N° d'ordre : 07/2007 – D/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'enseignement supérieur et de la recherche scientifique

Université des Sciences et de la technologie
Houari Boumediene

Faculté d'Electronique et d'Informatique

THESE

Présentée pour l'obtention du diplôme de DOCTORAT
En : Informatique
Spécialité : Informatique

Par : MAHDOUM Ali

Thème

Synthèse de haut niveau d'un compilateur de systèmes digitaux VLSI à intégrer sur la même puce

Soutenue le 19/07/2007, devant le Jury composé de :

Mr- Y. SMARA, Professeur, USTHB.	**Président**
Mr- N. BADACHE, Professeur, USTHB.	**Directeur de Thèse**
Mr- H. BESSALAH, Docteur, CDTA.	**Co-Directeur de Thèse**
Mr- M. BENMOHAMMED, Professeur, Univ. Constantine,	**Examinateur**
Mr- A. BERRACHEDI, Professeur, USTHB.	**Examinateur**
Mr- A. HENNI, Maître de Conférences, INI.	**Examinateur**

SOMMAIRE

LISTE DES FIGURES

LISTE DES TABLES

À ma famille,

à la mémoire du Cheikh Mohammed El Ghazali,
source de lumière et de sagesse.

REMERCIEMENTS

Je remercie le Professeur BADACHE et Docteur BESSALAH d'avoir accepté de diriger ce travail, pour leur disponibilité, ainsi que pour leur aide à concrétiser ce travail.

Je remercie également le Professeur SMARA de m'avoir honoré en présidant le Jury de cette thèse, ainsi que les Professeurs BENMOHAMMED et BERRACHEDI et Docteur HENNI d'avoir accepté de faire partie de ce Jury.

Enfin, mes remerciements vont également à tous ceux qui m'ont aidé de près ou de loin à la réalisation de ce travail.

R E S U M E

Les premiers compilateurs de silicium étaient principalement de deux types :
- ceux qui ont pour point d'entrée une description structurelle,
- ceux qui ont pour fichier d'entrée une description comportementale.

L'utilisation d'un compilateur du premier type permet d'obtenir un système VLSI dont l'architecture peut être quelconque. Néanmoins, la conception de cette architecture est à la charge de l'utilisateur dans la mesure où la compilation est conduite à partir d'une représentation structurelle.

Un compilateur du deuxième type permet de générer un système VLSI dont l'architecture est pré-spécifiée. Cependant, l'avantage de tels types de compilateurs est que les efforts de l'utilisateur sont concentrés non pas sur la conception d'architectures, mais plutôt sur la conception de systèmes.

Les travaux de recherche actuels dans ce domaine visent la conception de compilateurs ayant les avantages de ceux des deux types, c'est à dire conduire la compilation à partir d'une description comportementale tout en générant des circuits à caractéristiques différentes (variées). Ainsi, il est possible d'introduire des contraintes de surface, de vitesse et de consommation de puissance afin de guider le processus de synthèse. En outre, les technologies actuelles permettent d'intégrer tous les composants du système sur la même puce (*system on chip*), ce qui engendre des problèmes de recherche supplémentaires (la conception de systèmes monopuce requiert un savoir pluridisciplinaire allant du logiciel aux phénomènes physiques des technologies submicroniques : développement d'APIs, de systèmes d'exploitation en temps réel, de pilotes pour des périphériques aussi variés que nombreux, conception de circuits mixtes numérique/analogique, de circuits RF, résolution des phénomènes électro-thermiques, etc …). La prise en considération de tous ces problèmes nécessitant alors toute une compagnie, le projet a été cerné à l'aspect *Synthèse Hardware de systèmes digitaux à intégrer sur la même puce*. Toutefois, l'architecture à générer permet à plusieurs processeurs et mémoires de coexister avec la partie matérielle du système, ce qui permet de paralléliser même la partie Software et d'anticiper sur la possibilité d'intégrer une partie logicielle. Les grandes étapes de ce travail sont les suivantes :
- Génération d'une structure de données à partir d'une description en SYSTEMC pour la synthèse du compilateur
- Détermination du graphe des tâches
- Décomposition du système en sous-systèmes VLSI en tenant compte du compromis surface – vitesse – consommation de puissance
- Synthèse des tâches
- Définition du protocole de communications régissant les sous-systèmes
- Synthèse et analyse de chaque sous-système VLSI

Enfin, notons que la plupart des problèmes traités dans cette thèse ne sont pas polynomiaux : Ils sont $\sum_k^P$, et les plus simples d'entre-eux sont $\sum_1^P$ (=NP) complets.

Mots-clé : compilation de silicium, synthèse de haut niveau, systèmes monopuce, embarqués, temps réel, optimisations, contraintes de surface, vitesse, consommation de puissance, complexité algorithmique

INTRODUCTION GENERALE

La conception des tout premiers circuits intégrés se faisait selon une approche *bottom-up*, c'est-à-dire à partir de composants de base, jusqu'à arriver au circuit final. Le concepteur ne disposait alors que de très peu d'outils d'aide à la conception, rendant de ce fait la conception d'un circuit aussi petit soit il, fastidieuse et non exempte d'erreurs. Bien que l'aboutissement au produit final était possible grâce au savoir-faire et à l'expérience du concepteur, cette approche fût abandonnée au profit d'une autre. Cette dernière est appelée *gate arrays* et consiste en une conception physique d'un ensemble de fonctions où seule l'opération de métallisation n'est pas effectuée. Pour concevoir son circuit, il suffisait au concepteur de relier, par des interconnexions métalliques, les fonctions nécessaires à la conception de son circuit. Bien que cette approche lui permettait une conception plus rapide, il n'en demeure pas moins qu'une perte de surface en silicium (dont le coût est assez élevé) est engendrée par les fonctions présentes mais non utilisées par le circuit. Au cours du temps, alors qu'on concevait des circuits, des bibliothèques de cellules furent de plus en plus mises à jour, et mises à la disposition du concepteur. Ainsi celui-ci pouvait se servir de ces cellules pour concevoir un circuit plus complexe. Ceci lui permettait à la fois d'accélérer la conception (comme avec les *gate arrays*), tout en évitant une perte de surface de silicium inutile (inconvénient des *gate arrays*).

Une autre méthodologie de conception plus intéressante vint par la suite remplacer la méthodologie *bottom-up*. C'est une méthodologie où les bibliothèques de cellules sont toujours utilisées, mais est une méthodologie *top-down*, c'est-à-dire que la conception d'un circuit démarre à partir de sa représentation la plus abstraite possible, et progresse de détails en détails jusqu'au produit final. L'intérêt d'une telle méthodologie est qu'une erreur est d'autant moins coûteuse (en temps et en argent) qu'elle est commise à un niveau d'abstraction haut. Au fil du temps, de nombreux outils de CAO furent développés pour libérer le concepteur aussi bien de tâches fastidieuses, que pour l'aider à vérifier, analyser et obtenir des circuits plus optimaux, lesquelles opérations sont impossibles à faire manuellement pour des circuits devenant de plus en plus complexes avec le développement technologique.

Tout comme la compilation de langages et l'édition de liens qui permettent de démarrer d'un langage de haut niveau jusqu'au langage machine, on pensa alors à en faire de même pour les circuits : faire de *la compilation de silicium*, c'est-à-dire démarrer d'une description algorithmique du système VLSI (*Very Large Scale Integration*) à concevoir, puis par étapes successives, arriver à la conception *physique* du système, qui, avec le test, représentent la dernière étape avant la fabrication du système.

Les premiers compilateurs de silicium étaient principalement de deux types :

- ceux qui ont pour point d'entrée une description structurelle,
- ceux qui ont pour fichier d'entrée une description comportementale.

L'utilisation d'un compilateur du premier type permet d'obtenir un système VLSI dont l'architecture peut être quelconque. Néanmoins, la conception de cette architecture est à la charge de l'utilisateur dans la mesure où la compilation est conduite à partir d'une représentation structurelle.

Un compilateur du deuxième type permet de générer un système VLSI dont l'architecture est pré-spécifiée. Cependant, l'avantage de tels types de compilateurs est que les efforts de l'utilisateur sont concentrés non pas sur la conception d'architectures, mais plutôt sur la conception de systèmes.

Les travaux de recherche actuels dans ce domaine visent la conception de compilateurs ayant les avantages de ceux des deux types, c'est à dire conduire la compilation à partir d'une description comportementale tout en générant des circuits à caractéristiques différentes (variées). Ainsi, il est possible d'introduire des contraintes de surface, de vitesse et de consommation de puissance afin de guider le processus de synthèse.

Avec le développement considérable de la technologie, il est actuellement possible d'intégrer tous les composants du système sur la même puce (*system on chip*), ce qui engendre des problèmes de recherche supplémentaires (la conception de systèmes monopuce requiert un savoir pluridisciplinaire allant du logiciel aux phénomènes physiques des technologies submicroniques : développement d'APIs, de systèmes d'exploitation en temps réel, de pilotes pour des périphériques aussi variés que nombreux, conception de circuits mixtes numérique/analogique, de circuits RF, résolution des phénomènes électrothermiques, etc …). La prise en considération de tous ces problèmes nécessitant alors toute une compagnie, le projet a été cerné à l'aspect *Synthèse Hardware de systèmes digitaux à intégrer sur la même* puce. Toutefois, l'architecture à générer permet à plusieurs processeurs et mémoires de coexister, ce qui permet de paralléliser même la partie Software. Les grandes étapes de ce projet sont les suivantes :

- Génération d'une structure de données à partir d'une description en SYSTEMC pour la synthèse du compilateur
- Détermination du graphe des tâches
- Décomposition du système en sous-systèmes VLSI en tenant compte du compromis surface – vitesse – consommation de puissance
- Synthèse des tâches
- Définition du protocole de communications régissant les sous-systèmes
- Synthèse et analyse de chaque sous-système VLSI

Enfin, notons que la plupart des problèmes traités dans cette thèse ne sont pas polynomiaux : Ils sont $\sum_k^P$, et les plus simples d'entre-eux sont $\sum_1^P$ (=NP) complets.

La présente thèse est organisée comme suit. Au premier chapitre, nous donnerons quelques définitions et rappels pour mieux situer notre travail dans le contexte de la compilation de silicium. Suivra alors le deuxième chapitre où nous présenterons notre flot de conception. Le troisième chapitre portera sur la synthèse, l'analyse et la vérification d'un système digital VLSI en vue d'une implémentation *monopuce*. Ces tâches sont conduites à partir d'analyses syntaxique et lexicale dune description en SYSTEMC, ainsi que d'une détermination de graphe de tâches, présentées dans le même chapitre. Suivront alors la présentation de notre synthèse des différentes tâches et celle de notre protocole de communications. La synthèse, l'analyse et la vérification des constituants du système seront abordées au quatrième chapitre. Les résultats obtenus seront présentés et discutés au dernier chapitre. Enfin, nous terminerons par une conclusion générale suivie par une bibliographie.

Chapitre 1

Définitions et rappels

1.1. Introduction :

Avant de présenter notre travail dans le cadre de cette thèse, nous aimerions tout d'abord donner quelques définitions et rappels ayant trait à la compilation de silicium. Ceci permettrait de mieux situer notre travail dans le contexte. Nous aborderons alors les différents niveaux d'abstraction et discuterons des représentations comportementale, structurelle et physique d'une entité à un niveau d'abstraction donné.

1.2. Compilation de silicium :

La conception de systèmes VLSI dans le cadre de la compilation de silicium est conduite à partir du niveau d'abstraction *système*. Puis, à partir de tâches de synthèse, d'analyse et de vérification exécutées à chaque niveau d'abstraction, on aboutit jusqu'aux détails d'un transistor au niveau physique (dessin de masques). Cette transition passe par les niveaux *transfert de registres*, *module* et *cellule*, et ce, en considérant à chaque niveau de conception les représentations comportementale, structurelle et physique de l'entité au niveau d'abstraction donné. La Figure 1.1 donne une vue générale des étapes de conception d'un système VLSI dans le cadre de la compilation de silicium.

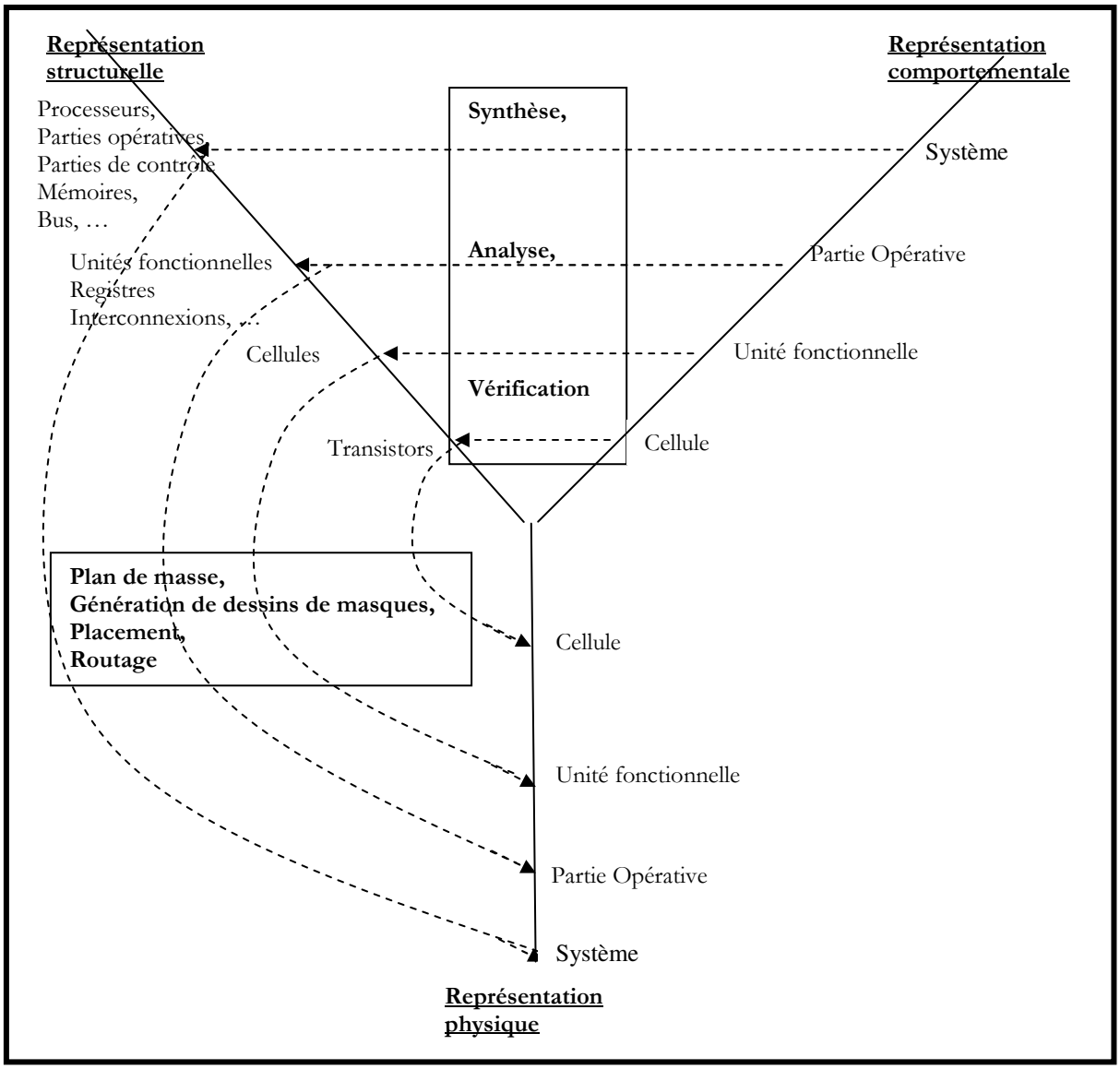

Fig. 1.1. Schéma synoptique d'un compilateur de silicium

Définition1 :

La synthèse est un outil ou un ensemble d'outils de CAO permettant de transformer, à un niveau de conception donné, la description comportementale d'une entité en une description structurelle, ou sa description structurelle en description physique. Parfois, la synthèse transforme la représentation de l'entité en une représentation de même type, mais plus raffinée, optimisée, ... De manière générale, la synthèse est conduite en tenant compte de paramètres d'optimisation, de contraintes, de spécifications, ...

Définition2 :

L'analyse consiste à déterminer, ou estimer les caractéristiques d'une entité qui concernent notamment les aspects de surface, de vitesse et de consommation de puissance. Cette analyse est une aide à la décision au concepteur pour passer à un autre niveau de conception plus bas, ou pour reconsidérer la conception de l'entité au niveau d'abstraction courant. Ceci lui éviterait une perte en temps et en argent de se concentrer sur les détails du produit final pour s'apercevoir qu'il ne répond pas à un cahier de charges, car le produit est issu d'une architecture mal étudiée, ...

Définition3 :

La vérification porte sur la fonctionnalité de l'entité à un niveau de conception donné. Elle peut être basée soit sur des tests, des simulations, ..., soit formelle. La vérification formelle est évidemment plus rigoureuse que les tests car ne pouvant être exhaustifs, mais à part certaines entités simples, elle est encore au stade de la recherche. Les outils de vérification industriels quant à eux s'articulent sur des jeux de test. Notons que l'analyse peut faire parfois office de vérification. Une analyse temporelle par exemple, permet d'analyser si les signaux sont générés conformément aux spécifications temporelles fixées, et en même temps de vérifier si les signaux générés sont conformes au comportement de l'entité considérée.

1.2.1. Niveau d'abstraction système :

1.2.1.1. Représentation comportementale

Le système VLSI à concevoir est décrit dans un langage donné tel que VHDL, VERILOG, C#, SYSTEMC,.... La fonctionnalité du système étant vérifiée, sa synthèse permet de transformer sa représentation comportementale en une représentation structurelle.

1.2.1.2. Représentation structurelle

La description algorithmique du système est transformée en un ensemble de composants tels que des ASICs (*Application Specific Integrated Circuits –Circuits Intégrés à Applications Spécifiques-*), des processeurs, des mémoires, des bus, ..., selon la complexité du système et le type de synthèse qui lui a été appliquée. De manière générale, sont représentés à ce niveau tous les composants (sous-systèmes) du système, communiquant via un bus global et éventuellement une mémoire globale. Chaque composant (sous-système) est un ensemble d'IPs, de processeurs, de mémoires locales, d'unités fonctionnelles et de registres communiquant via des interconnexions locales. De cette décomposition et de la synthèse des tâches affectées à chaque sous-système découleront les caractéristiques du système VLSI à concevoir. Cette synthèse de tâches est faite de manière telle que les tâches qui s'exécutent se terminent dans les délais (cas d'une application fonctionnant en temps réel) tout en tenant compte de l'aspect *consommation d'énergie* (exemple, [27]).

Une analyse et une vérification de l'entité structurelle obtenue du point de vue surface, vitesse et consommation de puissance permettront de valider ou pas cette structure. Dans le cas où le concepteur n'est pas satisfait par la structure obtenue, la synthèse du système est refaite, en considérant d'autres aspects introduits par le concepteur sous forme de contraintes, de spécifications supplémentaires, etc .... Une fois la structure validée, la synthèse, l'analyse et la vérification de chaque composant de la structure sont abordées à des niveaux d'abstraction plus bas. Notons toutefois que si les composants de la structure existaient (en bibliothèque) sous

forme physique, des outils de définition de plans de masse, de placement et de routage permettraient d'aboutir au stade final de la conception du système, c'est-à-dire à sa représentation physique.

1.2.1.3. Représentation physique

Comme il a été dit précédemment, si tous les composants du système existaient sous forme de dessins de masques, des outils de définition de plans de masse, de placement et de routage permettraient de générer la représentation physique du système, moyennant des opérations supplémentaires (synthèse guidée par des outils d'analyse et de vérification) portant sur la diminution des paramètres parasites, les problèmes liés aux horloges et aux interconnexions de manière générale, l'amplification des signaux, etc .... Dans le cas contraire, les composants n'existant pas sous forme physique ou structurelle sont à leur tour synthétisés, analysés et vérifiés à des niveaux d'abstraction plus bas.

1.2.2. Niveau d'abstraction transfert de registres :

1.2.2.1. Représentation comportementale

L'un des composants du système peut être implémenté par une partie opérative et une partie de contrôle. Un type de synthèse appliquée à la partie de contrôle (représentée par une machine à états finis) permettrait d'aboutir ultérieurement à une partie de contrôle ayant une surface intéressante et de bonnes caractéristiques électriques (exemple, [5]). La synthèse portant sur la partie opérative consiste en un ordonnancement d'opérations et d'allocation de ressources physiques. Un ordonnancement minimisant le nombre de cycles vise à produire une entité performante en matière de vitesse (exemple, [7]). L'allocation consiste à allouer respectivement un nombre minimal de registres et d'interconnexions aux différentes variables et transferts de données sans qu'il y'ait de conflit entre ces variables et ces transferts. Dans les circuits actuels, la consommation de puissance est devenue un paramètre très important dont il faut tenir compte. Aussi, l'allocation et l'ordonnancement peuvent être conduits en tenant compte de cette contrainte (exemple, [8]).

Dans une conception conjointe Hardware / Software, des processeurs et des mémoires peuvent faire partie du système pour exécuter la partie logicielle en intéraction avec la partie matérielle. Le code de l'application peut être aussi réarrangé de manière à pouvoir l'exécuter plus rapidement. La diminution des temps d'accès aux mémoires et la consommation d'énergie de ces dernières représente aussi un cas de synthèse pour aboutir à un système global ayant les caractéristiques désirées.

L'analyse de chaque composant (IP, processeur, mémoire, bus,....) porte sur sa surface (estimée en termes de registres, d'unités fonctionnelles ou *IPs –Intellectual Property-*, d'interconnexions, ...), sa vitesse (nombre de cycles d'exécution et fréquence de fonctionnement) et sa consommation de puissance ou d'énergie (exemple, [10], [11], [12]). Dans le cas où l'un des paramètres ne satisfait pas une contrainte, la synthèse du composant est reconsidérée.

1.2.2.2. Représentation structurelle

Comme nous l'avions dit précédemment, la synthèse d'une partie opérative conduit à un ensemble de registres, d'unités fonctionnelles ou *IPs*, des interconnexions sur lesquelles transitent des données entre les différents constituants de la partie opérative. Le fonctionnement de cette partie est régi par la partie de contrôle associée grâce à des signaux contrôlant le transfert des données.

La partie de contrôle est constituée de signaux de contrôle émanant de la partie opérative, de bascules pour mémoriser l'état de la machine, et des signaux de commande de la partie opérative.

Enfin, notons que la représentation structurelle dépend de l'entité considérée (processeur, mémoire, bus, ...) au niveau d'abstraction considéré, et que l'analyse et la vérification de la

structure obtenue détermineront s'il faut passer à sa génération physique (les composants de la structure existent sous forme physique), à un niveau d'abstraction plus bas (où chacun des constituants de la structure sera synthétisé, analysé et vérifié), ou alors reconsidérer la synthèse de l'entité au niveau d'abstraction courant, c'est-à-dire générer une autre configuration ayant d'autres caractéristiques.

1.2.2.3. Représentation physique

De la même manière qu'au niveau *système*, une synthèse physique peut être conduite pour transformer la représentation structurelle de l'entité considérée (bus, partie opérative, partie de contrôle, ...) en sa représentation physique. Cette synthèse (confrontée ultérieurement par une analyse et une vérification) dépendra aussi du composant considéré et consistera à générer sa structure physique de manière entièrement automatique (exemple, [6]), ou par placement et routage de ses constituants physiques. Notons que les structures physiques obtenues pour un niveau d'abstraction pourront servir ultérieurement comme briques de base pour la génération de la structure physique d'un autre système. Ainsi, il existe actuellement des bibliothèques de co-processeurs !! Dans le cas où la génération de l'entité physique n'est pas directement possible, chacun de ses constituants est synthétisé, analysé et vérifié au niveau d'abstraction directement inférieur.

1.2.3. Niveau d'abstraction module :

1.2.3.1. Représentation comportementale

Nous avions vu qu'au niveau *transfert de registres*, la synthèse permettait de générer, outre d'autres composants, des unités fonctionnelles. Chaque unité fonctionnelle est à son tour abordée au niveau d'abstraction *module*. Cette unité fonctionnelle nécessitera une synthèse adéquate permettant de générer une structure répondant à certains critères en matière de surface, vitesse et / ou consommation de puissance (exemple, [9]).

1.2.3.2. Représentation structurelle

La synthèse opérée sur le composant concerné constitue une *loupe* qui permet de voir les constituants de ce composant. Dans le cas où celui-ci répond négativement aux tâches d'analyse et de vérification, sa synthèse est reconduite. Autrement, soit sa représentation physique est directement entamée, soit ses détails sont abordés au niveau d'abstraction directement inférieur.

1.2.3.3. Représentation physique

Celle-ci peut être soit générée directement de manière automatique, soit en optant pour des opérations de plan de masse, de placement et de routage de ses constituants physiques disponibles. Evidemment, la synthèse physique de ce composant sera suivie par des tâches d'analyse et de vérification de l'entité physique obtenue. Une troisième possibilité serait d'aboutir à la représentation physique ultérieurement, c'est-à-dire une fois que les constituants de l'entité considérée seront synthétisés, analysés et vérifiés.

1.2.4. Niveau d'abstraction cellule :

1.2.4.1. Représentation comportementale

Un exemple simple en est l'expression Booléenne de fonctions logiques. Une forme de synthèse consisterait à optimiser les fonctions logiques de manière à utiliser ultérieurement moins de transistors.

1.2.4.2. Représentation structurelle

Elle consiste en un ensemble de transistors interconnectés et réalisant de manière électrique le comportement de la cellule. Chaque transistor est représenté par son type, ses dimensions. Avec les paramètres de la technologie de sa fabrication, l'ensemble des informations

(groupées sous forme de *netlist*) permettrait d'analyser les performances électriques de la cellule. Un autre type de synthèse de l'entité structurelle de la cellule consisterait de déterminer de manière automatique les dimensions des transistors permettant à la cellule d'avoir certaines caractéristiques. Une analyse et une vérification électriques permettront de décider s'il y'a lieu de passer à la représentation physique de la cellule, ou de la resynthétiser.

1.2.4.3. Représentation physique

Celle-ci peut être obtenue directement à partir d'une bibliothèque de cellules (choix de la cellule ayant les caractéristiques désirées), ou générée automatiquement. Dans ce cas, la synthèse est conduite sous les contraintes de surface, vitesse et / ou consommation de puissance. Elle peut aussi être conduite en obligeant les ports d'entrée et de sortie à être placés à des endroits bien définis, de manière à faciliter l'agencement (placement et routage) de la cellule dans son environnement. De même, une analyse et une vérification sont opérées sur la structure physique de la cellule, et dans lesquelles, les facteurs liés aux interconnexions et aux paramètres parasites (inconnus dans sa représentation structurelle) sont pris en compte.

1.3 Conclusion :

Dans ce chapitre, nous avons fait des rappels et introduit les définitions de certains termes qui seront rencontrés le long de la lecture de cette thèse. Notons aussi que beaucoup de travaux de recherche ont été faits à différents niveaux d'abstraction, mais nécessitent néanmoins des mises à jour dues aux changements rapides des technologies. Ainsi, un simulateur ou un outil de CAO de manière générale qui ne tenait pas compte de certains paramètres -car alors négligeables- n'est plus valide avec les nouvelles technologies où ces paramètres sont devenus importants.

Concernant les niveaux d'abstraction *transfert de registres* et *système*, appelés de manière indistincte *haut niveau de conception* (*high level design*), en plus de la mise à jour des outils de CAO existants, ils nécessitent aujourd'hui le développement d'autres outils car les systèmes sont devenus plus complexes (par exemple, un processeur peut être juste une brique de base dans un système actuel). Ceci est aussi dû à de nouvelles préoccupations telles que celles engendrées par une conception conjointe logicielle et matérielle, l'intégration de systèmes hybrides dans la même puce, la complexité des protocoles de communications, les problèmes de couplage et de phénomènes électrothermiques dans les systèmes monopuce, ... Ceux-ci sont autant de problèmes qui nécessitent des outils de CAO adéquats dont les plus simples sont $\sum_1^P$ complets, c'est-à-dire NP complets (exemple, [17], [18], [19]). Enfin, ces outils doivent être à même capables de pouvoir traiter en des temps CPU raisonnables, et de manière efficace, des problèmes dont l'espace des solutions est devenu très large. Cette problématique nécessite de ce fait le développement d'heuristiques adéquates, les méthodes exactes étant souvent de complexité exponentielle. De plus, une méthodologie rigoureuse est requise pour espérer concevoir un système répondant aux caractéristiques voulues (si c'est possible), et de détecter et de corriger les problèmes dès les premiers niveaux d'abstraction, ce qui permet de réduire les coûts et d'augmenter les chances d'obtenir un produit final fonctionnel.

Chapitre II

Méthodologie adoptée

2.1. Introduction :

Le développement considérable de la technologie permet aujourd'hui d'intégrer plusieurs circuits sur la même puce, donnant ainsi naissance aux systèmes *monopuce*. Toutefois, cette fantastique intégration n'est pas sans problèmes. En effet, beaucoup de problèmes sont engendrés par la présence, sur la même puce, de beaucoup de circuits, de circuits hétérogènes, impliquant principalement des protocoles de communications complexes, des phénomènes électrothermiques, de couplage, ainsi que des problèmes de fiabilité induits par une forte dissipation de puissance. Des solutions doivent être donc apportées à ces problèmes aussi bien du point de vue *conception*, que du point de vue *outils d'aide à la conception* de tels systèmes.

La complexité des systèmes actuels est telle qu'elle nécessite une approche rigoureuse et méthodique. Aussi, la conception d'un système implémenté par une partie logicielle et une partie matérielle est conduite selon les étapes indiquées dans la Figure 2.1.

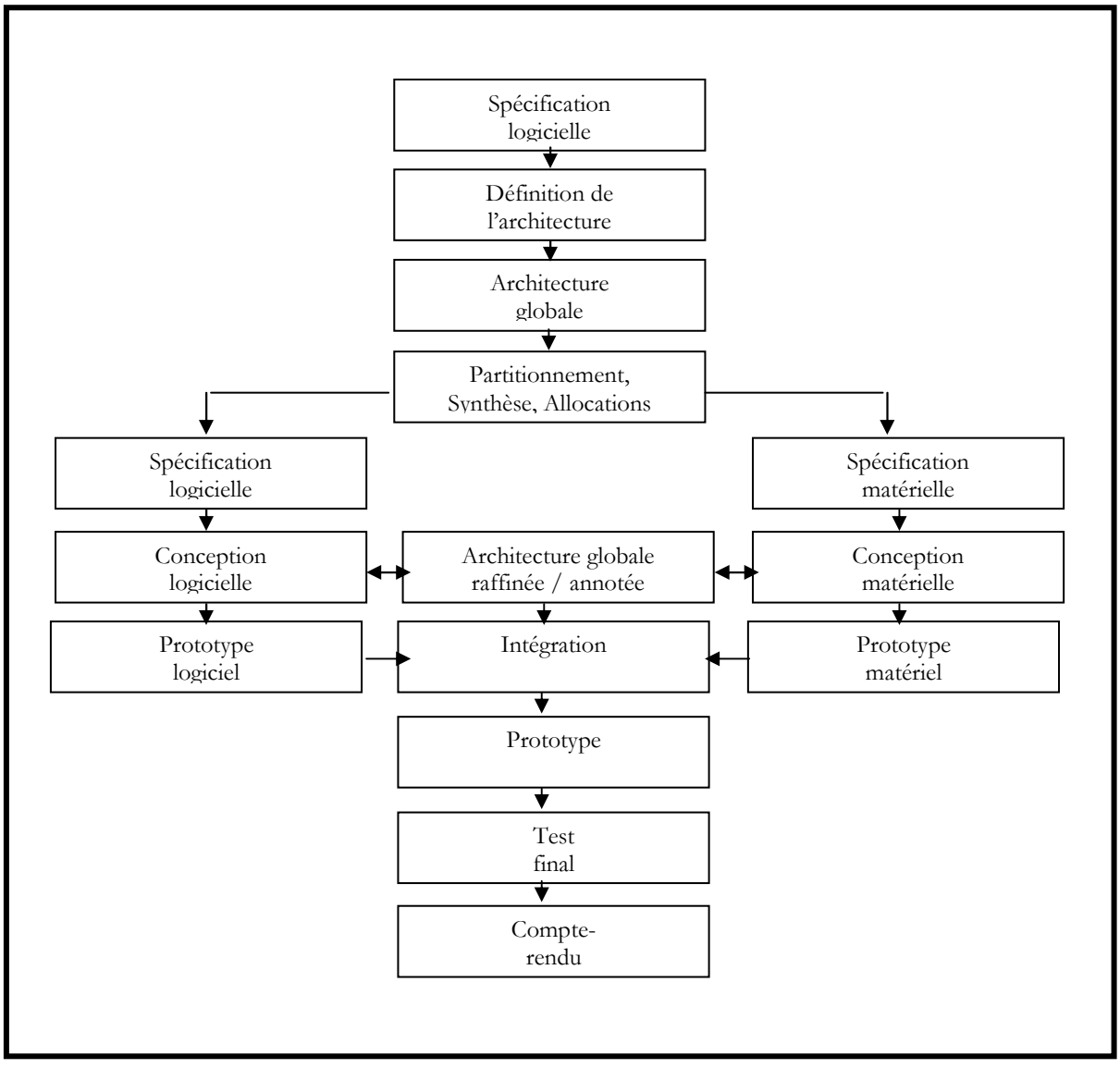

Fig.2.1. Méthodologie de conception conjointe logicielle et matérielle d'un système

Notons que l'étape *conception matérielle* est détaillée dans la Figure 1.1, dans le chapitre précédent. C'est cette étape qui fait l'objet de notre travail, et à laquelle nous allons donner d'autres détails dans le présent chapitre ainsi que dans les chapitres qui suivront. Nous allons alors présenter le long de ce chapitre notre propre méthodologie adoptée pour la synthèse,

l'analyse et la vérification en vue de la conception de la partie matérielle du système VLSI et de ses constituants.

2.2. Flot de conception :

Nous avons vu au premier chapitre que le système à concevoir était décrit dans un premier temps dans un langage donné (le langage SYSTEMC dont le choix est justifié dans le chapitre suivant). Aussi, des analyses lexicale et syntaxique sont opérées sur cette description. Nous en déduisons par la suite les tâches qui sont décrites dans le langage par des procédures et des fonctions.

Les tâches du système étant identifiées, nous construisons un graphe de tâches dont les nœuds sont représentés par ces tâches, et les arêtes définies d'une manière qui sera présentée au chapitre suivant.

Une synthèse est alors conduite à partir de ce graphe de tâches, et consiste à décomposer le système initial en sous-systèmes VLSI, puis à synthétiser chaque tâche. Une analyse et une vérification permettront alors de retenir cette décomposition ou d'opter pour une autre.

Enfin, la définition d'un protocole de communications permet d'aboutir à un ensemble de sous-systèmes interconnectés.

Des tâches de synthèse, d'analyse et de vérification sont aussi effectuées sur chacun des sous-systèmes, ce qui permet d'avoir des détails plus fins sur les caractéristiques de la configuration obtenue. La décision de retenir cette configuration ou pas est déterminée par des tâches d'analyse et de vérification.

Ainsi, notre méthodologie tient compte de la possibilité de générer plusieurs configurations (à caractéristiques différentes) du système à concevoir, ce qui donne au concepteur un éventail de choix lui permettant d'opter pour la configuration qui répond le mieux à son application. Le schéma synoptique de notre flot de conception est indiqué dans la figure 2.2.

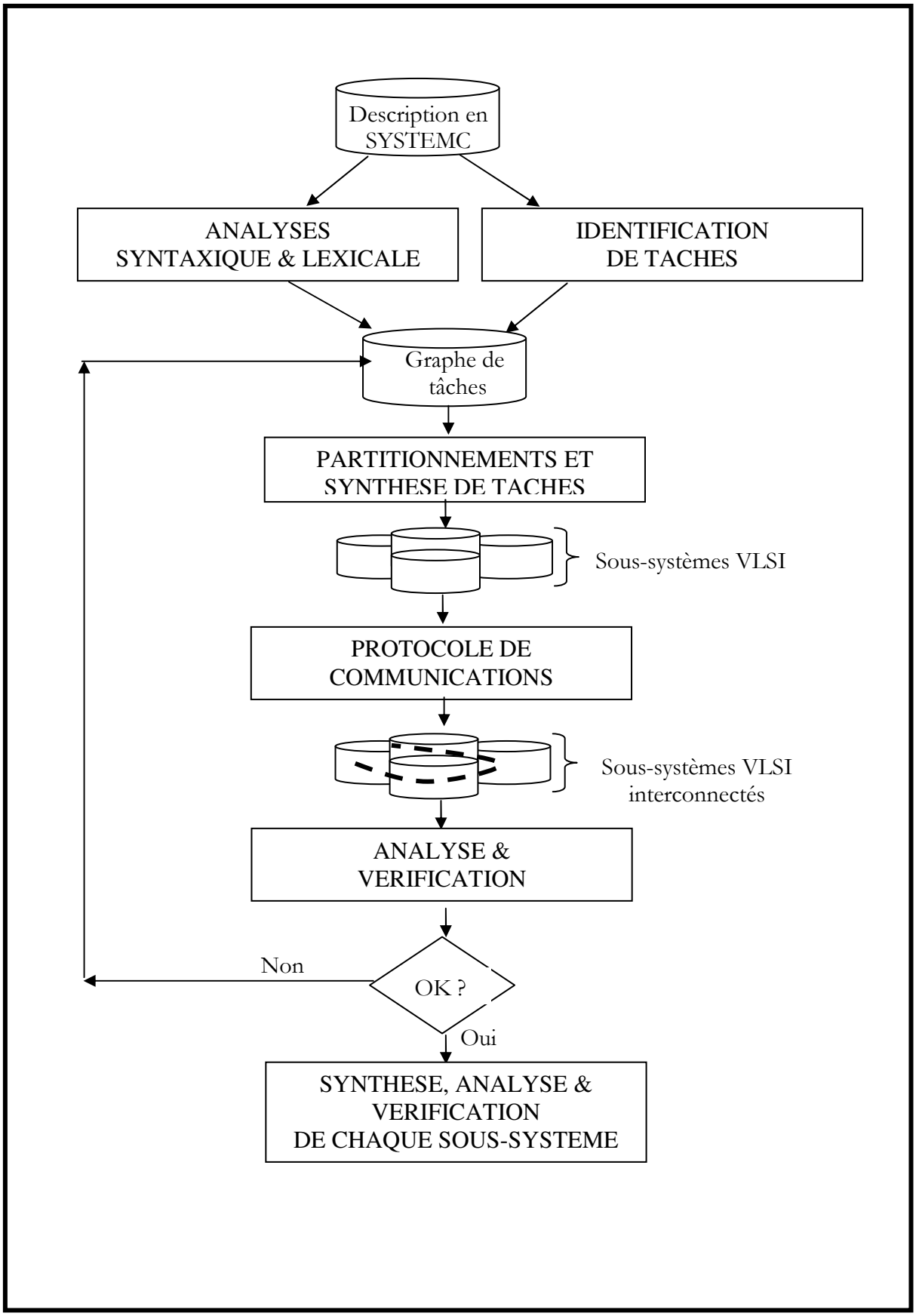

Fig.2.2. Schéma synoptique de notre flot de conception à de hauts niveaux d'abstraction

2.3. Algorithme général :

Avant d'entamer d'autres chapitres, nous aimerions donner l'algorithme général décrivant les différentes étapes de notre flot de conception. Ces différentes étapes seront évidemment détaillées au fur et à mesure que nous progresserons dans la lecture de cette thèse. Il est donné à ce niveau afin de donner au lecteur une vision globale de notre méthodologie, et sert comme introduction à la présentation des prochains chapitres.

```
Début
        Analyses syntaxique et lexicale d'une description en SYSTEMC() ;
        Déterminer G=(V,E) à partir de la description() ;
                // V={Tâches Ti} ; (Ti,Tj) ∈ E ssi Ti et Tj ne communiquent pas, ou ont
                // un certain degré faible de communications
        fin=0 ;
        tant que ( !fin)
        faire {Colorer_G() ;
              Mettre dans la même partition les tâches dont les nœuds correspondants dans G ont reçu la même
                couleur() ;
              Pour chaque sous-système, affecter les tâches aux différents composants (Processeurs, ASICs) du
                sous-système considéré().
               Synthétiser les différentes tâches() ;
               Définir Protocole_ des_Communications() ;
                Analyse() et Vérification() ;
                si (OK) Alors fin=1 ;
                sinon {Déterminer G'=( V, E' ) à partir de G() ;
                       G=G' ;
                       }
               fsi
              }
        ffaire
        // A ce niveau, il y'a obtention de sous-systèmes VLSI  interconnectés
        fin=0 ;
        tant que ( !fin)
        faire {pour chaque sous-système VLSI
              faire {  Ordonnancement & Allocation() ;
                       Déterminer_automate() ;
                       Génération_signaux_Ctrl_et_Cmd() ;
                       Synthèse_Partie_Ctrl() ;
                       Définir_Protocole_Communications_du_Sous_Système() ;
                    }
              ffaire
              Définir_Protocole_Communications()_des_Sous_Systèmes() ;
              Analyse() et Vérification() ;
              Si (OK) Alors fin=1 ;
              Sinon Ajout_et/ou_Suppression_de_contraintes() ;
              fsi
              }
        ffaire
Fin
```

Fig.2.3. Algorithme général de notre méthode

2.4. Conclusion :

Nous avons présenté dans ce chapitre notre méthodologie et notre algorithme général de synthèse de haut niveau des systèmes digitaux VLSI dans le cadre de la compilation des systèmes monopuce. Après les analyses syntaxique et lexicale d'une description en SYSTEMC et la génération du graphe des tâches, l'algorithme se compose de deux grandes parties, chacune d'elles faisant l'objet de présentation dans l'un des deux prochains chapitres.

Chapitre III

Synthèse, analyse et vérification de haut niveau de systèmes vlsi monopuce

3.1. Introduction:

Le développement considérable des technologies de fabrication de circuits ainsi que la complexité croissante des systèmes à concevoir font que les tâches de synthèse, d'analyse et de vérification doivent être conduites dès le niveau d'abstraction *système*, et ce, pour détecter et corriger, dès ce niveau, toute erreur. Ceci a pour conséquence de réduire les coûts de conception et d'augmenter les chances de fonctionnement du produit final.

Il s'agit aussi de produire, dès ce niveau, une architecture répondant aux caractéristiques voulues avant même de passer aux détails de conception de cette architecture. Le système est alors conduit à partir d'une description algorithmique, ce qui permet au concepteur une modification moins coûteuse en cas de non fonctionnalité du système. Plusieurs langages peuvent être utilisés à cet effet dont les plus connus sont VHDL (VHDL-AMS actuellement), VERILOG, C#, SYSTEMC, ... et le choix du langage est dicté par des critères qui seront abordés après cette introduction. Enfin, le protocole de communications doit être défini dès ce niveau, et ce, afin d'éviter de passer aux détails d'un protocole de communications qui ne réponde pas aux caractéristiques voulues. Ce sont ces points qui seront développés dans la suite de ce chapitre et qui seront synthétisés par une conclusion à la fin.

3.2. Langage SYSTEMC :

Pour décrire le comportement d'une entité donnée, des langages tels que VHDL et VERILOG sont utilisés. Le langage VHDL étant dédié aux circuits digitaux seulement, il a été généralisé pour pouvoir décrire des circuits mixtes analogiques / numériques. Il a ainsi donné naissance au langage VHDL-AMS qui est utilisé pour ce type de circuits présents actuellement dans plusieurs applications. VERILOG est aussi abondamment utilisé par les concepteurs des circuits, et le choix d'un langage par un autre est dicté soit par l'expérience acquise dans l'utilisation du langage (le concepteur préfère exploiter son expérience acquise dans l'utilisation d'un langage pour mieux représenter son circuit), soit par le flot de conception utilisé dans le laboratoire du concepteur et qui s'articule sur un langage bien défini. Dans certains cas, plusieurs langages peuvent être utilisés pour la conception du même circuit, selon le niveau de conception considéré. Il arrive aussi que pour le même niveau de conception, différents langages sont considérés car les *IPs* constituant le circuit en cours de conception sont décrits dans différents langages.

Cette utilisation simultanée de langages n'est pas évidemment sans problèmes :

- le concepteur doit avoir une maîtrise parfaite de *tous* les langages utilisés dans la conception d'un circuit donné
- des problèmes de portabilité se posent et qu'il faut résoudre par des interfaces

Notons aussi que bien que les langages VHDL-AMS et VERILOG aient montré leur efficacité dans les descriptions comportementales, ils présentent néanmoins certaines limitations :

- ils ont été utilisés pour des circuits qui n'étaient pas aussi complexes que ceux actuels, et dont la conception démarrait généralement à partir du niveau d'abstraction *transfert de registres*
- la complexité des systèmes actuels nécessite leur conception dans un environnement *orienté-objet*, caractéristique importante que n'offrent pas tous les langages
- la présence simultanée de plusieurs circuits, souvent hétérogènes, complique le protocole des communications qu'il faut étudier avant même de passer aux détails de sa conception, et ce, afin d'éviter des coûts de conception supplémentaires et inutiles

Afin de pallier ces problèmes, les compagnies *CADENCE* et *SYNOPSYS* qui sont des compagnies leader dans la conception des outils de CAO ont mis au point le langage *SYSTEMC*. Celui-ci est nettement plus efficace au niveau d'abstraction *système* du fait:

- qu'il est orienté-objet et se prête donc bien pour la description de systèmes complexes et hétérogènes
- qu'il offre une bibliothèque de protocoles de communications au profit du concepteur qui peut toujours avoir la possibilité de définir son propre protocole

L'autre but visé par ce langage est de faire de lui un langage standard dans le futur, ce qui éviterait au concepteur le recours à différents langages. Ceci étant, plusieurs investissements ont été faits par des compagnies dans le développement d'outils de CAO qui s'articulent sur des langages autres que SYSTEMC. Celles-ci n'étant pas prêtes à réinvestir, des outils de transformation de descriptions SYSTEMC en VHDL ou en VERILOG sont apparus sur le marché. Ainsi, au niveau d'abstraction *système* (ou peu d'investissements y ont été faits, et où de toute façon VHDL et VERILOG ne sont pas efficaces), le comportement du système est décrit à l'aide de SYSTEMC. Après les tâches de synthèse, d'analyse et de vérification conduites à ce niveau, les fichiers qui en résultent sont transcrits en VHDL ou VERILOG, et les flots de conception existants peuvent être alors utilisés pour la suite de la conception, c'est-à-dire à partir du niveau *transfert de registres*.

Précisons que SYSTEMC est basé sur C++ et a donc tous les avantages que possède ce dernier (dont l'aspect programmation orientée-objet). Il offre aussi la possibilité d'utiliser d'autres types prédéfinis se rapportant aux signaux, ports, protocoles de communications, ..., et intègre un noyau pour la simulation. Il n'est pas possible de décrire dans cette thèse le langage SYSTEMC. Le lecteur s'y intéressant peut se référer par exemple à [13] où plusieurs informations concernant ce langage (dont notamment sa description) sont données. Néanmoins, nous présenterons ci-après quelques uns de ses aspects qui le caractérisent par rapport aux autres langages, et qui font de lui un langage incontournable pour la conception au niveau d'abstraction *système*.

Au niveau d'abstraction *système*, SYSTEMC permet au concepteur de décrire son système en deux étapes successives :

- modules fonctionnels communiquant à travers des canaux de communications abstraits
- des durées exprimées en unités de temps absolues sont affectées aux processus pour une modélisation permettant une analyse de performances, de compromis, ...

Ces deux étapes sont ainsi définies afin d'éviter au concepteur une étude fine tenant compte de l'aspect temporel alors que le comportement du système tel que décrit ne répond pas aux spécifications voulues. Avant d'aborder les détails concernant ces deux étapes, nous donnons ci-après quelques définitions nécessaires pour leur présentation.

Méthode : C'est une fonction décrite en C++ et qui est membre d'une classe
Module : C'est une entité structurelle pouvant contenir des processus, des ports, des canaux, et d'autres modules (dans le cas d'une structure hiérarchique)
Signal : Il est utilisé à des fins de communications entre des processus du même module.
Interface : Elle définit un ensemble de déclarations de méthodes, mais ne définit aucune implémentation de méthodes, ni de champs de données.
Canal : Il implémente une ou plusieurs interfaces et sert comme composant pour les communications.
Port : C'est un objet à partir duquel un module accède à une interface de canal de communications.
Canal primitif : Il est atomique, c'est-à-dire qu'il ne contient ni modules, ni processus, et peut directement accéder à d'autres canaux.
Canal hiérarchique : C'est un module contenant d'autres processus ou modules, et peut directement accéder à d'autres canaux.
Evènement : Il permet d'activer des processus ou de les suspendre.
Sensitivité : Elle définit le moment où un processus est activé.
Sensitivité statique : Elle est déclarée de manière statique pendant la phase d'élaboration (c'est-à-dire juste avant la simulation), et ne peut être modifiée une fois que la simulation ait commencé.
Sensitivité dynamique : Elle peut être modifiée au cours de la simulation.

Thread : Séquence d'exécution non préemptive (pas de modèle pour contrôler la corruption de données partagées due à un accès non protégé de ces données. Il revient à l'utilisateur d'en tenir compte, en utilisant par exemple l'exclusion mutuelle).
Wait() : Une méthode suspendant l'exécution d'une séquence. Les arguments qui y sont utilisés déterminent quand l'exécution de la séquence peut reprendre.
SC_THREAD : Une méthode pour un module qui a sa propre séquence d'exécution, et qui peut appeler un code appelant à son tour wait(). Les processus SC_THREAD sont automatiquement activés, c'est-à-dire qu'ils le sont dès qu'un évènement de la liste de sensitivité le permet.
SC_METHOD : C'est une méthode de module qui n'a pas sa propre séquence d'exécution (toutes les opérations qui y sont décrites s'exécutent d'un seul coup, du début jusqu'à la fin), et qui ne peut appeler wait().
SC_CTHREAD : C'est une méthode de module ayant sa propre séquence d'exécution, et dont la liste de sensitivité ne contient qu'un seul évènement à front montant ou descendant. Elle peut appeler wait(), mais avec une liste d'arguments restreinte. Les SC_CTHREADS processus sont automatiquement activés (dès que l'évènement de la liste de sensitivité le permet).

Ces définitions étant données, nous présentons ci-après quelques notions portant sur les modules et les processus, ainsi que sur les communications.

3.2.1. Modules et processus

Un module typique peut contenir:
- un ou plusieurs processus pour spécifier une logique combinatoire ou séquentielle.
- d'autres modules pour spécifier une hiérarchie
- une ou plusieurs fonctions pouvant être appelées à partir d'un processus ou module instancié

Nous avions dit auparavant que SYSTEMC est basé sur le langage C++. Il est ainsi basé sur la notion de classes, chacune possédant ses propres membres (privés, protégés ou publics). Un processus (SC_METHOD, SC_THREAD, SC_CTHREAD) est déclaré comme étant une fonction membre d'une classe du module considéré, et enregistré comme processus dans le constructeur du module.

Les processus utilisent des signaux pour communiquer entre-eux dans le même module. Un processus peut activer un autre processus en affectant une nouvelle valeur à un signal qui les lie. Notons que les variables ne servent pas directement pour les communications entre les processus. Elles sont utilisées à l'intérieur d'un processus donné. Quant au choix du type de processus, les trois types peuvent être utilisés pour la simulation, mais seul le processus de type SC_METHOD est utilisé pour la synthèse au niveau transfert de registres.

La création d'un module obéit à la syntaxe suivante :

```
#include "systemc.h"
SC_MODULE(nom_du_module)
{
 // Déclarations des ports du module
        // Exemple : sc_in <int> port1 ; sc_out <float> port2 ;
          //              sc_inout <double> port3 ;

 // Déclarations des signaux
        // Exemple : sc_signal <bool> signal1 ;

 // Déclarations des variables
        // Exemple : int val1 ; float val2 ;
```

```
 // Déclarations des fonctions
        // Exemple : int exple_fonction() ;

 // Déclarations des processus (Un processus ne peut être que du type void, sans arguments)
        // Exemple :   void méthode1() ;
          //                void méthode2() ;
          //                void méthode3() ;

 SC_CTOR(nom_du module)  // Constructeur du module
 {
  // Enregistrement des processus et déclarations des listes de sensitivité
        // Exemple : SC_METHOD(méthode1) ;
          //                sensitive << signal1 << ready ;

          //              SC_METHOD(méthode2) ;
          //                sensitive << ack ;

          //              SC_SLAVE(méthode3, port1) ;
  }
};
```

Notons que SC_MODULE et SC_CTOR sont des macros C++ définies dans la bibliothèque des classes de SYSTEMC.

Concernant les communications, un module utilise des ports pour communiquer avec d'autres modules. Dans une conception hiérarchique, les signaux sont utilisés pour communiquer avec des modules instantiés ou pour des communications entre des processus du même module. Lorsqu'un signal est connecté à un port, il doit avoir le même type que celui du port. Les variables ne sont pas utilisées pour les communications. Elles servent uniquement pour stocker des données relatives à un processus donné.

Un processus *SC_METHOD* réagit à un ensemble de signaux appelé *liste de sensibilité* qui peut être utilisée de différentes manières :

- fonction *sensitive()* // ne s'applique pas pour la logique séquentielle
- *sensitive stream* // ne s'applique pas pour la logique séquentielle
- fonction *sensitive_pos()*
- fonction *sensitive_neg()*
- *stream sensitive_pos*
- *stream sensitive_neg*

```
Exemple :   sc_in <bool> clock, b, reset ;
            ……..
            sensitive_pos(clock) ;              // Déclaration de fonction
            sensitive_neg << b << reset ;       // Déclaration de stream
             // le processus réagira sur les évènements concernant les signaux clock, b et reset
```

En ce qui concerne les fonctions, il existe deux types :

- les fonctions qui servent à définir un processus (enregistrement de processus)
- les fonctions qui sont membres d'un module donné et qui sont appelées par les processus du module considéré

Exemple :

```
SC_MODULE(nom_module)
{
 int  fct() ;
 void méthode1() ;
 SC_CTOR(nom_du_module)  // Constructeur du module
 {
  SC_METHOD(méthode1) ;
        .....
 }
};
void nom_module :: méthode1()
 {
  int x ;
   .....
   x= fct() ;
  }
```

La lecture/écriture des ports et des signaux se fait en utilisant les méthodes *read()* et *write()*. Pour ce qui est des variables, les opérations de lecture/écriture se font par affectation.

```
Exemple : adresse=p1.read() ;              // méthode read()
          temp=adresse ;                   // affectation
          d_tmp=mémoire[adresse] ;    // affectation
          p2.write(d_tmp) ;                // méthode write()
```

Notons que la lecture et l'écriture des bits nécessitent quelques précautions.

Exemple :

```
sc_signal <sc_int<8>>a ;   // signal a à 8 bits
sc_int <8> b ;             // variable b
bool c ;                   // variable booléenne
b=a.read() ;               // lecture des 8 bits présents sur le port a
c=a[0] ;                   // affectation d'1 bit de a à c
```

Il est aussi important de souligner que lorsqu'une valeur est affectée à une variable, cette affectation prend effet immédiatement. Par contre, lorsque l'affectation concerne un signal ou à un port, la mise à jour n'est pas immédiate tant que le processus concerné n'est pas terminé. Ainsi, les autres processus utilisent toujours les précédentes valeurs de ces ports et signaux.

Exemple :

```
#include "systemc.h "
 SC_MODULE(exemple)
 {
  int a;
   sc_in <bool> clk, donnée ;
  sc_inout <bool> b, c;
```

```
void lecture_écriture_ports()
{a=5;                          // l'affectation est immédiate
 b.write(donnée.read());
 c.write(b.read()) ;           // l'écriture sur le port c se fait avec l'ancienne
                               // valeur de b, et non pas avec celle qui vient
                               // d'être lue à partir du port-signal donnée
}
SC_CTOR(exemple)
{
  SC_METHOD(exemple);
  sensitive_pos << clk;
 }
};
```

3.2.2. Composantes fondamentales d'une communication

Pour établir une communication, certaines informations sont nécessaires. Ce sont :

- la direction de la communication (*in*, *out*, *inout*)
- l'initiateur de la communication (*master*, *slave*)
- le type de la donnée transmise
- éventuellement, un index de la transaction (*abstract @*)
- les attributs du canal de communications pour configurer un canal ou une transaction (par exemple, une communication peut être bloquée après l'écoulement d'une certaine durée : *time-out*)
- le protocole du bus (détails du protocole)

Nous avions auparavant défini un port de communications. Un processus peut communiquer à travers un port-signal ou un port abstrait.
Un port-signal est connecté à un signal, et peut être de trois types :

- sc_in (port d'entrée connecté à un signal)
- sc_out (port de sortie connecté à un signal)
- sc_inout (port d'entrée/sortie connecté à un signal)

Dans le langage SYSTEMC, il existe des classes de ports pour les *ports-maîtres* et les *ports-esclaves*. Au niveau fonctionnel, ces ports sont abstraits pour décrire une communication abstraite à ce niveau. Lorsque la communication sera détaillée, un port abstrait sera implémenté par un port de bus. Il existe huit types de ports abstraits :

- sc_master < >
- sc_inmaster <T>
- sc_outmaster <T>
- sc_inoutmaster <T>
- sc_slave < >
- sc_inslave <T>
- sc_outslave <T>
- sc_inoutslave <T>

Il est à noter que T représente l'un de ces types suivants :

- un type défini dans le langage C++ (long, int, char, short, float, double, ...)
- un type défini dans SYSTEMC (sc_int<N>, sc_uint<N>, sc_bigint<N>, sc_biguint<N>, sc_bit, sc_logic, ...)
- un type défini par l'utilisateur

Notons aussi que *sc_master< >* et *sc_slave< >* concernent uniquement le contrôle, sans aucune indication concernant la direction ou le type de donnée transmise. Un exemple en serait de modéliser une interruption.

Nous donnons ci-après quelques exemples d'utilisation de ports abstraits en SYSTEMC :

```
i)
sc_inmaster<int> P1[10] ;      // Tableau de 10 ports de type sc_inmaster pour recevoir des
                               // données de type int
int data ;
P1[5]=data ;
ii)
sc_outmaster<int, sc_indexed<1024>> P2 ; // Port de type sc_outmaster avec un adressage allant
                                         // de 0 à 1023 pour transmettre des données de type int
P2[5]=777;
index=P2.get_address();  // index=5
memory[index]=P2 ;       // ranger 777 dans le 6ème mot-mémoire

iii)
sc_inmaster<int, 1024> P3[10] ;
int data ;
data=P3[5][68];  // data est égale à la valeur rangée à la 69ème position dans le 6ème port

iv)
sc_inoutslave<int> P1 ;
.....
if(P1.input())          // la direction du transfert de données sur le port P1 est obtenue par
                        // l'appel à la fonction input()
  val=P1;           // Port P1 défini comme port d'entrée
else P1=val;        // Port P1 défini comme port de sortie
```

Il est important de souligner que l'accès à un port donné définit un ordre d'exécution des processus concernés. Ainsi, dans la Fig.3.1, l'ordre d'exécution est : A, A11, A12, A13, A21 et A22 si le port P1 de A est accédé avant P2.

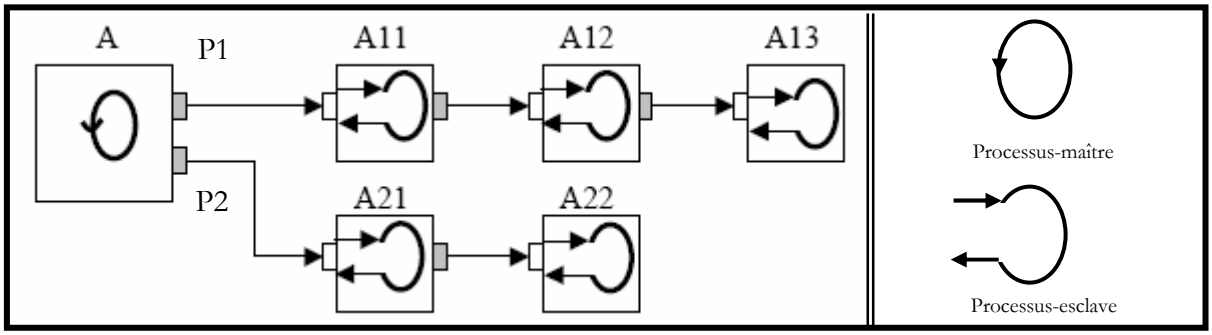

Fig.3.1. Exemple de communication point à point

Dans le cas d'une communication multipoints (Cf. Fig.3.2), lorsque le processus-maître accède à un de ses ports, il invoque tous les processus-esclaves qui sont dans sa liste. Un processus-esclave s'étant exécuté, il rend contrôle pour permettre à un autre processus-esclave (en tenant compte de l'ordre de profondeur) de s'exécuter à son tour. Si à un moment donné un

processus-esclave se bloque sur un *wait()*, alors les autres processus qui sont dans la liste se bloquent à leur tour, à cause de la profondeur.

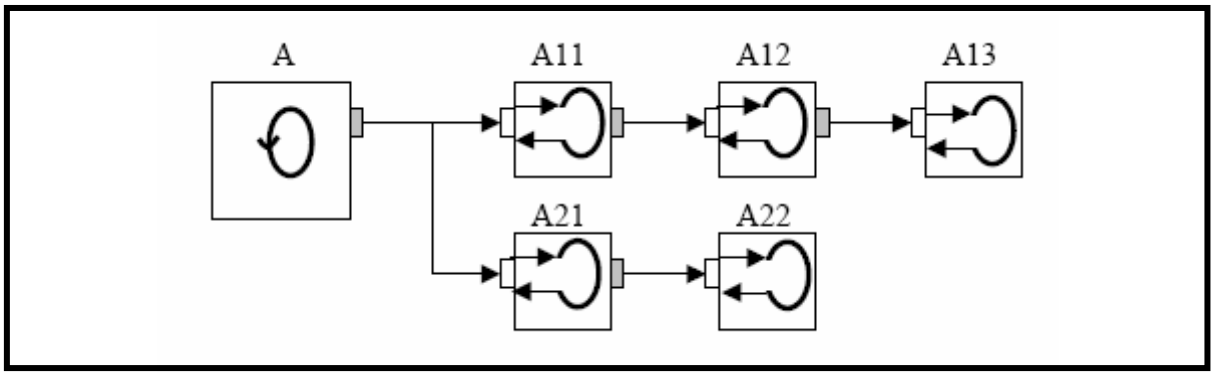

Fig.3.2. Exemple de communication multipoints

Notons que les processus-esclaves dans les deux listes (Fig.3.2) s'exécutent séquentiellement, mais de manière indéterminée. Les ordres d'exécution valides sont : A, A11, A12, A13, A21, A22, ou A, A21, A22, A11, A12, A13. L'objet *sc_link_mp<T>* ne définit aucun moyen d'arbitrage ou d'affectation de priorités, mais le concepteur pourra toujours définir un ordre de priorité d'exécution lors de la description détaillée du canal abstrait.

De même que pour les ports, les canaux de communications sont modélisés dans un premier temps de manière abstraite, puis affinés ultérieurement. En SYSTEMC, un canal abstrait est défini par: *sc_link_mp<T> link1*
où T est le type de la donnée transmise à travers le lien (canal) link1. Notons que :
- les ports connectés à un canal doivent avoir le même type que lui
- sc_link_mp<T> ne s'applique que pour connecter des ports abstraits

Les règles de communications point à point sont les suivantes :

- sc_master	--- sc_link_mp ---	sc_slave
- sc_outmaster	--- sc_link_mp ---	sc_inslave
- sc_inmaster	--- sc_link_mp ---	sc_outslave
- sc_inoutmaster	--- sc_link_mp ---	sc_inslave
- sc_inoutmaster	--- sc_link_mp ---	sc_outslave
- sc_outmaster	--- sc_link_mp ---	sc_inoutslave
- sc_inmaster	--- sc_link_mp ---	sc_inoutslave
- sc_inoutmaster	--- sc_link_mp ---	sc_inoutslave

Enfin, dans une conception hiérarchique, les règles de connexions pour les ports abstraits doivent obéir à certaines règles:
- un port d'un module-fils connecté à un port du module-parent doit être de même type que ce dernier
- un port-maître d'un module-parent doit être connecté à un port-maître du module-fils
- un port-esclave d'un module-parent doit être connecté à un port-esclave du module-fils
- un port d'entrée d'un module-fils peut être connecté à un port d'entrée ou d'entrée/sortie du module-parent
- un port de sortie d'un module-fils peut être connecté à un port de sortie ou d'entrée/sortie du module-parent
- un port d'entrée/sortie d'un module-fils peut être connecté à un port d'entrée/sortie du module-parent

L'exemple suivant, correspondant à la Fig.3.3, respecte les règles qui viennent d'être énumérées:

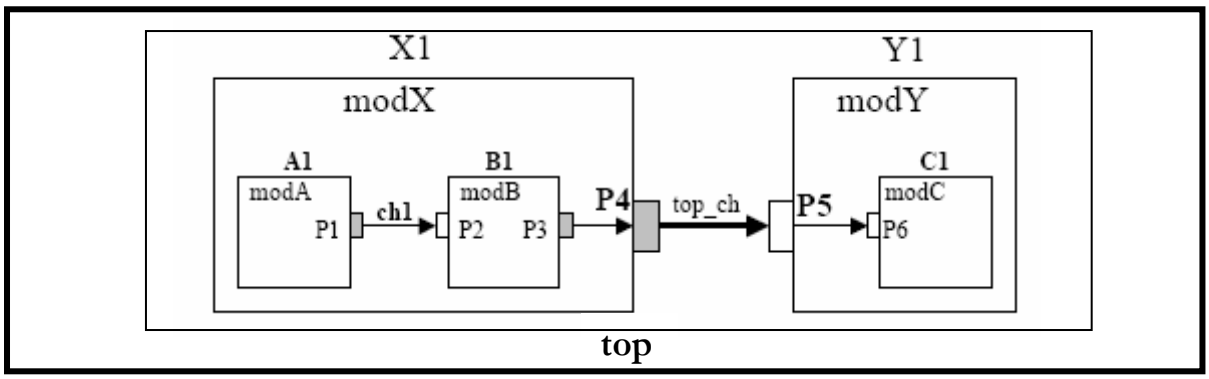

Fig.3.3. Exemple de modules hiérarchiques

```
SC_MODULE(modA) {
 sc_outmaster<int> P1;
 ... };

SC_MODULE(modB) {
 sc_inslave<int> P2;
 sc_outmaster<int> P3;
 ... };

SC_MODULE(modC) {
 sc_inslave<int> P6;
 ... };

SC_MODULE(modX) {
 sc_outmaster<int> P4;
 sc_link_mp<int> ch1;
 modA *A1;
 modB *B1;
 ...
 SC_CTOR(modX) {
 A1 = new modA("A1");
 A1(ch1);
 B1 = new modB("B1");
 B1(ch1, P4);
 } };

SC_MODULE(modY) {
 sc_inslave<int> P5;
 modC *C1;
 ...
 SC_CTOR(modY) {
  C1 = new modC("C1");
  C1(P5);
  }
};
```

```
SC_MODULE(top) {
 sc_link_mp<int> top_ch;
 modX *X1;
 modY *Y1;
 ...
 SC_CTOR(top) {
 X1 = new modX("X1");
 X1(top_ch);
 Y1 = new modY("Y1");
 Y1(top_ch);
 }
};
```

3.2.3. Simulation comportementale

Les éléments du langage SYSTEMC que nous venons de présenter sont les éléments les plus importants pour décrire de manière abstraite le système à concevoir. Le comportement du système est alors vérifié grâce à un noyau intégré dans SYSTEMC qui permet l'utilisation d'un simulateur comportemental obéissant à l'algorithme suivant :

1. Tous les signaux d'horloge qui doivent changer de valeur au temps courant sont mis à jour
2. Tous les processus de type SC_METHOD et SC_THREAD dont les entrées ont changé de valeur s'exécutent. Toutefois, alors que toutes les opérations d'un processus de type SC_METHOD sont exécutées, l'exécution des processus de type SC_THREAD est suspendue à la rencontre d'un *wait()*.
3. Mise à jour, à partir d'une liste, des sorties de tous les processus de type SC_CTHREAD (sans exécution de ceux-ci) qui sont contrôlés par des horloges. Ces nouvelles valeurs sont rangées dans une liste pour être utilisées ultérieurement en 5. Il y'a aussi une mise à jour de toutes les sorties des processus de type SC_METHOD et SC_THREAD qui ont été exécutés en 2.
4. Les pas 2 et 3 sont répétés jusqu'à ce qu'aucun signal d'entrée ne change de valeur
5. Exécution de tous les processus de type SC_CTHREAD contrôlés par une horloge dont les sorties ont été mises dans une liste au troisième pas de l'algorithme. Les nouvelles valeurs des sorties sont alors mises à jour au prochain front montant (ou descendant) du signal d'horloge qui les commande, au troisième pas de l'algorithme.
6. Le temps de simulation est incrémenté au prochain front montant (descendant) du signal d'horloge utilisé pour la simulation, et l'algorithme reprend au premier pas.

La simulation est contrôlée par l'appel à la fonction *sc_start()*. Cet appel se fait à partir de la routine *sc_main()*. Le temps de la simulation dépend alors de l'argument de la fonction *sc_start()*. La simulation peut être arrêtée en appelant la fonction *sc_stop()*. L'utilisateur peut connaître le temps courant de la simulation en faisant appel à la fonction *sc_simulation_time()*. La visualisation de la simulation peut se faire à l'aide de tout éditeur graphique supportant les fichiers suivants qui sont créés par SYSTEMC :

- des fichiers VCD (Value Change Dump),
- des fichiers ASCII WIF (Waveform Intermediate Format)
- des fichiers ISDB (Integrated Signal Data Base)

Ces fichiers contiennent les valeurs de toutes les variables et signaux que le concepteur veut visualiser, et ce, au fur et à mesure que celles-ci changent durant la simulation. Pour générer un fichier de type VCD par exemple, il faut faire appel à la fonction

*sc_create_vcd_trace_file(*nom_de_fichier*)* à partir de la routine *sc_main()*, une fois que tous les modules et signaux ont été définis. La fonction *sc_close_vcd_trace_file(*nom_de_fichier*)* permet de fermer le fichier concerné.

Exemple :

```
  #include "systemc.h"
  #include "producer.h"
  #include "consumer.h"
  #define NS 1e-9                                   // Unités de temps exprimées en ns
  sc_trace_file *my_trace_file;
  sc_signal<int> sortie;
  sc_signal<int> somme;
  sc_clock clk(char *, double, double);
  int sc_main(int argc, char **argv)
{
   sc_link_mp<int> link1;
   sc_link_mp<int> link2;
   sc_clock clk("my_clock",20,0.3);        // Définition de l'horloge: période= 20 ns
                                           //          durée du '1' logique : 20 * 0.3 ns= 6 ns
                                           // =>    durée du '0' logique= 20 - 6=14 ns
   producer p("A");
   consumer c("B");
   p(link1,clk);
   c(clk,link1,link2);
   my_trace_file=sc_create_vcd_trace_file("simuler");  // Création du fichier simuler.vcd
   sc_trace(my_trace_file,clk,"CLOCK");                // Signaux à visualiser: clk,
   sc_trace(my_trace_file,sortie,"OUT");               //                       sortie, et
   sc_trace(my_trace_file,somme,"SUM");                //                       somme
   sc_start(300);                                  // Début de la simulation de durée  300 ns
   sc_close_vcd_trace_file(my_trace_file);             // Fermeture du fichier simuler.vcd
   return 0;
}
```

3.3. Structures de données dérivées d'une description en SYSTEMC :

Entamer des tâches de synthèse, d'analyse ou de vérification au niveau d'abstraction *système* en considérant comme entité de base une opération, conduirait à une complexité astronomique. Il est plus simple et plus commode d'abstraire le système à un ensemble de tâches communicantes qui sera affiné au cours de la progression de la conception à différents niveaux d'abstraction. Aussi, l'entité atomique au niveau d'abstraction *système* est la tâche. Il s'agit aussi de déterminer quelle tâche communique avec telle autre. Un graphe G=(V,E) où V={tâches t_i} et E={e_i=(t_j,t_k) / t_j et t_k communiquent entre-elles} permet de donner une première représentation du système au niveau fonctionnel. Dans notre cas, la tâche est déterminée à partir de la description donnée en SYSTEMC, et correspond à une procédure ou une fonction. Pour ce faire, des analyses syntaxique et lexicale d'une description en SYSTEMC sont nécessaires pour pouvoir identifier les tâches à partir desquelles le graphe de tâches sera déduit.

3.3.1. Analyses syntaxique et lexicale d'une description en SYSTEMC

Les compilateurs de compilateurs sont utilisés pour générer des processeurs de langages à partir de descriptions de haut niveau. Une fois la grammaire du langage spécifiée, un compilateur de compilateurs crée un exécutable qui traite le texte écrit dans ce langage. Outre le fait d'analyser

le langage en entrée, il permet aussi à l'utilisateur d'introduire des actions (instructions en C, C++, ...) qui seront exécutées une fois vérifiée une règle donnée. Ceci correspond parfaitement à notre problématique qui consiste à analyser une description en SYSTEMC, et d'en déduire les tâches ainsi que leurs intéractions. Rappelons que la *tâche* est l'entité atomique qui nous servira comme brique de base pour la synthèse, l'analyse et la vérification du système à concevoir.

Notre traitement d'une description en SYSTEMC s'articule sur le flot d'étapes décrit dans la Fig.3.4. Ce flot utilise l'utilitaire *lex* de Linux ainsi que l'exécutable *accent* qui est un compilateur de compilateurs ayant différentes caractéristiques :

- le compilateur de compilateurs ACCENT n'est pas restreint à des classes spécifiques de grammaires libres du contexte
- des noms symboliques plutôt que des nombres (dans YACC par exemple) sont utilisés pour les attributs, ce qui limite les erreurs de description de grammaires
- ACCENT a traité (selon ses concepteurs) des milliers de lignes de JAVA en quelques secondes

Les fichiers de la Fig.3.4 sont les suivants:

- *grammar-SYSTEMC.acc* décrit la grammaire de SYSTEMC tout en incluant des actions (en C) identifiant une tâche (les nœuds du graphe de tâches correspondent aux différentes fonctions et procédures incluses dans la partie déclarative d'un module –reconnaissance de *SC_MODULE*- et les arêtes sont déduites à partir des paramètres d'appel des fonctions et procédures, et des ports de communication –reconnaissance de *sc_link_mp*-)
- *spec.lex* définit les terminaux (tokens)
- la compilation en C (commande *cc*) des fichiers *yygrammar.h*, *yygrammar.c* (analyse syntaxique), *lex.yy.c* (analyse lexicale), main.c (programme principal) et *art.o* (temps d'exécution de *accent*) produit les résultats des analyses et le graphe des tâches.

3.3.2. Détermination du graphe des tâches à partir d'une description en SYSTEMC

Le fichier *grammar-SYSTEMC.acc* contient les règles de grammaire de SYSTEMC. Pour chacune de ces règles, on peut inclure des actions (instructions en C) à exécuter lors de la reconnaissance de la règle associée.

Les nœuds du graphe des tâches sont alors déterminés au fur et à mesure que la règle associée à une fonction ou procédure d'un module est reconnue et écrits dans un fichier (moyennant la fonction *fprintf ()*).

Les arêtes s'obtiennent à chaque reconnaissance d'un appel de fonction ou procédure (pour des fonctions et procédures d'un même module) et à chaque occurrence du terminal (token) *sc_link_mp* pour des communications entre différents modules.

3.4. Synthèse :

3.4.1. Nombre de sous-systèmes et affectation de tâches aux sous-systèmes

Le graphe des tâches étant défini, celui-ci ne permet toujours pas de savoir quelles sont les tâches qui seront implémentées par les mêmes composantes du système. L'affectation des tâches aux sous-systèmes a une conséquence importante sur les caractéristiques du système portant sur la surface, la vitesse, la consommation de puissance, et même sur la complexité de conception aux niveaux d'abstraction inférieurs (composants matériels et logiciels, protocoles de communications,...). De plus, le nombre de sous-systèmes est tout aussi important et engendre les mêmes conséquences que celles qui viennent tout juste d'être citées. Alors comment ce nombre doit il être défini ?

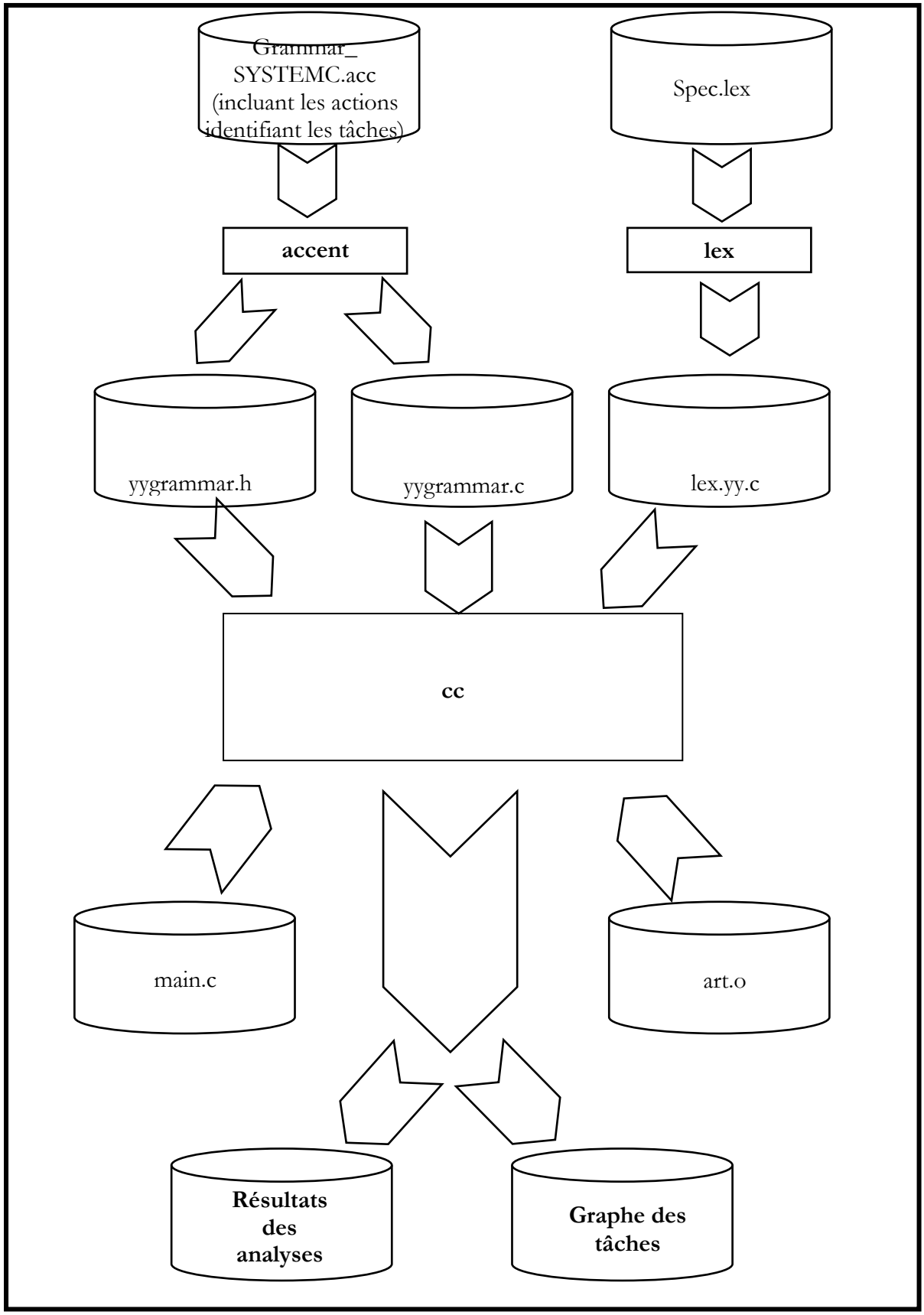

Fig.3.4. Analyses syntaxique et lexicale d'une description en SYSTEMC, et génération du graphe des tâches

```
%prelude {#include "yygrammar.h"}

%token ARROW, ARROW_STAR, DEC, EQ, GE, INC, LE, LOG_AND, LOG_OR, NE, SHL,
       SHR, ASS_ADD, ASS_AND, ASS_DIV, ASS_MOD, ASS_MUL, ASS_OR, ASS_SHL,
       ASS_SHR, ASS_SUB, ASS_XOR, DOT_STAR, ELLIPSIS, SCOPE,
       PRIVATE, PROTECTED, PUBLIC, BOOL, CHAR, DOUBLE, FLOAT, INT, LONG,
       SHORT, SIGNED, UNSIGNED, VOID, WCHAR_T, CLASS, ENUM, NAMESPACE,
       STRUCT, TYPENAME, UNION, CONST, VOLATILE, AUTO, EXPLICIT, EXPORT,
       EXTERN, FRIEND, INLINE, MUTABLE, REGISTER, STATIC, TEMPLATE,
       TYPEDEF, USING, VIRTUAL, ASM, BREAK, CASE, CATCH, CONST_CAST,
       CONTINUE, DEFAULT, DELETE, DO, DYNAMIC_CAST, ELSE, FALSE, FOR, GOTO,
       IF, NEW, OPERATOR, REINTERPRET_CAST, RETURN, SIZEOF, STATIC_CAST,
       SWITCH, THIS, THROW, TRUE, TRY, TYPEID, WHILE, SC_LINK_MP;
                              ……

translation_unit: declaration_seq_opt {printf("\n translation_unit1");
printf("\n !! SUCCES !!\n");}
                  ;
identifier_word: Identifier {printf("\n identifier_word1\n");}
                ;
identifier: identifier_word
            {printf("\n identifier1\n");}
          ;

id: identifier
                {printf("\n id1\n");}
  | identifier template_test '+' template_argument_list '>'
                {printf("\n id2\n");}
  | identifier template_test '+' '>' { ERRMSG("Empty template-argument-
list"); printf("\n id3\n");}
  | identifier template_test '-'
                {printf("\n id4\n");}
  | template_id
                {printf("\n id5\n");}
  ;
..........................
function_block: ctor_initializer_opt function_body
{ptr_node=(NODE *) malloc(sizeof(NODE)); inserer_dans_G(ptr_node);
;}
               ;
..........................
--------------------------------------------------------------------------
%{
#include "yygrammar.h"
%}
%%

" "   {printf("\n BLANCS\n"); /* skip blancs */}
"\t"  { }
"\n"  {yypos++; /* adjust line number and skip newline */}
"=="  {printf(" == "); return EQ;}
">="  {printf(" >= "); return GE;}
"<="  {printf(" <= "); return LE;}
..........................
"private"   {printf(" private "); return PRIVATE;}
"protected" {printf(" protected "); return PROTECTED;}
"public"    {printf(" public "); return PUBLIC;}
"bool"      {printf(" bool "); return BOOL;}
..........................
```

Fig.3.5. Parties des fichiers Grammar_SYSTEMC.acc et Spec.lex

Dans [14], ce nombre N est défini de manière arbitraire, et la répartition des tâches est faite sur les N sous-systèmes.

Dans [15], N est défini à partir de la fonctionnalité-même du système, c'est-à-dire qu'il y'a autant de sous-systèmes que de *threads* pouvant s'exécuter en parallèle (Cf. Fig.3.6).

Concernant [14], il apparaît clairement qu'un mauvais choix de N a une implication grave sur la conception du système aux autres niveaux de conception :

- En effet, si N est sous-estimé, les tâches seraient réparties sur un faible nombre de composants (densité très importante), ce qui fait que certains d'entre-eux auraient à exécuter beaucoup de tâches, nécessitant ainsi un nombre important de ressources physiques dans le même sous-système. Ceci engendrerait une conception difficile au niveau *dessin de masques* pour diminuer les paramètres parasites impliqués par un très grand nombre de modules et d'interconnexions présents dans un voisinage immédiat (congestion). Cette diminution ne pourrait se faire qu'au prix de la diminution des croisements des interconnexions (quand c'est possible), leur éloignement, ..., ce qui n'est pas sans conséquence néfaste sur la surface, donc sur le coût de la fabrication.

- Si N est très grand, les paramètres parasites seraient plus faibles, mais ceci amènerait des tâches à communiquer fréquemment à travers le bus global car il est très probable que ces tâches soient affectées à des sous-systèmes différents. La capacité du bus étant très importante, la consommation de la puissance le serait aussi. Cette méthode de conception n'est pas indiquée car elle n'est pas adaptée aux systèmes monopuce : une forte consommation de puissance entraînerait une augmentation de la température et aurait une conséquence sur la fiabilité du système et sur son coût de fabrication (encapsulation et refroidissement). Pour les systèmes portables, ceux embarqués sur des satellites,...., l'autonomie des batteries en serait affectée. De plus, la probabilité que des tâches communicantes soient affectées à des sous-systèmes différents étant importante, le bus global étant unique, plusieurs communications pouvant se faire en théorie en parallèle, doivent être différées afin qu'il n'y'ait pas de conflit de communications sur le bus global unique. Ainsi, les performances du système en matière de vitesse seraient aussi dégradées.

Concernant [15], la fonctionnalité du système (nombre de *threads* pouvant s'exécuter en parallèle) pourrait aussi engendrer des valeurs de N qui ne seront pas intéressantes pour la suite de la conception, et les mêmes problèmes évoqués pour une valeur de N mal choisie surgiraient. Comme cas extrêmes, si le système est décrit par un seul *thread* contenant un nombre très important d'opérations, alors il serait implémenté par un seul composant, avec tous les problèmes décrits et aggravés pour une faible valeur de N. Dans le cas où la fonctionnalité du système est décrite par un très grand nombre de *threads*, les problèmes évoqués précédemment pour de grandes valeurs de N seraient engendrés.

Dans notre cas, les arêtes E du graphe G=(V,E) sont définies telles qu'une arête existe entre 2 nœuds de V si et seulement si les tâches correspondantes à ces nœuds ont un certain degré de communications. Ces degrés de communications sont répartis en 4 ensembles :

- aucune communication entre 2 tâches données
- faible degré de communications entre 2 tâches
- degré de communications moyen entre 2 tâches
- fort degré de communications entre 2 tâches données

Dans un premier temps, il existe une arête pour tout couple de tâches qui ne sont pas fortement communicantes. Un partitionnement de tâches basé sur le fait que 2 tâches données soient affectées à la même partition si elles sont fortement communicantes aboutirait à un système tel que ces 2 tâches ne solliciteraient pas le bus global (fortement capacitif) pour communiquer. Celui-ci est utilisé uniquement par des transferts de données entre des tâches qui communiquent rarement, entraînant ainsi des charges et de décharges peu fréquentes de la capacité du bus global.

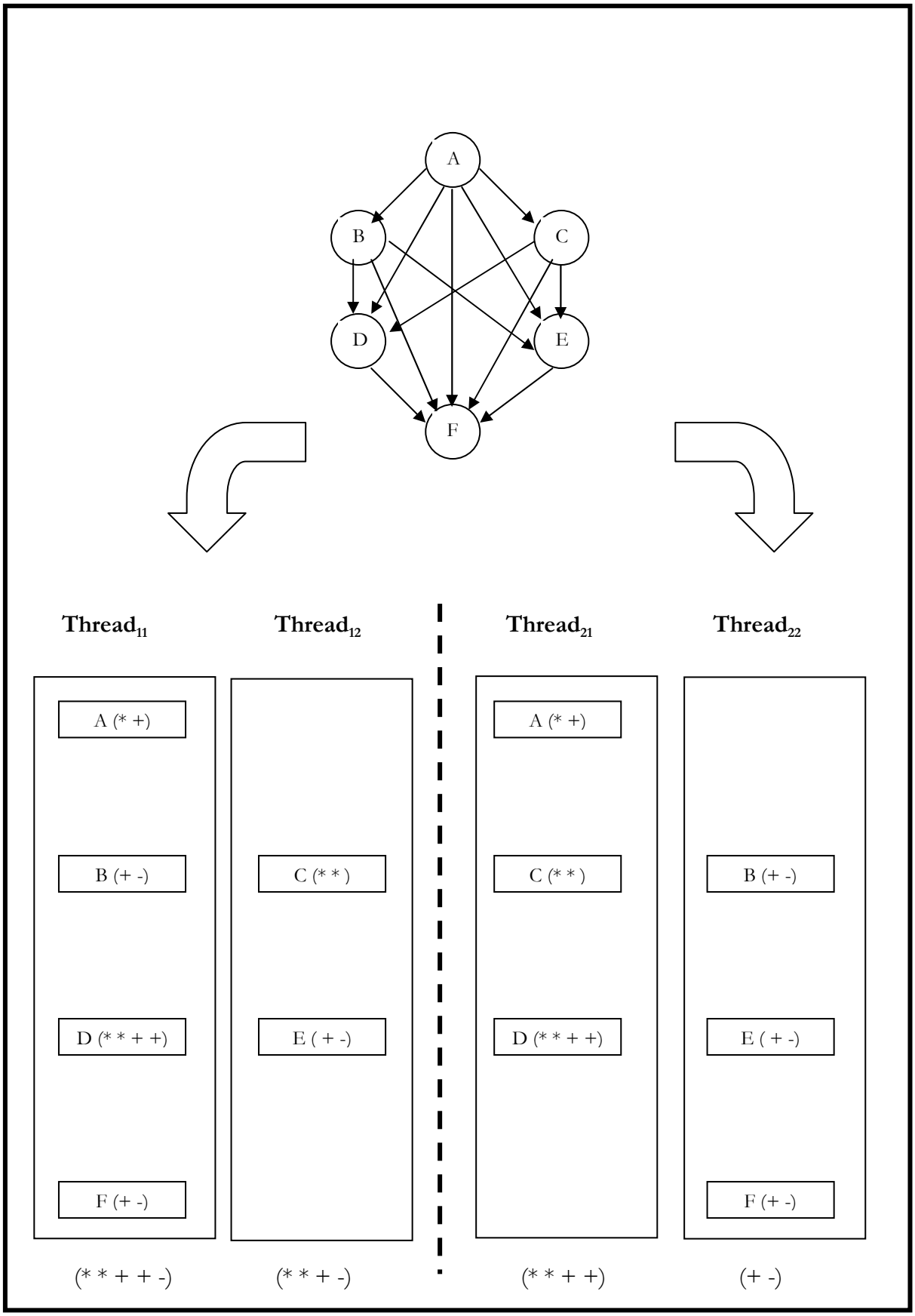

Fig.3.6. Méthode de détermination du nombre de sous-systèmes VLSI [15]

Toutefois, à l'inverse de [14] et de [15], si l'analyse donne des caractéristiques qui ne sont pas encore intéressantes (toujours aux niveaux d'abstraction élevés), les arêtes E de G seront redéfinies à partir de tâches qui ne communiquent pas, ou de celles qui ont un faible degré de communications, et ainsi de suite jusqu'à obtenir des caractéristiques intéressantes.

Ce premier partitionnement, qu'on peut appeler *macro-partitionnement* est suivi par un *micro-partitionnement*. En effet, si l'affectation des tâches à travers les différents sous-systèmes est bien connue, il reste à déterminer l'affectation des tâches à travers les processeurs et les ASICS du *même* sous-système. Ce micro-partitionnement est aussi dicté par les différentes contraintes, et le meilleur serait celui d'affecter autant que possible les tâches fortement communicantes dans le même composant (ASIC ou processeur) du même sous-système. Ceci aurait pour conséquence d'utiliser des bus locaux très fréquemment, ce qui permettrait de libérer le bus global du sous-système pour quelques transferts de données entre des tâches affectées au même sous-système. En d'autres termes, ceci permettait d'augmenter le parallélisme et de diminuer la consommation de puissance (le bus global est très capacitif par rapport aux bus locaux). Le traitement est similaire au *macro-partitionnement* sauf que dans le cas du *micro-partitionnement*, il y'a construction d'un graphe pour chaque sous-système. Pour chacun de ces graphes, les nœuds correspondent aux tâches affectées au même sous-système, et les arêtes sont définies comme dans le cas du macro-partitionnement. Enfin, notons que la décision d'affecter une tâche à un processeur ou un ASIC dépend de l'outil de décomposition Hardware-Software, ou du concepteur (la décomposition Hardware-Software est faite manuellement pour les outils industriels à cause de la diversité des applications et des résultats encore médiocres obtenus par une décomposition automatique).

Ainsi, cette méthodologie a pour avantages de générer plusieurs configurations à différentes caractéristiques (dès les plus hauts niveaux de conception), ce qui donne au concepteur la possibilité de choisir celle qui offre le meilleur compromis (surface – vitesse – consommation de puissance), ou celle qui répond le mieux à son application.

Jusqu'ici, on conjecture que le problème de partitionnement est NP-complet ([2]) et des encouragements sont offerts pour prouver *formellement* soit qu'il n'existe pas d'algorithme polynomial pour de tels problèmes, soit pour exhiber un algorithme polynomial permettant de les résoudre ([16]). Notons que les problèmes NP-complets peuvent être résolus par le *même* algorithme par des transformations polynomiales sur les données et les résultats de ces problèmes. Ceci fait que s'il existait un algorithme polynomial pour résoudre un problème NP-complet (permettant donc de résoudre *tous* les problèmes NP-complets), alors on aurait P = NP (Cf. Fig.3.7). Notons qu'un problème P2 est NP-complet:

- s'il appartient à la classe NP, c'est à dire s'il peut être résolu par un algorithme *non déterministe polynomial* et
- s'il existe une transformation polynomiale de P1 à P2, P1 étant NP-complet (tel que le problème 3-SAT démontré NP-complet par Cooke)

Il est aussi connu que le problème des graphes colorés est, comme le problème de partitionnement, NP-complet ([2]). De ce fait, notre problème d'affectation de tâches à différentes partitions dans le cas du macro-partitionnement (différents composants du même sous-système dans le cas du micro-partitionnement) peut être résolu par analogie : 2 tâches données dont les nœuds auraient reçu la même couleur seront affectées au même sous-système dans le cas du macro-partitionnement (au même composant du même sous-système dans le cas du micro-partitionnement). Autrement, elles seraient affectées à des sous-systèmes (composants) différents. Pour ce faire, j'ai utilisé une nouvelle technique de coloration de graphes que j'ai développée et qui a été implémentée en C++ par deux collègues ([17], [18], [19]). La Fig.3.8 donne un exemple d'illustration de la répartition des tâches sur des sous-systèmes par notre

technique des graphes colorés. Les détails de cette dernière étant indiqués dans [17], [18] et [19], nous donnons dans ce document juste ses grandes lignes.

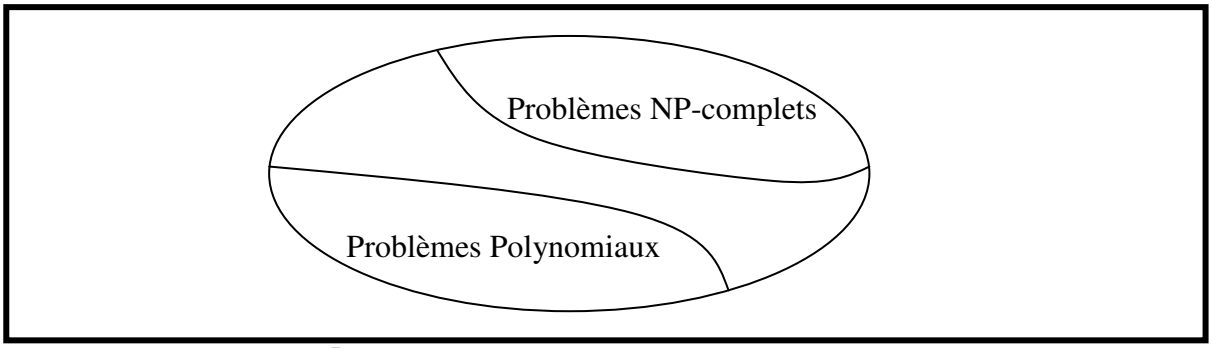

Fig.3.7. Classe NP= $\sum_1^P$

Soit à colorer un graphe G=(V,E). Ce problème est formulé de la manière suivante :
Trouver la fonction f minimisant K ($K \leq |V|$) telle que :
$f : V \rightarrow \{1,2, ... , K \}$
$f(v_i) \neq f(v_j) \ \forall \ (v_i , v_j) \in E$
La Fig.3.9 montre que le nombre chromatique est fortement dépendant de E.
Ayant affecté des degrés aux nœuds (un degré d'un nœud est le nombre de nœuds qui lui sont connectés), des études ([17], [18] et [19]) ont montré que ni l'ordre ascendant, ni l'ordre descendant des degrés donnait toujours le meilleur résultat. Aussi, une méthode triviale pour aboutir à un nombre chromatique intéressant serait de colorer G en tenant compte de tous les arrangements possibles opérés sur les nœuds. Cette méthode consisterait alors à considérer $|V|!$ colorations possibles (chaque coloration tenant compte d'un certain arrangement de nœuds), puis opter pour celle donnant la plus faible valeur de K. Il est évident qu'un tel algorithme n'est pas polynomial, et que son implémentation engendrerait un temps CPU prohibitif à partir d'une certaine valeur de $|V|$. Notre technique est un compromis entre *opter pour une seule coloration* et *faire toutes les colorations possibles*. Elle consiste donc à trouver un compromis entre la qualité de la solution et le temps CPU nécessaire pour la produire. Pour ce faire, nous affectons dans un premier temps tous les nœuds du même degré dans la même classe, obtenant ainsi M classes ($M \leq |V|$). Les colorations de G sont alors faites en tenant compte de tous les arrangements possibles sur les classes (M ! colorations), les nœuds appartenant à la même classe considérés *équivalents*, et colorés dans un ordre indifférent. Toutefois, le problème de temps CPU se pose toujours dans le sens où :

- si les nœuds ont tous des degrés différents, alors on aurait $M=|V|$
- si quelques nœuds seulement ont les mêmes degrés, la valeur de M resterait toujours importante, et le problème de temps CPU se poserait toujours

Nous avions alors opté pour une autre alternative qui consiste non pas à mettre dans la même classe des nœuds de même degré, mais plutôt des nœuds de degrés proches. En plus, nous fixons dès le départ un nombre de classes (que l'utilisateur de notre outil peut choisir à sa guise), puis répartissons les nœuds sur ces classes. Cette répartition, comme le montre la Table 3.2, n'est pas unique. Nous optons alors pour la meilleure qui consiste à donner une uniformité globale en termes de degrés de nœuds à travers les classes. Ceci est la solution au problème suivant :

$$\min \sum_{i=1}^{k} diff_i$$

tel que : *i)* $diff_i = d_{Mi} - d_{mi}$; i=1,2,…,k;

ii) $\sum_{i=1}^{k} |Cl_i| = N$;

où d_{Mi} et d_{mi} sont respectivement les degrés maximal et minimal des nœuds de la classe Cl_i. N est le nombre de nœuds (N=|V|), et k est le nombre de classes.

Considérons la Fig.3.10. Les degrés des noeuds sont ceux indiqués dans la Table 3.1, et les différentes répartitions possibles des nœuds à travers trois classes sont indiquées dans la Table 3.2. La troisième répartition est alors la solution au problème qui vient juste d'être présenté :

Cl_1={v1} ; Cl_2={v2, v3, v4} ; Cl_3={v5}

Ainsi, les arrangements possibles sur les trois classes sont les suivants:

Cl_1,Cl_2,Cl_3 → {v1},{v2,v3,v4}, {v5} // colorations successives de v1, v2, v3, v4, puis v5
Cl_2,Cl_1,Cl_3 → {v2,v3,v4},{v1},{v5} // colorations successives de v2, v3, v4, v1, puis v5
Cl_3,Cl_2,Cl_1 → {v5},{v2,v3,v4},{v1} // colorations successives de v5, v2, v3, v4, puis v1
Cl_1,Cl_3,Cl_2 → {v1},{v5},{v2,v3,v4} // colorations successives de v1, v5, v2, v3, puis v4
Cl_3,Cl_1,Cl_2 → {v5},{v1},{v2,v3,v4} // colorations successives de v5, v1, v2, v3, puis v4
Cl_2,Cl_3,Cl_1 → {v2,v3,v4},{v5},{v1} // colorations successives de v2, v3, v4, v5, puis v1

Pour chaque arrangement, notre technique tient compte des ordres ascendant et descendant des degrés des nœuds. Ainsi, étant donné un nombre M de classes, le nombre de colorations est M ! * 2. Notons que pour l'arrangement Cl_1, Cl_2, Cl_3, les deux colorations possibles sont :

* Colorer successivement v1, v2, v3, v4, puis v5 // Coloration différente de celle de Cl_3, Cl_2, Cl_1 dans l'ordre ascendant des degrés des nœuds (qui est v1, v4, v3, v2, puis v5)
* Colorer successivement v5, v4, v3, v2, puis v1 // Coloration différente de celle de Cl_3, Cl_2, Cl_1 dans l'ordre descendant des degrés des nœuds (qui est v5, v2, v3, v4, puis v1).

Avec un nombre de classes égal à 3, le graphe de la Fig.3.10 sera coloré de 12 manières différentes (=3 ! * 2), et la meilleure coloration qui sera retenue sera celle donnant le plus faible nombre chromatique. Enfin, notons que le nombre M de classes doit vérifier : M ≤ |V|et qu'il sera choisi par l'utilisateur de l'outil de telle sorte que les différentes colorations se fassent en un temps CPU raisonnable et/ou fixé.

Comme il a été dit précédemment, si l'analyse faite sur la configuration d'architecture obtenue révèle des caractéristiques qui n'intéressent pas le concepteur, notre méthodologie lui permet d'obtenir d'autres configurations à partir d'autres graphes. Ceux-ci sont générés en modifiant itérativement l'ensemble des arêtes jusqu'à obtention de la configuration désirée. Notons que la modification des arêtes (ajout et / ou suppression des arêtes) est facilitée et est conduite à la base des caractéristiques (surface, vitesse et consommation de puissance) des configurations déjà générées. La Fig.3.11 donne un exemple d'implémentation pour la répartition des tâches à travers les sous-systèmes obtenus dans l'exemple de la Fig.3.8.

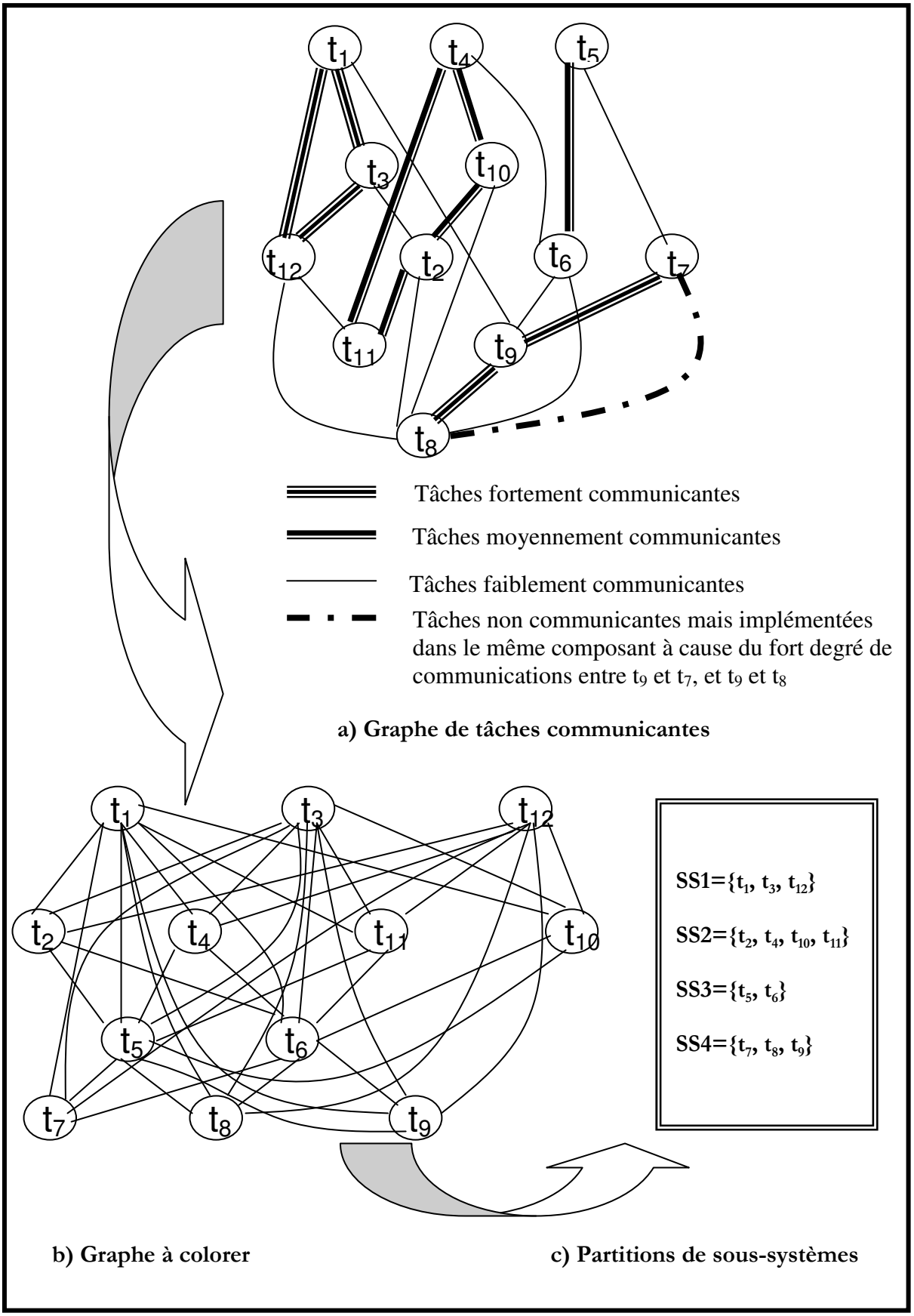

Fig.3.8. Exemple d'illustration de notre méthode pour la détermination des sous-systèmes et la répartition des tâches à travers ces sous-systèmes

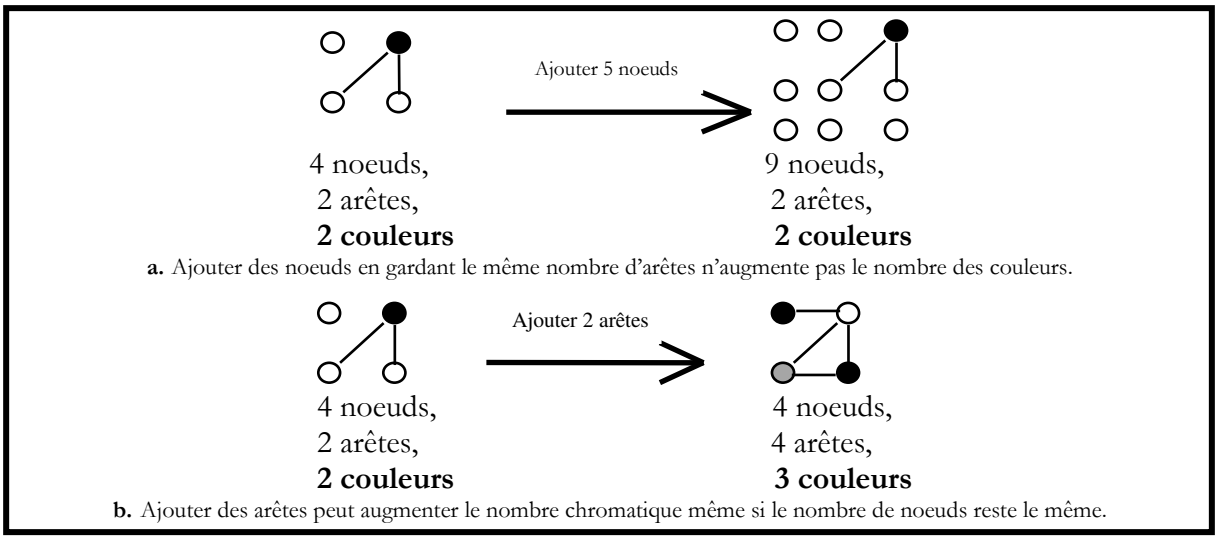

Fig. 3.9. Modification du nombre chromatique en fonction de E de G=(V,E)

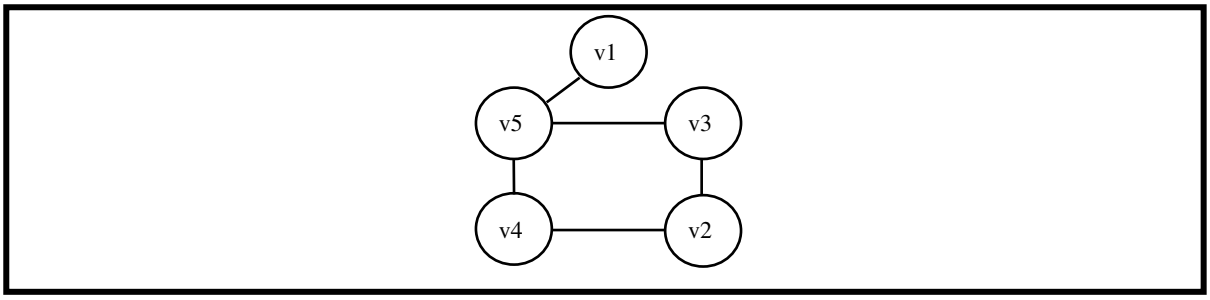

Fig.3.10. Un exemple de graphe

Table 3.1. Degrés des nœuds du graphe de la Fig.3.10

Noeud	v1	v2	v3	v4	v5
Degré	1	2	2	2	3

Table 3.2. Répartition de nœuds à travers des classes

Cl1		**Cl2**		**Cl3**		
#nodes	**$diff_1$**	**#nodes**	**$diff_2$**	**#nodes**	**$diff_3$**	**$diff_1$+ $diff_2$+ $diff_3$**
1	0	1	0	3	1	1
1	0	2	0	2	1	1
1	**0**	**3**	**0**	**1**	**0**	**0**
2	1	1	0	2	1	2
2	1	2	0	1	0	1
3	1	1	0	1	0	1

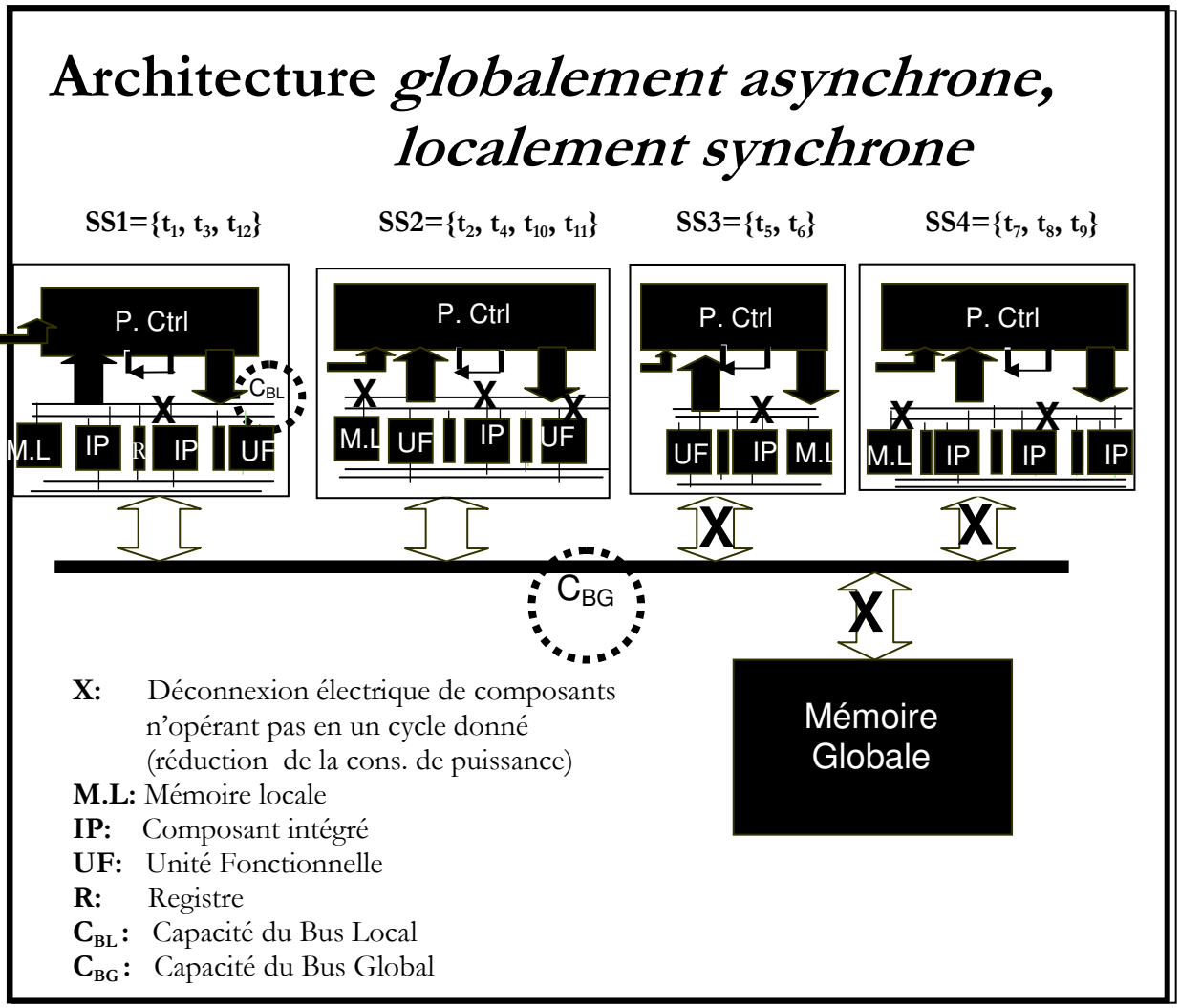

Fig.3.11. Exemple d'architecture correspondant à la répartition des tâches sur les sous-systèmes obtenus et indiqués dans la Fig.3.8

3.4.2. Synthèse des tâches

La répartition des tâches à travers les sous-systèmes étant faite, une synthèse de toutes les tâches composant le système est alors conduite ([27]). Celle-ci a pour objectifs d'optimiser l'architecture obtenue et de satisfaire les différentes contraintes, notamment pour les systèmes où les paramètres *énergie* et *temps* sont très critiques (systèmes embarqués, systèmes fonctionnant en temps réel).

Soit $S=\{P_1, P_2, \ldots, P_n, A_1, A_2, \ldots, A_m, B_g\}$un système composé de n processeurs, m ASICs et un bus global. Dans ce système, les tâches s'exécutent de manière concurrente et échangent des informations à travers le bus global. Chaque processeur ou ASIC dispose d'interconnexions locales et éventuellement d'une mémoire locale. Noter que dans chaque cycle d'exécution, le processeur P_i $(1 \leq i \leq n)$ exécute au plus une seule opération alors qu'un ASIC A_i $(1 \leq i \leq m)$ peut exécuter plusieurs opérations. Ainsi, dans le système hétérogène S, il existe des traces qui s'exécutent de manière concurrente (dans notre contexte, une trace est une succession de tâches). La figure 3.12 est un exemple d'un ensemble de traces.

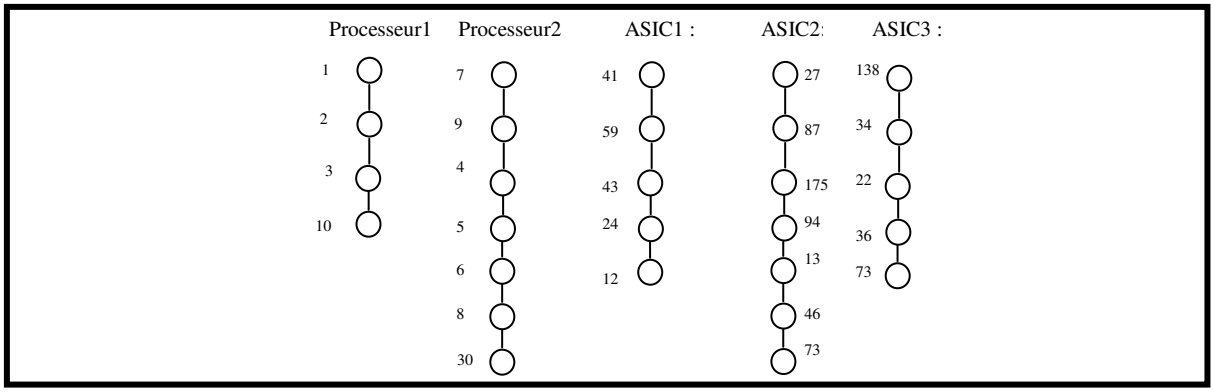

Fig.3.12. Ensemble de traces concurrentes affectées à des processeurs et des ASICs

Dans l'exemple de la figure 3.12, il est supposé que dans cet ensemble de traces concurrentes, le premier processeur exécute séquentiellement les tâches 1, puis 2, puis 3, puis 10 alors que le deuxième ASIC exécute les tâches 27, 87, …, 73. Il est à noter que lorsqu'un processeur exécute une seule opération en un cycle d'exécution donné, un ASIC peut par contre exécuter plusieurs opérations dans le même cycle. Avant de généraliser pour tout le système, considérons un processeur. Le même raisonnement est fait pour un ASIC en tenant toutefois compte des opérations qui s'exécutent de manière concurrente pendant le même cycle. Il s'agit alors d'affecter des tensions aux différentes tâches de sorte à ce que toutes les tâches s'exécutent dans les délais (cas des systèmes fonctionnant en temps réel) tout en minimisant l'énergie utilisée (paramètre très critique dans les systèmes portables).

Il existe trois méthodologies pour cette affectation :

- statique : les tensions sont affectées en tenant compte du cas le plus défavorable. Noter que dans un flot de données contrôlées, les opérations qui seront exécutées dépendent des différentes valeurs des conditions logiques, inconnues avant l'exécution de l'application.
- dynamique : les tensions sont affectées pendant l'exécution de l'application, au fur et à mesure qu'une tâche se termine.
- Quasi-statique :pendant la conception, une estimation (intervalle de valeurs) est faite, puis, au fur et à mesure que l'exécution d'une tâche se termine, une tension est affectée pour la tâche qui suit en fonction de l'énergie et du temps d'exécution de l'application consommés jusqu'ici

Dans certaines applications fonctionnant en temps réel, il est plus intéressant d'avoir un résultat (une image par exemple) acceptable dans les délais, plutôt que d'avoir le résultat exact en retard. Dans ce cas, on distingue, pour chaque tâche, les opérations qui doivent impérativement s'exécuter, de celles qui sont optionnelles. Le problème se généralise alors à l'affectation de tensions de sorte à maximiser le nombre d'opérations optionnelles, tout en satisfaisant les contraintes de temps et d'énergie.

Pour une tâche T_i, soient M_i et O_i les nombres de cycles pour exécuter les opérations obligatoires et optionnelles, respectivement. L'énergie consommée par T_i est :

$$E_i = C_i V_i^2 \left(M_i + O_i\right) \qquad (3.1)$$

où C_i est la capacité de charge et V_i est la tension d'alimentation affectée à T_i

Le surplus d'énergie, dû au basculement de la tension V_i à la tension V_j est donné par:

$$\varepsilon_{i,j}^{\Delta V} = C_r \left(V_i - V_j\right)^2 \qquad (3.2)$$

où C_r est la capacité du rail d'alimentation.

Le temps d'exécution d'une tâche exécutant (M_i+O_i) cycles est donné par :

$$\tau_i = k \frac{V_i}{\left(V_i - V_{th}\right)^{\alpha}} \left(M_i + O_i\right) \qquad (3.3)$$

où k est une constante dépendant de la technologie utilisée, α ($1.4 \leq \alpha \leq 2$) est un facteur de saturation de vitesse et V_{th} la tension de seuil.

Le surplus du temps d'exécution dû au basculement de la tension V_i à la tension V_j est donné par:

$$\delta_{i,j}^{\Delta V} = p\left|V_i - V_j\right| \qquad (3.4)$$

où p est une constante.

En supposant que le processeur supporte une alimentation multiple, que ces tensions d'alimentation appartiennent à $[V_{min}, V_{max}]$, la formulation du problème est la suivante :

- affectation statique :

Pour chaque tâche T_i ($1 \leq i \leq n$), trouver V_i et O_i pour le problème suivant :

$$\max \sum_{i=1}^{n} R_i\left(O_i\right) \qquad (3.5)$$

$$tel\ que : V_{\min} \leq V_i \leq V_{\max} \qquad (3.6)$$

$$s_{i+1} = t_i = s_i + \underbrace{p\left|V_{i-1} - V_i\right|}_{\delta_{i-1,i}^{\Delta V}} + \underbrace{k \frac{V_i}{\left(V_i - V_{th}\right)^{\alpha}} \left(M_i^{wc} + O_i\right)}_{\tau_i} \leq d_i \qquad (3.7)$$

$$\sum_{i=1}^{n} \left[\underbrace{C_r \left(V_{i-1} - V_i\right)^2}_{\varepsilon_{i-1,i}^{\Delta V}} + \underbrace{C_i V_i^2 \left(M_i^{wc} + O_i\right)}_{E_i} \right] \leq E_{\max} \qquad (3.8)$$

La fonction $R_i(O_i)$ est à maximiser (Eq.3.5). L'équation 3.6 montre que la tension V_i doit se trouver dans $[V_{min}, V_{max}]$. Le temps d'achèvement t_i (Eq.3.7) est la somme du temps de commencement d'exécution de T_i, du temps de basculement de voltages $\delta_{i-1,i}^{\Delta V}$, et du temps d'exécution τ_i. Les tâches doivent se terminer dans les délais impartis (Eq.3.7). L'énergie totale est la somme de l'énergie due aux basculements de tensions $\varepsilon_{i-1,i}^{\Delta V}$ et de E_i, l'énergie consommée par chaque tâche. L'énergie totale ne doit pas dépasser le budget d'énergie E_{max} alloué (Eq.3.8). Noter que cette approche utilise la valeur la plus défavorable M_i^{wc} des cycles d'exécution des opérations obligatoires.

- affectation dynamique :

Trouver V_i et O_i pour $c+1 \leq i \leq n$ optimisant le problème suivant :

$$\max \sum_{i=c+1}^{n} R_i(O_i) \quad (3.9)$$

$$tel\ que : V_{\min} \le V_i \le V_{\max} \quad (3.10)$$

$$s_{i+1} = t_{c+1} = s_{c+1} + \underbrace{p\left|V_{i-1} - V_i\right|}_{\delta^{\Delta V}_{i-1,i}} + \underbrace{k\frac{V_i}{\left(V_i - V_{th}\right)^{\alpha}}\left(M_{i} + O_i\right)}_{\tau_i} + \delta_i^{dyn} \le d_i \quad (3.11)$$

$$\sum_{i=c+1}^{n}\left[\underbrace{C_r\left(V_{i-1} - V_i\right)^2}_{\varepsilon^{\Delta V}_{i-1,i}} + \underbrace{C_i V_i^2\left(M_i + O_i\right)}_{E_i} + \varepsilon_i^{dyn}\right] \le E_{\max} - EC_c \quad (3.12)$$

L'approche dynamique exploite l'information une fois que des tâches s'achèvent et calcule en conséquence les nouvelles tensions pour maximiser la fonction objective en supposant que les tâches T_1, T_2, …, T_c se sont achevées, que l'énergie totale consommée jusqu'à l'exécution de T_c est EC_C, et que le temps d'achèvement de T_c est t_c. δ_i^{dyn} et ε_i^{dyn} sont respectivement les surplus de temps et d'énergie pour calculer de manière dynamique V_i et O_i pour chaque tâche T_i. Contrairement à l'approche statique, la valeur exacte de M_i est considérée plutôt que la valeur la plus défavorable M_i^{wc} (Eqs.3.7 & 3.8).

- affectation quasi-statique :

Elle a pour objectif de réduire le temps de résolution du problème d'optimisation pour l'approche dynamique en estimant au préalable (avant l'exécution de l'application) les valeurs des tensions et d'opérations optionnelles. Avant l'exécution d'une tâche donnée, une valeur de tension et un nombre d'opérations optionnelles sont sélectionnés à partir des intervalles d'estimation en exploitant les informations concernant le temps et l'énergie au cours de l'exécution de l'application. Le problème résolu par cette approche est le suivant :

$$\max \sum_{i=1}^{n} R_i(O_i) \quad (3.13)$$

$$tel\ que : V_{\min} \le V_i \le V_{\max} \quad (3.14)$$

$$s_{i+1} = t_i = s_i + \underbrace{p\left|V_{i-1} - V_i\right|}_{\delta^{\Delta V}_{i-1,i}} + \underbrace{k\frac{V_i}{\left(V_i - V_{th}\right)^{\alpha}}\left(M_{i} + O_i\right)}_{\tau_i} + \delta_i^{sel} \le d_i \quad (3.15)$$

$$\sum_{i=1}^{n}\left[\underbrace{C_r\left(V_{i-1} - V_i\right)^2}_{\varepsilon^{\Delta V}_{i-1,i}} + \underbrace{C_i V_i^2\left(M_i + O_i\right)}_{E_i} + \varepsilon_i^{sel}\right] \le E_{\max} \quad (3.16)$$

Noter que δ_i^{sel} et ε_i^{sel} sont respectivement le temps et l'énergie consommés par la sélection de la valeur de la tension et de celle des opérations optionnelles pour la tâche T_i.

Les trois approches qui viennent d'être présentées possèdent des avantages et des inconvénients :

- L'approche statique a pour avantage de ne consommer aucun surplus de temps et d'énergie du fait que l'affectation de tensions et du nombre d'opérations optionnelles se fait avant l'exécution de l'application concernée. Evidemment, l'inconvénient est que le nombre le plus défavorable d'opérations est considéré, exécutant ainsi le processeur plus rapidement que

nécessaire et augmentant inutilement la consommation d'énergie pour tous les cas qui ne sont pas défavorables.

- L'approche dynamique a pour but de déterminer les valeurs exactes des tensions et des opérations optionnelles du fait que l'affectation se fait au cours de l'exécution de l'application. L'inconvénient est que les surplus de temps et d'énergie nécessaires pour résoudre le problème d'optimisation sont très importants. Ceci fait que la fonction objective est moins maximisée qu'avec l'approche statique. Pire encore, les contraintes de temps et d'énergie peuvent ne pas être satisfaites.

- L'approche quasi-statique tend à éviter la considération du cas le plus défavorable (approche statique) tout en réduisant le temps et l'énergie nécessaires pour la sélection. Bien que cette approche soit intéressante, elle présente néanmoins un inconvénient concernant la qualité de la solution. En effet, cette approche opère comme indiqué dans l'exemple suivant :
  If $t_1 \leq 75\ \mu s \wedge EC_1 \leq 77\ \mu J$
  Then V_2=1.444 V ; O_2= 66924
  Else …..

  Ainsi, cette approche n'opère pas comme suit:
  If t_1= 75 μs ∧ EC_1 = 77 μJ
  Then V_2=1.444 V ; O_2= 66924
  Else …..

En conséquence, la même tension et le même nombre d'opérations optionnelles pour la tâche T_2 sont affectés lorsque l'énergie consommée par T_1 (EC_1) est égale à 77 μJ ou 1 μJ !!

Présentation de notre approche ([27]):

Du fait que les approches statiques et dynamiques ne sont pas utilisées en pratique à cause de leurs inconvénients, nous avons développé une technique qui améliore les approches quasi-statiques. Elle consiste à considérer la valeur exacte de M_i tout en réduisant le temps et l'énergie consommés par la sélection des tensions et des opérations optionnelles. Aussi, notre formulation du problème est la suivante :

$$\max \sum_{i=1}^{n} R_i(O_i) \tag{3.17}$$

$$tel\ que : V_{\min} \leq V_i \leq V_{\max} \tag{3.18}$$

$$s_{i+1} = t_i = s_i + \underbrace{p\left|V_{i-1} - V_i\right|}_{\delta^{\Delta V}_{i-1,i}} + \underbrace{k\frac{V_i}{(V_i - V_{th})^{\alpha}}(M_i + O_i)}_{\tau_i} + p1 * \delta^{sel}_{O\_T} \leq d_i \tag{3.19}$$

$$\sum_{i=1}^{n}\left[\underbrace{C_r(V_{i-1} - V_i)^2}_{\varepsilon^{\Delta V}_{i-1,i}} + \underbrace{C_i V_i^2 (M_i + O_i)}_{E_i}\right] + p1 * \varepsilon^{sel}_{O\_T} \leq E_{\max} \tag{3.20}$$

$$p1 = \begin{cases} 1 & \text{si } i=1 \\ 0 & \text{sinon} \end{cases}$$

Comparativement à l'approche quasi-statique où la sélection se fait *Nb_tasks* fois (*Nb_tasks* est le nombre de tâches), la sélection se fait une seule fois (Eqs.3.19 & 3.20) dès l'achèvement de la

première tâche tout en considérant la valeur exacte de M_i (avantage de l'approche dynamique). Avant de détailler notre approche, nous énumérons les quatre techniques dans la table 3.3.

Table 3.3. Sommaire de techniques d'affectation

Technique	**Nombre d'opérations obligaoires (M_i)**	**Surplus de temps**	**Surplus d'énergie**
Statique	$= M_i^{wc}$	0	0
Dynamique	$= M_i^{ex} \le M_i^{Q-S} \le M_i^{wc}$	$\sum_{i=1}^{Nb\_tasks} \delta_i^{dyn}$	$\sum_{i=1}^{Nb\_tasks} \varepsilon_i^{dyn}$
Précédentes Quasi-statiques	$= M_i^{Q-S} \le M_i^{wc}$	$Nb\_tasks * \delta^{sel}$	$Nb\_tasks * \varepsilon^{sel}$
La nôtre (O_T)	$= M_i^{ex} \le M_i^{Q-S} \le M_i^{wc}$	$1 * \delta_{O\_T}^{sel}$	$1 * \varepsilon_{O\_T}^{sel}$

Une application étant un ensemble de tâches, une tâche T_i étant un ensemble d'opérations $O_{i,1}$, O_{i2}, …, O_{i,n_i}, considérons la figure 3.13 représentant un flot de données contrôlées.

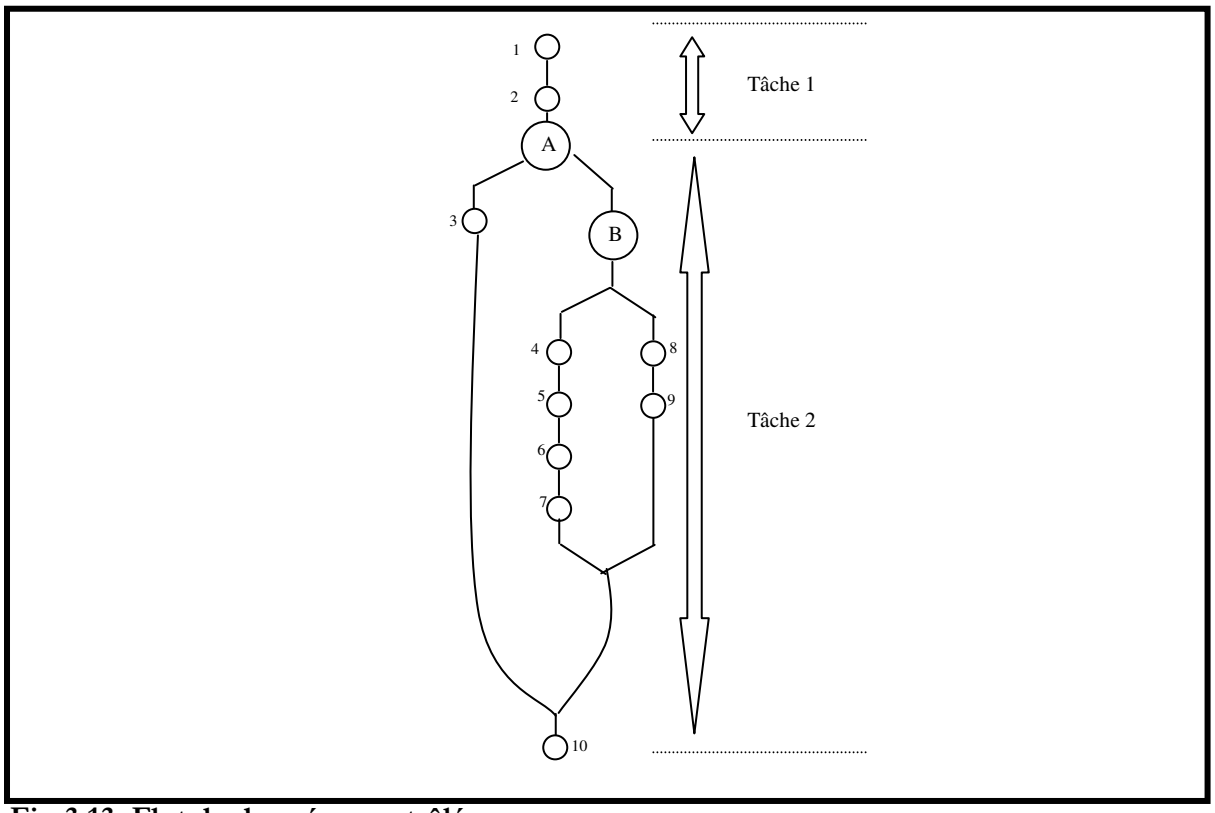

Fig.3.13. Flot de données contrôlées

Dans la figure 3.13, il y'a 10 opérations obligatoires et 2 variables logiques A et B. Du fait que les opérations qui seront réellement exécutées ne sont pas connues avant l'exécution de l'application, les techniques statiques affectent à M_i la valeur 7 qui est la plus défavorable (opérations 1, 2, 4, 5, 6, 7 et 10). Noter que dans le cas où la condition *A* est vraie, le nombre réel d'opérations qui seront exécutées n'est que 4 (opérations 1, 2, 3 et 10). Pour des applications réelles, ces 2 nombres

peuvent être très différents, ce qui entraînerait une mauvaise optimisation de la fonction objective. Afin d'éviter ceci, nous éclatons le flot de données contrôlées en un ensemble de traces, tel qu'indiqué dans la figure 3.14. Noter que les opérations optionnelles ne sont pas montrées dans les figures 3.13 et 3.14.

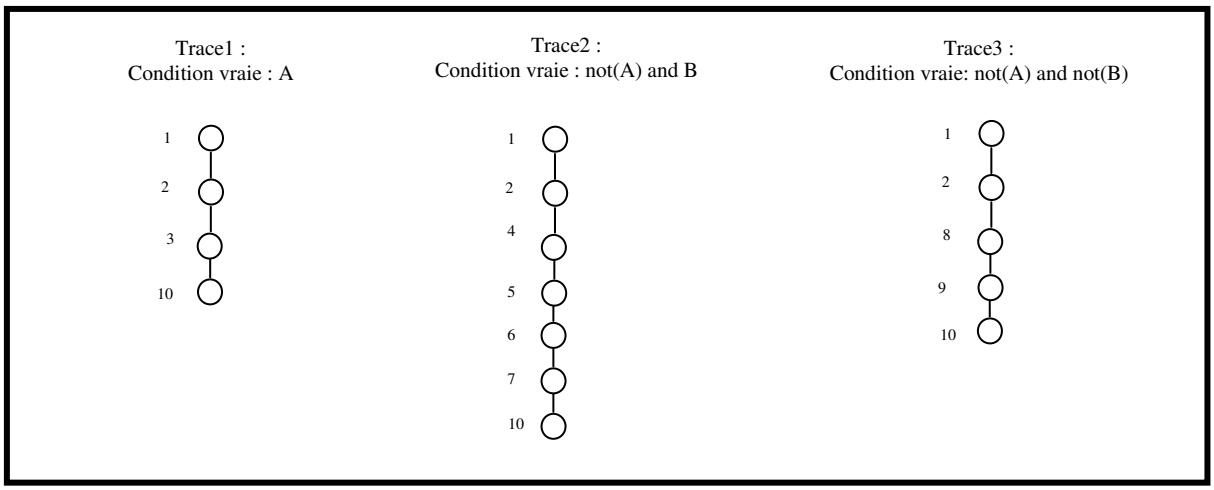

Fig.3.14. Traces contenues dans le flot de données contrôlées de la Fig.3.13

Supposons que les tensions affectées aux opérations de la figure 3.14 pour maximiser la fonction objective tout en satisfaisant les différentes contraintes sont celles indiquées dans la table 3.4.

Table 3.4. Exemple d'affectation de tensions à des opérations

Tâche	Opération	Tension (V) dans Trace 1	Trace 2	Trace 3
	1	1.0	1.0	3.3
1	2	1.0	1.0	1.0
	3	1.0		
	4		3.3	
	5		3.3	
2	6		1.0	
	7		3.3	
	8			3.3
	9			3.3
	10	1.0	3.3	1.0

Affecter des tensions différentes aux opérations de la même tâche pourrait augmenter les surplus d'énergie et de temps dus à des basculements excessifs de tensions (Eqs.3.2 & 3.4). Il est alors préférable d'affecter la même tension à de telles opérations. La tension que nous appliquons à chaque tâche lorsque la trace t est exécutée est alors :

$$V_{i,t} = \frac{1}{Nb_{Op\_i,t}} \sum_{j=1}^{Nb_{Op\_i,t}} V_{i,t,j} \quad (3.21)$$

où :
- $V_{i,t,j}$ est la tension affectée à l'opération j de la tâche i lorsque la trace t est exécutée.
- $Nb_{Op\_i,t}$ est le nombre d'opérations de la tâche i dans la trace t

En appliquant l'équation 3.21, les tensions moyennes affectées à chaque tâche lorsqu'une certaine trace est exécutée sont alors celles indiquées dans la table 3.5.

Table 3.5. Exemple d'affectation de tensions à des tâches

	Tension (V) dans		
Tâche	**Trace 1**	**Trace 2**	**Trace 3**
1	1.0	1.0	2.15
2	1.0	2.84	2.53

Il est à noter que l'approche conduisant au résultat le plus optimal est celle qui consiste à déterminer, pour chaque trace, l'affectation de tensions aux différentes tâches de sorte que le nombre d'opérations optionnelles soit maximisé, tout en satisfaisant les différentes contraintes de temps et d'énergie. La table 3.6 est un exemple d'une telle affectation.

Table 3.6. Affectation optimale de tensions et d'opérations optionnelles

	Trace 1					**Trace 2**					**Trace 3**				
Task	M_i	V_i	O_i	E_i (pJ)	D_i (ns)	M_i	V_i	O_i	E_i (pJ)	D_i (ns)	M_i	V_i	O_i	E_i (pJ)	D_i (ns)
1	100	1.13	90	9.73	4.04	120	1.11	95	11.01	4.57	80	1.12	41	6.20	2.57
2	80	1.12	20	5.09	6.18	65	1.14	17	4.17	6.32	70	1.13	17	4.43	4.43
3	60	1.13	30	4.59	8.10	62	1.13	30	4.69	8.29	61	1.13	30	4.64	6.37
4	540	1.15	236	39.18	24.74	516	1.13	201	36.21	23.66	520	1.12	210	36.86	22.03
5	75	1.12	20	4.83	26.77	200	1.12	80	14.24	29.64	150	1.13	45	9.92	26.19
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

Quoique la trace qui sera réellement exécutée ne sera connue que pendant l'exécution de l'application, nous avons développé une technique qui détecte la trace qui est en cours d'exécution et pour laquelle l'affectation V/O optimale (celle déterminée pendant la conception) peut être exploitée en conséquence. Toutefois, ceci n'est pas possible pour la première tâche. Pour l'exemple de la table 3.6, l'affectation V/O à la première tâche devrait être 1.13/90, 1.11/95 ou 1.12/41 ? Aussi, nous exploitons l'affectation V/O optimale à l'exception de la première tâche pour laquelle une autre affectation V/O est appliquée, et à l'exception de quelques opérations optionnelles pour quelques tâches.

Concernant la première tâche, et du fait qu'il n'est pas encore possible de détecter la trace en cours d'exécution, la tension qui lui est affectée est la moyenne de ses tensions en considérant toutes les traces, c'est-à-dire :

$$V_1 = \frac{1}{Nb\_traces} \sum_{j=1}^{Nb\_traces} V_{1,j} \quad (3.22)$$

Toutefois, dès l'achèvement de la première tâche, notre technique permet de déterminer la trace en cours d'exécution, et d'exploiter l'affectation V/O optimale en conséquence. Ceci est fait moyennant la procédure *détecte_trace()* présentée en annexe.

Soit $V_{1,t}$ la tension optimale affectée à la première tâche lorsqu'une certaine trace *t* est en cours d'exécution. Tenant compte du fait que l'affectation V/O optimale a été obtenue avec $V_{1,t}$ plutôt qu'avec V_1 (Eq. 3.22), les contraintes de temps et d'énergie peuvent être violées en utilisant V_1 au lieu de $V_{1,t}$. Afin d'exploiter au maximum l'affectation V/O optimale tout en satisfaisant les différentes contraintes, l'affectation de nombres d'opérations optionnelles doit être mise à jour pour certaines tâches. A partir des équations (3.1) et (3.3), nous avons ce qui suit :

$$E_{1,t} = C_1 V_{1,t}^2 \left(M_{1,t} + O_{1,t} \right) \quad (3.23)$$

$$\tau_{1,t} = k \frac{V_{1,t}}{\left(V_{1,t} - V_{th} \right)^{\alpha}} \left(M_{1,t} + O_{1,t} \right) \quad (3.24)$$

$$E_1 = C_1 V_1^2 \left(M_{1,t} + O_1 \right) \quad (3.25)$$

$$\tau_1 = k \frac{V_1}{\left(V_1 - V_{th} \right)^{\alpha}} \left(M_{1,t} + O_1 \right) \quad (3.26)$$

Afin de maintenir satisfaites les contraintes d'énergie et de temps, nous devons établir :

$$E_1 \le E_{1,t} \wedge \tau_1 \le \tau_{1,t} \quad (3.27)$$

A partir des équations (3.23), (3.24), (3.25), (3.26) et (3.27), nous obtenons :

$$O_{1,1,t} \le \left\lfloor \frac{V_{1,t}^2}{V_1^2} \left(M_{1,t} + O_{1,t} \right) - M_{1,t} \right\rfloor \quad (3.28)$$

et

$$O_{1,2,t} \le \left\lfloor \frac{V_{1,t}}{V_1} \frac{\left(V_1 - V_{th} \right)^{\alpha}}{\left(V_{1,t} - V_{th} \right)^{\alpha}} \left(M_{1,t} + O_{1,t} \right) - M_{1,t} \right\rfloor \quad (3.29)$$

où $E_{1,t}$, $\tau_{1,t}$ et $O_{1,t}$ sont obtenus avec l'affectation V/O optimale et correspondent à l'énergie, le temps et le nombre d'opérations optionnelles associés à la première tâche pour une trace *t* donnée. Du fait qu'on veut maximiser la fonction objective, le signe ≤ utilisé dans les équations (3.28) et (3.29) doit être remplacé par le signe =.
Du fait que certaines tensions $V_{1,t}$ sont supérieures à V_1 (Eq.3.22), nous devons avoir :
$O_{1,1,t} \le O_{1,\max,t}$ et $O_{1,2,t} \le O_{1,\max,t}$,où $O_{1,\max,t}$ est la valeur maximale des opérations optionnelles de la première tâche lorsque la trace t est exécutée. Aussi, afin de maintenir satisfaites les contraintes de temps et d'énergie, l'affectation V/O correspondant à la première tâche est la suivante :

V_1 est tel que défini par l'équation (3.22)

$$O'_{1,t} = \min\left(O_{1,1,t}, O_{1,2,t}, O_{1,\max,t} \right); 1 \le t \le Nb\_traces \quad (3.30)$$

Dans le cas où les contraintes de temps et d'énergie sont satisfaites, notre méthode donne la même affectation V/O qu'avec l'approche idéale, à l'exception de la première tâche si des tensions différentes devaient être affectées à la première tâche (Eqs.3.22 et 3.30). Si celles-ci sont égales, les équations (3.28) et (3.29) montrent que notre méthode donne le nombre maximal d'opérations optionnelles. Plus formellement, si :

i) Si $E_{\max} \geq E_{I\_C,t} + \varepsilon_{O\_T}^{sel} \wedge V_{1,r} = V_{1,t} \; \forall 1 \leq r,t \leq Nb\_traces$, nous obtenons :

$$R_{1,1} = R_{I\_C,t} - R_{O\_T,t} = \left(O_{1,t} + \sum_{i=2}^{N} R_{I\_C,i,t} \right) - \left(O_{O\_T,1,t} + \sum_{i=2}^{N} O_{O\_T,i,t} \right) = 0$$

où *I_C*, *O_T*, *i* et *t* dénotent le cas idéal, notre technique, la tâche i et la trace t, respectivement. $R_{I\_C,t}$ et $R_{O\_T,t}$ sont les nombres d'opérations optionnelles obtenues pour la trace t par la technique idéale et la nôtre, respectivement. L'erreur relative $Rel\_Err_{11}=R_{1,1}/R_{I\_C,t}$ est alors :

$$\text{Rel\_Err}_{11} = 0\ \% \qquad (3.31)$$

ii) Si $\exists t \neq r : V_{1,r} \neq V_{1,t}\ 1 \leq r,t \leq Nb\_traces \ \wedge \ E_{\max} \geq E_{I\_C,t} + \varepsilon_{O\_T}^{sel}$, nous obtenons :

$$\begin{aligned} R_{1,2} &= R_{I\_C} - R_{O\_T} \\ &= \max_t \left(O_{I\_C,1,t} + \sum_{i=2}^{N} O_{I\_C,i,t} \right) - \left(O_{O\_T,1,t} + \sum_{i=2}^{N} O_{O\_T,i,t} \right) \\ &= \max_t \left(O_{I\_C,1,t} - O_{O\_T,1,t} \right) \\ &= \max_{1 \leq t \leq Nb\_traces} \left(\left(O_{I\_C,1,t} + M_{1,t} \right) * \left(1 - \frac{V_{1,t}^2}{V_1^2} \right), \left(O_{I\_C,1,t} + M_{1,t} \right) * \left(1 - \frac{V_{1,t}}{V_1} \frac{\left(V_1 - V_{th} \right)^{\alpha}}{\left(V_{1,t} - V_{th} \right)^{\alpha}} \right) \right) \end{aligned}$$

L'erreur relative $Rel\_Err_{12}=R_{1,2}/R_{I\_C,t}$ est alors :

$$\text{Rel\_Err}_{12} = 100 * \text{R}_{12} / \text{R}_{\text{I\_C}}\ \% \qquad (3.32)$$

Il est à remarquer que si $V_{1,t}$ est très proche de V_1, alors l'erreur Rel_Err_{12} est très proche de 0%. Le cas le plus défavorable correspond à celui où V_{1t} est très proche de V_{min} et V_1 très proche de V_{max}. Mais du fait que V_1 est une tension *moyenne* (Eq.3.22), Rel_Err_{12} sera plutôt une erreur intéressante.

Dans le cas où le budget d'énergie alloué est égal à la quantité d'énergie consommée par la technique idéale (c'est-à-dire $E_{max}=E_{I\_C,t}$), certaines opérations optionnelles ne pourront pas être exécutées en appliquant notre technique du fait que les surplus d'énergie et de temps nécessaires pour la sélection de V/O ne sont pas nuls. En notant que ces surplus sont exploités par la technique idéale pour exécuter un certain nombre d'opérations optionnelles, déterminons ce nombre d'opérations que notre technique empêche de s'exécuter. Du fait que la ième opération s'exécute avant la jème (avec i < j), et en considérant les équations 3.1 & 3.3, N_L opérations ne pourraient s'exécuter :

$$N_L = \left\lceil \max_{1 \leq t \leq Nb\_traces} \left(\frac{\varepsilon_{O\_T}^{sel}}{E_{I\_C,O,N,t}} O_{I\_C,N,t}, \frac{\delta_{O\_T}^{sel}}{\tau_{I\_C,O,N,t}} O_{I\_C,N,t} \right) \right\rceil \qquad (3.33)$$

où N est le nombre de tâches, $E_{I\_C,O,N,t}$ et $\tau_{I\_C,O,N,t}$ sont respectivement l'énergie et le temps nécessaires pour exécuter la partie optionnelle de la dernière tâche.

- Cas 1 : si $\frac{\varepsilon_{O\_T}^{sel}}{E_{I\_C,O,N,t}} \le 1 \wedge \frac{\delta_{O\_T}^{sel}}{\tau_{I\_C,O,N,t}} \le 1$, le nombre d'opérations optionnelles que notre technique permet de s'exécuter pour la dernière tâche est :

$$O_{O\_T,N,t} = \min\left(\left(1-\frac{\varepsilon_{O\_T}^{sel}}{E_{I\_C,O,N,t}}\right)*O_{I\_C,N,t}, \left(1-\frac{\delta_{O\_T}^{sel}}{\tau_{I\_C,O,N,t}}\right)*O_{I\_C,N,t}\right)$$

- Cas 2 : si $\frac{\varepsilon_{O\_T}^{sel}}{E_{I\_C,O,N,t}} > 1 \vee \frac{\delta_{O\_T}^{sel}}{\tau_{I\_C,O,N,t}} > 1$, les N_L opérations qui ne pourront être exécutées sont distribuées à travers les tâches j, j+1, ..., N ($1 < j \le N$) telles que :

$$N_L + \sum_{k=j}^{N} O_{O\_T,k,t} = \sum_{k=j}^{N} O_{I\_C,k,t}$$

Pour les deux cas nous avons :

iii) Si notre affectation V/O pendant la phase de conception donne des tensions égales pour la première tâche, nous avons $O_{1,t,O\_T} = O_{1,t,I\_C}$, et la différence des nombres d'opérations optionnelles permises par la technique idéale et la nôtre est alors : $R_{21}=R_{I\_C} - R_{O\_T} = N_L$.
L'erreur relative serait alors :

$$Rel\_Err_{21} = 100*R_{21} / R_{I\_C} \% = \max_{1\le t\le Nb\_traces} \frac{100*N_L}{\sum_{i=1}^{N} O_{I\_C,i,t}} \% \quad (3.34)$$

iv) si $\exists t \ne r \; V_{1,r} \ne V_{1,t} \;\; 1 \le r,t \le Nb\_traces \wedge E_{\max} = E_{I\_C,t}$, nous obtenons :

$$R_{22} = R_{I\_C} - R_{O\_T} = \max_{1\le t\le Nb\_traces} \begin{pmatrix} \left(1-\frac{V_{1,t}^2}{V_1^2}\right)*\left(O_{1,t,I\_C}+M_{1,t}\right)+N_L, \\ \left(1-\frac{V_{1,t}*\left(V_1-V_{th}\right)^{\alpha}}{V_1*\left(V_{1,t}-V_{th}\right)^{\alpha}}\right)*\left(O_{1,t,I\_C}+M_{1,t}\right)+N_L \end{pmatrix}$$

L'erreur relative serait alors :

$$Rel\_Err_{22}=100*R_{22} / R_{I\_C} \% \quad (3.35)$$

Ainsi, l'algorithme général de notre méthode est celui indiqué dans la figure 3.15.

Algorithme général – V/O affectation
{
// Affectation V/O durant la phase de conception
Pour chaque trace
Faire pour chaque tâche i
Faire Déterminer V_i/O_i *optimisant* $R(O_i)$ *en tenant compte des contraintes indiquées dans (3.18), (3.19) et (3.20) ;*
ffaire
ffaire
// Affectation V/O durant la phase d'exécution
V_1 *est tel que défini dans l'équation (3.22));* // $V_{r,1}=V_{s,1}$ $\forall$ *traces r et s*
$O'_{1,t}$ *est défini par les équations (3.28), (3.29) et (3.30).*
Détermine_trace_t() ; //Déterminer la trace t qui est en cours d'exécution
$V_{i,t}$ *sont les tensions déterminées durant la phase de conception*
// i=2, 3, ..., N ; t=1, 2, ..., Nb_traces
si $E_{max} \geq E_{I\_C,t}+\varepsilon^{sel}$
alors $O_{i,t}$ *sont les valeurs déterminées durant la phase de conception*
// i=2, 3, ..., N ; t=1, 2, ..., Nb_traces
Sinon {Déterminer N_L *(Eq.3.33);*
si $\dfrac{\varepsilon^{sel}_{O\_T}}{E_{I\_C,O,N,t}} \leq 1 \quad \wedge \quad \dfrac{\delta^{sel}_{O\_T}}{\tau_{I\_C,O,N,t}} \leq 1,$
alors {$O_{i,t}$ *sont les valeurs déterminées durant la phase de conception*
// i=2, 3, ..., N-1 ; t=1, 2, ..., Nb_traces
$O_{N,t}=O_{N,t}-N_L$;
// la valeur initiale de $O_{N,t}$ *est le nombre maximal d'opérations optionnelles de la dernière tâche*
// pour une trace t donnée
}
sinon { $O_{i,t}$ *sont les valeurs déterminées durant la phase de conception*
// i=2, 3, ..., j-1 ; t=1, 2, ..., Nb_traces
$O_{j,t} = O_{j,t} - N_L$; $O_{i,t}=0$; *// i= j+1, ..., N ; t=1, 2, ..., Nb_traces*
// j vérifie : $N_L + \sum_{k=j}^{N} O_{O\_T,k,t} = \sum_{k=j}^{N} O_{I\_C,k,t}$
}
fsi
}
fsi
}

Fig.3.15. Algorithme général de notre méthode d'affectation V/O

L'affectation V/O durant la phase de conception ainsi que la procédure détectant la trace qui est en cours d'exécution (pour l'exploitation de l'affectation V/O optimale) sont présentées en annexe. Notons que la technique idéale n'existe pas en pratique. Elle a été formulée uniquement pour des fins de comparaison avec notre méthode. Les quatre erreurs relatives déterminées de manière formelle montrent l'efficacité de notre méthode, laquelle efficacité est confirmée par les résultats obtenus. Enfin, du fait que la technique idéale n'existe pas, notre technique se présente comme une bonne candidate pour ce problème d'affectation.

Intéressons-nous maintenant au système dans sa globalité. Transformons les tâches indiquées dans la figure 3.12 en opérations (figure 3.16). Dans cette figure, il est montré que

pendant que le premier processeur exécute l'opération 1 de la tâche 1, le troisième ASIC exécute de manière concurrente les opérations $O_{138,1}$ et $O_{138,3}$ de la tâche 138. Ceci nous amène à transformer les équations (3.1) et (3.3) de manière à ce que le temps et l'énergie consommés par une tâche soient correctement déterminés lorsque cette tâche est affectée à un ASIC. Il faudrait alors :

- utiliser (3.1) pour déterminer l'énergie consommée par une tâche qu'elle soit affectée à un processeur ou à un ASIC (pour un ASIC, le nombre d'opérations est la somme de toutes les opérations opérant durant le même cycle d'exécution)
- utiliser respectivement (3.3) ou (3.36) pour déterminer le temps d'exécution d'une tâche affectée à un processeur ou un ASIC

$$\tau_i = \sum_{j=1}^{N_i} \max_{1 \le l \le n_{i,j}} t_{i,j,l} \quad (3.36)$$

où N_i est le nombre de cycles d'exécution de la $i^{ème}$ tâche et n_{ij} est le nombre d'opérations de la tâche i qui sont ordonnancées dans le $j^{ème}$ cycle d'exécution.

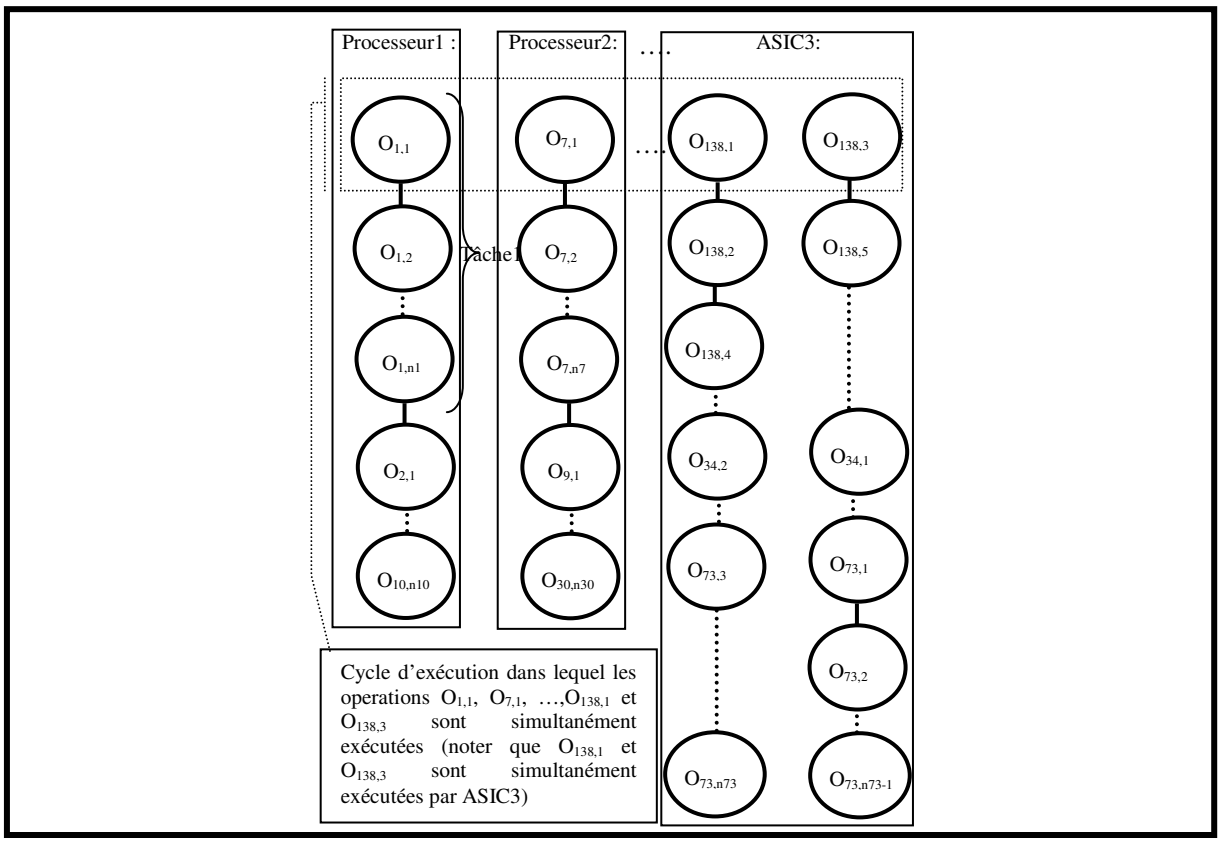

Fig.3.16. Ensemble de traces concurrentes

$t_{i,j,l}$ est la durée d'exécution de la $l^{ème}$ opération de la tâche i lorsqu'elle est ordonnancée dans le $j^{ème}$ cycle :

$$t_{i,j,l} = k \frac{V_{i,l}}{\left(V_{i,l} - V_{th}\right)^{\alpha}} \quad (3.37)$$

où $V_{i,l}$ est la tension affectée à la $l^{ème}$ opération de la tâche i. V_{th}, k et α sont tels que définis précédemment.

Notre affectation V/O concerne un système dont l'architecture est celle indiquée à la figure 3.11. Les opérations sont le plus souvent traitées localement, mais contrairement à un système constitué d'un seul processeur ou d'un seul ASIC, d'autres types de durée d'exécution et de consommation d'énergie sont à considérer :

- transfert de données dans le même sous-système
- communications à travers le bus global
- opérations de lecture/écriture (mémoires locales et globale)

Les algorithmes utilisés pour l'affectation V/O dans le cas d'un seul processeur sont alors utilisés pour le système global en remplaçant l'équation (3.3) par l'équation (3.36) lorsqu'il s'agit d'un ASIC (plutôt qu'un processeur) et en tenant compte des types de durée d'exécution et d'énergie supplémentaires. Ces types viennent d'être énumérés et seront développés dans la partie *analyse* de ce chapitre.

3.4.3. Protocole de communications entre les sous-systèmes

Les protocoles de communications sont décrits dans un premier temps de manière abstraite et ne sont affinés qu'ultérieurement. En effet, il serait illusoire de détailler un protocole qui de prime abord présente des problèmes. Au niveau d'abstraction *système*, le langage SYSTEMC est tout indiqué pour la description des communications entre les différents modules du système à concevoir. Ainsi, le protocole de communications abstrait entre les modules SOURCE et SINK de la figure 3.17 se décrit de manière simple comme suit :

Fig.3.17. Exemple de communication abstraite

```
SC_MODULE(source)
{ // output port
  sc_port<sc_fifo_write_if<int>> output ;

  // thread process
  void main_action()
  {while(true)
    {……
     output->write(data);   // écriture d'une donnée sur un port
    }
   }


   // constructeur
   SC_CTOR(source)
   {SC_THREAD(main_action);
   }
};
```

```
SC_MODULE(sink)
{ // input port
  sc_port<sc_fifo_read_if<int>> input ;

  // thread process
  void main_action()
  {while(true)
    …..
   data=input->read() ;
  ….
    }
}
// constructor
SC_CTOR(sink)
{SC_THREAD(main_action);
}
};

int sc_main(int argc, char *argv[])
{
  // instantiation du canal
  sc_fifo<int> a_fifo[10] ;
  // instantiation des modules
  source SOURCE;
  sink SINK ;
  // connection des modules
  SOURCE.output(a_fifo);
  SINK.input(a_fifo);
  …..
}
```

Une fois la vérification comportementale validée, le protocole peut être affiné. Il s'agit alors de remplacer le support de communications (fils) par un protocole défini soit par l'utilisateur, soit par un protocole de la bibliothèque de SYSTEMC. Supposons que l'on choisisse le protocole *handshaking*. Le canal *a_fifo* sera alors remplacé par le module suivant :

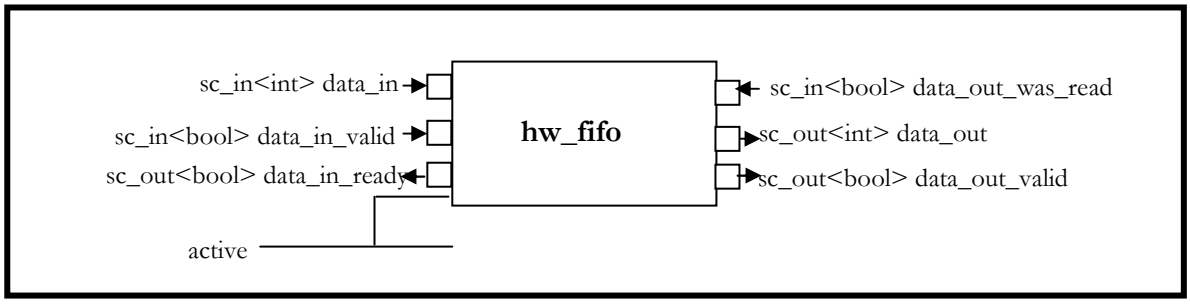

Fig.3.18. Protocole *handshaking*

La communication sera alors affinée par étapes successives :

- **Affinement du canal :**

On remplace sc_fifo par hw_fifo dans la description précédente.

```
int sc_main(int argc, char *argv[ ])
{hw_fifo HW_FIFO ;
  …. }
```

- **Insertion d'un adaptateur :**

Les modules SOURCE et SINK ne pouvant communiquer directement avec *hw_fifo*, il y'a lieu d'insérer un adaptateur entre *hw_fifo* et chacun des deux modules (Figure 3.19). Les différents ports des différents modules sont les suivants :
a: data_port ; b: valid_port ; c: ready_port ; d: data_in ; e : data_in_valid ; f : data_in_ready ; g : data_out_was_read ; h : data_out ; i : data_out_valid ; j : was_ready_port; k: data_out_port; l:data_out_port;

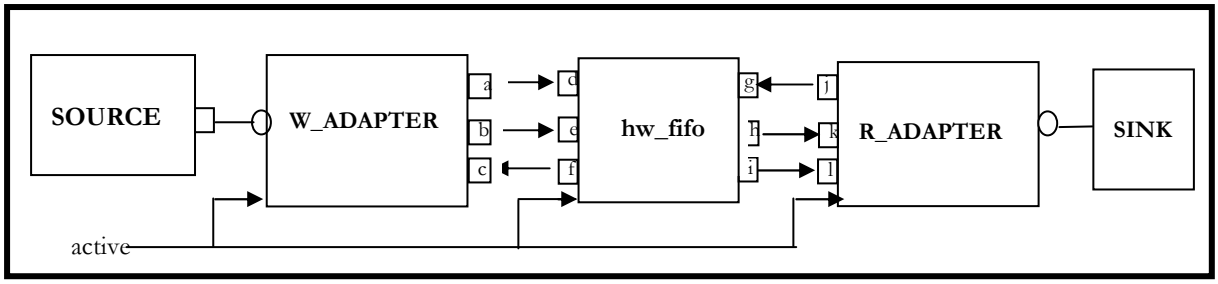

Fig.3.19. Communication affinée

L'adaptateur *w*_adapter montré ci-après implémente l'interface requise par le module SOURCE.

```
class w_adapter :public sc_module, sc_fifo_write_if<int>
{public:
  //Ports
  sc_out<int> data_port;
  sc_out<bool> valid_port;
  sc_in<bool> ready_port;
  sc_in_clk active;

  // Implémentation de write()
  virtual void write(const int& data)
  {// attendre jusqu'à ce que fifo soit prêt
    while(ready_port == false)
        wait(active.pos_edge()) ;
     data_port=data;
     valid_port=true;
      // attendre le prochain cycle d'horloge
      wait(active.pos_edge()) ;
      valid_port=false ;
  }
  ……..
};
```

Instructions à mettre à la place de

output->write(data)

dans void source ::main_action()

dans la description de la communication

abstraite

- **Instantiation de l'adaptateur :**

```
int sc_main(int argc, char* argv[ ])
{
  // Instantiation du canal
  source SOURCE ;
  sink SINK ;
  // Instantiation des adaptateurs
  w_adapter W_ADAPTER ;
   r_adapter R_ADAPTER ;     // définir la classe r_adapter de la même manière que pour
                                    // la classe w_adapter : définition des ports et de la fct read()
   // connecter les modules SOURCE et W_ADAPTER
   SOURCE.output(W_ADAPTER) ;
   // connecter W_ADAPTER à hw_fifo
    sc_signal<int> data_in ;
    sc_signal<bool> data_in_valid;
    sc_signal<bool> data_in_ready;
    ……. //  signaux nécessaires  pour l'instantiation avec R_ADAPTER
   W_ADAPTER.data_port(data_in);
   W_ADAPTER.valid_port(data_in_valid);
   W_ADAPTER.ready_port(data_in_ready);
   hw_fifo.data_in(data_in);
   hw_fifo.data_in_valid(data_in_valid);
   hw_fifo.data_in_ready(data_in_ready);
   ..... // connecter les modules hw_fifo et R_ADAPTER
   ..... // connecter R_ADAPTER et SINK
}
```

- **Affinement de l'interface :**

Il s'agit de fusionner les modules W_ADAPTER et SOURCE en un seul module (REFINED_SOURCE –voir Fig.3.20-) et les modules R_ADAPTER et SINK en un seul module (REFINED_SINK). Ainsi, la communication entre les modules REFINED_SOURCE et REFINED_SINK à travers le protocole de communications *handshaking* sera complètement définie.

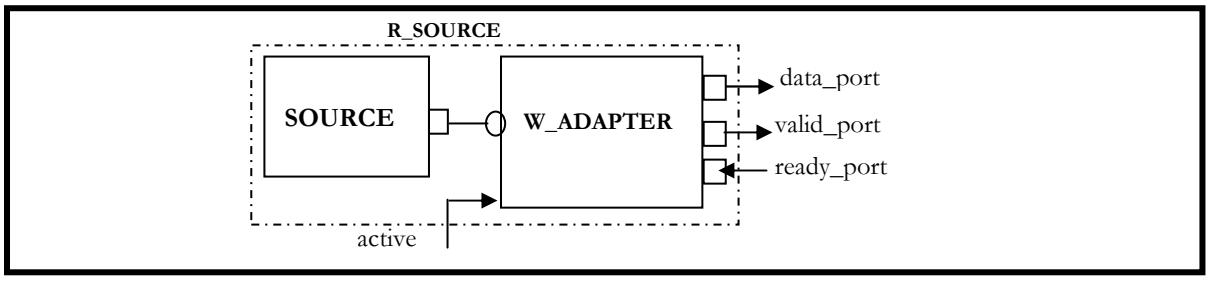

Fig.3.20. Affinement d'interface

Nous obtenons alors :

```
SC_MODULE(refined_source)
{
  // Port de sortie
  sc_out<int> data_port ;          ⎫  à la place de sc_port<sc_fifo_write_if<int>> output
  sc_out<bool> valid_port ;        ⎪
  sc_in<bool> ready_port;          ⎭  dans la définition de la communication abstraite
```

```
 sc_in_clk active;
 // Thread process
 void main_action()
 {
  while(true)
  {
    ……
   write(data);
    …..
  }
 }
  void write(const int &data)
  {
   // wait until fifo is ready
   while(ready_port == false)
       wait(active.pos_edge());
       ……..
       valid_port=false;
  }
   SC_CTOR(refined_source)
   {
    SC_THREAD(main_action);
   }
}
```

à la place de output->write(data) et de

virtual void write(const int & data)

```
int sc_main(int argc, char *argv[ ])
{
 // instantiation du canal
 hw_fifo HW_FIFO;
 // instantiation des modules affinés SOURCE et SINK
 refined_source R_SOURCE ;
 refined_sink R_SINK ;  // Le module refined_sink doit être défini de manière identique
                        //  à celle du module refined_source
 .....
 // Connecter R_SOURCE et hw_fifo
 sc_signal<int> data_in ;
 sc_signal<bool> data_in_valid;
 sc_signal<bool> data_in_ready;
 R_SOURCE.data_port(data_in);
 R_SOURCE.valid_port(data_in_valid);
 R_SOURCE.ready_port(data_in_ready);
 hw_fifo.data_in(data_in);
 hw_fifo.data_in_valid(data_in_valid);
 hw_fifo.data_in_ready(data_in_ready);
 .....  // Même traitement concernant le module R_SINK
}
```

Ainsi, comme nous venons de le voir, n'importe quelle communication entre deux ports peut être décrite de manière abstraite puis affinée en fonction du protocole de communications

utilisé. Pour notre part, nous avons utilisé le protocole *handshaking* que nous trouvons plus complet par rapport à d'autres. Toutefois, le traitement que nous venons de voir ne permet que de remplacer un fil de communications reliant deux points par une description de communications plus complète. De plus, l'utilisation de trois modules (module affiné-source, module du protocole, module affiné-destination) pour chaque communication point à point (très nombreuses dans un système monopuce *SOC*) entraînerait un certain nombre d'inconvénients ayant trait à la surface, le temps de transfert et la dissipation de puissance. Pour ce faire, nous nous sommes limités à cette modélisation uniquement pour les communications (peu nombreuses) entre des modules appartenant à différents sous-systèmes, tout en évitant l'utilisation simultanée du bus global par deux ou plusieurs transferts de données (conflits gérés par les parties de contrôle concernées –détails dans le prochain chapitre-). Pour les communications locales, nous avons développé une autre interface de communications qui sera aussi présentée dans le prochain chapitre. Concernant les communications entre les différents sous-systèmes, le principe est le suivant :

G=(V,E) // Graphe des tâches
P={partitions p_i} ; // Ensemble des sous-systèmes obtenus par partitionnement
Pour tout (v_i , v_j) ∈ E tel que $v_i \in p_k \wedge v_j \in p_l$, $k \neq l$
Faire {affiner le module dont la tâche i émet des données ; // fusion avec l'adaptateur comme vu
utiliser le module hw_fifo ; // hw_fifo est décrit précédemment
affiner le module dont la tâche j reçoit des données ; // fusion avec l'adaptateur comme vu
connecter les trois modules et utiliser write() et read() ;
}
Ffaire

Fig.3.21. Algorithme général du protocole des communications entre les différents sous-systèmes

Les trois modules sont activés sur le front montant du signal booléen *active*. Celui-ci émane de la partie de contrôle des sous-systèmes concernés par la communication, et nous verrons dans le prochain chapitre comment sont générés les différents signaux de commande.

3.5. Analyse :

3.5.1. Surface

Au niveau d'abstraction *système*, le paramètre *surface* ne peut être estimé qu'en termes de nombre de sous-systèmes et de la taille de leur protocole de communications. Ceci est dû au fait que les détails ne sont connus qu'au fur et à mesure que l'on progresse dans la conception à travers les différents niveaux d'abstraction. Aussi, s'il est jugé que la configuration courante conduira à une surface coûteuse, certaines arêtes dans le graphe des tâches (Fig.3.8) pourraient être supprimées pour forcer la synthèse à mettre certaines tâches dans le même sous-système, ce qui conduirait à une réduction simultanée du nombre de sous-systèmes et de la taille du protocole de communications, mais avec des performances moins intéressantes (parallélisme réduit). Evidemment, les arêtes à supprimer doivent concerner les tâches qui ont le plus fort degré de communications dans la configuration courante de l'architecture. Ceci aurait pour effet de libérer beaucoup plus le bus global au détriment de communications locales plus nombreuses.

3.5.2. Vitesse

Elle peut être estimée en considérant les différents délais suivants :

- temps de transfert de données à travers un bus local :

$$\tau_{i,LB,t} = pr_{i,L} * (t_{LB} * Nb\_data_i) \quad (3.38)$$

où $pr_{i,L} = \begin{cases} 1 \, si \, la \, tâche \, i \, transmet \, (reçoit) \, des \, données \, sur \, le \, (à \, partir \, du) bus \, local \\ \quad aux \, (de) \, registres \, locaux \, ou \, à \, des \, (de) \, mémoires \, locales \\ 0 \, sinon \end{cases}$

t_{LB} est le temps de transfert d'une donnée à travers le bus local
Nb_data_i est le nombre de données transmises par la tâche i

- temps de transfert de données à travers le bus global :

$$\tau_{i,GB,t} = pr_{i,G} * \left(t_{GB} * Nb\_data_i \right) \quad (3.39)$$

où $pr_{i,G} = \begin{cases} 1 \, si \, la \, tâche \, i \, transmet \, (reçoit) \, des \, données \, sur \, le \, (à \, partir \, du) bus \, global \\ \quad à \, des \, (de) \, registres \, externes \, ou \, à \, (de) \, la \, mémoire \, globale \\ 0 \, sinon \end{cases}$

t_{GB} est le temps de transfert d'une donnée à travers le bus global

- Temps de lecture/écriture d'une donnée en mémoire locale :

$$\tau_{i,LM,t} = pr_{i,L} * \left(t_{LM} * Nb\_data_i \right) \quad (3.40)$$

où t_{LM} est le temps de lecture/écriture d'une donnée en mémoire locale

- Temps de lecture/écriture d'une donnée en mémoire globale :

$$\tau_{i,GM,t} = pr_{i,G} * \left(t_{GM} * Nb\_data_i \right) \quad (3.41)$$

où t_{GM} est le temps de lecture/écriture d'une donnée dans la mémoire globale

Considérant une trace donnée t, nous avons pour un processeur P_i:

$$\tau_{P,i,t} = \sum_{j=1}^{Nb_t} \left(\tau_{j,t} + \tau_{j,LB,t} + \tau_{j,GB,t} + \tau_{j,LM,t} + \tau_{j,GM,t} \right) \quad (3.42)$$

où Nb_t est le nombre de tâches sur la trace t et $\tau_{j,t}$ est la somme des durées telles que définies dans les équations (3.3) et (3.4).

Considérant un ensemble donné S_k de traces concurrentes dans un ASIC A_i, nous avons :

$$\tau_{A,i,k} = \sum_{j=1}^{Nbk} \left(\tau_{j,t} + \tau_{j,LB,t} + \tau_{j,GB,t} + \tau_{j,LM,t} + \tau_{j,GM,t} \right) \quad (3.43)$$

où Nb_k est le nombre de tâches dans l'ensemble S_k et $\tau_{j,t}$ est la somme des durées telles que définies dans les équations (3.36) et (3.4).

La durée d'exécution des différentes tâches de l'application affectées aux différents processeurs et ASICs peut être estimée par :

$$\tau = \max_{\substack{1 \le l1 \le \#Processeurs \\ 1 \le l2 \le \#ASICs}} \left[\max_t \tau_{P,i,t,l1}, \max_k \tau_{A,i,k,l2} \right] \quad (3.44)$$

3.5.3. Consommation de puissance

Elle peut être estimée en considérant les différentes consommations d'énergie suivantes :

- Consommation d'énergie due à un transfert de données sur le bus local :

$$E_{i,LB,t} = pr_{i,L} * \left(C_{LB} * V_i^2 * Nb\_data_i \right) \quad (3.45)$$

- Consommation d'énergie due à un transfert de données sur le bus global :

$$E_{i,GB,t} = pr_{i,G} * \left(C_{GB} * V_i^2 * Nb\_data_i \right) \quad (3.46)$$

V_i est la tension affectée à la tâche *i* transmettant Nb_data_i sur le bus local (global) de capacité C_{LB} (C_{GB})

- Consommation d'énergie due à une lecture/écriture d'une donnée en mémoire locale :

$$E_{i,LM,t} = pr_{i,L} * Nb\_data_i * E_{LM} \quad (3.47)$$

- Consommation d'énergie due à une lecture/écriture d'une donnée en mémoire globale :

$$E_{i,GM,t} = pr_{i,M} * Nb\_data_i * E_{GM} \quad (3.48)$$

où E_{LM} (E_{GM}) est l'énergie dissipée pour la lecture ou l'écriture d'une donnée en mémoire locale (globale)

Considérant une trace *t* donnée, nous avons pour un processeur P_i:

$$E_{P,i,t} = \sum_{j=1}^{Nb_t} \left(E_{j,t} + E_{j,LB,t} + E_{j,GB,t} + E_{j,LM,t} + E_{j,GM,t} \right) \quad (3.49)$$

Nb_t est le nombre de tâches dans la trace *t*. E_{it} est la somme des énergies définies par les équations (3.1) et (3.2).

L'énergie moyenne consommée par P_i est alors :

$$E_{P,i,avg} = \frac{1}{NB_i} \sum_{i=1}^{NB_i} E_{P,i,t} \quad (3.50)$$

NB_i est le nombre de traces qui peuvent être exécutées par le processeur P_i

Considérant un ensemble S_k de traces concurrentes dans un ASIC A_i donné, nous avons :

$$E_{A,i,k} = \sum_{t=1}^{|S_k|} \sum_{j=1}^{Nb_{t,k}} \left(E_{j,t} + E_{j,LB,t} + E_{j,GB,t} + E_{j,LM,t} + E_{j,GM,t} \right) \quad (3.51)$$

$Nb_{t,k}$ est le nombre de tâches dans la trace *t* de l'ensemble S_k.

L'énergie moyenne consommée par A_i est alors:

$$E_{A,i,avg} = \frac{1}{NB_{SC}} \sum_{k=1}^{NB_{SC}} E_{A,i,k} \quad (3.52)$$

Finalement, l'énergie moyenne consommée par tout le système est :

$$E_{avg} = \sum_{i=1}^{Nb\_Proceseurs} E_{P,i,avg} + \sum_{i=1}^{Nb\_ASICs} E_{A,i,avg} \quad (3.53)$$

La puissance moyenne consommée par tout le système est : $P_{avg}=E_{avg}* f$; f est la fréquence de fonctionnement du système.

3.6. Vérification :

Au niveau d'abstraction *système*, la vérification consiste à vérifier le comportement du système vu comme étant un ensemble de sous-systèmes communiquant à travers un protocole de communications défini. SYSTEMC offre la possibilité de vérifier le système sans tenir compte de l'aspect *temps* (*untimed functional level*), puis de tenir compte de certaines contraintes temporelles (*bus cycle accurate*). Ceci a pour avantage de tenir compte des détails progressivement.

3.7. Conclusion :

Nous avons vu dans ce chapitre comment, à partir d'une description du système à concevoir, aboutir à un graphe de tâches moyennant des analyses syntaxique et lexicale. Un partitionnement des tâches basé sur leurs degrés de communications conduit à une configuration d'une architecture constituée de sous-systèmes interconnectés via un protocole de communications que nous avons également présenté. Nos méthodes de synthèse et d'analyse montrent enfin comment le concepteur peut aboutir à une configuration répondant à certaines contraintes de surface, de vitesse et de consommation de puissance. Cette configuration étant arrêtée, chacun des sous-systèmes qui la constitue fait l'objet d'opérations de synthèse, d'analyse et de vérification. Ceci fait l'objet du chapitre qui suit.

Chapitre IV

Synthèse, analyse et vérification de sous-systèmes vlsi

4.1. Introduction :

La synthèse du système ayant conduit à un certain nombre de sous-systèmes répondant aux différentes contraintes assujetties au *plus haut* niveau d'abstraction (le niveau *système*), il s'agit de synthétiser chacun de ces sous-systèmes, à un niveau d'abstraction *plus bas*, le niveau d'abstraction *transfert de registres*.

4.2. Synthèse :

4.2.1. Partie opérative

Un premier compromis sur les paramètres *surface*, *vitesse* et *consommation de puissance* a été abordé au niveau *système* en répartissant les différentes tâches sur un nombre de sous-systèmes répondant aux contraintes spécifiées au niveau d'abstraction cité. De même, au niveau d'abstraction *transfert de registres*, il est possible de générer, pour chaque sous-système, plusieurs configurations ayant des caractéristiques différentes. En effet, supposons que dans la description du sous-système, il existe N opérations du même type qui peuvent être exécutées durant le même cycle. Le sous-système peut être rendu performant en matière de vitesse si l'on allouait N entités physiques à l'opération donnée. Toutefois, ceci se ferait au détriment de la surface et de la consommation de puissance. D'un autre côté, si l'on allouait une seule entité physique aux N opérations, celles-ci ne pourraient être exécutées de manière parallèle. La problématique consiste alors à trouver le bon compromis permettant de concilier les trois types de contraintes. Ceci nous amène à *ordonnancer* les différentes opérations de manière à satisfaire les contraintes.

Etant donné un flot de données contrôlées (CDFG), il s'agit d'ordonnancer chacune des traces contenues dans ce flot. Le nombre de traces incluses dans ce flot est égal à $\prod_{i=1}^{\#S\_C} N_i$, S_C étant le nombre de structures de contrôle, et N_i étant le nombre de traces contenues dans la $i^{ème}$ structure de contrôle. Dans le cas où le nombre de traces conduirait à un ordonnancement non polynomial, il faudrait réduire le nombre de traces tout en cherchant un bon compromis entre le temps d'exécution de l'ordonnancement et le nombre de cycles. Une technique utilisée dans [28] consiste à grouper chacune des structures contrôlant des opérations non assujetties à des contraintes en une seule (*collapsing*). Si le temps de traitement reste toujours important, il faudrait insérer un certain nombre de points de coupure entre différentes paires de structures de contrôle successives. Ainsi, pour M structures de contrôle isolées, le nombre de traces engendré serait non pas $\prod_{i=1}^{M} M_i$, mais plutôt $\sum_{i=1}^{M} M_i$, M_i étant le nombre de traces contenues dans chacune de ces M structures de contrôle. Dans ce cas, il faut noter que deux opérations se trouvant dans deux structures de contrôle isolées ne peuvent être ordonnancées durant le même cycle d'exécution quand bien même elles soient indépendantes. D'où le problème de trouver un bon compromis entre le temps d'exécution de l'ordonnancement et le nombre de cycles qu'exécutera la partie opérative à concevoir.

Les différentes traces étant définies, l'ordonnancement s'articule, pour chacune de ces traces, sur deux grandes étapes :

- détecter toutes les opérations dépendantes et les affecter à des cycles d'exécution tels que :

  $C(O_i) < C(O_j)$ si les opérations O_i et O_j sont dépendantes et si O_j dépend de manière directe ou indirecte du résultat de O_i.

- si dans chacun des cycles obtenus il n'existe pas de contraintes liant des opérations ordonnancées jusqu'alors dans le même cycle, alors fin du traitement. Sinon, pour chaque

contrainte, ordonnancer les opérations assujetties à cette contrainte dans des cycles différents, et mettre à jour convenablement l'affectation de toutes les autres opérations.

Notons que les différentes contraintes sont introduites par l'outil de synthèse afin de trouver un bon compromis conciliant la surface, la vitesse et la consommation de puissance. A cet effet, nous avons développé un outil ([8]) qui permet de tenir compte de cette problématique, notamment pour la consommation de puissance, paramètre très critique dans les systèmes actuels. L'algorithme en est le suivant :

```
// Compromis vitesse – consommation de puissance
// au niveau transfert de registres
P= +∞;
tant que (P > P_desired)
faire { schedule_traces(); // ordonnancement
        P= -∞;
        pour chaque trace i
        faire {utiliser SPOT ([10], [11], [12]) pour déterminer POWER,  la puissance dissipée par cycle;
                si (POWER > P)
                alors P= POWER;
                fsi
                pour j=1 jusqu'à nb_cycles_i
                  // nb_cycles_i est le nombre de cycles dans la trace i pendant l'ordonnancement courant
                faire si nb_i_j > avg_nb_i
                        // nb_i_j est le nombre d'opérations de la trace i
                        // dans le cycle j de l'ordonnancement courant
                        // avg_nb_i est le nombre moyen d'opérations par cycle dans le prochain ordonnancement
                    alors {k_ij = ⌊nb_i_j / avg_nb_i⌋;
                           Construire Ng_ij = ⌈nb_i_j / k_ij⌉ graphs G_1,  G_2, ...,G_Ngij  tels que:
                                |V_k| = k_ij;  |E_k|=k_ij*(k_ij-1)/2;   k=1, 2, ..., Ng_ij
                           n=Ng_ij*k_ij;
                           si (n < nb_i_j)
                           alors {construire un graphe additionel G_a=(V_a,E_a) tel que:
                                   . |V_a|=nb_i_j – n=n_a_ij;
                                   . |E_a|=n_a_ij*(n_a_ij-1)/2
                                  }
                           fsi
                           pour chaque opération I_l dans le cycle j de l'ordonnancement courant,

                                il existe 1 et 1 seul noeud v_l dans  ⋃_{k=1}^{N_g} V_k ⋃ V_a

                           pour chaque graphe G_k=(V_k,E_k)  (1 ≤k ≤Ng_ij)
                           faire {pour chaque  arête e_lm ∈ E_k
                                  faire générer une contrainte entre les opérations I_l et I_m;
                                  fin
                                  pour chaque arête e_lm ∈ E_a // dans le cas où G_a existe
                                  faire générer une contrainte entre les opérations I_l and I_m;
                                  fin
                                  }
                           fin
                          }
                    fsi
                end
               }
    end
   }
end
```

Fig.4.1. Algorithme visant un bon compromis entre la vitesse et la consommation de puissance dans une partie opérative

Noter que dans l'algorithme présenté, le nombre de contraintes est égal à :

$$\sum_{i=1}^{\#traces} \sum_{j=1}^{\#c\_i} \left[\left[k_{ij} * \left(k_{ij} - 1 \right) / 2 \right] * Ng_{ij} + n_{a\_ij} * \left(n_{a\_ij} - 1 \right) / 2 \right]$$

$$= 1/2 \sum_{i=1}^{\#traces} \sum_{j=1}^{"c\_i} \left[\left[k_{ij} * \left(k_{ij} - 1 \right) \right] * Ng_{ij} + n_{a\_ij} * \left(n_{a\_ij} - 1 \right) \right]$$

où Ng_{ij} est le nombre de graphes construits pour le $j^{ème}$ cycle de la $i^{ème}$ trace, k_{ij} est le nombre de nœuds dans chacun des Ng_{ij} graphes, n_{a_ij} est le nombre de nœuds dans le graphe additionnel éventuel pour le $j^{ème}$ cycle de la $i^{ème}$ trace, $\#c\_i$ est le nombre de cycles dans l'ordonnancement courant de la trace i de sorte qu'il n'y'ait pas plus que *avg_nb_i* opérations dans chacun des cycles. Comme nous le constatons dans la dernière équation, le nombre de contraintes peut être très important et nécessite de ce fait une procédure de générations de contraintes automatique libérant du coup le concepteur de circuits d'une tâche fastidieuse.

L'idée générale de l'algorithme présenté est que la dissipation de puissance est d'autant plus importante qu'il existe un grand nombre d'opérations à exécuter dans le même cycle. L'algorithme consiste alors à générer les contraintes adéquates pour aboutir au compromis recherché. Afin de l'illustrer, considérons l'exemple suivant :

Exemple : Supposons que 26 (nb_i_j) opérations sont ordonnancées dans le même cycle d'exécution j d'une trace donnée i du flot de données. Soit P la puissance dissipée dans le circuit implémentant les opérations telles qu'elles sont ordonnancées. Supposons que $P > P_{désirée}$ et que le nombre moyen d'opérations dans le même cycle issu du prochain ordonnancement est 8 (*avg_nb_i*). Nous avons alors à ordonnancer les 26 opérations dans des cycles différents, et non pas dans le même cycle comme c'est le cas avec l'ordonnancement courant. Aussi, nous avons à introduire des contraintes en vue de réduire P. Ces contraintes sont alors générées comme suit :

$k = \lfloor nb\_i_j / avg\_nb\_i \rfloor = \lfloor 26/8 \rfloor = 4$

- construire $Ng = \lceil 26/4 \rceil = 6$ graphes $G_1, G_2, ..., G_6$
  $|V_k| = k = 4$; $|E_k| = |V_k| * (|V_k| - 1)/2 = 6$; $k = 1,2, ...,6$

- construire un graphe additionel $G_a = (V_a, E_a)$
  $|V_a| = nb\_i_j - Ng * |V_k| = 26 - 6*4 = 2$
- pour chaque opération O_l ($1 \leq l \leq 26$), il existe 1 et 1 seul nœud v_l dans $\bigcup_{k=1}^{6} V_k$
- générer une contrainte entre les opérations I_l et I_m si $e_{lm} \in E_k$; k=1, 2, ..., 6 (pour chacune des 24 opérations, il existe un nœud appartenant à V_k ; k=1, 2, ..., 6 (voir Fig.4.2)
- générer une contrainte entre les opérations I_l et I_m si $e_{lm} \in E_a$
- nous avons à ordonnancer au plus 8 (avg_nb_i) opérations par cycle
- le pas de contrôle j est remplacé par $|V_k| = 4$ nouveaux cycles

L'utilisation de l'algorithme précédemment décrit conduit pour cet exemple à la figure suivante :

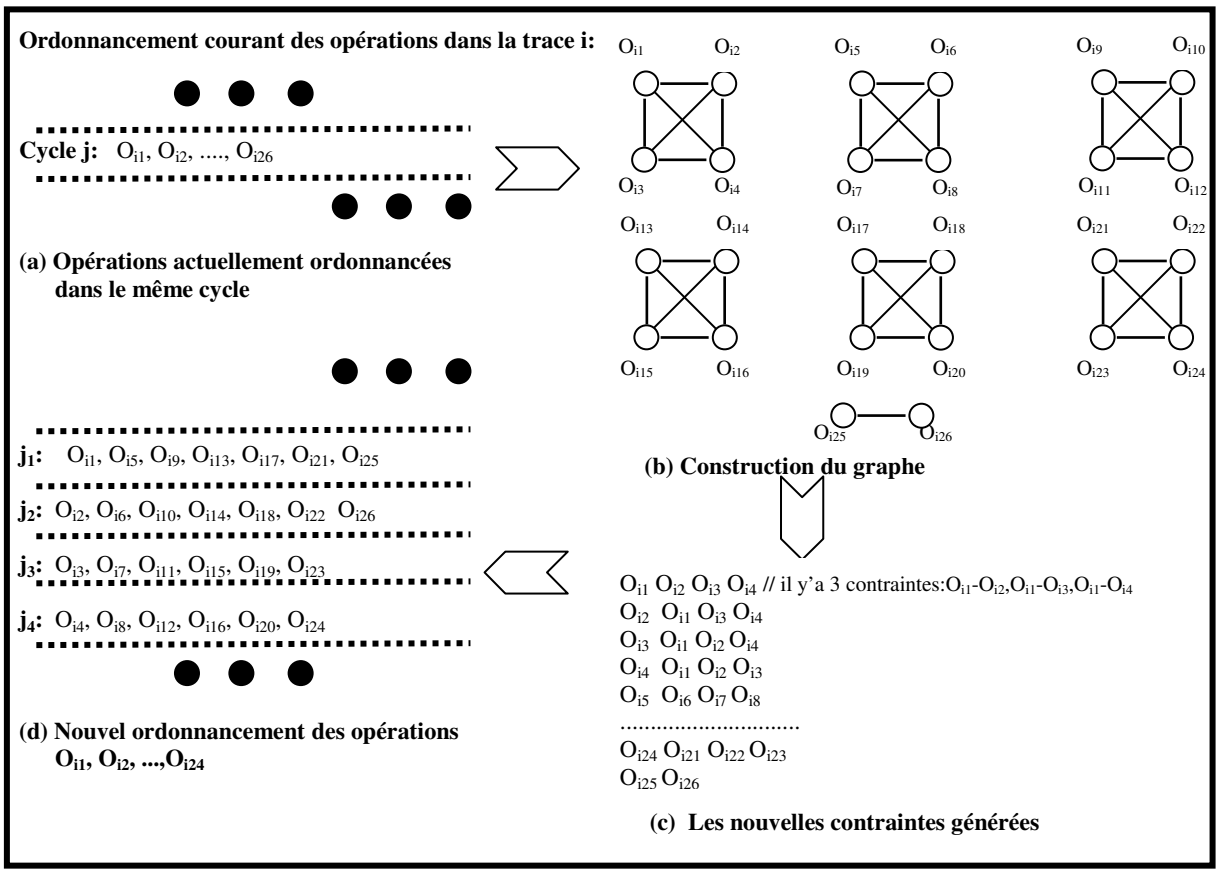

Fig.4.2. Exemple de génération de contraintes

Ayant obtenu l'ordonnancement adéquat, il s'agira d'allouer de manière efficace les entités utilisées dans le comportement de la partie opérative à des ressources physiques. En effet, s'il existait un million de variables dans la description, il serait aberrant de leur allouer un million de registres. Le problème consiste alors à déterminer le nombre optimal de registres interdisant d'écraser une variable (par une autre) dont l'utilisation est encore requise. Il en est de même pour les interconnexions : affecter une interconnexion à chaque transfert de données conduirait à une surface importante du circuit sans compter la dégradation du circuit en matière de vitesse et de consommation de puissance à cause des nombreux paramètres parasites engendrés. Il y'a donc lieu d'allouer un nombre optimal d'interconnexions sans pour autant générer des conflits dans des transferts de données simultanés. Ces deux problèmes d'allocation sont en fait NP-complets et peuvent être résolus par la méthode des graphes colorés.

Pour les variables, il s'agira simplement de :

- déterminer la table de vie des différentes variables à partir de la durée de vie de chaque variable (intervalle déterminé par le 1er cycle où une variable est utilisée dans une opération et le dernier cycle où elle l'est aussi. Pendant tout cet intervalle, le registre affecté à cette variable ne pourra être affecté à une autre).
- déterminer le graphe G=(V,E) tel que chaque nœud correspond à une variable donnée, et deux nœuds sont reliés par une arête si et seulement si les durées de vie correspondantes se chevauchent.
- colorer G, puis affecter le même registre aux variables dont les nœuds correspondants ont reçu la même couleur.

Pour les interconnexions, le traitement est sensiblement le même :

- grouper dans la même unité fonctionnelle fictive toutes les unités fonctionnelles qui sont deux à deux utilisées dans des cycles différents
- déterminer $G_1=(V_1,E_1)$ tel que chaque nœud correspond à un transfert de données d'une unité fonctionnelle fictive à un registre, et deux nœuds sont reliés par une arête si et seulement si les transferts de données associés s'opèrent en parallèle (opérations ordonnancées dans le même cycle)
- colorer G_1, puis affecter la même interconnexion aux transferts de données dont les nœuds associés ont reçu la même couleur
- déterminer $G_2=(V_2,E_2)$ tel que chaque nœud correspond à un transfert de données d'un registre à une unité fonctionnelle fictive, et deux nœuds sont reliés par une arête si et seulement si les transferts de données associés s'opèrent en parallèle (opérations ordonnancées dans le même cycle)
- colorer G_2, puis affecter la même interconnexion aux transferts de données dont les nœuds associés ont reçu la même couleur

Notons que notre outil de coloration de graphes a été présenté dans le troisième chapitre.

4.2.2. Partie de contrôle ([5])

Le comportement du sous-système peut être complètement défini par une machine à états finis décrivant les actions à exécuter selon des conditions données :

$X=\{x_1,x_2,...,x_{|X|}\}$ est l'ensemble des signaux de contrôle
$Y=\{y_1,y_2,...,y_{|Y|}\}$ est l'ensemble des états de la machine
$Z=\{z_1,z_2,...,z_{|Z|}\}$ est l'ensemble des signaux de commande
δ : XxY ->Y est la fonction qui détermine le prochain état de la machine
λ:XxY->Z(λ:Y-> Z) est la fonction générant les signaux de commande pour une machine de Mealy (Moore)

Générer la partie de contrôle directement à partir de la machine à états finis correspondante conduirait à une partie de contrôle offrant des caractéristiques peu intéressantes du point de vue surface, vitesse et consommation de puissance à cause d'un nombre énorme de transistors, de longues lignes d'interconnexion et de beaucoup de paramètres parasites. Certaines méthodes telles que celle dite *1 hot-encoding* consistent à minimiser la surface de la partie de contrôle en réduisant le nombre de lignes, d'où une réduction implicite du nombre de transistors et d'interconnexions entraînant du coup une amélioration de la vitesse et de la consommation de puissance. D'autres techniques plus intéressantes (dont la nôtre [5]) consiste à réduire beaucoup plus le nombre de lignes : étant donné N états, la codification ne se fait pas sur une longueur de $\lceil log_2\ N \rceil$bits, mais sur une longueur pouvant être supérieure. Cette relaxation sur la longueur des codes des états conduit parfois à de meilleures performances. Toutefois, dans certains cas, les résultats sont plutôt médiocres par rapport à ceux engendrés par des techniques telles que *1 hot-encoding*. Ceci est dû au fait que le nombre de colonnes obtenu peut être très important par rapport à celui induit par une codification sur $\lceil log_2\ N \rceil$bits, ce qui donnerait une surface plus importante que celle produite par *1 hot-encoding*.

Bien que notre technique opère sur une longueur de codes pouvant être différente de $\lceil log_2\ N \rceil$bits, elle surmonte ce problème. Mieux encore, la longueur du code peut être même *inférieure* à $\lceil log_2\ N \rceil$ bits, tout en réduisant du même coup le nombre de lignes. Ceci est possible en affectant le *même* code à des états *différents*, sans pour autant remettre en cause la fonctionnalité correcte de la machine à états finis et donc celle de la partie de contrôle. Ceci est une caractéristique propre à notre méthode ([5]) du fait que toutes les autres méthodes donnent une longueur d'au moins $\lceil log_2\ N \rceil$ bits. Avant de formuler le problème, considérons l'exemple suivant :

00 s7 s1 10
00 s8 s1 10
00 s9 s4 11
00 s10 s4 11
01 s5 s2 11
01 s6 s2 01
01 s7 s8 11
01 s1 s8 00
10 s1 s4 00
10 s2 s4 00
10 s3 s4 00
10 s5 s7 00
10 s6 s7 00
11 s1 s5 01
11 s4 s5 01
11 s2 s5 01
11 s8 s2 10
11 s3 s1 11

La première ligne signifie : si les signaux de contrôle sont 00 et que la machine se trouve à l'état s7, alors le nouvel état serait s1, et les signaux de commande seraient 10. Regroupons dans le même état (état composé) tous les états qui alimentent la partie de contrôle avec les mêmes signaux de commande et qui génèrent les mêmes états et signaux de commande. On a alors :

00 {s7,s8} s1 10
00 {s9,s10} s4 11
01 s5 s2 11
01 s6 s2 01
01 s7 s8 11
01 s1 s8 00
10 {s1,s2,s3} s4 00
10 {s5,s6} s7 00
11 {s1,s2,s4} s5 01
11 s8 s2 10
11 s3 s1 11

Ce regroupement conduit à un nombre de lignes *le plus minimal*. Il s'agira par la suite de réduire autant que possible le nombre de colonnes de la partie de contrôle. Avant de discuter ce point, nous donnons les définitions suivantes en nous servant de l'exemple.

- Les états simples sont: s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, E_1={s7,s8}, E_2={s9,s10} et E_3={s1,s2,s4}
- Les états composés sont: E_{c1}={s1,s2,s3}, E_{c2}={s5,s6}
- Les partitions sont P1, P2, P3 et P4, et correspondent respectivement aux signaux 00, 01, 10 et 11
- Count(état simple)=1
- Count(E_{c1})=2 car s1 (ou s2) appartient à la même partition (P4) que s3
- Count(E_{c2})=2 car s5 et s6 appartiennent à la même partition (P2)

Noter que E_1={s7,s8} est en fait un état simple puisqu'il n'existe aucune partition contenant à la fois s7 et s8. Par contre, E_{c1}={s1,s2,s3} est un état composé puisque la partition P4 (11) contient à la fois s1 et s2.

Soient c1=00, c2=01 et c3=11 les codes respectifs de s1, s2 et s3. Le code de E={s1,s2,s3} qui est l'intersection de c1, c2 et c3 est c=** ={00,01,10,11}. Puisque le code c couvre tous les codes possibles sur 2 bits, il ne serait pas possible de coder un état (s4 par exemple) sur 2 bits sans entraîner un non déterminisme si s4 se trouvait dans la même partition que E (c'est-à-dire alimentant la partie de contrôle avec les mêmes signaux de contrôle). Dans ce cas, la seule possibilité serait d'augmenter de 1 bit la longueur du code pour lever ce non déterminisme. Dans un cas plus général et complexe, une codification mal déterminée entraînerait une longueur de codes très importante, conduisant par la suite (aux niveaux d'abstraction plus bas) à une conception d'une partie de contrôle inefficace du point de vue surface, vitesse et consommation de puissance.

Nous définissons B={0,1} et B'={0,1,*} et formulons ce problème de codification comme suit :

Soit l'instance J={j; j $\in$ $B'^{|X|}$ est un signal de contrôle et correspond à une certaine partition P}
(∅) pour une machine de Mealy (Moore)
S={s ; s est un état simple}
C={E ; E est un état composé}

Notre formulation du problème est alors :

f : (J, S $\cup$ C) ⟶ {codes c tels que la longueur de codification c vaut k}
Trouver la valeur minimale de k telle que :

$f(j, E_m) \cap f(j, E_n) = \emptyset \; \forall \; E_m \neq E_n \; \wedge \; \exists$ partition P telle que $E_m \in P, E_n \in P \; \wedge$
Il n'existe pas E $\in$ C tel que $E_m \in E, E_n \in E$ in P

Faisons sur (Pb) les considérations suivantes :

i) Nous venons de voir que $c1 \cap c \neq \emptyset$ même si $c1 \neq c$ (car c couvre c1). Soit C = ∅ (il n'existe pas d'état composé). Afin de satisfaire les contraintes indiquées dans la formulation de (Pb), il suffit d'avoir : $c1=f(j, E_m) \neq c2=f(j,E_n)$ car c1 (c2) ne peut couvrir c2 (c1).

ii) Soit C ≠ ∅, mais $E_m \cap E_n = \emptyset \; \forall$ m ≠ n. Supposons qu'il existe une fonction *g* telle que : $f(j, E_m) = c_m \mid g(j, E_m)$ où $c_1 \mid c_2$ veut dire *concaténer le code c_2 à celui de c_1*. Afin de satisfaire la contrainte indiquée dans (Pb), il suffit d'assurer $c_m \neq c_n$ (noter que chaque bit de code de c_m ou c_n appartient à B). En d'autres termes, c_m (c_n) ne peut couvrir c_n (c_m).

Ces deux considérations simplifient énormément (Pb) en termes de temps CPU pour sa résolution :

- dans i), il suffit que deux états se trouvant dans la même partition aient des codes différents,
- dans ii), même si $g(j,E_m)$ couvre $g(j,E_n)$, les codes de E_m et de E_n ne peuvent se couvrir car $c_m \neq c_n$. Nous définissons c_i ; $1 \leq i \leq |C|$ comme suit : c_i est un code dont la longueur est égale à $\lceil \log_2 |C| \rceil$ et dont la valeur décimale appartient à [0, |C|-1]. Noter que |C| est le nombre des différents états composés, que $c_m \neq c_n$ si E_m et E_n n'appartiennent pas au même état composé, et que le code partiel des états simples composant E_i vaut c_i.

A cause de ces considérations, nous avions décomposé le problème en trois sous-problèmes :

(SP_1) : Résoudre (Pb) lorsque $C = \varnothing$
(SP_2) : Résoudre (Pb) lorsque $|C| > 0$ mais $E_m \cap E_n = \varnothing \ \forall\ m \neq n$
(SP_3) : Résoudre (Pb) si $|C| > 0 \ \wedge \ \exists\ E_i$ et E_j tels que $E_i \cap E_j \neq \varnothing$; $E_i \in C$; $E_j \in C$; $i \neq j$

Trois heuristiques ont été développées pour la résolution de ces sous-problèmes et tiennent compte de ce qui suit :

Deux états quelconques s_i et s_j reçoivent le même code sans incidence aucune sur le comportement correct de la partie de contrôle si

i) il n'existe aucune partition P_k telle que $s_i \in P_k$ et $s_j \in P_k$ ou,
ii) si de telles partitions P_k existent, s_i et s_j doivent faire partie d'un état composé inclus dans P_k

Preuve:

i) Soit $x=\{x_{i1},x_{i2},...,x_{i|x|}\}$ l'ensemble des signaux de contrôle introduits dans la partie de contrôle concaténés avec le code de s_i, et $x' =\{x_{j1},x_{j2},...,x_{j|x|}\}$ celui des signaux de contrôle concaténés avec le code de s_j. Comme s_i et s_j n'appartiennent pas à la même partition, nous avons $x \neq x'$. Par conséquent, le nouvel état de la machine et les signaux de commande seront déterminés de manière déterministe. ■

ii) Puisque dans chaque partition P_k il existe un état composé E_c tel que s_i et s_j y appartiennent, s_i et s_j induisent alors le même nouvel état et les mêmes signaux de commande. Du fait qu'il n'existe aucune partition où s_i et s_j induisent des états nouveaux différents ou des signaux de commande différents, la partie de contrôle ne peut pas être non déterministe. ■

Notre stratégie globale de codification se base sur le fait que :
- le code doit être adjacent au plus grand nombre possible de codes affectés aux états simples composant les mêmes états composés que l'état en cours de codification (notons H cet ensemble d'états composés) : l'intersection des codes des états simples composant ces états composés couvrirait un nombre faible de codes
- le code en cours de détermination ne doit pas être adjacent au plus grand nombre possible de codes affectés aux états composés qui se trouvent dans les mêmes partitions que celles où se trouvent les éléments de H (l'ensemble de telles partitions est dénoté P_s) : ceci est une bonne stratégie pour réserver certains codes aux états simples qui n'appartiennent pas aux états composés de H (défini précédemment) et pour augmenter la probabilité que l'intersection des codes des éléments de H ne couvrira pas des codes interdits, évitant ainsi d'augmenter inutilement la longueur du code

Soit e_{sj} l'état en cours de codification. Nous définissons :
$H=\{E_{ci}$; $e_{sj} \in E_{ci}$; count(E_{ci}) > 1$\}$ et $F=\{E_{ci}$; $E_{ci} \in P_s$; $E_{ci} \notin H$; count(E_{ci}) > 1 $\}$

Du fait qu'il devrait exister des codes appartenant aussi bien à C_H qu'à C_F, nous définissons
$D = C_H \cap C_F$ où C_H={codes des états simples composant E_{ci} ; $E_{ci} \in H$} et C_F={codes des états simples composant E_{ci} ; $E_{ci} \in F$}

Nous posons $C'_H = C_H - D$ et $C'_F = C_F - D$ à partir desquels deux valeurs *degré1* et *degré2* seront déterminées :

- degré1 est le nombre de codes inclus dans C'_H qui sont adjacents à un certain code c
- degré2 est le nombre de codes inclus dans C'_F et qui ne sont pas adjacents à un certain code c.

D peut être considéré comme étant un ensemble de codes *parasites*.

Nous procédons alors à la codification courante de l'état comme suit :

i) $\mathbf{C'_H = \varnothing \wedge C'_F = \varnothing}$ **:** c'est le cas où $C_H=C_F=D$. C'est alors le cas le plus défavorable, et n'importe quel code non interdit peut être affecté à e_{sj}

ii) $\mathbf{C'_H = \varnothing \wedge C'_F \neq \varnothing}$ **:** c'est le cas où $C_H=D$ et $C_H \subset C_F$. Du fait que $C'_H = \varnothing$, seul *degré2* peut déterminer la qualité du code c qui peut être affecté à e_{sj} :c ne doit pas être adjacent au plus grand nombre possible de codes inclus dans C'_F

iii) $\mathbf{C'_H \neq \varnothing \wedge C'_F = \varnothing}$ **:** c'est le cas où $C_F=D$ et $C_F \subset C_H$. Le code c qui rend la valeur de *degré1* la plus grande possible est affecté à e_{sj}

iv) $\mathbf{C'_H \neq \varnothing \wedge C'_F \neq \varnothing}$ **:** c'est le cas où $C_H \not\subset C_F$ et $C_F \not\subset C_H$. Nous déterminons c comme suit :

si $|C'_H| > |C'_F|$

alors c est le code qui rend la valeur de *degré1* la plus grande possible (*degré_a*);

/* si beaucoup de tels codes c existent, sélectionner celui qui est le moins adjacent aux codes inclus dans C'_F et qui satisfait degré1=degré_a */

sinon c est le code qui rend la valeur de *degré2* la plus petite possible (*degré_n*);

/* si beaucoup de tels codes c existent, sélectionner celui qui est le plus adjacent aux codes inclus dans C'_H et qui satisfait degré2=degré_n */

fsi

Notre méthode a été comparée à différentes autres méthodes moyennant l'utilisation de 43 MCNC FSM Benchmarks. Cette comparaison nous a montré que :

- notre méthode donne les meilleurs résultats dans 74% de cas
- seule notre méthode peut donner une longueur de codes inférieure à $\lceil \log_2 N \rceil$, N étant le nombre d'états de la machine à états finis (exploitation de notre lemme présenté précédemment) tout en assurant un comportement correct de la machine à états finis

4.2.3. Interface de communications entre une partie opérative et sa partie de contrôle

Nous avons présenté dans le précédent chapitre le protocole de communications entre les différents sous-systèmes. Nous présenterons dans ce qui suit celui qui régit les transferts de données dans le même sous-système. Notre protocole (implémenté en C par deux ex-élèves-ingénieurs dans le cadre de leur mémoire d'Ingéniorat en Informatique [29]) est basé sur l'utilisation de transistors de passage dont le signal sur la grille permet ou empêche le passage d'une donnée. Ce signal émane de la partie de contrôle correspondant à la machine à états finis du sous-système.

Il s'agit dans un premier temps de déterminer tous les transistors de passage à insérer entre les différents composants du sous-système (registres, unités fonctionnelles, interconnexions). L'exemple de la figure 4.3 montre comment contrôler le transfert d'une donnée issue du registre R1 vers le bus 1 ou le bus2, en fonction du comportement du sous-système.

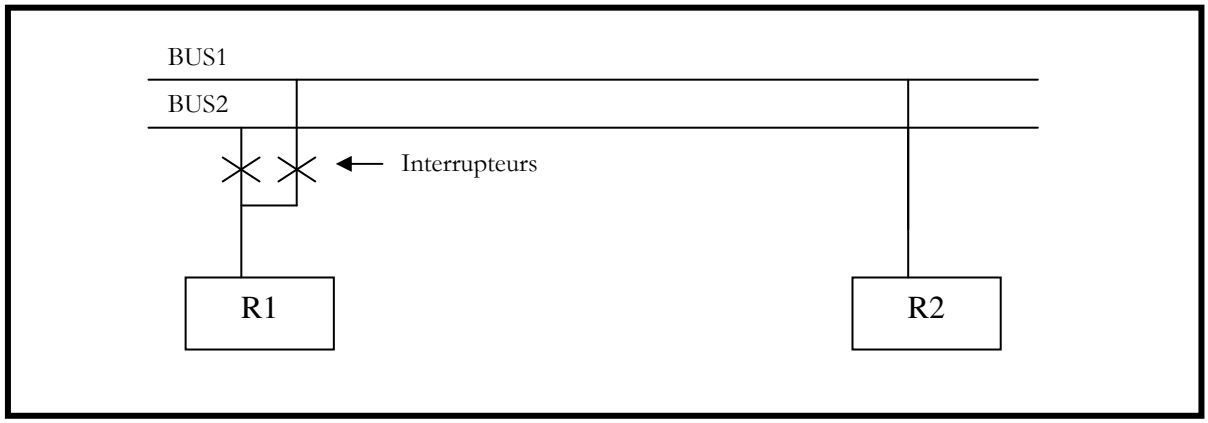

Fig.4.3. Exemple de contrôle de transfert de données

Ce comportement est déterminé à ce stade, à partir du fichier issu de l'allocation de ressources physiques, étape précédée elle-même par l'ordonnancement. La figure 4.4 en est un exemple montrant :

- le transfert du contenu du registre r5 vers la première entrée de l'additionneur à travers le bus br4
- le transfert du contenu du registre r6 vers la deuxième entrée de l'additionneur à travers le bus br3
- le transfert du résultat de l'addition vers le registre r4 à travers le bus bf2

L'instruction ordonnancée dans le premier cycle est exécutée sans condition aucune (condition=1=vrai), celle ordonnancée dans le deuxième cycle n'est excutée que si la condition booléenne B or A est vraie.

```
-----------------------------------------
cycle 1
[r4] := [r5]  +  [r6]   1
br4
br3
bf2
-----------------------------------------
cycle 2
[r3] := [r4] - [r2]     B or A
br3
br4
bf2
……
```

Fig.4.4. Exemple d'un fichier issu de l'allocation de ressources physiques

Ainsi, pour cet exemple, les signaux de commande qui émanent de la partie de commande et qui contrôlent les différents transferts de données lors du premier cycle sont mis à 1 lorsque :

- la machine à états finis se trouve dans le premier état et
- les signaux de contrôle valent ** (* pour A et * pour B, * représente 0 ou 1)

Durant ce cycle, tous les autres signaux de commande sont mis à 0.

Les signaux de commande contrôlant les différents transferts durant le deuxième cycle sont mis à 1 lorsque :

- la machine à états finis se trouve dans le second état et les signaux de contrôle valent 01 (A faux, B vrai)
  ou lorsque

- la machine à états finis se trouve dans le second état et les signaux de contrôle valent 10 (A vrai, B faux)
  ou lorsque

- la machine à états finis se trouve dans le second état et les signaux de contrôle valent 11 (A vrai, B vrai)

De même, durant ce cycle, tous les autres signaux de commande sont mis à 0.

Rappelant que dans un flot de données contrôlées il existe plusieurs traces, les transitions d'états se font de manière générale comme suit :

- de S_0 à l'état S_1 dans tous les cas (S_0 est l'état initial de la machine à états finis)

- de S_i à S_{i+1} ($i \geq 1$) pour toute trace contenant au moins une instruction dans le cycle S_{i+1} (la condition de transition est donc l'union des conditions d'exécution de telles traces)

- de S_i ($i \geq 1$) à S_0 pour toute trace dont le nombre maximal de cycles est égal à S_i (la condition de transition à l'état S_0 est donc l'union des conditions d'exécution de telles traces)

Appliquons cet algorithme pour l'exemple de la figure 4.5.

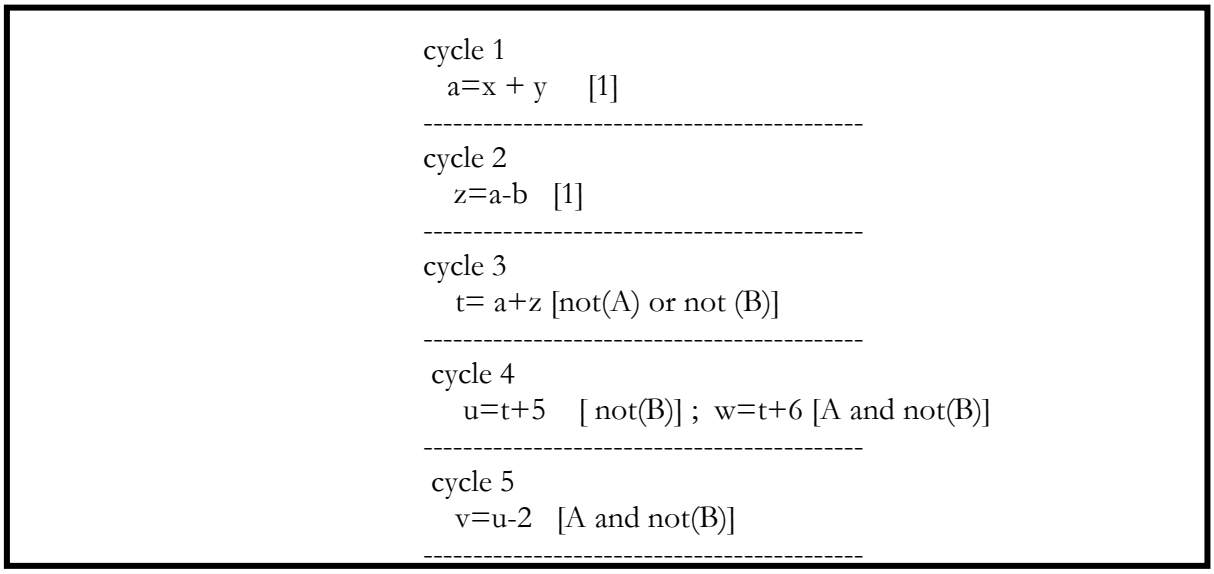

Fig.4.5. Exemple d'ordonnancement dans un flot de données contrôlées

Les traces issues du flot de données de la figure 4.5 sont les suivantes :

trace1 not(A) . not(B)	trace2 not(A) . B	trace3 A . not(B)	trace4 A . B
S_1-> 1 : a=x+y	1 : a=x+y	1 : a=x+y	1 : a=x+y
S_2-> 2 : z=a-b	2 : z=a-b	2 : z=a-b	2 : z=a-b
S_3-> 3 : t=a+z	3 : t=a+z	3 : t=a+z	
S_4-> 4 : u=t+5		4 : u=t+5 ; 6 : w=t+6	
S_5->		5 : v=u-2	

Fig.4.6. Traces issues d'un flot de données contrôlées

Dans la figure 4.5, il est indiqué que l'instruction 1 (a=x+y) s'exécute dans le cas où la variable logique 1 est vraie, ce qui est évidemment le cas. Aussi, cette instruction s'exécute pour n'importe quelle combinaison logique issue des variables A et B lorsque cette combinaison est vraie (voir Fig.4.6). Par contre, l'instruction 4 (u=t+5) ne s'exécute que si la condition not(B) est vraie, c'est-à-dire lorsque soit (not(A) and not(B)) est vraie, soit (A and not(B)) est vraie (voir Figs.4.5 & 4.6).

De la figure 4.6, on en déduit l'automate correspondant indiqué dans la figure 4.7. Cet automate se compose d'états (de S_0 à S_5), de transitions et d'actions à exécuter sous certaines conditions. L'état initial étant S_0, chacune des quatre traces comportant une instruction à l'état S_1, il y'a alors une transition de S_0 à S_1 dans tous les cas (variable logique =1). La transition de l'état S_3 à l'état S_4 n'est possible que si les conditions logiques (not(A) . not(B)) ou (A . not(B)) sont vraies (voir Fig.4.6), c'est à dire que si not(B) est vraie. Le même raisonnement est fait pour déterminer toutes les transitions.

Pour ce qui est des actions à exécuter, considérons par exemple le cas où la machine à états finis se trouve à l'état S_1. La figure 4.6 montre qu'il y'a transition vers l'état S_2 pour exécuter l'instruction 2 (z=a-b) quelles que soient les valeurs de A et B. Par contre, si la machine est à l'état S_3, il y'a transition vers l'état S_4 lorsque (not(B)) est vraie pour exécuter l'instruction 4 (u=t+5) et l'instruction 6 (w=t+6). Mais pour celle-ci, uniquement si la condition (A . not(B)) est vraie. Noter que si (A . not(B)) est vraie, la condition de transition (not(B)) l'est aussi, mais l'inverse ne l'est pas forcément.

Nous obtenons alors l'automate suivant:

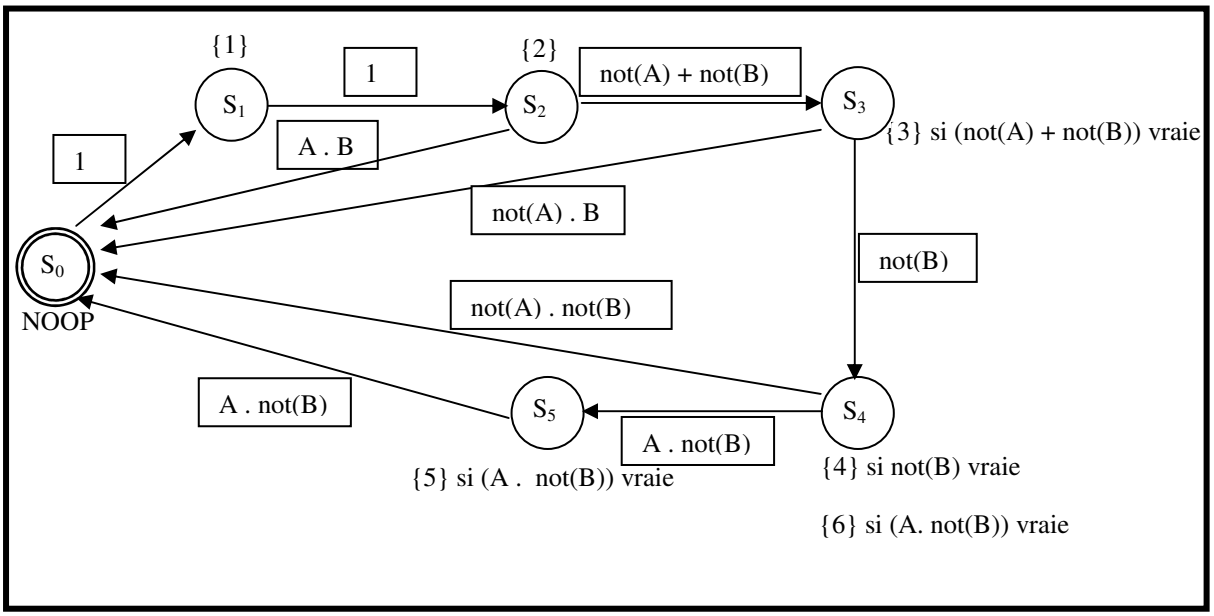

Fig.4.7. Exemple d'automate

La table de commande issue de cet automate est alors la suivante :

AB	état courant	état suivant	instructions à exécuter	signal
--	0	1	1	F 1
--	1	2	2	F 2
0-	2	3	3	F 3
-0	2	3	3	F 4
11	2	0	NOOP	F 5
-0	3	4	4	F 6
10	3	4	4, 6	F 7
01	3	0	NOOP	F 8
10	4	5	5	F 9
00	4	0	NOOP	F 10
10	5	0	NOOP	F 11

Fig.4.8. Exemple de table de commande

Considérons par exemple l'instruction 3 (t=a+z). Si les registres R_X, R_Y et R_Z ont été alloués respectivement pour les variables a, b et z pour le transfert de données entre l'additionneur et les registres à travers les interconnexions I_1, I_2 et I_3 comme indiqué dans la figure 4.9, les signaux de commande F3 et F4 vaudront 1 respectivement dans les cas où la machine à états finis se trouve à l'état S_2 et que le signal de contrôle A ou B vaut 0, c'est-à-dire lors du deuxième cycle où la condition (not(A) + not(B)) est vraie. Ceci correspond parfaitement au comportement désiré (Figs.4.5 & 4.6). La porte OU indiquée dans la figure 4.9 assure l'exécution de l'opération concernée quand l'un des signaux F3 ou F4 est vrai. L'inverseur utilisé génère le signal not(F3+F4) du fait que chaque interrupteur X est en fait implémenté par un transistor CMOS dont la grille du transistor NMOS est alimentée par un signal (F3+F4 dans cet exemple) et celle du transistor PMOS par le complément du signal (not(F3+F4) dans cet exemple).

Ainsi, à partir de la génération de la machine à états finis implémentant la partie de contrôle du sous-système considéré et la génération d'un fichier décrivant les nœuds (source, grille et drain) de tous les transistors de passage utilisés pour contrôler le flux des données, le

protocole de communications entre la partie opérative et sa partie de contrôle est totalement défini. Un exemple plus complet que celui de la figure 4.9 est celui indiqué à la figure 4.10.

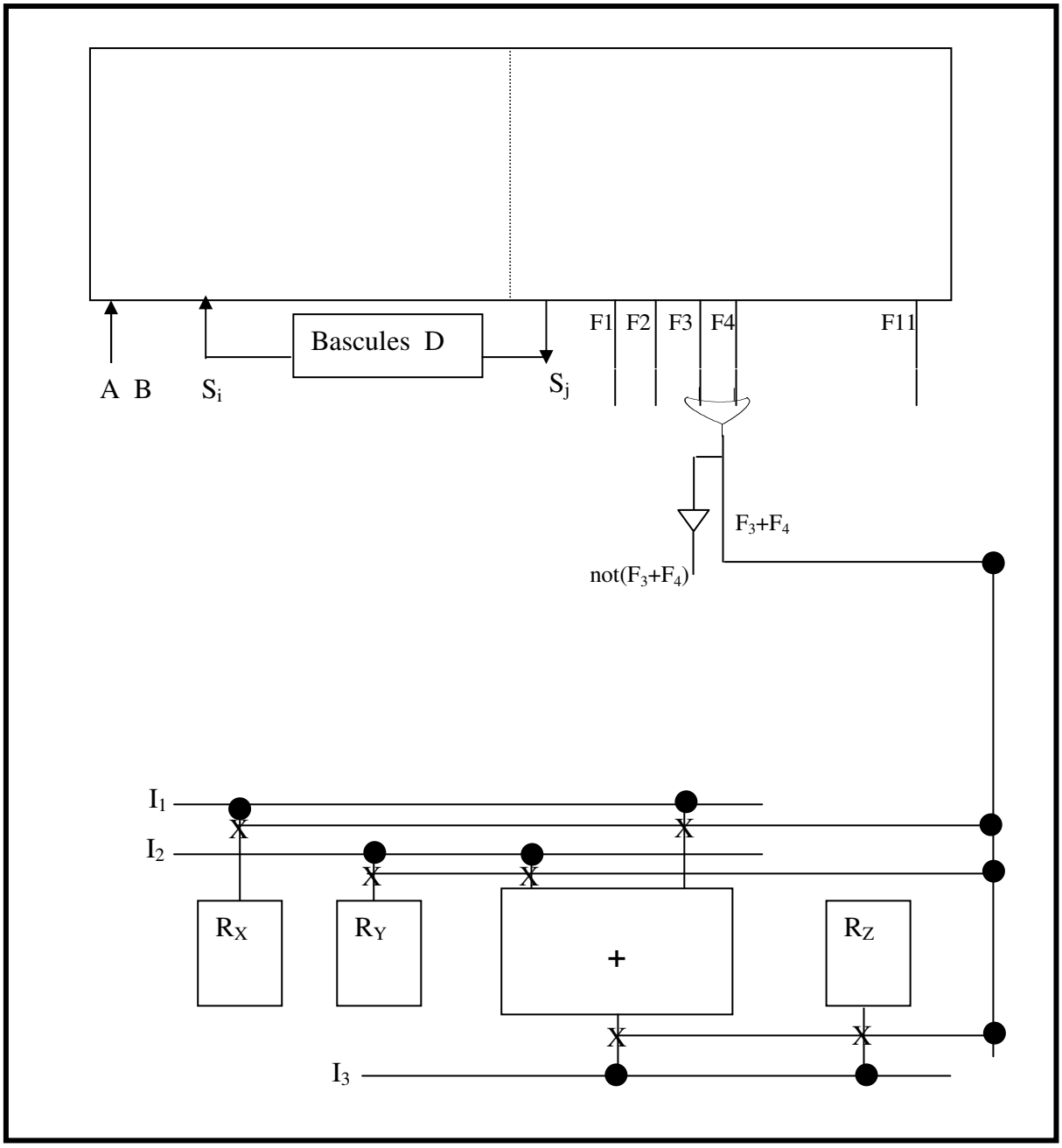

Fig.4.9. Exemple de contrôle de transfert de données

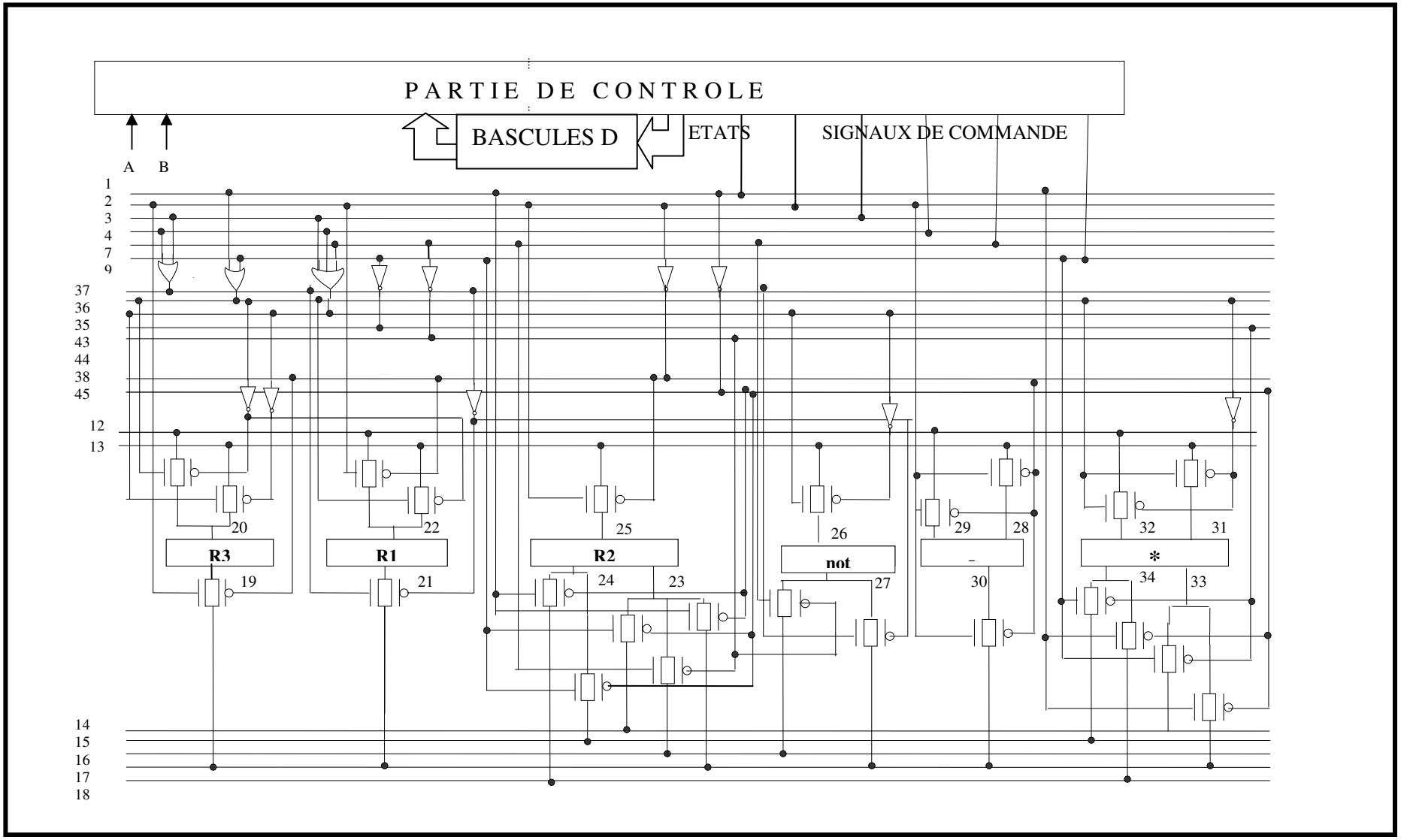

Fig.4.10. Exemple d'interface entre une partie opérative et une partie de contrôle d'un sous-système VLSI

Enfin, notons que le protocole de communications entre les différents sous-systèmes au niveau d'abstraction *transfert de registres* découle de celui déterminé au niveau d'abstraction *système* (section 3.4.3) auquel des transistors de passage sont introduits et dont les grilles sont contrôlées par des signaux déterminés tels que décrit précédemment. A partir du fichier d'allocation des ressources physiques issu lui-même du fichier d'ordonnancement où tout conflit éventuel dû à l'utilisation simultanée du bus global est réglé (ordonnancement exécuté avec ajout de contraintes pour les transferts de données concernés), l'automate, puis la table des commandes sont successivement générées pour permettre de transférer correctement des données à partir des registres ou de la mémoire locale d'un sous-système vers des registres ou la mémoire locale d'un autre sous-système, ou vers la mémoire globale. Ceci est contrôlé par des signaux émanant de la partie de contrôle de chaque sous-système (Fig.4.11 montrant un transfert de données entre les sous-systèmes i et j).

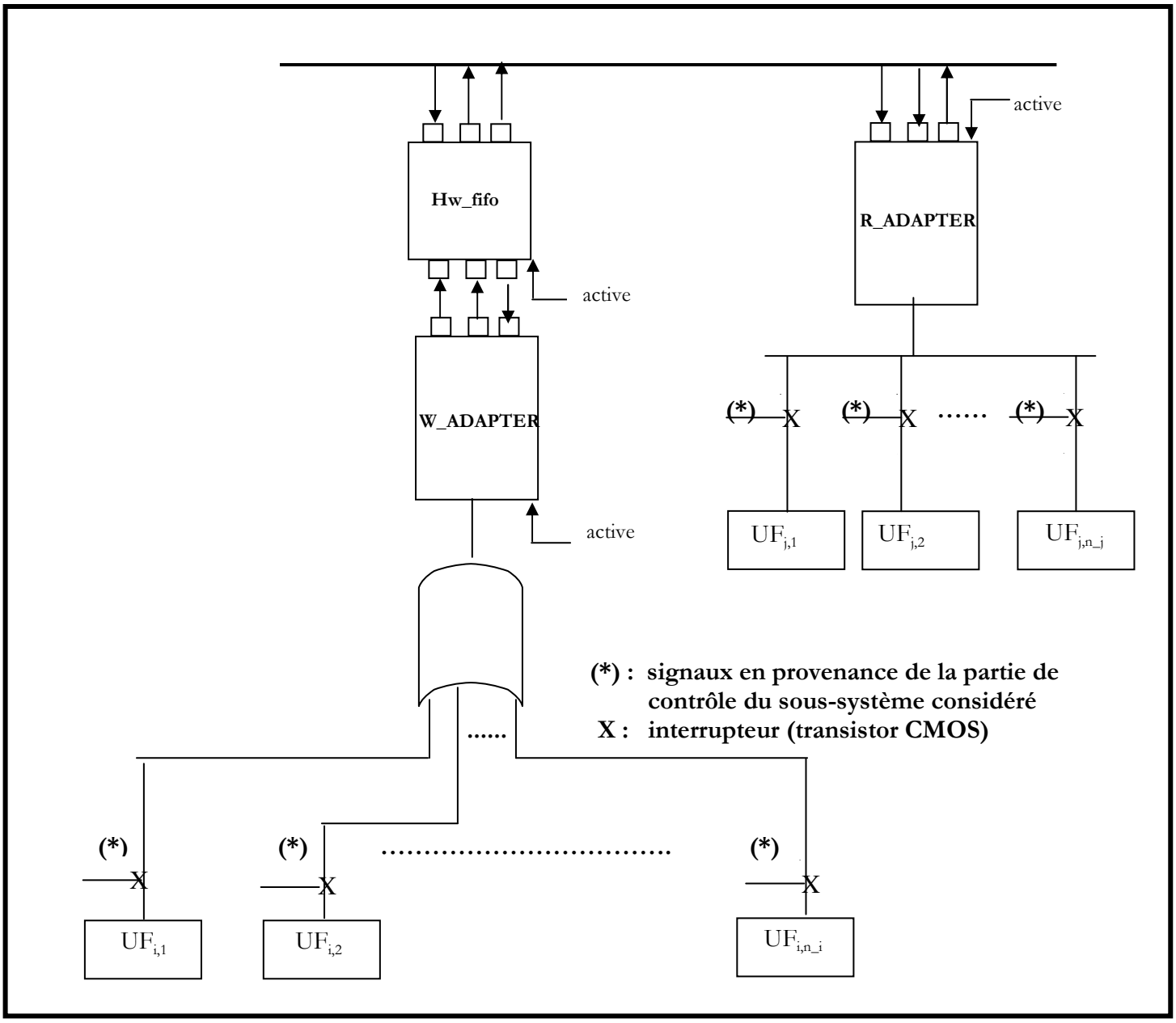

Fig.4.11. Protocole de communications entre différents sous-systèmes

4.3. Analyse :

4.3.1. Surface

Des bibliothèques très riches existent de nos jours et contiennent différentes instances (offrant des caractéristiques différentes) pour chacune des entités les plus couramment utilisées (additionneurs, multiplieurs, comparateurs, ..., voire carrément des microprocesseurs). Ces entités étant caractérisées par leur surface, vitesse et consommation de puissance, une affectation (*mapping*) de ces ressources peut être faite aux différentes unités fonctionnelles de la partie opérative de chaque sous-système pour trouver le compromis recherché.

Dans le cas où la meilleure surface obtenue est jugée importante, il est toujours possible d'introduire des contraintes pour forcer l'outil de synthèse (l'ordonnancement à ce niveau d'abstraction) à ordonnancer certaines opérations indépendantes utilisant le même type de ressource dans des cycles différents. Par exemple, si un ordonnancement donne la possibilité d'exécuter 20 multiplications simultanément, le nombre de multiplieurs doit être égal au moins à 20 – il est égal au maximum de multiplications qu'on peut exécuter durant le même cycle-. En supposant que ce maximum est de 20, il est toujours possible d'introduire des contraintes pour ces 20 multiplications afin de les exécuter en 2 ou plusieurs cycles, réduisant ainsi le nombre de multiplieurs, et du coup la surface. Evidemment, cette réduction de surface doit tenir compte de la performance de la partie opérative, et ce, en cherchant le meilleur compromis possible. Notons qu'un type de traitement similaire à celui-ci a été présenté en 4.2.1, et bien que concernant notamment le compromis vitesse –consommation de puissance, il tient compte aussi de manière indirecte le paramètre *surface*.

4.3.2. Vitesse

L'ordonnancement des opérations effectué à ce niveau d'abstraction donne le nombre de cycles que doit exécuter la partie opérative du sous-système considéré. L'affectation des ressources physiques d'une bibliothèque aux différentes unités fonctionnelles permettrait de connaître la durée d'un cycle. Ainsi, l'exécution des opérations assignées à la partie opérative considérée peut être estimée en unités de temps réelles (ns par exemple). Quoique le temps de transfert des opérations ne peut pas être connu de manière exacte à ce niveau d'abstraction (ce temps n'est connu de manière exacte que lorsque chaque interconnexion est matérialisée par un niveau de masque avec l'insertion éventuelle d'amplificateurs, et en tenant compte de tous les paramètres intrinsèques et parasites – résistances, capacités, inductances -), l'ordonnancement et le *mapping* donnent au concepteur une idée assez précise sur la performance de son système, et peut agir en conséquence pour l'insertion ou la suppression de contraintes en vue d'un autre ordonnancement satisfaisant le compromis désiré.

4.3.3. Consommation de puissance

Afin d'analyser ce paramètre très critique dans les systèmes actuels, nous avions développé un outil d'estimation de la consommation maximale de la puissance dynamique des circuits CMOS, à différents niveaux d'abstraction ([10], [11]). Cet outil a été ensuite amélioré en estimant aussi la consommation dynamique moyenne ([12]). Puis, en adoptant un modèle utilisé pour déterminer les fuites de courant dans un transistor MOS, nous avons généralisé notre outil SPOT pour estimer aussi bien la consommation dynamique que celle due aux fuites du courant.

La consommation de puissance est due principalement:

- aux courts circuits
- à l'activité dynamique
- aux courants de fuite

Courts circuits: Ils étaient très importants avec les technologies NMOS et pseudo-NMOS du fait qu'il y'a un passage de courant permanent entre l'alimentation et la masse pour une certaine

combinaison de signaux même lorsque le circuit est au repos. La technologie CMOS réduit considérablement ces courts circuits du fait de l'utilisation de transistors complémentaires NMOS et PMOS. Notons que les transistors PMOS se situent entre l'alimentation et le nœud de sortie de la porte alors que les transistors NMOS se trouvent entre le nœud de sortie de la porte et la masse. Aussi, ces deux types de transistors ne conduisent pas simultanément (en théorie), ce qui empêche toute circulation de courant entre l'alimentation et la masse. En fait, lors du basculement de signaux sur les grilles des transistors, il existe un instant où chaque transistor NMOS et son transistor PMOS associé conduisent simultanément. Il suffit alors de réduire ce temps de transition des signaux en diminuant les capacités de grille et les tensions de seuil des transistors, et en dimensionnant de manière adéquate les transistors.

Activité dynamique : Elle se produit à chaque charge ou décharge d'une capacité, notamment celle rattachée au nœud de sortie de la porte (Fig.4.12).

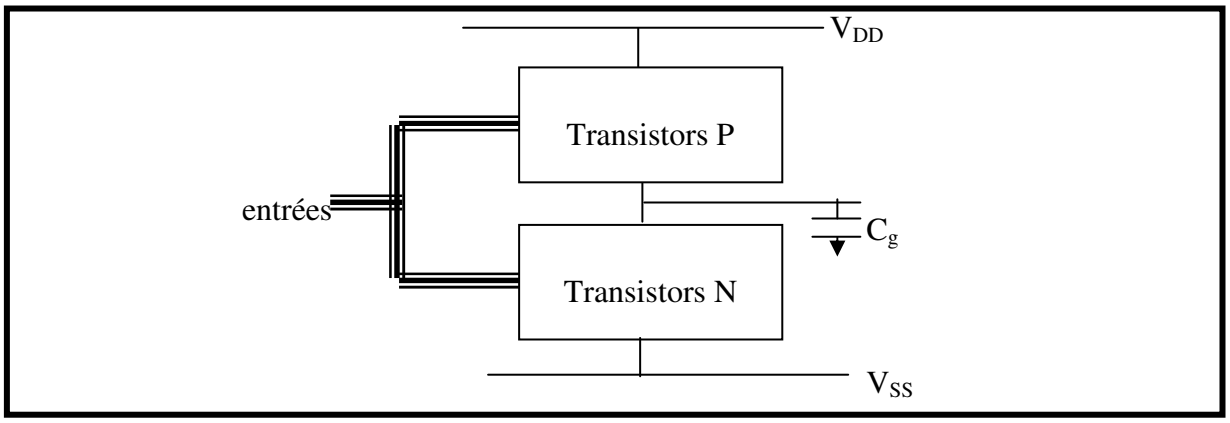

Fig.4.12. Exemple de porte CMOS

Le modèle utilisé est alors :

$$P_{avg} = 0.5 * V_{DD}^2 * f * C_g \quad (4.1)$$

Pour un circuit constitué de plusieurs portes, nous avons :

$$P_{dyn} = 0.5 * V_{DD}^2 * f \sum_{i=1}^{\#Portes} C_{g,i} * N_{g,i} \quad (4.2)$$

où V_{DD} est l'alimentation du circuit, f est sa fréquence de fonctionnement, $C_{g,i}$ est la capacité de charge, et $N_{g,i}$ est le nombre de transitions (charges/décharges de $C_{g,i}$ ou transitions 0->1 / 1->0). La dissipation de puissance dynamique dépend du style de conception puisque celui-ci influe sur certains paramètres de l'équation (4.2). Aussi, avant d'estimer (4.2) finement, nous avions déterminé les bornes des transitions afin d'orienter le concepteur sur le style de conception à adopter.

Circuit statique :
i) L'activité dynamique N dans un circuit statique vérifie :

$$0 \le N \le \sum_{i=1}^{\#Portes} Q_i / q_i$$

où q_i est le temps de propagation de la porte G_i et $Q_i = \sum_{k=1}^{S_i} \max\left(q_{j,k}\right); j = 1,2,...,M_{i,k}$; $M_{i,k}$ étant le nombre de portes au niveau k alimentant directement ou indirectement la porte G_i, S_i est le niveau de G_i.

Preuve: Une séquence d'entrée (V_0,V_0) n'implique aucune transition (N=0).
$1+Q_i/q$ pas sont requis pour stabiliser le nœud de sortie de G_i à une certaine valeur (q est le délai le plus faible parmi ceux des portes alimentant directement ou indirectement G_i). Du fait qu'entre 2 pas successifs G_i peut observer au plus q/q_i transitions, nous avons :

$N_{gi} = (Q_i / q)*(q/q_i) = Q_i / q_i$ ■

ii) Dans le cas où toutes les portes ont le même délai de propagation, le nombre de transitions $N_{g,i}$ que peut observer G_i vérifie: $0 \le N_{gi} \le s_i$; s_i est le niveau de G_i

Preuve: Nous avions vu précédemment que $N_{gi}=Q_i/q_i$. Puisque $q_{j,k}=q \ \forall j, k$, alors $Q_i = s_i*q$, et donc $N_{gi} = (s_i * q)/q=s_i$ ■

iii) Dans le cas où toutes les portes ont le même délai de propagation, l'activité dynamique N vérifie :

$N \le \sum_{k=1}^{S_i} s_i * nb_i$ où s_i est le niveau de G_i; nb_i est le nombre de portes au niveau i.

Preuve : Déduction directe de ce qui précède ($N_{gi}=s_i$) ■

Circuit statique ayant des transistors de passage entre deux niveaux successifs :
iv) L'activité dynamique de toute porte de niveau 1 (supérieur à 1) vaut 1 (2).

Preuve : Soit W_{0i} (W_{ti}) le vecteur alimentant G_i lorsque V_0 (V_t) est appliqué au circuit. W_{ti} est l'ensemble des entrées alimentant G_i telles qu'une entrée donnée est soit une entrée primaire issue de V_t, soit une entrée provenant de la sortie d'une porte G_j alimentant G_i.
Du fait de la présence de transistors de passage, W_{ti} reste inchangé jusqu'à ce que les sorties des portes alimentant G_i se stabilisent à des valeurs formant le vecteur W'_{ti}. Aussi, nous avons comme équations de transitions : $e_{1i}= f_i(W_{0i}) \ xor \ f_i(W_{ti})$ et $e_{2i}= f_i(W_{0i}) \ xor \ f_i(W'_{ti})$

Puisque $W_{ti}=W'_{ti}$ pour les portes de niveau1, nous avons $e_{2i}=0$ pour ces portes. En d'autres termes, les portes de niveau 1 observent 1 seule transition alors que les autres portes en observent 2 ■

Si aucune entrée primaire n'alimente les portes de niveau supérieur à 1, nous avons pour ces portes $W_{0i}=W_{ti}$ et donc $e_{1i}=0$. Du fait que $e_{2i}=0$ pour les portes de niveau 1, alors dans ce cas particulier toutes les portes observent au plus une seule transition.

v) L'activité dynamique (N) dans un circuit statique ayant des transistors de passage entre deux niveaux successifs vérifie $0 \le N \le nb_1 + 2 * \sum_{k=2}^{S} nb_k$; nb_k est le nombre de portes au niveau k ; S est le dernier niveau.

Preuve : Elle découle directement de iv). ■

Circuit statique avec précharge des nœuds du dernier niveau :
La précharge des nœuds du dernier niveau dans certains circuits réduit la consommation de puissance. Dans le cas d'un décodeur d'adresses-mémoire à n entrées, la précharge à 0 des 2^n sorties suivie par une précharge à 1 du nœud correspondant à l'adresse sélectionnée permet de ne décharger qu'une seule capacité au dernier niveau du décodeur. Nous avons:

vi) L'activité dynamique (N) pour ce type de circuits vérifie: $0 \le N \le N_S + 1 + \sum_i Q_i / q_i$ où S est le dernier niveau, Q_i et q_i sont tels que définis dans i), N_S est le nombre de portes du dernier niveau.

Preuve : Soit V_0 le vecteur d'entrées ayant précédemment alimenté le circuit. Les nœuds du dernier niveau étant préchargés à 0 (1), N_S est alors l'activité dynamique maximale des portes du dernier niveau. Quand le nouveau vecteur V_t est injecté au circuit, une seule capacité au dernier niveau est chargée (déchargée). Concernant les autres niveaux, l'activité dynamique maximale est $M = \sum_i Q_i / q_i$ -voir i)- . Au total, l'activité dynamique maximale est N_S+1+M. ■

Circuit dynamique :
vii) Dans un circuit dynamique à 4 phases d'horloge, chaque porte observe au plus $1+\lceil (S-i)/2 \rceil$ transitions. S est le dernier niveau et i est le niveau de la porte considérée.

Preuve : Posons $t_{p,h}=t_{e,h}=\varnothing_{jk} = t$ (c'est-à-dire que le temps de précharge du nœud de sortie de la porte h est égal au temps de l'évaluation) ; $1 \le j \le 4$; $k = (j \% 4) + 1$; $j \% 4$ dénote le reste de la division $j/4$. Pour simplifier la démonstration et sans perte de généralité, supposons qu'il n'existe qu'une seule porte G_i à chaque niveau i. Supposons que la précharge du nœud de sortie de G_1 commence à $t'=t_0$. Dans ce cas, G_i ($1 \le i \le S$) continue à précharger à $t_{bi}=t_0+(i-1)t/2$ (le circuit est à 4 phases d'horloge). Par conséquent, la valeur au nœud de sortie de la porte G_S est disponible à $t_S = t_0+(S-1)t/2 + 2t$. Ainsi, G_i ($1 \le i \le S$) continue à précharger et à évaluer durant $t'_S=t_S-t_{bi}=(S-i)t/2 + 2t$. Comme $t_{pi} = t_{ei} = t$, le nombre de précharges et d'évaluation pour G_i est $t'_S/t = 2 + (S-i) / 2$. Si on suppose que G_i fait une transition lors de la première évaluation de son nœud de sortie, le nombre de transitions pour la porte G_i est alors $\lceil (S-i)/2 \rceil + 1$. ■

viii) Dans un circuit dynamique à 4 phases d'horloges, l'activité dynamique N vérifie : $0 \le N \le 2|G| + \sum_{i=1}^{S} nb_i * \lceil S-i \rceil / 2$. S et nb_i tels que définis précédemment, G est l'ensemble des portes.

Preuve : Supposons que chaque porte observe une transition après la précharge de son nœud de sortie. Le nombre de telles transitions serait alors égal à $|G|$. De vii), il découle que le nombre de transitions dues à l'évaluation est égal à $\sum_{i=1}^{S} \left(1+\lceil (S-i)/2 \rceil\right) * nb_i = |G| + \sum_{i=1}^{S} \lceil (S-i)/2 \rceil * nb_i$, soit un total de transitions égal à $2|G| + \sum_{i=1}^{S} \lceil (S-i)/2 \rceil * nb_i$. ■

Le nombre de transitions déterminé précédemment pour certains styles de conception usuels est une borne maximale permettant de guider le concepteur quant au choix du style de conception à adopter. Toutefois, la valeur exacte du nombre de transitions que peut observer un circuit stabilisé avec un vecteur V_0 puis alimenté par un vecteur V_t demande un traitement plus

conséquent. Supposons que 3 entrées primaires alimentent un circuit. Connaître la puissance dynamique maximale consommée consiste à déterminer pour chaque porte, le nombre de transitions $N_{g,i}$ résultant de l'état actuel où se trouve le circuit (un vecteur V_0 alimentant le circuit) à l'état où un nouveau vecteur V_t est injecté au circuit. Pour un circuit à 3 entrées primaires par exemple, il faudrait injecter au circuit :
000 puis 000
000 puis 001
.....
000 puis 111
.....
111 puis 111

Pour chacune de ces 2^{2*3}=64 combinaisons, il faudrait déterminer en fonction du comportement de chaque porte et du style de conception du circuit, le nombre de transitions $N_{g,i}$ qu'observe cette porte, puis appliquer l'équation (4.2) pour déterminer les 64 dissipations de puissance et sélectionner la plus importante si l'on s'intéressait à la puissance dynamique maximale. De manière générale, ce traitement est fait 2^{2*N} fois pour un circuit à N entrées. Il est clair que cette approche conduit à un temps CPU prohibitif pour les circuits actuels du fait de la complexité algorithmique du problème. Pour ce faire, nous avions développé un algorithme génétique dont les caractéristiques seront énumérées ultérieurement. Le point de départ du développement de notre algorithme a été le suivant :

ix) La paire de vecteurs (V_0,V_0) n'implique aucune transition.

Preuve : Soit S_{i0} la valeur au nœud de sortie de G_i lorsqu'on applique V_0 au circuit. Lorsque le nouveau vecteur V_0 est appliqué au circuit, la valeur au nœud de sortie de G_i reste égal à S_{i0}. L'équation de transition est alors $e_i=S_{i0}$ *xor* $S_{i0}=0$. ■

En d'autres termes, si les 2 vecteurs successifs (V_0,V_t) ne diffèrent d'aucun bit, alors il n'y'a aucune transition, et donc aucune dissipation de puissance dynamique. Mais qu'en est t-il si les deux vecteurs diffèrent d'1 bit, ou de 2 bits ? etc .... Nous avions alors défini N classes C_1, C_2, ..., C_N (N étant le nombre d'entrées primaires) où chaque paire de vecteurs $(V_0,V_t) \in C_i$ si et seulement si V_0 et V_t diffèrent de i bits exactement. Noter que C_0 n'existe pas puisqu'on a vu que les paires de vecteurs de cette classe n'impliquent aucune transition. Afin de considérer des éléments représentatifs, nous avions décomposé chaque classe C_i en 2^i sous-ensembles, et les paires de vecteurs considérées sont : $(W_0,W_t)=\left(V_{0,S_{ikj}},V_{t,S_{ilj}},\right)$ où S_{ik} est le $k^{ième}$ sous-ensemble dans C_i ; $1 \le k \le 2^i$; $l=2^i-k+1$; $V_{0,S_{ikj}}\left(V_{t,S_{ilj}}\right)$ est le $j^{ième}$ vecteur dans S_{ik} (S_{il}) ; $1 \le j \le 2^{N-i}$. Ainsi, C_1, C_2, ..., C_N nous permettent de considérer un bon nombre de vecteurs représentatifs. Toutefois, pour une meilleure exploration de l'espace des paires de vecteurs, nous avions défini une classe additionnelle C_X qui contient les paires (V_{0j}, V_{tm}) telles que : $1 \le j \le 2^N$; $m= j+1\ (j-1)$ si j est impair (pair). La figure 4.13 montre les différentes classes pour N=3.

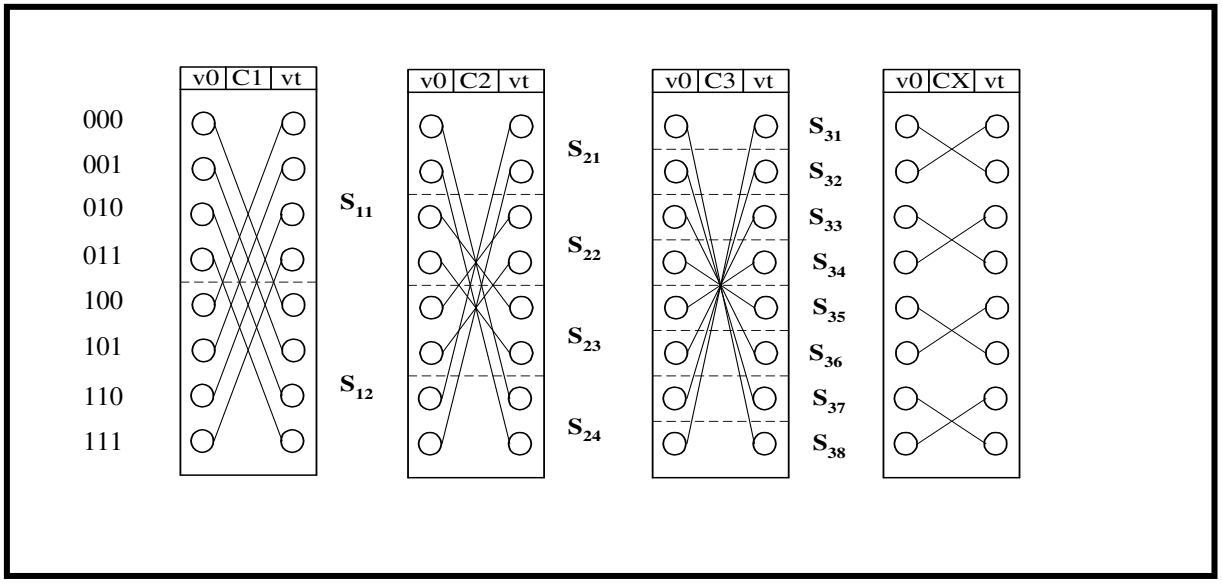

Fig.4.13. Opérations génétiques sur des vecteurs

Par exemple, $\left(V_{0,S_{111}}, V_{t,S_{121}}\right)$ vaut (000,100) du fait que $V_{0,S_{111}}$ est le 1er vecteur V_0 de S_{11} et que $V_{t,S_{121}}$ est le 1er vecteur V_t de S_{12}.

Une population de (N+1) paires de vecteurs est sélectionnée des classes C_1, C_2, ..., C_N et C_X à chaque génération. Pour les circuits où les entrées primaires n'alimentent que les portes du premier niveau, les paires de vecteurs $\left(V_{0,S_{ikj}}, V_{t,S_{ilj}}\right) \in C_i$ tels que $1 \le i \le N$; $2^{i-1}+1 \le k \le 2^i$ ainsi que les paires $\left(V_{0,j}, V_{0,m}\right) \in C_X$ telles que j est pair ne sont pas sélectionnées en vertu de ce qui suit :

x) (V_0, V_t) implique le même nombre de transitions que (V_t, V_0) pour les circuits où les entrées primaires n'alimentent que les portes du premier niveau.

Preuve : Soit g une porte quelconque et S_g son niveau. (V_0, V_t) donne la transition d'équation $e_{g,1}=f(W_0)$ *xor* $f(W_t)$ où f est le comportement de g et

$$W_0(W_t)\begin{cases} \subseteq V_0(V_t) \; si \; S_g = 1 \\ = \{o; o \; est \; la \; sortie \; \text{de g}' ; g' \; \text{alimente} \; g\} \end{cases}$$

Avec (V_t, V_0), la transition d'équation est : $e_{g,2}=f(W_t)$ *xor* $f(W_0)=e_{g,1}$ ■

Ainsi, pour accélérer le traitement, il est inutile de considérer toute paire (V_t, V_0) si l'estimation de la consommation de la puissance dynamique a été faite avec (V_0, V_t) du fait que ces paires induisent la même consommation de puissance dynamique.

Dans le cas où il n'est pas possible d'explorer toutes les paires de vecteurs construites dans une classe donnée, l'exploration dans les classes C_1, C_2, ..., C_N se fait alternativement dans les sous-ensembles $S'_{i,1} = \bigcup_{r=1}^{2^{i-1}} S_{i,r}$ et $S'_{i,2} = \bigcup_{r=2^{i-1}+1}^{2^i} S_{i,r}$; i étant le numéro de la classe. Pour la classe C_X, les vecteurs $(V_{0,j}, V_{t,m})$ sont déterminés tels que $j = n\,(n+2^{N-1})$ si n est impair (pair) ; $m = j+1\,(j-1)$ si j est impair (pair) ; n est le numéro de la génération et N le nombre d'entrées primaires. Notons

que *n*, le nombre de générations, est choisi en tenant compte du compromis sur la qualité de la solution et le temps CPU.

Cet algorithme a été testé avec succès sur des benchmarks ISCAAS et présente différentes caractéristiques auxquelles un algorithme génétique doit se conformer ([30]) :

- les éléments non significatifs dans le contexte du problème ne sont pas générés. Par exemple, dans le cas du problème du voyageur du commerce, il serait illusoire de générer un chromosome du type CBADEB.... du fait que la ville B est traversée deux fois. Dans notre cas, les paires de vecteurs (V_0 , V_t) sont générées adéquatement du fait que les longueurs des vecteurs ne peuvent être différentes du nombre d'entrées primaires.
- Une génération aléatoire peut alourdir le traitement en traitant de manière redondante les mêmes chromosomes pendant un certain nombre de générations. Dans notre cas, ceci ne peut se produire du fait que les paires de vecteurs sont définies à l'avance et que leur exploration se fait selon un mécanisme bien déterminé.
- Une génération aléatoire de chromosomes peut retarder la convergence vers une solution d'une certaine qualité. Dans notre cas, toutes les paires de vecteurs (V_0 , V_0) ne sont pas considérées du fait qu'elles n'entraînent aucune consommation de puissance dynamique. Pour un circuit à N entrées primaires, il y'a 2^N paires (V_0 , V_0). De plus, pour les circuits où seules les portes du premier niveau sont alimentées par des entrées primaires, nous avons vu qu'il serait inutile de considérer (V_t , V_0) si (V_0 , V_t) l'a été. En faisant cette observation, nous évitons d'explorer inutilement $2*2^N = 2^{N+1}$ paires de vecteurs.
- Une génération aléatoire peut conduire à un effet stationnaire, c'est à dire que la solution ne peut pas se détacher d'un ensemble d'optimums locaux. Dans notre cas, cet effet stationnaire ne peut se produire du fait qu'à chaque génération aucune paire déjà explorée n'est reconsidérée.
- Une caractéristique propre à notre algorithme est que même les chromosomes ayant amélioré la solution sont supprimés de la population, et ce, à chaque génération. La taille de la population étant fixée, ceci permet de traiter de nouveaux éléments à chaque génération, ce qui permet une meilleure exploration de l'espace des solutions.

Cet outil qui a été initialement développé pour l'estimation de la consommation de la puissance dynamique d'un module a été généralisé par la suite au cas de parties opératives. L'algorithme est celui indiqué dans la figure 4.14 et traite le cas où plusieurs unités fonctionnelles opèrent en parallèle.

Enfin, SPOT a été généralisé pour estimer, outre la consommation dynamique maximale, la consommation dynamique moyenne ainsi que la consommation due aux fuites du courant. Notons que ce dernier type de puissance était négligeable dans les anciennes technologies, mais représente dans les technologies courantes un pourcentage assez important de la consommation totale. Notre estimation de ce type de consommation est basée sur le modèle intégré dans le logiciel BACPAC, développé à Berkeley. Ce modèle est celui indiqué par l'équation (4.3).

```
Début
  { Power = - ∞ ;      list_vectors = Φ ;
   for i=1 to nb_vectors  /* nb_vectors est le nombre maximal de paires de vecteurs explorées */
   do {déterminer (v0,vt) en utilisant les classes C1, C2, …, CN et CX
       P= - ∞ ;
       foreach cycle c
       do {déterminer les unités fonctionnelles f1 ,f2 ,.....,fm qui exécutent durant c ;
           for i=1 to m
           do calculer Pi , la puissance dissipée quand (v0,vt) alimente fi ,
           end
           Pc = Σ_{i=1}^{m} Pi ;
           if (P < Pc)
           then P = Pc ;
           endif
          }
       end
       if (Power < P)
       then {Power = P ; list_vectors = Φ ;
             list_vectors ← (v0,vt) ;
            }
       else if (Power == P)
            then list_vectors = list_vectors ∪ (v0,vt) ;
            endif
       endif
      }
  end
  imprimer les paires de vecteurs impliquant une consommation de puissance dynamique
            égale à Power ;
  imprimer Power ;
 }
Fin
```

Fig.4.14. Estimation de la puissance dynamique maximale dans une partie opérative

$$P_{leak} = 0.28125 * V_{DD} * K * Nb_{trans} * W_{avg} * L * e^{-V_t / \alpha_V} \qquad (4.3)$$

où : - P_{leak} est la consommation due aux fuites du courant
- V_{DD} est la tension d'alimentation
- K=10 μA/μm
- Nb_{trans} est le nombre de transistors dans le circuit
- W_{avg} est la largeur moyenne des transistors
- L est la longueur du canal (en μm) du transistor dans la technologie considérée
- V_t est la tension de seuil initiale du transistor
- α_V = 0.095 V

Observant qu'une tension d'alimentation importante améliore la performance du circuit (charges et décharges de capacités rapides) mais au détriment de la consommation de puissance et qu'une tension de seuil faible d'un transistor fait commuter celui-ci rapidement mais au détriment d'une consommation due aux fuites du courant, une alternative peut être trouvée. Afin de réaliser un bon compromis entre la vitesse et la consommation de puissance, les technologies actuelles utilisent dans le même circuit deux tensions d'alimentation et deux tensions de seuil pour chaque type de transistor. Aussi, nous avions transformé pour SPOT, les équations (4.2) et (4.3) comme suit :

$$P_{dyn} = 0.5 * f * \left[V_{DD,L}^2 \sum_{i=1}^{|E_L|} C_{g,L,i} * N_{g,L,i} * V_{DD,H}^2 \sum_{i=1}^{|E_H|} C_{g,H,i} * N_{g,H,i} \right] \quad (4.4)$$

où : - P_{dyn} est la puissance dynamique du circuit
- f est la fréquence de fonctionnement du circuit
- E_L est l'ensemble des portes alimentées par $V_{DD,L}$
- E_H est l'ensemble des portes alimentées par $V_{DD,H}$
- $C_{g,L,i}$ est la capacité de charge de la i^{ème} porte de E_L
- $N_{g,L,i}$ est le nombre de charges/décharges de $C_{g,L,i}$
- $C_{g,H,i}$ est la capacité de charge de la i^{ème} porte de E_H
- $N_{g,H,i}$ est le nombre de charges/décharges de $C_{g,H,i}$

et

$$P_{leak} = 0.28125 * K * W_{avg} * L * \sum_{i=1}^{\#Portes} \left(Nb_{N,i} * 10^{-V_{t,N,i}/\alpha_V} + Nb_{P,i} * 10^{V_{t,P,i}/\alpha_V} \right) * V_{DD,i} \quad (4.5)$$

où :
- K, W_{avg} et L sont tels que définis précédemment
- $Nb_{N,i}$ est le nombre de transistors de type N dans la porte i
- $V_{t,N,,i}$ est la tension de seuil des transistors de type N dans la i^{ème} porte
- $Nb_{P,i}$ est le nombre de transistors de type P dans la porte i
- $V_{t,P,i}$ (< 0) est la tension de seuil des transistors de type P dans la i^{ème} porte
- $V_{DD,i}$ est la tension alimentant la porte i

4.4. Vérification :

Pour ce niveau d'abstraction, il existe actuellement sur le marché de nombreux outils de vérification pour vérifier la configuration obtenue moyennant les tâches de synthèse et d'analyse, et ce, avant que le concepteur ne décide de passer ou pas à un autre niveau d'abstraction (le niveau *module*).

4.5. Conclusion :

Nous avions présenté dans ce chapitre nos méthodes de synthèse et d'analyse appliquées à chacun des sous-systèmes obtenus à la première étape de synthèse. Ces méthodes ont été validées soit par des comparaisons à des résultats exacts (quand c'est possible), soit à des résultats publiés en utilisant des Benchmarks, comme nous le verrons dans le prochain chapitre.

Chapitre V

Résultats

5.1. Introduction :

Nous présentons dans ce chapitre les résultats obtenus pour les étapes importantes. Celles-ci concernent la décomposition du système en sous-systèmes VLSI, la synthèse des tâches affectées aux différents sous-systèmes VLSI, l'ordonnancement d'opérations d'une partie opérative assujetties à la contrainte de consommation de la puissance, la synthèse de parties de contrôle, l'interface de communications entre une partie opérative et sa partie de contrôle, et enfin l'estimation de la consommation de la puissance.

Ces résultats ont été pour la plupart obtenus sur une plateforme à base de processeurs de type Pentium 3 ou Pentium 4 s'exécutant sous le système d'exploitation Linux. Le langage de programmation utilisé est soit C, soit C++ dans le cas où une programmation orientée-objet était nécessaire.

5.2. Décomposition d'un système en sous-systèmes et affectation de tâches :

Comme nous l'avons vu, ce traitement se fait en deux étapes : un *macro-partitionnement* déterminant le nombre de sous-systèmes obtenus par décomposition du système VLSI, et un *micro-partitionnement* affectant les tâches du même sous-système aux différents composants (ASICs, processeurs) de ce sous-système. La table 5.1 montre les variations des caractéristiques obtenues pour différentes configurations du même système. Le graphe de tâches utilisé comme fichier de données a été généré de manière aléatoire en utilisant [29], [30].

Table 5.1. EXEMPLE DE CONFIGURATIONS POUR LE MEME SYSTEME VLSI

# Configuration	# Sous-systèmes	Temps (µs)	Consommation d'énergie (nJ)
1	15	755	735
2	10	815	344
3	6	870	197
4	4	1290	118

Les valeurs reportées dans la table 5.1 sont estimées à partir des équations établies dans les sections 3.5.2 et 3.5.3. Notons que l'énergie dissipée pour une configuration donnée tient compte de celle dissipée par les différents sous-systèmes et de celle due aux transferts de données entre les différents sous-systèmes. Le premier type d'énergie diminue avec le nombre de sous-systèmes (car le parallélisme est de plus en plus réduit) alors que le deuxième type d'énergie augmente avec la diminution du nombre de sous-systèmes (plus le nombre de sous-systèmes est réduit et plus le problème de couplage est accentué). Ainsi, à partir d'un certain nombre de configurations, le concepteur peut choisir celle qui répond le mieux à son application.

5.3. Synthèse de tâches:

Une étude comparative formelle de notre méthode ([27]) au cas idéal a été faite au troisième chapitre. Nous donnons ci-après les résultats obtenus pour les données indiquées dans la table 5.2 où M_i, O_i et D_i représentent respectivement les nombres d'opérations obligatoires et optionnelles, et la date limite d'achèvement de la tâche i. Noter que la durée d'exécution d_i d'une tâche i est égale à D_i-D_{i-1} (D_0=0 ; i ≠ 2) alors que d_2= D_2-d_1-$\delta^{sel}_{O\_T}$ (comme nous l'avons vu, la procédure de sélection est exécutée immédiatement après l'achèvement de la première tâche). Notre affectation V/O pendant la phase de conception est celle indiquée dans la table 5.3. Cette affectation maximise la fonction objective tout en satisfaisant les contraintes de temps et de consommation d'énergie. Il est supposé que la tension à affecter se situe dans l'intervalle [V_{min}=1V,V_{max}=3.3V]. Nous supposons aussi que le budget d'énergie alloué E_{max} est de 370 pJ, et que $\varepsilon^{sel}_{O\_T}$ et $\delta^{sel}_{O\_T}$ valent 2.20 pJ et 400 ns, respectivement. Ces valeurs sont déduites des équations (3.1) et (3.3) en remplaçant M_i+O_i par $\lceil log_2 Nb\_traces \rceil$+*2*Nb_tasks*. Noter que $\lceil log_2 Nb\_traces \rceil$ est due à la détermination de la trace qui est en cours d'exécution alors que *2*Nb_tasks* est dû à l'affectation V/O pour chaque tâche. Dès que la première tâche se termine, notre méthode détecte la trace qui est en cours d'exécution et procède en conséquence aux affectations V/O adéquates. La table 5.4 indique l'affectation V/O pendant l'exécution de l'application. Les résultats montrent que :

- Chaque tâche se termine avant le délai de son achèvement
- Pour chaque trace, l'énergie consommée est inférieure au budget d'énergie alloué
- Le nombre d'opérations optionnelles est maximal pour chaque tâche (à l'exception de la première) pour n'importe quelle trace
- Bien que notre affectation V/O soit déterminée d'abord durant la phase de conception, la valeur de $O_{1,t}$ est égale ou très proche de $O_{1,t,max}$
- La différence entre le temps d'achèvement d'une tâche et son délai d'achèvement est exploitée pour traiter les tâches restantes. Ainsi, la satisfaction de la contrainte de temps est augmentée
- Notre outil tient compte du cas où certaines tâches ne sont pas exécutées pour des raisons données (cases vides)

Les erreurs relatives (indiquées dans la table 5.5) de notre méthode par rapport au cas idéal ne dépassent pas 0.08%. Ces erreurs sont obtenues en appliquant (3.31) pour la trace 3 et (3.32) pour les autres traces. Noter que $E_{I\_C,T}$ représente l'énergie consommée par l'approche idéale.

Table 5.2. EXEMPLE DE DONNEES

Task	Trace 1			Trace 2			Trace 3		
	M_i	$O_{i,max}$	D_i (µs)	M_i	$O_{i,max}$	D_i (µs)	M_i	$O_{i,max}$	D_i (µs)
1	100	90	1.90	128	95	2.15	78	41	1.21
2	80	20	3.30	65	17	3.37	70	17	2.48
3	60	30	4.20	62	30	4.29	61	30	3.39
4	540	236	11.96	516	201	11.46	520	210	10.69
5	75	20	12.91	200	80	14.26	150	45	12.64
6	110	10	14.11	110	10	15.46	110	10	13.84
7	97	15	15.23	90	8	16.44	145	27	15.56
8	136	23	16.82	140	25	18.09	112	14	16.82
9	608	202	24.92	200	74	20.83	706	218	26.06
10	527	117	31.36	600	120	28.03	602	120	33.28
11	26	4	31.66	200	93	30.96	150	61	35.39
12	75	17	32.58	20	4	31.20			
13	36	8	33.02	146	51	32.50	117	26	36.83
14	29	3	33.34	377	92	37.19	200	63	39.46
15	810	205	43.49	810	205	47.34			
16	631	161	51.42	200	23	49.57	350	57	43.50
17	28	2	51.72	28	2	49.97			
18	417	113	57.02	184	67	52.38	209	36	45.95
19	809	300	68.11	295	46	55.79			
20	19	3	68.33	216	38	58.33	168	42	48.05

TABLE 5.3. AFFECTATION V/O PENDANT LA PHASE DE CONCEPTION

Task	Trace 1				Trace 2				Trace 3			
	O_i	V_i (V)	D_i (µs)	E_i (pJ)	O_i	V_i (V)	D_i (µs)	E_i (pJ)	O_i	V_i (V)	D_i (µs)	E_i (pJ)
1	90	1.13	1.70	9.70	94	1.12	2.01	11.14	41	1.11	1.09	5.86
2	20	1.12	3.00	5.14	17	1.14	3.15	4.17	17	1.13	2.27	4.41
3	30	1.13	3.81	4.61	30	1.13	3.97	4.69	30	1.13	3.08	4.67
4	236	1.12	10.87	38.99	201	1.13	10.43	36.71	210	1.12	9.71	36.67
5	20	1.15	11.72	5.18	80	1.12	12.96	14.16	45	1.13	11.45	10.00
6	10	1.12	12.82	6.12	10	1.12	14.05	6.12	10	1.13	12.54	6.04
7	15	1.12	13.83	5.65	8	1.14	14.93	4.96	27	1.12	14.10	8.70
8	23	1.13	15.27	8.05	25	1.13	16.42	8.44	14	1.13	15.24	6.46
9	202	1.12	22.62	40.64	74	1.13	18.88	14.01	218	1.12	23.63	46.41
10	117	1.13	28.41	32.93	120	1.12	25.43	36.15	120	1.12	30.19	36.24
11	4	1.15	28.68	1.51	93	1.13	28.06	15.20	61	1.13	32.08	10.85
12	17	1.13	29.51	4.76	4	1.10	28.28	1.39				
13	8	1.10	30.02	3.47	51	1.85	30.95	29.66	26	1.11	33.69	11.31
14	3	1.14	30.31	1.60	92	1.13	35.16	24.04	63	1.13	36.05	13.41
15	205	1.12	39.53	50.97	205	1.12	44.38	50.97				
16	161	1.12	46.70	40.04	23	1.12	46.40	11.30	57	1.15	39.70	22.20
17	2	1.08	46.99	1.51	2	1.15	46.67	1.48				
18	113	1.13	51.77	27.16	67	1.12	48.94	12.70	36	1.12	41.93	12.33
19	300	1.12	61.84	55.72	46	1.13	52.02	17.51				
20	3	1.21	62.04	1.44	38	1.13	54.31	13.23	42	1.12	43.86	10.87
	Total Energy (pJ)											
	$2.20 + \sum E_i$			347.39				320.23				248.63

TABLE 5.4. AFFECTATION V/O PENDANT LA PHASE D'EXECUTION

Task	Trace 1				Trace 2				Trace 3			
	O_i	V_i (V)	D_i (µs)	E_i (pJ)	O_i	V_i (V)	D_i (µs)	E_i (pJ)	O_i	V_i (V)	D_i (µs)	E_i (pJ)
1	89	1.12	1.71	9.48	93	1.12	1.98	11.09	41	1.12	1.06	5.97
2	20	1.12	3.01	5.14	17	1.14	3.12	4.17	17	1.13	2.24	4.41
3	30	1.13	3.82	4.61	30	1.13	3.94	4.69	30	1.13	3.05	4.67
4	236	1.12	10.88	38.99	201	1.13	10.40	36.71	210	1.12	9.68	36.67
5	20	1.15	11.73	5.18	80	1.12	12.93	14.16	45	1.13	11.42	10.00
6	10	1.12	12.83	6.12	10	1.12	14.02	6.12	10	1.13	12.51	6.04
7	15	1.12	13.84	5.65	8	1.14	14.90	4.96	27	1.12	14.07	8.70
8	23	1.13	15.28	8.05	25	1.13	16.39	8.44	14	1.13	15.21	6.46
9	202	1.12	22.63	40.64	74	1.13	18.85	14.01	218	1.12	23.60	46.41
10	117	1.13	28.42	32.93	120	1.12	25.40	36.15	120	1.12	30.16	36.24
11	4	1.15	28.69	1.51	93	1.13	28.03	15.20	61	1.13	32.05	10.85
12	17	1.13	29.52	4.76	4	1.10	28.25	1.39				
13	8	1.10	30.03	3.47	51	1.85	30.92	29.66	26	1.11	33.66	11.31
14	3	1.14	30.32	1.60	92	1.13	35.13	24.04	63	1.13	36.02	13.41
15	205	1.12	39.54	50.97	205	1.12	44.35	50.97				
16	161	1.12	46.71	40.04	23	1.12	46.37	11.30	57	1.15	39.67	22.20
17	2	1.08	47.00	1.51	2	1.15	46.64	1.48				
18	113	1.13	51.78	27.16	67	1.12	48.91	12.70	36	1.12	41.90	12.33
19	300	1.12	61.85	55.72	46	1.13	51.99	17.51				
20	3	1.21	62.05	1.44	38	1.13	54.28	13.23	42	1.12	43.83	10.87
	Total Energy (pJ)											
	$2.20 + \sum E_i$			347.17				320.18				248.74

TABLE 5.5. ERREURS RELATIVES POUR L'EXEMPLE DE LA TABLE 5.2 EN SUPPOSANT $E_{MAX} \geq E_{I\_C,T} + \varepsilon^{SEL}$

	# Opérations Optionnelles		
	Trace 1	**Trace 2**	**Trace 3**
Cas idéal	1579	1280	1017
Notre Technique	1578	1279	1017
Erreur relative (%)	0.06	0.08	0.00

Dans la table5.6, l'erreur relative ne dépasse pas 3.18%. Du fait que $E_{I\_C,max}$=344.97 pJ, $E_{I\_C} + \varepsilon^{sel} > E_{max}$ ne se réalise que pour la première trace (E_{O_T}=347.17 pJ). Les erreurs relatives pour les autres traces s'obtiennent à partir des équations (3.31) et (3.32) - $E_{max} \geq E_{I\_C} + \varepsilon^{sel}$ – et sont aussi montrées dans la table 5.4. La table 5.7 montre que nous obtenons des résultats intéressants en des temps CPU raisonnables pour des graphes générés automatiquement par *TGFF* ([31], [32]) et adaptés au problème considéré. *Avg_M* (*Avg_O*) est le nombre moyen des cycles d'exécution des opérations obligatoires (optionnelles), c'est à dire la somme de ces cycles en considérant toutes les traces, puis en divisant par le nombre de traces. En utilisant (3.31), (3.32), (3.34) ou (3.35), l'erreur relative moyenne ne dépasse pas 9.36%, ce qui confirme l'efficacité de notre technique. De plus, du fait que l'approche idéale n'existe pas réellement, notre technique peut constituer une alternative. Noter que le temps CPU correspondant à l'affectation V/O

pendant la phase de conception dépend fortement du nombre de traces qui est $\prod_{i=1}^{N\_Ctr\_V} N_i$, N_i étant le nombre de traces induit par la $i^{ème}$ variable de contrôle, et *N_Ctr_V* le nombre des conditions de contrôle séquentielles. Nous estimons qu'il est plus intéressant de dépenser un temps suffisant en vue d'obtenir des affectations V/O intéressantes plutôt que des affectations très éloignées de la solution optimale en des temps CPU très courts. Ceci est vrai pour autant que le temps CPU dépensé est nettement moins important que la durée de vie du système embarqué, ce qui est le cas. Aussi, les temps CPU indiqués dans la table 5.7 sont raisonnables et acceptables. Il existe une manière de réduire ces temps CPU en affectant les traces à différentes partitions de sorte qu'une partition donnée contienne des traces ayant des caractéristiques (délais, énergies) proches. Puis, au lieu de considérer toutes les traces, il suffirait de traiter une trace de chaque partition et les résultats obtenus pour chaque trace seraient reportés aux traces de la même partition. Malheureusement, ceci affecterait la qualité de la solution. Par contre, notre technique est très rapide pendant la phase d'exécution de l'application du fait que la complexité algorithmique de la sélection est $O(log_2 N)$, N étant le nombre de traces.

Table 5.6. ERREURS RELATIVES POUR L'EXEMPLE DE LA TABLE 5.2 EN SUPPOSANT $E_{MAX}=E_{I\_C,T}$

Trace	Trace 1	Trace 2	Trace 3
Erreur Relative	3.18 %	0.08 %	0.00 %
$E_{max}=E_{I\_C}$ **(pJ)**	347.17 - 2.20 = 344.97	320.18 - 2.20 = 317.98	248.74 - 2.20 = 246.54

Table 5.7. ERREURS RELATIVES POUR DES GRAPHES DE TACHES GENERES ALEATOIREMENT

Task Graph	# Traces	# Tasks	Avg_M	Avg_O	Avg_ Rel_Err (%)	CPU Time
1	10000	700	90000	35000	4.05	15h58'19"
2	900	630	70000	22000	5.75	1h3'32"
3	800	500	300000	25000	4.08	3h19'53"
4	750	820	328000	25000	6.80	3h22'58"
5	630	550	250000	12000	9.23	2h7'37"
6	600	700	150000	15000	9.36	1h16'55"
7	580	530	400000	35000	3.08	3h14'23"
8	550	620	580000	60000	2.67	4h29'58"
9	400	280	620000	50000	1.13	3h28'31"
10	300	410	800000	40000	2.09	3h14'37"
11	200	850	750000	35000	4.88	2h0'51"
12	100	1000	2000	200	4.49	14"

5.4. Ordonnancement d'opérations assujetti à la contrainte de la consommation de la puissance [8] :

L'ordonnancement d'un flot de données contrôlées nécessite tout d'abord une réduction du nombre de traces. Cette réduction se déroule en une ou deux étapes. Dans la première, pour toutes les structures contrôlant des opérations non assujetties à des contraintes, les traces induites par chacune de ces traces sont regroupées (*collapsing*) en une seule trace en mentionnant toutefois la condition d'exécution de chacune des opérations se trouvant dans ces traces. Si le nombre de traces obtenu permet un ordonnancement en un temps raisonnable, la deuxième étape peut être omise. Autrement, il s'agira d'insérer des points de coupure entre différentes structures de contrôle en visant un bon compromis entre le nombre de traces obtenu et le nombre de cycles d'exécution. La table 5.8 montre les résultats obtenus pour le cas défavorable (transformer toutes les traces en une seule) où un nombre très élevé de traces est transformé en une seule trace en un temps CPU égal à 239 s.

Table 5.8. REGROUPEMENT DE TRACES

Initial #DFGs	Obtained #DFGs	CPU Time (s)
$1.980704*10^{28}$	1	0
$8.98846*10^{307}$	1	0
2^{100000}	1	12
2^{500000}	1	239

La table 5.9 montre les résultats obtenus pour la deuxième étape de réduction de traces. Il est à noter que :

- 1 seul point de coupure a été inséré pour passer de $1.980704*10^{28}$ traces (ip#) à $2.81475*10^{14}$ traces (op#) quand le nombre de traces à ne pas dépasser est $9.90352*10^{27}$. Noter que le rapport op# / ip# est très faible
- Seulement 1008 points de coupure permettent de passer de $8.98846*10^{307}$ traces à 2526 traces (le rapport est très faible) quand le nombre de traces à ne pas dépasser est de 5000

Noter que les entrées :

- 1, 5, 9 et 13 dans la table 5.9 représentent le cas où le nombre requis de traces est supérieur ou égal au nombre initial (le nombre de points de coupure est donc égal à 0)
- 4, 8, 12 et 16 représentent le cas où le nombre requis de traces est inférieur au nombre minimal de traces (le nombre de points de coupure est alors maximal et est égal à 93, 1022, 99999 et 499999, respectivement). Dans de tels cas, les nombres obtenus sont égaux aux nombres minimaux tout en étant supérieurs aux nombres requis
- 3, 7, 11 et 15 représentent le cas où le nombre requis de traces est égal au nombre minimal de traces. Dans ce cas, le nombre de points de coupure est inférieur ou égal à N-1, N étant le nombre de structures de contrôle séquentielles. Noter que ce nombre peut être inférieur à N-1 car aucun point de coupure et 1 point de coupure entraînent le même nombre de traces pour chaque couple de structures de contrôle successives ($2+2=2^2$)
- 2, 6, 10 et 14 représentent le cas où le nombre requis de traces est supérieur au nombre minimal et inférieur au nombre initial de traces. Les résultats indiquent qu'un nombre minimal de points de coupure (1, 1008, 97956 et 497956) implique un rapport (nombre de traces obtenu / nombre initial de traces) très faible : $1.42*10^{-14}$, $2.81*10^{-305}$, 2^{-98977} et $2^{-499981}$, respectivement, tout en garantissant un nombre de traces inférieur ou égal au nombre désiré.

Table 5.9. INSERTION DE POINTS DE COUPURE

Entry #	# Control Structures	Initial #paths	Required # paths	Obtained # paths	#cuts	CPU Time (s)
1	94	$1.980704*10^{28}$	2^{94}	2^{94}	0	0
2	94	$1.980704*10^{28}$	$9.90352*10^{27}$	$2.81475*10^{14}$	1	0
3	94	$1.980704*10^{28}$	188	188	92	0
4	94	$1.980704*10^{28}$	187	188	93	0
5	1023	$8.98846*10^{307}$	$9.0*10^{307}$	$8.988466*10^{307}$	0	0
6	1023	$8.98846*10^{307}$	$5.0*10^{3}$	2526	1008	0
7	1023	$8.98846*10^{307}$	2046	2046	1020	0
8	1023	$8.98846*10^{307}$	2045	2046	1022	0
9	100000	2^{100000}	2^{100000}	2^{100000}	0	13
10	100000	2^{100000}	$1.0*10^{308}$	$8.988466*10^{307}$	97956	3809
11	100000	2^{100000}	$2.0*10^{5}$	$2.0*10^{5}$	99997	3845
12	100000	2^{100000}	$15.0*10^{4}$	$2.0*10^{5}$	99999	13
13	500000	2^{500000}	2^{500000}	2^{500000}	0	81
14	500000	2^{500000}	$1.69*10^{308}$	$8.988466*10^{307}$	497956	5256
15	500000	2^{500000}	$1.0*10^{6}$	$1.0*10^{6}$	499997	5867
16	500000	2^{500000}	$0.5*10^{6}$	$1.0*10^{6}$	499999	74

Nous avions vu que notre ordonnancement assujetti à la contrainte de la consommation de puissance était basé sur les graphes colorés. Bien que ce dernier problème soit NP-complet, la solution exacte est connue pour les benchmarks DIMACS mentionnés dans la table 5.10. Cette table reflète l'efficacité de notre technique dans la mesure où avec l'utilisation d'une seule classe, le temps CPU n'excède pas 4 s tout en générant la solution exacte pour les 14 premiers benchmarks. Pour le $15^{ème}$, $16^{ème}$ et $17^{ème}$ benchmark, la solution exacte est obtenue avec k=3 (k est le nombre de classes), k=7 et k=3, respectivement. Une solution proche de la solution exacte est obtenue pour le $18^{ème}$ benchmark (#couleurs=8, k=8) et le $19^{ème}$ (#couleurs=11, k=8). Il est aussi montré qu'un temps CPU de 30'58" nous a permis d'obtenir une solution proche (11) de la solution exacte (9) en utilisant 8 classes plutôt que celle obtenue (14) avec une seule classe. Enfin, cette table montre que le pourcentage d'erreur passe de 8.60 à seulement 1.92 lorsque plus d'une classe sont utilisées pour l'affectation des nœuds du graphe à travers les classes.

Table 5.10. COLORATION DE GRAPHES

DIMACS benchmark	**Exact solution**	**#colours (#classes =1)**	**% error**	**CPU Time (s)**	**(#colours, #classes)**	**% error**	**CPU Time (s)**
1. anna	11	11	0	0	(11 , 1)	0	0
2. david	11	11	0	0	(11 ,1)	0	0
3. myciel3	4	4	0	0	(4 , 1)	0	0
4. myciel4	5	5	0	0	(5 , 1)	0	0
5. myciel5	6	6	0	0	(6 , 1)	0	0
6. myciel6	7	7	0	0	(7 , 1)	0	0
7. huck	11	11	0	0	(11 , 1)	0	0
8. zeroin.i.3	30	30	0	0	(30 , 1)	0	0
9. miles750	31	31	0	0	(31 , 1)	0	0
10. miles500	20	20	0	0	(20 , 1)	0	0
11. miles1500	73	73	0	4	(73 , 1)	0	4
12. miles1000	42	42	0	1	(42 , 1)	0	1
13. games120	9	9	0	0	(9 , 1)	0	0
14. jean	10	10	0	0	(10 , 1)	0	0
15. myciel7	8	9	12.5	0	(8 , 3)	0	0
16. queen5_5	5	7	40	0	(5 , 7)	0	2
17. miles250	8	9	12.5	0	(8 , 3)	0	0
18. queen6_6	7	10	42.86	0	(8 , 8)	14.29	49
19. queen8_8	9	14	55.56	0	(11 ,8)	22.22	1858
%error average			**8.60**			**1.92**	

Les résultats obtenus dans le cadre de l'ordonnancement assujetti à la contrainte de consommation de puissance sont indiqués dans la table 5.11. Il est montré :

- qu'en général, l'ordonnancement basé sur la théorie des jeux [34] donne de meilleurs résultats que ceux obtenus par la programmation linéaire entière [33]
- nos résultats concernant la consommation de puissance sont plus proches de ceux de [34] qu'à ceux de [33] (noter, cependant, que pour l'instance *fir*, nous obtenons une valeur très différente de celles obtenues dans [33] et [34], et que le benchmark *fir* provient de [35])
- notre méthode est plus rapide que les 2 autres (la 1ère est basée sur la programmation linéaire entière connue pour être lente ([3], [26]) et la 2ème est basée sur la théorie des jeux avec une complexité algorithmique $O(N^S * N * S)$ pour un jeu impliquant N joueurs avec S stratégies pour chaque joueur
- notre méthode est plus flexible que les 2 autres du fait qu'elle produit différentes paires de vitesse et de consommation de puissance pour le même circuit, ce qui donne la possibilité au concepteur de choisir le meilleur compromis. Ce compromis est mieux illustré par les résultats indiqués dans la table 5.12

Table 5.11. ORDONNANCEMENT SOUS LA CONTRAINTE CONSOMMATION DE PUISSANCE

Bench. Circuit	[33]			[34]			Our Method With binding ; Without binding			
	Lat. (cycles)	Pwr (mW)	Run Time (s)	Lat. (cycles)	Pwr (mW)	Run Time (s)	Lat. (cycles)	MaximalPower (mW)	Average Power (mW)	Run Time (s)
Diff. eqn	4	22.9	35	5	16.0	37	4 ; 4	25.00 ; 25.00	20.00 ; 20.00	0 ; 20
Diff. eqn							5 ; 5	20.00 ; 25.00	16.00 ; 16.00	0 ; 20
Diff. eqn							7 ; 8	14.29 ; 14.01	11.43 ; 11.39	0 ; 20
FIR	10	61.7	221	10	50.4	165	9 ; 9	13.06 ; 13.06	10.44 ; 10.44	0 ; 91
FIR							10 ; 11	11.75 ; 11.58	9.40 ; 9.38	0 ; 91
FIR							13 ; 12	9.04 ; 9.26	7.23 ; 7.36	0 ; 91
IIR	8	12.8	100	9	11.2	96	8 ; 8	15.31 ; 15.31	12.25 ; 12.25	0 ; 74
IIR							9 ; 9	13.61 ; 13.61	10.89 ; 10.89	0 ; 74
IIR							11 ; 12	11.14 ; 11.06	8.91; 8.88	0 ; 74
Lattice	9	66.3	207	10	55.9	161	8 ; 9	80.63 ; 8.47	64.50 ; 64.47	0 ; 153
Lattice							10 ; 10	65.75 ; 65.75	51.60 ; 51.60	0 ; 153
Lattice							19 ; 18	33.95 ; 34.06	27.16 ; 27.41	0 ; 153
Ellip	17	204.0	259	19	191.3	204	16 ; 16	137.19 ; 137.19	109.75 ; 109.75	0 ; 188
Ellip							20 ; 21	119.20 ; 119.04	87.80 ; 87.74	0 ; 188
Ellip							24 ; 24	91.46 ; 91.46	73.17 ; 73.17	0 ; 188
WAVE	26	201.2	427	27	179.2	332	24 ; 25	304.38 ; 304.03	243.50 ; 243.46	194 ; 326
WAVE							25 ; 26	292.20 ; 191.85	233.76 ; 233.73	216 ; 326
WAVE							29 ; 28	251.90 ; 252.06	201.52 ; 201.57	265 ; 326
NC filter	27	324.3	501	27	286.7	377	25 ; 26	371.80 ; 371.65	297.44 ; 297.32	196 ; 364
NC filter							29 ; 31	320.52 ; 320.17	256.41 ; 256.13	211 ; 364
NC filter							33 ; 32	281.67 ; 282.09	225.33 ; 225.39	256 ; 364

Table 5.12. COMPROMIS VITESSE-CONSOMMATION DE PUISSANCE

Entry (#traces= 50)	# power constraints	# cycles	Average switching power (mW)	Maximal switching power (mW)	User-fixed switching power (mW)
1	0	28	11.870	20.000	15.0
2	15400	56	8.239	11.000	10.0
3	15400+7000=22400	68	5.402	8.500	7.0
4	22400+6800=29200	90	4.533	6.500	5.0
5	29200+94500=123700	634	0.4999	0.500	0.5

La première ligne de la table 5.12 indique des opérations ordonnancées en 28 cycles en absence de contrainte de consommation de puissance. Dans ce cas, la puissance dynamique moyenne (maximale) obtenue par SPOT ([10], [11], [12]) est 11.87 mW (20.0 mW) quand la valeur moyenne fixée par le concepteur de circuits est de 15 mW. Prenant ce premier résultat comme point de départ, des contraintes peuvent être ajoutées en vue d'obtenir le compromis vitesse - consommation de puissance désiré. Noter que ce nombre de contraintes peut être important (par exemple, non moins de 123700 contraintes sont nécessaires pour obtenir une consommation de 0.4999 mW pour un circuit exécutant ses opérations en 634 cycles contre une consommation de 11.87 mW et une exécution en 28 cycles). Ceci montre clairement que la génération des contraintes ne peut se faire de manière manuelle pour obtenir le compromis souhaité.

5.5. Synthèse de parties de contrôle:

Un type de synthèse de haut niveau de parties de contrôle consiste à affecter des codes adéquats (et non aléatoires) aux états de la machine à états finis implémentant une partie de contrôle donnée. Ceci, en vue d'anticiper sur l'obtention ultérieure (aux bas niveaux) d'une partie de contrôle offrant de bonnes caractéristiques en termes de surface, vitesse et consommation de puissance. L'outil SAFT que nous avons développé ([5]) vise de telles caractéristiques dans la mesure où la réduction simultanée des nombres de lignes et de colonnes entraîne une surface réduite, une bonne vitesse et une bonne consommation de puissance à cause d'interconnexions plus courtes, moins nombreuses (donc moins de paramètres parasites) et d'un nombre réduit de transistors. Cet outil a été comparé à d'autres méthodes en utilisant 43 MCNC FSMs Benchmarks, et les résultats obtenus sont ceux indiqués dans la table 5.13. Il est montré que :

- le temps CPU de notre méthode est de l'ordre de quelques secondes, à l'exception des benchmarks *s298* et *tbk* pour lesquels les temps CPU sont 14mn51s et 5h5mn25s, respectivement
- il n'existe aucun benchmark pour lequel SAFT s'est planté. Mieux, SAFT donne des longueurs de code intéressantes en un temps CPU n'excédant pas en général 13 s alors que ENC s'est planté sur les benchmarks *planet*, *styr* et *scf*. NOVA s'est planté sur le benchmark *tbk*, et DIET sur *planet*, *tbk* et *scf*. Noter que pour *planet*, notre longueur de code est inférieure à $\lceil \log_2 48 \rceil$ (48 est le nombre d'états pour le benchmark *planet*)
- à l'exception de *donfile* et *tbk* pour lesquels notre outil donne les plus mauvais résultats, notre longueur de code diffère de peu de celle donnée par le meilleur outil pour une instance donnée. Par contre, cette différence peut être très importante (supérieure ou égale à 150%) lorsque SAFT est le meilleur outil pour le benchmark considéré. Il y'a beaucoup de tels cas et nous citons seulement le benchmark *ex6* pour lequel notre longueur de code est 3 fois inférieure à celle produite par STOIC, le deuxième meilleur outil pour ce benchmark, et 5 fois plus petite que celles obtenues avec NOVA et KISS, les plus mauvais outils pour ce benchmark. Noter que les pourcentages utilisés dans cette table sont obtenus comme indiqués au bas de la table
- seul SAFT permet d'obtenir une longueur de code inférieure à $\lceil \log_2 n \rceil$, n étant le nombre d'états de la machine à états finis, tout en assurant à celle-ci un fonctionnement correct et déterministe. Ceci, grâce à l'attribution du même code à 2 ou plusieurs états (section 4.2.2). De telles longueurs de codes ont été possibles pour 15 benchmarks (ceux dont le caractère # apparaît à côté de ces longueurs)
- sur ces 43 benchmarks, SAFT a donné les meilleurs résultats pour 32 benchmarks, soit dans 74% des cas

Noter que dans la table 5.13, h1, h2 et h3 sont les heuristiques que nous avions développées pour résoudre SP1, SP2 et SP3, respectivement (section 4.2.2).

Table 5.13. CODIFICATION D'ETATS DE MACHINES À ETATS FINIS PAR DIFFERENTS OUTILS

FSM	Heur.	CPU T (s)	SAFT	STOIC	ENCORE	NOVA	ENC	KISS	DIET	MEILLEUR OUTIL
ex2	h3	2	**8**	5	6	6	?	6	6	STOIC 120% - 160%
ex3	h3	1	**5**	4	6	6	?	5	7	STOIC 125% - 175%
ex5	h3	0	**3 #**	4	5	6	?	5	5	**SAFT** 133% - 200%
ex6	h3	0	**1 #**	3	4	5	?	5	4	**SAFT** 300% - 500%
ex7	h3	1	**6**	4	6	?	?	?	6	STOIC 150% - 150%
keyb	h3	3	**2 #**	5	8	7	9	8	8	**SAFT** 250% - 450%
lion9	h3	1	**2 #**	4	4	?	?	4	4	**SAFT** 200% - 200%
shiftreg	h1	0	**3**	3	3	3	?	?	3	**SAFT**,STOIC,ENCORE,NOVA, DIET -% - -%
train11	h3	1	**5**	4	5	5	?	6	5	STOIC 125% - 150%
bbsse	h3	1	**4**	6	6	6	7	6	6	**SAFT** 150% - 175%
dk27	h3	0	**4**	4	3	3	?	?	?	ENCORE,NOVA 133% - 133%
ex1	h3	3	**4 #**	7	7	7	9	7	7	**SAFT** 175% - 225%
ex4	h1	0	**3 #**	6	4	?	?	?	4	**SAFT** 133% - 200%
opus	h1	0	**2 #**	7	4	?	?	?	?	**SAFT** 200% - 350%
pma	h1	1	**4 #**	8	?	?	?	?	?	**SAFT** 200% - 200%
s1	h1	2	**2 #**	6	5	5	7	5	5	**SAFT** 250% - 350%
sse	h3	1	**4**	6	6	?	?	?	6	**SAFT** 150% - 150%
tma	h1	1	**4 #**	10	?	?	?	?	?	**SAFT** 250% - 250%
bbara	h3	1	**7**	8	5	5	?	?	?	ENCORE,NOVA 140% - 160%
cse	h3	1	**4**	9	5	5	7	6	5	**SAFT** 125% - 225%
dk14	h3	1	**5**	6	4	5	?	4	4	ENCORE,KISS,DIET 125% - 150%
dk15	h3	0	**5**	4	4	4	?	4	4	STOIC,ENCORE,NOVA,KISS, DIET 125% - 125%
dk16	h3	2	**12**	6	8	10	12	10	8	STOIC 133% - 200%
dk17	h3	1	**4**	6	4	4	?	4	4	**SAFT**,ENCORE,NOVA,KISS,DIET 150% - 150%
dk512	h3	0	**4**	6	5	6	9	6	5	**SAFT** 125% - 225%
lion	h1	0	**1 #**	3	2	?	?	?	2	**SAFT** 200% - 300%
mc	h1	0	**1 #**	4	2	?	?	?	?	**SAFT** 200% - 400%
planet	h3	2	**5 #**	8	7	6	P	?	P	**SAFT** 120% - 160%
train4	h3	0	**2**	3	2	?	?	?	2	**SAFT** ,ENCORE,DIET 150% - 150%
tav	h1	1	**2**	?	2	?	?	?	2	**SAFT**,ENCORE,DIET -% - -%
s8	h3	0	**3**	?	3	?	?	?	3	**SAFT**,ENCORE,DIET -% - -%
bbtas	h2	0	**3**	?	3	?	?	3	?	**SAFT**,ENCORE,KISS -% - -%
beecount	h3	1	**3**	?	4	4	?	?	?	**SAFT** 133% - 133%

modulo12	h1	0	**4**	?	4	?	?	?	?	**SAFT**,ENCORE -% - -%
mark1	h1	0	**4**	?	4	?	?	5	4	**SAFT**,ENCORE,DIET 125% - 125%
kirkman	h1	13	**3 #**	?	6	?	11	?	6	**SAFT** 200% - 366.7%
s1a	h1	2	**2 #**	?	5	?	7	5	5	**SAFT** 250% - 350%
donfile	h3	3	**39**	?	6	15	12	12	7	ENCORE 116.7% - 650%
styr	h3	4	**6**	?	6	9	P	6	6	**SAFT**,ENCORE,KISS,DIET 150% - 150%
s298	h3	891	**15**	?	?	?	?	?	?	**SAFT** -% - -%
tbk	h3	18325	**96**	?	23	P	12	?	P	ENC 191.7% - 800%
scf	h1	8	**7**	?	8	?	P	8	P	**SAFT `** 114.3% - 114.3%
sand	h2	3	**5**	?	6	6	11	6	6	**SAFT** 120% - 220%

P : S'est planté
? : Résultat non disponible
 # : Longueur de code < $\lceil \log_2 n \rceil$ (n est le nombre des états des machines à états finis)
1er pourcentage = 2ème meilleure longueur de code / meilleure longueur de code
2ème pourcentage = plus mauvaise longueur de code / meilleure longueur de code

En utilisant ESPRESSO [36] un outil de synthèse logique sur les codes produits par SAFT, des valeurs intéressantes de surface ont été obtenues. Du fait que les valeurs de surface ont été estimées comme étant le produit des nombres de termes produits et de colonnes, et que les nombres de termes produits ne sont pas connus pour les outils STOIC[37] et ENC[38], nous avons comparé nos résultats uniquement à ceux publiés dans [39]. Ces derniers concernent les outils ENCORE[39], NOVA /ih, NOVA /ig et 1-hot encoding. Nous avions aussi comparé nos résultats à ceux produits par ESPRESSO exécutés sur des codes aléatoires de longueur $\lceil \log_2 n \rceil$, n étant le nombre d'états de la machine à états finis. Ces différents résultats sont indiqués dans la table 5.14 dans laquelle les colonnes correspondent au nom du benchmark, le nombre de termes-produits (nombre de cubes) et les surfaces obtenues par SAFT, ESPRESSO, 1-hot encoding, NOVA/ih, NOVA/ig et ENCORE. La dernière colonne mentionne le meilleur outil pour le benchmark considéré. De cette table, il ressort que :

- il y'a 11 benchmarks pour lesquels SAFT donne des valeurs de surface qui sont au moins 165% inférieures à celles obtenues avec le deuxième meilleur outil pour ces benchmarks (pour *s1a*, les deux pourcentages sont 1072% et 1351%)
- ESPRESSO donne de mauvais résultats lorsqu'il est exécuté avec des codes aléatoires
- De ces 43 benchmarks, SAFT donne les meilleures valeurs de surface pour 22 benchmarks (donc dans 51% de cas), suivi par ENCORE (23.2%), NOVA/ih (21%), NOVA/ig (13.8%), 1hot-encoding (6.9%), puis ESPRESSO (4.6%).

Table 5.14. SURFACES DE PARTIES DE CONTRÔLE

FSM	SAFT		ESPRESSO		1-hot encod.		NOVA/ih		NOVA/ig		ENCORE		BEST TOOL
	#cubes	area	#cubes	area	#cubes	area	#cubes	area	#cubes	area	#cubes	area	
ex2	**40**	**1200**	43	903	38	798	29	609	36	756	32	672	NOVA/ih 110% - 197%
ex3	**22**	**462**	26	468	21	378	18	324	18	324	17	306	ENCORE 106% - 153%
ex5	**20**	**300**	20	360	19	342	14	252	18	324	16	288	NOVA/ih 114% - 143%
ex6	**11**	**231**	31	837	23	621	25	675	25	675	25	675	**SAFT** 269% - 362%
ex7	**23**	**552**	20	360	20	360	17	306	17	306	18	324	NOVA/ih,NOVA/ig 106% - 180%
keyb	**19**	**418**	107	3317	77	2387	48	1488	55	1705	51	1581	**SAFT** 356% - 794%
lion9	**7**	**77**	15	255	10	170	8	136	9	153	8	136	**SAFT** 177% - 331%
shiftreg	**4**	**48**	4	48	9	108	4	60	8	120	6	72	**SAFT**,ESPRESSO 125% - 250%
train11	**13**	**260**	18	306	11	187	9	153	12	204	10	170	NOVA/ih 111% - 200%
bbsse	**17**	**561**	36	1188	30	990	30	990	30	990	30	990	**SAFT** 176% - 212%
dk27	**9**	**144**	12	156	10	130	9	117	8	104	8	104	NOVA/ig,ENCORE 113% - 150%
ex1	**43**	**2107**	66	3432	44	2288	48	2496	51	2652	45	2340	**SAFT** 109% - 163%
ex4	**11**	**330**	20	660	21	693	19	627	19	627	21	693	**SAFT** 190% - 210%
opus	**9**	**198**	20	560	19	532	16	448	16	448	18	504	**SAFT** 226% - 283%
pma	**53**	**1908**	64	2496	?		?		?		?		**SAFT** 131% - 131%
s1	**15**	**420**	88	3256	92	3404	80	2960	87	3219	86	3182	**SAFT** 705% - 810%
sse	**17**	**561**	35	1155	30	990	30	990	30	990	30	990	**SAFT** 176% - 206%
tma	**31**	**992**	42	1470	?		?		?		?		**SAFT** 148% - 148%
bbara	**34**	**1054**	31	682	34	748	25	550	25	550	25	550	NOVA/ih,g,ENCOR 124% - 192%
cse	**44**	**1452**	63	2079	57	1881	46	1518	45	1485	45	1485	**SAFT** 102% - 143%
dk14	**26**	**676**	39	897	25	500	29	580	27	540	27	540	1-HOT ENCODING 108% - 179%
dk15	**17**	**442**	26	442	17	289	19	323	18	306	18	306	1-HOT ENCODING 106% - 153%
dk16	**78**	**3354**	91	2002	55	1210	59	1298	72	1584	58	1276	1-HOT ENCODING 105% - 277%
dk17	**20**	**380**	30	570	20	320	19	304	19	304	17	272	ENCORE 112% - 210%
dk512	**22**	**374**	26	442	21	357	18	306	19	323	19	323	NOVA/ih 106% - 144%
lion	**5**	**40**	7	77	8	88	6	66	6	66	7	77	**SAFT** 165% - 220%
mc	**5**	**70**	10	170	10	170	9	153	9	153	8	136	**SAFT** 194% - 243%
planet	**102**	**4896**	117	5967	92	4692	91	4641	89	4539	90	4590	NOVA/ig 101% - 131%
train4	**7**	**77**	8	88	7	77	6	66	6	66	6	66	NOVA/ih,g,ENCOR 117% - 133%
tav	**10**	**180**	11	198	12	216	11	198	11	198	11	198	**SAFT** 110% - 120%
s8	**16**	**288**	19	342	14	252	10	180	10	180	9	162	ENCORE 111% - 211%
bbtas	**13**	**195**	14	210	16	240	9	135	12	180	10	150	NOVA/ih 111% - 178%
beecount	**10**	**190**	18	342	12	228	13	247	12	228	10	190	**SAFT**,ENCORE 120% - 180%

modulo 12	**15**	**225**	15	225	24	360	12	180	12	180	14	210	NOVA/ih,NOVA/ig 117% - 200%
mark1	**29**	**1102**	27	1026	19	722	21	798	19	722	18	684	ENCORE 106% - 161%
kirkman	**57**	**2223**	186	7812	61	2562	79	3318	77	3234	61	2562	**SAFT** 115% - 351%
s1a	**9**	**252**	76	2812	92	3404	76	2812	80	2960	73	2701	**SAFT** 1072% - 1351%
donfile	**24**	**2928**	41	820	24	480	35	700	48	960	18	360	ENCORE 133% - 813%
styr	**79**	**3634**	142	6106	111	4773	94	4042	103	4429	93	3999	**SAFT** 110% - 168%
s298	**750**	**42750**	922	33192	?		?		?		?		ESPRESSO 129% - 129%
tbk	**180**	**54540**	175	14000	173	13840	154	4620	176	5280	128	3480	ENCORE 133% - 1567%
scf	**139**	**18209**	145	18995	151	19781	148	19388	146	19126	140	18340	**SAFT** 101% - 109%
sand	**69**	**3174**	101	4646	114	5244	101	4646	102	4692	100	4600	**SAFT** 145% - 148%

? : Résultat non disponible

1er pourcentage = 2ème meilleure surface / meilleure surface

2ème pourcentage = plus mauvaise surface / meilleure surface

5.6. Interface de communications entre une partie opérative et sa partie de contrôle :

Cette interface a été implémentée par l'insertion de transistors de passage de type CMOS qui jouent le rôle d'interrupteurs. En effet, selon le comportement de la partie opérative, une partie de contrôle en est déduite pour régir les différents transferts de données. Cette étape de synthèse se compose des étapes suivantes:

- détermination de l'automate décrivant le comportement de la partie opérative à partir du fichier d'allocation de ressources physiques, issu lui-même du fichier d'ordonnancement des opérations de cette partie opérative
- détermination de la table de commande (constituée de signaux de contrôle, d'états d'entrée et de sortie de la machine à états finis et des signaux de commande) à partir de l'automate
- numérotation des drains et des sources des différents transistors. Cette numérotation servira pour savoir quel transistor est relié à quel registre, unité fonctionnelle et interconnexion
- numérotation des grilles des transistors pour savoir quelle grille est reliée à quel signal de commande (provenant de la partie de contrôle). Dans le cas où une entité (registre, unité fonctionnelle, interconnexion) est sollicitée deux ou plusieurs fois (pour lire ou ranger une donnée dans un même registre, effectuer une opération avec une même unité fonctionnelle, transférer des données sur une même interconnexion), le numéro de la grille du transistor NMOS est celui de la sortie d'une porte OU ayant pour entrées les signaux de commande activant cette entité à différents temps d'exécution de la partie opérative. Le signal alimentant la grille du transistor PMOS étant complémentaire à celui relié à la grille du transistor NMOS du même transistor de passage, le numéro de la grille du transistor PMOS est alors celui de la sortie de l'inverseur ayant pour entrée la sortie de la porte OU connectée au transistor NOMS du même transistor de passage

Ces tâches sont toutes polynomiales et leur implémentation a conduit aux résultats de la table 5.15 obtenus en des temps CPU très faibles.

Table 5.15. CONCEPTION DE L'INTERFACE ENTRE UNE PARTIE OPERATIVE ET SA PARTIE DE CONTROLE

Nombre de registres	Nombre d'unités fonctionnelles	Nombre de bits	Nombre de transistors	Temps CPU (s)
3	3	32	1536	0
6	5	32	2176	0
6	6	32	2816	0
15	6	32	2880	0
16	8	64	8960	0
120	50	32	762000	155

5.7. Consommation de la puissance:

La première version de notre outil SPOT ([10], [11]) consistait à estimer la consommation *maximale* de la puissance *dynamique* à différents niveaux d'abstraction, tout en tenant compte de différents styles de conception et en optant pour le modèle de délais général (les délais des portes du circuit peuvent être différents). Les résultats obtenus sur des benchmarks ISCAAS sont ceux de la Table 5.16. Notre heuristique (basée sur un algorithme génétique) a été comparée à la méthode exacte dans la mesure du possible (la complexité du problème n'étant pas polynomiale – elle est égale à $O(2^{2*N})$, N étant le nombre d'entrées primaires du circuit-, la solution exacte ne peut donner le résultat en un temps CPU raisonnable lorsque la valeur de N est importante). Il est à remarquer que la puissance dynamique dissipée pour le benchmark *too large* est moins importante que celle dissipée pour *c1355* à cause du nombre de niveaux dans ce circuit (28) plus important que celui de *too large* (4 seulement) et du nombre de portes (634 contre 86 seulement). Nous avions aussi testé notre outil pour deux parties opératives. La première est constituée des circuits *c17* et *decod* exécutant en parallèle et dont les résultats alimentent *majority*. Dans la seconde, les circuits *b1*, *cm82a* et *cm138a* s'exécutent séquentiellement. L'énergie dissipée par cycle est plus importante pour la première opérative (c'est la somme des énergies dissipées par *c17* et *decod*) que pour la seconde. Les résultats montrent aussi que l'erreur relative maximale est de 14%. Cette erreur a été obtenue pour un certain nombre de générations de la population de notre algorithme génétique exécuté sur une station Sun dont la fréquence du processeur est de 50 MHz. Enfin, notons qu'en plus de déterminer la puissance dynamique maximale, SPOT génère aussi les paires de vecteurs ayant induit cette consommation, et ce, pour des simulations plus fines. Ainsi, au lieu de procéder à 2^{2*N} simulations électriques (ce qui est fastidieux et prendrait des mois .... voire des années), le concepteur de circuits se limiterait à quelques simulations fines pour mieux cerner la valeur de la puissance dissipée.

Table 5.16. ESTIMATION DE LA CONSOMMATION MAXIMALE DE LA PUISSANCE DYNAMIQUE DES CIRCUITS CMOS (fréquence du processeur=50 MHz)

Circuit	#prim. inputs	#gates	#stage levels	Exact Method(mW)	CPU time	Heur. Method(mW)	CPU time	% error
c17	5	7	3	0.5625	3″	0.5625	1″	0%
too large	38	86	4	-	-	1.0000	2h31'20″	-
c1355	43	634	28	-	-	78.2500	3h36'17″	-
decod	5	36	4	0.7500	45″	0.7500	5″	0%
count	35	94	34	-	-	7.0000	3h53'07″	-
cordic	23	204	26	-	-	19.8750	2h46'32″	-
alu4	14	224	24	-	-	12.8750	4h24'07″	-
comp	32	110	12	-	-	1.7500	2h52'51″	-
apex6	135	476	16	-	-	17.3750	1h44'47″	-
parity	16	30	8	-	-	0.5000	5'17″	-
majority	5	4	4	0.2500	3″	0.2500	1″	0%
b1	3	12	4	0.7500	0″	0.7500	0″	0%
cm82a	5	12	4	0.8750	6″	0.7500	1″	14%
cm151a	12	18	10	-	-	2.3750	1'45″	-
cm152a	11	2	2	-	-	0.1250	28″	-
cm138a	6	18	4	0.5000	45″	0.5000	5″	0%
9syml	9	88	12	6.6250	6h16'32″	5.6975	4'45″	14%
x2	10	24	4	-	-	2.0000	1'31″	-
cm42a	4	26	6	1.2500	4″	1.1000	1″	12%
z4ml	7	16	4	1.0000	5'54″	1.0000	22″	0%
data path1	15	47	8	-	-	1.3125	3'36″	-
data path2	14	42	12	-	-	0.7500	2'54″	-

SPOT a été par la suite amélioré pour déterminer aussi la consommation dynamique moyenne ([12]), puis généralisé pour tenir compte de la consommation due aux fuites de courants, consommation non négligeable avec les nouvelles technologies où les tensions de seuil des transistors sont très faibles. De plus, afin d'aboutir à un bon compromis entre la puissance totale dissipée et la vitesse du circuit, différentes tensions peuvent alimenter les portes d'un même circuit, comme différentes tensions de seuil de transistors peuvent être utilisées pour les transistors d'un même circuit. Rappelons que la puissance (vitesse) est d'autant importante (intéressante) que la tension d'alimentation est grande, et que la puissance due aux fuites de courant (temps de commutation du transistor) est d'autant importante (court) que la tension de seuil du transistor est faible. Ces deux paramètres *vitesse* et *consommation de puissance* ne pouvant être optimisés à la fois (ils sont antagonistes), il s'agit alors de déterminer un bon compromis à ce problème d'optimisation combinatoire. Ce travail ayant été fait (avec toutefois une généralisation en cours) en dehors du cadre que dans celui assigné dans cette thèse, nous soulignons seulement que SPOT exploite les résultats obtenus par l'outil d'aide à la conception de circuits digitaux à faible consommation de puissance pour estimer la consommation totale du circuit en tenant compte du fait que différentes tensions d'alimentation (de seuil) sont affectées aux différentes portes (transistors) d'un même circuit. Nous citons ci-après les principaux fichiers utilisés et générés par la dernière version de SPOT :

- le fichier TENSIONS (supposé obtenu par l'outil d'aide à la conception de circuits digitaux à faible consommation de puissance) indiquant les valeurs de la tension d'alimentation, celle de la tension de seuil du transistor N, et celle du transistor P, et ce, pour chaque porte du circuit - Figure 5.1 -
- le fichier décrivant la fréquence de fonctionnement d'un circuit, son style de conception, son comportement, la capacité de charge et le délai de chaque porte – Figure 5.2 -
- le fichier des résultats indiquant les différentes paires de vecteurs ayant maximisé la dissipation de la puissance, les valeurs maximale et moyenne de la dissipation de puissance, la valeur (déterminée à partir des valeurs maximale et moyenne) susceptible d'être proche de la solution exacte (rappelons que le problème considéré n'est pas polynomial), la consommation de la puissance due aux fuites de courants dans les transistors et la consommation de la puissance totale – Figure 5.3 -

```
3.3 0.1 -0.1
3.3 0.3 -0.3
3.3 0.1 -0.1
1.1 0.3 -0.3
3.3 0.1 -0.1
3.3 0.3 -0.3
1.1 0.1 -0.1
3.3 0.3 -0.3
3.3 0.1 -0.1
3.3 0.3 -0.3
3.3 0.1 -0.1
3.3 0.3 -0.3
1.1 0.1 -0.1
3.3 0.3 -0.3
3.3 0.1 -0.1
1.1 0.3 -0.3
3.3 0.1 -0.1
3.3 0.3 -0.3
3.3 0.1 -0.1
1.1 0.3 -0.3
3.3 0.1 -0.1
3.3 0.3 -0.3
3.3 0.1 -0.1
3.3 0.3 -0.3
3.3 0.3 -0.3
```

Fig.5.1. Fichier TENSIONS

```
32.
static1
3G 6G % BBG % (not(3G) or not (6G)   )  % 100. % 2. %
BG 3G % BAG % ( not (BG)  or not (3G)   )  % 80. % 1. %
BBG 7G % B9G % ( not (BBG)  or not (7G)   )  % 100. % 2. %
2G BBG % B6G % ( not (2G)  or not (BBG)   )  % 100. % 1. %
B6G B9G % 23G % ( not (B6G)  or not (B9G)   ) % 100. % 1.5 %
2G BBG % B7G % ( not (2G)  or not (BBG)   )  % 100. % 2.5 %
B6G B7G % 25G % ( not (B6G)  or not (B7G)   )  % 40. % 2.5 %
25G BBG % B4G % ( not (25G)  or not (BBG)   )  % 100. % 2. %
B6G B4G % 21G % ( not (B6G)  or not (B4G)   )  % 100. % 2. %
39G 6G % B9BG % ( not (39G)  or not (6G)   )  % 40. % 2. %
B9G 3G % B9AG % ( not (B9G)  or not (3G)   )  % 50. % 2. %
B9BG 7G % B99G % ( not (B9BG)  or not (7G)   )  % 100. % 1. %
28G BBG % B86G % ( not (28G)  or not (BBG)   )  % 100. % 1. %
B66G B9AG % 263G % ( not (B66G)  or not (B9AG)   )  % 100. % 2. %
28G B9BG % B77G % ( not (28G)  or not (B9BG)   )  % 100. % 2. %
B6G B77G % 259G % ( not (B6G)  or not (B77G)   )  % 100. % 2. %
259G BBG % B44G % ( not (259G)  or not (BBG)   )  % 100. % 2. %
B6G B4G % 271G % ( not (B6G)  or not (B4G)   )  % 100. % 2. %
B61G B4G % 272G % ( not (B61G)  or not (B4G)   )  % 100. % 2. %
38G 259G % B17G % ( not (38G)  or not ( 259G )   )  % 100. % 2. %
B6G B17G % 229G % ( not ( B6G )  or not (B17G)   )  % 100. % 2. %
229G BBG % B48G % ( not (229G)  or not (BBG)   )  % 100. % 2. %
B6G B48G % 279G % ( not (B6G)  or not (B48G)   )  % 100. % 2. %
B48G B4G % 222G % ( not (B48G)  or not (B4G)   )  % 100. % 2. %
B44G B6G % 22G % ( not (B44G)  or not (B6G)   )  % 100. % 2. %
```

Fig.5.2. Fichier COMPORTEMENT DU CIRCUIT

```
Vector pairs maximizing the switching dissipation of the circuit:

B61G# 1
B66G# 1
38G# 1
2G# 1
7G# 1
28G# 1
BG# 1
3G# 1
39G# 1
6G# 1
B61G 0
B66G 0
38G 0
2G 0
7G 0
28G 0
BG 0
3G 0
39G 0
6G 0

THE MAXIMAL SWITCHING POWER IS 7.492320e-04 Watts

THE AVERAGE SWITCHING POWER IS 2.063370e-04 Watts

THE EXPECTED ACTUAL SWITCHING POWER IS 5.114403e-04 Watts

P_leakage= 6.208938e-05 Watts

THE TOTAL POWER DISSIPATION IS:

P_tot (P_expected + P_leak) = 5.735297e-04 Watts
```

Fig.5.3. Fichier RESULTATS (consommation due à l'activité dynamique et aux fuites du courant)

CONCLUSION GENERALE

Les systèmes actuels sont devenus très complexes de par leur taille (aujourd'hui, un processeur est juste un composant dans de tels systèmes) et de par leur hétérogénéité (circuits mixtes analogiques et numériques présents sur la même puce). Ceci a engendré l'apparition de nouveaux problèmes, ce qui a nécessité la mise à jour d'anciens outils de CAO, voire carrément leur remise en cause et le développement de nouveaux. Pour des fins de compromis et de rapidité de conception, certains systèmes peuvent être conçus de manière conjointe Hardware et Software. Une méthodologie de conception de tels systèmes a été présentée à l'introduction du deuxième chapitre. Notons toutefois que la décomposition Hardware – Software reste de nos jours manuelle pour ce qui est des outils industriels du fait de la diversité des applications, rendant toute décomposition automatique infructueuse. Ceci étant, des travaux de recherche sont toujours menés dans ce cadre. Une erreur étant d'autant plus coûteuse qu'elle est détectée à un niveau d'abstraction bas, une méthodologie rigoureuse de conception doit être adoptée afin de détecter les problèmes dès les premiers niveaux d'abstraction.

Une telle méthodologie pour la conception de la partie Hardware a été présentée dès le premier chapitre de cette thèse, et sur laquelle s'est articulé notre travail centré sur la synthèse Hardware de systèmes VLSI monopuce à de hauts niveaux d'abstraction. Nous avions particulièrement mis l'accent sur le compromis vitesse – consommation de puissance afin de réduire la consommation de la puissance de millions de transistors présents sur la même puce sans pour autant dégrader les performances du système, ces deux paramètres étant antagonistes. Notons aussi que cette réduction de consommation de puissance est le point focal dans toute conception de système monopuce à cause du grand nombre d'éléments présents sur la même puce (une grande consommation de puissance entraînerait une augmentation de température, qui, à une certaine valeur affecterait la fiabilité du système) et à cause des nombreux systèmes embarqués et portables nécessitant tous une consommation modérée d'énergie puisque les batteries NiMh (les plus sophistiquées de nos jours) n'assurent pas une grande autonomie. Nous avions alors abordé ce compromis en décomposant le système en sous-systèmes VLSI et en affectant les tâches aux différents sous-systèmes. Une synthèse de ces tâches s'en est alors suivie pour que le système puisse les exécuter dans les délais (cas des systèmes fonctionnant en temps réel) tout en consommant le minimum d'énergie. Puis, un protocole de communications au niveau d'abstraction *système* a été présenté. Enfin, une méthode d'analyse a été développée en vue de déterminer les caractéristiques d'une configuration du système, ce qui permettrait au concepteur soit de la retenir et passer à un niveau de conception plus bas, soit de refaire la synthèse du système en ajoutant et/ou en supprimant des contraintes.

Chacun des sous-systèmes obtenus par notre synthèse fait ensuite lui-même l'objet d'une synthèse ayant toujours pour but de satisfaire les différentes contraintes portant sur la surface, la vitesse et la consommation de puissance. Un sous-système étant constitué d'une partie opérative et d'une partie de contrôle associée, notre synthèse a porté sur chacune de ces deux parties, suivie par une technique de génération d'interface (jouant le rôle de protocole de communications) entre ces deux parties. Les outils d'analyse que nous avions développé permettent alors de retenir

la configuration obtenue du sous-système, ou revoir sa conception en ajoutant et/ou supprimant des contraintes.

Notons que la plupart des problèmes engendrés par cette problématique ne sont pas polynomiaux : Ils sont $\sum_k^P$, et les plus simples d'entre-eux sont $\sum_1^P$ (=NP) complets.

Enfin, quoique le développement d'un cycle complet soit l'affaire d'une compagnie (centaines de personnes), nous comptons développer dans le futur d'autres outils de CAO, notamment en vue d'une conception conjointe Hardware – Software de systèmes, d'autant plus que nos outils de synthèse et d'analyse tiennent compte déjà de la présence simultanée de processeurs et d'ASICs sur notre architecture cible. L'intégration ultérieure de nos outils avec ceux de la conception des circuits analogiques aboutirait à un flot de conception plus complet.

Annexe

```
Algorithm1: // Determination of the supply voltage of each operation of each task, the time and the energy
            //consumed by the mandatory part of each task when some trace is being run
for each trace t
do{Total_Energy= ε^sel_O_T      Total_Time= δ^sel_O_T ;
      for each task i included in trace t
      do {O_i=0; Assign V_min to each operation of task i;
          E=Energy(i); // E is the energy consumed by task i
          if(E  <= E_max – Total__Energy)
          then {D=Throughput(i); // D is the throughput of i
                if(D > deadline_of_task i – Total_Time)
                then { N_k=Set_voltages_to_Vmax();
                       if(N_k == 0)
                       then {print "Cannot process task i
                                  due to the time  constraint";
                             exit();
                            }
                       endif
                     }
                endif
               }
          else { print "Cannot process task i due to the
                    energy  constraint";
                 exit();
               }
          endif
          Total_Energy += E;
          Total_Time += D;
          Energy[i]=E;
          Throughput[i] = D;
         }
       end
     }
end
```

```
Algorithm2: // Determination of the supply voltage of each operation of each task, the time and the energy
            //consumed by both the mandatory and the optional part of each task when some trace is being run
for each trace t
do { for each task i included in trace t
     do {Oi=Oi,max; // We first try to process all the operations of the optional part
 Label_Min: for(j=Mi to Mi+Oi-1)
              do V[i][j]=Vmin; // We attempt to perform the optional part with a low energy
                               // Notice that for the mandatory part, the supply voltages have been
                               // assigned during the 1rst step/
                               // Mi and Oi are the number of execution cycles of the mandatory
                               // and the optional parts, respectively
            end
            O_f=0; // O_f is the number of operations that are being assigned Vmax
            O_g=Oi,max;  // O_g is the number of operations
                         // that are being assigned Vmin
            Label_E:
            E=Energy(i); // E is the energy consumed by task i
            if(E <= Emax – Total__Energy)
            then {   D=Throughput(i); // D is the throughput of i
                     if(D <= deadline_of_task i – Total_Time)
                     then { if(Oi = Oi,max)
                              then continue; // The optional part of task i can be entirely processed
                              endif
                            // Oi ≠ Oi,max
                            // Oi will be gradually increased to the greatest value which satisfies the
                            // time and the energy constraints
                             while(E <= Emax-Total_Energy &&
                                    D <= deadline_of_task I – Total_Time && Oi < Oi,max)
                            do {P_Oi=Oi;
                                Oi+=⌈((Oi,max-Oi)/2.)⌉;
                                if(Oi == P_Oi)
                                then continue;
                               endif
                               E=Energy(i);
                               D=Throughput(i);
                               }
                           end
                           if(Oi == Oi,max && constraints met)
                           then {E=Energy(i);
                                 D=Throughput(i);
                                 continue;
                                }
                           endif
                           // Oi value has violated the time or the energy constraint: the best solution is
                           // in the range [P_Oi, Oi[
                           B_Oi=Oi;
                           Oi=P_Oi;
                           Oi=⌈((P_Oi+B_Oi)/2.)⌉;
                           E=Energy(i);
                           D=Throughput(i);
                           while(E <= Emax-Total_Energy && D <= deadline_of_task i – Total_Time)
                           do {P_Oi=Oi;
                              Oi=⌈((P_Oi+B_Oi)/2.)⌉;
```

```
                    if(Oi == P_Oi)
                    then continue;
                    endif
                    E=Energy(i);
                    D=Throughput(i);
                   }
                end
               while(E > Emax-Total_Energy || D > deadline_of_task_i – Total_Time)
               do {B_Oi=Oi;
                    Oi=⌈((P_Oi+B_Oi)/2.)⌉;
                    if(Oi == B_Oi)
                    then {Oi=P_Oi;
                           E=Energy(i);
                          D=Throughput(i);
                          continue;
                         }
                   endif
                    E=Energy(i);
                   D=Throughput(i);
                  }
               end
               continue;
               }
        else {if(O_f == Oi)
               then {Oi=⌊(Oi/2.)⌋;
                    if(Oi = 0)
                    then {E=Energy(i);
                          D=Throughput(i);
                          break; // No Optional operation can be performed for task i
                         }
                    endif
                   O_f=0; O_g=Oi; goto Label_Min;
                  }
               else { N_k=Set_voltages_to_Vmax();
                      if(!N_k)
                      then if(!Oi)
                           then {E=Energy(i);
                                 D=Delai(i);
                                 break;    // unable to process the optional part of task i
                                }
                           else {Oi=⌈Oi/2⌉;
                                 O_f=0;  O_g=Oi;
                                 goto label_Min;
                                }
                      endif
                    endif
                   O_f += N_k;
                   O_g -= N_k;
                   goto Label_E;
                  }
             endif
           }
       endif
     }
else { if(O_g == Oi)
       then {Oi=⌊(Oi/2.)⌋;
```

```
                    if(Oi=0)
                     then {E=Energy(i);
                           D=Throughput(i);
                           break;  // No Optional operation can  be performed for task i
                          }
                     endif
                    O_g=Oi;  O_f=0;  goto Label_E
                   }
               else {N_k=Set_voltages_to_Vmin();
                    if(!N_k)
                    then if(!Oi)
                         then {E=Energy(i);
                               D=Throughput(i);
                               break;
                               }
                          else {  Oi=⌈Oi/2;⌉
                                  O_g=Oi;   O_f=0;
                                  goto Label_Min;
                                 }
                         endif
                     endif
                    O_g += N_k;
                    O_f -= N_k;
                    goto Label_E;
                    }
                 endif
             }
         endif
         Total_Time+=D;
         Energy[i]+=E;
         Total_Energy+=E;
         Throughput[i]=D;
         Vf[t,i]=mean of voltages assigned to the operations of task i when trace t  is involved;
        }
       end
     }
end


Procedure Set_voltages_to_Vmax() // procedure attempting to fulfill the time constraint
{
 j=(mandatory)?0:Mi;
 // mandatory=1 (0) means that the mandatory (optional) part of task i is being considered
for(N_k=0; E < Emax-Total_Energy && j < Oi+Mi && D > deadline_of_task i –  Total_Time; j++)
do    if(V[num_task][j] == Vmin)
      then {V[num_task][j]=Vmax;
            P_E=E;
            P_D=D;
            E=Energy(num_task);
            D=Throughput(num_task);
            N_k++;
           }
    endif
end
if(N_k)
then if(E > Emax-Total_Energy)
```

```
      then {V[num_task][j-1]=Vmin;
             E=P_E;
            D=P_D;
            N_k--;
           }
    endif
endif
return (N_k);
}


Procedure Set_voltages_to_Vmin() // Procedure attempting to fulfil the energy constraint
{
for(j=Mi, N_k=0; D < deadline_of_task i-  Total_Time && j <= Oi+Mi && E > Emax-Total_Energy;
                                                                                    j++)
do    if(V[num_task][j] == Vmax)
       then {V[num_task][j]=Vmin;
             P_D=D;
             P_E=E;
             D=Throughput(num_task);
             E=Energy(num_task);
             N_k++;
            }
      endif
end
if(N_k)
then if(D > deadline_of_task_i –  Tolal_Time)
       then {V[num_task][j-1]=Vmax;
              D=P_D;
              E=P_E;
              N_k--;
             }
      endif
endif
return (N_k);
}
```

```
Procedure Détecte_trace()   // The values given below are related to the V/O assignment of some application
V1 = 1.13 V;      // V1 is determined by Equation 22
Sort the M1 values in the decreased order then store them in M_TRACES[] array
L_B=0;                              // Lower Bound value
U_B=Nb_traces-1;                // Upper Bound value
j= ⌈(L_B+U_B)/2 ⌉;
found=0;
while (! found)
do {if M1 = M_TRACES[j]
     then {found=1;
            while(j ≥ 0 && M_TRACES[j] = M1)
            do j--;
           end
            j++;   // Position of the 1rst task whose Mi value equals M1
           while (j < Nb_traces  && M_TRACES[j++] = M1)
           do  if E1=9.73 && D1=4.04
               then {   O1=α1 ;     // α1 is determined  as shown in Fig.3.15, Page 49
                        V2=1.13V; O2=20;
                        V3=2.12 V; O3=25;
                        V4=1.14V; O4=97;
                        V5=1.22V; O5=24;
                         …
                     }
               else    if  E1=8.49 && D1=5.06
                    then {O1=α1;   //α1 is determined as shown in Fig.3.15, Page 49
                             V2=1.14 V; O2=17
                             V3=1.13V; O3=30;
                             V4=1.13V; O4=201;
                             V5=1.12 V; O5=80;
                               …
                           }
                    else ……
                ……
           end
          }
    else { if M1 < M_TRACES[j]
           then U_B=j;
           else L_B=j;
           end if
            j= ⌈(L_B+U_B)/2 ⌉;    }
    end if
  }
end
```

Bibliographie

[1] Weste & Eshraghian "Principles of CMOS VLSI Design: A systems perspective" Addison Wesley, 1993

[2] M.R. Garey, D.S. Johnson "Computers and Intractability: a guide to the theory of NP-completeness" Freeman, San Fransisco

[3] D. Gajski, N. Dutt, A. Wu, S. Lin "High level synthesis" Kluwer Academic Publishers, 1997

[4] L. Gauthier, A. A. Djerraya "Conception des logiciels embarqués pour les systèmes monopuces" Lavoisier, 2003

[5] A. Mahdoum "SAFT : An Efficient States Assignment for Finite States Machine Tool" INFORMATION Journal, Vol.3, N°3, July 2000, Nippon Press

[6] A. Mahdoum "Démonstrations sur l'outil SAFT" Forum de la Conférence DATE, 4-8 March 2002, Palais des Congrès, Paris, France (résumé de l'outil publié dans DESIGNER'S FORUM PROCEEDINGS, DATE'02, 4-8 March 2002, Paris, France)

[7] H. Djamah, H. Hadj Moussa, A. Mahdoum "CMORGEN: A CMOS Regular Structure Generator" Synthèse (Revue de l'Université Badji Mokhtar, Annaba), Vol.6, N°6, 1997

[8] A. Mahdoum, M. Bougherara, L. Beha, N. Badache, H. Bessalah "SCHEDULE++: A User-Constrained Scheduling Tool for Controlled Data Flow Graphs" IEEE/SCS'04, 18-21 Mars 2004, Monastir, Tunisie

[9] A. Mahdoum, N. Badache, H. Bessalah "A Low-Power Scheduling Tool for SOC Designs" IEEE/IWSSIP'05, 22-24 September 2005, Chalkida, Greece

[10] A. Mahdoum, A. Boutammine, D. Touahri, N. Toubaline "FREEZER1 : Un Outil d'Aide à la Conception de Circuits Digitaux à Faible Consommation de Puissance" IEEE/FTFC'05, 18-20 Mai 2005, Paris, France

[11] A. Mahdoum "SPOT : Un Outil à base d'un Algorithme Génétique pour Estimer la Consommation Maximale de la Puissance Dynamique des Circuits CMOS" CSCA'99, Nov.99, Hôtel Sheraton, Alger

[12] A. Mahdoum "SPOT : An Estimation of the Switching Power Dissipation for CMOS Circuits and Data Paths Tool" SASIMI'97, 1-2 Dec. 97, Osaka, Japan

[13] A. Mahdoum "Démonstrations sur l'outil SPOT" Forum de la Conférence DATE, 4-8 March 2002, Palais des Congrès, Paris, France (résumé de l'outil publié dans DESIGNER'S FORUM PROCEEDINGS, DATE'02, 4-8 March 2002, Paris, France)

[14] http://www.systemc.org

[15] Y. Fei, N. K. Jha "Functional Partitioning for low Power Distributed systems of systems-on a-chip" Proc. 15th Intl. Conf. on VLSI Design, 2002

[16] M. Takahashi, N. Ishiura, A. Yamada "Thread Composition Method for Hardware Compiler Bach Maximizing Resource Sharing among Processes" IEICE Trans. on Fundamentals, Vol. E83-A, N°12, Dec. 2000

[17] http://www.claymath.org

[18] H. Belkouche, F. Louiz, A. Mahdoum "FEWERCOLORS : A New Technique for Solving the Graph Coloring Problem" DCIS'2000, Montpellier, France

[19] A. Mahdoum, F. Louiz, H. Belkouche "FEWERCOLORS : A New Technique for Solving the Graph Coloring Problem" 7th IEEE/ICECS, Beirut, Lebanon

[20] A. Mahdoum, F. Louiz, H. Belkouche "Démonstrations sur l'outil FEWERCOLORS" Forum de la Conférence DATE, 4-8 March 2002, Palais des Congrès, Paris, France (résumé de l'outil publié dans DESIGNER'S FORUM PROCEEDINGS, DATE'02, 4-8 March 2002, Paris, France)

[21] C.A.R. Hoare "Processus séquentiels communicants" Manuels Informatiques, MASSON

[22] http://accent.compilertools.net/accent-0.7.tar.gz

[23] N. Graham "Learning C++" Mc GRAW HILL, INC

[24] A. B. Fontaine, P. Hammes "UNIX System V, système et environnement" MASSON

[25] A. Bellaouar and M. I. Elmasry "Low-Power Digital VLSI Design" Norwell, Kluwer Academic Publishers, 1995

[26] C. Mead & L. Conway "Introduction aux systèmes VLSI" Inter Editions

[27] M. Sakarovitch "Optimisation combinatoire, méthodes mathématiques et algorithmiques : programmation discrète" HERMANN, 1984

[28] A. Mahdoum, N. Badache, H. Bessalah "An Efficient Assignment of Voltages and Optional Cycles for Maximizing Rewards in Real-Time Systems with Energy Constraints" Journal Of Low Power Electronics (JOLPE), Vol.2, N°2, August 2006, American Scientific Publishers

[29] R. A. Bergamashi, S. Raje, I. Nair and L. Trevillyan, "Control-flow versus data-flow-based scheduling combining both approaches in an adaptive scheduling system" IEEE Transactions on VLSI , Vol. 5, N° 1, pp. 82-100, 1997

[30] A. Mechti & D. Melzi "Elaboration d'une commande de transfert de données dans un circuit" Mémoire d'Ingénieur d'Etat en Informatique, Univ. Saad Dahlab, Blida, 2003-2004

[31] A. Mahdoum "Revue du livre *Representations for Genetic and Evolutionary Algorithms,* écrit par *Franz Rothlauf*, édité par SPRINGER-VERLAG Editions", revue du livre publiée dans The Computer Journal, Vol.49, N°5, Sept. 2006, Oxford Journals

[32] R.P. Dick, D. L. Rhodes and W. Wolf, "TGFF: Task graphs for free" Proceedings of the International Workshop Hardware/Software Codesign", pp.97-101, 1998

[33] http://helsinki.ee.princeton.edu/~dickrp/tgff

[34] W. T. Shiue, C. Chakrabarti "ILP-based scheme for low power scheduling and resource binding" Proc. Intl. Symp. on Circuits and Systems, Vol.3, pp 279-282, 2000

[35] N. Ranganathan, Ashok K. Murugavel "A Low Power Scheduler Using Game Theory" CODES+ISSS'03, pp.126-131, October 2003, Newport Beach, California, USA

[36] http://poppy.snu.ac.kr/inspire/CDFG/ftp/benchmarks.tar.gz

[37] R. Rudell "ESPRESSO" U. C. Berkeley, USA

[38] I.Pomeranz, K.T Cheng "STOIC: State assignment based on output/input functions" IEEE Trans. on CAD, Vol.12, N°8, August 93

[39] Alexander Saldanha, Tiziano Villa, RobertK.Brayton, Alberto L.Sangiovanni-Vincentelli "Satisfaction of input and output encoding constraints" IEEE Transactions on CAD, Vol.13, N°5, May 94

[40] C.J. Shi, J. A.. Brzozowski "An efficient algorithm for constrained encoding and its applications" IEEE Transactions on CAD, Vol.12, N°12, December 93

[41] Giovanni De Micheli, Robert K. Brayton, Alberto Sangiovanni-Vincentelli "Optimal states assignment for finite states machimes" IEEE Transactions on CAD- Vol. CAD-4, N°3, July 85

[42] S.Yang, M.J. Ciesielski "Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization" IEEE Trans. On CAD, Vol.10, pp.4-12, January 1991

[43] I.Pomeranz, K.T Cheng "STOIC : State assignment based on output/input functions" IEEE Trans. On CAD, Vol.12, N°8, August 93

[44] T.Villa, A.Sangiovanni-Vincentelli "NOVA :State assignment for optimal two-level logic implementations" IEEE Trans. On CAD, Vol.9, pp.905-924, Sept.1990

[45]R.L.Rudell, A.Sangiovanni-Vincentelli "Multiple-valued minimization for PLA optimization" IEEE Trans. On CAD Vol.6, pp.727-750, Sept. 1987

[46] D. Krumme, K. Venkataraman and G.Cybenko "Hypercube embedding is NP-complete" Proc. SIAM Hypercube conference, Sept.85

[47] Tsutomo Sasao "Variable assignment and output phase optimization of PLAs" IEEE Transactions on Computers. Vol.C-33, N°10, October 84

[48] Ming Huei Young and Saburo Muroga "Symmetric minimal covering problem and minimal PLAs with symmetric variables" IEEE Transactions on Computers. Vol.C-34, N°6, June 85

[49] P.J.M Van Laarhoven, E.H.L. Aarts, M.Davio "PHIPLA:A new algorithm for logic minimization" IEEE Proceedings of the 22nd ACM/IEEE Design Automation Conference Las Vegas, June 23-26, 1985

[50] H. Fujiwara, K.Kinoshita "A design of Programmable Logic Arrays with universal tests" IEEE Transactions on Computers. Vol.C-30, N°11, Nov. 81

[51] R. Brayton, A.Sangiovanni Vincentelli, G.Hachtel "Multi-level logic synthesis" Proc. IEEE, Vol.78, Feb.90

[52]G.Cybenko, D.Krumme, K.Venkataraman "Fixed hypercube embedding" Info. Process. Letters, Vol.25, April 87

[53] S.Devadas, H-T.Ma, R.Newton and A.Sangiovanni-Vincentelli "A synthesis and optimization procedure for fully and easily testable sequential machines" IEEE Transactions on CAD, Vol.8, October 89

[54] D. Ku, G. De Micheli "Relative scheduling under timing constraints" DAC'90, pp. 59-64

[55] N. K. Jha "Low power system scheduling and synthesis" ICCAD'2001, pp. 259-263

[56] C. G. Lyuh, T. Kim, C. L. Liu "An integrated data path optimization for low power based on network flow methods" ICCAD'2001, pp. 553-559

[57] I. Kadayif, M. T. Kandemir, U. Sezer "An ILP based approach for parallelizing applications in on-chip multiprocessors" DAC'02, pp. 703-708

[58] S. Davidson, D. Landskov, B. Shriver, P. Mallet "Some experiments in local microcode compaction for horizontal machines" IEEE Trans. Computers, C30, 1981, pp 460-477

[59] P.G. Paulin, J. P. Knight "Force-directed scheduling for the behavioural synthesis of ASIC's" IEEE Transactions on Computer-Aided-Design, Vol. 8, 1989, pp 661-679

[60] A. C. Parker, J. T. Pizarro, M. Mlinar "MAHA: a program for datapath synthesis" 23rd ACM/IEEE DAC, 1986, pp. 461-466

[61] B. M. Pangrle, D. Gajski "State synthesis and connectivity binding for microarchitecture compilation" ACM/IEEE ICCAD, 1986, pp. 210-213

[62] S. M. Heemstra De Groot, S. H. Gerez, O. E. Hermann "Range-chart guided iterative data flow graph scheduling" IEEE Trans. Circuits Systems, Vol. 39, 1992, pp 351-364

[63] W. Wolf, A. Takach, C. Y. Huang, R. Manno "The Princeton university behavioral synthesis system" 29th ACM/IEEE DAC, 1992, pp. 182-187

[64] R. Camposano, "Path-based scheduling for synthesis" IEEE Transactions, Vol. 10, N°1, 1991, pp 85-93

[65] R. A. Bergamashi, R. Composano, M. Payer "Scheduling under resource constraints and module assignment" INTEGRATION: the VLSI Journal, Vol. 12, 1991, pp 1-19

[66] A. Jerraya, I. Park, K. O'Brien "'AMICAL : an interactive high level synthesis" IEEE European Conference on Design Automation, 1993

[67] R. A. Bergamashi, D. J. Allerton "A graph-based silicon compiler for concurrent VLSI systems" IEEE CompEuro Conference, 1988, pp. 36-47

[68] K. Wakabayashi, H. Tanaka "Global scheduling independent of control dependencies based on condition vectors" 29th ACM/IEEE DAC, 1992, pp. 112-115

[69] R. A. Bergamashi, S. Raje, I. Nair, L. Trevillyan "Control-flow versus data-flow-based scheduling : combining both approaches in an adaptive scheduling system" IEEE Transactions on VLSI, Vol. 5, N° 1, 1997, pp 82-100

[70] Y. Xie, W. Wolf "Allocation and scheduling of conditional task graph in hardware/software co-synthesis" DATE'01, 2001, pp. 620-625

[71] J. Teich, T. Blickle, L. Thiele "System-level synthesis using evolutionary algorithms" CODES/CASHE'97, 1997, pp. 167-171

[72] H. Motallebpoor, C. Lucas, P. Jabbehdar, M. Nourani "Data flow graph scheduling using genetic algorithms" ICEE'99, 1999, pp. 125-132

[73] A.P. Chandrakasan, M.Potkonjak, R.Mehra, J. Rabaey & R.W. Brodersen "Optimizing Power using Transformations" IEEE Trans. on CAD of ICS, Vol. 14, N°1, Jan.95

[74] F. Najm "A Survey of Power Estimation Techniques in VLSI Circuits" IEEE Trans. On VLSI Systems, Vol.2, pp 446-455, Dec.94

[75] M. Pedram "Logic Synthesis and Physical Design for Low Power" IEEE Solid State Circuits & Technology Committee, Workshop on Low Power Electronics, August26, 1993

[76] M. Cirit "Estimating dynamic power consumption of CMOS circuits" IEEE Int. Conf.CAD, 1987

[77] A. Ghosh, S. Devadas, K. Keutzer & J. White "Estimation of average switching activities" in Proc. DAC,92

[78] N. Kimura & J. Tsujimoto "Calculation of total dynamic current of VLSI using a switch level timing simulator (RSIM_FX)" CICC, 1991

[79] A. Salz, M. Horowitz "IRSIM : An incremental MOS switch level simulator" Proc. 26th ACM/IEEE, DAC June89

[80] P.E. Landman, M.J. Rabaey "Power Estimation for High Level Synthesis" Proc.Euro.Conf.Design Automation93" Paris, France, Feb.93

[81] A. Shen, A. Ghosh, S. Devadas, K. Keutzer "On average power dissipation and random pattern testability of CMOS combinational logic networks" Proc.IEEE ICCAD, 1992

[82] S. Devadas, K. Keutzer, J. White "Estimation of Power Dissipation in CMOS Combinational Circuits Using Boolean Function Manipulation" IEEE Trans. on CAD, Vol. II, N°3, March 1992

[83] A.P. Chandrakasan, M. Potkonjak, J.Rabaey & R.W Brodersen "HYPER-LP: A System for Power Minimization Using Architectural Transformations" IEEE Journal of SSC, 1992

[84]H. Trickey "Flamel : A high level hardware compiler" IEEE Trans. on CAD, Vol.6, N°2, 1987

[85]B.S. Haroun & M.I. Elmasry "Architectural synthesis for DSP silicon compilers" IEEE Trans. on CAD for IC, Vol.8, N°4, 1989

[86] M. Nemani, F.N. Najm "Towards a high level power estimation capability" IEEE Trans. on CAD of ICS, Vol.15, N°6, June96

[87]P. Landman & J. Rabaey "Architectural power analysis : the dual bit type method" IEEE Trans. on VLSI Systems, pp 173,187, June 95

[88]D. Marculescu, R. Marculescu, M. Pedram "Information theoretic measures for power analysis" IEEE Trans. on CAD of ICS, Vol.15, N°6, June 96, pp 599-610

[89]A.M. Hill & S.M. Kang "Determining accuracy bounds for simulation based switching activity estimation" IEEE Trans. on CAD of ICS, Vol.15, N°6, June 96, pp 611-618

[90] K.A De Jong & W.M Spears "Using Genetic Algorithms to solve NP complete problems" Proc.3rd Int. Conf. Genetic Algorithms J.D. Schaffer Ed. George Mason University, 1989

[91] P. Chaudhuri "Parallel Algorithms : Design and analysis" Prentice Hall Editions

[92] C.J. Tseng, D.P. Siewiorek "Automated synthesis of data paths in digital systems" IEEE Trans. On CAD, Vol. CAD-5, pp.379-395, July 1986

[93] O. Wing, S. Huang, R. Wang "Gate matrix layout" IEEE Trans. On CAD vol.CAD-4 N°3, pp.220-231, July 1985

[94] N. Togawa, T. Hisaki, M. Yanagisawa, T. Ohtsuki "A high level synthesis system for digital signal processing based on data flow graph enumeration" IEICE Trans. On Fundamentals Vol.E81-A, N°12, pp.2563-2575, Dec. 1998

[95] H. Cho, G.D. Hachtel, E. Macii, B. Plessier, F. Somenzi "Algorithms for approximate FSM traversal" 30th DAC, pp.25-30, 1993

[96] Y.L. Wu, D. Chang, M. Marek-Sadowska, S. Tsukiyama "On improved FPGA Greedy routing architectures" IEICE Trans. On Fundamentals, Vol.E81-A, N°12, pp.2485-2491

[97] K.Yi, S.Y. Ohm, C.S. Jhon "An efficient FPGA technology mapping tightly coupled with logic minimization" IEICE Trans. On Fundamentals vol.E80-A N°10, pp.1807-1812, Oct.1997

[98] S.Y. Yuan, S.Y. Kuo "A new technique for optimization problems in graph theory", IEEE Trans. On Computers, Vol.47, N°2, pp.190-196, Feb. 1998

[99] J. Gu, Q.P. Gu, D.Z. Du "Convergence properties of optimization algorithms for the SAT problem" IEEE Trans. On Computers, Vol. 45, N°2, pp. 209-218, Feb. 1996

[100] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati and J.-Y. Chung "Imprecise computations" The IEEE Proceedings, Vol. 82, N° 1, pp. 83–94, 1994

[101] C. Rusu, R. Melhem and D. Mossé "Maximizing rewards for real-time applications with energy constraints" ACM Transactions on Embedded Computing Systems, Vol. 2, N° 4, pp. 537-559, 2003

[102] Y. Zhang, Z. Lu, J. Lach, K. Skadron and M. R. Stan "Optimal procrastinating voltage scheduling for hard real-time systems" Proceedings of the Design Automation Conference, pp. 905-908, 2005

[103] T. Okuma, H. Yassura and T. Ishihara "Software energy reduction techniques for variable voltage processors" IEEE Design & Test of Computers, Vol. 18, N° 2, pp. 31-41, 2001

[104] D. Shin, S. Lee and J. Kim "Intra-task voltage scheduling for low-energy hard real-time applications" IEEE Design & Test of Computers, Vol. 18, N° 2, pp. 20-30, 2001

[105] S. M. Martin, K. Flautner, T. Mudge and D. Blaauw "Combined dynamic voltage scaling and adaptive body biasing for low power microprocessors under dynamic workloads" Proceedings of the International Conference on Computer Aided Design, pp. 721-725, 2002

[106] L. A. Cortes, P. Eles and Z. Peng "Quasi-static assignment of voltages and optional cycles for maximizing rewards in real-time systems with energy constraints" Proceedings of the Design Automation Conference, pp. 889-894, 2005