. Dans la suite, nous présentons brièvement ces catégories de circuit en commençant par les circuits dont le fonctionnement respecte fondamentalement la notion d'asynchronisme, puis des circuits avec des hypothèses temporelles de plus en plus fortes. Cette approche va donc des circuits purement asynchrones vers des circuits synchrones.

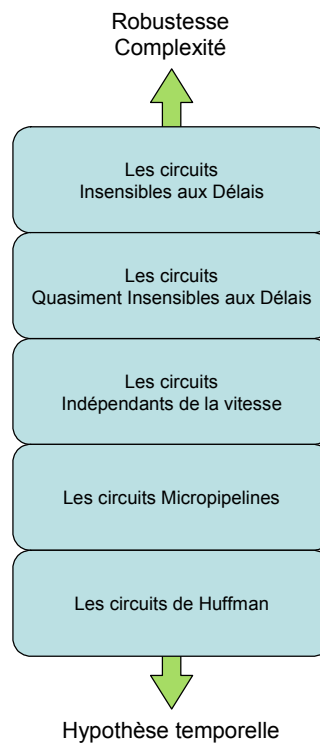


Figure 16 Différentes classes de circuits asynchrones.

1.2.7.1 Circuits insensibles aux délais (DI, « Delay Insensitive »)

Les circuits de ce type fonctionnent fondamentalement dans le mode asynchrone. Ils utilisent le modèle de délais « non borné » pour les fils et les portes. Aucune hypothèse temporelle n'est introduite. Informellement, cela signifie qu'ils sont fonctionnellement corrects indépendamment des délais introduits par les fils ou les éléments logiques.

Basés sur les travaux de [WESL67] et re-formalisés par [UDDI86], les circuits DI sont supposés répondre toujours correctement à une sollicitation externe pourvu qu'ils aient assez de temps pour calculer. 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 entrée et/ou la fin de son traitement. Les circuits émetteurs doivent attendre un compte rendu avant d'émettre une nouvelle donnée. Les contraintes de réalisation pratique, qu'impose l'utilisation de ce modèle, sont très fortes. De plus, la plupart des circuits conçus aujourd'hui utilisent des portes logiques à une seule sortie. L'adoption du modèle de délai « non borné » ne permet pas leur utilisation. En effet, les portes logiques standard ont leurs sorties qui peuvent changer si une seule de leurs entrées change. Comme nous l'avons souligné, toutes les composantes de ce type de circuits doivent s'assurer du changement des entrées avant de produire une sortie. Si la sortie change alors qu'une seule des entrées change, seul le changement de l'entrée active est acquittée, sans pouvoir tester l'activité des autres entrées. La seule porte à une sortie qui respecte cette règle est la porte de Muller (cf. §1.2.4). Malheureusement, les fonctions réalisables avec seulement des portes de Muller sont très limitées ([MART90]). La seule solution est d'avoir recours à un modèle de circuit de type « portes complexes » pour les composants élémentaires ([EBER91][JOSE93]). Dans ce cas la construction de ces circuits se fait à partir de composants standard plus complexes que de simples portes logiques et qui peuvent posséder plusieurs entrées et plusieurs sorties.

1.2.7.2 Circuits quasi-insensibles aux délais (QDI, « Quasi-Delay Insensitive »)

Comme son nom l'indique, la classe des circuits QDI est un sur-ensemble de la classe DI. Cette classe adopte le même modèle de délai « non borné » pour les connexions mais y ajoute la notion de *fourche isochrone* (« isochronic fork ») [MART90][BERK92] et un modèle de type « porte simple » et « non borné » pour les composants élémentaires du circuit.

Une *fourche* est un fil qui connecte un expéditeur unique à deux (ou plusieurs) récepteurs. Elle est qualifiée d'*isochrone* lorsque les délais entre l'expéditeur et les récepteurs sont identiques. Cette hypothèse a des conséquences importantes sur le modèle et les réalisations possibles. Elle résout notamment le problème de l'utilisation de portes logiques à une seule sortie, causé par la classe DI. Car si les fourches sont isochrones, il est permis 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. En conséquence, on peut autoriser l'acquittement d'une des branches de la fourche isochrone. Alain Martin a montré dans [MART90b] que l'hypothèse temporelle de fourche isochrone est la plus faible à ajouter aux circuits DI pour les rendre réalisables avec des portes à plusieurs entrées et une seule sortie. Les circuits QDI sont donc réalisables avec des cellules standard telles qu'on les utilise pour la conception de circuits synchrones.

Finalement, dans les circuits QDI, les signaux peuvent mettre un temps arbitraire à se propager dans les portes ainsi que dans les connexions (avec l'hypothèse de fourche isochrone) sans que cela remet en cause la fonctionnalité du circuit. De telles réalisations sont, bien entendu, très délicates à mettre au point compte tenu des contraintes imposées à la conception. En contrepartie, elles représentent un degré de robustesse quasi-parfait. En théorie, aucune erreur inhérente au circuit ne peut se produire. En pratique, la contrainte de fourche isochrone est assez faible, et est facilement remplie par une conception soignée, en particulier au niveau du routage et des seuils de commutation. Il suffit, finalement, que la dispersion des temps de propagation jusqu'aux extrémités de la fourche soit inférieure aux délais des opérateurs qui lui sont connectés (au minimum une porte et un fil dont une sortie interagit avec la fourche [MART90][BERK92]).

1.2.7.3 Circuits indépendants de la vitesse (SI, « Speed Independence »)

Les circuits SI – initialement décrit par [MILL65] – font l'hypothèse que les délais dans les fils sont négligeables, tout en conservant un modèle « non borné » pour les délais dans les portes. Ce modèle, qui n'est pas réaliste dans les technologies actuelles, est en pratique équivalent au modèle QDI à l'exception des fourches qui sont toutes considérées comme isochrones. 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 les modèles QDI et SI équivalents. Hauck dans [HAUC95] montre avec le schéma de la Figure 17 comment une fourche isochrone peut être représentée par un circuit SI.

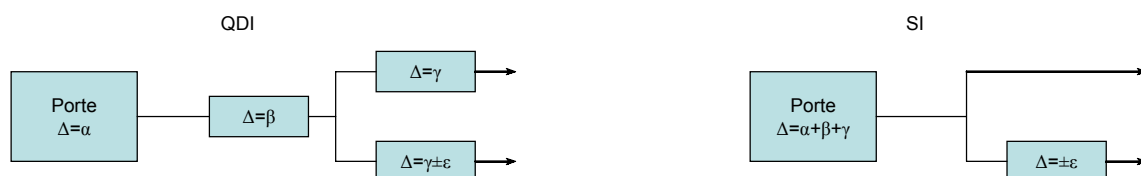


Figure 17 Equivalence entre les modèles QDI et SI.

Cependant, il semble plus pertinent d'utiliser le modèle QDI car celui-ci identifie les fourches isochrones de celles qui ne le sont pas. Ceci offre le moyen de vérifier les connexions du circuit qui ne sont pas insensibles aux délais au cours des différentes étapes de conception, et seulement celles-ci.

1.2.7.4 Circuits micropipelines

La technique micropipeline a été initialement introduite par Ivan Sutherland [SUTH89]. Il faut en fait considérer que les circuits micropipelines correspondent plutôt à une classe d'architectures de circuit asynchrone que directement à un modèle de délais de circuit asynchrone. Les circuits de cette classe sont composés de parties de contrôle insensibles aux délais qui commandent des chemins de données conçus en utilisant le modèle de délais borné. La structure de base de cette classe de circuits est le contrôle d'une file (FIFO) (cf. Figure 18). Elle se compose d'éléments identiques connectés tête-bêche.

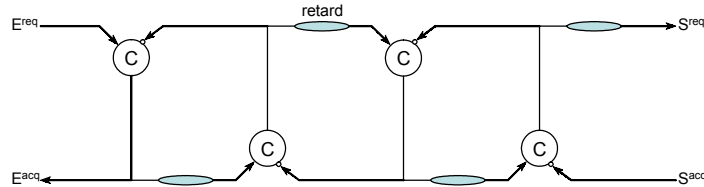


Figure 18 Structure de base des circuits micro pipelines.

Cette structure implémente un protocole deux phases. Le circuit réagit à des transitions de signaux et non pas à des états. On parle également d'une logique à événements : chaque transition étant associée à un événement. Ainsi, tous les signaux sont supposés à zéro initialement. Une transition positive sur E^{req} provoque une transition positive sur E^{acq} 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. Ce fonctionnement de type FIFO est bien insensible aux délais.

Cette structure est une structure de contrôle simple. Elle peut alors être utilisée à contrôler le chemin de données possédant des opérateurs de mémorisation et des opérateurs de traitements combinatoires. Dans ce cas, le codage de données et le protocole associé est du type deux phases « données groupées ». La Figure 19 illustre un micropipeline avec traitement.

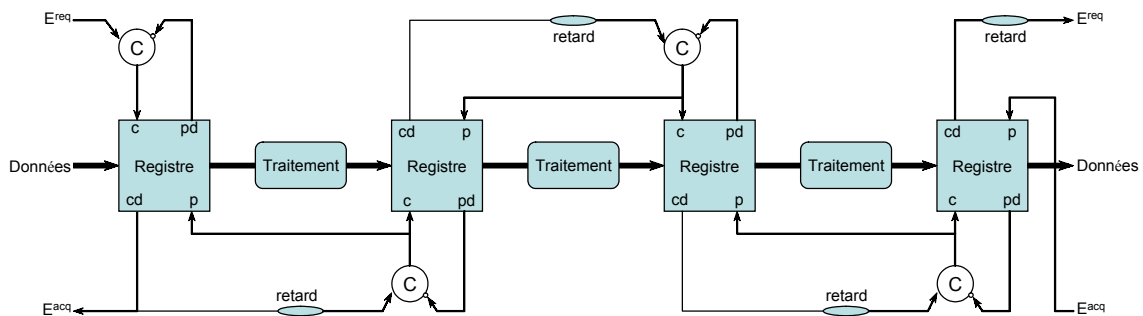


Figure 19 Structure micro pipeline avec traitement et registre.

Avec la logique qu'elle contient, cette structure se comporte comme le circuit de base (cf. Figure 18). C'est une FIFO qui peut contenir un nombre variable de données, les données progressant le plus loin possible dans la logique en fonction du nombre d'états vides. Dans cette structure, le contrôle et les registres sont spécifiques. Quant à elle, la logique combinatoire est équivalente à la logique synchrone. Le problème d'aléas est écarté grâce à l'insertion de délai sur les signaux de contrôle. L'hypothèse de délai consiste alors à vérifier que les données sont stables avant la mémorisation de celles-ci. Dans ce cas, les signaux de contrôle s'apparentent en fait à des horloges locales. Comme les délais sont fixes, cette structure ne permet pas de tirer parti de la variation dynamique du temps de traitements des opérateurs.

La motivation première pour le développement de cette classe de circuits était de permettre un pipeline élastique. En effet, le nombre de données présentes 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 il est autorisé la mémorisation des données seulement lorsqu'elles sont stables (à la sortie des portes combinatoires).

Un certain nombre d'optimisations peut être apporté à cette structure initiale. Des circuits SI peuvent être obtenus en détectant localement la fin de calcul de la logique. Ceci peut être effectué par exemple en remplaçant la logique combinatoire par la logique à précharge [FURB96]. Il est aussi possible de supprimer les *latches* explicites en utilisant de la logique différentielle à précharge [RENA96]. Des bascules de mémorisation sur double front peuvent être utilisées dans le cas de protocole deux phases [YUN96]. Enfin, des protocoles quatre phases optimisés permettent aussi d'obtenir un taux de remplissage de la structure important afin d'augmenter les performances [FURB96b].

De nombreuses méthodes de conception s'attachent à concevoir des circuits micropipeline avec des modèles de délais plus ou moins différents. L'idée générale de ces approches est de séparer dans la spécification le contrôle des données et de les implémenter séparément dans des styles différents. En particulier, les contrôleurs des registres de pipeline peuvent être obtenus avec un grand nombre de méthodes et avec des hypothèses de délais plus ou moins fortes.

1.2.7.5 Circuits de Huffman

Cette catégorie regroupe des circuits qui utilisent un modèle de délai identique aux circuits synchrones. Ils supposent que les délais dans tous les éléments du circuit et dans les connexions sont bornés ou même de valeurs connues. Les hypothèses temporelles sont donc du même ordre que pour la conception de circuits synchrones. Leur conception repose sur l'analyse des délais dans tous les chemins et les boucles de façon à dimensionner les temps d'occurrence des signaux de contrôles locaux. Ces circuits sont d'autant plus difficiles à concevoir et à caractériser qu'une faute de conception ou de délai les rend totalement non fonctionnels.

1.2.7.6 Conclusion

Il est à noter que le terme de circuits « auto séquencés » (« self-timed ») – comme décrit [SEIT80] – est souvent utilisé dans la littérature pour désigner les circuits asynchrones mais il ne correspond en fait à aucune définition précise. Il désigne simplement des circuits dont l'enchaînement des étapes est déterminé par des signaux internes aux circuits plutôt que par un signal d'horloge. Il qualifie donc les circuits non synchrones plutôt qu'il ne renseigne sur la technique asynchrone mise en œuvre.

La présentation des différentes classes de circuits asynchrones vient d'être faite. Celles-ci proposent, dans leur mode de fonctionnement, des hypothèses temporelles plus ou moins fortes. Plus cette contrainte est relâchée, moins le circuit est robuste d'un point de vue comportemental. Mais, l'ensemble des fonctionnalités implantables est plus grand. Pour chacune de ces classes, il peut être associé une ou plusieurs méthodologies de conception développées par différents groupes de recherche de la communauté asynchrone. Ceci est le reflet du vaste spectre de solutions qu'offre le relâchement des synchronisations. A ce jour, il n'existe pas de vision unificatrice de ces différentes méthodologies de conceptions. Ainsi, il n'y a pas de consensus sur le type de circuits asynchrones le

plus approprié : cela dépend des objectifs et des moyens visés. Si les avantages présentés dans ce chapitre sont considérés, les circuits QDI, SI et micropipeline semblent les mieux adaptés pour concevoir des systèmes robustes dans les technologies actuelles tout en visant de bonnes performances.

Dans le flot de conception de circuits asynchrones TAST que représente ce travail de thèse, seuls les circuits QDI et micropipelines sont ciblés. La méthode de synthèse proposée est illustrée sur la classe de circuits QDI, mais comme nous le montrerons par la suite, il est possible d'utiliser cette méthode pour la synthèse des contrôleurs des circuits micropipelines.

1.3 Méthodologies de conception des circuits asynchrones

Les circuits asynchrones ne possèdent pas les problèmes liés à l'horloge et ont un certain nombre d'atouts tels que faible consommation, calcul en temps minimum ou moyen, modularité, robustesse, facilité de migration technologique, ... Cependant, ils sont difficiles à concevoir au niveau bas tel que le niveau des portes. Il est donc nécessaire d'avoir une méthodologie de conception de circuits permettant au concepteur de décrire la spécification d'un circuit souhaité et ensuite de synthétiser cette description au niveau matériel. Ainsi, une méthode de spécification de circuits de haut niveau, par exemple dans un langage de description, et une méthode de synthèse à partir de cette spécification de circuit s'avèrent nécessaires.

Deux raisons font que les méthodologies synchrones traditionnelles ne peuvent pas être utilisées pour la conception des circuits asynchrones :

- Les langages de description de matériel utilisés pour les circuits synchrones, tels que VHDL et Verilog, ne permettent pas de décrire parfaitement des communications via des canaux dans la philosophie asynchrone. De plus, ils ne sont pas suffisamment flexibles pour modéliser la concurrence et la séquentialité des blocs fonctionnels asynchrones.
- Le phénomène d'aléa – étant le problème le plus sensible qui cause un fonctionnement incorrect d'un circuit asynchrone – nécessite une synthèse exempte d'aléa, ce que n'adresse pas les outils de CAO standard.

Ainsi, une méthodologie de conception propre aux circuits asynchrones est indispensable. Dans ce paragraphe, les méthodologies essentielles de conception de circuits asynchrones de la littérature sont brièvement présentées en abordant d'abord les méthodes de spécification et ensuite les méthodes de synthèse. En conclusion, les méthodologies et les outils associés sont discutés.

1.3.1 Méthodologies de spécification

Les méthodes de spécification et de représentation des circuits asynchrones peuvent se scinder en deux groupes :

- les méthodes basées sur un langage de description de haut niveau tel que VHDL, CHP, Occam, Tangram, Balsa.
- les méthodes basées sur une description de comportement en graphes telle que les réseaux de Petri, les graphes d'états, les STG.

Ces deux groupes diffèrent par leurs méthodes de conception et produisent en général des implémentations de circuit différentes. De manière générale, 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, alors que les méthodes basées sur un graphe sont préférées pour l'analyse temporelle et la synthèse. Il existe un compromis dans le choix de la méthode de

spécification. Les méthodes de spécification basées sur un langage facilitent la tâche de conception d'un circuit (permettent de décrire structurellement et hiérarchiquement le circuit à haut niveau), mais le circuit synthétisé à partir de ces spécifications n'est pas optimal. Si les méthodes de spécification basées sur un graphe s'avèrent ardues et pénibles pour le concepteur (notamment pour des circuits complexes), le circuit synthétisé est rapide et efficace.

La combinaison des deux approches offre alors la possibilité de décrire des systèmes complexes, de les analyser temporellement, de les synthétiser et d'optimiser les circuits générés. Le but de ce paragraphe est de tenter de donner une vue globale des méthodes de spécification de circuits asynchrones en langages de description de matériel et en représentations sous forme de graphes.

1.3.1.1 Méthodes basée sur un langage de description

Les langages qui sont employés pour spécifier des circuits asynchrones incluent CSP [HOAR78], CHP [MART90b], Occam [MAY90][BRUN91], Tangram [BERK91][BERK93], Balsa [BARD97], VHDL [ZHEN98] et Verilog [BLUN00]. 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 séquençement des transitions de signaux. Néanmoins, ils n'utilisent pas le concept de canal de communication, tel qu'il est défini dans §1.2.1. Il faut donc spécifier des paquetages qui permettent au concepteur de définir la communication entre processus concurrents communicants, tel que présenté dans [RENA99][MYER01].

A l'opposé, les autres langages sont dérivés du langage CSP. Ces langages utilisent le même concept que CSP : 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 de 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 Méthodes basée sur un graphe

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 [PETR62], les graphes de transitions de signaux (STG ou « Signal Transition Graph » en anglais) [CHU87], les diagrammes de changements [VARS90], les machines à états asynchrones [YUN92][DAVI93][HUFF54] et les graphes d'états [MULL59].

Le réseau de Petri, présenté plus tard, permet une représentation graphique qui peut décrire des événements concurrents autant que séquentiels. Cette notation est très adéquate pour spécifier le comportement des circuits asynchrones. Par conséquent, de nombreuses méthodes de spécification des comportements des circuits asynchrones utilisent ce type de réseau. A titre d'exemple, les réseaux M (« M-Nets ») proposés par Seitz [SEIT70], les réseaux I (« I-Nets ») définis par Molnar [MOLN85] et les graphes de signaux développés par Yakovlev [ROSE85] sont une classe restreinte de réseaux de Petri (appelée les 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. Etant toujours une classe restreinte de réseaux de Petri, un autre modèle important, qui est actuellement largement étudié et utilisé par la communauté asynchrone, est le graphe de transitions de signaux (STG). Le STG, présenté dans §1.3.3.5, 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 ([CHU87][CORT02][MYER92][YKMA94][SENT92]).

Une méthode alternative est basée sur la machine à états asynchrones de type Mealy (ASM ou « Asynchronous State Machine » en anglais). 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.

Comme les spécifications de circuits asynchrones obtenues avec ces approches sont généralement au niveau des transitions de signaux, l'écriture de telle spécification est ardue et est sujette aux erreurs surtout si la taille du circuit à concevoir est conséquente.

1.3.2 Méthodologies de synthèse

Pour automatiser la synthèse de circuits asynchrones, il est nécessaire d'utiliser une représentation spécifique du circuit. Le coût et la qualité de la synthèse des circuits asynchrones dépendent du type de représentation qui est supporté. De manière générale, les représentations plus flexibles et plus expressives permettent la synthèse de circuits plus rapides et plus complexes, mais la réalisation de tels outils de synthèse est plus ardue en termes de complexité et de ressources. Les spécifications plus restreintes facilitent le processus de synthèse, mais les circuits dérivés peuvent être lents et redondants. Il est à noter que le concepteur devra disposer de méthodes de spécification propres aux circuits asynchrones complexes.

Les méthodologies de synthèse de circuits asynchrones peuvent être grossièrement scindées en trois groupes : synthèse par traduction, synthèse logique et synthèse par compilation.

1.3.2.1 Synthèse dirigée par la syntaxe

Les méthodes de synthèse par traduction commencent par une spécification de circuits sous forme d'un programme dans un langage de description à base de processus concurrents communicants (cf. §1.3.1.1). Elles sont utilisées pour traduire directement les structures du langage en blocs de circuit. On parle alors de synthèses dirigées par la syntaxe. Des exemples de ces méthodes sont celles proposées par van Berkel [BERK98] (cf. §1.3.3.1), Brunvand [BRUN89], Ebergen [EBER91] et Burns [BURN88].

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 2 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 2 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 dans 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 [BURN88b]. 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 n'existe pas de place pour la créativité : une conception pauvre peut facilement donner le même résultat qu'une conception élégante. 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.

1.3.2.2 Synthèse logique

Une méthode alternative – la méthode de synthèse logique – se concentre sur la synthèse de circuits dont la spécification décrit le comportement des signaux du circuit. La spécification utilisée dans cette approche est typiquement basée sur une classe restreinte de réseau de Petri (cf. §1.3.1.2). Le comportement d'un circuit est décrit par un graphe construit en énumérant exhaustivement les états atteignables : le graphe d'états [CHU87]. Le graphe d'états représente le fonctionnement désiré de l'implémentation et peut être mappé sur du matériel. N'étant pas considérée dans l'approche, 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).

Le formalisme le plus utilisé pour spécifier un circuit SI dans la méthode de synthèse logique est le graphe de transitions de signaux (STG). Ces graphes ne peuvent pas directement être synthétisés en circuit sans respecter un certain nombre de contraintes telles que la complétude du codage des états CSC (cf. §1.3.3.5). Les méthodes basées sur les STG – Chu [CHU87], Meng [MENG89][MENG91], Myers [MYER92], Cortadela [CORT02] – possèdent plusieurs limitations. Tout d'abord, la syntaxe des STG est restrictive. Bien que le STG puisse décrire l'opérateur de choix entre des entrées, cette capacité est sévèrement limitée par les conditions structurelles. 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. Deuxièmement, les méthodes basées sur les STG 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 customisés 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 [BEER92] 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.

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.

1.3.2.3 Synthèse par compilation

Une autre approche proposée par Alain Martin dans [MART90b] 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 [BURN88c] est très complexe. Cette méthode sera détaillée dans le paragraphe 1.3.3.3.

1.3.3 Méthodologies et outils de synthèse actuelles des circuits asynchrones

Il existe aujourd'hui quelques méthodes et outils de synthèse de circuits asynchrones. Aucune n'est commercialisée. Ce paragraphe ne se veut pas exhaustif sur l'étude de l'ensemble des méthodologies de conception et des outils de synthèse de circuits asynchrones (ceci pourrait être l'objet d'un document complet), mais se veut un reflet des principales méthodes existantes.

1.3.3.1 Méthode Tangram

Philips développe depuis longtemps des réalisations asynchrones basées sur un système propriétaire Tangram [BERK88][BERK93], dont les succès sont bien connus dans la communauté asynchrone [BERK94][BERK95][GAGE98][KESS00][KESS00b]. Tangram est en fait un ensemble comprenant un langage de description de haut niveau, dit également Tangram, et un compilateur associé qui permet de traduire les structures de langage en des structures de composants « poignée de main » (cf. Figure 20).

Tangram est un langage de description de haut niveau, dont la syntaxe ressemble aux langages de programmation traditionnels tels que C, Pascal. Utilisant les concepts de CSP, Tangram sert à modéliser des processus concurrents communicants par passage de messages synchrones via des canaux de communication point à point.

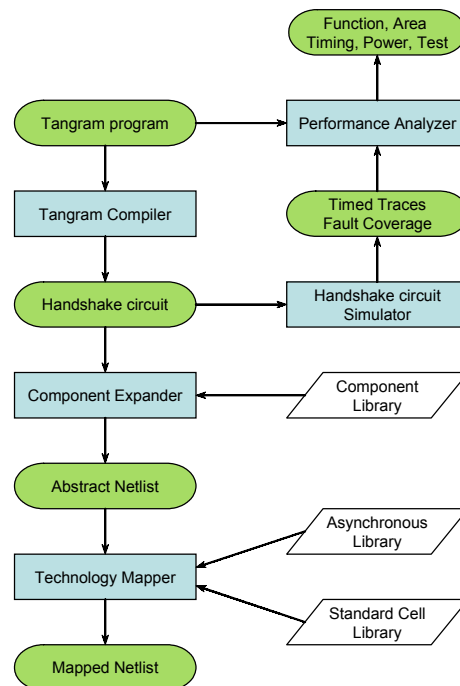


Figure 20 Méthode de Tangram

La méthode de synthèse Tangram utilise un format intermédiaire basé sur les circuits de type « poignée de main » [BERK93]. Les circuits de type « poignée de main » sont ceux qui communiquent avec les autres circuits par des canaux directs utilisant le protocole de communication 4 phases « données groupées ». Grâce à la compilation dirigée par la syntaxe, un circuit asynchrone décrit en langage Tangram est traduit de façon directe 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 et aussi 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 peuvent être spécifiés et conçus a) manuellement, b) en utilisant les méthodes STG (cf. §1.3.3.5) ou c) en utilisant les étapes de la méthode de Caltech (cf. §1.3.3.3).

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. Cependant, comme montré au §1.3.1.1, il est difficile d'optimiser le résultat d'une telle approche de compilation dirigée par la syntaxe. L'optimisation dans la méthode Tangram est au niveau primaire : il s'agit de remplacer les structures des composants « poignée de main » par des équivalents plus efficaces.

1.3.3.2 Méthode Balsa

Cette méthode est développée par l'université de Manchester. Ressemblant tout à fait à la méthode Tangram, Balsa – étant un cadre pour la conception de circuits asynchrones et un langage de description de tels circuits – adopte une approche de la compilation 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 21.

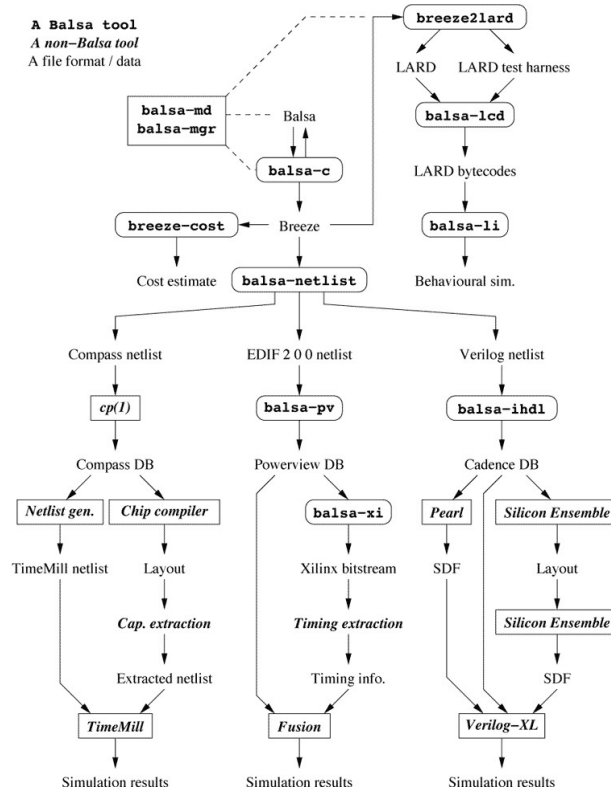


Figure 21 Méthode Balsa

Comme Tangram, le langage Balsa [BARD97][BARD00] 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.

Le format intermédiaire utilisé principalement dans le flot Balsa est le format Breeze qui définit le réseau de circuits « poignée de main ». Breeze sert également aux outils finals à fournir des implémentations pour les spécifications Balsa, ce qui rend les outils finals indépendants des outils frontaux. Dans le système Balsa, la simulation fonctionnelle est effectuée par LARD [ENDE94], un simulateur développé dans le cadre du projet Amulet. La simulation après synthèse est réalisée grâce à des outils commerciaux.

Bien que le succès de cette méthode de conception ait été illustré par la conception du microprocesseur Amulet3i [BARD00b], il apparaît que, comme les autres méthodes basées sur la synthèse dirigée par la syntaxe, l'implémentation générée des circuits n'est pas efficace.

1.3.3.3 Méthode de Caltech

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 [HOAR78], cette modélisation CHP [MART90b] 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.

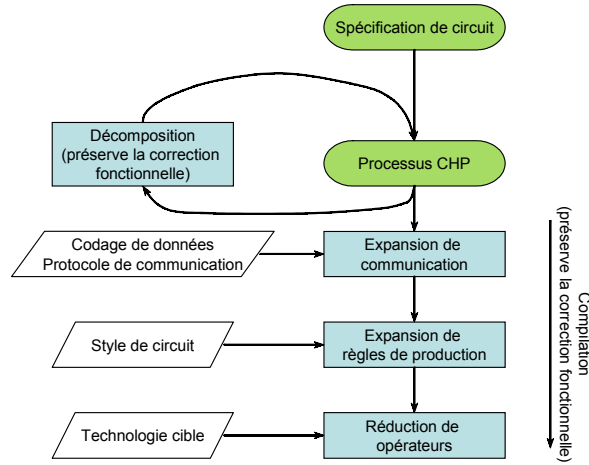


Figure 22 Méthode de Caltech

Le principe de la méthode repose sur la modélisation de chaque processus en termes de communication : au plus haut niveau, le protocole est implicite, seules les actions de communication (lecture, écriture) apparaissent. Le développement du code permet de faire intervenir différents types de protocole, de codage et de conventions (choix du processus actif sur le canal, séquençement de la communication) pour aboutir, par étapes successives, à une description explicite des signaux. Sophistiqués et complexes, les étapes de développement préservent la sémantique mais sont pour la plupart effectuées manuellement (Figure 22).

Etape de la décomposition de processus : chaque processus est itérativement affiné en une collection de processus interactifs simples.

Etape de l'expansion de communication (HSE ou « Handshake Expansion ») : chaque canal de communication est remplacé par les fils explicites conformément au codage choisi. Egalement dans cette étape, chaque action de communication (la lecture et l'écriture) est substituée par des séquences de transitions de signaux respectant le protocole utilisé. Une lecture de canal $E?x$, par exemple, est remplacée par une séquence de transitions de signaux du type :

$$[E^{req}] ; x := \text{données} ; E^{acq} \uparrow ; [\neg E^{req}] ; E^{acq} \downarrow$$

Cela signifie : « attend la requête », « lit les données », « acquitte le canal », « attend la descente de la requête », et « libère le canal ». A cette étape, il est possible d'ajouter des variables et/ou de re-ordonner des transitions de signaux afin d'obtenir une spécification satisfaisant la condition de CSC (cf. §1.3.3.5).

Etape de l'expansion des règles de production : une règle de production (PR ou « Production Rule » en anglais) se compose d'une condition (garde) et d'une action : l'action est exécutée quand la condition est vraie. Dans cette étape, chaque expansion de communication est remplacée par un ensemble de règles de production. A titre d'exemple, $E^{req} \wedge S^{acq} \mapsto S^{req} \uparrow$ et $\neg E^{req} \wedge \neg S^{acq} \mapsto S^{req} \downarrow$. Les règles de production – étant elles-mêmes des processus concurrents simples – spécifient le comportement des signaux internes et signaux de sortie d'un processus. Les gardes doivent assurer que les transitions de signaux se produisent dans l'ordre du programme : la sémantique du programme CHP initial est conservée. Pour cela, il est parfois nécessaire de renforcer les gardes ou d'introduire des variables locales. En outre, les gardes peuvent être modifiées et être réalisées de façon symétrique afin d'obtenir une implémentation de circuit basée sur des cellules standard.

Etape de la réduction des opérateurs : c'est l'étape dans laquelle les PR sont regroupées et associées pour former des composants matériels comme les C éléments généralisés. Les deux PR ci-dessus, par exemple, peuvent être rassemblées pour former une porte de Muller à deux entrées.

Appliquée à plusieurs conceptions dont le microprocesseur asynchrone très connu MIPS R3000 [MART97], la méthode de synthèse du professeur Martin génère des circuits QDI avec le protocole de communication 4 phases. Elle génère des circuits plus optimisés que ceux générés par les autres méthodes de synthèse dirigée par la syntaxe car l'optimisation peut être appliquée au niveau bas lors de la génération des portes. Cependant, elle n'est pas totalement optimisée à notre connaissance.

1.3.3.4 Méthode NCL

« Null Convention Logic » [FANT96][LIGT00] est une logique propriétaire proposée et brevetée par Theseus Logic. Elle est basée sur le codage 3 états (cf. §1.2.2.2.2) et l'utilisation de portes à seuil (en l'anglais « threshold gates »), représentées Figure 23, qui sont en fait des portes de Muller généralisées. La « valeur ajoutée » se situe au niveau de l'optimisation logique : bien que la synthèse ne soit pas sérieusement vérifiée, il semble que les possibilités de projection technologique soient supérieures avec ces portes généralisées.

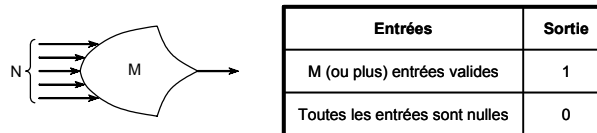


Figure 23 Fonctionnement d'une porte à seuil ($M \leq N$)

La philosophie des circuits générés par la méthode NCL repose sur le motif de base représentée Figure 24. 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.

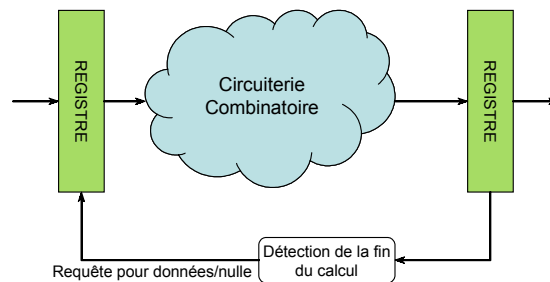


Figure 24 Motif de base de circuits généré par la méthode NCL

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 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). Il 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) : 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 circuit asynchrone gèrent d'une manière ou d'une autre les signaux d'acquittement selon des règles bien définies : souvent, cela représente plus de travail que la partie combinatoire elle-même.

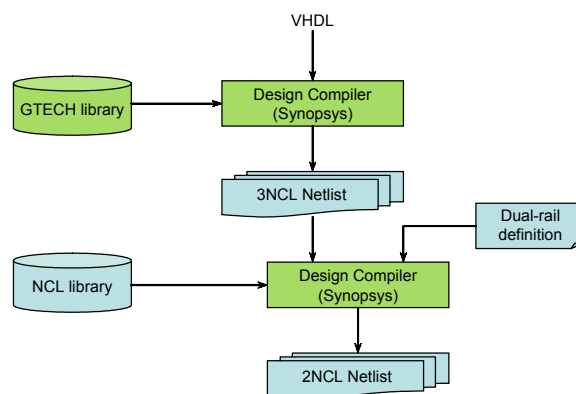


Figure 25 Synthèse dans la méthode NCL

La conception d'un circuit asynchrone décrit en VHDL est synthétisée par l'outil de synthèse commercial Synopsys comme montrée Figure 25. La synthèse [LIGT00] 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 [KOND02]. 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, *a fortiori* si celui-ci 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'acquiescement 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 acquiescements partiels (n'acquiescer que certaines branches du circuit), ni de procéder à toutes les combinaisons complexes où l'acquiescement 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 déjà 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 acquiescements 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'acquiescement est correcte ni qu'elle réalise le schéma de communication souhaité. D'autant plus que la valeur ajoutée de la génération manuelle des acquiescements est nulle, on ne fait que réaliser soi-même un protocole dont la description serait relativement aisée avec un modèle haut niveau.

1.3.3.5 Méthode basée sur les STG (Petrify)

Cette méthode – proposée par [CORT02] – traite la conception des circuits de type « indépendant de la vitesse » (SI) spécifiés par un graphe de transitions de signaux (STG). La Figure 26 représente le flot de conception proposé.

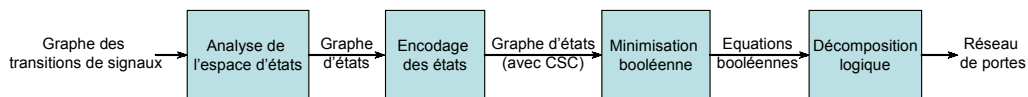


Figure 26 Flot de conception de la méthode STG

Basée sur le réseau de Petri [PETR62][YAKO00], 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 27, est en fait un réseau de Petri limité par les caractéristiques suivantes :

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

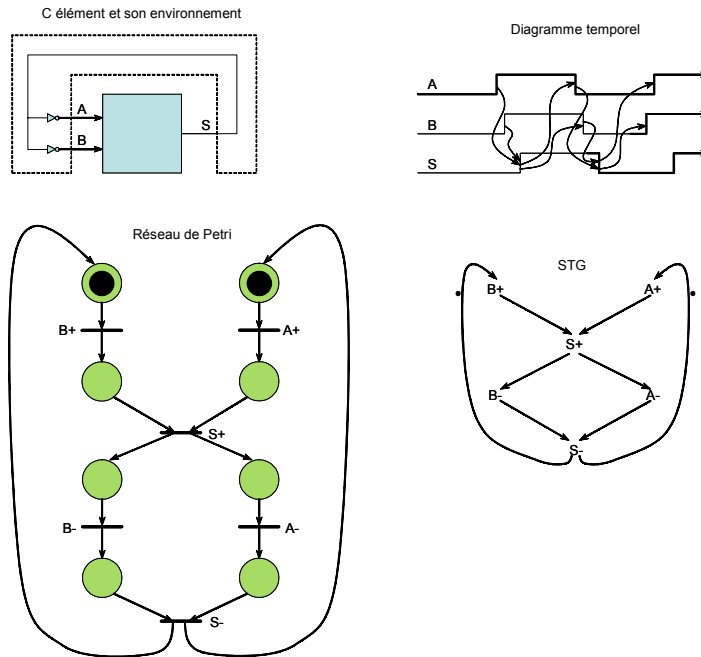


Figure 27 Porte de Muller et son environnement

Dans cette méthode, un circuit asynchrone SI à concevoir 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 (les transitions de signaux). Pour calculer la fonction de chaque sortie du circuit – la tâche de synthèse de circuit – l'espace d'états atteignables ([MULL59]) 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 [KISH94] 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 est dit violer 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.

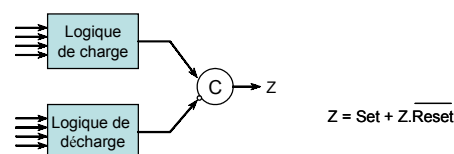


Figure 28 Implémentation en utilisant un C élément

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'entrée 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 (la condition CSC).
4. Choisir un modèle d'implémentation et calculer les équations booléennes pour les fonctions de charge et de décharge (*set* et *reset*). A partir de ces équations, l'implémentation du circuit est effectuée en utilisant des portes atomiques (telles que les portes complexes AOI) 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* [CORT97]. 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.

1.3.3.6 Méthode basée sur ASM (Minimalist)

Cette méthode, proposée par Nowick [NOWI93], traite la conception de contrôleurs asynchrones fonctionnant en mode rafale (cf. §1.2.6.2). 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 : à 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 [UNGE69]). A titre d'exemple, la Figure 29 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.

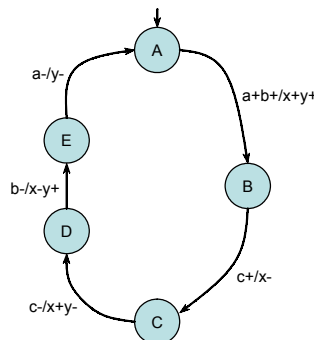


Figure 29 Exemple de la spécification à mode rafale

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 [DAVI93b].

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 ») (cf. Figure 30). Les caractéristiques suivantes font que la machine proposée par Nowick est différente des machines de Huffman.

- 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.

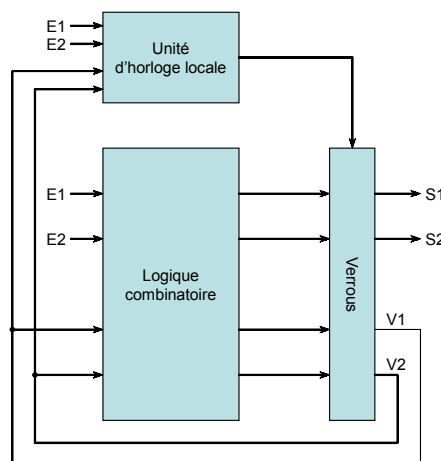


Figure 30 Schéma de la machine auto-synchronisée

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* [FUHR99], 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 [DEAN92], 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

Les problèmes liés à l'horloge dans la conception de circuits synchrones permettent d'ouvrir une voie vers la nouvelle approche de la conception de circuits asynchrones. Contrairement à la synchronisation globale utilisée dans les circuits synchrones, la synchronisation dans les circuits asynchrones est effectuée localement grâce à une signalisation bidirectionnelle entre tous les éléments du circuit. Cette signalisation est implémentée via des canaux de communication avec un protocole de communication et un codage de données. De nombreux types de circuits asynchrones existent : ils sont classés en fonction du modèle de délais pour les portes et pour les fils.

Différentes méthodologies de conception des circuits asynchrones sont présentées. Ces méthodologies sont déterminées par la méthode de spécification et la méthode de synthèse. Deux principales méthodes existent pour spécifier des circuits asynchrones : la première basée sur un langage de description de haut niveau et la seconde basée sur un graphe. Quant aux méthodes de synthèse, elles peuvent se scinder en trois groupes : les synthèses dirigées par la syntaxe, les synthèses par compilation et les synthèses logiques. Plusieurs méthodologies de conception pour les circuits asynchrones sont présentées. Il apparaît qu'aucune méthodologie ne adresse aujourd'hui tous les problèmes de la conception de circuits asynchrones.

Comme on l'a vu précédemment, puisque les langages de description de matériel sont utiles pour concevoir hiérarchiquement les circuits complexes, et que les algorithmes d'analyse temporelle et de synthèse sont plus facilement appliqués aux représentations sous forme de graphes, nous emploierons dans notre outil de synthèse une méthode de spécification combinant les approches basées sur les langages et sur les graphes. Dans TAST [DINH02][DINH02b], un circuit asynchrone est modélisé par un programme décrit dans un langage de haut niveau appelé CHP étendu qui combine les langages CHP et VHDL. Ce programme est ensuite transformé en graphes (réseau de Petri et graphes de flot de données) qui sont utilisés pour la synthèse. Le but de notre approche est donc de tirer parti des avantages d'un langage de haut niveau pour la spécification et des avantages des réseaux de Pétri pour la synthèse.

Chapitre 2

SPECIFICATION SYNTHÉTISABLE DE CIRCUITS ASYNCHRONES

Comme nous l'avons vu précédemment, les méthodes de spécification de circuits asynchrones peuvent se scinder en deux groupes :

- les méthodes basées sur les langages de description de haut niveau,
- les méthodes utilisant des graphes.

Les langages de description de haut niveau sont expressifs et adéquats pour décrire les fonctionnements globaux de systèmes complexes avec une structure hiérarchique et modulaire, sans se soucier du détail de l'implémentation (protocole de communication, modèle de circuits), alors que les graphes sont en général préférés pour l'analyse temporelle et la synthèse. La combinaison des deux approches offre la possibilité de décrire des systèmes complexes, de les analyser temporellement, de les synthétiser et d'optimiser les circuits générés.

Un langage de description de haut niveau est bien adapté à la modélisation de systèmes asynchrones complexes grâce à son niveau élevé d'abstraction. Cependant, de telles descriptions ne peuvent pas être automatiquement synthétisées par les outils de synthèse. Cela signifie que les outils de synthèse et les technologies cibles imposent souvent certaines contraintes ou limitations qu'il faut prendre en compte pour aboutir à une description « synthétisable ». Il faut donc définir un sous-ensemble du langage qui peut être synthétisé.

Dans ce chapitre, la première partie représente le langage de description de matériel utilisé pour spécifier des circuits asynchrones, appelé CHP (« Communication Hardware Processes »). Ce travail constitue une extension du langage présenté initialement par Alain Martin [MART90b]. Celui-ci est enrichi de concepts empruntés à VHDL pour répondre à nos besoins de synthèse et de simulation. La deuxième partie est consacrée à une représentation des circuits asynchrones sous forme de graphes : les PN-DFG (« Petri Net - Data Flow Graph »). Cette combinaison de graphes sert non seulement à vérifier formellement des circuits (avant et après la synthèse) mais aussi à réaliser un lien entre le *front-end* et le *back-end* de l'outil de conception représenté plus tard. Enfin, la dernière partie propose une spécification synthétisable de circuits asynchrones, et explicite la transformation d'un programme CHP en vue de l'obtention d'une implémentation efficace du circuit.

2.1 Langage CHP

Les circuits asynchrones sont constitués par des blocs fonctionnels qui communiquent entre eux par des canaux. Ces circuits peuvent intégrer beaucoup de parallélisme, ce qui implique en général l'utilisation d'un grand nombre de canaux de communication lors de la spécification. Une approche intégrant explicitement le concept de canal de communication est donc particulièrement appropriée. En conséquence, un formalisme choisi pour décrire les circuits asynchrones est basé sur la notion de processus concurrents, qui communiquent entre eux par des passages explicites de messages via des canaux et des assignations explicites de variables. Le formalisme le mieux adapté à ces exigences est le langage CSP (« Communicating Sequential Processes »), conçu et spécifié par Hoare [HOAR78].

CHP est le langage résultant du travail d'Alain Martin [MART90b], qui s'est inspiré du CSP, mais aussi des commandes gardées de Dijkstra [DIJK76]. Dans ce langage CHP, quelques extensions mais aussi quelques limitations ont été ajoutées pour s'adapter à la spécification des circuits asynchrones. Les restrictions de la version du CHP d'Alain Martin sont imposées par la réalisation physique en VLSI. Ce langage est loin de fournir toutes les facilités d'un langage de haut niveau comme C ou VHDL notamment au niveau des types et des opérateurs arithmétiques. Ainsi, il existe un type de donnée unique : le type booléen. Tous les autres types peuvent être créés par l'association de variables booléennes, et l'usage d'opérateurs sur des types autres que booléens fait implicitement référence à des fonctions qui traitent des booléens. La description de systèmes complexes à l'aide de ce langage, comparativement à l'utilisation de VHDL ou Verilog, est donc plus difficile. Malheureusement ces derniers sont des langages sémantiquement synchrones.

Les extensions apportées visent à faciliter la tâche du concepteur et à rendre le langage CHP plus puissant, plus lisible, plus efficace afin qu'il soit adapté à la vérification par simulation et à l'implémentation matérielle par synthèse. En effet, la gestion de l'implémentation des systèmes complexes nous a conduit à ajouter au langage CHP la possibilité d'analyser la consommation, le rayonnement électromagnétique et la sécurité. De plus, la modélisation structurelle était quasi inexistante dans la proposition initiale du CHP, tous les processus étaient déclarés à « plat ». Aussi, afin de pouvoir gérer la complexité des systèmes VLSI, il a été défini une approche de modélisation hiérarchique. La syntaxe présentée en annexe, inspirée des langages de description de matériel, est simple et régulière. La hiérarchie est seulement basée sur des processus et sur des composants et la connectivité entre blocs repose uniquement sur des canaux ; il n'est pas possible de déclarer des signaux. Enfin, il est possible d'effectuer des calculs arithmétiques avec les opérandes de type signé et non signé en multi-rails « 1-parmi-B ».

L'annexe présente concrètement les différents aspects de la syntaxe et de la sémantique du langage CHP, tels que les types de données, déclarations de canaux et de variables locales, les structures de contrôle autant que les opérateurs de composition. L'accent est mis en particulier sur les points nouveaux apportés au langage (voir l'annexe B).

2.2 Représentation intermédiaire d'un circuit asynchrone

2.2.1 Réseau de Petri (« Petri Net »)

Un réseau de Petri [PETE81] est un modèle de graphe bipartite formel pour modéliser des flots d'information et de contrôle dans des systèmes, en particulier les systèmes exposant les propriétés asynchrones et concurrentes. Le réseau de Petri contient deux types de nœud : les rondes (appelés des places) représentent des événements et les barres (dites des transitions) représentent des

conditions. Des jetons (symbolisés par des points noirs) sont affectés à plusieurs places. A tout moment, les positions des jetons dans le réseau définissent l'état du système (marquage).

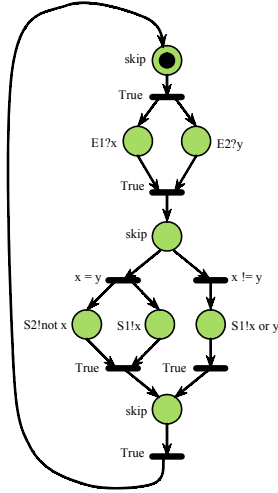


Figure 31 Exemple de réseau de Petri

Une sémantique est associée au réseau de Petri : quand toutes les places précédant une transition portent des jetons et quand la condition dans la transition est vraie, les jetons sont enlevés des places et mis sur les places suivant la transition. Comme les jetons circulent dans le réseau, ils tirent des transitions sur leurs chemins selon des règles prédéfinies, ce qui change l'état du système. De tel mouvement de jetons (jeu de jeton) décrivent les différentes branches d'exécution possibles du système.

Formellement, un réseau de Petri est défini par un quadruplé (P, T, R, M_0) avec :

- P : l'ensemble des places
- T : l'ensemble des transitions
- $R \subset (P \times T) \cup (T \times P)$: la fonction de relation entre places et transitions
- $M_0 : (P \rightarrow \mathbb{N})$ l'état du réseau correspondant à l'initialisation.

La fonction R se définit par un couple $(\text{Pré}, \text{Post})$:

- $\text{Pré} : P \times T \rightarrow \mathbb{N}$ une application d'incidence avant ;
- $\text{Post} : T \times P \rightarrow \mathbb{N}$ une application d'incidence arrière correspondant aux arcs

Pré contient la valeur entière associée à l'arc allant d'une place à une transition alors que Post contient la valeur entière associée à l'arc allant d'une transition à une place.

La fonction R définie dans ce travail de thèse est « $\text{Pré} = 1$ » et « $\text{Post} = 1$ ». Cela signifie que le nombre maximum de jetons pouvant transiter dans un arc est égal à 1. Ce réseau de Petri est appelé *réseau ordinaire* dans la littérature.

Certaines définitions sont données afin de faciliter la lecture de ce manuscrit.

- Un réseau de Petri est dit *borné* si à n'importe quel état du réseau, toutes les places possèdent un nombre borné de jetons. Un réseau borné se caractérise par un espace d'états fini.
- Un réseau de Petri est *vivace* si pour chaque marquage atteignable depuis le marquage initial, il existe une séquence de transitions telle que chaque transition puisse être franchie.

2.2.2 Graphe de flot de données (« Data Flow Graph »)

Le graphe de flot de données représente les dépendances entre des opérations et des données. Supposons que n_{ops} est le nombre d'opérations. Un graphe de flot de données $G_d(V, E)$ est

un graphe directif dont l'ensemble des nœuds $V = \{v_i; i = 1, 2, \dots, n_{ops}\}$ correspond un pour un avec l'ensemble des opérations. Il est supposé que les opérations requièrent un ou plusieurs opérandes et génèrent un ou plusieurs résultats. Par exemple, d'une addition à deux opérandes, il résulte une somme et une retenue. L'ensemble des arcs directifs $E = \{(v_i, v_j); i, j = 1, 2, \dots, n_{ops}\}$ correspond aux transferts de données entre les opérations. A titre d'exemple, la Figure 32 illustre le graphe de flot de données pour l'expression $x := a + b * c * d + 1$.

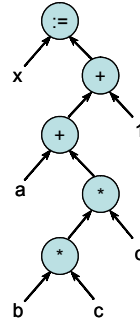


Figure 32 Graphe de flot de données : $x := a + b * c * d + 1$

2.2.3 Combinaison de PN-DFG

La méthode de synthèse proposée opère à partir d'une description comportementale des circuits à réaliser. Cette dernière est exprimée généralement sous une forme intermédiaire. Cette forme sert à exprimer l'ensemble des opérations manipulant des données et un enchaînement impératif de ces données. Ceci conduit à discerner assez rapidement :

- la partie opérative chargée de la transformation des données nécessaires à la réalisation du circuit. Une description flot de données à partir d'un graphe de flot de données (DFG) permet de représenter les dépendances fonctionnelles entre les données et les opérations, tout en optimisant éventuellement les performances et la complexité des opérations à partir de l'analyse de la forme intermédiaire.
- la partie contrôle chargée de piloter l'unité d'exécution en définissant les transformations à effectuer, les données à utiliser, les commandes et les validités de données. Une description de type réseau de Petri s'impose pour caractériser le comportement de ce que l'on qualifie généralement de partie commande ou automate de calcul

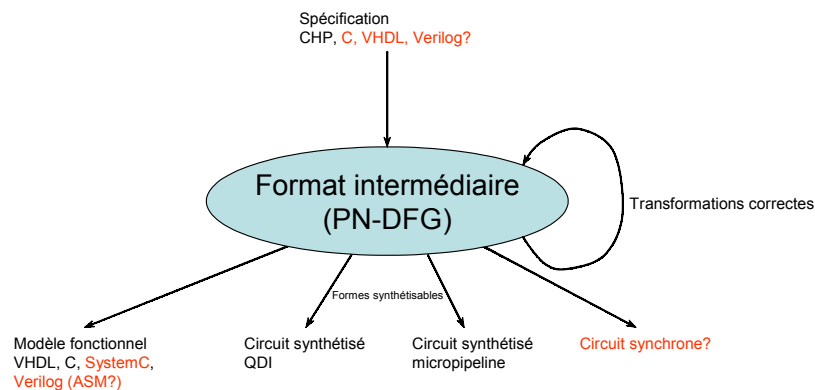


Figure 33 Représentation intermédiaire de circuits asynchrones

La Figure 33 illustre le modèle de représentation intermédiaire de circuits asynchrones. Ce modèle permet de séparer le *front-end* et le *back-end* de l'outil. L'avantage le plus important de l'indépendance entre ces parties est l'extensibilité de l'outil. En effet, on peut facilement ajouter un

outil de compilation d'un autre langage de description dans le *front-end* ou un outil prototypé un autre style d'implémentation de circuits dans le *back-end*, tout en passant par cette forme intermédiaire.

2.3 DTL : spécification synthétisable de circuits asynchrones

De même qu'il existe des règles dites RTL (« Register Transfer Level ») pour la synthèse des circuits synchrones, nous avons défini des règles d'écriture, dites DTL (« Data Transfer Level »), pour les programmes lors de la conception de circuits asynchrones ([DINH02d]). Le RTL spécifie les transferts de données entre les registres. À l'inverse, les programmes DTL indiquent comment les données sont transférées entre les processus, garantissant une synthèse rapide et systématique. La spécification DTL est le résultat de l'analyse des formes synthétisables de circuits asynchrones. De ce fait, on peut considérer les règles DTL comme une spécification synthétisable de circuits asynchrones. Les règles DTL sont des restrictions dans l'écriture des programmes décrivant des circuits asynchrones. Elles figent le style d'écriture d'une description de circuit asynchrone en vue de la synthèse. Un circuit asynchrone, qui remplit les règles de la spécification DTL, est automatiquement synthétisé en cellules standard par la méthode de synthèse proposée dans cette thèse. Toutefois, si la spécification d'un circuit asynchrone n'est pas conforme aux règles DTL, elle peut être transformée par décomposition en un ensemble de processus conformes à la spécification DTL et donc synthétisables (voir §2.3.3).

La spécification DTL est la forme synthétisable de l'outil de synthèse TAST, qui effectue la synthèse des circuits asynchrones en cellules standard.

2.3.1 Analyse des formes synthétisables

Très souvent, les langages de description de circuits sont des langages de spécification de haut niveau, pouvant décrire le comportement de systèmes plus généraux que les circuits électroniques. Aujourd'hui il n'y a pas d'outil de synthèse, capable de synthétiser toutes les constructions de ces langages. C'est pourquoi un groupe de travail pour chaque langage s'est constitué dans le but d'identifier un sous-ensemble des langages qui peuvent être synthétisé automatiquement. C'est le cas pour le sous-ensemble de VHDL et Verilog dit RTL qui donne une spécification synthétisable des circuits synchrones.

De la même façon que le modèle de conception RTL, la spécification DTL d'un circuit asynchrone est venue de l'étude de la spécification et de l'implémentation des éléments de mémorisation. Dans la conception des circuits asynchrones, la mémoire peut apparaître à différents niveaux. Nous classifions la mémoire en trois catégories :

- La mémoire peut être indiquée au niveau du langage. Une mémoire est exigée dès qu'un calcul utilise une variable calculée dans une étape précédente. Nous appelons ce type de mémoire *mémoire fonctionnelle* parce qu'elle provient de la spécification elle-même.
- Une mémoire apparaît également à chaque fois qu'un circuit asynchrone implémente un rendez-vous entre les signaux de données. Notez que nous considérons ici seulement des signaux de données et non pas des signaux d'acquiescement. Ce type de mémoire est nommé *mémoire de données* parce qu'elle s'applique seulement au calcul de données.
- Enfin, la mémoire est aussi nécessaire pour implémenter des protocoles de communication de type « poignée de main » parce que le séquençement des actions de communication est modifié par des machines d'états finis. Nous l'appelons *mémoire de protocole* parce qu'elle est liée au protocole utilisé.

Nous traitons la mémoire de protocole quand nous implémentons le protocole plus loin dans le manuscrit. Quant à la « mémoire de données », elle est constituée d'éléments de mémoires existant dans la bibliothèque de cellules. Par exemple, l'élément C (porte de Muller) est un rendez-vous utilisé dans ce cas. Ainsi, les sous paragraphes suivants se concentrent sur l'identification des éléments mémorisants de type « mémoire fonctionnelle » dans la description d'un circuit asynchrone. Cette identification est à la base de la définition de DTL.

2.3.1.1 Représentation d'un circuit asynchrone

Un circuit asynchrone peut être vu comme une fonction qui réagit sous certaines conditions. Cette réaction prend la forme d'une réponse (la sortie) en fonction de stimuli (l'entrée). Deux cas peuvent être envisagés :

- Le circuit produit un résultat en fonction de la valeur de ses entrées exclusivement : c'est un circuit combinatoire.
- Le circuit produit un résultat en fonction de la valeur de ses entrées et de sa propre évolution au cours du passé : c'est un circuit séquentiel. La notion d'évolution est nécessaire. Elle est liée à l'état courant et aux transitions. Le circuit évolue en effectuant au cours du temps des transitions successives d'un état à un autre.

Dans cet aspect, un circuit est souvent modélisé à l'aide de machines d'états finis de Mealy (on abrégera par FSM – « Finite State Machine »). Une FSM est un sextuplet $M = (I, O, S, s^0, \delta, \lambda)$. Les vecteurs $I = \langle i_1, i_2, \dots, i_n \rangle$ et $O = \langle o_1, o_2, \dots, o_m \rangle$ représentent respectivement les entrées et les sorties de la machine, alors que $S = \langle s_1, s_2, \dots, s_p \rangle$ contient des éléments qui stockent son état courant. Ces éléments sont connus sous le nom de variables d'état. Le symbole $s^0 = \langle s_1^0, s_2^0, \dots, s_p^0 \rangle$ dénote l'état initial de la FSM. Les vecteurs de fonction δ et λ , calculent les nouvelles valeurs pour les variables d'état et pour les sorties, à partir des valeurs courantes des variables d'état et des valeurs des entrées. On associe une composante δ_k ($k=1\dots p$) de δ à chaque variable d'état et une composante λ_j ($j=1\dots m$) de λ à chaque sortie de la FSM.

La particularité de cette représentation vient du fait que S n'est pas le vecteur des états de la FSM, mais celui des variables qui codent ces états. En conséquence, si S_k est le domaine de définition de s_k , et que son cardinal est égal à 2 (codage binaire), alors le nombre total d'états d'une telle machine vaut 2^p .

Dans le cas où l'ensemble des variables d'état est vide, seule la fonction de sortie λ a un sens : le circuit modélisé est combinatoire.

Une correspondance directe peut être faite entre les entrées, les sorties et les états d'une FSM d'une part, et celles d'un circuit d'autre part. La Figure 34 représente schématiquement le modèle formel adopté.

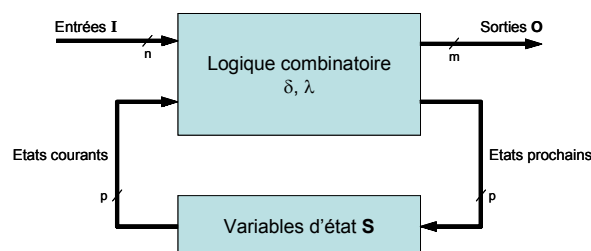


Figure 34 Modèle FSM d'un circuit asynchrone/synchrone

Avant d'aller analyser les formes synthétisables de circuits asynchrones, il est important de rappeler qu'un programme écrit dans un langage de description de haut niveau – décrivant un circuit asynchrone – peut être constitué des processus dont chacun correspond à un bloc fonctionnel du

circuit. Ils communiquent entre eux par passage de messages via des canaux point à point. Ainsi, le terme « processus » dans ce chapitre fait allusion aux processus d'un programme décrivant un circuit.

2.3.1.2 Mémoire causée par une variable locale

Dans cette partie, nous considérons seulement l'utilisation des variables locales aux processus. Une variable déclarée, affectée et lue peut représenter une mémoire fonctionnelle. Deux types de variable sont envisagées :

- Variables de mémorisation : ces variables correspondent à des éléments mémoire du processus. Leurs valeurs dépendent de l'état précédent du processus. Elles sont les véritables mémoires de l'architecture, typiquement ce sont des variables d'état, des registres, des mémoires (L'ensemble des variables d'état S est non vide).
- Variables intermédiaires : au cours de l'exécution du processus, ces variables sont lues sur des canaux d'entrée, peuvent faire l'objet de calculs intermédiaires (commandes gardées, opérations arithmétiques, fonctions complexes, *etc.*) puis émises sur des canaux de sortie (L'ensemble des variables d'état S est vide).

Pour être combinatoire, un processus ne doit pas avoir de variable mémorisante. Cela signifie que son implémentation n'a pas de mémoire fonctionnelle. Un tel processus émet sur ses canaux de sortie des valeurs qui sont exclusivement calculées en utilisant l'information indiquée par ses canaux d'entrée.

Au contraire, le processus qui exige des variables mémorisantes, par définition, est séquentiel. Dans ce cas, le calcul de ses sorties implique les entrées aussi bien que des variables d'état assignées pendant le calcul précédent.

2.3.1.3 Mémoire causée par l'opérateur séquentiel

Évidemment, dans un programme décrivant un circuit asynchrone, l'utilisation de l'opérateur séquentiel, symbolisé par « ; » en langage CHP, peut créer des états, et donc des mémoires fonctionnelles. Par exemple, dans un processus décrit en langage CHP : $*[E?x ; E?x ; S!x]$, le canal d'entrée E est lu deux fois successivement et la dernière valeur lue est ensuite propagée en sortie par l'écriture sur le canal de sortie S . Une variable d'état est nécessaire dans ce processus pour mémoriser les états correspondants à la première et à la deuxième lecture du canal E .

Cependant, tous les opérateurs séquentiels ne produisent pas une mémoire fonctionnelle. Par exemple, dans le processus ci-dessus, l'opérateur séquentiel séparant la deuxième lecture du canal E et l'écriture dans le canal S n'implique pas un état parce qu'il y a une dépendance vraie entre les actions. La séquentialité est assurée par le protocole de communication : la propagation de x depuis le canal d'entrée vers le canal de sortie suffit pour implémenter la séquentialité.

D'un point de vue implémentation, les dépendances vraies n'exigent aucun matériel supplémentaire. Dans un simple half-buffer $*[E?x ; S!x]$, les valeurs d'entrée venant du canal d'entrée seront propagées au canal de sortie. L'opérateur séquentiel n'a dans ce cas aucun coût. Au contraire, la séquentialité a un coût quand les dépendances vraies n'existent pas. Ce coût correspond à la mise en place des états exigés pour assurer le comportement séquentiel.

2.3.1.4 Mémoire causée par l'opérateur parallèle

L'utilisation de l'opérateur parallèle peut entraîner l'apparition de mémoires fonctionnelles s'il existe une variable partagée en écriture. Une variable est dite partagée en écriture si elle est affectée, alors qu'elle est lue ou modifiée dans une branche d'exécution concurrente. Ainsi, une

variable partagée en écriture est déclarée non synthétisable. Cependant, une variable seulement partagée en lecture n'exige aucune mémoire puisqu'elle sert de variable intermédiaire.

D'une façon identique, un canal de communication partagé en écriture n'est pas synthétisé. De plus, un canal partagé en lecture n'est pas non plus synthétisé car cela viole la propriété de communication point à point.

Hormis ces restrictions, les opérateurs parallèles n'induisent aucune mémoire.

2.3.1.5 Initialisation d'un système

Dans la modélisation d'un système asynchrone avec des processus concurrents communicants, deux types d'initialisation sont envisagées : l'initialisation de variable locale et l'initialisation de canal de communication.

- L'initialisation de variable locale est très courante : elle permet de confirmer correctement l'état initial d'un processus.
- L'initialisation de canal de communication permet d'envoyer une valeur initiale à un canal de sortie d'un processus une seule fois, avant que le processus ne commence une exécution bouclée infiniment. Il est nécessaire de pouvoir initialiser les flux de données lorsque des processus concurrents communicants sont bouclés. Sans cela, le circuit serait bloqué.

Les deux types d'initialisation exigent des mémoires fonctionnelles. Dans le cas de l'initialisation d'une variable locale, cette dernière représente un élément mémorisant. Ainsi, ce type d'initialisation n'est pas synthétisable. De la même façon, l'initialisation d'un canal de communication implique une écriture séquentielle sur un canal de sortie, ce qui exige un état. Cependant, cette initialisation peut être implémentée de façon particulière en ajoutant un opérateur d'initialisation sur le canal de sortie. Le paragraphe §5.1.2.2 représentera l'implémentation de cet opérateur d'initialisation.

2.3.2 Règles DTL

Basé sur l'analyse des éléments de mémorisation impliqués dans la réalisation d'un circuit asynchrone, nous avons défini un ensemble de règles de modélisation et d'écriture, afin que le circuit puisse être réalisé par une FSM conforme au modèle présenté dans la Figure 34. Au sein d'un processus, nous avons défini la spécification DTL comme une spécification qui limite l'utilisation de variables d'état et donc de la mémoire fonctionnelle. Un programme de description d'un circuit asynchrone est dit conforme à la spécification DTL si il est seulement composé de processus combinatoires et de processus ayant des initialisations de canal de communication. Par conséquent, pour être conforme à l'écriture DTL, un processus doit respecter toutes les règles suivantes :

- 1. Les variables partagées ne sont pas autorisées sauf dans le cas où les variables sont consultées uniquement.**
- 2. Les variables doivent être affectées avant d'être lues.**
- 3. Les valeurs envoyées aux canaux de sortie doivent exclusivement dépendre des valeurs reçues sur les canaux d'entrée.**
- 4. Les mêmes canaux ne peuvent être accédés séquentiellement que dans le cas où les canaux de sortie sont initialisés au début du processus.**
- 5. Les mêmes canaux ne peuvent être accédés concurremment.**
- 6. Deux instructions séquentielles doivent exprimer une dépendance vraie.**

Justifications :

Les règles 1-3 restreignent les utilisations de variables de mémorisation. L'utilisation de variables partagées, si elle ne viole pas la sémantique de conflit, exige une mémoire pour mémoriser l'ancienne valeur. Une variable et un canal doivent avoir leur propre valeur avant d'être consultés. Ils ne gardent pas leur valeur au cours de leur évolution. Par conséquent, admettre les règles 1-3 signifie que seulement l'utilisation des variables intermédiaires est autorisée. Utiliser uniquement les variables intermédiaires permet d'avoir un processus de type consommation – production : toute valeur émise sur un canal de sortie est toujours une fonction de valeurs lues sur les canaux d'entrée. D'après la définition (voir §2.3.1.1), ces processus sont qualifiés de processus combinatoires.

Ensuite, la restriction sur l'opérateur séquentiel (règles 4 et 6) permet de déduire de manière systématique, la relation entre les canaux d'entrée et de sortie. Il ne peut pas exister de conflit d'état entre deux protocoles successifs sur un même canal. La règle 5 garantit qu'il n'y a pas de conflit de données sur un même canal. C'est à dire qu'un canal d'entrée ne peut être consommé deux fois concurremment. De même, un canal de sortie ne peut envoyer deux valeurs différentes concurremment.

Une fois, les règles sus-citées satisfait, un programme décrivant un circuit asynchrone peut être synthétisé systématiquement en cellules standard par la méthode de synthèse proposée dans les chapitres suivants. Les règles DTL formalisent la forme synthétisable des programmes décrivant des circuits asynchrones. Deux problèmes se posent à nous à ce niveau :

- Que faire si un programme viole une de ces règles ?
- Permettent-elles ces règles aux concepteurs de décrire n'importe quels circuits asynchrones ?

Les sections suivantes répondent à ces questions.

2.3.3 Transformation d'un programme en DTL

Un processus, spécifiant un bloc fonctionnel d'un circuit asynchrone, qui ne satisfait pas les conditions requises pour être DTL peut être transformé en un ensemble de processus compatibles DTL par décomposition. La décomposition considère deux types de processus : les processus ayant une ou des variables de mémorisation et les processus ayant un ou des opérateurs séquentiels. Nous avons choisi le langage CHP pour illustrer cette décomposition, mais ce choix n'influe pas sur l'idée générale proposée. La méthode peut s'appliquer également à tout autre langage de processus communicants tel que Tangram, Balsa.

2.3.3.1 Extraction des variables de mémorisation

Supposons que l'on veuille modéliser un élément mémorisant x dans un processus P et que ce dernier ait un certain nombre d'entrées et de sorties. On extrait cet élément du processus P par la création d'un nouveau processus associé à cet élément afin d'obtenir un fonctionnement de type machine d'états finis comme représenté Figure 34 (calcul des sorties de P en fonctions des entrées et de l'état courant et calcul de l'état suivant). Le processus P n'a donc plus de variable de mémorisation. Le processus modélisant les éléments mémorisants peut s'écrire de deux manières différentes :

- **Comme un registre**

```

Regx ≡ *[RWx?rw;
    [rw = 0 => Wx?x      -- écriture de x
    @ rw = 1 => Rx!x      -- lecture de x
    ]
    ]
    
```

Ce type de processus n'est pas DTL tel que défini dans §2.3.2 puisqu'il contient la variable x qui est un élément mémoire. Cependant, si le processus Reg_x est synthétisé et inclus, une fois pour toute, dans la bibliothèque de cellules, alors la synthèse d'un circuit ayant un tel processus est directe.

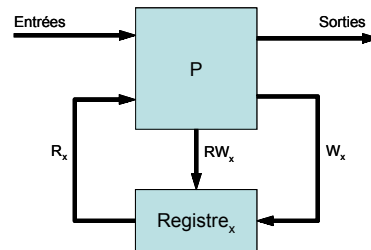


Figure 35 Modélisation d'un élément mémoire comme un registre

Le processus d'origine contenant la variable x est alors transformé de la manière suivante :

- Tout accès en écriture à la variable x est remplacé par la séquence [$RW_x!0, W_x!x$]
- Tout accès en lecture à la variable x est remplacé par la séquence [$RW_x!1, R_x?x$]

La Figure 35 présente la vue équivalente du processus P après extraction de la variable. La variable est dorénavant stockée dans le registre et les sorties de P ne sont plus fonctions que des entrées et du contenu du canal R_x . Le processus P est alors conforme à la spécification DTL.

- **Comme une variable tournante**

```

Bouclex ≡ [Cx!0;
    *[ Nx?x ; Cx!x ]
    ]
    
```

Dans ce cas, le processus possède un canal d'entrée C_x qui lit la valeur courante de x et un canal de sortie N_x qui produit la prochaine valeur de x . En fait, ce processus est équivalent à un « full-buffer » avec initialisation du canal de sortie.

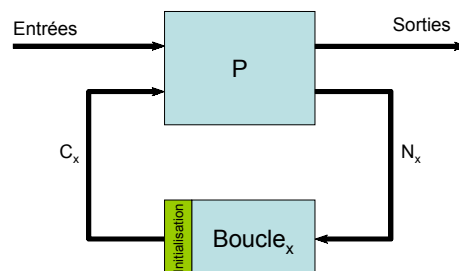


Figure 36 Modélisation d'un élément mémoire comme une variable tournante

La Figure 36 donne la vue schématique du processus P avec l'élément de mémoire x modélisé comme une variable tournante. Contrairement à la modélisation précédente où la variable était mémorisée dans un registre qui peut être accédé aléatoirement en mode écriture ou lecture, la variable est calculée par le processus P et mise en attente dans la boucle de retour jusqu'à sa future utilisation. Dans cette modélisation, l'effet mémoire ne provient donc plus d'une véritable ressource mémoire mais d'un simple processus qui lit et retransmet la donnée. Le processus de retour (Boucle_x) possède une séquence d'accès fixe : « envoyer une donnée sur C_x » qui est toujours suivie

par « recevoir une autre donnée sur N_x ». Ainsi, une mise à jour de x correspond à l'acquiescement de la valeur actuelle dans $Boucle_x$ ($C_x?$) et à la réception d'une nouvelle valeur calculée par P ($N_x!x$). Réciproquement, une consultation de x correspond à la consommation de x et à la réécriture de sa valeur pour qu'elle soit disponible à l'accès suivant. Dans ce cas, une optimisation consiste uniquement à consulter le contenu de la variable pour éviter de réécrire sa valeur, permet de limiter l'activité et donc d'optimiser la consommation.

Le processus P contenant la variable x est alors transformé de la manière suivante :

- Tout accès en écriture à la variable x est remplacé par la séquence [$C_x?$, $N_x!x$]
- Tout accès en lecture à la variable x est remplacé par la séquence [$C_x?x$; ... ; $N_x!x$] ou $\#C_x$ dans le cas optimisé

Il est à noter que pour éviter le problème blocage généré dans un cycle fermé (tel que celui créé par les deux processus P et $Boucle_x$ ci-dessus), ce dernier doit contenir au moins 3 « demi-buffer ». L'initialisation du processus $Boucle_x$ est en fait un demi-buffer ajouté dans la boucle (il y a déjà deux « demi-buffer » : un pour le processus P et l'autre pour le processus $Boucle_x$ lui-même), ce qui la rend non bloquée.

En résumé, nous avons montré deux façons de modéliser un élément mémoire dans un processus décrivant un circuit asynchrone. Les deux méthodes de modélisation que nous présentons ici en langage CHP sont générales. Elles offrent des compromis intéressants et ne s'utilisent pas dans les mêmes conditions. La modélisation d'un élément mémoire en un processus de retour est particulièrement efficace lorsque le processus d'origine doit effectuer une lecture et une écriture de la variable dans le même cycle. Ceci s'explique par le fait que le processus $Boucle_x$ se comporte comme un registre de pipeline. De plus, il est possible d'effectuer le calcul dans le processus de retour, ce qui simplifie le processus d'origine P . La modélisation en registre est intéressante pour un fonctionnement de type registre lorsque les accès aux éléments sont aléatoires en lecture et écriture.

• Illustration des 2 méthodes sur un exemple

Dans ce paragraphe, nous présentons les deux méthodes d'extraction de variable de mémorisation sur l'exemple d'une mémoire. La Figure 37 représente le code CHP fonctionnel de la mémoire et son schéma. Le canal RW (« Read-Write ») contient l'information de contrôle lecture et écriture, le canal AD (« Address ») l'adresse mémoire, DW (« Data Write ») la donnée à écrire et enfin DR (« Data Read ») la donnée lue. Supposons que cette mémoire dispose de deux éléments (on peut réaliser *a priori* n'importe quelle taille de mémoire).

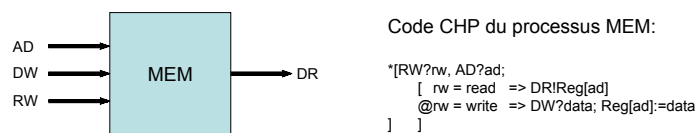
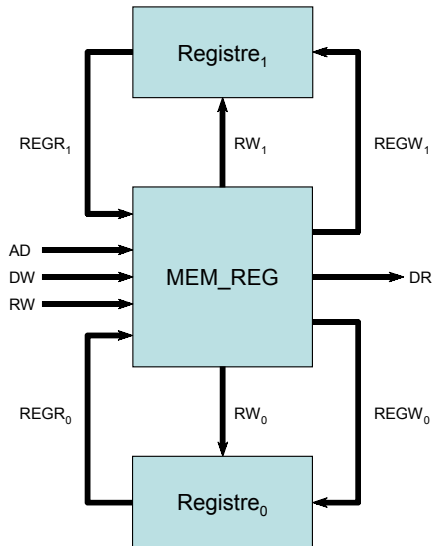


Figure 37 Exemple d'une mémoire

Evidemment, le processus CHP de cette mémoire n'est pas conforme aux règles DTL : la règle 3 est violée. En effet, la variable Reg est une variable de mémorisation. Sa valeur, envoyée au canal de sortie DR dans la phase lecture, dépend de la valeur stockée précédemment. Par conséquent, cette variable doit être extraite afin que le processus MEM puisse être conforme aux règles DTL et donc synthétisable.

Comme proposées ci-dessus, deux techniques sont possibles pour extraire la variable Reg . La première consiste à extraire la variable sous forme des registres. Deux registres sont nécessaires dans ce cas, ce qui est représenté Figure 38.



Code CHP du processus MEM_REG:

```

*[RW?rw, AD?ad;
  [ rw = read  => [ ad = 0 => RW0!rw, REGR0?data; DR!data
                    @ ad = 1 => RW1!rw, REGR1?data; DR!data
                  ]
  @ rw = write => [ ad = 0 => DW?data; RW0!rw, REGW0!data
                    @ ad = 1 => DW?data; RW1!rw, REGW1!data
                  ]
] ]
    
```

Code CHP du processus Registre0:

```

*[RW0?rw;
  [ rw = read  => REGR0!data
    @ rw = write => REGW0?data
  ] ]
    
```

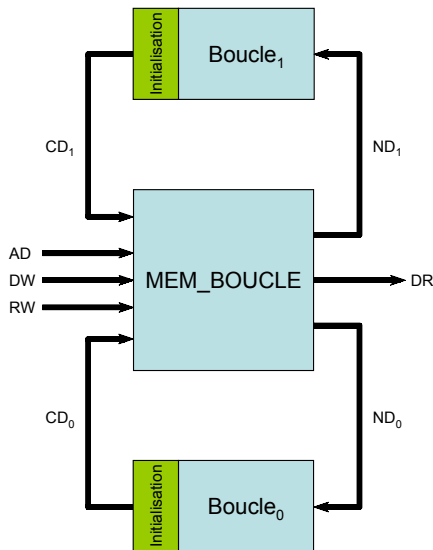
Code CHP du processus Registre1:

```

*[RW1?rw;
  [ rw = read  => REGR1!data
    @ rw = write => REGW1?data
  ] ]
    
```

Figure 38 Extraction de la mémoire : utilisation de registre

La deuxième solution est d'utiliser des processus tournants. La décomposition de la mémoire est montrée Figure 39. Les informations à conserver sont actuellement stockées dans les processus bouclés.



Code CHP du processus MEM_BOUCLE:

```

*[RW?rw, AD?ad;
  [ rw = read  => [ ad = 0 => CD0?data; DR!data, ND0!data
                    @ ad = 1 => CD1?data; DR!data, ND1!data
                  ]
  @ rw = write => [ ad = 0 => DW?data, CD0?; ND0!data
                    @ ad = 1 => DW?data, CD1?; ND1!data
                  ]
] ]
    
```

Code CHP du processus Boucle0:

```

[ CD0!0;
  *[ND0?data; CD0!data]
]
    
```

Code CHP du processus Boucle1:

```

[ CD1!0;
  *[ND1?data; CD1!data]
]
    
```

Figure 39 Extraction de la mémoire : utilisation de processus tournant

2.3.3.2 Extraction des opérateurs séquentiels

Des accès séquentiels à un canal ne sont pas autorisés dans la spécification DTL puisqu'ils imposent de la mémoire. Certaines solutions sont possibles pour extraire les opérateurs séquentiels et transformer le programme en machines à états comme celles de la Figure 34. En particulier, on peut encoder les états associés aux opérateurs séquentiels, puis les extraire comme des variables tournantes. Le principe de ces solutions est le suivant.

Supposons que l'on veuille modéliser le séquençement de deux blocs d'instructions conformes à la spécification DTL : $P \equiv *[A; B]$

On affecte un état à chaque action qui suit un opérateur séquentiel. Un état initial est alors réservé pour exécuter l'ensemble des actions qui précèdent le premier opérateur séquentiel. On a

donc un seul état requis pour le bloc B, en plus de l'état initial. La variable d'état peut être extraite sous forme de variable tournante, on obtient alors :

```
P1 ≡ *[CS?cs ;
    [cs = 0 => A, NS!1
    @cs = 1 => B, NS!0
]
P2 ≡ [CS!0 ; *[ NS?ns ; CS!ns]]
```

Ici, le processus d'origine P est équivalent à deux processus parallèle P1 et P2 : $P \equiv P1 // P2$. Le processus P1 prend en charge l'exécution des blocs d'instructions du processus initial du décodage de l'état courant et du calcul de l'état suivant. Le processus P2 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 1. Certaines techniques sont proposées pour réduire le nombre d'états nécessaires et l'activité des processus décomposés. Elles permettent d'optimiser la consommation et d'augmenter la performance du circuit. La différence entre ces techniques porte en particulier sur la manière d'affecter et d'encoder les états du processus initial.

Pour illustrer clairement ces techniques d'amélioration, on prend un exemple d'un décodeur d'instructions. Selon la valeur du canal G, différentes séquences d'ordres sont générées :

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

La première solution est la solution de base où on code les états associés à B et D. On obtient le processus P1 et P2 suivants :

```
P1 ≡ *[ CS?cs ;
    [cs = 0 => G?g ;
        [g = 0 => A, NS!1
        @g = 1 => C, NS!2
        @g = 2 => E, NS!0
        ]
    @cs = 1 => B, NS!0
    @cs = 2 => D, NS!0
]
P2 ≡ [CS!0 ; *[ NS?ns ; CS!ns]]
```

Notez que le nombre d'états requis peut être réduit si certains des blocs sont factorisés dans plusieurs commandes gardées.

On remarque dans cette solution que B et D sont dans les gardes différentes de G. On peut donc encoder directement les opérateurs séquentiels dans chaque garde par des variables d'état. L'introduction des variables d'état dans le processus P nous donne :

```
P ≡ *[ G?g ;
    [g = 0 => [A, ns:=1]; [B, ns:=0]
    @g = 1 => [C, ns:=1]; [D, ns:=0]
    @g = 2 => E
]
]
```

On extrait la variable d'état *ns* et on obtient les processus P1 et P2 suivants :

```

P1 ≡ *[ [#G = 0 => CS?cs;
        [cs = 0 => A, NS!1
        @cs = 1 => B, NS!0, G?]
    @#G = 1 => CS?cs ;
        [cs = 0 => C, NS!1
        @cs = 1 => D, NS!0, G?]
    @#G = 2 => E, G?
]      ]
P2 ≡ [CS!0 ; *[ NS?ns ; CS!ns]]

```

Le nombre d'états dans ce type de solution est le nombre maximum d'opérateurs séquentiels sur les branches du processus P, plus l'état initial.

Le fait que G soit toujours acquitté au dernier état permet donc d'éviter de mémoriser tous les états du processus. Il faut noter aussi que le canal G est consommé à la fin de l'exécution des actions de P1, tandis que le programme initial exécute cette lecture au début. Ceci peut facilement être corrigé en ajoutant une mémoire tampon sur le canal G.

De plus, la dernière garde [$@\#G = 2 \Rightarrow E$] qui n'était pas séquentielle à l'origine n'est pas complexifiée inutilement dans ce cas. Elle reste simple car elle ne consomme pas d'état initial. Ceci permet d'optimiser la consommation et les performances des cas simples et favorables.

• Exemple

Pour illustrer la méthode d'extraction des opérateurs séquentiels, nous présentons d'ici un exemple dans lequel les deux valeurs consécutives sur un canal d'entrée sont lues, calculées et puis émises sur un canal de sortie.

```

P ≡ *[E?x ; E?y ; S!f(x, y)]

```

Comme l'énonce précédemment, deux états sont nécessaires pour ce processus P : un état pour coder les actions suivant l'opérateur séquentiel et un état initial. La variable d'état est extraite par un processus tournant. En outre, nous devons mémoriser la valeur de la première lecture du E afin de calculer la sortie ultérieurement. Cette valeur est stockée en utilisant aussi un autre processus tournant. Par conséquent, le processus P est décomposé comme la montre Figure 40.

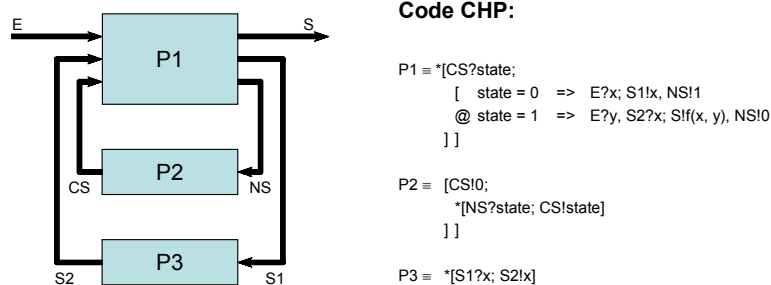
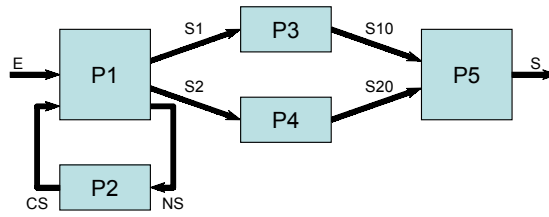


Figure 40 Exemple de la décomposition de processus

Une autre solution pour mémoriser les valeurs des lectures du canal d'entrée est d'utiliser deux processus concurrents. Un démultiplexeur sert à distribuer les valeurs lues dans deux processus différents et à la sortie, un multiplexeur est utilisé pour regrouper ces valeurs et pour calculer la fonction de sortie. Cette décomposition est illustrée dans la Figure 41.


Code CHP:

```

P1 = *[CS?state;
    [ state = 0 => E?x; S1!x, NS!1
      @ state = 1 => E?y; S2!y, NS!0
    ] ]

P2 = [CS!0;
    *[NS?state; CS!state]
  ]

P3 = *[S1?x; S10!x]

P4 = *[S2?y; S20!y]

P5 = *[S10?x0, S20?y; S!f(x, y)]
    
```

Figure 41 Autre solution pour la décomposition de processus

2.3.4 Conclusion sur la spécification DTL

Nous avons défini la spécification DTL. C'est une forme synthétisable des langages basés sur les processus concurrents communicants qui est analogue à la spécification RTL des langages de description de matériel (VHDL, Verilog). La spécification DTL restreint l'utilisation des variables de mémorisation et des opérateurs séquentiels qui requièrent des variables d'état. Ces restrictions proviennent directement de l'analyse des éléments mémorisants dans une spécification de circuits asynchrones.

Les méthodes de transformation de programmes, au cas où ils ne sont pas conforme à la spécification DTL, en programmes compatible DTL sont également proposées. Ces transformations permettent d'extraire les variables de mémorisation et les opérateurs séquentiels. Elles mènent alors à des programmes sans mémoire fonctionnelle. Le programme obtenu en appliquant ces transformations est intrinsèquement un programme de type machine d'états. Il se compose de manière parallèle d'un processus combinatoire – qui calcule les sorties du programme et les prochains états à partir des entrées et des états courants – et d'un processus propageant les états.

La spécification DTL ne dépend ni du langage de description, ni du modèle d'implémentation des circuits. Le langage CHP sert à illustrer les transformations de programmes, mais ce choix n'influe pas sur la généralité des transformations. N'importe quel langage de description de matériel basé sur des processus concurrents communicants (tel que Tangram, Balsa) peut parfaitement être adapté. De plus, aucune supposition n'est faite sur le modèle d'implémentation des circuits. Le DTL permet de modéliser un grand nombre de modèles de circuits asynchrones (QDI, SI, micropipeline, ...) et de cibler plusieurs implémentations matérielles (cellules standard, cellules complexes, FPGA, ...). De ce fait, la spécification DTL définit un style d'écriture général des circuits asynchrones. De façon anecdotique on peut remarquer que cibler des circuits synchrones est aussi possible.

Elle fournit aussi un modèle de description transparent qui permet au concepteur de contrôler l'architecture du circuit à travers la modélisation. Autrement dit, il est facile de prévoir le circuit implémenté sous-jacent au programme DTL. Par ailleurs, le pipeline et la re-synchronisation peuvent être accompli au niveau langage. Le style DTL est suffisamment flexible pour permettre le contrôle de la vitesse du circuit aussi bien que sa consommation. Cet aspect sera discuté dans le chapitre suivant.

Le style DTL permet une génération directe et une compilation rapide de différents types de circuits asynchrones, sans exiger l'exploration de l'espace d'états au niveau signal. Cet avantage de la spécification DTL sera détaillé dans le Chapitre 4 plus tard.

Enfin, la spécification DTL assure l'existence d'une implémentation QDI en portes logiques.

2.4 Conclusion

Dans ce chapitre, nous avons abordé des spécifications de circuits asynchrones à différents niveaux.

- Au niveau conception, le langage de description de haut niveau CHP est introduit. Ce langage – une extension des travaux du professeur Alain Martin et de ses collègues – permet de faciliter la description de circuits asynchrones en vue de la vérification par simulation et de l'implémentation matérielle par la synthèse. Grâce à lui, les communications entre des processus concurrents communicants sont exprimées à travers des messages au sein des canaux.
- Au niveau de la représentation interne, la structure combinée de réseaux de Petri et de graphes de flot de données est proposée pour représenter de manière intermédiaire les circuits asynchrones vis-à-vis non seulement de la simulation et de la synthèse mais aussi de la vérification formelle. Cette structure est obtenue par la compilation du langage CHP.
- Au niveau implémentation matérielle, ce chapitre contribue à la définition des règles DTL – une spécification synthétisable de circuits asynchrones. Comme la spécification RTL pour les circuits synchrones, la spécification DTL restreint l'utilisation des éléments mémorisants. Ainsi, un programme décrivant un circuit asynchrone conforme à la spécification DTL ne possède pas de mémoire fonctionnelle. Il peut être systématiquement synthétisé en portes par la méthodologie de synthèse proposée et automatisée dans les chapitres qui suivent. Des programmes non conformes à la spécification DTL peuvent être transformés afin qu'ils puissent être synthétisés par la suite. Se basant sur l'extraction des éléments qui induisent des mémoires, diverses techniques de transformation ont été données. La spécification DTL montre qu'elle peut être appliquée aux différents langages de description, différentes classes de circuits asynchrones et différentes implémentations finales.

La spécification DTL a été intégrée comme la forme synthétisable des circuits asynchrones dans l'outil de synthèse TAST qui est développé dans le cadre de ce travail de thèse pour prototyper la synthèse de circuits asynchrones QDI. La propriété synthétisable de la spécification d'un circuit asynchrone, représentée sous forme d'une combinaison de réseaux de Petri et de graphes de flot de données, est vérifiée. Cette vérification correspond à un module dédié « CheckDTL » qui est abordé plus tard dans le Chapitre 6.

Chapitre 3

MODELE DES CIRCUITS CIBLES

Dans les circuits synchrones, les problèmes de propagation et de distribution d'horloge deviennent de plus en plus difficiles à résoudre. En particulier, comme le décrit [DALL98], ces problèmes affectent facilement les circuits complexes fonctionnant à fréquence élevée. Il faut donc explorer des approches alternatives. Une approche viable, de plus en plus commercialisée, est la conception de circuits asynchrones. Parmi les nombreuses catégories de circuits asynchrones développés, les circuits pipelinés à grain fin ont démontré des performances en vitesse élevée [SUTH01], [NYST01], [SING00], [SING00b], [SING01], [LINE98], [FERR02], [OZDA02], [OZDA02b] et [TUGS02]. De plus, ce type de circuits permet d'éviter la génération, l'optimisation et la vérification de contrôleurs complexes [YUN98]. Ces structures implémentent un contrôle distribué composé de contrôleurs élémentaires de faible complexité.

Afin de trouver une méthode pour réaliser des circuits et des systèmes asynchrones à grande vitesse et à faible consommation, nous étudions les techniques de conception des pipelines asynchrones. Dans ce chapitre, nous présentons tout d'abord le mécanisme du pipeline asynchrone. Une étude aborde dans la suite les travaux sur les pipelines asynchrones. Nous concluons ce chapitre par la proposition d'un modèle de circuit qui constituera la cible de la méthode de synthèse de circuits QDI proposée.

3.1 Pipeline asynchrone

Généralement, un système asynchrone se compose d'un ensemble de composants fonctionnels qui communiquent localement entre eux. Ils utilisent un protocole de communication de type « poignée de main » via des canaux. Pour augmenter le débit de sortie du système avec un faible surcoût en surface, les composants du système peuvent être arrangés en plusieurs étages concurrents. Cette décomposition s'appelle un pipeline. A titre d'exemple, une pile de type FIFO (« First-In, First-Out ») dans laquelle les jetons, envoyés à l'entrée, traversent chaque étage et atteignent la sortie dans l'ordre « premier arrivé, premier sorti » constitue un pipeline.

Dans un pipeline synchrone, les jetons avancent d'un étage à l'autre à chaque cycle d'horloge. Au contraire, dans un pipeline asynchrone, il n'y a 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.

Comme montré au paragraphe 1.2, il existe un grand nombre de catégories de circuits asynchrones. Celles-ci dépendent :

- de la taille des composants fonctionnels
- du parallélisme des protocoles de communication de type « poignée de main »
- du codage de données
- des hypothèses sur le temps qui assurent le fonctionnement correct du circuit.

Dans ce chapitre, nous étudions les pipelines asynchrones QDI utilisant le protocole 4 phases avec le codage de données « 1-parmi-N ». Ce codage est insensible aux délais qui inclue le codage double-rails. L'extension à l'encodage « M-parmi-N » est également possible.

3.1.1 Définitions

3.1.1.1 Mécanisme de pipeline

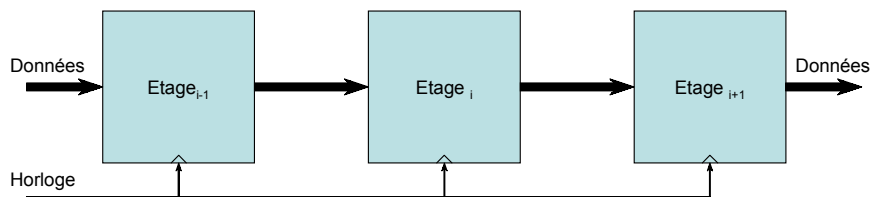


Figure 42 Pipeline synchrone

La Figure 42 illustre un pipeline synchrone. L'occurrence d'un front d'horloge sur tous les registres de pipeline provoque le déplacement des données. L'horloge impose donc une synchronisation relativement forte des données entre elles. Une fois entrées dans le pipeline, deux données sont toujours séparées du même nombre d'étages. Ce n'est pas le cas en asynchrone.

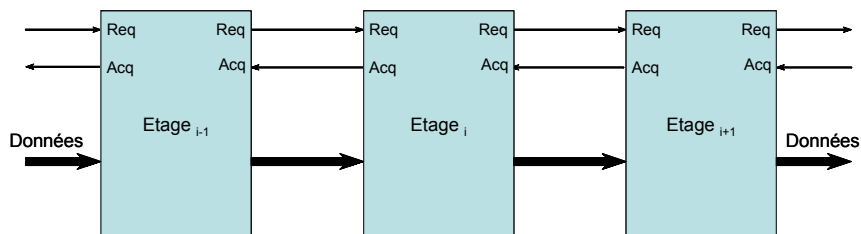


Figure 43 Pipeline asynchrone

La Figure 43 représente le mécanisme d'un pipeline asynchrone. Le signal de requête est utilisé pour signaler à un étage que les données valides sont disponibles sur ses entrées et qu'il peut effectuer son calcul. Ceci fournit une faible consommation à grain fin car il n'y a aucun calcul à exécuter quand aucune requête n'est produite. L'absence du signal de requête verrouille le pipeline en aval. Cela nous impose alors de bloquer les données en amont en attendant de traiter des données courantes. Ceci est assuré par l'attente du signal d'acquittement. Ce signal d'acquittement sert à informer l'étage en amont que des données ont été bien reçues et que de nouvelles données peuvent être générées. De cette façon, une bulle est créée avant que les données puissent se déplacer entre les étages, et il n'y a aucun risque de perdre le flot de données entre les étages. Aucune nouvelle donnée ne peut être produite par l'étage en amont tant qu'il n'a pas reçu l'acquittement. L'étage en aval reste bloqué tant que l'acquittement qu'il a envoyé n'a pas été reçu. (Autrement dit, des 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.

Les pipelines asynchrones ont plusieurs propriétés intéressantes qui peuvent être exploitées pour la conception des circuits. Ces propriétés incluent la faible consommation à grain fin, les blocages localisés, la faible variation du courant lors des commutations et la possibilité d'utiliser des verrous transparents sans réduire la vitesse du flot de données [SUTH89]. Ces propriétés résultent de la disposition des étages du pipeline asynchrone qui sont activés en fonction des étages voisins se trouvant en amont et en aval. Une propriété intéressante est la capacité d'un étage du pipeline à s'activer ou à se désactiver en fonction de la validité de ses données. Par ailleurs, un étage du pipeline peut prendre des décisions locales de contrôle grâce au mécanisme de blocage du pipeline.

3.1.1.2 Performance du pipeline

La *capacité mémoire* d'un étage du pipeline est le nombre maximum de jetons qui peut être contenu sans bloquer l'entrée du pipeline.

Le *débit* d'un pipeline est le nombre de jetons par seconde qui passe par un étage donné.

On appelle *latence* le temps nécessaire pour qu'une donnée traverse les étages d'un pipeline.

Le *temps de cycle* d'un étage est le temps minimum 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 (dans le cas général, il faudrait considérer le maximum parmi les latences des différentes boucles mises en jeu). Il correspond également 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.

3.1.2 Pipeline linéaire vs pipeline non-linéaire

De nouveaux pipelines asynchrones ont été récemment introduits ([SING00], [SING00b], [SING01], [LINE98], [SUTH89], [OZDA02], [OZDA02b], [TUGS02] et [FERR02]). Cependant, la plupart cible des applications de pipelines linéaires, telles que les FIFO. Plus un circuit asynchrone est complexe, et plus les structures de pipeline utilisées sont élaborées. Nous présentons dans ce qui suit des pipelines non-linéaires asynchrones et les problèmes qui leur sont associés.

3.1.2.1 Pipeline linéaire

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

3.1.2.2 Pipeline non-linéaire : fourche et convergence

Un étage de pipeline non-linéaire est un étage qui peut avoir plusieurs canaux d'entrée et plusieurs canaux de sortie. Les pipelines de fourche, les pipelines de convergence, les multiplexages et les démultiplexages sont des exemples de pipelines non-linéaires.

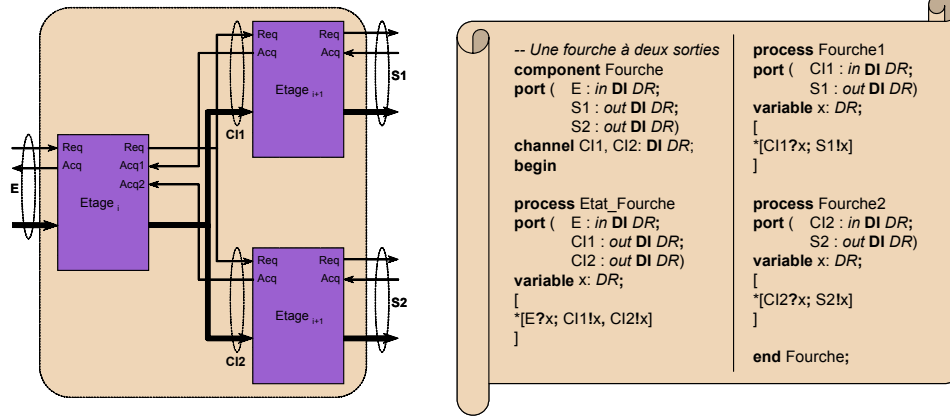


Figure 44 Pipeline non-linéaire : fourche

Une fourche (« fork ») 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 44).

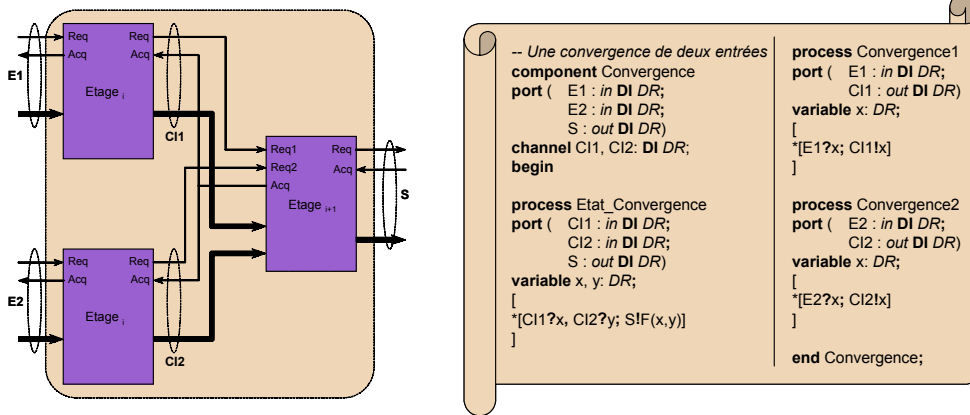


Figure 45 Pipeline non-linéaire : convergence

Une convergence (« join ») 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 45).

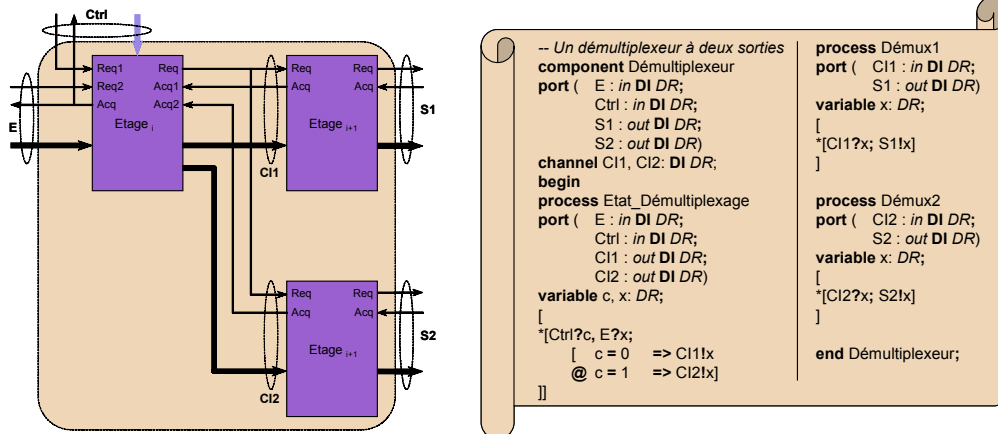


Figure 46 Pipeline non-linéaires : démultiplexage

Les fourches et les convergences complexes sont utilisées pour des lectures ou des écritures conditionnelles dans les canaux. La valeur du canal de contrôle régit la condition. Les exemples typiques sont les démultiplexages (« split ») et les multiplexages (« merge »). Ces pipelines sont présentés Figure 46 et Figure 47.

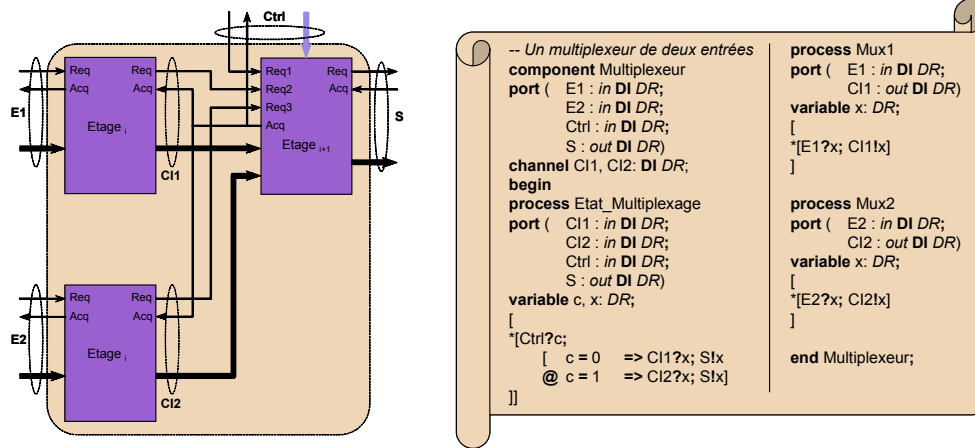


Figure 47 Pipeline non-linéaires : multiplexage

Dans le cas du protocole 4 phases, une fourche ayant plusieurs canaux de sortie, doit recevoir un signal d'acquittement de tous ces derniers pour revenir à zéro. De la même façon, une convergence, qui reçoit des données à partir de ses canaux d'entrée, doit émettre un signal d'acquittement sur tous les canaux d'entrée. Un démultiplexage est une combinaison de fourche et de convergence où un canal de contrôle est utilisé pour déterminer quelle sortie est générée. Le contrôle indique l'envoi de la donnée d'entrée à un des canaux de sortie, n'importe quelle combinaison des canaux de sortie, ou aucun. Cette dernière option est notée « skip ». Un multiplexage est une convergence où le signal de contrôle ou choix, vient d'un autre étage du pipeline. Le signal de choix contrôle quel canal entrant doit être lu.

3.1.2.3 Problèmes associés au traitement des fourches et convergences

Deux défis principaux se présentent pour la conception des pipelines non-linéaires : la synchronisation d'un étage de pipeline avec de multiples destinations (pour les fourches) et la synchronisation d'un étage de pipeline avec de multiples sources (pour les convergences). Cette partie discute ces problèmes en détail et les stratégies pour les régler.

3.1.2.3.1 Environnement lent ou blocage dans les fourches

Dans beaucoup de pipelines asynchrones linéaires existants – comme le pipeline classique PS0 de Williams et Horowitz [WILL91], et également comme les pipelines « look-ahead » [SING00] et ceux de grande capacité [SING00b] – certains acquittements entre les étages du pipeline sont essentiellement des impulsions chronométrées, c'est-à-dire les communications inter-étages ne sont pas persistantes. En particulier, après qu'un étage du pipeline affirme son signal d'acquittement, entraînant une précharge de l'étage précédent, il suppose que cette précharge est assez rapide. Par conséquent, il ne vérifie pas explicitement la fin de la précharge avant de désactiver son signal d'acquittement. Cette supposition de synchronisation, appelée « supposition de précharge rapide », est en général facilement satisfaite. Ainsi, la non-persistance n'est ordinairement pas problématique dans les pipelines linéaires : on peut raisonnablement supposer que tous les étages réagissent assez rapidement aux impulsions d'acquittement [WILL91].

Cependant, dans le cas où le chemin de donnée possède une fourche, la non-persistance peut être un défi. Les signaux d'acquittement multiples sont reçus par l'étage de fourche. Ces signaux sont donc des impulsions, qui peuvent être non recouvrantes. Par conséquent, les acquittements ne peuvent être correctement fusionnés en utilisant une porte de rendez-vous simple.

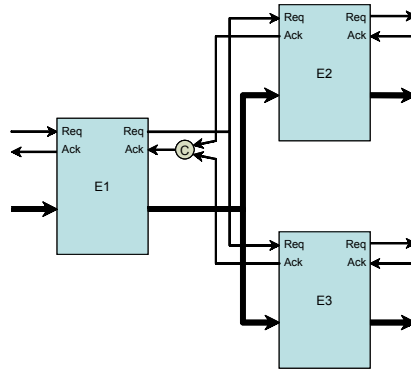


Figure 48 Pipeline de fourche

A titre d'exemple, la Figure 48 représente une fourche à deux sorties, où l'étage de fourche E1 contrôle les étages E2 et E3. Pour un fonctionnement correct, E1 doit recevoir les deux acquittements de E2 et E3. Cependant, les étages E2 et E3, et les étages ultérieurs de ces derniers, peuvent fonctionner indépendamment les uns des autres. Si l'étage E3 est retardé (ou bloqué), alors l'acquittement de E3 pour E1 est retardé. Pendant ce temps, un signal d'acquittement non persistant de E2, qui n'est pas retardé, est reçu par E1. Par conséquent, les deux acquittements ne peuvent pas être reçus en même temps par E1. Ainsi la porte de Muller contenu dans la fourche ne peut pas produire le signal de précharge pour E1.

Ce problème peut être reformulé comme une contrainte de synchronisation relative entre les signaux [CORT97]. Il faut désactiver la requête de l'étage de fourche avant la désactivation des acquittements des étages suivants (E2 et E3), préservant de ce fait le protocole de type « poignée de main » quatre phases relatif aux canaux.

Deux solutions générales peuvent être proposées pour résoudre le problème [OZDA02b] :

- La première solution consiste à mettre des conditions sur les signaux d'acquittement reçus des étages suivants pour les rendre persistants. Dans ce cas, les étages dans la branche non retardée de la fourche (c.-à-d., les successeurs de E2) n'ont aucune autre contrainte sur leur comportement. Donc cette solution n'a besoin que de changements locaux – des changements dans les étages suivants la fourche.
- La deuxième solution exige des modifications plus globales. En particulier, la partie de contrôle de chaque étage de pipeline suivant la fourche est modifiée afin de rendre tous les signaux d'acquittement persistants. Par conséquent, la contrainte de précharge rapide est éliminée, permettant de combiner avec des stratégies plus simples des signaux d'acquittement qui sont actuellement persistants. Dans ce cas, des étages postérieurs dans une branche non retardée de la fourche (*i.e.* successeurs de E2) sont encore contraints dans leur comportement (Ils peuvent seulement passer à une précharge sur la nouvelle donnée, mais ne pas entrer les phases de décharge ultérieures).

3.1.2.3.2 Environnement lent ou blocage dans les convergences

Le deuxième défi est celui de la synchronisation des canaux d'entrée dans une convergence, comme représenté Figure 49.

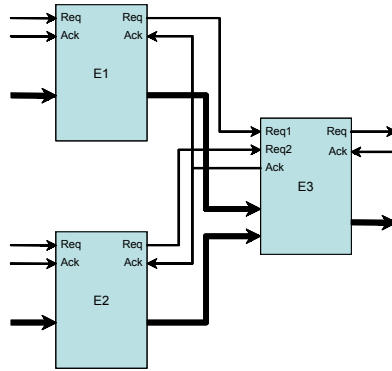


Figure 49 Pipeline de convergence

Un problème peut surgir si un bloc de fonction « gourmand » est utilisé pour implémenter l'étage E3. Si E3 est un bloc de fonction « gourmand », alors il peut produire des sorties après la consommation de données d'un étage précédent. Il n'attend pas la présence des données de l'autre étage. Par exemple, on suppose que E3 contient une fonction « Ou » double rails qui évalue rapidement (dès qu'un bit d'entrée arrive). Après l'évaluation d'une donnée provenant de E2, il enverra un signal d'acquiescement à E1 et E2, bien que E1 n'ait pu produire des données. En particulier, si l'étage d'entrée E1 est lent ou bloqué, il peut recevoir un acquiescement de E3 trop tôt. La sortie de l'étage lent peut être traitée comme un nouveau jeton de donnée non désiré, empêchant ainsi la synchronisation entre les étages du pipeline.

Ce problème peut également être formulé comme une contrainte de synchronisation relative entre les signaux : l'étage de convergence doit produire un signal d'acquiescement seulement après que tous les canaux d'entrée aient des données valides, préservant de ce fait le protocole « poignée de main » quatre phases sur tous les canaux d'entrée.

Deux solutions générales peuvent être également proposées pour résoudre ce problème :

- Une solution est d'utiliser simplement les blocs de fonction « non-gourmands » ; c'est-à-dire, chaque bloc de fonction vérifie explicitement la validité de toutes ses entrées avant de produire un résultat valide. De tels blocs de fonction s'appellent parfois en littérature « weakly-indicating » ou blocs de logique « condition faible » (« weak condition ») [SEIT80], [VARS90], [NIEL94] et [LINE98]. Cependant, le terme « condition faible » est souvent utilisé de façon plus restrictive que « non-gourmand » : les blocs de logique « condition faible » vérifient non seulement la validité de toutes les entrées avant de produire un résultat, mais ils vérifient également explicitement la remise à zéro de toutes les entrées avant de remettre la sortie à l'état initial. Par conséquent, les blocs « non-gourmands » sont parfois appelés sous le nom de « semi-condition faible ».
- La deuxième solution permet l'utilisation des blocs de fonction gourmands, tout en assurant la génération du signal d'acquiescement. Celui-ci est produit seulement après que des données de tous les étages d'entrée aient été reçues. Cette dernière solution exige la modification de la partie de contrôle.

3.2 Travaux antérieurs

La conception de circuit asynchrone à grande vitesse devient de plus en plus une alternative attrayante à la conception de circuit synchrone en raison de son absence de distribution d'horloge et des problèmes liés aux différences de temps de propagation des signaux d'horloge. De plus, elle fournit naturellement une interface robuste avec des composants plus lents. Un certain nombre de pipelines asynchrones rapides à grain fin a été proposé pour la conception des circuits asynchrones à

grande vitesse. Parmi eux, IPCMOS [SCHU00], GasP [SUTH01][COAT01] et les circuits en mode impulsion [NYST01]. 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 [SING00][SING00b] et statiques [SING01], 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 (*i.e.* sans fourches et convergences), bien que certaines solutions préliminaires pour manipuler les convergences ont été proposées dans [SING00b]. De plus, une approche initiale pour traiter les environnements lents ou les blocages dans un nombre limité de cas de pipelines linéaires a été présentée dans [SING00]. Pourtant, les problèmes de synchronisation qui arrivent en utilisant des fourches et convergences arbitraires sont beaucoup plus complexes, et les approches proposées par Singh et Nowick n'abordent pas ces problèmes.

Les divers modèles de pipelines se caractérisent par le temps de latence, le temps de cycle, et la robustesse de la synchronisation. Le modèle le plus robuste est le modèle QDI proposé par Martin et Lines [LINE98] dans lequel la correction d'un circuit asynchrone est garantie indépendamment du délai dans les portes et dans les fils. La communication entre processus via des canaux 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 [MART97]. Nous détaillerons ces modèles de pipeline dans les paragraphes qui suivent.

Un autre modèle commun de pipeline est celui de type « données groupées » dans lequel les chemins de données synchrones peuvent être utilisés conjointement avec des signaux de requête et d'acquittement [SUTH89]. Ces modèles de pipeline ont les avantages d'être moins gros en surface et de moins consommer que les autres modèles. Ils présentent également l'avantage que l'on peut réutiliser les méthodes des circuits synchrones pour la conception de chemins de données.

Beerel a proposé dernièrement certains modèles de pipelines qui ciblent l'encodage double rails ou « 1-parmi-N » et aussi l'encodage de type « données groupées » ([FERR02], [OZDA02], [OZDA02b] et [TUGS02]). Ces modèles offrent des pipelines à grain fin à haute vitesse et des pipelines linéaires ainsi que des pipelines non-linéaires. Cependant, ces modèles de pipelines présentent des inconvénients lors de la synthèse automatisée car ils demandent une bibliothèque de cellules complexes. [FERR02] propose une approche, appelée « single-track », dans laquelle un canal encodé par un codage « 1-parmi-N » est utilisé pour porter des données et servant également pour le signal d'acquittement. Cette approche est une extension des travaux de Berkel [BERK96].

3.3 Modèle des circuits cibles

Nous venons de présenter l'existence de divers types de circuits qui implémentent le mécanisme de pipeline dans la conception de circuits asynchrones. Parmi ces circuits et ces types de pipelines, ceux proposés initialement par Caltech [LINE98], les types QDI, sont les plus robustes. Ils nous permettent d'obtenir des circuits robustes, à faible consommation, et qui ont des performances assez élevées en vitesse.

Une autre propriété intéressante de ces types de pipelines est qu'il est facile de construire un circuit asynchrone par composition. Cette propriété est essentielle pour permettre une synthèse automatique des circuits asynchrones.

Nous présentons tout d'abord les différents types de pipelines de Caltech, et proposons ensuite la structure d'un circuit asynchrone pour la synthèse en se basant sur ces modèles.

3.3.1 Types de pipeline linéaire de base

3.3.1.1 Séquentiel

Comme indiqué par son nom, le protocole séquentiel est utilisé pour implémenter des blocs fonctionnels logiques qui n'ont pas besoin de synchronisation de type « poignée de main » entre l'entrée et la sortie. Autrement dit, la communication entre l'entrée et la sortie est transparente. La Figure 50 représente un module implémenté en protocole séquentiel avec une entrée E et une sortie S . Le bloc fonctionnel logique F se charge du calcul de la sortie. Le signal d'acquittement de l'entrée est directement le signal d'acquittement de la sortie.

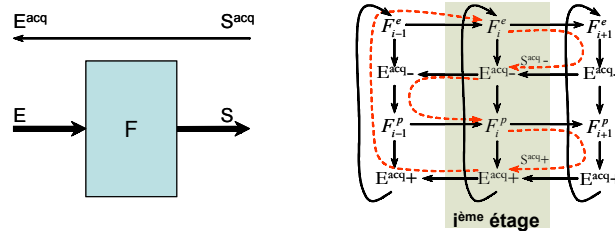


Figure 50 Protocole séquentiel : schéma et protocole de communication en STG

La Figure 50 illustre également la communication entre les étages de pipeline implémentée en protocole séquentiel sous forme d'un graphe de transitions de signaux STG [CHU87]. En lignes pointillées les communications du $i^{\text{ème}}$ étage avec ses étages voisins sont dessinées.

Ce protocole est le protocole le plus séquentiel. La donnée sur le canal E est transmise de l'entrée vers la sortie puis est acquittée une fois que la sortie est acquittée à son tour. Cette séquence est valable aussi bien pour la phase de calcul que 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 WCHB (« Weak Condition Half-Buffer »)

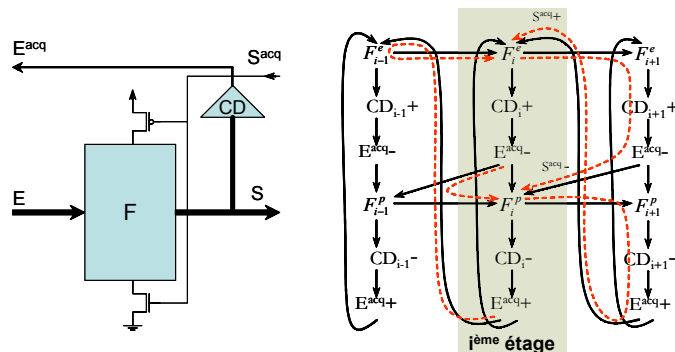


Figure 51 WCHB : schéma et son protocole de communication en STG

La Figure 51 illustre un pipeline linéaire de type WCHB [LINE98] avec le canal d'entrée E et le canal de sortie S . Le protocole de communication entre un étage du pipeline et ses étages voisins est représenté sous forme d'un graphe de transitions STG.

La Figure 52 représente un « buffer » double rails WCHB et son circuit au niveau transistor qui est déjà optimisé. Il est à noter que les signaux d'acquittement E^{acq} et S^{acq} sont actifs à niveau bas

et que les canaux utilisent un codage « 1-parmi-N ». Le bloc CD (« Completion Detect ») détecte la fin du calcul des sorties. L'opération de ce « buffer » est la suivante. Au départ, après le signal de reset, tous les rails de données sont au niveau bas et les rails d'acquittement sont au niveau haut. Quand la donnée arrive, un des rails d'entrée ($E0$ ou $E1$) passe au niveau haut. La sortie de la porte de Muller correspondante ($S0$ ou $S1$) passe à 1, faisant descendre le signal d'acquittement de l'entrée E^{acq} . L'environnement acquitte la donnée reçue par la descente du signal S^{acq} . En même temps, à l'entrée, l'environnement réagit à la descente du signal E^{acq} par la remise à 0 des rails de données. Conjointement avec S^{acq} au niveau bas, les rails de sortie sont remis à l'état initial.

Selon [LINE98], puisque les canaux E et S ne peuvent pas simultanément contenir deux données distinctes, ce circuit s'appelle un « demi-buffer » (« half-buffer »). Selon le protocole de communication dans la Figure 51, ce « demi-buffer » WCHB a une latence de deux transitions en direct (E vers S) et de trois transitions en retour (pour le chemin de S^{acq} à E^{acq}). Le temps de cycle du « demi-buffer » WCHB est donc de 10 transitions au total. De ce fait, le « demi-buffer » WCHB est actuellement plus rapide que des « demi-buffers » basés sur d'autres modèles de pipeline QDI que nous décrirons par la suite.

En fait, pour ce « demi-buffer », la validité et la neutralité de la donnée sur S impliquent la validité et la neutralité de la donnée correspondante sur E . Le canal E n'a pas besoin d'être vérifié si le canal S a été vérifié, ce qui fait qu'on qualifie la logique de logique « condition faible ».

Grâce à la symétrie dans les phases montante et descendante de la sortie d'un « demi-buffer » WCHB sur les signaux E_i ($i = 0, 1$) et S^{acq} , ce protocole permet d'avoir une implémentation très régulière d'un circuit asynchrone. Le bloc fonctionnel de « condition faible » dans l'exemple de la Figure 52 est une porte Muller simple. Cependant, pour les pipelines non-linéaires plus complexes, le bloc fonctionnel de « condition faible » peut être implémenté en technologie CMOS par des réseaux N et P beaucoup plus complexes. Ces derniers donnent un temps de latence direct plus lent et des surfaces plus importantes.

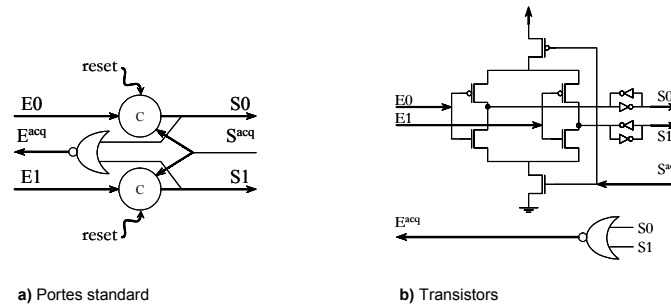


Figure 52 Implémentation du « demi-buffer » double-rails en protocole WCHB

3.3.1.3 PCHB (« PreCharge Half-Buffer »)

La Figure 53 représente le modèle de pipeline linéaire PCHB [LINE98]. Contrairement au modèle WCHB, le test de la validité et de la neutralité de données entrantes s'effectue explicitement en utilisant un circuit détecteur en entrée du circuit. Dans la figure, ce détecteur est indiqué comme ECD et celui de sortie comme SCD. Grâce au protocole, représenté en STG Figure 53, le modèle PCHB permet de remettre plus tôt à zéro la sortie dès que celle-ci est acquittée. Cette pré-charge s'effectue sans attendre l'invalidité du canal d'entrée. Le signal d'acquittement d'entrée est repositionné pour la donnée suivante une fois que le canal d'entrée est remis à zéro.

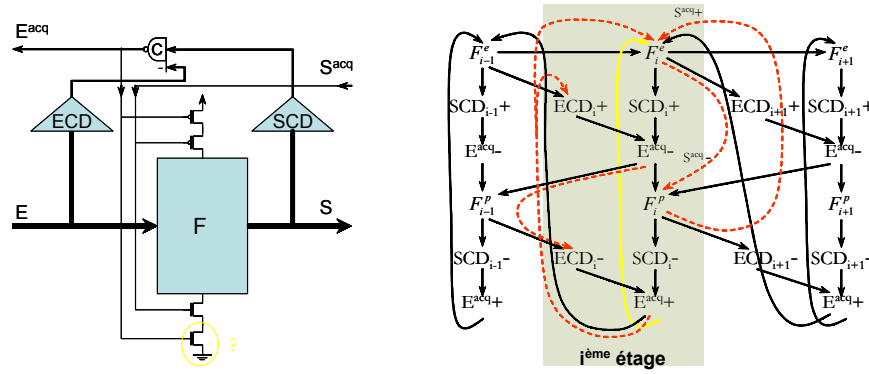


Figure 53 PCHB : modèle schématique et son protocole de communication en STG

Le bloc fonctionnel F n'est pas forcément en logique de « condition faible » et ainsi peut évaluer avant que toutes les entrées soient arrivées (si la logique le permet). Pourtant, le modèle produit seulement le signal d'acquiescement E^{acq} après que toutes les entrées soient arrivées et que la sortie ait été évaluée. En particulier, ECD et SCD sont combinés en utilisant une porte de rendez-vous pour produire le signal d'acquiescement : le circuit est bien QDI.

Dans ce modèle, puisque l'élément C est inversé, le signal d'acquiescement est actif au niveau bas. Le temps de cycle du modèle, selon son graphe de transitions STG décrit Figure 53, peut être facilement déduit : $T_{PCHB} = 3 t_{\text{éval}} + 2 t_{CD} + 2 t_C + t_{\text{pré}}$. Cela fait généralement 14 transitions. Dans cette formule, t_{CD} prend 2 transitions, t_C prend une transition et $t_{\text{éval}}$ et $t_{\text{pré}}$ prennent 2 transitions.

La Figure 54 présente le « demi-buffer » double rails en protocole PCHB.

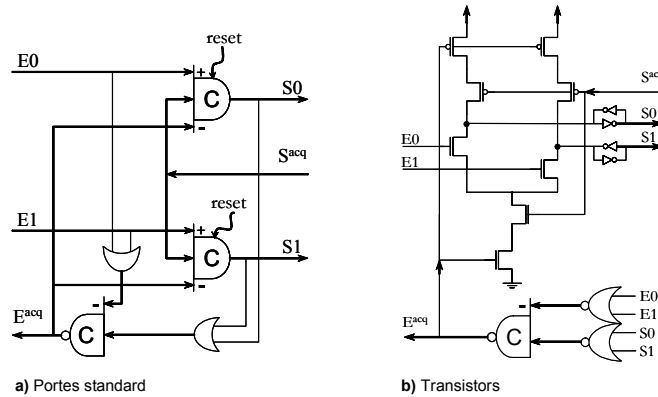


Figure 54 Implémentation du « demi-buffer » double-rails en protocole PCHB

3.3.1.4 PCFB (« PreCharge Full-Buffer »)

Le modèle PCFB [LINE98] et son protocole de communication en STG sont donnés Figure 55. Ce modèle est plus concurrent que le PCHB car ses signaux d'acquiescement sont mis à zéro en parallèle au prix d'une variable supplémentaire notée en . Il permet de mémoriser une donnée complète dans le pipeline. En particulier, la donnée sur le canal d'entrée peut simultanément différencier la donnée sur le canal de sortie, d'où le nom de « buffer » complet (« full-buffer ») pour ce type de pipeline.

Le temps de cycle de ce modèle est $T_{PCFB} = 2 t_{\text{éval}} + 2 t_{CD} + 2 t_C + t_{\text{pré}}$ qui donne 12 transitions en général.

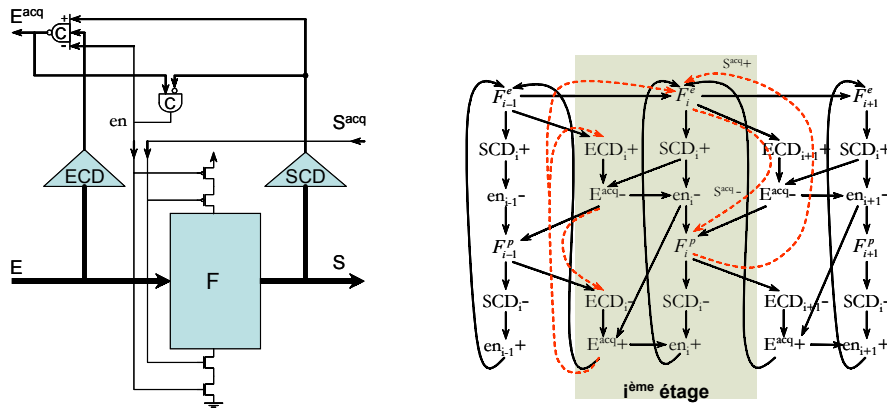


Figure 55 PCFB : modèle schématique et protocole de communication en STG

A titre de remarque, il est à noter que le bloc fonctionnel de la Figure 51 est différent de ceux des Figure 53 et Figure 55. Le bloc de calcul F dans le modèle WCHB est réalisé en logique « condition faible » : la phase montante et la phase descendante sont symétriques (Figure 52). Par contre, les modèles PCHB et PCFB sont réalisés de façon dissymétrique entre la phase montante et la phase descendante comme illustrée Figure 54.

En outre, comme la validité et la neutralité des données entrantes sont explicitement vérifiées dans les deux modèles PCHB et PCFB, leurs blocs fonctionnels de pré-charge sont plus petits et rapides. Cela se traduit par un nombre de transistors et de transitions plus grand par cycle.

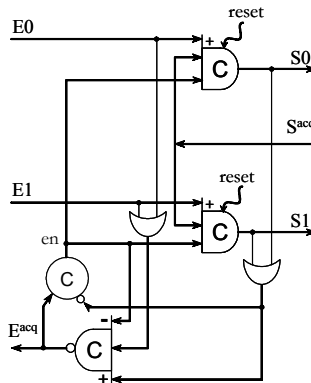


Figure 56 Implémentation du « buffer » double rails avec le protocole PCFB

3.3.2 Structure du circuit cible

Les structures de pipelines linéaires présentées dans les derniers paragraphes ne sont pas suffisantes pour implémenter des circuits asynchrones QDI. Ils demandent en plus des structures non-linéaires plus complexes. Ces structures comprennent les fourches, les convergences, les multiplexages et les démultiplexages. Nous présentons dans ce paragraphe ces structures avec différents protocoles (séquentiel, WCHB, PCHB et PCFB). L'ensemble des structures de pipelines linéaires et non-linéaires sert à construire des circuits asynchrones QDI avec le protocole correspondant. En particulier, un circuit asynchrone est construit en composant les structures pipelines de base.

Dans le système asynchrone, un circuit asynchrone, compatible avec la spécification DTL (cf. paragraphe 2.3.2), est toujours implémenté par un bloc de calcul et un bloc de contrôle. Le bloc de calcul, qui est complètement combinatoire, prend en charge le calcul des sorties et des états suivants. Il est implémenté en logique « condition faible ». Le bloc de contrôle, chargé de la synchronisation entre le circuit et les autres modules du système, est appelé un « buffer ». En particulier, ce contrôle doit implémenter un protocole de communication de type « poignée de

main » avec les modules auxquels ce circuit est connecté. En fait, le bloc de calcul effectue la partie combinatoire et le « buffer » fonctionne comme une partie séquentielle. Du point de vue de l'architecture, un circuit asynchrone est clairement séparé en deux parties. Ces deux parties sont mises en cascade (pipeline). La Figure 57 illustre deux cas possibles : le « buffer » est placé soit en entrée, soit en sortie du circuit. Dans ces deux cas, alors que le « buffer » peut être réalisé en utilisant un des protocoles mentionnés ci-dessus, le bloc combinatoire est implémenté en utilisant le protocole séquentiel (cf. §3.3.1.1). Il est à noter qu'un circuit asynchrone peut être implémenté sans « buffer ». Dans ce cas, ce circuit devient un circuit séquentiel. Ce type de circuit a déjà été présenté dans le paragraphe 3.3.1.1.

Ces modèles d'implémentation ressemblent à l'architecture micropipeline présentée dans [SUTH89]. La différence principale est la façon d'implémenter le chemin de données : le traitement des données est insensible aux délais tandis qu'elle est du type « données groupées » pour le micropipeline.

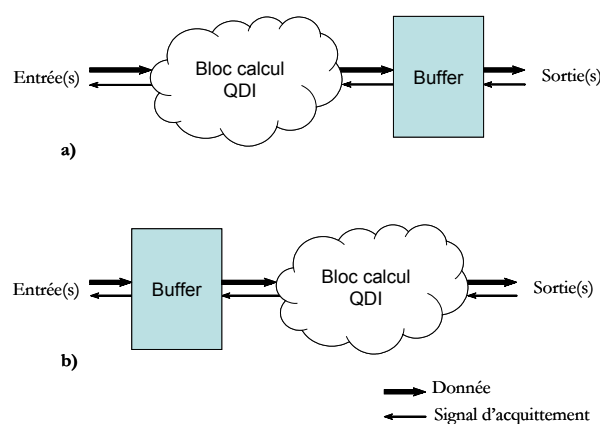


Figure 57 Implémentation de circuits asynchrones QDI avec a) buffers à la sortie et b) buffers en entrée

Nous présentons par la suite les différentes structures non-linéaires pour l'implémentation du bloc de calcul et du buffer. Ces structures, dans l'esprit de l'extension des types de pipeline présentés ci-dessus, comprennent les fourches, les convergences, les multiplexages et les démultiplexages. La façon d'implémenter ces structures doit satisfaire les contraintes qui leur sont associées (cf. §3.1.2.3).

3.3.2.1 Structures implémentées en séquentiel

Afin d'implémenter un circuit asynchrone QDI avec le protocole séquentiel, les structures séquentielles de base sont nécessaires. Nous avons présenté la structure pipeline linéaire en protocole séquentiel dans le paragraphe 3.3.1.1. Nous étendons les structures séquentielles non-linéaires en présentant ici quatre structures de base qui permettent la conception de façon automatique d'un circuit asynchrone en protocole séquentiel.

La Figure 58 représente une fourche à deux sorties implémentée avec le protocole séquentiel. Les fourches à plus de deux sorties peuvent être obtenues de la même manière. Le chemin de données, représenté en gras, est transparent entre l'entrée et les deux sorties. Par contre, pour résoudre le problème de blocage ou de ralentissement dans un des canaux de sortie, il faut une porte de Muller pour synchroniser les deux signaux d'acquittement des sorties. De ce fait, le canal d'entrée est acquitté quand les deux sorties ont bien reçues la donnée.

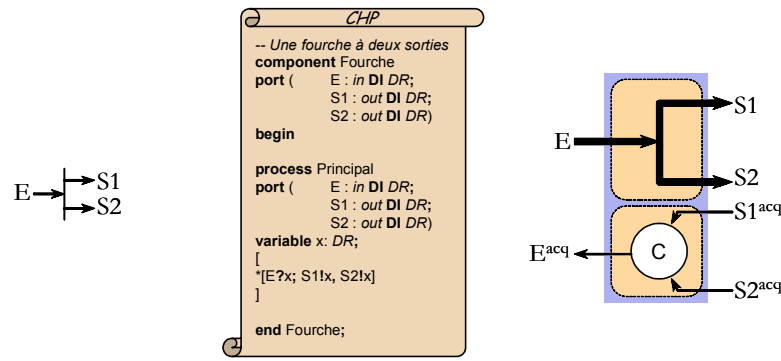


Figure 58 Pipeline de fourche en protocole séquentiel

A titre de remarque, dans la Figure 58 les données envoyées sur les canaux de sorties S1 et S2 sont exactement les données du canal d'entrée E. Dans des cas plus complexes, les données sur S1 et S2 sont différentes et également différentes de celles lues sur E. Les blocs de calcul combinatoires sont alors nécessaires pour calculer les sorties S1 et S2. Ces blocs, indiqués F1 et F2 dans la Figure 59, sont réalisés en logique « condition faible ». Ils ont un seul chemin de données entrant et un seul chemin de données sortant. Par conséquent, ces blocs n'influent pas sur le protocole de communication et nous les aborderons plus tard. Dans la suite de ce chapitre, les blocs de calcul sont considérés transparents et sont donc omis dans la discussion.

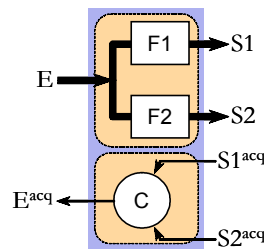


Figure 59 Pipeline de fourche en protocole séquentiel : avec les blocs de calcul

De façon duale à la fourche, une structure de type convergence implémentée en protocole séquentiel est décrite Figure 60. Elle 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 deux entrées E1 et E2. Ceci est explicitement accompli par la logique « condition faible » du bloc de calcul F. Contrairement à la structure de fourche, les signaux d'acquittement des canaux E1 et E2 proviennent directement du signal d'acquittement du canal S. Cela est justifié par la contribution concurrente des canaux E1 et E2 à la génération de S.

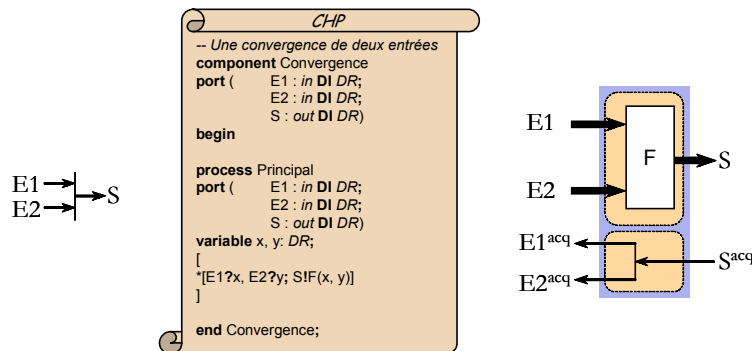


Figure 60 Pipeline de convergence en protocole séquentiel

Les structures plus complexes de fourche et de convergence en protocole séquentiel sont montrées Figure 61 et Figure 62. Celles-ci sont le démultiplexage et le multiplexage contrôlés par un canal C. Selon les valeurs de ce canal, une des sorties est choisie dans le cas du démultiplexage

pendant qu'une des entrées est sélectionnée pour le cas du multiplexage. Très souvent, le canal de contrôle C est codé en « 1-parmi-N ». Dans ce cas, chaque rail du canal C correspond à un cas de la sélection. A titre d'exemple, pour le démultiplexage à deux sorties, il suffit d'encoder le canal C en double-rails : rail 0 pour sélectionner la sortie S1 et rail 1 pour sélectionner la sortie S2.

En fait, les signaux de sélection peuvent être plus complexes si bien que des blocs de décodage sont nécessaires pour calculer les signaux de sélection. Ces blocs de calcul sont ignorés dans la discussion, leur réalisation sera abordée dans les chapitres suivants.

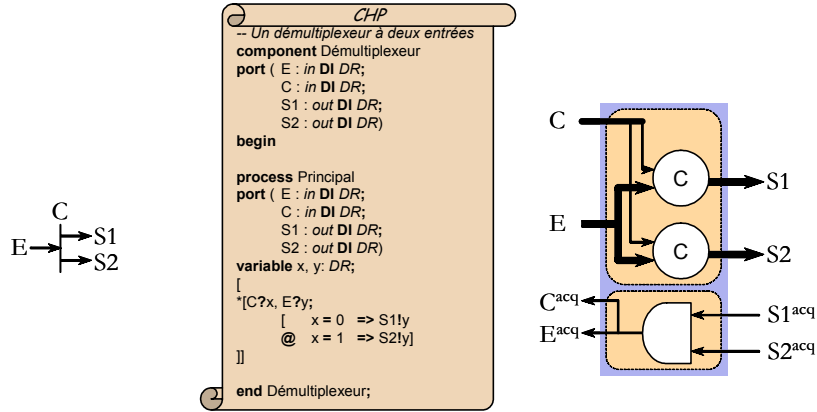


Figure 61 Pipeline de démultiplexage en protocole séquentiel

La porte « OR » à la sortie de l'opérateur de multiplexage est en fait une porte avec des données exclusives qui sélectionne une des données entrantes. Cette exclusivité est garantie par l'exclusivité dans le calcul des signaux de sélection.

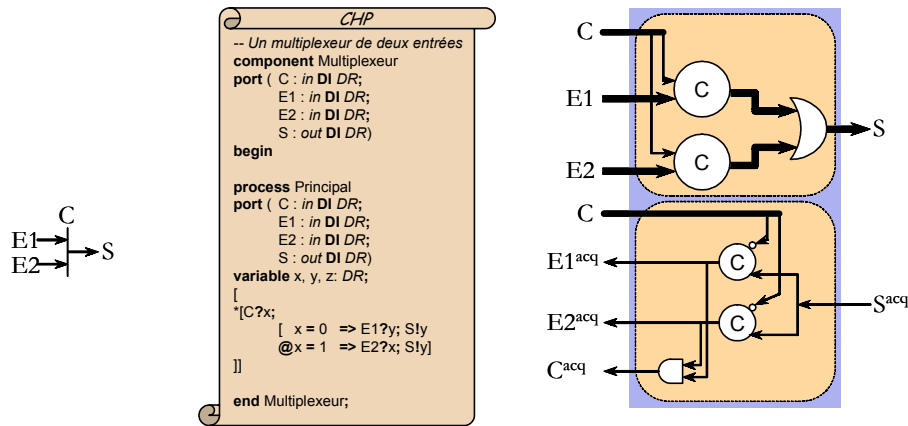


Figure 62 Pipeline de multiplexage en protocole séquentiel

Dans la génération des signaux d'acquiescement, l'exclusivité sur les canaux de sortie du démultiplexage garantit qu'un des signaux d'acquiescement des sorties est acquitté. Une porte avec des données exclusives est nécessaire. Comme les signaux d'acquiescement sont actifs au niveau bas, une porte « AND » est donc utilisée. En revanche, dans le cas du multiplexage (Figure 62), nous n'acquiesçons que le canal sélectionné. Cette sélection s'effectue par les signaux de sélection eux-mêmes. Les inverseurs placés sur l'entrée des portes de Muller de sélection sont justifiés par le fait que les signaux d'acquiescement sont actifs au niveau bas. Le canal de contrôle C est acquitté quand un des canaux d'entrée est acquitté. Ceci est réalisé par une porte « AND » qui joue le rôle d'une porte « OR » avec des données exclusives.

3.3.2.2 Structures implémentées en WCHB

A la différence de celles implémentées en protocole séquentiel, dans les structures de base implémentées avec le protocole WCHB, les données sont envoyées sur les canaux de sortie quand

ces derniers sont disponibles. En particulier, les données entrantes doivent attendre le signal d'acquittement de la sortie. Cela nécessite l'usage d'une porte de Muller pour faire le protocole de type « poignée de main » à la sortie.

Grâce à la logique « condition faible », la présence des données sur les sorties implique la validation des données sur les entrées. Ainsi la génération du signal d'acquittement pour les canaux d'entrée est simplement un test de validation de données sur les canaux de sortie. Encore une fois, l'inverseur sur le signal d'acquittement des canaux d'entrée signifie que l'acquittement est actif au niveau bas.

La Figure 63 illustre une fourche et une convergence implémentées avec le protocole WCHB.

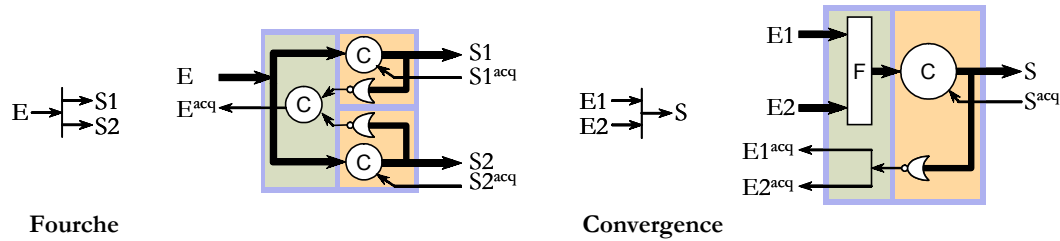


Figure 63 Pipeline de fourche et pipeline de convergence en protocole WCHB

Une remarque importante dans ce paragraphe est que l'implémentation de ces structures peut être obtenue de façon généralisée en appliquant les décompositions discutées Figure 57. En particulier, ces structures peuvent être considérées comme une concaténation d'un bloc de calcul avec un bloc de contrôle. Le bloc de calcul qui se charge de la fonction principale de la structure est implémenté avec le protocole séquentiel. Le bloc de contrôle est implémenté avec le protocole de communication choisi (dans ce cas, WCHB) pour effectuer la synchronisation avec les autres modules. A titre d'exemple, les structures Figure 63 montrent cette implémentation avec le bloc de contrôle placé à la sortie. Les fonctions des structures (la fourche et la convergence) sont en fait les structures séquentielles qui ont été présentées Figure 58 et Figure 60. Le protocole de communication de type « poignée de main » sur les canaux de sortie est réalisé par des buffers WCHB. Cette structure de combinaison correspond exactement à la structure de circuits représentée Figure 57.

La Figure 64 représente un démultiplexage et un multiplexage en protocole WCHB. Il est à noter que la porte « AND » dans les deux schémas réalise un « OR » exclusif car les signaux d'acquittement sont actifs au niveau bas.

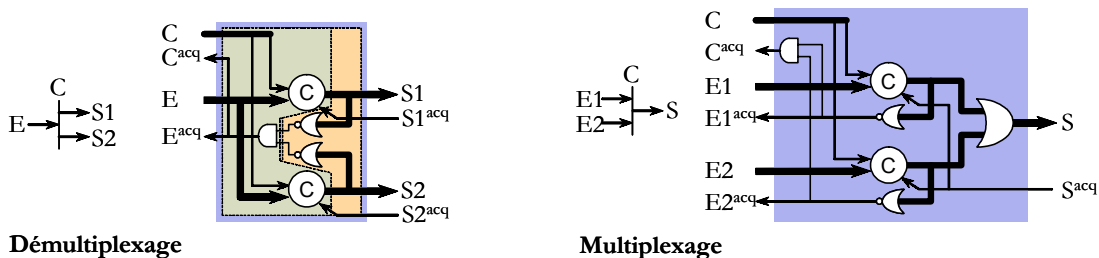


Figure 64 Pipeline de démultiplexage et pipeline de multiplexage en protocole WCHB

3.3.2.3 Structures implémentées en PCHB

La Figure 65 représente une fourche et une convergence implémentées avec le protocole PCHB. La porte de Muller dans le schéma de la fourche synchronise les deux canaux de sortie. Elle peut être optimisée avec la porte de Muller dissymétrique pour générer le signal d'acquittement de

canal d'entrée. Quant au schéma de la convergence, grâce à la logique « condition faible » du bloc de calcul F, la présence des données à la sortie du bloc F implique la validité des données sur les deux canaux d'entrée. Par conséquent, le test de validité sur les canaux de sortie est nécessaire et suffisant pour acquitter les canaux d'entrée.

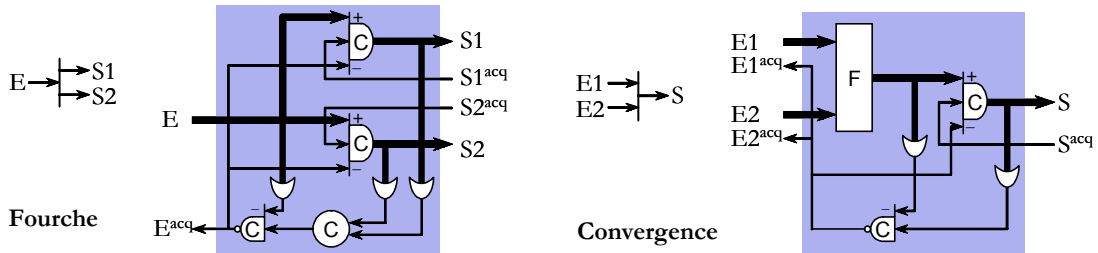


Figure 65 Pipeline de fourche et pipeline de convergence en protocole PCHB

Les structures de démultiplexage et de multiplexage avec le protocole PCHB sont représentées Figure 66. Il est à noter que dans le cas du multiplexage, puisqu'un seul canal d'entrée est acquitté, le canal de contrôle C sert aussi à sélectionner le canal correspondant à acquitter.

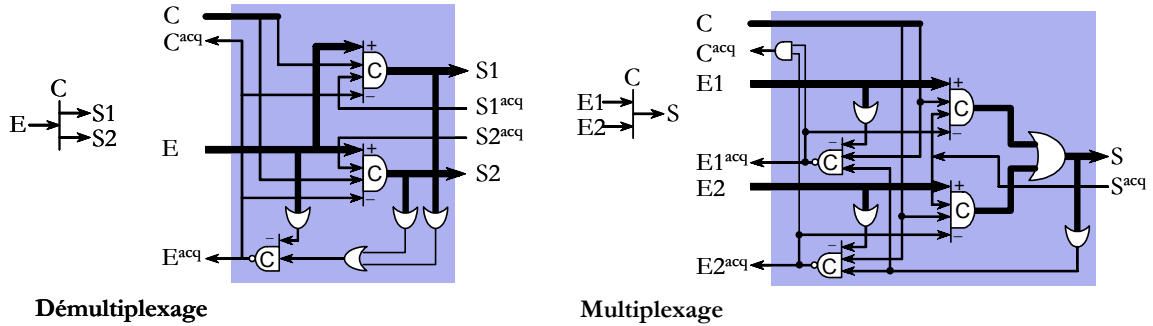


Figure 66 Pipeline de démultiplexage et pipeline de multiplexage en protocole PCHB

3.3.2.4 Structures implémentées en PCFB

A titre indicatif, nous présentons Figure 67 les schémas d'une fourche et d'une convergence implémentées avec le protocole PCFB.

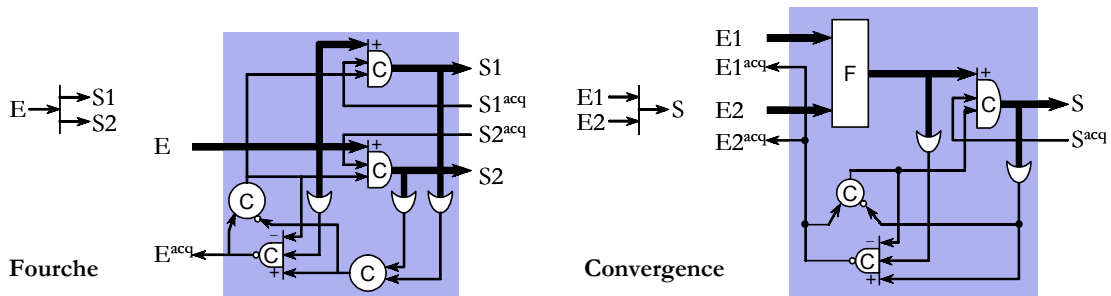


Figure 67 Pipeline de fourche et pipeline de convergence implémentées en protocole PCFB

Les structures de démultiplexage et de multiplexage sont décrites Figure 68.

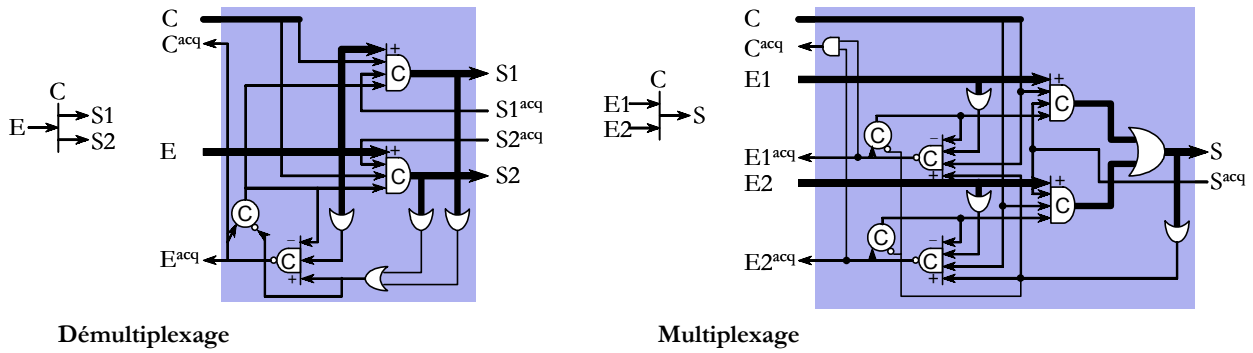


Figure 68 Pipeline de démultiplexage et pipeline de multiplexage en protocole PCFB

3.4 Conclusion

Dans ce chapitre, nous avons présenté le mécanisme de pipeline asynchrone. Comme on l'a vu précédemment, contrairement au pipeline synchrone, le pipeline asynchrone dispose de plusieurs propriétés intéressantes (faible consommation, blocage localisé, capacité de données variable, *etc.*). Ces propriétés ont été exploitées lors de la conception de circuits asynchrones à grande vitesse et à faible consommation. Cependant, pour les pipelines non-linéaires plus complexes (comme les fourches, les convergences, les multiplexages et les démultiplexages), les problèmes liés au temps de propagation dans les différentes branches du pipeline sont soulevés. Des solutions ont également été proposées.

Motivés par les propriétés des pipelines asynchrones, plusieurs travaux de recherche sur le pipeline asynchrone ont été effectués. Les pipelines les plus robustes sont actuellement les pipelines QDI, initiés par Andrew Lines. Partant de ce type de pipeline, nous avons proposé des modèles de circuits cibles. Ces modèles se composent d'une partie de calcul et d'une partie de contrôle, permettent de synthétiser des circuits asynchrones de façon automatique et systématique. Enfin, nous avons développé les différentes structures de pipeline non-linéaire – telles que fourche, convergence, multiplexage, démultiplexage, ... – avec les protocoles (séquentiel, WCHB, PCHB et PCFB).

Chapitre 4

GENERATION D'UNE FORME INDEPENDANTE DES PROTOCOLES ET DE LA TECHNOLOGIE

Les circuits asynchrones promettent un nombre d'avantages importants pour la conception des circuits numériques complexes. Leur modularité, leur faible consommation, leur temps de calcul moyen, et l'élimination de la distribution de l'horloge encouragent leur étude. Cependant, l'implémentation asynchrone d'un circuit doit satisfaire des conditions plus restrictives que son homologue synchrone. Un circuit asynchrone doit être non seulement fonctionnellement équivalent à la spécification mais aussi exempt d'aléa.

Les circuits asynchrones les plus robustes sont les circuits DI. Aucune hypothèse temporelle n'est introduite pour ce type de circuits. Mais malheureusement, les circuits DI sont très limités d'un point de vue pratique. De ce fait, le type de circuits asynchrones QDI est largement utilisé pour effectuer une implémentation asynchrone car il est robuste. La correction fonctionnelle d'un circuit QDI ne dépend pas des délais des portes et des interconnexions, à l'exception de l'hypothèse faite sur les interconnexions constituant une fourche isochrone. De plus, lors d'une migration de la technologie, aucune modification sur ce type de circuits n'est nécessaire pour garantir la correction fonctionnelle (la validité des fourches isochrones doit toujours être vérifiée).

Un éventail de techniques de synthèse des circuits asynchrones existe. Ces techniques utilisent des modèles basés sur les événements comme les réseaux de Petri, les STG, ... (voir §1.3). Le réseau de Petri est un formalisme puissant pour modéliser des systèmes concurrents car il intègre les notions de séquentialité, concurrence et choix entre les événements. Sa caractéristique la plus intéressante est la capacité à décrire explicitement un système complexe en une représentation concise.

Nous proposons dans ce chapitre une méthode de synthèse des circuits asynchrones de type QDI qui sont décrits dans un langage de haut niveau, CHP. Cette méthode se base sur un réseau de Petri de haut niveau dont chaque place représente une action de communication ou de calcul. Présenté dans le Chapitre 2, ce réseau de Petri est vérifié de façon à garantir qu'il soit conforme à la spécification DTL. Cette vérification assure également l'existence d'une implémentation QDI du circuit. Nous proposons également une forme de représentation des dépendances des signaux du circuit : les équations de dépendances. Elles servent à spécifier les relations de causalité entre des

signaux. Deux techniques sont offertes pour générer les équations de dépendances tout en exploitant le réseau de Petri. Les équations de dépendances sont indépendantes des styles d'implémentation des circuits ainsi que de la technologie cible. Nous verrons dans le Chapitre 5 comment les différentes implémentations de circuits sont créées à partir de ces équations de dépendances.

La méthode de synthèse proposée est prototypée par l'outil de synthèse automatique TAST qui sera présenté Chapitre 6.

4.1 Flot de synthèse de circuits QDI

La méthode de synthèse proposée dans ce travail de thèse est représentée en détails par le flot de synthèse donné Figure 69. Servant à synthétiser les circuits asynchrones de type QDI en portes logiques standard, elle montre les étapes de synthèse de la spécification CHP jusqu'au réseau final de portes.

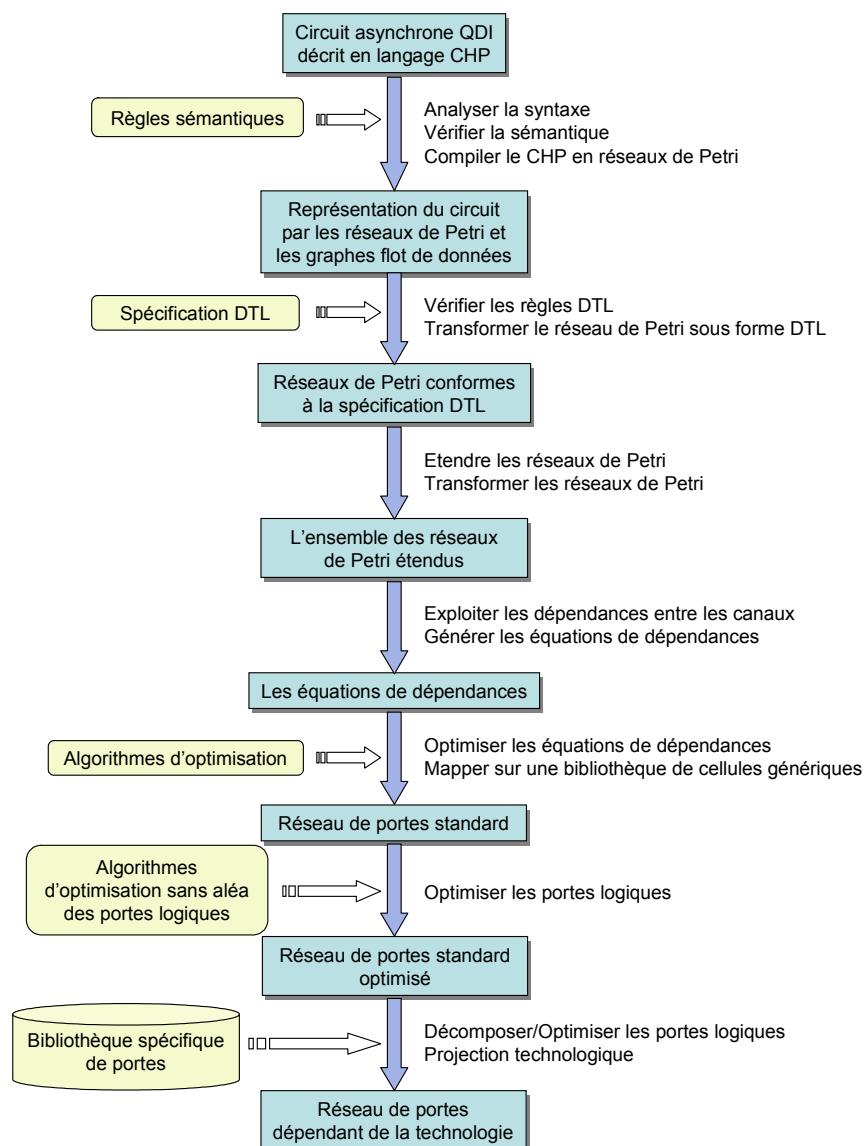


Figure 69 Flot détaillé de la synthèse de circuits asynchrones QDI

La synthèse part de la description, en le langage de description de haut niveau CHP, d'un circuit asynchrone. Le programme CHP est analysé par un « *parser* » lexical et syntaxique. Cette analyse transforme ce programme en une représentation intermédiaire sous la forme d'une structure arborescente. Grâce à des règles de sémantiques prédéfinies, une étude sur cette structure est ensuite

effectuée afin de vérifier la correction sémantique de la spécification. Ces étapes seront détaillées dans le Chapitre 6 où l'outil TAST sera présenté. Un compilateur est développé pour traduire cette structure intermédiaire en réseaux de Petri et graphes de flot de données. Servant à spécifier hiérarchiquement le comportement (la partie contrôle) autant que la dépendance de données (la partie opérative) entre des canaux du circuit, la combinaison de ces deux formes de représentation (cf. §2.2.3) est également utilisée par les outils *back-end* pour fournir des implémentations matérielles. De cette façon, les circuits asynchrones complexes à grande échelle peuvent facilement être décrits à l'aide d'un langage de haut niveau et ensuite représentés sous forme d'un graphe prêt à la synthèse. De plus, cette forme de représentation intermédiaire permet de séparer le *front-end* et le *back-end* de l'outil. L'avantage le plus important de l'indépendance entre le *front-end* et le *back-end* est que, on peut ajouter un outil de compilation d'un autre langage de description dans le *front-end* ou un outil de génération d'un autre style d'implémentation de circuits dans le *back-end*, tout en passant par cette forme intermédiaire. Cet avantage sera présenté comme une perspective de ce travail.

On valide ensuite que le réseau de Petri est conforme à la spécification DTL (voir §2.3). Cette vérification a pour but d'assurer que le réseau de Petri est synthétisable et qu'une implémentation QDI du circuit existe. Ce réseau de Petri est ensuite étendu en déployant (« *unfolding* ») toutes les branches. Cette étape permet d'obtenir un ensemble de réseaux de Petri dont chacun représente une branche de celui d'origine (*i.e.* un choix).

Certains algorithmes sont développés pour exploiter exhaustivement les réseaux de Petri et ses graphes de données associés afin de trouver les relations de dépendances entre les canaux du circuit. Ces relations sont représentées sous forme d'équations de dépendances exprimant les dépendances de causalité entre les canaux de sortie et les canaux d'entrée. Aucune hypothèse n'est faite sur le protocole de communication et la technologie cible lors de l'extraction des équations de dépendances. Par conséquent, les équations de dépendances sont indépendantes du protocole de communication et de la technologie cible. Certaines simplifications peuvent être effectuées sur les équations de dépendances.

Dans le Chapitre 5, grâce à ces équations de dépendances, les équations logiques des sorties du circuit sont déduites. Une première implémentation sans aléa du circuit en réseau de portes peut être tout de suite générée en utilisant une bibliothèque de cellules génériques. Le réseau de portes standard est particulièrement intéressant car il permet de faciliter un prototypage rapide du circuit en utilisant les cellules standard. De cette manière, la synthèse de circuits asynchrones ne se base pas sur des cellules dédiées à l'asynchrone (la conception de type « *semi-custom* »). Par ailleurs, à partir des équations logiques des sorties, les algorithmes d'optimisation et de décomposition spécifiques aux circuits QDI peuvent être appliqués. Les algorithmes pour les circuits synchrones ne peuvent être directement utilisés dans ce cas car le circuit obtenu peut avoir des aléas et l'hypothèse de fourche isochrone ne peut être garantie. Les techniques d'optimisation et de décomposition peuvent être également intégrées dans la projection technologique des portes génériques avec des cellules d'une bibliothèque spécifique. Nous en parlerons dans le Chapitre 5 plus tard.

4.2 Du CHP aux réseaux de Petri et graphes de flot de données

Depuis des années, les réseaux de Petri (cf. §2.2.1) ont été largement utilisés pour représenter des comportements mixtes concurrents et séquentiels des systèmes. Les graphes de flot de données (cf. §2.2.2) sont adéquats pour modéliser la dépendance de données des circuits asynchrones. Par conséquent, la combinaison des réseaux de Petri et des graphes de flot de données est capable de

modéliser des circuits asynchrones à tout niveau de complexité et de hiérarchie. Cette combinaison est adoptée comme la forme de représentation intermédiaire dans l'approche proposée.

Afin de traduire un programme CHP en réseau de Petri et graphes de flot de données, nous considérons les représentations en réseau de Petri de toutes les structures du langage CHP. Les structures de contrôle sont représentées par des réseaux de Petri alors que les dépendances de données sont représentées par des DFG qui sont associés aux places et aux transitions du réseau de Petri. Les actions de communication ou de calcul CHP sont associées aux places du réseau de Petri alors que les gardes sont associées aux transitions.

4.2.1 Expressions

Une expression du langage CHP est représentée par un DFG dont les feuilles sont les variables, canaux ou constantes du programme. Les nœuds du graphe sont les opérateurs arithmétiques, les actions de communication, ...

A titre d'exemple, une affectation $x := a + b * c * d + 1$ peut être éclatée en un ensemble de quatre opérateurs simples (addition, multiplication). La représentation sous forme d'un DFG est décrite Figure 70. Les données entrées et sorties, représentées respectivement par les variables et constantes $\{a, b, c, d, 1\}$ et par $\{x\}$, sont explicitement marquées comme les feuilles du graphe.

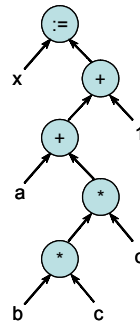


Figure 70 Exemple d'un graphe de flot de données $x := a + b * c * d + 1$

Il est à noter que l'affectation pourrait avoir été représentée par un autre DFG différent en exploitant la commutativité et l'associativité de l'addition et de la multiplication.

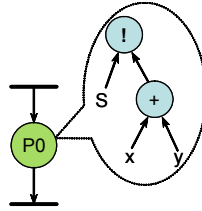
4.2.2 Gardes

Puisqu'une garde du langage CHP est une expression booléenne, elle est alors traitée comme une expression. En particulier, une garde est représentée par un DFG. En plus des opérateurs arithmétiques, les nœuds de ce DFG peuvent inclure les opérateurs de comparaison (« = », « /= », ...) et de « probe ».

Contrairement aux expressions dans les instructions du langage CHP, les DFG représentant les expressions de garde sont associés aux transitions du réseau de Petri.

4.2.3 Instructions

Une instruction du langage CHP est une action de communication sur les canaux (par exemple : envoi, réception) ou une affectation de variables locales. Une autre instruction spéciale est le « *skip* ». Ces instructions sont représentées par un DFG qui est associé à une place du réseau de Petri.

Figure 71 Représentation en PN-DFG d'une instruction $S !x+y$

Un exemple est donné Figure 71. Dans cet exemple, la valeur de l'expression $x+y$ est envoyée sur le canal de sortie S . Cette instruction est représentée par une place $P0$ du réseau de Petri et le DFG associé.

4.2.4 Opérateur de séquence

Deux instructions CHP en séquence sont traduites en réseau de Petri par deux places successives séparées par une transition comme présentée Figure 72. Cette transition a une garde toujours vraie.

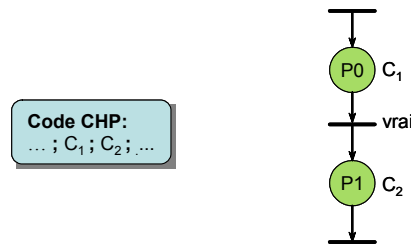


Figure 72 Représentation en réseau de Petri de l'opérateur de séquence

4.2.5 Opérateur de parallélisme

La Figure 73 illustre la représentation en réseau de Petri de l'opérateur de parallélisme. Dans cette représentation, les structures de divergence et de convergence en « ET » sont utilisées.

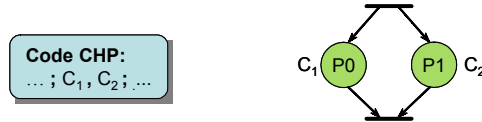


Figure 73 Représentation en réseau de Petri de l'opérateur de parallélisme

4.2.6 Opérateur de sélection

La représentation en réseau de Petri de l'opérateur de sélection est décrite Figure 74. Dans ce cas, les structures de divergence et de convergence en « OU » sont utilisées.

Ce réseau de Petri sert à modéliser l'opérateur de sélection déterministe et indéterministe. La propriété déterministe du choix est stockée dans la place de divergence (place $P0$ Figure 74) pour être traitée ultérieurement.

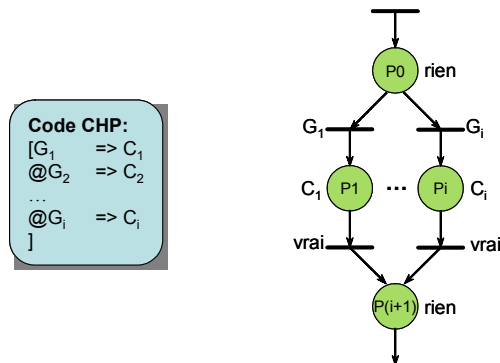


Figure 74 Représentation en réseau de Petri de l'opérateur de sélection

4.2.7 Opérateur de répétition

Contrairement à l'opérateur de sélection, la représentation de l'opérateur de répétition en réseau de Petri est une structure non symétrique. Quand aucune garde (de G_1 à G_i) n'est vraie, l'exécution de la répétition se termine.

Comme le réseau de Petri de l'opérateur de sélection, le réseau de Petri de l'opérateur de répétition peut modéliser l'opérateur de répétition déterministe et indéterministe.

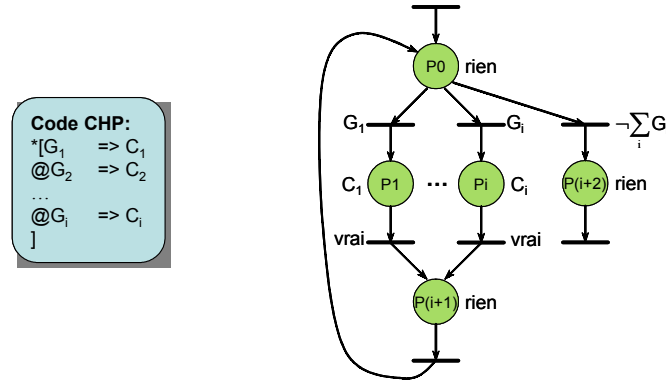


Figure 75 Représentation en réseau de Petri de l'opérateur de répétition

4.3 Vérification DTL du réseau de Petri

Les règles DTL définissent la forme synthétisable des circuits asynchrones. La vérification DTL a alors pour but de garantir que les circuits asynchrones modélisés par des réseaux de Petri peuvent être synthétisés par des étapes qui suivent. La vérification des règles DTL s'effectue sur le réseau de Petri. Elle se compose des étapes suivantes :

- Vérification de l'initialisation
- Vérification de l'accès séquentiel
- Vérification de l'accès parallèle
- Vérification du contexte

4.3.1 Vérification de l'initialisation

Selon les règles DTL annoncées précédemment (cf. §2.3.2), l'initialisation d'un canal de sortie est possible, même si cela se traduit par un accès séquentiel à ce canal. La vérification de l'initialisation consiste à vérifier que lorsqu'un canal est initialisé, tous les digits de données représentant dans le canal sont initialisés avec des valeurs constantes. Par ailleurs, il faut vérifier qu'un canal est initialisé une seule fois dans le processus.

4.3.2 Vérification de l'accès séquentiel

Cette vérification garantit qu'un canal de communication dans le réseau de Petri n'est pas accédé séquentiellement, ce qui nécessite des états supplémentaires. L'algorithme de vérification, présenté plus tard, suit le principe suivant : toutes les branches d'exécution du réseau de Petri sont parcourues. Pour chaque branche d'exécution du réseau, nous notons les accès aux canaux en passant les places en revue. Quand un canal est accédé en écriture une deuxième fois, nous vérifions si la première écriture correspond à une initialisation du canal. Si ce n'est pas le cas ou si un canal est accédé en lecture en séquence, nous signalons une violation de la règle DTL sur les accès séquentiels.

4.3.3 Vérification de l'accès parallèle

Nous vérifions qu'un canal de communication dans le réseau de Petri ne subit pas d'accès concurremment. Le principe de l'algorithme de vérification est le suivant : toutes les branches d'exécution du réseau de Petri sont parcourues. Pour les branches parallèles distinctes, les accès à chaque canal sont enregistrés. Ces accès sont ensuite comparés. Si un canal est accédé en écriture et concurremment, nous signalons une violation de la règle DTL sur les accès parallèles.

Cet algorithme, qui est présenté Chapitre 6, vérifie l'accès parallèle aux variables locales de manière identique.

4.3.4 Vérification du contexte

La vérification du contexte est utilisée pour garantir une vraie dépendance de données pour les affectations séquentielles. Elle doit vérifier que chaque variable/canal de sortie est affectée avec des valeurs connues. Cela signifie que ces valeurs doivent être calculées à partir des variables reçues par les canaux d'entrée. Cette vérification est facilement réalisée grâce à un environnement contextuel construit en passant les places du réseau de Petri en revue suivant le flot d'exécution. Cet environnement stocke les informations des variables telles que leur type de données, leur valeur. Le mécanisme de fonctionnement de l'environnement est le suivant.

En début de vérification, l'environnement est vide. En parcourant le réseau de Petri, chaque place est analysée. On ajoute alors un élément à l'environnement de la façon suivante:

- Quand on rencontre une lecture d'un canal d'entrée vers une variable locale, cette dernière est sauvegardée dans l'environnement avec ses informations comme, par exemple, le canal d'entrée. Si la variable de la lecture est déjà dans l'environnement, alors son ancienne entrée est remplacée par la nouvelle. Autrement dit, nous écrasons l'ancienne valeur.
- Quand on rencontre l'affectation d'une variable locale, l'expression de l'affectation est calculée en remplaçant les variables lues ou calculées auparavant dans l'environnement. Si l'expression d'affectation d'une variable locale contient une autre variable qui n'a pas été affectée précédemment, alors une erreur de dépendance est détectée. De ce fait, l'expression obtenue dépend uniquement des canaux d'entrée et des constantes. L'expression qui passe ce test est ensuite sauvegardée avec la variable dans l'environnement.
- Quand on rencontre une écriture dans un canal, l'expression d'écriture est calculée de la même manière que l'expression d'une affectation. Les variables dans l'expression sont remplacées par leurs valeurs sauvegardées dans l'environnement. Si l'une des variables n'est pas trouvée dans l'environnement, alors l'expression dépend d'une valeur inconnue et donc une erreur est levée.

On vérifie également dans ce parcours la compatibilité des types de données des opérandes, sauf l'opérateur d'affectation qui peut faire appeler la fonction de conversion des types. Il s'agit de vérifier l'égalité des bases de digits et des longueurs des opérandes (nombre de digits).

L'algorithme de vérification du contexte est décrit dans le Chapitre 6.

4.3.5 Conclusion sur la vérification DTL

La vérification DTL assure fondamentalement au niveau de la spécification que :

- les valeurs envoyées sur les canaux de sortie sont des fonctions combinatoires des valeurs lues dans les canaux d'entrée,
- il n'y a pas d'état implicite, sauf l'état nécessaire à l'initialisation des canaux. Il est nécessaire de mentionner que les états liés au protocole de communication ne sont pas considérés à ce

niveau. Ces états seront adressés plus tard dans l'étape de génération du réseau de portes où le protocole de communication est implémenté. Pour les états implicites aux canaux initialisés, nous les traitons de façon particulière dans le Chapitre 5,

- il existe une implémentation QDI en portes standard du circuit.

Certaines vérifications supplémentaires sont effectuées pour vérifier que le réseau de Petri DTL est synthétisable. Elle concerne la vérification :

- des opérateurs non synthétisables tels que la division, le modulo, ...
- des segments inatteignables du réseau de Petri. Par exemple, une garde ayant la condition toujours « faux » implique que le segment du réseau qui suit la transition est inatteignable,
- de l'étude de l'exclusion mutuelle des gardes pour les conditions de garde. Les gardes dans l'opérateur de sélection/répétition déterministe doivent être mutuellement exclusives. Cela signifie qu'il n'y a pas de valeurs qui rendent deux conditions vraies. Cette vérification est faite en utilisant les diagrammes de décision binaire (BDD) lorsque c'est possible.

A la fin de ces vérifications, le réseau de Petri obtenu est synthétisable et il existe une implémentation QDI en portes standard.

4.4 Transformation de réseaux de Petri DTL

4.4.1 Expansion des structures de choix

Dans la génération des équations de dépendances donnée dans la suite, l'expression d'écriture de chaque canal de sortie doit être produite. En général, un réseau de Petri d'un circuit asynchrone a plusieurs branches de choix dans lesquelles un même canal de sortie est affecté. La fonction de sortie pour ce canal est une réunion exclusive de toutes les fonctions de sortie des différentes branches.

Pour faciliter la recherche de ces fonctions, le réseau de Petri doit être déployé en plusieurs réseaux de Petri. Chaque réseau déployé n'a plus de branche de choix. En effet, il correspond à une branche d'exécution du programme avec un ensemble défini de conditions pour les choix qui définit une exécution. Par conséquent, cette transformation permet de calculer une seule fonction de sortie pour chaque canal de sortie et ce pour chaque branche d'exécution. L'expression finale pour chaque canal de sortie est ensuite l'union exclusive des expressions.

Un exemple est donné Figure 76 pour illustrer cette transformation.

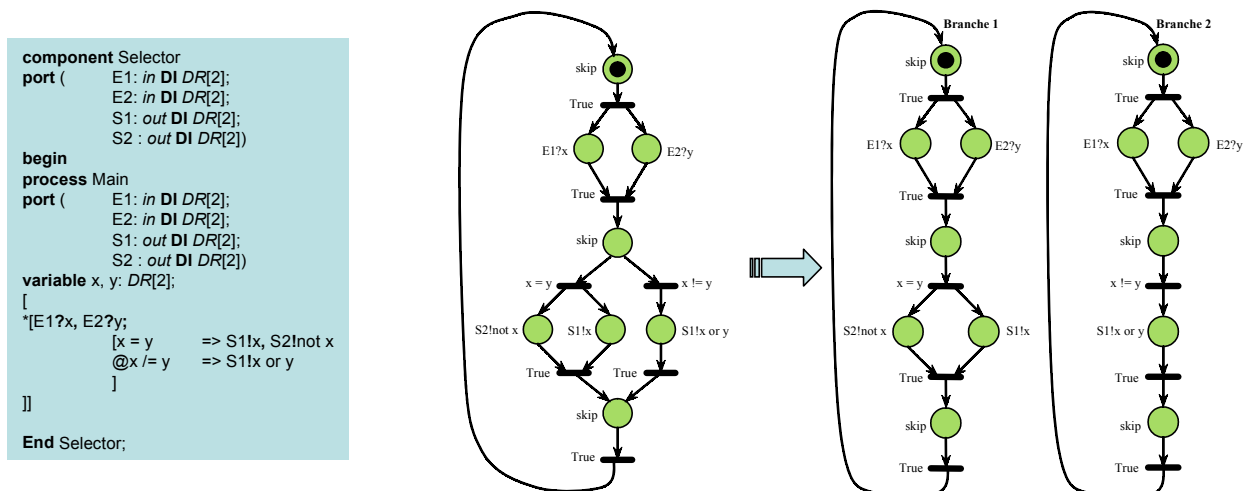


Figure 76 Expansion des structures de choix du réseau de Petri

4.4.2 Expansion des types de données

Sous sa forme générale, un type de données du langage CHP est représenté sous forme d'un vecteur de digits. Le type $MR[B][L]$ est un vecteur de L digits de base B . Un digit d'un canal de communication ou d'une variable locale peut être accessible indépendamment des autres dans le langage CHP. De ce fait, les expressions pour l'écriture de chaque digit d'un canal de sortie peuvent être différentes. C'est pourquoi les canaux et les variables d'un programme CHP doivent être exprimés au niveau des digits.

La deuxième transformation effectuée sur le réseau de Petri est donc d'expanser sous forme de digits les canaux de communication et les variables locales dans toutes les expressions, qui sont représentées par des DFG. A titre d'exemple, une place de réseau de Petri d'une communication $S!x$, où le canal S et la variable x sont de type $MR[2][3]$, est expansé en trois places comme illustrées Figure 77. Cette forme permet de générer facilement les équations de dépendances pour chaque digit des canaux de sortie. Le réseau de Petri avec les canaux de communication et variables développés s'appelle le réseau de Petri expansé au niveau digit.

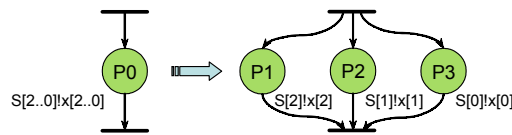


Figure 77 Expansion le réseau de Petri sous forme de digits

4.5 Génération des équations de dépendances

Dans cette partie, nous présentons la génération des équations de dépendances, une des étapes importantes de la synthèse de circuits asynchrones. Nous commençons par introduire les équations de dépendances et leurs parties constituantes. Ensuite nous adressons la manière de générer les équations de dépendances correspondantes aux structures de pipeline présentées dans le Chapitre 3. En se basant sur cette génération, nous discutons la génération des équations de dépendances d'un circuit asynchrone quelconque. Puis, un exemple est donné pour illustrer cette méthode. Des règles d'optimisation des équations de dépendances sont ensuite proposées.

4.5.1 Définition

L'équation de dépendances est une équation conventionnelle utilisée pour formaliser le fonctionnement d'un circuit asynchrone. Comme nous l'avons montré dans le paragraphe 2.3, étant conforme à la spécification DTL, un circuit asynchrone peut être exprimé comme un circuit dans lequel les sorties ne dépendent que des entrées. La relation de dépendance entre les sorties et les entrées du circuit est représentée par des équations de dépendances des sorties en fonction des entrées.

Les équations de dépendances d'un circuit asynchrone expriment la sémantique des canaux de communication. Elles contiennent toutes les informations qui font changer les sorties du circuit. Ces paramètres incluent :

- la validité des données sur les canaux d'entrée dont dépend la sortie,
- les conditions qui gardent l'émission de la donnée sur la sortie considérée,
- la disponibilité du canal de sortie considéré,
- et la fonction de calcul du canal de sortie considéré en fonction des canaux d'entrée.

Pour les circuits asynchrones encodés avec un codage insensible aux délais (cf. §1.2.2), il existe deux types de sortie : les données des canaux de sortie et les acquittements des canaux

d'entrée. De ce fait, il y a deux types d'équations de dépendances : l'un pour les rails de données des canaux de sortie et l'autre pour les signaux d'acquittement des canaux d'entrée. Nous définissons :

- E_i : un canal d'entrée du circuit
- E_i^{acq} : le signal d'acquittement du canal E_i
- S_k : un canal de sortie du circuit.
- S_k^{acq} : le signal d'acquittement du canal S_k . (Il est à noter que la disponibilité d'un canal de sortie est représentée par la présence de son signal d'acquittement.)
- S_E : le canal de sortie vers lequel la donnée issue du canal d'entrée E est envoyée.
- S : une sortie de données du circuit.
- $\sum_i V(E_i)$: les validités des données des canaux d'entrée E_i . (Pour les circuits asynchrones avec un codage insensible aux délais, la validité des données sur un canal d'entrée E_i implique que tous les digits du canal doivent porter des valeurs valides. Une valeur valide avec un codage « 1-parmi-N » est représentée par N fils de sorte qu'à tout moment, un et seulement un fil est actif. Alors, le test de validité des données sur E_i est réalisé par le test de l'ensemble des digits de E)
- $\sum_j G_j$: les conditions qui gardent l'émission de la donnée sur une sortie.
- F_S : la fonction de calcul de la sortie S . (L'implémentation de cette fonction sera présentée Chapitre 5)

Ainsi, l'équation de dépendances générale pour la sortie de données S est exprimée par :

Equation 4-1
$$S = F_{dep}^{données}(\sum_i V(E_i); \sum_j G_j; S_E^{acq}; F_S)$$

et l'équation de dépendances générale pour le signal d'acquittement d'un canal d'entrée E par :

Equation 4-2
$$E^{acq} = F_{dep}^{acq}(\sum_i V(E_i); \sum_j G_j; S_E^{acq}; V(S_E))$$

Dans ces deux équations de dépendances, $F_{dep}^{données}$ symbolise la fonction de dépendances pour les données des canaux de sortie, et F_{dep}^{acq} symbolise celle pour le signal d'acquittement des canaux d'entrée.

Ces équations sont interprétées de la façon suivante : si les conditions de garde G_j sont satisfaites, que les données sont présentes sur les canaux d'entrée E_i et que le canal de sortie est prêt à recevoir des données, alors la donnée calculée par la fonction F_S est émise sur la sortie S . Dans la même situation plus la validité de données sur les canaux de sortie, le signal d'acquittement E^{acq} passe au niveau actif. Les arguments conditionnels sont donnés pour le cas général afin de couvrir tous les cas. Par conséquent, ils peuvent être redondants. Pour un certain modèle de circuits, il est possible de les simplifier. Les simplifications pour les différents modèles de circuits seront abordées Chapitre 5.

Pour les circuits asynchrones avec un codage insensible aux délais, un canal de communication est représenté par les digits de données de N rails plus un signal d'acquittement. La dépendance de données entre les sorties et les entrées peut être représentée en deux niveaux d'abstraction : niveau des canaux (niveau digit) et niveau des signaux.

- Au niveau des canaux, les équations de dépendances sont exprimées en digits. Chaque digit des canaux de sortie est généralement exprimé en fonction des digits des canaux d'entrée et des signaux d'acquiescement des canaux de sortie. Pour une représentation compacte, si tous les digits d'un canal de sortie sont homogènes, ce canal peut être parfois représenté en une seule équation de dépendances.
- Par contre au niveau des signaux, chaque rail de données (fil de signal) des canaux de sortie est explicitement calculé en fonction des rails de données des canaux d'entrée et des signaux d'acquiescement des canaux de sortie.

Les équations de dépendances ci-dessus (Equation 4-1 et Equation 4-2) peuvent s'appliquer aux deux niveaux d'abstraction. En particulier, la sortie S dans l'Equation 4-1 peut être un rail de données d'un canal de sortie au niveau des signaux ou le canal lui-même au niveau des canaux. Pour une meilleure compréhension, les équations de dépendances d'un circuit asynchrone sont détaillées pour les deux niveaux d'abstraction.

4.5.1.1 Equations de dépendances au niveau des canaux

Au niveau d'abstraction des canaux, les équations de dépendances traitent les digits des canaux. Cela signifie que dans toutes les expressions de gardes et de données, les canaux de communication et les variables locales sont représentées sous forme de digits.

Pour illustrer les équations de dépendances au niveau des canaux d'un circuit asynchrone, nous supposons que E et S symbolisent respectivement des canaux d'entrée et des canaux de sortie ayant L digits de B rails. En langage CHP, ces canaux sont déclarés comme le type de données $MR[B][L]$. Cette hypothèse est générale car tous les types de données du langage CHP peuvent être exprimés en type $MR[B][L]$.

- L'équation de dépendances de chaque digit du canal S est indiquée dans l'Equation 4-3. S_n et F_{S_n} sont respectivement le $n^{\text{ème}}$ digit du canal S et sa fonction de calcul.

Equation 4-3
$$S_n = F_{dep}^{données} \left(\sum_i V(E_i); \sum_j G_j; S^{acq}; F_{S_n} \right)$$

En général, les équations de dépendances des digits du canal S sont différentes car chaque digit peut être individuellement accédé dans le langage CHP. Pour constituer le canal S , ses digits sont regroupés comme spécifié par l'Equation 4-4 suivante :

Equation 4-4
$$S = F_{regroupe} \left(\sum_{n=0}^{L-1} S_n \right)$$

- L'équation de dépendances du signal d'acquiescement du canal E est déduite à partir de l'Equation 4-2 :

Equation 4-5
$$E^{acq} = F_{dep}^{acq} \left(\sum_i V(E_i); \sum_j G_j; S_E^{acq}; V(S_E) \right)$$

Il est important de mentionner que dans le cas où la donnée issue du canal E est envoyée vers plusieurs canaux S_{E_i} différents, il faut générer l'Equation 4-5 pour chaque canal S_{E_i} distinct. Cette génération correspond en fait à la génération de chaque acquiescement des étages dans la fourche, ce pour résoudre le problème lié à la fourche (cf. §3.1.2.3). Le signal d'acquiescement E^{acq} est alors un rendez-vous de ces sorties. L'équation de dépendances du signal d'acquiescement du canal E devient comme suit :

Equation 4-6
$$E_i^{acq} = F_{dep}^{acq} \left(\sum_i V(E_i); \sum_j G_j; S_{E_i}^{acq}; V(S_{E_i}) \right)$$

Equation 4-7
$$E^{acq} = F_{RDV}(\sum_i E_i^{acq})$$

Les expressions dans $\sum_j G_j$ et F_S des équations Equation 4-3, Equation 4-5 et Equation 4-6 sont représentées sous forme de digits.

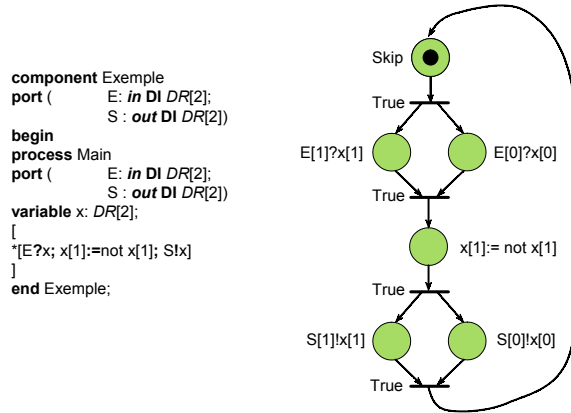


Figure 78 Exemple de multi-rails en langage CHP et son réseau de Petri

A titre d'exemple, un programme CHP d'un exemple et son réseau de Petri sont donnés Figure 78. Nous discutons ultérieurement la façon de générer les équations de dépendances de cet exemple (cf. §4.5.3). Les équations de dépendances au niveau des canaux sont données :

Equation 4-8
$$\begin{cases} S_1 = F_{dep}^{données}(V(E); S^{acq}; F_{S_1} = not E_1) \\ S_0 = F_{dep}^{données}(V(E); S^{acq}; F_{S_0} = E_0) \\ S = F_{regroupe}(S_1, S_0) \\ E^{acq} = F_{dep}^{acq}(V(E); S^{acq}; V(S)) \end{cases}$$

Dans cet exemple, comme les données issues du canal E sont émises inconditionnellement sur le canal S , il n'existe pas d'expression de garde $\sum_j G_j$ dans le système d'Equation 4-8.

4.5.1.2 Equations de dépendances au niveau des signaux

Au niveau d'abstraction des signaux, les équations de dépendances traitent exclusivement les signaux des canaux. Cela signifie que les équations de dépendances sont exprimées à partir des rails de données et que dans toutes les expressions de gardes et de données, les digits des canaux de communication et les variables locales sont expansés en signaux.

Pour illustrer les équations de dépendances au niveau des signaux d'un circuit asynchrone, nous reprenons les définitions du paragraphe précédent (voir §4.5.1.1).

- L'équation de dépendances de chaque rail d'un digit du canal S est indiquée dans l'Equation 4-9. Dans cette équation, S_{d-r} est le $r^{ième}$ rail du $d^{ième}$ digit du canal S et $F_{S_{d-r}}$ est la fonction de calcul correspondante.

Equation 4-9
$$S_{d-r} = F_{dep}^{données}(\sum_i V(E_i); \sum_j G_j; S^{acq}; F_{S_{d-r}})$$

Pour constituer le canal S , les rails sont d'abord regroupés en digits et les digits sont à leur tour regroupés comme spécifiés par Equation 4-10 et Equation 4-11

Equation 4-10
$$S_d = F_{regroupe}^{rail}(\sum_{r=0}^{B-1} S_{d-r})$$

$$\text{Equation 4-11} \quad S = F_{regroupe}^{digit} \left(\sum_{d=0}^{L-1} S_d \right)$$

- L'équation de dépendances au niveau des signaux du signal d'acquittement du canal E est la même que celle au niveau des canaux sauf que dans cette équation, les expressions dans $\sum_j G_j$ sont expansées au niveau signal :

$$\text{Equation 4-12} \quad E^{acq} = F_{dep}^{acq} \left(\sum_i V(E_i); \sum_j G_j; S_E^{acq}; V(S_E) \right)$$

Dans le cas où la donnée issue du canal E est envoyée vers plusieurs canaux S_{E_i} différents, il faut générer l'Equation 4-12 pour chaque canal S_{E_i} distinct. Le signal d'acquittement E^{acq} est alors un rendez-vous de ces sorties. L'équation de dépendances du signal d'acquittement du canal E devient alors :

$$\text{Equation 4-13} \quad E_i^{acq} = F_{dep}^{acq} \left(\sum_i V(E_i); \sum_j G_j; S_{E_i}^{acq}; V(S_{E_i}) \right)$$

$$\text{Equation 4-14} \quad E^{acq} = F_{RDV} \left(\sum_i E_i^{acq} \right)$$

Les expressions dans $\sum_j G_j$ et F_{S_d-r} des équations Equation 4-9, Equation 4-12 et Equation 4-13 traitent des signaux.

A titre d'exemple, les équations de dépendances de l'exemple de la Figure 78 sont exprimées au niveau des signaux :

$$\text{Equation 4-15} \quad \begin{cases} S_{1_1} = F_{dep}^{donnees} (V(E); S^{acq}; F_{S_{1_1}} = E_{1_0}) \\ S_{1_0} = F_{dep}^{donnees} (V(E); S^{acq}; F_{S_{1_0}} = E_{1_1}) \\ S_1 = F_{regroupe}^{rail} (S_{1_1}, S_{1_0}) \\ S_{0_1} = F_{dep}^{donnees} (V(E); S^{acq}; F_{S_{0_1}} = E_{0_1}) \\ S_{0_0} = F_{dep}^{donnees} (V(E); S^{acq}; F_{S_{0_0}} = E_{0_0}) \\ S_0 = F_{regroupe}^{rail} (S_{0_1}, S_{0_0}) \\ S = F_{regroupe}^{digit} (S_1, S_0) \\ E^{acq} = F_{dep}^{acq} (V(E); S^{acq}, V(S)) \end{cases}$$

En résumé, les équations de dépendances des sorties de circuits asynchrones sont présentées aux deux niveaux d'abstraction. Elles ne font aucune hypothèse sur l'implémentation de circuits (modèle de circuits, type de circuits, technologie cible). Elles n'expriment que les dépendances des sorties en fonction des entrées.

4.5.2 Equations de dépendances des structures de base du réseau de Petri

Avant d'explicitier la génération des équations de dépendances d'un circuit asynchrone, il est nécessaire de discuter la génération des équations de dépendances des structures de base des réseaux de Petri. Ces dernières correspondent aux structures de pipeline abordées dans le Chapitre 3 : pipeline linéaire, fourche et convergence. Rappelons que les équations de dépendances d'un circuit asynchrone sont générées à partir des réseaux de Petri vérifiés et transformés. Ainsi les structures de choix n'existent plus, et ne sont donc pas considérées.

Les structures de base sont représentées dans ce paragraphe sous forme de programmes CHP et des réseaux de Petri correspondants. Les équations de dépendances données dans la suite sont indiquées généralement au niveau des canaux. Les équations de dépendances au niveau des signaux sont déduites facilement si la fonction de sortie et la condition de garde sont connues.

4.5.2.1 Equations de dépendances de la structure place/transition/place

- **Le code CHP et son réseau de Petri**

Les programmes CHP qui font apparaître cette structure séquentielle ont les formes suivantes :

...C1 ; C2...

ou

...C1 ; [G => C2]

Dans ces programmes, $C1$ et $C2$ sont les commandes CHP et G est une condition de garde. Les réseaux de Petri de ces programmes sont donnés Figure 79 ci-dessous. Pour le premier programme CHP, il peut être considéré que la condition de garde G porte la valeur vraie (« True »). Les conditions de garde « vraie » peuvent être oubliées dans les équations de dépendances. Dans le but de généraliser la génération des équations de dépendances de cette structure, nous mettons systématiquement la garde G .

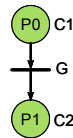


Figure 79 Réseau de Petri de la structure séquentielle

Ces programmes signifient que la commande $C2$ est exécutée quand la commande $C1$ a été exécutée et que la condition G est satisfaite (sa valeur évaluée « vraie »).

- **Equations de dépendances :**

Equation 4-16
$$S_{C2} = F_{dep}^{données}(V(C1); G; S_{C2}^{acq}; F_{S_{C2}})$$

Equation 4-17
$$E_{C1}^{acq} = F_{dep}^{acq}(V(C1); G; S_{C2}^{acq}; V(S_{C2}))$$

Dans ces équations, S_{C2} est le canal de sortie de la commande $C2$ et E_{C1}^{acq} est le signal d'acquiescement du bloc $C2$ pour le bloc $C1$. La validité des canaux d'entrée dans $C1$ est représentée par $V(C1)$. $F_{S_{C2}}$ symbolise la fonction de calcul du S_{C2} .

- **Exemple :**

Un simple exemple de cette catégorie est donné comme suit

... E ?x ; S !x ...

Dans cet exemple, la commande $C1$ contient un canal d'entrée (E) et la commande $C2$ un canal de sortie (S). Puisque la condition de garde est toujours « vraie », elle est enlevée dans les équations de dépendances, qui sont présentées ci-dessous :

Equation 4-18
$$S = F_{dep}^{données}(V(E); S^{acq}; F_S = E)$$

Equation 4-19
$$E^{acq} = F_{dep}^{acq}(V(E); S^{acq}; V(S))$$

4.5.2.2 Equations de dépendances de la structure de divergence en « ET »

- **Le code CHP et son réseau de Petri**

Les programmes CHP qui font apparaître la structure de divergence en « ET » ont les formes suivantes :

...C1 ; C2, C3...

ou

...C1 ; [G => C2, C3]

Les réseaux de Petri de ces programmes sont décrits Figure 80. Les commandes $C2$ et $C3$ sont parallèlement exécutées une fois que la commande $C1$ est finie et que la condition G est évaluée « vraie ».

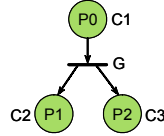


Figure 80 Réseau de Petri de la structure de divergence en « ET »

- **Equations de dépendances :**

Equation 4-20 $S_{C2} = F_{dep}^{données}(V(C1); G; S_{C2}^{acq}; F_{S_{C2}})$

Equation 4-21 $S_{C3} = F_{dep}^{données}(V(C1); G; S_{C3}^{acq}; F_{S_{C3}})$

Equation 4-22 $E_{C1_C2}^{acq} = F_{dep}^{acq}(V(C1); G; S_{C2}^{acq}; V(S_{C2}))$

Equation 4-23 $E_{C1_C3}^{acq} = F_{dep}^{acq}(V(C1); G; S_{C3}^{acq}; V(S_{C3}))$

Equation 4-24 $E_{C1}^{acq} = F_{RDV}(E_{C1_C2}^{acq}, E_{C1_C3}^{acq})$

Comme nous l'avons étudié dans le Chapitre 3, l'acquittement de l'étage de fourche ($C1$) doit résoudre le problème lié : à savoir quand l'une des branches de la fourche est lente ou bloquée. De ce fait, les signaux d'acquittement des blocs $C2$ et $C3$ apparaissent dans l'équation de dépendances du signal d'acquittement de $C1$. Ce système d'équation de dépendances est toujours correct même si les données issues de $C1$ ne sont pas envoyées vers tous les canaux de sortie de $C2$ et de $C3$. Ce qui compte dans cette structure est que $C1$ déclenche simultanément les fonctionnements des blocs $C2$ et $C3$. Par conséquent, le bloc $C1$ est acquitté quand les blocs $C2$ et $C3$ ont été acquittés.

- **Exemple :**

Un simple exemple de cette catégorie est donné comme suit :

... E ?x ; S1 !x, S2 !x ...

Les équations de dépendances de cet exemple sont les suivantes :

Equation 4-25 $S1 = F_{dep}^{données}(V(E); S1^{acq}; F_{S1} = E)$

Equation 4-26 $S2 = F_{dep}^{données}(V(E); S2^{acq}; F_{S2} = E)$

Equation 4-27 $E_{S1}^{acq} = F_{dep}^{acq}(V(E); S1^{acq}; V(S1))$

Equation 4-28 $E_{S2}^{acq} = F_{dep}^{acq}(V(E); S2^{acq}; V(S2))$

Equation 4-29 $E^{acq} = F_{RDV}(E_{S1}^{acq}, E_{S2}^{acq})$

4.5.2.3 Equations de dépendances de la structure de convergence en « ET »

- **Le code CHP et son réseau de Petri :**

Les programmes CHP qui font apparaître la structure de convergence en « ET » ont les formes suivantes :

...C1, C2 ; C3...

ou

...C1, C2 ; [G => C3]

Les réseaux de Petri de ces programmes sont décrits Figure 81. La commande $C3$ est exécutée quand les commandes $C1$ et $C2$ sont terminées et quand la condition G est évaluée « vraie ».

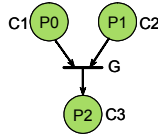


Figure 81 Réseau de Petri de la structure de convergence en « ET »

- **Equations de dépendances :**

Equation 4-30 $S_{C3} = F_{dep}^{données}(V(C1), V(C2); G; S_{C3}^{acq}; F_{S_{C3}})$

Equation 4-31 $E_{C1}^{acq} = F_{dep}^{acq}(V(C1), V(C2); G; S_{C3}^{acq}; V(S_{C3}))$

Equation 4-32 $E_{C2}^{acq} = F_{dep}^{acq}(V(C1), V(C2); G; S_{C3}^{acq}; V(S_{C3}))$

Une remarque s'impose sur la génération des équations de dépendances de cette structure de convergence : Equation 4-31 et Equation 4-32 sont identiques. Ceci est justifié par le fait que les entrées de la convergence sont acquittées simultanément. Par contre, quand les données arrivent sur une des entrées de la convergence, il faut attendre les données sur les autres entrées. C'est pourquoi nous testons la validité des deux entrées dans l'équation de données du bloc $C3$.

- **Exemple :**

Un simple exemple de cette catégorie est donné comme suit :

... E1 ?x, E2 ?y ; [x=y => S !x] ...

Les équations de dépendances de cet exemple sont les suivantes :

Equation 4-33 $S = F_{dep}^{données}(V(E1), V(E2); x = y; S^{acq}; F_S = E1)$

Equation 4-34 $E1^{acq} = E2^{acq} = F_{dep}^{acq}(V(E1), V(E2); x = y; S^{acq}; V(S))$

4.5.3 Génération des équations de dépendances d'un réseau de Petri quelconque

Nous présentons dans ce paragraphe la génération des équations de dépendances à partir des réseaux de Petri qui sont conformes à la spécification DTL ([DINH02c][DINH02d]). Les circuits asynchrones, représentés par les réseaux de Petri conformes aux règles DTL, se trouvent sous la forme « consommation; production ». Cette structure est équivalente au programme CHP « $E_i?; S_i!$ » dans lequel il y a d'abord consommation des canaux d'entrée en parallèle, puis production concurrente des canaux de sortie. En particulier, les sorties du circuit peuvent être uniquement calculées en fonction des entrées courantes, non pas de valeurs du passé. Les variables locales jouent un rôle intermédiaire : elles servent de containers de données à propager les valeurs lues en entrées vers les sorties. Par ailleurs, elles ne doivent plus être visibles dans les fonctions finales des canaux de sortie.

Dans cet esprit, un environnement contextuel est construit pour chaque réseau de Petri. Il évolue au cours du parcours du réseau de Petri pour garder les informations sur les valeurs des variables locales. Sa réalisation se fait comme suit : lorsque l'on rencontre une écriture dans une

variable locale, la valeur de cette dernière est calculée par la substitution des variables opérandes par leurs valeurs (stockées dans l'environnement). En particulier, la valeur d'une variable écrite à partir d'un canal d'entrée, est identique à la valeur de ce canal. La valeur de la variable locale ne dépend donc que des données de canaux d'entrée. Une vérification est ensuite réalisée pour s'assurer de l'existence de cette variable dans l'environnement. Dans le cas où elle est inexistante, elle est rajoutée avec sa nouvelle valeur calculée. Dans le cas contraire, son ancienne valeur est écrasée.

Comme le montre §4.2, le programme décrivant un circuit asynchrone est compilé en un ensemble de réseaux de Petri. Chaque réseau de Petri correspond à une seule branche d'exécution (*i.e.* un choix) du programme original. Pour générer les équations de dépendances d'un circuit asynchrone, celles de chaque réseau de Petri sont d'abord créées. Les équations de dépendances de chaque canal utilisé sont ensuite obtenues en réunissant exclusivement les équations issues de chacun des réseaux de Petri. Cette opération est illustrée par l'Equation 4-35 avec S^c le canal de sortie correspondant au choix c et N le nombre de réseaux de Petri (ou nombre de choix dans la spécification CHP). $E^{c/acq}$ symbolise le signal d'acquiescement du canal E qui correspond au choix c .

Equation 4-35

$$\begin{cases} E^{acq} = F_{union}^{acq} \left(\sum_{c=0}^{N-1} E^{c/acq} \right) \\ S = F_{union}^{donnees} \left(\sum_{c=0}^{N-1} S^c \right) \end{cases}$$

Les équations de dépendances du circuit asynchrone, représenté par l'ensemble de réseaux de Petri, sont générées par l'algorithme suivant.

Pour chaque réseau dans l'ensemble de réseaux de Petri

Pendant le parcours:

Construire un environnement contextuel correspondant

Enregistrer les canaux d'entrée consommés

Mémoriser les conditions de garde en passant les transitions

Calculer, grâce aux valeurs des variables dans l'environnement contextuel, la fonction de sortie de chaque canal de sortie utilisé afin d'obtenir la fonction qui dépend uniquement des valeurs de canaux d'entrée

Former l'équation de dépendances pour chaque canal de sortie avec les dépendances nécessaires

Générer l'équation de dépendances pour l'acquiescement de chaque canal d'entrée consommé

Créer les équations de dépendances des canaux en unissant les équations de chaque réseau de Petri (l'Equation 4-35).

- **Exemple d'un multiplexeur**

Pour illustrer la méthode de génération d'équations de dépendances proposée, nous présentons un exemple de multiplexage. Ce multiplexeur, décrit par le code CHP suivant, lit un des canaux d'entrée selon la valeur du canal de contrôle *Ctrl*. Puis, il réalise le calcul nécessaire sur la valeur lue. La valeur calculée dans chaque choix est finalement émise sur le canal de sortie. Pour montrer la génération des équations de dépendances de canaux multi-digits, les canaux sont déclarés en type MR[2][2].

```

component multiplexer
port (  Ctrl : in DI DR;
        InMux0 : in DI MR[2][2];
        InMux1 : in DI MR[2][2];
        OutMux : out DI MR[2][2])
begin

process Main
port (  Ctrl : in DI DR;
        InMux0 : in DI MR[2][2];
        InMux1 : in DI MR[2][2];
        OutMux : out DI MR[2][2])
variable c: DR;
variable x: MR[2][2];
[
*[Ctrl?c;
  [c = 0 => InMux0?x; y := x;
  @c = 1 => InMux1?x; y := not x;
];
  OutMux!y
]
]]
end multiplexer;

```

Le programme CHP est d'abord compilé en format intermédiaire : le réseau de Petri et les DFG (Figure 82a). Grâce aux transformations présentées dans §4.4, ce réseau est ensuite transformé en plusieurs réseaux de Petri qui ne possèdent plus de choix (Figure 82b).

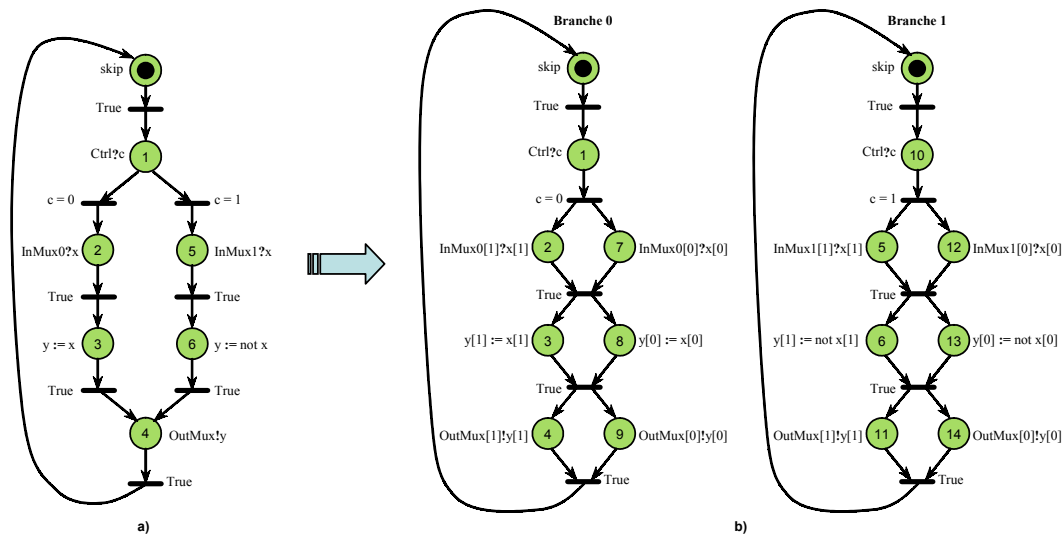


Figure 82 Réseaux de Petri du multiplexeur

Pour simplifier l'illustration, les équations de dépendances sont seulement représentées au niveau des canaux. Les équations de dépendances au niveau des signaux sont facilement déductibles.

Pour le premier réseau de Petri de la Figure 82b, en appliquant l'algorithme de la génération des équations de dépendances ci-dessus, les dépendances entre les canaux sont données Tableau 1.

	Variable/Canal	Valeur	Condition de garde
	C	Ctrl	-
	X[1]	InMux0[1]	c = 0
	X[0]	InMux0[0]	c = 0
	Y[1]	InMux0[1]	c = 0
	Y[0]	InMux0[0]	c = 0
Canaux de sortie	OutMux[1]	InMux0[1]	c = 0
	OutMux[0]	InMux0[0]	c = 0
Canaux d'entrée consommés	Ctrl	OutMux	c = 0
	InMux0	OutMux	c = 0

Tableau 1 Les dépendances entre les canaux du premier PN (branche 0)

Il est à noter que dans ce tableau, la colonne « valeur » contient les expressions/fonctions pour les variables locales et les canaux de sortie. Dans le cas des canaux d'entrée, elle contient les canaux de sortie vers lesquels les données du canal d'entrée sont envoyées. A partir de ce tableau, les équations de dépendances du premier réseau de Petri sont générées.

$$\text{Equation 4-36} \quad \begin{cases} OutMux_0^0 = F_{dep}^{données}(V(Ctrl), V(InMux0); Ctrl = 0; OutMux^{acq}; InMux0_0) \\ OutMux_1^0 = F_{dep}^{données}(V(Ctrl), V(InMux0); Ctrl = 0; OutMux^{acq}; InMux0_1) \end{cases}$$

$$\text{Equation 4-37} \quad \begin{cases} Ctrl^{0/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux0); Ctrl = 0; OutMux^{acq}; V(OutMux^0)) \\ InMux0^{0/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux0); Ctrl = 0; OutMux^{acq}; V(OutMux^0)) \end{cases}$$

Les équations de dépendances pour les signaux d'acquittement des canaux *Ctrl* et *InMux0* dans ce cas sont identiques. Le signal d'acquittement du canal *Ctrl* est alors conduit par celui du canal *InMux0*.

De la manière identique, le tableau récapitulatif du deuxième réseau de Petri est également créé comme le montré Tableau 2.

	Variable/Canal	Valeur	Condition de garde
	C	Ctrl	-
	X[1]	InMux1[1]	c = 1
	X[0]	InMux1[0]	c = 1
	Y[1]	not InMux1[1]	c = 1
	Y[0]	not InMux1[0]	c = 1
Canaux de sortie	OutMux[1]	not InMux1[1]	c = 1
	OutMux[0]	not InMux1[0]	c = 1
Canaux d'entrée consommés	Ctrl	OutMux	c = 1
	InMux1	OutMux	c = 1

Tableau 2 Les dépendances entre les canaux du deuxième PN (branche 1)

Les équations de dépendances pour ce réseau de Petri sont les suivantes :

$$\text{Equation 4-38} \quad \begin{cases} OutMux_0^1 = F_{dep}^{données}(V(Ctrl), V(InMux1); Ctrl = 1; OutMux^{acq}; not InMux1_0) \\ OutMux_1^1 = F_{dep}^{données}(V(Ctrl), V(InMux1); Ctrl = 1; OutMux^{acq}; not InMux1_1) \end{cases}$$

$$\text{Equation 4-39} \quad \begin{cases} Ctrl^{1/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux1); Ctrl = 1; OutMux^{acq}; V(OutMux^1)) \\ InMux1^{1/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux1); Ctrl = 1; OutMux^{acq}; V(OutMux^1)) \end{cases}$$

Comme l'Equation 4-37, le signal d'acquittement du canal *Ctrl* dans ce réseau de Petri peut être conduit par celui du canal *InMux1*.

Les équations de dépendances finales du multiplexeur sont générées en unissant les équations de dépendances de chaque réseau :

$$\text{Equation 4-40} \quad \begin{cases} OutMux_0 = F_{union}^{données}(OutMux_0^0, OutMux_0^1) \\ OutMux_1 = F_{union}^{données}(OutMux_1^0, OutMux_1^1) \end{cases}$$

$$\text{Equation 4-41} \quad \begin{cases} InMux0^{acq} = InMux0^{0/acq} \\ InMux1^{acq} = InMux1^{1/acq} \\ Ctrl^{acq} = F_{union}^{acq}(Ctrl^{0/acq}, Ctrl^{1/acq}) \end{cases}$$

4.5.4 Génération des équations de dépendances : une solution alternative

Nous présentons dans ce paragraphe une solution alternative pour la génération des équations de dépendances. Une notion à noter lors de la discussion est que les canaux de sortie présentés dans ce paragraphe peuvent être les canaux de sortie primaires du circuit ou bien les variables locales. Une variable locale est considérée comme un canal de communication interne comprenant, avec un codage insensible aux délais « 1-parmi-N », les fils de données plus un fil d'acquittement. Ces canaux internes sont justement les canaux intermédiaires car ils ne créent pas d'état supplémentaire. Ces canaux internes doivent disparaître dans les équations de dépendances finales du circuit. Mais en raison de la clarté des équations de dépendances générées et de la facilité de la génération de circuit à la suite, nous présentons ici les équations de dépendances contenant les variables locales traitées comme des canaux.

Dans cet esprit, la génération des équations de dépendances est réalisée comme décrit par l'algorithme suivant.

Pour chaque réseau dans l'ensemble des réseaux de Petri

Pendant le parcours récursif:

Reconnaître les structures de base du réseau de Petri (cf. §4.5.2)

Générer les équations de dépendances pour chaque canal (y compris les variables locales)

Remplacer cette structure par une place fictive

Reformer l'équation de dépendances pour chaque canal de sortie avec les dépendances nécessaires

Générer l'équation de dépendances pour l'acquittement de chaque canal d'entrée consommé

Créer les équations de dépendances des canaux en unissant les équations de chaque réseau de Petri (l'Equation 4-35).

- **Exemple**

L'exemple de multiplexage du paragraphe précédent est repris. Chaque réseau de Petri est séparément analysé lors de la génération des équations de dépendances. En considérant les variables locales comme des canaux, les équations de dépendances sont récursivement créées en appliquant la génération des équations de dépendances sur les structures de base reconnues. En particulier, les places 1, 2 et 7 dans le premier réseau de Petri (Figure 82b) constituent une structure de divergence

en « ET ». Les équations de dépendances de cette structure sont représentées par l'Equation 4-42. Le chiffre exposant dans ces équations spécifie le réseau de Petri traité. Par exemple, X_1^0 symbolise le deuxième digit du canal X du premier réseau de Petri, tandis que X^0 représente le canal entier pour le premier réseau de Petri.

$$\text{Equation 4-42} \quad \begin{cases} X_1^0 = F_{dep}^{données}(V(Ctrl), V(InMux0); Ctrl = 0; X^{acq}; F_{X_1^0} = InMux0_1) \\ X_0^0 = F_{dep}^{données}(V(Ctrl), V(InMux0); Ctrl = 0; X^{acq}; F_{X_0^0} = InMux0_0) \\ Ctrl^{0/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux0); Ctrl = 0; X^{acq}; V(X^0)) \\ InMux0^{0/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux0); Ctrl = 0; X^{acq}; V(X^0)) \end{cases}$$

Cette structure de fourche est ensuite considérée comme une place fictive contenant une communication sur le canal X . Cette place et les places 3 et 8 constituent une autre structure de divergence en « ET ». En appliquant encore une fois la génération associée aux divergence en « ET », nous obtenons

$$\text{Equation 4-43} \quad \begin{cases} Y_1^0 = F_{dep}^{données}(V(X); Y^{acq}; F_{Y_1^0} = X_1^0) \\ Y_0^0 = F_{dep}^{données}(V(X); Y^{acq}; F_{Y_0^0} = X_0^0) \\ X^{0/acq} = F_{dep}^{acq}(V(X); Y^{acq}; V(Y^0)) \end{cases}$$

A titre de remarque, puisque aucune condition de garde n'est nécessaire pour l'exécution des places 3 et 8, seule la validité du canal X est exigé.

Pareillement, les équations de dépendances pour les places 4 et 9 sont générées :

$$\text{Equation 4-44} \quad \begin{cases} OutMux_1^0 = F_{dep}^{données}(V(Y); OutMux^{acq}; F_{OutMux_1^0} = Y_1^0) \\ OutMux_0^0 = F_{dep}^{données}(V(Y); OutMux^{acq}; F_{OutMux_0^0} = Y_0^0) \\ OutMux^0 = F_{regroupe}(OutMux_1^0, OutMux_0^0) \\ Y^{0/acq} = F_{dep}^{acq}(V(Y); OutMux^{acq}; V(OutMux^0)) \end{cases}$$

Les systèmes d'équation Equation 4-42, Equation 4-43 et Equation 4-44 constituent les équations de dépendances correspondantes au premier choix du circuit.

De la même manière, nous obtenons les équations de dépendances pour le deuxième réseau de Petri :

Equation 4-45

$$\left\{ \begin{array}{l}
 X_1^1 = F_{dep}^{données}(V(Ctrl), V(InMux1); Ctrl = 1; X^{acq}; F_{X_1^1} = InMux1_1) \\
 X_0^1 = F_{dep}^{données}(V(Ctrl), V(InMux1); Ctrl = 1; X^{acq}; F_{X_0^1} = InMux1_0) \\
 X^1 = F_{regroupe}(X_1^1, X_0^1) \\
 Ctrl^{1/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux1); Ctrl = 1; X^{acq}; V(X^1)) \\
 InMux1^{1/acq} = F_{dep}^{acq}(V(Ctrl), V(InMux1); Ctrl = 1; X^{acq}; V(X^1)) \\
 Y_1^1 = F_{dep}^{données}(V(X); Y^{acq}; F_{Y_1^1} = X_1^1) \\
 Y_0^1 = F_{dep}^{données}(V(X); Y^{acq}; F_{Y_0^1} = not X_0^1) \\
 Y^1 = F_{regroupe}(Y_1^1, Y_0^1) \\
 X^{1/acq} = F_{dep}^{acq}(V(X); Y^{acq}; V(Y^1)) \\
 OutMux1^1 = F_{dep}^{données}(V(Y); OutMux^{acq}; F_{OutMux1^1} = Y_1^1) \\
 OutMux0^1 = F_{dep}^{données}(V(Y); OutMux^{acq}; F_{OutMux0^1} = Y_0^1) \\
 OutMux^1 = F_{regroupe}(OutMux1^1, OutMux0^1) \\
 Y^{1/acq} = F_{dep}^{acq}(V(Y); OutMux^{acq}; V(OutMux^1))
 \end{array} \right.$$

Enfin, les équations de dépendances des canaux primaires de chaque réseau sont finalement unies pour générer les équations de dépendances finales du circuit. Dans cet exemple, les équations de dépendances sont :

Equation 4-46

$$\left\{ \begin{array}{l}
 OutMux = F_{union}(OutMux^1; OutMux^0) \\
 Ctrl^{acq} = F_{union}^{acq}(Ctrl^{0/acq}, Ctrl^{1/acq}) \\
 InMux0^{acq} = InMux0^{0/acq} \\
 InMux1^{acq} = InMux1^{1/acq}
 \end{array} \right.$$

En résumé, les systèmes d'équation (Equation 4-42, Equation 4-43, Equation 4-44, Equation 4-45 et Equation 4-46) sont les équations de dépendances du multiplexeur.

4.5.5 Optimisation des équations de dépendances

Normalement, les équations de dépendances générées peuvent posséder des redondances. Ces redondances sont liées à la génération automatique des équations de dépendances. Lors de la génération des équations de dépendances, les tests de validité des canaux garantissent la présence de données sur les entrées. Ces tests sont systématiquement réalisés, sans prendre en compte les relations entre les canaux testés avec les expressions des gardes et des fonctions de calcul des sorties. Ceci a pour objectif d'assurer que les équations de dépendances sont correctes dans tous les cas. En effet, certains tests de validité peuvent être redondants. Ils peuvent être implicitement inclus dans les expressions des gardes ou des fonctions de calcul. Par conséquent, ces tests peuvent être enlevés.

De plus, les redondances peuvent être concernées le protocole de communication adopté des circuits. Dans certains protocoles de communication – par exemple le protocole séquentiel, WCHB – la génération des signaux de sortie pour les données implique la fonction des dépendances nécessaires pour la génération des signaux d'acquiescement des canaux d'entrée. Ainsi, les équations de dépendances des signaux d'acquiescement dans ce cas seront simplifiées : elles dépendent uniquement les test des canaux de sortie. Comme ces redondances sont liées au protocole de communication des circuits, nous les aborderons dans le Chapitre 5.

Nous définissons ici les conditions dans lesquelles certains des tests de validité des canaux d'entrée peuvent être supprimés dans les équations de dépendances générées.

Il est à rappeler que les règles d'optimisation présentées dans ce paragraphe sont seulement appliquées aux circuits asynchrones encodés avec un codage insensible aux délais. En fait, seul le test de validité d'un canal insensible aux délais pose problème car il est coûteux (il faut tester tous les rails de données de tous les digits du canal). Pour des circuits encodés avec un codage différent que le DI, la validité d'un canal se résume au test du signal de requête du canal.

Pour représenter les règles de simplification des équations de dépendances, nous reprenons les équations de dépendances définies §4.5.1 (Equation 4-1 et Equation 4-2).

$$\text{Equation 4-47} \quad S = F_{dep}^{données} \left(\sum_i V(E_i); \sum_j G_j; S^{acq}; F_S \right)$$

$$\text{Equation 4-48} \quad E^{acq} = F_{dep}^{acq} \left(\sum_i V(E_i); \sum_j G_j; S_E^{acq}; V(S) \right)$$

4.5.5.1 Règle 1

Cette règle sert à éliminer des redondances de test de validité des canaux d'entrée dans les équations de dépendances en exploitant les expressions de garde. L'utilisation de données issues d'un canal d'entrée dans l'expression de garde implique déjà la présence de données sur le canal. Dans ce cas, le test de validité de ce canal est superflu. La règle 1 de l'optimisation des équations de dépendances indique que : si un canal d'entrée encodé avec un codage insensible aux délais apparaît dans l'expression de garde, il n'est pas nécessaire de tester la validité de ce canal dans les équations de dépendances concernées. Cette règle est représentée par la condition suivante :

$$\forall V(E) \in \sum_i V(E_i) : E \in \sum_j G_j$$

Cette règle est appliquée aux deux équations de dépendances Equation 4-47 et Equation 4-48.

4.5.5.2 Règle 2

La règle 2 de l'optimisation des équations de dépendances concerne l'élimination des tests de validité des canaux d'entrée en exploitant les expressions de calcul. De la même façon qu'avec la règle 1, si un canal d'entrée encodé avec un codage insensible aux délais apparaît dans l'expression de calcul, le test de validité de ce canal dans les équations de dépendances concernées est redondant et peut donc être enlevé. La règle 2 est illustrée par la condition :

$$\forall V(E) \in \sum_i V(E_i) : E \in F_S$$

4.5.5.3 Exemple

Pour illustrer les règles d'optimisation des équations de dépendances, les équations de dépendances du multiplexeur présentées §4.5.3 sont reprises. En appliquant la règle 1, le test de validité du canal de contrôle *Ctrl* dans l'Equation 4-36 et l'Equation 4-37 est enlevé car ce canal a été vérifié dans l'expression de garde (*Ctrl*=0). Toujours dans l'Equation 4-36, puisque le canal *InMux0* apparaît dans l'expression de calcul du canal de sortie *OutMux*, le test du canal *InMux0* est également éliminé (la règle 2). L'Equation 4-36 et Equation 4-37 deviennent alors :

$$\text{Equation 4-49} \quad \begin{cases} OutMux_0^0 = F_{dep}^{données} (Ctrl = 0; OutMux^{acq}; InMux0_0) \\ OutMux_1^0 = F_{dep}^{données} (Ctrl = 0; OutMux^{acq}; InMux0_1) \end{cases}$$

$$\text{Equation 4-50} \quad \begin{cases} InMux0^{0/acq} = F_{dep}^{acq}(V(InMux0); Ctrl = 0; OutMux^{acq}; V(OutMux^0)) \\ Ctrl^{0/acq} = InMux0^{0/acq} \end{cases}$$

De la même façon, les équations de dépendances qui correspondent au deuxième réseau de Petri du multiplexeur sont optimisées en appliquant les règles 1 et 2. Elles deviennent :

$$\text{Equation 4-51} \quad \begin{cases} OutMux_0^1 = F_{dep}^{données}(Ctrl = 1; OutMux^{acq}; not InMux1_0) \\ OutMux_1^1 = F_{dep}^{données}(Ctrl = 1; OutMux^{acq}; not InMux1_1) \end{cases}$$

$$\text{Equation 4-52} \quad \begin{cases} InMux1^{1/acq} = F_{dep}^{acq}(V(InMux1); Ctrl = 1; OutMux^{acq}; V(OutMux^1)) \\ Ctrl^{1/acq} = InMux1^{1/acq} \end{cases}$$

4.5.6 Conclusion sur les équations de dépendances

La génération des équations de dépendances est une des étapes très importantes de la méthode de synthèse de circuits asynchrones proposée dans ce manuscrit. La génération correcte des équations de dépendances d'un circuit asynchrone assurera une implémentation correcte du circuit car à partir de ces dernières, le réseau de portes du circuit est créé.

L'équation de dépendances est une forme générale décrivant le fonctionnement d'un circuit asynchrone. Elle capture toutes les informations concernées l'implémentation des circuits asynchrones telles que les signaux d'acquittement. Aucune hypothèse de protocole de communication ni de technologie cible n'est introduite lors de cette étape. Ces paramètres sont pris en compte dans l'étape qui suit, l'étape de génération du circuit. Ceci est un avantage important de la méthode de synthèse proposée car à partir de ces équations de dépendances, nous pouvons opter pour une implémentation d'un circuit asynchrone, un protocole de communication, ainsi qu'une technologie cible. De plus, il est possible de choisir une implémentation QDI du circuit ou une implémentation micropipeline, voire d'autres types de circuit. Elles peuvent être toutes générées grâce à ces équations de dépendances.

Un avantage supplémentaire de la génération des équations de dépendances est la possibilité de s'interfacer avec les autres outils d'optimisation/décomposition propres aux circuits asynchrones. A partir des équations de dépendances, les équations logiques peuvent être déduites. Ces équations logiques peuvent être entrées dans d'autres outils de synthèse pour une implémentation spécifique du circuit. Cette propriété des équations de dépendances nous permet d'intégrer un autre outil d'optimisation, de décomposition ou de projection technologique dans l'environnement TAST.

4.6 Conclusion

Une méthode de compilation de circuits asynchrones, de la spécification en langage CHP vers une forme indépendante, vient d'être présentée dans ce chapitre. Cette méthode permet de synthétiser des circuits asynchrones à partir d'une spécification CHP. Le langage CHP est utilisé à titre indicatif. Un autre langage peut être également utilisé. Le réseau de Petri et les graphes de flot de données sont ici utilisés comme une forme de représentation intermédiaire. Plusieurs vérifications ont été introduites sur cette forme de représentation pour garantir que les circuits sont conformes à la spécification DTL, ce qui permet une implémentation des circuits.

La notion d'équation de dépendances a aussi été présentée pour exprimer des circuits asynchrones synthétisables sous une forme indépendante des protocoles de communication et de la

technologie cible. Cette forme ne dépend pas non plus du type de circuit : la méthode proposée peut être appliquée aux autres types de circuits tels que SI, micropipeline, ...

Nous avons également proposé certaines règles de simplification qui permettent d'optimiser les équations de dépendances des circuits encodés avec un codage insensible aux délais, tel que « 1-parmi-N ».

La méthode de synthèse proposée élimine le problème de l'explosion des états en évitant la génération explicite de tous les marquages au niveau signal. Elle peut servir à synthétiser des circuits asynchrones complexes. Dans ce travail de thèse, toutes les étapes présentées de cette méthode ont été automatisées dans l'outil TAST. Les algorithmes et l'implémentation de l'outil TAST seront présentés Chapitre 6.

Chapitre 5

SYNTHESE DE CIRCUITS QDI

Après avoir introduit un premier aperçu des aspects critiques de la conception des circuits asynchrones et mis en place une forme de représentation des circuits asynchrones (les équations de dépendances), qui est non seulement indépendante du protocole de communication et de la technologie cible mais aussi un point de départ pour la synthèse bas niveau (les tâches d'optimisation, de décomposition et de projection technologique). Nous voyons depuis ce départ comment structurer la synthèse afin de la simplifier en étapes élémentaires. Le partitionnement du problème peut être représenté de la manière suivante :

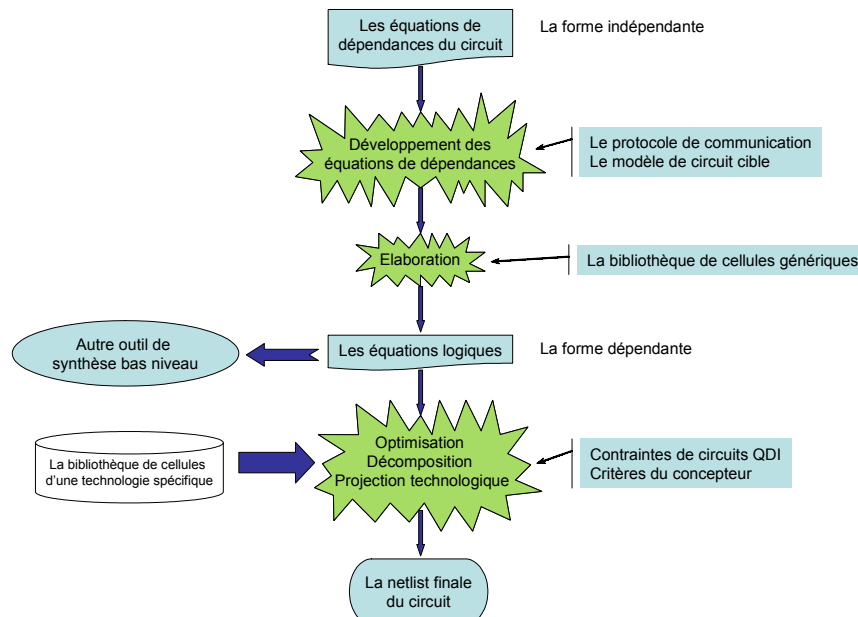


Figure 83 Synthèse bas niveau

Au niveau de la représentation des circuits asynchrones sous forme d'équations de dépendances, il semblait important de conserver les informations structurales concernant le flot de données représenté par le programme original traité. C'est pourquoi les équations de dépendances sont séparément générées pour chaque garde (chaque « chemin » choisi d'après la valeur des données en entrée du processus). Chaque sortie du programme regroupe donc un certain nombre d'équations de dépendances (une par garde) correspondant à la production de la sortie dans un cas différent. Les équations de dépendances pour les canaux de sortie du programme traité sont réécrites comme suit :

$$\text{Equation 5-1} \quad \begin{cases} S^c = F_{dep}^{données} \left(\sum_i V(E_i); \sum_j G_j; S^{acq}; F_{S^c} \right) \\ S = F_{union}^{données} \left(\sum_c S^c \right) \end{cases}$$

De la même façon, les entrées du programme sont acquittées depuis chaque garde. Les équations de dépendances pour les signaux d'acquiescement des canaux d'entrée sont décrites de la façon suivante :

$$\text{Equation 5-2} \quad \begin{cases} E^{c/acq} = F_{dep}^{acq} \left(\sum_i V(E_i); \sum_j G_j; S_E^{c/acq}; V(S_E^c) \right) \\ E^{acq} = F_{union}^{acq} \left(\sum_c E^{c/acq} \right) \end{cases}$$

Ainsi, la structure du programme initial (décrit en langage CHP) se trouve conservée en totalité et on peut construire un circuit dont le flot de données correspond à l'imbrication des gardes du code original. Parmi les arguments des équations de dépendances ci-dessus, on trouve :

$\sum_i V(E_i)$: des signaux de validité pour s'assurer la validité d'un bus ou multi-rails (un canal d'entrée) avant de produire une sortie

$\sum_j G_j$: des expressions de garde dans l'ordre d'imbrication

$S_E^{c/acq}$: le signal d'acquiescement (afin de s'assurer que la sortie est disponible vis-à-vis des processus auxquels elle est connectée)

F_{S^c} : la fonction de calcul à propager en sortie (qui peut être arbitrairement complexe)

Parmi ces signaux, la redondance permettant d'éliminer une partie des arguments est gérée par les règles d'optimisation des équations de dépendances définies précédemment (Chapitre 4). Il ne reste plus par la suite qu'à interpréter les équations de dépendances générées, en introduisant le protocole de communication de type « poignée de main » et la technologie cible afin de les transformer, plus concrètement, en représentation de portes issues d'une bibliothèque spécifique de cellules. Les tâches d'optimisation, de décomposition et de projection technologique peuvent avoir lieu : A ce stade, une première modélisation permet de gérer un réseau de portes proche de la structure finale.

La littérature consacrée aux tâches d'optimisation, de décomposition et de projection technologique ne propose pas une infinité de solutions algorithmiques efficaces, loin de là. Dans les faits, le problème est souvent décomposé en plusieurs phases distinctes :

- L'optimisation logique, réalisée avant la projection technologique proprement dite, a souvent pour objectif de décomposer le circuit – et l'optimiser – en utilisant des portes élémentaires « atomiques » présentes dans la bibliothèque.
- De cette manière, la projection technologique est toujours possible à partir de ces portes et consiste à les regrouper pour utiliser des portes plus grosses de la bibliothèque (optimisation et projection technologique simultanées). A partir d'un tel circuit, le processus de la projection technologique revient à « remonter » le circuit en amont depuis une sortie pour tenter de le « recouvrir » avec des portes de la bibliothèque. Cette technique, bien que très efficace, a néanmoins ses inconvénients comme le fait de ne pouvoir converger que vers des portes à une sortie.

- Enfin, il est toujours possible d'exécuter une série de règles de transformations sur le circuit obtenu afin de réaliser une dernière optimisation logique. Celle-ci est cependant très délicate et dépend en majeure partie de la liste de règles permises à ce stade (il ne s'agit pas de détruire le travail déjà effectué auparavant : les règles doivent être très rigoureusement définies)

Cette décomposition n'est pas universelle, mais elle est utilisée dans un grand nombre de logiciels de circuit synchrone. Mais pour l'appliquer au domaine de l'asynchrone, il faut que les contraintes de circuits QDI soient respectées et que l'absence d'aléas soit vérifiée et assurée. Deux solutions s'ouvrent donc :

- soit on opte pour un ensemble de calculs sur des circuits non QDI en sachant rendre le résultat final QDI avec une bonne efficacité (Figure 84). Basée sur un ensemble de transformations élémentaires, cette méthode se rapproche de la technique synchrone à ceci près que l'ensemble des règles de transformation [BURN96] n'est pas, de loin, le même.
- soit on effectue l'ensemble des calculs sur des circuits QDI par construction, en étant assuré que le résultat final sera bien QDI (Figure 85). Cette méthode semble être la plus prometteuse (et la plus accessible). Cependant, elle nécessite la définition d'une série de règles précises afin de ne pas « sortir » du modèle QDI pendant l'optimisation et la projection technologique.

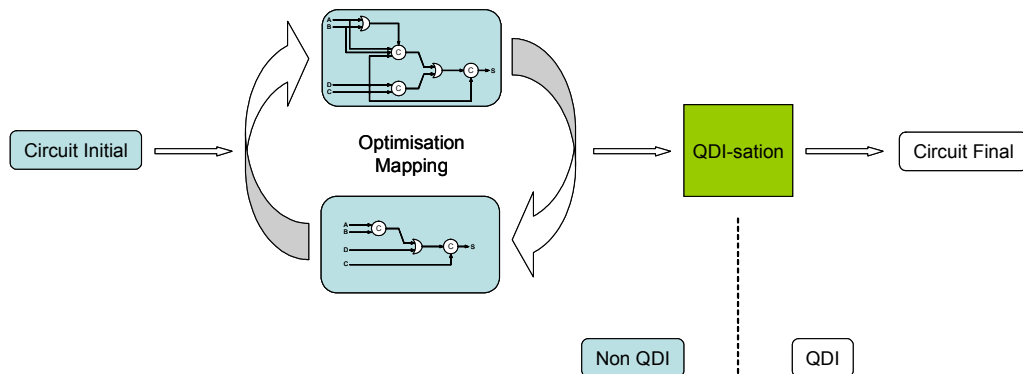


Figure 84 Première solution d'optimisation et de projection technologique

Comparativement aux méthodes synchrones, l'optimisation logique et la projection technologique asynchrone demandent une rigueur supplémentaire dont le prix à payer en termes de calculs n'est pas négligeable. Des deux méthodes exposées ci-dessus, la seconde semble clairement la meilleure car la méthode de transformation d'un circuit non QDI optimal en un circuit QDI optimal n'existe pas. Nous ne considérons donc que la deuxième méthode dans la suite. Armé de ce choix, on est confronté à une liste d'autres choix tout aussi complexes que nous aborderons par la suite.

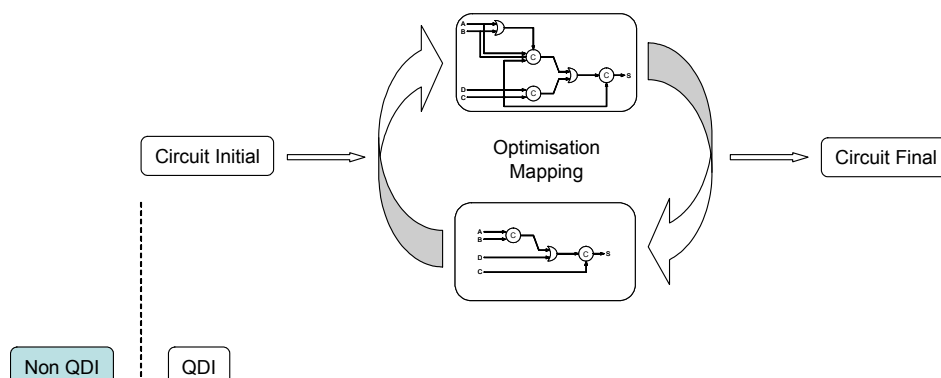


Figure 85 Deuxième solution d'optimisation et de projection technologique

Dans ce chapitre, un réseau de portes (la *netlist*) de circuit asynchrone est initialement généré en introduisant un protocole de communication de type « poignée de main » quatre phases pour

communiquer avec les autres circuits auxquels il est connecté. Ce réseau de portes, dans un premier temps, est mappé sur une bibliothèque de cellules contenant des portes génériques. Ces portes sont décrites seulement au niveau comportemental, mais pas au niveau physique. Les tâches d'optimisation, de décomposition et de projection technologique sont ensuite abordées tout en préservant les caractéristiques des circuits asynchrones QDI : fourche isochrone, absence d'aléa, *etc.*

5.1 Génération des circuits QDI en utilisant des cellules génériques

L'un des intérêts de la méthode considérée dans ce travail de thèse réside dans le fait que la transformation non-QDI $\rightarrow$ QDI est effectuée en amont de la chaîne d'optimisation. Ainsi, elle n'a pas à prendre en compte des critères d'optimisation – qui seront traités par la suite – de telle sorte que l'on peut se permettre de compléter le circuit à volonté (avec des termes potentiellement redondants) afin d'aboutir à une version correcte (*i.e.* QDI). De nombreuses méthodes existent qui se réfèrent toutes à des modèles généraux simplistes (tels que des modélisations en *minterms* [MILL65][DAVI92], c'est-à-dire avec de la logique à deux niveaux : un niveau de portes de Müller et un niveau de portes « OR »).

Les portes génériques utilisées dans ce paragraphe sont les portes élémentaires – comme les portes « AND », les portes « OR », les portes d'inverseur, *etc.* – plus les portes de Muller (symétrique et dissymétrique, sans ou avec le signal de reset) [RIGA02]. Ces portes sont fonctionnellement définies de façon générique de telle sorte qu'il n'y a aucune contrainte sur leur *fan-in*. En effet, elles sont volontairement générées avec un *fan-in* nécessaire. C'est par la suite que les portes de *fan-in* élevé sont décomposées.

Dans ce paragraphe, les schémas de circuits asynchrones, utilisant les protocoles de communication abordés au Chapitre 3 (séquentiel, WCHB, PCHB, PCFB), seront présentés. Les modèles de circuit cible proposés dans le Chapitre 3 sont utilisés afin de structurer le circuit final. Il s'agit de mettre la partie de contrôle (*buffer*), qui a charge de réaliser la communication de type « poignée de main » avec les autres circuits, avant ou après la partie de calcul. Pour simplifier la présentation du problème, seule la décomposition dans laquelle le *buffer* est mis après la partie de calcul sera présentée. L'autre décomposition peut être effectuée de manière identique.

5.1.1 Développement des équations de dépendances en introduisant le protocole de communication

Dans ce paragraphe, le développement des équations de dépendances est présenté avec deux protocoles de communication : le protocole séquentiel et le protocole WCHB. Les autres protocoles sont déductibles.

5.1.1.1 Protocole séquentiel

Avec le protocole séquentiel, les sorties de données et les signaux d'acquiescement sont réalisés séparément. En particulier, les sorties de données d'un canal de sortie sont uniquement calculées en fonction des données entrantes, tandis que les signaux d'acquiescement sont directs. Le test de la disponibilité des canaux de sortie et le test de validité des sorties ne sont plus nécessaires. Les équations de dépendances deviennent donc :

$$\text{Equation 5-3} \quad \begin{cases} S^c = F_{SEQ}^{données}(\sum_i V(E_i); \sum_j G_j; F_{S^c}) \\ S = F_{union}^{données}(\sum_c S^c) \end{cases}$$

Equation 5-4

$$\begin{cases} E^{c/acq} = F_{SEQ}^{acq}(\sum_i V(E_i); \sum_j G_j; S_E^{acq}) \\ E^{acq} = F_{union}^{acq}(\sum_c E^{c/acq}) \end{cases}$$

L'implémentation du circuit avec le protocole séquentiel est présentée Figure 86. Dans cette implémentation, $F_{SEQ}^{données}$ et F_{SEQ}^{acq} sont respectivement les fonctions de dépendances de données et d'acquiescement correspondant au protocole séquentiel. Elles sont considérées comme un rendez-vous entre leurs paramètres de façon symétrique sur la phase montante et sur la phase descendante. En particulier, les inverseurs sur le test de validité des entrées et sur la condition de garde dans le bloc « Direct », qui implémente la fonction F_{SEQ}^{acq} , viennent du fait que les signaux d'acquiescement sont actifs au niveau bas.

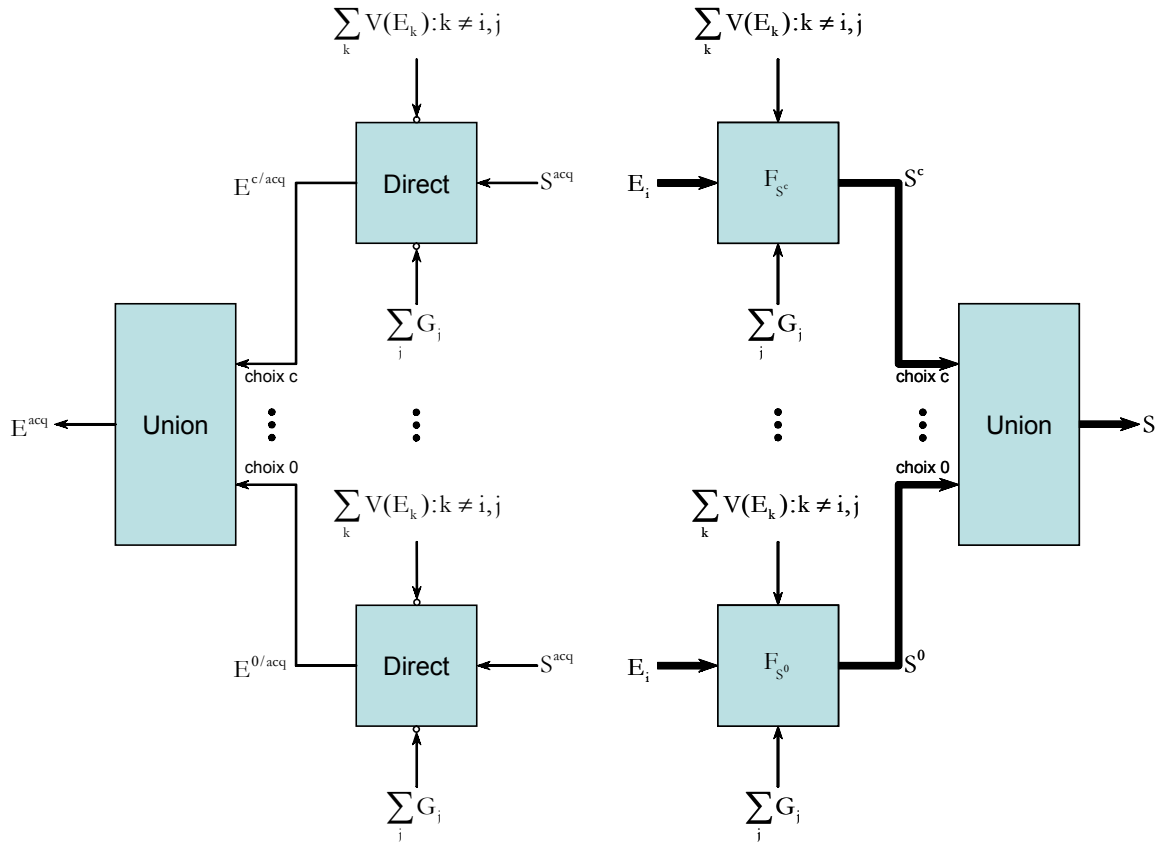


Figure 86 Implémentation de circuits avec le protocole séquentiel

A titre d'exemple, les équations de dépendances du multiplexeur §4.5.3 sont réécrites en introduisant le protocole séquentiel :

Equation 5-5

$$\begin{cases} OutMux_0^0 = F_{SEQ}^{données}(Ctrl = 0; InMux_0^0) \\ OutMux_1^0 = F_{SEQ}^{données}(Ctrl = 0; InMux_0^1) \\ OutMux_0^1 = F_{SEQ}^{données}(Ctrl = 1; not InMux_1^0) \\ OutMux_1^1 = F_{SEQ}^{données}(Ctrl = 1; not InMux_1^1) \\ OutMux_0 = F_{union}^{données}(OutMux_0^0, OutMux_0^1) \\ OutMux_1 = F_{union}^{données}(OutMux_1^0, OutMux_1^1) \end{cases}$$

$$\text{Equation 5-6} \quad \left\{ \begin{array}{l} InMux0^{acq} = InMux0^{0/acq} = F_{SEQ}^{acq}(Ctrl = 0; OutMux^{acq}) \\ InMux1^{acq} = InMux1^{1/acq} = F_{SEQ}^{acq}(Ctrl = 1; OutMux^{acq}) \\ Ctrl^{0/acq} = InMux0^{acq} \\ Ctrl^{1/acq} = InMux1^{acq} \\ Ctrl^{acq} = F_{union}^{acq}(Ctrl^{0/acq}, Ctrl^{1/acq}) \end{array} \right.$$

Nous illustrons simplement dans la Figure 87 le schéma du circuit obtenu avec le protocole séquentiel. La génération de la *netlist* de portes des circuits QDI est présentée en détail dans le paragraphe qui suit.

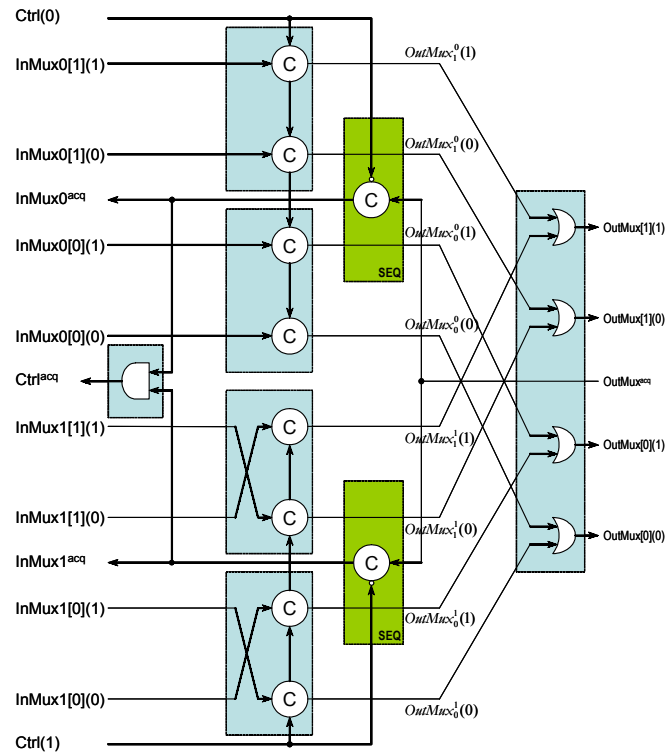


Figure 87 Implémentation du multiplexeur en protocole séquentiel (optimisée)

5.1.1.2 Protocole WCHB

Dans les circuits implémentés avec le protocole WCHB, les données issues des canaux d'entrée sont propagées vers la sortie si et seulement si la sortie est prête (ce qui est indiqué par son signal d'acquiescement). Les canaux d'entrée consommés sont acquittés une fois que les données sont envoyées. Les canaux de sortie sont remis à zéro quand les entrées sont rétablies à zéro. Ce protocole est alors symétrique sur la phase montante et sur la phase descendante.

La fonction de dépendances F_{dep} (pour les données ainsi que pour les acquittements) est interprétée comme un rendez-vous des données. Elle attend que la sortie soit prête à recevoir des données (son signal d'acquiescement passe au niveau actif) pour que les données, calculées par la fonction F_{sc} et conditionnées par les tests de validité et les conditions de garde, soient propagées vers la sortie. Cette action est également répétée dans la phase de remise à zéro.

Les équations de dépendances dans ce cas sont définies comme illustrée ci-après.

Equation 5-7
$$\begin{cases} S^c = F_{WCHB}^{données}(\sum_i V(E_i); \sum_j G_j; F_{S^c}; S^{acq}) \\ S = F_{union}^{données}(\sum_c S^c) \end{cases}$$

Equation 5-8
$$\begin{cases} E^{c/acq} = F_{WCHB}^{acq}(\sum_i V(E_i); \sum_j G_j; S_E^{acq}; V(S_E^c)) \\ E^{acq} = F_{union}^{acq}(\sum_c E^{c/acq}) \end{cases}$$

Une optimisation peut être réalisée à ce stade. Comme nous l'avons énoncé dans §3.3.1.2, la présence de données sur les sorties implique que les entrées possèdent des données valides et qu'elles sont consommées. Ceci est aussi vrai pour la phase de remise à zéro du protocole. De ce fait, les canaux d'entrée peuvent être acquittés grâce à uniquement un test de validité des canaux de sortie. En particulier, les signaux d'acquiescement peuvent être calculés en fonction des signaux de données qui viennent d'être générés par l'Equation 5-7. L'Equation 5-8 devient alors :

Equation 5-9
$$\begin{cases} E^{c/acq} = F_{valide}^{acq}(V(S_E^c)) \\ E^{acq} = F_{union}^{acq}(\sum_c E^{c/acq}) \end{cases}$$

L'implémentation de circuits avec le protocole WCHB est représentée Figure 88. Cette implémentation est réalisée en mettant en pipeline un bloc de calcul, implémenté avec le protocole séquentiel, et un *buffer* en protocole WCHB.

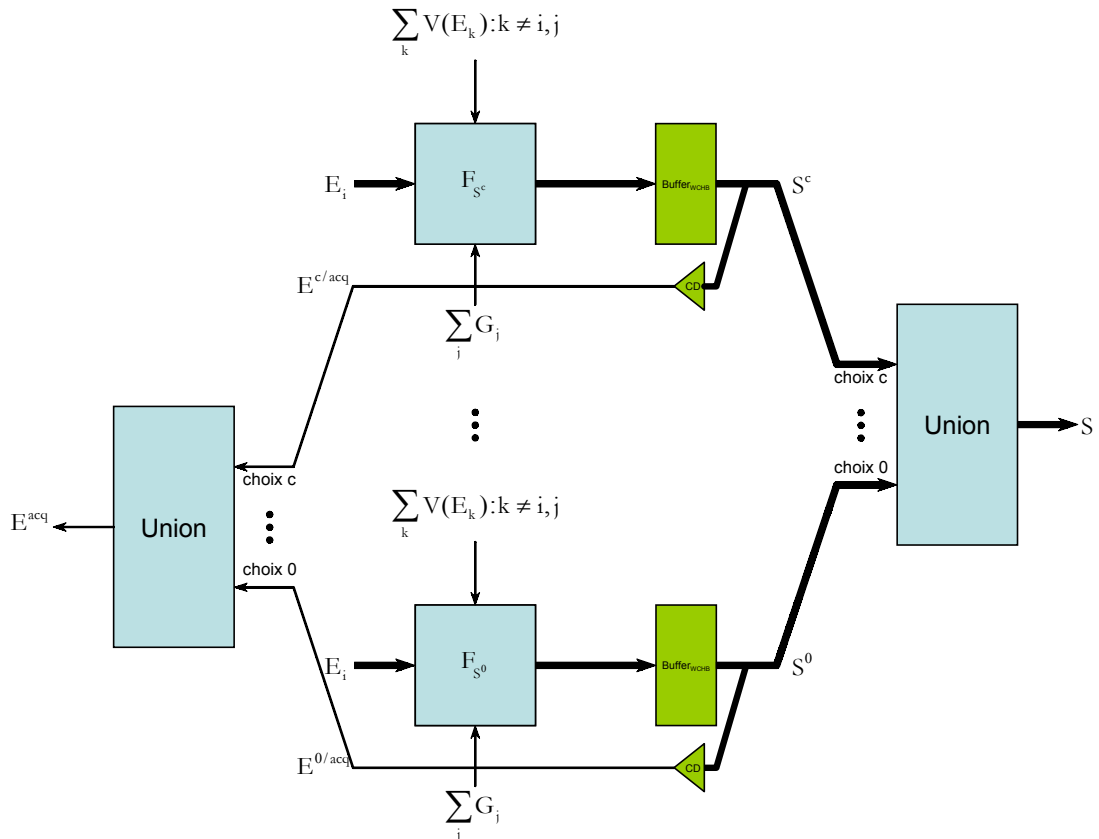


Figure 88 Implémentation de circuits en protocole WCHB

A titre d'exemple, les équations de dépendances du multiplexeur (cf. §4.5.3) avec le protocole WCHB sont les suivantes.

Equation 5-10

$$\begin{cases}
 OutMux_0^0 = F_{WCHB}^{données}(Ctrl = 0; InMux_0^0; OutMux^{acq}) \\
 OutMux_1^0 = F_{WCHB}^{données}(Ctrl = 0; InMux_0^1; OutMux^{acq}) \\
 OutMux_0^1 = F_{WCHB}^{données}(Ctrl = 1; not\ InMux_1^0; OutMux^{acq}) \\
 OutMux_1^1 = F_{WCHB}^{données}(Ctrl = 1; not\ InMux_1^1; OutMux^{acq}) \\
 OutMux_0 = F_{union}^{données}(OutMux_0^0, OutMux_0^1) \\
 OutMux_1 = F_{union}^{données}(OutMux_1^0, OutMux_1^1)
 \end{cases}$$

Equation 5-11

$$\begin{cases}
 InMux_0^{acq} = InMux_0^{0/acq} = F_{valide}^{acq}(V(OutMux_0^0); V(OutMux_1^0)) \\
 InMux_1^{acq} = InMux_1^{1/acq} = F_{valide}^{acq}(V(OutMux_0^1); V(OutMux_1^1)) \\
 Ctrl^{0/acq} = InMux_0^{acq} \\
 Ctrl^{1/acq} = InMux_1^{acq} \\
 Ctrl^{acq} = F_{union}^{acq}(Ctrl^{0/acq}, Ctrl^{1/acq})
 \end{cases}$$

Nous illustrons simplement dans la Figure 89 le schéma du circuit obtenu avec le protocole de communication WCHB. La génération de la *netlist* de portes des circuits QDI est présentée en détail dans le paragraphe qui suit.

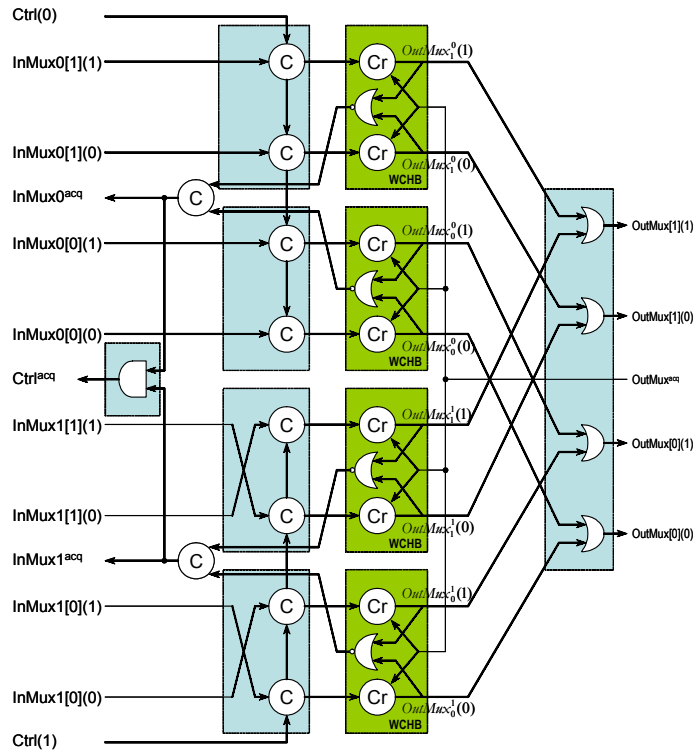


Figure 89 Implémentation du multiplexeur en protocole WCHB

5.1.2 Génération de la netlist de portes

Dans la logique insensible aux délais (DI), les canaux, et donc les variables qui portent les valeurs de canaux, sont encodés en codage « 1-parmi-N ». En langage CHP, ces variables sont déclarées comme le type de données MR[B][L], cela signifie qu'elles sont composées de L digits de base B . Pour les opérateurs binaires effectués sur les opérandes de type MR, deux algorithmes de calcul existent.

- **L'algorithme à 2-niveaux** : cet algorithme calcule tous les *minterms* possibles des digits des deux opérandes et effectue le calcul. A titre d'exemple, l'algorithme est illustré Figure 90. L'avantage de cet algorithme est la régularité et la vitesse du circuit généré. Ce circuit, regroupant des *minterms* de longueur $2*L$ de tous les digits des deux opérandes, peut être réalisé en structure PLA comme montrée dans [SEIT'80]. Cependant, cet algorithme est inefficace et coûteux en surface car il demande, pour l'opérateur d'égalité par exemple, B^L *minterms* de longueur $2*L$ chacun pour comparer les opérandes de L digits de base B . D'ailleurs, avec un nombre de digits L suffisamment grand, ces *minterms* doivent être décomposés en plusieurs petits *minterms*, ce qui est un problème assez difficile pour les circuits QDI.

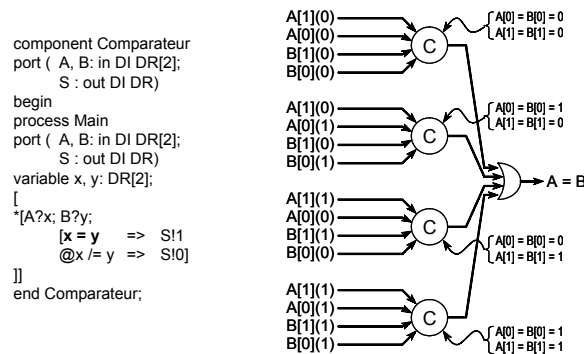


Figure 90 Algorithme à 2-niveaux pour l'opérateur d'égalité

- **L'algorithme multi-niveaux** : l'algorithme consiste à faire l'opération sur les digits des opérandes séparément l'un de l'autre, puis de regrouper ces résultats. Un exemple de cet algorithme est représenté Figure 93. L'avantage de cet algorithme est la préservation des structures du circuit. De plus, cet algorithme est beaucoup plus efficace en surface comparé à celui à 2-niveaux. Il demande, toujours pour l'opérateur d'égalité, seulement $B*L$ *minterms* de longueur 2 pour comparer les digits de deux opérandes et un *minterm* de longueur L pour regrouper les comparaisons de digits. Un autre avantage de cet algorithme est qu'il pose moins de problème pour décomposer les *minterms*. Seul le *minterm* de regroupement a besoin d'être décomposé. C'est pourquoi cet algorithme est choisi pour présenter les opérateurs dans la suite. Ainsi, il a été intégré dans l'outil de synthèse (TAST) qui prototype ce travail de thèse.

5.1.2.1 Test de validité

Le fonctionnement des circuits asynchrones est « piloté » par l'activité sur les canaux d'entrée. Cela signifie que le circuit fonctionne quand les données apparaissent sur les entrées et qu'il est mis en veille quand aucun des canaux ne possède de données. Le test de validité d'un canal est alors nécessaire pour observer la présence de données sur les canaux d'entrée afin de déclencher l'activité du circuit. En outre, il sert également à vérifier la validité de données d'un canal, ce qui est utilisé non seulement pour résoudre le problème de blocage dans les circuits de convergence (cf. §3.1.2.3.2) mais encore pour acquitter les canaux d'entrée avec certains protocoles de communication (voir les protocoles WCHB, PCHB, PCFB).

Avec les canaux codés en « 1-parmi- N », une valeur de données valide est représentée par N fils tels qu'à tout moment, un et seulement un fil est actif. Par contre, l'absence de données est indiquée par les zéros sur tous les fils. De ce fait, le test de validité d'un canal est simplement réalisé avec une porte « OR » sur les N fils du canal.

Dans le cas général où un canal se compose de digits, la validité du canal est la validité de tous ses digits. Une porte rendez-vous est donc nécessaire comme le montre la Figure 91.

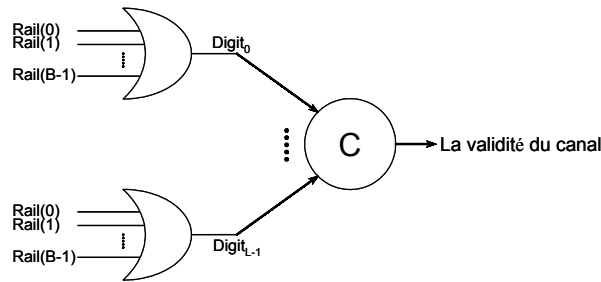


Figure 91 Test de validité d'un canal de communication

5.1.2.2 Initialisation de canaux

L'initialisation d'un canal de sortie consiste à envoyer une valeur sur ce canal au début avant que le processus ne commence son exécution bouclée infinie. Elle est très utile pour initialiser des flots de données bouclées contenant des processus concurrents communicants. L'initialisation d'un canal implique l'écriture séquentielle sur le canal, ce qui nécessite un état supplémentaire. Ainsi, un circuit d'initialisation est ajouté au canal pour le mettre dans l'état souhaité. Ce circuit d'initialisation est un cas dégénéré de *buffer* : il dépend du protocole de communication choisi. La Figure 92 présente un circuit d'initialisation pour les circuits utilisant le protocole WCHB. La porte de Muller avec le signal « *set* » (notée « *Cs* ») est utilisé pour mettre le rail de sortie $S(0)$ au niveau « 1 » au départ, ce qui initialise le canal S avec « 0 ».

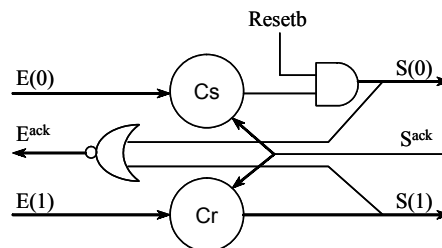


Figure 92 Circuit d'initialisation pour des buffers WCHB

Les circuits d'initialisation pour les autres protocoles (PCHB, PCFB) font partie de la thèse d'un autre doctorant.

5.1.2.3 Implémentation QDI des gardes

5.1.2.3.1 Opérateur d'égalité « = »

L'opérateur égalité est nécessairement utilisé avec deux opérandes de même type. Dans le cas échéant, un des opérandes doit être converti afin être compatible. Cette conversion est automatiquement effectuée par l'outil de synthèse dans le cas où l'un des opérandes est une constante, de sorte que le débordement est tronqué et que le « manque » est rempli par des zéros. Par contre, pour convertir le type de données d'un opérande, l'opérateur de conversion (« *casting* ») est employé. Cet opérateur est présenté dans la suite.

Pour que les deux opérandes soient fonctionnellement égaux, les digits des deux opérandes doivent être égaux respectivement. L'algorithme de comparaison utilisé est l'algorithme multi-niveaux. En la logique « condition faible », la présence ou l'absence de la donnée sur un canal de sortie signifie que les données des canaux d'entrée sont respectivement valides ou invalides. De ce fait, une porte « rendez-vous » est forcément nécessaire pour regrouper les tests des différents digits.

- **Variable vs. constante**

Pour comparer une variable, codé en « 1-parmi-N », avec une constante, le fil de donnée correspondant à la constante est utilisé. La constante est toujours convertie dans le type de données de la variable. Cette comparaison est souvent employée dans des multiplexeurs ou des démultiplexeurs pour choisir les bons canaux. En particulier, selon la valeur d'un canal de contrôle, une action de communication sera prise en compte. Par exemple dans la première garde du multiplexeur du §4.5.3, pour comparer la valeur issue du canal *Ctrl* avec « 0 », le fil de donnée *Ctrl*(0) est utilisé et pour comparer cette valeur avec « 1 », le fil de donnée *Ctrl*(1) est utilisé.

- **Deux variables différentes**

Le circuit de comparaison de deux variables différentes est représenté Figure 93. Le nombre de portes de Muller utilisées est $B*L+1$, avec B et L respectivement la base et la longueur des deux variables.

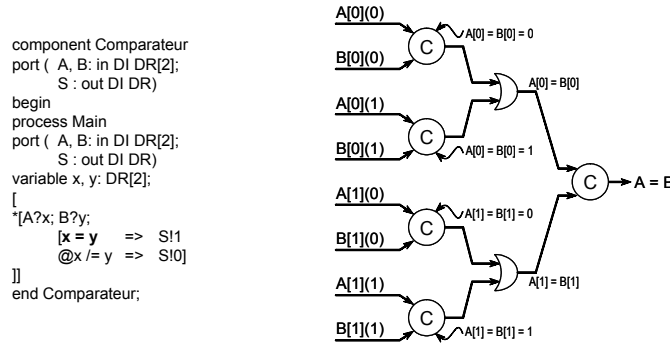


Figure 93 Implémentation QDI pour le test d'égalité

5.1.2.3.2 Opérateur d'inégalité « /= »

De la même façon que l'opérateur d'égalité, l'opérateur d'inégalité est employé sur deux opérandes de même type. Si ce n'est pas le cas, il faut convertir un des opérandes pour qu'il soit compatible.

Contrairement à l'opérateur d'égalité, deux opérandes sont différents si un de ses digits est différent. Par conséquent, l'algorithme de comparaison consiste à tester chaque paire de digits des deux opérandes, puis de regrouper exclusivement ces résultats. Cependant, il existe deux façons d'effectuer le circuit. La première façon est illustrée Figure 94. Le nombre de portes de Muller

utilisées est $\binom{B}{2} * L = \frac{B!}{(B-2)!} * L$.

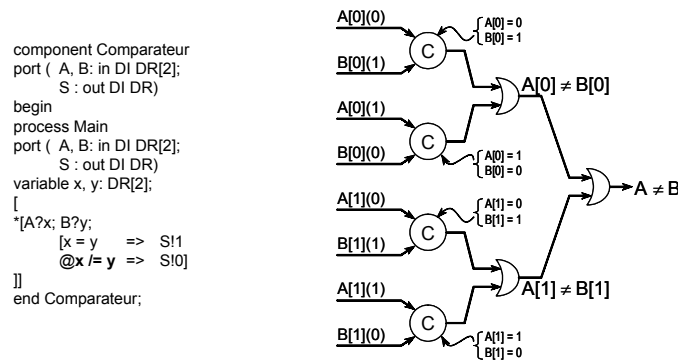


Figure 94 Implémentation non-QDI du test d'inégalité

Cette implémentation ne respecte pas la propriété QDI. Imaginons les gardes du processus *Comparateur* sont implémentées par les circuits dans Figure 93 et Figure 94. Si, par exemple, $A[0]=B[0]$ et $A[1] \neq B[1]$, ce qui fait $A \neq B$, alors la deuxième garde est sélectionnée. Toutefois, il

n'empêche pas de faire commuter les portes dans la branche $A[0]=B[0]$ du circuit de la Figure 93. Ces portes commutées ne seront pas acquittées par la suite : la propriété QDI est violée.

Nous optons donc pour la méthode d'implémentation suivante qui garantit que le circuit généré est QDI.

- **Variable vs. constante**

Pour tester une variable, codée en « 1-parmi-N », avec une constante, tous les fils de données différents de ceux correspondant à la constante sont testés. A titre d'exemple, la Figure 95 présente le test d'inégalité entre une variable et une constante.

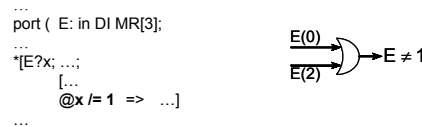


Figure 95 Test d'inégalité entre une variable et une constante

Ce test est répété pour chaque digit de la variable.

- **Deux variables différentes**

La Figure 96 illustre l'algorithme de comparaison entre deux variables de l'exemple donné Figure 93. Cette implémentation est QDI car toutes les combinaisons de digits (*minterms*) sont testées et une seule d'entre elles commute.

L'inconvénient de la version QDI est qu'elle est coûteuse en surface. Le nombre de portes de Muller (à 2 entrées) utilisées pour le premier étage est $L*B^L$ tandis que celui (de L entrées) du deuxième étage est $(2^L - 1)$. Une première vue en terme d'optimisation en surface du circuit est l'utilisation du moins de digits possibles (L petit). L'optimisation globale se fait plus tard.

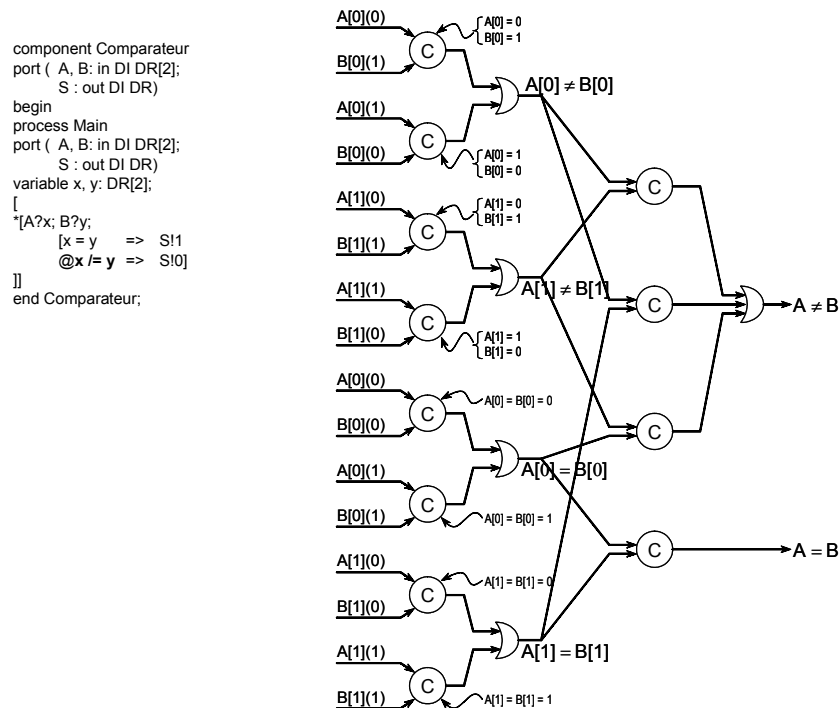


Figure 96 Implémentation QDI pour le test d'inégalité

5.1.2.3.3 Opérateur de sonde (« probe »)

L'opérateur de sonde sert à tester l'activité d'un canal sans consommer la valeur présente dans ce même canal. Il est donc implémenté comme le circuit pour le test de validité. La différence

dans ce cas est qu'on ne termine pas la communication sur le canal. La Figure 97 présente un exemple utilisant l'opérateur de sonde.

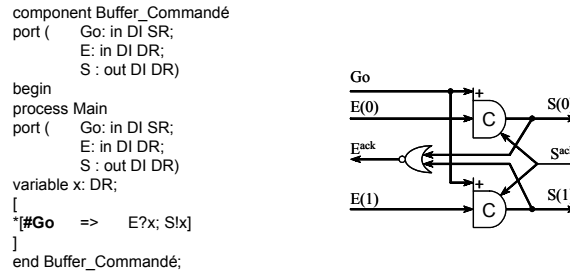


Figure 97 Exemple avec l'opérateur de sonde

Comme nous l'avons montré dans §2.1, l'opérateur de sonde peut être utilisé pour comparer la valeur présente dans un canal avec une autre valeur prédéfinie. Cette comparaison est réalisée comme l'opérateur d'égalité présentée au-dessus.

5.1.2.3.4 Opérateur booléen : « NOT », « OR », « AND »

En logique insensible aux délais, l'opérateur booléen « NOT » n'est pas simplement effectué par un inverseur. La Figure 98 représente un exemple de l'opérateur « NOT ». L'opérateur « NOT » dans cet exemple est équivalent avec l'opérateur d'inégalité (cf. Figure 95).

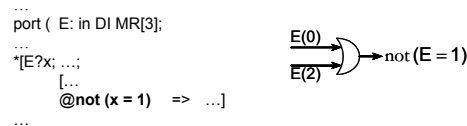


Figure 98 Implémentation de l'opérateur booléen « NOT »

Pour garantir la propriété QDI, l'opérateur booléen « AND » est réalisé par une porte de rendez-vous. Cette porte de Muller assure la correction du circuit non seulement sur la phase montante mais aussi sur la phase descendante. La Figure 99 représente un exemple de l'implémentation d'opérateur booléen « AND » (garde G1).

L'opérateur booléen « OR » inclusif requiert que ses deux opérandes soient valides pour assurer l'insensibilité aux délais. Seules les valeurs de ses opérandes sont utilisées pour l'opération. De ce fait, le test de validité d'autre opérande est nécessaire pour chaque opérande si les deux opérandes ne proviennent pas de la même source. A titre d'exemple, prenons l'exemple du programme de la Figure 99. Les variables x et y ne proviennent pas de la même canal. Afin de garantir une implémentation d'insensible aux délais de la garde G2, le test sur le canal x ($not\ x=0$) doit être associé avec le test de validité du canal y et pareillement, le test sur le canal y ($not\ y=0$) doit être associé avec le test de validité du canal x . Le circuit QDI de la garde G2 est illustré dans la Figure 99. Il est noter que seule la porte « OR » à la sortie (en gris) est inclusive. Les autres portes « OR » du circuit sont exclusives : l'exclusivité de ces portes sont assurée par l'exclusivité des rails des canaux (codage insensible aux délais). Bien entendu, ce circuit va être optimisé par la suite (cf. §5.3.2)

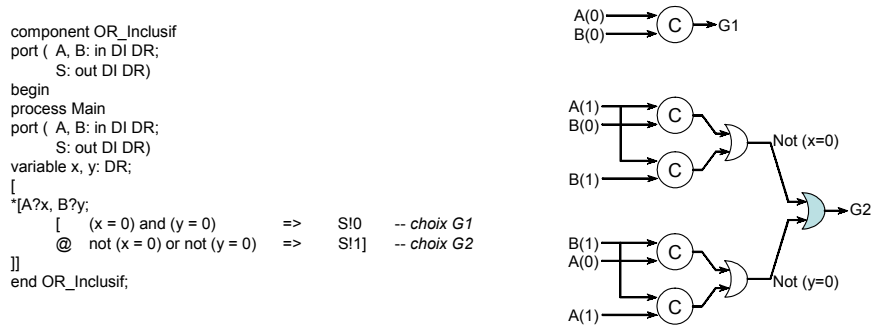


Figure 99 Exemple d'opérateurs booléens dans des gardes

5.1.2.3.5 Implémentation des gardes imbriquées

Nous venons de présenter les implémentations QDI des fonctions de garde. Dans la mesure où les fonctions de gardes peuvent être arbitrairement complexes et représentées sous des formes d'imbrication, les signaux de calcul de chaque garde sont responsables de la représentation non QDI, comme on peut le voir dans le cas général (cf. Figure 100) :

[Garde n-1 => [Garde n => [Garde n+1 ...

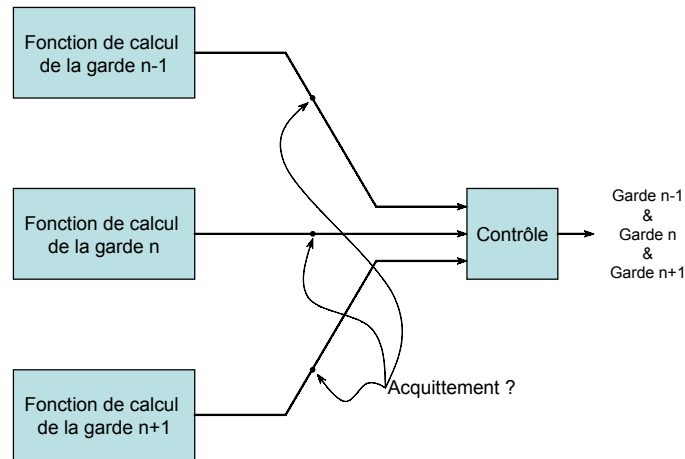


Figure 100 Problème d'acquittement des signaux intermédiaires

Cependant, on peut profiter de la notion d'exclusivité des gardes pour imposer les contraintes QDI. Pour un code CHP de la forme générale :

```

*[G1  =>  [G1_1 =>  [G1_1_1  => ...
                    @G1_1_2  => ...
                    ]
  @G1_2=> ...
]
@G2  => ...
]
    
```

dans lequel les gardes sont imbriquées et exclusives, un chemin parmi les gardes serait, par exemple, {G1, G1_1, G1_1_1}. Le problème dans ce cas est de générer un circuit qui n'active pas de signal intermédiaire dans la garde G2 – non acquitté par la suite – sachant que ce chemin n'est pas utilisé. Il s'agit de faire en sorte que tout signal intermédiaire activé corresponde à une garde réellement empruntée et l'imbrication des fonctions de calcul des gardes répond à cet impératif : tant que la garde G1 n'a pas été activée, on ne permet pas aux signaux intermédiaires de calcul de la garde G1_1 de s'activer (car le chemin final pourrait passer par la garde G2). Par conséquent, on protège la fonction de calcul de la garde G1_1 par le signal d'activation de la garde G1. Cette technique est illustrée par la Figure 101.

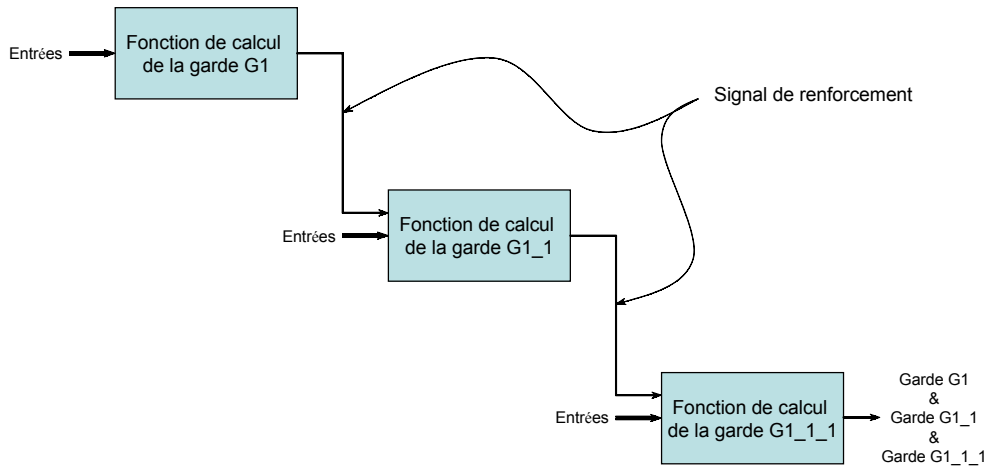


Figure 101 Renforcement des gardes imbriquées

5.1.2.4 Implémentation QDI des fonctions de calcul

5.1.2.4.1 Opérateur de conversion (« casting »)

L'opérateur de conversion est associé à l'action d'affectation. Quand une variable est affectée par une expression dont le type de données est différent de celui de la variable, l'opérateur de conversion est appelé pour convertir le type de données de l'expression vers le type de la variable afin que les deux types soient compatibles. Cet opérateur est réalisé en logique QDI, c'est-à-dire, toutes les combinaisons des fils d'entrée sont considérées pour fabriquer les fils de sortie. En effet, il n'y a aucune porte qui commute sans être acquittée. Un exemple de conversion de type de données est représenté Figure 102.

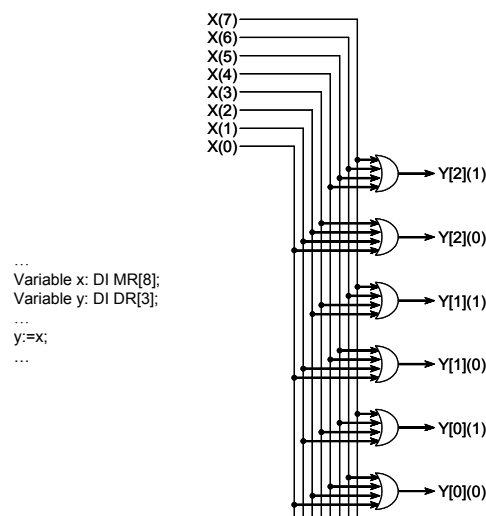


Figure 102 Implémentation QDI d'un exemple de conversion de type

5.1.2.4.2 Opérateur d'inversion (« NOT »)

L'opérateur d'inversion d'un digit de base B est le calcul du complément de la base $(B-1)$ de ce digit (cf. §2.1). Avec l'encodage « 1-parmi- N », l'opération d'inversion n'a besoin d'aucune porte. Elle est effectuée par l'inversion de façon symétrique des fils de données comme cela est montré sur l'exemple de la Figure 103.

L'opérateur « NOT » sur une variable de plusieurs digits est réalisé en effectuant l'inversion des fils sur chaque digit le constituant.

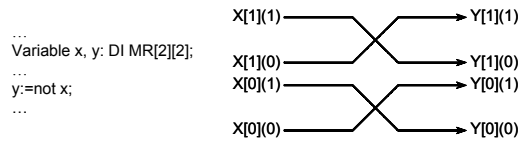


Figure 103 Implémentation QDI de l'opérateur « NOT »

5.1.2.4.3 Opérateur de calcul (« AND », « OR », « XOR », « NAND », « NOR », « XNOR »)

Les opérateurs de calcul logique (« AND », « OR », « XOR ») sont réalisés en logique insensible aux délais pour garantir la propriété QDI du circuit généré. En particulier, tous les *minterms* sont utilisés pour réaliser le circuit. Les Figure 104, Figure 105 et Figure 106 représentent les opérateurs de calcul logique « AND », « OR » et « XOR » respectivement.

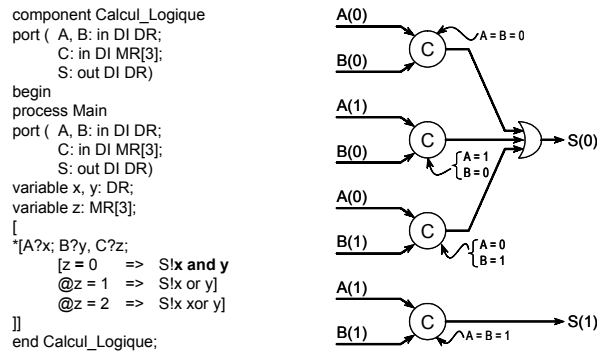


Figure 104 Opérateur de calcul « AND »

Les opérateurs de calcul logique (« NAND », « NOR », « XNOR ») sont effectués en combinant un opérateur de calcul logique correspondant et un opérateur d'inversion. L'opérateur « NAND », par exemple, est une combinaison d'un « AND » et d'un « NOT ». Elle est alors réalisée en permutant les rails de sortie de l'opérateur de calcul « AND ».

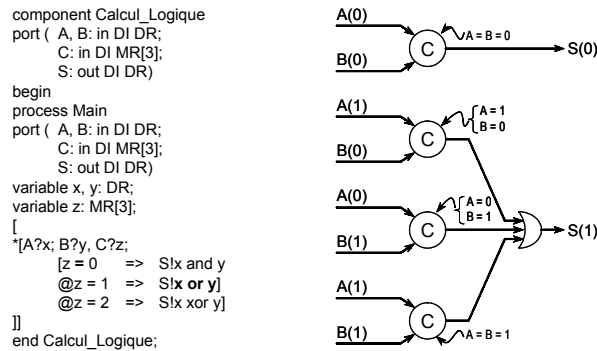


Figure 105 Opérateur de calcul « OR »

L'opération de calcul est effectuée sur chaque digit des opérandes. Il est à noter que ces opérateurs de calcul logique s'appliquent non seulement aux opérandes de codage double-rails mais aussi aux opérandes de codage multi-rails. L'opération sur des digits de multi-rails est basée sur l'opération des digits double-rails. Elle est définie de façon binaire. Cela signifie que l'opérateur est réalisé en effectuant l'opération binaire sur les digits. A titre d'exemple, l'opération « AND » sur les constantes "3"[10] et "5"[10] est faite en effectuant l'opération sur leurs formes binaires "0.1.1"[2] et "1.0.1"[2], ce qui donne "0.0.1"[2] ou "1"[10] en résultat.

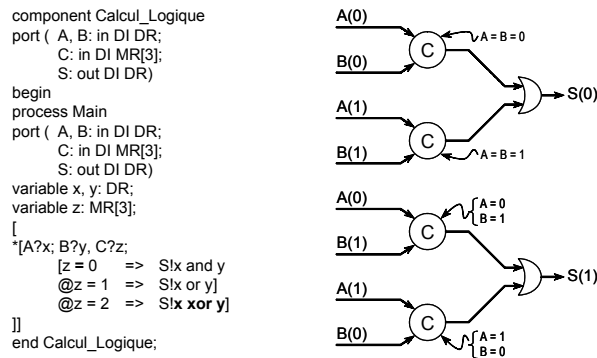


Figure 106 Opérateur de calcul « XOR »

Le nombre total de *minterms* nécessaires pour les opérandes de type $MR[B][L]$ est $L \cdot 2^B$ dont la longueur est 2.

5.1.2.4.4 Opérateurs arithmétiques

Les opérateurs arithmétiques sont implémentés en appelant un générateur d'opérateurs arithmétiques qui fait partie du travail d'un autre doctorant ([FRAG03]). Intégré dans l'outil TAST, ce générateur peut générer des additionneurs et des multiplieurs multi-rails d'une taille quelconque (signés ou non signés).

5.2 Un exemple de synthèse

Dans ce paragraphe, nous présentons la génération de circuits en réseau de cellules génériques. Comme nous l'avons énoncé au début du chapitre, les circuits – basés sur les modèles d'implémentation abordés §5.1.1 – sont générés de façon à garantir qu'ils soient QDI même si ils possèdent potentiellement des redondances. Il faut donc garantir que tous les signaux intermédiaires générés respectent la propriété QDI : ils doivent être acquittés.

Les implémentations des gardes et des blocs de calcul sont présentées dans les paragraphes précédents. Les blocs « union » dans ces modèles sont réalisés différemment en fonction des signaux à générer. Pour les signaux d'acquiescement des canaux d'entrée, l'union des signaux est effectuée par les portes « AND » car les signaux d'acquiescement sont actifs au niveau bas. Par contre, l'union des rails de données des canaux de sortie est effectuée par les portes « OR ». Ces portes sont en fait des portes exclusives mais grâce à la garantie d'exclusivité des choix, des portes « Or-inclusif » peuvent être utilisées.

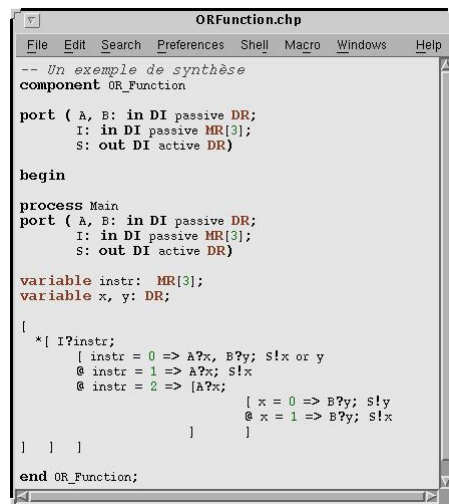


Figure 107 Exemple de la génération de la netlist de portes génériques

Pour illustrer la génération de la *netlist* de portes génériques et les problèmes associés, la génération de circuits en portes d'un exemple complet est présentée. La description de ce circuit est exprimée par le programme CHP ci-dessus (Figure 107). Dans cette description, en fonction de l'instruction (la valeur du canal I), les actions de communication correspondantes sont exécutées. De façon plus concrète, la première instruction correspond au calcul de l'opération « OR » sur les deux canaux d'entrée tandis que la deuxième consiste à envoyer la donnée issue du canal A vers la sortie. La troisième instruction est en fait une autre façon d'écrire la première instruction, mais permet d'illustrer la construction de gardes imbriquées.

Si on choisit le protocole de communication WCHB, les équations de dépendances optimisées – grâce aux règles d'optimisation présentées au §4.5.5 et §5.1.1.2 – de ce circuit sont les suivantes.

$$\text{Equation 5-12} \quad \left\{ \begin{array}{l} S^0 = F_{WCHB}^{données}(I = 0; A \text{ or } B; S^{acq}) \\ S^1 = F_{WCHB}^{données}(I = 1; A; S^{acq}) \\ S^2 = F_{WCHB}^{données}(I = 2, A = 0; B; S^{acq}) \\ S^3 = F_{WCHB}^{données}(V(B); I = 2, A = 1; A; S^{acq}) \\ S = F_{union}^{données}(S^0, S^1, S^2, S^3) \end{array} \right.$$

$$\text{Equation 5-13} \quad \left\{ \begin{array}{l} I^{0/acq} = A^{0/acq} = B^{0/acq} = F_{WCHB}^{acq}(V(S^0)) \\ I^{1/acq} = A^{1/acq} = F_{WCHB}^{acq}(V(S^1)) \\ I^{2/acq} = A^{2/acq} = B^{2/acq} = F_{WCHB}^{acq}(V(S^2)) \\ I^{3/acq} = A^{3/acq} = B^{3/acq} = F_{WCHB}^{acq}(V(S^3)) \\ A^{acq} = I^{acq} = F_{union}^{acq}(I^{0/acq}, I^{1/acq}, I^{2/acq}, I^{3/acq}) \\ B^{acq} = F_{union}^{acq}(I^{0/acq}, I^{2/acq}, I^{3/acq}) \end{array} \right.$$

Le modèle WCHB de la Figure 88 est appliqué, ce qui nous donne le schéma de la Figure 108. Quatre choix clairement identifiés de 0 à 3 dans ce schéma correspondent respectivement aux chemins : « $I=0$ », « $I=1$ », « $I=2 \text{ and } A=0$ » et « $I=2 \text{ and } A=1$ ». Les buffers WCHB placés à la sortie de chaque branche, permettent d'acquitter les canaux d'entrée consommés. En particulier, les canaux I et A – utilisés dans les quatre branches – sont exclusivement acquittés par un des quatre signaux d'acquiescement des buffers (les choix sont mutuellement exclusifs). Par conséquent, leurs acquiescements sont conduits par une porte « AND » à 4 entrées. Quant au canal B, comme il n'est pas consommé dans le deuxième choix ($I=1$), son signal d'acquiescement est seulement généré par trois signaux d'acquiescement sur quatre (porte « AND » à 3 entrées).

Les signaux de données sont les unions des signaux générés par 4 buffers (correspondant à 4 choix). L'exclusivité de ces portes « OR » est assurée par l'exclusivité des choix.

Les gardes imbriquées (les deux derniers choix) sont hiérarchiquement structurées en utilisant la technique présentée au paragraphe 5.1.2.3.5. En particulier, dans la quatrième garde (« $I=2 \text{ and } A=1$ »), le canal B est consommé mais jamais utilisé. Par conséquent, il faut qu'il soit valide afin d'autoriser l'écriture sur le canal de sortie. Comme nous l'avons énoncé précédemment (cf. §5.1.2.1 et §5.1.2.3.4), ce test doit être implémenté de façon à garantir que le circuit généré soit toujours QDI. De ce fait, la redondance peut avoir lieu et l'optimisation peut bien entendu être appliquées pour simplifier le circuit. Nous verrons dans la suite (§5.3.2) comment optimiser ce circuit.

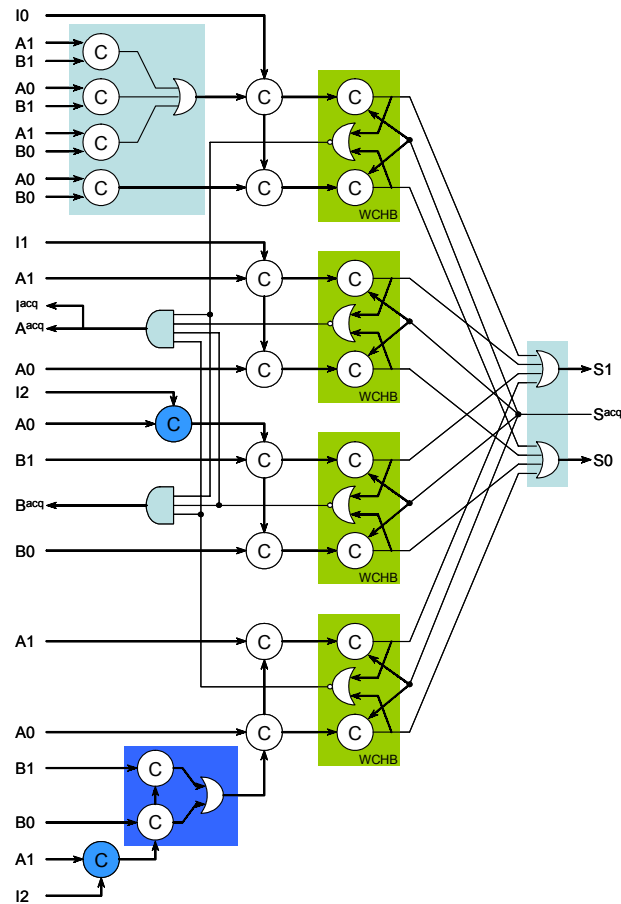


Figure 108 Implémentation en protocole WCHB de l'exemple de la Figure 107

Un autre problème peut se poser lors de la synthèse : à savoir si l'on veut décomposer certaines portes de *fan-in* élevé. Bien que les portes utilisées dans cette phase de synthèse sont les portes génériques générées à la volée (seul le fonctionnement est pris en compte), on veut dans certains cas limiter leur *fan-in* en les décomposant, tout en conservant le fonctionnement correct du circuit. Il s'agit de respecter les contraintes QDI telles que les fourches isochrones, aléas, ... Par exemple, d'après la génération présentée, si le nombre de choix dans un circuit est assez grand (facilement possible, notamment pour les encodeurs/décodeurs), les portes « OR » à la sortie et les portes « AND » pour la génération des signaux d'acquiescement peuvent avoir des *fan-in* élevés. La décomposition de ces portes peut être réalisée comme pour la logique synchrone : elle ne pose pas un problème connu car leurs entrées sont mutuellement exclusives.

En résumé, le circuit généré par la méthode de synthèse est par construction QDI. Le réseau de portes génériques est généré à ce niveau. Les équations logiques des sorties peuvent également être déduites. La *netlist* de portes du circuit est prête pour les tâches d'optimisation/projection. Dans le paragraphe suivant, nous présentons l'optimisation et la projection technologique pour des circuits asynchrones QDI.

5.3 Optimisation et projection technologique

L'objectif principal de la synthèse est de « traduire » la description de haut niveau en description de bas niveau, c'est-à-dire de permettre la construction du circuit en termes de signaux logiques propres aux implémentations matérielles. Cependant, la synthèse ne tient pas – ou peu – compte des contraintes de surface (nombre de portes) ou de vitesse (étages de portes) qui sont inhérents à toute réalisation. En effet, il existe possiblement une infinité d'implémentations

matérielles correspondant à la même description de haut niveau, et l'objectif ultime est de trouver un optimum par rapport aux contraintes posées par le concepteur.

De même que la synthèse est sensible aux contraintes de haut niveau : protocole, codage, informations macroscopiques, l'optimisation logique se base sur les résultats concrets de la synthèse (équations logiques) pour en proposer une implémentation fonctionnellement équivalente mais optimisée selon les critères matériels imposés par le concepteur. En fait, la présence de cette étape est motivée par l'extrême complexité posée par le problème de l'optimisation matérielle : pouvoir réaliser la même fonction logique avec un nombre de portes inférieur ou avec une rapidité/consommation inférieure est une perspective que l'on ne peut ignorer au risque de sacrifier une large part de l'efficacité finale du produit.

L'objectif de la projection technologique est de « traduire » un circuit logique quelconque (composé de portes de complexité variable) en un circuit fonctionnellement équivalent composé de portes (appelées cellules standard) définies à l'avance, dont le dessin (*layout*) a été réalisé et optimisé pour constituer le niveau de granularité le plus fin au moment du routage. Pour des raisons pratiques de développement, ces cellules sont en nombre réduit, de telle sorte que les contraintes appliquées à la tâche sont multiples. Il s'agit de décomposer le schéma du circuit initial afin de revenir à des portes disponibles dans la librairie, mais aussi à factoriser certaines parties pour optimiser le circuit en termes de vitesse et de surface : autant de changement de configurations qui rendent la tâche complexe et gourmande en puissance de calcul. Les limites de la tâche sont donc fixées par les librairies de cellules utilisées, elles-mêmes dépendantes des choix de protocoles asynchrones.

Telle qu'on l'a présentée au début de ce chapitre, la méthode d'optimisation et la projection technologique choisies reposent sur l'utilisation bien maîtrisée d'un certain nombre de règles « valides » permettant au circuit :

- de rester dans le cadre du modèle QDI
- de converger simultanément vers un circuit optimisé selon les critères fixés
- de converger vers un ensemble de portes de la bibliothèque spécifique.

Ces trois axes représentent chacun un problème à part entière, et l'on peut en venir à se demander si les deux derniers ne pourraient pas être dé-corrélés pour en simplifier le traitement. La chose est envisageable – tout comme pour le cas synchrone – mais les critères sont ici bien plus exigeants que traditionnellement, de telle sorte qu'il semble pour l'instant très délicat de trouver une modélisation dans laquelle l'optimisation et la projection technologique soient réellement indépendantes. Il apparaît donc dans un premier temps plus simple de développer un ensemble de règles permettant d'accomplir les deux à la fois, avec une efficacité certes limitée.

Dans ce qui suit, nous présentons les problèmes d'optimisation logique, de décomposition et de projection technologique de façon séparé. Avant de les décrire et de proposer les solutions adoptées, nous présentons les contraintes QDI.

5.3.1 Contraintes QDI

5.3.1.1 Aléa

Le circuit logique généré à l'issue de la synthèse doit être modélisé de telle sorte qu'il ne comporte pas d'aléa, c'est-à-dire sans possibilité de « glitch ». Dans le modèle asynchrone QDI, les signaux traversent globalement deux phases appelées phase d'activation (ou phase montante : la validité des signaux s'établit petit à petit) et phase de désactivation (ou phase descendante : les signaux retournent dans leur état non valide). Un signal ne doit en aucun cas s'activer

intempestivement. Plusieurs techniques de « renforcement » permettent de garantir que des *glitches* ne se produiront pas, mais elles requièrent souvent une grosse capacité de calcul.

5.3.1.2 Fourche isochrone

Il existe un certain nombre de cas complexes dans lesquels le modèle QDI est violé, et l'ensemble de cas n'est pas véritablement bien connu. Le principe de l'approche que nous adoptons est donc de savoir débiter, après synthèse, avec un circuit « certifié QDI » et de n'appliquer que des règles parfaitement connues pour conserver ce modèle. Pour être clair, le problème se pose principalement dans les cas où un signal intermédiaire peut s'activer sans que sa redescende ne soit vérifiée par la suite. C'est un cas très fréquent, il mobilise l'ensemble des calculs et oblige à effectuer la plupart des factorisations possibles. Par conséquent, la puissance de calcul constituera certainement un facteur limitatif dans la conception par cette approche.

Il est indispensable de bien connaître la structure de dépendances dans le circuit pour déduire le caractère isochrone de chaque nœud. Rappelons à cette occasion à quel point il est crucial de connaître la structure d'exclusivité (et d'exhaustivité) entre les signaux : savoir que deux signaux sont par construction exclusifs permet de procéder à un nombre de simplifications algorithmiques décisives qui bénéficient au temps de calcul.

5.3.2 Optimisation logique

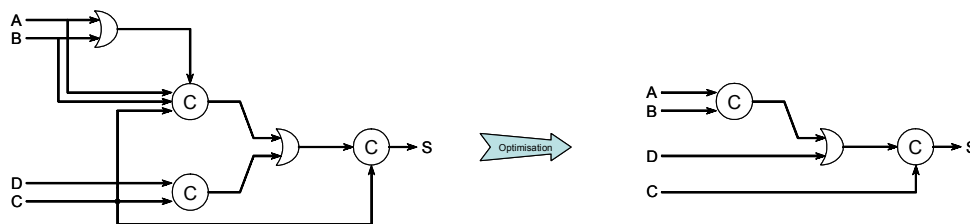


Figure 109 Exemple d'optimisation logique

La Figure 109 est la représentation typique de l'optimisation d'un bloc généré automatiquement par la synthèse (sans souci d'optimisation) et dont l'implémentation finale équivalente permet un gain en surface, consommation et vitesse. Des optimisations de ce type ne sont absolument pas rares, si bien que tout logiciel se doit de gérer le problème sérieusement.

Les bases théoriques de l'optimisation logique [GIOV94] sont sensiblement les mêmes que celles de la logique numérique. Pendant longtemps, les tableaux de Karnaugh et les lois de De Morgan ont été les principaux outils permettant au concepteur d'entrevoir l'implémentation optimale de son circuit, en procédant à la main ou avec l'aide de logiciels. Aujourd'hui, la théorie des BDDs (« Binary Decision Diagram » ou arbres de décision binaire) permettant de modéliser efficacement les fonctions logiques par ordinateur se révèle des plus satisfaisantes, de telle façon qu'elle est largement utilisée dans les logiciels du commerce (synchrone). Mais elle est loin d'être la seule : il est tout à fait possible de procéder par des tests d'équivalence fonctionnelle (analyse symbolique, analyse exhaustive dirigée, etc.) ou de réaliser l'optimisation – qui reste l'objectif de cette étape – en utilisant récursivement certaines règles de simplification bien choisies (fonction d'évaluation efficace d'un système expert, ...)

Autant l'objectif de l'optimisation est relativement bien défini, autant les moyens mis en œuvre à ce jour nécessitent des ressources de calcul importantes, algorithmiquement parlant, tout test d'égalité fonctionnelle (nécessaire à la majorité des étapes de l'optimisation) représente un problème *NP-complet* dont on ne peut prévoir à l'avance le temps de résolution. Les logiciels se basent donc sur des suppositions permettant de restreindre les calculs nécessaires, d'écarter

l'exploration d'un certain nombre de possibilités jugées non optimales... mais il est rarement possible de conclure sur le caractère optimal de la solution adoptée.

Présentés au §6.7, les algorithmes d'optimisation logique adoptés sont les algorithmes synchrones « renforcés » des contraintes QDI. Le logiciel développé a pour objectif de parcourir l'ensemble des factorisations possibles pour le circuit et n'exécute que celles qui réalisent un certain optimum (selon les critères prédéfinis) et pour lesquelles les contraintes QDI sont respectées. Ce dernier point n'est pas pour autant figé : il semble que dans certains cas, l'ajout d'une porte dans une fourche isochrone soit possible si l'on prend soin de faire quelques hypothèses faibles sur le temps de parcours. Cette ouverture vers les circuits appelés QⁿDI ([PEET90]) permet de réaliser à l'aide d'un même logiciel des circuits strictement QDI et des circuits basés sur un degré d'hypothèses temporelles croissant. Les promesses de cet objectif ambitieux pourraient se révéler très intéressantes car elles regroupent sous un même noyau logiciel la décision de factorisation optimale et la validation de cette factorisation selon des hypothèses temporelles très souples.

Hormis l'approche d'optimisation logique globale qui prend une puissance de calcul non négligeable, certaines routines d'optimisations évidentes sont effectuées avant, afin d'alléger dès le début l'optimisation globale du circuit. Il s'agit d'un premier niveau d'optimisation du circuit. Parmi ces routines, citons les plus utilisées :

- Génération de signaux de validité globaux pour ne pas avoir à créer une fonction redondante à chaque fois
- Elimination de la redondance entre les signaux de test de validité et les signaux de test d'une voie sur un multi-rails. Dans ces cas, la validité est assurée par la présence de la voie seule, le signal de validité globale est superflu.
- Elimination des portes « sans consistance » : portes à une entrée, portes sans sortie, *etc.* Ces portes apparaissent dans des traitements généraux sur le circuit : il faut donc les éliminer par une analyse globale.
- Elimination des signaux « constants » (VCC et GND) entraînant des modifications potentiellement importantes dans le circuit (élimination de tout un arbre, *etc.*).
- Elimination des portes avec les entrées croisées : les portes de Muller, par exemple, ayant les entrées provenant des différents rails d'un digit du canal d'entrée peuvent être supprimées.
- Factorisation des portes semblables lorsque deux portes ont exactement les mêmes entrées et le même type (et qu'une au moins des sorties peut être remplacée). Il s'agit bien sûr d'un cas de factorisation triviale.

Ces optimisations sont répétées pour chaque itération d'optimisation afin de pouvoir propager leurs effets vers les étages en aval. Elles peuvent être effectuées en alternant avec les optimisations QDI présentées §6.7.

5.3.3 Décomposition : situation du problème

Sans faire appel à des concepts de réduction logique (comme la simplification, la factorisation, ...), on peut en asynchrone procéder de manière radicalement plus pragmatique. Les portes à optimiser dans le circuit sont le plus souvent des portes possédant un grand nombre d'entrées (*fan-in*). Aucune porte de la bibliothèque ne peut assurer une telle fonction. Quelle que soit la méthode envisagée (à quelques rares exceptions près), il est alors nécessaire de scinder la porte d'une manière ou d'une autre pour factoriser une partie des entrées ou simplement modéliser la porte complexe avec des portes plus simples.

Si le fait de grouper plusieurs portes ensemble ne pose pas de problème majeur (il y a élimination de signaux), l'opération consistant à scinder une porte complexe en portes plus simples comporte, elle, un certain nombre d'inconnues qu'il s'agit de résoudre efficacement.

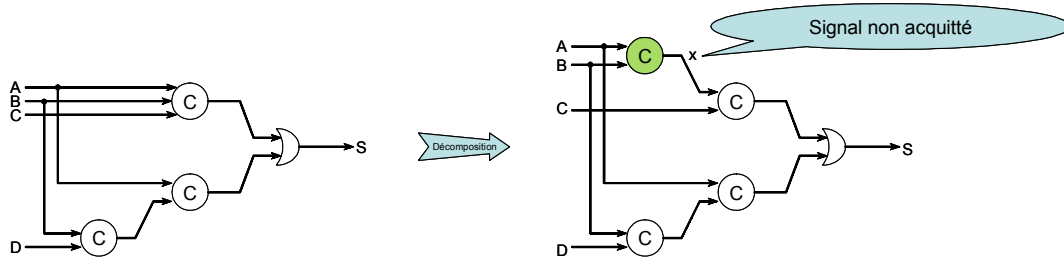


Figure 110 Problème de décomposition des circuits QDI

L'exemple dans la Figure 110 ci-dessus montre que la décomposition classique (telle qu'on peut la pratiquer dans le cas synchrone) n'est pas valide dans un nombre important de cas en asynchrone. Le signal intermédiaire x , issu de la décomposition, représente la même fonction dupliquée et peut tout à fait être activé sans que la porte en aval le soit. Dans ce cas, si le chemin en bas est lui-même activé, la redescende du nouveau signal intermédiaire n'est pas vérifiée. En effet, le circuit n'est absolument pas QDI et la décomposition n'est pas valide.

Il est possible d'élaborer un certain nombre de règles de décomposition portant sur le signal intermédiaire créé pour valider ou non l'opération. Ces règles, montrées et prouvées formellement dans [BURN96], sont basées sur la dépendance d'un signal par rapport aux entrées/sorties du processus. En effet, il est possible de dupliquer une même fonction dans deux chemins de données totalement indépendants, mais pas sur le même chemin de données.

La validité de telles opérations exige une analyse non triviale du circuit demandant elle-même beaucoup de ressources de calcul. Il convient donc de garder à l'esprit que le problème de la division des portes est central dans la phase d'optimisation/projection technologique et que le gérer avec efficacité représente également un objectif primordial.

Dans l'outil de synthèse TAST, la décomposition de portes de *fan-in* élevés est effectuée dans les cas où la propriété d'insensibilité aux délais est garantie par l'exclusivité des signaux. Un exemple de cette décomposition est présenté dans le paragraphe 5.2. Pour les portes de Muller, aucune décomposition n'est effectuée automatiquement. Par contre, la description de circuits – représentée par un programme CHP – peut être modifiée pour limiter de *fan-in* des portes

5.3.4 Projection technologique : situation du problème

L'ultime étape avant le routage (placement des éléments sur le masque définitif du circuit) est la projection technologique. Ainsi qu'on l'a vu au début de ce chapitre, la projection technologique consiste à transformer le schéma logique optimisé en schéma de portes existantes issues de la bibliothèque de cellules prédéfinies. Les fonctions logiques complexes n'ayant pas d'équivalent en *layout* (plan d'une région fonctionnelle du circuit), il s'agit de les décomposer en portes élémentaires présentes dans la librairie. Indéniablement, le problème d'optimisation se pose une fois de plus dans la mesure où la décomposition n'est jamais unique. Si en revanche la librairie de cellules contient des sur-éléments de portes à implémenter, une factorisation est possible. La projection technologique est intimement lié à l'optimisation de telle manière que beaucoup de logiciels traitent les deux conjointement.

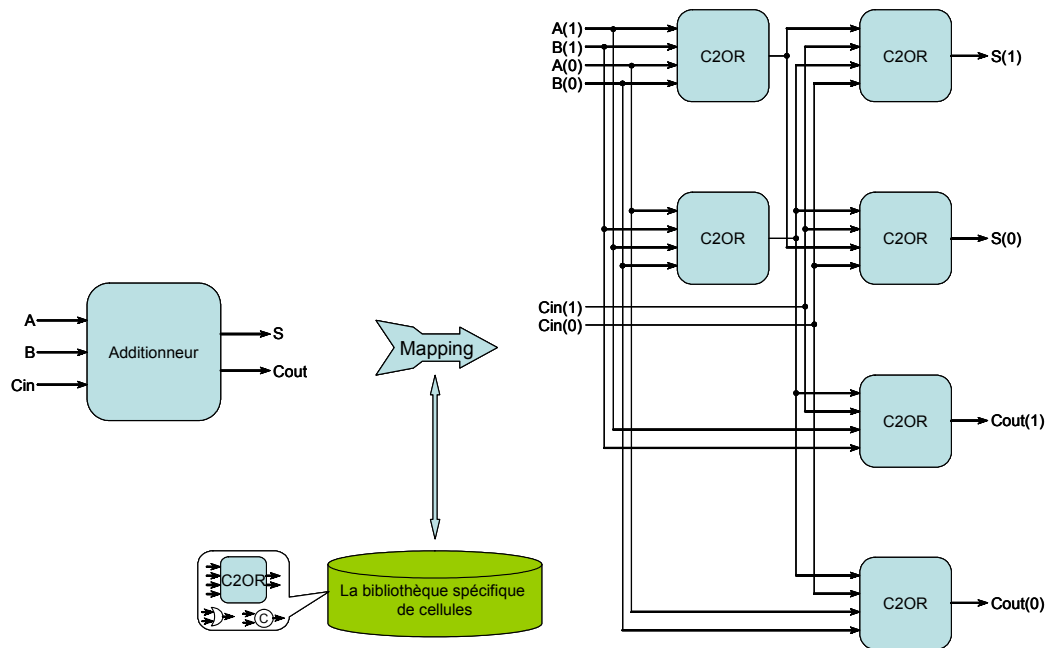


Figure 111 Principe général de la projection technologique

Plusieurs techniques ([CORT97b], [CHOU99], [BEER96], [GIOV94]) ont été utilisées pour réaliser la projection technologique de structures matérielles très diverses. L'implémentation de circuits intégrés basés sur des cellules standard requiert naturellement une phase de projection technologique essentielle qui fait le lien entre le circuit optimal proposé et le circuit effectivement possible à implémenter avec les moyens disponibles dans la bibliothèque. Le principe de ce type de circuit (« semi-custom design ») est basé sur la réduction du temps de développement global. En augmentant la taille du niveau de granularité le plus fin (du transistor à la cellule), on simplifie le routage (moins d'éléments) tout en réduisant le travail à fournir par le concepteur. La contrepartie de ce choix est immédiate : on ne peut optimiser le circuit au niveau du transistor. Le compromis entre les deux types de circuits les plus répandus (*full-custom* au niveau transistor et *semi-custom* au niveau cellule standard) ne se traduit pas nécessairement par des circuits beaucoup moins performants en cellules standard. Dans ce type de circuits, les cellules sont elles-mêmes « dessinées » mieux que le logiciel de routage ne saurait le faire.

Certaines autres structures matérielles ont motivé le développement de logiciels de projection technologique différents. Le circuit de composant programmable type FPGA ou CPLD a toujours nécessité de « projeter » un circuit logique sur un ensemble de portes limitées. Certaines méthodes basées sur l'exécution d'automates de reconnaissance ont été implémentées avec un succès évident (autant concernant leur simplicité que leur efficacité).

Ainsi, la projection logique, opération complexe à réaliser, est très liée au problème d'optimisation où l'équivalence des fonctions logiques doit être couramment testée. La progression par étapes vers le circuit optimal espéré s'effectue grâce à la définition de règles de base permises pour conserver la fonctionnalité du circuit (règles électriques, *fan-in*, nombre d'étages de portes, *etc.*).

5.4 Conclusion

Nous venons de présenter la synthèse automatique de circuits asynchrones de type QDI en portes standard. Cette synthèse part des équations de dépendances des circuits qui expriment les dépendances de sorties en fonction des entrées. Le protocole de communication et le modèle de circuit cible choisi sont pris en compte à ce niveau. L'implémentation QDI de plusieurs opérateurs tels que le test de validité, l'initialisation, opérateurs logiques (sonde de canal, « AND », « OR », ...),

opérateurs de comparaison (égalité, inégalité, ...) et opérateurs de calcul (convertisseur de types de données, « AND », « OR », « XOR », ...), est présentée. Grâce à eux, les circuits asynchrones sont générés. Ils sont garantis QDI par construction.

Une des étapes la plus cruciale pour la synthèse, notamment pour les circuits asynchrones de type QDI, est l'optimisation et la projection technologique (donc la décomposition). La synthèse de circuits QDI doit non seulement garantir l'équivalence du circuit (avant et après la synthèse) vis-à-vis du fonctionnement dans le mode entrée-sortie mais encore prendre soin des contraintes QDI telles que les fourches isochrones. Certaines optimisations pour les circuits QDI sont proposées et implémentées dans TAST, mais elles sont encore partielles. D'autres, plus agressives sont proposées au Chapitre 6 et seront implémentées dans le futur.

Chapitre 6

SYNTHETISEUR DE CIRCUITS QDI : TAST

6.1 Introduction

Bien que les circuits asynchrones promettent d'apporter beaucoup d'avantages par rapport aux circuits synchrones, leurs applications sont encore en petit nombre. L'incompatibilité des outils dédiés aux circuits synchrones, le souci de la correction de fonctionnement des circuits du aux aléas, le manque d'une méthodologie de conception de haut niveau, ... font que les circuits asynchrones sont peu utilisés à l'heure actuelle par l'industrie. Dans de telles circonstances, un environnement de conception et des outils d'aide à la conception des circuits asynchrones sont susceptibles d'améliorer l'introduction/adoption des circuits asynchrones dans l'industrie. Dans le but de mieux gérer la conception des circuits asynchrones, nous développons un environnement de conception, TAST [DINH02][DINH02b], qui comporte l'ensemble des outils d'aide à la conception de circuits asynchrones. Le but ultime de ces outils associés avec un langage de conception et de simulation puissant, tel que le langage de description des processus communicants de haut niveau CHP, est de générer le réseau de portes d'un circuit asynchrone à partir de sa description comportementale de haut niveau. Ce réseau de portes sert à réaliser le dessin des masques (*layout*) du circuit. Cet environnement présenté dans ce chapitre est un prototype de la méthodologie de conception décrite dans les chapitres précédents.

S'il est un concept sur lequel toute la conception numérique s'appuie largement, c'est bien celui de la hiérarchie. La nécessité d'évoluer depuis une description haut niveau vers la réalisation matérielle reflète précisément le degré de complexité élevé qui ne permet pas de considérer simultanément toutes les caractéristiques d'un circuit dans son ensemble. Le mode de conception dénommé « top-down », largement utilisé en asynchrone, correspond au développement d'un circuit en commençant par les aspects comportementaux pour évoluer au fur et à mesure vers les aspects les plus techniques et enfin aboutir à la réalisation. C'est dans ce cadre que la description de type CHP s'affranchit des caractéristiques d'implémentation pour ne considérer que les éléments de type communication ou fonction macroscopique. L'objectif d'un logiciel de CAO dans ce domaine est donc d'assister le concepteur dans les tâches les plus complexes afin d'aboutir en fin de compte au circuit le plus performant (selon les critères choisis) en un temps minimal.

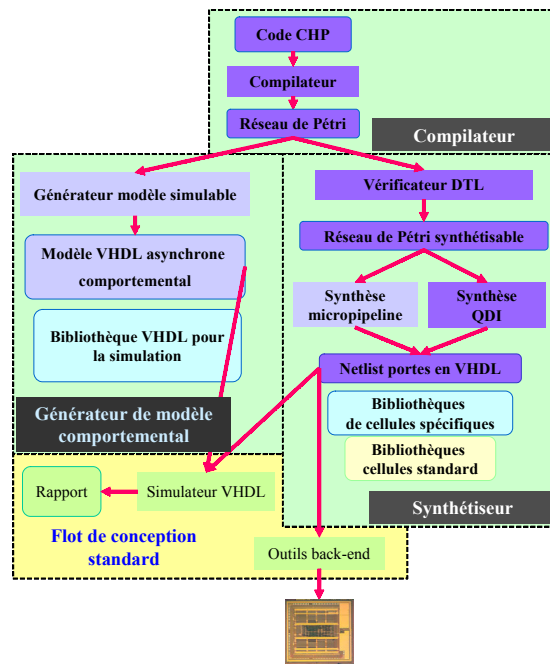


Figure 112 Flot de synthèse implémenté dans TAST

La Figure 112 représente le flot de conception des circuits asynchrones implémenté dans l'environnement de conception TAST. Partant du code initial – de type CHP – correspondant à la description haut niveau, le logiciel est en charge de détailler les « concepts » (communications, opérations logiques) utilisés pour progresser vers des degrés d'abstraction de moins en moins élevés. La synthèse est précisément la première étape de ce développement : le logiciel y modélise chacune des opérations « haut niveau » avec des opérations plus explicites (en utilisant des informations pré-programmées : protocole, codage, types d'implémentation). Le plus souvent, cette étape s'achève avec la définition d'équations logiques réalisant le plus efficacement les spécifications données dans le code initial. Cependant, l'intervention du concepteur durant le processus de synthèse n'est pas rare : soit elle est automatique (paramètres prédéfinis), soit elle consiste à orienter le processus pour optimiser chaque niveau de la hiérarchie (équilibre des pipelines, meilleur traitement du chemin de données critique, *etc.*).

Par ailleurs, les étapes de synthèse nécessitant une part non négligeable d'analyse de code (initial et intermédiaire), de nombreux logiciels accomplissent une série de vérifications permettant d'écarter le plus tôt possible les cas non réalisables ou posant un problème quelconque. Parmi eux, une vérification importante concerne la compatibilité des règles DTL. Dans le cas où le circuit n'est pas conforme aux règles DTL, celui-ci se refuse de synthétiser pour éliminer les solutions *a priori* non fonctionnelles. La synthèse assure donc une série de tâches complexes permettant de passer de concepts abstraits à des équations logiques déjà bien proches de l'implémentation finale : il s'agit d'une phase mettant en jeu de nombreuses compétences algorithmiques.

Mentionnons simplement quelques techniques algorithmiques essentielles à ce niveau telles que les réseaux de Petri qui permettent de modéliser les états fonctionnels d'un système numérique et les relations qui les lient avec une complexité algorithmique réduite. De nombreuses méthodes de synthèse proposées utilisent ces techniques afin d'accomplir des analyses fonctionnelles et de simplifier les équations logiques finales.

Une autre étape du développement de la description haut niveau du circuit vers l'implémentation matérielle qui est cruciale non seulement pour les circuits asynchrones mais aussi pour les circuits synchrones, est l'optimisation. Le circuit généré par l'étape de synthèse doit être

optimisé selon les critères définis par le concepteur vis-à-vis de la performance en surface et en vitesse, tout en gardant la correction fonctionnelle. Il s'agit surtout d'éviter les aléas dus aux transitions de signaux dans des circuits asynchrones. L'étape d'optimisation de circuits est réalisée en lien avec une technologie spécifique : les circuits à optimiser seront continuellement traduits en des implémentations de bas niveau correspondantes à une bibliothèque spécifique de cellules. Telle qu'on l'a présentée dans le Chapitre 5, l'optimisation et la projection technologique choisies doivent permettre au circuit de rester dans le cadre du modèle QDI, de converger simultanément vers un circuit optimisé selon les critères fixés et de converger vers un ensemble de portes de la bibliothèque spécifique. Ces trois axes représentent chacun un problème à part entière. La chose est envisageable – quand on veut séparer les deux tâches pour en simplifier le traitement – mais les critères sont ici bien plus exigeants que traditionnellement, de telle manière qu'il semble pour l'instant très délicat de trouver une modélisation dans laquelle l'optimisation et la projection technologique soient réellement indépendantes. Il apparaît donc dans un premier temps plus simple de développer un ensemble de règles permettant d'accomplir les deux à la fois, avec une efficacité limitée.

Ainsi, TAST apporte des contributions sur plusieurs points :

- un environnement complet se composant des outils d'aide à la conception de circuits asynchrones de la description haut niveau jusqu'à l'implémentation bas niveau
- une description haut niveau des circuits asynchrones avec une complexité importante et une hiérarchie de profondeur peu importante
- une méthode de synthèse capable de produire des réalisations dans des modèles différents de circuits asynchrones (QDI, micropipeline) avec diverses fonctions de coût (encodage de données, protocole de communication), permettant l'exploration de différents styles de conception
- une co-simulation des implémentations fonctionnelles et matérielles en utilisant des outils commerciaux

Le logiciel TAST est implémenté en langage C ANSI. Il contient actuellement 50.000 lignes de code structuré en de nombreuses bibliothèques d'outils qui être réutilisées.

6.2 Interface utilisateur

L'environnement TAST est implémenté comme un *shell* à partir duquel il est possible d'appeler les différents outils. En effet, il est un interpréteur de commandes dans lequel les commandes du concepteur sont manipulées et interprétées : substitution des variables dans les commandes et expansion des commandes avec les alias. Enfin selon la commande et ses arguments, l'outil correspondant est invoqué pour accomplir la tâche désirée. Les avantages de l'implémentation de l'environnement sous forme d'un interpréteur de commandes sont les suivants.

- les commandes peuvent être saisies manuellement ou grâce à un fichier de commandes (un script), ce qui permet notamment de lancer la synthèse d'un circuit asynchrone complexe dans un ordre prédéfini
- les tâches de conception peuvent se faire de façon progressive – notamment l'optimisation de circuits qui nécessite parfois plusieurs passes – sans refaire certaines étapes (les résultats des étapes sont toujours dans la mémoire)

Il existe une politique de travail dans l'environnement TAST : il s'agit de faire cohabiter et d'identifier des objets stockés dans TAST. Les circuits asynchrones à concevoir et leurs résultats (les arborescences syntaxiques, les réseaux de Petri, les équations de dépendances, les équations logiques,

les réseaux de portes, ...) sont symbolisés dans la mémoire de façon qu'ils peuvent être facilement identifiés les uns des autres. Chaque objet est identifié par son type et un nom assigné par le concepteur. Un ensemble de réseaux de Petri représentant un circuit asynchrone, par exemple, est identifié par le type PN et un nom. De cette façon, plusieurs circuits peuvent être représentés en plusieurs formes dans la mémoire : chaque forme correspond à une action effectuée sur le circuit ou à la même action mais avec des paramètres différents. En particulier, le concepteur peut effectuer, sur un même circuit asynchrone, plusieurs synthèses et les optimisations avec différentes options (protocole de communication, technologie cible, paramètre d'optimisation, ...) afin de comparer leurs résultats. Il est à noter qu'à tout moment, on peut savoir quels sont les objets stockés dans la mémoire.

Il existe également une interface graphique à partir de laquelle le concepteur peut graphiquement effectuer la conception de circuits asynchrones, ce qui rend la tâche de conception plus facile. La Figure 113 illustre l'interface graphique de l'utilisateur qui est simplement l'interface de commandes. Dans ce qui suit, nous présentons les différents modules en accompagnant les commandes correspondantes qui sont essentielles à l'environnement TAST.

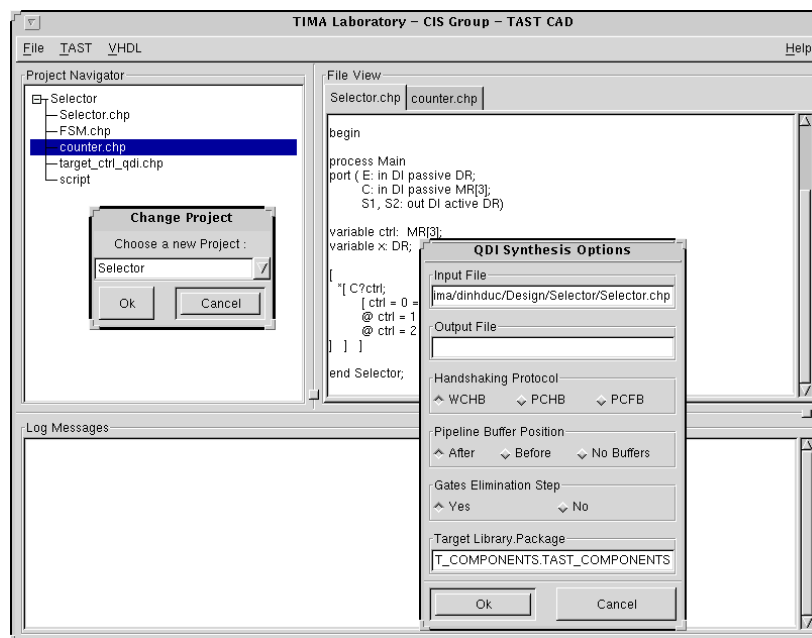


Figure 113 Interface graphique de l'utilisateur

Hormis les outils servant à la conception et à la gestion de circuits asynchrones, TAST offre aussi un environnement de travail commode au concepteur. Certains de ses avantages sont énumérés ci-dessous :

- Il existe des commandes permettant de connaître les ressources (temps de calcul, la mémoire, ...) utilisées par le système avant et après une tâche.
- L'utilisation du mécanisme de complétion, s'il n'y a pas d'ambiguïté sur les noms des commandes ou des fichiers (y compris avec leurs chemins de répertoire) permet de compléter automatiquement les caractères entrés.
- Les dernières commandes exécutées sont gardées dans un mini-tampon et peuvent être facilement rappelées et même éditées.
- Toutes les commandes du système d'exploration dans lequel TAST est hébergé, peuvent être invoquées à partir de l'environnement TAST, ce qui permet au concepteur de lancer ces commandes sans quitter TAST.

- Certaines commandes usuelles du système d'exploitation telles que « cd », « pwd », « ls », ... peuvent être directement appelées dans TAST : elles sont considérées comme les commandes de l'environnement TAST.
- Les commandes de TAST peuvent facilement être personnalisées en utilisant des « alias » : elles peuvent être abrégées ou redéfinies avec les options utilisées couramment.
- Les variables d'environnement prédéfinies dans les systèmes d'exploitation Unix/Linux peuvent être utilisées dans les commandes ou les scripts de TAST.
- Les aides en ligne sur les commandes de TAST peuvent être pratique pour les concepteurs lors de la conception

6.3 Module parseur syntaxique et analyseur contextuel du langage CHP

Avant de pouvoir s'attaquer aux problèmes cruciaux que représente la synthèse, le concepteur de circuits asynchrones doit interpréter la description du circuit en langage de haut niveau CHP. TAST la traduit en une structure intermédiaire – la combinaison des réseaux de Petri et les graphes flot de données – utilisé par la suite. Cette étape, appelée *parsing*, s'effectue à l'aide des outils LEX/YACC intégrés à l'environnement TAST. Le module parseur syntaxique du langage CHP se compose d'un analyseur lexical et d'un analyseur syntaxique. L'analyse lexicale est la première phase de la compilation lors de laquelle on réunit les symboles en « lexèmes » (c'est-à-dire en éléments signifiants de base dans une grammaire). En fait, pour chaque lexème reconnu, l'analyseur génère un code correspondant (appelé « token ») qui va être utilisé par l'analyseur syntaxique. L'analyse syntaxique est une phase de la compilation pendant laquelle est vérifiée la conformité vis-à-vis d'une grammaire d'une succession de « lexèmes ». Autrement dit, après la reconnaissance lexicale, l'analyse syntaxique consiste à reconnaître des structures du langage CHP en se basant sur la grammaire hors contexte du langage donnée sous forme de règles. Chaque reconnaissance fait appel à une fonction qui construit la structure de données correspondante à la règle. L'interaction entre l'analyseur lexical et l'analyseur syntaxique, résumée de manière schématique à la Figure 114, est couramment implantée en faisant de l'analyseur lexical un sous-programme de l'analyseur syntaxique. Pour ce fait, une structure de données – la table des symboles – est utilisée. Elle sert à stocker les informations qui concernent les identificateurs du programme source (par exemple leur type, leur valeur, leur portée, nombre et type, ...). Le remplissage de cette table (la collecte des informations) a lieu lors des phases d'analyse. Les informations contenues dans la table des symboles sont nécessaires lors des analyses syntaxique et sémantique, ainsi que lors de la génération du réseau de Petri dans la suite.

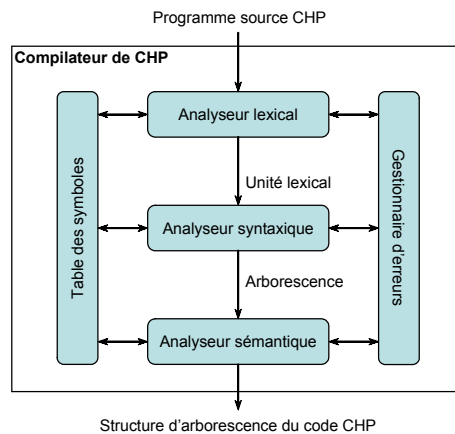


Figure 114 Interaction dans le module parseur du langage CHP

L'idée de base du *parsing* est en effet de générer une structure d'arborescence représentant l'image du code source CHP, à partir de laquelle on obtient la structure intermédiaire réseau de Petri et graphe flot de données. On tente le moins possible de regrouper ensemble les éléments qui pourraient l'être et on sépare les arguments les uns des autres. De cette manière, la structure initialement décrite dans le code source CHP se trouve conservée. C'est ce qu'on pourrait nommer « la mise en forme hiérarchique ».

Suite à la phase de *parsing*, la structure d'arborescence peut être incohérente (la connexion des ports de même type par exemple). Par conséquent, une vérification contextuelle est nécessaire. Chargé de cette tâche, l'analyseur contextuel parcourt l'arborescence pour vérifier sa cohérence. Il vérifie ainsi les déclarations, les types de données, la visibilité des variables ainsi que la connectivité des ports. Suivant la vérification, des conversions de types de données peuvent être réalisées sur les constantes. Ainsi afin d'être compatibles, ces dernières peuvent être tronquées ou remplies avec des « zéro » dans les cas d'un débordement ou d'un manque, respectivement.

Chaque phase du compilateur du langage CHP peut indiquer des erreurs. Il s'agit de les détecter et d'informer l'utilisateur le plus précisément possible : erreur de syntaxe, erreur de sémantique, erreur de contexte, erreur système, erreur interne. Après avoir détecté une erreur, il s'agit ensuite de la traiter de telle manière que la compilation puisse continuer et que d'autres erreurs puissent être détectées (un compilateur qui s'arrête à la première erreur n'est pas très performant). Bien sûr, il y a des limites à ne pas dépasser et certaines erreurs (ou un trop grand nombre d'erreurs) peuvent entraîner l'arrêt de l'exécution du *parsing*.

Dans l'environnement TAST, pour analyser un programme CHP décrivant le circuit asynchrone à concevoir, la commande « readchp » est appelée. Cette commande accomplit l'analyse syntaxique (et éventuellement l'analyse lexicale) et l'analyse sémantique. Le résultat de cette commande est une structure d'arborescence représentant le code source CHP du circuit asynchrone. Cette arborescence – stockée en mémoire – peut être réécrite en un fichier texte CHP grâce à la commande « writechp » servant initialement à vérifier la compilation.

6.4 Module traducteur en réseau de Pétri

Ce module – appelé la commande « chp2pn » – est utilisé pour compiler l'arborescence syntaxique stockée dans la mémoire en une forme intermédiaire : les réseaux de Petri et les graphes de flot de données. Comme l'énonce le Chapitre 4, servant à séparer le *front-end* et le *back-end* de l'outil de synthèse, la représentation intermédiaire a la propriété d'être générale et de ne pas faire

d'hypothèse sur un type de circuits précis. L'utilisation d'une forme intermédiaire indépendante de l'implémentation cible permet :

- un « reciblage » facilité : on peut créer un outil de synthèse pour une autre implémentation de circuits (différent codage de données, protocole de communication, modèle de circuits, ...) en modifiant seulement le *back-end* des outils de conception,
- de traduire en un autre langage, ce qui sert à simuler fonctionnellement le programme initial,
- d'appliquer un optimiseur de réseau de Petri indépendant de l'implémentation finale à cette représentation intermédiaire,
- d'appliquer une vérification formelle [BORR03].

Dans le cadre du projet TAST, tous les outils intégrés dans l'environnement TAST sont organisés autour de la forme intermédiaire. Dans le *front-end*, nous avons considéré la traduction de cette forme intermédiaire d'un circuit asynchrone en langages VHDL et C (voire SystemC ou autres) pour effectuer la simulation fonctionnelle du circuit grâce aux outils standard intégrés. Dans le *back-end*, deux types d'implémentations de circuits asynchrones sont ciblés à partir de cette représentation intermédiaire : l'implémentation QDI – qui est présentée ce travail de thèse – et l'implémentation micropipeline.

Pour « traduire » l'arborescence syntaxique en réseaux de Petri annotés par des graphes de flot de données, il faut disposer de fonctions traduisant chacune des structures du langage, représentées par l'arborescence, en un réseau de Petri. L'idée qui nous a permis de trouver la méthode de conception des réseaux de Petri est la généralisation de la notion de commandes gardées à celle de processus gardés. En effet, comme nous l'avons présenté §4.2.3, nous considérons chaque instruction comme un processus gardé par une condition de franchissement. Ainsi la construction du réseau de Petri se fera par composition récursive de ces processus. Le processus de base, *i.e.* une simple instruction, est ainsi composé d'une transition portant la condition vraie et d'une place comportant l'instruction. Il est à noter que chaque réseau de Petri contient les informations concernant des canaux et des variables comme le type de données, le sens de la connectivité, ...

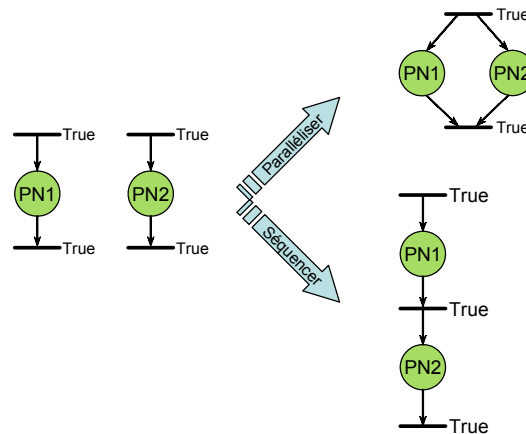


Figure 115 Construction des réseaux de Pétri

Après avoir obtenu les premiers réseaux de Petri, le compilateur cherche à optimiser les réseaux de Petri qu'il engendre, de manière à les rendre soit plus concis, soit plus efficaces, voire les deux. Il contiendra une fonction d'optimisation des réseaux de Petri. Typiquement l'algorithme d'optimisation revient à remplacer, à un stade ou un autre de la construction, une structure par une structure équivalente plus concise et/ou efficace. Des optimisations peuvent intervenir à plusieurs stades de la construction. Plusieurs peuvent être faites à l'analyse sémantique en tenant compte des

renseignements obtenus par consultation de la table des symboles. D'autres sont parfois réalisables sur les réseaux de Petri eux-mêmes; la construction a souvent tendance à produire naïvement les réseaux de Petri dans lesquels apparaissent des constructions manifestement trop lourdes qu'il est possible de simplifier dans une phase ultérieure. Ces optimisations ne dépendent pas de l'implémentation cible :

- propagation des constantes, s'il existe une variable ayant toujours la même valeur tout le long d'une partie du programme
- élimination de morceaux de réseau de Petri inutiles, si les places et les transitions n'apportent aucune information sur l'exécution des réseaux de Petri
- détection de code « mort », c'est-à-dire d'une partie du programme qui n'est jamais atteinte.

6.5 Module vérificateur de règles DTL

Appelé par la commande « checkDTL », le module vérificateur de règles DTL permet de contrôler la nature synthétisable des réseaux de Petri représentant le programme du circuit asynchrone. Plusieurs vérifications sont effectuées dans ce module.

6.5.1 Vérification des accès séquentiels et parallèles

L'accès séquentiel aux mêmes canaux (en lecture ou en écriture) n'est pas autorisé afin que les réseaux de Petri puissent être synthétisés par la suite. Bien entendu cette restriction ne s'applique pas à l'initialisation des canaux, une exception des accès séquentiels.

La vérification de l'accès parallèle s'applique à des canaux autant qu'à des variables. Elle interdit l'accès parallèle à un canal de communication (lecture ou écriture). Quant à elles, les variables locales sont autorisées en lecture concurrente (consultation du contenu de la variable) et non en écriture.

Pour tirer le meilleur parti du parcours des réseaux de Petri, les vérifications des accès séquentiels et parallèles sont conjointement réalisées par un algorithme unique. Puisque les restrictions sur les canaux et sur les variables sont légèrement différentes, les algorithmes de vérification pour les canaux et pour les variables diffèrent du point de vue de l'opérateur de comparaison, mais la méthode de parcours du réseau de Petri reste la même.

6.5.2 Vérification des dépendances et de l'initialisation

Comme annoncé au paragraphe 4.3.4, la vérification des dépendances sert à garantir une vraie dépendance de données pour les affectations séquentielles. Elle est vérifiée par des substitutions successives des valeurs lues et/ou calculées des canaux d'entrée. La compatibilité de types de données des opérandes est également vérifiée dans cet algorithme. Il s'agit de s'assurer que les bases de digits et les longueurs des opérandes sont égales respectivement.

La vérification de l'initialisation, quant à elle, consiste à vérifier que lorsqu'un canal est initialisé, tous les digits de données du canal sont initialisés avec des valeurs constantes.

Hormis ces vérifications, le module vérificateur contrôle également les opérateurs qui ne sont pas synthétisés actuellement : les opérateurs modulo, multiplication, division, ...

Ces vérifications sont réalisées après la vérification des accès séquentiels et parallèles. Elles ont un caractère commun : elles se font au cours des exécutions du réseau de Petri. De ce fait, un réseau de Petri initial doit être « déployé » en plusieurs réseaux de Petri (chaque réseau correspond à

une branche d'exécution) afin de simplifier les vérifications. Ils ne possèdent plus ni de places de sélection (divergence en « OU ») ni de places de fusion (convergence en « OU »).

6.6 Module générateur des équations de dépendances et logiques

A partir des réseaux de Petri vérifiés et transformés, les équations de dépendances des sorties en fonction des entrées sont générées. Comme énoncées précédemment, elles sont indépendantes de l'implémentation des circuits : protocole de communication, modèle de circuits, technologie cible. Les réseaux de Petri sont parcourus pour collecter des dépendances des sorties (signaux de données, acquittements) en fonction des entrées, permettant de générer ensuite les équations de dépendances du circuit.

L'étape suivante de la synthèse de circuits asynchrones est de développer les équations de dépendances obtenues avec les paramètres de synthèse adoptés. Cette étape est le prototype de la méthode présentée §5.1.1. Les paramètres de synthèse comprennent le protocole de communication (séquentiel, WCHB, PCHB ou PCFB) et le modèle de circuit (il s'agit de la position du *buffer* dans le circuit généré). Selon les paramètres contrôlés par le concepteur, le circuit asynchrone est généré dans un premier temps sous forme d'équations logiques. Constituant l'image matérielle du circuit QDI, ces équations logiques sont les entrées de l'optimisation et la projection technologique dont les algorithmes sont présentés par la suite. A ce stade, on peut créer le fichier texte (format EQN) représentant les équations logiques.

Ce module – la commande « *synthesizeQDI* » – est le noyau de la synthèse de circuits asynchrones QDI. Il permet de synthétiser des circuits asynchrones QDI représentés par les réseaux de Petri conformes à la spécification DTL. Les paramètres sont pris en compte comme les options de la commande. Les valeurs par défaut sont prédéfinies : le protocole WCHB et les buffers à la sortie du circuit.

6.7 Module optimiseur

L'optimisation repose sur les circuits QDI générés sous forme d'équations logiques. L'idée de l'optimisation implémentée dans l'outil TAST est que les circuits QDI à travers les transformations d'optimisation restent toujours dans le « cadre » QDI. Pour ce faire, les transformations d'optimisation adoptées doivent garantir les contraintes QDI des circuits.

La représentation en *minterms* à deux niveaux des circuits asynchrones QDI possède une assertion intéressante : « si tous les signaux de sortie du circuit sont des fonctions à deux étages Muller-Or de signaux d'entrée/sortie – garde exclusives – alors le circuit respecte les contraintes QDI (il n'existe alors plus de signal intermédiaire susceptible de poser des problèmes de stabilité) ».

Partant de ces idées, la première étape dans l'optimisation de circuits asynchrones QDI est de transformer toutes les fonctions multi-niveaux (comme les fonctions de garde ou les fonctions de sortie) afin de n'obtenir finalement que les différents *minterms*. Cela signifie que les circuits générés sont à deux niveaux : Muller-Or. Comme le montre les paragraphes précédents, un circuit asynchrone décrit par un programme CHP conforme aux règles DTL, sous la forme « consommation ; production », peut toujours être transformé en un circuit à deux niveaux Muller-Or.

Comme énoncé précédemment, les transformations d'optimisation ne doivent pas rendre les circuits QDI non-QDI. Elles doivent garantir que les aléas n'apparaissent pas et que les fourches sont isochrones. Ainsi, un algorithme d'optimisation de circuits QDI peut être un algorithme

efficace pour les circuits synchrones, renforcé de la vérification des contraintes QDI. Il s'agit de maîtriser les fourches isochrones dans l'algorithme. Par conséquent, nous partons d'un algorithme connu, efficace mais simple ([MIS]), auquel nous ajoutons la vérification des fourches isochrones. Si cette vérification est prouvée correcte et efficace, l'adoption d'un autre algorithme d'optimisation est tout à fait possible.

Nous ne présentons pas l'algorithme d'optimisation des circuits QDI dans ce manuscrit. Cet algorithme sera approfondi et implémenté dans le futur.

Pour manipuler les équations logiques du circuit, nous avons également développé une bibliothèque d'outils. Cette bibliothèque contient des fonctions principales permettant de mettre à « plat » un système d'équations afin de diminuer leur nombre d'étages (niveau). Elle permet aussi de trouver un facteur d'un système d'équations avec le nombre maximal de littéraux gagné, ce qui sert pour l'optimisation.

6.8 Module gestionnaire du réseau de portes (*netlist*)

Ce module génère les réseaux de portes du circuit à partir des équations logiques optimisées et « projetées » sur une technologie spécifique. En fait, la génération des réseaux de portes du circuit peut être réalisée à tout moment des que les équations logiques sont créées. Il s'agit de générer les réseaux de portes de circuits avant ou après l'optimisation et la projection technologique.

Les réseaux de portes des circuits asynchrones dans l'environnement TAST sont engendrés sous formes les netlists VHDL. Ces fichiers sont prêts à la simulation après synthèse par les outils commerciaux (ModelSim de Mentor ou VSS de Synopsys). Les réseaux de portes générés comprennent les réseaux de portes de chaque processus CHP (la commande « *writenetlist* ») et un réseau de niveau le plus haut contenant des composants instanciés (« *writetop* »). Il s'agit de décrire le circuit asynchrone de façon hiérarchique : chaque processus CHP correspond à un composant VHDL (entité, architecture) et un programme CHP composé de processus correspond à une *netlist* de composants instanciés. Un exemple de réseaux de portes est donné Figure 117.

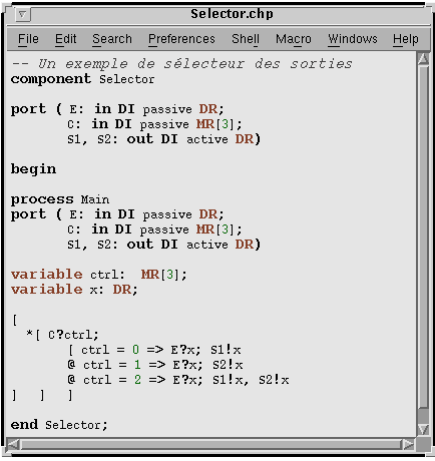
6.9 Module générateur de rapports

Les rapports créés par le générateur de rapports concernent les traitements successifs réalisés tout au long de l'utilisation de l'environnement. Ils permettent de mieux visualiser la synthèse réalisée afin de traquer les erreurs et les corriger par la suite. De plus, les commandes utilisées dans une session de travail sont enregistrées dans un fichier script, ce qui peut être utile à l'avenir.

Le générateur de rapports génère également des rapports de la structure hiérarchique du programme CHP initial et le bilan statistique sur l'utilisation des cellules dans chaque processus et dans le circuit global. Ces rapports, générés en appelant la commande « *countgate* », servent à estimer la surface du circuit implémenté.

6.10 Une session typique de travail

Nous illustrons ici l'environnement de travail TAST avec un exemple de sélecteur. Cet exemple provient du document édité pour le tutorial sur l'outil TAST à l'occasion de la conférence ASYNC'02 et de l'école d'été ACiD'02 ([DINH02]). Il décrit en langage CHP un sélecteur (Figure 116) dont le fonctionnement est le suivant. Selon la valeur du canal de contrôle C, la valeur lue du canal d'entrée E est propagée vers un ou plusieurs canaux de sortie.



```

Selector.chp
File Edit Search Preferences Shell Macro Windows Help
-- Un exemple de sélecteur des sorties
component Selector
port ( E: in DI passive DR;
      C: in DI passive MR[3];
      S1, S2: out DI active DR)
begin
process Main
port ( E: in DI passive DR;
      C: in DI passive MR[3];
      S1, S2: out DI active DR)

variable ctrl: MR[3];
variable x: DR;

[
  *| C?ctrl;
  | ctrl = 0 => E?x; S1!x
  @ ctrl = 1 => E?x; S2!x
  @ ctrl = 2 => E?x; S1!x, S2!x
] ] ]
end Selector;

```

Figure 116 Exemple d'un sélecteur

\$ tast	# démarrer l'environnement TAST
TAST - TIMA Asynchronous circuits Synthesis Tools	
By Anh Vu DINH DUC	
TAST > commands	# lister les commandes disponibles
ReadChp	-> read a CHP file
WriteChp	-> rewrite a CHP
Chp2Pn	-> convert CHP to PN
CheckDTL	-> check the DTL specification on the PN
WritePn	-> rewrite a PN
SynthesizeQDI	-> synthesize a circuit QDI from its PN
Elim	-> eliminate duplicated cells in the circuit netlist
CountGate	-> report on the gates used
WriteNetlist	-> write circuit's netlist
WriteTop	-> write circuit's top-level netlist
WriteAll	-> write circuit's netlists (including top-level netlist) as one file
ReadEqn	-> read an equation file (EQN format)
WriteEqn	-> write equations
CollapseEqn	-> collapse equations
Help	-> display the help information
?	-> synonym for 'Help'
History	-> history occupation
Script	-> execution of a script
Commands	-> list of existing commands
Time	-> CPU time information
Alias	-> Alias
Falias	-> get Alias from file
Unalias	-> remove an alias
FreeSymbol	-> free entities
Status	-> list of the defined entities
Memory	-> memory occupation
List	-> list files in directory
ls	-> synonym for 'list'
cd	-> change directory
pwd	-> print the current working directory
view	-> view the contents of a file
!	-> execution of a system command (subprocess)

```

Quit                -> quit TAST

*** Execution script will be directed to the "/tmp/script0002".

TAST > pwd                                # voir le répertoire courant
/tima/dinhduc/tutorial

TAST > ls                                # chercher le circuit
bin                examples        results        scripts

TAST > ls examples
Counter           Filter           Selector

TAST > cd examples/Selector
Current Dir is : /tima/dinhduc/tutorial/examples/Selector

TAST > help readchp                      # montrer la syntaxe de « readchp »

READCHP                TAST User's Manual

NAME
    READCHP - input of CHP source

SYNOPSIS
    readchp [OPTIONS] CHP_file

DESCRIPTION
    readchp loads to the memory a CHP source file which describes an asynchronous circuit.
    Some primary verification of context are taken while the CHP file is parsed.

OPTIONS
    -name Symbol_Name
        + Description          : used to symbolize the loaded CHP in the memory
        + Accepted value       : string
        + Default value        : "blank" name

SEE ALSO
    writetchp, chp2pn

TAST > readchp Selector.chp                # lire la description du circuit en CHP
Parsing syntax and lexicon ... ok.
Renaming ..... ok
Processing type ..... ok
Verifying context ..... ok

TAST > chp2pn                            # compiler le CHP en réseaux de Petri et graphes de flot de données
Compiling PN-DFG ..... ok

TAST > checkDTL                          # vérifier les règles DTL

```

Checking DTL specification ok

synthétiser le circuit avec le protocole WCHB et la bibliothèque propriété à TAST

TAST > synthesizeQDI -HSE WCHB -lib TAST

WARNING: Buffer is put at the output par default.

Processing component SELECTOR ...

Processing MAIN ok

TAST > countgate

Analyser la netlist générée

INFO: Statistics on used gates in the netlist of circuit "SELECTOR"

Process "MAIN":

Gate TAST_AND3	: 2
Gate TAST_OR2	: 4
Gate TAST_NOR2	: 4
Gate TAST_MULLER2	: 10
Gate TAST_MULLER2_R	: 4
Gate TAST_MULLER3_R	: 4

TOTAL for the circuit "SELECTOR":

Gate TAST_AND3	: 2
Gate TAST_OR2	: 4
Gate TAST_NOR2	: 4
Gate TAST_MULLER2	: 10
Gate TAST_MULLER2_R	: 4
Gate TAST_MULLER3_R	: 4

TAST > elim

réaliser l'optimisation

Component SELECTOR:

8 duplicated gates are removed

Total: 8 duplicated gates are removed

TAST > countgate

Analyser le résultat

INFO: Statistics on the used gates in the netlist of the circuit "SELECTOR"

Process "MAIN":

Gate TAST_AND3	: 1
Gate TAST_OR2	: 4
Gate TAST_NOR2	: 3
Gate TAST_MULLER2	: 6
Gate TAST_MULLER2_R	: 4
Gate TAST_MULLER3_R	: 2

TOTAL for the circuit "SELECTOR":

Gate TAST_AND3	: 1
Gate TAST_OR2	: 4
Gate TAST_NOR2	: 3
Gate TAST_MULLER2	: 6
Gate TAST_MULLER2_R	: 4

Gate TAST_MULLER3_R : 2

```

TAST > writenetlist -of Selector.vhd                                # Enregistrer le réseau de portes
Writing ..... ok

TAST > lxemacs Selector.vhd &      # Montrer le fichier VHDL en utilisant l'éditeur de texte XEmacs
                                     # Lancer les commandes ci-dessus en utilisant le script

TAST > script /tmp/script0002 -of Selector.log
[ ... le même résultat mais sans l'interaction ... ]

TAST > quit
Bye
$
    
```

La netlist de portes généré par TAST avec la bibliothèque de cellules standard TAST ([RIGA02]) est illustrée ci-dessous.

```

-----
-- Title       : MAIN
-- Project      : SELECTOR
-----

-- File        : Selector.vhd
-- Author       : dinhduc
-- Company      :
-- Created      : 2003-01-04
-- Last update  : 2003-01-04
-- Platform     : SunOS
-----

-- Description: netlist of the process MAIN

-- Generated from the CHP source "Selector.chp" by TAST
-- Copyright (c) 2002
-----

-- Revisions :
-- Date       Version   Author    Description
-- 2002-12-04  1.0      dinhduc  Created
-----

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

LIBRARY TAST_COMPONENTS;
USE TAST_COMPONENTS.tast_components.all;

ENTITY MAIN IS
PORT
(
    Resetb: in std_ulogic;
    E: in std_ulogic_vector (1 downto 0);
    E_ack: out std_ulogic;
    C: in std_ulogic_vector (2 downto 0);
    C_ack: out std_ulogic;
    S1: out std_ulogic_vector (1 downto 0);
    S1_ack: in std_ulogic;
    S2: out std_ulogic_vector (1 downto 0);
    S2_ack: in std_ulogic);
-- attribute MR_base of E: signal is 2;
-- attribute MR_length of E: signal is 1;
-- attribute MR_base of C: signal is 3;
-- attribute MR_length of C: signal is 1;
-- attribute MR_base of S1: signal is 2;
-- attribute MR_length of S1: signal is 1;
-- attribute MR_base of S2: signal is 2;
-- attribute MR_length of S2: signal is 1;
END MAIN;

ARCHITECTURE MAIN_arch OF MAIN IS

    SIGNAL GND: std_ulogic:= '0';
    SIGNAL VCC: std_ulogic:= '1';
    SIGNAL Net_17, Net_18, Net_19, Net_20, Net_21: std_ulogic;
    SIGNAL Net_22, Net_23, Net_24, Net_25, Net_26: std_ulogic;
    SIGNAL Net_27, Net_28, Net_31, Net_32, Net_33: std_ulogic;
    SIGNAL Net_39, Net_41, Net_42, Net_43, Net_44: std_ulogic;

    BEGIN -- MAIN_arch

        Inst_Name_17: TAST_MULLER2 port map (Net_17, E(1), C(0));
        Inst_Name_18: TAST_MULLER2 port map (Net_18, E(0), C(0));
        Inst_Name_19: TAST_MULLER2_R port map (Resetb, Net_19, Net_17, S1_ack);
        Inst_Name_20: TAST_MULLER2_R port map (Resetb, Net_20, Net_18, S1_ack);
        Inst_Name_21: TAST_NOR2 port map (Net_21, Net_19, Net_20);
        Inst_Name_22: TAST_MULLER2 port map (Net_22, E(1), C(1));
        Inst_Name_23: TAST_MULLER2 port map (Net_23, E(0), C(1));
        Inst_Name_24: TAST_MULLER2_R port map (Resetb, Net_24, Net_22, S2_ack);
        Inst_Name_25: TAST_MULLER2_R port map (Resetb, Net_25, Net_23, S2_ack);
        Inst_Name_26: TAST_NOR2 port map (Net_26, Net_24, Net_25);
        Inst_Name_27: TAST_MULLER2 port map (Net_27, E(1), C(2));
        Inst_Name_28: TAST_MULLER2 port map (Net_28, E(0), C(2));
        Inst_Name_31: TAST_MULLER3_R port map (Resetb, Net_31, Net_27, S1_ack, S2_ack);
        Inst_Name_32: TAST_MULLER3_R port map (Resetb, Net_32, Net_28, S1_ack, S2_ack);
        Inst_Name_33: TAST_NOR2 port map (Net_33, Net_31, Net_32);
        Inst_Name_39: TAST_AND3 port map (Net_39, Net_21, Net_26, Net_33);
        E_ack <= Net_39;
        C_ack <= Net_39;
        Inst_Name_41: TAST_OR2 port map (Net_41, Net_19, Net_31);
        S1(1) <= Net_41;
        Inst_Name_42: TAST_OR2 port map (Net_42, Net_20, Net_32);
        S1(0) <= Net_42;
        Inst_Name_43: TAST_OR2 port map (Net_43, Net_24, Net_31);
        S2(1) <= Net_43;
        Inst_Name_44: TAST_OR2 port map (Net_44, Net_25, Net_32);
        S2(0) <= Net_44;

    END MAIN_arch;
    
```

Figure 117 Netlist VHDL de portes

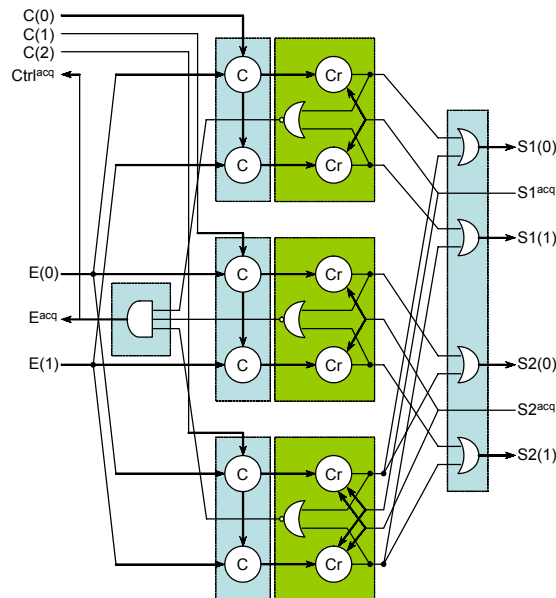


Figure 118 Schéma de portes du circuit Sélectionneur

La Figure 118 donne le schéma de porte du circuit obtenu.

6.11 Conclusion

Aujourd'hui, le développement d'outils de conception automatiques pour les circuits asynchrones constitue un verrou très important et difficile à lever. Afin de faciliter l'accessibilité des circuits asynchrones au monde industriel et ainsi de contribuer à ce que leur potentiel devienne en réalité, un environnement de conception de circuits asynchrones de type QDI a été développé. Il est principalement composé d'un compilateur pour le *front-end* et d'un synthétiseur pour le *back-end*. Un format intermédiaire sert à faire le lien entre ces deux parties. Il est basé sur un réseau de Petri de haut niveau qui tient lieu de sémantique opérationnelle pour le sous-ensemble déterministe du langage de description. Le compilateur prend en entrée des descriptions de haut niveau de circuits asynchrones en langage CHP. Il transforme ces descriptions en un réseau de Petri. Le synthétiseur s'appuie sur ce format intermédiaire pour générer des circuits QDI au niveau portes.

Cet outil de conception complet offre un environnement agréable permettant au concepteur de décrire aisément des circuits asynchrones à haut niveau, puis de les synthétiser de façon automatique en réseaux de portes grâce à une bibliothèque de cellules spécifique à la technologie. La correction fonctionnelle des circuits QDI générés reste tout de même à prouver formellement, même si l'utilisation d'outils de vérification automatiques donne des résultats convaincant ([BORR03]).

Le logiciel TAST a fait l'objet d'une démonstration à DATE 2002 (Paris) et de tutoriaux à la conférence ASYNC 2002 (Manchester) et à l'école d'été ACiD 2002 (Grenoble).

Il sera par ailleurs utilisé à des fins de formation dans les domaines de l'informatique et de la conception de circuits asynchrones.

Conclusion

L'objectif de cette thèse est de fournir des méthodologies et des outils qui faciliteront et accéléreront la conception des systèmes asynchrones. Ainsi, on espère favoriser leur acceptation et leur diffusion dans l'industrie. Pour ce faire, un flot de conception « descendant » est proposé. Ce flot part d'une spécification de haut niveau du système asynchrone à générer, et grâce à des outils automatiques de raffinement, se propose de générer le circuit niveau porte correspondant. Ce flot offre aussi la possibilité de simuler fonctionnellement le système.

De façon générale, les méthodologies de conception des circuits asynchrones diffèrent par leur méthode de spécification et leur méthode de synthèse. Un état de l'art sur les méthodes de spécification et les méthodes de synthèse est présenté, permettant d'identifier leurs points forts et leurs points faibles. Dans cette vision, les méthodologies actuelles de la conception de circuits asynchrones ont été étudiées.

La méthode de spécification proposée dans cette thèse est basée sur un langage de description de haut niveau : le CHP. Initialement proposé par l'université Caltech, ce langage est étendu afin de répondre à nos besoins vis-à-vis de la simulation et de la synthèse. En particulier, il offre :

- les types de données multi-rails signés et non signés
- la description hiérarchique et modulaire des circuits asynchrones
- la possibilité de déterminer la probabilité d'occurrence d'un choix
- une programmation aisée
- une progression continue du fonctionnel au structurel
- une lisibilité conjointe de l'algorithme et de la structure
- une transposition directe à l'implémentation en circuits asynchrones
- la prise en compte simultanée des spécifications fonctionnelles et des spécifications matérielles.

En vue de séparer le *back-end* et le *front-end* de la méthode de synthèse des circuits asynchrones, une représentation intermédiaire du circuit est nécessaire. Elle est basée sur les réseaux de Petri et les graphes de flot de données. Cette combinaison permet de représenter des circuits asynchrones complexes sous une forme compacte formelle.

La méthode de synthèse proposée est basée sur la spécification DTL. Celle-ci fournit un ensemble de règles qui limite de façon intrinsèque les éléments de mémorisation des descriptions de circuits. Les règles DTL garantissent que les descriptions – écrites en CHP – sont synthétisables vers des circuits asynchrones. Les transformations sur des spécifications de circuits sont proposées. Elles permettent d'extraire les éléments de mémorisation en décomposant des spécifications non DTL en plusieurs processus afin de les rendre conformes à la spécification DTL.

Après une étude sur les techniques actuelles de pipeline, le modèle de circuits cibles a été adopté. Il repose sur les pipelines QDI (séquentiel, WCHB, PCHB et PCFB), offrant une

implémentation de circuits robuste et efficace en vitesse. Ce modèle a été utilisé pour implémenter des structures pipelines non linéaires (fourche, convergence, multiplexage, démultiplexage).

La génération de circuits QDI est basée sur les équations de dépendances générées à partir de la forme intermédiaire. Les équations de dépendances expriment les dépendances de contrôle et de données des sorties en fonction des entrées du circuit. Elles sont indépendantes du protocole de communication et de la technologie cible, permettant de cibler les circuits asynchrones QDI mais aussi d'autres types de circuits asynchrones (micropipeline, SI, ...). Deux algorithmes de génération des équations de dépendances ont été introduits, permettant de générer systématiquement les équations de dépendances des sorties de données ainsi que les signaux d'acquittement des circuits asynchrones (codage insensible aux délais). Cela est réalisé à deux niveaux d'abstraction différents : au niveau des digits et au niveau des signaux. Les équations de dépendances générées peuvent contenir des redondances. Ces dernières sont éliminées grâce à des règles d'optimisation.

Les implémentations QDI au niveau porte des circuits sont générées en développant les équations de dépendances avec un protocole de communication. L'implémentation QDI de nombreux opérateurs tels que le test de validité d'un canal, l'initialisation d'un circuit, les opérateurs de garde, les opérateurs de calcul, est définie. Ces implémentations peuvent être réalisées avec de la logique à deux niveaux ou de la logique multi-niveaux. Une tâche extrêmement difficile lors de la synthèse, notamment la synthèse de circuits asynchrones, est l'optimisation. L'optimisation de circuits asynchrones est soigneusement réalisée afin de respecter les contraintes QDI imposées (éviter les aléas et garder les fourches isochrones). Il est proposé de façon pragmatique les optimisations évidentes ainsi que les algorithmes d'optimisation et de décomposition généraux pour les circuits générés, tout en respectant (et vérifiant) les contraintes QDI.

Les circuits QDI optimisés – sous forme d'une description VHDL de niveau porte – sont « projetés » sur une bibliothèque de cellules spécifique à la technologie. Cette bibliothèque comporte les portes élémentaires standard et les portes de Muller (circuits précaractérisés ou FPGAs). Cependant, la *netlist* de circuits générée et optimisée peut aussi cibler une technologie spécifique pour l'asynchrone.

Enfin, l'outil de conception automatique de circuits asynchrones TAST a été développé pour prototyper la méthode de conception proposée. Il est en fait un environnement de conception composé principalement d'un compilateur et d'un synthétiseur offrant la possibilité de cibler plusieurs formats de sortie (description comportementale en VHDL, langage C, description structurelle au niveau porte en VHDL) à partir de descriptions de haut niveau décrites dans un langage proche du CHP. Cet outil a été utilisé pour la conception plusieurs applications concrètes (filtres, DES, ...). Il a aussi servi comme outil de « travaux pratiques » à la conférence ASYNC 2002 et à l'école d'été ACiD 2002. Il sera aussi utilisé pour des formations d'étudiants à la conception de circuits asynchrones.

L'objectif initial de cette thèse a été atteint et représente une contribution sur les méthodologies et l'automatisation de la conception des circuits asynchrones.

TAST est un projet très ambitieux et certaines parties sont à compléter avant de le diffuser largement dans la recherche et l'industrie. Suite aux travaux présentés dans cette thèse, plusieurs travaux sont en cours pour améliorer l'outil. Certains d'entre eux sont énumérés ici :

- validation de la tâche d'optimisation des circuits générés,
- réalisation d'un outil de *back-end* pour synthétiser les circuits asynchrones de type micropipeline,

- réalisation d'un simulateur de réseau de Petri avec mesures statistiques, permettant de déterminer la probabilité d'occurrence d'une place, ce qui est utile pour les analyses ultérieures (consommation, émission électromagnétique, *etc.*),
- finition et optimisation du générateur d'opérateurs arithmétiques,
- génération d'éléments de bibliothèque spécifique à l'asynchrone et implémentation de la fonction de projection technologique,
- description des circuits par un autre langage (VHDL, SystemC, ...).

Perspectives

Le principal enjeu au niveau du flot de conception réside aujourd'hui dans la définition et le développement d'un outil complet de simulation et de synthèse pour générer automatiquement divers styles de circuits asynchrones répondant à diverses contraintes.

Le premier thème consiste à utiliser des places annotées en probabilité afin de guider la synthèse pour la faible consommation, le faible rayonnement électromagnétique et la sécurité.

Le deuxième concerne l'étude et le développement d'algorithmes de synthèse de circuits asynchrones contraints par des critères de sécurité, de faible consommation et de faible émission électromagnétique.

Une autre perspective est l'étude et l'identification des fourches isochrones dans les circuits générés afin de concevoir un outil de *back-end* qui permet de générer automatiquement les contraintes de routage spécifiques aux circuits FPGAs.

Enfin, cet outil peut être étendu de façon à synthétiser automatiquement des arbitres. Pour ce faire, il faut ajouter dans l'outil un module de reconnaissance des choix indéterministes qui est susceptible de les implémenter en utilisant des cellules particulières (synchroniseur, bloc d'exclusion mutuelle).

Bibliographie

- [AKEL96] V. Akella, N. H. Vaidya, and R. Redinbo. Limitations of VLSI implementation of delay-insensitive codes. In *International Symposium on Fault-Tolerant Computing (FTCS)*, June 1996
- [BARD00] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000
- [BARD00b] A. Bardsley and D. A. Edwards. Synthesising an asynchronous DMA controller with Balsa. *Journal of Systems Architecture*, 46:1309-1319, 2000
- [BARD97] Bardsley A., Edwards D., Compiling the Language Balsa to Delay-Insensitive Hardware, in Kloos C. D. and Cerny E., Ed., *Hardware Description Languages and their Applications (CHDL)*, (1997), pp. 89-91.
- [BEER92] P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581-587. IEEE Computer Society Press, November 1992
- [BEER96] P. A. Beerel, K. Y. Yun, and W. C. Chou. Optimizing average-case delay in technology mapping of burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*. March 1996
- [BERK88] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150-166, Rye Brook, New York, 1988. IEEE Computer Society Press
- [BERK91] Berkel K. V., Kessels J., Roncken M., Saeijs R. and Schalijs F., The VLSI-programming language Tangram and its translation into handshake circuits, in *Proc. European Conference on Design Automation (EDAC)* (1991), pp 384-389.
- [BERK92] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103-128, June 1992
- [BERK93] Berkel K. V., Handshake circuits: an asynchronous architecture for VLSI Programming, *vol. 5 of International Series on Parallel Computation*, Cambridge University Press, 1993.
- [BERK94] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalijs. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429-1439, December 1994
- [BERK95] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalijs, and Rik van de Wiel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *Asynchronous Design*

Methodologies, pages 72-79. IEEE Computer Society Press, May 1995

- [BERK96] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 122-133. IEEE Computer Society Press, March 1996
- [BERK98] Van Berkel C., Saeijs R., Compilation of communicating processes into delay-insensitive circuits, in *International Conference on Computer Design (ICCD)*, (1998), IEEE Computer Society Press.
- [BERK99] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223-233, February 1999
- [BLUN00] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 84-92. IEEE Computer Society Press, April 2000
- [BORR03] D. Borriore, M. Boubekeur, E. Dumitrescu, Marc Renaudin, J.-B. Rigaud, and A. Sirianni, An Approach to the Introduction of Formal Validation in an Asynchronous Circuit Design Flow, *Proc. of the 26th Annual Hawaiï International Conference on System Sciences*, Big Island, Hawaiï, 6-9 January, 2003
- [BRED72] 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
- [BRED75] J. G. Bredeson. Synthesis of Multiple-Input Change Hazard-Free Combinational Switching Circuits without Feedback. *International Journal of Electronics (GB)*, 39(6): 615-624. December 1975
- [BRUN89] Brunvand E., Sproull R. F., Translating concurrent programs into delay-insensitive circuits, in *International Conference on Computer-Aided Design (ICCAD)*, (1989), IEEE Computer Society Press.
- [BRUN91] Brunvand E., Translating concurrent communicating programs into asynchronous circuits, *PhD thesis, Carnegie Mellon University*, 1991.
- [BRZO89] J. A. Brzozowski and J. C. Ebergen. Recent developments in the design of asynchronous circuits. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Fundamentals of Computation Theory (FCT)*, volume 380 of *Lecture Notes in Computer Science*, pages 78-94, Szeged, Hungary, 1989. Springer-Verlag
- [BURN88] Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. *Master's thesis*, California Institute of Technology, 1988
- [BURN88b] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Advanced Research in VLSI*, pages 35-50. MIT Press, 1988
- [BURN88c] Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. In G. J. Milne, editor, *The Fusion of Hardware Design and*

- Verification*, pages 99-116. Elsevier Science Publishers, 1988
- [BURN91] Steven M. Burns. Performance Analysis and Optimization of Asynchronous Circuits. *PhD thesis*, California Institute of Technology, 1991
- [BURN96] S. M. Burns. General condition for 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
- [BUSH96] M. E. Bush and M. B. Josephs. Some limitations to speed-independence in asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996
- [CHAN73] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421-422, April 1973
- [CHOU99] Wei-Chun Chou, Peter A. Beerel, and Kenneth Y. Yun. Average-case technology mapping of asynchronous burst-mode circuits. *IEEE Transactions on Computer-Aided Design*, 18(10):1418-1434, October 1999
- [CHU87] Chu T. A., Synthesis of self-timed VLSI Circuits from Graph-theoretic Specifications, *PhD Thesis, Massachusetts Institute of Technology*, 1987.
- [CLUS86] E. J. McCluskey. Logic Design Principles: with emphasis of testable semicustom circuits. *Prentice-Hall, Englewood Cliffs, NJ*, 1986
- [COAT01] William S. Coates, Jon K. Lexau, Ian W. Jones, Scott M. Fairbanks, and Ivan E. Sutherland. FLEETzero: An asynchronous switch fabric chip experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 173-182. IEEE Computer Society Press, March 2001
- [CORT02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Logic Synthesis of Asynchronous Controllers and Interfaces. *Springer-Verlag*, 2002
- [CORT97] 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
- [CORT97b] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, November 1997
- [DALL98] W. J. Dally, J. Poulton. Digital Systems Engineering. *Cambridge University Press, Cambridge, UK*, 1998
- [DAVI92] Ilana David, Ran Ginosar, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2-11, January 1992
- [DAVI93] Davis A., Coates B. and Stevens K., The PostOffice experience: Designing a large asynchronous chip, in *Proc. of the 26th Annual Hawaii International Conference*

- on System Science* (1993), IEEE Computer Science Press, pp. 409-418.
- [DAVI93b] A. Davis, B. Coates, and K. Stevens. 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
 - [DAVI97] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. *Technical Report UUCS-97-013*, Dept. of Computer Science, University of Utah, September 1997
 - [DEAN92] Mark E. Dean. STRiP: A Self-Timed RISC Processor Architecture. *PhD thesis, Stanford University*, 1992
 - [DIJK76] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, *Englewood Cliffs, N.J.* 1976
 - [DINH02] A.V. Dinh Duc et al., TAST, tutorial given at *the 8th IEEE Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, Manchester, UK, Apr. 8-11, 2002, ISRN: TIMA-RR--02/04/01-FR
 - [DINH02b] A.V. Dinh Duc et al., TAST CAD Tools, 2nd Asynchronous Circuit Design *Workshop of the European Commission's Fifth Framework Programme (ACID'02)*, Munich, Germany, 28-29 January 2002. Also exist as TIMA Technical Report ISRN: TIMA-RR--02/03/08-FR
 - [DINH02c] A.V. Dinh Duc, L. Fesquet and M. Renaudin, Synthesis of QDI asynchronous circuits from DTL-Style Petri Net, in *Proc. of 11th IEEE/ACM Intl. Workshop on Logic and Synthesis (IWLS)*, pp. 191-196, June 2002
 - [DINH02d] A.V. Dinh Duc et al., DTL: The Foundations of a General Design Methodology for Asynchronous Circuits, Technical report, TIMA, 2002
 - [EBER91] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107-119, 1991
 - [EICH65] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90-99, March 1965
 - [ENDE94] F.B. Endecott and S. B. Furber. Modelling and simulation of asynchronous systems using the LARD hardware description language. In *Proc. of the 12th European Simulation Multiconference*, Manchester, Society for Computer Simulation International, pages 39-43, June 1994
 - [FANT96] Karl M. Fant and Scott A. Brandt. NULL conventional 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
 - [FERR02] 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
 - [FRAG03] Joao Frago, Gilles Sicard, Marc Renaudin, Generalized 1-of-M QDI

- Asynchronous Adders, published to 3rd ACiD-WG Workshop, 27-28 January 2003, Crete, Greece
- [FUHR99] 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 TR CUCS-020-99*, Columbia University, NY, July 1999
- [FURB96] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, March 1996
- [FURB96b] S. B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247-253, June 1996
- [FURB99] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243-256, February 1999
- [GAGE98] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 96-107, 1998
- [GIOV94] Giovanni de Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Press, 1994
- [HAUC95] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69-93, January 1995
- [HOAR78] Hoare C. A. R., *Communicating Sequential Processes*, Communications of the ACM 21, vol. 8, (April, 1978), pp. 666-677
- [HUFF54] Huffman D. A., *The Synthesis of Sequential switching Circuits*, J. Franklin Institute (March, April 1954).
- [JERR02] A.A. Jerraya, C. Landrault, E. Martin, M. Renaudin, K. Torrki. Conception logique et physique des systèmes monopuces, *chapitre 5*, Lavoisier, 2002.
- [JOSE93] M. B. Josephs and J. T. Udding. An overview of DI algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proc. Hawaii International Conf. System Sciences*, volume I, pages 329-338. IEEE Computer Society Press, January 1993
- [JOSE99] Mark B. Josephs, Steven M. Nowick, and C. H. (Kees) van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234-242, February 1999
- [KESS00] Joep Kessels, Torsten Kramer, Gerrit den Besten, Ad Peeters, and 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
- [KESS00b] Joep Kessels, Torsten 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
- [KESS99] Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE*, 87(2):257-267, February 1999
- [KISH94] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. Concurrent Hardware: The Theory and Practice of Self-Timed Design. *Series in Parallel Computing*. John Wiley & Sons, 1994
- [KOND02] Alex Kondratyev, Lawrence Neukom, Oriol Roig, Alexander Taubin, and Karl Fant. Checking delay-insensitivity: 104 gates and beyond. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 149-157, April 2002
- [KRUP98] H. Krupnova, A.V. Dinh Duc, G. Saucier, Real Time Prototyping and A Case Study, In *Proc. of the IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 13-18, Belgium, June 1998.
- [KRUP98b] H. Krupnova, A.V. Dinh Duc, G. Saucier, A Knowledge-Based System for Prototyping on FPGAs, In *Proc. of the International Workshop on Field Programmable Logic and Applications (FPL)*, pages 89-98, Tallinn, August 1998.
- [LIGT00] 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
- [LINE98] A. M. Lines. Pipelined Asynchronous Circuits. *M.Sc. Thesis, California Institute of Technology*, June 1995, revised 1998
- [MARS94] Alan Marshall, Bill Coates, and Polly Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8-21, 1994.
- [MART86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226-234, 1986
- [MART90] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263-278. MIT Press, 1990
- [MART90b] 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
- [MART92] Alain J. Martin. Tomorrow's digital hardware will be asynchronous and verified. In J. van Leeuwen, editor, *Information Processing 92, Vol. I: Algorithms, Software, Architecture*, volume A-12 of *IFIP Transactions*, pages 684-695. Elsevier Science Publishers, 1992
- [MART97] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS

- R3000 microprocessor. In *Advanced Research in VLSI*, pages 164-181, September 1997
- [MAY90] D. May. Compiling OCCAM into Silicon. *Developments in Concurrency and Communication*. Edited by C.A.R. Hoare, Addison Wesley, pages 87-106, 1990
- [MENG89] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185-1205, November 1989
- [MENG91] Teresa H.-Y. Meng. Synchronization Design for Digital Systems. *Kluwer Academic Publishers*, 1991
- [MILL65] R. E. Miller. Sequential Circuits and Machines, volume 2 of Switching Theory. *John Wiley & Sons*, 1965
- [MIS] 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, No 6, November 1987
- [MOLN85] Molnar C. E., Fang T. P. and Rosenberger F. U., Synthesis of delay-insensitive modules, in 1985 *Chapel Hill Conference on VLSI*, (1985), H. Fuchs, Ed., Computer Science Press, pp. 67-86.
- [MULL59] Muller D. E. and Bartky W. S., A Theory of Asynchronous Circuits, in *Proc. of an International Symposium of the Theory of Switching* (1959), pp. 204-243
- [MYER01] Chris Myers. Asynchronous Circuit Design. *John Wiley & Sons*, 2001
- [MYER92] 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
- [NIEL94] Christian D. Nielsen. Evaluation of function blocks for asynchronous design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 454-459. IEEE Computer Society Press, September 1994
- [NIEL99] 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
- [NOWI93] Steven M. Nowick. Automatic Synthesis of Burst-Mode Asynchronous Controllers. *PhD thesis, Stanford University*, Department of Computer Science, 1993
- [NYST01] Mika Nyström, Asynchronous Pulse Logic. *PhD thesis, California Institute of Technology*, May 2001.
- [OZDA02] Recep O. Ozdag and Peter A. Beerel. High-speed QDI asynchronous pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 13-22, April 2002
- [OZDA02b] Recep O. Ozdag, Montek Singh, Peter A. Beerel, and Steven M. Nowick. High-speed non-linear asynchronous pipelines. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1000-10007, March 2002

- [PEET90] Ad Peeters. Decomposition of delay-insensitive circuits. Computing Science Notes 90/04, Dept. of Math. and C.S., Eindhoven Univ. of Technology, April 1990
- [PETE81] J.L. Peterson. Petri Net Theory and the Modeling of Systems. *Prentice-Hall, Englewood Cliffs, NJ*, 1981
- [PETR62] C.A. Petri. Kommunikation mit automaten. *PhD Thesis, Bonn, Institut für Instrumentelle Mathematik*, 1962
- [PIGU98] C. Piguet and J. Zahnd. Design of speed-independent CMOS cells from signal transition graphs. In Anne-Marie Trullemans-Anckaert and Jens Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 357-366, October 1998
- [RENA00] M. Renaudin. Asynchronous Circuits and Systems: A Promising Design Alternative. In *MicroElectronic Engineering*. Volume 54(1-2): 133-149, December 2000
- [RENA96] Marc Renaudin, Bachar El Hassan, and Alain Guyot. New asynchronous pipeline scheme: Application to the design of a self-timed ring divider. *IEEE Journal of Solid-State Circuits*, 31(7):1001-1013, July 1996
- [RENA98] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 22-31, 1998
- [RENA99] M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 135-144, April 1999
- [RIGA02] J.B. Rigaud, Spécification de bibliothèques pour la synthèse de circuits asynchrones, *PhD thesis (in French)*, INP of Grenoble, 2002
- [ROSE85] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199-207, Torino, Italy, July 1985. IEEE Computer Society Press
- [SCHU00] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. “Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz”, in *ISSCC 2000*, pp. 292–293
- [SEIT70] Charles L. Seitz. Asynchronous machines exhibiting concurrency, 1970. *Record of the Project MAC Concurrent Parallel Computation*
- [SEIT80] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems, chapter 7*. Addison-Wesley, 1980
- [SENT92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. *Technical report, U.C. Berkeley*, May 1992
- [SIEG93] P. Siegel, G. De Micheli, and D. Dill. Automatic technology mapping for

- generalized fundamental-mode asynchronous designs. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 61-67, June 1993
- [SING00] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 198-209. IEEE Computer Society Press, April 2000
- [SING00b] Montek Singh and Steven M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 111-118. IEEE Computer Society Press, April 2000
- [SING01] Montek Singh and Steven M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. International Conf. Computer Design (ICCD)*, pages 9-17, November 2001
- [SPAR01] Jens Sparsø and Steve Furber, editors. Principles of Asynchronous Circuit Design: A Systems Perspective. *Kluwer Academic Publishers*, 2001
- [SUTH01] Ivan Sutherland and Scott 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, March 2001
- [SUTH89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720-738, June 1989
- [TERA99] 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
- [TUGS02] Sunan Tugsinavisut and Peter A. Beerel. Control circuit templates for asynchronous bundled-data pipelines. In *Proc. Design, Automation and Test in Europe (DATE)*, page 1098, March 2002
- [UDDI86] Jan Tjijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197-204, 1986
- [UNGE69] S. H. Unger. Asynchronous Sequential Switching Circuits. *Wiley-Interscience, John Wiley & Sons Inc.*, New York, 1969
- [UNGE71] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437-1444, december 1971
- [VAR90] Varshavsky V. I., Ed. Self-timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems, *Kluwer Academic Publishers*, Dordrecht, The Netherlands, 1990
- [VIVE01] P. Vivet. Une Méthodologie de Conception de Circuits Intégrés Quasi-Insensibles aux Délais: Application à l'Etude et à la Réalisation d'un Processeur RISC 16-bit Asynchrone. *PhD thesis (in French)*, INP of Grenoble, 2001
- [WAKE94] John F. Wakerly, Digital Design: Principles and Practices, *Second Edition*, Prentice, 1994
- [WESL67] 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

- [WILL91] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651-1661, November 1991
- [YAKO00] A. Yakovlev, L. Gomes and L. Lavagno. Hardware Design and Petri Nets. *Kluwer Academic Publishers*, 2000
- [YKMA94] 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
- [YUN92] Yun K. Y., Dill D. L. and Nowick S. M., Synthesis of 3D asynchronous state machines, in *Proc. International Conference on Computer Design (ICCD)*, (1992), IEEE Computer Society Press, pp. 346-350.
- [YUN96] 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 (ASYNC)*. March 1996
- [YUN98] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI Systems*, 6(4):643-655, December 1998
- [ZHEN98] Zheng H., Specification and compilation of timed systems, *Master thesis, University of Utah*, 1998.

Annexe

A) Système de numération

1) Nombre non signé

Un entier N est représenté en précision arbitraire par " $d_{L-1}.d_{L-2}...d_0$ "[B][L], ce qui signifie un vecteur de L digits représentés dans la base B . La longueur L et la base B sont les nombres naturels. La valeur de ce nombre d'entier est donnée par

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

Exemple : "2.0.0.3"[10] est le nombre décimal 2003 et "0.1.2"[3] est égal $0*3^2+1*3^1+2*3^0$, ce fait 5 dans la base décimale.

Pour éviter cette notion dans l'écriture des nombres dans les bases usuelles telles que binaire ou décimale, les notions standard (comme celles dans le langage VHDL) est ainsi supportées. Par exemple, "5" ou même 5 est équivalent avec "5"[10], '1' est représenté pour "1"[2].

2) Nombre signé

Un nombre signé N peut être représenté en une formule générale suivante :

$$N = \begin{cases} \sum_{i=0}^{L-1} d_i B^i & \text{if } d_{L-1} < \frac{B}{2} \\ (d_{L-1} - B)B^{L-1} + \sum_{i=0}^{L-2} d_i B^i & \text{if } d_{L-1} \geq \frac{B}{2} \end{cases}$$

Dans cette formule, B est la base et L le nombre de digits (soit la longueur) du nombre N . Cette formule peut être réécrite comme suite :

$$N = \begin{cases} d_{L-1}B^{L-1} + \sum_{i=0}^{L-1} d_i B^i & \text{if } d_{L-1} < \frac{B}{2} \\ (d_{L-1} - B)B^{L-1} + \sum_{i=0}^{L-2} d_i B^i & \text{if } d_{L-1} \geq \frac{B}{2} \end{cases}$$

Cette formule exprime que le nombre N est composé de L digits et que uniquement le digit le plus significatif (à gauche) porte le signe. Les autres digits se comportent comme les digits non signés. Il est à noter que dans cette forme de représentation, on n'utilise pas les signes (plus ou moins) pour spécifier un nombre. A titre d'exemple, le tableau ci-dessous illustre le digit de signe (le digit le plus significatif) dans la base décimale ($L=1$ et $B=10$)

Digit	Valeur
"0"	0
"1"	1
"2"	2
"3"	3

“4”	4
“5”	-5
“6”	-4
“7”	-3
“8”	-2
“9”	-1

B) Syntaxe et sémantique du langage CHP

L'objectif de cette annexe est de présenter la syntaxe et la sémantique du langage CHP telles que nous les avons définies et implémentées dans l'outil TAST. Ce langage présenté – appelé CHP évolutif – est une version extensive de celle initiée par Alain Martin [MART90b]. L'accent sera donné en particulier sur les points nouveaux apportés au langage.

1) Types de données

a) Type de données non signé

- **MR[B]** : type de Multi-Rails dans une base B. Au niveau d'implémentation de matérielle, un canal de ce type comporte B fils (rail) conformes au codage insensible aux délais « 1-parmi-B ». Exemple :

variable $b : \text{MR}[B] ;$ -- déclaration d'une variable b avec le type MR
 $b := \text{“x”}[B] ;$ -- affectation de b ($0 \leq x \leq B - 1$)

- **MR[B][L]** : vecteur de L éléments de type Multi-Rails dans une base B. Une variable de ce type peut représenter un nombre entre 0 et $(B^L - 1)$. Exemple :

variable $b : \text{MR}[B][3] ;$ -- déclaration d'un vecteur de 3 éléments
 $b := \text{“x.y.z”}[B] ;$ -- affectation de ce vecteur ($0 \leq x, y, z \leq B - 1$)

Ces deux types sont les types de base sur lesquels on peut définir certains types pour faciliter la programmation avec le langage.

- **DR** : type de double-rails – équivalent au type $\text{MR}[2]$
- **DR[L]** : vecteur de L éléments de type double-rails – équivalent au type $\text{MR}[2][L]$
- **BIT** : représente un nombre binaire (un bit : 0 ou 1) – équivalent au type $\text{MR}[2]$
- **BIT[L]** : vecteur des nombres binaires – équivalent au type $\text{MR}[2][L]$
- **BOOLEAN** : représente une valeur booléenne (0 ou 1, vraie ou fausse) – équivalent à $\text{MR}[2]$
- **NATURAL[Max]** : représente un nombre naturel allant de 0 à Max – équivalent au type $\text{MR}[2][L]$ (L est le nombre d'entier supérieure de $(\text{Log}_2[\text{Max}+1])$).
- **SR** : type de mono-rail, sert uniquement à synchroniser des processus communicants (par conséquent, ce type n'est pas utilisé pour déclarer des variables) – équivalent au type $\text{MR}[1]$

b) Type de données signé

Tous les types de données non signés représentés ci-dessus (hormis le type SR) possèdent leurs équivalences signées. Le système de numération pour les nombres signés est représenté l'annexe A).

- **SMR[B]** : type de multi-rails signés dans une base B. Il représente soit un nombre entre $-B/2$ et $B/2$ si B est un nombre pair, soit un nombre entre $-(B-1)/2$ et $(B-1)/2$ si B est un nombre impair. Exemple :

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

- **SMR[B][L]** : vecteur d'éléments de type multi-rails signé. Comme représenté annexe A), uniquement le digit le plus significatif porte le signe. Exemple :

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

Comme les types de données non signés, il existe également les types de données signés définis, tels que

- **SDR** : type double-rails signé – équivalent au type SMR[2]
- **SDR[L]** : vecteur d'éléments double-rails signé – équivalent au type SMR[2][L]
- **INTEGER[Max]** : représente un nombre allant de $-(Max+1)$ à Max – équivalent au type SMR[2][L]

2) Conversion de type de données

CHP est un langage typé : les parties gauche et droite d'une action de communication (lecture, écriture) sont demandées d'avoir un même type de données. La seule façon de convertir des types de données afin de les rendre compatibles est l'affectation. 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'exemple, si une variable a est déclarée avec un type de données MR[3] et une variable b un type DR, l'affectation $a := b$ fait appeler automatiquement la procédure de conversion de type, ce qui permet de convertir la variable b en type de a (MR[3]).

3) Opérateurs

Les opérateurs suivants sont possibles dans notre langage CHP. Ils sont tous définis pour les types de données MR et SMR.

- Comparaison : $=$ (égal), $\neq$ (inégal),
 $<$ (inférieur), $\leq$ (inférieur ou égal),
 $>$ (supérieur), $\geq$ (supérieur ou égal)
- Arithmétique : $+$, $-$, $*$, mod , neg , abs , sll , sla , srl , sra , rol , ror
- Logique : not , nand , and , nor , or , xnor , xor
- Action de communication : $!$ (écriture), $?$ (lecture), $\#$ (sonde)
- Affectation/conversion : $:=$
- Composition : $“;”$ (séquence) et $“,”$ (concurrency)
- Divers : skip , others

4) Structure de contrôle

En CHP, la modélisation du contrôle (instructions conditionnelles, boucles, *etc.*) emprunte la syntaxe des commandes gardées de Dijkstra [DIJK76].

<i>Sélection déterministe</i> Tant qu'aucune garde n'est vraie, on attend. Dès qu'une garde et une seule est vraie, on exécute l'instruction correspondante et termine la sélection.	<pre>[guard1 => bloc1 @ guard2 => bloc2 @ ...]</pre>
<i>Sélection indéterministe</i> Tant qu'aucune garde n'est vraie, on attend. Si une garde et une seule est vraie, on exécute l'instruction correspondante et termine la sélection. Si plusieurs gardes sont vraies, un tirage aléatoire d'une garde parmi l'ensemble des gardes vraies est effectué et l'instruction correspondante à la garde sélectionnée est ensuite exécutée.	<pre>[guard1 => bloc1 @@ guard2 => bloc2 @@ ...]</pre>
<i>Répétition déterministe</i> Tant qu'une garde et une seule est vraie, on exécute l'instruction correspondante. Si aucune garde n'est vraie, on termine la commande de répétition.	<pre>*[guard1 => bloc1 @ guard2 => bloc2 @ ...]</pre>
<i>Répétition indéterministe</i> Tant qu'une ou plusieurs gardes sont vraies, on sélectionne une seule garde vraie pour exécuter l'instruction correspondante. Si aucune garde n'est vraie, on termine la commande de répétition.	<pre>*[guard1 => bloc1 @@ guard2 => bloc2 @@ ...]</pre>

5) Modélisation structurelle de programme

La modélisation structurelle était quasi inexistante dans la proposition initiale du CHP [MART90b], tous les processus étaient déclarés à plat. Afin de pouvoir gérer la complexité des problèmes VLSI, il a été défini une approche de modélisation hiérarchique. La syntaxe présentée ici est bien sûr empruntée aux langages de description de matériel mais légèrement adaptée afin d'obtenir une syntaxe simple et régulière. La hiérarchie est seulement basée sur des processus et des composants. C'est effectivement moins modulaire mais beaucoup moins prolixe qu'en VHDL où il faut à la fois déclarer entité, architecture, composant, processus et configuration. La connectivité entre blocs repose uniquement sur des canaux, il n'est pas possible de déclarer de signaux.

<i>Composant</i> Un composant est la vue globale d'un circuit. Sa vue externe déclare, si il y en a, des paramètres génériques et des ports de communication. Le corps d'un composant est constitué d'instructions concurrentes tel que des instances de composants et des déclarations de processus. Les canaux de communications doivent être déclarés, ils permettent de connecter les différents processus et instances de composants.	<pre>COMPONENT component_name PORT (port_list) {declaration part} BEGIN <component body> END component_name ;</pre>
<i>Processus</i> Contrairement au VHDL, chaque processus doit déclarer ses ports de communications, tout comme un composant. Ceci permet de déclarer proprement les attributs des canaux (codage, direction et protocole) pour ensuite effectuer les vérifications nécessaires. Le processus doit définir ses variables locales et son corps est constitué d'instructions : il peut contenir une partie initialisation ainsi qu'une boucle infinie	<pre>PROCESS process_name PORT (port_list) {declaration part} [instruction_list]</pre>

La modélisation structurelle offerte par cette syntaxe est suffisamment riche pour envisager l'écriture de problème complexe. On peut noter en conclusion les différences suivantes avec le

VHDL : i) la notion VHDL de configuration et de couple entité/architecture n'est pas disponible mais peut s'implémenter avec une politique de renommage des composants ; ii) les possibilités de synchronisation/parallélisme en CHP sont plus riches sachant que l'opérateur de concurrence est disponible dans les processus ; iii) les entêtes de processus CHP doivent être déclarés contrairement au VHDL ce qui permet une meilleure lisibilité structurelle et fonctionnelle de chaque processus.

6) Exemple

La Figure 119 donne un exemple de modélisation structurelle.

```

Component C2
Port
    (E1, E2: IN DI DR;
     S1, S2: OUT DI DR)

Channel A, B: DI DR;
Begin

Process P0
Port
    (E1: IN DI DR;
     A, B: OUT DI DR)
Variable x: DR;
[AI0;
 *[E1?x; ...; AIx, BI(x+1)]
]

C1_instance: C1 port map (A, S1)

Process P1
Port
    (E2, B: IN DI DR;
     S2: OUT DI DR)
[
 *[E2?, B?; ...; S2!]
]
End C2;

```

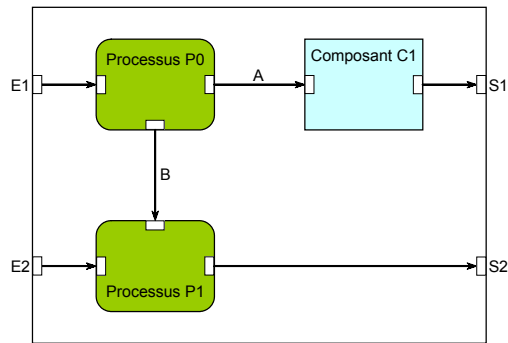


Figure 119 Exemple de modélisation structurelle : code CHP et schéma correspondant

RESUME

Contrairement aux circuits synchrones, les circuits asynchrones fonctionnent avec un mécanisme de synchronisation local (sans signal d'horloge). Ils ont montré depuis de nombreuses années leur pertinence vis-à-vis des circuits synchrones grâce à leurs propriétés de robustesse, de faible consommation, de faible bruit et de modularité. Cependant, le manque actuel de méthodes et d'outils de conception est un frein à leur développement. Ce travail de thèse porte sur la définition d'une méthodologie de conception de circuits intégrés asynchrones quasi-insensibles aux délais (QDI). Les circuits QDI font partie de la classe des circuits asynchrones les plus robustes, propriété avantageuse pour les technologies à venir.

La méthode de conception proposée permet d'une part la modélisation dans un langage de haut niveau, et d'autre part la génération de circuits en portes logiques élémentaires et en portes de Muller. Cette méthode a été prototypée par le développement d'un outil de conception automatique de circuits asynchrones TAST («TIMA Asynchronous Synthesis Tools»). C'est un environnement de conception principalement composé d'un compilateur et d'un synthétiseur offrant la possibilité de générer des circuits asynchrones QDI avec différents modèles de circuits cibles (séquentiel, WCHB, PCHB et PCFB) en partant de descriptions de haut niveau décrites en langage CHP. Le résultat produit par le synthétiseur est une description VHDL de niveau porte qui peut cibler soit une technologie spécifique pour l'asynchrone, soit une bibliothèque de cellules standard (circuits précaractérisés ou FPGAs).

MOTS-CLES

Circuits asynchrones, méthodologies de conception, synthèse de circuits, circuits quasi-insensibles aux délais, processus concurrents communicants, protocoles de communications, langage CHP, réseaux de Pétri, équations de dépendances, cellules standard, CAO, outils de synthèse.

TITLE

AUTOMATIC SYNTHESIS OF QDI ASYNCHRONOUS CIRCUITS

ABSTRACT

Contrary to the synchronous circuits, the asynchronous circuits operate with a mechanism of local synchronization (without clock signal). Since many years, they showed their relevance with respect to the synchronous circuits thanks to their properties of robustness, low power, low noise and modularity. However, the current lack of design methods and associated tools prevents them from being widely spread. This dissertation deals with a new design methodology for quasi-delay insensitive integrated asynchronous circuits (QDI). The QDI circuits are the most robust class of the practical asynchronous circuits, which is an advantageous property for upcoming technologies.

The suggested design method allows on the one hand to model circuits in a high-level language, and on the other hand to generate circuits using only elementary logical gates and Muller gates. This method was prototyped by the development of an automatic design tool for asynchronous circuits, namely TAST ("TIMA Asynchronous Synthesis Tools"). It is a design environment mainly made up of a compiler and a synthesizer targeting QDI asynchronous circuits with various handshaking protocols (sequential, WCHB, PCHB and PCFB) from high level descriptions (language based on concurrent communicating processes). The result produced by the synthesizer is a VHDL gate netlist which can target either an asynchronous specific technology, or a standard cell library (precharacterized circuits or FPGAs).

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 2-84813-010-5 (broché)

ISBNE : 2-84813-011-3 (électronique)