Méthodologie pour l'application industrielle de la synthèse comportementale

E. Berrebi

THESE

présentée par

Elisabeth BERREBI

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Microélectronique**

MÉTHODOLOGIE POUR L'APPLICATION INDUSTRIELLE DE LA SYNTHÈSE COMPORTEMENTALE

Date de soutenance : le 11 décembre 1997

Composition du jury :

Président :	Bernard Courtois
Rapporteurs :	Michel Auguin
	Eric Martin
Examinateurs :	Ivo Bolsens
	Joseph Borel
	Ahmed Amine Jerraya
Invité :	Jean Fréhel

Thèse CIFRE préparée au sein de
du Laboratoire TIMA, 46, avenue Félix Viallet, 38014 Grenoble Cedex
et de Central R&D / SGS-Thomson Microelectronics 850, rue Jean Monnet, 38031 Crolles Cedex

A mes parents

Remerciements

C'est dans le cadre d'un contrat CIFRE entre le Laboratoire TIMA de l'INPG et Central R&D de SGS-Thomson Microelectronics (ST) à Crolles que s'inscrit ma thèse. Elle s'est déroulée pendant les 22 premiers mois au sein de l'équipe SLS ("System-Level Synthesis") du Laboratoire TIMA en collaboration avec ST et également l'IMEC en Belgique, et s'est poursuivie au sein de l'équipe "CMOS Digital Design" à Crolles dans le Groupe DAIS ("Design Automation and Integrated Systems") de ST.

Ma gratitude s'adresse à tous ceux qui ont à un moment ou à un autre contribué à mener à bien ce travail et notamment à M. Bernard COURTOIS, Directeur de Recherche au CNRS, pour m'avoir accueillie au sein du Laboratoire TIMA qu'il dirige, et me faire l'honneur de présider le jury.

Je remercie mes directeurs de thèse M. Ahmed JERRAYA, Directeur de Recherche au CNRS, et M. Jean FRÉHEL de ST, pour leurs discussions et leurs encouragements qui ont permis le bon déroulement de ce travail, Jean FRÉHEL étant à l'origine de mon sujet de thèse et Ahmed JERRAYA m'ayant fait participer en outre aux expériences enrichissantes et variées de l' équipe SLS.

J'exprime ma reconnaissance à M. Joseph BOREL, Vice-Président de SGS-Thomson, Directeur de la Conception et de la CAO, à M. Ivo BOLSENS, Vice-Président de l'IMEC, Directeur de la division VSDM, et mes rapporteurs M. Michel AUGUIN de l'Université de Nice et M. Eric MARTIN de l'Université de Bretagne Sud de m'honnorer de leur participation au jury en consacrant une partie de leur temps précieux.

Ma gratitude s'adresse également à MM. Yves DUFLOS, Michel HARRAND, Jean-Claude HERLUISON, Jean-Pierre MOREAU, Pierre PAULIN et Jean-Pierre SCHOELLKOPF de ST pour avoir enrichi et aiguillé les premières discussions sur ce travail, et particulièrement à Polen KISSION, alors en thèse au Laboratoire TIMA, Serge VERNALDE de l'IMEC, ainsi que Stefan de TROCH alors à l'IMEC, pour leur étroite et précieuse collaboration sur la première application de la thèse.

Je suis aussi reconnaissante aux deux générations de thésards de l'équipe SLS et à tous les membres de l'équipe CMOS Digital Design à ST, avec lesquels j'ai travaillé dans une ambiance très amicale au cours de cette thèse sans avoir hésité à me faire profiter de leurs compétences scientifiques. Merci également aux membres des autres équipes du Laboratoire TIMA et au personnel administratif pour leur aimable collaboration.

Merci à Daniel WEIL du CNET-Meylan, Antoine HANCZAKOWSKI de ST et Jérôme AVEZOU de Synopsys pour leur éclairage sur l'utilisation de l'outil *Behavioral Compiler*.

J'exprime ma profonde gratitude à Laurence NIRMA de ST pour ses encouragements et ses conseils pendant les mois de rédaction. Une aide précieuse m'a été apportée par Polen KISSION, Jean-Claude HERLUISON, François NAçABAL, José SANCHES, Rodolphe SUESCUN, Thierry LEPLEY et Selim ABOU-SAMRA pour réaliser et améliorer ce document, qu'ils en soient remerciés.

Je remercie enfin mes proches qui m'ont encouragée et conseillée tout le long de ce parcours.

Références bibliographiques

[1] J. Monnier, F. Grosvalet, "La densité d'intégration pourra être encore multipliée par dix", Electronique International Hebdo, n°260, p. 7, 10 Avril 1997.

[2] Dataquest, “Market Trends”, *Electronic Design Automation*, chapitre 3, "Major Trends", p. 11, 28 aout 1995.

[3] J. Borel, "Technologies for Multimedia Systems on a Chip", ISSCC, pp18-21, février 1997.

[4] N.G. Einspruch, J.L. Hilbert, *Application Specific Integrated Circuits (ASIC) Technology*, VLSI Electronics, Vol. 23, Academic Press Inc., Harcourt Brace Jovanovich Publishers, 1991.

[5] A.A. Jerraya, H. Ding, P. Kission, M. Rahmouni, *Behavioral Synthesis and Component Reuse with VHDL*, Kluwer Academic Publishers, 1996.

[6] T. Ben Ismail, *Synthèse au niveau système et conception de systèmes mixtes logiciels /matériels*, Chapitre 5, "Le découpage de systèmes au niveau-système", Thèse INPG, janvier 1996.

[7] J. Huisken, F. Welten, "FADIC : Architectural Synthesis Applied in IC Design", 33ème ACM/IEEE Design Automation Conference, pp.579-584, juin 1996.

[8] M.T.-C. Lee, Y.-C. Hsu, B. Chen, M. Fujita, "Domain-Specific High-Level Modeling and Synthesis for ATM Switch Design Using VHDL", 33ème ACM/IEEE Design Automation Conference, pp. 585-590, juin 1996.

[9] P. Kission, E. Berrebi, H. Ding, M. Rahmouni, P. Vijaya Raghavan, A.A. Jerraya, “AMICAL : Interactive Behavioral Synthesis Based on VHDL for Control Flow Dominated Systems”, *Journal of the Brazilian Computer Society*, Volume 2, novembre 1995.

[10] J. Rabaey, H. De Man, J. Vanhoff, G. Goossens, F. Catthoor, "Cathedral II : A Synthesis System for multiprocessor DSP Systems", editions D. Gajsksi, *Silicon Compilation*, Addison-Wesley, 1988.

[11] D.W. Knapp, *Behavioral Synthesis*, Prentice-Hall, 1996.

[12] R. Goering, "EDA & ASICs coping with complexity", *Electronic Engineering Times*, 13 mars 1995.

[13] E. Berrebi, P. Kission, S. Vernalde, S. De Troch, J.C. Herluison, J. Fréhel, A.A. Jerraya, I. Bolsens, "Combined Control Flow Dominated and Data Flow Dominated High-Level Synthesis", 33ème ACM/IEEE Design Automation Conference, juin 1996.

[14] R. Camposano, "Tutorial on Behavioral Synthesis", 33rd ACM/IEEE Design Automation Conference, pp.33-34, juin 1996.

[15] R.A. Berghamaschi, "Productivity Issues in High-Level Design : Are Tools Solving the Real Problems", 32nd ACM/IEEE Design Automation Conference, pp. 674-677, juin 1995.

[16] R.A. Walker, R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.

[17] D. Gajski, L. Ramachandran, "Introduction to High-Level Synthesis", IEEE Design & Test of Computers, pp.44-54, 1994.

[18] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, Mc Graw-Hill, 1994.

[19] C.A. Mead, L.A. Conway, *Introduction to VLSI Systems*, Addisson-Wesley Publishing Company, 1980.

[20] IEEE, "VHDL Language Reference Manual 1076.3", VHDL, 1993.

[21] D.E. Thomas, P. Moorby, *The VERILOG Hardware Description Language*, Kluwer Academic Publishers, 1991.

[22] D. Ku, G. De Micheli, “Hardware C - A language for Hardware Design - Version 2.0”, Computer Systems Laboratory, Université de Stanford, Rapport Technique CSL-TR-90-419, 1990.

[23] P.N. Hilfinger, "A high-level language and silicon compiler for digital signal processing", IEEE Custom Integrated Circuits Conference, pp 72-75, mai 1985.

[24] VHDL Newsletter, janvier 1997.

[25] IMEC, *The Cathedral-2/3 Silicon Compiler for Real Time Signal Processing*, 1993

[26] M. Rahmouni, *Ordonnancement et optimisations pour la synthèse de haut niveau des circuits de contrôle*, Thèse INPG, Février 1997.

[27] T.C. Hu, "Parallel sequencing and assembly line problems", *Operations Research*, pp. 841-848, novembre 1961.

[28] P.G. Paulin, J.P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s”, *IEEE Transactions on Computer-Aided Design*, vol. 8, N°6, juin 1989.

[29] T.E. Fuhrman, "Industrial Extensions to University High-Level Synthesis Tools : Making it work in the Real World", 28th ACM/IEEE Design Automation Conference, pp. 520-525, juin 1991.

[30] M. Genoe, P. Vanoostende, G. Van Wauwe, "On the Use of VHDL-based behavioral synthesis for telecom ASIC design", pp. 96-101, 1995.

[31] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn, "The System Artchitect's Workbench", DAC, juin 1988.

[32] G. De Micheli, D.C. Ku, F. Mailhot, T. Truong, "The Olympus Synthesis System", IEEE Design & Test, pp37-53, octobre 1991.

[33] J. Granacki, D. Knapp, A.C. Parker, "The ADAM Advanced Design Automation System : Overview, Planner and Natural Language Interface", 22nd ACM/IEEE Design Automation Conference, 1985.

[34] M. Koster, J. Biesenack, "The Siemens High-Level Synthesis System, CALLAS", High-Level Synthesis Workshop, novembre 1992.

[35] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Sofware System", 23rd ACM/IEEE Design Automation Conference, 1986.

[36] M. Auguin, F. Boeri, C. Carriere, "Automatic exploration of VLIW processor architectures from a designer's experience based specification", *3rd International Workshop on Hardware/Software Codesign*, pp. 108-115, Septembre 1994.

[37] A. Greiner, F. Pêcheux, "ALLIANCE : A Complete Set of CAD Tools for Teaching VLSI Design", 3ème Eurochip Workshop on VLSI Design Training, 1992.

[38] R. Camposano, R.A. Berghamaschi, C.E. Haynes, M. Payer, S.M. Wu, *Trends in High-Level Synthesis*, chapitre "The IBM High-Level Synthesis System", Kluwer Academic Publishers, 1991.

[39] E. Martin, O. Sentieys, H. Dubois, J.-L. Philippe, "GAUT : An Architectural Synthesis Tool for Dedicated Signal Processors", EuroDAC-EuroVHDL, 1993.

[40] M.L. Flottes, D. Hammad, B. Rouzeyre, "Automatic Synthesis of BISTed Data Paths From High Level Specification", !!!, 1994.

[41] S. Bhatia, N.K. Jha, "Genesis : A Behavioral Synthesis System for Hierarchical Testability", Electronic Design & Test Conference, mars 1994.

[42] Y.-L. Lin, "Recent Developments in High-Level Synthesis", *ACM Transaction on Design Automation of Electronic Systems*, Vol. 2, N°1, pp. 2-21, Janvier 1997.

[43] R. San Martin, J.P. Knight, "Power-Profiler : Optimizing ASICs Power Consumption at the Behavioral Level", 32ème ACM/IEEE Design Automation Conference, pp. 42-47, juin 1995.

[44] J. Fréhel, J.P. Moreau, P. Paulin, "Motion Estimator Design and Verification Workshop", rapport interne SGS-Thomson-Microelectronics, janvier 1995.

[45] V. Anjubault, A. Hanczackowski, "Evaluation of Synopsys Behavioral Compiler at ST-Crolles", rapport interne SGS-Thomson-Microelectronics, mars 1997.

[46] P. Kission, E. Closse, L. Bergher, A.A. Jerraya, "Industrial experimentation of High-Level Synthesis", Euro-DAC/Euro-VHDL, 1993.

[47] K. O'Brien, I. Park, A.A. Jerraya, "DLS : A Scheduling Algorithm for Control-Flow Dominated Circuits", EDAC, 1993.

[48] K. O'Brien, M. Rahmouni, A.A. Jerraya, B. Courtois, "Synthesis for Control-Flow Dominated Machines", dans *Application-Driven Architecture Synthesis*, F. Catthoor, L. Svensson, Kluwer Academic Publishers, 1993.

[49] A.A. Jerraya, K. O'Brien, "SOLAR : An Intermediate Format for System-Level Modelling and Synthesis", J. Rozenblit, K. Buchenrieder (eds), *Computer Aided Software/Hardware Engineering*, IEEE Press, Piscataway, N. J., pp. 147-175, 1994.

[50] M. Romdhani, G. Marchioro, R. Suescun, A.A. Jerraya, "SOLAR Reference Manual", Rapport technique, Laboratoire TIMA Grenoble, 133 pages, septembre 1997.

[51] S. Note, F. Catthoor, G. Goosens, H. De Man, “CATHEDRAL-III: Architecture-driven high-level synthesis for high throughput DSP applications.”, 28th ACM/IEEE Design Automation Conference, juin 1991.

[52] Synopsys Inc. *Synopsys Online Documentation*, Version 3.5.a, VSS Expert Interfaces Manual, 6. C-Language Interface Reference, 1996.

[53] P. Pype, "News is good on behavioral synthesis", *Electronic Engineering Times*, 13 mars 1995.

[54] D. Knapp, T. Ly, D. Mac Millen, R. Miller, “Behavioral Synthesis Methodology for HDL-Based Specification and Validation”, 32ème ACM/IEEE Design Automation Conference, juin 1995.

[55] T. Ly, D. Knapp, R. Miller, D. Mac Millen, “Scheduling using Behavioral Templates”, 32nd ACM/IEEE Design Automation Conference, juin 1995.

[56] P. Kission, "Exploitation de la hiérarchie et de la réutilisation de blocs existants par la synthèse de haut niveau", Thèse INPG, janvier 1996.

[57] C. DiLisi, "Computers in Molecular Biology : Current Applications and Emerging Trends", *Science*, number 240, pp. 47-57, avril 1988.

[58] A. Wise, “Introduction To Motion Picture Coding and the CCITT Algorithm”, Decembre 1989.

[59] P.G. Paulin, J. Fréhel, M. Harrand, E. Berrebi, C. Liem, F. Naçabal, J.-C. Herluison, "High-Level Synthesis and Codesign Methods : An Application to a Videophone Codec", EuroDAC/EuroVHDL, 1995.

[60] M. Harrand, M. Henry, P. Chaisemartin, P. Mougeat, Y. Durand, A. Tournier, R. Wilson, J-C. Herluison, J-C. Longchambon, J-L. Bauer, M. Runtz, J. Bulone, "A Single Chip Videophone Video Coder/Decoder", *IEEE International Solid-State Circuits Conference*, pp.292-293, 1995.

[61] M. Harrand, J. Bulone, "Mise au point en “grandeur réelle” d’un CODEC de visiophone", *Electronique* n°41, pp. 53-63, octobre 1994.

[62] J.-P. Feste, "Compression d'images : pour y voir plus clair!", Electronique, n°22, novembre 1992.

[63] S. Vernalde, P. Schaumont, I. Bolsens, H. De Man, J. Fréhel, "Synthesis of high throughput DSP ASICs using Application Specific Datapaths", *DSP & Multimedia Technology*, juin 1994.

[64] K. Vahtra, "ASIC Design Partitionning", Synopsys Methodology Program.

[65] P. Chambers, "The Ten Commandments Of Excellent Design", *Electronic Design*, Avril 1997.

[66] F. Naçabal, O. Deygas, P. Paulin, M. Harrand, "C-VHDL Co-simulation : Industrial Requirements for Embedded Control Processors", Euro-DAC, Designer Session, Genève, pp.55-60, septembre 1996

Résumé

La synthèse architecturale fait l'objet de recherches intensives depuis 1985. Quelques expériences ont été menées depuis 5 ans. Mais son application industrielle est très récente. Le but de cette thèse est de spécifier les contraintes industrielles pour des outils de synthèse architecturale et une méthode de conception adaptée afin d'introduire à terme la synthèse comportementale dans le flot de conception industriel. Les difficultés industrielles sont dues à la complexité des circuits et à des incompatibilités éventuelles avec les environnements de conception existants.

Pour la conception de circuits complexes, nous présentons ici une méthode modulaire à base de synthèse architecturale. Nous spécifions aussi les caractéristiques nécessaires à un outil de synthèse comportementale pour son intégration dans le flot de conception industriel existant.

Nous avons eu l'idée de combiner deux outils complémentaires de synthèse comportementale. L'application de cette méthode à un circuit industriel, nous a fourni de premiers résultats prometteurs : une réduction de la longueur des descriptions au cinquième, une réduction du temps de conception ainsi qu'un surplus en surface de seulement 5% par rapport à la méthode classique manuelle.

Cependant, en appliquant notre méthode à un circuit plus complexe, nous avons mis en évidence les limites, à ce jour, des outils de synthèse architecturale utilisés dans cette thèse. Le temps de conception gagné par l'automatisation de la génération de l'architecture est perdu dans l'intégration des outils dans le flot de conception industriel existant.

Abstract

Architectural synthesis is subjected to intensive research since 1985. A few experiments have been carried out for 5 years. But its industrial application is very recent. The aim of this thesis is to specify the industrial constraints for architectural synthesis tools and to suggest an adapted design method in order to introduce behavioral synthesis in the industrial design flow. Industrial issues are due to the complexity of the circuits and to possible incompatibilities with the existing design environments.

For the design of complex circuits, we present here a modular method based on architectural synthesis. We also specify the features required for a behavioral synthesis tool for its integration in the existent industrial design flow.

We have carried out a promising experiment combining two complementary high level synthesis tools, revealing a reduction of the description length to the fifth, a reduction of the design time and a reasonnable area overhead of 5% comparing to the classical manual method results.

However, the application of this method on a circuit more complex has shown the limitations of the architectural synthesis tools used in this thesis, by now. Design time gained by the automation of the architecture generation was lost in the integration of the tools in the existent industrial design flow.

Table des matières

Chapitre I : Introduction

Pour situer le cadre de cette thèse, l'introduction retrace brièvement l'évolution des circuits intégrés. Elle fait ressortir l'incapacité des méthodes et outils de conception actuels à profiter des progrès des technologies sur le silicium. C'est aujourd'hui un problème majeur pour l'industrie du semi-conducteur qui se tourne vers la recherche dans le domaine de la CAO.

Notre propos ici est de réduire le temps de conception des circuits exclusivement matériels, ou ASICs (*Application Specific Integrated Circuits*). On trouve dans cette partie une description du flot de conception classique de ces circuits et le vocabulaire associé ainsi que les objectifs et la contribution de ce travail.

1. Problèmes de productivité

Le stade actuel des technologies submicroniques (CMOS 0,25µm) permet déjà "d'intégrer 30 millions de transistors logiques avec 16 Mbits de DRAM sur une puce" [1]. A l'usine de fabrication de semi-conducteurs de SGS-Thomson Microelectronics, à Crolles, on prévoit les premiers prototypes en technologie CMOS 0,18µm pour 1998. Il faudra à chaque fois, 2 à 3 ans pour atteindre successivement les technologies 0,12µm, 0,10µm puis 0,08µm [1]. "Dans dix ans, des circuits de 300 millions de transistors seront réalisables" [1].

Depuis 1987, le domaine de la conception des circuits intégrés constate un décalage croissant entre les technologies utilisées et le gain de productivité en conception [2]. En 1995 Dataquest [2] prévoyait que “pendant la conception d’un circuit de 4 millions de portes, l’industrie des semi-conducteurs serait capable de fabriquer des circuits de 16 millions de portes”. La tendance va en s’accentuant. Cet écart constitue un facteur limitatif pour l’exploitation des capacités des nouvelles technologies à venir. Il va donc falloir faire face à ce décalage par des méthodes de conception plus rapides. Elles devront largement faire appel à la réutilisation de composants déjà réalisés et à une automatisation plus poussée. Cette perspective lance un défi au domaine de la CAO et des méthodologies de conception.

2. Flot de conception d'un système intégré

L'évolution de la technologie a permis l'intégration de systèmes de plus en plus complexes sur une même puce. La figure I.1 illustre le flot général qui débute par le partitionnement du système en sous-systèmes ou parties logicielles et matérielles, suivant les spécifications. La colonne de gauche représente le flot de conception de chaque partie matérielle. Les parties matérielles sont transposées sur un ASIC ou un FPGA (*Field Programmable Gate Array*). Après validation de la description comportementale, on élabore manuellement ou automatiquement une description de l'architecture au niveau transfert de registre que l'on transpose ensuite sur une bibliothèque de cellules ou sur un FPGA. La colonne de droite représente le flot de conception de chaque partie logicielle. Depuis environ cinq ans, il existe des outils de génération de code qui, à partir d'un algorithme en C, fournissent le code assembleur pour programmer les coeurs de DSP et les micro-contrôleurs ciblés [3].

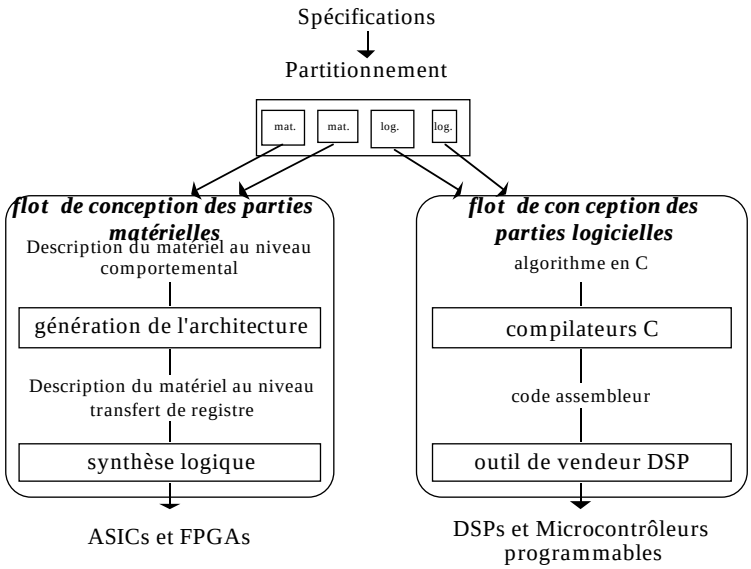


Figure I.1 : Flot de conception d'un système intégré.

3. Les ASICs

Dans le cadre de cette thèse, nous nous limitons à la conception des parties matérielles, il s'agit desASICs.

3.1. Définition

Un ASIC désigne un circuit intégré conçu pour une application particulière ou pour une utilisation finale comme lecteur de disque compact ou système de télécommunication [4]. Contrairement aux autres circuits intégrés, comme les mémoires ou les micro-processeurs, les ASICs n'ont pas un domaine d'application étendu. De plus, ils requièrent une méthodologie de conception fondée sur une large utilisation d'outils et de systèmes de CAO.

3.2. Flot de conception d'un ASIC

La figure I.2 reproduit le flot de conception classique d'un ASIC. Avant de commenter ce flot, il est utile d'exposer brièvement la classification des modèles de représentation de circuits. Ces représentations sont communément classées suivant le niveau d'abstraction. A chacun de ces niveaux, elles se singularisent par un concept temporel particulier et des primitives de description. Le tableau I.1 [5] résume ces caractéristiques.

Niveau de description	Étape de calcul	Primitives
Niveau physique et logique	Délai	Portes, transistors
Niveau transfert de registre	Cycle d'horloge	Registres, opérateurs, transferts
Niveau algorithmique	Calcul	Calcul, contrôle
Niveau système	Transaction	Processus, communication

Tableau I.1 : Concepts de temps et niveaux de spécification lors de la conception

Le niveau le plus bas est le niveau physique et logique. L'étape de base est le délai. On décrit le circuit en termes de transistors et de portes interconnectés par des fils.

Le niveau suivant est le niveau transfert de registre, ou RTL (*Register Transfer Level).* L'étape de base est le cycle d'horloge pour traverser la logique combinatoire entre deux bancs de registres. Généralement les descriptions déterminent les tâches à exécuter à chaque cycle d'horloge. Elles assemblent des registres, des opérateurs et des transferts entre ces registres et ces opérateurs.

On qualifie de comportemental ou algorithmique le niveau supérieur. L'étape de base est le calcul. Un calcul se compose d'un ensemble d'opérations exécutées entre deux points successifs d'entrées/sorties et/ou de synchronisation. Elle peut durer plusieurs cycles d'horloge. Généralement, la description utilise un ensemble de protocoles d'échange de données pour interagir avec le monde extérieur.

Le niveau le plus haut est le niveau système. L'étape de base est la transaction de communication pour contrôler l'enchaînement des processus. La primitive de description de base est le processus.

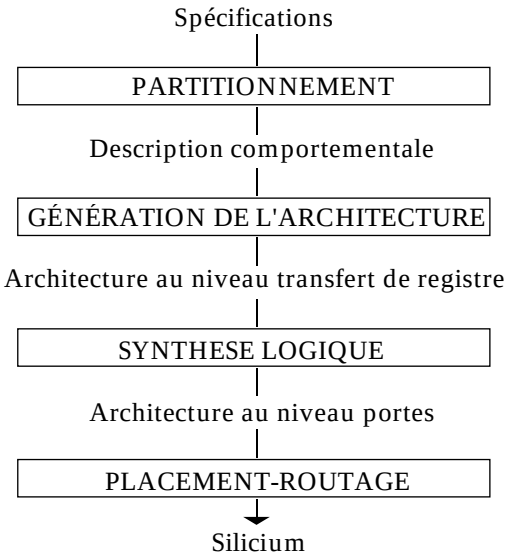


Figure I.2 : Flot de conception d'un ASIC.

Le flot de la figure I.2 se décompose en quatre étapes successives qui sont le partitionnement et la description des algorithmes, la génération de l'architecture, la synthèse logique qui transpose l'architecture sur une bibliothèque de cellules de base et enfin le placement-routage sur le silicium.

La première étape se situe au niveau système, c'est le partitionnement du circuit. Le partitionnement [6], consiste, à partir des spécifications, à découper le circuit en un ensemble de sous-systèmes communicants. Ce découpage peut être guidé par de nombreux paramètres comme la fonctionnalité, les fréquences de fonctionnement, la synchronisation et la concurrence de tâches, etc. Après avoir réparti les séquences de tâches concurrentes dans des sous-systèmes distincts, le concepteur décrit le comportement de chacun de ces sous-systèmes. Ces descriptions servent à valider la fonctionnalité globale du circuit.

Chaque description comportementale validée est ensuite traduite en une architecture au niveau transfert de registre, c'est l'étape de génération de l'architecture. La synthèse logique traduit l'architecture obtenue en une description au niveau portes, optimisée en surface et en performance. Enfin, l'étape de placement et de routage des descriptions des architectures résultantes sur le silicium termine ce flot. Aujourd'hui l'étape critique se situe entre la compréhension des spécifications et la description de l'architecture définitive du circuit, comme on le verra au § 4.1.

4. Contexte de la thèse

4.1. Genèse de ce travail

Au département R&D de SGS-Thomson Microelectronics à Crolles (ST), la réalisation d'une architecture sur une bibliothèque de cellules standard, puis son placement-routage sont automatisés par des outils de CAO. Mais les étapes de partitionnement du circuit, à partir des spécifications, et de génération de l'architecture, à partir du modèle comportemental, demeurent complètement manuelles. La figure I.3 représente la répartition du temps consacré aux différentes étapes du flot de conception d'une puce de 573 000 transistors, achevée en 1994 à ST. Il en ressort que c'est la description des modèles RTL qui a requis plus de la moitié du temps de conception à elle seule!

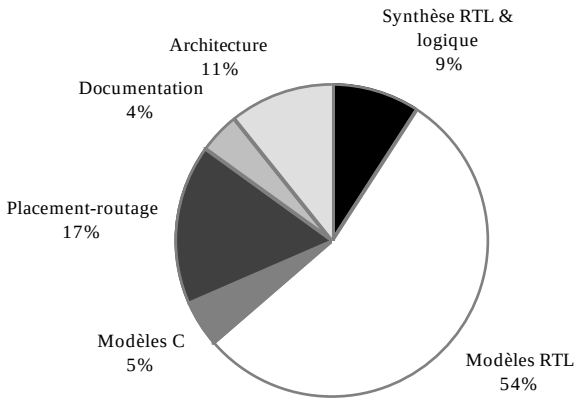


Figure I.3 : Répartition du temps entre les différentes étapes du flot de conception.

Depuis 1985, la synthèse architecturale fait l'objet de nombreuses recherches pour répondre au besoin croissant de productivité industrielle afin de réduire les phases critiques de conception. Des outils de synthèse architecturale apparaissent pour servir de support à ces recherches et certains sont commercialisés. Quelques circuits industriels particuliers ont ainsi pu être conçus [7, 8].

Au Laboratoire de recherche TIMA de l'INPG, l'équipe SLS (*System Level Synthesis*) développe un outil de synthèse architecturale, *Amical* [9], pour automatiser l'étape de description de l'architecture à partir de la description comportementale. L'étape entre la compréhension des spécifications et la description comportementale du circuit reste manuelle. Cet outil est spécialisé pour les circuits de contrôle. Au même moment, d'autres outils se sont spécialisés dans la conception de circuits de calcul de données. *Cathedral-2/3* [10], développé à l'IMEC (Leuven, Belgique), était dans ce domaine l'outil le plus avancé au début de la thèse.

Cette thèse, régie par une convention CIFRE entre TIMA et ST, avait pour objectif d'évaluer les résultats de l'application de la synthèse architecturale au domaine industriel et notamment le gain de temps de conception. Il fallait déterminer une méthode adaptée.

Pour ST il s'agissait d'évaluer, à ce jour, les techniques de synthèse architecturale sur des circuits complexes hétérogènes, à parties contrôle et opératives complexes, par rapport à la méthode usuelle. L'approfondissement des problèmes rencontrés permet de se préparer à l'automatisation de cette étape et au choix d'outils de synthèse architecturale.

Pour TIMA, il s'agissait d'étudier les problèmes réels qui apparaissent lors de la synthèse architecturale d'un circuit complexe hétérogène dans des conditions industrielles. Il peut apparaître des problèmes de méthodologie, de synchronisation, de validation, de réutilisation de composants déjà existants, etc. Les résultats de ces études servent à préciser les directions de recherches.

4.2. Objectifs et contribution

Le circuit choisi pour cette expérience est l'Estimateur de Mouvement du Visiophone CODEC HCMOS5 (technologie 0,5 µm). Il présente la particularité de contenir à la fois une partie contrôle complexe (séquence de tâches, protocoles asynchrones, etc.) et des opérateurs de calcul sophistiqués (traitement du signal, hautes performances, etc.). Au début de la thèse, aucun outil n'était capable de réaliser la synthèse architecturale d'un tel circuit. Pour la synthèse architecturale de l'exemple complet il a fallu faire appel à deux outils de synthèse architecturale appelés *Amical* et *Cathedral-2/3*. *Amical* synthétise la partie contrôle. *Cathedral-2/3* synthétise la partie traitement du signal.

En fait, cette combinaison d'outils de synthèse architecturale constitue l'une des contributions originales de cette thèse. En attendant l'émergence d'outils complets, elle permettait de recouvrir un domaine d'application le plus large possible et d'anticiper certains problèmes indépendants du nombre d'outils utilisés. Cette expérience a nécessité une étroite collaboration entre l'équipe SLS de TIMA qui développe *Amical*, l'équipe VSDM de l'IMEC qui développait *Cathedral-2/3*, et des personnes du département DAIS de Central R&D de ST-Crolles impliquées dans le projet du VideoCODEC.

Après le succès de l'évaluation des outils *Amical* et *Cathedral-2/3*, ST a jugé intéressant d'appliquer la même méthode pour concevoir un circuit beaucoup plus complexe en termes d'accès aux mémoires et comportant 12 modes de fonctionnement différents. Nous avons donc tenté de tirer profit de la première expérience. Mais nous avons montré que les limites de la méthode appliquée et des outils de synthèse comportementale ne permettaient pas la réalisation industrielle efficace d'un circuit d'une telle complexité. Ces limites sont évoquées à la fin du document.

La contribution de cette thèse réside en cinq points :

- la réalisation d'exemples et de circuits intégrés dans l'industrie jusqu'aux niveaux transfert de registre ou portes, en utilisant des outils de synthèse architecturale.
- la combinaison de plusieurs outils de synthèse architecturale pour la conception de circuits complexes.
- l'analyse des problèmes rencontrés au cours des expériences et des résultats de celles-ci.
- la mise au point d'une méthode de conception permettant d'intégrer la synthèse architecturale dans un flot de conception existant.
- la participation à l'évolution de l’outil de synthèse architecturale *Amical*.

5. Structure de la thèse

Le chapitre II dresse l'état de l'art de la synthèse architecturale par rapport aux besoins industriels.

Le chapitre III s'intéresse en particulier à trois outils de synthèse différents utilisés et évalués à ST sur des circuits réels : *Amical* [9], *Cathedral-2/3* [10] et *Behavioral Compiler* [11], outil commercial développé par Synopsys. A partir de critères particuliers nous montrons leur spécificité et leur complémentarité suivant les domaines d'application. Nous en déduisons la nécessité éventuelle de l'utilisation mixte d'outils puis nous proposons un flot de conception original.

Le chapitre IV illustre l'application du flot présenté au chapitre III avec uniquement l'outil *Amical*. Nous y développons une méthode de conception modulaire de circuits avec des parties contrôle et opératives complexes, à base de synthèse architecturale.

Le chapitre V fournit les résultats de l'application de la synthèse architecturale sur un exemple industriel. Ces résultats sont donnés en termes de gain en productivité, coût en surface et performance, de l'automatisation du passage de la description comportementale à l'architecture du circuit.

Le chapitre VI récapitule les problèmes résolus par la synthèse architecturale et pose les défis qu'elle doit relever pour faciliter la prochaine automatisation du flot de conception : le partitionnement et la synthèse des communications. Nous retenons également les démarches jugées positives d'après l'expérience du chapitre V et tentons d'améliorer celles qui le nécessitaient.

Chapitre II : La Synthèse Architecturale dans le contexte industriel

Les travaux sur la synthèse architecturale se sont intensifiés vers 1985. Depuis 1990 on évalue et utilise des outils de synthèse architecturale universitaires ou commerciaux dans le milieu industriel [7, 8, 12, 13]. Il y a cinq ans on prévoyait que la synthèse architecturale réduirait le temps de conception d'un facteur 10 par rapport à la synthèse classique. Mais aujourd'hui des expériences affichent un facteur variable, mais toujours intéressant suivant les applications, de 3 à 10 [14, 15]. Il faut fournir un important travail de méthodologie pour rendre effectif le gain attendu en productivité, essentiellement du fait que les outils sont spécialisés pour des domaines d'application spécifiques.

Dans ce chapitre, nous présentons brièvement les étapes de la synthèse architecturale. Nous distinguons le traitement des applications suivant leur orientation : flot de données ou flot de contrôle. Nous caractérisons en particulier les circuits candidats à ce type de synthèse. Nous dressons l'état de l'art de la synthèse architecturale à travers les outils qui existent. Enfin nous tentons de décrire les besoins de l'industrie et ses contraintes dans ce domaine.

1. La synthèse architecturale

1.1. Principe général

La synthèse architecturale est le processus (figure II.1) qui traduit des spécifications comportementales sous forme algorithmique en une architecture matérielle capable de les exécuter [5, 16, 17, 18]. On l'appelle aussi synthèse comportementale ou synthèse de haut niveau. Généralement, un contrôleur et un chemin de données composent l'architecture. Le chemin de données est décrit sous la forme d'une liste structurelle d'un ensemble de composants RTL interconnectés, comme des unités arithmétiques et logiques (UALs), des registres et des multiplexeurs. La description comportementale spécifie la fonction à réaliser par le circuit. Elle peut être textuelle ou graphique. Un outil de synthèse comportementale agit comme un compilateur qui, à une spécification de haut niveau, fait correspondre une architecture. Pour modifier l'architecture, le concepteur peut en premier lieu modifier les paramètres de synthèse, sinon il peut revenir sur la description comportementale. Cette automatisation soulage la tâche du concepteur par des descriptions plus courtes, plus lisibles et plus flexibles.

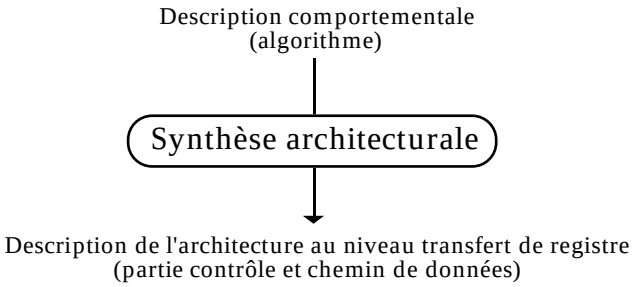


Figure II.1 : La Synthèse Comportementale

L’outil de synthèse comportementale ordonnance automatiquement les opérations d'une description algorithmique dans des cycles d’horloge, et alloue et partage des ressources matérielles à travers plusieurs cycles d’horloge [17]. La suite de cette partie fait référence au chapitre I de l’ouvrage intitulé *Behavioral Synthesis and Component Reuse with VHDL* de Jerraya [5]. Elle décrit les étapes successives de la synthèse architecturale en commentant la figure II.2.

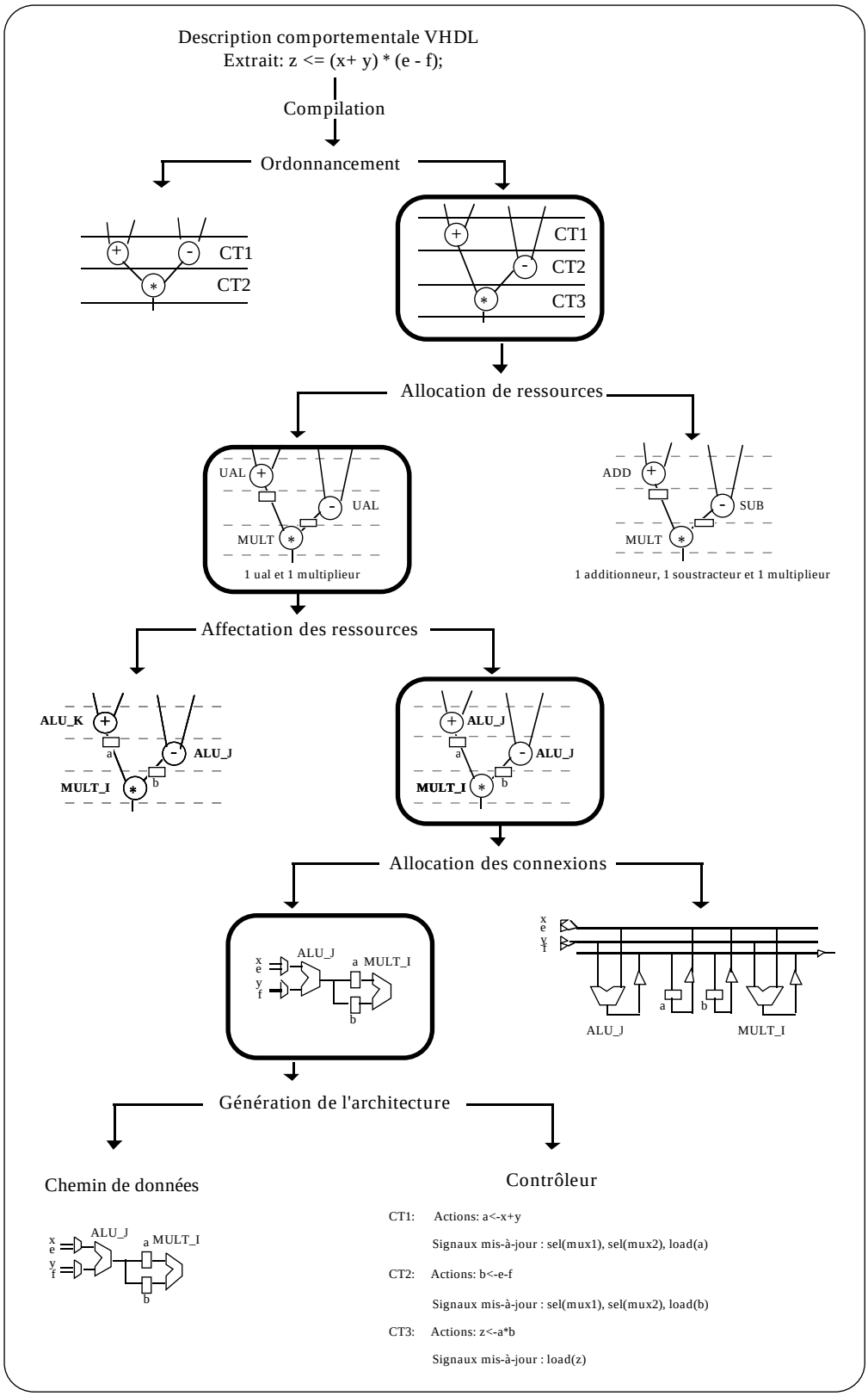


Figure II.2 : Les étapes de la synthèse architecturale

La synthèse comportementale traduit une spécification fonctionnelle en une architecture RTL. Traditionnellement, elle se décompose en cinq étapes principales [18] : traduction en une forme intermédiaire, ordonnancement, allocation, affectation des ressources, génération de l'architecture. La figure II.2 illustre ces étapes à travers un exemple simple. Chaque étape peut fournir plusieurs solutions. Dans le schéma de la figure II.2 on se restreint à deux solutions alternatives par étape. A chaque étape, la solution encadrée en gras est sélectionnée pour l'étape suivante. Il est important de noter que seule l'étape d'ordonnancement est spécifique à la synthèse comportementale, toutes les autres étapes peuvent être réalisées par la synthèse logique classique.

1.2. Compilation et génération de la forme intermédiaire

Cette étape peut inclure des transformations comme pour les compilateurs, appelées transformations de haut niveau dans la littérature. Elle a pour objectif de s'affranchir de tous les détails relatifs au langage de description et au style d'écriture. De telles transformations incluent la propagation de constante, l'élimination de code mort, le déroulement de boucles et l'expansion des procédures. Il existe deux classes de formes intermédiaires. Les premières se fondent sur des graphes, on les appelle formes intermédiaires orientées langage. Les secondes se fondent sur des machines d'états finis, on les appelle formes intermédiaires orientées architecture. Lorsque l'on utilise une forme intermédiaire orientée langage, cette étape est immédiate puisque les modèles de graphes sont très proches de la plupart des langages de description comportementaux. Mais lorsque l'on utilise une forme intermédiaire orientée architecture, cette étape nécessite des transformations complexes pour obtenir une machine d'états finis à partir d'une description. Celles-ci peuvent même inclure un ordonnancement et une allocation partiels.

1.3. Ordonnancement

L'ordonnancement est le partitionnement de la description comportementale en sous-graphes, chacun étant exécuté en une seule étape de contrôle. Une étape de contrôle correspond à une transition d'une Machine d'Etats Finis (MEF). Cela peut inclure plusieurs opérations à exécuter en parallèle. On suppose bien sûr, que le nombre de ressources est suffisant pour exécuter des opérations en parallèle. Certains algorithmes d'ordonnancement sont contraints en ressources et / ou en performance. Il existe différents modèles d'ordonnancement. On distinguera dans la section de ce chapitre les modèles orientés flot de données des modèles orientés flot de contrôle. Sur la figure II.2 deux ordonnancements s'ensuivent; dans la solution de gauche l'exécution de la description comportementale requiert deux étapes de contrôle *CT1* et *CT2*. Cependant dans cette solution, l'exécution de *CT1* nécessite deux opérateurs pour exécuter en parallèle l'addition et la soustraction. La solution de droite requiert trois étapes de contrôle, mais il n'y a plus de parallélisme de tâches. Pour simplifier on pourra supposer qu'une étape de contrôle prend un cycle d'horloge.

1.4. Allocation

L'allocation détermine le nombre et le type de ressources nécessaires pour exécuter la description comportementale. Cette étape fixe le nombre et le type des unités d'exécution, d'unités de mémorisation (registres, bancs de registres) et d'unités de communications (multiplexeurs, bus, fils). Bien entendu, le nombre de ressources peut restreindre le parallélisme permis dans le chemin de données et ainsi restreindre l'ordonnancement. Sur la figure II.2, en partant d'une description ordonnancée, représentée par un graphe flot de données, la tâche d'allocation de ressources nécessite d'allouer des opérateurs pour exécuter les opérations du modèle comportemental et des registres pour mémoriser les valeurs intermédiaires utilisées dans plus d'un cycle d'horloge. Dans ce cas également, on représente deux solutions d'allocation. Dans la solution de gauche, on alloue deux types d'opérateurs *UAL* et *MULT*. Dans la solution de droite, on alloue trois types d'opérateurs *ADD*, *SUB* et *MULT*.

1.5. Affectation des ressources

Cette étape détermine les ressources utilisées pour chaque opération de la description comportementale. La figure II.2 propose deux solutions. Dans la solution de gauche, on affecte les opérations + et - à deux UALs distinctes. Tandis que dans la solution de droite, ces deux opérations sont exécutées par la même UAL. Deux registres *a* et *b* mémorisent les sorties des deux premières opérations.

1.6. Allocation des connexions

L'allocation des connexions détermine les ressources nécessaires pour la communication entre les unités du chemin de données qui sont les registres et les opérateurs. On distingue deux types d'architecture, les architectures à base de bus et les architectures à base de multiplexeurs. En accord avec le style de l'architecture, cette étape va produire les chemins nécessaires pour les échanges de données. Sur la figure II.2, les chemins de communication de la solution de gauche utilisent des multiplexeurs. Ceux de la solution de droite utilisent des bus. Des interrupteurs, ou composants trois-états contrôlent l'accès aux bus. Le chemin de données résultant inclut tous les chemins nécessaires pour les transferts entre les entrées et les sorties, les registres internes et les opérateurs.

1.7. Génération de l'Architecture

Cette étape produit une description RTL du circuit. Le contrôleur et le chemin de données sont construits à partir du résultat des étapes précédentes. Sur la figure II.2, en partant de la solution à base de multiplexeurs on délivre un modèle "contrôleur-chemin de données". Le contrôleur inclut la séquence des étapes de contrôle et l'activation des composants du chemin de données pendant chaque étape de contrôle.

Les étapes d'ordonnancement et d'allocation sont interdépendantes. Plusieurs algorithmes existent pour chacune d'elles. Ils privilégient les contraintes en surface et/ou en performance.

2. Synthèses orientées flot de données ou flot de contrôle

La restriction d'un outil de synthèse architecturale à un domaine d'application spécifique permet de réduire la complexité de cette transformation et de produire de meilleurs résultats [5]. Pour la synthèse architecturale on distingue généralement dans la littérature deux domaines d'application, les applications orientées flot de données et les applications orientées flot de contrôle.

Une application orientée flot de données traite des flots de données réguliers. Les entrées et les sorties du circuit sont des signaux à débit fixe. On spécifie généralement le comportement comme un ensemble périodique d'opérations à exécuter sur chaque nouvelle donnée. Un filtre est un exemple typique d'application dominée flot de données. En synthèse classique ces circuits, pour lesquels on recherche surtout une grande performance, sont modélisés en RTL par du flot de données pur ou une machine de Moore [19].

Une application orientée flot de contrôle est actionnée par un ensemble de commandes à interpréter. Chaque commande, séquence de contrôle ou de données, peut impliquer la lecture ou l'écriture de structures de données complexes et l'exécution d'un algorithme spécifique relatif à cette commande. Le calcul peut dépendre des données. L'application comporte des protocoles de requête / acquittement avec l'environnement, des boucles dépendant des données et une réaction rapide aux interruptions. Plus généralement, des exemples de machines dominées flot de contrôle incluent les systèmes de contrôle temps réel, les contrôleurs complexes distribués et des applications des télécommunications, pour la correction de code, le détramage.

A ce jour la plupart des outils et méthodes sont spécialisés dans l'un ou l'autre des domaines. Cette spécialisation des outils détermine les caractéristiques suivantes :

- leur langage d'entrée,
- leur architecture cible,
- leur type d'ordonnancement.

Cette section montre chacune de ces caractéristiques dans une optique de flot de contrôle, puis de flot de données. Nous distinguerons donc par la suite les synthèses orientées flot de contrôle et les synthèses orientées flot de données.

2.1. Description d'entrée flot de données ou flot de contrôle

La description d'entrée peut être textuelle ou graphique. Mais depuis l'avènement de la synthèse logique, les concepteurs adoptent largement les langages de description textuelle de matériel qui offrent une plus grande souplesse que les solutions graphiques. Certains langages permettent en effet d'assembler plusieurs niveaux d'abstraction comme VHDL [20] et Verilog [21]. D'autres langages sont dédiés au niveau comportemental comme Hardware C [22], Silage [23]. Au début de la conception d'un circuit, le choix du langage de description dépend de son admissibilité par les outils de simulation et de synthèse, et de sa capacité à abstraire le comportement souhaité.

Le rôle des langages de description de matériel a changé ces dernières années. L'utilisation de ces langages pour la simulation et la documentation a été étendue avec succès à leur utilisation pour la synthèse et la vérification formelle. Les langages VHDL et Verilog peuvent décrire des structures, du flot de données et des processus comportementaux. Ils permettent ainsi la simulation de systèmes composés de sous-blocs de niveaux d'abstraction différents. Mais avec l'automatisation du flot de conception, VHDL et Verilog deviennent aussi les langage d'interface des environnements de conception. Ils garantissent que l'on synthétise effectivement ce que l'on simule. Le prochain standard VHDL, prévu pour 2000, devrait orienter le langage vers la conception de systèmes [24]. Par la suite dans ce texte on illustrera des extraits d'algorithmes en VHDL et plus rarement en Silage.

Il existe deux styles de description : le style concurrent et le style séquentiel. Dans le style concurrent, toutes les instructions de la description sont exécutées en parallèle et ce, perpétuellement. L'ordre des instructions n'importe donc pas. Par contre dans le style séquentiel, les instructions sont décrites dans un ordre particulier. Une description séquentielle comporte des points de synchronisation avec l'extérieur. Ce sont les instructions *wait* en VHDL. Ces points de synchronisation ordonnent les instructions d'entrées / sorties dans le temps. En VHDL, entre deux points de synchronisation consécutifs les instructions peuvent être exécutées en parallèle dans la mesure où il n'y a pas de dépendance de données entre elles et ce, sans conséquence sur la synchronisation avec l'extérieur.

Une application "orientée flot de contrôle" se compose essentiellement de protocoles asynchrones, de compteurs de boucle “tant que” effectuant un test sur des données d’entrée ou internes. Pour la description d'entrée d’une telle application, il est plus commode d'utiliser un style séquentiel. Le code qui suit, figure II.3, est une description VHDL d’un circuit calculant le plus grand commun diviseur (PGCD) de deux nombres suivant l’algorithme d’Euclide. Il illustre parfaitement une application orientée flot de contrôle. On peut noter que dans le cas de cet exemple, la boucle de calcul interne dépend des données externes, *xi* et *yi*. Ainsi, il est difficile de déterminer le nombre d'interactions effectuées par cette boucle durant la synthèse comportementale.

```
entity pgcd is port(
        clk, start, din     : in    bit;
        xi, yi              : in    integer;
        dout                : out   bit;
        output              : out   integer);
end pgcd;

architecture behavior of pgcd is begin
process
variable x, y: integer;
begin
    wait until ((start='1') and (clk='1' and clk'event));

    calculation : loop
        wait until ((din='1') and (clk='1' and clk'event));
        dout <= '0';
        x := xi;
        y := yi;
        while (x /= y) loop
            if (x < y) then y := y - x;
            else x := x - y;
            end if;
        end loop;
        wait until ((din='0') and (clk='1' and clk'event));
        dout <= '1';
        output <= x;
    end loop calculation;

end process;
end behavior;
```

Figure II.3 : Description VHDL de l’algorithme de calcul du PGCD

Pour réaliser la synthèse de haut niveau, les descriptions sont généralement traduites en un graphe de contrôle. La figure II.4 montre le graphe de contrôle découlant de la description de la figure II.3. Les noeuds du graphe correspondent à des opérations et les arcs à des relations d'antécédence.

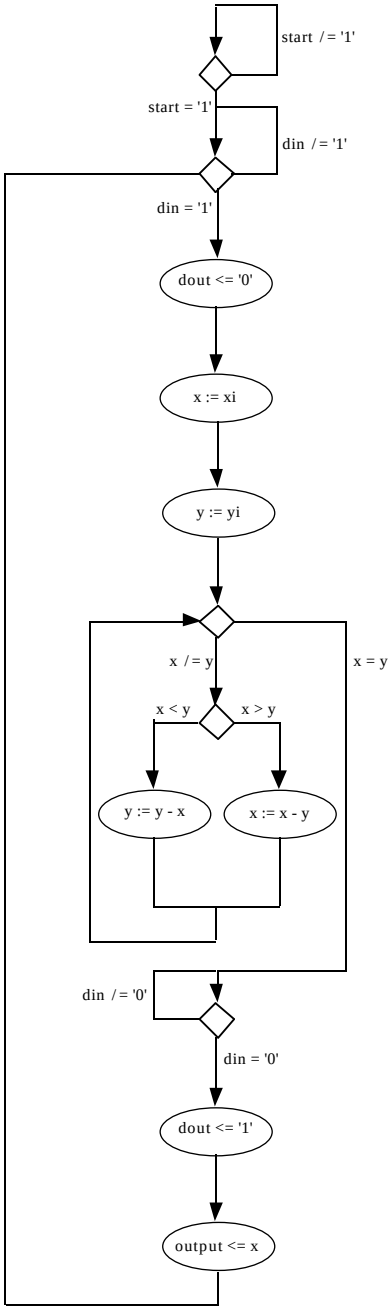


Figure II.4 : Graphe flot de contrôle de la description décrite en VHDL.

Une application "orientée flot de données" se compose essentiellement de calculs concurrents activés à chaque nouvelle donnée d'entrée, avec un minimum d'instructions de contrôle. Pour la description d'entrée d'une telle application, il est plus commode d'utiliser un style concurrent. Le code qui suit, figure II.5, est une description, dans le langage fonctionnel synchrone Silage [23], d'un adaptateur intégré dans un filtre à débit variable [25]. Il illustre une application orientée flot de données.

```
#define word fix<24,8>

func main (In, Coef : word) Out : word =
begin
    tmp@@1 = 0;
    tmp@@2 = 0;
    tmp = In + Coef;
    Out = tmp + tmp@1 + tmp@2;
end;
```

Figure II.5 : Description Silage d'un adaptateur.

On remarque par rapport à VHDL que l'affectation du signal interne *tmp* suffit à le déclarer. Une autre particularité de Silage est d'admettre les expressions de délai qui consistent en un signal retardé, un opérateur de délai '@' et une valeur de délai. La valeur du délai est obligatoirement un scalaire positif. Par exemple dans la figure II.5, *tmp@1* et *tmp@2* désignent respectivement les valeurs de *tmp* à l'issue du dernier et de l'avant dernier cycle de calcul. C'est pourquoi l'opérateur @@ initialise ces signaux retardés.

Pour réaliser la synthèse de haut niveau, les descriptions sont généralement traduites en un graphe de flot de données. La figure II.6 montre le graphe de flot de données correspondant à la figure II.5. Les noeuds du graphe correspondent à des opérations et les arcs à des valeurs qui indiquent des dépendances de données. Les boîtes représentent des registres de mémorisation.

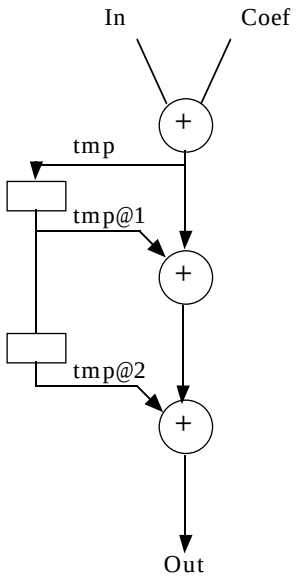


Figure II.6 : Graphe flot de données de la description décrite en Silage.

2.2. Architectures cibles

L'architecture générique cible, communément utilisée par les outils de synthèse architecturale, a pour base un contrôleur et un chemin de données. Cette architecture globale convient à la fois aux applications dominées flot de données et dominées flot de contrôle. La partie contrôle est généralement modélisée par une machine d'états finis qui peut être réalisée sur une ROM ou par une solution câblée. Le chemin de données se compose d'unités de mémorisation (registres, mémoires), d'unités fonctionnelles (UALs, additionneurs, etc.), d'unités d'entrée/sortie et d'unités de communication (bus, multiplexeurs). La figure II.7 illustre l'architecture à base de multiplexeurs correspondant à la description VHDL de l'algorithme du calcul du PGCD.

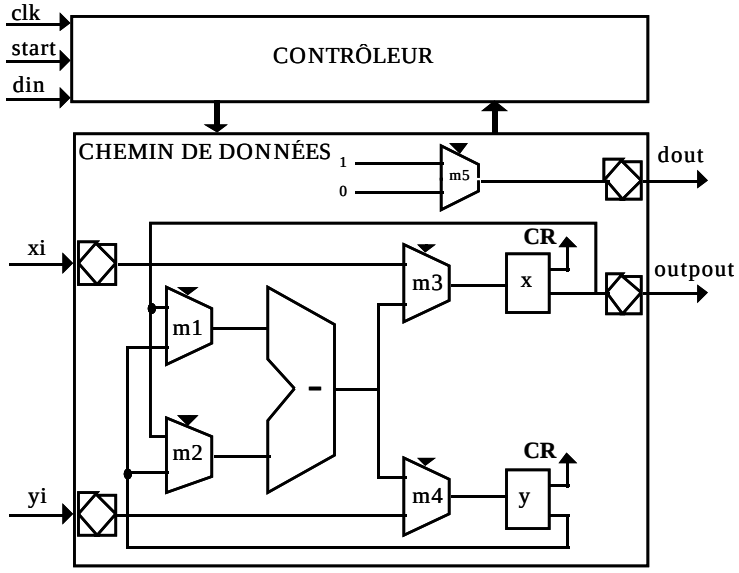


Figure II.7 : Architecture RTL à base de multiplexeurs du PGCD

La distinction entre une architecture cible pour des applications dominées flot de données et des applications dominées flot de contrôle se situe au niveau des importances respectives du contrôleur et du chemin de données. Dans le cas des applications flot de données, la synthèse de la partie opérative est généralement dominante. Les étapes cruciales sont l'allocation et la génération de la partie opérative. Le contrôle réalise une séquence statique d'opérations. Dans le cas des circuits de contrôle, la synthèse du contrôle est dominante. Le contrôleur peut comporter des dizaines, voire des centaines d'états et réalise généralement une séquence complexe des opérations exécutées par la partie opérative. La séquence peut être dynamique, c'est-à-dire dépendre des données.

2.3. Différents types d'ordonnancement

On a vu au chapitre II, §1.2, que la première étape dans la synthèse de haut niveau est de déduire une représentation interne à partir de la description d'entrée. Différents types de représentation existent. Le choix de cette représentation dépend des algorithmes de synthèse utilisés et en particulier de l'algorithme d'ordonnancement (chapitre II, §1.3). L'ordonnancement est la tâche la plus complexe dans la synthèse comportementale. Elle définit l'ordre d'exécution des opérations de la description comportementale.

L'ordonnancement associe chaque opération du comportement à un point du temps [17]. Dans les systèmes synchrones, on mesure le temps en étapes de contrôle. L'ordonnancement a pour objet d'optimiser le nombre d'étapes de contrôle nécessaires à l'exécution d'une fonction, dans la limite des ressources matérielles disponibles et du temps de cycle [5]. Un algorithme d'ordonnancement doit tenir compte des constructions de contrôle telles que, les boucles et les branchements conditionnels, les dépendances de données exprimées dans un graphe flot de données, et les contraintes sur les ressources matérielles. Dans les circuits synchrones, les contraintes de base portent sur l'utilisation unique d'une même ressource matérielle au cours d'une étape de contrôle. Autrement dit, au cours d'une étape de contrôle, les registres ne peuvent être chargés qu'une seule fois, la logique combinatoire ne peut évaluer qu'une seule fois (les rétroactions sont interdites). Les bus ne peuvent porter qu'une seule valeur pendant une étape de contrôle. D'autres contraintes sur un circuit peuvent spécifier la taille, le temps de cycle et la consommation.

De façon générale les algorithmes d'ordonnancement peuvent être classés dans deux catégories, l'ordonnancement orienté flot de données et l'ordonnancement orienté flot de contrôle. On les définit comme tous les algorithmes utilisant respectivement un graphe de flot de données ou un graphe de flot de contrôle. Pour montrer les principales différences entre les deux classes, prenons deux approches d'ordonnancement, l'une utilisant un graphe flot de données (GFD), l'autre un graphe flot de contrôle (GFC) comme l'illustre la figure II.8 [26].

L'ordonnancement d'un GFD consiste à affecter chaque opération à une étape de contrôle tout en respectant les contraintes. Les contraintes fixent soit le nombre d'étapes de contrôle, soit le nombre de ressources à utiliser. Une opération ne peut être affectée qu'à une seule étape de contrôle. Un GFD est la représentation la plus parallèle. Le parallélisme découle des dépendances de données parmi les opérations. L'ordonnancement orienté flot de données réduit parfois le nombre de parallélismes à cause des contraintes.

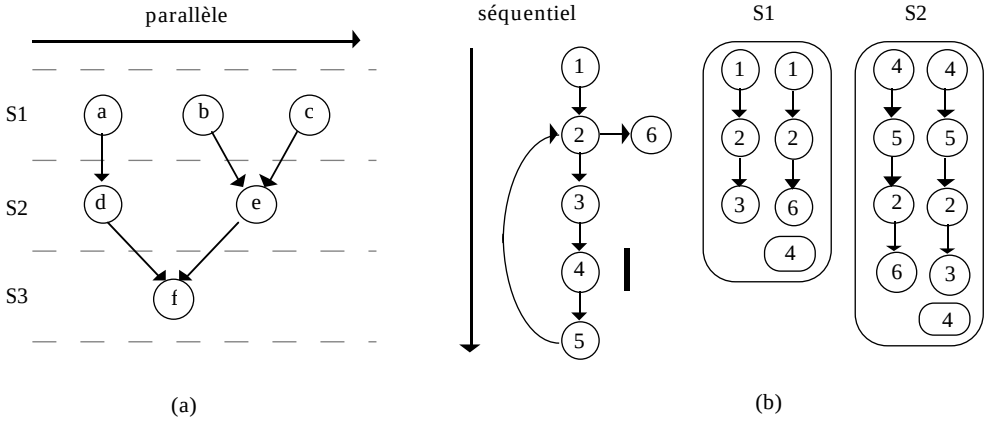


Figure II.8 : Ordonnancements (a) flot de données (b) flot de contrôle

Walker fait l'inventaire dans [16], des principaux algorithmes d'ordonnancement orienté flot de données, développés et intégrés dans des outils de synthèse de haut niveau:

- l'ordonnancement ASAP, (*As Soon As Possible*), attribue toutes les opérations aux étapes de contrôle le plus tôt possible.
- l'ordonnancement par liste [27] attribue les opérations à des étapes de contrôle, étape par étape, en prenant en compte le nombre de ressources disponibles. Il construit une liste d'opérateurs à données disponibles qu'il trie suivant une fonction de priorité pouvant porter sur le temps ou les ressources.
- l'ordonnancement orienté par les forces [28] équilibre le nombre d'opérations dans chaque étape de contrôle.

Contrairement à l'ordonnancement orienté flot de données, l'ordonnancement orienté flot de contrôle utilise un GFC comme représentation sous-jacente. Un GFC est par nature une représentation séquentielle de la description comportementale. Ainsi, l'ordonnancement orienté flot de contrôle extrait le maximum de parallélisme tout en respectant les contraintes. Il consiste à affecter une séquence d'opérations à une étape de contrôle, puis une opération peut être affectée à plus d'une étape de contrôle comme les opérations 3 et 6 de la figure II.8(b).

Le tableau II.1 liste le vocabulaire relatif à l'ordonnancement qui est utilisé par la suite dans ce manuscrit. Il s'agit de noms ou d'expressions usuellement connus sous leur dénomination anglaise.

- Le **chaînage** (en Anglais *chaining*) de deux opérations qui se suivent consiste à les attribuer à une même étape de contrôle. Elles s'exécutent en séquence.

- Le **multi-cycle** (en Anglais *multi-cycling*) s'effectue lorsqu'une opération requiert un temps supérieur à la période de l'horloge. Le compilateur prend alors l'initiative d'étendre l'opération sur différents cycles.

- Le **pipeline du chemin de données** (en Anglais *data path pipelining*) consiste à insérer un étage de registres à l'intérieur du chemin de données.

- Le **pipeline du contrôle** (en Anglais *control pipelining*) consiste à insérer une barrière temporelle entre le contrôleur et le chemin de données. Ainsi lorsque la partie contrôle évalue une condition dans un cycle d'horloge, la partie opérative exécute les opérations qui en dépendent dans le cycle suivant.

- Le **repliement de boucle** (en Anglais *loop folding* ou *loop pipelining* ou *software pipelining*) est particulièrement intéressant lorsque les spécifications algorithmiques ou comportementales contiennent de nombreuses constructions de boucles. Ces boucles peuvent limiter le débit des données si elles sont ordonnancées en séquence. En considérant le corps d'une boucle comme une unique opération multi-cycle, l'ordonnancement insère un pipeline dans l'exécution d'une boucle toutes les N+1 itérations avant la fin de la $N^{ème}$ itération. Cette technique qui augmente le parallélisme entre les tâches à ordonnancer, peut considérablement améliorer le débit.

- Le **déroulement de boucle** (en Anglais *loop unrolling*) consiste à dérouler une boucle. Il peut se faire dans le temps ou sur le matériel.

Tableau II.1 : Vocabulaire relatif à l'ordonnancement.

3. La synthèse comportementale dans la boucle de conception

Cette section traite des principaux problèmes liés à l'introduction de la synthèse comportementale dans le processus de conception. Parmi les points critiques, citons la correction des bogues des descriptions comportementales, le lien avec la synthèse logique, mais aussi la difficulté à décrire la fonctionnalité complète au niveau comportemental, et la validation des résultats de la synthèse comportementale.

La figure II.9 montre un flot de conception à base de synthèse architecturale avec les procédés de vérification correspondant et les boucles de conception.

A partir du niveau de spécification initial, on commence par un partitionnement manuel du circuit, puis on le décrit au niveau comportemental. La description obtenue est simulée (validation B) pour vérifier la fonctionnalité globale. La boucle B1 permet d'en corriger les bogues.

A partir du niveau comportemental, la synthèse architecturale produit automatiquement une architecture RTL du circuit. La simulation de la description RTL (validation R) vérifie le comportement au niveau du cycle d'horloge. La boucle "R1" sert à corriger des bogues de la description comportementale en considérant le style d'écriture accepté par la synthèse architecturale. La boucle "R2" met au point les directives et autres entrées de l'outil de synthèse pour reproduire la fonctionnalité d'entrée avec les composants adéquats de la bibliothèque.

A partir du niveau transfert de registre, l'architecture du circuit est transposée sur un assemblage de cellules interconnectées. Puis elle est optimisée par un outil de synthèse logique suivant les directives manuelles. La simulation de la description au niveau portes (validation G) vérifie les délais. La boucle "G1" corrige les bogues de la description comportementale pour les initialisations manquantes de variables et de signaux, non détectées pendant la validation "R". La boucle "G2" ajuste les directives pour la synthèse comportementale pour l'optimisation des performances, comme le pipeline du contrôle. La boucle "G3" met au point les directives pour la synthèse logique pour l'optimisation des performances telle que le replacement des registres (pour *retiming*) ou l'insertion d'un pipeline à l'intérieur du chemin de données des unités fonctionnelles. Il s'ensuit le placement-routage du circuit à partir du niveau portes.

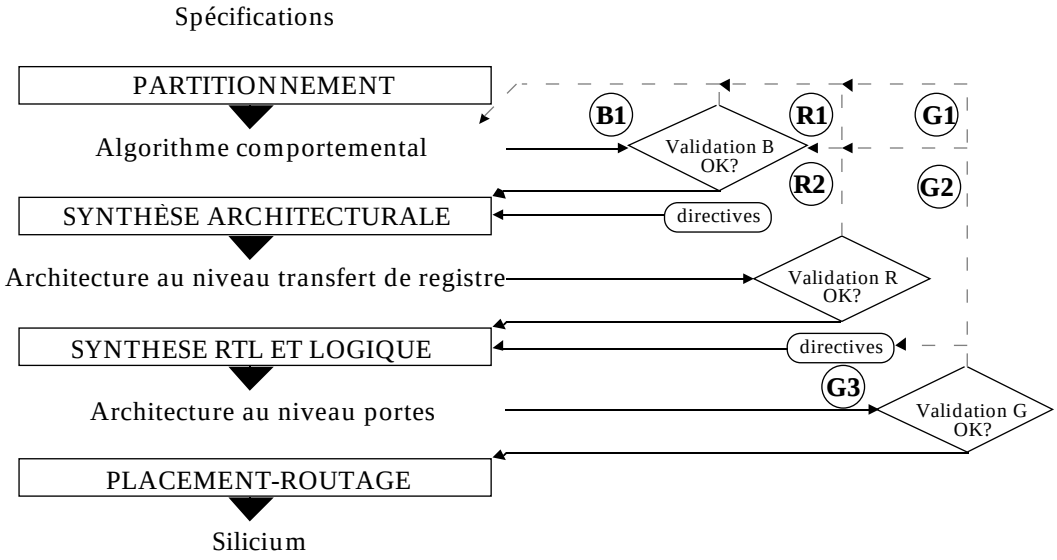


Figure II.9 : Les boucles de conception.

En raison du niveau d'abstraction et de la quantité d'informations manipulées, la simulation comportementale (validation B) prend moins de temps que la simulation RTL (validation R), elle-même beaucoup plus rapide que la simulation "portes" (validation G). Pour les mêmes raisons, la synthèse comportementale est plus rapide que la synthèse logique.

La boucle B1 inclut la simulation comportementale, une analyse des résultats de simulation et une éventuelle modification de la description comportementale. Les différentes tâches correspondant aux différentes boucles figurent dans le tableau II.2.

	Simulation comport.	Mise à jour des spécifications comport.	Synthèse comport.	Mise à jour des directives comport.	Simulation RTL	Synthèse logique	Mise à jour des directives de synthèse logique	Simulation au niveau portes
B1	x	x						
R1		x	x		x			
R2			x	x	x			
G1		x	x			x		x
G2			x	x		x		x
G3			x			x	x	x

Tableau II.2 : Tâches de conception dans les boucles de conception

Selon le tableau II.2, il est facile de voir que, la boucle comportementale B1 est plus rapide que les boucles RTL (R1, R2), elles-mêmes beaucoup plus rapides que les boucles "portes" (G1, G2, G3).

D'un autre côté, il est facile de montrer que chaque boucle comportementale peut éviter plusieurs boucles RTL et une boucle RTL peut éviter plusieurs boucles "portes". En fait, les bogues sont plus faciles à détecter et à corriger aux niveaux supérieurs.

Il est alors clair que l'utilisation de la synthèse comportementale contribue à réduire le temps de conception. Même lorsque l'on ne l'utilise pas, de nombreux concepteurs préfèrent commencer avec une description comportementale pour vérifier les spécifications du circuit. Ainsi, plusieurs itérations de la boucle B1 précèdent la conception RTL.

Cependant, en pratique, sans synthèse architecturale, la description comportementale n'est pas mise à jour pendant le processus de conception. Le concepteur décrit à la main l'architecture RTL. Puis il corrige les bogues détectés, non plus dans la description comportementale, mais dans la description RTL. Bien que la conception débute avec une description comportementale, après quelques itérations dans les processus de conception, celle-ci devient obsolète et la description RTL devient le modèle de référence.

Par contre avec la synthèse architecturale, les modèles de niveau inférieur sont produits automatiquement. Dans ce cas, la description comportementale demeure le modèle de référence pendant tout le processus de conception.

4. Les outils de synthèse de haut niveau

Il existe trois catégories de logiciels pour la synthèse de haut niveau [12]:

- les outils de synthèse comportementale qui acceptent une description algorithmique et en réalisent automatiquement l'ordonnancement et l'allocation des ressources multi-cycles;
- les outils comportementaux interactifs qui fournissent une assistance pour l'ordonnancement et l'allocation des ressources multi-cycles;
- les outils de génération de code graphiques qui acceptent un diagramme bloc et délivrent une description en VHDL ou en Verilog compatible avec les outils de synthèse logique.

Dans le cadre de cette thèse nous nous intéressons exclusivement aux outils à entrée textuelle, c'est-à-dire aux deux premières catégories énoncées ci-dessus. Par contre nous adopterons plutôt la même classification que celle qui existe pour les circuits, outils orientés flot de données ou DSP, outils orientés flot de contrôle et outils à application générale. Certains outils sont évalués ou même utilisés dans l'industrie [29, 30]. Le chapitre III étudie en détail trois d'entre eux, représentatifs des tendances générales. Le tableau II.3 liste quelques outils orientés flot de contrôle. Le tableau II.4 liste quelques outils orientés flot de données.

• *SAW*	de l'université de Carnegie Mellon [31],
• *Amical*	de TIMA - INP-Grenoble [9],
• *Olympus*	de l'université de Stanford [32],
• *VSS-ISE*	de l'université d'Irvine,
• *Adam*	de USC [33],
• *Callas / Caddy*	de l'université de Karlsruhe [34],
• *Mimola*	de l'université de Dortmund [35],
• *Scoop*	de l'université de Montpellier,
• *CAPSYS*	de I3S à Nice [36],
• *Alliance*	du MASI, Université de Paris [37],
• *HIS*	d'IBM [38],
• *Behavioral Compiler*	de Synopsys Inc. [11]
• *Visual Architect*	de Cadence Alta Group,
• *View Schedule*	de Viewlogic Systems, Inc.,
• *ArchGen*	de CAE Plus.

Tableau II.3 : Outils de synthèse orientés flot de contrôle.

• *Cathedral-2/3*	de l'IMEC [10, 25],
• *GAUT*	du CNET [39],
• *Lager*	de l'université de Berkeley,
• *Behavioral Compiler*	de Synopsys Inc. [11],
• *Mistral 2 DSP Compiler*	de Mentor Graphics Corp.,
• *COSSAP*	de Synopsys Inc.,
• *SPW*	de Cadence Alta Group.

Tableau II.4 : Outils de synthèse de orientés flot de données.

Actuellement dans l'industrie, on fait très peu appel aux outils de synthèse architecturale. *HIS, Mistral, Phideo, Behavioral Compiler* sont utilisés. Mais ils sont pour la plupart issus de la recherche. Les orientations de recherche ont eu pour conséquence de spécialiser ces outils qui ne couvrent donc pas tous les aspects nécessaires de la synthèse architecturale. Les industriels préfèrent donc se fier à des outils commerciaux, encore rares, dont la maintenance est assurée. Dans §5, nous tentons de décrire les requêtes de l'industrie vis-à-vis de la synthèse de haut niveau.

5. Les besoins industriels

Cette section s'appuie sur l'expérience d'introduction de la synthèse architecturale à ST. Il s'agit de formuler les besoins que doivent satisfaire les outils de synthèse comportementale avant de pouvoir être utilisés dans l'industrie.

Les principaux besoins de ST portent sur le type de description à admettre en entrée et à délivrer en sortie. Ils portent aussi sur la prise en compte de bibliothèques de composants et des informations incluses, en particulier le traitement des mémoires. Ils portent enfin sur les capacités de l'outil à explorer plusieurs solutions architecturales, et sur les optimisations possibles. Mais l'introduction de la synthèse comportementale dans le flot de conception automatique industriel ne se justifie que par un gain de productivité prouvé.

5.1. Descriptions d'entrée

Tout d'abord le langage de la description d'entrée doit être normalisé comme C ou VHDL ou répandu comme Verilog, de façon à pouvoir être compatible avec des outils de validation disponibles. Les descriptions en entrée et en sortie de l'outil doivent aussi pouvoir s'intégrer dans le même environnement de simulation.

Comme pour la synthèse logique, le style de la description comportementale d'entrée est déterminant pour obtenir un résultat optimal. En effet, il existe parfois plusieurs façons de décrire un même fonctionnement. Un filtrage du style de la description d'entrée de l'outil économiserait du temps de conception pour guider le concepteur vers la solution la plus intéressante. De plus, un outil de synthèse comportementale ne doit pas se limiter au seul comportement, mais également prendre en compte les parties décrites au niveau transfert de registre, de façon à unifier le flot de conception pour toutes les parties matérielles et ainsi à conserver les mêmes informations durant tout le flot de conception.

5.2. Gestion des mémoires

La gestion des mémoires est particulièrement contraignante dans l'industrie, car chaque société de fournisseur de silicium dispose généralement de sa propre bibliothèque de mémoires. L'outil doit donc accepter tous les types de mémoire.

Par ailleurs, les modèles de ces mémoires sont complexes et par conséquent ralentissent les étapes de validation. Pour diminuer le temps de validation, on peut profiter du niveau comportemental pour manipuler une mémoire comme un tableau ou un enregistrement. On doit permettre des lectures simultanées et mêmes des écritures simultanées, lorsque c'est fonctionnellement possible. C'est à l'outil de synthèse de décider d'après la bibliothèque de mémoires disponibles comment organiser la ou les mémoires et d'insérer des cycles là où il le faut, pour reproduire les mêmes séquences d'accès. Les outils courants de synthèse comportementale (*Amical*, *Cathedral-2/3*, *Behavioral Compiler*) requièrent une description VHDL, Verilog ou Silage avec des structures de mémoires et leur gestion totalement spécifiées.

5.3. Optimisations et exploration architecturale

L'outil doit pouvoir supporter plusieurs types d'ordonnancement, tout en guidant le concepteur vers la meilleure solution. Les optimisations doivent porter en priorité sur la performance, sur la puissance puis sur la surface. La testabilité devrait pouvoir être optimisée et facilitée. Il faut noter que très peu d'outils de synthèse comportementale traitent le problème du test [40, 41, 42] et que les optimisations sont généralement orientées vers la surface. Ils sont également peu nombreux à prendre en compte l'optimisation de la consommation [42, 43].

L'outil doit pouvoir intégrer les paramètres utiles au niveau transfert de registre de toute bibliothèque de portes ou de modules déjà existants. Ces paramètres permettent de premières estimations, absolues sinon relatives, en surface, durée du cycle, nombre de cycles, consommation, occupation des ressources, etc. Ces estimations servent de référence aux algorithmes d'optimisation de l'architecture. La synthèse devrait intégrer plusieurs types d'optimisation. Mais le choix de l'algorithme, s'il s'impose, doit dépendre des directives du concepteur.

La description de sortie doit absolument avoir le même comportement fonctionnel que la description d'entrée indépendamment des directives de synthèse du concepteur. En d'autres termes, il faut interdire toutes les transformations pouvant changer le comportement.

Finalement il faut noter l'importance de l'interactivité. Ce concept permet de prendre en compte l'intelligence et l'expérience du concepteur durant le processus de synthèse. Il arrive souvent que le concepteur débute la conception de son circuit avec une idée en tête. Dans ce cas il est important de pouvoir guider l'outil pour atteindre cette solution. Il peut arriver aussi que le concepteur ait des idées pour améliorer une architecture produite automatiquement. Là aussi, il faut pouvoir guider l'outil de synthèse pour réaliser l'optimisation voulue. Très peu d'outils de synthèse architecturale permettent une interaction avec le concepteur [9].

5.4. Gain en temps de conception

L'introduction de la synthèse comportementale dans un flot de conception automatique préexistant ne se justifie que par un gain important en productivité. Comme on l'a vu au §3, le flot est théoriquement plus rapide lorsqu'il comprend la synthèse architecturale. Mais il ne faut pas oublier le temps supplémentaire nécessaire pour la gestion de l'outil-même.

Un outil de synthèse comportementale doit posséder les qualités généralement requises pour tout outil de CAO. Il doit être compatible en entrée comme en sortie avec les environnements existants et facile à installer. Il ne doit pas nécessiter un apprentissage supérieur à deux mois et par conséquent doit être facile à utiliser. Le temps CPU utile à la synthèse-même ne doit pas excéder quelques minutes, si la conception d'un circuit nécessite plusieurs passages par la boucle R2 de la figure II.9. La sauvegarde de scripts mis au point lors de l'exploration architecturale est essentielle ainsi que la possibilité d'exécuter la synthèse en tâche de fond.

Le temps perdu à l'addition d'un nouvel outil dans le flot de conception doit au moins être compensé par le temps gagné dans la mise au point plus souple des modèles comportementaux. Le temps de simulation d'un modèle comportemental doit être inférieur au temps de simulation de référence, (le temps de simulation de référence est celui du même modèle, décrit à la main au niveau transfert de registre). Par ailleurs, le temps de simulation du modèle RTL issu de l'outil doit être équivalent au temps de référence. Des descriptions trop hiérarchiques ou incluant trop d'objets de type signal, composant, etc. peuvent ralentir la validation.

La vérification de la description RTL est une étape nécessaire pour valider le résultat de la synthèse de haut niveau. Par contre il est essentiel que cette vérification utilise le même jeu de test qui a servi à la validation de la description comportementale. La figure II.10 illustre ce schéma de vérification.

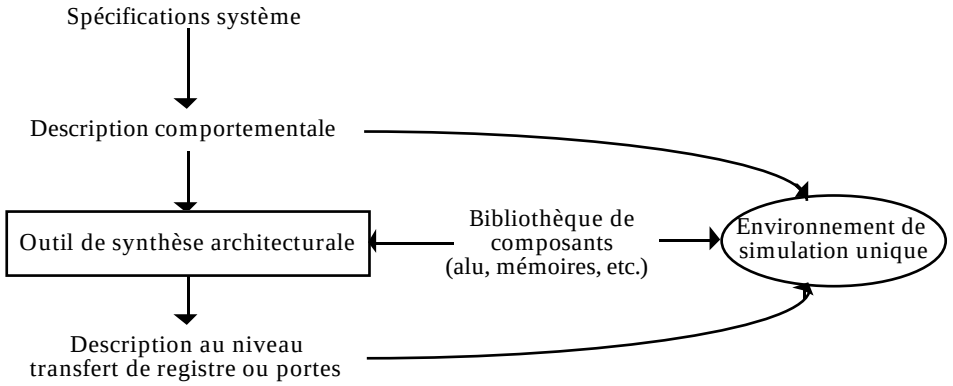


Figure II.10 : Vérification des résultats de la synthèse architecturale.

Enfin et surtout, il doit être plus facile de corriger les bogues des descriptions en entrée comme en sortie de l'outil, que de corriger les bogues du même modèle, décrit à la main au niveau transfert de registre. Même si l'outil produit automatiquement la description de sortie, celle-ci doit être lisible et faire référence à la description d'entrée.

6. Conclusion

Ce chapitre a décrit la synthèse comportementale, ses fonctions et son insertion dans le flot de conception. Nous y avons rappelé les besoins industriels après avoir dressé l'état de l'art des outils. Les principaux besoins de l'industrie portent sur le style des descriptions à l'entrée et à la sortie de l'outil, la compatibilité de ces descriptions avec un flot de conception préexistant, la gestion des mémoires, et les possibilités d'exploration et d'optimisation architecturale. Malheureusement, peu d'outils existants couvrent ces besoins. Par contre, comme il sera détaillé dans le chapitre suivant, la combinaison de plusieurs outils de synthèse dans un même flot de conception peut donner des résultats satisfaisants.

Chapitre III : Trois Types d'Outils Existants

Ce chapitre décrit trois outils de synthèse de haut niveau applicables dans l'industrie et traite de leurs domaines d'application respectifs. En effet, ces outils sont représentatifs des trois principales classes d'outils de synthèse architecturale : les outils orientés flot de contrôle, les outils orientés flot de données et les outils à orientation mixte. Nous présentons ici un outil de recherche orienté flot de contrôle, *Amical*, un outil de recherche orienté flot de données, *Cathedral-2/3* et un outil commercial à orientation mixte, *Behavioral Compiler*. Nous insistons sur les aspects de validation des descriptions en entrée et en sortie de chaque outil. Un tableau final récapitule leurs caractéristiques respectives. Nous concluons sur l'intérêt d'une conception combinant plusieurs outils complémentaires.

Le choix de ces outils s'explique par les raisons suivantes. Initialement, le travail de cette thèse s'est fondé sur l'utilisation d'*Amical* et de *Cathedral-2/3* pour concevoir un circuit test : l'Estimateur de Mouvement [44]. *Behavioral Compiler* n'était alors pas disponible sur le marché. Cet outil a fait son apparition au cours du projet et a fait l'objet d'une évaluation [45] à ST sur le même circuit test pour comparer les deux flots de conception.

1. *Amical*

Le système de synthèse architecturale *Amical* [5,9] s'adresse aux applications dominées par le flot de contrôle pour les réaliser sur des ASICs ou des FPGAs. L'outil est compatible avec les environnements de simulation et de synthèse logique existants. En partant d'une description purement VHDL, *Amical* produit une description destinée aux outils de synthèse logique existants. La figure III.1 montre que la description d'entrée doit contenir un processus décrit en VHDL comportemental, séquentiel *P1*. Eventuellement d'autres processus RTL *P2*, ainsi que des instructions flot de données ou structurelles concurrentes *P3* peuvent être intégrés.

Amical permet l'utilisation d'unités fonctionnelles complexes ainsi que la réutilisation de composants existants dans un nouveau circuit, grâce à une bibliothèque externe d'unités fonctionnelles. L'interactivité de l'outil accroît les possibilités d'exploration de l'espace des solutions architecturales en temps réel, d'autant plus que le temps de réponse d'*Amical* est assez court, généralement de quelques secondes.

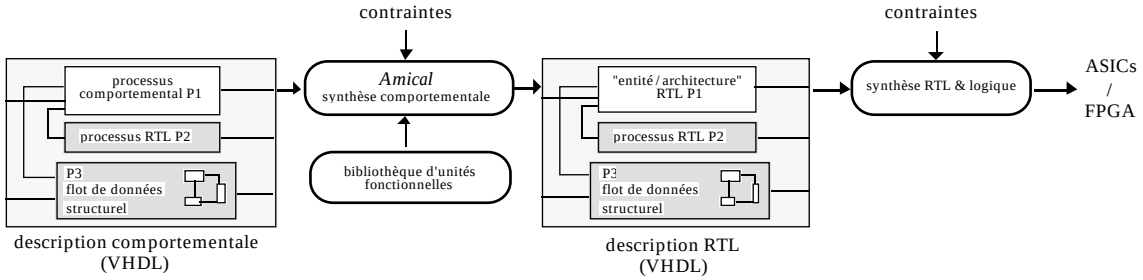


Figure III.1 : Flot de conception utilisant Amical

1.1. Domaine d’application

Amical s'adresse aux applications dominées flot de contrôle pour les réaliser sur des ASICs ou des FPGAs. L'outil est dédié aux contrôleurs séquentiels dont la description du comportement fait appel à des boucles dépendant des données, à des compteurs, à des interruptions de traitement, ou à des protocoles asynchrones. D'autres comportements, comme ceux que l'on trouve dans les applications de traitement du signal, pourraient être décrits en entrée de l'outil. Mais pour obtenir des résultats efficaces, le niveau de raffinement nécessaire rejoindrait celui d'une description RTL.

Plusieurs exemples ont déjà servi à l'évaluation d'*Amical*, et ont donné de bons résultats. Il s'agit d'un contrôleur de répondeur téléphonique et d'un système de décodage MPEG-AUDIO [46]. Bien que ce soit de grands circuits, leur synthèse architecturale requiert moins de 5 minutes.

1.2. Architecture produite par *Amical*

Amical transforme une description comportementale VHDL en une architecture contenant un contrôleur et un chemin de données (figure III.2). Le contrôleur est une Machine d’Etats Finis (MEF). Le chemin de données se compose d’unités fonctionnelles allouées et connectées entre elles ainsi qu'avec le monde extérieur, par l’intermédiaire d’un réseau de communication. Ce dernier est composé de registres, de bus, d'interrupteurs, de multiplexeurs, etc.

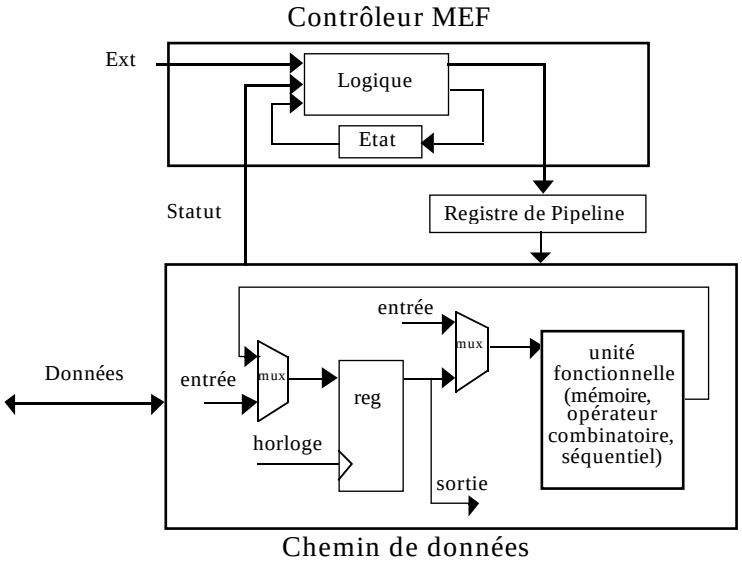


Figure III.2 : Architecture produite par Amical

1.2.1. Contrôleur

Le contrôleur agit comme un contrôleur central synchrone. Il séquence les opérations exécutées par les unités fonctionnelles et le réseau de communication. La synthèse produit automatiquement le contrôleur, représenté par une machine d'états finis "symbolique". Pour chaque cycle élémentaire, le chemin de données (unités fonctionnelles et réseau de communication) reçoit une nouvelle commande. Une commande peut inclure l'activation de plusieurs chemins ou transferts.

1.2.2. Chemin de données

L'architecture du chemin de données est parallèle. Elle peut inclure plusieurs unités fonctionnelles travaillant en même temps. Le degré de parallélisme est déterminé automatiquement ou manuellement, pendant la synthèse. Une unité fonctionnelle peut exécuter plusieurs fonctions multi-cycles. Les unités fonctionnelles interagissent à travers un réseau de communication composé de bus, d'interrupteurs, de multiplexeurs et de registres.

L'architecture est hétérogène puisqu'elle permet de mélanger des composants provenant d'autres environnements de conception. Une unité fonctionnelle peut être le résultat d'un générateur de modules, le résultat d'un outil de synthèse architecturale comme *Amical* ou *Cathedral-2/3* ou un composant matériel existant produit manuellement. Ces unités fonctionnelles peuvent réaliser différentes sortes d'opérations comme les opérations arithmétiques, la mémorisation, le calcul d'adresse, des opérations de traitement du signal, d'entrées / sorties, etc.

Le réseau de communication se compose de bus, d'interrupteurs et de registres dans le cas d'une architecture à base de bus. Il se compose de multiplexeurs et de registres dans le cas d'une architecture à base de multiplexeurs. Le réseau est construit de façon à permettre la communication entre les unités fonctionnelles et le monde extérieur. Les transferts parallèles requis par l'architecture déterminent le nombre de bus.

Le concept d'unités fonctionnelles d'*Amical* permet d'inclure des circuits, complexes pour leur réutilisation, comme unités fonctionnelles. Actuellement, le système restreint les opérations exécutées par chaque unité fonctionnelle. Il exige que chaque opération ait un temps d'exécution statique prévisible. Avec un tel schéma, une opération complexe à temps d'exécution variable (ex : opération incluant des calculs dépendant des données et / ou des instructions d'attente asynchrones) doit être décomposée en plusieurs opérations (ex : opération de déclenchement, de vérification de fin de calcul et de récupération des résultats) ayant un temps d'exécution fixe.

1.3. Flot de synthèse

La figure III.3 illustre le flot de synthèse par *Amical*. La description comportementale est un processus VHDL utilisant des sous-systèmes complexes à travers des appels de procédure et de fonction. Cependant, au moins une des unités fonctionnelles disponibles dans la bibliothèque doit pouvoir exécuter chaque procédure ou fonction utilisée. Pendant les différentes étapes de la synthèse architecturale, *Amical* considère les unités fonctionnelles comme des boîtes noires. Les seules informations requises pour chaque unité fonctionnelle concernent les opérations qu'elles sont capables d'exécuter et le protocole qu'elles suivent. Cependant, comme l'outil est indépendant des bibliothèques de cellules, on ne peut obtenir d'estimation réalistes en performance et surface.

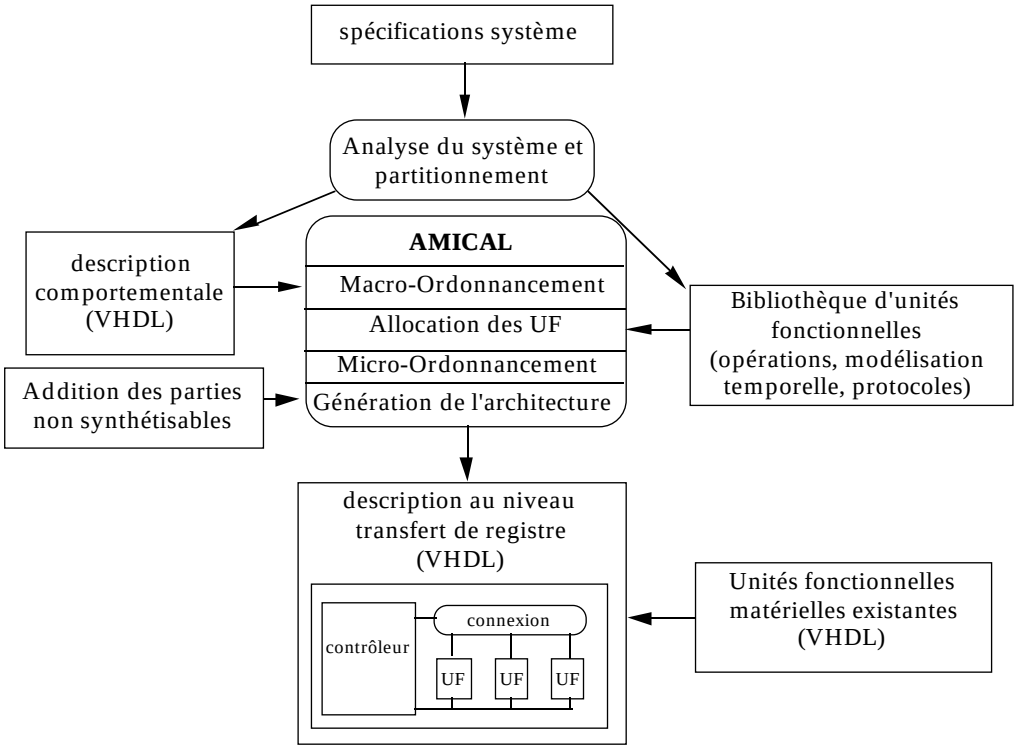


Figure III.3 : Flot de synthèse dans Amical.

1.3.1. Les étapes de synthèse

Les différentes étapes de la synthèse sont le macro-ordonnancement, l'allocation des unités fonctionnelles, le micro-ordonnancement, et la génération de l'architecture.

Le macro-ordonnancement applique l'algorithme de boucle dynamique [47]. C'est un ordonnancement à base de chemins. L'outil d'ordonnancement lit la description VHDL et produit une machine d'états finis représentée comme une table de transitions. Chaque transition correspond à l'exécution d'une étape de contrôle pour une condition donnée. Toutes les opérations de la transition donnée peuvent être exécutées en parallèle. Une opération peut correspondre à une opération standard en VHDL (telle que +, -) ou à un appel de procédure.

A ce stade, l'exécution de chaque opération peut nécessiter plusieurs cycles d'horloge. On désigne aussi une transition par macro-cycle ou macro-étape. En fait, une transition correspond à un simple graphe flot de données, synthétisé par les étapes classiques qui sont l'ordonnancement et l'allocation. L'objectif de ces étapes est de décomposer chaque transition (appelée également macro-cycle) en un ensemble d'étapes de contrôle de base s'exécutant chacune en une seule étape de contrôle. On désigne aussi par micro-cycles ces étapes de contrôle de base.

Après l'ordonnancement, la synthèse architecturale débute avec deux sortes d'informations. La description ordonnancée (un ensemble de graphes de flot de données) et une bibliothèque externe d'unités fonctionnelles. Les étapes suivantes réalisent l'allocation et l'affectation. L'étape d'allocation des unités fonctionnelles fait appel à un algorithme de type EMUCS [48]. Elle associe une unité fonctionnelle à chaque opération dans la table d'états. Un second ordonnancement, appelée micro-ordonnancement, applique l'algorithme ASAP à la représentation obtenue en respectant le schéma

d'exécution de chaque opération. Chaque opération est décomposée en un ensemble de transferts ordonnancés dans des micro-cycles. Chaque micro-cycle contient un ensemble de transferts parallèles dont l'exécution prend un cycle d'horloge de base.

La dernière étape de synthèse est une génération d'architecture classique. La description au niveau du cycle est transposée sur une architecture composée d'un chemin de données et d'un contrôleur. Cette étape inclut la génération du chemin de données (allocation des communications, c'est-à-dire des multiplexeurs et des bus) et la génération du contrôleur.

1.3.2. Optimisations

Amical optimise le nombre d'unités fonctionnelles en fonction des opérations qu'elles sont capables d'exécuter et de leurs paramètres physiques (surface, consommation), lors des étapes successives d'ordonnancement et d'allocation imbriquées.

Comme l'outil n'analyse pas la durée de vie des variables, il n'effectue pas d'optimisation du nombre des registres. Cette optimisation reste manuelle au niveau de la description comportementale.

Dans le cas d'une architecture à base de multiplexeurs, le nombre et la taille des multiplexeurs ne sont pas optimisés.

Le pipeline du contrôle est possible (tableau II.1 et figure III.2).

1.3.3. Synchronisation

Le contrôleur et le réseau de communication de l'architecture produite ne peuvent être synchronisés que par une seule horloge. Cependant le reste de l'architecture peut contenir d'autres horloges.

Le signal de mise à zéro peut être synchrone ou asynchrone. S'il est synchrone, la description résultante ne pourra y réagir qu'au premier cycle de chaque étape de contrôle. En effet, une étape de contrôle peut durer plusieurs cycles d'horloge alors que la condition dans laquelle est testé le signal de mise à zéro n'est évaluée que dans le premier cycle.

1.3.4. Validation

La description d'entrée de l'outil *Amical* consiste en une "entité-architecture" VHDL comprenant obligatoirement un processus à synthétiser. L'écriture de ce processus utilise un sous-ensemble du style comportemental du langage VHDL. Cette restriction provient des limitations du langage de description interne SOLAR [49] (ex : non-admission des opérations sur des bits) ainsi que des limitations de traitement de l'outil (ex : non-admission des fonctions de conversion). Cette limitation a été levée dans la dernière version de SOLAR [50].

La validation de la description comportementale est généralement l'étape qui nécessite le plus de temps, durant la conception d'un circuit (figure I.3); surtout dans le cas de circuits de taille importante dont les spécifications ne sont pas figées. Elle sert d'abord à mettre au point le comportement du circuit, puis à vérifier la non-régression du comportement après adaptation de la description à la synthèse par *Amical*. Le style séquentiel et la concision d'écriture au niveau comportemental facilitent le suivi des spécifications changeantes grâce à la flexibilité et lisibilité du code, par rapport à une description au niveau transfert de registre. Cette concision facilite donc les corrections et réduit le temps de compréhension d'une description pour sa réutilisation.

Amical produit une description VHDL au niveau transfert de registre. Une machine d'états finis de Mealy sous la forme d'un "*case*" sur son état courant décrit le contrôleur. La partie opérative est une structure contenant : les composants de base (registres, multiplexeurs, etc.), les unités fonctionnelles instanciées, ainsi que leurs interconnexions. A ce niveau, les noms des signaux incluent les noms des signaux et variables apparaissant au niveau comportemental. Cette description, produite par l'outil automatique, est aisément lisible. Cependant elle compte trois niveaux de hiérarchie, ce qui peut affecter les performances de la synthèse logique. La dernière version d'Amical adopte une nouvelle structure avec moins de niveaux de hiérarchie dans le VHDL RTL produit.

La validation de la description RTL s'avère nécessaire, dans la mesure où l'ordonnancement a pu insérer des cycles par rapport au fonctionnement du modèle comportemental. Ce problème est évoqué par Bergamaschi dans [15]. Cette transformation intervient si la description comportementale contient des dépendances de données ou des opérations sur des variables non chaînées par l'outil. La lisibilité de la description de sortie d'*Amical* facilite la localisation fonctionnelle des bogues, par la visualisation possible de la valeur de l'état courant du contrôleur. Mais le problème reste d'identifier la source des bogues. Lorsque celle-ci est trouvée, seule une connaissance approfondie de l'outil permet de corriger la description d'entrée ou les scripts de synthèse.

2. *Cathedral-2/3*

Cathedral-2/3 [10, 25, 51] est un outil de synthèse d'algorithmes de traitement de signal ou DSP (*Digital Signal Processing*), synchrones et testables, en ASICs, à partir d'une description de haut niveau. *Cathedral-2/3* se classe parmi les outils orientés flot de données. L'utilisation de *Cathedral-3* a montré que les temps de synthèse varient de quelques minutes à une heure suivant la complexité du circuit.

La figure III.4 illustre le flot de conception utilisant *Cathedral-2/3*. La description d'entrée est écrite en langage Silage sous forme d'instructions concurrentes. Un fichier de directives contraint l'outil à réaliser les instructions sur deux types d'opérateurs. Le premier type désigne les EXUs (*Execution Unit*) issues de la bibliothèque de l'outil. Le second type désigne les opérateurs spécifiques rapides appelés ASUs (*Application Specific Unit*) et créés sur demande. *Cathedral-2* [10] produit alors une architecture à base d'un contrôleur micro-programmé commandant des unités fonctionnelles dans un chemin de données. Parmi celles-ci, on trouve les ASUs détectées lors du flot de conception et synthétisées puis optimisées par *Cathedral-3* [51]. La description de sortie est produite en VHDL au niveau transfert de registre ou au niveau portes.

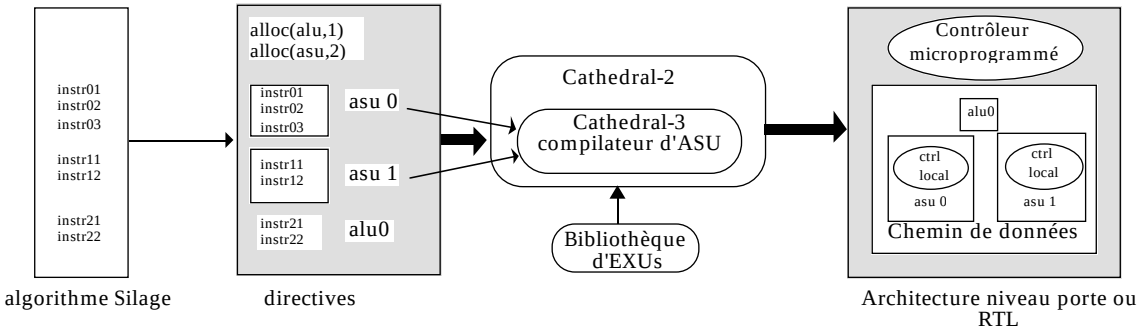


Figure III.4 : Flot de conception utilisant Cathedral-2/3.

2.1. Domaine d'application

Cathedral-2/3 s'adresse aux applications orientées traitement du signal (ou DSP) pour les réaliser sur des ASICs. En particulier *Cathedral-3* vise les applications à haut débit. Ces applications sont caractérisées par des graphes flot de signal montrant de nombreuses répétitions et une certaine récursivité. D'autres comportements, comme ceux des applications dominées flot de contrôle, pourraient être décrits à l'entrée de l'outil. Mais pour obtenir des résultats efficaces, le niveau de raffinement nécessaire rejoindrait au mieux celui d'une description RTL.

2.2. Architecture produite par Cathedral-2/3

L'architecture matérielle produite par *Cathedral-2/3* (figure III.5(a)) est un processeur VLIW (*Very Large Instruction Word*) synchrone dont le contrôleur est micro-programmé, comme c'est généralement le cas des architectures des DSP en ASIC. Une architecture VLIW consiste en un ensemble de ressources arithmétiques, de mémoires, et d'entrées / sorties qui fonctionnent en parallèle. Les opérateurs parallèles sont directement accédés par les champs des instructions.

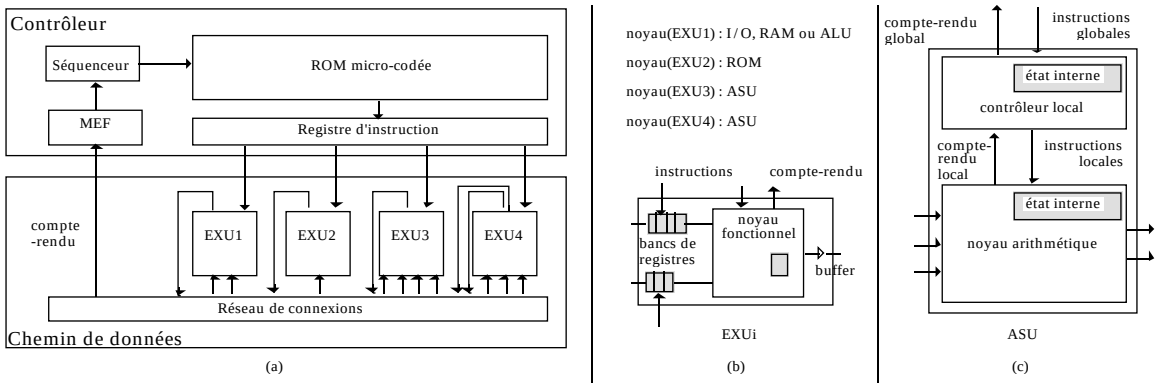


Figure III.5 : Architecture produite par Cathedral-2 / 3

2.2.1. Chemin de données

Le chemin de données (figure III.5(a)) comprend un ensemble d'unités d'exécution (EXUs) d'opération et d'unités d'applications spécifiques interconnectées. C'est aussi le cas dans l'outil CAPSYS [36]. Chaque délai d'exécution est un entier multiple du cycle de l'horloge de base, en général 1. Par défaut l'architecture produite par *Cathedral-2/3* est à base de multiplexeurs. Mais une option permet de choisir une solution optimale mixte à base de multiplexeurs et de bus.

Au cours de la synthèse de l'architecture, le degré de raffinement se situe au niveau des FBB (*Functional Building Blocks*) ou blocs de construction fonctionnels. Un FBB est un élément de logique générique paramétré et programmable, contrôlé indépendamment des autres FBB. Il peut être combinatoire ou séquentiel. Des exemples sont les bancs de registres, les multiplexeurs, les additionneurs, les opérateurs à décalage, les PLAs, les comparateurs et les décodeurs.

L'équivalent d'une unité fonctionnelle dans *Amical* est une unité d'exécution dans *Cathedral-2* (figure III.4). Une EXU (figure III.5(b)) se compose d'un noyau fonctionnel alimenté en entrée par des bancs de registres double-ports. Ce noyau fonctionnel est à base de FBB. La bibliothèque d'EXUs prédéfinies est limitée, mais elle peut être étendue par des EXUs définies par l'utilisateur. Les EXUs sont donc des opérateurs figés dont la fonctionnalité est rigide.

Une autre solution consiste à utiliser un deuxième type d'unités fonctionnelles, les unités d'application spécifique, ASUs, produites à la demande par *Cathedral-3*. Elles peuvent profiter des techniques d'optimisation logique intégrées dans *Cathedral-3* comme la propagation de constantes, l'élimination de logique redondante, etc. Le concepteur choisit de réaliser certaines parties critiques d'un algorithme par une ASU dans le but d'accélérer l'exécution de ces parties en temps réel, ou de réduire leur surface, ou encore d'obtenir les deux en même temps. Les fonctions arithmétiques et le contrôle de l'ASU ne sont plus prédéfinis ni mémorisés dans une bibliothèque. A la place, le compilateur les produit à partir de spécifications de haut niveau fournies par le concepteur sous la forme de directives d'affectation. Un vérificateur temporel calcule la latence d'une ASU en termes de cycles d'horloge.

Une ASU (figure III.5(c)) se compose donc d'un noyau arithmétique et d'un contrôleur logique local. Cette architecture évite la perte systématique d'un cycle, nécessaire au contrôleur global micro-programmé pour contrôler les EXUs. Le noyau des ASUs est réalisé à base de cellules standard. Le chemin de données, constitué de FBB est contrôlé par des micro-instructions. Cependant un multiplexage dans le temps de l'ASU peut sélectionner plusieurs sous-algorithmes. C'est ce qui arrive lorsque l'on affecte plus d'une expression à la même ASU. Dans un tel cas l'ASU produite est programmable et peut être commandée dans différents modes par le contrôleur global pour exécuter les différentes instructions.

2.2.2. Contrôleur

Le contrôleur est constitué d'une machine d'état de Moore, d'un séquenceur, d'une ROM micro-codée et de registres d'instructions. En général la largeur du registre d'instruction est très grande (50 à 200 bits, d'où le nom VLIW). Il existe un registre d'instruction pour chaque FBB du chemin de données. Le contrôleur ne produit qu'un seul fil de contrôle. Toutes les ressources suivent le même flot de programme de sorte qu'un seul séquenceur est maintenu. Le décodage local dans la MEF et le cache peuvent réduire la largeur du mot d'instruction et en conséquence la taille de la ROM micro-codée.

2.3. Flot de synthèse

La figure III.6 illustre le flot de synthèse par *Cathedral-2/3*. Le système se compose de deux outils, *Cathedral-2* et *Cathedral-3*, et d'une bibliothèque d'unités d'exécution figées. Exceptée la description comportementale, le concepteur fournit les spécifications temporelles et d'autres directives pour guider les optimisations de l'outil. Les optimisations peuvent même être logiques car la synthèse logique est intégrée.

2.3.1. Les étapes de synthèse

Le flot commence dans *Cathedral-2* par l'organisation des mémoires. Celle-ci consiste à compiler des structures de données multi-dimensionnelles (scalaires, tableaux, matrices de signaux) sur des mémoires ASICs distribuées. Il faut considérer plusieurs paramètres car le nombre d'unités de mémorisation influence la surface du circuit, tandis que la complexité des calculs d'adresse influence le débit des données.

Le flot continue avec l'allocation des ressources matérielles. Plusieurs EXUs peuvent parfois exécuter la même opération. Des directives permettent de choisir la plus intéressante selon le concepteur.

La synthèse se poursuit dans *Cathedral-3* si une directive demande l'allocation d'une ASU, sinon on passe directement à la génération du code.

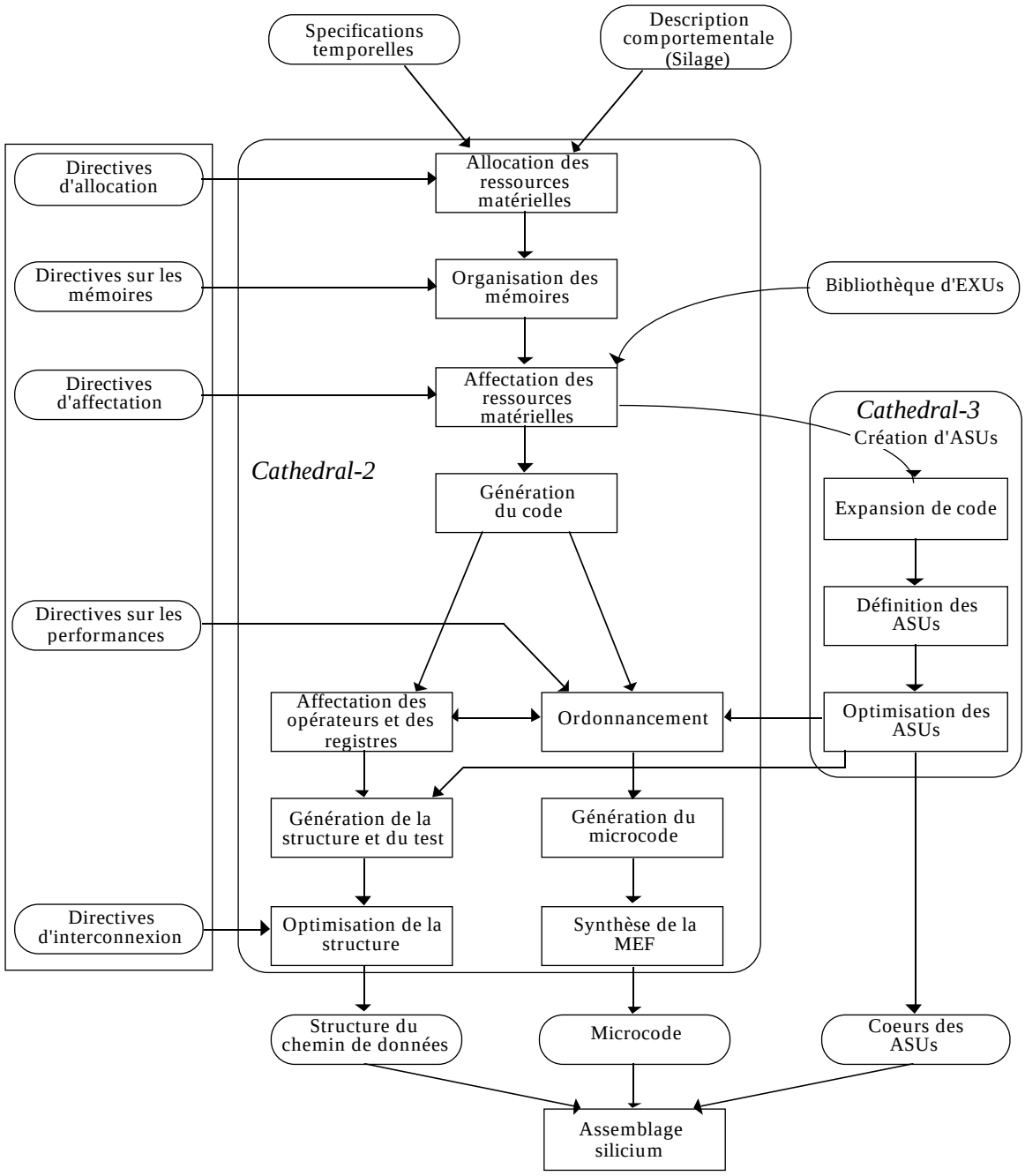


Figure III.6 : Flot de conception dans Cathedral-2/3

Pour la synthèse d'ASUs, *Cathedral-3* commence par la génération d'un graphe flot de données et de signaux par l'expansion du code d'origine. Après l'attribution des différentes parties du graphe à des micro-instructions d'une ASU, il est possible de générer la liste de matériel nécessaire en utilisant la bibliothèque de FBB disponibles. Une partie du graphe peut contenir des prises de décision locales, c'est pourquoi cette étape de définition de l'ASU produit un contrôleur local. Enfin l'outil optimise le résultat obtenu en termes de performance et de surface.

Il est possible de récupérer le résultat de *Cathedral-3* sans repasser par *Cathedral-2* si l'on n'a besoin que du composant ASU.

Pour les ressources matérielles allouées, *Cathedral-2* met à plat les constructions de haut niveau en Silage sous forme de primitives RTL pendant l'étape de génération de code. L'outil analyse le flot de données de l'application et détermine les dépendances de données. Puis il met à jour et optimise les opérations conditionnelles et l'ordre des boucles de l'algorithme. Les transferts de registres doivent s'effectuer le plus souvent possible sous des conditions mutuellement exclusives de façon à partager les ressources matérielles requises pour ces transferts.

L'ordonnancement du résultat obtenu sert à ordonner les transferts de registres sur l'axe des temps de telle façon que l'exécution de l'algorithme prenne le minimum de cycles. A partir des directives temporelles l'outil termine les affectations et les allocations de matériel.

Pendant l'étape de génération de la structure, l'outil alloue les connexions et vérifie la testabilité du circuit. *Cathedral-2* peut alors délivrer le micro-code qui sert d'entrée à l'environnement de génération du contrôleur.

2.3.2. Optimisations

Les optimisations sont nombreuses au cours des différentes étapes de la synthèse. Chaque tâche opère quelques transformations selon ses propres critères d'optimisation, comme la minimisation du nombre de cycles, du nombre de multiplexeurs, ou le partage des ressources. On peut influencer le résultat par quelques directives autorisées par l'outil et mémorisées dans des fichiers. Des estimations en performance et en surface sont disponibles après chaque étape de synthèse lorsque la progression dans la synthèse l'autorise.

Une option permet de réaliser le repliement des boucles qui introduit davantage de parallélisme dans le graphe de flot de données en insérant un pipeline dans l'exécution du corps de la boucle. La taille du contrôleur est alors sensiblement augmentée.

Le concepteur peut si nécessaire demander à *Cathedral-3* d'insérer un ou plusieurs étages de pipeline dans une ASU. Ces étages sont insérés automatiquement par l'outil sur toute la largeur du flot de données aux endroits les plus intéressants, selon ses propres estimations.

Enfin Cathedral-2/3 offre les optimisations logiques classiques telles que l'élimination des redondances, la propagation de constante, le replacement de registres.

2.3.3. Synchronisation

Le concepteur définit la fréquence de fonctionnement de son circuit.

Au niveau comportemental, la description Silage ne contient aucune information relative à l'horloge. C'est pendant la synthèse que le concepteur définit la fréquence souhaitée en accord avec les performances de l'architecture estimées par l'outil. La description de sortie est synchronisée sur cette horloge.

Aucune insertion de signal de mise à zéro n'est prévue dans l'outil, même si l'architecture produite contient des registres. Pour initialiser les registres, on décrit un signal fonctionnellement équivalent au niveau comportemental, en spécifiant les valeurs d'initialisation.

2.3.4. Validation

La description comportementale est écrite en Silage. Elle est traduite en C à l'entrée de *Cathedral-2/3* pour sa vérification fonctionnelle. Pour valider la description comportementale, on a le choix entre exécuter le code C ou co-simuler ce code C avec le code VHDL de la configuration de test qui servira à la validation du résultat VHDL de la synthèse. La synthèse est effectuée à partir du GFD (défini au chapitre II, §2.1) issu du Silage, et non du C qui a servi à la validation. Le résultat est une description VHDL au niveau portes. Le fait que *Cathedral-2/3* utilise des langages différents en entrée et en sortie pose des problèmes supplémentaires pour la validation. Si l'on veut utiliser la même configuration de test pour valider la description comportementale et l'architecture produite au niveau transfert de registre, il faut recourir à la co-simulation. Dans la suite nous supposons que la configuration de test est décrite en VHDL.

Pour la co-simulation du code C, il est possible d'inclure ce code dans une architecture VHDL. Le C communique avec le VHDL à travers une interface CLI (*C Language Interface* propre à Synopsys [52]), synchronisée par l'horloge du système. Même si l'exécution du code C est plus rapide que l'exécution du code VHDL, cette interface peut légèrement ralentir la co-simulation par rapport à une simulation purement VHDL, lorsqu'il y a de nombreux échanges de données à chaque cycle d'horloge. La visualisation des signaux internes (non ports) est impossible, puisqu'on ne peut entrer dans le C lors de la simulation. Le seul moyen de les observer est de les déclarer comme ports de sortie dans l'entité!

La description résultante peut sortir au niveau transfert de registre ou au niveau portes. Dans les deux cas elle consiste en une liste de composants à deux entrées (or, nand, inv, etc.) de largeur un bit interconnectés. Ce niveau de raffinement des composants de la bibliothèque ralentit l'étape de validation au niveau transfert de registre et la synthèse logique si l'on souhaite la réaliser avec un autre outil.

Un même circuit ne possède pas la même entité aux niveaux comportemental, RTL ou portes. Cela est dû à l'insertion de l'horloge et des ports de tests en cours de synthèse, ainsi qu'au positionnement des paramètres génériques au niveau comportemental pour la synchronisation entre le front montant de l'horloge et la transmission ou la récupération des données du C. Cela signifie qu'il faut créer deux configurations de simulation différentes : l'une pour le modèle comportemental, l'autre pour le modèle de sortie, ce qui est laborieux et source d'erreurs lorsque l'on modifie fréquemment l'interface.

Si l'on arrête la synthèse par *Cathedral-2/3* au niveau transfert de registre, le temps de simulation se rapproche plutôt de celui d'une simulation au niveau portes. De plus, les noms des signaux de ces descriptions générées automatiquement ne correspondent pas aux noms des signaux de la description comportementale, (sauf ceux de l'entité), ce qui rend extrêmement difficile la localisation d'un bogue. Par contre, nous avons pu apprécier de toujours valider la fonctionnalité des descriptions résultantes dès le premier flot. Peut-être est-ce dû à une interactivité limitée de *Cathedral-3* par rapport à d'autres outils?

3. *Behavioral Compiler*

Le domaine d'application de *Behavioral Compiler* (*BC*) est la conception matérielle de circuits totalement synchrones (excepté éventuellement pour la remise à zéro), pour produire des ASICs et des FPGAs [11, 53]. Le temps de synthèse avant d'atteindre le RTL varie de quelques minutes à quelques heures suivant la complexité du circuit.

L'entrée de *BC* consiste en une entité VHDL associée à une architecture ou en un module Verilog, et en un ensemble de contraintes fixées par l'utilisateur (figure III.7). Dans la suite on ne considère que l'entrée VHDL. La sortie RTL de *BC* écrite dans un format interne ne peut entrer que dans l'outil de synthèse logique *Design Compiler*. En réalité ces deux outils interagissent et ainsi utilisent la même bibliothèque synthétique *DesignWare*.

La description VHDL en entrée peut contenir un ou plusieurs processus, des instances de composants et des instructions purement flot de données. On distingue deux types de processus [11] : les processus comportementaux (séquentiels) et les processus RTL. Un processus RTL est ignoré par la synthèse comportementale et peut, comme les instances de composants et les instructions purement flot de données, entrer directement dans la synthèse logique. S'il existe plusieurs processus comportementaux, ils sont ordonnancés indépendamment les uns des autres; autrement dit, l'ordonnancement d'un processus n'aura aucune incidence sur celui d'un autre. Il n'est donc pas possible de partager des ressources entre différents processus. Par conséquent c'est au concepteur d'établir entre les processus une synchronisation qui ne sera pas affectée par les ordonnancements successifs. Lorsque tous les processus comportementaux sont prêts pour la synthèse logique, *BC* en assure les interconnexions.

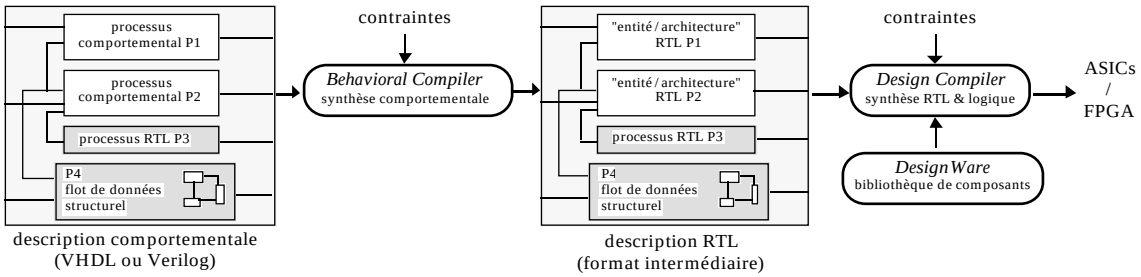


Figure III.7 : Flot de conception utilisant Behavioral Compiler

3.1. Architecture produite par BC

Le modèle architectural produit consiste (figure III.8) en un chemin de données, des mémoires, des entrées / sorties et un contrôleur. Pour chaque processus, *BC* fournit un chemin de données et un contrôleur.

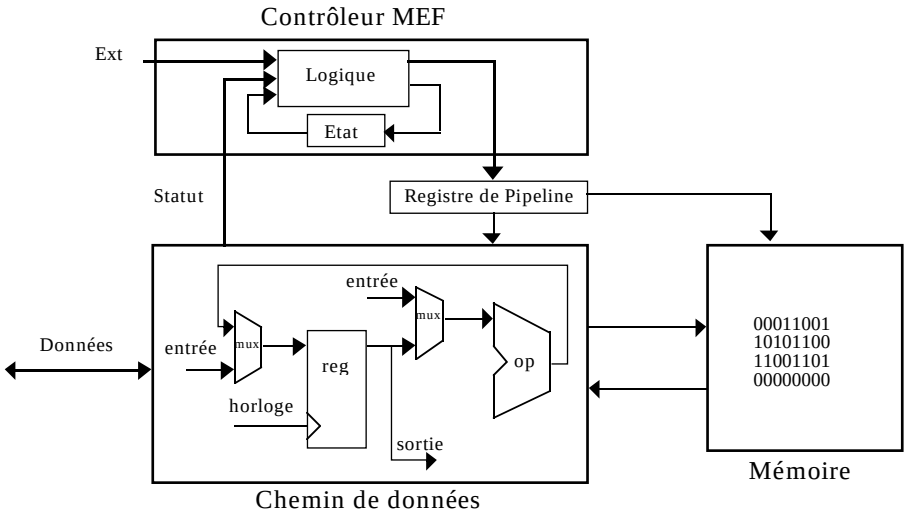


Figure III.8 : Architecture produite par Behavioral Compiler

3.1.1. Chemin de données

Le chemin de données est un circuit “mux-registre-mux-opérateur” ou “mux-registre-mux-registre”, comme pour *Amical*. Cela signifie que son architecture est à base de multiplexeurs, de registres et d'opérateurs de calcul. Les entrées des opérateurs sont issues des multiplexeurs, qui sélectionnent les sorties des registres. Les résultats des opérateurs sont mémorisés dans des registres après sélection par une barrière de multiplexeurs. Ce type d'architecture permet de partager les ressources, registres et opérateurs, pendant le fonctionnement du circuit. Par défaut toutes les sorties sont mémorisées. Cependant il existe un attribut qui, lorsqu'il est associé à un port de sortie, donne éventuellement l'ordre de ne pas mémoriser les données de ce port. Lorsqu’il y a peu de flot de données, comme dans les circuits orientés mémoires, le chemin de données devient un circuit “mux-registre” utilisé principalement pour les mémoires, les entrées / sorties et le contrôle.

3.1.2. Contrôleur

Le contrôleur est une machine d’états finis de Mealy : les sorties dépendent des entrées et de l’état précédent. Des registres optionnels peuvent être présents à l’entrée ou à la sortie du contrôleur pour permettre d’insérer un étage de pipeline lorsque le chemin critique traverse le chemin de données et le contrôleur. Cela réduit le cycle d’horloge, éventuellement au prix d’un ordonnancement plus long.

3.1.3. Mémoires

BC permet au concepteur de spécifier des mémoires en utilisant des variables tableaux et une directive qui affecte la variable à une et une seule mémoire particulière de la bibliothèque synthétique. Ces mémoires physiques peuvent être synchrones, asynchrones, simple ou multi-ports. Mais la version actuelle de l'outil ne permet pas le partage des mémoires entre plusieurs processus. L'utilisation d'un ou plusieurs tableaux dans la description comportementale entraîne l'inclusion d'une ou de plusieurs mémoires. A partir des informations de la bibliothèque, *BC* fournit automatiquement des interfaces avec les RAMs et les ROMs. L'outil ordonne et alloue ces interfaces en insérant un étage de pipeline si nécessaire. Cependant on constate que les cycles de lecture / écriture sont plus lents après synthèse parce que les transferts sont synchrones. Les conflits d'accès aux RAMs ne sont pas gérés. Le concepteur reste responsable des protocoles de communication.

BC admet également les structures dynamiques d'enregistrement et incluent alors des mémoires : mais l'accès à un champ signifie l'accès à tout l'enregistrement. On ne peut donc accéder à deux champs d'un même enregistrement dans le même cycle d'horloge.

3.2. Flot de synthèse

La figure III.9 illustre le flot de synthèse de *BC*. Après avoir analysé la syntaxe et la fonction de la description comportementale, *BC* élabore le circuit pour préparer l'ordonnancement et construit une représentation abstraite du circuit. *Design Compiler* caractérise alors le circuit temporellement au niveau portes. Puis l'allocation des ressources matérielles fait appel à la bibliothèque de composants réutilisables du *DesignWare*. Des comptes-rendus détaillés sur l'ordonnancement et l'allocation sont alors disponibles. Ils servent de base à une exploration des architectures possibles en modifiant

les contraintes avant l'optimisation au niveau portes. A ce moment-là, une description RTL peut être fournie pour une vérification rapide. Enfin *BC* connecte tous les éléments précédemment ordonnancés et, avec *DC* il construit et optimise la description au niveau portes.

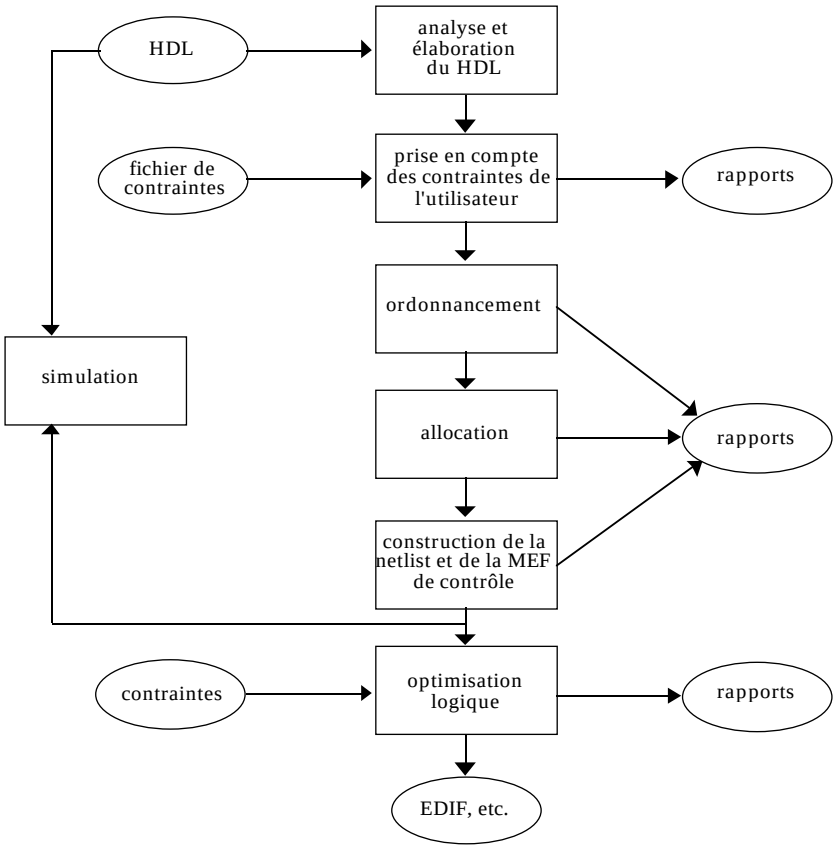


Figure III.9 : Flot de synthèse de BC

3.2.1. Les étapes de synthèse

Le processus de synthèse peut être réalisé de manière itérative. Les itérations consistent principalement à explorer les compromis entre la période de l'horloge et l'allocation des ressources, et à placer les contraintes d'ordonnancement sur les opérations et les boucles. En contraignant l'ordonnancement et en modifiant la période d'horloge, le concepteur système est capable d'explorer l'espace des solutions. Il lui est possible :

- d'imposer un nombre minimum, maximum ou exact de cycles pour un processus ou une boucle;
- de forcer une opération dans un nombre donné de cycles.

L'exploration architecturale utilise des résultats intermédiaires et des estimations de coût. *BC* délivre les comptes-rendus suivants :

- le bilan de l'ordonnancement : il donne le nombre de cycles requis et une estimation de la surface.
- le compte rendu des opérations : il montre comment les opérations sont ordonnancées à travers les étapes de temps disponibles, cependant il est difficile à lire.
- le compte rendu des variables : il montre comment les valeurs intermédiaires d'un calcul sont mémorisées dans des registres, cependant il est difficile à lire.

L'étape d'ordonnancement dans *BC* distingue trois modes d'entrées / sorties [54] et autorise plusieurs types d'optimisations pour augmenter le parallélisme des tâches. Ces trois modes d'entrées / sorties sont:

- le mode "cycle" (pour *cycle based*): Tout le code VHDL situé entre deux instructions *wait* consécutives (définies au chapitre II, §2.1) est exécuté en un cycle.
- le mode "macro-cycle" : Toutes les opérations VHDL situées entre deux instructions *wait* consécutives sont exécutées en un nombre fixe de cycles. Les entrées / sorties peuvent se situer dans chacun de ces cycles.
- le mode "entrées / sorties flottantes" : Toutes les opérations, y compris les opérations d'entrées / sorties peuvent flotter en dehors des instructions *wait*.

Dans le cas idéal, le concepteur souhaite utiliser les mêmes stimuli pour vérifier les descriptions aux niveaux comportemental et RTL. Pour un ordonnancement en mode “cycle”, il n’y a aucun problème. Pour un ordonnancement en mode “macro-cycles” cela reste faisable, car des cycles d'horloge supplémentaires ont pu être insérés. Cependant pour le mode “entrées / sorties flottantes” cela devient plus difficilement contrôlable, car l'ordonnancement a pu intervertir l'ordre de certaines opérations .

Or, l’ordonnancement en mode “entrées / sorties flottantes” donne aux concepteurs la plus grande flexibilité d’exploration. Par conséquent, les modes “entrées / sorties flottantes" et “macro-cycle” servent à explorer l’espace des solutions et un ordonnancement en mode “cycle” sert à générer la description RTL finale. En pratique le mode le plus utilisé est le mode “macro-cycle”.

L'ordonnancement de *BC* utilise des ”gabarits comportementaux” [55] qui permettent de fixer l'ordre relatif d'un ensemble d'opérations. Cette construction simple permet le respect des contraintes temporelles, la modélisation des opérations séquentielles, le chaînage de certaines opérations et l'ordonnancement hiérarchique.

L'étape d'allocation dans BC fait appel à des unités fonctionnelles génériques de la bibliothèque *DesignWare* et directement aux cellules de la bibliothèque de portes afin de fournir des estimations en termes de performance et de surface dès la fin de la synthèse architecturale.

3.2.2. Optimisations

L'ordonnancement réalise plusieurs optimisations. La première caractéristique de l'ordonnancement de *BC* est le chaînage. *BC* accède à des modèles temporels au niveau du bit pour ordonnancer dans un cycle unique des opérations qui dépendent des données. L'outil admet aussi des opérations multi-cycles au moyen d'une variable. On peut aussi appliquer le repliement et le déroulement de boucle. Toutes ces notions ont déjà été définies dans le tableau II.1.

L'analyse de la durée de vie des variables permet le partage de registres entre plusieurs variables.

BC peut utiliser des opérateurs avec un pipeline de la bibliothèque du *DesignWare* pour améliorer les performances. Ces opérateurs contiennent des registres pour mémoriser les résultats intermédiaires. Ainsi il est possible de décomposer un chemin critique d'un circuit, en cours de synthèse, en deux étapes ou plus.

3.2.3. Synchronisation

Dans la description comportementale l'attente d’un signal qui n’est pas l’horloge primaire n’est pas permise. Les mises à zéro synchrones et asynchrones sont acceptées avec les mêmes restrictions que pour *Amical* (chapitre II, §1.3.3)

3.2.4. Validation

La nature de la description produite pose un sérieux problème de correction des bogues lors de la validation au niveau transfert de registre. La description comportementale est du même style que la description comportementale à l'entrée d'*Amical*. Le sous-ensemble VHDL comportemental est par contre beaucoup plus étendu. Les pointeurs et les opérations sur les bits sont admis. Mais les limitations d'écriture, les problèmes de gestion des mémoires partagées allongent la durée de la validation de la description comportementale par rapport aux autres outils présentés.

La description RTL est sous une forme binaire, dans le format interne des outils de Synopsys. Il contient des informations sur les contraintes à appliquer lors de la synthèse logique par le *Design Compiler*. Pour les mêmes raisons que pour la description sortie d'*Amical*, la validation de la description après synthèse par *BC* est indispensable pour s'assurer des bonnes directives durant l'étape d'ordonnancement. Il est donc possible d'obtenir une description VHDL mais son code n'est pas synthétisable. Par ailleurs, ce code est illisible et complique la validation de la synthèse. Certaines options permettent d'extraire la machine d'états finis produite, mais il reste très laborieux d'identifier l'origine des erreurs et de savoir ce qu'il faudrait modifier pour les corriger.

La simulation au niveau portes peut dans certains cas réutiliser les stimuli du niveau comportemental pour comparaison.

4. Combinaison de plusieurs outils pour la conception de circuits complexes.

Le tableau III.1 résume les caractéristiques principales de chaque outil présenté dans ce chapitre, à savoir : le domaine d'application, la forme intermédiaire ou modèle sous-jacent, la flexibilité du processus de synthèse, le langage de la description d'entrée, le type d'architecture produite et le schéma de synthèse.

Ainsi, *Amical* est un outil de synthèse comportementale, orienté flot de contrôle. Le modèle de représentation interne du circuit est une machine d'états finis étendue avec co-processeur [5]. L'étape d'allocation des ressources fait appel à une bibliothèque d'unités fonctionnelles extensible. Le flot de synthèse est interactif. *Amical* traduit une description comportementale VHDL en une description RTL VHDL structurelle, comportant une partie contrôle, un chemin de données et éventuellement un ou plusieurs co-processeurs. Enfin, le schéma de synthèse consiste en un macro-ordonnancement, suivi d'une allocation puis d'un micro-ordonnancement suivi de l'allocation des connexions.

Cathedral-2/3 est orienté flot de données. Le modèle de représentation interne du circuit est une machine d'états finis étendue avec des opérateurs sur des données [25]. L'étape d'allocation des ressources fait appel à une bibliothèque d'unités fonctionnelles interne, extensible par les ASUs issues de *Cathedral-3*. Le pipeline de ces ASUs est automatique. L'interactivité de l'outil est limitée. La spécification d'entrée est décrite en Silage et la sortie en VHDL. Une étape d'ordonnancement suit l'étape d'allocation.

BC a de nombreux points communs avec *Amical* au niveau du style de la description comportementale et avec *Cathedral-2/3* au niveau des optimisations. Seule l'entrée est décrite en VHDL ou en Verilog, et les mémoires se situent à l'extérieur du chemin de données.

Caractéristiques	*Amical*	*Cathedral-2/3*	*Behavioral Compiler*
Domaine d'application	Flot de contrôle	Flot de données	Mixte flot de données / contrôle
Modèle sous-jacent	MEFC	MEFD	MEFD
Flexibilité du processus de synthèse	bibliothèque externe et synthèse interactive	bibliothèque interne extensible par des ASUs, pipeline, optimisation logique	bibliothèque externe pipeline, optimisation logique
Spécification d'entrée	VHDL	Silage	VHDL / Verilog
Sortie	Partie Contrôle Partie Opérative Co-Processeur	Partie Contrôle Partie Opérative	Partie Contrôle Partie Opérative Mémoire
Schéma de synthèse	Ordonnancement / Allocation sur deux niveaux	Ordonnancement / Allocation	Ordonnancement / Allocation

Tableau III.1 : Comparaison et récapitulatif des trois outils.

4.1. Intérêt de l'utilisation de plusieurs outils

Les outils de synthèse comportementale ont été orientés pour une classe limitée d'applications pour être efficaces. La spécialisation des domaines d'application de la plupart des outils explique donc la nécessité d'utiliser des outils complémentaires dans le cas de circuits hétérogènes. Le tableau III.1 fait ressortir la spécificité des domaines d'application et la complémentarité des outils sur certains aspects.

Pour les circuits complexes il arrive souvent que l'on trouve des parties orientées flot de contrôle et des parties orientées flot de données. Pour la conception de tels systèmes les problèmes se situent plutôt au niveau du partitionnement et de l'intégration des différentes parties au sein du système.

Pour des raisons d'efficacité on peut être amené à utiliser les outils les plus performants dans leur domaines d'application de façon complémentaire, si c'est nécessaire. Ce type d'utilisation mixte d'outils a des conséquences méthodologiques particulières sur la vérification, avant et après synthèse, et sur l'intégration des résultats.

Cette méthode d'utilisation d'outils complémentaires revêt plusieurs intérêts :

- Tout d'abord l'étude de la faisabilité.
- La connaissance des problèmes rencontrés lors de l'utilisation mixte d'outils de synthèse comportementale.
- Enfin le fait de choisir deux outils spécifiques à deux classes d'application suffit-il à couvrir la classe d'application mixte?

4.2. Conséquences méthodologiques

L'utilisation mixte d'outils pose deux problèmes : l'intégration des architectures résultantes des différentes synthèses dans une même architecture globale et la vérification de celle-ci.

L'utilisation de plusieurs outils de synthèse implique éventuellement des langages de description différents, comme le VHDL (normalisé), le C (qui possède une interface avec le VHDL), Silage, etc. Par conséquent seuls les langages de description traduisibles en C ou en VHDL sont considérés ici pour les simulations multi-langages.

La vérification est une étape essentielle et particulièrement délicate dans le cas d'une méthode mixte. Pour la conception d'un même circuit, l'utilisation de deux outils de synthèse de haut niveau différents nécessite des simulations mixtes. Ces simulations impliquent des niveaux d'abstraction différents et des langages différents. L'environnement de simulation doit pouvoir intégrer des blocs de niveaux de description différents.

Un autre problème est l'utilisation de concepts temporels différents. Au niveau comportemental, la synchronisation est assurée par la causalité; au niveau transfert de registres, elle est orchestrée par des horloges.

Lorsque les langages de description et les concepts temporels, éventuellement différents d'un outil à l'autre au niveau comportemental, permettent d'envisager la simulation de l'ensemble, il faut intégrer les différentes architectures dans une architecture unique. Pour cela il faut définir des modèles de communication entre ces blocs et abstraire leur comportement réciproque, l'un vis-à-vis de l'autre. Ce mode de communication est par conséquent soumis à la synthèse.

Le reste de cette section est dédié à la présentation d'une méthode de conception et des étapes de vérification nécessaires pour la synthèse de haut niveau d'un circuit avec une partie contrôle complexe, par le compilateur *Amical (*orienté flot de contrôle), et un chemin de données complexe, par l'outil *Cathedral-2/3* (orienté flot de données).

4.3. Méthode de conception utilisant à la fois *Cathedral-2/3* et *Amical*

La méthode de conception proposée procède de façon hiérarchique. Une première étape permet la décomposition d'une spécification système en un contrôleur global et des sous-systèmes éventuellement orientés flot de données réalisant des tâches spécifiques. Cette décomposition permet de traiter des circuits complexes en les décomposant en sous-systèmes et en traitant chaque partie par un outil de synthèse de haut niveau dédié. Bien sûr, l'intégration des résultats de différents compilateurs est l'étape finale de cette méthode. Les sections suivantes décrivent les différentes étapes de la méthode. La figure III.10 donne une vue globale de ce flot de conception.

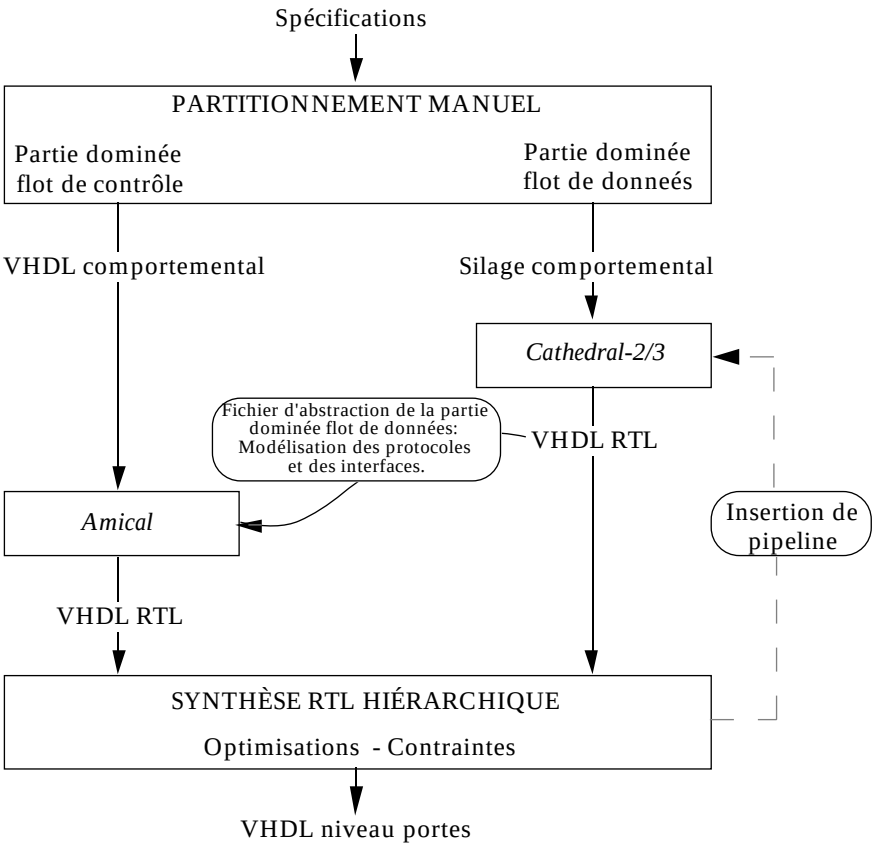


Figure III.10 : Flot de conception utilisant Amical *et* Cathedral-2/3.

Le processus de conception part d'une spécification du système. Le circuit est ensuite découpé en une partie dominée flot de contrôle et une ou plusieurs parties dominées flot de données. La partie dominée flot de contrôle est soumise à une synthèse architecturale par *Amical*. La partie dominée flot de données est soumise à une synthèse architecturale par *Cathedral-2/3*. La partie issue de *Cathedral-2/3* est utilisée par *Amical* sous forme d'unité fonctionnelle. Pour ce faire, le concepteur doit abstraire le comportement de l'unité fonctionnelle pour être vue comme une boîte noire par le contrôleur global. La description résultante peut être soumise à un outil de synthèse logique acceptant du VHDL en entrée. Les sections qui suivent détaillent l'intégration de la partie issue de la synthèse par *Cathedral-2/3* dans le reste du circuit et la validation de l'ensemble.

4.4. Partitionnement d'un circuit en une partie dominée flot de données et une partie dominée flot de contrôle

La spécification initiale donne la fonction du circuit. Une spécification de haut niveau d'un système complexe nécessite seulement de décrire le séquencement des tâches à exécuter par les sous-systèmes. A partir des spécifications, le concepteur doit déterminer quel outil synthétisera quelle tâche.

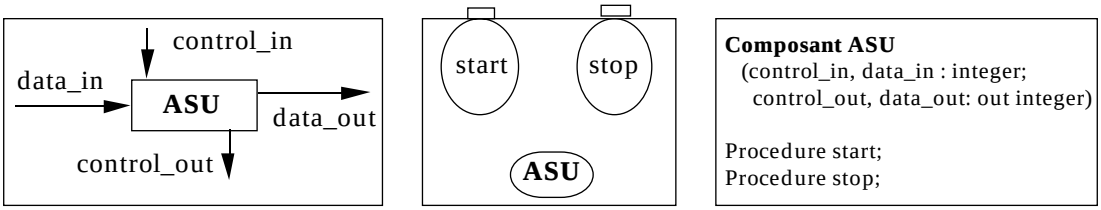
Dans notre cas, cette étape consiste à réaliser le partitionnement du circuit en une partie dominée flot de contrôle et une partie dominée flot de données en prenant en compte la façon de les décrire (Silage et VHDL comportemental), leur façon de communiquer avant et après synthèse et en particulier leur capacité à abstraire la partie dominée flot de données pour sa réutilisation comme unité fonctionnelle. Cette étape sera réalisée manuellement.

4.5. Synthèse par *Cathedral-2/3*

Dès que les tâches à synthétiser par *Cathedral-2/3* sont définies, par le concepteur à partir des spécifications initiales, elles sont décrites en Silage. Après la génération de code et la définition de la partie dominée flot de données dans *Cathedral-3*, on peut connaître sa performance. Si les contraintes temporelles ne sont pas satisfaisantes, on peut insérer un ou plusieurs étages de pipeline pour augmenter le débit.

4.6. Abstraction et réutilisation de la partie dominée flot de données

La partie dominée flot de données, issue de *Cathedral-2/3*, sera activée comme une unité fonctionnelle par la partie dominée flot de contrôle assimilée à un contrôleur global. Elle doit être abstraite pour être réutilisée lors de la conception du circuit complet [56]. Cette étape définit le protocole de communication entre le contrôleur global et le composant. Pour *Amical*, chaque unité fonctionnelle peut être spécifiée à trois niveaux d'abstraction différents. A partir d'une réalisation de la partie dominée flot de données (figure III.11(a)). le concepteur produit une vue conceptuelle (figure III.11(b)). L'objet dominé flot de données est capable d'exécuter un ensemble de fonctions (*start*, *stop*, etc.). On utilise cette abstraction pour produire un modèle comportemental des composants (figure III.11(c)). Ces opérations cachent les détails de réalisation pendant la réutilisation de ce composant.



(a) Vue d'implantation (b) Vue conceptuelle (c) Vue comportementale

Figure III.11 : Les trois niveaux d'abstraction.

Cette solution autorise le parallélisme avec l'activité d'autres unités fonctionnelles. Le protocole consiste à sélectionner le mode de traitement de la partie dominée flot de données et à attendre le signal de contrôle de sortie signifiant la fin du calcul et la disponibilité des résultats. Une inclusion du composant dans l'architecture VHDL RTL peut faciliter le protocole.

4.7. Synthèse par *Amical*

Elle commence par deux informations requises pour la synthèse : une spécification comportementale et une bibliothèque d'unités fonctionnelles. L'utilisation de sous-systèmes complexes est faite à travers des appels de procédure ou de fonction. Pour chaque procédure ou fonction utilisée, la bibliothèque doit inclure au moins une unité fonctionnelle capable d'exécuter l'opération correspondante.

Pour permettre l'allocation de la partie dominée flot de données, *Amical* a besoin d'un fichier d'abstraction dédié. Ce fichier inclut l'interface du bloc en spécifiant ses signaux de contrôle, ses paramètres d'appel correspondant aux paramètres de l'opération, l'ensemble des opérations ainsi que le protocole de passage des paramètres pour chaque opération.

4.8. Vérification et simulation multi-niveaux

La validation des spécifications et les différentes étapes de synthèse est un stade critique lorsque différents outils avec différents langages de spécification sont impliqués.

Pour vérifier que le comportement global du circuit reste le même pour tous les niveaux d'abstraction, une configuration unique de test RTL est utilisée pendant tout le flot de conception pour valider: les spécifications comportementales, les spécifications RTL produites par la synthèse de haut niveau et finalement la liste de portes produite par la synthèse logique.

La figure III.11 résume les différents cas de co-simulation et leurs rôles. Aucune étape n'est inutile. Chacune permet de mettre en évidence certains problèmes et de détecter les erreurs des scripts de synthèse. La simulation 1 ajuste les descriptions comportementales. Les co-simulations 2 et 3 valident respectivement la synthèse par *Cathedral-2/3* et par *Amical*. La simulation 4 valide la synthèse du protocole d'activation de la partie dominée flot de données. Les simulations 5 et 6 valident respectivement chaque synthèse logique. Enfin la simulation 7 vérifie les délais et valide le flot de conception complet.

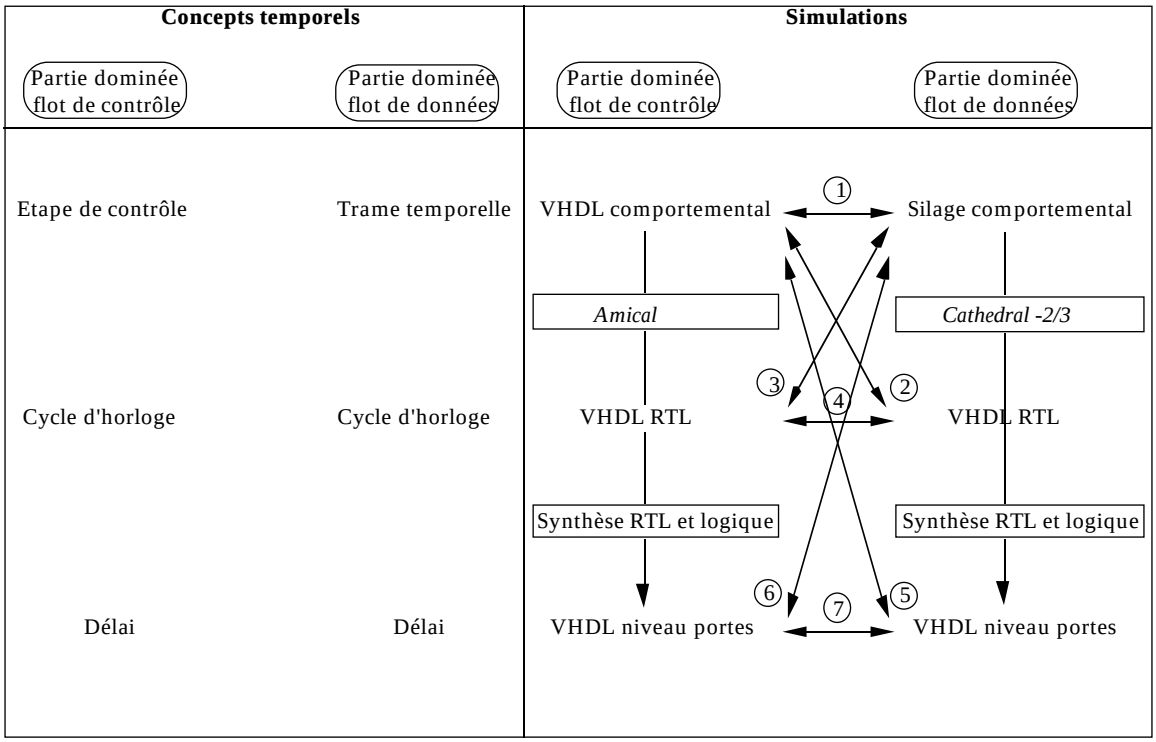


Figure III.12 : Simulations multi-niveaux.

5. Conclusion

Ce chapitre a présenté trois outils de synthèse architecturale de types différents. Le premier, *Amical*, est dédié aux applications orientées flot de contrôle. Le second, *Cathedral-2/3*, est dédié aux applications de traitement de signal et plus particulièrement aux applications à haut débit, grâce à des insertions de pipeline puissantes. Le troisième, *Behavioral Compiler*, est orienté vers les deux domaines d'application. L'étude des ces différents types d'outils a permis de constater qu'il serait intéressant d'utiliser les outils spécifiques à chaque domaine d'application de façon complémentaire, dans la mesure où l'on peut les intégrer dans un même flot de conception.

Ce chapitre a aussi introduit un flot permettant de combiner les outils *Amical* et *Cathedral-2/3*. Ce flot de conception et de validation est sophistiqué par rapport à un flot à base d'un environnement unique comme le propose *BC*. Vu que *BC* n'était pas disponible, au début de ce projet, il faut répondre à la question : ce flot a-t-il un intérêt par rapport à la méthode de conception classique RTL? Les résultats du chapitre V répondent favorablement à cette question. Par ailleurs, il s'agit de se donner le plus de chance pour couvrir l'ensemble des caractéristiques des ASICs, pour ainsi constater les avantages et les limitations de la synthèse architecturale.

Chapitre IV : Étude de Cas - Conception Modulaire utilisant la Synthèse Comportementale.

L'utilisation modulaire de la synthèse comportementale pour la conception d'un système complexe à partir de sa description VHDL est développée dans ce chapitre. Nous y expliquons la méthode de conception hiérarchique qui à servi à l'évaluation des outils *Amical* et *Cathedral-2/3*. La fonctionnalité du circuit étudié ici est semblable à celle du circuit du chapitre V. Mais elle est beaucoup plus simple, ce qui allège la présentation de la méthode.

L'exemple choisi est un système fondé sur un algorithme de recherche sur des fenêtres. Ce dernier est une application répandue pour la recherche d'une séquence spécifique de mots dans une chaîne de mots. Un tel système est ici conçu à partir d'une architecture avec un contrôleur global gérant trois sous-systèmes parallèles : deux modules mémoire et un co-processeur réalisant la recherche. Chaque sous-système inclut un contrôleur local et sera conçu par la synthèse architecturale. Ce chapitre présente la conception des sous-systèmes et du contrôleur global décrits en VHDL en utilisant la

synthèse comportementale. L'outil de synthèse comportementale *Amical* est utilisé ici pour illustration, mais le style de description comportementale peut être adapté pour d'autres outils de synthèse comportementale avec des entrées VHDL.

1. Introduction

Le système est une application fondée sur l'algorithme de recherche sur des fenêtres, WSS (*Window Searching System*). Son principe est analogue à celui de nombreux algorithmes cherchant une séquence spécifique de mots (*sequence*) dans une chaîne de mots (*string*). Les trois exemples qui suivent utilisent ce type d'algorithme. Le premier est la recherche de gênes (*sequence*) dans une chaîne (*string*) d'ADN [57]. Le second est la traduction d'une phrase ou d'un ensemble de mots (*sequence*) d'après un dictionnaire de phrases (*string*). Le troisième est l'estimation de mouvement pour le choix de codage d'une suite d'images. Il s'agit de trouver la position d'une portion de l'image courante (*sequence*) dans une portion de l'image précédente (*string*) plus grande que la fenêtre courante [58].

Le modèle, étudié ici, exécute l'algorithme du WSS. Son architecture est constituée d'un contrôleur global et d'un chemin de données incluant deux modules mémoires (respectivement pour *sequence* et *string*) et un co-processeur cherchant la séquence la plus proche dans la chaîne. La figure IV.1 montre une vue globale d'une telle architecture.

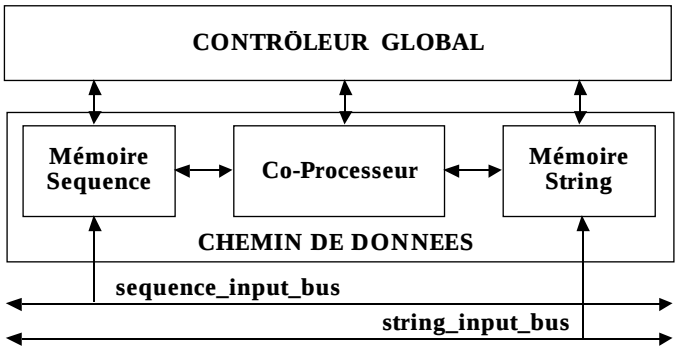


Figure IV.1 : Une architecture globale du système fondé sur l'algorithme de recherche d'une fenêtre.

Cette architecture a pour objectif de permettre le fonctionnement concurrent de ces trois unités. Les modules mémoires peuvent être remplis pendant que le co-processeur exécute l'algorithme de recherche. Le contrôleur global a la charge de coordonner ces trois composants. L'utilisation de mémoires double-ports permet des opérations de lecture et d'écriture simultanées, et de ce fait, de les remplir en parallèle avec le processus de recherche. Quatre problèmes doivent être résolus pendant la synthèse comportementale d'un tel circuit :

- Une fois analysé, le système peut être décomposé facilement en deux sous-systèmes principaux séparés : un contrôleur global et des co-processeurs. Comment procéder à la vérification et la synthèse comportementale modulaire d'un tel système?
- Comme il est décrit plus haut, le chemin de données utilise des unités fonctionnelles (deux modules mémoires et un co-processeur) travaillant de façon concurrente. Comment spécifier un tel parallélisme pour la synthèse comportementale?
- Pour valider la description comportementale, des modèles précis des unités fonctionnelles doivent être utilisés. Ces modèles incluent les valeurs temporelles caractéristiques telles que les temps de "set-up", de "hold", etc. Comment des composants existants tels que les RAMs ou les co-processeurs peuvent-ils être réutilisés au niveau comportemental?
- Pour atteindre les performances souhaitées, des interconnexions directes entre les unités fonctionnelles sont nécessaires (au lieu d'être gérées par le contrôleur global). Comment ces interconnexions locales sont-elles traduites après la synthèse comportementale?

2. Spécifications du système

Cette section décrit la fonctionnalité, l'algorithme de tout le système WSS, les composants de mémorisation, ainsi que les contraintes en performance.

2.1. Fonctionnalité

Le système, fondé sur l'algorithme de recherche d'une fenêtre, doit sélectionner, dans une chaîne (*string*) de n caractères, la séquence de caractères la plus proche d'une séquence spécifique (*sequence*) de caractères. On suppose que la chaîne se compose de 23 caractères W_i, $(0 \leq i \leq 22)$. On présume aussi que la séquence de référence est constituée de 8 caractères Y_j,, $(0 \leq j \leq 7)$. La chaîne se décompose en un ensemble de rafales (en Anglais *bursts*). Dans de nombreuses réalisations, la rafale correspond au nombre de données qui peuvent être chargées à partir de la mémoire externe. Dans le cas présent, une rafale regroupe 8 caractères. La chaîne est alors constituée de trois rafales (n=3). On appellera *string*, la fenêtre de recherche, et *sequence*, la fenêtre courante. Les deux fenêtres sont chargées par des rafales en utilisant deux bus externes (figure IV.1). La figure IV.2 montre une représentation visuelle du processus de recherche. Le résultat consiste en un vecteur de mouvement variant entre 0 et 15 et en une distorsion correspondante, entre la séquence de référence et la séquence la plus semblable. Le $24^{ème}$ caractère de la chaîne est ignoré, du fait de la limitation de la taille du vecteur de mouvement. Le vecteur de mouvement X correspond à la plus petite distorsion trouvée :

$$D_X = \mathrm{Min}_{0 \leq x \leq 15}(D_X)$$

Et une distorsion au vecteur x est définie par :

$$Dx = \Sigma_{0 \leq i \leq 7} | Y_i - W_{i+x} |$$

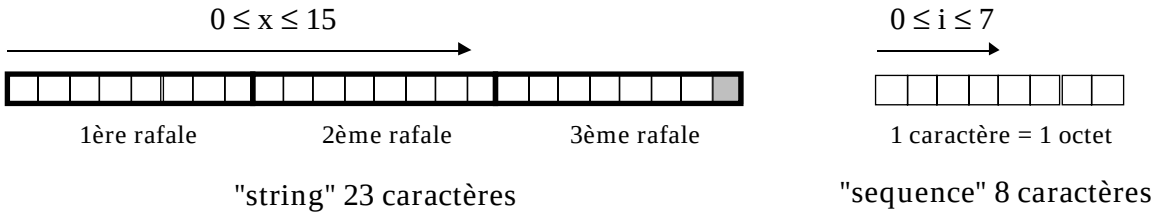


Figure IV.2 : Représentation visuelle du processus de recherche.

2.2. Algorithme

La figure IV.3(a) montre une vue globale de l'algorithme de recherche : après une étape d'initialisation, la séquence est chargée. La chaîne est chargée par n rafales de 8 caractères. Cela prend n-1 itérations. Tant que la fin de la chaîne n'est pas atteinte, une nouvelle rafale est chargée. Dès que deux rafales consécutives de la chaîne sont chargées, le processus de recherche peut commencer. Chaque tâche dans la figure IV.3(a) possède un label portant la durée t_i. Le temps nécessaire à l'accomplissement d'une itération de tout l'algorithme est :

$$T_a = t_1 + t_2 + t_3 + (n-1).(t_3 + t_4)$$

où n est le nombre de rafales requis pour remplir une chaîne.

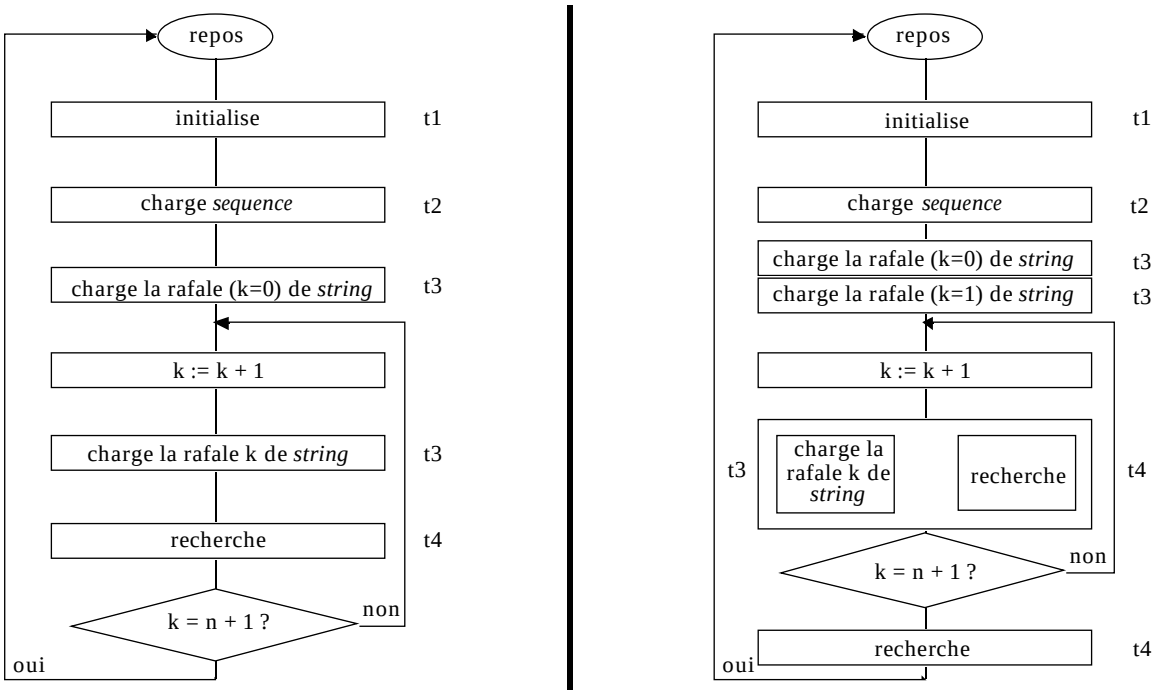


Figure IV.3 : Algorithmes de recherche.

2.3. Mémoires

Cet algorithme de recherche se fonde sur des mémoires accessibles en lecture et en écriture pour *sequence* et *string*. Comme elles sont lues simultanément par le co-processeur, elles seront stockées dans deux RAMs différentes. La chaîne et la séquence sont chargées à partir d'une mémoire externe en utilisant un protocole de requête / acquittement. Selon la figure IV.2, elles sont remplies rafale par rafale. Pour permettre la concurrence entre le contrôleur global et le remplissage des mémoires, des contrôleurs locaux sont nécessaires. Ainsi on utilise deux modules consistant chacun en une mémoire et en un contrôleur local : *mem_sequence* renferme une RAM synchrone 8x8 contenant la séquence de référence, et *mem_string* renferme une RAM synchrone 24x8 contenant la chaîne.

2.4. Parallélisme

Deux types de parallélisme sont possibles : entre les chargements des deux mémoires avant la boucle interne, et entre l'opération de recherche et le chargement de la prochaine rafale à traiter.

Le parallélisme entre le remplissage des deux mémoires dépend de la façon dont les rafales arrivent. Nous supposons que chaque rafale est disponible après un protocole de requête/acquittement sur un bus externe. Comme le montre la figure IV.1 deux bus externes peuvent servir : l'un pour les rafales de *sequence* et l'autre pour les rafales de *string*. Dans ce cas, les deux mémoires peuvent être écrites en parallèle. Pour réduire le nombre de connexions externes, nous utiliserons un seul bus (*Data_In*). Ainsi, les mémoires ne peuvent pas être remplies en parallèle.

Le parallélisme entre l'opération de recherche et le chargement de la prochaine rafale à traiter, suppose que *mem_string* contient une RAM permettant de lire et d'écrire simultanément. Cela conduit au choix de RAMs double-ports.

Le séquencement et le parallélisme des différentes tâches figurent dans une nouvelle version de l'algorithme dans la figure IV.3(b). Le temps T_b pour l'exécution globale est plus court pour cette solution.

$$T_b = t_1 + t_2 + 2t_3 + (n-2) \cdot \max(t_3, t_4)$$

3. Partitionnement du système

Cette section décrit le partitionnement du système et l'architecture abstraite du circuit selon les spécifications systèmes. L'architecture abstraite précède la dernière étape de synthèse d'*Amical* qui est la génération de l'architecture RTL vue au chapitre III, §1.3.1.

3.1. Partitionnement

Le partitionnement du WSS résulte naturellement des spécifications du système global. En fait, en plus d'un contrôleur global, trois sous-systèmes sont nécessaires pour ce système : deux mémoires incluses dans *mem_sequence* et *mem_string*, et *co-processor* qui réalise l'algorithme de recherche. Ces composants peuvent être issus d'outils de synthèse comportementale ou d'un autre environnement de conception. En raison de leur complexité, les contrôleurs de mémoire et le co-processeur peuvent être générés par la synthèse comportementale. Ainsi, quatre descriptions comportementales sont requises, correspondant à *mem_sequence*, à *mem_string*, au co-processeur et à l'ensemble du système. La synthèse de haut niveau de l'architecture globale et les modèles d'abstraction des composants sont décrits dans les sections suivantes.

3.2. Architecture abstraite

A partir du partitionnement, on peut déduire une architecture abstraite du WSS : un contrôleur global en charge de coordonner trois sous-systèmes. Chaque tâche de la figure IV.3(b) peut être exécutée par l'un de ces trois sous-systèmes : "charge sequence" par *mem_sequence*, "charge string" par *mem_string* et les deux modes de "recherche" par *co-processor*. Le co-processeur doit mémoriser les résultats partiels jusqu'à la recherche finale sur la dernière rafale (k = n). La figure IV.4 montre l'architecture abstraite résultante du WSS. Les sous-systèmes sont interconnectées. Les mémoires *mem_sequence* et *mem_string* reçoivent des entrées externes *Data_In*. Le co-processeur va quérir ces données dans les mémoires.

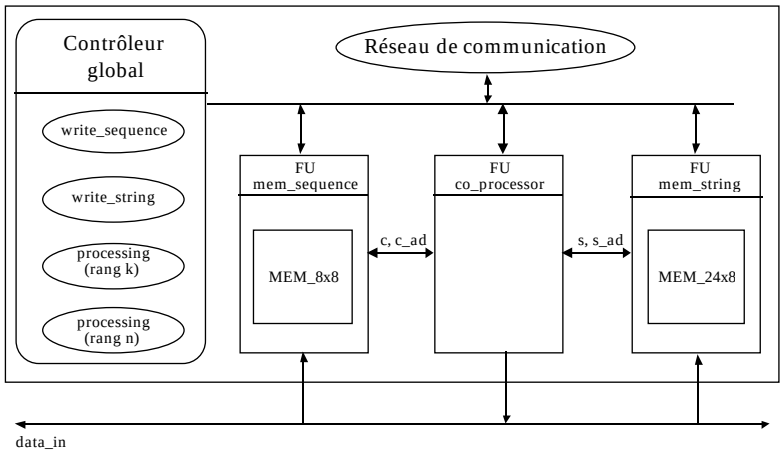


Figure IV.4 : Architecture abstraite du WSS.

3.3. Flot de conception pour une utilisation modulaire de la synthèse comportementale

Cette section applique la méthode présentée au chapitre III, §4.3 pour l'utilisation modulaire de la synthèse comportementale et nous utiliserons ici un seul outil de synthèse pour la conception des différents blocs. Cette simplification facilite une présentation détaillée de l'approche. Le système WSS se compose de trois blocs. En raison de leur complexité et dans le but de faciliter la tâche du contrôleur global et de respecter les performances, nous avons décidé que chaque bloc devrait contenir son propre contrôleur local. Le système global, de la même façon que ces blocs, sera décrit au niveau comportemental puis soumis à la synthèse comportementale. La figure IV.5 montre le flot de conception correspondant.

Les composants sont conçus dans un premier temps. Nous utiliserons *Amical* pour la synthèse des deux modules mémoires et du co-processeur. Dans un second temps, l'abstraction du co-processeur sert à son utilisation pour la description de l'ensemble du système. Finalement le contrôleur global est synthétisé en utilisant *Amical*.

Dans ce chapitre les sections 4 et 5 décrivent respectivement les composants et le WSS global conçus avec *Amical.*

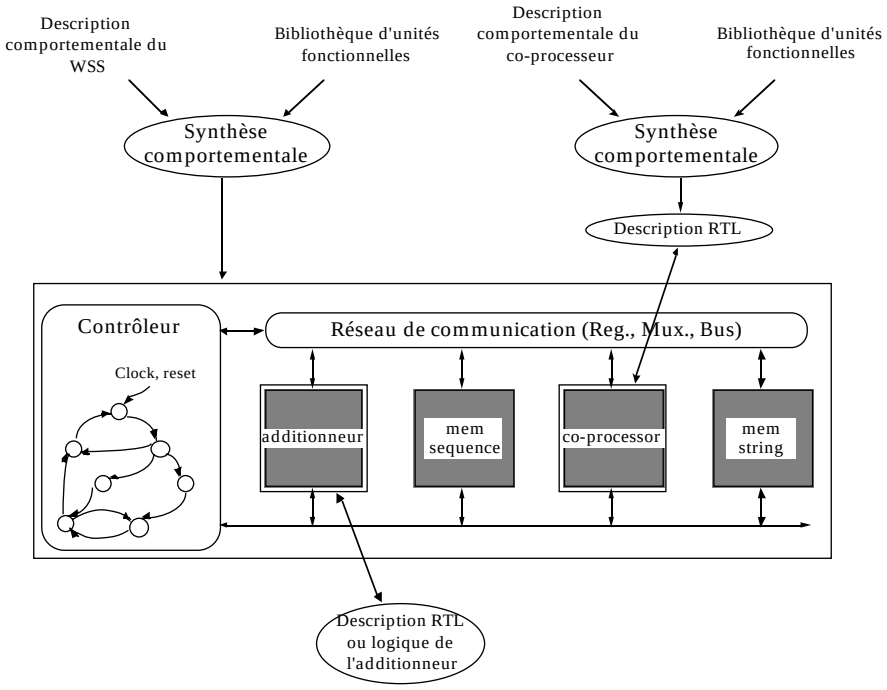


Figure IV.5 :Utilisation modulaire de la synthèse de haut niveau.

3.4. Style de la description VHDL

Au cours de la conception nous avons décidé d'utiliser une méthode de validation complète fondée sur une configuration unique de test et permettant de simuler toutes les parties du système à différents niveaux d'abstraction comme il est introduit au chapitre III, §4.8.

Ce choix impose plusieurs restrictions sur le style d'écriture du VHDL pour ce qui concerne les communications avec les mémoires et parfois avec le bus externe au niveau du cycle d'horloge. L'étape d'ordonnancement ne devait pas insérer de cycle supplémentaire au milieu de ces communications contraintes. Pour les composants nous avons mélangé le "mode cycle avec des entrées / sorties fixes" et le "mode comportemental avec des entrées / sorties fixes". Puisque les blocs mémoires interagissent avec le bus de

données externe et le co-processeur au niveau du cycle d'horloge, toutes les parties relatives à cette communication sont décrites en utilisant le mode cycle avec entrées / sorties fixes. Le reste du comportement, relatif à la communication avec le contrôleur global, est décrit en utilisant le mode d'état comportemental avec entrées / sorties fixes. Le contrôleur global est aussi décrit dans le mode d'état comportemental avec entrées / sorties fixes.

Comme la plupart des outils de synthèse, *Amical* interprète l'instruction *wait* comme une transition synchrone, même si la liste de sensibilité de cette instruction *wait* n'est pas relative à l'horloge. Afin d'avoir des descriptions synchrones équivalentes avant et après synthèse, le style d'écriture impose la synchronisation au niveau comportemental. Cette synchronisation est faite en utilisant la fonction IEEE “rising_edge” appliquée à l'horloge. On peut noter que l'instruction *wait* peut inclure des expressions booléennes complexes utilisant des signaux autres que l'horloge. Tous les composants et le contrôleur global sont connectés au signal de mise à zéro appelé *reset*. Les paquetages suivants sont utilisés pour les déclarations de sous-types et de composants. Les déclarations de sous-types sont nécessaires pour la synthèse par *Amical*, pour distinguer les différentes largeurs de signaux lorsque l'on utilise des types vecteurs.

```
1   LIBRARY ieee;
2           USE ieee.std_logic_1164.ALL;
3           USE ieee.std_logic_arith.ALL;
4
5   PACKAGE pkg_types IS
6
7           SUBTYPE bit1    IS std_ulogic;
8
9           SUBTYPE bit2    IS std_ulogic_vector ( 1 DOWNTO 0);
10          SUBTYPE bit3    IS std_ulogic_vector ( 2 DOWNTO 0);
11          SUBTYPE bit4    IS std_ulogic_vector ( 3 DOWNTO 0);
12          SUBTYPE bit5    IS std_ulogic_vector ( 4 DOWNTO 0);
13          SUBTYPE bit8    IS std_ulogic_vector ( 7 DOWNTO 0);
14          SUBTYPE bit11   IS std_ulogic_vector (10 DOWNTO 0);
15
16          SUBTYPE bit2_r  IS std_logic_vector  ( 1 DOWNTO 0);
17          SUBTYPE bit3_r  IS std_logic_vector  ( 2 DOWNTO 0);
18          SUBTYPE bit4_r  IS std_logic_vector  ( 3 DOWNTO 0);
19          SUBTYPE bit5_r  IS std_logic_vector  ( 4 DOWNTO 0);
20          SUBTYPE bit8_r  IS std_logic_vector  ( 7 DOWNTO 0);
21          SUBTYPE bit11_r IS std_logic_vector  (10 DOWNTO 0);
```

```
22
23 END pkg_types;
```

Figure IV.6 : Paquetage pour les déclarations de sous-types.

La figure IV.7 donne le paquetage de déclaration des composants. Les composants *mem_sequence*, *mem_string* et *co-processor* sont respectivement déclarés aux lignes 38-52, 85-100 et 102-118 du paquetage *pkg_components*. Chaque bloc mémoire contient un contrôleur local gérant une RAM double-ports {26-36} pour *mem_sequence* et {73-83} pour *mem_string*. En fait chaque RAM double-ports renferme elle-même le véritable composant mémoire (7-24 pour *dpram_8x8* et 54-71 pour *dpram_24x8*) avec un peu de logique combinatoire autour.

```
1  LIBRARY ieee;
2          USE ieee.std_logic_1164.ALL;
3          USE work.pkg_types.ALL;
4
5  PACKAGE pkg_components IS
6
7          COMPONENT dpram_8x8
8          PORT(
9                  q2      : OUT   bit8;
10                 q1      : OUT   bit8;
11                 d2      : IN    bit8;
12                 d1      : IN    bit8;
13                 a2      : IN    bit3;
14                 a1      : IN    bit3;
15                 oen2    : IN    bit1;
16                 oen1    : IN    bit1;
17                 wen2    : IN    bit1;
18                 wen1    : IN    bit1;
19                 csn2    : IN    bit1;
20                 csn1    : IN    bit1;
21                 ck2     : IN    bit1;
22                 ck1     : IN    bit1
23                 );
24         END COMPONENT;
25
26         COMPONENT mem_8x8
27         PORT(
28                 q2      : OUT   bit8;
29                 d1      : IN    bit8;
30                 a2      : IN    bit3;
31                 a1      : IN    bit3_r;
32                 sel_read: IN    bit1;
33                 sel_write: IN   bit1;
34                 ck      : IN    bit1
35                 );
36         END COMPONENT;
37
38         COMPONENT mem_sequence
39         PORT(
40                 clk     : IN    bit1;
```

```
41                  reset     : IN      bit1;
42                  sel_read: IN      bit1;
43                  c_sel     : IN      bit1;
44                  c_req     : OUT     bit1;
45                  c_ack     : IN      bit1;
46                  c_valid   : IN      bit1;
47                  data_in   : IN      bit8;
48                  c_ad      : IN      bit3;
49                  c         : OUT     bit8;
50                  c_done    : OUT     bit1
51                  );
52          END COMPONENT;
53
54          COMPONENT dpram_24x8
55          PORT(
56                  q2        : OUT     bit8;
57                  q1        : OUT     bit8;
58                  d2        : IN      bit8;
59                  d1        : IN      bit8;
60                  a2        : IN      bit5;
61                  a1        : IN      bit5;
62                  oen2      : IN      bit1;
63                  oen1      : IN      bit1;
64                  wen2      : IN      bit1;
65                  wen1      : IN      bit1;
66                  csn2      : IN      bit1;
67                  csn1      : IN      bit1;
68                  ck2       : IN      bit1;
69                  ck1       : IN      bit1
70                  );
71          END COMPONENT;
72
73          COMPONENT mem_24x8
74          PORT(
75                  q2        : OUT     bit8;
76                  d1        : IN      bit8;
77                  a2        : IN      bit5;
78                  a1        : IN      bit5_r;
79                  sel_read: IN      bit1;
80                  sel_write: IN     bit1;
81                  ck        : IN      bit1
82                  );
83          END COMPONENT;
84
85          COMPONENT mem_string
86          PORT(
87                  clk       : IN      bit1;
88                  reset     : IN      bit1;
89                  sel_read: IN      bit1;
90                  s_sel     : IN      bit1;
91                  burst     : IN      bit2_r;
92                  s_req     : OUT     bit1;
93                  s_ack     : IN      bit1;
94                  s_valid   : IN      bit1;
95                  data_in   : IN      bit8;
96                  s_ad      : IN      bit5;
97                  s         : OUT     bit8;
98                  s_done    : OUT     bit1
99                  );
100         END COMPONENT;
101
102         COMPONENT co_processor
103         PORT (
104                 clk       : IN      bit1;
```

```
105                 reset     : IN      bit1;
106                 c         : IN      bit8;
107                 s         : IN      bit8;
108                 p_sel     : IN      bit1;
109                 mode      : IN      bit1;
110                 sel_read: OUT     bit1;
111                 c_ad      : OUT     bit3;
112                 s_ad      : OUT     bit5;
113                 dmin       OUT      bit11;
114                 vector    : OUT     bit4;
115                 done0     : OUT     bit1;
116                 done1     : OUT     bit1
117                 );
118         END COMPONENT;
119 END pkg_components;
```

Figure IV.7 : Paquetage de déclaration des composants.

4. Spécifications comportementales des sous-systèmes

La première étape de la conception modulaire est la synthèse de haut niveau des sous-systèmes à réutiliser comme composants. Les trois composants : *mem_sequence*, *mem_string* et *co-processor* sont décrits. Cette section aboutit à la description VHDL comportementale complète de chaque module.

4.1. Le co-processeur

Le co-processeur a la charge d'exécuter le processus de comparaison introduit au chapitre IV, §2.1. Le co-processeur est connecté à deux mémoires, au bus externe et au contrôleur global selon la figure IV.8.

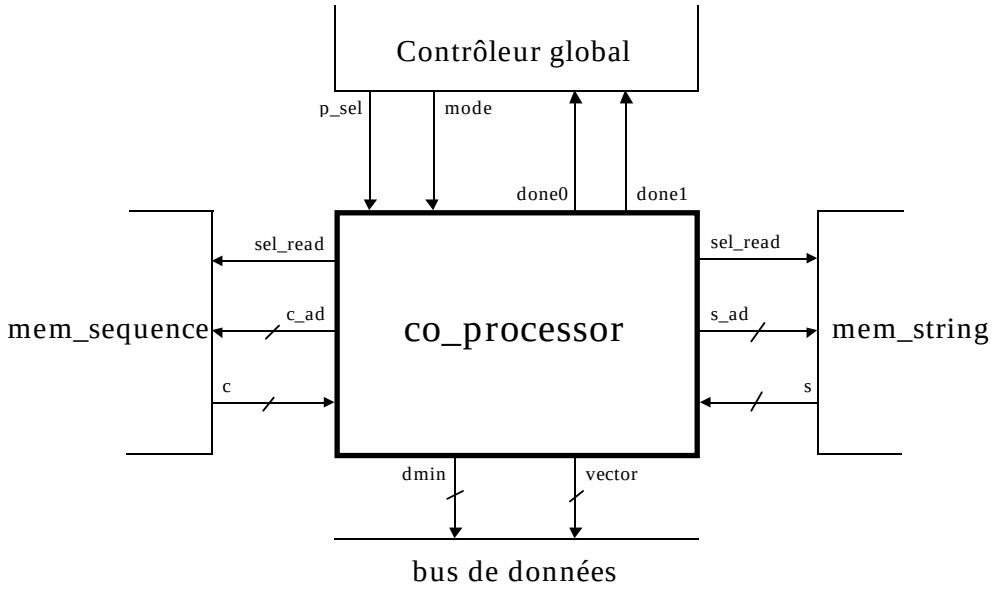


Figure IV.8 : Connexions du co-processeur.

Le protocole de communication avec les mémoires prend deux cycles. Pendant le premier cycle le co-processeur calcule les adresses, sélectionne les deux mémoires dans le mode de lecture *sel_read* et envoie les adresses aux mémoires *c_ad*, *s_ad*. Pendant le second cycle, les données issues des mémoires sont disponibles *c*, *s*.

Les signaux *p_sel* et *mode* actionnent le co-processeur. *p_sel* déclenche le co-processeur et *mode* détermine le mode de fonctionnement du co-processeur.

Le co-processeur possède deux modes de fonctionnement suivant la position de la chaîne traitée, car la dernière itération nécessite quelques calculs spécifiques. Toutes les itérations exceptée la dernière sont traitées dans le premier mode. Ce mode est sélectionné par (*mode* = '0') et (*p_sel* = '1'). A la fin du calcul, le co-processeur positionne à '1' le signal *done0*. La réalisation de la dernière itération utilise un mode spécifique (*mode* = '1' et *p_sel* = '1'). A la fin du calcul, il rétablit *done1* à '1'. Les signaux *done0*, *done1*, *mode* et *p_sel* sont externes.

En plus des opérations standard (+, *incr*, -), le co-processeur utilise plusieurs calculs spécifiques combinant des opérations arithmétiques, des concaténations et des conversions de type. Toutes ces opérations sont dissimulées dans un opérateur spécial appelé *min_addr* en raison des restrictions du style VHDL admis par *Amical* vues au chapitre III, §1.3.4. Chacune de ces opérations est exécutée comme une procédure VHDL comme nous le précisons dans la suite.

La communication avec les mémoires synchrones impose que les signaux d'entrée soient stables sur le front montant de l'horloge. Cette contrainte temporelle est difficile à spécifier dans des modèles comportementaux ignorant l'horloge. Cela provient du fait que toutes les affectations prennent zéro délai et par conséquent, pendant la simulation comportementale, toutes les affectations de signaux auront lieu sur le front montant de l'horloge. Pour éviter ce problème, lors des simulations comportementales et RTL, tous les positionnements à l'entrée des mémoires sont retardés en utilisant une clause "after" (*after 2 ns*). Ce style n'altère pas le style comportemental de la description puisque ces procédures ne sont pas mises à plat pendant la synthèse comportementale lorsque l'on utilise *Amical.*

La figure IV.9 montre l'architecture abstraite du co-processeur. Chaque appel de procédure, fonction ou opération requiert une unité fonctionnelle capable de l'exécuter et permet l'allocation de l'unité fonctionnelle pendant la synthèse comportementale avec *Amical*. Les procédures *assign*, *init_min* et *sel_min*, la fonction *dist* et l'opérateur + sont respectivement exécutés par les unités fonctionnelles nommées *min_addr*, *alu* et *incr*.

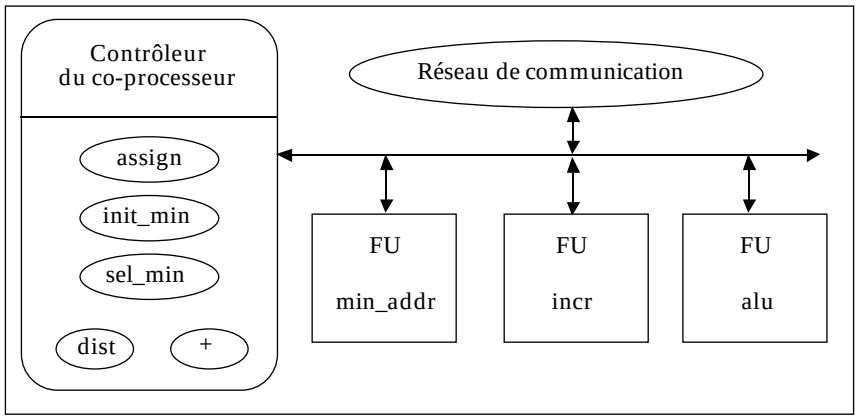


Figure IV.9 : Architecture abstraite du co-processeur.

La description comportementale VHDL du co-processeur est présentée dans la figure IV.10. Son architecture se compose d'un seul processus {25-108}.

```
1   LIBRARY ieee;
2           USE ieee.std_logic_1164.all;
3           USE ieee.std_logic_arith.all;
4           USE ieee.std_logic_signed.all;
5           USE work.pkg_types.all;
6
7   ENTITY co_processor IS
8   PORT (  clk       : IN      bit1;
9           reset     : IN      bit1;
10          c         : IN      bit8;
11          s         : IN      bit8;
12          p_sel     : IN      bit1;
13          mode      : IN      bit1;
14          sel_read: OUT     bit1;
15          c_ad      : OUT     bit3;
16          s_ad      : OUT     bit5;
17          dmin      : OUT     bit11;
18          vector    : OUT     bit4;
19          done0     : OUT     bit1;
20          done1     : OUT     bit1);
21          END co_processor;
22
23 ARCHITECTURE behavior OF co_processor IS
24 BEGIN
25    PROCESS
26    VARIABLE      i          : bit3_r;
27    VARIABLE      x          : bit4_r;
28    VARIABLE      d, min     : bit11_r;
29
30          FUNCTION dist (c, s: IN bit8; d: IN bit11_r) RETURN bit11_r IS
31          VARIABLE val8: bit8_r;
32          VARIABLE val11: bit11_r;
33          BEGIN
34                  val8 := to_stdlogicvector(c) - to_stdlogicvector(s);
35                  val8 := abs(val8);
36                  val11 := val8 + d;
```

```
37                RETURN(val11);
38        END dist;
39
40        PROCEDURE assign (i: IN bit3_r; x: IN bit4_r;
41        SIGNAL c_ad: OUT bit3; SIGNAL s_ad: OUT bit5) IS
42        BEGIN
43                c_ad <= to_stdulogicvector(i) AFTER 2 ns;
44                s_ad <= to_stdulogicvector(("00" & i) + ('0' & x)) AFTER 2 ns;
45        END assign;
46
47        PROCEDURE init_min (SIGNAL dmin: OUT bit11) IS
48        BEGIN
49                min := "11111111000";
50                dmin <= "00000000000";
51        END init_min;
52
53        PROCEDURE sel_min (d: IN bit11_r; x: IN bit4_r;
54        SIGNAL dmin : OUT bit11;
55        SIGNAL vector : OUT bit4) IS
56        BEGIN
57                IF unsigned(d) <= unsigned(min)
58                THEN
59                        min := d;
60                        vector <= to_stdulogicvector(x);
61                END IF;
62                dmin <= to_stdulogicvector(min);
63        END sel_min;
64
65        BEGIN
66                sel_read <= '0';
67                done0 <= '1';
68                done1 <= '1';
69
70                WAIT UNTIL p_sel='1' AND rising_edge(clk) AND reset='1';
71
72                IF mode='0' OR reset='0' THEN
73                        i := "000";
74                        x := "0000";
75                        d := "00000000000";
76                        init_min(dmin);
77                        vector <= "0000";
78                        done0 <= '0';
79                ELSE
80                        IF mode='1' THEN
81                                done1 <= '0';
82                        END IF;
83                END IF;
84
85                WAIT UNTIL p_sel='0' AND rising_edge(clk) AND reset='1';
86
87                loop_x: WHILE reset='1' LOOP
88                        d := "00000000000";
89                        loop_i: WHILE reset='1' LOOP
90                                sel_read <= '1';
91                                assign(i, x, c_ad, s_ad);
92
93                                WAIT UNTIL rising_edge(clk);
94
95                                sel_read <= '0';
96                                d := dist(c,s,d);
97                                i := i + 1;
98                                IF i=0 THEN
99                                        EXIT loop_i;
100                               END IF;
```

```
101                         END LOOP loop_i;
102                         sel_min(d, x, dmin, vector);
103                         x:= x + 1;
104                         IF (x=0 OR x=8) THEN
105                                 EXIT loop_x;
106                         END IF;
107                 END LOOP loop_x;
108         END PROCESS;
109 END behavior;
```

Figure IV.10 : description comportementale du co-processeur.

Le processus commence par la détection de *p_sel* actif pour les deux modes {70}, selon le protocole de la figure IV.11. L'initialisation {72-83} dépend du mode. Puis, deux boucles imbriquées suivent: *loop_x* pour balayer tous les vecteurs *x* {87-107} et *loop_i* pour l'accumulation de la distorsion {89-101}. Une itération de la boucle d'accumulation prend un cycle d'horloge. Pour chaque RAM une nouvelle adresse doit être positionnée à chaque cycle d'horloge pour accéder à une séquence, c'est pourquoi il y a une instruction *wait* sur le front montant de l'horloge à l'intérieur des boucles imbriquées {93}.

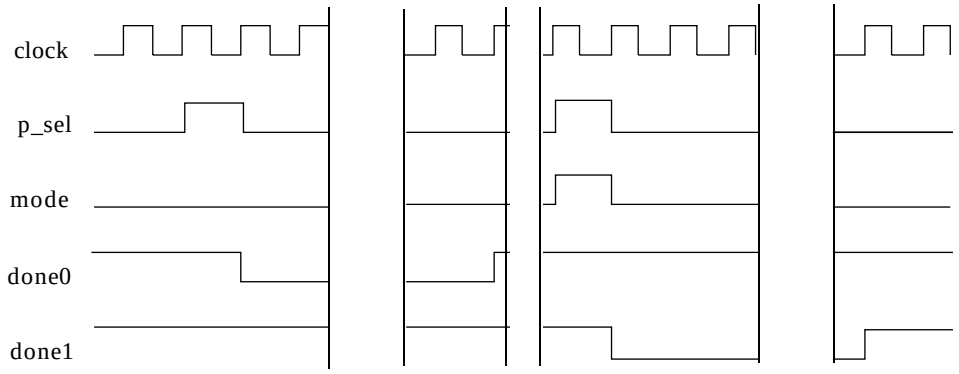
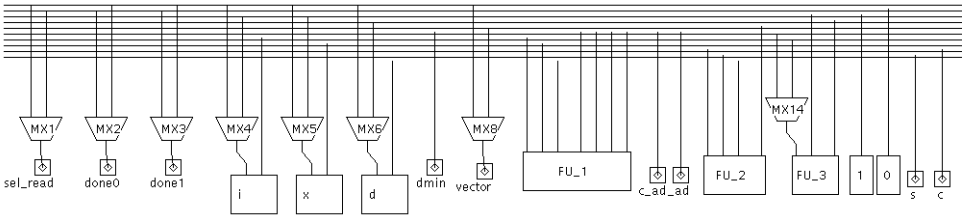


Figure IV.11 : Protocole d'activation entre le contrôleur global et le co-processeur.

La figure IV.12 montre le chemin de données du co-processeur, issu d'*Amical* à partir de la description VHDL de la figure IV.10. Cela schématise une architecture de chemin de données à base de multiplexeurs. Les unités fonctionnelles *FU_1*, *FU_2* et *FU_3* correspondent respectivement aux unités fonctionnelles (*min_addr*, *alu* et *incr*), et le contrôleur est une MEF de 7 états et 18 transitions. Il contrôle le chemin de données à travers 23 fils de contrôle.


MX1
MX2
MX3
MX4
MX5
MX6
MX8
MX14
sel_read
done0
done1
i
x
d
dmin
vector
FU_1
c_ad_ad
FU_2
FU_3
1
0
s
c

La description du processus principal de *mem_sequence* n'inclut pas la communication avec le co-processeur. Les connexions entre les modules mémoires et le co-processeur apparaissent dans la partie de l'architecture qui utilise une instruction de "port map". *Amical* ne synthétise pas cette communication qui sera insérée pendant l'étape de la génération de l'architecture (chapitre III, §1.3.1).

La synthèse de haut niveau de *mem_sequence* affecte deux unités fonctionnelles : la RAM (*dpram_8x8* pour la procédure *write_ram*) et un opérateur d'incrément (*c_inc* pour l'opération +).

La figure IV.14 illustre l'architecture abstraite de *mem_sequence*. La procédure *mem_write* et l'opérateur + sont respectivement utilisés par les unités fonctionnelles appelées *mem_8x8* et *c_inc*.

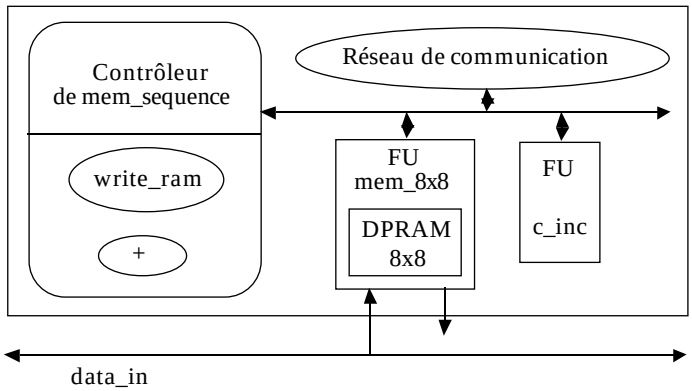


Figure IV.14 : Architecture abstraite de mem_sequence.

La description comportementale de *mem_sequence* apparaît dans la figure IV.15. Elle est commentée dans cette section. *mem_sequence* comporte une RAM pour mémoriser la séquence courante ainsi qu'un contrôleur local.

```
1  LIBRARY ieee;
2          USE ieee.std_logic_1164.ALL;
3          USE ieee.std_logic_unsigned.ALL;
4          USE work.pkg_types.ALL;
5          USE work.pkg_components.ALL;
```

```
6
7   ENTITY mem_sequence IS
8   PORT(  clk       : IN      bit1;
9          reset     : IN      bit1;
10         sel_read: IN      bit1;
11         c_sel     : IN      bit1;
12         c_req     : OUT     bit1;
13         c_ack     : IN      bit1;
14         c_valid   : IN      bit1;
15         data_in   : IN      bit8;
16         c_ad      : IN      bit3;
17         c         : OUT     bit8;
18         c_done    : OUT     bit1);
19 END mem_sequence;
20
21 ARCHITECTURE behavior OF mem_sequence IS
22 SIGNAL          a1, addr          : bit3_r;
23 SIGNAL          sel_write         : bit1;
24 BEGIN
25   PROCESS
26
27         PROCEDURE write_ram(sel_write: IN bit1; data: IN bit8) IS
28         BEGIN
29                 a1 <= addr AFTER 2 ns;
30         END write_ram;
31
32         BEGIN
33                 c_req <= '0';
34                 c_done <= '1';
35                 addr <= "000";
36                 sel_write <= '0';
37
38                 WAIT UNTIL (c_sel='1' AND rising_edge(clk) AND reset='1');
39
40                 c_done <= '0';
41                 c_req <= '1';
42
43                 WAIT UNTIL (c_ack='1' AND rising_edge(clk)) OR reset/='1';
44
45                 c_req <= '0';
46                 write_loop: LOOP
47                         IF c_valid/='1' AND addr<7 THEN
48
49                                 WAIT UNTIL (c_valid='1' AND rising_edge(clk))
                                   OR reset/='1';
50
51                                 IF reset/='1' THEN
52                                         EXIT write_loop;
53                                 END IF;
54                         END IF;
55                         addr <= addr + 1;
56                         write_ram(sel_write, data_in);
57                         sel_write <= '1';
58                         IF addr=7 OR reset/='1' THEN
59                                 WAIT UNTIL rising_edge(clk);
60                                 EXIT write_loop;
61                         END IF;
62                         WAIT UNTIL rising_edge(clk);
63                         sel_write <= '0';
64                 END LOOP write_loop;
65
66         END PROCESS;
67
68         sequence_mem : mem_8x8
```

```
69              PORT MAP ( c, data_in, c_ad, a1, sel_read, sel_write, clk);
70 END behavior;
```

Figure IV.15 : Description comportementale de mem_sequence

L'architecture VHDL consiste en une partie comportementale (processus {32-66}) et une partie structurelle {68-69}. Le composant de mémorisation *mem_8x8* est déclaré dans le paquetage *pkg_components* et instancié comme *sequence_mem* dans la partie structurelle. La partie comportementale consiste en un unique processus avec trois étapes principales : les initialisations {33-38}, la requête {40-43} et le chargement {45-64}.

Pendant l'initialisation, les signaux et les variables sont affectés à leur valeur initiale suivie d'un *wait* {38} jusqu'à la validation du signal d'activation : *c_sel*. L'adresse et les signaux de sortie (comme le signal d'autorisation d'écriture *sel_write* de la RAM) reçoivent leurs valeurs initiales.

L'étape de requête gère un protocole de requête *c_req* / acquittement *c_ack*, avec les opérateurs externes grâce à l'instruction *wait* {43}.

La phase d'écriture est le coeur du processus. Elle consiste en une boucle dans laquelle *data_in* est écrit dans *sequence_ram* {56} à l'adresse incrémentée *a1* {29}, tant que la fin de la rafale n'est pas atteinte {58}.

L'écriture des données suit deux conditions selon le chronogramme illustré par la figure IV.16. Premièrement, une donnée n'est valide que si elle arrive un cycle après le signal *c_valid* actif {47, 54}. Deuxièmement, l'autorisation d'écriture n'a lieu que lorsque la donnée est validée {57}, sinon l'écriture est interrompue {63}.

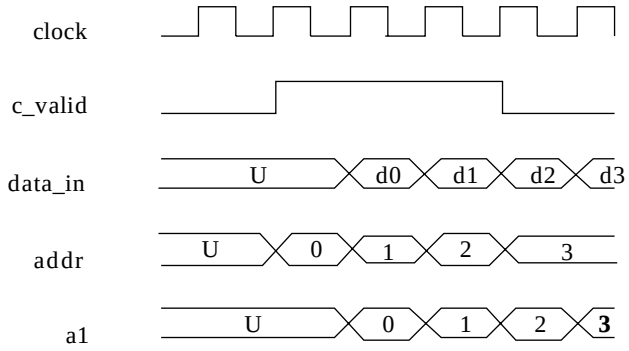
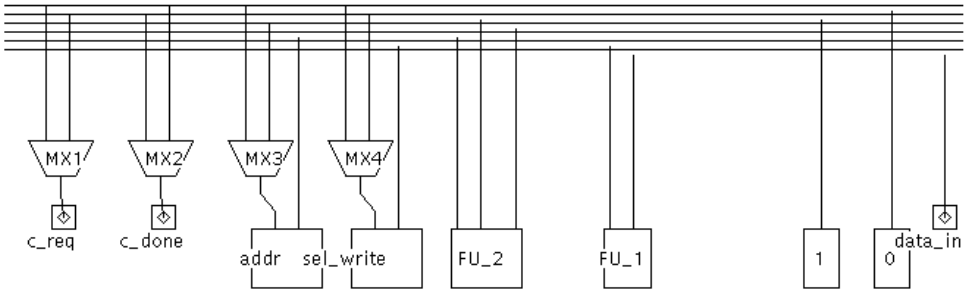


Figure IV.16 : Écriture dans la RAM.

La dernière étape est l'activation du signal de contrôle de sortie *c_done* signalant la fin du chargement jusqu'à la prochaine sélection {34}.

La contrainte de cette description est d'avoir un comportement au cycle d'horloge près pour remplir la RAM. Si *c_valid* vaut '1' pour toute la rafale sans interruption, la boucle "while" est alors exécutée à chaque cycle d'horloge. Une instruction *wait* apparaît dans la boucle, dans ce cas, pour les affectations de signaux. Comme l'adresse *a1* doit être prête à l'arrivée de la donnée *data_in*, il faut anticiper son calcul. Par conséquent, un autre signal *addr* est nécessaire. L'affectation effective de l'adresse est insérée à l'intérieur d'une déclaration de procédure dans *write_ram* {29} (figure IV.15). Elle permet de respecter les temps de "set-up" pendant la simulation comportementale (*after 2 ns*), et de manipuler une description purement comportementale.

La figure IV.17 montre le chemin de donnée de *mem_sequence*, issu d'*Amical* à partir de la description VHDL de la figure IV.15. Il schématise une architecture de chemin de données à base de multiplexeurs. Les unités fonctionnelles *FU_1* et *FU_2* correspondent respectivement aux composants *mem_8x8* et à l'opérateur d'incrément *c_inc*. Le contrôleur produit est une MEF de 6 états et 15 transitions. Il contrôle le chemin de données à travers 10 fils de contrôle.

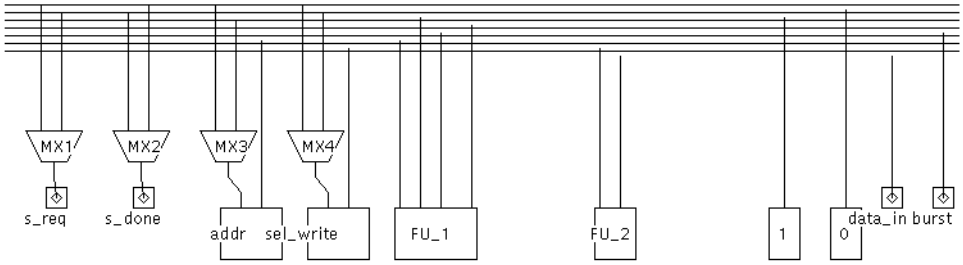

MX1
MX2
MX3
MX4
c_req
c_done
addr
sel_write
FU_2
FU_1
1
0
data_in

```
36                      a1 <= addr;
37              END write_ram;
38
39              BEGIN
40                      s_req <= '0';
41                      s_done <= '1';
42                      addr <= "00000";
43                      sel_write <= '0';
44
45                      WAIT UNTIL (s_sel='1' AND rising_edge(clk) AND reset='1');
46
47                      init(burst, addr);
48                      s_done <= '0';
49                      s_req <= '1';
50
51                      WAIT UNTIL ((s_ack='1' AND rising_edge(clk)) OR reset/='1');
52
53                      s_req <= '0';
54                      write_loop: LOOP
55                              IF s_valid/='1' AND addr/=7 AND addr/=15 AND addr<23 THEN
56
57                                      WAIT UNTIL (s_valid='1' AND rising_edge(clk))
                                        OR reset/='1';
58
59                                      IF reset/='1' THEN
60                                              EXIT write_loop;
61                                      END IF;
62                              END IF;
63                              addr <= addr + 1;
64                              write_ram(sel_write, data_in);
65                              sel_write <= '1';
66                              IF addr=7 OR addr=15 OR addr>=23 OR reset/='1' THEN
67                                      WAIT UNTIL rising_edge(clk);
68                                      EXIT write_loop;
69                              END IF;
70                              WAIT UNTIL rising_edge(clk);
71                              sel_write <= '0';
72                      END LOOP write_loop;
73
74              END PROCESS;
75
76              string_mem : mem_24x8
77              PORT MAP (s, data_in, s_ad, a1, sel_read, sel_write, clk);
78
79 END behavior;
```

Figure IV.18 : Description comportementale de mem_string.

L'architecture abstraite de *mem_string* est la même que l'architecture abstraite de *mem_sequence*. La procédure supplémentaire *init* est exécutée par l'unité fonctionnelle *s_inc*.

La figure IV.19 montre le chemin de données de *mem_string*, issu d'*Amical* à partir de la description VHDL de la figure IV.18. Elle schématise une architecture de chemin de données à base de multiplexeurs. Les unités fonctionnelles *FU_1* et *FU_2* correspondent respectivement au composant *mem_24x8* et à l'opérateur d'incrément *s_inc*. Le contrôleur produit, est une MEF à 6 états et 15 transitions. Il contrôle le chemin de données à travers 11 fils de contrôle.



Le WSS interagit directement avec le monde extérieur à travers deux signaux : la commande (commence le calcul) et le statut (indique l'état occupé ou la disponibilité des résultats). L'interface du WSS inclut plusieurs autres signaux qui sont utilisés par le co-processeur et les modules de mémoire pour communiquer avec le monde extérieur. Le schéma des interconnexions apparaît dans la figure IV.20.

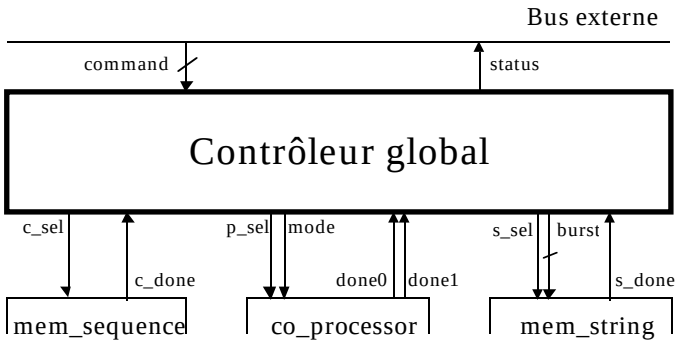


Figure IV.20 : Connexions du WSS.

La communication entre le contrôleur global et les sous-systèmes utilise un protocole très simple (figure IV.21). Pour activer un composant, le contrôleur global a besoin de positionner les signaux de sélection correspondants. Lorsqu'un composant est occupé, il remet à zéro le signal *done* correspondant. Cette interconnexion entre le contrôleur global et le sous-système figure dans IV.20.

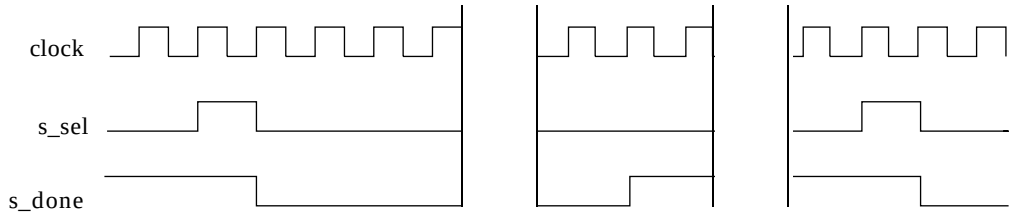


Figure IV.21 : Protocole d'activation de mem_string

Il existe deux façons de décrire un tel système. Les composants peuvent être utilisés comme composants externes ou comme unités fonctionnelles (figure IV.4).

Dans le premier cas, les signaux de communication sont directement accessibles dans la description comportementale. La figure IV.22 donne une description VHDL du contrôleur global comme une unité séparée. Le système complet est décrit dans la figure IV.23 comme l'assemblage de quatre composants : *top*, *mem_sequence*, *mem_string* et *co-processor*. L'architecture VHDL de *top* se compose d'un processus qui modélise le contrôleur global. Dans WSS, l'architecture structurelle contient *top* ainsi que les trois instances correspondant aux trois sous-systèmes. Dans ce cas, tout changement dans le protocole de communication entre les composants et le contrôleur global, impliquerait une modification de la description comportementale du contrôleur global. Par exemple, *mem_string* est activé trois fois à l'intérieur du contrôleur global. Un changement du protocole de communication entre le contrôleur global et *mem_string* entraînera la modification de la description du contrôleur global en trois endroits différents (figure IV.22 {51, 64, 77})

```
1   LIBRARY ieee;
2               USE ieee.std_logic_1164.ALL;
3               USE ieee.std_logic_arith.ALL;
4               USE ieee.std_logic_unsigned.ALL;
5               USE work.pkg_types.ALL;
6
7   ENTITY top IS
8   PORT (  clk         : IN        bit1;
9               reset       : IN        bit1;
10              command: IN     bit1;
11              c_sel       : OUT       bit1;
12              s_sel       : OUT       bit1;
13              burst       : OUT       bit2_r;
14              p_sel        OUT        bit1;
15              mode        : OUT       bit1;
16              c_done      : IN        bit1;
17              s_done      : IN        bit1;
18              done0       : IN        bit1;
19              done1       : IN        bit1;
20              status      : OUT       bit1);
21 END top;
22
23 ARCHITECTURE behavior OF top IS
24 CONSTANT         n                   : integer:= 2;
25 SIGNAL           burst_int           : bit2_r;
26 BEGIN
27   PROCESS
28
29              BEGIN
```

```
30          mode <='0';
31          burst_int <= "00";
32          p_sel <= '0';
33          c_sel <= '0';
34          s_sel <= '0';
35          status <= '0';
36          main_loop : LOOP
37
38                  WAIT UNTIL c_done ='1' AND s_done ='1' AND done0='1'
39                  AND done1='1' AND rising_edge(clk) AND reset='1';
40
41                  status <= '1';
42                  IF command/='1' THEN
43
44                          WAIT UNTIL command='1' AND reset='1'
                            AND rising_edge(clk);
45
46                  END IF;
47                  mode <='0';
48                  burst_int <= "00";
49                  status <= '0';
50                  c_sel <= '1';
51                  s_sel <= '1';
52
53                  WAIT UNTIL c_done='0' AND s_done = '0' AND rising_edge(clk);
54
55                  c_sel <= '0';
56                  s_sel <= '0';
57
58                  WAIT UNTIL c_done='1' AND s_done = '1' AND rising_edge(clk);
59
60                  IF reset/='1' THEN
61                          EXIT main_loop;
62                  END IF;
63                  burst_int <= burst_int + 1;
64                  s_sel <= '1';
65
66                  WAIT UNTIL s_done = '0' AND rising_edge(clk);
67
68                  s_sel <= '0';
69
70                  WAIT UNTIL s_done = '1' AND rising_edge(clk);
71
72                  IF reset/='1' THEN
73                          EXIT main_loop;
74                  END IF;
75                  WHILE burst_int < n AND reset='1' LOOP
76                          burst_int <= burst_int + 1;
77                          s_sel <= '1';
78                          p_sel <= '1';
79
80                          WAIT UNTIL s_done='0' AND done0='0'
                            AND rising_edge(clk);
81
82                          s_sel <= '0';
83                          p_sel <= '0';
84                          WAIT UNTIL s_done='1' AND done0='1'
                            AND rising_edge(clk);
85
86                  END LOOP;
87                  IF reset/='1' THEN
88                          EXIT main_loop;
89                  END IF;
90                  mode <= '1';
```

```
91                              p_sel <= '1';
92
93                              WAIT UNTIL done1='0' AND rising_edge(clk);
94
95                      END LOOP main_loop;
96              END PROCESS;
97
98              burst <= burst_int;
99
100 END behavior;
```

Figure IV.22 : Description comportementale VHDL du contrôleur global instancié dans l'architecture structurelle du WSS.

```
1   LIBRARY ieee;
2               USE ieee.std_logic_1164.ALL;
3               USE ieee.std_logic_arith.ALL;
4               USE ieee.std_logic_unsigned.ALL;
5               USE work.pkg_types.ALL;
6               USE work.pkg_components.ALL;
7
8   ENTITY wss IS
9   PORT (  reset     : IN      bit1;
10              clk       : IN      bit1;
11              command: IN     bit1;
12              c_req     : OUT     bit1;
13              c_ack     : IN      bit1;
14              c_valid   : IN      bit1;
15              s_req     : OUT     bit1;
16              s_ack     : IN      bit1;
17              s_valid   : IN      bit1;
18              data_in   : IN      bit8;
19              dmin      : OUT     bit11;
20              vector    : OUT     bit4;
21              status    : OUT     bit1);
22 END wss;
23
24 ARCHITECTURE rtl OF wss IS
25 CONSTANT         n                       : integer:= 2;
26 SIGNAL           c_sel, c_done           : bit1;
27 SIGNAL           s_sel, s_done           : bit1;
28 SIGNAL           p_sel, done0, done1     : bit1;
29 SIGNAL           mode, sel_read          : bit1;
30 SIGNAL           burst                   : bit2_r;
31 SIGNAL           s_ad                    : bit5;
32 SIGNAL           c_ad                    : bit3;
33 SIGNAL           s, c                    : bit8;
34 BEGIN
35
36              tp: top
37              PORT MAP(clk, reset, command, c_sel, s_sel, burst, p_sel, mode,
                c_done, s_done, done0, done1, status);
38
39              mq: mem_sequence
40              PORT MAP(clk, reset, sel_read, c_sel, c_req, c_ack,
                c_valid, data_in, c_ad, c, c_done);
41
42              mt: mem_string
43              PORT MAP(clk, reset, sel_read, s_sel, burst, s_req, s_ack,
                s_valid, data_in, s_ad, s, s_done);
44
45              pr: co_processor
46              PORT MAP(clk, reset, c, s, p_sel, mode, sel_read, c_ad, s_ad,
```

```
            dmin, vector, done0, done1);
47
48 END rtl;
```

Figure IV.23 : Description structurelle VHDL de WSS.

Une autre solution pour décrire ce modèle consiste à dissimuler le protocole de communication entre le contrôleur global et les sous-systèmes en utilisant des procédures. Une nouvelle description du WSS est donnée dans la figure IV.24. La sélection de *mem_string* s'effectue par un appel de procédure appelé *write_string*. Dans ce cas, une modification du protocole de sélection de *mem_string* induit seulement un unique changement dans la description comportementale du contrôleur global. La procédure *write_string* {42-45}, qui a la charge de sélectionner *mem_string*, est la seule partie du code qui doive être modifiée. Bien sûr lorsque le protocole de communication devient plus complexe et utilise plusieurs signaux, cette dissimulation du protocole devient beaucoup plus intéressante.

```
1   LIBRARY ieee;
2           USE ieee.std_logic_1164.ALL;
3           USE ieee.std_logic_arith.ALL;
4           USE ieee.std_logic_unsigned.ALL;
5           USE work.pkg_types.ALL;
6           USE work.pkg_components.ALL;
7
8   ENTITY wss IS
9   PORT (  reset     : IN      bit1;
10          clk       : IN      bit1;
11          command: IN     bit1;
12          c_req     : OUT     bit1;
13          c_ack     : IN      bit1;
14          c_valid   : IN      bit1;
15          s_req     : OUT     bit1;
16          s_ack     : IN      bit1;
17          s_valid   : IN      bit1;
18          data_in   : IN      bit8;
19          dmin      : OUT     bit11;
20          vector    : OUT     bit4;
21          status    : OUT     bit1);
22 END wss;
23
24 ARCHITECTURE behavior OF wss IS
25 CONSTANT       n                       : integer:= 2;
26 SIGNAL         c_sel, c_done           : bit1;
27 SIGNAL         s_sel, s_done           : bit1;
28 SIGNAL         p_sel, done0, done1     : bit1;
29 SIGNAL         mode, sel_read          : bit1;
```

```
30 SIGNAL          burst                    : bit2_r;
31 SIGNAL          s_ad                     : bit5;
32 SIGNAL          c_ad                     : bit3;
33 SIGNAL          s, c                     : bit8;
34 BEGIN
35   PROCESS
36
37          PROCEDURE write_sequence(val : IN bit1) IS
38          BEGIN
39                  c_sel <= val;
40          END write_sequence;
41
42          PROCEDURE write_string(val : IN bit1; burst: IN bit2_r) IS
43          BEGIN
44                  s_sel <= val;
45          END write_string;
46
47          PROCEDURE run(val : IN bit1) IS
48          BEGIN
49                  mode <= val;
50                  p_sel <= '1' AFTER 2 ns, '0' AFTER 22 ns;
51          END run;
52
53          BEGIN
54
55                  WAIT UNTIL done0 = '1' AND done1 = '1' AND rising_edge(clk);
56
57                  burst <= "00";
58                  write_sequence('0');
59                  write_string('0', "00");
60                  status <= '1';
61                  IF command/='1' THEN
62
63                          WAIT UNTIL command='1' AND rising_edge(clk);
64
65                  END IF;
66                  status <= '0';
67                  write_sequence('1');
68                  write_string('1', burst);
69
70                  WAIT UNTIL c_done='0' AND s_done = '0' AND rising_edge(clk);
71
72                  write_sequence('0');
73                  write_string('0', burst);
74
75                  WAIT UNTIL c_done='1' AND s_done = '1' AND rising_edge(clk);
76
77                  burst <= burst + 1;
78                  write_string('1', burst);
79
80                  WAIT UNTIL s_done = '0' AND rising_edge(clk);
81
82                  write_string('0', burst);
83
84                  WAIT UNTIL s_done = '1' AND rising_edge(clk);
85
86                  WHILE burst < n LOOP
87                          burst <= burst + 1;
88                          write_string('1', burst);
89
90                          WAIT UNTIL s_done = '0' AND rising_edge(clk) ;
91
92                          write_string('0', burst);
93                          run('0');
```

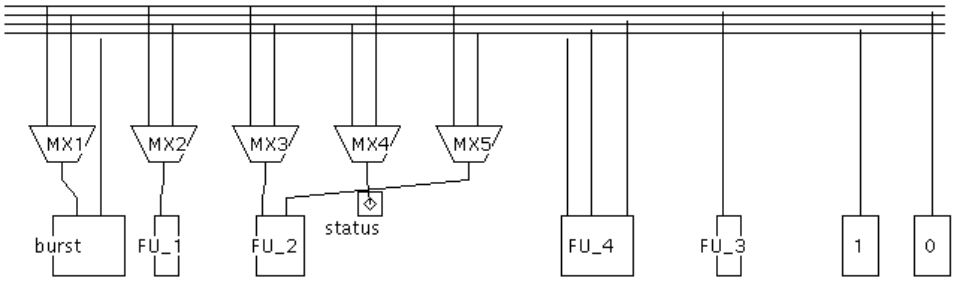
```
94
95                          WAIT UNTIL s_done='1' AND done0='1' AND rising_edge(clk);
96
97                  END LOOP;
98                  run('1');
99
100         END PROCESS;
101
102         mq: mem_sequence
103         PORT MAP(clk, reset, sel_read, c_sel, c_req, c_ack,
            c_valid, data_in, c_ad, c, c_done);
104
105         mt: mem_string
106         PORT MAP(clk, reset, sel_read, s_sel, burst, s_req, s_ack,
            s_valid, data_in, s_ad, s, s_done);
107
108         pr: co_processor
109         PORT MAP(clk, reset, c, s, p_sel, mode, sel_read, c_ad, s_ad,
            dmin, vector, done0, done1);
110
111 END behavior;
```

Figure IV.24 : Description VHDL comportementale de WSS.

Deux descriptions, données dans les figures IV.22 et IV.24, impliquent deux schémas de synthèse différents, lorsque l'on utilise *Amical* :

- Les sous-systèmes sont ignorés par la synthèse comportementale (description figure IV.22). Les procédures sont mises à plat dans la description du contrôleur global.
- Les sous-systèmes sont traités comme des unités fonctionnelles (description figure IV.24). Dans ce cas, les procédures sont interprétées comme des opérations s'exécutant sur des co-processeurs (composants). En plus du VHDL comportemental, il est nécessaire d'abstraire les sous-systèmes comme des unités fonctionnelles pour pouvoir les utiliser.

La figure IV.25 montre le chemin de données du WSS, issu d'*Amical* à partir de la description comportementale de la figure IV.24. Il s'agit d'une architecture de chemin de données à base de multiplexeurs. Les unités fonctionnelles *FU_1*, *FU_2*, *FU_3* et *FU_4* correspondent respectivement à *mem_sequence*, *mem_string*, *co_processor* et à l'opérateur d'incrément *w_inc*. Le contrôleur produit, est une MEF de 9 états et 21 transitions. Il contrôle le chemin de données à travers 9 fils de contrôle.



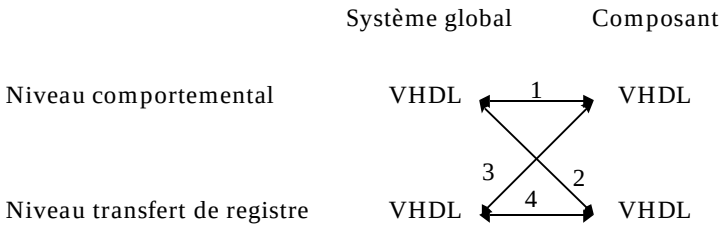


Figure IV.26 : Simulations multi-niveaux.

7. Conclusion

Ce chapitre a présenté une application modulaire de la synthèse comportementale pour la conception d'un système de recherche sur des fenêtres.

Le système se décompose en quatre modules : un co-processeur, deux modules mémoires, et un contrôleur global. Le processus de conception commence avec la description des trois premiers composants. Le contrôleur peut être conçu comme un simple composant interagissant avec les autres modules ou comme un processeur qui utilise les trois premiers modules comme des composants comportementaux.

Pour chaque module, nous avons détaillé la fonction, l'architecture abstraite, la description VHDL et l'architecture produite par *Amical*. Dans le style de la description VHDL, il a fallu distinguer le "mode cycle avec entrées/sorties fixes" et le "mode état comportemental avec entrées/sorties fixes". Ces modes étaient imposés pour permettre la simulation du modèle comportemental et le modèle RTL synthétisé par *Amical* en utilisant la même configuration de test.

Le chapitre V donne les résultats de synthèse pour un cas industriel, il s'agit de l'Estimateur de Mouvement d'un codeur d'images pour visiophone développé à ST. La fonctionnalité du WSS et la méthode utilisée pour sa conception ne sont en fait que des simplifications de la fonctionnalité et de la méthode utilisée pour la conception de l'Estimateur de Mouvement.

Chapitre V : Application à la Conception Industrielle - L'Estimateur de Mouvement

Ce chapitre présente la synthèse de haut niveau d'un opérateur complexe, l'Estimateur de Mouvement, du Vidéo CODEC H261 de ST dans des conditions industrielles [13, 59]. Cette expérience applique la méthode hiérarchique décrite au chapitre IV et combine deux outils de synthèse comportementale : le compilateur orienté flot de contrôle, *Amical*, et le compilateur orienté flot de données, *Cathedral-2/3*. La présence d'une partie dominée flot de contrôle (avec des protocoles de communication complexes) et d'une partie dominée flot de données (avec des calculs à haut débit) rend difficile la synthèse de ce circuit par un seul outil de synthèse de haut niveau. Cependant, pour combiner ces outils il a fallu définir un flot de conception sophistiqué permettant des simulations multi-niveaux, multi-langages. Ce flot est décrit dans le chapitre III, §4, figure III.10.

Pour évaluer le coût de cette méthode de “haut niveau” par rapport à la méthode classique, nous avons appliqué les mêmes conditions industrielles et utilisé le même environnement de simulation décrit au niveau du cycle d’horloge pour la validation des descriptions comportementale et RTL. Cependant, sans des communications asynchrones entre cet opérateur et l’environnement externe, cette expérience aurait été impossible. En comparant les deux modes de conception, il ressort que la synthèse de haut niveau induit une augmentation de surface négligeable de 5% contre une réduction du temps de conception et de la taille de la description (divisée par 5). La flexibilité en est aussi améliorée.

1. L’Estimateur de Mouvement du Visiophone CODEC

L’Estimateur de Mouvement considéré fait partie du Visiophone CODEC [60, 61]. Le CODEC code et décode des séquences d’images à travers un pipeline de 12 opérateurs communicants avec des bus de commande et de données (figure V.1). La partie d’encodage peut compresser des séquences d’images sans détérioration significative de la qualité pour éviter les redondances d'information entre deux images successives. La décision de la compression dépend des résultats de l’Estimateur de Mouvement. De ce fait, cet opérateur constitue un goulet d’étranglement et agit sur l’efficacité de l’ensemble de la puce.

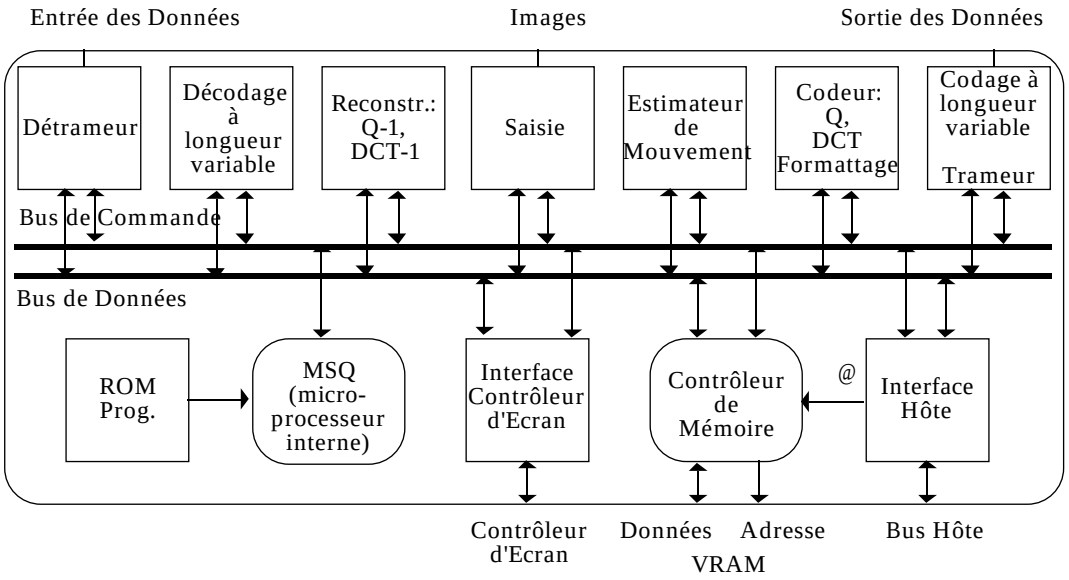


Figure V.1 : Architecture du Visiophone CODEC.

Tous les opérateurs du CODEC partagent la même Video-RAM (DRAM avec un accès série et un accès parallèle). Un contrôleur de mémoire spécifique gère les conflits d'accès à cette mémoire. Chaque opérateur contient sa propre mémoire locale. Les transferts entre la VRAM et les mémoires locales sont gérés par un protocole asynchrone complexe.

L'objectif de l'Estimateur de Mouvement est de déterminer le vecteur de mouvement d'une partie mobile d'une image, par comparaison avec l'image précédente, mémorisée dans la VRAM (figure V.1).

Le CODEC réalise l'algorithme de codage d'image de la norme H261 développée par l'organisme CCITT [58, 62]. Selon cet algorithme, on découpe une image en 99 macro-blocs pour permettre le parallélisme entre les différents opérateurs. La fenêtre de recherche du vecteur de mouvement est limitée aux 256 combinaisons de 16 vecteurs horizontaux et de 16 vecteurs verticaux dans l'image précédente comme l'illustre la figure V.2.

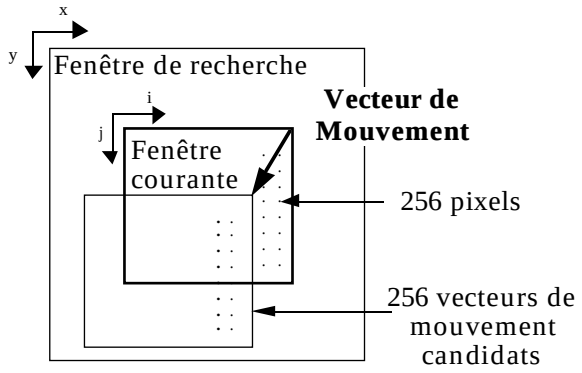


Figure V.2 : Fonctionnalité de l'Estimateur de Mouvement

L'Estimateur de Mouvement calcule la "distance" ou distorsion entre le macro-bloc courant et le macro-bloc cible, pour les 256 vecteurs de mouvement. Il peut ainsi déterminer le vecteur de mouvement recherché : ce vecteur correspond à la plus petite distorsion trouvée (la première trouvée dans le cas de plusieurs minima). En fait, pour chaque macro-bloc, on applique la formule qui suit, ce qui représente 97,3 millions d'opérations par seconde pour le débit requis de 15 images par seconde :

$$D_{min} = Min_{0\le x,y\le 15} [\Sigma_{0\le i,j\le 15} abs(s_{i+x,j+y} - c_{i,j})]$$

D_{min} : distorsion minimum trouvée entre le macro-bloc courant et les 256 macro-blocs de la fenêtre de recherche.

$c_{i,j}$: luminescence du pixel de coordonnées (i,j) dans le macro-bloc courant.

$s_{i+x,j+y}$: luminescence du pixel de coordonnées (i,j) dans le macro-bloc correspondant au vecteur de mouvement (x,y) de la fenêtre de recherche.

La fonction de l'Estimateur de Mouvement est principalement une opération dominée flot de données. Cependant, elle implique également un flot de contrôle très complexe. En effet, en raison de l'architecture distribuée du CODEC, l'estimateur de mouvement possède deux mémoires locales qui sont remplies en utilisant un protocole externe complexe. De plus il communique avec le Multi-SéQuenceur (qui ordonne l'ensemble des opérations de la puce) à travers un protocole bien défini.

L’Estimateur de Mouvement avait déjà été réalisé précédemment suivant le flot de conception classique RTL. C'est un circuit dont la conception est laborieuse en raison du manque de régularité de ses spécifications et de leurs fréquentes modifications. Ces raisons sont à l'origine du choix de ce circuit pour l'évaluation des outils de synthèse comportementale. Nous avons donc appliqué le flot de conception combinant deux outils de synthèse de haut niveau, exposé dans la figure III.10. On présente par la suite les résultats de l'expérience.

2. Application de la méthode et résultats

Cette section présente la manière dont nous avons appliqué le flot de conception du chapitre III, (§4, figure III.10) et la méthode exposée au chapitre IV à l’Estimateur de Mouvement et les résultats obtenus. Pour comparer la nouvelle méthode à la méthode classique, des contraintes étaient imposées sur la fréquence, sur les mémoires et sur les communications externes :

- Suivre des protocoles externes décrits au niveau du cycle : un bus de données externe met à jour ces RAMs synchrones par l'intermédiaire d’un protocole de requête/acquittement entre l’Estimateur de Mouvement, le MSQ et le contrôleur de la VRAM, qui organise la circulation des données.
- La fréquence de la puce s’élève à 13,5 MHz, mais la distorsion doit être calculée trois fois plus rapidement, (à 40,5 MHz), pour un débit de 15 images par seconde impliquant 97,3 millions d’opérations par seconde.
- Pour le stockage du macro-bloc et de la fenêtre de recherche, les spécifications fournissaient le modèle précis de deux mémoires avec leurs caractéristiques temporelles.

2.1. Le partitionnement

La conception de l'Estimateur de Mouvement commence par le partitionnement du circuit en une partie dominée flot de contrôle et une partie dominée flot de données qui seront respectivement traitées par *Amical* et *Cathedral-2/3*.

Le partitionnement est déterminant pour l'efficacité du flot de conception. Il peut être guidé par les spécifications fonctionnelles (une macro-fonction par module), par la facilité à décrire les modules (description séquentielle ou concurrente, liste d'instances de composants), par les types de communication en jeu (protocoles asynchrones, synchrones, flot de données), par le partage de ressources, etc. Un "bon partitionnement" devrait décomposer un circuit en sous-systèmes ou modules homogènes. Un module est homogène si un seul style suffit à décrire son architecture. Dans la suite nous allons considérer quatre types de modules.

2.1.1. Les tâches principales

La figure V.3 illustre la coordination des principales tâches impliquées dans le calcul de l'Estimateur de Mouvement. Dans le but d'atteindre la vitesse requise, l'algorithme est organisé pour une exécution des tâches en parallèle. Dans ce cas, ces tâches incluent le traitement et la mise à jour de la fenêtre de recherche.

Après l'étape d'initialisation, le remplissage de la mémoire cache courante commence la boucle de traitement : les 256 luminescences de pixels du macro-bloc courant sont chargées.

Pour chaque macro-bloc courant, le calcul complet du vecteur de mouvement est divisé en deux étapes successives : le "traitement de la 1ère moitié" et le "traitement de la seconde moitié". Pendant chaque étape, on anticipe la mise à jour d'une partie de la

fenêtre de recherche afin de ne pas perdre de temps pour le macro-bloc suivant.

Pour cet exemple, le partitionnement entre les tâches destinées à être traitées par *Amical* et *Cathedral-2/3* était guidé par les exigences en parallélisme et en fréquence. D'une part, les tâches à 40,5 MHz, le calcul de la distorsion et les accès aux pixels, devaient être assurés par la partie dominée flot de données. D'autre part, le reste du circuit, séquencement des instructions, gestion des mémoires et des communications, devait être synchronisé par l'horloge à 13,5 MHz. Ainsi, la description en Silage, pour une synthèse par *Cathedral-2/3*, était la solution la plus adaptée au co-processeur. Par ailleurs il était commode de décrire le contrôleur et tout l'assemblage du circuit en VHDL pour une synthèse par *Amical*.

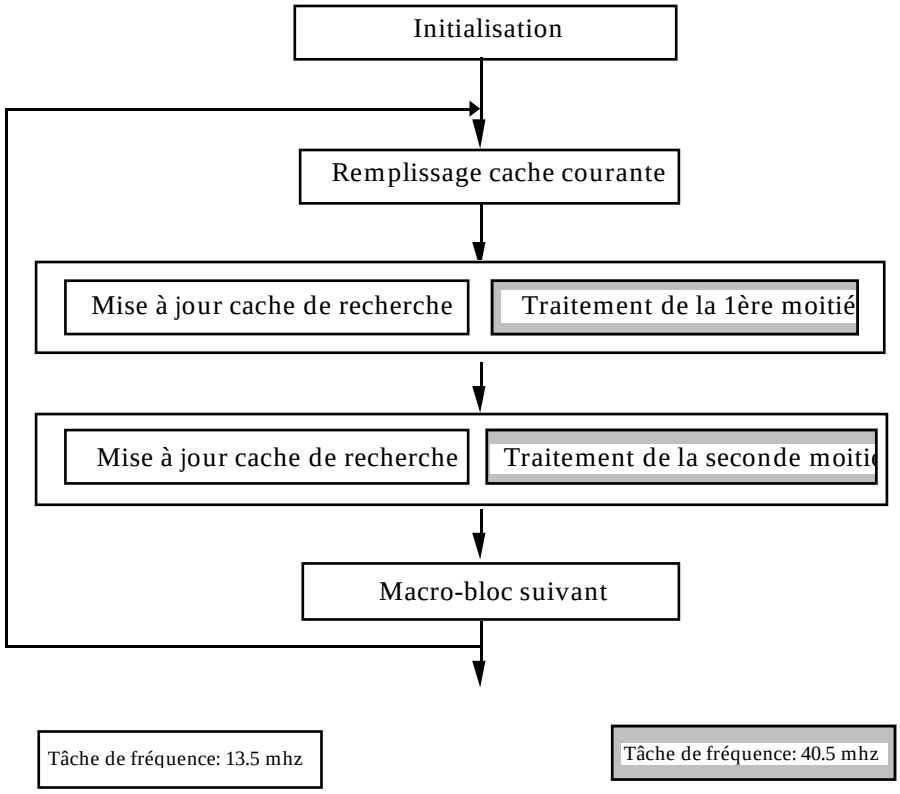


Figure V.3 : Algorithme du Contrôleur Global de l'Estimateur de Mouvement

2.1.2. Architecture du circuit

L'architecture globale du circuit résultant (figure V.4) se compose d'un ensemble d'unités fonctionnelles coopérantes *ALU, co-processeur, mémoire cache de recherche, mémoire cache courante* contrôlées par le *Contrôleur Global*.

L'UAL calcule l'adresse en écriture dans la mémoire cache courante.

Le co-processeur est inclus dans une architecture VHDL RTL afin de rendre son interface avec le reste du circuit plus flexible et d'admettre les fonctions de conversion. Il est actionné à travers un mécanisme de "protocole-inclusion" défini dans [59], (appel de procédure et instruction d'attente jusqu'au signal de fin). La description en Silage de l'ASU et sa synthèse avec *Cathedral-2/3* ont été réalisées à l'IMEC au sein de l'équipe VSDM.

Nous avons décidé d'inclure la RAM contenant la fenêtre de recherche, en ajoutant un contrôleur local, et de l'activer au moyen d'un autre "protocole-inclusion". En effet, sa mise à jour partielle apparaît à quatre reprises dans le processus principal (dont deux dans l'étape d'initialisation), et la séquence des calculs d'adresse s'avère complexe pour cette mémoire.

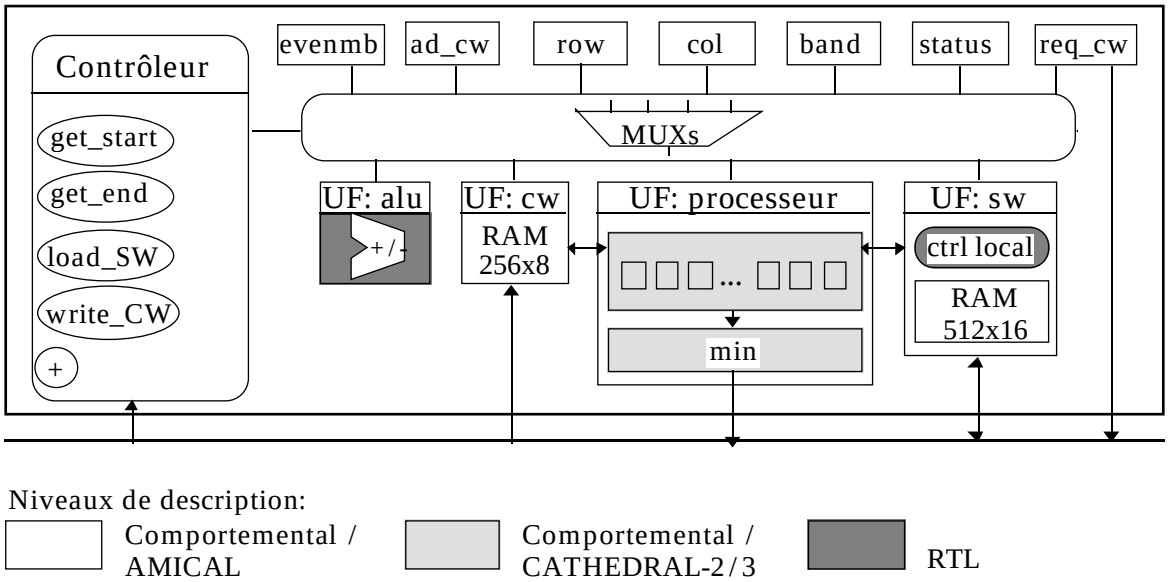


Figure V.4 : Architecture de l'Estimateur de Mouvement

Comme la mémoire qui contient le macro-bloc courant n'est remplie qu'une seule fois dans la boucle de traitement, et comme le calcul d'adresse, dans ce cas, n'a besoin que d'un opérateur d'incrément, nous l'avons juste instanciée sans logique supplémentaire.

De ce fait, les communications synchronisées par l'horloge à 40,5 MHz ont été ignorées par *Amical*. Le Contrôleur Global travaille à 13,5 MHz tandis que le co-processeur travaille à 40,5 MHz (figure V.3).

2.2. Résultats de synthèse

Nous avons utilisé le même outil commercial de synthèse logique que pour la précédente conception de l'Estimateur de Mouvement, suivant la méthode classique. Dans les deux méthodes, les circuits ont été réalisés avec une bibliothèque spécifique de cellules standard (en technologie HCMOS5 0,5 µm).

Le compilateur a fourni les estimations présentes dans les tableaux V.1, V.2 et V.3. Ces résultats (nombre de lignes VHDL et Silage, nombre de portes et registres, surface, chemin critique) sont comparés aux résultats obtenus par la méthode manuelle. On comparera séparément les parties dominées flot de contrôle et flot de données, et l'ensemble du circuit.

Pour atteindre la fréquence requise, nous avons inséré un niveau de pipeline dans l'ASU puis replacé les registres (pour *retiming*) avec *Cathedral-3.* Nous avons également introduit un pipeline du contrôle dans le circuit global avec *Amical*. Les résultats correspondants sont analysés plus loin.

Paramètres	**Classique**	***Amical***	**Comparaison**
Nb cellules logiques combinatoires	1146	1173	+2%
Nb cellules logiques séquentielles	82	100	+22%
Surface totale	**0,300 mm^2**	**0,342 mm^2**	**+14%**
Chemin critique	**65,9 ns**	**60,0 ns**	**-9%**
Lignes de code pour la spécification	**668 (VHDL RTL)**	**136 (VHDL comp.)**	**-80%**

Tableau V.1 : Comparaison entre la méthode classique RTL et la méthode de "haut niveau" avec Amical

Paramètres	**Classique**	***Cathedral-2/3***	**Comparaison**
Nb cellules logiques combinatoires	3153	2630	-16%
Nb cellules logiques séquentielles	551	877	+59%
Surface totale	**1,17 mm^2**	**1,22 mm^2**	**+4%**
Chemin critique	**22,0 ns**	**22,4 ns**	**+2%**
Lignes de code pour la spéci fication	**1381 (VHDL RTL)**	**300 (Silage)**	**-78%**

Tableau V.2 : Comparaison entre la méthode classique RTL et la méthode de "haut niveau" avec Cathedral-2/3

Paramètres	Classique	Haut Niveau	Comparaison
Nb cellules logiques combinatoires	4299	3669	-15%
Nb cellules logiques séquentielles	633	1010	+60%
Surface totale	**1,47 mm^2**	**1,54 mm^2**	**+5%**
Lignes de code pour la spécification	**2049**	**436**	**-79%**

Tableau V.3 : Comparaison entre la méthode classique RTL et la méthode de "haut niveau" pour l'ensemble du circuit

Une configuration de test RTL existait déjà. Elle a permis de valider les descriptions comportementales, puis de combiner les niveaux d'abstraction (ex : co-processeur au niveau portes / contrôleur global au niveau comportemental). Nous avons validé toutes les étapes de vérification jusqu'au niveau portes.

2.3. Analyse des résultats

L'avantage de cette méthode est évidemment l'automatisation du passage au niveau transfert de registre et ainsi, la flexibilité vis-à-vis des modifications architecturales, sans augmentation significative de surface. Une analyse détaillée des résultats ci-dessus et de la boucle de conception suivent dans cette section.

La description comportementale est flexible en raison de sa faible longueur (divisée par 5!). Le surplus en surface de 5% sur l'ensemble du circuit est raisonnable : il s'élève en particulier à 7% pour la partie dominée flot de contrôle et à 4% pour la partie dominée flot de données. Il est dû à l'introduction des étages de pipeline. L'augmentation de surface ne représente en fait que moins de 1% pour l'ensemble de la puce.

Le point commun entre les deux outils de synthèse architecturale réside dans l'augmentation du nombre de portes de logique séquentielle et dans la réduction du nombre de portes de logique combinatoire. Les organisations régulières des architectures génériques, les mémorisations nécessaires à travers les niveaux hiérarchiques et les insertions d'étages de pipeline en sont à l'origine. Il faut aussi tenir compte du compromis entre logique séquentielle et logique combinatoire.

3. Analyse du gain en temps de conception

3.1. Répartition du temps de conception

La figure V.5 présente la répartition des temps de validation pour les différentes étapes du flot de conception. Mais comme cette évaluation s'arrête à la validation de la description au niveau portes, il est difficile d'établir une comparaison entre les figures I.3 et V.5. Cependant, on peut noter le rapport entre le temps de conception mis pour la synthèse logique et le temps mis avant de valider la description RTL obtenu.

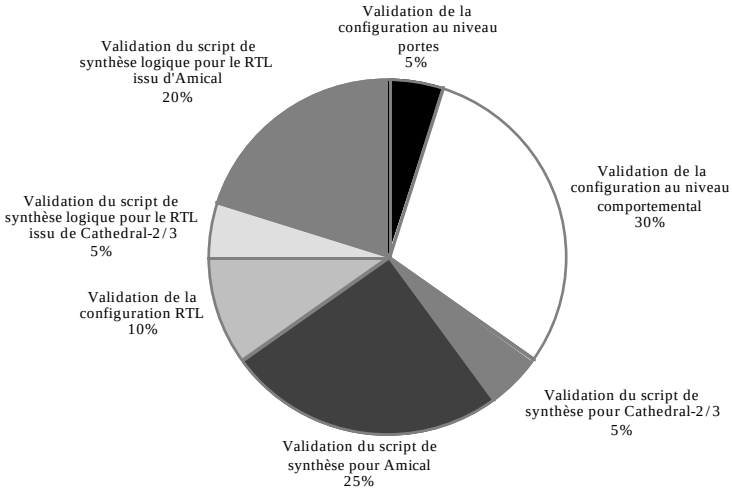


Figure V.5 : Répartition du temps de conception entre les différentes étapes du flot de synthèse logique ou comportementale.

On constate que la mise au point des descriptions comportementales est l'étape la plus longue. Ce temps se justifie par l'évolution des spécifications au cours du projet et la mise au point des descriptions comportementales pour leur admission par les outils de synthèse respectifs.

A l'issue de la synthèse, les descriptions RTL ne nécessitent aucune mise au point pour leur admission par un outil commercial de synthèse logique (*Design Compiler*) avec une entrée VHDL.

Il apparaît également que l'étape de validation de la synthèse de la partie dominée flot de données est plus courte que celle de la partie dominée flot de contrôle. L'explication provient d'une part de la régularité de la partie calcul et de l'irrégularité de la partie contrôle, et d'autre part de la faible interactivité de *Cathedral-3* par rapport à *Amical*.

3.2. Comparaison des temps de conception manuels et automatiques

Pour évaluer le gain de temps de la méthode de conception automatique, nous avons pris comme référence le temps de conception de la méthode classique, soit approximativement 18 homme x mois. Le temps de conception de la méthode à base de synthèse de haut niveau se décompose comme suit :

- Description et validation de la partie décrite en Silage jusqu'à validation du RTL : environ 1 homme x mois
- Description et validation du VHDL comportemental et du contrôleur de la fenêtre de recherche et des scripts de synthèse pour *Amical* jusqu'à validation du RTL : environ 4 homme x mois
- Description des scripts de synthèse logique jusqu'à la validation au niveau portes : environ 2 homme x mois

Le temps de conception global s'élève donc à environ 7 homme x mois, soit un facteur 1/2 à 1/3 par rapport au temps de référence. Cette réduction du temps de conception est l'un des avantages majeurs de la méthode automatique. Les nombreuses raisons sont évoquées dans la suite.

Les outils de synthèse de haut niveau sont beaucoup plus rapides que les traductions manuelles. La synthèse par *Amical* dure quelques minutes. La mise au point des fichiers d'abstraction des unités fonctionnelles peut demander quelques passages par *Amical*. La synthèse par *Cathedral-2/3* dure quelques minutes pour la partie synthèse de haut niveau. L'introduction de pipeline et les optimisations au niveau portes prennent une heure supplémentaire. Les itérations avec différents niveaux de pipeline durent 10 minutes. Les optimisations au niveau portes peuvent aussi être réalisées par des outils de synthèse logique disponibles dans le commerce.

Le nombre d'itérations à travers le synthétiseur logique est réduit grâce à la génération automatique des descriptions RTL VHDL. De plus, cela dépend seulement de la mise au point des descriptions RTL en VHDL des unités fonctionnelles. Une itération dure plusieurs heures. Par ailleurs, la mise au point du script de synthèse dure le même temps quelle que soit la méthode.

L'étape de correction des bogues est délicate et hiérarchique en commençant par la validation des sous-systèmes, pour la méthode automatique. En particulier, nous avons décrit une configuration de test indépendante pour le co-processeur afin de lui fournir un flot de données régulier (avec les mêmes stimuli que pour la configuration de test global imposée) et de vérifier les résultats. Ensuite, la validation du circuit global devenait possible.

4. Conclusion

Le succès de la synthèse architecturale de cet Estimateur de Mouvement par *Amical* et *Cathedral-2/3* a montré plusieurs gains appréciables, par rapport à la méthode manuelle classique. La longueur de la description a été divisée par 5. Le temps de conception a également diminué. Il en découle une flexibilité du modèle du circuit, en particulier vis-à-vis de l'évolution des normes en transmission de l'image et des technologies.

Ainsi, pour réaliser la synthèse de haut niveau de circuits industriels hétérogènes, il est possible de faire appel à la combinaison la plus adaptée d'un outil orienté flot de contrôle et d'un outil orienté flot de donnée. Dans ce cas, le concepteur doit considérer que le partitionnement entre ces parties est déterminant pour la suite de la conception et que la vérification nécessite des simulations multi-niveaux et éventuellement multi-langages avec une unique configuration de test. La solution idéale serait d'utiliser un environnement unique qui supporterait les deux types de synthèse.

Chapitre VI : Évaluation et Perspectives de la Synthèse de Haut Niveau

Au commencement de cette thèse, en 1994, à ST débutait l'évaluation industrielle des outils de synthèse comportementale. En 1997, au terme de la thèse, les conclusions de cette évaluation sont partagées suivant les domaines d'application. La synthèse comportementale orientée flot de données profite de la maturité de ses travaux de recherche, par rapport à la synthèse comportementale orientée flot de contrôle. Ainsi, les premières applications industrielles de la synthèse comportementale ont servi à la réalisation de circuits orientés flot de données, comme les filtres de traitement du signal [63]. Mais les circuits à partie contrôle complexe n'offrent pas encore le gain attendu en productivité. En effet, l'utilisation des outils est encore trop contraignante sur plusieurs points :

- chaque outil de synthèse architecturale impose des règles d'écriture à ses descriptions d'entrée.
- la gestion des mémoires, dont la diversité a souvent été sous-estimée par les concepteurs d'outils de CAO,
- la durée de l'apprentissage de l'outil, etc.

C'est pourquoi les utilisateurs reviennent parfois à la méthode de conception classique ou se reportent sur des architectures programmables. Il reste donc encore à perfectionner ces outils pour simplifier leur utilisation et élargir leur domaine d'applications.

Ce chapitre fait le point sur les problèmes résolus par la synthèse de haut niveau et sur les défis qu'il lui reste à remporter pour devenir plus efficace dans l'industrie. Lorsque la synthèse architecturale et ses méthodes d'application seront maîtrisées, il sera possible d'utiliser les outils de synthèse système déjà nécessaires.

Après le succès de l'évaluation des outils *Amical* et *Cathedral-2/3* présentée au chapitre V, ST a décidé d'appliquer la même méthode pour concevoir un circuit beaucoup plus complexe en termes d'accès aux mémoires et comportant 12 modes de fonctionnement différents. Nous avons donc tenté de tirer profit de la première expérience. Mais après quelques mois de travail, ce projet a été abandonné. Le circuit a finalement été réalisé en peu de temps sur une architecture programmable, totalement différente de l'architecture matérielle initiale. Cette nouvelle architecture a tiré les enseignements des problèmes rencontrés lors de la synthèse de haut niveau. Les raisons de l'abandon, relatives aux outils de synthèse comportementale et à la méthode appliquée, sont évoquées dans la suite. C'est donc sur cette seconde expérience que se fondent les conclusions de cette thèse.

1. Les problèmes résolus par la Synthèse de Haut Niveau

Même si les outils de synthèse comportementale n'ont pas encore atteint leur pleine maturité à ce jour, ils offrent déjà des possibilités prometteuses, comme l'automatisation de l'ordonnancement et de l'introduction de pipeline, l'exploration architecturale, la réutilisation de composants.

1.1. Automatisation et optimisation de l'ordonnancement

Sans conteste, c'est l'automatisation de l'ordonnancement qui apporte le confort le plus important : facilité et souplesse d'écriture par rapport à la description manuelle d'une machine d'états finis. De plus, les algorithmes d'ordonnancement utilisés optimisent suivant les cas, le séquencement ou le parallélisme des tâches. Ces optimisations ne seraient pas systématiques si elles étaient manuelles. Une conséquence très importante est la facilité à valider la description comportementale par rapport à la description RTL.

1.2. Introduction automatique d'étages de pipeline

Lors de la transposition du résultat de la synthèse comportementale sur une bibliothèque de cellules, les analyses temporelles peuvent montrer que cette architecture est loin d'atteindre les performances requises. Dans ce cas il vaut mieux introduire un étage de pipeline sur le chemin critique. L'introduction automatique d'étages de pipeline par l'outil de synthèse comportementale évite une reprise de la description source, avec les erreurs qu'elle peut entraîner. Il peut s'agir d'un repliement de boucle, d'un pipeline de chemin de données, ou d'un pipeline du contrôle (tableau II.1).

Cependant certains outils ne permettent pas d'isoler le chemin critique à raccourcir. L'insertion d'un étage de pipeline se fait uniformément, sur l'ensemble de l'architecture. On constate, comme dans l'application sur l'Estimateur de Mouvement, une augmentation non négligeable du nombre de bascules.

1.3. Réutilisation de composants

Certains outils de synthèse comportementale autorisent la réutilisation d'éléments déjà décrits et validés. Ces éléments sont intégrés dans une bibliothèque et leur comportement est abstrait de façon à être pris en compte par l'outil de synthèse. Ils sont inclus par un appel d'opération, de fonction ou de procédure dans la description comportementale. On peut citer *Amical* avec sa bibliothèque d'unités fonctionnelles ainsi que *Behavioral Compiler* avec sa bibliothèque de composants *DesignWare*. La bibliothèque *DesignWare* n'intègre à ce jour que des composants à nombre de cycles d'exécution fixe, quel que soit leur complexité (mémoires, opérateurs combinatoires, etc.). Tandis que la bibliothèque d'unités fonctionnelles d'*Amical* admet également des composants à nombre de cycles d'exécution variable.

Rappelons un exemple de réutilisation de composants dont, le nombre de cycles d'exécution est variable et imprévisible. Dans l'expérience décrite dans le chapitre V, §2.1.2, *Amical* "réutilise" l'une des mémoires avec son contrôleur. Or la disponibilité des données d'entrée dépend de l'occupation du bus de données du système (chapitre V, §1). Pour permettre la réutilisation de ce type de composant nous avons mis au point un protocole de communication asynchrone entre le contrôleur global de l'estimateur de mouvement et le contrôleur de remplissage de la mémoire. Et c'est ce protocole qui a été abstrait lors de l'intégration de ce composant dans la bibliothèque d'unités fonctionnelles. En réalité, ce type de tâche incombe plutôt à la conception système qui consiste à réaliser le partitionnement d'un circuit en modules et à synthétiser les communications entre ces parties. Mais comme cette étape est encore manuelle aujourd'hui, cela signifie que ces contrôleurs devraient se trouver au même niveau hiérarchique et que leur communication devrait être mise au point avant la synthèse architecturale.

De la même façon, nous avons "réutilisé" le co-processeur issu de *Cathedral-2/3*. Il était intéressant d'intégrer un bloc issu d'un outil de synthèse comportementale dans la bibliothèque extensible d'unités fonctionnelles d'un second outil de synthèse. En effet, cette expérience contribue à se préparer à la généralisation de la conception de blocs réutilisables. Car la réutilisation est la solution la plus simple au problème de productivité. C'est pourquoi dans le chapitre V nous avons choisi la méthode hiérarchique proposée au chapitre IV. Mais dans ce cas, l'expérience montre qu'il est beaucoup plus simple d'appliquer l'autre méthode du chapitre IV, qui met à plat le contrôleur et les composants qu'il commande.

La figure VI.1 illustre les différents types de réutilisation dans le flot de conception, en supposant l'existence de la synthèse système à venir. En règle générale on peut affirmer qu'une étape de synthèse utilise des bibliothèques d'objets décrits au niveau d'abstraction inférieur. Dans la pratique ce sera à la synthèse système d'utiliser des bibliothèques de composants à nombre de cycles d'exécution variables ou très importants, décrits eux-mêmes au niveau comportemental.

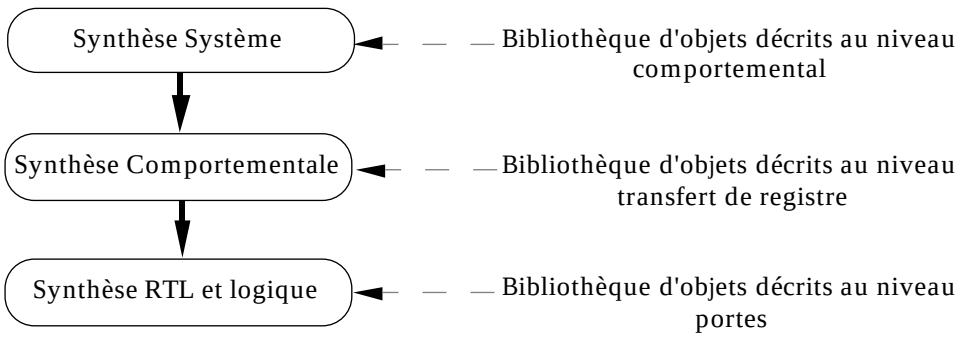


Figure VI.1 : La réutilisation dans le flot de conception.

2. Les défis lancés à la Synthèse de Haut Niveau

Les expériences menées dans l'industrie montrent que le gain en productivité n'est pas encore suffisant. Il reste à améliorer les outils, à simplifier leur utilisation et à généraliser les méthodes de conception à base de synthèse comportementale. Bergamaschi liste encore d'autres problèmes dans [15].

Actuellement, le concepteur doit bien connaître l'outil et ses algorithmes d'optimisation pour l'utiliser de façon satisfaisante. Le partitionnement et les styles d'écriture dans la description comportementale sont en cause car ils ont encore trop d'impact. Par exemple, une description comportementale correcte ne produit pas forcément une description correcte au niveau transfert de registres après synthèse architecturale. Comme il est expliqué au chapitre III, §1.3.4, l'ordonnancement peut insérer des cycles d'horloge supplémentaires par rapport au niveau comportemental. Or il reste très difficile de corriger les bogues d'une description RTL générée automatiquement. Enfin, la gestion des mémoires et le traitement de leurs interfaces sont fastidieux. Des efforts sur tous ces points qui allongent le temps de conception sont primordiaux.

2.1. Les conséquences du partitionnement.

La qualité d'une synthèse architecturale dépend des données d'entrée qui résultent des étapes de partitionnement et de description manuelle du comportement. Un partitionnement ou un style de description comportemental mal adaptés conduisent à des difficultés pour obtenir une architecture validée, atteindre les performances requises et gérer le flot de conception. L'utilisateur peut revenir sur son partitionnement d'origine au cours de la conception pour les raisons qui suivent.

Partitionnement RTL / comportemental

Aujourd'hui, il arrive que, même lorsque l'on fait appel à la synthèse de haut niveau, certaines parties doivent être décrites au niveau transfert de registre. La description d'entrée d'un outil de synthèse architecturale peut par exemple comprendre des parties dont le comportement au niveau du cycle d'horloge ne doit pas changer après synthèse. Par sécurité, ou suivant le style de description recommandé par l'outil, le concepteur décrit directement ces parties au niveau transfert de registre après les avoir extraites de la description comportementale.

Hiérarchie / Mémorisation

Un découpage induisant plusieurs niveaux de hiérarchie peut entraîner des mémorisations multiples qui peuvent être évitées dans une solution manuelle : des données à transférer d'un niveau de hiérarchie à un niveau inférieur doivent souvent être mémorisées aux deux niveaux. Car les connexions entre les registres internes du circuit et les sous-systèmes peuvent être interrompues par un multiplexeur intermédiaire. En effet les possibilités de partage de ressources n'assurent pas toujours des connexions permanentes. Il faudrait que la synthèse d'un niveau de hiérarchie prennent en compte les niveaux inférieurs. Un "bon" partitionnement permettra à des registres internes d'être visibles de partout sans nécessiter de duplication pour cause de partage de ressources à travers un multiplexeur.

Localisation du pipeline

L'introduction de pipeline reste parfois plus intéressante lorsqu'elle est manuelle, car le concepteur ne peut la localiser ni avec *Amical* ni avec *Cathedral-2/3*, comme le montre l'analyse des résultats de la synthèse de l'estimateur de mouvement (chapitre V, §2.3). Pour obtenir un résultat équivalent de façon automatique, il faudrait extraire la

partie critique et en faire une autre entité. Avec *Behavioral Compiler* on peut spécifier des boucles pour y insérer un pipeline, mais si la partie critique n'appartient pas à une boucle il faut la sortir de la description et la synthétiser à part.

Ordonnancement multi-processus

D'autre part il serait intéressant de pouvoir appliquer un ordonnancement multi-processus afin de synchroniser les tâches entre les processus et de partager les ressources entre eux. Or ce type d'ordonnancement est très difficile à mettre au point. Il reste au stade de recherche. Aucun outil actuellement disponible sur le marché n'intègre ce genre de transformation. En conclusion il appartient au concepteur de synchroniser les processus entre eux par des protocoles de dialogue et éventuellement de casser les processus pour partager les ressources entre eux.

2.2. La gestion des mémoires

Les concepteurs d'outils de synthèse comportementale ne se sont pas suffisamment préoccupés de la gestion des mémoires malgré de nombreuses recherches théoriques [42]. Aujourd'hui on peut dire que la synthèse de haut niveau n'a pas apporté une très grande souplesse par rapport à l'utilisation des mémoires. La plupart des outils (dont *Behavioral Compiler, Amical, Cathedral-2/3*) incluent des mémoires à la place des tableaux. Mais les difficultés surgissent lorsqu'il faut utiliser une mémoire spécifique : respect du protocole d'accès, simulation au niveau comportemental avec le modèle de la mémoire qui sera utilisée, ou simplement avec un tableau.

En fait, la synthèse architecturale comprend une partie automatique pour la globalité du circuit, mais aussi une partie manuelle pour la gestion des mémoires avec *Amical*. En effet, lorsque le concepteur décrit le comportement du circuit, il revient aux méthodes classiques, à chaque interface avec une mémoire. Ceci a pour conséquence de limiter les performances de l'ordonnancement (une donnée disponible par cycle). Le concepteur doit au préalable avoir étudié le nombre de mémoires nécessaires et leur organisation avant la synthèse. La synthèse de haut niveau sera d'une très grande assistance lorsqu'elle proposera des organisations de mémoires optimales. On pourra ainsi constituer une bibliothèque de mémoires statistiquement les plus intéressantes.

Par ailleurs, dès qu'une mémoire ou tableau est partagée par plusieurs processus, la modélisation et la simulation deviennent délicates avec les outils existants, et particulièrement avec la version actuelle de *Behavioral Compiler*. L'inférence des mémoires telle qu'elle existe actuellement néglige les aspects de testabilité de la mémoire en rendant périlleuse l'insertion de BIST (*Boundary Scan Insertion Test*, ce composant sert à tester le contenu d'une mémoire). C'est pourquoi à ce jour il est préférable d'instancier simplement le composant mémoire avec son BIST dès le niveau comportemental à l'extérieur des architectures qui comprennent des processus comportementaux, aux dépends d'une simulation comportementale plus lente.

2.3. Style de la description RTL

Les outils actuels de synthèse logique sont sensibles au style d'écriture de la description d'entrée. La sortie des outils de synthèse de haut niveau n'est pas toujours présentée suivant un style facilitant l'exploitation des capacités de la synthèse logique. Les problèmes les plus fréquents, dus à l'automatisation de la génération de ce code, sont:

- une utilisation excessive de la hiérarchie, provenant des allocations d'éléments de bibliothèque, qui complique et allonge la synthèse logique,

- un découpage non nécessaire et peu efficace, entre parties contrôle et opératives dans *Amical*. En effet, le contrôleur issu de la synthèse décode une instruction à chaque cycle d'horloge. La largeur de l'instruction est supérieure ou égale au nombre de ressources matérielles allouées dans le chemin de données. Cette instruction intermédiaire qui connecte les deux parties alourdit l'ensemble de la description RTL et empêche des optimisations logiques si le circuit n'est pas mis à plat. De plus, dans le cas d'un pipeline du contrôle, il faut un registre pour toute l'instruction, alors que seuls certains fils sont concernés.
- des noms d'éléments (signaux, unités fonctionnelles, etc.) trop longs ou non significatifs qui peuvent rendre la correction des bogues très fastidieuse,
- l'absence de commentaire pouvant aider à corriger les bogues.

Les descriptions délivrées, par de nombreux outils de synthèse architecturale (*Amical*, *Cathedral-2/3*, etc.), sous forme de listes de composants interconnectés sont donc difficiles à exploiter aussi bien pour la validation que pour la synthèse logique.

En effet, les documents méthodologiques de synthèse logique [64, 65] conseillent généralement un découpage en blocs de 250 à 2000 portes. En dehors de cette fourchette les temps de synthèse s'allongent de façon disproportionnée. En pratique il existe deux façons de synthétiser l'architecture RTL résultante:

La première procède de façon hiérarchique. Elle consiste à synthétiser chaque ressource instanciée dans la description RTL, puis à les assembler. Les optimisations logiques sont locales aux instances sans traverser les frontières de celles-ci. Cette compilation est particulièrement inefficace pour une description issue de *Cathedral-2/3* car les instances sont au niveau du bit.

La seconde procède à la mise à plat du circuit avant synthèse logique. On dépasse alors fréquemment le seuil des 2000 portes et, comme l'optimisation s'étend à un espace logique très important, les temps de synthèse deviennent longs et les résultats en termes de performance et de surface ne sont pas forcément optimaux.

Notre évaluation des outils a mis en évidence leur points faibles concernant la validation des descriptions d'entrée et de sortie. Cependant, il ne faut pas perdre de vue que l'intérêt de la synthèse de haut niveau est l'augmentation de la productivité et de la flexibilité de la conception architecturale. L'élévation du niveau d'abstraction doit permettre une validation fonctionnelle plus rapide que dans le flot classique, par des descriptions plus concises. Il se trouve que les contraintes de style d'écriture à l'entrée des outils *Amical* et *Behavioral Compiler* font perdre le précieux gain de temps acquis par la simulation comportementale. Et pour finir, l'expérience prouve qu'il faut valider l'étape de conception architecturale. Mais les descriptions de sortie des outils générées automatiquement sont inutilisables par rapport au confort souhaité par le concepteur. Manifestement le style d'écriture des descriptions de sortie (à part celle d'*Amical*) ne prend absolument pas en compte ces aspects. On constate pour la description RTL sortie de *Cathedral-2/3*, un temps de simulation aussi important que pour une description au niveau portes. Le code de sortie est illisible et nécessite des commentaires pour faciliter la localisation des bogues et éventuellement permettre au concepteur le raffinement manuel de cette description s'il le souhaite.

2.4. Simplification de l'utilisation des outils

Les outils de synthèse architecturale évalués requièrent ou produisent un trop grand nombre de fichiers pendant la synthèse d'un circuit, sans omettre la gestion des bibliothèques d'unités fonctionnelles spécifiques à un outil qui alourdit le flot. Il serait plus simple de pouvoir construire une bibliothèque extensible de composants existants et décrits dans un langage de description de matériel. L'élaboration de cette bibliothèque ne nécessiterait pas de temps considérable. La bibliothèque ne dépendrait pas de l'outil de synthèse, mais de celui qui la construit. Actuellement dans *Amical* et *Behavioral Compiler*, l'élaboration des bibliothèques requièrent des fichiers d'abstraction dans des formats internes à l'outil, entraînant des redondances d'informations difficiles à valider.

3. Perspectives du flot de conception

Avant de proposer des perspectives du flot de conception, il est utile de rappeler brièvement le flot de conception actuel. Dans l'équipe "CMOS Digital Design" en Central R&D à ST, le flot de conception rôdé à ce jour est décrit par la figure I.1. Alors que les outils de synthèse architecturale n'ont pas encore fait de percée pour les parties matérielles, les compilateurs de code C en assembleur semblent bien intégrés dans ce flot pour les parties logicielles. En fait, le champ d’application de la synthèse de haut niveau est plus étroit que prévu. Pour des applications avec un contrôle complexe, les concepteurs optent pour une solution logicielle, plus flexible et moins risquée qu'une solution matérielle, lorsque la bande passante le permet.

Néanmoins, la synthèse de haut niveau reste nécessaire pour les raisons qui suivent.

Premièrement, on ne peut réaliser de partie logicielle qui réagisse à des signaux extérieurs en un ou deux cycles d'horloge. Or il est primordial de pouvoir décrire ce type de fonctionnement. On peut citer en exemple des contrôleurs de remplissage de mémoire qui testent souvent les valeurs de plusieurs signaux. Dans d'autres cas, on peut imaginer que la bande passante ne permette pas d'utiliser un contrôleur logiciel nécessitant en moyenne 5 à 6 fois plus de cycles d'exécution qu'un contrôleur matériel.

La figure VI.4 montre un extrait de code assembleur issu d'un compilateur ciblé du type de celui qui est décrit dans [66]. La première ligne, référencée 187 contient en commentaire l'instruction C d'origine. Les lignes référencées de 188 à 197 renferment le résultat de la compilation du code C en assembleur. On constate qu'il faut 10 cycles pour exécuter les opérations et 3 ou 6 cycles pour ne pas les exécuter. On aurait pu décrire cette instruction en VHDL pour une réalisation matérielle qui n'aurait nécessité qu'un seul cycle. Entre les solutions matérielles et logicielles il faut trouver un compromis entre flexibilité et rapidité d'exécution.

```
                                  187  ;  if ((a < b) && (c < d)) {a = e; c = b;}
0x00000000        002c0000        188       LDA RAM#V_a
0x00000001        003c0001        189       CMPA SIGNED#V_b
0x00000002        0004000a        190       BGE ROM#main_L1
0x00000003        002c0002        191       LDA RAM#V_c
0x00000004        003c0003        192       CMPA SIGNED#V_d
0x00000005        0004000a        193       BGE ROM#main_L1
0x00000006        002c0004        194       LDA RAM#V_e
0x00000007        00300000        195       STA RAM#V_a
0x00000008        002c0001        196       LDA RAM#V_b
0x00000009        00300002        197       STA RAM#V_c
```

Figure VI.2 : Exemple d'une instruction conditionnelle en C traduite en assembleur.

Deuxièmement, avec la complexité des futurs systèmes et l'avènement des outils de synthèse système, il sera nécessaire d'automatiser l'étape qui lie la sortie de ces outils à l'entrée des outils de synthèse logique.

Enfin troisièmement, la méthode classique partant du niveau transfert de registre souffre de peu de confort (flexibilité, lisibilité, rapidité de validation), par rapport à ce que la synthèse architecturale est susceptible d'offrir.

La figure VI.5 illustre un flot de conception système idéal. A partir des spécifications, un outil de synthèse système fournirait un partitionnement et la synthèse des communications entre des parties logicielles et des parties matérielles suivant la flexibilité et les performances requises. Cette tâche est appliquée manuellement en Central R&D à ST. La suite du flot reprend la figure I.1.

La synthèse comportementale a donc un champ d'application certain : la génération de l'architecture de toutes les parties matérielles dont les spécifications d'entrée seront issues des outils de synthèse système.

Cependant pour atteindre ce but, il faut un environnement unique pour les parties matérielles. Cet avis est partagé par Lin dans [42]. Les expériences menées pendant cette thèse montrent que les outils de synthèse comportementale sont efficaces pour des domaines d'applications spécifiques. Mais il est préférable de ne pas compliquer davantage le futur flot de conception et donc de disposer d'un environnement unique pour les parties matérielles.

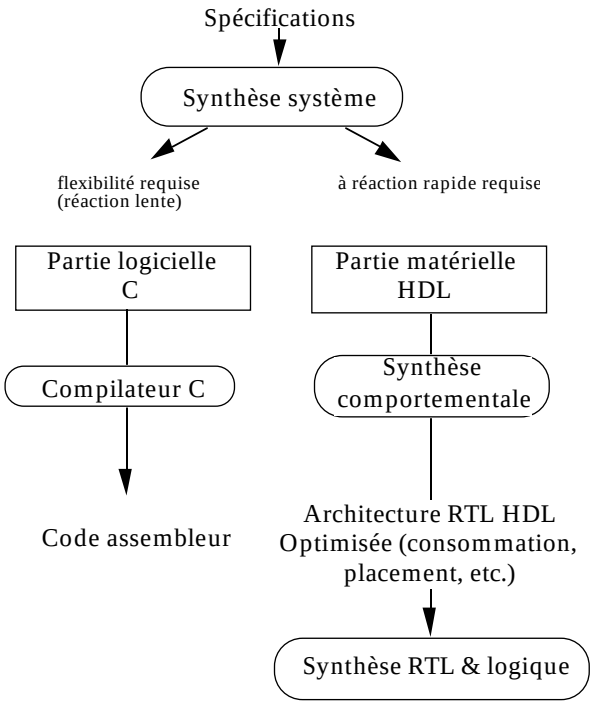


Figure VI.5 : Perspectives de flot de conception

4. Conclusion

Ce chapitre a fait le bilan sur les points forts de la synthèse comportementale, notamment l'ordonnancement et l'introduction automatique de pipeline. Les points faibles restent la gestion des mémoires, le style de la description de sortie des outils, le manque de simplicité dans l'emploi des outils de synthèse. La conclusion de cette étude est qu'il existe un champ d'application pour la synthèse de haut niveau, mais peut-être plus étroit que prévu, comme le constate Camposano dans [14]. A l'avenir de plus en plus les parties matérielles seront soumises à la synthèse de haut niveau, permettant de faire le pont plus tard entre les outils de synthèse système et les outils de synthèse logique. Il faut donc préserver un comportement au niveau transfert de registre à travers la synthèse de haut niveau lorsque c'est requis.

Conclusion Générale

Notre objectif a été d'évaluer les résultats de l'application de la synthèse architecturale en milieu industriel et d'étudier l'impact de cette nouvelle technologie sur la productivité. Nous avons analysé la réduction du temps de conception des circuits exclusivement matériels, ou ASICs. Comme les premiers outils de synthèse architecturale sont apparus peu avant le début de la thèse, il n'existait pas encore de méthode d'utilisation de ces outils en milieu industriel. Il fallait en définir une. Au terme de la thèse, des méthodes émergent.

Dans le cadre de cette thèse, nous avons montré qu'il est aujourd'hui possible d'utiliser certains outils de synthèse architecturale pour concevoir des ASICs en milieu industriel. Nous avons mis au point une méthode de conception s'appuyant sur l'utilisation mixte d'outils de synthèse comportementale complémentaires. Son principe se fonde sur le partitionnement fonctionnel du circuit entre parties dominées flot de contrôle ou flot de données en vue de l'utilisation de l'outil le plus approprié.

Le chapitre I a situé le cadre de cette thèse en faisant ressortir l'incapacité des méthodes et outils de conception actuels à profiter des progrès des technologies sur le silicium. On trouve dans ce chapitre une description du flot de conception classique de ces circuits.

Le chapitre II a décrit la synthèse comportementale et son insertion dans le flot de conception. Nous y expliquons les besoins industriels après avoir dressé l'état de l'art des outils. Peu d'outils couvrent ces besoins à ce jour.

Dans le chapitre III nous avons présenté, pour trois outils de synthèse architecturale de types différents, leur architecture cible, leurs étapes de synthèse, leurs optimisations et l'étape de validation. Le premier, *Amical*, est dédié aux applications orientées flot de contrôle. Le second, *Cathedral-2/3*, est dédié aux applications de traitement de signal à haut débit. Le troisième, *Behavioral Compiler*, disponible en fin de thèse, est à orientation mixte. En comparant les caractéristiques de ces trois outils, nous avons montré leur complémentarité sur certains aspects. Nous avons proposé dans ce chapitre un flot de conception combinant les outils *Amical* et *Cathedral-2/3*, ce qui a permis une plus grande couverture du domaine des ASICs.

Dans le chapitre IV nous avons présenté une application modulaire de la synthèse comportementale pour la conception avec *Amical* d'un système de recherche sur des fenêtres. Le système se décompose en quatre modules : un co-processeur, deux modules mémoires, et un contrôleur global. Le processus de conception commence avec la synthèse des trois premiers composants. Le contrôleur peut être conçu comme un simple composant interagissant avec les autres modules ou comme un processeur qui utilise les trois premiers modules comme des composants comportementaux. Pour chaque module, nous avons détaillé la fonction, l'architecture abstraite, la description VHDL et l'architecture produite par *Amical*. Le circuit étudié et la méthode utilisée pour le concevoir ne sont que des simplifications de l'expérience présentée dans le chapitre suivant.

Dans le chapitre V nous avons appliqué la méthode proposée à la fin du chapitre III à un cas industriel. Il s'agit de l'Estimateur de Mouvement du VideoCODEC développé à SGS-Thomson. Nous avons donné ici les résultats de la synthèse architecturale de ce circuit par *Amical* et *Cathedral-2/3*. Nous les avons comparés à ceux obtenus par la méthode manuelle classique. Grâce à notre technique, la longueur de la description d'entrée dans le flot de conception automatique est divisée par 5. Cette concision augmente la flexibilité, en particulier vis-à-vis de l'évolution des normes en transmission de l'image et des technologies. Le coût de l'automatisation en surface est négligeable et s'élève à 5%. La conception fut 2 à 3 fois plus rapide que la conception par la méthode classique.

Pour explorer les limites de la méthode proposée, nous l'avons appliquée au cas d'un circuit plus complexe. Dans le chapitre VI nous avons évalué la synthèse comportementale et nous avons analysé ses perspectives au vu des résultats des expériences menées dans cette thèse. Nous avons mis en évidence les points forts de cette automatisation et les points à améliorer. Enfin nous avons dressé les perspectives de la synthèse architecturale face à la conception logicielle.

Il reste encore un long travail de méthodologie pour donner les moyens aux outils de synthèse architecturale de poursuivre leur évolution vers une meilleure efficacité. Le gain en temps de conception peut encore augmenter. La validation reste le point critique. L'utilisation des outils doit également être simplifiée. La qualité du partitionnement est déterminante.

La conclusion de cette étude est qu'il existe un champ d'application certain pour la synthèse de haut niveau, mais plus étroit que prévu : les parties matérielles des circuits issus de la synthèse système. A l'avenir la synthèse architecturale fera le pont entre les outils de synthèse système et les outils de synthèse logique. Il faut donc préserver un comportement au niveau transfert de registre à travers la synthèse de haut niveau lorsque c'est requis. Nous espérons donc que notre travail aura cerné les vrais problèmes de l'industrie auxquels la recherche sur la synthèse architecturale devrait répondre.