Optimisation des chemins de données arithmétiques par l'utilisation des systèmes de numération redondants

Sophie Belloeil

▶ **To cite this version:**

Sophie Belloeil. Optimisation des chemins de données arithmétiques par l'utilisation des systèmes de numération redondants. Arithmétique des ordinateurs. Université Pierre et Marie Curie Paris VI, 2009. Français. ⟨NNT : ⟩. ⟨tel-01348675⟩

HAL Id: tel-01348675

https://hal.sorbonne-universite.fr/tel-01348675v1

Submitted on 25 Jul 2016

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

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

HAL Authorization

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

Spécialité Informatique

Présentée par Sophie BELLŒIL DUPUIS

Pour obtenir le titre de
Docteur de l'Université Paris VI

OPTIMISATION AUTOMATIQUE DES CHEMINS DE DONNÉES ARITHMÉTIQUES PAR L'UTILISATION DES SYSTÈMES DE NUMÉRATION REDONDANTS

Soutenue le 25 mai 2009, devant le jury composé de :

LIRIDA NAVINER	Rapporteur
EMMANUEL CASSEAU	Rapporteur
JEAN-CLAUDE BAJARD	Examinateur
NICOLAS FEL	Examinateur
ALIX MUNIER-KORDON	Examinatrice
HABIB MEHREZ	Directeur de thèse
ROSELYNE CHOTIN-AVOT	Co-Directrice de thèse

Résumé

Cette thèse présente l'optimisation des chemins de données arithmétiques par l'intégration automatique du système des notations redondantes dans le flot de conception VLSI, de façon à le rendre plus accessible.
Les travaux effectués se découpent en deux phases.

La première a pour objectif d'incorporer les opérateurs redondants et mixtes et le savoir-faire lié à leur usage dans la synthèse bas niveau. Les bonnes performances intrinsèques de ces opérateurs montrent l'intérêt potentiel d'une telle approche. Trois algorithmes d'optimisation sont proposés, basés sur la redéfinition des enchaînements entre opérateurs arithmétiques.

La seconde est consacrée à la mise en place de l'environnement de conception dans lequel seront utilisés ces algorithmes. Cet environnement répond aux besoins liés à l'arithmétique et fournit un langage de description de circuits ayant un haut niveau d'abstraction.

Ces algorithmes ont été appliqués sur différents circuits arithmétiques et les résultats obtenus confirment que l'intégration automatique de l'arithmétique redondante améliore nettement les performances par rapport à une implantation classique de ces circuits.

Mots Clefs :

Arithmétique des ordinateurs, arithmétique redondante, opérateurs arithmétiques, optimisations arithmétiques, synthèse de chemins de données, environnement de conception VLSI

Abstract

This thesis presents the optimisation of data-paths thanks to the automatic incorporation of the redundant number system on VLSI conception flow, in order to make it more accessible.
The work is divided into two parts.

The goal of the first part is to incorporate redundant and mixed operators and the knowledge in their usage to low level synthesis. The intrinsic good performances of those operators show the potential interest of such an approach. Three optimisation algorithms have been proposed, based on the choice of the notations used between arithmetical operators.

The second part is dedicated to the conception environment in which the algorithmes are going to be used. This environment meets the needs of the arithmetic and provides a circuits description langage with a high level of abstraction.

Those algorithms have been applied to several arithmetic circuits and the results confirm that the automatic incorporation of redundant arithmetic improves the performances with regard to the classical implementation of those circuits.

Key Words :

Computer arithmetic, redundant arithmetic, arithmetic operators, arithmetic optimisations, data-paths synthesis, VLSI conception flow

Remerciements

Je souhaite avant tout exprimer toute ma reconnaissance au professeur Alain Greiner, pour m'avoir accueillie dans le département SOC du LIP6 et pour l'intérêt qu'il a porté à mes travaux. Merci également à lui d'avoir "orienté mon choix" vers le DEA ASIME et vers une thèse.

J'adresse mes remerciements à madame Lirida Naviner, Maître de Conférences à l'ENST, ainsi qu'à monsieur Emmanuel Casseau, Professeur à l'ENSSAT, pour l'honneur qu'ils m'ont fait en acceptant d'être les rapporteurs de mon travail, et pour le soin qu'ils ont apporté à l'examen de ce manuscrit.

Je remercie également madame Alix-Munier Kordon, Professeur au LIP6, monsieur Jean-Claude Bajard, Professeur au LIRMM, et monsieur Nicolas Fel du CEA, pour avoir accepté d'être mes examinateurs.

Mes remerciements suivants seront pour mon directeur de thèse, le professeur Habib Mehrez, et pour ma co-directrice de thèse, madame Roselyne Chotin-Avot, qui ont encadré mes recherches avec disponibilité et gentillesse. Merci pour leurs conseils avisés, notament en ce qui concerne la rédaction de ce manuscrit.

Je remercie tous les membres du département SOC, principalement ceux avec qui j'ai collaboré autour des projets *ArithLib* et *Stratus* : Ludovic Noury, Christophe Alexandre, Jean-Paul Chaput, Hugo Clément, Christian Masson et Marek Sroka. Je leur suis très reconnaissante pour la contribution qu'ils ont apportée à ces travaux.

J'en profite pour remercier également tous les enseignants m'ayant donné goût à la micro-électronique et à la CAO durant mon cursus, principalement Anne Derieux, Franck Wajsbürt, Farouk Vallette et Ludovic Jacomme.

Evidemment j'embrasse toute ma famille, et les remercie vivement pour m'avoir permis d'effectuer mes études dans les meilleures conditions qui soient, et pour m'avoir soutenue dans tous mes choix.

Mes derniers remerciements seront pour toi Damien, pour les conseils avisés du collègue, pour le soutien et l'amour sans faille du mari, et ce, depuis bientôt dix ans. Une page est sur le point de se tourner, à nous d'écrire les suivantes et de les rendre tout aussi heureuses.

Table des matières

* **Résumé** **i**

* **Abstract** **iii**

* **Remerciements** **v**

* **Table des matières** **vii**

* **Table des figures** **xiii**

* **Liste des tableaux** **xvii**

* **Acronymes** **xix**

* **Introduction** **1**

1 Cadre de la thèse **7**
1.1 Optimisations arithmétiques 8
1.1.1 Phase d'initialisation 8
1.1.2 Opérateur de somme 13
1.1.3 Glissement 14
1.1.4 Circuits séquentiels 15
1.1.5 Méthodologie générale 16
1.2 Environnement de conception 17
1.2.1 Langage *mou* 17
1.2.2 Portabilité 18
1.3 Problématique 19
1.4 Conclusion 20

2 Arithmétique redondante **21**
2.1 Généralités 21
2.1.1 Représentations 21
2.1.2 Arithmétique mixte 22
2.2 Architectures 23
2.2.1 Addition 23
2.2.2 Multiplication 27
2.3 Performances 30
2.3.1 Addition 31
2.3.2 Multiplication 33

2.4 Applications . . . 34
2.4.1 PGCD . . . 34
2.4.2 DCT . . . 35
2.4.3 DCU . . . 36
2.5 Conclusion . . . 36

3 Etat de l'art **37**
3.1 Utilisation d'additionneurs mixtes dans la synthèse RTL . . . 37
3.1.1 Approche *tout en redondant* . . . 37
3.1.2 Approche *Allocation optimale* . . . 40
3.1.3 Gestion des blocs non arithmétiques et des multiplieurs . . . 41
3.1.4 Compromis Surface/Délai . . . 42
3.1.5 Prise en compte de la vue physique . . . 44
3.2 Extension de la représentation Carry-Save . . . 44
3.2.1 Optimisation des *CSA* au niveau bit . . . 45
3.2.2 Signaux *multiple vecteurs* . . . 45
3.3 Réorganisation des graphes . . . 45
3.3.1 Ordonnancement de graphe . . . 46
3.3.2 Exploration des optimisations temps / surface . . . 47
3.4 Synthèse haut niveau . . . 47
3.5 Conclusion . . . 48

4 Optimisation de chemins de données arithmétiques **49**
4.1 Définition du problème . . . 49
4.1.1 Optimisation arithmétique dans le flot de conception VLSI . . . 49
4.1.2 Approches choisies . . . 51
4.1.3 Formalisation . . . 54
4.2 Reconnaissance de motifs . . . 56
4.2.1 Algorithme . . . 56
4.2.2 Description des règles . . . 57
4.2.3 Ensemble des règles . . . 58
4.2.4 Exemple de reconnaissance de motifs . . . 62
4.2.5 Prise en compte des sorties multiples . . . 62
4.3 Algorithmes de labellisation . . . 65
4.3.1 Fonction d'allocation . . . 65
4.3.2 Algorithme *redondant dès que possible* . . . 66
4.3.3 Obtention de l'architecture à partir du graphe . . . 66
4.3.4 Exemple de la labellisation *redondant dès que possible* . . . 66
4.3.5 Fonction d'évaluation d'un nœud . . . 68
4.3.6 Fonction de coût d'un arc . . . 71
4.3.7 Algorithme d'allocation optimale . . . 71
4.3.8 Exemple de la labellisation version optimale . . . 72
4.3.9 Prise en compte des sorties multiples . . . 80

4.4 Conclusion 83

5 Environnement de conception **85**
5.1 Introduction 85
5.2 Environnements existants 86
5.2.1 Environnements industriels 86
5.2.2 Alliance 87
5.2.3 Coriolis 88
5.2.4 Genoptim 89
5.2.5 Conclusion 90
5.3 L'environnement Stratus 91
5.3.1 Présentation générale 91
5.3.2 Justification du choix du langage Python 92
5.3.3 Structure de données 92
5.3.4 Langage de description 94
5.3.5 Plate-forme de conception 96
5.3.6 Conclusion 97
5.4 Le *package* arithmétique 98
5.4.1 Présentation générale 98
5.4.2 Description arithmétique 99
5.4.3 Mécanismes d'optimisation 100
5.4.4 Bibliothèque ArithLib 100
5.5 Conclusion 101

6 Applications **103**
6.1 Filtre 103
6.2 Unité de calcul de distance 106
6.3 Opérateur de DCT 109
6.3.1 Partition 109
6.3.2 Opérateur complet 112
6.4 Transformée de Fourier 115
6.4.1 le "papillon" 115
6.4.2 FFT 118
6.5 Conclusion 121

∗ **Conclusions et perpectives** **123**

∗ **Publications** **127**

∗ **Bibliographie** **129**

Annexes **135**

A Description d'une règle **137**
A.1 Description xml . . . 137
A.2 Exemple . . . 137

B Ensemble des règles Carry-Save **139**
B.1 Transformation d'une addition en un nombre Carry-Save . . . 139
B.2 Modification de la sortie d'un multiplieur . . . 143

C Exemple **147**
C.1 Cellule . . . 147
C.1.1 Description haut niveau . . . 147
C.1.2 Description bas niveau . . . 147
C.2 Phase d'initialisation en redondant . . . 149
C.3 Cellule générée . . . 149

D Stratus **153**
D.1 Introduction . . . 154
D.1.1 Stratus . . . 154
D.1.2 Example . . . 156
D.2 Description of a netlist . . . 161
D.2.1 Nets . . . 161
D.2.2 Instances . . . 163
D.2.3 Generators . . . 164
D.3 Description of a layout . . . 165
D.3.1 Place . . . 165
D.3.2 PlaceTop . . . 166
D.3.3 PlaceBottom . . . 167
D.3.4 PlaceRight . . . 169
D.3.5 PlaceLeft . . . 170
D.3.6 SetRefIns . . . 171
D.3.7 DefAb . . . 172
D.3.8 ResizeAb . . . 173
D.4 Place and Route . . . 175
D.4.1 PlaceSegment . . . 175
D.4.2 PlaceContact . . . 176
D.4.3 PlacePin . . . 177
D.4.4 PlaceRef . . . 178
D.4.5 GetRefXY . . . 179
D.4.6 CopyUpSegment . . . 179
D.4.7 PlaceCentric . . . 180
D.4.8 PlaceGlu . . . 181
D.4.9 FillCell . . . 182

D.4.10 Pads . . . 182
D.4.11 Alimentation rails . . . 183
D.4.12 Alimentation connectors . . . 184
D.4.13 PowerRing . . . 184
D.4.14 RouteCk . . . 185
D.5 Instanciation facilities . . . 186
D.5.1 Buffer . . . 186
D.5.2 Multiplexor . . . 186
D.5.3 Shifter . . . 189
D.5.4 Register . . . 191
D.5.5 Constants . . . 191
D.5.6 Boolean operations . . . 193
D.5.7 Arithmetical operations . . . 194
D.5.8 Comparison operations . . . 195
D.6 Virtual library . . . 196
D.7 Data Base . . . 199
D.7.1 Model . . . 199
D.7.2 Nets . . . 200
D.7.3 Instances . . . 202

E ArithLib **205**
E.1 Adders . . . 205
E.1.1 lib.fixed.adder.ripple.Ripple Class Reference . . . 205
E.1.2 lib.fixed.adder.sklansky.Sklansky Class Reference . . . 206
E.1.3 lib.fixed.adder.multi_add.MultiAdd Class Reference . . . 208
E.1.4 lib.fixed.adder.mixed.Mixed Class Reference . . . 209
E.1.5 lib.fixed.adder.redundant.Redundant Class Reference . . . 211
E.1.6 lib.fixed.adder.derived.complementer.c2ripple.C2Ripple Class Reference . . . 212
E.1.7 lib.fixed.adder.derived.distance.distance.Distance Class Reference 213
E.1.8 lib.fixed.adder.derived.comparator.sklansky.Sklansky Class Reference . . . 215
E.1.9 lib.fixed.adder.derived.incrementer.sklansky.Sklansky Class Reference . . . 216
E.1.10 lib.fixed.adder.derived.increment_decrement.ripple.Ripple Class Reference . . . 218
E.2 Summation . . . 219
E.2.1 lib.fixed.sum.dadda.Dadda Class Reference . . . 219
E.3 Multipliers . . . 221
E.3.1 lib.fixed.multiplier.nr.Booth Class Reference . . . 221
E.3.2 lib.fixed.multiplier.mixed.Direct Class Reference . . . 222
E.3.3 lib.fixed.multiplier.redundant.Direct Class Reference . . . 224
E.3.4 lib.fixed.multiplier.constant.multcst.MultCst Class Reference . . . 225

E.4 Converters 226
E.4.1 lib.fixed.adder.converter.BsCs Class Reference 226
E.4.2 lib.fixed.adder.converter.CsBs Class Reference 228
E.5 Other 229
E.5.1 lib.fixed.other.bool.Bool Class Reference 229
E.5.2 lib.fixed.other.bool.Boolx Class Reference 230
E.5.3 lib.fixed.other.shifter.Shifter Class Reference 231
E.5.4 lib.fixed.other.coding.Encoder Class Reference 232

Table des figures

1.1 Légende . . . 9
1.2 Phase d'initialisation . . . 9
1.3 Impact des blocs non redondants . . . 10
1.4 Dédoublement . . . 11
1.5 Conditionnement par l'analyse . . . 12
1.6 Regroupement . . . 13
1.7 Fusion . . . 14
1.8 Glissement . . . 15
1.9 Circuit séquentiel . . . 16
1.10 Etapes de la méthode . . . 17

2.1 Additionneur séquentiel . . . 23
2.2 Additionneur de Sklansky . . . 24
2.3 Additionneur redondant . . . 24
2.4 Additionneur redondant Borrow-Save avec sortie Carry-Save . . . 25
2.5 Additionneur redondant Borrow-Save avec sortie Borrow-Save . . . 26
2.6 Additionneur mixte . . . 26
2.7 Additionneur mixte Borrow-Save avec sortie Carry-Save . . . 27
2.8 Additionneur mixte Borrow-Save avec sortie Borrow-Save . . . 27
2.9 Struture d'un multiplieur classique . . . 28
2.10 Struture d'un multiplieur redondant . . . 29
2.11 Struture d'un multiplieur mixte . . . 30
2.12 Comparaison en surface des additionneurs . . . 31
2.13 Comparaison en délai des additionneurs . . . 32
2.14 Comparaison en surface des multiplieurs . . . 33
2.15 Comparaison en délai des multiplieurs . . . 34

3.1 Solution au problème de troncature des bits de poids faible . . . 39
3.2 Solution au problème de hiérarchie . . . 40
3.3 Optimisation à travers les multiplexeurs . . . 41
3.4 Optimisation à travers les multiplieurs . . . 42
3.5 Architectures compromis temps/surface . . . 43
3.6 Ordonnancement de graphe . . . 47

4.1 Flot de conception VLSI . . . 50
4.2 Synthèse . . . 51
4.3 Allocation optimale en temps . . . 53
4.4 Problème dû à une soustraction . . . 54
4.5 Exemple de règle . . . 56

4.6 Exemple de règle n°1 . . . 59
4.7 Exemple de règle n°2 . . . 59
4.8 Exemple de règle n°3 . . . 59
4.9 Exemple de règle n°4 . . . 59
4.10 Règle erronée . . . 60
4.11 Exemple de règle n°5 . . . 60
4.12 Exemple de règle n°6 . . . 61
4.13 Exemple de règle n°7 . . . 61
4.14 Déroulement de la reconnaissance de motifs . . . 63
4.15 Reconnaissance de motifs : Problème des sorties multiples . . . 64
4.16 Reconnaissance de motifs : Prise en compte des sorties multiples . . . 64
4.17 Déroulement de la labellisation redondant dès que possible . . . 68
4.18 Différents cas de connexion . . . 69
4.19 Déroulement de la labellisation temps optimal 1 . . . 74
4.20 Déroulement de la labellisation temps optimal 2 . . . 75
4.21 Déroulement de la labellisation temps optimal 3 . . . 76
4.22 Déroulement de la labellisation temps optimal 4 . . . 77
4.23 Déroulement de la labellisation temps optimal 5 . . . 78
4.24 Déroulement de la labellisation temps optimal 6 . . . 79
4.25 Algorithme de labellisation *redondant dès que possible* : Prise en compte des sorties multiples . . . 80
4.26 Algorithme de labellisation *allocation optimale* : Prise en compte des sorties multiples . . . 81

5.1 Descriptions fournies par **Stratus** . . . 98
5.2 Diagramme de classes . . . 99

6.1 Architecture d'un filtre . . . 103
6.2 Résultats du filtre : Surface . . . 104
6.3 Résultats du filtre : Chaîne longue . . . 104
6.4 Résultats du filtre : Temps d'exécution . . . 105
6.5 Architectures optimisées d'un filtre avec quatre coefficients . . . 106
6.6 Architecture d'une DCU . . . 107
6.7 Résultats de la DCU : Surface . . . 108
6.8 Résultats de la DCU : Chaîne longue . . . 108
6.9 Résultats de la DCU : Temps d'exécution . . . 109
6.10 Partition de la DCT . . . 110
6.11 Résultats de la partition de la DCT : Surface . . . 111
6.12 Résultats de la partition de la DCT : Chaîne longue . . . 111
6.13 Résultats de la partition de la DCT : Temps d'exécution . . . 112
6.14 Opérateur de DCT . . . 113
6.15 Résultats de la DCT complète : Surface . . . 114
6.16 Résultats de la DCT complète : Chaîne longue . . . 114
6.17 Résultats de la DCT complète : Temps d'exécution . . . 115

6.18 Architecture d'un papillon FFT 115
6.19 Résultats du papillon : Surface 116
6.20 Résultats du papillon : Chaîne longue 117
6.21 Résultats du papillon : Temps d'exécution 117
6.22 Architecture d'une FFT 118
6.23 Résultats de la FFT : Surface 119
6.24 Résultats de la FFT : Chaîne longue 119
6.25 Résultats de la FFT : Temps d'exécution 120

A.1 Exemple de règle 137

B.1 Motif 1 139
B.2 Motif 2 139
B.3 Motif 3 140
B.4 Motif 4 140
B.5 Motif 5 140
B.6 Motif 6 140
B.7 Motif 7 141
B.8 Motif 8 141
B.9 Motif 9 141
B.10 Motif 10 141
B.11 Motif 11 142
B.12 Motif 12 142
B.13 Motif 13 142
B.14 Motif 14 142
B.15 Motif 15 143
B.16 Motif 16 143
B.17 Motif 17 143
B.18 Motif 18 144
B.19 Motif 19 144
B.20 Motif 20 144
B.21 Motif 21 144
B.22 Motif 22 145
B.23 Motif 23 145
B.24 Motif 24 145

C.1 Exemple 150

Liste des tableaux

2.1 PGCD : Résultats 35
2.2 PGCD étendu : Résultats 35
2.3 DCT : Résultats 35
2.4 DCU : Résultats 36

3.1 Résultats Kim : Approche tout en redondant 38
3.2 Performances compromis temps/surface 43

4.1 Architectures 67
4.2 Fonctions d'évaluation des additionneurs 70
4.3 Fonctions d'évaluation des multiplieurs 70

C.1 Paramètres de l'initialisation en redondant 149

Acronymes

ASIC Application-Specific Integrated Circuit - Circuit intégré spécifique conçu pour réaliser de façon optimale une fonction précise dans un système

BS Notation Borrow-Save

CAO Conception Assistée par Ordinateur

CS Notation Carry-Save

CSA Carry-Save adder

C2 Notation en Complément à 2

DAG Direct Acyclic Graph - Graphe acyclique orienté

DCT Discrete Cosine Transform - Transformée cosinus discrète

DCU Distance Computation Unit - Unité de calcul de distance

FA Full-Adder - Cellule élémentaire additionneur complet

FIR Finite Impulse Response - Filtres numériques à réponse impulsionnelle finie

FFT Fast Fourrier Transform - Transformée discrète de Fourrier rapide

FPGA Field Programmable Gate Array - Circuit spécifique dont la fonctionnalité est programmée par l'utilisateur

HA Half-Adder - Cellule élémentaire demi-additionneur

LSB Lower Significant Bit - Bit/chiffre d'un nombre de poids le moins significatif

MSB Most Significant Bit - Bit/chiffre d'un nombre de poids le plus significatif

NR Notation Non-Redondante

PGCD Plus Grand Commun Diviseur

PPM Plus-Plus-Minus - Cellule élémentaire plus-plus-moins

R Notation Redondante

RNS Residu Number System - Notation par résidus

RTL Register Transfert Level - Niveau transfert de registres

VHDL VHSIC (Very High Speed Integrate Circuit) Design Langage - Langage de description de circuits intégrés

VLSI Very Large Scale Integration - Intégration à très grande échelle

Introduction

Contexte de recherche

Depuis l'apparition des circuits intégrés dans les années cinquantes, leurs performances ont évolué de manière exponentielle. La loi de Moore, exprimée en 1975, indiquait que la densité des transistors dans les circuits intégrés doublerait tous les deux ans. Cette loi s'est révélée exacte jusqu'à nos jours. Un tel rythme d'évolution technologique implique une mise sur le marché de plus en plus rapide des composants. Une telle rapidité couplée avec une complexité croissante des circuits accroît grandement la pression pour les concepteurs de circuits.

La tendance actuelle pour les concepteurs est donc de se focaliser principalement sur le comportement et la validation fonctionnelle de leur circuits, en laissant de côté les aspects concernant l'architecture, les performances, les optimisations possibles ... domaines typiquement délégués aux chaînes de *CAO*. Cette tendance est logiquement couplée avec le développement de langages de description de circuits ayant un niveau d'abstraction de plus en plus haut, de façon à atteindre des niveaux d'intégration très élevés.

C'est dans ce contexte que l'utilisation d'outils optimisant les circuits de façon automatique selon un critère donné (surface, chaîne longue, etc ...) trouve toute sa signification. Ceci se vérifie principalement dans des domaines peu connus des concepteurs de circuits, tel celui de l'optimisation arithmétique. Ce type d'optimisation consiste à utiliser différentes architectures d'opérateurs arithmétiques afin d'étudier leur impact sur les performances des circuits. Cela permet ainsi d'obtenir l'architecture la plus performante selon le critère souhaité.

L'arithmétique redondante a été introduite par Avizienis dans les années soixante [Avi61]. Ce type d'arithmétique permet de coder les entiers de telle sorte qu'un même nombre peut avoir différents codages. En effet, elle est constituée, entre autres, de la représentation Carry-Save, dans laquelle un nombre est codé par deux éléments dont la somme représente ce nombre. Cette représentation est dite à "retenues conservées" car les retenues sont gardées au niveau du résultat final au lieu d'être propagées. Elle permet de ce fait de se défaire de la séquentialité inhérente aux propagations de retenue i.e. de réaliser des additions totalement parallèles. Les additions se font donc en temps constant quelle que soit la dynamique des opérandes.

On en déduit aisément que l'utilisation d'une telle arithmétique peut permettre d'améliorer de façon significative les performances des circuits, par rapport à l'utilisation de l'arithmétique classique en complément à deux, avec laquelle l'affranchissement de la propagation de la retenue est impossible.

L'arithmétique redondante est couramment utilisée de façon transparente dans différents opérateurs arithmétiques tels que la multiplication et la division, pour effectuer rapidement les sommes internes à ces opérateurs et mémoriser les résultats intermédiaires. Son utilisation de façon explicite est cependant beaucoup moins répandue. Elle commence à se généraliser seulement depuis quelques années. En effet, son utilisation est délicate pour des concepteurs de circuits n'ayant pas forcément les connaissances requises en arithmétique.

Différentes études ont donc été menées dans le but d'automatiser l'utilisation de ce système de numération dans la synthèse de circuits, plus particulièrement dans la synthèse d'architectures possédant des enchaînements d'opérateurs arithmétiques importants, architectures appelées *chemins de données arithmétiques*. Ces études démontrent l'intérêt de cette approche : une amélioration importante des performances des chemins de données arithmétiques, ce, du point de vue de deux des critères principaux d'optimisation que sont la surface et le délai.

Cadre de la thèse

Cette thèse a été réalisée au sein du département "Systèmes Embarqués sur Puce" (SOC) du Laboratoire d'Informatique de Paris 6 (LIP6).

Elle s'inscrit comme un prolongement de la thèse de Yannick Dumonteix réalisée dans ce même laboratoire et soutenue en octobre 2001 [Dum01] : "Optimisation de chemins de données arithmétiques par l'utilisation de plusieurs systèmes de numération". Cette thèse avait pour but d'étudier l'impact de l'utilisation de l'arithmétique redondante dans le flot de conception *VLSI*. Elle se terminait par la spécification d'une méthodologie de conception.

Elle est par ailleurs étroitement liée à la plate-forme Coriolis [Cor], une plate-forme de prototypage rapide du laboratoire LIP6.

Objectifs

Dans le contexte de la conception de chemins de données arithmétiques, notre objectif est de mettre en place une méthodologie *d'aide à la conception* se focalisant autour de deux aspects principaux que sont l'optimisation automatique de ces chemins de données grâce à l'utilisation de l'arithmétique redondante et le fait de faciliter le travail du concepteur.

Optimiser les circuits en utilisant l'arithmétique redondante

Notre but est de tirer parti des performances intrinsèques des opérateurs arithmétiques redondants, tels que les additionneurs et les multiplieurs. Leur utilisation étant

relativement méconnue, nous voulons fournir des outils permettant d'effectuer, de façon automatique, la modification d'un chemin de données arithmétique décrit en arithmétique classique afin d'utiliser ce type d'opérateurs.

Nous voulons évaluer différentes approches, tant du point de vue des algorithmes utilisés (temps de calcul, adaptation à nos besoins) que des résultats obtenus (performances des circuits). L'objectif final est l'amélioration des performances du circuit, des points de vue du délai et de la surface, tout en conservant le comportement initial.

Notons que nous nous focalisons ici sur les deux critères délai et surface en laissant de côté l'aspect consommation. Cet aspect devient de plus en plus important depuis quelques années, les téléphones mobiles et autres ordinateurs portables étant le fer de lance de cet intérêt. Cependant, les outils nécessaires à son évaluation sont encore émergeants, à l'opposé des outils évaluant le délai et la surface, fournis par toutes les chaînes de *CAO* actuelles.

De plus, le délai et la surface sont indépendants du contexte d'utilisation des différents blocs, les évaluer pour un circuit en fonction des blocs qui le composent ne pose pas de problème majeur. A contrario, la consommation est fortement corrélée au contexte d'utilisation : influence, interaction entre les blocs, évolution de la consommation pendant le fonctionnement normal du circuit, etc. [Pig04, AMP+08]. Déterminer la consommation d'un circuit en fonction de celle des différents blocs le composant n'est pas une chose aisée.

Une solution serait d'évaluer la consommation des différents choix architecturaux que l'on serait amené à faire en effectuant une simulation logique incluant des modèles d'énergie, mais cela s'avèrerait très coûteux pour des chemins de données complexes.

Faciliter le travail des concepteurs

Notre but est que les concepteurs de circuits aient à disposition un environnement facilitant le travail de conception, entre autres, un langage de description ayant un haut niveau d'abstraction et étant adapté à la description de circuits arithmétiques.

Cela sous entend par exemple de pouvoir effectuer la description d'un chemin de données arithmétique de façon simplifiée i.e. en considérant les opérateurs arithmétiques comme de simples mnémoniques. Ainsi, le concepteur s'affranchit des problèmes d'architectures d'opérateurs ou de dynamiques des opérandes. L'outil devra donc compléter les informations manquantes de manière automatique. Il fournira également des services permettant aux concepteurs de vérifier aisément leurs circuits.

Pour cela, toutes les étapes du flot de conception visant à obtenir une vue structurelle en portes d'un circuit seront encapsulées dans un seul et même outil.

Notre but est également d'incorporer toutes les étapes jusqu'au dessin des masques et l'analyse des temps de propagation pour unifier le flot complet, étant entendu que notre outil ne sera qu'un interfaçage faisant appel à des outils extérieurs.

Contenu du mémoire

Arithmétique redondante

Nous commençons tout d'abord par expliciter le cadre de cette thèse dans le Chapitre 1, principalement en faisant une synthèse des conlusions de la thèse de Yannick Dumonteix. Suite à cela, nous exposons notre problématique.

Les travaux de cette thèse étant essentiellement basés sur l'utilisation de l'arithmétique redondante, nous définissons ensuite naturellement cette arithmétique dans le Chapitre 2, et présentons les architectures et les performances des opérateurs qui lui sont associés.

Ce chapitre est également consacré à l'introduction de l'arithmétique redondante au coté de l'arithmétique classique pour laquelle nous définissons l'arithmétique dite *mixte*.

Optimisations automatiques

Le but principal de cette thèse est d'automatiser l'utilisation de l'arithmétique redondante dans le flot de conception de circuits afin de la rendre plus accessible aux concepteurs de circuits n'ayant pas forcément le savoir-faire arithmétique nécessaire.

Nous commençons donc par effectuer un état de l'art des travaux déjà existants dans le Chapitre 3.

Nous présentons ensuite de façon détaillée dans le Chapitre 4 les différents algorithmes d'optimisation que nous avons mis en œuvre.

Environnement de conception

Nous nous intéressons, dans le Chapitre 5, à l'environnement de conception dans lequel vont être utilisés nos algorihmes.

Nous faisons un bref récapitulatif de différents environnements de conception existants de façon à montrer qu'aucun de ces environnements ne répond parfaitement à nos besoins, c'est pourquoi nous avons fait le choix d'enrichir un environnement, l'environnement Coriolis, avec, entre autres, un nouveau langage procédural de description de circuits. Ce langage a été conçu de façon à être le plus clair et intuitif possible tout en étant assez complet (description d'une vue structurelle, d'une vue physique, de stimuli, ...), afin de ne pas le restreindre au cadre de la conception de chemins de données arithmétiques. D'un point de vue arithmétique, ce langage a la particularité de répondre aux besoins de différenciation entre les représentations arithmétiques.

Résultats

Nous présentons dans le Chapitre 6 les architectures des différents circuits que nous avons utilisés pour évaluer notre outil.

Cette évaluation consiste à comparer les différents algorithmes d'optimisation que nous avons développés. Cette comparaison est faite sur les performances de ces circuits en utilisant l'arithmétique classique par rapport à celles obtenues après optimisation par nos algorithmes.

Nous terminons par une conclusion et des perspectives.

Chapitre 1 Cadre de la thèse

Cette thèse s'inscrit dans la continuité de la thèse de Yannick Dumonteix [Dum01] soutenue en octobre 2001 : "Optimisation de chemins de données arithmétiques par l'utilisation de plusieurs systèmes de numération". Il a en effet étudié l'impact de l'introduction de l'arithmétique redondante dans le flot de conception, étude qui lui a permis de formaliser une méthode complète d'outils d'optimisation utilisant l'arithmétique redondante.

Il a tout d'abord étudié l'usage conjoint de l'arithmétique redondante et de l'arithmétique classique. Il a pour cela défini une nouvelle arithmétique qu'il a appelée *arithmétique mixte*, étant la combinaison de ces deux représentations. Dans le but de montrer l'intérêt de cette nouvelle arithmétique, il a présenté les architectures des différents opérateurs en faisant une étude comparative entre les performances intrinsèques des opérateurs redondants et mixtes et celles des opérateurs classiques.

Il a ensuite déterminé l'impact de l'utilisation de ces nouvelles architectures dans la conception de circuits en étudiant les enchaînements d'opérateurs. Cette étude lui a permis de proposer des règles d'optimisations automatisables consistant à remplacer un enchaînement d'opérateurs utilisant l'arithmétique classique par des opérateurs utilisant l'arithmétique redondante. L'utilisation de ces règles permet d'optimiser automatiquement les chemins de données arithmétiques.

Son travail s'est concrétisé par la spécification d'une *méthodologie d'aide à la conception de chemins de données*. Dans cette spécification il a tout d'abord explicité un cadre de développement permettant la mise en œuvre automatisée des règles d'optimisation présentées, et a défini ensuite ce qu'il a appelé un *proto-langage*.

Dans un premier temps, nous faisons une synthèse de la méthode d'optimisation formalisée par Yannick Dumonteix. Nous présentons les différentes étapes d'optimisation, étapes qui ont servi de base à nos travaux. Nous présentons également l'environnement de conception, avec entre autres les fonctionnalités de son proto-langage, correspondant principalement aux contraintes dues à l'arithmétique redondante.

Dans un second temps, nous exposons notre problématique. Basés sur les conclusions de Yannick Dumonteix, nos travaux se résument à l'utilisation de façon automatique des règles qu'il a présenté ainsi que la mise en place du proto-langage qu'il a défini.

1.1 Optimisations arithmétiques

La méthodologie d'optimisation présentée par Yannick Dumonteix comprend deux phases distinctes :

- l'optimisation des équations arithmétiques et logiques, composée de plusieurs étapes que nous allons détailler,
- la redistribution des parties arithmétiques et logiques.

1.1.1 Phase d'initialisation

Passage tout en redondant

L'utilisation de l'arithmétique redondante consiste à instancier des opérateurs redondants à la place des opérateurs classiques d'un chemin de données arithmétique, de telle façon que :

- son interface soit inchangée,
- son comportement soit inchangé,
- son architecture soit modifiée de façon à exploiter au mieux les opérateurs redondants.

Cette méthode d'initialisation d'un chemin de données arithmétique est présentée comme apportant un gain toujours positif grâce aux meilleures performances intrinsèques des opérateurs redondants. Nous reviendrons en détails sur les performances des différents opérateurs arithmétiques dans le chapitre suivant.

Dans la suite de ce manuscrit, les signaux non redondants (noté *NR*) seront représentés par une simple flèche et les signaux redondants (noté R) par une double flèche (en effet, les nombres redondants sont codés sur deux termes). Les opérateurs redondants (opération entre deux nombres redondants) et mixtes (opération entre un nombre redondant et un nombre non redondant) seront donc représentés avec des signaux redondants (doubles flèches) pour leurs entrées/sorties redondantes et des signaux non redondants (simples flèches) pour leurs entrées/sorties non redondantes.

Un code des couleurs sera également adopté de façon à rendre plus lisibles les opérateurs utilisés : cyan pour les opérateurs classiques, magenta pour les opérateurs redondants et mixtes, et blanc pour les convertisseurs permettant de passer d'une représentation à l'autre.

La Figure 1.1 présente les différentes conventions utilisées.

Considérons l'exemple de la Figure 1.2. Notre postulat de base est que les additionneurs redondants sont plus petits et plus rapides que les additionneurs classiques. De plus, comme nous le verrons en détails dans le Chapitre 2, un nombre redondant codé en représentation Carry-Save est la somme de deux éléments, on parle alors de somme non encore effectuée ; cela permet d'effectuer la somme de quatre termes avec un seul additionneur redondant plutôt qu'avec trois additionneurs classiques.

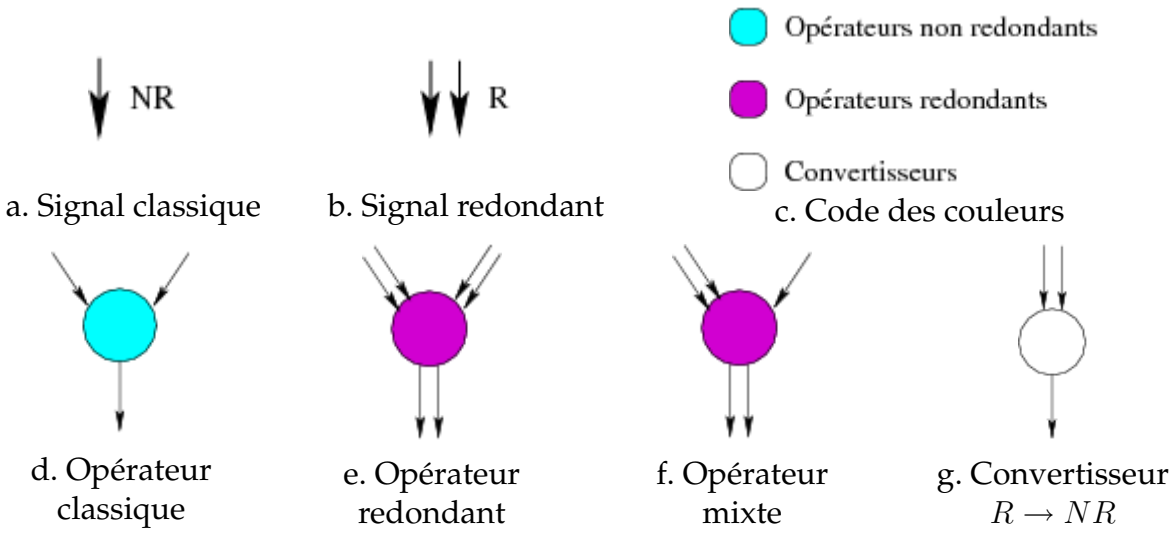


FIG. 1.1 – Légende

On déduit aisément que le *passage tout en redondant* (cf. Figure 1.2.b) génère une architecture plus petite et plus rapide que celle en arithmétique classique (cf. Figure 1.2.a).

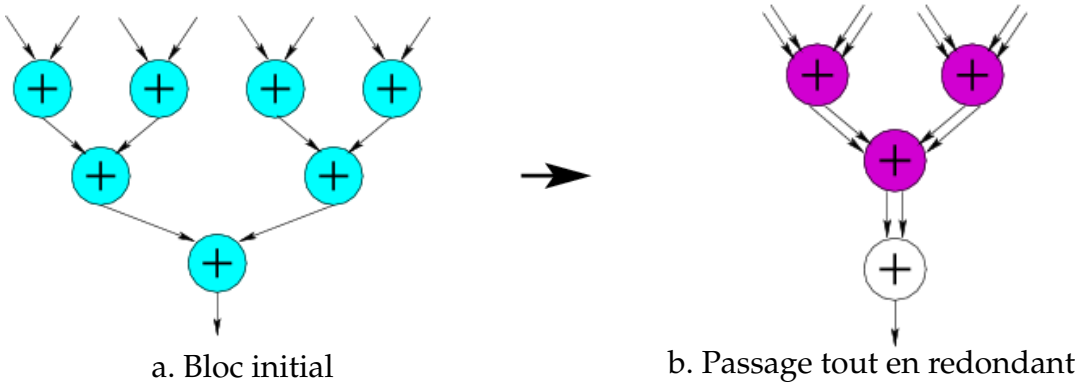


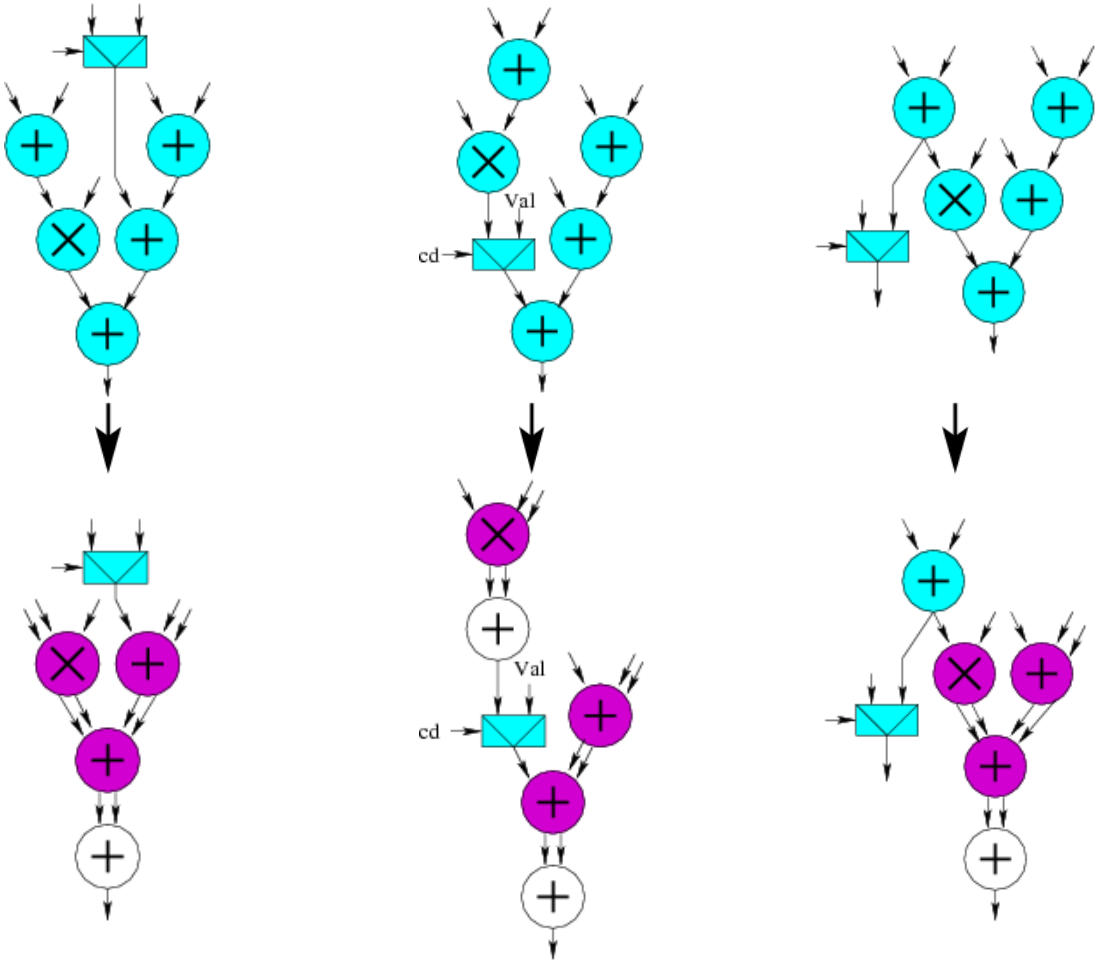
FIG. 1.2 – Phase d'initialisation

Passage en redondant dès que possible

L'exemple présenté précédemment est un cas *idéal* se réduisant uniquement à une succession d'opérateurs arithmétiques. Un chemin de données arithmétique ne se réduit cependant pas à cela, il faut également prendre en compte les opérateurs non arithmétiques, tels que les multiplexeurs et les opérateurs booléens. Ces opérateurs doivent garder les signaux à leur interface en non redondant, il est donc possible qu'ils *compromettent* le passage en redondant en obligeant l'utilisation d'opérateurs classiques. C'est pourquoi il ne faut pas parler de *passage tout en redondant* mais de *passage en redondant dès que possible*.

Trois exemples sont présentés dans la Figure 1.3. Ces exemples représentent les trois cas d'enchaînement qui peuvent survenir entre un opérateur arithmétique et un opérateur non arithmétique :

- Figure 1.3.a : le bloc non arithmétique n'est pas situé *à l'intérieur* de la chaîne arithmétique, il n'a donc pas d'influence sur les optimisations possibles,
- Figure1.3.b : le bloc non arithmétique est à l'intérieur de la chaîne arithmétique, les signaux à son interface doivent donc rester en représentation non redondante, on parle alors de limitation de type opérateur,
- Figure 1.3.c : la sortie d'un opérateur arithmétique est connecté à la fois à un bloc arithmétique et au bloc non arithmétique, cette seconde connexion empêche de passer ce signal en représentation redondante, on parle alors de limitation de type donnée.



a. Bloc sans influence

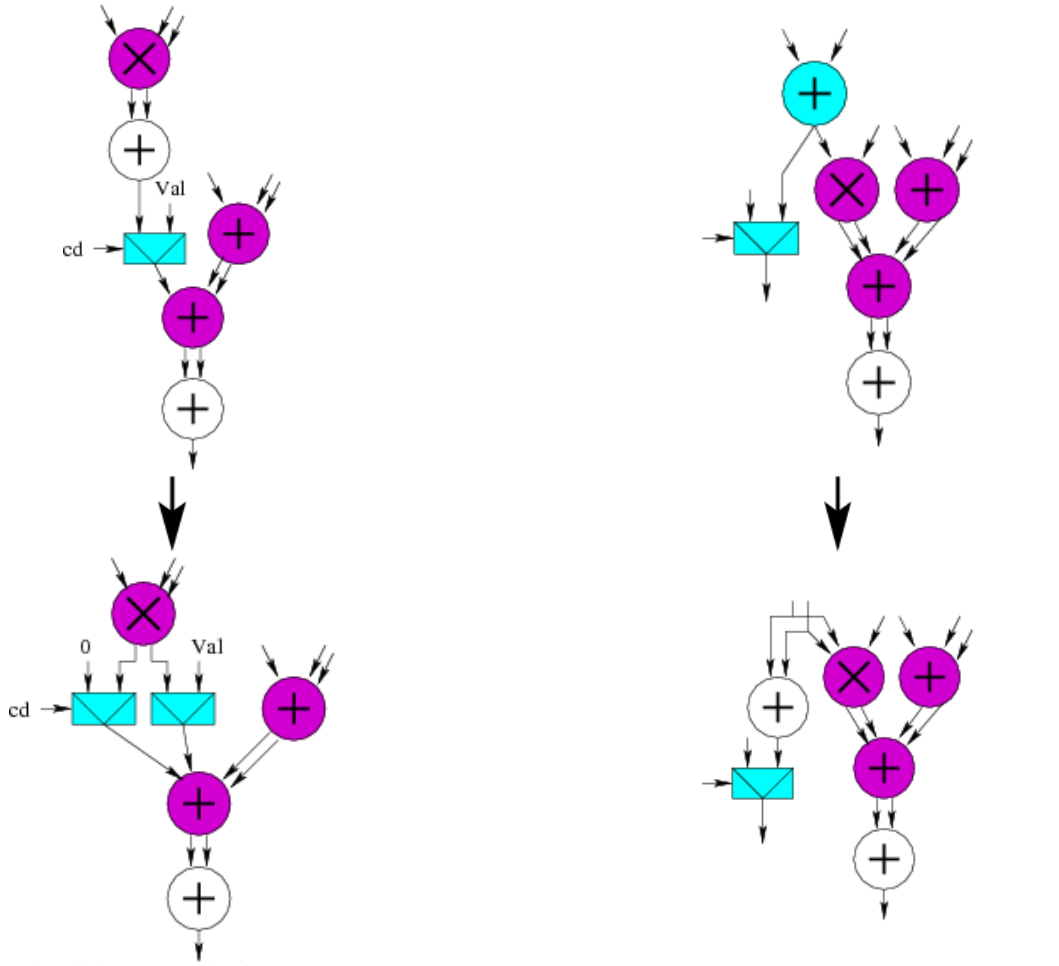
b. Limitation de type opérateur

c. Limitation de type donnée

FIG. 1.3 – Impact des blocs non redondants

De façon à pouvoir utiliser quand même des opérateurs redondants et ainsi améliorer les performances du chemin de données arithmétique, des solutions sont proposées pour les deux cas dans lesquels le bloc non arithmétique empêche le passage en redondant comme montré dans la Figure 1.4 :

- dans le cas de la limitation de type opérateur, il est possible de dédoubler l'opérateur de façon à relier chacun des deux opérateurs à un des termes du nombre redondant comme montré dans la Figure 1.4.a,
- dans le cas de la limitation de type données, il est possible de dédoubler la donnée, de ce fait, on relie la donnée en redondant au bloc arithmétique, et on relie la donnée en non redondant au bloc non arithmétique comme montré dans la Figure 1.4.b.



a. Dédoublement de l'opérateur b. Dédoublement de la donnée

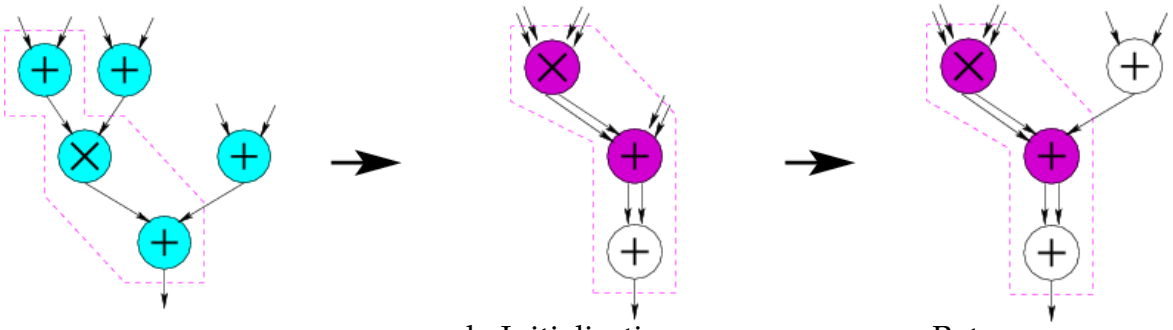
FIG. 1.4 – Dédoublement

Conditionnement par l'analyse

Les optimisations présentées précédemment apportent un gain substantiel dû aux bonnes performances intrinsèques des opérateurs redondants. Cependant, ce gain n'est pas forcément optimal pour le délai car ces optimisations ne prennent pas en compte la dimension temporelle du chaînage des opérateurs.

En effet, dans certains cas, il est préférable de ne pas transformer un opérateur en sa version redondante pour optimiser la chaîne longue. Plus précisément, l'objectif est d'exploiter les déséquilibres de chemins de façon à réduire le nombre d'entrées redondantes des opérateurs se situant dans la chaîne longue afin d'utiliser des opérateurs mixtes. Cela permet de cette façon de minimiser la chaîne longue car les opérateurs mixtes sont plus petits et plus rapides que leur équivalent redondant. Le principe est donc de partir de la version totalement redondante d'un chemin de données et d'insérer à bon escient des convertisseurs de notation redondante vers notation classique (noté $R \rightarrow NR$) permettant ainsi l'utilisation des opérateurs mixtes.

Dans l'exemple présenté dans la Figure 1.5.a, la chaîne longue passe par le chemin de gauche. Les chemins entre les deux entrées de l'opérateur de sortie sont donc déséquilibrés, l'un étant plus long que l'autre. L'idée est que l'insertion d'un convertisseur $R \rightarrow NR$ ne modifie pas le fait que la chaîne longue passe par le chemin de gauche (le chemin de droite est toujours plus court), elle permet par contre de transformer l'opérateur redondant (cf. Figure 1.5.b) en un opérateur mixte plus rapide (cf. Figure 1.5.c).



a. Bloc

b. Initialisation en redondant

c. Retour en non redondant

FIG. 1.5 – Conditionnement par l'analyse

Remarques :

1. Cette technique repose sur l'analyse des temps de propagation et doit être réitérée après chaque modification de l'architecture pour converger vers la solution optimale ; le coût potentiel d'un tel mécanisme paraît donc assez élevé.
2. Cette technique est à effectuer uniquement si l'optimisation voulue est du point de vue du délai ; en effet, elle améliore le délai au détriment de la surface, par rapport à l'optimisation tout en redondant.

1.1.2 Opérateur de somme

Une fois le chemin de données obtenu avec des opérateurs redondants, des optimisations sont encore possibles : l'étude des enchaînements d'opérateurs permet d'extraire de nouvelles règles d'optimisations.

A la base de ces règles se trouve l'opérateur de somme qu'est l'arbre de Wallace [Mul89]. Ces règles ne font donc pas parti à proprement parler des optimisations liées à l'arithmétique redondante, mais leur sont connexes.

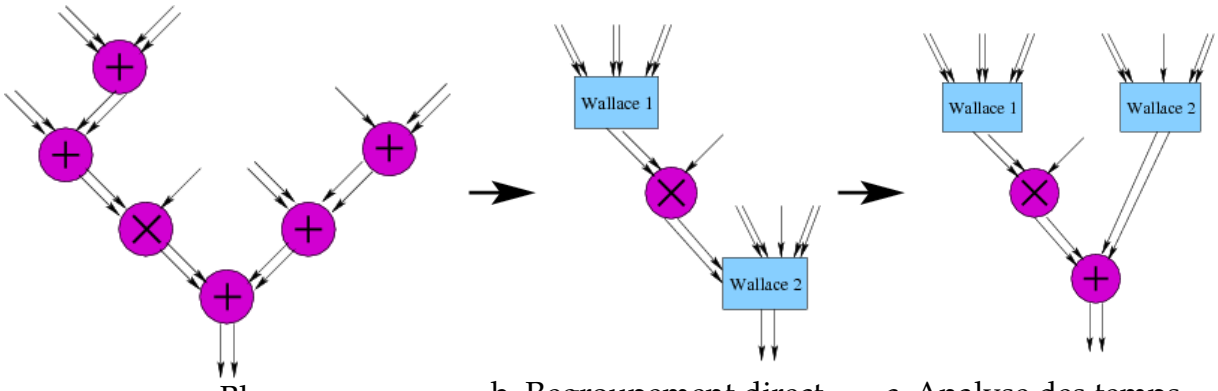
Regroupement

Il s'agit du regroupement d'une suite d'opérateurs effectuant la même opération au sein d'un unique opérateur.

La Figure 1.6 présente un exemple d'un tel regroupement dans lequel une suite d'additionneurs est regroupée au sein d'un arbre de Wallace.

Cependant, l'architecture présentée dans l'exemple est une architecture typique menant à une contre-performance : le principe est le même que le *passage tout en redondant* nécessitant un retour en non redondant de façon à être optimal. En effet, en effectuant un regroupement de tous les additionneurs en un seul abre de Wallace, le temps de propagation subit un retard (cf. Figure 1.6.b) puisque le temps de propagation d'un arbre de Wallace est supérieur à celui d'un additionneur redondant. Ainsi il serait plus judicieux de garder l'additionneur redondant de sortie et d'utiliser un arbre de Wallace en parallèle avec la multiplication, plutôt que d'utiliser un arbre de Wallace après la multiplication. Une analyse de temps s'impose donc afin de *découper* les regroupements (cf. Figure 1.6.c).

La méthode à mettre en place est une nouvelle fois d'effectuer tous les regroupement possibles dans un premier temps puis de les *découper* en fonction de l'analyse des temps.



a. Bloc b. Regroupement direct c. Analyse des temps

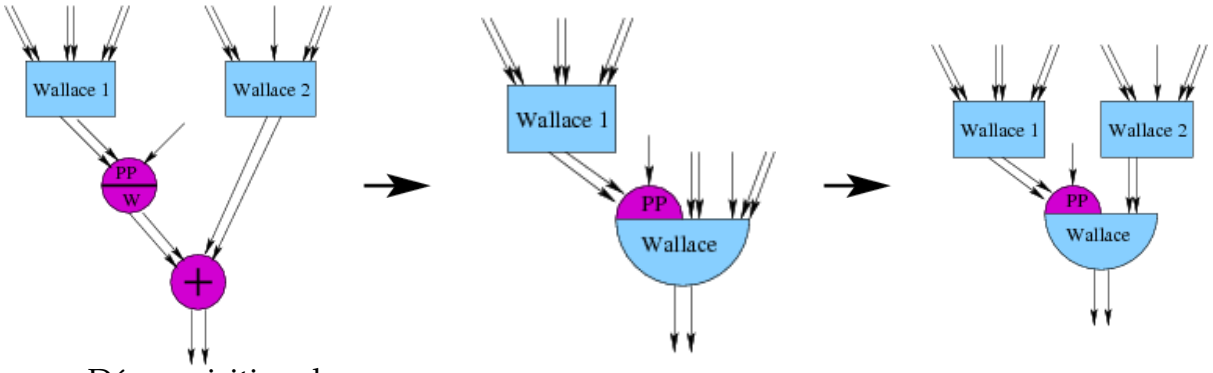
FIG. 1.6 – Regroupement

Fusion

Il s'agit de l'absorption d'un opérateur par un autre de nature différente.

La Figure 1.7 présente l'exemple d'une fusion, dans laquelle un multiplieur est découplé en deux parties (cf. Figure 1.7.a) de façon à pouvoir effectuer un regroupement d'additionneurs avec l'arbre de Wallace interne du multiplieur.

Dans cet exemple, on soulève le même problème que précédemment : effectuer des fusions sans analyse des temps peut mener à des contre-performances (cf. Figure 1.7.b). La solution à adopter est donc la même : effectuer une analyse temporelle de façon à *découper* les fusions (cf. Figure 1.7.c).



a. Décompisition du multiplieur : produits partiels + somme

b. Fusion directe

c. Analyse des temps

FIG. 1.7 – Fusion

Synthèse

Comme cela a été montré dans les deux exemples précédents, le gain en temps n'est pas toujours positif en ce qui concerne les regroupements et les fusions. En effet, ces optimisations dépendent très fortement du contexte.

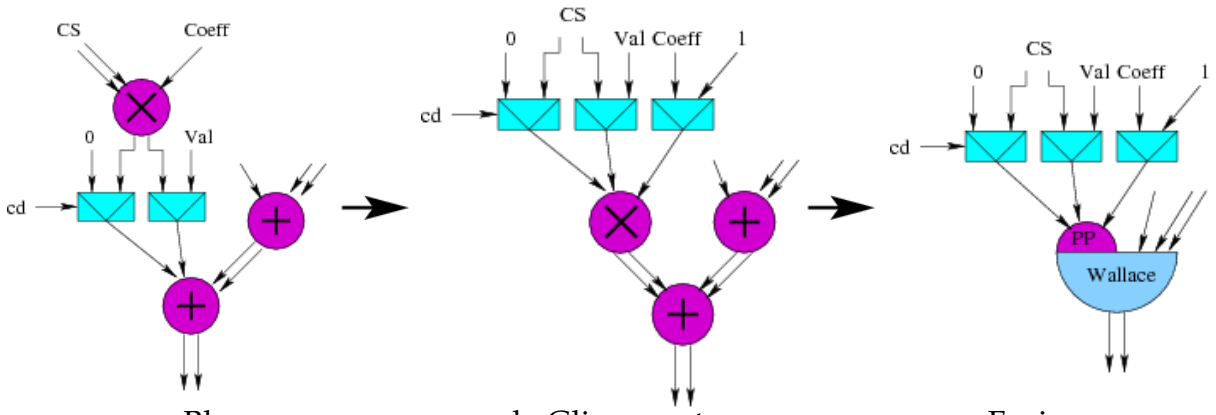
Il faut donc mettre en place une méthodologie d'optimisation pouvant calculer les temps de propagation des opérateurs de façon à être capable de corriger les éventuels effets indésirables.

1.1.3 Glissement

Le glissement est une redistribution des ressources logiques de façon à dégager des portions arithmétiques plus importantes pour permettre davantage d'optimisations arithmétiques.

Là encore, ce n'est pas une technique propre à l'utilisation de l'arithmétique redondante : elle a pour but de *préparer* les circuits en vue d'une optimisation arithmétique.

Dans l'exemple de la Figure 1.8.a, nous pouvons voir que les multiplexeurs à l'intérieur de la chaîne arithmétique ne permettent pas de fusionner les deux additionneurs avec le multiplieur. Le déplacement des deux multiplexeurs se traduit par l'allongement du chemin de données purement arithmétique (cf. Figure 1.8.b), ce qui permet d'effectuer la fusion (cf. Figure 1.8.c).



a. Bloc b. Glissement c. Fusion

FIG. 1.8 – Glissement

1.1.4 Circuits séquentiels

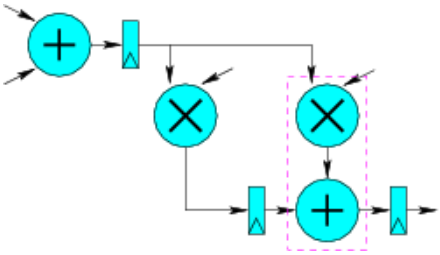
Dans les sections précédentes nous nous sommes restreint aux circuits combinatoires. Yannick Dumonteix a également fait l'étude de l'intérêt de sa méthode en ce qui concerne les circuits séquentiels.

En théorie, les registres peuvent être doublés comme n'importe quel bloc non arithmétique en utilisant la technique de dédoublement présentée dans la Figure 1.4.a. Cependant, leur qualité de *barrière temporelle* fait que cette action peut être source de gain dans certains cas et de contre performance dans d'autres, comme montré dans l'exemple de la Figure 1.9.

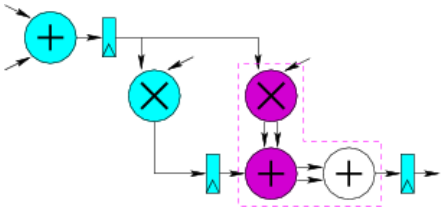
Un bloc arithmétique séquentiel est montré dans la Figure 1.9.a, et l'optimisation de ses enchaînements combinatoires dans la Figure 1.9.b. Deux sortes d'optimisation à travers les registres sont montrées dans les Figures 1.9.c et 1.9.d. Elles se différencient par la place choisie pour le convertisseur final :

- dans la première, le temps de propagation est amélioré car la traversée des registres par des nombres redondants a permis la disparition de conversions $R \rightarrow NR$,
- la seconde amène cependant à une contre performance, en effet dans ce deuxième cas :

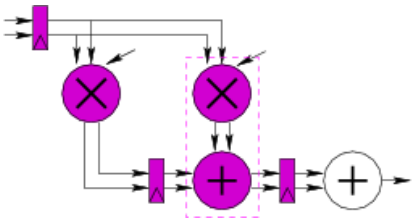
  1. la traversée du registre noté 1 par un nombre redondant a impliqué l'augmentation du délai de l'arbre combinatoire rattaché à la sortie de ce registre



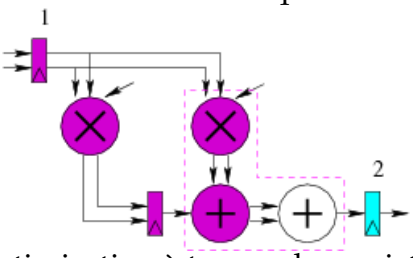
a. Bloc



a. Optimisation des enchaînement combinatoires uniquement



b. Optimisation à travers les registres : Sortie combinatoire



c. Optimisation à travers les registres : Sortie sur registre

FIG. 1.9 – Circuit séquentiel

(les opérateurs de cet arbre voient leur nombre d'entrées redondantes augmenter),

2. en parallèle, la traversée du registre noté 2 par un nombre classique a elle aussi impliqué l'augmentation du délai de l'arbre combinatoire rattaché à l'entrée de ce registre (donc le même que précédemment) de part la nécessité de conserver la conversion $R \rightarrow NR$.

La conclusion donnée est donc que le passage des notations redondantes à travers les registres doit être conditionnée par une analyse des temps de propagation.

1.1.5 Méthodologie générale

Les différentes phases composant la méthodologie d'optimisation de Yannick Dumonteix viennent d'être présentées :

- **passage en redondant partout où c'est possible** : choix des architectures en utilisant autant d'opérateurs redondants / mixtes que possible,
- **glissements** : redistribution des données afin d'avoir des chaînes arithmétiques à optimiser plus longues,
- **regroupements** et **fusions** : utilisation d'arbres de Wallace,
- **retours en notation non redondante** : analyse des temps de propagation pour introduire des retours en notation non redondante de façon à améliorer la chaîne critique.

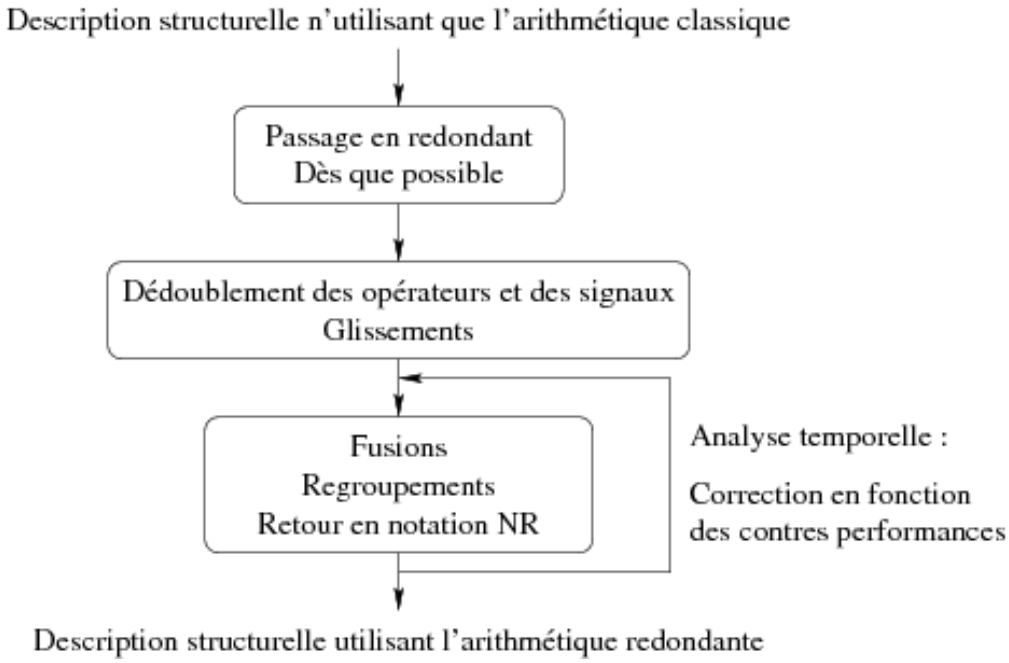


FIG. 1.10 – Etapes de la méthode

Ces phases sont encapsulées dans une méthodologie générale, comme montré dans la Figure 1.10. Après ces différentes étapes, l'architecture des opérateurs ainsi que les interconnexions dans le circuit ont été spécifiées puis éventuellement modifiées de façon à améliorer les performances.

1.2 Environnement de conception

La méthodologie de conception présentée par Yannick Dumonteix va au-delà de la mise en œuvre des différentes phases d'optimisations arithmétiques que nous venons de décrire. Elle aborde également divers autres aspects tels la conception des circuits (langage de description) ou leur portabilité (indépendance entre la description et la technologie cible), toujours dans l'optique de faciliter le but premier des concepteurs : la rapidité de conception.

1.2.1 Langage *mou*

Du point de vue de la description des circuits, le principe de la méthodologie présentée est que, dans la description initiale du circuit à optimiser, les différents opérateurs ne sont pas *complètement spécifiés* : les concepteurs fournissent une description la moins définie possible, i.e. une description structurelle dans laquelle les opérateurs ne sont connus que par leurs interconnexions et leur fonctionnalité ; le terme employé est *description molle*.

La création des opérateurs peut alors se faire avec une forme implicite ($a + b$ par exemple pour l'opération d'addition), plutôt qu'avec une forme explicite comme cela est fait couramment dans les langages de description de vues structurelles (instanciation de l'opérateur avec architecture choisie et interconnexions).

D'un point de vue arithmétique, les signaux doivent contenir l'information de la différenciation entre les différentes représentations arithmétiques des nombres.

1.2.2 Portabilité

Après les étapes d'optimisation, la description obtenue est une description bas niveau la plus précise possible, i.e. une liste de blocs élémentaires avec leur architecture et leurs interconnexions, dans une technologie cible donnée.

Cependant, l'idée de portabilité est elle aussi très importante dans un but de rapidité de conception, par réutilisation de l'existant. Une description intermédiaire prend alors tout son sens : après optimisation, mais avant projection vers une technologie cible. On parle alors de description en portes fonctionnelles ou virtuelles plutôt que réelles.

De cette façon, le concepteur a à disposition une version optimisée de son circuit indépendante de la technologie cible. Il peut ainsi faire suivre les évolutions technologiques au circuit, et également le changer de technologie, cela sans coût additif.

Remarques :

1. Notons que la version optimisée d'un circuit est indépendante de la technologie cible uniquement si l'algorithme d'optimisation utilisé est lui aussi indépendant de cette technologie.

   Comme nous le verrons, un de nos algorithmes dépend de la technologie cible, du fait qu'il prend en compte les temps de propagation des cellules. On obtient alors bien un circuit en portes virtuelles, mais l'optimisation effectuée est optimale pour une technologie en particulier, rien ne garantit qu'elle le soit pour une technologie différente.

   Il est donc nécessaire d'adapter l'algorithme suivant la technologie utilisée, i.e. les temps de propagation des cellules virtuelles sur lesquelles il se base.

2. Il est suggéré que cette description avant projection soit faite dans le langage d'entrée, de façon à pouvoir, entre autres, réutiliser les développements antérieurs.

   Une fois la projection vers une technologie cible effectuée, il est par contre conseillé que la description soit dans un langage connu de façon à être facilement interfaçable avec le reste du flot de conception.

1.3 Problématique

L'objectif de cette thèse est de proposer un outil de conception de circuits permettant l'optimisation des chemins de données arithmétiques grâce à l'utilisation de l'arithmétique redondante tout en facilitant le travail des concepteurs.

Pour cela, nous nous appuierons sur les travaux de Yannick Dumonteix qui a été l'un des premiers à étudier l'impact de l'introduction de l'arithmétique redondante dans le flot de conception *VLSI*. Il a traité pour cela la problématique concernant la mise en place et l'évaluation de l'arithmétique redondante au côté de l'arithmétique classique, définissant pour cela l'arithmétique mixte.

Cette étude a été étoffée par la conception *à la main* de divers circuits arithmétiques utilisant cette arithmétique tels un opérateur de transformée en cosinus discrète [DCM00] et un opérateur de calcul de distance [DBM01]. Ces diverses études ont abouti à de meilleures performances des architectures mises en œuvre.

Suite à ces résultats concluants, Yannick Dumonteix a défini diverses optimisations arithmétiques applicables de façon automatique. Il a alors encapsulé ces optimisations dans une méthodologie d'optimisation complète.

Cependant, appliquer *à la main* la méthodologie qu'il a présentée n'est pas chose aisée pour des concepteurs de circuits n'ayant pas forcément le savoir-faire arithmétique nécessaire. Cela s'avère par ailleurs difficile voire impossible sur de gros circuits. Il en découle naturellement l'idée d'encapsuler cette méthodologie dans un outil qui la mettra en œuvre de manière automatique.

Une telle automatisation a pour but de rendre l'utilisation de l'arithmétique redondante beaucoup plus accessible, permettant aux concepteurs n'ayant pas le savoir-faire requis de pouvoir néanmoins tirer partie de cette arithmétique et d'améliorer de façon significative les performances de leurs circuits.

Notre problématique se focalise donc sur la mise en place de l'automatisation de la méthodologie d'optimisation arithmétique à travers deux étapes fondamentales :

1. L'optimisation arithmétique à proprement parler : différents types d'algorithmes sont utilisables de façon à mettre en pratique automatiquement les différentes optimisations proposées par Yannick Dumonteix. Nous voulons donc définir le meilleur procédé pour encapsuler ces optimisations.
2. L'environnement de conception : nous voulons *faciliter le travail* des concepteurs en caractérisant précisément leurs besoins du point de vue du langage d'entrée, des services à fournir, etc. Ceci sous-entend fournir un langage complet, clair et intuitif. Ce langage doit également permettre de définir un chemin de données par une description simplifiée, i.e. considérer les opérateurs élémentaires (arithmétiques, booléens) comme de simples mnémoniques. Il doit enfin répondre aux besoins de l'arithmétique i.e. permettre la spécification de la représentation des signaux.

1.4 Conclusion

Dans ce chapitre nous avons présenté les différentes optimisations arithmétiques utilisant l'arithmétique redondante, introduites par Yannick Dumonteix dans sa thèse, ainsi que l'environnement de conception associé. Suite à cela, nous avons exposé notre problématique.

Nous allons maintenant présenter plus en détail les caractéristiques de l'arithmétique redondante de façon à démontrer l'intérêt de notre démarche.

Chapitre 2 Arithmétique redondante

Nous présentons ici l'arithmétique redondante et l'arithmétique mixte. Cette présentation est brève, car ne faisant pas parti de cette thèse. Elle a pour but d'expliquer les principes fondamentaux de ces arithmétiques de manière à mieux comprendre l'intérêt de leur utilisation. Le lecteur intéressé par plus de détails pourra se reporter à la thèse de Yannick Dumonteix [Dum01]. Nous montrons ensuite les performances des différents opérateurs arithmétiques, avant de nous intéresser à différentes architectures qui ont été mises en œuvre en utilisant l'arithmétique redondante.

2.1 Généralités

2.1.1 Représentations

L'arithmétique redondante doit son nom au fait qu'il est possible d'affecter plusieurs codages à un même nombre [Avi61]. Elle est constituée de deux représentations.

La représentation Carry-Save

En base 2, les chiffres sont des éléments de 0, 1, 2 et leur codage, sur deux bits de même poids, est tel que $cs_i = cs_i^0 + cs_i^1$. Un nombre Carry-Save est donc codé sur deux termes CS^0 et CS^1 tel que le terme CS en notation classique est : $CS = CS^0 + CS^1$. On parle d'addition non effectuée.

Cette notation permet de représenter des nombres non signés (CS_{NS}) avec des termes sont codés en notation simple de position, et des nombres signés (CS_S) avec des termes en complément à deux :

$$CS_{NS} = \sum_{i=0}^{i=N-1} cs_i^0 \cdot 2^i + \sum_{i=0}^{i=N-1} cs_i^1 \cdot 2^i$$

$$CS_S = -(cs_i^0 + cs_i^1) \cdot 2^{N-1} + \sum_{i=0}^{i=N-2} cs_i^0 \cdot 2^i + \sum_{i=0}^{i=N-2} cs_i^1 \cdot 2^i$$

La représentation Borrow-Save

En base 2, les chiffres sont des éléments de -1, 0, 1 et leur codage, sur deux bits de même poids, est tel que $bs_i = bs_i^+ + bs_i^-$. Un nombre Borrow-Save est donc codé sur deux termes BS^+ et BS^- tel que le terme BS en notation classique est : $BS = BS^+ - BS^-$. On parle de soustraction non effectuée.

Le principe est le même que pour la notation Carry-Save : de la même façon, un nombre Borrow-Save permet de représenter des nombres non signés (BS_{NS}) et des nombres signés (BS_S) :

$$BS_{NS} = \sum_{i=0}^{i=N-1} bs_i^+ \cdot 2^i - \sum_{i=0}^{i=N-1} bs_i^- \cdot 2^i$$

$$BS_S = (-bs_{N-1}^+ \cdot 2^{N-1} + \sum_{i=0}^{i=N-2} bs_i^+ \cdot 2^i) - (-bs_{N-1}^- \cdot 2^{N-1} + \sum_{i=0}^{i=N-2} bs_i^- \cdot 2^i)$$

2.1.2 Arithmétique mixte

Un chemin de données arithmétiques ne se réduit pas à des opérateurs arithmétiques, il contient également des opérateurs non arithmétiques, comme les multiplexeurs ou les opérateurs booléens par exemple. Ces opérateurs doivent conserver les signaux à leur interface en arithmétique classique, c'est pourquoi l'utilisation de l'arithmétique redondante seule dans un circuit s'avère illusoire. De plus l'interface du chemin de données ne doit pas non plus être modifiée.

L'idée qui a été émise est donc d'utiliser conjointement l'arithmétique redondante et l'arithmétique classique. Ainsi, les opérateurs arithmétiques peuvent avoir des entrées/sorties aussi bien en représentation redondante qu'en représentation classique. Cela permet de traiter les signaux pour lesquels la représentation classique est nécessaire.

C'est ce que nous appelons **l'arithmétique mixte**.

Utiliser conjointement ces deux arithmétiques nécessite donc :

1. d'avoir des opérateurs qui acceptent toutes les combinaisons de représentations possibles en interface : $NR\ op\ NR \rightarrow NR$, $NR\ op\ NR \rightarrow R$, $R\ op\ R \rightarrow NR$, $R\ op\ R \rightarrow R$, $NR\ op\ R \rightarrow NR$, $NR\ op\ R \rightarrow R$,
2. d'avoir les convertisseurs nécessaires au passage d'une notation à une autre : en pratique, cela s'avère assez simple étant donné que passer de la notation Carry-Save à une notation classique se fait en additionnant les deux termes du nombre Carry-Save avec un additionneur classique, et passer d'une notation Borrow-Save à une notation classique se fait de même avec un opérateur de soustraction.

2.2 Architectures

2.2.1 Addition

Arithmétique classique

L'addition est l'opération la plus usitée dans la conception de chemins de données arithmétiques. De nombreuses architectures ont donc été étudiées en arithmétique classique de façon à la rendre la plus rapide possible [Mul89], nous n'en présentons ici que deux, ayant des performances surface/temps opposées :

Additionneur à propagation de retenue séquentielle L'algorithme employé est l'algorithme intuitif et purement séquentiel d'une addition *à la main*, comme montré dans la Figure 2.1. On parlera de *Carry Ripple Adder*. La complexité de la surface et du délai de cet additionneur sont en $O(N)$ (N étant la dynamique des nombres à additionner).

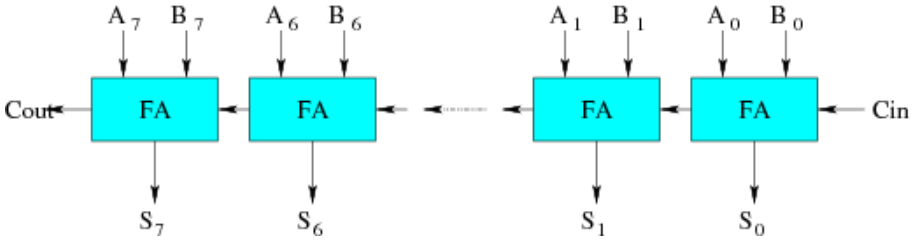


FIG. 2.1 – Additionneur séquentiel

Le délai important de cette architecture est dû à la propagation de la retenue, qui est réalisée de façon purement séquentielle. Les autres architectures d'additionneurs ont donc pour but de réduire ce temps de propagation.

Additionneur de Sklansky Cet additionneur, aussi appelé CLA pour *Carry Lookahead Adder*, fait partie des additionneurs à retenue anticipée. Le principe est d'anticiper la propagation de la retenue de façon à s'affranchir de l'attente du calcul de l'addition du bit précédent. L'architecture correspondante est montrée dans la Figure 2.2. C'est une structure ayant une complexité de $O(Nlog_2(N))$ en surface et $O(log_2(N))$ en délai.

La chaine critique a donc pu être améliorée par rapport au *Ripple*, mais au détriment de la surface.

Soustraction Il n'y a pas d'architecture spécifique pour la soustraction en arithmétique classique. Effectuer une soustraction se fait en additionnant à un nombre le complément à deux du nombre à soustraire : $A - B = A + not(B) + 1$. En pratique, un additionneur est utilisé, avec une série d'inverseurs pour obtenir le complément à un du nombre à soustraire et la retenue entrante de l'additionneur positionnée à 1.

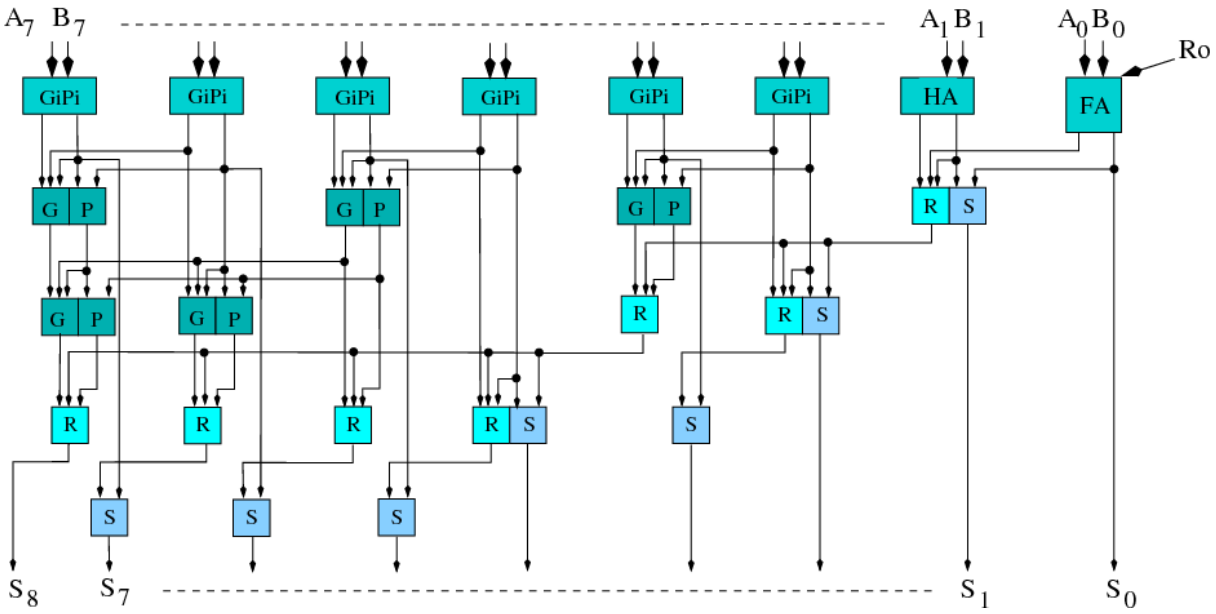


FIG. 2.2 – Additionneur de Sklansky

Arithmétique redondante

L'additionneur redondant effectue l'addition de deux nombres redondants et fournit un résultat en redondant.

Addition Carry-Save La Figure 2.3 représente l'architecture de l'additionneur $CS + CS \rightarrow CS$. Cet additionneur a une complexité de $O(N)$ en surface et $O(1)$ en délai.

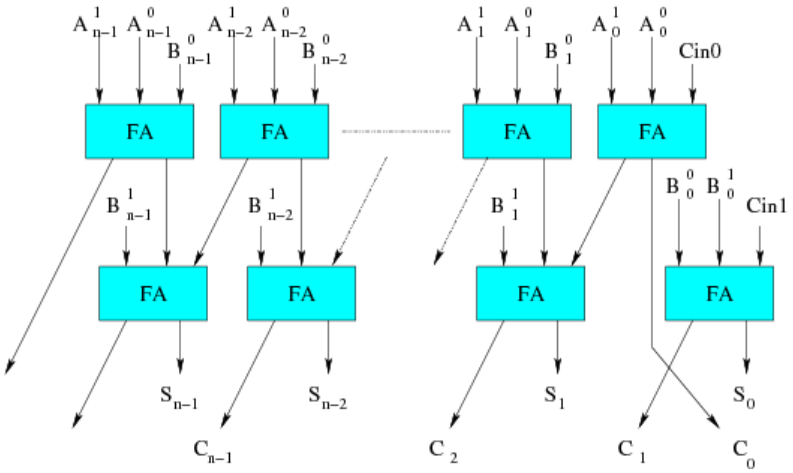


FIG. 2.3 – Additionneur redondant

Plus précisément, sa surface est de l'ordre de grandeur de deux fois celle du *Ripple*, l'additionneur "le plus petit" en arithmétique classique, et il est en temps constant, quelle que soit la dynamique de ses entrées (temps correspondant à la traversée de deux étages d'additionneurs complets - noté *FA* pour *full-adder*).

En effet, le fait qu'un nombre Carry-Save soit la somme de deux nombres entraîne que l'on propage en parallèle la somme et la retenue. On s'affranchit donc de la propagation de retenue dans le calcul.

Addition Borrow-Save L'utilisation de la notation Borrow-Save permet de réaliser les soustractions avec des architectures redondantes. Un additionneur redondant effectue soit l'addition de deux nombres Borrow-Save : $BS_0 + BS_1 = BS_0^+ - BS_0^- + BS_1^+ - BS_1^-$, soit l'addition d'un nombre Borrow-Save et d'un nombre Carry-Save : $BS + CS = BS^+ - BS^- + CS^0 + CS^1$.

Deux types d'architectures existent pour ces différentes opérations : soit avec une sortie en représentation Carry-Save, soit avec une sortie en représentation Borrow-Save :

- pour obtenir une sortie Carry-Save, le principe est le même qu'en arithmétique classique : une série d'inverseurs pour chaque nombre Borrow-Save en entrée, et une ou deux des retenues d'entrée positionnée à '1' (cf. exemple de l'addition de deux nombres Borrow-Save dans la Figure 2.4),

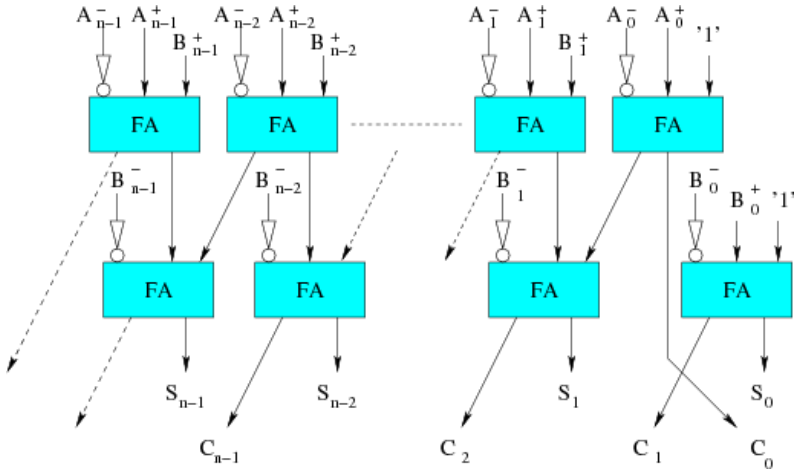
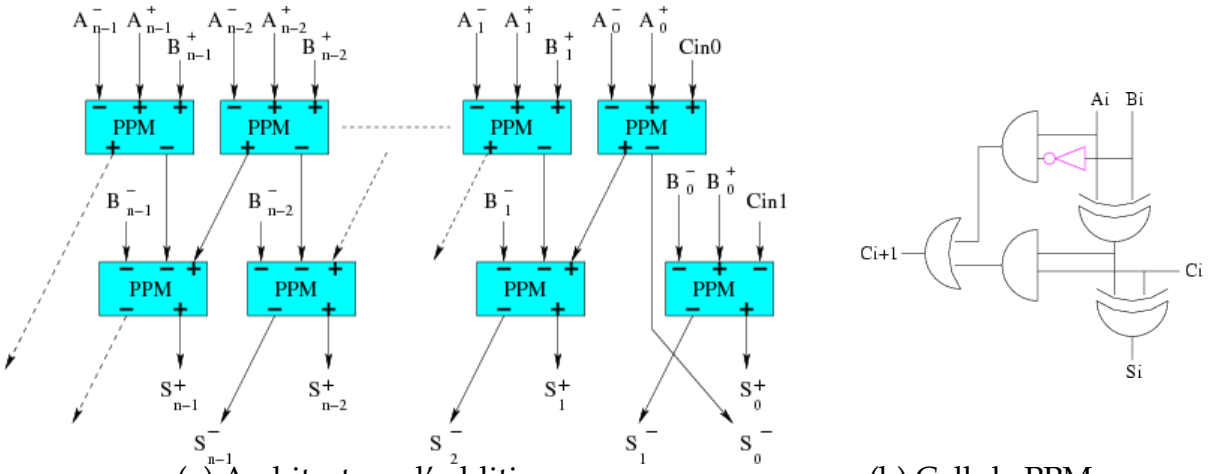


FIG. 2.4 – Additionneur redondant Borrow-Save avec sortie Carry-Save

- pour obtenir une sortie Borrow-Save, l'architecture est une nouvelle fois identique à celle pour le Carry-Save, à ceci près qu'il faut modifier la cellule de base qu'est le *FA* en une cellule dite *plus-plus-minus* (notée *PPM*, cf. Figure 2.5.b) [JDK91] (cf. exemple de l'addition de deux nombres Borrow-Save dans la Figure 2.5.a).



(a) Architecture d'additionneur (b) Cellule PPM

FIG. 2.5 – Additionneur redondant Borrow-Save avec sortie Borrow-Save

Arithmétique mixte

L'additionneur mixte effectue l'addition d'un nombre redondant et d'un nombre non redondant et fournit un résultat en redondant.

Addition Carry-Save La Figure 2.6 représente l'architecture de l'additionneur mixte $CS + NR \rightarrow CS$. Cet additionneur a une complexité de $O(N)$ en surface et $O(1)$ en délai.

Il est naturellement lui aussi en temps constant (temps correspondant à la traversée d'un étage de *FA*) et sa surface est identique à celle d'un *Ripple*.

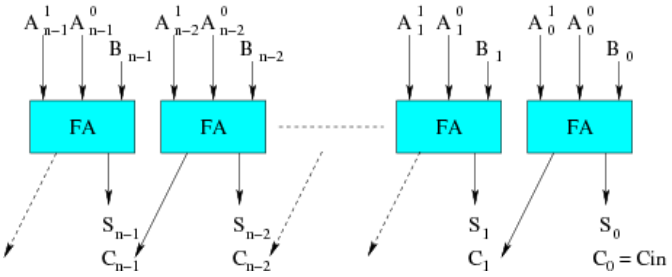


FIG. 2.6 – Additionneur mixte

Remarque :

Là où un additionneur redondant réduit le nombre d'opérandes de 4 à 2, un additionneur mixte le réduit de 3 à 2. Un additionneur mixte représente ainsi la moitié du calcul d'un additionneur redondant. En d'autres termes, un additionneur redondant peut être considéré comme la concaténation de deux additionneurs mixtes.

Addition Borrow-Save En utilisant la représentation Borrow-Save, un additionneur mixte effectue l'addition entre un nombre Borrow-Save et un nombre en complément à 2 : $BS + NR = BS^+ - BS^- + NR$.

Là aussi, deux architectures différentes existent en fonction de la représentation de la sortie de l'additionneur. Le principe est identique à celui de l'additionneur redondant, comme montré dans les Figures 2.7 et 2.8.

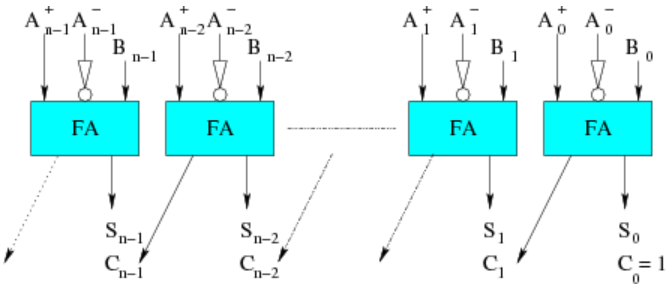


FIG. 2.7 – Additionneur mixte Borrow-Save avec sortie Carry-Save

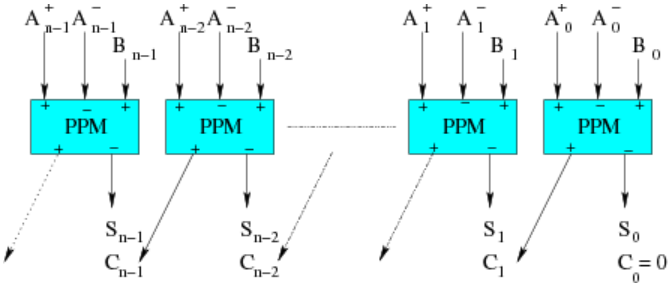


FIG. 2.8 – Additionneur mixte Borrow-Save avec sortie Borrow-Save

Synthèse

Les architectures d'additionneurs redondants et mixtes allient une petite taille (comparable à celle du plus petit additionneur classique) et un temps de propagation constant. On en déduit l'atout majeur de l'arithmétique redondante : **l'addition de deux nombres se fait en temps constant, quelle que soit la taille des opérandes à sommer**. L'addition étant un opérateur fondamental, le gain potentiel est important.

2.2.2 Multiplication

Dans ce qui suit, nous présentons les différentes étapes nécessaires pour effectuer une multiplication. Ces étapes nous servent à montrer l'impact de l'utilisation de l'arithmétique redondante sur la multiplication. Nous ne présentons pas en détail l'architecture des différentes étapes, les lecteurs intéressés peuvent se référer à [DM00, Dum01, AFT02].

Arithmétique classique

Tout comme pour l'addition, un algorithme intuitif de multiplication par additions / décalages conduit à une architecture ayant un mauvais délai (complexité en $O(N)$).

L'idée émise de façon à minimiser ce délai consiste à augmenter le nombre de 0 du nombre multiplicateur en le recodant dans une base plus grande, c'est le recodage de *Booth modifié*. Ce recodage permet de diviser globalement par deux le nombre de produits partiels.

Dans la Figure 2.9, nous pouvons voir les différentes étapes nécessaires à la réalisation d'une telle multiplication : le recodage du multiplicateur (délai en $O(1)$), le calcul des produits partiels (délai en $O(1)$), puis la somme des produits partiels, elle même décomposée en un arbre de Wallace (délai en $O(log_{3/2}(N/2))$) et un additionneur final de *Sklansky* (délai en $O(log_2(N))$). Cet algorithme permet donc d'obtenir des multiplieurs arborescents ayant une complexité en délai en $O(log_{3/2}(N/2))$.

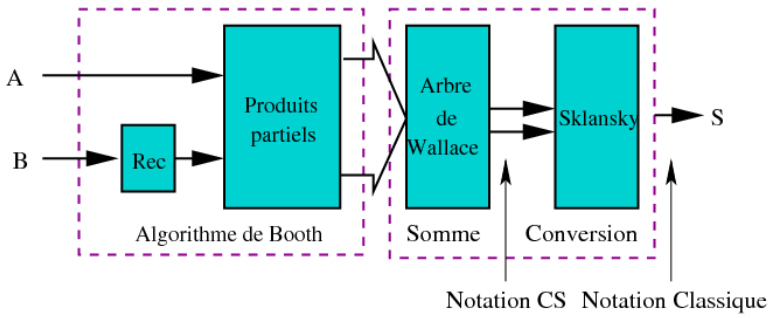


FIG. 2.9 – Struture d'un multiplieur classique

Remarque :

Nous avons déjà évoqué le fait que l'arithmétique redondante a tout d'abord été utilisée dans l'architecture interne des multiplieurs. Comme on peut le voir, un arbre de Wallace est utilisé pour effectuer la somme des produits partiels. Or, son rôle est de diminuer le nombre d'opérandes à sommer de n à 2 : il fournit un résultat en représentation Carry-Save. C'est pourquoi dans l'architecture classique d'un multiplieur, il est suivi d'un additionneur, qui a pour rôle de convertir le nombre Carry-Save obtenu en un nombre en représentation classique. Cette particularité de l'architecture du multiplieur montre d'ores et déjà l'intérêt de permettre l'utilisation de la notation Carry-Save : elle permet la suppression du convertisseur final.

Arithmétique redondante

Multiplication Carry-Save La multiplication de deux nombres *CS* consiste en l'opération suivante : $CS_0 * CS_1 = (CS_0^0 + CS_0^1) * (CS_1^0 + CS_1^1)$. Le résultat est fourni en représentation Carry-Save.

La Figure 2.10 représente les modifications à apporter à l'architecture classique de façon à utiliser les notations redondantes. L'architecture présentée est de type *direct*, dans laquelle il n'est pas effectué un recodage de Booth. Une étape de recodage est cependant nécessaire pour prendre en compte deux entrées en format Carry-Save. En effet, le calcul des produits partiels ne peut gérer les cas où un chiffre égal à 2 est présent à la fois dans les deux opérandes. L'étape de recodage sert donc à modifier les propriétés des entrées de façon à éviter ce cas.

L'étape de produits partiels est également différente de la version classique étant donné qu'elle prend en compte deux nombres redondants en entrée i.e. l'équivalent de quatre nombres classiques (au lieu de deux pour l'architecture classique).

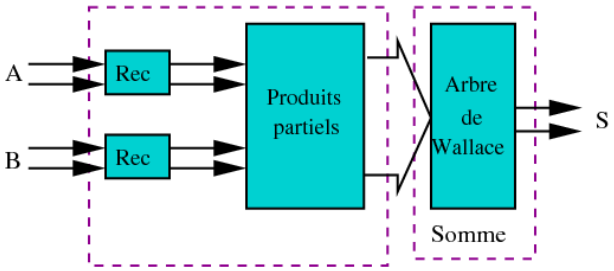


FIG. 2.10 – Struture d'un multiplieur redondant

Multiplication Borrow-Save Un multiplieur redondant effectue soit la multiplication de deux nombres *BS* : $BS_0 * BS_1 = (BS_0^+ - BS_0^-) * (BS_1^+ - BS_1^-)$, soit la multiplication d'un nombre *BS* et d'un nombre *CS* : $BS * CS = (BS^+ - BS^-) * (CS^0 + CS^1)$.

D'un point de vue architecture, seule l'étape de recodage est modifiée, en fonction de la représentation de chaque entrée. En effet, on profite du fait qu'un recodage est de toute façon nécessaire pour effectuer les produits partiels pour recoder la ou les entrées Borrow-Save en Carry-Save. De ce fait, la construction des produits partiels (puis la sommation) reste la même. La sortie est donc en représentation Carry-Save.

Arithmétique mixte

Multiplication Carry-Save La multiplication entre un nombre *CS* et un nombre *NR* consiste en l'opération suivante : $CS * NR = (CS^0 + CS^1) * NR$.

Les modifications à apporter à l'architecture classique se situent au niveau du calcul des produits partiels, de façon à prendre en compte un nombre Carry-Save et un nombre classique i.e. l'équivalent de trois nombres, comme montré dans la Figure 2.11. Comme on peut le voir, cette architecture ne nécessite pas d'étape de recodage. En effet, l'étape de recodage servait à s'assurer, dans la multiplication redondante, qu'il n'y aurait jamais deux nombres égaux à 2 en même temps lors de la phase de produits partiels, cas qui n'est de toute façon pas possible dans le cas d'un nombre redondant et d'un nombre classique.

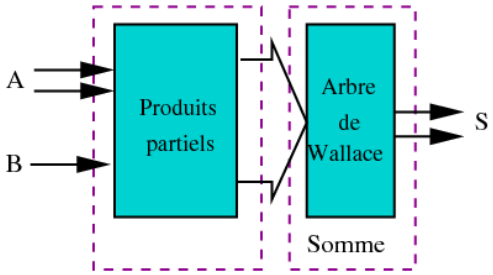


FIG. 2.11 – Struture d'un multiplieur mixte

Multiplication Borrow-Save La multiplication entre un nombre *BS* et un nombre *NR* consiste en l'opération suivante : $BS * NR = (BS^+ - BS^-) * NR$.

Les modifications à apporter à l'architecture classique se situent au même niveau que pour la multiplication Carry-Save.

Synthèse

Comme nous l'avons déjà souligné, le fait qu'un multiplieur fournisse un résultat en représentation Carry-Save se traduit par la suppression de l'additionneur final de conversion dans son architecture. Cependant, plus un multiplieur a d'entrées redondantes, plus la phase de calcul des produits partiels est coûteuse, principalement quand il y a deux entrées redondantes, car un recodage des entrées redondantes est alors nécessaire. L'idéal est donc un multiplieur dont les entrées sont en représentation classique et la sortie en représentation redondante.

A un niveau unitaire, l'intérêt des multiplieurs redondant et mixte est moins évident que pour les additionneurs. Il se fera ressentir principalement avec l'enchaînement des opérateurs.

2.3 Performances

Dans cette partie, nous présentons un récapitulatif des résultats que nous avons obtenus pour les additionneurs et multiplieurs mixtes et redondants, comparées à ceux de ces mêmes opérateurs en arithmétique classique, en terme de surface et de délai. Cette évaluation a pour objectif de montrer l'intérêt de telles architectures.

Les opérateurs évalués ont été décrits avec le langage **Stratus** et font partie de la bibliothèque **ArithLib**, tous deux développés durant cette thèse et qui seront étudiés au Chapitre 5.

L'évaluation se fait avec emploi de la bibliothèque de cellules précaractérisées de la chaîne **Alliance** [GP92] (en 0.35μm) et les outils de placement/routage de la chaîne de *CAO* Cadence : **Encounter**.

2.3.1 Addition

Les Figures 2.12 et 2.13 présentent les performances des additionneurs classiques, redondants et mixtes. Les additionneurs utilisés sont les additionneurs mixtes et redondants (avec les différentes représentations possibles *CS* et *BS* en entrées / sorties), ainsi que deux additionneurs classiques : le *Sklansky*, qui est la référence en matière de rapidité pour les additionneurs classiques, et le *Ripple*, qui est le plus petit additionneur classique. Pour chaque additionneur, plusieurs évaluations sont faites en faisant varier la dynamique des entrées (de 4 à 128 bits).

Les résultats présentés montrent que les performances obtenues pour les opérateurs d'addition sont en accord avec ce que les architectures présageaient, les additionneurs mixtes et redondants sont :

- **petits** :
  - l'ordre de grandeur est celui du *Ripple* pour les additionneurs mixtes avec sortie Carry-Save, et de deux fois cette surface pour les additionneurs redondants avec sortie Carry-Save,
  - les additionneurs mixtes et redondants avec sorties Borrow-Save sont un peu plus grands que leur équivalent avec sortie Carry-Save, mais restent plus petits que le *Sklansky*,

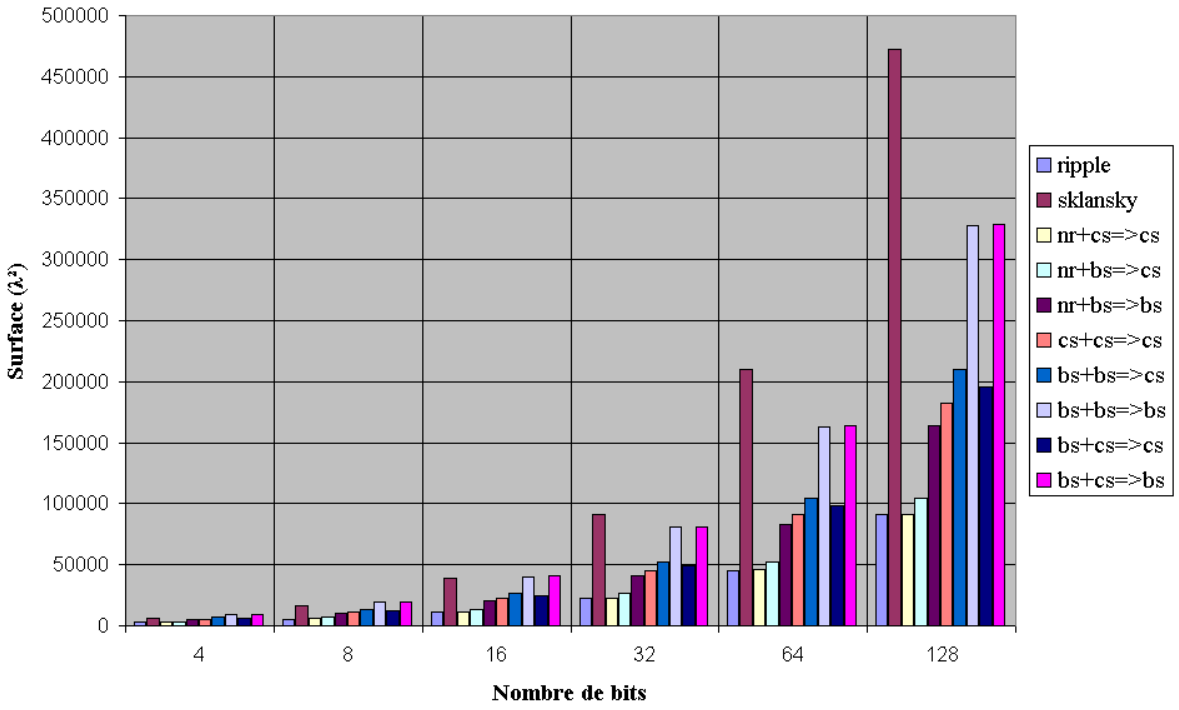


FIG. 2.12 – Comparaison en surface des additionneurs

- **rapides et en temps constant** :
  - les additionneurs mixtes sont systématiquement plus rapides que le *Sklansky*,
  - les additionneurs redondants le sont également, quand la dynamique des entrées est supérieure ou égale à 8 bits,
  - le temps de propagation des additionneurs redondants est environ deux fois celui des additionneurs mixtes.

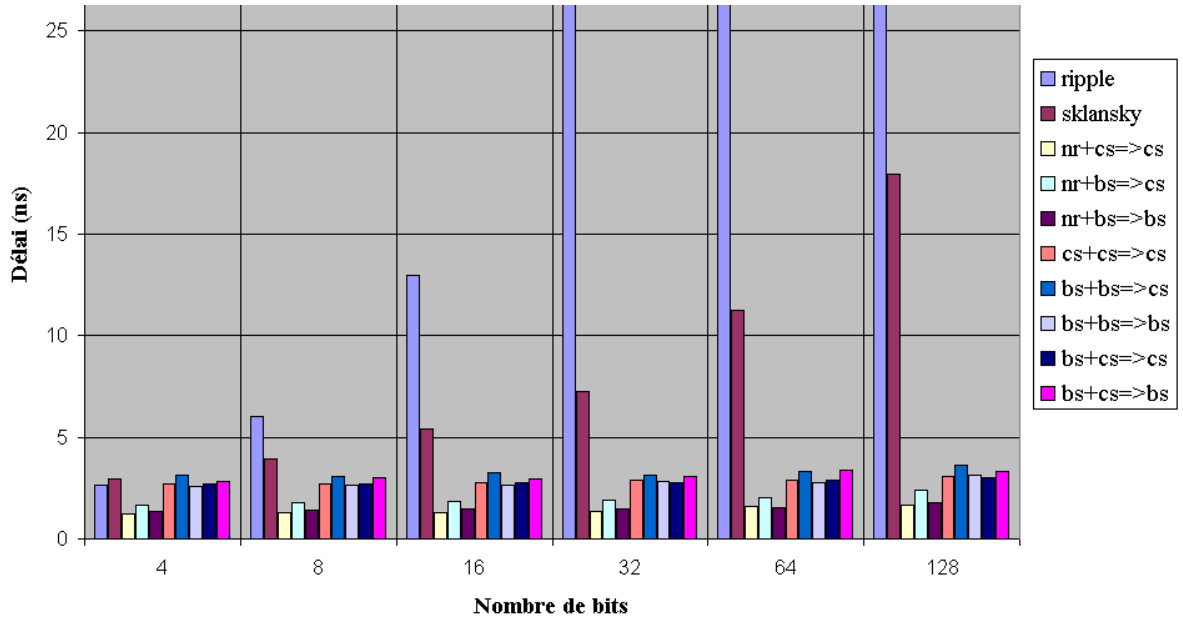


FIG. 2.13 – Comparaison en délai des additionneurs

Remarques :

1. Les données en délai du *Ripple* sont tronquées à partir de 32 bits dans le graphique, étant donné les mauvaises performances de cet additionneur, sur 128 bits, son délai est par exemple de 304 ns, valeur qui aurait rendue le graphique peu lisible.
2. Les performances de l'additionneur redondant sont d'autant plus importantes que, là où le *Sklansky* diminue le nombre d'opérandes de 2 à 1, l'additionneur redondant le diminue de 4 à 2, soit potentiellement deux fois le même travail.
3. La différence de taille entre les additionneurs avec sortie Carry-Save et ceux avec sortie Borrow-Save est principalement dûe au fait que la cellule dite *PPM* (cf. Figure 2.5.b) n'existe pas dans la bibliothèque de cellules utilisée, contrairement au *FA*, les différentes portes la composant ont donc été utilisées, amenant à une cellule de base moins optimisée. Il est envisageable de créer cette cellule de base de façon à obtenir de meilleurs résultats.

2.3.2 Multiplication

Les Figures 2.14 et 2.15 montrent les mêmes expérimentations que précédemment, mais avec la multiplication. Les multiplieurs comparés sont : le classique, le mixte et le redondant (différentes représentations en entrées / sorties pour les deux derniers), le classique étant avec utilisation de l'algorithme de Booth, et les deux autres avec utilisation de l'algorithme dit direct [Dum01]. Comme précédemment, plusieurs évaluations sont faites en faisant varier la dynamique des entrées (de 4 à 24 bits).

Les résultats présentés sont là aussi en accord avec ce que nous avions déduit de l'architecture des multiplieurs. Les multiplieurs redondants et mixtes sont :

- **gros** :
  - les multiplieurs mixtes sont légèrement plus petits que le multiplieur classique,
  - les multiplieurs redondants sont eux un peu plus gros, à cause des phases de recodage des entrées redondantes et de calcul des produits partiels,

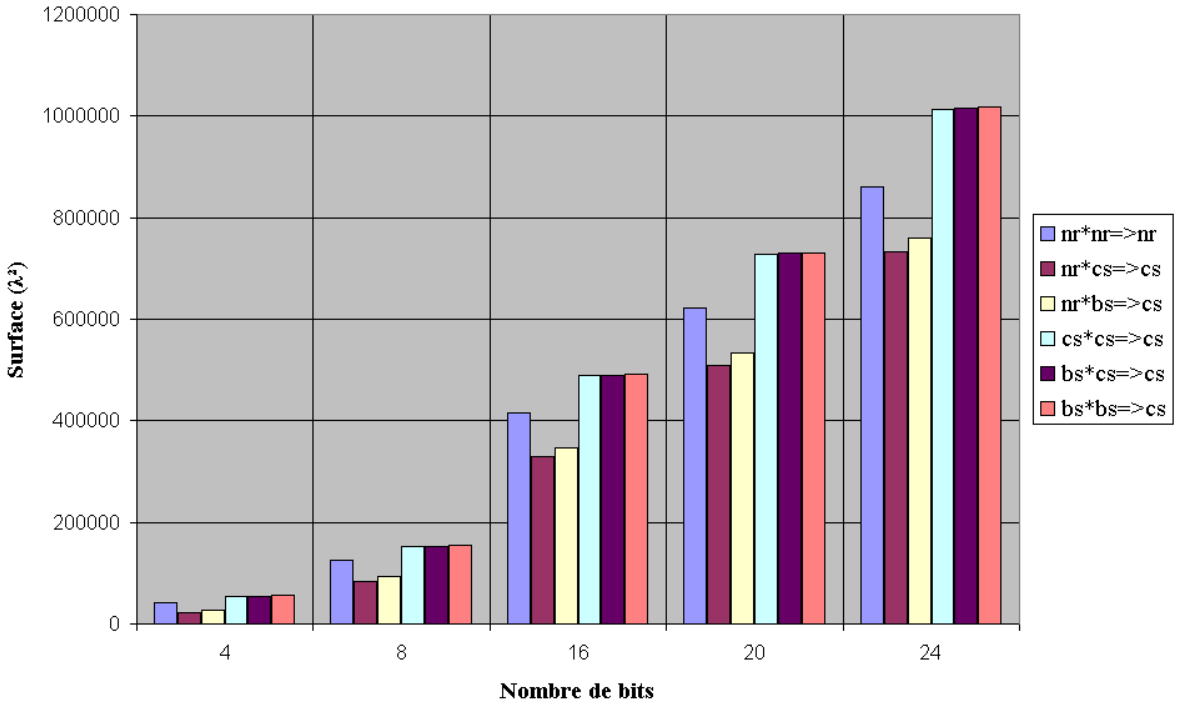


FIG. 2.14 – Comparaison en surface des multiplieurs

- **rapides** :
  - ils sont tous plus rapides que le multiplieur classique.

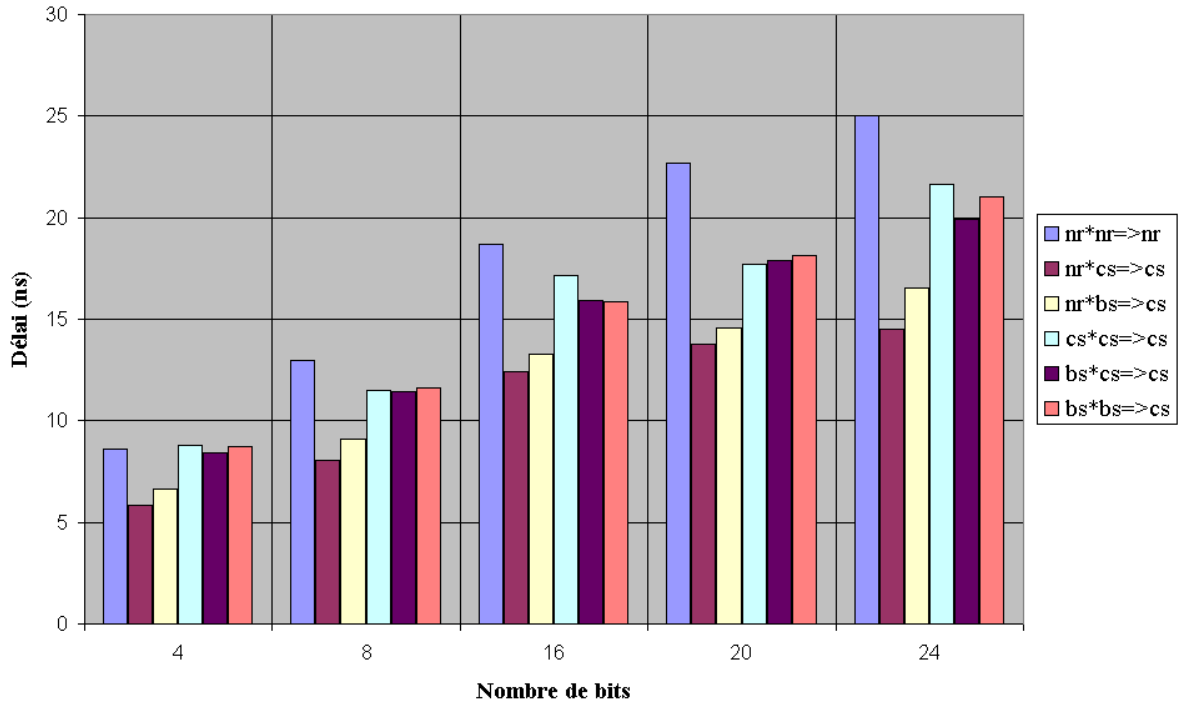


FIG. 2.15 – Comparaison en délai des multiplieurs

2.4 Applications

Au-delà des bonnes performances intrinsèques des opérateurs redondants et mixtes, c'est le chaînage d'opérations arithmétiques qui permet d'exploiter au mieux ces architectures de par la suppression des conversions. La meilleure façon de montrer l'intérêt de l'utilisation de l'arithmétique redondante dans la conception de chemins de données arithmétiques est donc d'utiliser *à la main* cette arithmétique dans la réalisation de différentes architectures [VBG+94, LCSJJ03, CMRS04]. Nous détaillons ici plusieurs résultats extraits de la littérature et obtenus de cette façon.

2.4.1 PGCD

Plusieurs algorithmes existent de façon à trouver le *PGCD* et le *PGCD* étendu[1]. Deux grandes approches existent : algorithmes utilisant le bit de poids faible des nombres et ceux utilisant le bit de poids fort. Dans la première approche, des architectures classiques sont utilisées, tandis que des architectures redondantes sont utilisées dans la seconde. Ces deux approches sont comparées dans [Guy91], elles-mêmes divisées en deux sous approches : additions / soustractions effectuées en série (comparaison pour un algorithme) ou en parallèle (comparaison pour quatre algorithmes).

[1]Calcul des deux entiers vérifiant l'égalité de Bézout i.e. s et t tels que $a * s + b * t = pgcd(a, b)$

	bits / cycle			transistors / bit		
	Classique	Redondant	%	Classique	Redondant	%
Série (1 algo.)	0.6	1	+67%	75	240	+220%
Parallèle (4 algo.)	0.6 à 0.95	0.95 à 1.05	jusqu'à +75%	140 à 220	140 à 150	jusqu'à -36%

TAB. 2.1 – PGCD : Résultats

	bits / cycle			transistors / bit		
	Classique	Redondant	%	Classique	Redondant	%
Série (1 algo.)	-	0.48	-	-	320	-
Parallèle (4 algo.)	0.22 à 0.32	0.46 à 0.50	jusqu'à +127%	265 à 305	155 à 165	jusqu'à -49%

TAB. 2.2 – PGCD étendu : Résultats

Les résultats présentés dans les Tableaux 2.1 et 2.2 proviennent de simulations et font une estimation du délai en considérant le ratio bits par cycle, et de la surface avec le ratio nombre de transistors par bit.

Ils montrent que les architectures utilisant l'arithmétique redondante ont de meilleurs ratios bits / cycle (jusqu'à +127%) couplés à de meilleurs ratios transistors / bit (jusqu'à -49%) sauf dans le cas de l'algorithme série du *PGCD* (+220%).

2.4.2 DCT

Un macro-générateur effectuant la *DCT* 2 dimensions a été réalisé dans [DCM00] (architecture décrite dans le Chapitre 6), avec comme paramètres les tailles des données et des coefficients ainsi que le nombre d'étages de pipeline. Deux implantations de ce macro-générateur ont été réalisées : avec ou sans pipeline, et avec les entrées sur 8 bits, les coefficients sur 12 bits et les sorties sur 16 bits.

Les performances de ces implantations sont présentées après placement / routage et comparées à celles d'une implantation classique.

	Surface (mm^2)			Fréquence (MHz)		
	Classique	Redondant	%	Classique	Redondant	%
sans pipeline	0.37	0.36	-2.7%	36	45	+25%
3 étages de pipeline	0.38	0.41	+7.9%	75	116	+54.7%

TAB. 2.3 – DCT : Résultats

Les résultats présentés dans le Tableau 2.3 montrent qu'il n'y a pas d'amélioration significative d'un point de vue de la surface, mais une amélioration importante du

point de vue du délai, +25% pour la fréquence pour la version sans pipeline et +54.7% pour la version avec pipeline.

2.4.3 DCU

L'implantation d'une *DCU* a été réalisée avec un opérateur de carré redondant (amélioration en temps de 10 à 20% par rapport à l'opérateur classique) dans [DBM01] (architecture décrite dans le Chapitre 6). Plusieurs implantations ont été effectuées, en fonction de la dynamique des entrées : de 8 à 32 bits.

Les performances sont ici aussi présentées après placement / routage et comparées à celles d'une implantation classique.

Nombre de bits	Surface (μm^2)			Délai (ns)		
	Classique	Redondant	%	Classique	Redondant	%
8	6259	7166	+14.5%	10.7	9.9	-7.5%
16	19293	22491	+16.6%	14.3	13.0	-9.1%
24	44222	51480	+16.4%	16.9	14.8	-12.4%
32	86350	99849	+15.6%	20.4	17.5	-14.2%

TAB. 2.4 – DCU : Résultats

Les résultats présentés dans le Tableau 2.4 se résument à une amélioration du délai et un surcoût de la surface : par exemple sur 32 bits, -14.2% en délai et +15.6% en surface. On peut en déduire que, pour ce circuit, les produits surface.délai ne sont pas améliorés en utilisant l'arithmétique redondante. Cela ne diminue cependant pas l'intérêt de l'usage de l'arithmétique redondante si l'on considère que le critère primordial à optimiser est le délai.

2.5 Conclusion

Dans ce chapitre nous avons présenté les caractéristiques de l'arithmétique redondante. Nous avons montré que l'usage de cette arithmétique seule dans la conception de circuits s'avère illusoire, nous avons donc présenté une arithmétique étant la combinaison des arithmétiques redondante et classique : l'arithmétique mixte.

Nous avons présenté les architectures des différents opérateurs d'addition et de multiplications utilisant cette arithmétique, ainsi que leurs performances. Le point crucial est que l'addition en arithmétiques redondante/mixte est exempte de toute propagation de retenue, elle se fait donc en temps constant quelque soit la dynamique des opérandes à sommer.

Suite aux bonnes performances intrinsèques des opérateurs présentés, nous avons fait un état de l'art des différents blocs arithmétiques réalisés à la main avec cette arithmétique. Les résultats présentés sont très encourageants et montrent l'intérêt d'outils d'optimisation automatiques utilisant l'arithmétique mixte.

Chapitre 3 Etat de l'art

Au-delà des travaux de Yannick Dumonteix qui a énoncé des règles d'optimisation automatisables, diverses études ont été menées depuis quelques années en ce qui concerne les algorithmes d'optimisation à proprement parlé.

Nous nous intéressons dans ce chapitre à plusieurs catégories d'outils, certains faisant parti de la synthèse de haut niveau[1], d'autres de la synthèse *RTL*[2]. La synthèse de haut niveau s'apparente moins à notre vision de l'optimisation, mais les outils présentés peuvent fournir des postulats intéressants applicables à la synthèse *RTL*.

3.1 Utilisation d'additionneurs mixtes dans la synthèse RTL

Différents travaux sont menés par Taewhan Kim sur la synthèse *RTL* de circuits utilisant l'arithmétique redondante depuis la fin des années 90. Ces travaux sont essentiellement basés sur des algorithmes utilisant des additionneurs mixtes $CS + NR \rightarrow CS$ (notés *CSA*). En effet, ils partent du principe que ce type d'additionneurs est le plus usité car, alliant une chaîne critique très courte et une petite surface.

Plusieurs approches ont été présentées, certaines instanciant autant de *CSA* que possibles, d'autres cherchant l'allocation optimale, d'autres encore travaillant au niveau bit de façon à améliorer encore les performances. Nous présentons ici ces travaux, par ordre chronologique.

3.1.1 Approche *tout en redondant*

Dans [KJT98a] et [KJT98b], l'algorithme utilisé est décrit comme suit : *A partir d'un graphe non cyclique d'opérations arithmétiques, le but est de transformer les opérations en utilisant le plus de* CSA *possibles tout en concervant la fonctionnalité du circuit*.

Il s'apparente donc à la phase d'optimisation *redondant dès que possible* proposée par Yannick Dumonteix, à ceci près qu'il n'utilise que des additionneurs mixtes.

[1] Passage d'une description algorithmique à une description *RTL*
[2] Passage d'une description *RTL* à une description structurelle

Cet algorithme parcourt le circuit de ses sorties jusqu'à ses entrées en détectant des arbres d'additions à optimiser. Il est décomposable en trois phases :

1. identification d'un arbre d'opérations à transformer,
2. translation de cet arbre en expression d'additions,
3. conversion de l'addition trouvée en un arbre de *CSA*.

Il est appliqué de façon itérative jusqu'à ce qu'il n'y ait plus d'arbre à transformer.

De façon à ne pas limiter cet algorithme à des circuits ne contenant que des additions, des opérations comme la soustraction et la multiplication sont transformées de façon à pouvoir être optimisées.

Pour la soustraction, l'addition du premier terme et du complément à 2 du second est effectuée ($x - y = x + not(y) + 1$).

Pour la multiplication, deux solutions sont présentées : effectuer une somme de produits ou une multiplication partielle suivie d'une addition. La première solution consiste en une décomposition totale de la multiplication en opérations décalage + addition, ce qui est bénéfique dans ce contexte, mais augmente de façon très importante le nombre d'additionneurs lorsque les données sont de grande taille. La seconde n'a pas ce désavantage mais est moins flexible (optimisation de l'additionneur final uniquement).

Les résultats présentés dans [KJT98b] sont répertoriés dans le Tableau 3.1.

Ils consistent en une comparaison de la chaîne critique et de la surface (exprimée en $unit$ i.e. équivalent nombre de cellules de base, le *FA*) de circuits réalisés avec ou sans l'utilisation de l'outil d'optimisation arithmétique. Ces valeurs sont obtenues avec Design Compiler [Syn], pour une technologie 0.35μm.

	Surface ($unit$)		Temps (ns)	
	Classique	Redondant	Classique	Redondant
Somme de 4 opérandes (16 bits)	2 952 ref	2 130 -28%	6.69 ref	5.48 -14%
Somme de 8 opérandes (16 bits)	6 786 ref	4 327 -36%	9.48 ref	6.24 -34%
Somme de 16 opérandes (16 bits)	14 740 ref	8 590 -42%	15.97 ref	7.27 -54%
A*B + C*D + E*F (8 bits)	11 444 ref	10 210 -11%	7.32 ref	6.25 -15%
A+B-C-D-E-F (8 bits)	5 328 ref	3 697 -31%	7.94 ref	6.03 -24%

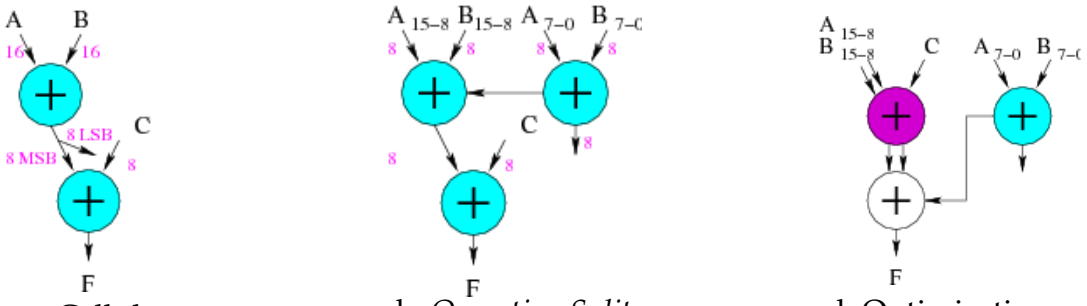
TAB. 3.1 – Résultats Kim : Approche tout en redondant

Ils montrent une diminution significative de la chaîne longue ainsi que de la surface pour les circuits obtenus après optimisation arithmétique, par rapport à ceux obtenus avec une synthèse *RTL* classique.

Par ailleurs, deux concepts sont présentés : *operation duplication* et *operation split*.

Le premier est une solution au problème de *limitation de type donnée* vu dans la Partie 1.1.1. Il s'apparente au concept de *dédoublement de la donnée* que nous avons présenté dans les Figures 1.3.b et 1.4.a pour résoudre ce problème. De façon à être appliqué à l'algorithme présenté, l'idée est que chaque opérateur avec une sortie connectée à plusieurs opérateurs est la racine d'un nouvel arbre d'additionneurs. Puis, pour chaque arbre d'additionneurs ainsi obtenu, deux sorties sont fournies : une en représentation CS (à connecter aux opérateurs arithmétiques), l'autre en représentation NR avec un convertisseur $CS \rightarrow NR$ (à connecter aux opérateurs non arithmétiques).

Le second est une solution au problème de *troncature de données* i.e. troncature des bits de poids faible d'un signal, comme montré dans la Figure 3.1. En effet, une modification du nombre tronqué en sa version redondante est susceptible d'engendrer une modification du comportement du circuit. L'idée présentée est donc d'effectuer deux additions avec propagation de retenue entre les deux, de façon à pouvoir optimiser l'un des additionneurs.



a. Cellule b. *Operation Split* d. Optimisation

FIG. 3.1 – Solution au problème de troncature des bits de poids faible

Des résultats sont également présentés pour montrer l'intérêt de ces deux méthodes. Ils peuvent être résumés comme suit :

- *limitation de type donnée* : l'application du concept de *operation duplication* sur des circuits de 4 ou 5 opérations dont 2 avec la sortie connectée à plusieurs opérateurs (appelé *sortie multiple*) permet d'améliorer le temps de 2 à 8% par rapport à l'optimisation sans appliquer ce concept, au détriment d'une augmentation de 9 à 26% pour la surface,
- *troncature de données* : l'application du concept de *operation split* sur des circuits de 4 ou 5 opérations dont 2 avec troncature de données permet d'améliorer le temps de 10 à 32% et la surface de 9 à 18% par rapport à l'optimisation sans appliquer ce concept.

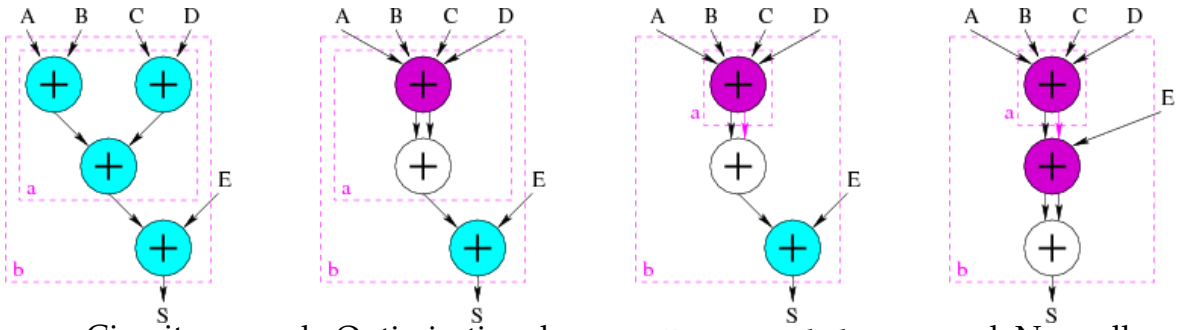
3.1.2 Approche *Allocation optimale*

Dans [UKL99] est présenté un algorithme en temps polynomial qui, à partir d'une expression arithmétique, a pour but de déterminer la structure optimale de *CSA* minimisant la chaîne longue. Cela part de la constatation de base que la chaîne critique d'un circuit est le critère le plus important à améliorer.

Les résultats présentés sur différentes opérations (mêmes conditions que précédemment) montrent une amélioration significative du temps et de la surface par rapport à la version classique, par exemple, -23% en temps et -39% en surface pour une somme de 9 nombres, -25% en temps et -20% en surface pour l'opération $A * B + C * D + E * F + G * H$.

Un nouveau concept est de plus présenté ici, appelé *auxiliary port*. C'est une solution au problème de *hiérarchie* dans les circuits. Le postulat de base est que la conception de *gros* circuits se fait en utilisant la hiérarchie (exemple dans la Figure 3.2.a). Optimiser chaque sous-circuit séparément amène à une réalisation qui n'est pas optimale pour le circuit complet à cause des convertisseurs $CS \rightarrow NR$ en sortie de chaque sous-circuit, comme montré dans la Figure 3.2.b.

Le concept présenté consiste en la création d'un nouveau port appelé *auxiliary port*. Dans l'exemple présenté, il consiste en *remonter* la sortie du circuit a à l'entrée du convertisseur, comme montré dans la Figure 3.2.c. La sortie de ce circuit devient donc en représentation *CS*, c'est pourquoi un deuxième port est nécessaire. Suite à cela, une nouvelle optimisation peut être faite (cf. Figure 3.2.d).



a. Circuit hiérarchique — b. Optimisation de chaque sous bloc — c. *Remontée* de la sortie — d. Nouvelle optimisation

FIG. 3.2 – Solution au problème de hiérarchie

Les résultats présentés se font, entres autres, sur un circuit contenant 9 opérateurs répartis en 3 sous-circuits.

Ce concept améliore le temps de -25% par rapport à une synthèse classique, et -17% par rapport à une optimisation "classique". Pour la surface, les valeurs sont de -14% et -5%. Les performances obtenues sont donc meilleures grâce à ce nouveau concept.

3.1.3 Gestion des blocs non arithmétiques et des multiplieurs

Trois concepts sont présentés dans [KU00a] et [KU00b] : l'optimisation *à travers la hiérarchie* (notion restreinte de [UKL99]), celle *à travers les multiplexeurs* et celle *à travers les multiplieurs*.

Les algorithmes précédents étant limités à des circuits purement arithmétiques, le concept d'optimisation *à travers les multiplexeurs* permet de surmonter cette limitation.

Trois étapes sont effectuées, comme montré dans la Figure 3.3 :

1. *move-up* (remontée des opérations avant le multiplexeur, cf. Figure 3.3.b),
2. l'optimisation des blocs arithmétiques (cf. Figure 3.3.c),
3. *move-down* (les convertisseurs finaux de chaque arbre sont redescendus après le multiplexeur, on obtient donc un seul convertisseur, mais deux multiplexeurs, cf. Figure 3.3.d).

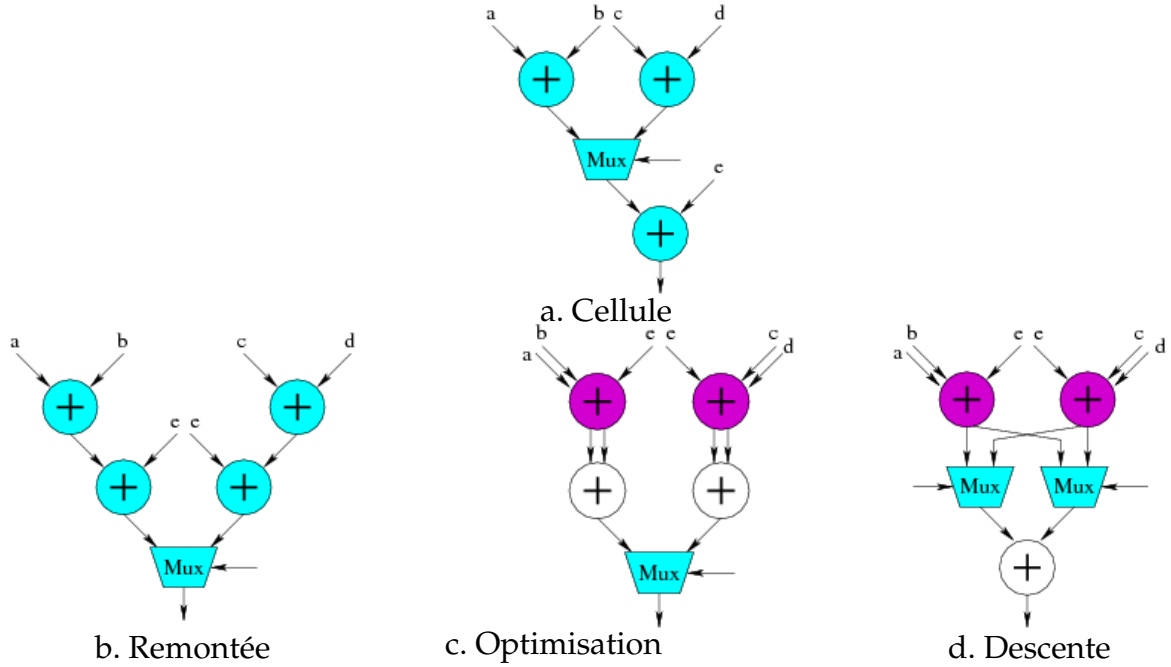


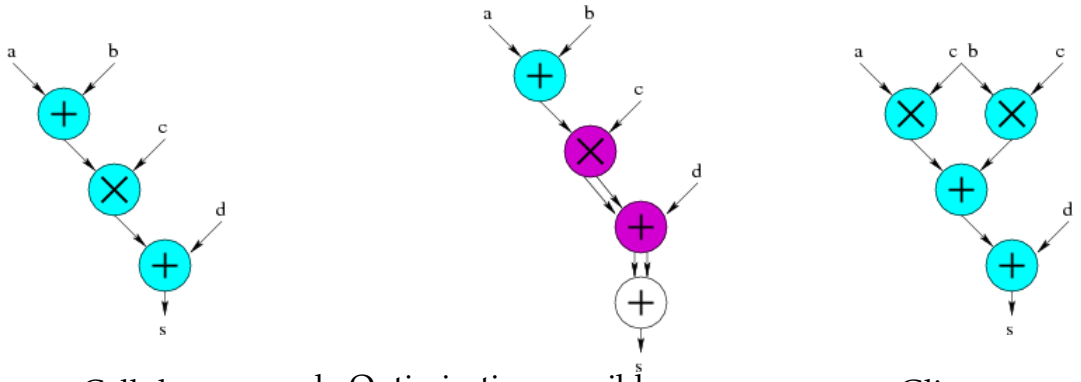
FIG. 3.3 – Optimisation à travers les multiplexeurs

Le concept d'optimisation *à travers les multiplieurs* a pour but quant à lui de compenser le fait qu'un multiplieur ne puisse être qu'une feuille des arbres d'additions formés.

Dans la Figure 3.4 sont présentées une cellule (a) et sa version optimisée (b). L'additionneur final du multiplieur a bien été optimisé, mais il n'a pas été possible d'optimiser l'additionneur d'entrée de la cellule. L'idée est donc de faire *glisser* le multiplieur

en entrée de façon à élargir la portion d'additionneurs, comme montré dans la Figure 3.4.c. L'additionneur pourra de cette façon être transformé en *CSA*.

Il est cependant conseillé de n'effectuer cette solution que lorsque les contraintes de temps sont primordiales, car l'augmentation en surface est alors très importante.



a. Cellule b. Optimisation possible c. Glissement

FIG. 3.4 – Optimisation à travers les multiplieurs

Les résultats présentés comparent ces nouvelles approches à des synthèses *RTL* classiques ainsi qu'à la précédente approche présentée dans [KJT98b].

Ils se résument à une amélioration systématique du temps critique grâce à l'introduction de ces nouveaux concepts, et des performances en surface tantôt meilleures, tantôt moins bonnes.

Remarques :

1. Le principe des deux nouveux concepts présentés s'apparente au principe de *glissement* que nous avons présenté dans la Figure 1.8.
2. Dans notre approche, nous prévoyons d'utiliser les multiplieurs redondant et mixte. Les multiplieurs ne sont donc pas contraints d'être des feuilles des arbres à optimiser. Ce principe de *glissement* ne nous est donc pas utile en ce qui concerne les mutliplieurs.

3.1.4 Compromis Surface/Délai

Le problème de *limitation de type donnée* est de nouveau traité dans [KK01] dans lequel il est fait une étude du compromis entre temps et surface.

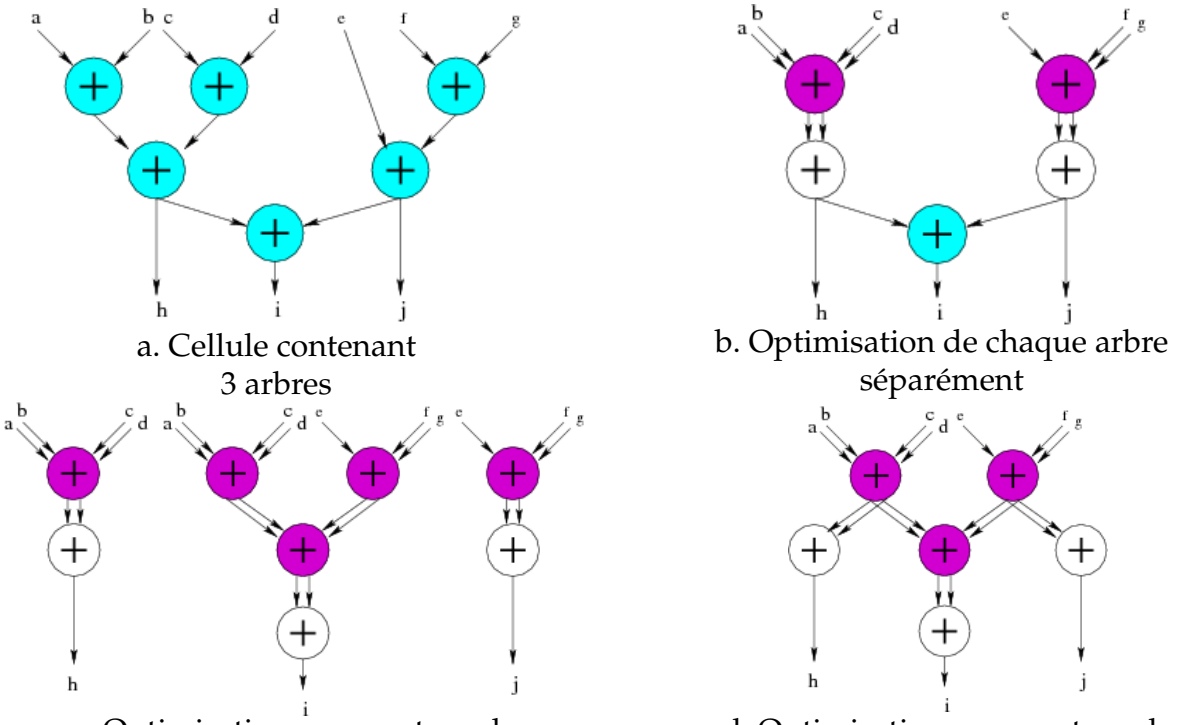
Trois optimisations sont comparées :

1. optimisation des arbres séparément,
2. optimisation des différentes opérations indépendamment, sans partage de ressources,

3. optimisation des opérations avec partage de ressources.

Le principe présenté s'apparente au principe de *dédoublement de la donnée* déjà exposé. L'étude faite ici montre l'intérêt de ce principe par rapport aux autres approches possibles, principalement l'*optimisation sans partage de ressource*, ce qui n'avait pas été envisagé jusque-là.

Les architectures des différentes optimisations possibles d'un circuit sont présentées dans la Figure 3.5 et leurs performances respectives dans le Tableau 3.2.



a. Cellule contenant 3 arbres

b. Optimisation de chaque arbre séparément

c. Optimisation sans partage de ressources

d. Optimisation avec partage de ressource

FIG. 3.5 – Architectures compromis temps/surface

	Surface ($unit$)			Temps (ns)		
	16 bits	32 bits	64 bits	16 bits	32 bits	64 bits
a	13 000	41 000	102 000	13.3	14.8	15.3
b	9 000	27 000	56 000	12.3	13.1	14.2
c	14 000	31 000	68 000	10.4	10.8	10.8
d	12 000	28 000	59 000	10.8	11.6	12.1

TAB. 3.2 – Performances compromis temps/surface

Ces résultats montrent que l'optimisation sans partage de ressources est la moins performante du point de vue de la surface. En effet, il y est fait moins d'optimisations que dans les deux autres.

Les meilleures performances en temps sont obtenues avec l'optimisation sans partage de ressources. En effet, le nombre de portes traversées est le même que pour celle effectuant un compromis, mais la traversée des portes est plus rapide du fait que chaque opérateur a sa sortie connectée à un seul autre opérateur, et a donc une capacité de sortie moindre. Le surcoût en surface avec cette optimisation la rend cependant difficilement utilisable.

C'est donc l'optimisation avec partage de ressource qui est présentée comme étant la meilleure, car elle génère le meilleur rapport temps/surface.

3.1.5 Prise en compte de la vue physique

Dans [UK02], Kim s'intéresse à la vue physique des circuits.

Le postulat de base est que le temps de propagation dans les fils d'interconnexion devient au fur et à mesure des années de plus en plus important par rapport au temps de propagation des portes. De nouveaux phénomènes existent avec les nouvelles technologies telles que la diaphonie.

L'idée directrice est toujours de se baser sur l'utilisation des *CSA*, mais le but est ici de rendre les interconnexions les plus régulières possibles.

L'algorithme présenté se déroule en deux phases :

1. synthèse optimale d'un algorithme avec utilisation des *CSA* (optimisation au niveau mot),
2. raffinement des interconnexions au niveau du bit.

La synthèse est celle qui a déjà été présentée. Le raffinement est une optimisation au niveau bit qui ne modifie pas la topologie faite par la phase 1 au niveau mot. Il est constitué de deux méthodes.

La première est la fusion de deux demi-additionneurs (noté *HA* pour *half-adder*) en un *FA* (en effet, des *HA* ont pu être instanciés lors de la première phase à la place de *FA* pour les *CSA* additionnant des nombres de tailles différentes par exemple).

La seconde est le réordonnancement des entrées d'un *FA*, de façon à rendre plus régulier le positionnement des entrées en fonction du dessin en masque du *FA*.

Les résultats présentés comparent cette nouvelle approche à une approche effectuant elle aussi une optimisation du temps au niveau bit [SMOR98], montrant que la nouvelle approche améliore le chemin critique.

3.2 Extension de la représentation Carry-Save

Les travaux de Alan N. Willson Jr. ont la particularité d'effectuer des modifications de la représentation Carry-Save pour élargir les possibilités d'optimisation.

3.2.1 Optimisation des *CSA* au niveau bit

Dans [KYW99] est présentée une optimisation au niveau bits. Une nouvelle représentation est introduite, appelée la représentation *relax carry-save* (notée RCS), étant donnée qu'elle permet à chaque bit d'un nombre d'être codé sur 1, 2 ou 3 termes. Lors de l'addition de nombres n'ayant pas la même taille, plutôt que d'instancier systématiquement des *FA* pour les bits de poids faible et des *HA* pour les bits de poids fort (additionnant 2 bits au lieu de 3 pour les bits pour lesquels on dépasse la taille d'un des 2 opérandes), l'idée est de trouver la configuration optimale entre ces deux cellules.

L'algorithme présenté construit une matrice $K - 2 * N$, N étant la taille du plus grand nombre à sommer, et K le nombre d'opérandes à sommer. Pour chaque élément de la matrice il définit quelle porte utiliser entre le *HA* et le *FA*. La contrainte est de respecter la notation Carry-Save classique à la sortie de la matrice de façon à pouvoir effectuer la conversion finale $CS \rightarrow NR$.

Des résultats sont présentés, comparant les coûts de différents filtres utilisant soit une instanciation de *CSA* classique, soit cette nouvelle approche. Ils montrent que la nouvelle approche donne de meilleures performances, avec des coûts améliorés de 15% à 30%.

3.2.2 Signaux *multiple vecteurs*

Les problèmes traités dans [YYW01] sont le traitement des registres (principes de *operation forward* et *operation backward*), et celui des *sorties multiples* (principes de *operation duplicate* et *operation merge*).

Ces problèmes sont traités avec une nouvelle notion, celle de signaux dits *multiple vecteurs* i.e. une extension de la notation Carry-Save qui propose de coder les nombres sur plus de 2 termes.

Les résultats présentés comparent trois architectures pour différents circuits : une avec le nouvel algorithme présenté, une autre classique, et enfin une qui n'utilise pas ce nouveau principe [YKW00].

Par rapport à une architecture classique, le temps est toujours amélioré (jusqu'à -45.5%) et la surface toujours dégradée (jusqu'à +60.3%). Par rapport à une architecture utilisant la représentation Carry-Save, mais pas de signaux *multiple vecteurs*, les conclusions sont les mêmes (jusqu'à -56% en temps et +15% en surface).

On peut en déduire que cette nouvelle approche permet d'améliorer encore plus le temps au détriment de la surface.

3.3 Réorganisation des graphes

3.3.1 Ordonnancement de graphe

Les travaux de Paolo Ienne se basent sur la modification de graphes de façon à ce que les outils de synthèse puissent utiliser les CSAs au mieux.

L'idée directrice est que le frein principal à une bonne utilisation des CSAs réside dans la présence d'opérateurs non arithmétiques. L'algorithme présenté dans [IV04] a donc pour but d'aggrandir les portions d'additionneurs chainés de façon à maximiser l'usage des CSAs.

Cet algorithme a pour entrée un *DAG* $G(V, E)$ représentant le circuit à optimiser. V correspond à l'ensemble de opérateurs, et E, l'ensemble des signaux. Les nœuds du graphe G sont ordonnés de telle façon que pour chaque arc (u, v) de G, u apparaisse avant v dans l'ordonnancement. La fonction $Ord()$ donne la position d'un nœud. De plus, une fonction $Class()$ est définie, qui renvoie la classe d'un nœud : A si c'est un opérateur arithmétique, L sinon.

Le but de l'algorithme est décrit comme suit : *Etant donné un graphe G, un graphe H équivalent est créé (i.e. représentant un circuit avec le même comportement que le circuit représenté par G) pour lequel tous les nœuds u et v tel que $Class(u) = A$ et $Class(v) = L$ vérifient $Ord(u) < Ord(v)$ ou $Ord(v) < Ord(u)$*. Le graphe H est dit *trié*.

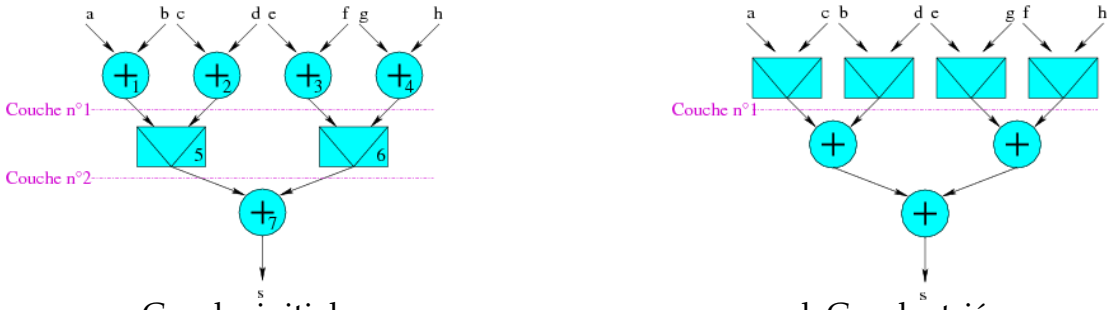
Le graphe obtenu a *séparé* les blocs arithmétiques des autres, et est donc apte à être optimisé.

Cet algorithme n'est cependant pas toujours applicable. Une solution est donc présentée, consistant à trier un graphe *autant que possible*. En définissant le nombre de *couches* d'un graphe comme étant le nombre de transition A/L et L/A, le second algorithme présenté permet d'obtenir le graphe H avec le nombre minimum de couches.

Dans la Figure 3.6 est présenté un exemple. A partir d'un graphe non ordonné contenant deux couches (cf. Figure 3.6.a : $Class(1) = A$, $Class(5) = L$ et $Class(7) = A$, or, $Ord(1) < Ord(5) < Ord(7)$), on obtient un graphe ordonné contenant le minimum de couches possible, une seule (cf. Figure 3.6.b).

Les résultats présentés comparent les performances de circuits synthétisés, avec optimisation ou non du graphe original. Pour certains circuits, une implantation faite à la main est également présentée.

Ces résultats montrent que le délai est amélioré (jusqu'à -43%) pour tous les circuits testés sauf un (+8%) par rapport à la synthèse directe et que les différences de surface varient entre -20% et +33%. Pour les circuits créés manuellement, le délai est moins bon, et la surface plus petite.



a. Graphe initial d. Graphe trié

FIG. 3.6 – Ordonnancement de graphe

Remarque :
Ce principe peut s'apparenter à une méthodologie permettant d'appliquer de façon automatique le principe de **glissement** déjà présenté.

3.3.2 Exploration des optimisations temps / surface

L'algorithme présenté précédemment est également la base de [VI04]. Les règles appliquées de façon à pouvoir effectuer le tri sont présentées et classées en deux catégories : celles qui détériorent la surface et celles qui ne le font pas. L'idée est de déterminer l'ordre dans lequel appliquer ces règles en fonction du critère à privilégier : par exemple, pour privilégier la surface, n'appliquer que les règles ne détériorant pas la surface, puis appliquer les autres dans un second temps (si besoin) jusqu'à obtenir un graphe trié.

Les résultats présentés montrent deux optimisations : une se focalisant sur le délai, et la seconde se focalisant sur la surface. La première améliore le délai (jusqu'à -46%) avec une détérioration de la surface (jusqu'à +833%). Pour la seconde, les améliorations en délai sont quasiment équivalentes, tandis que l'optimisation en surface est largement meilleure (jamais de dégradation et jusqu'à 23% d'amélioration).

3.4 Synthèse haut niveau

Depuis 1995 sont réalisés des travaux concernant l'arithmétique redondante à l'Ecole Normale Supérieur de Lyon traitant de l'introduction des notations redondantes dans la synthèse de haut niveau. Ces travaux se concluent par la thèse d'Olivier Peyran [Pey97].

Ces travaux présentent toutes les contraintes spécifiques qu'induit l'utilisation de l'arithmétique mixte dans la synthèse de haut niveau, et plus particulièrement dans le processus d'ordonnancement ; ils visent à déterminer de façon automatique la représentation de chaque signal d'un circuit [MMP96b].

Les additionneurs et multiplieurs mixtes et redondants sont utilisés, ainsi que les diviseurs. De ce fait, les deux notations Carry-Save et Borrow-Save sont utilisées.

Deux postulas de base sont présentés pour résumer l'arithmétique mixte [MMP96a] :

1. les opérateurs avec sortie redondante sont plus petits et plus rapides que leurs équivalents avec sortie non redondante,
2. les opérateurs avec entrées non redondantes sont plus petits et plus rapides que leurs équivalents avec entrée redondante (il faut en particulier citer les multiplieurs redondants qui ont une surface très importante).

Une formulation en programmation linéaire est ainsi proposée [MMP97, MP97a, MP97b, MP97c], ayant pour but d'utiliser le plus possible d'opérateurs mixtes, i.e. utiliser des opérateurs redondants uniquement *quand cela est nécessaire* et insérer dès que possible des convertisseurs $R \rightarrow NR$ de façon à limiter le nombre d'opérateurs redondants et ainsi limiter l'augmentation de la surface et de la consommation.

3.5 Conclusion

Nous avons pu constater que, suite à la réalisation fructueuse de différents circuits utilisant l'arithmétique redondante présentée dans la Section 2.4, plusieurs outils la mettant en œuvre émergent depuis quelques années, en synthèse *RTL* et en synthèse de haut niveau.

Les bons résultats obtenus par les différents outils présentés nous confortent dans l'idée que l'automatisation de l'utilisation de l'arithmétique redondante est une voix avantageuse à explorer, d'autant plus qu'aucun de ces outils n'exploite toutes les règles d'optimisation présentées dans la thèse de Yannick Dumonteix. Entre autre, seule la représentation Carry-Save a jusqu'à ce jour été utilisée dans la synthèse *RTL*, pas la représentation Borrow-Save, et seuls des additionneurs mixtes ont été utilisés, jamais des multiplieurs.

Notre choix s'est porté sur la synthèse logique, étant donné que les différentes optimisations que nous avons mis en œuvre ont été inspirées de la thèse de Yannick Dumonteix. Notons que nous avons pu reconnaître certains de ses postulats dans les outils présentés, mais qu'ils n'ont pas encore tous été utilisés. Comme nous le verrons, nous nous sommes également inspirés de postulats présentés dans ce chapitre en les adaptant à notre approche.

Un des buts de cette thèse est alors clair : mettre en œuvre les différentes optimisations possibles présentées par Yannick Dumonteix et évaluer les avantages et désavantages de chacune. Nous évaluons, entre autres, l'intérêt de l'usage des multiplieurs redondants et mixtes, ainsi que celui de l'utilisation de la représentation Borrow-Save pour les circuits contenant des soustractions. Nous comparons enfin l'approche *tout en redondant* à l'approche cherchant la solution optimale en ce qui concerne la chaîne longue.

Chapitre 4 Optimisation de chemins de données arithmétiques

Le but principal de cette thèse est d'utiliser de façon automatique l'arithmétique redondante, de façon à la rendre plus accessible. En d'autres termes, nous voulons intégrer automatiquement des opérateurs redondantes dans chemins de données arithmétiques décrit en arithmétique classique. Comme on a pu le voir, les bonnes performances intrinsèques de tels opérateurs montrent l'intérêt de cette approche.

Nous effectuons dans ce chapitre une formalisation précise de notre problème, et détaillons la place de nos outils dans le flot de conception *VLSI*.

Nous présentons ensuite les trois algorithmes que nous avons mis en œuvre, basés sur la redéfinition des enchaînements entre opérateurs arithmétiques. Le premier est à base de reconnaissance de motifs, les deux autres, à base de labellisation de graphe.

4.1 Définition du problème

4.1.1 Optimisation arithmétique dans le flot de conception VLSI

La Figure 4.1 présente le flot de conception *VLSI*. A partir d'une description haut niveau (i.e. description algorithmique), la phase de synthèse sert à obtenir une description bas niveau (i.e. une description structuelle en portes caractérisées d'une bibliothèque). Les phases de placement et routage servent à obtenir le dessin des masques. On connaît à ce stade la surface du circuit. Une extraction permet enfin d'obtenir une estimation de la chaîne critique. Différentes phases de vérification existent de façon à valider chaque étape.

Dans la Figure 4.2 est présentée en détail l'étape du flot qui nous intéresse plus particulièrement : la synthèse, telle qu'elle est décrite dans [Jac99]. Cette phase est découpée en trois étapes : synthèse haut-niveau, synthèse *RTL* et synthèse logique. La synthèse *RTL* est également découpée en trois étapes (symbolisées par des cadres en pointillés) : la compilation (passage d'une description comportementale synchrone en une série d'affectations concurrentes), l'optimisation (transformation des affectations concurrentes selon des critères) et la projection structurelle (passage d'une description en portes virtuelles à une description en portes caractérisées d'une bibliothèque).

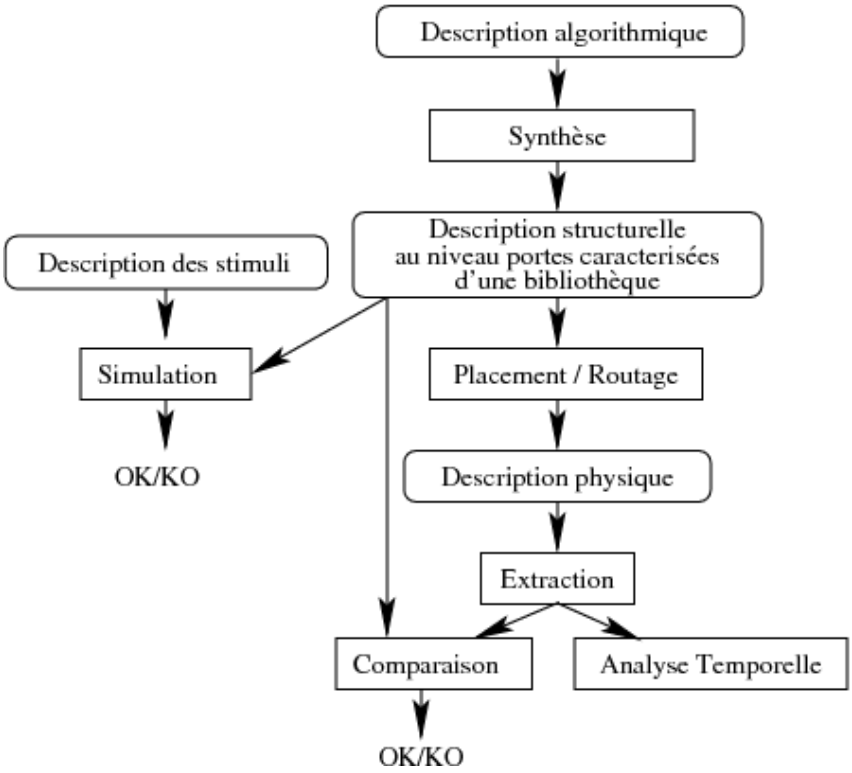


FIG. 4.1 – Flot de conception VLSI

Remarque :
La synthèse *RTL* est différente pour la partie contrôle et la partie chemin de données d'un circuit. Nous ne nous intéressons dans cette thèse qu'à la synthèse de chemins de données et nous focalisons donc sur les affectations concurrentes combinatoires et arithmétiques avec registres.

La Figure 4.2 permet également de voir à quel niveau se situe l'optimisation arithmétique : entre les phases de synthèse logique et de projection structurelle.

L'optimisation arithmétique prend en entrée une description structurelle au niveau portes virtuelles qu'elle modifie. Plus spécifiquement, elle prend une description structurelle hiérarchique de haut niveau, ce que nous appelons la description *molle*, car les opérateurs arithmétiques ne sont pas complètement spécifiés dans la façon de décrire le circuit. Elle fournit une description structurelle de bas niveau virtuelle i.e. description hiérarchique en portes logiques indépendantes de la technologie cible.

Il y a donc une étape préliminaire à la projection structurelle consistant à faire la traduction entre la description structurelle de bas niveau virtuelle et la description structurelles en portes réelles qui sert de point d'entrée à la projection structurelle classique.

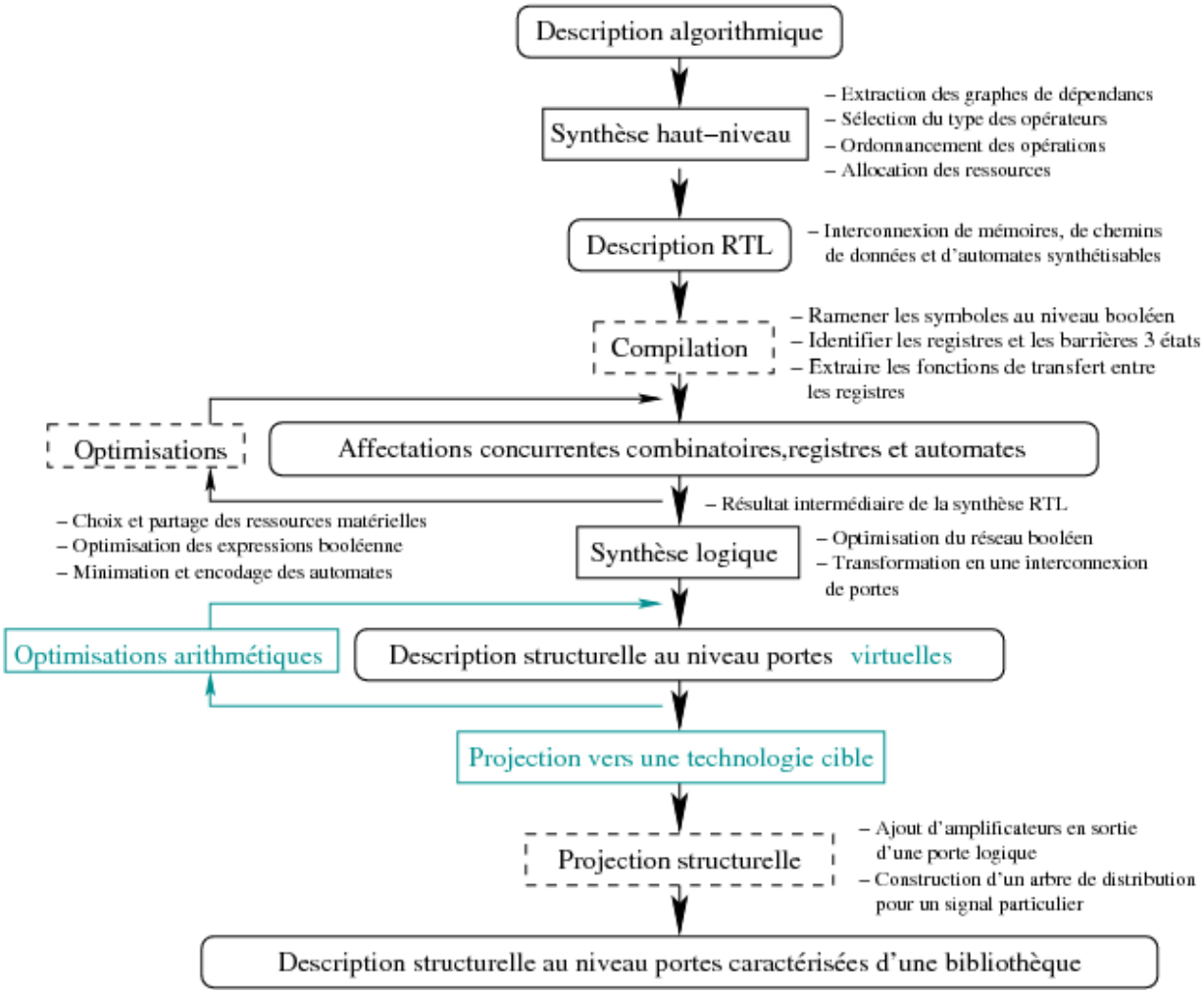


FIG. 4.2 – Synthèse

Notre méthodologie d'optimisation se résume à la modification du chemin de données de façon à utiliser des opérateurs redondants et mixtes. De cette façon la description finale obtenue du chemin de données est optimisée en fonction de notre savoir-faire arithmétique et assure la projection vers une technologie cible.

4.1.2 Approches choisies

Nous faisons ici un bref récapitulatif des différentes approches que nous avons traitées. Comme nous allons le voir, trois algorithmes ont été mis en place.

Nous présentons également les différentes solutions possibles pour prendre en compte les soustractions, solutions qui ont effectivement été développées et comparées.

Reconnaissance de motifs

Dans le but d'initialiser un chemin de données arithmétique en *redondant dès que possible*, la première approche qui nous a paru naturelle est d'encapsuler notre savoir-faire arithmétique dans des **règles** consistant à trouver des **motifs** dans le circuit et de les remplacer par d'autres afin de modifier sa structure. Typiquement, un motif est un regroupement d'opérateurs, et la règle est la façon de le remplacer par un autre motif utilisant l'arithmétique redondante, dans lequel les opérateurs ont donc de meilleures performances.

Algorithme de labellisation

Dans un second temps, nous avons mis en place un algorithme dit de labellisation qui transforme systématiquement tous les signaux possibles en représentation redondante. De cette façon les opérateurs arithmétiques d'un circuit sont systématiquement transformés en leur équivalent redondant ou mixte.

Cette approche est moins flexible que la précédente étant donnée qu'elle ne modifie pas la structure du circuit. Cependant, l'algorithme correspondant est théoriquement plus rapide à exécuter, ce qui est un point important.

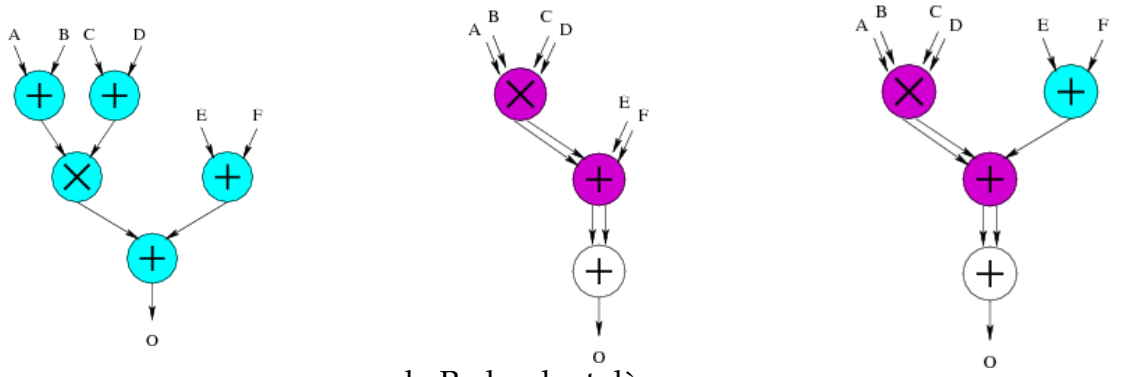
Allocation optimale

L'initialisation en redondant dès que possible apporte un gain substantiel. L'avantage d'une telle approche est d'avoir un outil rapide qui garantit la plus faible surface et une diminution de la chaîne longue. Cependant, cette approche n'apporte pas le chemin critique optimal, comme nous avons pu le voir dans la Section 1.1.1.

Reprenons l'exemple présenté dans la Figure 4.3 :

- la chaîne critique de la Figure 4.3.a contient deux additionneurs classiques et un multiplieur classique,
- l'initialisation en redondant dès que possible réduit cette chaîne critique à un multiplieur avec sortie redondante, un additionneur redondant et un additionneur classique, comme montré dans la Figure 4.3.b,
- étant donné qu'un additionneur classique est plus rapide qu'un multiplieur avec sortie redondante, l'addition de E et F peut s'effectuer en parallèle avec la multiplication sans que cela ne détériore la chaîne critique. Ne pas transformer cet additionneur en redondant modifie l'additionneur redondant de la chaîne critique en un additionneur mixte, c'est pourquoi la chaîne est alors raccourcie, comme montré dans la Figure 4.3.c.

Les performances de ces architectures confirment qu'avec une allocation optimale en temps, l'optimisation de la chaîne longue est plus performante qu'avec une allocation *redondant dès que possible*, avec un surcoût en surface.



a. Cellule b. Redondant dès que possible c. Allocation optimale

Nombre de bits	Surface (mm^2)			Temps (ns)		
	a	b	c	a	b	c
8	0.24	0.21	0.24	60.5	54.7	50.6
	ref	-9.8%	0%	ref	-9.6%	-16.4%
16	0.66	0.61	0.68	81.64	65.71	61.71
	ref	-8%	+1.9%	ref	-19.5%	-24.4%

FIG. 4.3 – Allocation optimale en temps

Nous avons donc mis en place un second algorithme de labellisation, qui n'a pas pour but de transformer en *redondant dès que possible*, mais de trouver la solution donnant la chaîne critique optimale. Cette allocation optimale en temps est le résultat du regroupement de la phase d'initialisation en redondant et de celle d'analyse des temps de propagation avec retour en notation non redondante présentée dans la Section 1.1.1. Le critère de temps étant primordial, nous avons choisi de développer un algorithme regroupant ces deux phases. Cet algorithme permet, grâce à une fonction de coût, d'obtenir la meilleure chaîne critique possible (avec une dégradation de la surface par rapport à la solution obtenue avec l'algorithme de labellisation direct).

Soustraction

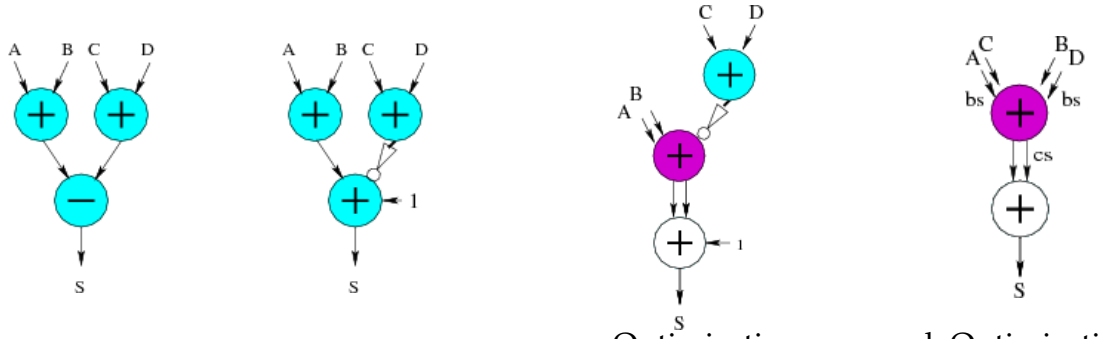
En ce qui concerne la façon d'optimiser la soustraction, nous avons tout d'abord fait le choix de n'utiliser que la notation Carry-Save, ce qui est couramment fait dans la littérature et apporte de bons résultats. Pour cela, les soustractions sont transformées en additions, en utilisant le complément à 2 du terme à soustraire.

Dans un second temps nous avons fait le choix d'utiliser les architectures d'additionneurs et de multiplieurs utilisant la notation Borrow-Save. Notre but était de contourner l'usage unique de la représentation Carry-Save et d'évaluer le gain potentiel. En effet, dans certains cas, l'insertion d'un inverseur au milieu d'une chaîne d'opérateurs arithmétiques peut empêcher certaines optimisations, comme cela est montré

dans la Figure 4.4 :

- la Figure 4.4.a montre un chemin de données à optimiser,
- la Figure 4.4.b montre la transformation des soustractions en additions,
- la Figure 4.4.c est l'architecture provenant de l'optimisation Carry-Save : on voit que l'insertion d'un inverseur dans la chaîne arithmétique empêche l'utilisation d'un opérateur redondant, l'optimisation n'est donc pas parfaite,
- la Figure 4.4.d montre l'architecture obtenue avec optimisation directe grâce à l'utilisation d'un additionneur utilisant la représentation Borrow-Save : on voit que le problème précédent n'existe pas dans ce cas.

Les performances des différentes architectures montrent bien que c'est l'optimisation Borrow-Save qui engendre la meilleure chaîne longue.



a. Circuit : $(A+B)-(C+D)$

b. Transformation : $(A+B)+\overline{(C+D)}$

c. Optimisation CS : $(A+B)+\overline{(C+D)}$

d. Optimisation BS : $(A-C)+(B-D)$

Nombre de bits	Surface (mm^2)			Temps (ns)		
	b	c	d	b	c	d
64	0.63	0.48	0.3	47.3	51.5	36.1
	ref	-24.7%	-51.8%	ref	+8.9%	-23.7%
128	1.41	1.05	0.65	62.6	66.8	45.4
	ref	-25.7%	-53.6%	ref	+6.7%	-27.5%

FIG. 4.4 – Problème dû à une soustraction

Tous les algorithmes développés offrent la possibilité d'utiliser ou non la représentation Borrow-Save.

4.1.3 Formalisation

Nous nous intéressons donc à la phase d'initialisation en redondant d'un chemin de données. Comme nous venons de le présenter, deux étapes de la méthodologie définies dans le Chapitre 1 sont traitées : l'initialisation en redondant dès que possible, et le retour en notation classique.

Le problème posé est donc l'optimisation automatique d'un chemin de données en arithmétique classique en introduisant des opérateurs redondants et mixtes.

Les différents algorithmes développés ont pour entrée un chemin de données arithmétique, et fournissent en sortie, le même chemin de données, qui a été optimisé. Ces algorithmes sont basés sur la théorie des graphes et le chemin de données est donc représenté par un graphe.

Définition 4.1.

Un chemin de données est représenté par un graphe direct acyclique (*DAG*) $G = (N, \mathcal{A})$ tel que :

1. les opérateurs (arithmétiques ou non), les entrées et les sorties représentent l'ensemble des nœuds N du graphe G ;
2. les signaux représentent l'ensemble des arcs $(x, y) \in \mathcal{A}$.

Notations :

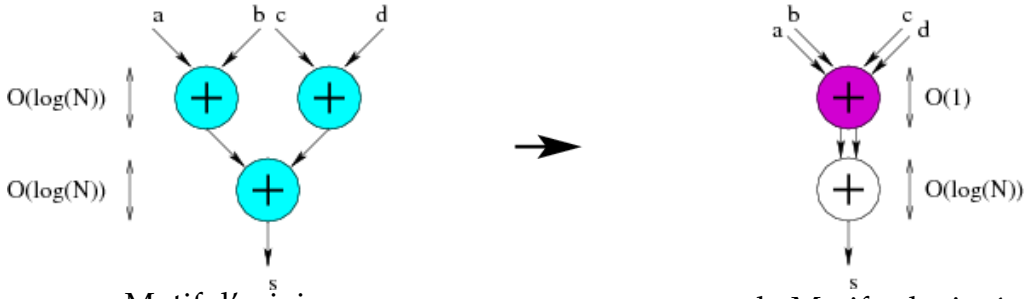
- pour chaque nœud $x \in N$, $\Gamma^+(x)$ est l'ensemble des successeurs directs de x,
$$\Gamma^+(x) = \{y \in N, (x, y) \in \mathcal{A}\},$$
- de la même façon, $\Gamma^-(x)$ est l'ensemble des prédécesseurs directs de x,
$$\Gamma^-(x) = \{y \in N, (y, x) \in \mathcal{A}\},$$
- $E \subset N$ est l'ensemble des nœuds représentant une entrée du circuit,
$$E = \{x \in N, \Gamma^-(x) = \emptyset\},$$
- $S \subset N$ est l'ensemble des nœuds représentant une sortie du circuit,
$$S = \{x \in N, \Gamma^+(x) = \emptyset\},$$
- $C \subset N$ est l'ensemble des connecteurs,
$$C = E \cup S,$$
- $N_a \subset N$ est l'ensemble des nœuds représentant un opérateur arithmétique ; les éléments de N_a peuvent donc correspondre à des opérateurs classiques, mixtes ou redondants ; les éléments des $N - N_a$ correspondent alors aux opérateurs non arithmétiques et aux entrées/sorties,
- $\mathcal{A}_a$ est l'ensemble des arcs correspondants à un signal pouvant être mis en représentation redondante,
$$\mathcal{A}_a = \{(x, y) \in \mathcal{A}, x \in N_a, y \in N_a\},$$
- dans certains cas, nous restreignons notre approche aux arborescences τ pour lesquelles chaque nœud $x \in N$ a un et un seul successeur dans τ.

4.2 Reconnaissance de motifs

Le but de la reconnaissance de motifs est d'appliquer l'étape d'initialisation en *redondant dès que possible*. Pour cela, un groupe d'opérateurs non redondants (motif d'origine) est remplacé par des opérateurs redondants et mixtes (motif substitué). Nous appelons *règle* l'ensemble des deux motifs.

Les deux principes fondamentaux à respecter sont que les deux motifs d'une règle doivent toujours avoir la même interface et le même comportement.

Un exemple est montré dans la Figure 4.5, dans laquelle la règle présentée remplace trois additionneurs classiques par un additionneur redondant et un additionneur classique (le convertisseur).



a. Motif d'origine b. Motif substitué

FIG. 4.5 – Exemple de règle

Nous allons présenter dans cette section l'algorithme à proprement parlé puis les différents ensembles de règles que nous avons établi. Nous expliquerons la gestion de la soustractions et des sorties multiples. Nous détaillerons enfin un exemple.

4.2.1 Algorithme

Nous avons fait le choix d'effectuer un parcours du graphe des sorties vers les entrées. Ce choix s'inspire des travaux de Taewhan Kim [KJT98a] qui a proposé plusieurs algorithmes à base de parcours de graphes à partir des sorties. A chaque étape, un motif est recherché de façon à être remplacé par un autre. Cet algorithme est donc un processus récursif que l'on applique à chacun des arcs connectés à un nœud de sortie du graphe.

Cette approche est valable pour les arborescences. Son application directe à des *DAG* pose problème comme nous le verrons en détail par la suite. Nous présentons donc dans un premier temps l'algorithme applicable aux arborescences et présenterons dans un second temps comment il a été adapté aux *DAG*.

L'algorithme (à appliquer à l'arc prédécesseur direct au nœud de sortie de l'arbre) peut être défini comme suit :

A partir d'un arc (x, y) d'une arborescence τ :

- *on recherche si un motif peut être appliqué,*
- *si c'est le cas, on applique la règle correspondant au motif trouvé pour le remplacer par le motif correspondant.*

On effectue ces deux étapes jusqu'à ce qu'il n'y ait plus de motif possible i.e. jusqu'à ce que l'on ait traité tous les prédécesseurs directs de x. Une fois tous les motifs remplacés, on effectue par récursion la même opération avec tous les prédécesseurs directs de x. La récursion s'arrête lorsqu'on arrive à un arc étant un successeur direct d'un nœud d'entrée du graphe.

Algorithm 4.1 Algorithme de reconnaissance de motifs

Entrée : un arc (x, y) d'une arborescence τ

1: **si** $x \in E$ **alors**
2: $stop$
3: **fin si**
4: **tant que** $\exists motif\, M$ **faire**
5: Remplacer M selon la règle associée
6: **fin tant que**
7: **pour tout** $z \in \Gamma^{-}(x)$ **faire**
8: $Algorithm(z, x)$
9: **fin pour**

Au niveau de la comparaison entre le circuit et le motif, cela se résume à évaluer si il existe une portion du graphe représentant le circuit ayant pour racine (x, y) identique au motif.

Cette comparaison se fait à partir de l'arc appelé arc de sortie du motif. Pour chaque opérateur du motif (resp. signal), la portion de graphe est évaluée de façon à définir si elle possède un opérateur de même type (resp. un signal avec la même représentation).

Si la portion du graphe évaluée correspond au motif sur tous les critères, on conclut que la règle correspondant au motif est applicable.

Au niveau du remplacement du motif, la portion du graphe représentant le circuit est tout simplement remplacée par le graphe représentant le motif à substituer.

4.2.2 Description des règles

La partie fondamentale de cette approche, au delà de l'algorithme utilisé, est la description de l'ensemble des règles. Cet ensemble doit permettre de prendre en compte tous les cas de connexions possibles entre opérateurs arithmétiques, et d'encapsuler notre savoir faire concernant l'arithmétique redondante.

Nous présentons la façon de décrire une règle dans l'Annexe A. Cette façon de décrire les règles est la plus simple et intuitive possible. Cela nous a permis de développer un outil modulaire. En effet, notre outil peut de cette façon s'adresser à des personnes :

- n'ayant pas de connaissance en arithmétique, et utilisant donc l'ensemble de règles que nous avons proposé de part notre savoir-faire arithmétique,
- ayant des connaissances en arithmétique, et voulant faire des tests d'architecture et faisant donc le choix d'introduire leur propre ensemble de règles.

4.2.3 Ensemble des règles

De part notre expérience, nous avons défini des ensembles de règles constituées de motifs à trouver dans le graphe et de motifs les remplaçant. Les motifs remplaçants ont de meilleurs performances que les anciens. Les ensembles ont été définis de telle façon que les règles soient ordonnées i.e. s'il arrive que plusieurs règles soient applicables, c'est la première de la liste qui doit être appliquée.

Nous proposons trois ensembles de règles : un n'utilisant que la représentation Carry-Save et ne prenant donc en compte que les additions et les multiplications, et deux autres utilisant la représentation Borrow-Save (le premier utilisant les additionneurs avec sortie en Borrow-Save, le second utilisant les additionneurs avec sortie en Carry-Save) et prenant en compte addition, multiplication et soustraction.

Comme nous allons de voir dans les Figures 4.6 à 4.13, nous avons fait le choix de définir des motifs avec une profondeur de 2 i.e. des motifs possédant exactement 2 nœuds se chaînant : un opérateur dit "racine" et un ou deux opérateurs lui étant connectés. En effet, considérer des motifs plus grands aurait rendu le nombre de motifs total beaucoup trop important de façon à traiter tous les cas de connexions.

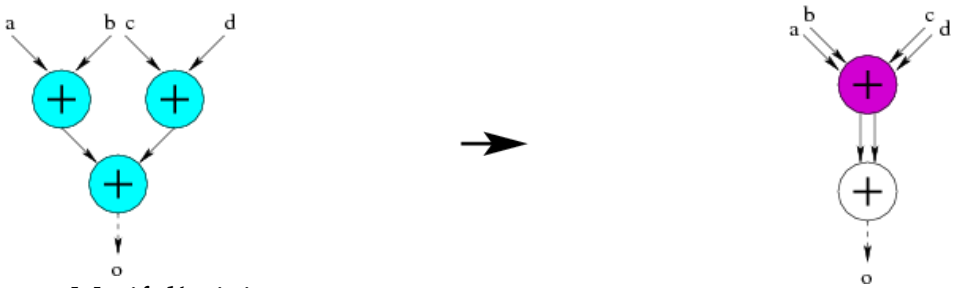
Nous allons dans ce qui suit présenter les différents types de règles. Nous expliciterons les améliorations apportées par chacune des règles de façon à montrer l'intérêt de la méthode.

Transformation d'une addition en un nombre Carry-Save

Le principe fondamental appliqué dans les différentes règles est d'effectuer une addition par l'utilisation d'un nombre Carry-Save plutôt que d'utiliser un additionneur classique. Différents cas se présentent :

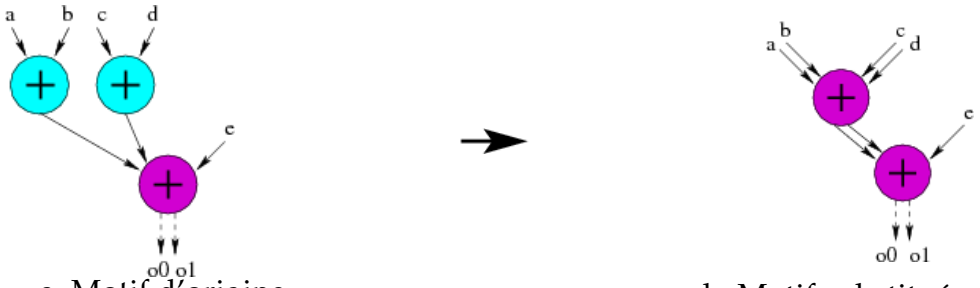
- si deux additionneurs sont connectés à un autre opérateur, ils sont "fusionnés" en un additionneur redondant (cf. Figures 4.6 et 4.7),
- si un seul additionneur est connecté à un opérateur, il est "effacé" et les nombres à son entrée sont transformés en un nombre Carry-Save (cf. Figure 4.8).
- l'ordre des entrées d'un additionneur peut être modifié de façon à pouvoir utiliser des opérateurs redondants (cf. Figure 4.9).

Toutes ces règles sont applicables grâce aux différentes propriétés de l'addition telle que l'associativité ou la commutativité. Elles sont par ailleurs également applicables si l'opérateur racine est un multiplieur.



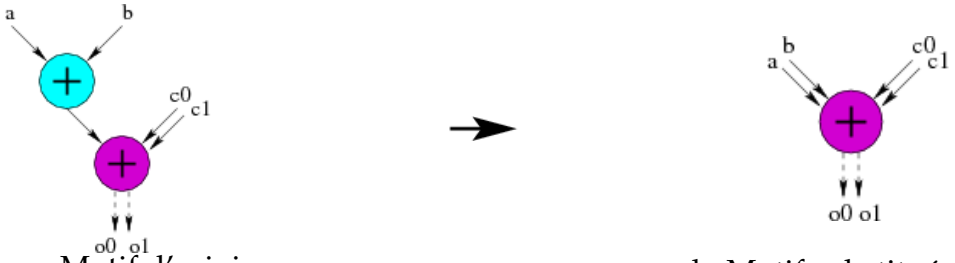
a. Motif d'origine b. Motif substitué

FIG. 4.6 – Exemple de règle n°1



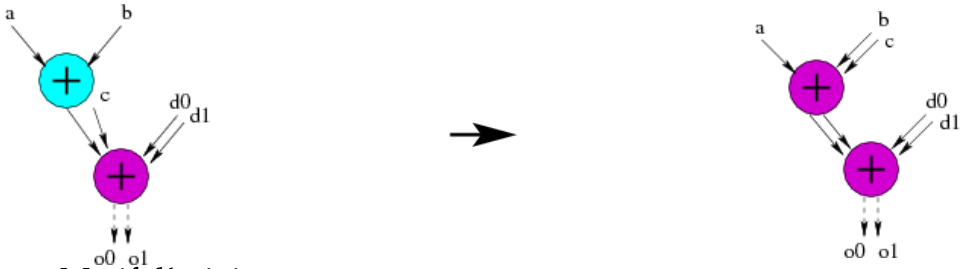
a. Motif d'origine b. Motif substitué

FIG. 4.7 – Exemple de règle n°2



a. Motif d'origine b. Motif substitué

FIG. 4.8 – Exemple de règle n°3



a. Motif d'origine b. Motif substitué

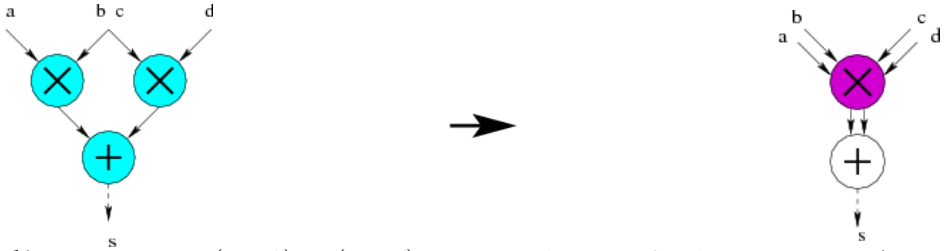
FIG. 4.9 – Exemple de règle n°4

Remarque :

Si l'opérateur racine est un opérateur non redondant, il nous faut ajouter un convertisseur dans le motif à substituer de façon à ne pas modifier l'interface du motif (exemple de la Figure 4.6).

Modification de la sortie d'un multiplieur

Dans le cas où l'opérateur racine est connecté à des multiplieurs, le même type d'optimisation n'est pas possible. Il est incorrect de fusionner deux multiplieurs classiques en un multiplieur redondant, comme montré dans la Figure 4.10. Pour pouvoir effectuer ce type de modification, il faudrait une représentation redondante définie comme étant la multiplication de deux nombres.

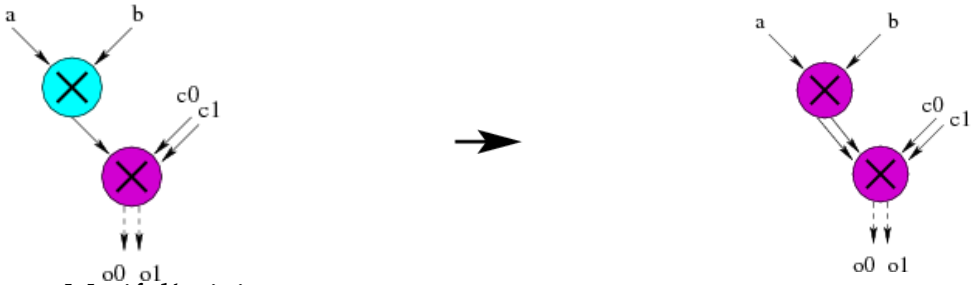


a. Motif d'origine : $s = (a * b) + (c * d)$ b. Motif substitué : $s = (a + b) * (c + d)$

FIG. 4.10 – Règle erronée

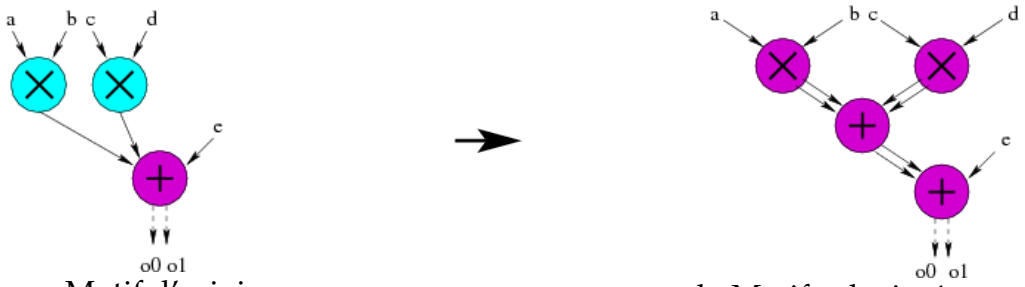
Ce qu'il est possible de faire est de transformer une sortie non redondante en une sortie redondante, comme montré dans les Figures 4.11 et 4.12. Comme on l'a vu dans le Chapitre 2, cela permet de supprimer l'additionneur final interne au multiplieur.

Même si l'ajout d'un additionneur redondant peut alors être nécessaire (cf. Figure 4.12), les performances sont améliorées. En effet, l'additionneur redondant ajouté est beaucoup plus performant que les deux additionneurs classiques internes aux multiplieurs qui sont supprimés.



a. Motif d'origine b. Motif substitué

FIG. 4.11 – Exemple de règle n°5

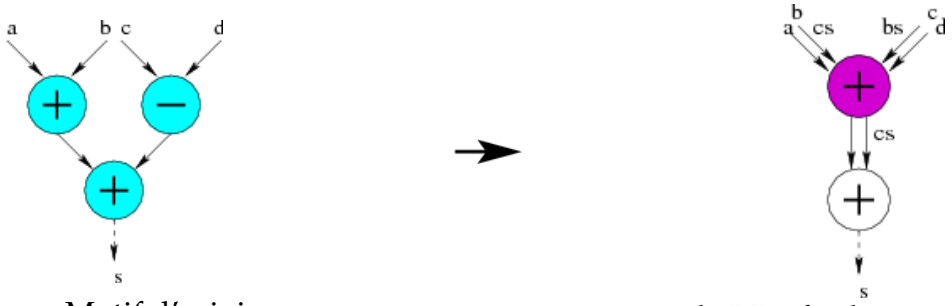


a. Motif d'origine b. Motif substitué

FIG. 4.12 – Exemple de règle n°6

Gestion de la soustraction

Les exemples de règles présentés précédemment font un récapitulatif des différentes connexions possible entre additions et multiplications. Pour les ensembles de règles prenant en compte également les soustractions, la même démarche est faite entre additions et soustractions et entre multiplications et soustractions. Un exemple est montré dans la Figure 4.13.



a. Motif d'origine b. Motif substitué

FIG. 4.13 – Exemple de règle n°7

Remarque :

Etant donné que dans ce cas, trois opérations différentes sont à prendre en compte (addition, multiplication et soustraction) et non plus deux (addition et multiplication), le nombre de règles croît de façon non négligeable.

Reprenons l'exemple précédent (Figure 4.13), de nombreux cas sont à prendre en compte pour gérer tous les cas de connexions :

- $(a - b) + (c - d)$,
- $(a - b) + (c + d)$,
- $(a + b) - (c + d)$,
- $(a - b) - (c - d)$,
- $(a - b) - (c + d)$.

Là où il existait un seul motif pour gérer ce type de connexion entre additions, il existe six motifs différents pour prendre en compte la soustraction.

Synthèse

Nous venons de présenter les principes mis en application dans les différents ensembles de règles que nous avons élaborés, à travers plusieurs exemples.

L'ensemble n'utilisant que la représentation Carry-Save comporte 24 règles, celui utilisant les architectures d'additionneurs Borrow-Save avec sortie Borrow-Save en comporte 84, et celui utilisant les architectures d'additionneurs Borrow-Save avec sortie Carry-Save 148. Une liste exhaustive du premier ensemble est donnée dans l'Annexe B.

4.2.4 Exemple de reconnaissance de motifs

Dans la Figure 4.14 est présenté un exemple du déroulement de l'algorithme de reconnaissance de motifs :

a. le premier arc à être traité est l'arc prédécesseur direct au nœud de sortie du graphe, arc appelé (x, y), un motif $M1$ est trouvé à partir de cet arc, et est remplacé en un motif $M1'$ consistant à transformer l'addition $E + F$ en un nombre Carry-Save,

b. par récursion, le nouvel arc à être traité est l'arc noté (w, x), le motif $M2$ est trouvé et remplacé par le motif $M2'$, la sortie du multiplieur est modifiée,

c. par récursion, l'arc (v, w) est traité, le motif $M3$ est trouvé et remplacé par le motif $M3'$, les additionneurs $A + B$ et $C + D$ sont supprimés grâce à l'utilisation d'un multiplieur redondant.

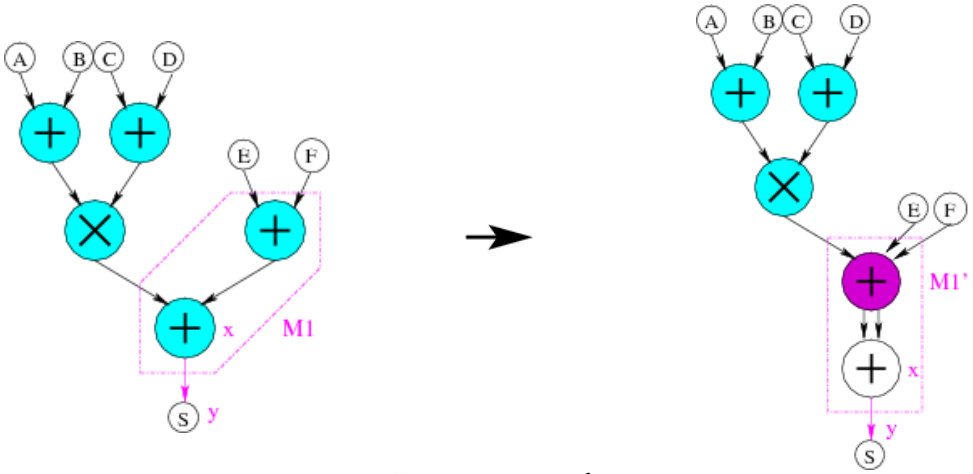
La récursion se continue jusqu'aux arcs successeurs directs des nœuds d'entrée du graphe, mais il n'y a aucun autre motif trouvé, c'est pourquoi ces étapes ne sont pas représentées dans la Figure.

4.2.5 Prise en compte des sorties multiples

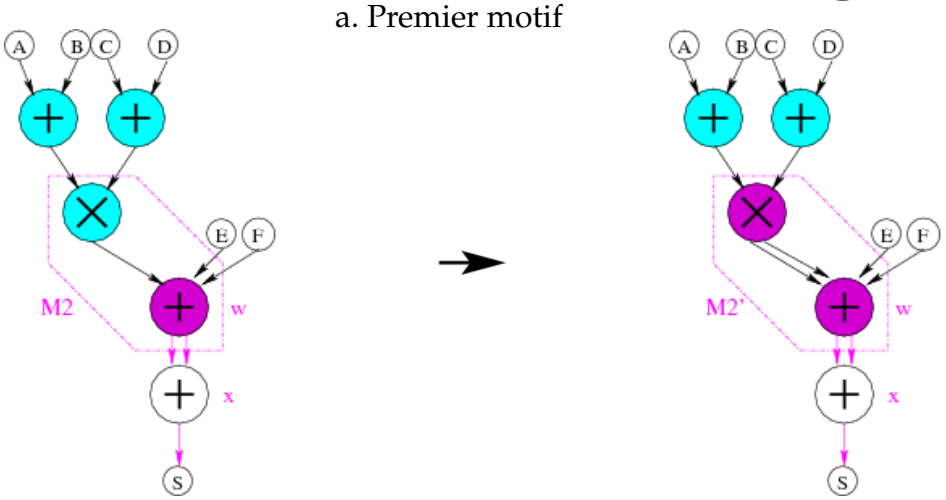
La méthode de reconnaissance de motifs pose problème dans le cas des *DAG* dans lesquels la sortie d'un opérateur peut être connectée à plusieurs autres opérateurs.

Comme on peut le voir dans la Figure 4.15, l'application d'un motif peut dans ce cas faire perdre une information utile. En effet, si l'on met de côté le fait qu'un additionneur a une sortie à la fois connectée à un autre additionneur et étant une sortie du chemin de données, le motif pouvant être appliqué à la sortie $s2$ est celui présenté dans la Figure 4.6 (cf. Figure 4.15.a). Appliquer ce motif fait que l'additionneur ayant $s1$ comme sortie est absorbé par un additionneur redondant additionnant $temp1$, $temp2$, E et F. La sortie $S1$ (somme de $temp1$ et $temp2$) est donc "perdue" (cf. Figure 4.15.b).

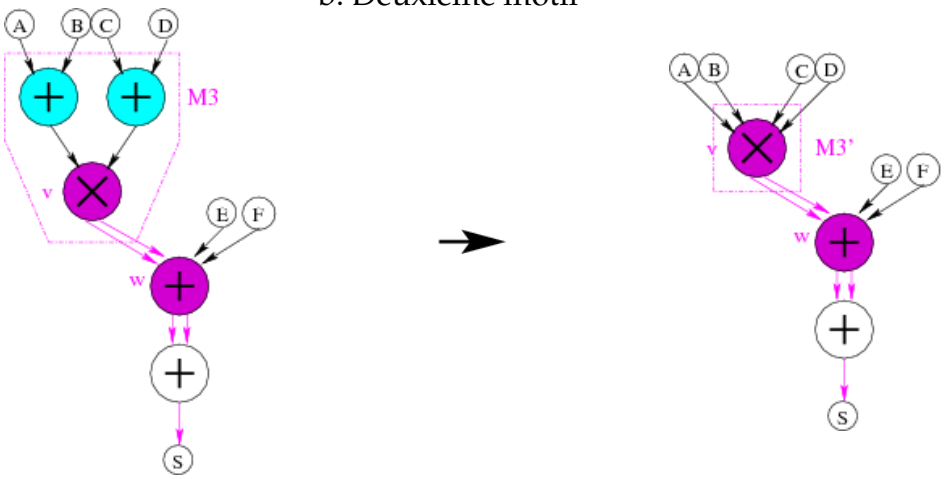
Nous avons donc dû adapter notre méthode de façon à tenir compte de ce problème. Nous avons mis en place trois façons de procéder dans un cas comme celui-ci, comme montré dans la Figure 4.16 :



a. Premier motif

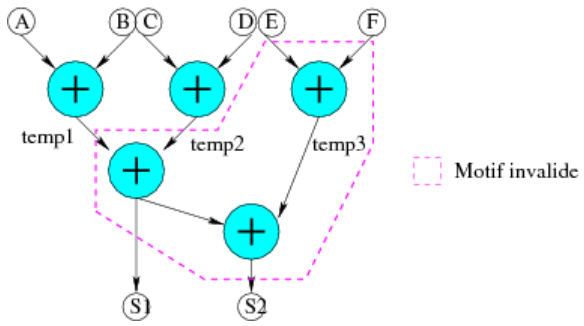


b. Deuxième motif

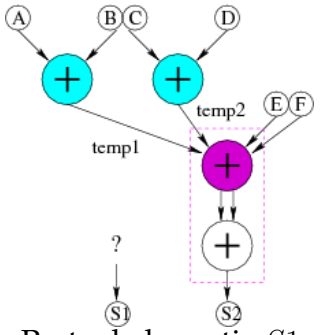


c. Troisième motif

FIG. 4.14 – Déroulement de la reconnaissance de motifs



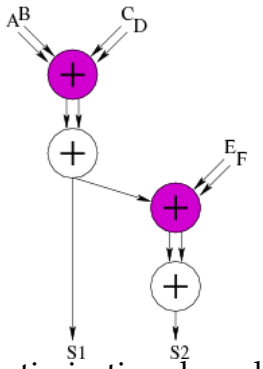
a. Règle applicable théoriquement à partir de l'arc de la sortie $S2$



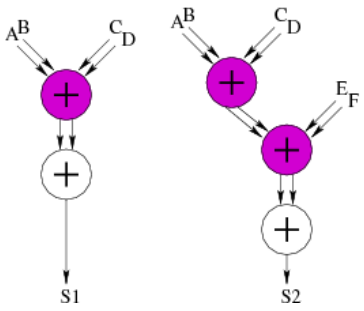
b. Perte de la sortie $S1$ si la règle est appliquée

FIG. 4.15 – Reconnaissance de motifs : Problème des sorties multiples

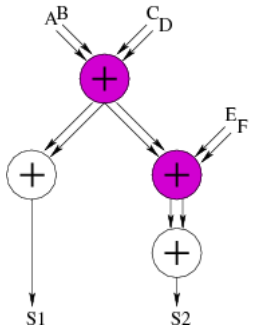
- **optimisation des arbres séparément** : pas d'application de motif dans le cas où cela génèrerait de la perte d'information (cf. Figure 4.16.a),
- **optimisation sans partage de ressources** : dédoublement de tous les opérateurs nécessaires de façon à pouvoir appliquer le motif voulu sans subir de perte d'information (cf. Figure 4.16.b),
- **optimisation avec partage de ressources** : étape précédente suivie d'une phase de fusion entre opérateurs effectuant la même opération (cf. Figure 4.16.c).



a. Optimisation des arbres séparément



b. Optimisation sans partage de ressources



c. Optimisation avec partage de ressources

FIG. 4.16 – Reconnaissance de motifs : Prise en compte des sorties multiples

Ces trois façons de procéder sont inspirées des travaux de Taewhan Kim [KK01], elles ont été présentées dans la Figure 3.5. La troisième solution est celle qui nécessite le plus de temps de calcul pour l'outil d'optimisation. En effet, comme nous venons de le définir, elle n'est pas applicable directement à la reconnaissance de motifs, et se fait donc en deux étapes, l'optimisation sans partage des ressources suivie d'une phase de fusion des opérateurs "en double".

C'est cependant celle qui est la plus optimale en terme de résultat, car conduisant au meilleur compromis surface/temps.

4.3 Algorithmes de labellisation

Nous avons ensuite développé deux algorithmes dits *de labellisation*. En effet, nous sommes parti du principe que la reconnaissance de motifs était adaptée aux arbres, mais que son adaptation aux *DAG* ainsi qu'aux architectures Borrow-Save était coûteuse.

Nous nous sommes donc tournés vers des algorithmes basés sur des techniques d'optimisation de graphes qui (toujours à partir d'une description structurelle d'un chemin de données) ne modifient pas la structure du graphe mais uniquement la représentation des arcs, et donc, l'architecture des opérateurs.

Dans cette partie, nous commençons par définir la fonction d'allocation qui permet de déterminer la représentation des arcs. Nous présentons ensuite les deux algorithmes développés. Comme précédemment pour la reconnaissance de motifs, nous présentons d'abord les algorithmes applicables aux arborescences, et montrons ensuite comment les adapter aux *DAG*.

4.3.1 Fonction d'allocation

Nous avons défini une fonction d'allocation de façon à distinguer les arcs correspondant à des signaux en représentation redondante de ceux correspondant à des signaux en représentation classique. Les raisons pour lesquelles un signal ne peut pas être mis en représentation redondante sont :

1. c'est une entrée/sortie du circuit,
2. c'est une entrée/sortie d'un opérateur non arithmétique.

Définition 4.2.

Une allocation est une fonction $a : \mathcal{A} \rightarrow \{0, 1\}$ telle que,

1. $\forall (x, y) \in \mathcal{A} - \mathcal{A}_a, a(x, y) = 0$;
2. $\forall (x, y) \in \mathcal{A}_a, a(x, y) = 1$ si (x, y) est en représentation redondante, $a(x, y) = 0$ sinon.

Notation :
Pour toute allocation a, l'ensemble des arcs en représentation redondante est

$$\mathcal{R}(a) = \{(x, y) \in \mathcal{A}_a, a(x, y) = 1\}.$$

4.3.2 Algorithme *redondant dès que possible*

L'algorithme *redondant dès que possible* passe tous les arcs de $\mathcal{A}_a$ en représentation redondante (d'où $\mathcal{R}(a) = \mathcal{A}_a$).

Cet algorithme, décrit ci-dessous, peut être défini comme suit :

A partir d'une arborescence τ représentant un chemin de données arithmétique, les arcs en représentation classique sont transformés dès que possible en leur version redondante, tout en préservant la fonctionnalité du circuit.

Algorithm 4.2 Algorithme de labellisation, redondant dès que possible

Entrée : arborescence τ

1: **pour tout** $(x, y) \in \mathcal{A}$ **faire**
2: **si** $x \in \mathcal{A}_a$ **alors**
3: $a(x, y) = 1$
4: **fin si**
5: **fin pour**

4.3.3 Obtention de l'architecture à partir du graphe

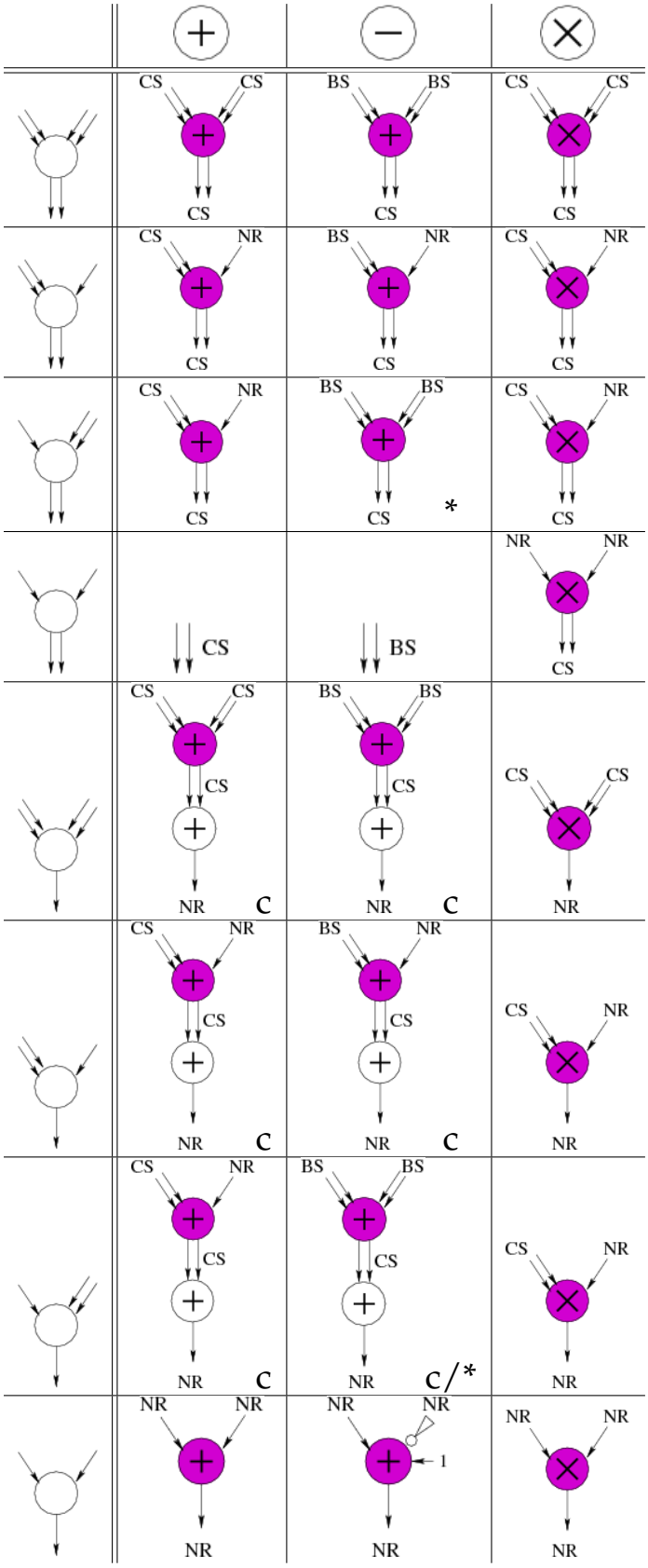
Pour que cette méthode soit viable, il faut qu'une architecture existe pour tous les cas possibles de connexions et d'opérations. Les différentes connexions possibles sont présentées dans le Tableau 4.1 avec les architectures correspondantes.

Comme on peut le voir, les trois architectures existantes pour chaque opération (l'architecture classique, la mixte et la redondante) suffisent à prendre en compte tous les cas. Une conversion $CS \rightarrow NR$ est tout simplement effectuée si la sortie d'un opérateur redondant doit être non redondante et que ce cas n'est pas pris en compte par l'opérateur (cas marqués par la lettre c).

Notons cependant deux cas qui ne sont pas une correspondance 1 pour 1 de la connexion vers l'architecture (les deux cas marqués d'une $*$). Dans ces deux cas l'opération à effectuer est $A - B$ avec A en représentation classique et B en représentation Carry-Save i.e. $A - (B^0 + B^1) = A - B^0 - B^1$. Cette opération n'est pas réalisable avec un opérateur mixte, on utilise donc un opérateur redondant avec entrées en représentation Borrow-Save en connectant les deux membres du nombre B aux termes négatifs des deux nombres Borrow-Save et en ajoutant un terme 0 comme quatrième membre à additionner : $(A - B^0) + (0 - B^1)$.

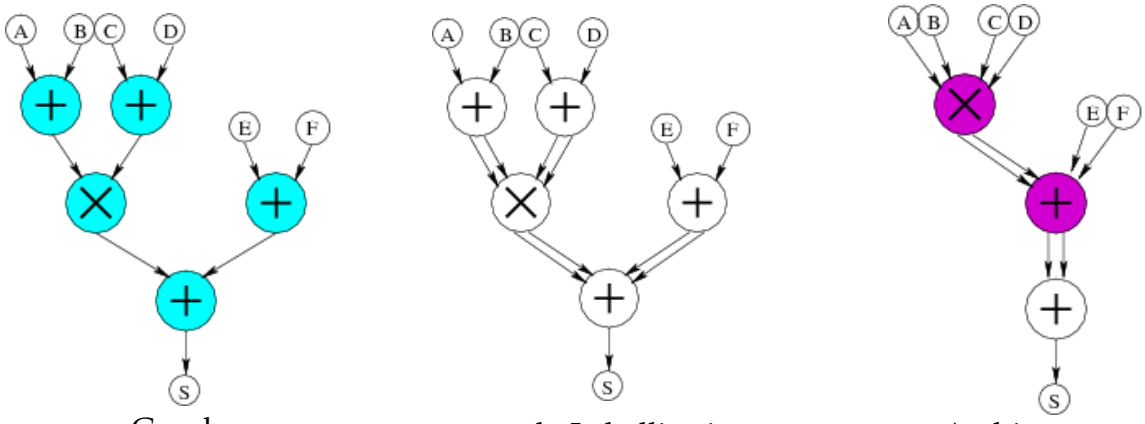
4.3.4 Exemple de la labellisation *redondant dès que possible*

Dans la Figure 4.17 est présenté un exemple du déroulement de l'algorithme de labellisation *redondant dès que possible* :



TAB. 4.1 – Architectures

- le circuit à optimiser est présenté dans la Figure 4.17.a,
- tous les arcs possibles sont labellisés en redondants, i.e. tous les arcs étant connectés à deux opérateurs redondants (cf. Figure 4.17.b),
- l'obtention de l'architecture à partir du graphe optimisé est réalisée selon les correspondances montrées dans le Tableau 4.1 (cf. Figure 4.17.c).



a. Graphe b. Labellisation c. Architecture

FIG. 4.17 – Déroulement de la labellisation redondant dès que possible

4.3.5 Fonction d'évaluation d'un nœud

De façon à trouver l'allocation optimale pour les différents opérateurs arithmétiques d'un circuit, il nous faut pouvoir estimer le coût d'une allocation en terme de temps de propagation. Pour cela, nous définissons une fonction d'évaluation d'un nœud x, notée $\mathcal{E}$, comme suit :

Définition 4.3.

Soit a une fonction d'allocation comme défini dans la Définition 4.2.
Une fonction d'évaluation d'un nœud x est une fonction $\mathcal{E} : N \rightarrow \mathbb{R}_+$ telle que,

1. si $x \notin N_a$, alors x ne peut pas être optimisé et son évaluation est constante : $\mathcal{E}(x) = 0$,
2. si $x \in N_a$, l'évaluation de x dépend de l'architecture de x et donc de la représentation des deux arcs prédécesseurs de x (notés $(w1, x)$ et $(w2, x)$) et de l'arc successeur de x (noté (x, y)) suivant les cas de la Figure 4.18 :
   (a) si $a(x, y) = 1$:
      i. si $(a(w1, x) = 1$ et $a(w2, x) = 1 : \mathcal{E}(x)$ est l'évaluation de l'architecture (a),
      ii. si $(a(w1, x) = 0$ et $a(w2, x) = 1 : \mathcal{E}(x)$ est l'évaluation de l'architecture (b1),

iii. si $(a(w1,x) = 1$ et $a(w2,x) = 0 : \mathcal{E}(x)$ est l'évaluation de l'architecture (b2),

iv. si $(a(w1,x) = 0$ et $a(w2,x) = 0 : \mathcal{E}(x)$ est l'évaluation de l'architecture (c),

(b) si $a(x,y) = 0$:

i. si $(a(w1,x) = 1$ et $a(w2,x) = 1 : \mathcal{E}(x)$ est l'évaluation de l'architecture (d),

ii. si $(a(w1,x) = 0$ et $a(w2,x) = 1 : \mathcal{E}(x)$ est l'évaluation de l'architecture (e1),

iii. si $(a(w1,x) = 1$ et $a(w2,x) = 0 : \mathcal{E}(x)$ est l'évaluation de l'architecture (e2),

iv. si $(a(w1,x) = 0$ et $a(w2,x) = 0 : \mathcal{E}(x)$ est l'évaluation de l'architecture (f).

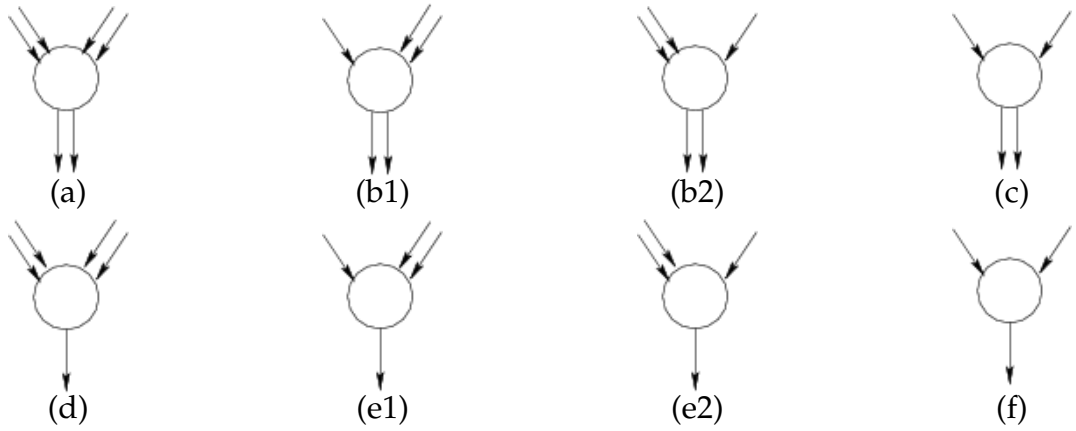


FIG. 4.18 – Différents cas de connexion

Remarque :
Etant donné que, pour le cas des opérateurs arithmétiques, l'évaluation d'un nœud dépend de l'architecture de l'opérateur correspondant, nous noterons pour plus de lisibilité $\mathcal{E}_a(x)$, $\mathcal{E}_{b1}(x)$, $\mathcal{E}_{b2}(x)$, $\mathcal{E}_c(x)$ ... en fonction du cas traité.

Dans le cas des opérateurs arithmétiques, l'évaluation d'un nœud dépend donc de l'architecture correspondante. Il nous faut donc évaluer chacune des architectures.

Plusieurs choix sont possibles pour cette évaluation, ayant différents niveaux de précision. La façon la plus simple d'évaluer les architectures est de considérer leur complexité. Cependant, cette solution nous a paru trop peu précise. La façon la plus précise est d'obtenir le temps de propagation de chaque opérateur après placement/routage. Cependant, cette solution nous a paru trop coûteuse en terme

de temps de calcul, et peut ne pas être significative étant donné que le temps obtenu aurait été fortement dépendant du contexte du placement/routage de l'opérateur seul.

Nous avons donc décidé d'évaluer les différentes architectures en créant des fonctions d'évaluation effectuant la somme des temps de traversée (notés $\mathcal{T}$) de chaque couche de portes les constituant, en fonction du nombre de bits N des signaux d'entrée (nous noterons donc $\mathcal{E}_f(+, N)$, par exemple, pour l'évaluation d'un additionneur classique sur N bits).

Les temps de traversée des portes sont définis pour chaque bibliothèque cible. Les fonctions d'évaluation de chaque opérateur sont obtenues à partir de ces temps de base. Ces fonctions sont répertoriées dans les Tableaux 4.2 et 4.3, en fonction de la taille des nombres (notée N).

Opération	Coût correspondant
NR op $NR \rightarrow NR$	$max(\mathcal{T}(xr2), \mathcal{T}(a2)) + (\mathcal{T}(a2) + \mathcal{T}(o2)) * \lfloor log_2(N) \rfloor + \mathcal{T}(xr2)$
NR op $CS \rightarrow CS$	$\mathcal{T}(FA)$
NR op $BS \rightarrow CS$	$\mathcal{T}(inv) + \mathcal{T}(FA)$
NR op $BS \rightarrow BS$	$\mathcal{T}(PPM)$
CS op $CS \rightarrow CS$	$2 * \mathcal{T}(FA)$
CS op $BS \rightarrow CS$	$2 * \mathcal{T}(FA)$
CS op $BS \rightarrow BS$	$2 * \mathcal{T}(PPM)$
BS op $BS \rightarrow CS$	$\mathcal{T}(inv) + 2 * \mathcal{T}(FA)$
BS op $BS \rightarrow BS$	$2 * \mathcal{T}(PPM)$

TAB. 4.2 – Fonctions d'évaluation des additionneurs

Opération	Coût correspondant
NR op $NR \rightarrow NR$	$\mathcal{T}(pp) + \mathcal{T}(FA) * \lfloor log_{1,5}(N/2) \rfloor + \mathcal{E}_f(+, 2 * N)$
R op $NR \rightarrow CS$	$\mathcal{T}(pp) + \mathcal{T}(FA) * \lfloor log_{1,5}(N+1) \rfloor$
R op $R \rightarrow CS$	$\mathcal{T}(pp) + \mathcal{T}(FA) * \lfloor log_{1,5}(N+1) \rfloor$

TAB. 4.3 – Fonctions d'évaluation des multiplieurs

La mise en place de ces fonctions nécessite de maîtriser les architectures de chaque opérateur utilisé, mais il nous a paru plus significatif que d'utiliser simplement la complexité des opérateurs.

Remarque :

Nous ne sommes pas rentré dans les détails pour les architectures de multiplieurs. Pour chaque architecture, l'évaluation des produits partiels diffère, étant par contre toujours une valeur constante, indépendante du nombre de bits.

4.3.6 Fonction de coût d'un arc

Après avoir défini l'évaluation d'un nœud, définissons le coût d'un arc (x, y), noté $\mathcal{C}$, comme suit :

Définition 4.4.

Soit a une fonction d'allocation comme défini dans la Définition 4.2.
Soit un arc (x, y).
Soit w_0 et w_1 les deux nœuds prédécesseurs de x, et $\mathcal{E}(x)$, l'évaluation du nœud x.
Une fonction de coût de (x, y) est une fonction $\mathcal{C} : \mathcal{A}, \{0, 1\} \rightarrow \mathbb{R}_+$ calculée de façon récursive comme suit :

$$\mathcal{C}((x, y), a) = max[\mathcal{C}((w_0, x), a), \mathcal{C}((w_1, x), a)] + \mathcal{E}(x)$$

Le cas de base de la récursion étant si x est un nœud d'entrée de l'arbre.
Dans ce cas : $\mathcal{C}((x, y), 0)$ = 0 et $\mathcal{C}((x, y), 1)$ est indéfini.

4.3.7 Algorithme d'allocation optimale

L'algorithme *d'allocation optimale* a pour but de minimiser la fonction de coût en utilisant une technique de programmation dynamique. Pour cela, il construit de façon récursive l'ensemble des signaux devant être mis en représentation redondante de façon à minimiser la fonction de coût.

Soit $e = (x, y) \in \mathcal{A}$ un arc de l'arbre τ, et $\tau(x)$ un sous-arbre de τ ayant pour racine y. Nous définissons $\mathcal{O}(e, 1)$ l'ensemble des arcs de $\tau(x)$ en représentation redondante pour une solution optimale du graphe $\tau(x)$ avec la contrainte que $a(e) = 1$. De même, $\mathcal{O}(e, 0)$ est l'ensemble des arcs de $\tau(x)$ en représentation redondante pour une solution optimale du graphe $\tau(x)$ avec la contrainte que $a(e) = 0$.

Cet algorithme peut être résumé comme suit :

- *si le nœud x n'est pas un opérateur arithmétique :*
  - *chaque fonction d'allocation vérifie $a(e) = 0$ et donc $\mathcal{O}(e, 1)$ n'est pas défini,*
  - *si x est un nœud d'entrée du graphe, $\mathcal{O}(e, 0)$ est l'ensemble vide, sinon, $\forall x \in \Gamma^-(x)$, (w, x) doit être en représentation classique et $\mathcal{O}(e, 0)$ est l'union des différents ensembles : $\mathcal{O}(e, 0) = \bigcup_{w \in \Gamma^-(x)} \mathcal{O}((w, x), 0)$,*
- *sinon, le nœud x est un opérateur arithmétique, et a donc deux prédecesseurs directs, notés w_0 et w_1.*
  - *pour calculer $\mathcal{O}(e, 0)$, nous fixons $a(e) = 0$ i.e. la sortie de x en représentation classique. $\mathcal{O}(e, 0)$ est alors, parmis les quatre possibilités suivantes, celle correspondant au coût minimum de (x, y) (calculé avec l'évaluation de x) :*
    - *les arcs (w_0, x) et (w_1, x) sont en représentation redondante (cas (d) de la Figure 4.18),*

  - *l'arc (w_0, x) est en représentation classique et l'arc (w_1, x) en représentation redondante (cas (e1) de la Figure 4.18),*
  - *l'arc (w_0, x) est en représentation redondante et l'arc (w_1, x) en représentation classique (cas (e2) de la Figure 4.18),*
  - *les arcs (w_0, x) et (w_1, x) sont tous deux en représentation classique (cas (f) de la Figure 4.18),*
- *$\mathcal{O}(e, 1)$ est calculé de la même façon en considérant que la sortie de x est en représentation redondante, i.e. en considérant les quatre possibilités des cas (a), (b1), (b2) et (c) de la Figure 4.18.*

Nous présentons ci-après l'algorithme d'allocation optimal.

L'algorithme calculant $\mathcal{O}((x, y), 0)$ est à appliquer à y, la racine de l'arbre τ. En effet, étant donné que l'arc racine d'un arbre est le prédécesseur direct du nœud de sortie, cet arc ne peut être mis sous forme redondante. La solution optimale d'un arbre τ est donc $\mathcal{O}((x, y), 0)$ avec y la racine de τ.

Dans l'algorithme présenté, les notations O_1, O_2, O_3 et O_4 correspondent aux différents ensembles d'arcs en représentation redondante, en fonction des quatre possibilités de connexions.

De la même façon, $\mathcal{C}_1^0$, $\mathcal{C}_2^0$, $\mathcal{C}_3^0$ et $\mathcal{C}_4^0$ (resp. $\mathcal{C}_1^1$, $\mathcal{C}_2^1$, $\mathcal{C}_3^1$ et $\mathcal{C}_4^1$) correspondent aux différents coûts possibles de l'arc (x, y), si celui ci n'est pas en représentation redondante (resp. est en représentation redondante), en fonction des différents ensembles O_1, O_2, O_3 et O_4.

4.3.8 Exemple de la labellisation version optimale

Dans les Figures 4.19 à 4.24 est présenté un exemple du déroulement de l'algorithme de labellisation initialisant un circuit en redondant de façon à obtenir la meilleure chaîne longue.

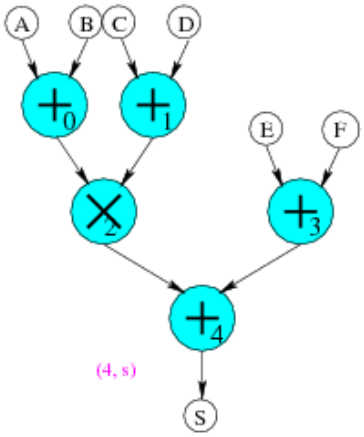
Chaque appel récursif y est détaillé, ainsi que le calcul des différents coûts et la création des ensembles de signaux redondants correspondants.

Comme nous avons déjà pu le voir précédemment, cet exemple est un cas typique dans lequel la version optimale ne va pas aboutir à la version entièrement redondante, comme montré dans la Figure 4.23.n. En effet, le temps pour effectuer la multiplication est beaucoup plus important (9367) que celui pour effectuer l'addition, même si cette dernière reste en représentation classique (4199). Le choix qui est pris est donc de garder cette addition en représentation classique (l'addition classique s'effectue en parallèle avec la multiplication) de façon à ce que l'additionneur final soit un additionneur mixte et non pas un additionneur redondant.

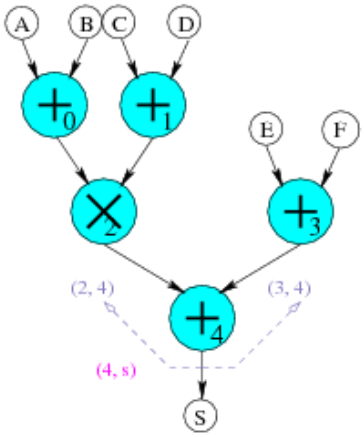
Algorithm 4.3 Algorithme de labellisation, version optimale

Entrée :
$e = (x, y) \in \mathcal{A}$ l'arc racine de l'arbre τ.

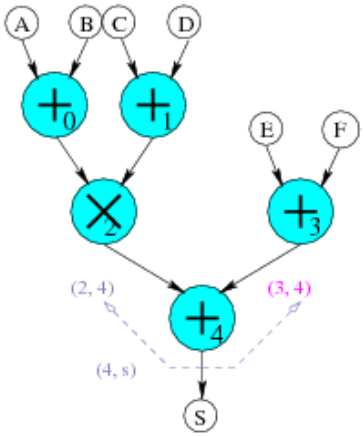
si $x \in N - N_a$ **alors**
  $\mathcal{O}(e, 1)$ n'est pas défini
  si $x \in E$ **alors**
    $\mathcal{O}(e, 0) = \emptyset$
  sinon
    $\mathcal{O}(e, 0) = \bigcup_{w \in \Gamma^-(x)} \mathcal{O}((w, x), 0)$
  fin si
sinon
  $O_1 = \mathcal{O}((w_0, x), 0) \cup \mathcal{O}((w_1, x), 0)$
  $O_2 = \mathcal{O}((w_0, x), 0) \cup \mathcal{O}((w_1, x), 1) \cup \{(w_1, x)\}$
  $O_3 = \mathcal{O}((w_0, x), 1) \cup \mathcal{O}((w_1, x), 0) \cup \{(w_0, x)\}$
  $O_4 = \mathcal{O}((w_0, x), 1) \cup \mathcal{O}((w_1, x), 1) \cup \{(w_0, x), (w_1, x)\}$
  $\mathcal{C}_1^1((x, y), 1) = max[\mathcal{C}((w_0, x), 0), \mathcal{C}((w_1, x), 0)] + \mathcal{E}_a(x)$
  $\mathcal{C}_2^1((x, y), 1) = max[\mathcal{C}((w_0, x), 0), \mathcal{C}((w_1, x), 1)] + \mathcal{E}_{b1}(x)$
  $\mathcal{C}_3^1((x, y), 1) = max[\mathcal{C}((w_0, x), 1), \mathcal{C}((w_1, x), 0)] + \mathcal{E}_{b2}(x)$
  $\mathcal{C}_4^1((x, y), 1) = max[\mathcal{C}((w_0, x), 1), \mathcal{C}((w_1, x), 1)] + \mathcal{E}_c(x)$
  $\mathcal{C}(e, 1) = \min_{i \in \{1,2,3,4\}}[\mathcal{C}_i^0(e, 1)]$
  si $\mathcal{C}(e, 1) = \mathcal{C}_1^1$ **alors**
    $\mathcal{O}(e, 1) = O_1$
  sinon si $\mathcal{C}(e, 1) = \mathcal{C}_2^1$ **alors**
    $\mathcal{O}(e, 1) = O_2$
  sinon si $\mathcal{C}(e, 1) = \mathcal{C}_3^1$ **alors**
    $\mathcal{O}(e, 1) = O_3$
  sinon si $\mathcal{C}(e, 1) = \mathcal{C}_4^1$ **alors**
    $\mathcal{O}(e, 1) = O_4$
  fin si
  $\mathcal{C}_1^0((x, y), 1) = max[\mathcal{C}((w_0, x), 0), \mathcal{C}((w_1, x), 0)] + \mathcal{E}_d(x)$
  $\mathcal{C}_2^0((x, y), 1) = max[\mathcal{C}((w_0, x), 0), \mathcal{C}((w_1, x), 1)] + \mathcal{E}_{e1}(x)$
  $\mathcal{C}_3^0((x, y), 1) = max[\mathcal{C}((w_0, x), 1), \mathcal{C}((w_1, x), 0)] + \mathcal{E}_{e2}(x)$
  $\mathcal{C}_4^0((x, y), 1) = max[\mathcal{C}((w_0, x), 1), \mathcal{C}((w_1, x), 1)] + \mathcal{E}_f(x)$
  $\mathcal{C}(e, 0) = \min_{i \in \{1,2,3,4\}}[\mathcal{C}_i^0(e, 1)]$
  si $\mathcal{C}(e, 0) = \mathcal{C}_1^0$ **alors**
    $\mathcal{O}(e, 0) = O_1$
  sinon si $\mathcal{C}(e, 0) = \mathcal{C}_2^0$ **alors**
    $\mathcal{O}(e, 0) = O_2$
  sinon si $\mathcal{C}(e, 0) = \mathcal{C}_3^0$ **alors**
    $\mathcal{O}(e, 0) = O_3$
  sinon si $\mathcal{C}(e, 0) = \mathcal{C}_4^0$ **alors**
    $\mathcal{O}(e, 0) = O_4$
  fin si
fin si



a. Calcul de $\mathcal{O}((4, s), 0)$

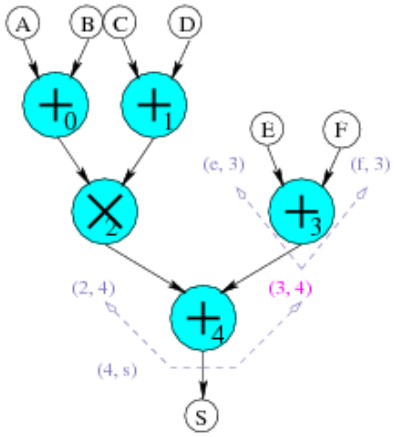


b. Appel récursif sur $(2, 4)$ et $(3, 4)$

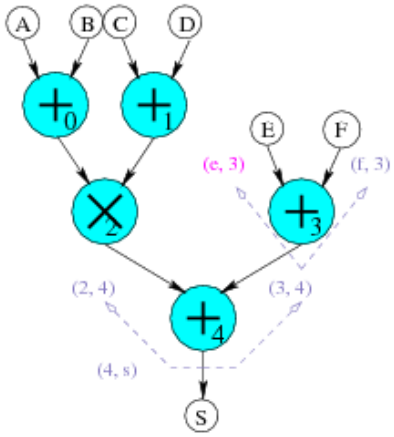


c. Calculs de $\mathcal{O}((3, 4), 0)$ et $\mathcal{O}((3, 4), 1)$

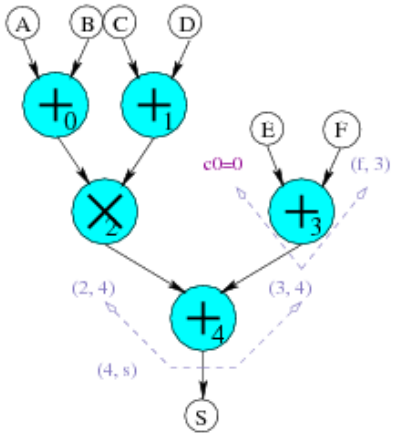
FIG. 4.19 – Déroulement de la labellisation temps optimal 1



d. Appel récursif sur $(E, 3)$ et $(F, 3)$

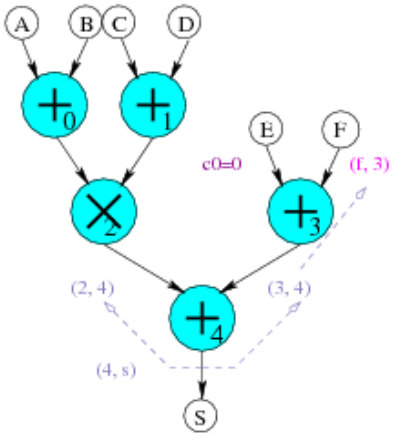


e. Calcul de $\mathcal{O}((E, 3), 0)$

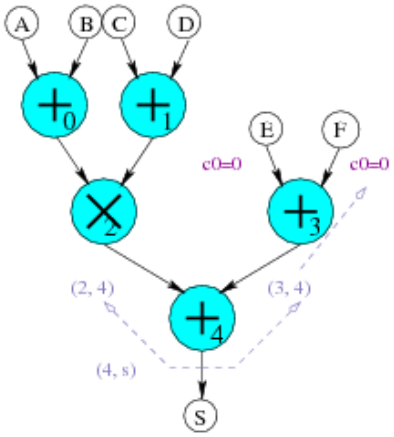


f. Résultat de $(E, 3)$:
$\rightarrow C^0 = 0, \mathcal{O}((E, 3), 0) = \emptyset$
$\rightarrow C^1 = ND, \mathcal{O}((E, 3), 1) = ND$

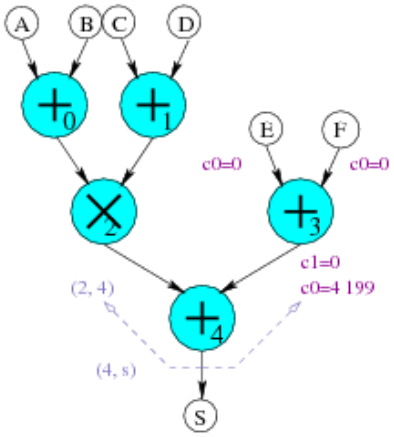
FIG. 4.20 – Déroulement de la labellisation temps optimal 2



g. Calcul de $\mathcal{O}((F,3),0)$

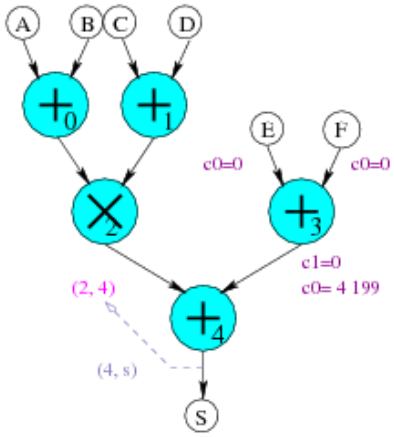


h. Résultat de $(F,3)$:
$\rightarrow C^0 = 0, \mathcal{O}((F,3),0) = \emptyset$
$\rightarrow C^1 = ND, \mathcal{O}((F,3),1) = ND$

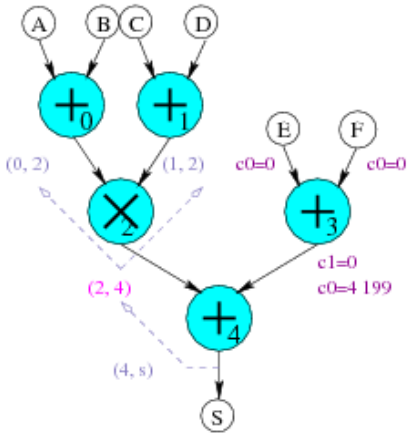


i. Résultat de $(3,4)$:
$O_1 = \mathcal{O}((E,3),0) \cup \mathcal{O}((F,3),0) = \emptyset,$
$O_2 = \mathcal{O}((E,3),0) \cup \mathcal{O}((F,3),1) \cup \{(F,3)\} = ND,$
$O_3 = \mathcal{O}((E,3),1) \cup \mathcal{O}((F,3),0) \cup \{(E,3)\} = ND,$
$O_4 = \mathcal{O}((E,3),1) \cup \mathcal{O}((F,3),1) \cup \{(E,3),(F,3)\} = ND,$
$\rightarrow C^0 = C_1^0 = max(0,0) + 4199 = 4199$
$\rightarrow C^1 = C_1^1 = max(0,0) + 0 = 0$
$\rightarrow \mathcal{O}((3,4),0) = O_1 = \emptyset$
$\rightarrow \mathcal{O}((3,4),1) = O_1 = \emptyset$

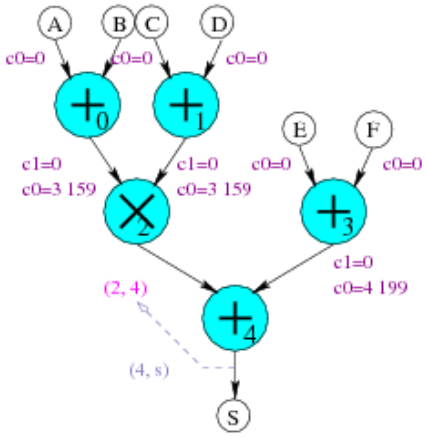
FIG. 4.21 – Déroulement de la labellisation temps optimal 3



j. Calcul $\mathcal{O}((2,4),0)$

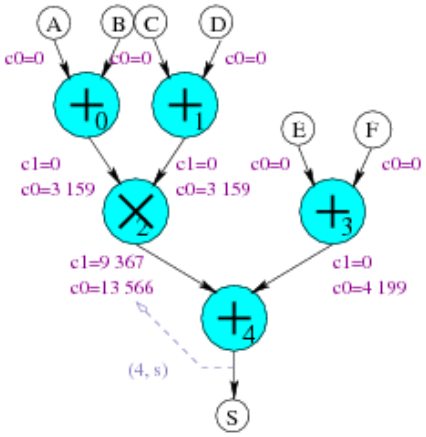


k. Appel récursif sur $(0,2)$ et $(0,1)$



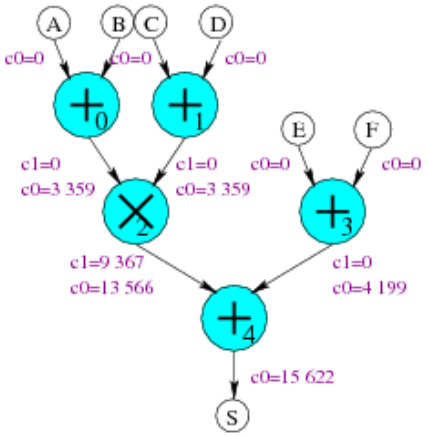
l. De la même façon que pour les étapes e à i, résultats de $(0,2)$ et $(0,1)$:
$\rightarrow C^0 = max(0,0) + 3359 = 3359$
$\rightarrow C^1 = max(0,0) + 0 = 0$
$\rightarrow \mathcal{O}((0,2),0) = \emptyset$
$\rightarrow \mathcal{O}((0,2),1) = \emptyset$
$\rightarrow \mathcal{O}((0,1),0) = \emptyset$
$\rightarrow \mathcal{O}((0,1),1) = \emptyset$

FIG. 4.22 – Déroulement de la labellisation temps optimal 4



m. Résultat de $(2,4)$

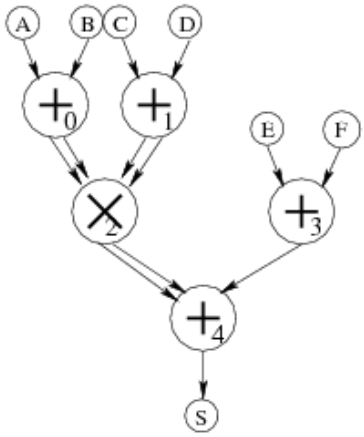
$O_1 = \mathcal{O}((0,2),0) \cup \mathcal{O}((1,2),0)$,
$O_2 = \mathcal{O}((0,2),0) \cup \mathcal{O}((1,2),1) \cup \{(1,2)\}$,
$O_3 = \mathcal{O}((0,2),1) \cup \mathcal{O}((1,2),0) \cup \{(0,2)\}$,
$O_4 =$
$\mathcal{O}((0,2),1) \cup \mathcal{O}((1,2),1) \cup \{(0,2),(1,2)\}$,
$C_1^0 = max(3359, 3359) + 11360 = 14719$
$C_2^0 = max(0, 3359) + 12425 = 15784$
$C_3^0 = max(3359, 0) + 12425 = 15784$
$C_4^0 = max(0, 0) + 13566 = 13566$
$\rightarrow C^0 = min(C_1^0, C_2^0, C_3^0, C_4^0) = C_4^0$
$\rightarrow \mathcal{O}((2,4),0) = O_4 = \{(0,2),(1,2)\}$
$C_1^1 = max(3359, 3359) + 7161 = 10520$
$C_2^1 = max(0, 3359) + 8226 = 11585$
$C_3^1 = max(3359, 0) + 8226 = 11585$
$C_4^1 = max(0, 0) + 9367 = 9367$
$\rightarrow C^1 = min(C_1^1, C_2^1, C_3^1, C_4^1) = C_4^1$
$\rightarrow \mathcal{O}((2,4),1) = O_4 = \{(0,2),(1,2)\}$



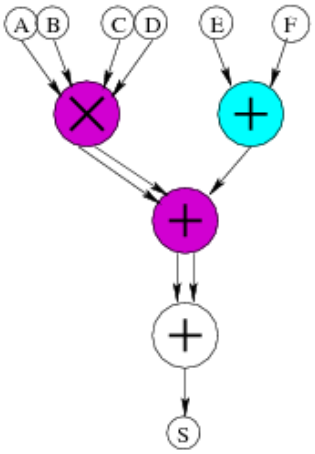
n. Résultat $(4,s)$

$O_1 = \mathcal{O}((2,4),0) \cup \mathcal{O}((3,4),0)$,
$O_2 = \mathcal{O}((2,4),0) \cup \mathcal{O}((3,4),1) \cup \{(3,4)\}$,
$O_3 = \mathcal{O}((2,4),1) \cup \mathcal{O}((3,4),0) \cup \{(2,4)\}$,
$O_4 =$
$\mathcal{O}((2,4),1) \cup \mathcal{O}((3,4),1) \cup \{(2,4),(3,4)\}$,
$C_1^0 = max(13566, 4199) + 4199 = 17765$
$C_2^0 = max(13566, 0) + 6255 = 19821$
$C_3^0 = max(9367, 4199) + 6255 = 15622$
$C_4^0 = max(9367, 0) + 7506 = 16873$
$\rightarrow C^0 = min(C_1^0, C_2^0, C_3^0, C_4^0) = C_3^0$
$\rightarrow \mathcal{O}((4,s),0) = O_3 =$
$\{(0,2),(1,2),(2,4)\}$

FIG. 4.23 – Déroulement de la labellisation temps optimal 5



o. Labellisation finale : Les arcs devant être mis en représentation redondante sont :
$\mathcal{O}((4, s), 0) = \{(0, 2), (1, 2), (2, 4)\}$



p. Architecture

FIG. 4.24 – Déroulement de la labellisation temps optimal 6

4.3.9 Prise en compte des sorties multiples

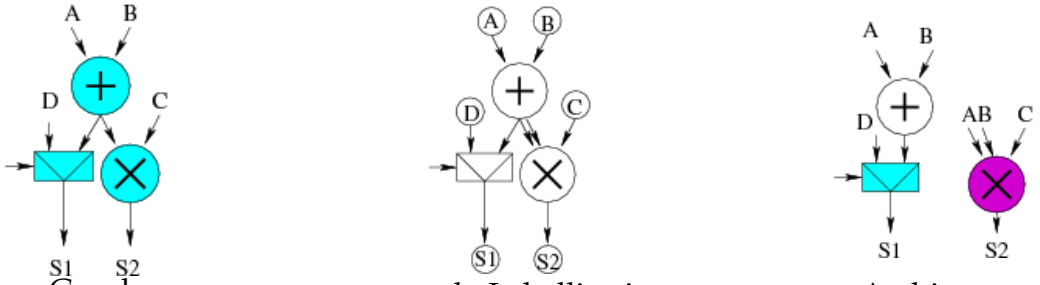
Algorithme *redondant dès que possible*

En ce qui concerne l'algorithme *redondant dès que possible*, la prise en compte des sorties multiples est on ne peut plus simple.

Si la sortie d'un opérateur est connectée à plusieurs opérateurs, dont deux sont de type différents, comme montré dans la Figure 4.25.a, chacun des arcs successeurs directs de cet opérateur sera labellisé indépendamment des autres, comme cela est représenté dans la Figure 4.25.b.

A partir du graphe ainsi optimisé, l'obtention de l'architecture se fait comme suit (cf. Figure 4.25.c) :

1. l'opérateur est utilisé avec sortie Carry-Save (dans le cas présent où l'opérateur est un additionneur, il est donc *effacé* et un nombre CS est utilisé),
2. un convertisseur $CS \rightarrow NR$ est utilisé de façon à obtenir une version NR de ce signal,
3. tous les opérateurs arithmétiques (resp. non arithmétiques) ayant pour entrée ce signal sont connectés "à la version redondante" (resp. "non redondante") du signal.



a. Graphe b. Labellisation c. Architecture

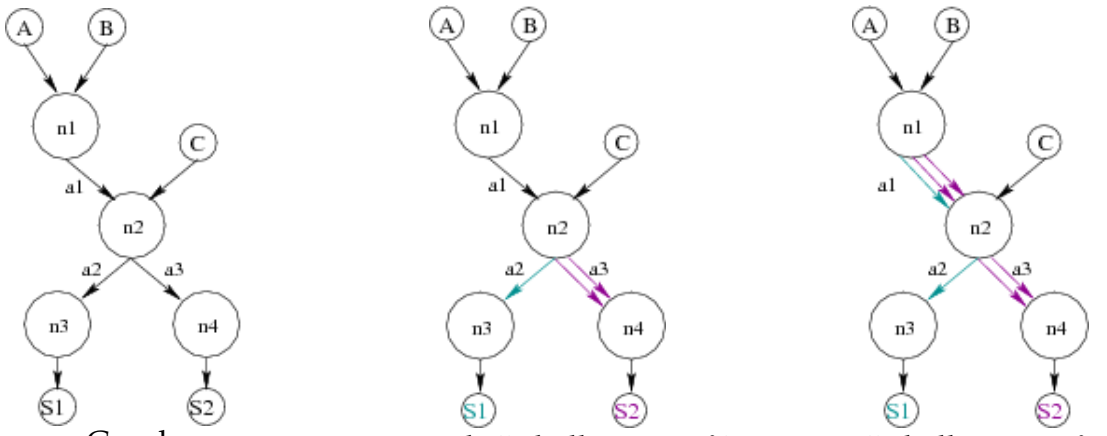
FIG. 4.25 – Algorithme de labellisation *redondant dès que possible* : Prise en compte des sorties multiples

Ce mécanisme, aisé à mettre en place, correspond au principe de *dédoublement de la donnée* présenté dans le Chapitre 1 et donne comme solution l'équivalent de *l'optimisation avec partage des ressources* déjà présentée, soit la meilleure solution existante pour prendre en compte les sorties multiples. Il permet donc d'adapter l'algorithme *redondant dès que possible* aux *DAG*. Il suffit alors d'appliquer l'algorithme à tous les arcs prédecesseurs directs de nœuds de sortie des *DAG*.

Algorithme d'allocation optimale

L'application de l'algorithme cherchant la solution optimale à des sorties multiples peut engendrer des problèmes si deux choix de labellisation différents peuvent être fait sur un même arc.

Nous commençons ici par expliciter plus en détail le problème posé. Nous montrerons ensuite que c'est un cas qui ne peut exister, et qu'ainsi la solution à mettre en place pour l'algorithme d'allocation optimale est la même que pour l'algorithme redondant dès que possible.



a. Graphe b. Labellisation n°1 c. Labellisation n°2

FIG. 4.26 – Algorithme de labellisation *allocation optimale* : Prise en compte des sorties multiples

La Figure 4.26 fait un récapitulatif des deux cas de labellisation pouvant être faits dans le cas d'une sortie multiple. A partir du graphe présenté dans la Figure 4.26.a :

- les deux arcs successeurs du nœud `n2` peuvent avoir une labellisation différente,
- l'un des arcs prédécesseurs du nœud `n2` peut avoir deux labellisations différentes en fonction du "nœud de départ" de la récursion.

Le premier cas est celui qui a été traité dans la section précédente (cf. Figure 4.26.b), on a donc montré qu'il n'engendrait pas de problème particulier.

Il n'a pas encore été question du second cas (cf. Figure 4.26.c), car l'algorithme redondant dès que possible ne peut générer un tel cas. En effet, les types (arithmétique ou non) des deux nœuds prédécesseur et successeur d'un arc sont immuables.

En ce qui concerne l'algorithme d'allocation optimale, ce n'est pas le type d'un nœud qui définit la labellisation d'un arc, mais la fonction de coût. On pourrait alors envisager que ce cas soit possible.

La seule cause susceptible de provoquer ce cas serait qu'un nœud successeur de l'arc ait plusieurs arcs successeurs directs labellisés de façon différente. Le choix fait lors de l'exploration de l'arc `a2` pourrait alors entrainer un choix pour l'arc `a1` qui soit différent du choix pris lors de l'exploration de l'arc `a3`.

Si un tel cas se produisait, un mécanisme coûteux serait à mettre en place quant au choix à faire pour l'architecture des opérateurs correpondants aux nœuds `n1` et `n2`. Ce mécanisme devrait mettre en place un dédoublement de ces opérateurs. Or, là où dans le premier cas, le mécanisme à mettre en place est peu coûteux et donne de bons résultats, dans ce cas-ci, il serait possible que les dédoublements engendrés par l'arc

`a1` soient propagés jusqu'aux nœuds d'entrée du *DAG*, et engendrent un gros surcoût pour la surface. Cela remettrait complètement en cause la pertinence de la méthode.

Nous allons cependant montrer que cette cause ne peut en aucun cas générer des choix distincts sur un même arc, en nous appuyant sur la définition de notre fonction de coût.

Théorème 4.1.

Le choix de labellisation d'un arc prédécesseur d'un nœud est toujours le même, quelle que soit la labellisation des différents arcs successeurs du nœud.

Preuve :
Soit le nœud x.
Soient $w1$ et $w2$ les prédécesseurs de x, $y1$ et $y2$ les successeurs de x.

Voici les valeurs que peuvent prendre les ensembles $\mathcal{O}((x, y1), 0)$ et $\mathcal{O}((x, y1), 1)$, en fonction des quatre possibilités de connexions :
$O_1 = \mathcal{O}((w1, x), 0) \cup \mathcal{O}((w2, x), 0)$,
$O_2 = \mathcal{O}((w1, x), 0) \cup \mathcal{O}((w2, x), 1) \cup \{(w2, x)\}$,
$O_3 = \mathcal{O}((w1, x), 1) \cup \mathcal{O}((w2, x), 0) \cup \{(w1, x)\}$,
$O_4 = \mathcal{O}((w1, x), 1) \cup \mathcal{O}((w2, x), 1) \cup \{(w1, x), (w2, x)\}$,
Les coûts correspondants sont les suivants, avec $(x, y1)$ non redondant :
$C_1^0 = max(\mathcal{C}((w1, x), 0), \mathcal{C}((w2, x), 0)) + \mathcal{E}_f(x)$
$C_2^0 = max(\mathcal{C}((w1, x), 0), \mathcal{C}((w2, x), 1)) + \mathcal{E}_{e1}(x)$
$C_3^0 = max(\mathcal{C}((w1, x), 1), \mathcal{C}((w2, x), 0)) + \mathcal{E}_{e2}(x)$
$C_4^0 = max(\mathcal{C}((w1, x), 1), \mathcal{C}((w2, x), 1)) + \mathcal{E}_d(x)$
$\rightarrow C^0 = min(C_1^0, C_2^0, C_3^0, C_4^0)$
De la même façon, avec $(x, y1)$ redondant :
$C_1^1 = max(\mathcal{C}((w1, x), 0), \mathcal{C}((w2, x), 0)) + \mathcal{E}_c(x)$
$C_2^1 = max(\mathcal{C}((w1, x), 0), \mathcal{C}((w2, x), 1)) + \mathcal{E}_{b1}(x)$
$C_3^1 = max(\mathcal{C}((w1, x), 1), \mathcal{C}((w2, x), 0)) + \mathcal{E}_{b2}(x)$
$C_4^1 = max(\mathcal{C}((w1, x), 1), \mathcal{C}((w2, x), 1)) + \mathcal{E}_a(x)$
$\rightarrow C^1 = min(C_1^1, C_2^1, C_3^1, C_4^1)$
Or, par définition des fonctions d'évaluation des nœuds, on a :
$\mathcal{E}_d = \mathcal{E}_a + \mathcal{E}_f(+)$
$\mathcal{E}_{e1} = \mathcal{E}_{b1} + \mathcal{E}_f(+)$
$\mathcal{E}_{e2} = \mathcal{E}_{b2} + \mathcal{E}_f(+)$
$\mathcal{E}_f = \mathcal{E}_c + \mathcal{E}_f(+)$
De cette façon, $C^0 = C_1^0 \Rightarrow C^1 = C_1^1$, $C^0 = C_2^0 \Rightarrow C^1 = C_2^1$, etc ... i.e. le choix des ensembles sera le même qu'elle que soit la labellisation de $(x, y1)$ et $(x, y2)$:
$\mathcal{O}((x, y1), 0) = \mathcal{O}((x, y1), 1) = \mathcal{O}((x, y2), 0) = \mathcal{O}((x, y2), 1)$

4.4 Conclusion

Trois algorithmes ont été présentés, le premier à base de reconnaissance de motifs, les deux autres à base de labellisation de graphe.

Graphe

Chacun des trois algorithmes sert à choisir une architecture pour chaque opérateur et une représentation pour chaque signal d'un circuit modélisé sous forme de graphe.

Une différence à noter est qu'avec la reconnaissance de motifs, la structure du graphe peut être modifiée, alors qu'elle ne l'est pas dans le cas des algorithmes de labellisation.

Sorties multiples

Les trois algorithmes présentés ont été appliqués dans un premier temps aux arbres et dans un second temps aux *DAG* (prise en compte des opérateurs avec une sortie connectée à plusieurs opérateurs, dits opérateurs à *sorties multiples*). Les deux algorithmes à base de labellisation se sont avérés être plus facilement adaptables à cette extension que celui à base de reconnaissance de motifs.

En effet, parmis les trois optimisations possibles concernant les opérateurs à sorties multiples (optimisation des arbres séparément, optimisation sans partage des ressources, optimisation avec partage des ressources, cf. Figure 3.5), ce sont les algorithmes de labellisation qui ont été adaptables le plus aisément à l'optimisation avec partage des ressources considérée comme étant la meilleure (car donnant le meilleur compromis temps / surface).

Pour la reconnaissance de motifs, c'est l'optimisation de chaque arbre séparément qui s'est la mieux prêtée à l'algorithme. Pour appliquer l'optimisation avec partage des ressources, il a au contraire fallu deux étapes de travail, la première étant l'optimisation sans partage de ressource et la seconde, la fusion des opérateurs dédoublés.

Remarque :

Mettre en œuvre ces trois optimisations a été un développement intéressant qui nous a permis de comparer les performances obtenues et de vérifier le postulat disant que l'optimisation sans partage de ressource pouvait amener à la meilleure chaîne critique.

Soustraction

Les trois algorithmes présentés fournissent au moins deux façons de prendre en compte la soustraction : avec utilisation de la représentation Carry-Save uniquement, ou avec utilisation conjointe des représentations Carry-Save et Borrow-Save.

La reconnaissance de motifs fournit deux optimisations différentes en ce qui concerne l'utilisation de la représentation Borrow-Save : une avec les architectures d'additionneurs avec sortie Carry-Save, l'autre avec les architectures avec sortie Borrow-Save. Autant il n'y a eu aucune modification à apporter à l'algorithme, autant les deux listes de motifs se sont avérées très importantes (beaucoup plus que celle n'utilisant que le Carry-Save), donc difficiles à rendre exhaustives.

En ce qui concerne la labellisation, nous avons mis en place la gestion des architectures d'additionneurs Borrow-Save avec sortie en représentation Carry-Save.

Synthèse

En conclusion, l'algorithme à base de reconnaissance de motifs, le premier à avoir été implanté, nous a servi de base pour tester toutes les options d'optimisation et évaluer les pours et les contres de chacune.

L'élaboration des algorithmes de labellisation dans un second temps nous a montré que l'algorithme à base de reconnaissance de motifs n'était cependant pas le plus direct / simple à implanter, car il était difficilement adaptable aux *DAG* et à la prise en compte de la notation Borrow-Save.

Une comparaison détaillée des performances obtenues avec les différents algorithmes est faite dans le Chapitre 6. De plus, un récapitulatif des différentes options existantes ainsi qu'un exemple d'utilisation sont présentés dans l'Annexe C.

Chapitre 5 Environnement de conception

Ce chapitre est consacré à l'environnement de conception dans lequel sont utilisés les différents outils d'optimisation arithmétiques mis en œuvre.

Nous commençons par faire une brève description de la méthodologie générale de la conception de circuits *VLSI*.

Nous faisons ensuite un récapitulatif des différents environnements de conception existants susceptibles de convenir à nos besoins. Nous nous intéressons aux différents aspects composant ces environnements : leur structure de données (représentation des circuits en mémoire), leur langage de description, ainsi que leur plate-forme de conception (ensemble des fonctionnalités et outils utilisés). Nous faisons également une analyse de leur caractéristiques de manière critique.

Nous présentons ensuite l'environnement de conception que nous avons personnellement mis en œuvre dans ce travail de thèse : **Stratus**.

Nous nous intéressons enfin plus précisément à l'aspect arithmétique lié à cet environnement, avec en particulier la bibliothèque d'opérateurs arithmétiques qui a été développée.

5.1 Introduction

La réalisation d'un circuit intégré est une opération très complexe qui demande de nombreuses connaissances et beaucoup de temps. Des méthodologies de travail et de nombreux outils d'aide à la conception existent de façon à faciliter cette opération. Différentes méthodologies existent, mais sont toujours composées de deux étapes fondamentales : la conception logique et la conception physique.

La conception logique représente la spécification et la validation du comportement fonctionnel. Elle a comme point de départ le cahier des charges du circuit à réaliser, avec, entre autres, son comportement fonctionnel. A partir de ce comportement fonctionnel, différentes étapes permettent de hiérarchiser le circuit en modules plus ou moins complexes jusqu'à obtenir une vue en portes.

La conception physique représente l'assemblage des différents composants d'un circuit : le placement des différents blocs et le routage des différentes interconnexions.

A chaque étape, l'architecture doit être validée. Plus cela est fait fréquemment, plus

une erreur peut être détectée rapidement. C'est donc une phase primordiale.

Comme nous venons de le préciser, une des étapes de conception conciste à hiérarchiser le circuit à concevoir en différents modules plus simples. Les modules dits "de base" sont des *générateurs* : des programmes informatiques permettant de générer une description structurelle d'un module en fonction d'un ensemble de paramètres. C'est ici que rentre en jeu la notion de réutilisation, primordiale lors de la conception de circuits, de façon à gagner du temps.

Dans le contexte de développement de chemins de données arithmétiques, ces modules seront dans la plupart des cas des opérateurs arithmétiques. Il parait donc intéressant (voire indispensable !) d'avoir à disposition ces différents opérateurs : addition, multiplication ... L'idée est d'avoir à disposition une bibliothèque de générateurs arithmétiques.

De nombreux outils de conception existent, basés sur la méthodologie que nous venons de décrire, chacun d'eux répondant à des exigences particulières. Il est donc nécessaire d'en déterminer les caractéritiques nous paraissant utiles, de façon à faire un choix.

Nous visons un environnement de conception fournissant :

- un langage de description adapté aux besoins de l'arithmétique (i.e. permettant, entres autres, la différenciation entre représentations arithmétiques),
- une structure de données (représentation des données en mémoire) adaptée aux besoins des outils d'optimisation arithmétique (i.e. permettant principalement d'être enrichie et modifiée de façon aisée),
- une plate-forme de conception permettant de bénéficier des différents outils transformant la description initiale en un dessin des masques,
- une bibliothèque d'opérateurs arithmétiques.

5.2 Environnements existants

5.2.1 Environnements industriels

Caractéristiques

Plusieurs environnements industriels ont en commun la caractéristique d'être organisés autour d'une structure de données centralisée. A travers cette structure, les outils interagissent et raffinent les données des circuits. Deux de ces industriels sont **Cadence** [Cad], proposant la plate-forme de conception **Encounter** et **Synopsys** [Syn], proposant la plate-forme de conception **Galaxy**. La première est plutôt axée placement/routage, et la seconde, plutôt synthèse, avec son produit phare, le synthétiseur **Design Compiler**. Cependant, cette différence tend à s'estomper car la tendance actuelle vise à intégrer le flot complet.

Dans le but de devenir la référence, **Cadence** a été la première à diffuser sa structure de données interne : **OpenAcces**. Suite à cette évolution **Synopsys** a fait le choix de donner un accès partiel à sa structure de données : **MilkyWay**.

D'un point de vu langage d'entrée, les industriels utilisaient initialement des formats de fichiers propriétaires. Il y eut ensuite la mise en place de formats standards permettant une inter-opérabilité minimale entre les différents fournisseurs. Du point de vue de la description logique, ce sont les formats *VHDL* ou Verilog qui sont dorénavant généralement utilisés.

Langage

VHDL est un langage spécialisé dédié à la description matérielle de circuits intégrés. Il est le premier langage de description de circuit à avoir été élaboré [Sha86]. C'est un langage dit HDL pour "Harware Description Langage", spécialisé (inspiré du langage Ada), qui permet de décrire différents niveaux d'abstractions, allant de l'algorithme aux portes. De nombreux autres langages spécialisés ont ensuite été développés [Mag92] avec, en particulier, Verilog [Smi96] (inspiré du langage C).

Analyse

Bien que couvrant un large spectre de niveaux d'abstractions, *VHDL* et Verilog possèdent une sémantique de simulation et n'ont pas été définis pour être directement utilisables en synthèse [Jac99]. Cela induit, entre autres, une forte verbosité de l'écriture, et une difficulté de réutilisation. Les différents outils de synthèse des plates-formes industrielles ont défini leur propre sous ensemble synthétisable, et ont donc restreint de façon importante la puissance du langage utilisé.

Comme nous avons pu le voir dans le chapitre précédent, nous nous plaçons dans un contexte fortement orienté vers la synthèse et pour lequel la réutilisation est le maître mot. Avoir à disposition un langage orienté vers la simulation et pour lequel l'utilisation de générateurs n'est pas aisée sont deux raisons suffisantes pour déterminer que ce type de langage n'est pas adapté à nos besoins.

Deuxième point non négligeable, même si ces industriels ont fait le choix de donner un accès à leur structure de données, cela avait pour but de faciliter l'interopérabilité entre industriels. L'ensemble du contenu de ces environnements demeure inaccessible au monde académique.

5.2.2 Alliance

Caractéristiques

Alliance [All, GP92] est une plate-forme de développement d'outils *CAO*et de conception de circuits intégrés mise en place au sein du laboratoire **LIP6**.

La structure de données interne d'**Alliance** est **MBK** [Pé94], qui se divise en trois sous-structures, de façon à décrire les trois vues (comportementale, logique et physique) d'un circuit intégré. **Alliance** est encore, à ce jour, l'unique plate-forme universitaire à proposer une série d'outils couvrant la quasi-totalité du flot *VLSI*, du niveau RTL au dessin des masques.

Alliance s'interface avec de nombreux formats standards, *VHDL* par exemple pour la vue logique. Elle possède de plus un langage permettant la description procédurale de générateurs de macro-blocs régulier, le langage **Genlib** [GP94].

Langage

Le langage **Genlib** est une bibliothèque de fonctions C permettant de décrire des circuits, proposant une utilisation de ce langage au niveau le plus élémentaire. Il offre ainsi un accès simplifié au langage tout en permettant à un utilisateur averti de profiter de l'ensemble de ces fonctionnalités.

Genlib a été conçu pour décrire des générateurs logiques et des générateurs physiques. Le but premier a été de permettre le développement rapide de générateurs de blocs optimisés portables. En ce sens, c'est un langage adapté à la synthèse dont les avantages résident en la séparation des vues logique et physique et l'utilisation de la hiérarchie.

Analyse

Genlib est, contrairement à *VHDL* ou Verilog, un langage orienté vers la synthèse et non pas vers la simulation. Il est spécialisé dans la description de la structure, et ne permet pas la description comportementale. De plus, il intègre fortement la notion de hiérarchie. Cependant, ce langage ne répond pas aux besoins de différenciation entre représentations arithmétiques.

Une possibilité aurait été d'adapter **Alliance** et **Genlib** à nos besoins. Cela aurait pu être notre choix si le LIP6 n'avait pas commencé en 2000 la définition d'une nouvelle plate-forme de développement, la plate-forme **Coriolis**.

5.2.3 Coriolis

Caractéristiques

Coriolis [Cor, ACC+05, Ale07] est la nouvelle plate-forme de conception du LIP6. C'est une plate-forme fortement orientée placement/routage.

Hurricane est la structure de données interne de **Coriolis**. Cette base de donnée est écrite en C++ et exploite donc la notion de langage objet. Par ailleurs, elle permet (contrairement à **MBK**) de représenter de manière unifiée le circuit selon les différents niveaux de représentation. Elle unifie donc les aspects logiques et physiques des circuits en une seule vue. Cela permet aux outils d'interagir à travers elle plutôt que d'être découplés les uns des autres et de devoir utiliser des fichiers pour "communiquer".

Coriolis s'interface avec plusieurs formats standards et utilise le langage de description **Genlib+**.

Langage

Nous avons personnellement défini et mis en œuvre le langage de description **Genlib+** préalablement à cette thèse. Notre but était de définir un langage ayant la même API que **Genlib** et utilisant un langage objet (Python en l'occurence) de façon à s'interfacer simplement avec **Coriolis**.

Analyse

Genlib+ est un langage orienté vers la synthèse. Tout comme **Genlib**, ce langage ne correspond pas parfaitement à nos besoins, principalement parce qu'il ne répond pas aux besoins de différenciation entre représentations arithmétiques.

Cependant, le fait d'avoir à disposition la base de donnée **Hurricane** est un point intéressant.

5.2.4 Genoptim

Caractéristiques

Genoptim n'est pas une plate-forme de conception de circuits mais un environnement d'aide à la conception de générateurs portables [Hou97, Vau97]. Il nous a donc paru intéressant de nous y intéresser également.

Genoptim se base sur le concept de bibliothèques de cellules logiques *virtuelles* qui permet la conception de blocs *VLSI* portables sur différentes technologies. Il fournit également une bibliothèque d'opérateurs arithmétiques contenant, entre autres, additionneurs et multiplieurs.

Le point d'entrée de **Genoptim** est un langage du même nom destiné à la synthèse.

Langage

Le langage **Genoptim** est une bibliothèque de fonctions C. Il a été conçu pour décrire des opérateurs arithmétiques de grande complexité. Il fournit différents types de fonctionnalités, aux niveaux structurel (réalisation de la liste d'interconnexions des portes), topologique (placement des cellules) et de la validation (intégration d'éléments nécessaires à la validation fonctionnelle dans les générateurs). Il fournit également un mécanisme de projection structurelle se traduisant par une insertion, au niveau des cellules de base, d'un niveau hiérarchique *virtuel* : ce niveau regroupe les fonctions logiques les plus couramment utilisées et peut réaliser toutes les fonctions virtuelles à partir des deux portes de base que sont le *nand* et l'inverseur. Il fournit enfin différents mécanismes en fonction de la bibliothèque cible, tels des algorithmes de placement et d'insertion d'amplificateurs.

Analyse

Le langage **Genoptim** paraît être celui le plus adapté à nos besoins, étant un langage dédié au développement d'opérateurs arithmétiques, et intégrant des mécanismes appréciables tels le niveau hiérarchique virtuel. Cependant, bien que dédié à l'arithmétique, ce langage ne répond pas aux besoins de différenciation entre représentations arithmétiques, un signal restant un vecteurs de bits.

De plus, ce langage est limité de part le statut propriétaire de ses sources. Cela empêche tout ajout de macro C qui permettrait de modéliser les représentations arithmétiques.

Tout comme **Genlib**, il n'a pour vocation que la génération de structure et ne dispose pas de mécanismes permettant de modifier dynamiquement sa structure.

5.2.5 Conclusion

Comme nous avons pu le voir, il n'existe à ce jour aucun langage et aucune structure de données répondant aux besoins de différenciation entre représentations arithmétiques. De plus, aucun de ces langages n'exploite la notion objet qui pourrait permettre de définir de façon aisée différents types de signaux. C'est la raison première pour laquelle nous avons envisagé de développer un nouveau langage.

Par ailleurs, tous les langages que nous avons évoqués sont très verbeux dès qu'il s'agit de décrire de la structure. Notre but était alors de créer un langage permettant de réduire la verbosité de façon à effectuer la description structurelle des circuits tout en gardant un niveau de compacité et de lisibilité au moins équivalent à une description comportementale.

On a pu voir les prémices de simplification d'écriture dans des langages comme JHDL [JHD] (enrichissement du langage Java) et MyHDL [Dec04] (enrichissement du langage Python). Ces langages ne correspondent pas non plus à nos besoins, le premier étant dédié à la description de circuits matériels reconfigurables pour FPGA, et le second étant dédié à la simulation. Cependant ces langages montrent l'engouement croissant durant les dernières années pour utiliser des langages orientés objet pour la description de circuits.

Le premier choix a été d'écarter les environnements industriels, trop peu ouverts vers l'extérieur pour permettre de façon aisée l'ajout de fonctionnalités ou la modification de leur base de données. Parmi les environnements académiques, l'environnement **Coriolis** nous a paru être le plus adapté, de part sa structure de données et son accessibilité.

Genlib+ (notre précédent langage) ne couvrait cependant pas tous nos besoins. Nous avons donc fait le choix de définir et de mettre en œuvre un nouveau langage, nommé **Stratus**, répondant à nos contraintes.

Un de nos buts était de gagner en concision par rapport à **Genlib+**. Nous conservons une description structurelle, mais élevons le niveau d'abstraction. En ce sens, ce langage fournit différents mécanismes appréciables servant à faciliter la description de vues structurelles. De plus, d'un point de vue arithmétique, **Stratus** a été conçu dans le but de permettre l'utilisation de différents systèmes de numération, ce qui n'existait jusqu'alors pas. Enfin, nous avons fourni des fonctions permettant de décrire une vue physique de façon détaillée en vue d'exploiter les possibilités offertes par la plateforme **Coriolis**, ainsi que des stimuli permettant d'effectuer la simulation du circuit.

5.3 L'environnement Stratus

5.3.1 Présentation générale

Stratus est un langage procédural de description de circuit ; c'est également une structure de données, développée pour répondre aux besoins de la description de circuits et de l'arithmétique. Cette structure de données est liée à la structure de données **Hurricane**, et permet de représenter la partie logique des circuits.

Nous avons fait ce choix car la structure de données **Hurricane** est beaucoup plus vaste que nos besoins, et il nous paraissait plus intéressant de se focaliser sur un sous-ensemble d'**Hurricane**, pour, une fois les manipulations terminées, et la vue structurelle prête, utiliser **Hurricane** de façon à obtenir le dessin des masques.

Par ailleurs, la structure de données **Hurricane** ne permet la manipulation des signaux qu'à un niveau bit. Or nous voulions avoir la possibilité de manipuler, de façon simple, des vecteurs de bits.

Structure de données

La structure de données liée à **Stratus** permet de représenter en mémoire les différents éléments d'une description structurelle : instances, signaux, connectique ... Elle exploite pleinement la notion de langage objet. Trois classes servent à décrire les circuits : une permettant de décrire une cellule (i.e. le circuit en cours de création), une pour les instances (i.e. les sous-blocs composant le circuit) et une pour les signaux (i.e. les connexions permettant de relier les différents sous-blocs entre eux).

Langage de description de circuits

Le but est de s'inscrire dans le contexte d'outils *d'aide à la conception de chemin de données*. Différentes méthodes sont donc mises à disposition de façon à élever le niveau d'abstraction, tout en restant dans le domaine du structurel.

De façon à élargir les possibilités en matière de description de circuits, **Stratus** permet également d'effectuer la description physique d'un circuit ainsi que la description de séquences de stimuli.

Plate-forme de conception

Dans le but d'intégrer toutes les étapes du flot de conception, **Stratus** encapsule l'utilisation des différents outils de placement automatique et de routage fournis par la plate-forme **Coriolis**.

Il fournit également la possibilité de faire appel à un simulateur.

Par ailleurs, de façon à s'interfacer avec différents outils, plusieurs descriptions (*VHDL* entre autres) peuvent être générées à partir d'une description faite en **Stratus**.

5.3.2 Justification du choix du langage Python

Le choix d'enrichir un langage nous a paru naturel, car beaucoup plus accessible, et dont le résultat est équivalent voire meilleur qu'un langage spécialisé. En effet, on fournit dans ce cas des fonctions spécifiques à la manipulation d'objets *VLSI* tout en ayant à disposition toutes les possibilités offertes par un langage généraliste.

Le choix d'un langage orienté objet nous a paru le plus adapté à la description de circuits car nous permettant de créer une structure de données répondant parfaitement à nos besoins. Nous voulions également un langage fournissant du prototypage et un cycle de développement rapide, ce qui a orienté notre choix vers les langages interprétés et non pas compilés. Notre choix s'est finalement porté sur le langage Python.

Python est développé depuis 1989 sous la direction de Guido van Rossum [sf]. Ce langage permet une approche modulaire et orientée objet de la programmation. Il est doté entre autres d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions. De plus, sa syntaxe facilite son apprentissage et en fait un candidat idéal pour l'apprentissage de la programmation orientée objet (par exemple, de part le fait que les blocs sont identifiés par l'indentation, il oblige le programmeur à écrire un code lisible, ce qui en fait un bon langage pédagogique).

5.3.3 Structure de données

Trois classes existent de façon à décrire les circuits : une pour les cellules (classe `Model`), une pour les instances (classe `Inst`) et une pour les signaux (classe `Net`).

Classe Model

La création d'un circuit se fait en décrivant une classe dont on définit une méthode pour chacune des étapes de la description (`Netlist`, `Layout` ...). Cette classe doit dériver de la classe `Model`, fournissant attributs et méthodes nécessaires, que nous détaillerons par la suite.

Un modèle est caractérisé par un nom et le jeu de paramètres permettant de le définir.

L'une des forces de **Stratus** est la possibilité de décrire des cellules génériques, pour ensuite créer différentes instances en donnant différentes valeurs aux paramètres.

Attributs Cette classe possède différents attributs, dont : le dictionnaire des paramètres (`_param`), la liste des instances de la cellule (`_st_insts`), la liste des signaux externes (`_st_ports`), la liste des signaux internes (`_st_sigs`).

Méthodes Outre les méthodes décrites par le concepteur de circuits pour les différentes étapes de conception, plusieurs méthodes sont fournies, telles que la sauvegarde de la cellule sur disque dans différents formats de sortie standards (`Save`), ou au format **Stratus** (`SaveStratus`), la création de la cellule dans la base de données **Hurricane** à partir de la base de données **Stratus** (`HurricanePlug`) et la visualisation du dessin des masques grâce à l'éditeur de **Coriolis** (`View`).

Classe Inst

Le premier élément constituant une cellule est l'instance.

Une instance est caractérisée par son modèle de référence et sa connectique. On peut également lui donner un nom.

Attributs Cette classe possède différents attributs, dont : son nom (`_name`), son modèle (`_model`), son dictionnaire de connexions (`_map`), et la cellule à laquelle elle appartient (`_st_cell`).

Classe net

Le deuxième élément constituant une cellule est le signal.

Un signal (ou *net*) est caractérisé par sa taille (étant donné qu'en **Stratus** les signaux sont des vecteurs de bits) et son nom.

En pratique, ce n'est pas la classe `net` qui est utilisée, mais l'une des classes dérivées, en fonction du type de signal voulu (`SignalIn`, `SignalOut`, `Signal` etc). Ces classes possèdent alors les attributs et méthodes de la classe `net` en commun.

Attributs Ces classes possèdent différents attributs, dont : le nom du signal (`_name`), sa taille (`_arity`), un booléen signifiant si le signal est externe ou interne (`_ext`), sa direction (`_direct`, pour les signaux externes uniquement), et la cellule à laquelle il appartient (`_st_cell`).

Méthodes Ces classes possèdent également différentes méthodes. Ces méthodes sont les principaux procédés permettant d'élever le niveau d'abstraction du langage. Nous y reviendrons dans la section suivante.

5.3.4 Langage de description

Niveaux de description

L'idée directrice est de fournir au concepteur un langage permettant différents niveaux de description, allant d'une description structurelle haut niveau dite "molle", à une description structurelle bas niveau. Le concepteur peut alors choisir la façon dont il veut concevoir son chemin de données en fonction de ses besoins. Comme mentionné précédemment, **Stratus** permet de décrire un circuit avec différents niveaux de description.

La description bas niveau C'est une description de la vue structurelle d'un circuit sous forme d'instanciations explicites de cellules avec connectique.

La description haut niveau C'est une description dans laquelle l'architecture des opérateurs n'est pas précisée.

Cette description peut s'apparenter à une description comportementale, en ce sens que c'est la fonctionnalité souhaitée qui est décrite, indépendamment de la structure ou de la technologie. Il n'y a par contre pas d'étape de synthèse au premier sens du terme : chaque fonctionnalité décrite correspond à une (et une seule) structure, créée à la volée dans la base de données, c'est pourquoi nous préférons parler de description structurelle haut niveau.

Le but de cette description est double : faciliter le travail des concepteurs en leur permettant de mettre de côté des problèmes tels que l'architecture des opérateurs, ou la taille des signaux, tout en diminuant la verbosité d'écriture. Elle sert principalement dans le cadre de l'optimisation arithmétique, pour laquelle un compromis est effectué entre facilité d'écriture et maîtrise du résultat de la synthèse.

Bibliothèques de cellules

Toutes les bibliothèques de cellules statiques sont utilisables avec **Stratus**, notamment celles de la chaîne de *CAO* **Alliance** [GP92] (cellules standards, cellules *full-customs*, plots ..), mais aussi celles des fondeurs.

Il fournit également une bibliothèque virtuelle, mécanisme permettant de changer, de façon aisée, la bibliothèque cible d'un circuit, sans avoir à le modifier. Il suffit alors de rédiger un fichier explicitant la correspondance vers la bibliothèque voulue (le format xml est utilisé pour cette description). Les cellules disponibles dans cette bibliothèque sont toutes les cellules booléennes *basiques* : des portes and, $nand$, or, nor, inv ... ainsi qu'un *HA* et un *FA*.

Stratus fournit enfin deux bibliothèques dynamiques (i.e. leurs cellules sont produites à partir de générateurs paramétrables écrits avec **Stratus**). En effet, la réutilisation est devenue le maître mot de la conception de SOC, puisque c'est la seule méthode permettant la conception de très gros circuits à un coût acceptable [SCMLN01, SCM01].

- **Dpgen** : bibliothèque de générateurs pré-placés (tels qu'une RAM, une Fifo, un Shifter ...) dédiés à la génération de chemins de données (cette bibliothèque est calquée sur la bibliothèque du même nom en langage Genlib, fournie par **Alliance** [GP94], et se base donc sur les cellules des différentes bibliothèques d'**Alliance**),
- **ArithLib** : bibliothèque d'opérateurs arithmétiques (basée sur la bibliothèque virtuelle).

Description de l'interface

Dans la méthode `Interface` sont décrits tous les ports externes de la cellule. **Stratus** fournit les types de signaux courament usités : entrées / sorties (`SignalIn, SignalOut, SignalInOut, Tristate`), horloge (`CkIn`), alimentations (`VddIn, VssIn`). La création d'un signal s'effectue en donnant le nom du signal et son arité.

Notons que les ports d'alimentation et d'horloge sont typés différemment des autres ports d'entrée de façon à pouvoir facilement leur appliquer des traitements spécifiques, lors de la phase de routage principalement.

Description de la vue structurelle

La création la vue structurelle se fait en fonction du niveau de description choisi :

- la première façon de faire est inspirée de langages tels que **Genlib** ou **Genoptim** : c'est l'instanciation explicite de portes avec la connectique,
- la seconde façon consiste à tirer profit des différents services mis en place permettant de rendre plus concise la description structurelle, que nous voulons aussi lisible qu'une description comportementale.

Voici l'exemple de l'instanciation d'une porte AND :

```
Inst ( "a2_x2"
    , map = { 'i0' : in0
            , 'i1' : in1
            , 'q' : out
            , 'vdd' : vdd
            , 'vss' : vss
            }
   )
```

ou bien, en notation simplifiée :

```
out <= in0 & in1
```

Dans la notation dite *explicite*, le modèle à utiliser est donné (`a2_x2`) ainsi que le nom de l'instance (`mon_inst_and2`) suivi par une liste de paires (dictionnaire `map`), comprenant des noms de ports de l'instance et les signaux auxquels ces ports sont connectés. Dans la notation simplifiée, seule l'opération est spécifiée, grâce à la surcharge de l'opérateur `&`.

Naturellement cette notation est disponible pour différents opérateurs booléens et arithmétiques : ou (`|`), ou exclusif (`^`), inverseur (`~`), addition (`+`), multiplication (`*`).

La structure résultant de cette description est exactement la même que si la description structurelle explicite est utilisée. Le but est simplement de réduire la verbosité.

Différentes méthodes de la classe `net` sont également fournies, dans un même but de simplification. Leur rôle est de simplifier l'instanciation de multiplexeurs, *buffers*, registres :

- `Mux` : `out <= cmd.Mux ("0,4" :in0, "1-3,5" :in1, "def" :0)`
- `Buffer` : `out <= in.Buffer()`
- `Register` : `out <= ck.Reg(in)`

Description de la vue physique

Un placement, à la main, peut être décrit dans **Stratus**. Il se fait par technique d'aboutement. Différentes fonctions existent pour faire du placement absolu et relatif : **Place**, **PlaceTop**, **PlaceRight**, ...

Différentes fonctions existent également de façon à créer des segments, des vias, ... Cependant, **Stratus** n'est pas fait pour faire du routage à la main à proprement parler, il n'y a pas de fonctions pour décrire des directives de routage ou des contraintes de routage à la main.

Description des stimuli pour la simulation

Stratus permet également de décrire de façon procédurale des séquences de stimuli permettant de valider fonctionnellement - par simulation - les composants matériels générés. Cette génération de patterns existe pour deux formats différents : le format .pat de la chaîne **Alliance** et le format *VHDL* IEEE.

5.3.5 Plate-forme de conception

Encapsulation des outils de placement / routage

Le placement automatique se fait par le biais d'une fonction permettant de choisir quel placeur de la plate-forme **Coriolis** utiliser, et de le paramétrer. Une fonction identique sera mise en place pour encapsuler l'utilisation des outils de routage quand ceux-ci seront finalisés, ainsi que pour l'outil d'analyse des temps de propagation.

Au delà de l'encapsulation des différents outils, quelques fonctionnalités ont été ajoutées dans le cadre de l'utilisation de **Stratus** dans l'enseignement : spécification de primitives de routage automatiques tels que la mise en place de rappels d'alimentations ou d'une grille d'horloge.

Ces traitements sont en dehors du cadre de cette thèse, nous n'entrerons donc pas plus dans les détails dans ce chapitre.

Encapsulation d'un simulateur

De façon à valider fonctionnellement les circuits en cours de développement avec la séquence de stimuli définie, **Stratus** permet l'appel à un simulateur.

Sauvegarde

Différents *drivers* ont été développés de façon à exporter une cellule décrite au format **Stratus** dans différents formats standards : *VHDL* et Verilog principalement.

Optimisation arithmétique

L'utilisation des différents algorithmes d'optimisation arithmétique se fait par le biais d'une méthode permettant de choisir quel algorithme utiliser, quel critère privilégier, quelle représentation utiliser, ...

5.3.6 Conclusion

Seule une description générale des caractéristiques de **Stratus** a été faite ici. Un exemple de l'utilisation de **Stratus** est montré dans l'Annexe C. De plus, la documentation (en anglais) de **Stratus** est présentée dans l'Annexe D.

Au delà du cadre de cette thèse, le but de cet environnement de conception était d'être le plus complet possible en permettant de n'utiliser qu'un seul langage informatique pour toutes les étapes de description d'un circuit. En effet, la plupart des chaînes industrielles utilisent *VHDL* ou Verilog pour la description, puis le langage tcl/tk [TCL] pour l'appel aux différents outils. L'utilisation de **Stratus** permet de rendre ces traitements unifiés.

La Figure 5.1 présente un récapitulatif, dans un flot de conception, des différentes descriptions qui peuvent être réalisées grâce au langage **Stratus** ainsi que les vues obtenues. Nous montrons également dans cette Figure les étapes du flot qui sont réalisées par la chaîne de *CAO* **Coriolis**, par le biais du langage **Stratus**.

Stratus est enseigné aux étudiants suivant le Master mention Informatique spécialité ACSI (Spécialité Architecture et Conception des Systèmes Intégrés) de l'UPMC depuis 2005. Il est utilisé dans plusieurs unités d'enseignement dans le but de familiariser les étudiants à la conception des circuits intégrés numériques *VLSI*, et à l'utilisation d'outils de *CAO*. L'une de ces unités d'enseignement a pour objectifs l'implantation *VLSI* d'un micro-processeur 32 bits.

Stratus a par ailleurs été utilisé dans la conception d'une architecture matérielle générique d'analyse de signaux à large bande dans le domaine temps-fréquence : l'architecture F-TFR (Fast Time-Frequency Representation), fabriquée en technologie 0.13μm [NDMT07, Nou08]. Il a également été utilisé dans la conception d'un micro-réseau sur puce [SPG07, She08], et de matrices FPGA [MMMT04, PMM08]. Tous ces circuits ont été fabriqués et fonctionnent correctement.

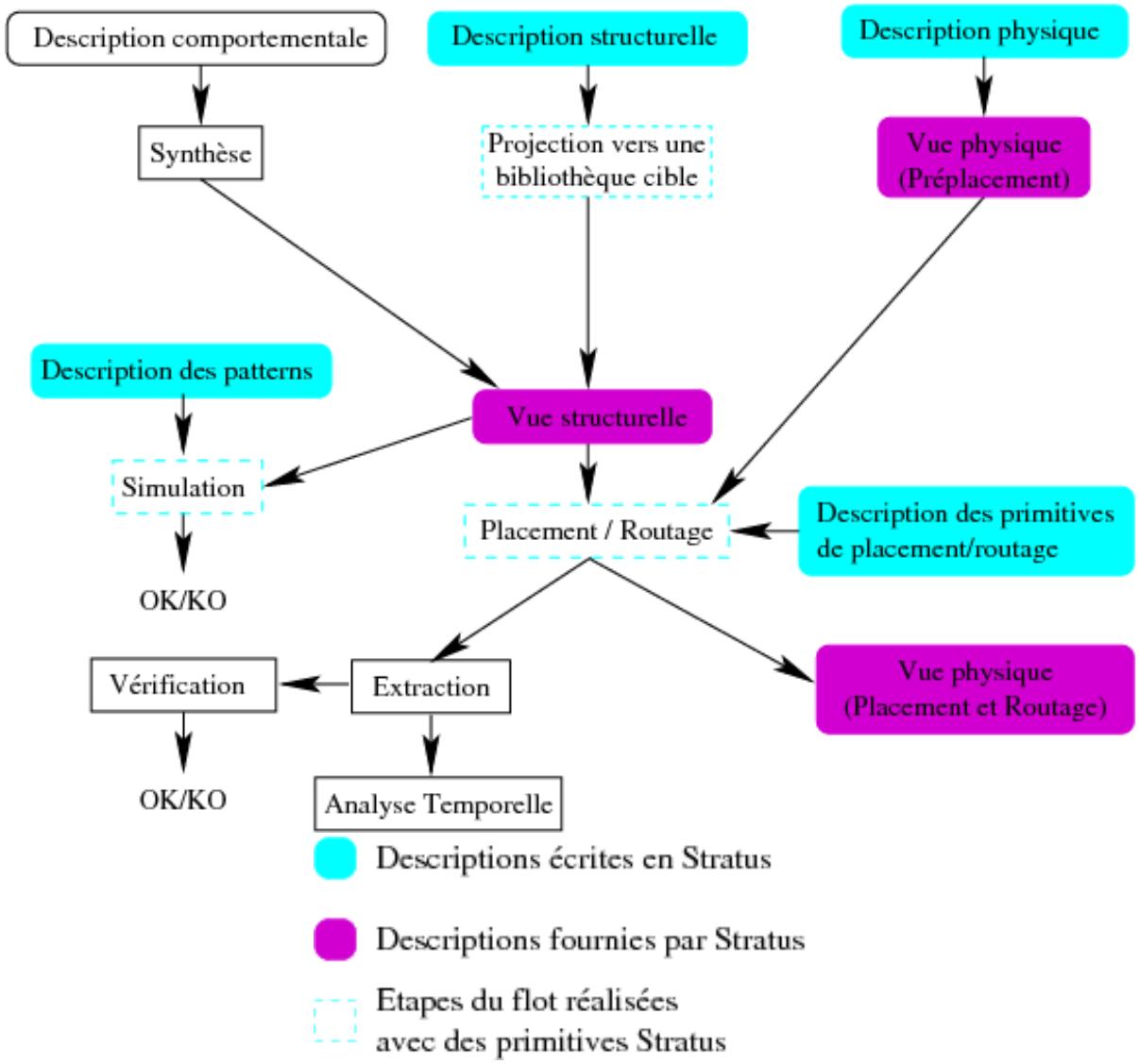


FIG. 5.1 – Descriptions fournies par **Stratus**

5.4 Le *package* arithmétique

5.4.1 Présentation générale

Le langage **Stratus** a été conçu dans le but de faciliter la conception de chemins de données et d'être adapté aux différents traitements d'optimisation. L'idée directrice était d'offrir au concepteur un langage permettant différents niveaux de description,

allant d'une description *molle* haut niveau, à une description structurelle bas niveau. Lors de l'utilisation de la description *molle* de **Stratus**, un générateur est automatiquement instancié (le générateur par défaut pouvant être modifié grâce à des méthodes ad-hoc).

Le but du *package* arithmétique est de modifier le comportement de l'outil : lors de l'utilisation de la description *molle*, un outil d'optimisation doit trouver la meilleure architecture possible en fonction de paramètres donnés.

De façon à pouvoir inclure l'arithmétique redondante dans les traitements d'optimisation, il faut également fournir la possibilité de décrire les différents types de signaux en matière de représentation arithmétique : Complément à 2, Carry-Save, Borrow-Save. C'est également le rôle de ce *package*.

5.4.2 Description arithmétique

Pour pouvoir décrire des chemins de données avec de l'arithmétique redondante, il faut pouvoir décrire des signaux redondants. Le *package* arithmétique de **Stratus** modifie donc la création des signaux en ajoutant un nouveau paramètre permettant de décrire le type arithmétique des signaux. Les signaux se créent comme suit : `signal = ASignalType ( "name", arity, repr = "cs" )`, pour un signal Carry-Save par exemple. Cette instruction crée deux signaux, un nombre Carry-Save étant composé de deux termes, utilisables comme suit : `signal[0]` et `signal[1]`.

La création de ces nouvelles classes tire parti de la notion d'héritage multiple fournie par Python, comme on peut le voir dans la Figure 5.2. Ces nouveaux signaux ont toutes les nouvelles propriétés arithmétiques (attribut `_repr` par exemple définissant leur représentation arithmétique de par leur héritage à la classe `anet`) tout en ayant les propriétés propres à leur type.

De plus, lors de l'utilisation de la surcharge de l'opérateur + comme suit : `s <= a + b`, l'architecture de l'additionneur à utiliser est automatiquement choisie en tenant compte des représentations des différents signaux.

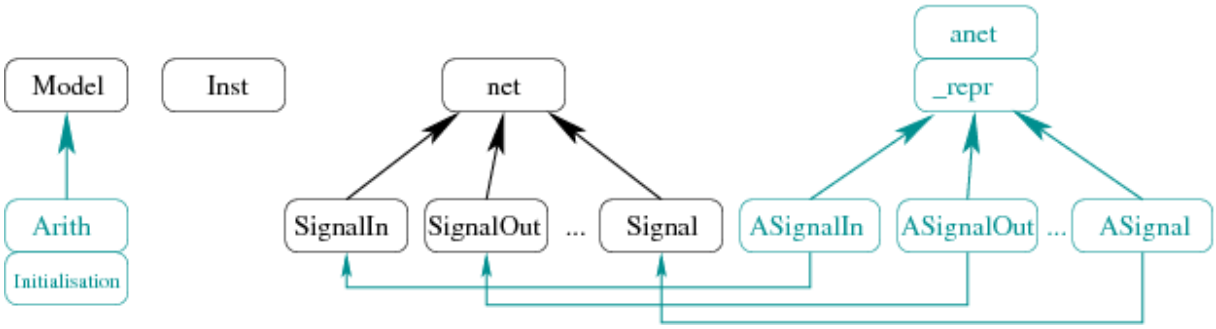


FIG. 5.2 – Diagramme de classes

5.4.3 Mécanismes d'optimisation

Le *package* arithmétique permet de tirer parti des différents niveaux de description proposés par **Stratus**. La façon la plus simple de procéder est en effet de fournir à l'outil d'optimisation un circuit décrit en notation *molle*. A partir de cette description, c'est à l'outil d'effectuer la spécification complète de la vue structurelle en choisissant le meilleur générateur à disposition. L'outil se base sur la bibliothèque virtuelle de **Stratus**, il fournit donc une description structurelle intermédiaire : après spécification complète des différents opérateurs et des interconnexions et avant projection vers une technologie cible.

L'utilisation des mécanismes d'optimisation se fait tout simplement en appelant une méthode d'initialisation en redondant (`Initialisation`). Pour cela, une nouvelle classe dérivant de la classe `Model` a été créée, comme montré dans la Figure 5.2.

Les arguments de cette méthode servent à préciser quel algorithme d'optimisation utiliser parmis les différents algorithmes vus dans le chapitre précédent, et donner les paramètres de ces algorithmes : avec ou sans fonction de coût, prise en compte ou non des architectures Borrow-Save pour l'optimisation des soustractions.

Un exemple d'utilisation et une liste des différents paramètres est donné dans l'Annexe C.

5.4.4 Bibliothèque ArithLib

La bibliothèque **ArithLib** est la bibliothèque d'opérateurs arithmétiques dynamiques dont nous avons parlé en faisant la liste des bibliothèques de cellules fournies par **Stratus** dans la section précédente. Cette bibliothèque regroupe les opérateurs arithmétiques selon leur représentation.

Des règles ont été respectées pour que l'utilisation des blocs conçus soit aisée : les blocs sont fortement paramétrables et ont subi une vérification la plus complète possible. Le fait qu'ils soient indépendants de la technologie est également primordial, dans un but de réutilisabilité. En effet, ces blocs sont alors réutilisables avec différentes bibliothèques de cellules sans surcoût. De plus, une standardisation des interfaces, une bonne documentation et un code bien écrit et lisible sont des plus améliorant de façon non négligeable la facilité d'utilisation. Autant de règles que nous nous sommes efforcés de suivre dans l'élaboration de cette bibliothèque.

La documentation de cette bibliothèque est fournie dans l'Annexe E.

Notons juste ici que deux additionneurs classiques sont fournis : le *Ripple* et le *Sklansky*. Les additionneurs de type mixte et redondant sont également fournis, prenant tous deux en compte la représentation Carry-Save et la représentation Borrow-Save (avec sortie *BS* ou *CS*).

Les multiplieurs de type classique (algorithme de Booth), mixte et redondant (algorithme direct) sont également à disposition, utilisant les différentes représentations arithmétiques.

5.5 Conclusion

Nous avons présenté dans cette section l'environnement de conception avec lequel nous avons travaillé.

Tout d'abord d'un point de vue concepteur de circuits, étant donné que nous avons présenté le langage de description que nous avons développé pendant cette thèse. Ensuite, d'un point de vue concepteur d'outils de *CAO*, étant donné que nous avons présenté la structure de données sur laquelle se basent le langage et nos différents outils d'optimisation arithmétique.

La mise en place du langage s'est faite avec une vision d'*aide à la conception*, en ce sens que nous nous sommes efforcés de développer un langage clair, intuitif, aisé à appréhender. En parallèle, la mise en place de la structure de données s'est faite sans perdre de vue notre but premier, être adapté à l'implantation d'algorithmes d'optimisation de circuits.

Chapitre 6 Applications

Nous faisons dans ce chapitre un récapitulatif des résultats en termes de surface et de chaîne longue que nous avons obtenus en appliquant nos différents algorithmes à plusieurs circuits arithmétiques. Ces résultats sont également comparés à ceux des architectures classiques de façon à montrer les améliorations obtenues.

Toutes les architectures présentées ont été obtenues avec l'utilisation de la bibliothèque de cellules précaractérisées de la chaîne Alliance [GP92] (en 0.35μm) et les outils de placement/routage/analyse de timing de Cadence : **Encounter**.

6.1 Filtre

Différentes architectures de filtre *FIR* (Finite Impulse Filter) existent [dBNNC03, dBNN04, NMDT04] ; nous nous intéressons particulièrement à celle présentée dans la Figure 6.1.

Comme on peut le voir, cette architecture ne contient ni soustraction ni opérateur avec sorties multiples. Les différents algorithmes s'utilisent donc avec uniquement la notation Carry-Save (et sans pré-traitement nécessaire) et sans une gestion spécifique en ce qui concerne les sorties multiples.

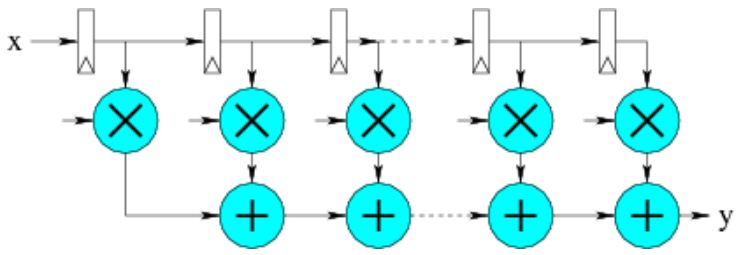
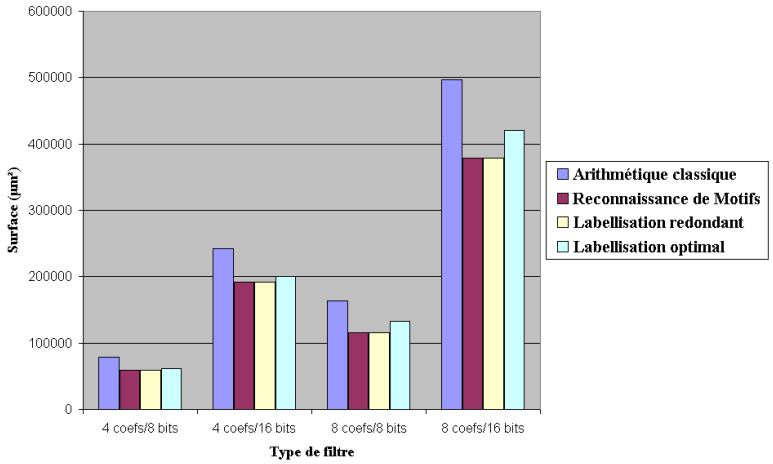


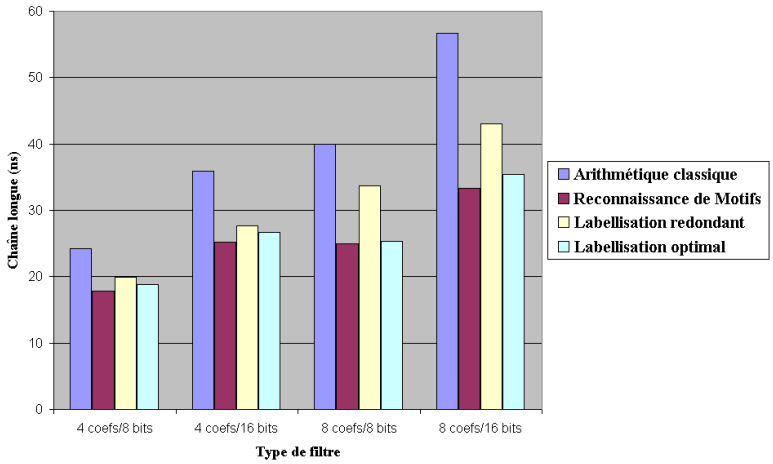
FIG. 6.1 – Architecture d'un filtre

Les résultats obtenus sont présentés dans les Figures 6.2 à 6.4. Ils résultent de l'optimisation de quatre versions différentes du filtre : 4 coefficients, valeurs sur 8 bits, 4 coefficients, valeurs sur 16 bits, 8 coefficients, valeurs sur 8 bits, 8 coefficients, valeurs sur 16 bits.



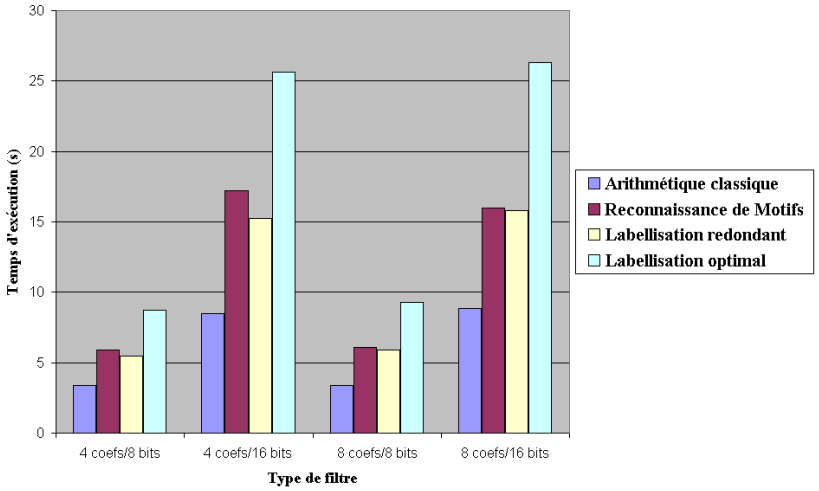
	Arithmétique Classique	Reconnaissance de Motifs		Labellisation Redondant		Labellisation Optimal	
	μm^2	μm^2	%	μm^2	%	μm^2	%
4coef/8bits	79 012	58 469	-26%	58 469	-26%	62 027	-21.5%
4coef/16bits	242 799	192 055	-20.9%	192 055	-20.9%	200 535	-17.4%
8coef/8bits	162 912	114 978	-29.4%	114 978	-29.4%	132 398	-18.7%
8coef/16bits	497 117	378 555	-23.8%	378 555	-23.8%	421 093	-15.3%

FIG. 6.2 – Résultats du filtre : Surface



	Arithmétique Classique	Reconnaissance de Motifs		Labellisation Redondant		Labellisation Optimal	
	ns	ns	%	ns	%	ns	%
4coef/8bits	24.27	17.88	-26.3%	19.93	-17.9%	18.83	-22.4%
4coef/16bits	35.89	25.23	-29.7%	27.71	-22.8%	26.62	-25.8%
8coef/8bits	40.02	24.99	-35.6%	33.71	-15.8%	25.35	-36.7%
8coef/16bits	56.68	33.34	-41.2%	43.07	-24%	35.41	-37.5%

FIG. 6.3 – Résultats du filtre : Chaîne longue



	Arithmétique Classique	Reconnaissance de Motifs		Labellisation Redondant		Labellisation Optimal	
	s	s	%	s	%	s	%
4coef/8bits	3.4	5.92	+74.1%	5.48	+61.2%	8.73	+156.7%
4coef/16bits	8.47	17.2	+103%	15.23	+79.8%	25.64	+202.7%
8coef/8bits	3.4	6.09	+79.1%	5.92	+74.1%	9.26	+172.3%
8coef/16bits	8.84	15.96	+80.5%	15.77	+78.4%	26.31	+197.6%

FIG. 6.4 – Résultats du filtre : Temps d'exécution

Ces résultats montrent tout d'abord que tous les algorithmes améliorent la surface et la chaîne longue, par rapport à une implantation en arithmétique classique. On remarque ensuite que chacun des algorithmes donne un résultat différent :

- d'un point de vue surface, ce sont les deux algorithmes *redondant dès que possible* qui fournissent le meilleur résultat,
- d'un point de vue chaîne longue, c'est la reconnaissance de motif qui donne le meilleur résultat, suivi de l'algorithme d'allocation optimals, puis de l'algorithme de labellisation redondant dès que possible.

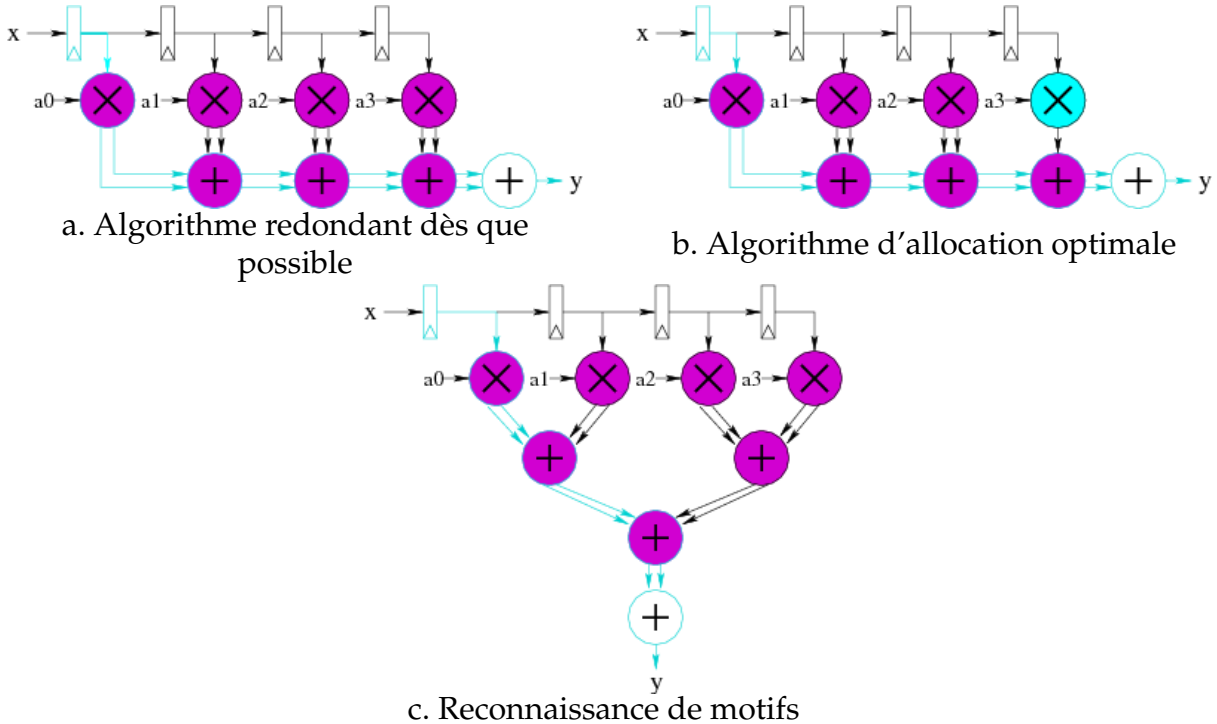
Cette différence est due au fait que les trois algorithmes ne génèrent pas la même architecture, comme montré dans la Figure 6.5 (exemple avec quatre coefficients).

Comparons tout d'abord les deux algorithmes de labellisation. L'algorithme d'allocation optimale améliore le délai au détriment de la surface par rapport à l'algorithme redondant dès que possible : en effet, le choix est fait de transformer uniquement trois multiplieurs dans leur version redondante (et donc laisser 1 multiplieur dans sa version classique, pour 4 coefficients, 5 pour 8 coefficients, ...), de façon à ce qu'au moins un additionneur de la chaîne longue (entre la sortie du premier registre et la sortie du circuit) soit un additionneur mixte et plus un additionneur redondant.

Cependant, le résultat obtenu par l'algorithme d'allocation optimale est moins performant que celui donné par la reconnaissance de motifs. En effet, alors que les algorithmes de labellisation "ne font que" modifier l'architecture choisie pour chaque

opérateur et pas la connectique, l'ensemble de motifs choisi exploite les propriétés de commutativité et d'associativité de l'addition pour effectuer de légères modifications à la connectique d'un circuit. Cette différence engendre qu'il y a au moins un additionneur de moins dans la chaîne longue de l'architecture générée par l'algorithme basé sur la reconnaissance de motifs.

Par ailleurs, notons que les améliorations sont plus importantes pour les circuits avec huit coefficients que pour ceux avec quatre coefficients. Autrement dit, plus la chaîne d'opérateurs arithmétiques est grande, meilleures sont les optimisations.



a. Algorithme redondant dès que possible

b. Algorithme d'allocation optimale

c. Reconnaissance de motifs

FIG. 6.5 – Architectures optimisées d'un filtre avec quatre coefficients

D'un point de vue temps d'exécution, nous pouvons voir que l'algorithme d'allocation optimale est plus lent que l'algorithme de labellisation redondant dès que possible, ce qui est en cohérence avec la théorie. Nous voyons également que la reconnaissance de motifs est elle à peine plus lente que l'algorithme redondant dès que possible.

6.2 Unité de calcul de distance

L'architecture d'une unité de calcul de distance (*DCU* : Distance Computation Unit) est composée de deux parties, comme montré dans la Figure 6.6 [DBM01]. La première effectue le calcul de la distance $(A_i - B_i)^2$, tandis que la seconde effectue la somme entre ces distances.

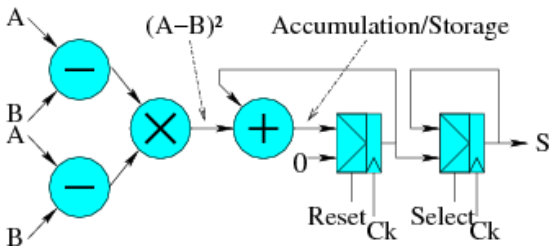


FIG. 6.6 – Architecture d'une DCU

Cet opérateur contenant des soustractions, nous présentons les résultats obtenus avec les différentes façons de prendre en compte cette opération. Sont donc comparées trois différentes gestions de la soustraction : l'utilisation des architectures Carry-Save uniquement, après transformation des soustractions en additions (*Carry-Save*), et l'utilisation des architectures Borrow Save, les additionneurs ayant une sortie Carry-Save (*Borrow-Save 1*) ou Borrow-Save (*Borrow-Save 2*).

Par ailleurs cette architecture ne contient pas d'opérateur avec sorties multiples. Nous utilisons donc la reconnaissance de motifs sans la gestion spécifique pour les sorties multiples.

Les résultats sont présentés dans les Figures 6.7 à 6.9. Ils résultent de l'optimisation d'une *DCU* avec les données en entrée sur 16 bits.

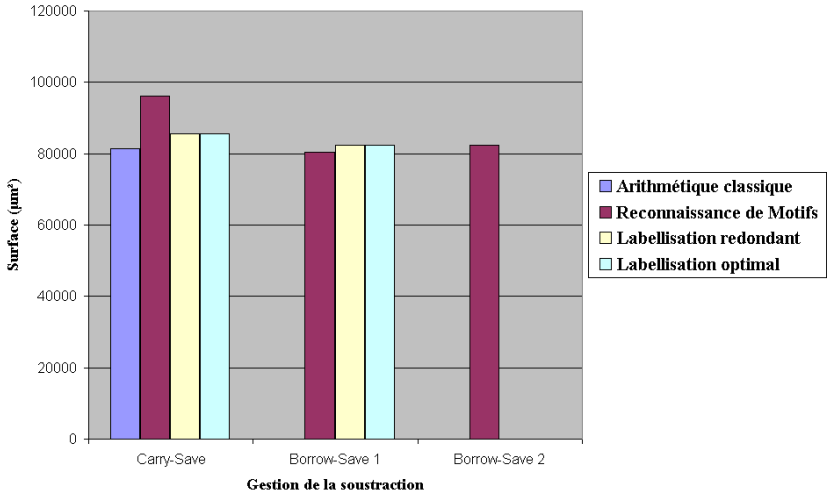
Ces résultats se résument à :

- dans le pire des cas, une légère détérioration de la surface, principalement pour l'optimisation Carry-Save,
- une amélioration de la chaîne longue pour tous les cas d'optimisation, avec de meilleures performances pour les optimisations Borrow-Save.

Ces résultats concordent avec notre postula de base disant que les architectures Borrow-Save sont plus appropriées que les architectures Carry-Save pour l'optimisation de circuits contenant des soustractions. En effet, l'utilisation d'architectures Borrow-Save dans ce cas permet l'utilisation d'un multiplieur redondant avec entrées Borrow-Save, alors que l'utilisation d'architectures Carry-Save uniquement oblige à effectuer une inversion sur l'entrée à soustraire et une addition à '1' avec un additionneur.

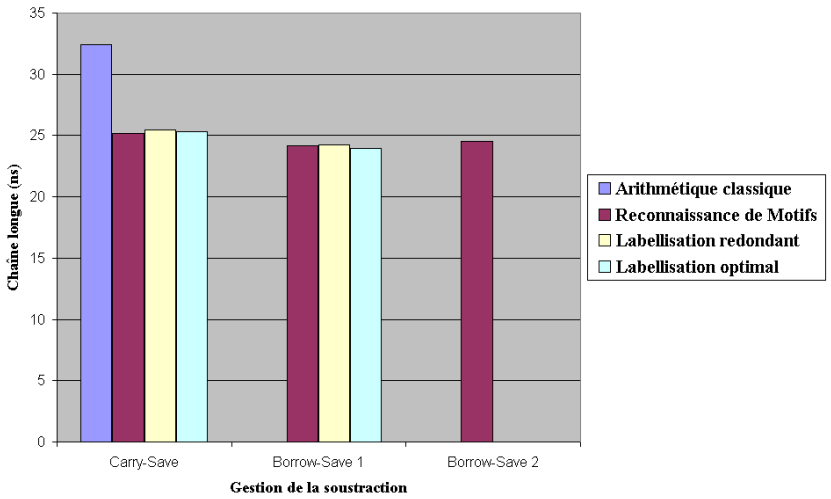
D'un point de vue temps d'exécution, on constate que l'algorithme de labellisation redondant dès que possible est le plus rapide des trois.

Dans l'article [DBM01], différentes implémentations d'une *DCU* ont été faites "à la main", en arithmétique classique et en arithmétique redondante. Les conclusions présentées se résument à une optimisation du délai allant de 10% à 15% pour un surcoût en surface limité à 17%. De notre côté, le délai est amélioré de 21.4% à 26.1% selon l'algorithme utilisé, avec une surface tantôt améliorée (-1.3%), tantôt dégradée (+5.2%). Nous pouvons en conclure que nos différents algorithmes produisent un meilleur délai couplé à un meilleur produit surface.délai.



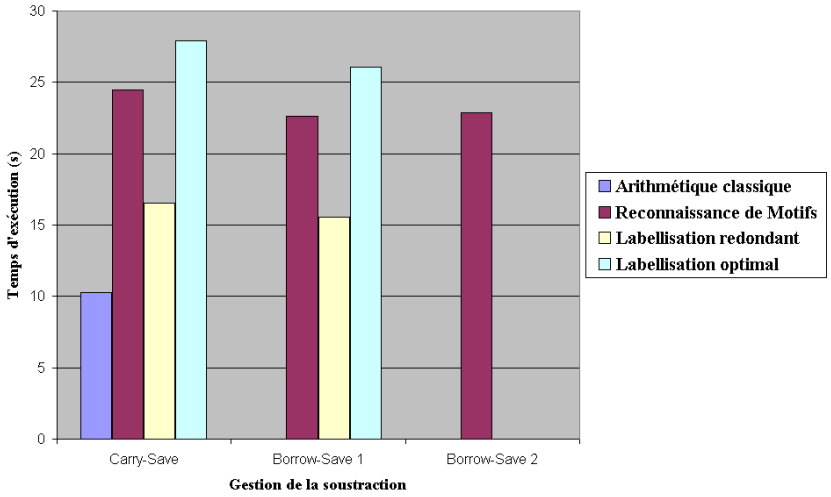
	Arithmétique classique	Reconnaissance de Motifs		Labellisation Redondant		Labellisation Optimal	
	μm^2	μm^2	%	μm^2	%	μm^2	%
Carry-Save	81 383	96 150	+18.1%	85 581	+5.1%	85 581	+5.1%
Borrow-Save 1	-	80 329	-1.3%	82 332	+1.2%	82 332	+1.2%
Borrow-Save 2	-	82 332	+1.2%	-	-	-	-

FIG. 6.7 – Résultats de la DCU : Surface



	Arithmétique classique	Reconnaissance de Motifs		Labelllisation Redondant		Labellisation Optimal	
	ns	ns	%	ns	%	ns	%
Carry-Save	32.43	25.18	-22.4%	25.48	-21.4%	25.32	-21.9%
Borrow-Save 1	-	24.14	-25.6%	24.24	-25.2%	23.97	-26.1%
Borrow-Save 2	-	24.55	-24.3%	-	-	-	-

FIG. 6.8 – Résultats de la DCU : Chaîne longue



	Arithmétique classique	Reconnaissance de Motifs		Labellisation Redondant		Labellisation Optimal	
	s	*s*	%	*s*	%	*s*	%
Carry-Save	9.62	23.12	+140.3%	15.46	+60.7%	25.79	+168.1%
Borrow-Save 1	-	23.04	+139.5%	14.12	+46.8%	24.71	+156.9%
Borrow-Save 2	-	24.15	+151%	-	-	-	-

FIG. 6.9 – Résultats de la DCU : Temps d'exécution

6.3 Opérateur de DCT

Différentes architectures de *DCT* (Discrete Cosinus Transform) existent. Nous nous sommes intéressés plus particulièrement au graphe de Loeffer [CLM89, DCM00]. Deux versions différentes de ce graphe ont été optimisées :

- une version avec 40 opérateurs arithmétiques (permettant d'obtenir tous les résultats en 1 cycle),
- une version avec 12 opérateurs arithmétiques (et un séquenceur) (permettant d'obtenir tous les résultats en 8 cycles).

Cet opérateur, quelque soit son implantation, possède de nombreux opérateurs avec sorties multiples, il est donc un bon exemple pour tester les trois comportements de la reconnaissance de motifs en cas de sorties multiples. Il contient également des soustractions, nous testons donc les trois façons de gérer cette opération.

6.3.1 Partition

L'architecture de la partition de la *DCT* est montrée dans la Figure 6.10.

Les résultats sont présentés dans les Figures 6.11 à 6.13. Ils proviennent de l'optimisation d'une *DCT* avec les données sur 8 bits, les coefficients sur 12 bits et les résultats sur 16 bits.

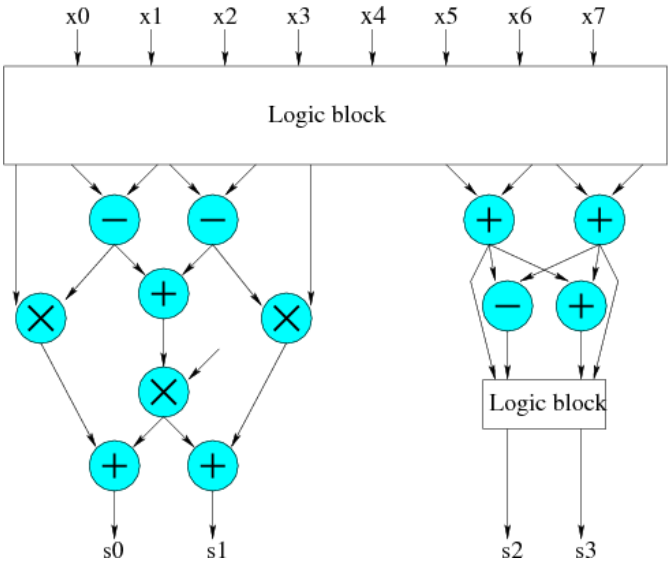


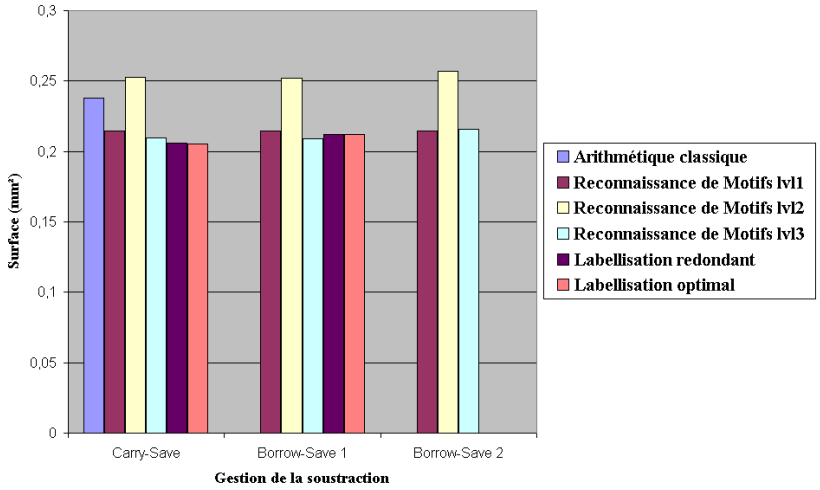
FIG. 6.10 – Partition de la DCT

En ce qui concerne la reconnaissance de motifs, ces résultats sont concordants avec la théorie : les optimisations sans et avec partage des ressources (*lvl2* et *lvl3*) donnent de meilleures chaînes longues que l'optimisation des arbres séparément (*lvl1*), tandis que l'optimisation des arbres séparément et l'optimisation avec partage des ressources donnent de meilleures surface que l'optimisation sans partage des ressources.

En ce qui concerne les algorithmes de labellisation, notons tout d'abord qu'ils donnent sensiblement les mêmes résultats. Le choix est donc fait par l'algorithme d'allocation optimale de transformer tous les opérateurs possibles en leur version redondante. Ces deux algorithmes donnent des résultats similaires aux meilleurs résultats fournis par la reconnaissance de motifs (proches de l'optimisation des arbres avec partage des ressources).

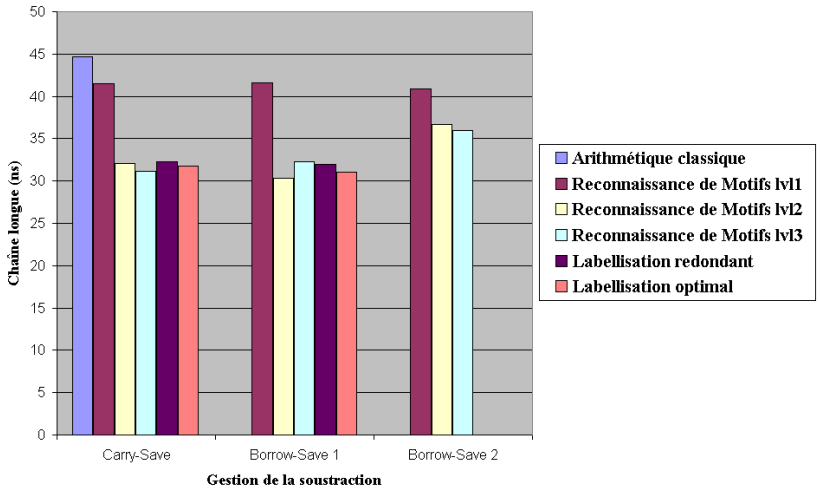
Par ailleurs, en ce qui concerne l'utilisation des architectures Borrow-Save, les résultats ne sont pas particulièrement plus performants qu'avec l'utilisation des architectures Carry-Save. Cela confirme le fait que le plus important est le contexte, et qu'il existe donc des cas pour lesquels les architectures Borrow-Save ne sont pas les plus optimales.

D'un point de vue temps d'exécution, on voit que l'algorithme de labellisation redondant dès que possible est, de loin, le plus rapide de tous. L'algorithme d'allocation optimale est lui le plus lent. Quant à la reconnaissance de motifs, elle est de plus en plus lente, plus l'option choisie en fonction des sorties multiples est complexe. En effet, là où seuls 3 motifs peuvent être remplacés quand on se limite à l'optimisation des arbres séparément (exemple de l'optimisation Carry-Save), l'optimisation sans partage de ressource permet d'en remplacer 12 (avec un surcoût de 9 instances dupliquées), ainsi que l'optimisation avec partage des ressources (qui contient également le post traitement permettant de fusionner 2 instances dupliquées inutilement).



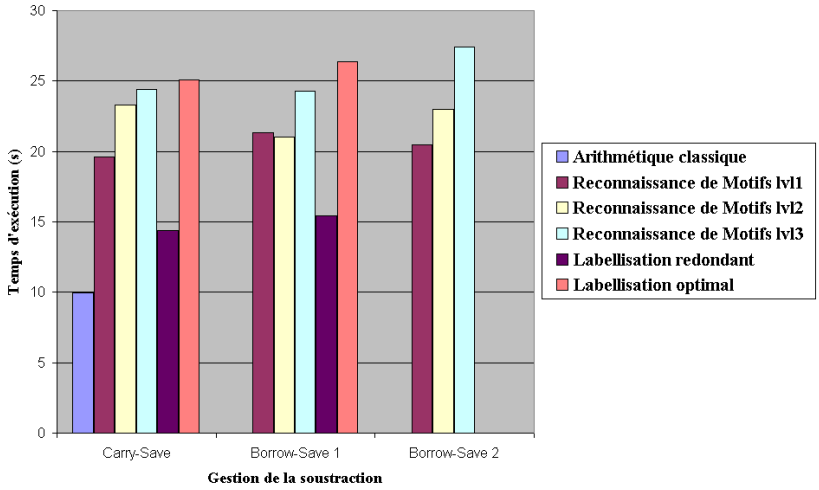
	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	mm^2	mm^2	%	mm^2	%	mm^2	%	mm^2	%	mm^2	%
CS	0.24	0.21	-12.5%	0.25	+4%	0.21	-12.5%	0.20	-16.7%	0.20	-16.7%
BS 1	-	0.21	-12.5%	0.25	+4%	0.21	-12.5%	0.21	-12.5%	0.21	-12.5%
BS 2	-	0.21	-12.5%	0.25	+4%	0.22	-8.3%	-	-	-	-

FIG. 6.11 – Résultats de la partition de la DCT : Surface



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	ns	ns	%	ns	%	ns	%	ns	%	ns	%
CS	44.67	41.47	-7.2%	32.09	-28.2%	31.18	-30.2%	32.29	-27.7%	31.72	-29%
BS 1	-	41.6	-6.9%	30.3	-32.2%	32.24	-27.8%	31.92	-28.5%	31.04	-30.5%
BS 2	-	40.91	-8.4%	36.63	-18%	35.91	-19.6%	-	-	-	-

FIG. 6.12 – Résultats de la partition de la DCT : Chaîne longue



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	s	*s*	%	*s*	%	*s*	%	*s*	%	*s*	%
CS	9.95	19.6	+97%	23.3	+134%	24.4	+145%	14.4	+45%	25.1	+152%
BS 1	-	21.35	+114%	21	+111%	24.3	+144%	15.4	+55%	26.4	+165%
BS 2	-	20.5	+106%	23	+131%	27.4	+175%	-	-	-	-

FIG. 6.13 – Résultats de la partition de la DCT : Temps d'exécution

Dans l'article [DCM00], différentes implémentations d'une *DCT* 2 dimensions (la combinaison de deux *DCT* 1 dimension avec adaptation des parties opératives) ont été faites "à la main", en arithmétique classique et en arithmétique redondante. Les conclusions présentées montrent une optimisation du délai de 20% et de la surface de 3%.

De notre côté, le délai est amélioré jusqu'à 32.2% selon l'algorithme utilisé, avec une surface tantôt améliorée (jusqu'à -16.7%), tantôt dégradée (+4%).

Nous pouvons en conclure que nos différents algorithmes permettent, selon le cas, d'optimiser de façon plus performante, l'un ou l'autre des deux critères.

6.3.2 Opérateur complet

L'architecture de l'opérateur complet de la *DCT* est montrée dans la Figure 6.14.

Les résultats sont présentés dans les Figures 6.15 à 6.17. Ils proviennent de l'optimisation d'une *DCT* avec les données sur 8 bits, les coefficients sur 14 bits et les résultats sur 16 bits.

En ce qui concerne la surface, tous les algorithmes sauf l'optimisation sans partage des ressources de la reconnaissance de motifs donnent une surface équivalente à celle d'origine. L'optimisation sans partage des ressources conduit elle, dans le pire des cas, à une surface preque doublée.

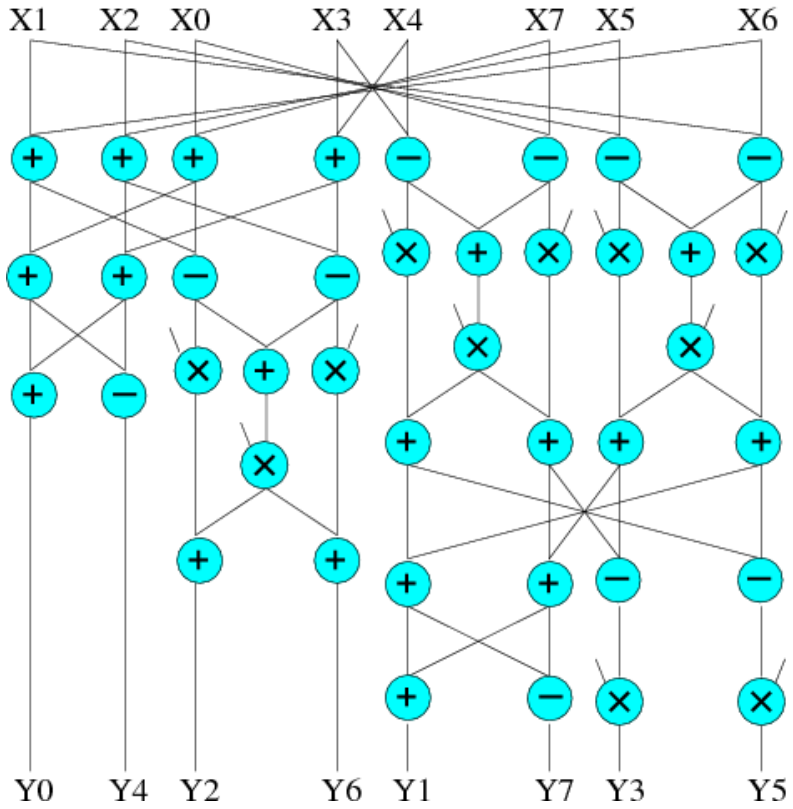


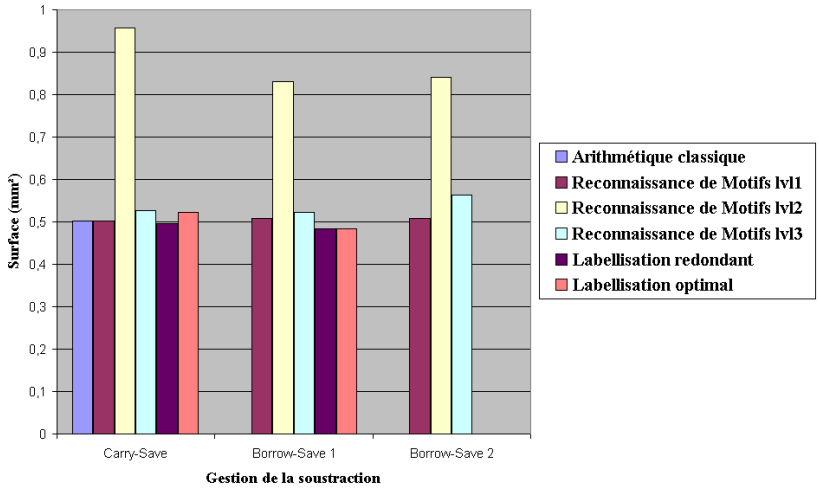
FIG. 6.14 – Opérateur de DCT

En ce qui concerne le délai, tous les algorithmes conduisent à une optimisation plus ou moins importante, les meilleurs résultats provenant des algorithmes de labellisation, et de l'optimisation avec partage des ressources.

Par aileurs, les résultats des algorithmes de labellisation sont similaires ; en effet, le choix est fait lors de l'algorithme d'allocation optimale de passer tous les opérateurs arithmétiques possibles en leur version redondante.

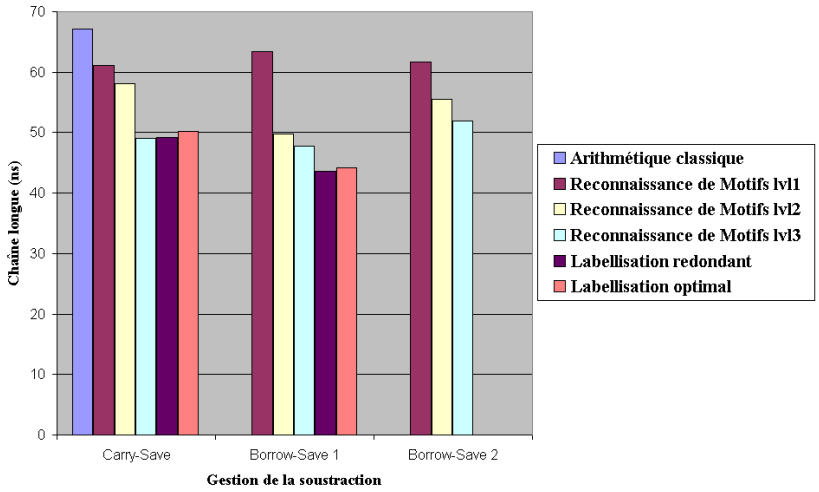
En ce qui concerne la gestion de la soustraction, on peut voir que dans ce cas, l'utilisation d'architectures Borrow-Save (avec additionneurs avec sorties Carry-Save) mène aux meilleures performances.

D'un point de vue temps d'exécution, c'est cette fois l'optimisation avec partage des ressources qui est la plus coûteuse. Cela est dû au post traitement visant à fusionner les instances inutilement dupliquées, qui devient de plus en plus coûteux plus le circuit sur lequel on travaille est grand (72 motifs remplacés au lieu de 11, 80 instances dupliquées et 33 instances fusionnées). Les quatre autres algorithmes ont un temps à peu près équivalent (avec cependant toujours l'algorithme d'allocation optimale un peu plus lent que l'algorithme redondant dès que possible, et l'optimisation sans partage des ressources un peu plus lente que l'optimisation des arbres séparément).



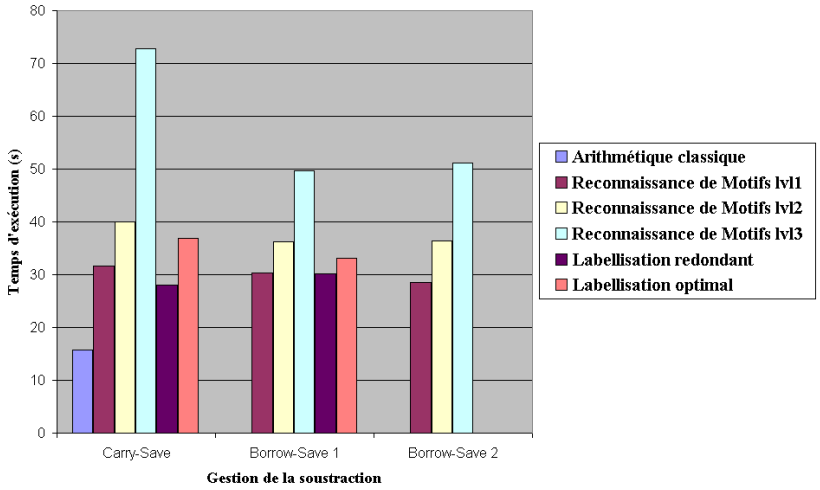
	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	mm^2	mm^2	%	mm^2	%	mm^2	%	mm^2	%	mm^2	%
CS	0.50	0.50	0%	0.96	+92%	0.53	+6%	0.50	0%	0.52	+4%
BS 1	-	0.51	+2%	0.83	+66%	0.52	+4%	0.48	-4%	0.48	-4%
BS 2	-	0.51	+2%	0.84	+68%	0.56	+12%	-	-	-	-

FIG. 6.15 – Résultats de la DCT complète : Surface



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	ns	*ns*	%	*ns*	%	*ns*	%	*ns*	%	*ns*	%
CS	67.06	61.04	-9%	58.11	-13.3%	49.09	-26.8%	49.18	-26.7%	50.23	-25.1%
BS 1	-	63.39	-5.5%	49.71	-25.9%	47.83	-28.7%	43.57	-35%	44.21	-34.1%
BS 2	-	61.75	-7.9%	55.50	-17.2%	51.99	-22.5%	-	-	-	-

FIG. 6.16 – Résultats de la DCT complète : Chaîne longue



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	s	*s*	%	*s*	%	*s*	%	*s*	%	*s*	%
CS	15.8	31.6	+100%	40.1	+153.8%	72.8	+360.7%	28	+77.2%	36.9	+133.5%
BS 1	-	30.3	+91.8%	36.2	+129.1%	49.6	+213.9%	30.1	+90.5%	33.1	+109.5%
BS 2	-	28.5	+80.4%	36.4	+130.4%	51.2	+224%	-	-	-	-

FIG. 6.17 – Résultats de la DCT complète : Temps d'exécution

6.4 Transformée de Fourier

6.4.1 le "papillon"

Le "papillon" est l'élément de base de l'opérateur de la transformée de Fourier (notée *FFT* pour *Fast Fourier Transform*). Son architecture est présentée dans la Figure 6.18.

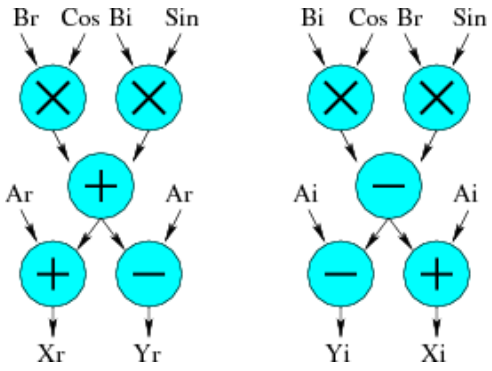


FIG. 6.18 – Architecture d'un papillon FFT

Cet opérateur contient des soustractions, nous présentons donc les résultats obtenus avec les différentes façon de gérer cet opérateur. Il possède également des opéra-

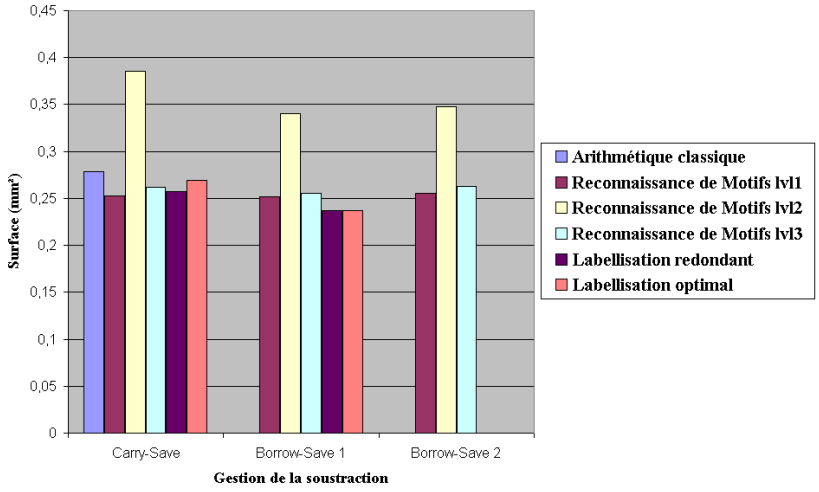
teurs possédant des sorties multiples. Nous présentons donc les différents comportements possibles avec la reconnaissance de motif.

Les résultats sont présentés dans les Figures 6.19 à 6.21. Il proviennent de l'optimisation d'un papillon avec les données sur 16 bits et les résultats sur 32 bits.

En ce qui concerne la surface, il y a une légère amélioration, sauf pour l'optimisation sans partage des ressources. Par ailleurs, les performances sont meilleures avec l'utilisation des architectures utilisant la notation Borrow-Save.

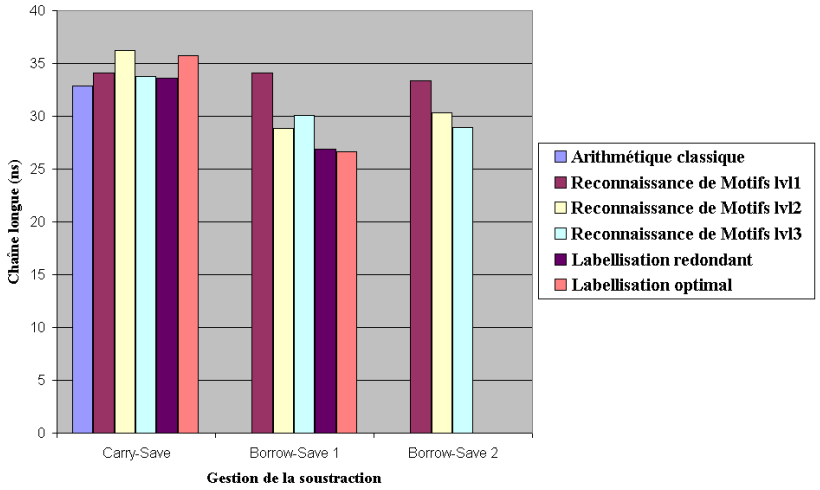
Cela est encore plus prononcé en ce qui concerne la chaîne longue : cet opérateur est un exemple typique montrant une dégradation de la chaîne longue avec l'utilisation des architectures Carry-Save, et une bonne optimisation dès que des architectures Borrow-Save sont utilisées.

D'un point de vue temps d'exécution, les conclusions sont les mêmes que pour tous les autres opérateurs du même ordre de grandeur : c'est l'algorithme d'allocation optimale qui est le plus lent, l'algorithme de labellisation redondant dès que possible est le plus rapide, et la reconnaissance de motifs entre les deux, étant de plus en plus lente en fonction de la gestion des sorties multiples.



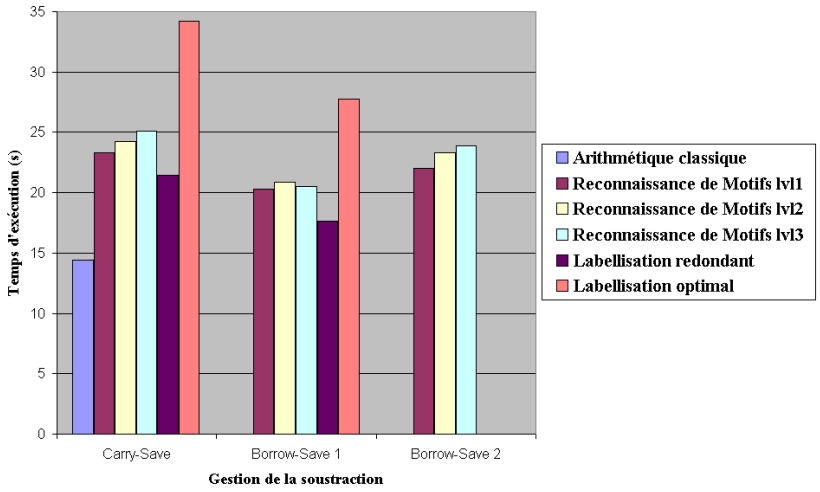
	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	mm^2	mm^2	%	mm^2	%	mm^2	%	mm^2	%	mm^2	%
CS	0.28	0.25	-10.7%	0.39	+39.3%	0.26	-7.1%	0.26	-7.1%	0.27	-3.6%
BS 1	-	0.25	-10.7%	0.34	+21.4%	0.25	-10.7%	0.24	-14.3%	0.24	-14.3%
BS 2	-	0.25	-10.7%	0.35	+25%	0.26	-7.1%	-	-	-	-

FIG. 6.19 – Résultats du papillon : Surface



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	ns	*ns*	%	*ns*	%	*ns*	%	*ns*	%	*ns*	%
CS	32.85	34.06	+3.7%	36.27	+10.4%	33.79	+2.9%	33.59	+2.2%	35.7	+8.7%
BS 1	-	34.11	+3.8%	28.83	-12.2%	30.06	-8.5%	26.92	-18%	26.67	-18.8%
BS 2	-	33.39	+1.6%	30.34	-7.6%	28.94	-11.9%	-	-	-	-

FIG. 6.20 – Résultats du papillon : Chaîne longue



	Arithmétique classique	Reconnaissance de Motifs						labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	s	*s*	%	*s*	%	*s*	%	*s*	%	*s*	%
CS	14.44	23.28	+61.2%	24.27	+68.1%	25.12	+74%	21.44	+48.5%	34.21	+136.9%
BS 1	-	20.28	+40.4%	20.84	+44.3%	20.54	+42.2%	17.62	+22%	27.73	+92%
BS 2	-	22.02	+52.5%	23.34	+61.6%	23.87	+65.3%	-	-	-	-

FIG. 6.21 – Résultats du papillon : Temps d'exécution

6.4.2 FFT

A partir de l'élément de base qu'est le papillon, différentes architectures existent de façon à obtenir un opérateur *FFT*. Chacune de ces architectures correspond à un algorithme particulier. Parmi les algorithmes existants, seize ont été analysés dans [Meh91], répartis en deux catégories : les algorithmes à entrelacement temporel et les algorithmes à entrelacement fréquentiel.

Nous nous sommes intéressés plus particulièrement à un des algorithmes à entrelacement temporel, dont l'architecture est montrée dans la Figure 6.22 (Version avec 8 entrées/sorties). Cet algorithme est un algorithme à géométrie constante. Les accès nœuds sources et nœuds puits sont séquentiels. Les entrées sont désordonnées, les sorties et les coefficients ordonnés.

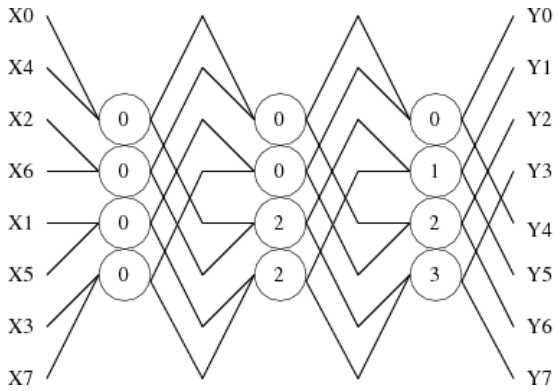


FIG. 6.22 – Architecture d'une FFT

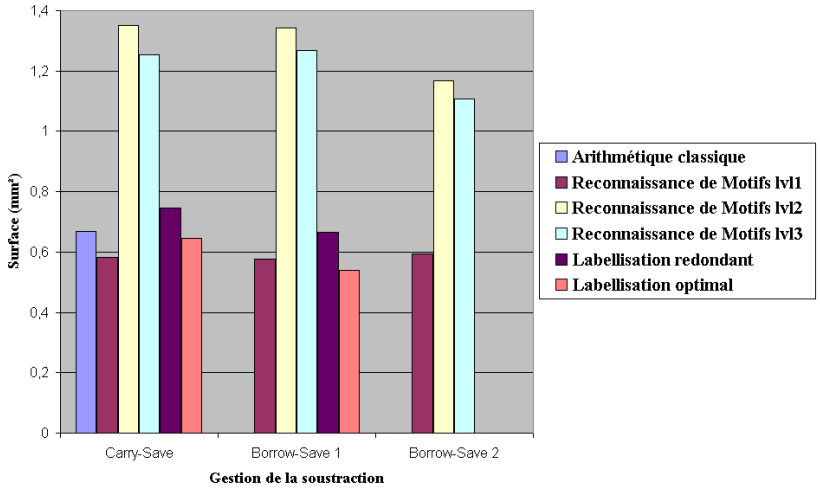
Comme précédemment, nous présentons les résultats obtenus avec tous les algorithmes à notre disposition, en ce qui concerne le traitement de la soustraction et celui des sorties multiples.

Les résultats sont présentés dans les Figures 6.23 à 6.25. Il proviennent de l'optimisation d'une *FFT* avec les données et coefficients sur 4 bits et les résultats sur 16 bits.

Ce circuit contient douze opérateurs de type papillon. Les ordres de grandeurs ne sont pas du tout les mêmes que précédemment.

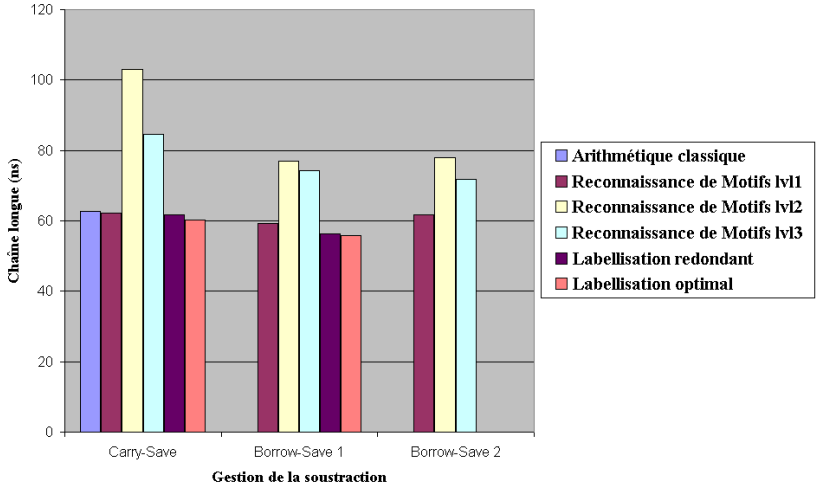
Commençons par la surface. L'optimisation sans partage des ressources dégrade la surface comme précédemment. Par contre ici, l'optimisation avec partage des ressources n'est guère meilleure. On voit avec cet exemple les limites d'une approche dédoublement des opérateurs / optimisation / fusion des opérateurs. Pour ce qui est de l'algorithme de labellisation redondant dès que possible, il dégrade la surface avec utilisation des architectures Carry-Save et l'améliore avec utilisation des architectures Borrow-Save. L'algorithme d'allocation optimale lui l'améliore dans tous les cas.

En ce qui concerne la chaîne longue, les algorithmes agrandissant la surface engendrent également des chaînes longues augmentées. Cette conclusion n'est pas justifiée d'un point de vue arithmétique, c'est l'optimisation des arbres séparément qui devrait



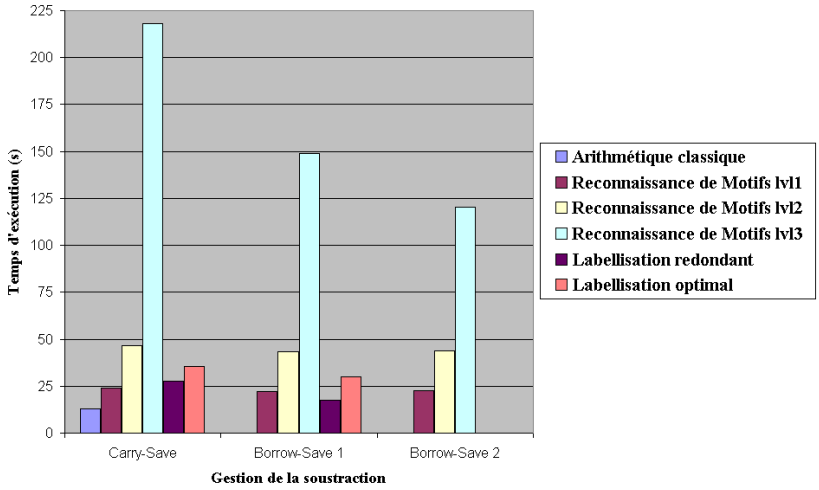
	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	mm^2	mm^2	%	mm^2	%	mm^2	%	mm^2	%	mm^2	%
CS	0.67	0.58	-13.4%	1.35	+101.5%	1.25	+86.6%	0.75	+11.9%	0.65	-3%
BS 1	-	0.57	-14.9%	1.34	+100%	1.27	+89.5%	0.66	-1.5%	0.54	-19.4%
BS 2	-	0.59	-11.9%	1.17	+74.6%	1.11	+54.7%	-	-	-	-

FIG. 6.23 – Résultats de la FFT : Surface



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	ns	ns	%	ns	%	ns	%	ns	%	ns	%
CS	62.64	62.25	-0.6%	102.91	+64.3%	84.54	+35%	61.82	-1.3%	60.16	-4%
BS 1	-	59.29	-5.3%	76.91	+22.8%	74.34	+18.7%	56.35	-10%	55.71	-11%
BS 2	-	61.77	-1.4%	77.87	+24.3%	71.9	+14.8%	-	-	-	-

FIG. 6.24 – Résultats de la FFT : Chaîne longue



	Arithmétique classique	Reconnaissance de Motifs						Labellisation Redondant		Labellisation Optimal	
		Lvl1		Lvl2		Lvl3					
	s	*s*	%	*s*	%	*s*	%	*s*	%	*s*	%
CS	13.11	24.14	+84.1%	46.74	+256.5%	218.22	+1564%	27.81	+112.1%	35.45	+170.4%
BS 1	-	22.28	+69.9%	43.27	+230%	148.76	+1034%	17.68	+34.8%	30	+128.8%
BS 2	-	22.73	+73.4%	43.88	+234.7%	120.23	+817%	-	-	-	-

FIG. 6.25 – Résultats de la FFT : Temps d'exécution

donner les moins bons résultats (ce sont d'ailleurs les moins bons résultats parmi les autres algorithmes). Cependant, d'un point de vu pratique, on travaille ici sur un circuit suffisamment gros pour que les phases de placement / routage influencent fortement les performances temporelles ; ces deux optimisations génèrent des architectures "pénalisées" par leur surface (presque double) dans lesquelles le temps passé dans les fils est très important comparé à des architectures plus petites. Pour ce qui est des trois autres algorithmes, c'est l'algorithme d'allocation optimale qui donne les meilleures résultats, et l'utilisation des architectures Borrow-Save.

Notons par ailleurs que, que ce soit du point de vue de la surface ou de la chaîne longue, l'algorithme d'allocation optimale donne ici des performances différentes de celles obtenues avec l'algorithme de labellisation redondant dès que possible. Cela est dû au fait que plusieurs instances restent dans leur version classique : 32 sur 60 par exemple pour l'optimisation Borrow-Save.

L'analyse des temps d'exécution confirme la conclusion qui avait été émise lors de l'étude de la *DCT* complète : pour les gros circuits, l'algorithme d'allocation optimale est beaucoup plus rapide que l'optimisation avec partage des ressource de la reconnaissance de motifs. L'algorithme de labellisation redondant dès que possible est comme toujours le plus rapide.

6.5 Conclusion

Nous avons présenté l'utilisation de nos différents algorithmes sur plusieurs opérateurs arithmétiques.

Les différentes architectures ainsi obtenues ont été comparées, entre elles, et par rapport à des architectures n'utilisant pas l'arithmétique redondante.

Nous avons ainsi pu montrer que de façon générale, nous obtenons des performances intéressantes, tant du point de vue de la surface que de celui de la chaîne longue.

En ce qui concerne l'utilisation des architectures Borrow-Save lors de l'optimisation de circuits contenant des soustractions, on peut conclure qu'elle mène, dans le pire des cas, à des performances identiques à celles obtenues sans utiliser ces architectures. Dans la plupart des cas, les performances obtenues sont meilleures, et ce, quelque soit l'algorithme utilisé.

En ce qui concerne la gestion des opérateurs avec sorties multiples, notre conclusion est que l'optimisation avec partage des ressources est celle qui produit les meilleures performances. C'est en total accord avec notre étude théorique.

Enfin, en ce qui concerne plus particulièrement les algorithmes que nous avons mis en place, les deux algorithmes de labellisation apparaissent comme une meilleure approche que la reconnaissance de motifs.

En effet, pour pouvoir comparer les différents algorithmes entre eux, il faut comparer l'optimisation avec partage des ressources. Cette version de la reconnaissance de motifs est beaucoup trop coûteuse, et ne conduit pas à des performances optimales.

La reconnaissance de motifs est donc une approche intéressante, mais est plutôt destinées aux circuits ne possédant pas d'opérateurs avec sorties multiples. En ce qui concerne le filtre par exemple, on a pu voir que cette approche permet des modifications d'architectures plus importantes que celles des deux autres algorithmes, et qu'elle peut donc amener à des performances meilleures.

Parmi les deux algorithmes de labellisation, l'approche *tout en redondant* allie de bons résultats et un temps de calcul très peu important. L'approche cherchant la solution optimale quant à elle amène également à de bonnes performances, avec un temps de calcul généralement doublé. Dans la plupart des cas, la chaîne longue est légèrement meilleure et la surface un peu dégradée par rapport à l'autre approche.

Enfin nous avons pu constater que nos algorithmes génèrent des architectures ayant des surfaces et chaînes longues meilleures que celles faites à la main (pour les circuits pour lesquels cette comparaison a pu être faite). Si l'on prend également en compte le temps nécessaire à la conception de ces circuits à la main, et les connaissances en arithmétique nécessaires, on en conclut aisément l'intérêt de tels outils automatiques.

Conclusions et perpectives

Conclusions

Le but premier de cette thèse était d'automatiser l'utilisation de l'arithmétique redondante dans le flot de conception de circuits de façon à la rendre plus accessible aux concepteurs de circuits n'ayant pas forcément le savoir-faire arithmétique nécessaire. Cela s'est fait en deux étapes.

La première étape a consisté à mettre en place l'environnement de conception. En effet, plusieurs points sont nécessaires à la mise en œuvre d'outils d'optimisation arithmétique.

Tout d'abord, nous nous sommes intéressés à la structure de données nécessaire à la description de circuit et répondant aux besoins de l'arithmétique. Nous avons ensuite développé un langage de description de circuit basé sur cette structure de données. Ce langage a été conçu de façon à être le plus clair et intuitif possible de façon à faciliter le travail des concepteurs.

En effet, au delà de la notion d'optimisation automatique, notre but était de fournir un cadre de conception qui rende la conception de chemins de données arithmétiques simple et rapide. En ce sens, nous avons fourni un langage de description ayant un haut niveau d'abstraction permettant de considérer les opérateurs arithmétiques comme de simples mnémoniques. Cela permet aux concepteurs de s'affranchir des problèmes d'architectures d'opérateurs ou de dynamiques des opérandes.

Une fois l'environnement constitué, nous avons mis en place une bibliothèque d'opérateurs arithmétiques paramétrables, indispensable au développement de chemins de données arithmétiques, et à l'utilisation d'outils d'optimisation.

La seconde étape a été consacrée à la mise en place des algorithmes d'optimisation à proprement parler.

Trois algorithmes ont été implantés, le premier à base de reconnaissance de motifs, les deux autres à base de labellisation de graphe.

Chacun de ces algorithmes permet de traiter des circuits pouvant être modélisés sous forme d'arbre, mais également sous forme de graphe.

Par ailleurs, chacun de ces algorithmes fournit (au moins) deux façons différentes de prendre en compte les soustractions. En effet, il est d'usage courant d'effectuer cette opération en la substituant par une addition (addition du complément à deux du nombre à soustraire). Nous avons donc exploré cette voix. Nous avons fait le choix d'également explorer l'usage des architectures Borrow-Save. Cette étude nous a montré que cette approche apporte des résultats au moins aussi performants que ceux de la précédente approche, et dans le plupart des cas, nettement plus performants.

L'utilisation de ces différents algorithmes sur différents opérateurs arithmétiques nous a permis non seulement d'apprécier l'utilité de l'usage de l'arithmétique redondante, en comparant les performances des architectures obtenues à celles des architectures classiques. Cela nous a permis également de comparer les différentes approches entre elles.

Si l'on se réfère aux perspectives de la thèse de Yannick Dumonteix concernant l'aide à la conception, il faisait référence à la mise au point d'une version logicielle de l'aide à la conception qu'il avait présenté. En effet, rappelons le, cette thèse s'est déroulée comme étant le prolongement de la thèse de Yannick Dumonteix, qui avait étudié l'impact de l'introduction de l'arithmétique redondante dans le flot de conception *VLSI*. L'environnement de conception conçu a largement été inspiré de son travail, ainsi que la façon d'utiliser l'arithmétique redondante. Notre travail peut être décrit comme l'expression de ses réflexions faites sur l'arithmétique.

Cependant, outre la mise au point de l'environnement ainsi que des outils d'optimisation, notre contribution s'est située dans la recherche des différents algorithmes d'optimisation. En particulier, nous avons développé et évalué une approche innovante consistant à ne pas transformer de façon systématique tous les opérateurs arithmétiques d'un circuit en leur version redondante, mais utiliser une fonction de coût permettant de choisir la meilleure architecture possible pour chaque opérateur en fonction d'un critère précis. Cette approche a été mise en place avec pour critère le temps de propagation.

Perspectives

Arithmétique

Seul l'aspect "utilisation d'opérateurs redondants" a été pris en compte dans cette thèse. Les particularités de la somme et les propriétés de commutativité et d'associativité n'ont pas été étudiées. Ces particularités peuvent permettre de nouvelles optimisations.

Tout d'abord les regroupements et les fusions, évoqués dans la Section 1.1.2. Ces deux mécanismes exploitent les propriétés de l'addition de façon à pouvoir utiliser des opérateurs de somme et sont à utiliser après le passage en redondant. Cependant, ils requièrent une analyse des temps de propagations pour être efficace. Il faudrait donc trouver un algorithme, comparable à l'algorithme d'allocation optimale mis en place, cherchant l'utilisation optimale des sommateurs.

Deuxième mécanisme intéressant, le glissement, comme évoqué dans la Section 1.1.3. Cette redistribution des ressources logiques permet de dégager des portions arithmétiques plus importantes pour permettre davantage d'optimisations arithmétiques. Un second algorithme mettant en place automatiquement ce mécanisme serait également appréciable.

Par ailleurs, nous avons évoqué dans le Chapitre 3 l'extension de la représentation Carry-Save, dite représentation Relax Carry-Save, qui permet à chaque bit d'un nombre d'être codé sur 1, 2 ou 3 bits. Il serait intéressant d'évaluer les possibles optimisations apportées par l'usage de cette nouvelle représentation. Avec des conclusions positives, on pourrait dans un premier temps envisager étendre cette nouvelle façon de faire à la représentation Borrow-Save. Dans un second temps, il nous faudrait trouver la façon d'automatiser l'utilisation de ces nouvelles représentations.

Au delà des systèmes de représentations classiques et redondants, il pourrait être intéressant de faire l'étude de l'introduction d'autres systèmes : les systèmes flottants et les systèmes par recodage de résidus pour ne citer qu'eux.

Les premiers ne répondent pas aux mêmes besoins que les systèmes classiques et redondants, d'autant plus qu'ils sont normés, ce qui les rend rigides. Les seconds répondent eux aux mêmes besoins, et posent des problèmes similaires (coût meilleures performances / conversions, ...). Ils paraissent donc être les plus intéressants à traiter.

Ce travail devrait reprendre le travail fait pour l'arithmétique redondante, en commencant par l'étude de l'arithmétique *RNS*, l'élaboration des opérateurs élémentaires, et enfin, l'insertion de l'arithmétique *RNS* dans l'arithmétique mixte. Une fois ce travail fait, la seconde phase serait d'introduire cette arithmétique dans nos différents algorithmes.

Optimisation

Nous nous sommes focalisés dans ce manuscrit à deux critères d'évaluation des performances que sont le délai et la surface. Nous avons laissé de côté l'aspect consommation. Ce critère devient cependant de plus en plus important depuis quelques années.

Dans un premier temps il serait intéressant d'évaluer la consommation intrinsèque des différents opérateurs arithmétiques. De bons résultats inciteraient à évaluer ce critère sur différents circuits arithmétiques et tirer des conclusions sur la dépendance entre consommation d'un circuit et consommation des différents opérateurs le composant. On en tirerait alors des conclusions sur l'intérêt des algorithmes *redondants dès que possible* en ce qui concerne la consommation.

Dans un second temps, il faudrait s'intéresser à ce critère dans notre algorithme d'allocation optimale. Il nous faudrait pour cela effectuer la même démarche que celle faite pour le temps de propagation : déterminer les fonctions d'évaluation des différents opérateurs arithmétiques et déterminer la fonction de coût permettant d'effectuer le choix entre différents architectures.

Par ailleurs, notre algorithme d'allocation optimale est un algorithme mono-critère. Une alternative intéressante serait d'adapter cet algorithme à une version bi-critères, voire multi-critères. Cela permettrait de prendre en compte en parallèle les différents critères qui nous intéressent, et non plus se focaliser sur un seul. Mais cela demanderait

de mettre en œuvre des algorithmes d'optimisation de graphes bien plus complexes.

Un point non négligeable également serait de mettre en place une gestion de la hiérarchie. En effet, nos algorithmes ne sont adaptés qu'aux circuits *à plat* i.e. instanciant directement les différents opérateurs arithmétiques.

Il faudrait s'inspirer du principe présenté dans [UKL99] et détaillé dans la Section 3.1.

Une dernière voie à explorer serait de transposer notre travail au monde des *FPGA* [NDLG99, GGN04]. En effet, nous nous sommes focalisés dans ce manuscrit au monde des *ASIC*. De récents travaux utilisent des *CSA* dans les *FPGA* [PABI08a, PABI08b] et présentent des résultats prometteurs.

Nous avons mis en avant la rapidité de l'évolution technologique couplée avec la complexité croissante des circuits à concevoir. Dans cette optique, l'avantage des *FPGA* est leur grande souplesse qui permet de les réutiliser à volonté dans différents algorithmes en un temps très court. Le progrès de ces technologies permet de faire des composants toujours plus rapides et à plus haute intégration, ce qui permet de programmer des applications importantes. On ciblera, en particulier, des circuits *FPGA* dits "à grains variables" contenant des blocs durs d'opérateurs avec des représentations mixtes [PMFM08].

Dans l'optique d'adapter nos algorithmes aux *FPGA*, rappelons que, toujours dans un but de rapidité de conception, nous avons mis en avant l'idée de portabilité en choisissant de générer des circuits optimisés en portes *virtuelles* indépendantes de la technologie cible.

Du point de vue de nos algorithmes *redondant dès que possible*, il suffirait alors de s'assurer que la synthèse vers circuits *FPGA* des architectures optimisées est possible. Il faudrait ensuite évaluer les résultats obtenus pour en tirer des conclusions.

Du point de vue de l'algorithme d'allocation optimale, il faudrait adapter les fonctions d'évaluation des différents opérateurs arithmétiques et la fonction de coût au monde des *FPGA*.

Publications

[BCMM08] Sophie Belloeil, Roselyne Chotin-Avot, Habib Mehrez, Alix Munier-Kordon Automatic Allocation of Redundant Operators in Arithmetic Data path Optimization In *DASIP'2008 Conference on Design and Architectures for Signal and Image Processing*, Bruxelles, Belgique, novembre 2008, pp. 176-183

[BCM08] Sophie Belloeil, Roselyne Chotin-Avot, Habib Mehrez Arithmetic Data path Optimization using Borrow-Save Representation In *ISVLSI'2008 IEEE computer society annual symposium on VLSI*, Montpellier, France, avril 2008

[BDMCM07] Sophie Belloeil, Damien Dupuis, Christian Masson, Jean-Paul Chaput, Habib Mehrez Stratus : A procedural circuit description language based upon Python In *ICM'2007 International Conference on Microelectronics*, Le Caire, Egypte, décembre 2007, pp.275-278

[BCM07] Sophie Belloeil, Roselyne Chotin-Avot, Habib Mehrez, Data Path Optimization using Redundant Arithmetic and Pattern Matching, In *DASIP'2007 Workshop on Design and Architectures for Signal and Image Processing*, Grenoble, novembre 2007

[BCCMM06] Sophie Belloeil, Jean-Paul Chaput, Roselyne Chotin-Avot, Christian Masson, Habib Mehrez, Stratus : Un environnement de développement de circuits, In *Proc. JPCNFM'06 9es Journées pédagogiques du CNFM*, Saint Malo, novembre 2006, pp. 57-61

[BH05] Sophie Belloeil et Habib Mehrez, Optimisation de chemins de données par l'utilisation de l'arithmétique redondante, In *Proc. JNRDM'05 Journées Nationales du Réseau Doctoral en Microélectronique*, Paris, mai 2005, pp. 268-270

Bibliographie

[ACC+05] Christophe Alexandre, Hugo Clement, Jean-Paul Chaput, Marek Sroka, Christian Masson, and Remy Escassut. Tsunami : An integrated timing-driven place and route research platform. In *Procceedings of the Design, Automation, and Test in Europe (DATE'05)*, pages 920–921, 2005. 88

[AFT02] Boris D. Andreev, Eby G. Friedman, and Edward L. Titlebaum. Efficient implementation of a complex ± 1 multiplier. In *Procceedings of the* 12^{th} *ACM Great Lakes symposium on VLSI (GLSVLSI'02)*, pages 83–88, 2002. 27

[Ale07] Christohe Alexandre. *Coriolis : une plate-forme ouverte pour l'évaluation de flots de conception VLSI fortement intégrés*. PhD thesis, Université Pierre et Marie Curie Paris VI, septembre 2007. 88

[All] Alliance. http ://www-asim.lip6.fr/recherche/alliance. 87, 232

[AMP+08] A. Abril, H. Mehrez, F. Pétrot, J. Gobert, and C. Miro. Estimation et optimisation de la consommation dans les soc utilisant la simulation précise au cycle. *Technique et science informatiques*, 27(1-2) :203–233, 2008. 3

[Avi61] A. Avizienis. Signed-digit number representation for fast parallel arithmetic. *IRE Trans. Electronic Computers*, 10 :389–400, 1961. 1, 21

[Cad] Cadence. http ://www.cadence.com. 86

[CLM89] A. Lightenberg C. Loeffler and G. Moschytz. Practical fast 1d-dct algorithls with 11 multiplications. In *Procceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'89)*, pages 988–991, 1989. 109

[CMRS04] A. Cilardo, A. Mazzeo, L. Romano, and G.P. Saggese. Carry-save montgomery modular exponentiation on reconfigurable hardware. In *Procceedings of the Design, Automation, and Test in Europe (DATE'04)*, page 30206, 2004. 34

[Cor] Coriolis. http ://www-asim.lip6.fr/recherche/coriolis. 2, 88

[Dad65] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 19 :349–356, 1965. 220

[DBM01] Yannick Dumonteix, Yann Bajot, and Habib Mehrez. A fast and low-power distance computation unit dedicated to neural networks, based on redundant arithmetic. In *Procceedings of IEEE International Symposium on Circuits and Systems (ISCAS'01)*, volume 4, pages 878–881, 2001. 19, 36, 106, 107

[dBNN04] Lirida Alves de Barros Naviner and Jean-François Naviner. On customized decimation filter implementation. In *Proceedings of IEEE International Conference on Industrial Technology (ICIT'04)*, pages 304–309, 2004. 103

[dBNNC03] Lirida A. de B Naviner, Jean-François Naviner, and Elizabeth Colin. Decimation filter cascade for channel selection in multistandard receivers. *2005 IEEE International Symposium on Micro-NanoMechatronics and Human Science*, 1 :245–248, décembre 2003. 103

[DCM00] Yannick Dumonteix, Roselyne Chotin, and Habib Mehrez. Use of redundant arithmetic on architecture and design of a high performance DCT macro-bloc generator. In *Proceedings of the* 15^{th} *Conference on Design of Circuits and Integrated Systems (DCIS'00)*, pages 428–433, 2000. 19, 35, 109, 112

[Dec04] Jan Decaluwe. Myhdl. a python-based hardware description language. *Linux J.*, 2004(127) :5, 2004. 90

[DM00] Yannick Dumonteix and Habib Mehrez. A family of redundant multipliers dedicated to fast computation for signal processing. In *Procceedings of IEEE International Symposium on Circuits and Systems (ISCAS'00)*, volume 5, pages 325–328, 2000. 27

[Dum01] Yannick Dumonteix. *Optimisations des chemins de données arithmétiques par l'utilisation de plusieurs systèmes de numérations*. PhD thesis, Université Pierre et Marie Curie Paris VI, octobre 2001. 2, 7, 21, 27, 33, 210, 212, 222, 223, 225, 227, 228

[GGN04] Khaled Grati, Adel Ghazel, and Lirida Naviner. Optimized fpga-based implementation of down-sampling filter for wide band radio receiver. In *Proceedings of IEEE International Conference on Industrial Technology (ICIT'04)*, pages 243–245, 2004. 126

[GP92] A. Greiner and F. Pecheux. Alliance : A complete set of cad tools for teaching vlsi design, 1992. 30, 87, 94, 103

[GP94] Alain Greiner and Frédéric Pétrot. Using c to write portable cmos vlsi module generators. In *Procceedings of the Conference on European Design Automation (EURO-DAC'94)*, pages 676–681, 1994. 88, 95

[Guy] Alain Guyot. Cours d'operateurs arithmetiques, virgule flottante. http ://tima-cmp.imag.fr/∼guyot/Cours/Arithmetique/flottant. 214

[Guy91] Alain Guyot. Ocapi : architecture of a vlsi coprocessor for the gcd and the extended gcd of large numbers. *Procceedings of IEEE* 10^{th} *Symposium on Computer Arithmetic (ARITH'91)*, pages 226–231, 1991. 34

[Hou97] Alain Houelle. *GenOptim : un environnement d'aide à la conception de générateurs de circuits portables optimisés en performance et en surface*. PhD thesis, Université Pierre et Marie Curie Paris VI, juin 1997. 89

[IV04] Paolo Ienne and Ajay K. Verma. Arithmetic transformations to maximise the use of compressor trees. In *Procceedings of IEEE International Workshop on Electronic Design, Test and Applications (DELTA'04)*, page 219, 2004. 46

[Jac99] Ludovic Jacomme. *Analyse sémantique de descriptions VHDL synchrones en vue de la synthèse*. PhD thesis, Université Pierre et Marie Curie Paris VI, octobre 1999. 49, 87

[JDK91] Y. Herreros J. Duprat and S. Kla. New redundant representations of complex numbers and vectors. In *Procceedings of IEEE 10^{th} Symposium on Computer Arithmetic (ARITH'91)*, pages 2–9, 1991. 25

[JHD] JHDL. http ://www.jhdl.org. 90

[KJT98a] Taewhan Kim, William Jao, and Steve Tjiang. Arithmetic optimization using carry-save-adders. In *Procceedings of the 35^{th} Design Automation Conference (DAC'98)*, pages 433–438, 1998. 37, 56

[KJT98b] Taewhan Kim, William Jao, and Steve Tjiang. Circuit optimization using carry-save-adder cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10) :974–984, 1998. 37, 38, 42

[KK01] Youngtae Kim and Taewhan Kim. Accurate exploration of timing and area trade-offs in arithmetic optimization using carry-save-adders. In *Procceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'01)*, 2001. 42, 64

[KU00a] Taewhan Kim and Junhyung Um. A practical approach to the synthesis of arithmetic circuits using carry-save adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(5) :615–634, 2000. 41

[KU00b] Taewhan Kim and Junhyung Um. A timing-driven synthesis of arithmetic circuits using carry-save-adders. In *Procceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'00)*, pages 313–316, 2000. 41

[KYW99] Kei-Yong Khoo, Zhan Yu, and Alan N. Willson Jr. Bit-level arithmetic optimization for carry-save additions. In *Procceedings of IEEE/ACM International Conference on Computer-Aider Design (ICCAD'99)*, pages 14–19, 1999. 45

[LCSJJ03] Ja-Sheng Liu, I-Hsin Chen, Yi-Chen Tsai Shyh-Jye, and Jou. Low-power digital cdma receiver. In *Procceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'03)*, pages 581–582, 2003. 34

[Mag92] Serge Maginot. Evaluation criteria of hdls : Vhdl compared to verilog, udl/i & m. In *Procceedings of the Conference on European Design Automation (EURO-DAC'92)*, pages 746–751, 1992. 87

[Meh91] Habib Mehrez. *Des architectures VLSI pipelines pour les algorithmes numeriques a flots de donnees en representations arithmetiques virgule fixe et virgule [Microforme]*. PhD thesis, 1991. 118

[MMMT04] Hayder Mrabet, Zied Marrakchi, Habib Mahrez, and André Tissot. Automatic layout of scalable embedded field programmable gate array. In *Procceedings of the International Conference on Electrical Electronic and Computer Engineering (ICEEC'04)*, pages 469–472, 2004. 97

[MMP96a] Anne Mignotte, Jean-Michel Muller, and Olivier Peyran. Mixed arithmetic : Introduction and design structure. In *2nd International conference on Massively Parallel Computing Systems*, 1996. 48

[MMP96b] Anne Mignotte, Jean Michel Muller, and Olivier Peyran. A model for using redundant number systems in special-purpose architectures. In *Proceedings of IMACS Multiconference Computational Engineering in Systems Applications (CESA'96)*, 1996. 47

[MMP97] Anne Mignotte, Jean Michel Muller, and Olivier Peyran. Synthesis for mixed arithmetic. *Research Report, LIP*, 1997. 48

[MP97a] A. Mignotte and O. Peyran. Reducing the complexity of ilp formulations for synthesis. In *Procceedings of the International Symposium on System Synthesis (ISSS'97)*, 1997. 48

[MP97b] A. Mignotte and O. Peyran. Scheduling using mixed arithmetic : an ilp formulation. In *Procceedings of the European Conference on Design and Test (EDCT'97)*, page 621, 1997. 48

[MP97c] Anne Mignotte and Olivier Peyran. Scheduling using mixed arithmetic : an ilp formulation. *Research Report, LIP*, 1997. 48

[Mul89] Jean Michel Muller. *Arithmétique des ordinateurs, opérateurs et fonctions élémentaires*. Masson, 1989. 13, 23, 220

[NDLG99] Lirida Naviner, Jean-Luc Danger, Charles Laurent, and Andres Gargia. Efficient implementation for high accuracy dct processor based on fpga. In *Proceedings of IEEE Midwest Symposium on Circuits and Systems (MWSCAS'99)*, pages 508–511, 1999. 126

[NDMT07] Ludovic Noury, François Durbin, Habib Mehrez, and André Tissot. A generic asic architecture for real time time-frequency analysis of non-stationary large bandwidth signals. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference (IMTC'2007)*, 2007. 97

[NMDT04] Ludovic Noury, Habib Mehrez, François Durbin, and André Tissot. Use of multiple numeration systems for architecture and design of a high performance fir filter netlist generator. In *Procceedings of International Conference on Microelectronics (ICM'04)*, pages 547–550, 2004. 103

[Nou08] Ludovic Noury. *Contribution à la conception de processeurs d'analyse de signaux à large bande dans le domaine temps-fréquence : l'architecture F-TFR*. PhD thesis, Université Pierre et Marie Curie Paris VI, juin 2008. 97

[PABI08a] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. Improving synthesis of compressor trees on fpgas via integer linear programming. In *Procceedings of the Design, Automation, and Test in Europe (DATE'08)*, pages 1256–1261, 2008. 126

[PABI08b] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. A novel fpga logic block for improved arithmetic performance. In *Proceedings of the 16th*

international ACM/SIGDA symposium on Field programmable gate arrays (FPGA'08), pages 171–180, 2008. 126

[Pey97] Olivier Peyran. *Synthèse d'architectures intégrées utilisant des arithmétiques redondantes*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire d'Informatique du Parallélisme de l'École Normale Supérieure de Lyon, décembre 1997. 47

[Pig04] Christian Piguet. *Low-power electronics design*. CRC Press, 2004. 3

[PMFM08] Husain Parvez, Zied Marrakchi, Umer Farooq, and Habib Mehrez. A new coarse-grained fpga architecture exploration environment. In *Proceedings of International Conference on Field-Programmable Technology (ICFPT'2008)*, pages 285–288, 2008. 126

[PMM08] Husain Parvez, Hayder Mrabet, and Habib Mehrez. Generic techniques and cad tools for automated generation of fpga layout. In *Proceedings of 4th IEEE Conference on Ph.D. Research in MicroElectronics and Electronics (PRIME'2008)*, pages 141–144, 2008. 97

[Pé94] Frédérique Pétrot. *Outil d'Aide au développement de bibliothèques VLSI portables*. PhD thesis, Université Pierre et Marie Curie Paris VI, octobre 1994. 88

[SCM01] Guillaume Savaton, Emmanuel Casseau, and Eric Martin. A methodology for behavioral virtual component specification targeting. In *Procceedings of Forum on Specification & Design Languages (FDL'01)*, pages 3–9, 2001. 94

[SCMLN01] Guillaume Savaton, Emmanuel Casseau, Eric Martin, and Catherine LAMBERT-NEBOUT. Composants virtuels comportementaux pour applications de compression d'images. In *Procceedings of Colloque GRETSI'01*, 2001. 94

[sf] Python software foundation. http ://www.python.org. 92

[Sha86] Moe Shahdad. An overview of vhdl language and technology. In *Procceedings of the 23^{rd} Design Automation Conference (DAC'86)*, pages 320–326, 1986. 87

[She08] Abbas Sheibanyrad. *Asynchronous Implementation of a Distributed Network-on-Chip*. PhD thesis, Université Pierre et Marie Curie Paris VI, mars 2008. 97

[Smi96] Douglas J. Smith. Vhdl & verilog compared & contrasted - plus modeled example written in vhdl, verilog and c. In *Procceedings of the 33^{rd} Design Automation Conference (DAC'96)*, pages 771–776, 1996. 87

[SMOR98] P. Stelling, C.U. Martel, V.G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Trans. on Computers*, 47(3), 1998. 44

[SPG07] Abbas Sheibanyrad, Ivan Miro Panades, and Alain Greiner. Systematic comparison between the asynchronous and the multi-synchronous implementations of a network on chip architecture. In *Procceedings of the Design, Automation, and Test in Europe (DATE'07)*, 2007. 97

[Syn] Synopsys. http ://www.synopsys.com. 38, 86

[TCL] TCL/TK. http ://www.tcl.tk. 97

[UK02] Junhyung Um and Taewhan Kim. Layout-aware synthesis of arithmetic circuits. In *Procceedings of the* 39^{th} *Design Automation Conference (DAC'02)*, pages 207–212, 2002. 44

[UKL99] Junhyung Um, Taewhan Kim, and C. L. Liu. Optimal allocation of carry-save-adders in arithmetic optimization. In *Procceedings of IEEE/ACM International Conference on Computer-Aider Design (ICCAD'99)*, pages 410–413, 1999. 40, 41, 126

[Vau97] Nicolas Vaucher. *Méthodogie de conception d'architectures VLSI génériques appliquée au traitement numérique*. PhD thesis, Université Pierre et Marie Curie Paris VI, juin 1997. 89

[VBG+94] A. Vacher, M. Benkhebbab, A. Guyot, T. Rousseau, and A. Skaf. A vlsi implementation of parallel fast fourier transform. In *Procceedings of the European Conference on Design and Test (EDCT'94)*, pages 250–255, 1994. 34

[VI04] Ajay K. Verma and Paolo Ienne. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Procceedings of IEEE/ACM International Conference on Computer-Aider Design (ICCAD'04)*, pages 791–798, 2004. 47

[YKW00] Zhan Yu, Kei-Yong Khoo, and Alan N. Willson, Jr. The use of carry-save representation in joint module selection and retiming. In *Procceedings of the* 37^{th} *Design Automation Conference (DAC'00)*, pages 462–467, 2000. 45

[YYW01] Zhan Yu, Meng-Lin Yu, and Alan N. Willson, Jr. Signal representation guided synthesis using carry-save adders for synchronous data-path circuits. In *Procceedings of the* 38^{th} *Design Automation Conference (DAC'01)*, 2001. 45

[Zim] Reto Zimmermann. Computer arithmetic : Principles, architectures, and vlsi design. http ://www.iis.ee.ethz.ch/∼zimmi. 206, 207, 209, 216, 217, 219, 233

Annexes

Annexe A
Description d'une règle

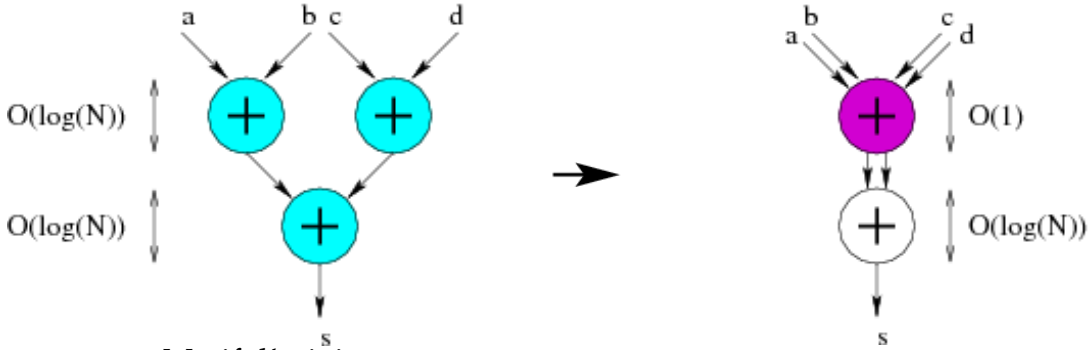
A.1 Description xml

Nous avons choisi le format xml de façon à ce que la description des règles soit aisée et intuitive.

Différentes balises sont utilisées, *rule* pour les règles par exemple. Dans chaque règle, deux motifs sont décrits (balise *pattern*), le motif d'origine et le motif à substituer. Dans chacun de ces motifs sont décrits les nœuds (balise *instance*) et les arcs (balise *net*) les composant. L'architecture de l'opérateur correspondant à chaque nœud (mot-clé *model*) est décrite, ainsi que la connectique entre nœuds (balise *port*, avec le nom du port *name* et son sens *direct*).

A.2 Exemple

Reprenons la règle présentée dans la Figure 4.5 du Chapitre 4.2, nous décrivons ci-après la façon dont cette règle est créée en langage xml.



a. Motif d'origine

b. Motif substitué

FIG. A.1 – Exemple de règle

```
<rule id="add_fusion">
   <pattern type="old">
      <instance id="1" model="fixed.adder.sklansky.Sklansky">
         <port name="i0" direct="input" map="temp1"></port>
         <port name="i1" direct="input" map="temp2"></port>
         <port name="o" direct="input" map="s"></port>
      </instance>
      <instance id="2" model="fixed.adder.sklansky.Sklansky">
         <port name="i0" direct="input" map="a"></port>
         <port name="i1" direct="input" map="b"></port>
         <port name="o" direct="output" map="temp1"></port>
      </instance>
      <instance id="3" model="fixed.adder.sklansky.Sklansky">
         <port name="i0" direct="input" map="c"></port>
         <port name="i1" direct="input" map="d"></port>
         <port name="o" direct="output" map="temp2"></port>
      </instance>
      <net id="a"></net>
      <net id="b"></net>
      <net id="c"></net>
      <net id="d"></net>
      <net id="s"></net>
      <net id="temp1"></net>
      <net id="temp2"></net>
   </pattern>
   <pattern type="new">
      <instance id="1" model="fixed.adder.sklansky.Sklansky">
         <port name="i0" direct="input" map="temp.0"></port>
         <port name="i1" direct="input" map="temp.1"></port>
         <port name="o" direct="input" map="s"></port>
      </instance>
      <instance id="2" model="fixed.adder.redundant.Redundant">
         <port name="i0_0" direct="input" map="a"></port>
         <port name="i0_1" direct="input" map="b"></port>
         <port name="i1_0" direct="input" map="c"></port>
         <port name="i1_1" direct="input" map="d"></port>
         <port name="o_0" direct="output" map="temp.0"></port>
         <port name="o_1" direct="output" map="temp.1"></port>
      </instance>
      <net id="a"></net>
      <net id="b"></net>
      <net id="c"></net>
      <net id="d"></net>
      <net id="s"></net>
      <net id="temp" repr="cs"</net>
   </pattern>
</rule>
```

Annexe B Ensemble des règles Carry-Save

Nous présentons ci-après les différentes règles constituant l'ensemble des règles Carry-Save. Pour chaque règle sont présentés les deux motifs *Avant/Après*. A ces motifs correspondent deux graphes montrant l'évolution de la surface et de la chaîne longue, en fonction du nombre de bits des données.

B.1 Transformation d'une addition en un nombre Carry-Save

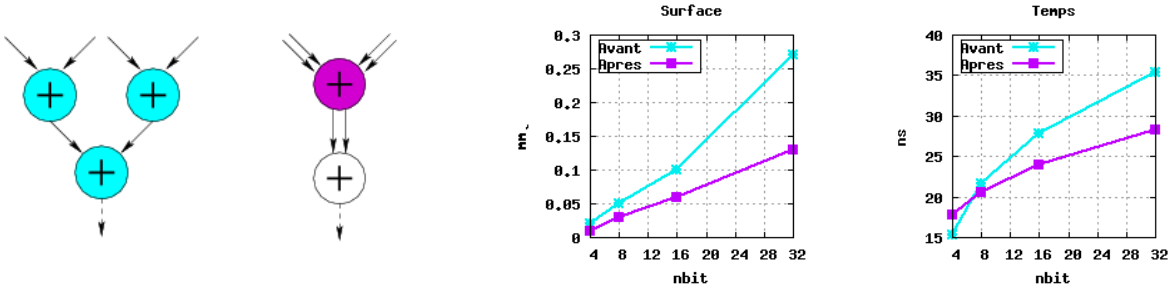


FIG. B.1 – Motif 1

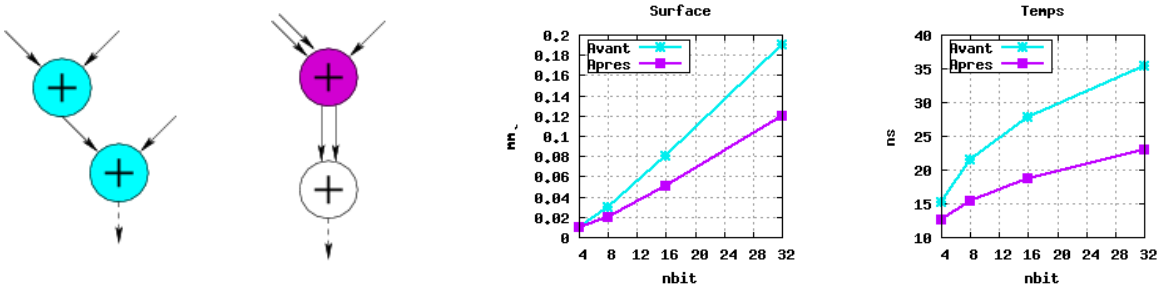


FIG. B.2 – Motif 2

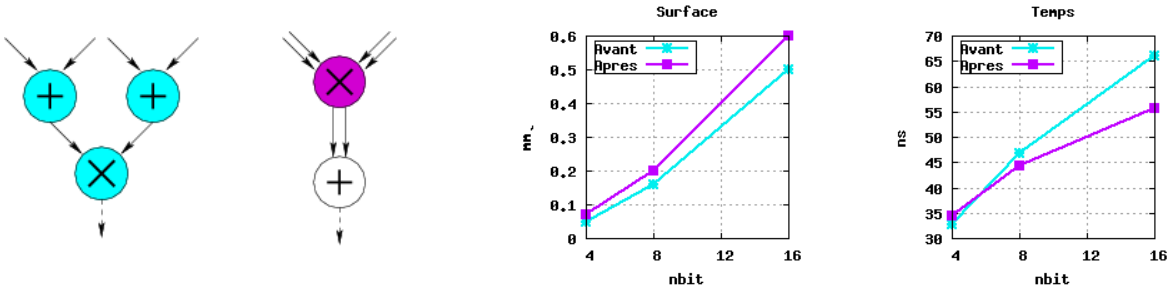


FIG. B.3 – Motif 3

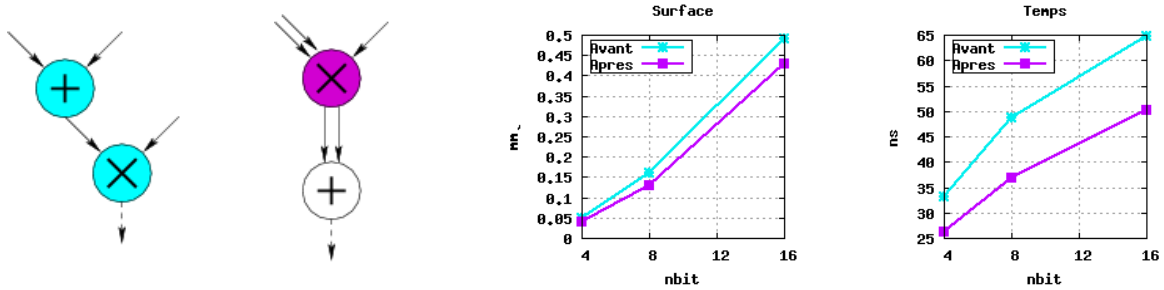


FIG. B.4 – Motif 4

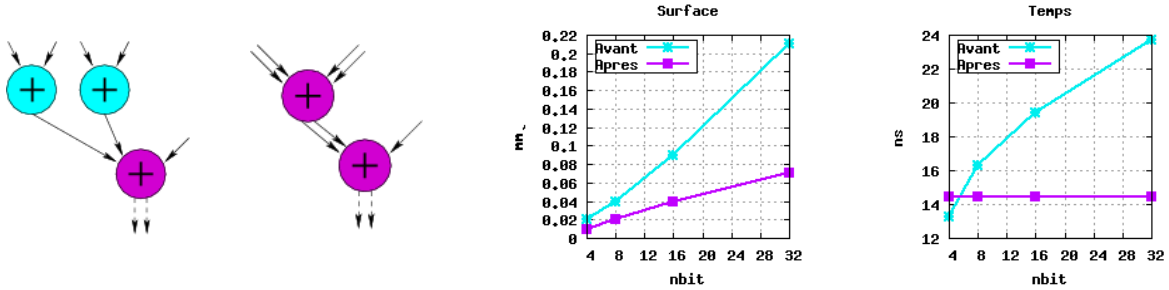


FIG. B.5 – Motif 5

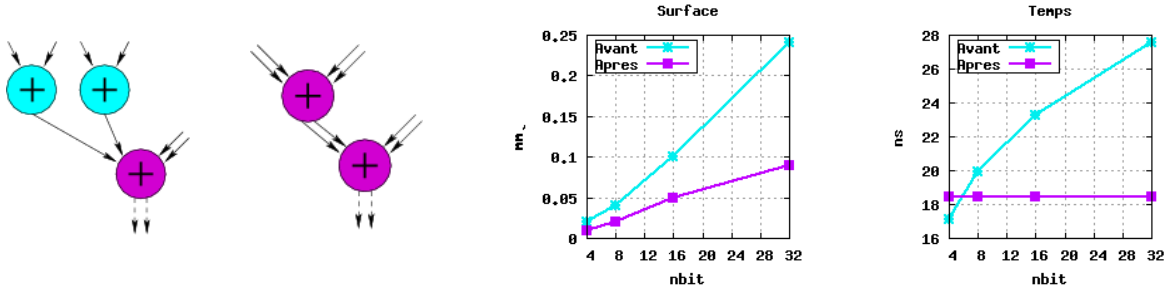


FIG. B.6 – Motif 6

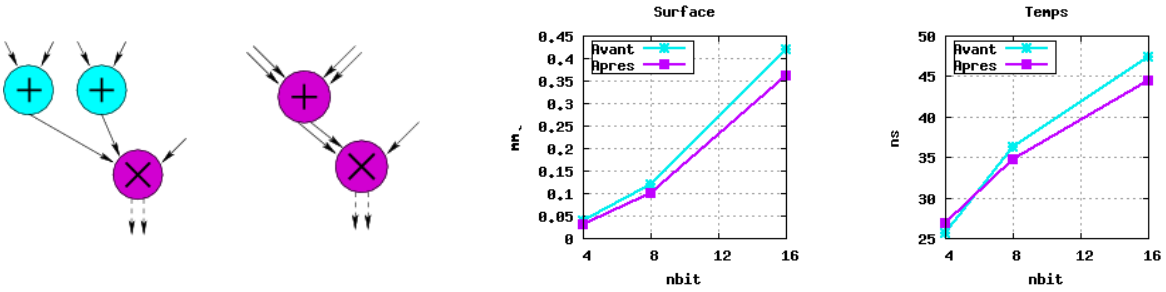


FIG. B.7 – Motif 7

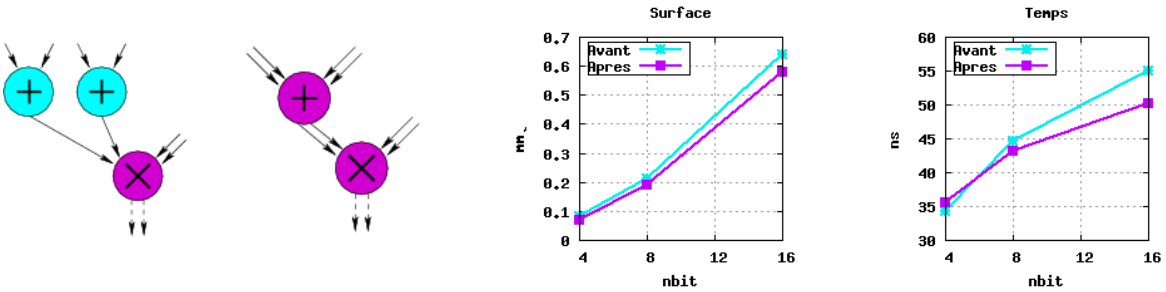


FIG. B.8 – Motif 8

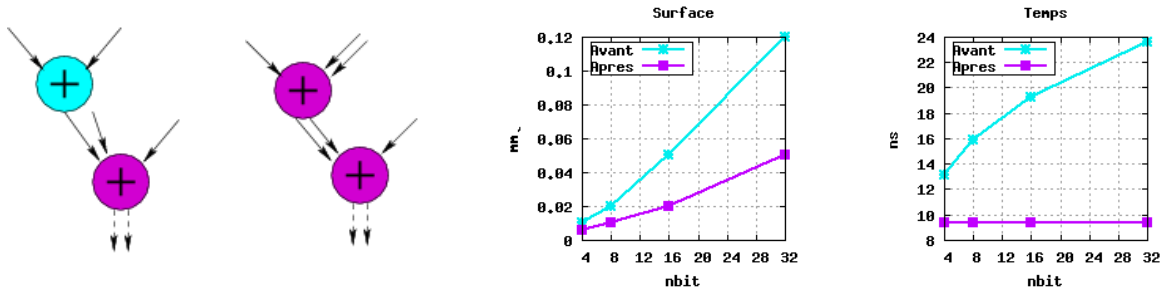


FIG. B.9 – Motif 9

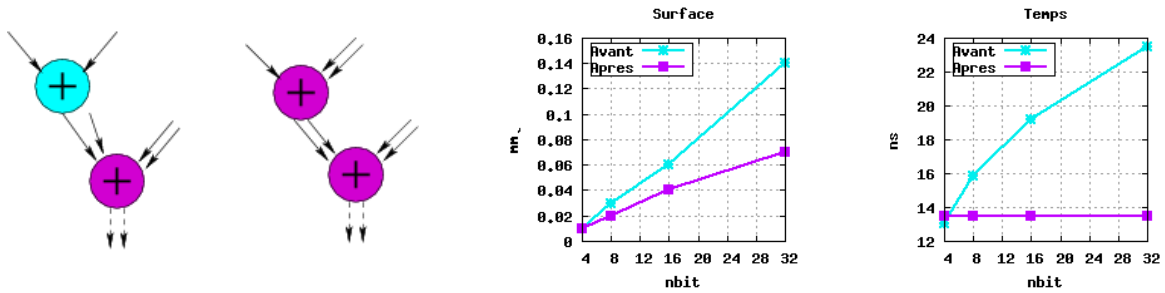


FIG. B.10 – Motif 10

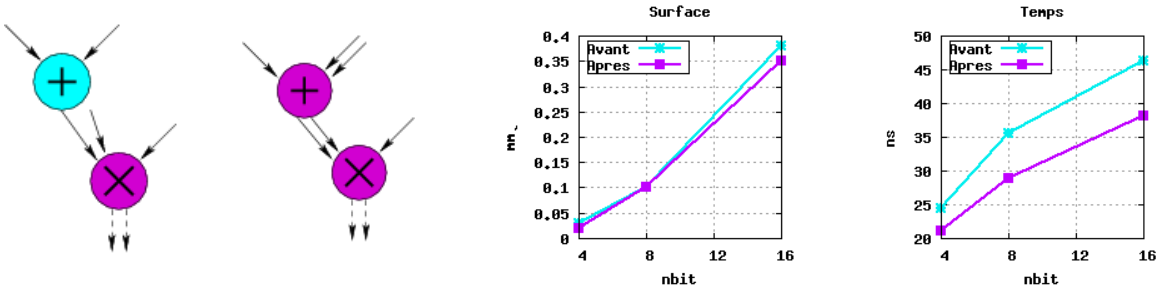


FIG. B.11 – Motif 11

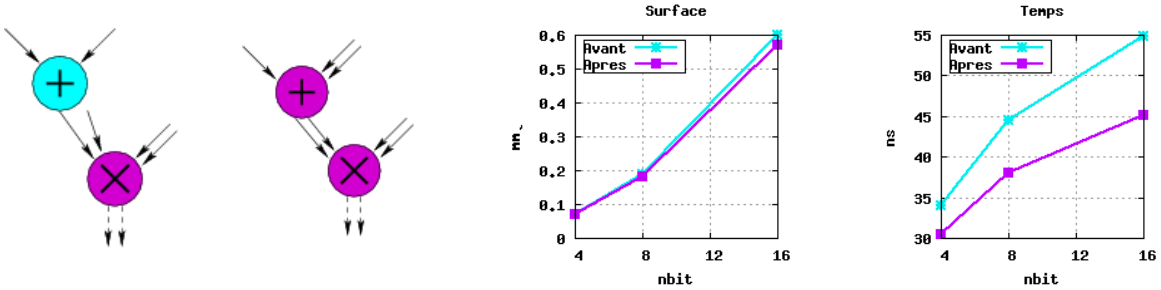


FIG. B.12 – Motif 12

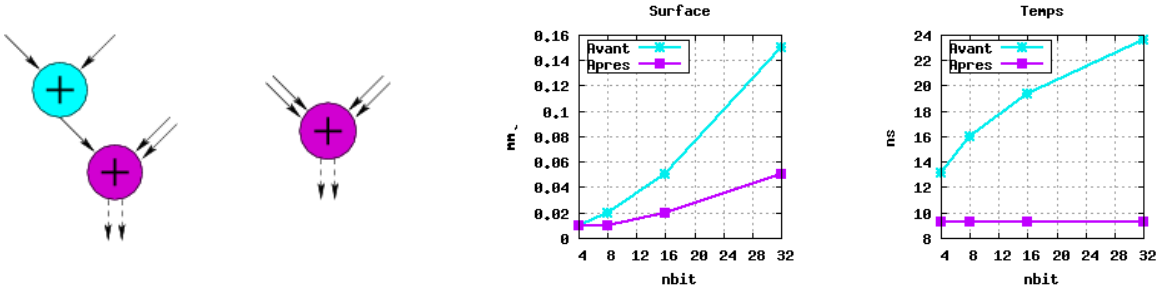


FIG. B.13 – Motif 13

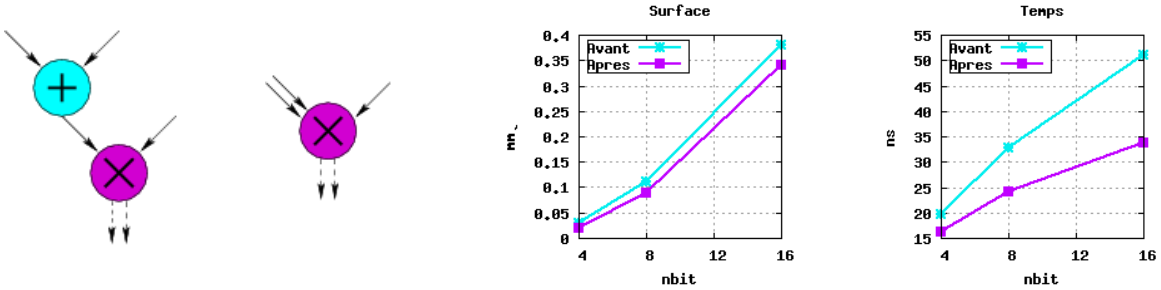


FIG. B.14 – Motif 14

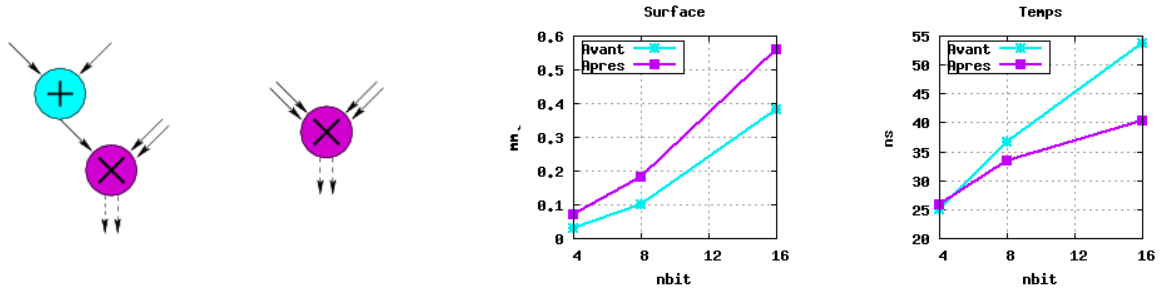


FIG. B.15 – Motif 15

B.2 Modification de la sortie d'un multiplieur

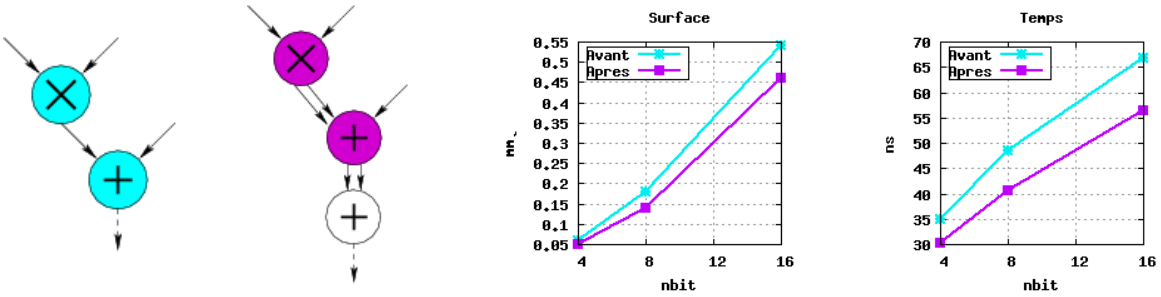


FIG. B.16 – Motif 16

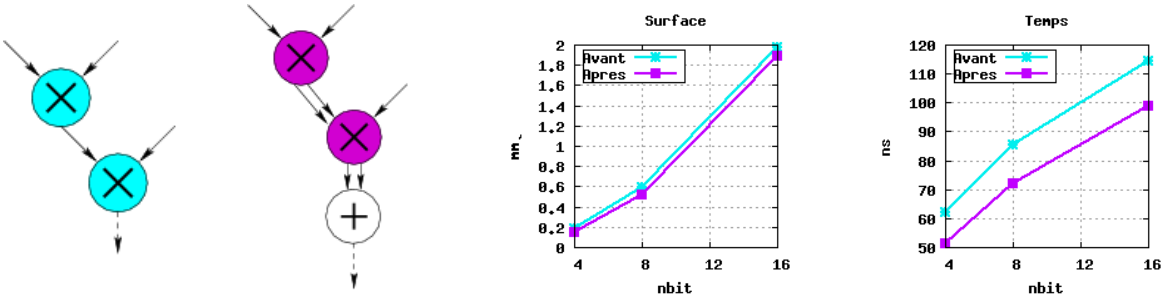


FIG. B.17 – Motif 17

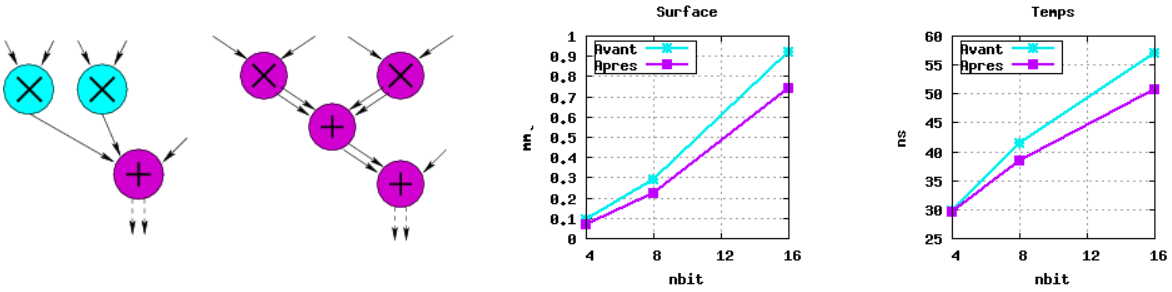


FIG. B.18 – Motif 18

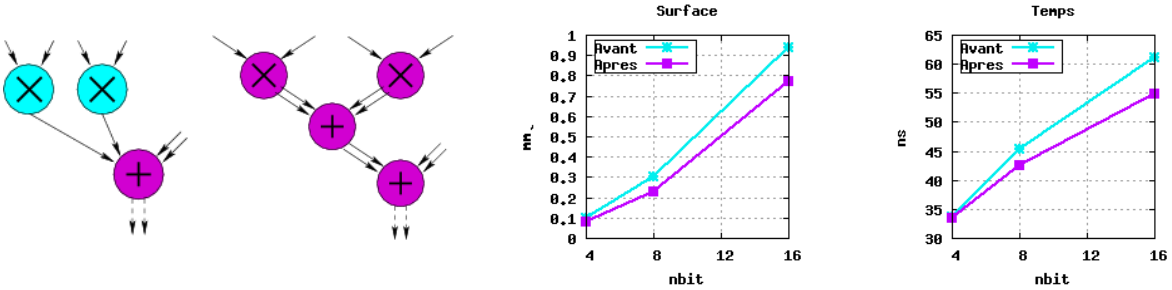


FIG. B.19 – Motif 19

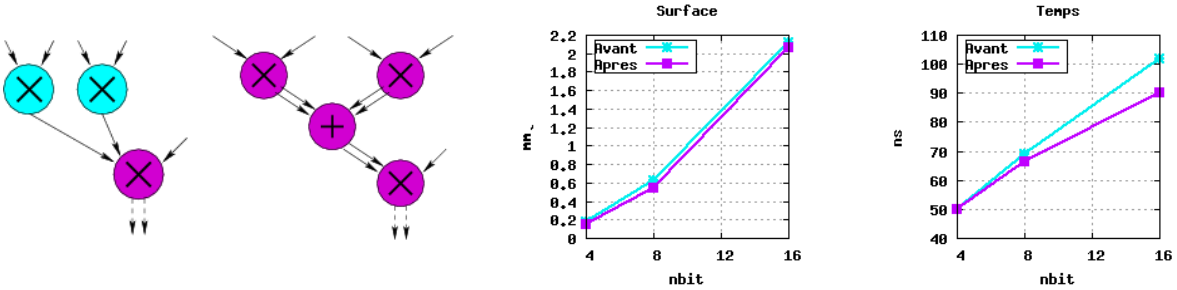


FIG. B.20 – Motif 20

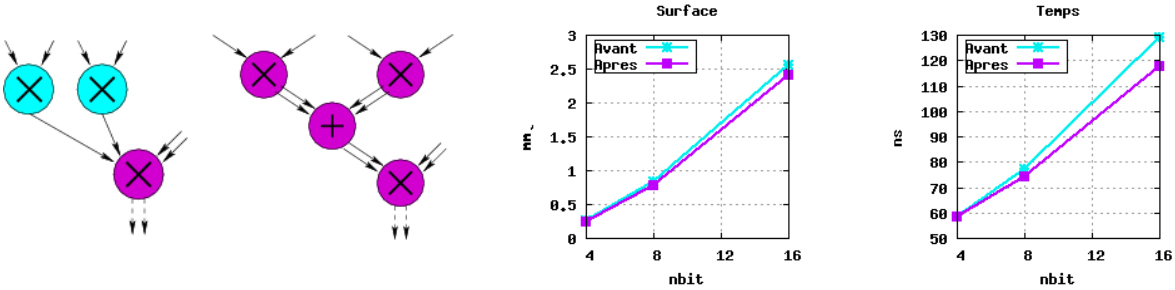


FIG. B.21 – Motif 21

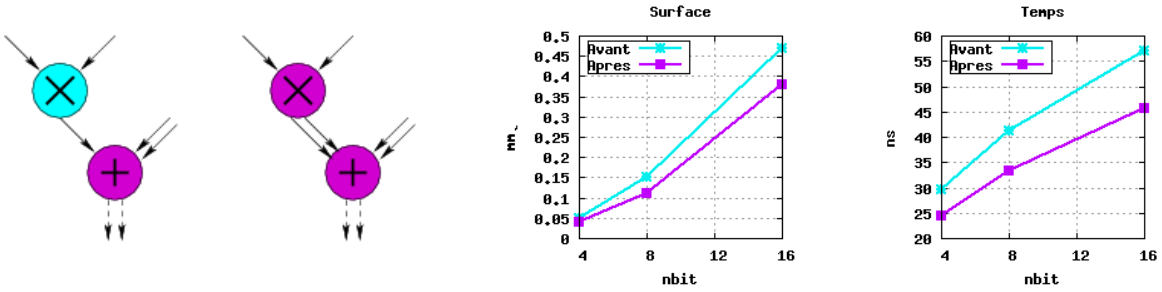


FIG. B.22 – Motif 22

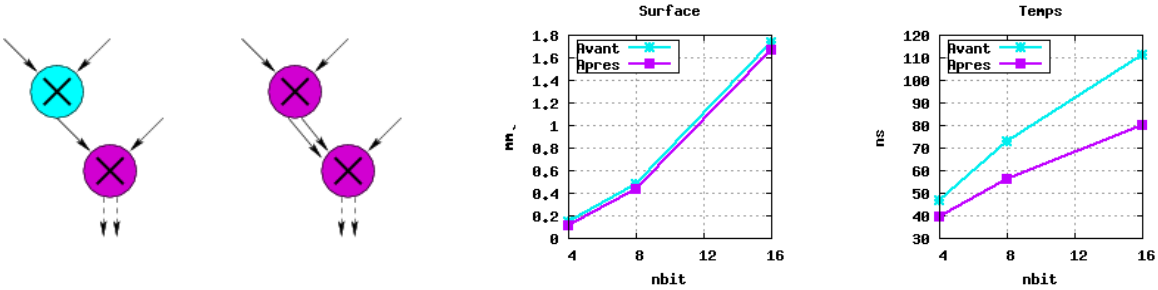


FIG. B.23 – Motif 23

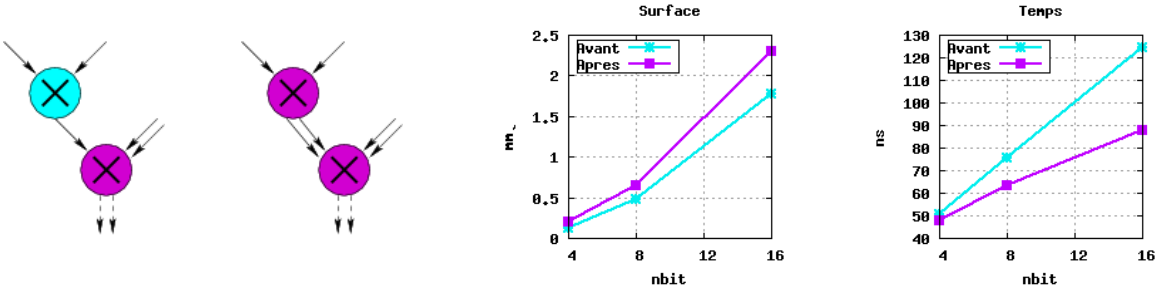


FIG. B.24 – Motif 24

Annexe C Exemple

C.1 Cellule

C.1.1 Description haut niveau

```
class Chemin ( Arith ) :

 def Interface ( self ) :
  self.n = param['nbit']
  self.A = SignalIn ( "a", self.n )
  self.B = SignalIn ( "b", self.n )
  self.C = SignalIn ( "c", self.n )
  self.D = SignalIn ( "d", self.n )
  self.E = SignalIn ( "e", self.n )
  self.F = SignalIn ( "f", self.n )
  self.S = SignalOut ( "s", self.n*2 )

  self.vdd = VddIn ( "vdd" )
  self.vss = VssIn ( "vss" )

 def Netlist ( self ) :
  self.S <= ( ( self.A + self.B ) * self.E )
        + ( self.C + self.D + self.F )
```

C.1.2 Description bas niveau

```
class Chemin ( Arith ) :

 def Interface ( self ) :
  self.n = param['nbit']
  self.A = SignalIn ( "a", self.n )
  self.B = SignalIn ( "b", self.n )
  self.C = SignalIn ( "c", self.n )
  self.D = SignalIn ( "d", self.n )
  self.E = SignalIn ( "e", self.n )
  self.F = SignalIn ( "f", self.n )
  self.S = SignalOut ( "s", self.n*2 )

  self.vdd = VddIn ( "vdd" )
  self.vss = VssIn ( "vss" )
```

```
def Netlist ( self ) :
  temp0 = Signal ( "temp0", self.n )
  temp1 = Signal ( "temp1", self.n )
  temp2 = Signal ( "temp2", self.n )
  temp3 = Signal ( "temp3", self.n*2 )

  Generate ( "fixed.adder.sklansky.sklansky", "adder_8bits"
         , param = { "nbit" : self.n } )
  Generate ( "fixed.adder.sklansky.Sklansky", "adder_8x16bits"
         , param = { "nbit0" : 16, "nbit1" : self.n } )
  Generate ( "fixed.multiplier.nr.Booth", "mult_8bits"
         , param = { "nbit" : self.n } )

  Inst ( "adder_8bits"
      , map = { "i0" : self.A
              , "i1" : self.B
              , "o" : temp0
              , "vdd" : self.vdd
              , "vss" : self.vss
              } )
  Inst ( "adder_8bits"
      , map = { "i0" : self.C
              , "i1" : self.D
              , "o" : temp1
              , "vdd" : self.vdd
              , "vss" : self.vss
              } )
  Inst ( "mult_8bits"
      , map = { "i0" : self.E
              , "i1" : temp0
              , "o" : temp2
              , "vdd" : self.vdd
              , "vss" : self.vss
              } )
  Inst ( "adder_8bits"
      , map = { "i0" : self.F
              , "i1" : temp1
              , "o" : temp3
              , "vdd" : self.vdd
              , "vss" : self.vss
              } )
  Inst ( "adder_8x16bits"
      , map = { "i0" : temp2
              , "i1" : temp3
              , "o" : self.S
              , "vdd" : self.vdd
              , "vss" : self.vss
              } )
```

L'intérêt de la notation haut niveau prend alors tout son sens.

C.2 Phase d'initialisation en redondant

L'initialisation du circuit se fait comme montré ci-après :

```
exemple = Chemin ( "chemin", param = { 'nbit' : 32 } )

exemple.Interface()
exemple.Netlist()

exemple.Initialisation ( "algorithm"
                  , param = { 'sub' : 'None', 'version' : 'greedy' } )
exemple.Print()
exemple.HurricanePlug()
exemple.Save()
exemple.Pattern()
exemple.Simul()
```

Voici un récapitulatif des différents paramètres à fournir à la méthode `Initialisation` :

	mot-clé	valeurs
Algorithme	-	patternMatching, algorithm
Gestion de la soustraction	sub	None, subToAdd, CarrySave, borrowSave
Gestion des DAG	level	1, 2, 3 (valable que pour la reconnaissance de motifs)
Version de l'algorithme	version	greedy, exhaustive (valable que pour les algorithmes de labellisation)
Impression	toPrint	travel, cell, pattern

TAB. C.1 – Paramètres de l'initialisation en redondant

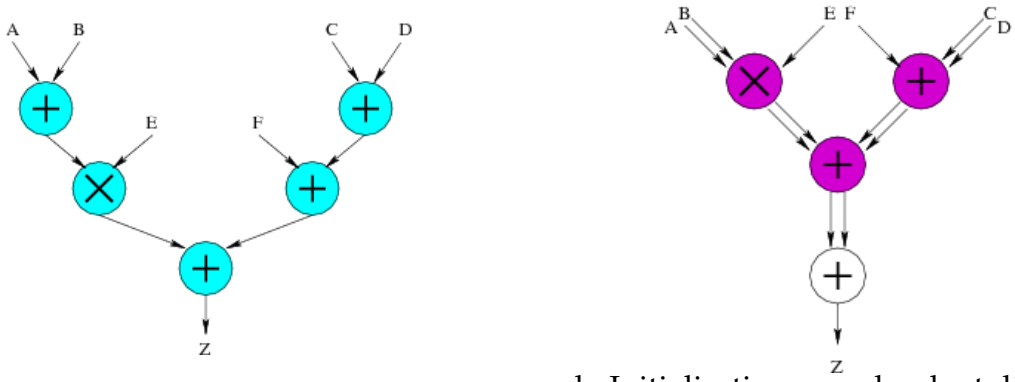
C.3 Cellule générée

La Figure C.1 montre la cellule initiale est le résultat d'une initialisation en redondant dès que possible. Après la phase d'initialisation, il est possible d'obtenir la vue modifiée du circuit dans le langage Stratus.

```
class Chemin ( Arith ) :

  def Interface ( self ) :

    self.a = ASignalIn ( "a", 8, repr = "ca2" )
    self.b = ASignalIn ( "b", 8, repr = "ca2" )
    self.c = ASignalIn ( "c", 8, repr = "ca2" )
    self.d = ASignalIn ( "d", 8, repr = "ca2" )
    self.e = ASignalIn ( "e", 8, repr = "ca2" )
    self.f = ASignalIn ( "f", 8, repr = "ca2" )
    self.s = ASignalOut ( "s", 16, repr = "ca2" )
```



a. Cellule

b. Initialisation en redondant dès que possible

FIG. C.1 – Exemple

```
  self.vdd = VddIn ( "vdd" )
  self.vss = VssIn ( "vss" )

def Netlist ( self ) :

  self.net_out_1_copy1 = ASignal ( "net_out_1_copy1", 16, repr = "cs" )
  self.net_out_3_copy3 = ASignal ( "net_out_3_copy3", 8, repr = "cs" )
  self.s_copy4_sl = ASignal ( "s_copy4_sl", 16, repr = "cs" )

  Generate ( "fixed.multiplier.mixed.Direct"
         , "fixed_multiplier_mixed_direct_nbit0_8_nbit1_8_o_cs"
         , param = {'nbit0': 8, 'nbit1': 8, 'o': 'cs'} )
  Generate ( "fixed.adder.mixed.Mixed"
         , "fixed_adder_mixed_mixed_nbit0_8_nbit1_8"
         , param = {'nbit0': 8, 'nbit1': 8} )
  Generate ( "fixed.adder.redundant.Redundant"
         , "fixed_adder_redundant_redundant_nbit0_16_nbit1_8"
         , param = {'nbit0': 16, 'nbit1': 8} )
  Generate ( "fixed.adder.sklansky.Sklansky"
         , "fixed_adder_sklansky_sklansky_nbit_16"
         , param = {'nbit': 16} )

  self.inst1 = Inst ( "fixed_multiplier_mixed_direct_nbit0_8_nbit1_8_o_cs"
          , "instance1_fixed_multiplier_mixed_direct_nbit0_8_nbit1_8_o_cs"
          , map = { "vss" : self.vss
                , "o_1" : self.net_out_1_copy1[1]
                , "o_0" : self.net_out_1_copy1[0]
                , "i1" : self.e[0]
                , "i0_0" : self.a[0]
                , "i0_1" : self.b[0]
                , "vdd" : self.vdd
                }
          )
```

```
self.inst2 = Inst ( "fixed_adder_mixed_mixed_nbit0_8_nbit1_8"
        , "instance2_fixed_adder_mixed_mixed_nbit0_8_nbit1_8"
        , map = { "vss" : self.vss
              , "o_1" : self.net_out_3_copy3[1]
              , "o_0" : self.net_out_3_copy3[0]
              , "i1" : self.f[0]
              , "i0_0" : self.c[0]
              , "i0_1" : self.d[0]
              , "vdd" : self.vdd
              }
        )

self.inst3 = Inst ( "fixed_adder_sklansky_sklansky_nbit_16"
        , "instance3_fixed_adder_sklansky_sklansky_nbit_16"
        , map = { "i1" : self.s_copy4_sl[1]
              , "i0" : self.s_copy4_sl[0]
              , "vss" : self.vss
              , "vdd" : self.vdd
              , "o" : self.s[0]
              }
        )

self.inst4 = Inst ( "fixed_adder_redundant_redundant_nbit0_16_nbit1_8"
       , "instance4_fixed_adder_redundant_redundant_nbit0_16_nbit1_8"
       , map = { "vss" : self.vss
             , "o_1" : self.s_copy4_sl[1]
             , "o_0" : self.s_copy4_sl[0]
             , "i0_0" : self.net_out_1_copy1[0]
             , "i0_1" : self.net_out_1_copy1[1]
             , "vdd" : self.vdd
             , "i1_1" : self.net_out_3_copy3[1]
             , "i1_0" : self.net_out_3_copy3[0]
             }
       )
```

Annexe D Stratus

Sommaire

D.1 Introduction 154
D.1.1 Stratus 154
D.1.2 Example 156
D.2 Description of a netlist 161
D.2.1 Nets 161
D.2.2 Instances 163
D.2.3 Generators 164
D.3 Description of a layout 165
D.3.1 Place 165
D.3.2 PlaceTop 166
D.3.3 PlaceBottom 167
D.3.4 PlaceRight 169
D.3.5 PlaceLeft 170
D.3.6 SetRefIns 171
D.3.7 DefAb 172
D.3.8 ResizeAb 173
D.4 Place and Route 175
D.4.1 PlaceSegment 175
D.4.2 PlaceContact 176
D.4.3 PlacePin 177
D.4.4 PlaceRef 178
D.4.5 GetRefXY 179
D.4.6 CopyUpSegment 179
D.4.7 PlaceCentric 180
D.4.8 PlaceGlu 181
D.4.9 FillCell 182
D.4.10 Pads 182
D.4.11 Alimentation rails 183
D.4.12 Alimentation connectors 184
D.4.13 PowerRing 184

D.4.14 RouteCk 185
D.5 Instanciation facilities 186
D.5.1 Buffer 186
D.5.2 Multiplexor 186
D.5.3 Shifter 189
D.5.4 Register 191
D.5.5 Constants 191
D.5.6 Boolean operations 193
D.5.7 Arithmetical operations 194
D.5.8 Comparison operations 195
D.6 Virtual library 196
D.7 Data Base 199
D.7.1 Model 199
D.7.2 Nets 200
D.7.3 Instances 202

D.1 Introduction

D.1.1 Stratus

Name

Stratus – Procedural design language based upon *Python*

Description

Stratus is a set of *Python* methods/functions dedicated to procedural generation purposes. From a user point of view, *Stratus* is a circuit's description language that allows *Python* programming flow control, variable use, and specialized functions in order to handle vlsi objects.

Based upon the *Hurricane* data structures, the *Stratus* language gives the user the ability to describe netlist and layout views.

Creation of a cell

A cell is a hierachical structural description of a circuit in terms of ports (I/Os), signals (nets) and instances.

The creation of a cell is done by creating a new class, derivating for class `Model`, with different methods :

- Method `Interface` : Description of the external ports of the cell :
  - SignalIn, SignalOut, ...
- Method `Netlist` : Description of the netlist of the cell :
  - Inst, Signal
- Method `Layout` : Description of the layout of the cell :
  - Place, PlaceTop, PlaceBottom, PlaceRight, PlaceLeft ...

Two methods are provided :

- Method `View` : Opens/Refreshes the editor in order to see the created layout
- Method `Save` : Saves the created cell
  - no argument : creation of a netlist file (format file thanks to CRL_OUT_LO)
  - PHYSICAL : creation of a netlist file AND a layout file (format files thanks to CRL_OUT_LO and CRL_OUT_PH)
  - STRATUS : creation of a python/stratus file
    - FileName : optionnal argument when using Save(STRATUS) in order to choose the name of the file to be generated
    - Be careful : if one wants to create a stratus file AND a netlist, always use Save(STRATUS) before Save() !

Syntax

A *Stratus* file must have a .py extension and must begin as follow :

```
#!/usr/bin/env python

from stratus import *
```

The creation of a class is done as follow :

```
class myClass ( Model ) :
   ...

exemple = myClass ( name, param )
```

In order to execute a *Stratus* file (named `file` for example), one has two choices :

```
python file.py
```

Or :

```
chmod u+x file.py
./file.py
```

The names used in *Stratus*, as arguments to *Stratus* functions, should be alphanumerical, including the underscore. The arguments of *Stratus* are case sensitive, so `VDD` is not equivalent to `vdd`.

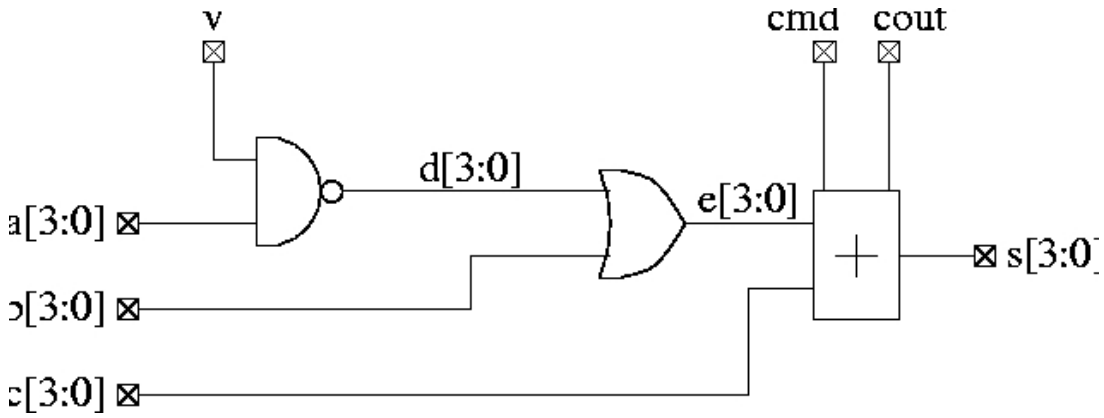
Vectorized connectors or signal can be used using the [N :M] construct.

Environment variables

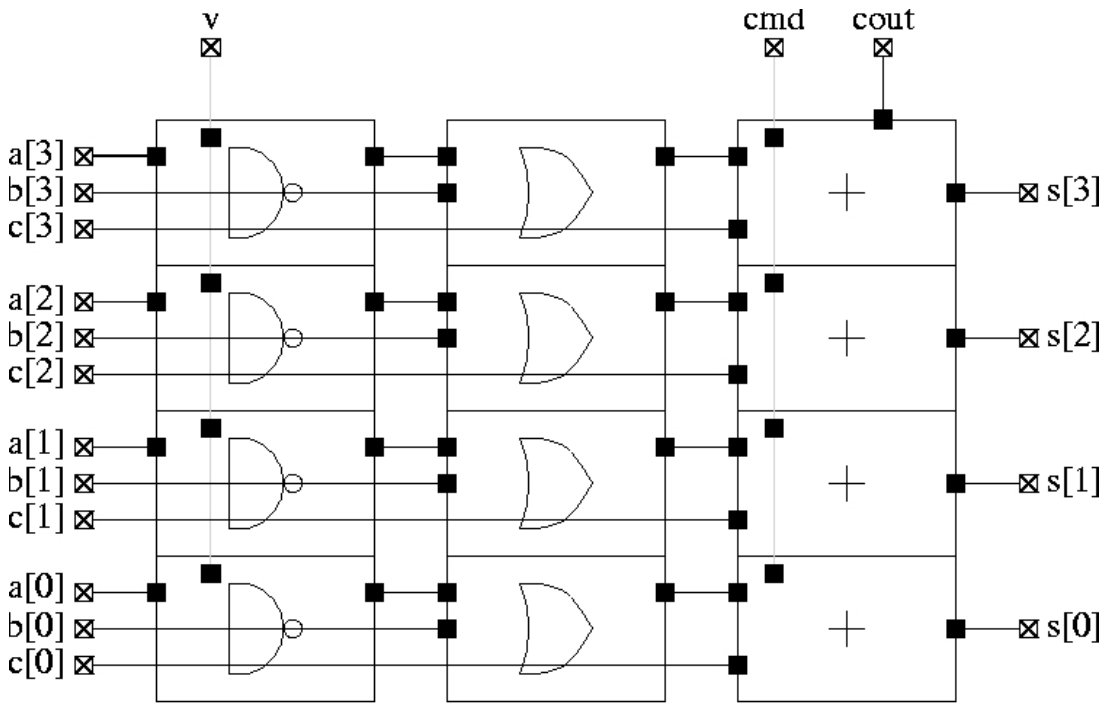
- CRL_IN_LO, default value : `def`
- CRL_OUT_LO, default value : `def`
- CRL_IN_PH, default value : `def`
- CRL_OUT_PH, default value : `def`
- CRL_CATA_LIB, default value : `.`
- CRL_CATAL_NAME, default value : `CATAL`

D.1.2 Example

The circuit



The data-path



Description of the circuit with *Stratus* : file addaccu.py

```
#!/usr/bin/env python

from stratus import *

class addaccu ( Model ) :

  def Interface ( self ) :
    self.nbit = self._param['nbit']

    self.a    = SignalIn ( "a", self.nbit )
    self.b    = SignalIn ( "b", self.nbit )
    self.c    = SignalIn ( "c", self.nbit )
    self.v    = SignalIn ( "v", 1 )
    self.cmd  = SignalIn ( "cmd", 1 )

    self.cout = SignalOut ( "cout", 1 )
    self.s    = SignalOut ( "s", self.nbit )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    d_aux = Signal ( "d_aux", self.nbit )
    e_aux = Signal ( "e_aux", self.nbit )
    ovr   = Signal ( "ovr", 1  )

    Generate ( "DpgenNand2", "nand2%d" % self.nbit
             , param = { 'nbit'     : self.nbit
                       , 'physical' : True
                       }
             )
    self.instNand2 = Inst ( "nand2%d" % self.nbit, "instance_nand2"
                          , map   = { 'i0'   : Cat ( self.v, self.v, self.v, self.v )
                                    , 'i1'   : self.a
                                    , 'nq'   : d_aux
                                    , 'vdd'  : self.vdd
                                    , 'vss'  : self.vss
                                    }
                           )

    Generate ( "DpgenOr2", "or2%d" % self.nbit
             , param = { 'nbit'     : self.nbit
                       , 'physical' : True
                       }
             )
    self.instOr2   = Inst ( "or2%d" % self.nbit, "instance_or2"
                          , map   = { 'i0'   : d_aux
                                    , 'i1'   : self.b
                                    , 'q'    : e_aux
                                    , 'vdd'  : self.vdd
                                    , 'vss'  : self.vss
                                    }
                          )
```

```
54
55      Generate ( "DpgenAdsb2f", "adder%d" % self.nbit
56               , param = { 'nbit'     : self.nbit
57                         , 'physical' : True
58                         }
59               )
60      self.instAdd2  = Inst ( "adder%d" % self.nbit, "instance_add2"
61                            , map    = { 'i0'      : e_aux
62                                       , 'i1'      : self.c
63                                       , 'q'       : self.s
64                                       , 'add_sub' : self.cmd
65                                       , 'c31'     : self.cout
66                                       , 'c30'     : ovr
67                                       , 'vdd'     : self.vdd
68                                       , 'vss'     : self.vss
69                                       }
70                            )
71
72    def Layout ( self ) :
73      Place      ( self.instNand2, NOSYM, XY ( 0, 0 ) )
74      PlaceRight ( self.instOr2,   NOSYM )
75      PlaceRight ( self.instAdd2,  NOSYM )
76
```

Creation of the circuit : file test.py

```
1 #!/usr/bin/env python
2
3 from stratus import *
4 from addaccu import addaccu
5
6 nbit = Param ( "n" )
7
8 dict = { 'nbit' : nbit }
9
10 inst_addaccu = addaccu ( "inst_addaccu", dict )
11
12 inst_addaccu.Interface()
13 inst_addaccu.Netlist()
14 inst_addaccu.Layout()
15 inst_addaccu.View()
16
17 inst_addaccu.Save ( PHYSICAL )
```

How to execute the file

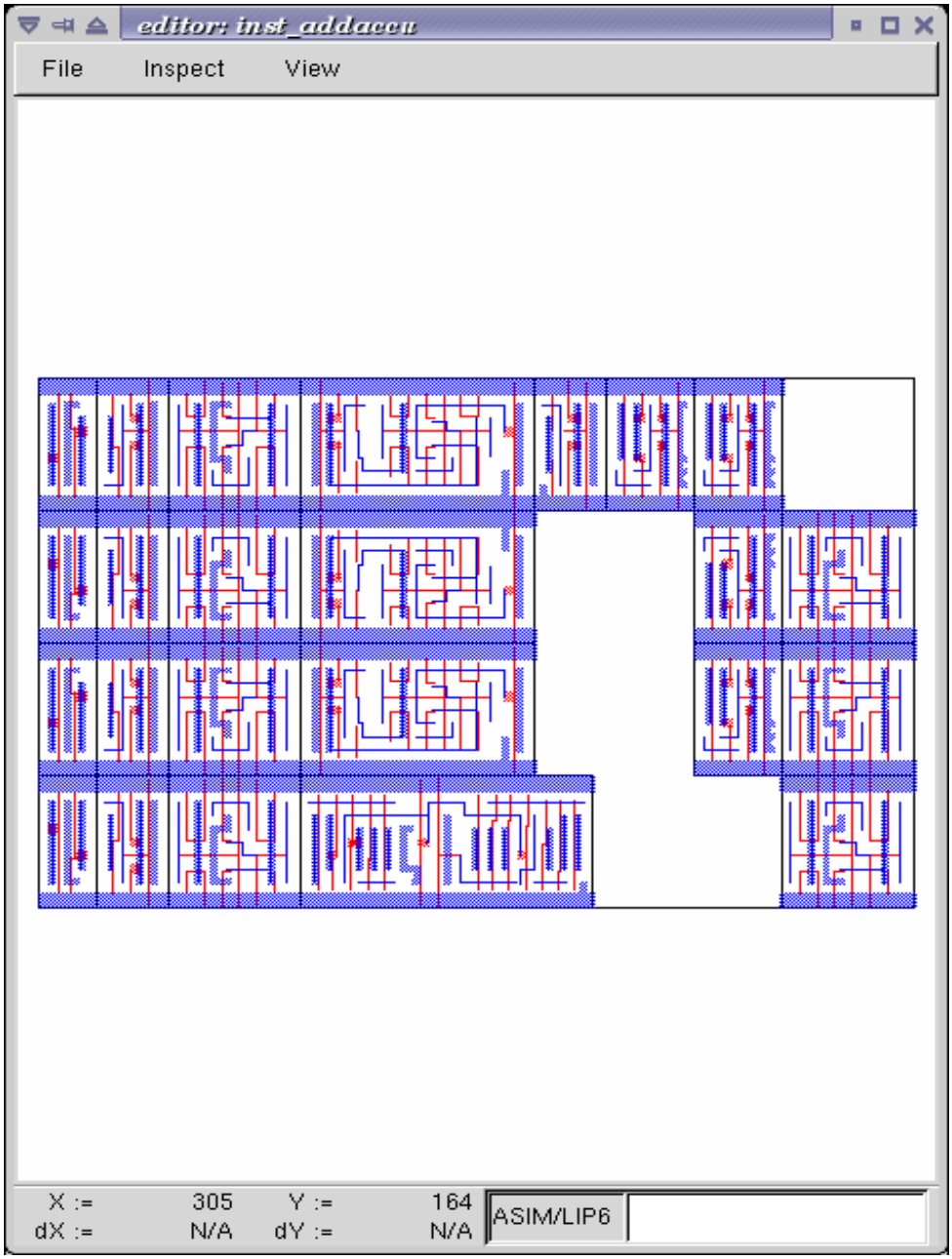
```
python test.py -n 4
```

or :

```
chmod u+x test.py
./test -n 4
```

The editor

The method `View` permits to open an editor in which one can see the cell being created as shown in the picture below.



Function Param

This function allows the user to give parameters when creating a cell.
When one wants to give values to two parameters, one can type on the shell :

```
python test.py -n 4 -w 8
```

The file `test.py` has then to contain :

```
nbit, nword = Param ( "n", "w" )
```

The letters typed on the shell must be the ones given as parameters of function `Param`.

How to instanciate your generator in another generator

One can create a generator and instantiate it in another generator.
To do that, the model name of the generator must have the form : "file_name.class_name".
Note that if the two generators are not in the same directory, the directory of the generator to be instantiated has to be added in the CRL_CATA_LIB environment variable.

For example, in order to instanciate the circuit created above in a cell :

```
n = 4
Generate ( "addaccu.addaccu", "my_addaccu_%dbits" % n
       , param = { 'nbit' : n } )

Inst ( "my_addaccu_%dbits" % n
    , map = { 'a' : self.netA
            , 'b' : self.netB
            , 'c' : self.netC
            , 'v' : self.netV
            , 'cmd' : self.netCmd
            , 'cout' : self.netCout
            , 's' : self.netS
            , 'vdd' : self.vdd
            , 'vss' : self.vss
            }
    )
```

D.2 Description of a netlist

D.2.1 Nets

Name

SignalIn, SignalOut ... – Creation of nets

Synopsys

```
netA = SignalIn ( "a", 4 )
```

Description

How to create and use nets.

Nets

Different kinds of nets are listed below :
- `SignalIn` : Creation of an input port
- `SignalOut` : Creation of an output port
- `SignalInOut` : Creation of an inout port
- `SignalUnknown` : Creation of an input/output port which direction is not defined
- `TriState` : Creation of a tristate port
- `CkIn` : Creation of a clock port
- `VddIn` : Creation of the vdd alimentation
- `VssIn` : Creation of the vss alimentation
- `Signal` : Creation of an internal net

Parameters

All kind of constructors have the same parameters :
- `name` : the name of the net (mandatory argument)
- `arity` : the arity of the net (mandatory argument)
- `indice` : for bit vectors only : the LSB bit (optional argument : set to 0 by default)

Only `CkIn`, `VddIn` and `VssIn` do not have the same parameters : there is only the `name` parameter (they are 1 bit nets).

Functions and methods

Some functions/methods are provided in order to handle nets :
- function `Cat` : Concatenation of nets, beginning with the MSB

```
Inst ( 'DpgenInv'
    , map = { 'i0' : Cat ( A, B )
            , 'nq' : S
            , 'vdd' : vdd
            , 'vss' : vss
            }
    )
```

Or :

```
tab = []
tab.append ( A )
tab.append ( B )

Inst ( 'DpgenInv'
    , map = { 'i0' : Cat ( tab )
            , 'nq' : S
            , 'vdd' : vdd
            , 'vss' : vss
            }
    )
```

If A and B are 2 bits nets, the net `myNet` will be such as :

```
myNet[3] = A[1]
myNet[2] = A[0]
myNet[1] = B[1]
myNet[0] = B[0]
```

– function `Extend` : Creation of a net which is an extension of the net which it is applied to

```
temp = Signal ( "temp", 5 )
tempExt = Signal ( "temp_ext", 8 )

tempExt <= temp.Extand ( 8, 'one' )
```

– method `Alias` : Creation of an alias name for a net

```
cin.Alias ( c_temp[0] )
cout.Alias ( c_temp[4] )
for i in range ( 4 ) :
 Inst ( "Fulladder"
     , map = { 'a' : a[i]
             , 'b' : b[i]
             , 'cin' : c_temp[i]
             , 'sout' : sout[i]
             , 'cout' : c_temp[i+1]
             , 'vdd' : vdd
             , 'vss' : vss
             }
     )
```

Errors

Some errors may occur :
– `Error in SignalIn :`
`the lenght of the net must be a positive value.`
One can not create a net with a negative lenght.

D.2.2 Instances

Name

Inst – Creation of instances

Synopsys

```
Inst ( model
    , name
    , map = connectmap
    )
```

Description

Instantiation of an instance. The type of the instance is given by the `model` parameter. The connexions are made thanks to the `connectmap` parameters.

Parameters

– `Model` : Name of the mastercell of the instance to create (mandatory argument)
– `name` : Name of the instance (optional)
When this argument is not defined, the instance has a name created by default. This argument is usefull when one wants to create a layout as well. Indeed, the placement of the instances is much easier when the conceptor has chosen himself the name f the instances.</para>
– `connectmap` : Connexions in order to make the netlist

`param` and `map` are dictionnaries as shown in the example below.

Example

```
Inst ( 'a2_x2'
    , map = { 'i0' : in0
            , 'i1' : in1
            , 'q' : out
            , 'vdd' : vdd
            , 'vss' : vss
            }
    )
```

Errors

Some errors may occur :

- `Error in Inst : the model Model does not exist. Check CRL_CATA_LIB.`
  Either one has made a mistake in the name of the model, either the environment variable is not correct.
- `Error in Inst : port does not exist in model Model.`
  One port in map is not correct.
- `Error in Inst : one input net is not dimensionned.`
  The size of the output nets is automatically calculated bus the input nets must be dimensionned before being connected.

D.2.3 Generators

Name

Generate – Interface with the generators

Synopsys

```
Generate ( model, modelname, param = dict )
```

Description

The `Generate` function call is the generic interface to all generators.

Arguments

- `model` : Specifies which generator is to be invoked
  - If the generator belongs to the Dpgen library provided by Stratus, the model name of the generator is simply the name of the class of the generator.
  - If the generator is created by the user, the model name of the generator must have the form : "file_name.class_name". (Note that if the the generator is not in the working directory, the directory of the generator to be instantiated has to be added in the CRL_CATA_LIB environment variable)
- `modelname` : Specifies the name of the model to be generated
- `dict` : Specifies the parameters of the generator

Parameters

Every generator has it's own parameters. They must be described in the map `dict`.

Every generator provides a netlist view. Two other views can be generated, if they are provided by the generator. Two parameters have to be given, in order to choose those views :

– 'physical' : True/False, generation of the physical view (optionnal, False by default)
– 'behavioral' : True/False, generation of the behavioral view (optionnal, False by default)

Errors

Some errors may occur :
– `[Stratus ERROR] Generate : the model must be described in a string.`

D.3 Description of a layout

D.3.1 Place

Name

Place – Places an instance

Synopsys

```
Place ( ins, sym, point )
```

Description

Placement of an instance.
The instance has to be instantiated in the method `Netlist`, in order to use the `Place` function.

Parameters

– `ins` : Instance to place.
– `sym` : Geometrical operation to be performed on the instance before beeing placed.
The `sym` argument can take eight legal values :
– `NOSYM` : no geometrical operation is performed
– `SYM_Y` : Y becomes -Y, that means toward X axe symetry
– `SYM_X` : X becomes -X, that means toward Y axe symetry
– `SYMXY` : X becomes -X, Y becomes -Y
– `ROT_P` : a positive 90 degrees rotation takes place
– `ROT_M` : a negative 90 degrees rotation takes place
– `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
– `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place

– `point` : coordinates of the lower left corner of the abutment box of the instance in the current figure.

Example

```
Place ( myInst, NOSYM, XY ( 0, 0 ) )
```

Errors

Some errors may occur :
– `[Stratus ERROR] Placement : the instance doesn't exist.`
The instance must be instanciated in order to be placed.
– `[Stratus ERROR] Placement : the first argument is not an instance.`
– `[Stratus ERROR] Placement : the instance is already placed.`
One can not place an instance twice
– `[Stratus ERROR] Place : wrong argument for placement type.`
The symetry given as argument is not correct.
– `[Stratus ERROR] Place : wrong argument for placement, the coordinates must be put in a XY object.`
The coordinates are not descrobed the bood way.

D.3.2 PlaceTop

Name

PlaceTop – Places an instance at the top of the "reference instance"

Synopsys

```
PlaceTop ( ins, sym, offsetX, offsetY )
```

Description

Placement of an instance.
The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The bottom left corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the top left corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

Parameters

– `ins` : Instance to place.

- `sym` : Geometrical operation to be performed on the instance before beeing placed.
  The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

Example

```
Place ( myInst1, NOSYM, 0, 0 )
PlaceTop ( myInst2, SYM_Y )
```

Errors

Some errors may occur :
- `[Stratus ERROR] Placement : the instance doesn't exist.`
  The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.`
  One can not place an instance twice
- `[Stratus ERROR] PlaceTop : no previous instance.`
  One can use `PlaceTop` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceTop : wrong argument for placement type.`
  The symetry given as argument is not correct.

D.3.3 PlaceBottom

Name

PlaceBottom – Places an instance below the "reference instance"

Synopsys

```
PlaceBottom ( ins, sym, offsetX, offsetY )
```

Description

Placement of an instance.
The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The top left corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the bottom left corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

Parameters

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed.
  The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

Example

```
Place ( myInst1, NOSYM, 0, 0 )
PlaceBottom ( myInst2, SYM_Y )
```

Errors

Some errors may occur :
- `[Stratus ERROR] Placement : the instance doesn't exist.` The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.` One can not place an instance twice

- `[Stratus ERROR] PlaceBottom : no previous instance.`
  One can use `PlaceBottom` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceBottom : wrong argument for placement type.`
  The symetry given as argument is not correct.

D.3.4 PlaceRight

Name

PlaceRight – Places an instance at the right of the "reference instance"

Synopsys

```
PlaceRight ( ins, sym, offsetX, offsetY )
```

Description

Placement of an instance.
The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The bottom left corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the bottom right corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

Parameters

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed.
  The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

Example

```
Place ( myInst1, NOSYM, 0, 0 )
PlaceRight ( myInst2, NOSYM )
```

Errors

Some errors may occur :

- `[Stratus ERROR] Placement : the instance doesn't exist.`
  The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.`
  One can not place an instance twice
- `[Stratus ERROR] PlaceRight : no previous instance.`
  One can use `PlaceRight` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceRight : wrong argument for placement type.`
  The symetry given as argument is not correct.

D.3.5 PlaceLeft

Name

PlaceLeft – Places an instance at the left of the "reference instance"

Synopsys

```
PlaceLeft ( ins, sym, offsetX, offsetY )
```

Description

Placement of an instance.
The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The bottom right corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the bottom left corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

Parameters

- `ins` : Instance to place.

- `sym` : Geometrical operation to be performed on the instance before beeing placed.
  The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

Example

```
Place ( myInst1, NOSYM, 0, 0 )
PlaceLeft ( myInst2, NOSYM )
```

Errors

Some errors may occur :
- `[Stratus ERROR] Placement : the instance doesn't exist.`
  The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.`
  One can not place an instance twice
- `[Stratus ERROR] PlaceLeft : no previous instance.`
  One can use `PlaceLeft` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceLeft : wrong argument for placement type.`
  The symetry given as argument is not correct.

D.3.6 SetRefIns

Name

SetRefIns – Defines the new "reference instance" for placement

Synopsys

```
SetRefIns ( ins )
```

Description

This function defines the new "reference instance", used as starting point in the relative placement functions.

It's regarding the abutmentbox of the instance `ins` that the next instance is going to be placed, if using the appropriate functions.

Note that the more recently placed instance becomes automaticaly the "reference instance", if SetRefIns isn't called.

Parameters

– `ins` : defines the new "reference instance"

Example

```
Place ( myInst1, NOSYM, 0, 0 )
PlaceRight ( myInst2, NOSYM )

% myInst3 is on top of myInst1 instead of myInst2
SetRefIns ( myInst1 )
PlaceTop ( myInst3, SYM_Y )
```

Errors

Some errors may occur :

– `[Stratus ERROR] SetRefIns : the instance doesn't exist.`
If the instance has not been instanciated, it is impossible do to any placement from it.

– `[Stratus ERROR] SetRefIns : the instance ... is not placed.`
If the instance has not been placed, it is impossible do to any placement from it.

D.3.7 DefAb

Name

DefAb – Creates the abutment box of the current cell

Synopsys

```
DefAb ( point1, point2 )
```

Description

This function creates the abutment box of the current cell.

Note that one does not have to call this function before saving in order to create the abutment box. The abutment box is created nevertheless (given to placed instances). This function is usefull if one wants to create an abutment before placing the instances.

Parameters

- `point1` : coordinates of the bottom left corner of the created abutment box.
- `point2` : coordinates of the top right corner of the created abutment box.

Example

```
DefAb ( XY(0, 0), XY(500, 100) )

Place ( self.inst, NOSYM, XY(0, 0) )
```

Errors

Some errors may occur :

- `[Stratus ERROR] DefAb : an abutment box already exists. Maybe you should use ResizeAb function.`
  One has called DefAb but the current cell already has an abutment box.
  In order to modify the current abutment box, the function to call is ResizeAb.
- `[Stratus ERROR] DefAb : wrong argument, the coordinates must be put in a XY object.`
  The type of one of the arguments is not correct. Coordinates must be put in a `XY` object.
- `[Stratus ERROR] DefAb : Coordinates of an abutment Box in y must be multiple of the slice. Coordinates of an abutment Box in x must be multiple of the pitch.`
  One has called DefAb with non authorized values.

D.3.8 ResizeAb

Name

ResizeAb – Modifies the abutment box of the current cell

Synopsys

```
ResizeAb ( dx1, dy1, dx2, dy2 )
```

Description

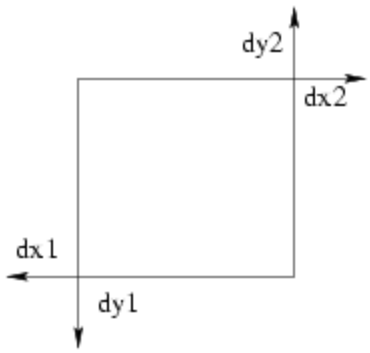
This function modifies the abutment box of the current cell.
The coordinates of the abutment box are the coordinates of the envelop of the abutment boxes of each instance plus the delta values given as argument.

Note that one can not call this function in order to create the abutment box. This fonction only modifies the already created abutment box.

Parameters

- `(dx1, dy1)` : Values to be substracted to the lower left corner of the previous abutment box.
- `(dx2, dy2)` : Values to be added to the upper right corner of the previous abutment box.

The Values are used as follow :



Example

```
% Expansion of the abutment box at the top and the bottom
ResizeAb ( 0, 100, 0, 100 )
```

Errors

Some errors may occur :

- `[Stratus ERROR] ResizeAb :`
  `Coordinates of an abutment Box in y must be multiple of the slice.`
  `Coordinates of an abutment Box in x must be multiple of the pitch.`
  One has called ResizeAb with non authorized values
- `[Stratus ERROR] ResizeAb :`
  `one of the values of dx1 or dx2 (dy1 or dy2) is incompatible with the size of the abutment box.`

`Coordinates of an abutment Box in x must be multiple of the pitch.`
One has called ResizeAb with a value which deteriorates the abtument box

D.4 Place and Route

D.4.1 PlaceSegment

Name

PlaceSegment – Places a segment

Synopsys

```
PlaceSegment ( net, layer, point1, point2, width )
```

Description

Placement of a segment.
The segment is created between `point1` and `point2` on the layer `layer` and with width `width`. It belongs to the net `net`.
Note that the segment must be horizontal or vertival.

Parameters

- `net` : Net which the segment belongs to
- `layer` : Layer of the segment.
  The `layer` argument is a string wich can take different values, thanks to the technology (file described in HUR_TECHNO_NAME)
  - NWELL, PWELL, ptie, ntie, pdif, ndif, ntrans, ptrans, poly, ALU1, ALU2, ALU3, ALU4, ALU5, ALU6, VIA1, VIA2, VIA3, VIA4, VIA5, TEXT, UNDEF, SPL1, TALU1, TALU2, TALU3, TALU4, TALU5, TALU6, POLY, NTIE, PTIE, NDIF, PDIF, PTRANS, NTRANS, CALU1, CALU2, CALU3, CALU4, CALU5, CALU6, CONT_POLY, CONT_DIF_N, CONT_DIF_P, CONT_BODY_N, CONT_BODY_P, via12, via23, via34, via45, via56, via24, via25, via26, via35, via36, via46, CONT_TURN1, CONT_TURN2, CONT_TURN3, CONT_TURN4, CONT_TURN5, CONT_TURN6
- `point1`, `point2` : The segment is created between those two points

Example

```
PlaceSegment ( myNet, "ALU3", XY (10, 0), XY (10, 100), 2 )
```

Errors

Some errors may occur :

- `[Stratus ERROR] PlaceSegment : Argument layer must be a string.`
- `[Stratus ERROR] PlaceSegment : Wrong argument, the coordinates of the segment must be put in XY objects.`
- `[Stratus ERROR] PlaceSegment : Segments are vertical or horizontal.`
  The two references given as argument do not describe a vertical or horizontal segment. Wether coordinate x or y of the references must be identical.

D.4.2 PlaceContact

Name

PlaceContact – Places a contact

Synopsys

```
PlaceContact ( net, layer, point, width, height )
```

Description

Placement of a contact.
The contact is located at the coodinates of `point`, on the layer `layer` and has a size of 1 per 1. It belongs to the net `net`.
Note that the segment must be horizontal or vertival.

Parameters

- `net` : Net which the contact belongs to
- `layer` : Layer of the segment.
  The `layer` argument is a string wich can take different values, thanks to the technology (file described in HUR_TECHNO_NAME)
  - NWELL, PWELL, ptie, ntie, pdif, ndif, ntrans, ptrans, poly, ALU1, ALU2, ALU3, ALU4, ALU5, ALU6, VIA1, VIA2, VIA3, VIA4, VIA5, TEXT, UNDEF, SPL1, TALU1, TALU2, TALU3, TALU4, TALU5, TALU6, POLY, NTIE, PTIE, NDIF, PDIF, PTRANS, NTRANS, CALU1, CALU2, CALU3, CALU4, CALU5, CALU6, CONT_POLY, CONT_DIF_N, CONT_DIF_P, CONT_BODY_N, CONT_BODY_P, via12, via23, via34, via45, via56, via24, via25, via26, via35, via36, via46, CONT_TURN1, CONT_TURN2, CONT_TURN3, CONT_TURN4, CONT_TURN5, CONT_TURN6
- `point` : Coodinates of the contact
- `width` : Width of the contact
- `height` : Height of the contact

Example

```
PlaceContact ( myNet, "ALU2", XY (10, 0), 2, 2 )
```

Errors

Some errors may occur :
- `[Stratus ERROR] PlaceContact : Argument layer must be a string.`
- `[Stratus ERROR] PlaceContact : Wrong argument, the coordinates of the contact must be put in a XY object.`

D.4.3 PlacePin

Name

PlacePin – Places a pin

Synopsys

```
PlacePin ( net, layer, direction, point, width, height )
```

Description

Placement of a pin.
The pin is located at the coodinates of `point`, on the layer `layer`, has a a direction of `direction` and size of 1 per 1. It belongs to the net `net`.

Parameters

- `net` : Net which the pin belongs to
- `layer` : Layer of the segment.
  The `layer` argument is a string wich can take different values, thanks to the technology (file described in HUR_TECHNO_NAME)
  - NWELL, PWELL, ptie, ntie, pdif, ndif, ntrans, ptrans, poly, ALU1, ALU2, ALU3, ALU4, ALU5, ALU6, VIA1, VIA2, VIA3, VIA4, VIA5, TEXT, UNDEF, SPL1, TALU1, TALU2, TALU3, TALU4, TALU5, TALU6, POLY, NTIE, PTIE, NDIF, PDIF, PTRANS, NTRANS, CALU1, CALU2, CALU3, CALU4, CALU5, CALU6, CONT_POLY, CONT_DIF_N, CONT_DIF_P, CONT_BODY_N, CONT_BODY_P, via12, via23, via34, via45, via56, via24, via25, via26, via35, via36, via46, CONT_TURN1, CONT_TURN2, CONT_TURN3, CONT_TURN4, CONT_TURN5, CONT_TURN6
- `direction` : Direction of the pin
  - UNDEFINED, NORTH, SOUTH, EAST, WEST
- `point` : Coodinates of the pin
- `width` : Width of the pin

– `height` : Height of the pin

Example

```
PlacePin ( myNet, "ALU2", NORTH, XY (10, 0), 2, 2 )
```

Errors

Some errors may occur :
– `[Stratus ERROR] PlacePin : Argument layer must be a string.`
– `[Stratus ERROR] PlacePin : Illegal pin access direction. The values are : UNDEFINED, NORTH, SOUTH, EAST, WEST.`
– `[Stratus ERROR] PlacePin : Wrong argument, the coordinates of the pin must be put in a XY object.`

D.4.4 PlaceRef

Name

PlaceRef – Places a reference

Synopsys

```
PlaceRef ( point, name )
```

Description

Placement of a reference.
The reference is located at the coordinates of `point`, with name `name`.

Parameters

– `point` : Coodinates of the reference
– `name` : Name of the reference

Example

```
PlaceRef ( XY (10, 0), "myref" )
```

Errors

Some errors may occur :
– `[Stratus ERROR] PlaceRef : Wrong argument, the coordinates of the reference must be put in a XY object.`
– `[Stratus ERROR] PlaceRef : Argument layer must be a string.`

D.4.5 GetRefXY

Name

GetRefXY – Returns the coordinates of a reference

Synopsys

```
GetRefXY ( pathname, refname )
```

Description

Computation of coordinates.

The point returned (object XY) represents the location of the reference of name `refname` within the coodinates system of the top cell. The reference `refname` is instanciated in an instance found thanks to `pathname` which represents an ordered sequence of instances through the hierarchy.

Parameters

- `pathname` : The path in order to obtain, from the top cell, the instance the reference `refname` belongs to
- `refname` : The name of the reference

Example

The cell which is being created (the top cell), instanciates a generator with instance name "my_dpgen_and2". This generator instanciates an instance called "cell_1" which the reference "i0_20" belongs to.

```
GetRefXY ( "my_dpgen_and2.cell_1", "i0_20" )
```

Errors

Some errors may occur :

- `[Stratus ERROR] GetRefXY :`
  `The instance's path must be put with a string.`
- `[Stratus ERROR] GetRefXY :`
  `The reference must be done with it's name : a string.`
- `[Stratus ERROR] GetRefXY :`
  `No reference found with name ... in masterCell ...`

D.4.6 CopyUpSegment

Name

CopyUpSegment – Copies the segment of an instance in the current cell

Synopsys

```
CopyUpSegment ( pathname, netname, newnet )
```

Description

Duplication of a segment.

The segment is created with the same cordinates and layer as the segment corresponding to the net `netname` in the instance found thanks to `pathname`. It belongs to the net `newnet`.

Note that if several segments correspond to the net, they are all going to be copied.

Parameters

- `pathname` : The path in order to obtain, from the top cell, the instance the net `netname` belongs to
- `netname` : The name of the net which the segment belongs to
- `net` : The net which the top cell segment os going to belong to

Example

```
CopuUpSegment ( "my_dpgen_and2.cell_1", "i0", myNet )
```

Errors

Some errors may occur :

- `[Stratus ERROR] CopyUpSegment :`
  `The instance's path must be put with a string.`
- `[Stratus ERROR] CopyUpSegment :`
  `The segment must be done with it's name : a string.`
- `[Stratus ERROR] CopyUpSegment :`
  `No net found with name ... in masterCell ...`
  There is no net with name `netname` in the instance found thanks to the path `pathname`.
- `[Stratus ERROR] CopyUpSegment :`
  `No segment found with net ... in masterCell ...`
  The net with name `netname` has no segment. So the copy of segment can not be done.
- `[Stratus ERROR] CopyUpSegment :`
  `the segment of net ... are not of type CALU.`
  In other words, the net is not an external net. The copy can be done only with external nets.

D.4.7 PlaceCentric

Name

PlaceCentric – Placement of an instance in the middle of an abutment box

Synopsys

```
PlaceCentric ( ins )
```

Description

This function places an instance in the middle of and abutment box.
The instance has to be instantiated in the method `Netlist` in order to use this function.

Parameters

– `ins` : Instance to place

Errors

Some errors may occur :
– `[Stratus ERROR] PlaceCentric: the instance does not exist.`
The instance must be instanciated in order to be placed.
– `[Stratus ERROR] PlaceCentric :`
`the instance's size is greater than this model.`
The instance must fit in the abutment box. The abutment box may not be big enough.

D.4.8 PlaceGlu

Name

PlaceGlue – Automatic placement of non placed instances

Synopsys

```
PlaceGlue ( cell )
```

Description

This function places, thanks to the automatic placer Mistral of Coriolis, all the non placed instances of the cell.

Parameters

– `cell` : the cell which the fonction is applied to

D.4.9 FillCell

Name

FillCell – Automatic placement of ties.

Synopsys

```
FillCell ( cell )
```

Description

This function places automatically ties.

Parameters

– `cell` : the cell which the fonction is applied to

Errors

Some errors may occur :
– `[Stratus ERROR] FillCell : Given cell doesn't exist.`
The argument is wrong. Check if one has created the cell correctly.

D.4.10 Pads

Name

PadNorth, PadSouth, PadEast, PasWest – Placement of pads at the periphery of the cell

Synopsys

```
PadNorth ( args )
```

Description

These functions place the pads given as arguments at the given side of the cell (PadNorth : up north, PadSouth : down south ...). Pads are placed from bottom to top for PadNorth and PadSouth and from left to right for PadWest and PasEast.

Parameters

– `args` : List of pads to be placed

Example

```
PadSouth ( self.p_cin, self.p_np, self.p_ng, self.p_vssick0
       , self.p_vddeck0, self.p_vsseck1, self.p_vddeck1, self.p_cout
       , self.p_y[0], self.p_y[1], self.p_y[2]
       )
```

Errors

Some errors may occur :

- `[Stratus ERROR] PadNorth : not enough space for all pads.`
  The abutment box is not big enough in order to place all the pads. Maybe one could put pads on other faces of the cell.
- `[Stratus ERROR] PadNorth : one instance doesn't exist.`
  One of the pads given as arguments does not exist
- `[Stratus ERROR] PadNorth : one argument is not an instance.`
  One of the pads is not one of the pads of the cell.
- `[Stratus ERROR] PadNorth : the instance ins is already placed.`
  One is trying to place a pad twice.
- `[Stratus ERROR] PadNorth : pad ins must be closer to the center.`
  The pad name ins must be put closer to the center in order to route the cell

D.4.11 Alimentation rails

Name

AlimVerticalRail, AlimHorizontalRail – Placement of a vertical/horizontal alimentation call back

Synopsys

```
AlimVerticalRail ( nb )
```

Description

These functions place a vertical/horizontal alimentation call back. It's position is given by the parameter given.

Parameters

- `nb` : coordinate of the rail
  - For AlimVerticalRail, `nb` is in pitches i.e. 5 lambdas
  - For AlimHorizontalRail, `nb` is in slices i.e. 50 lambdas

Example

```
AlimVerticalRail ( 50 )
AlimVerticalRail ( 150 )

AlimHorizontalRail ( 10 )
```

Errors

Some errors may occur :

- `[Stratus ERROR] AlimHorizontalRail : Illegal argument y, y must be between ... and ...`
  The argument given is wrong : the call back would not be in the abutment box.
- `[Stratus ERROR] Placement of cells : please check your file of layout with DRUC.`
  The placement of the cell needs to be correct in order to place a call back. Check the errors of placement.

D.4.12 Alimentation connectors

Name

AlimConnectors – Creation of connectors at the periphery of the core of a circuit

Synopsys

```
AlimConnectors()
```

Description

This function creates the connectors in Alu 1 at the periphery of the core.

D.4.13 PowerRing

Name

PowerRing – Placement of power rings.

Synopsys

```
PowerRing ( nb )
```

Description

This function places power rings around the core and around the plots.

Parameters

- `nb` : Number of pair of rings vdd/vss

Example

```
PowerRing ( 3 )
```

Errors

Some errors may occur :

- `[Stratus ERROR] PowerRing : Pads in the north haven't been placed.`
  The pads of the 4 sides of the chip must be placed before calling function PowerRing.
- `[Stratus ERROR] PowerRing : too many rings, not enough space.`
  Wether The argument of PowerRing is to big, or the abutment box of the chip is to small. There's no space to put the rings.

D.4.14 RouteCk

Name

RouteCk – Routing of signal Ck to standard cells

Synopsys

```
RouteCk ( net )
```

Description

This function routes signal Ck to standard cells.

Parameters

- `net` : the net which the fonction is applied to

Errors

Some errors may occur :

- `[Stratus ERROR] RouteCk : Pads in the north haven't been placed`
  The pads must be placed before calling RoutageCk.

D.5 Instanciation facilities

D.5.1 Buffer

Name

Buffer – Easy way to instantiate a buffer

Synopsys

```
netOut <= netIn.Buffer()
```

Description

This method is a method of net. The net which this method is applied to is the input net of the buffer. The method returns a net : the output net.

Note that it is possible to change the generator instanciated with the `SetBuff` method.

Example

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn ( "a", 4 )

    self.S = SignalOut ( "s", 4 )

    self.Vdd = VddIn ( "vdd" )
    self.Vss = VssIn ( "vss" )

  def Netlist ( self ) :

    self.S <= self.A.Buffer()
```

D.5.2 Multiplexor

Name

Mux – Easy way to instantiate a multiplexor

Synopsys

```
netOut <= netCmd.Mux ( arg )
```

Description

This method is a method of net. The net which this method is applied to is the command of the multiplexor. The nets given as parameters are all the input nets. This

method returns a net : the output net.
There are two ways to describe the multiplexor : the argument `arg` can be a list or a dictionnary.

Note that it is possible to change the generator instanciated with the `SetMux` method.

Parameters

- List :
  For each value of the command, the corresponding net is specified. All values must be specified.
  For example :

```
out <= cmd.Mux ( [in0, in1, in2, in3] )
```

  The net out is then initialised like this :

```
if cmd == 0 : out <= in0
if cmd == 1 : out <= in1
if cmd == 2 : out <= in2
if cmd == 3 : out <= in3
```

- Dictionnary :
  A dictionnary makes the correspondance between a value of the command and the corresponding net.
  For example :

```
out <= cmd.Mux ( {"0" : in0, "1" : in1, "2" : in2, "3" : in3} )
```

  This initialisation corresponds to the one before. Thanks to the use of a dictionnary, the connections can be clearer :
  - `'default'` : This key of the dictionnary corresponds to all the nets that are not specified
    For example :

```
out <= cmd.Mux ( {"0" : in0, "default" : in1} )
```

    This notation corresponds to :

```
if cmd == 0 : out <= in0
else : out <= in1
```

    Note that if there is no `'default'` key specified and that not all the nets are specified, the non specified nets are set to 0.
  - `#` and `?` : When a key of the dictionnary begins with `#`, the number after the `#` has to be binary and each `?` in the number means that this bit is not precised
    For example :

```
out <= cmd.Mux ( {"#01?" : in0, "default" : in1} )
```

    This notation corresponds to :

```
if cmd in ( 2, 3 ) : out <= in0
else : out <= in1
```

– , and – : When keys contains thoses symbols, it permits to enumerate intervals For example :

```
out <= cmd.Mux ( {"0,4" : in0, "1-3,5" : in1} )
```

This notation corresponds to :

```
if cmd in ( 0, 4 ) : out <= in0
elif cmd in ( 1, 2, 3, 5) : out <= in1
else : out <= 0
```

Example

```
class essai ( Model ) :

 def Interface ( self ) :
  self.A = SignalIn ( "a", 4 )
  self.B = SignalIn ( "b", 4 )
  self.C = SignalIn ( "c", 4 )
  self.D = SignalIn ( "d", 4 )

  self.Cmd1 = SignalIn ( "cmd1", 2 )
  self.Cmd2 = SignalIn ( "cmd2", 4 )

  self.S1 = SignalOut ( "s1", 4 )
  self.S2 = SignalOut ( "s2", 4 )

  self.Vdd = VddIn ( "vdd" )
  self.Vss = VssIn ( "vss" )

 def Netlist ( self ) :

  self.S1 <= self.Cmd1.Mux ( [sefl.A, self.B, self.C, self.D] )

  self.S2 <= self.Cmd2.Mux ( { "0" : self.A
                      , "1,5-7" : self.B
                      , "#1?1?" : self.C
                      , "default" : self.D
                     } )
```

Errors

Some errors may occur :

– `[Stratus ERROR] Mux : all the nets must have the same lenght.`
All the input nets pust have the same lenght.

– `[Stratus ERROR] Mux : there are no input nets.`
The input nets seem to have been forgotten.

– `[Stratus ERROR] Mux : wrong argument type.`
The connections of the buses are not described by a list nor a dictionnary.

– `[Stratus ERROR] Mux :`
`the number of nets does not match with the lenght of the`

`command.`
When using a list, the number of nets has to correspond to the number of possible values of the command.
- `[Stratus ERROR] Mux : wrong key.`
  One of the key of the dictionnary is not un number, neither a list or an interval.
- `[Stratus ERROR] Mux :`
  `when an interval is specified, the second number of the interval`
  `must be greater than the first one.`
  When creating an interval with "-", the second number has to be greater than the first one.
- `[Stratus ERROR] Mux :`
  `the binary number does not match with the lenght of the command.`
  When using the # notation, each digit of the binary number corresponds to a wire of the cmd. The leghts have to correspond.
- `[Stratus ERROR] Mux : after #, the number has to be binary.`
  When using the # notation, the number has to be binary : one can use 0, 1 or ?.

D.5.3 Shifter

Name

Shift – Easy way to instantiate a shifter

Synopsys

```
netOut <= netCmd.Shift ( netIn, direction, type )
```

Description

This method is a method of net. The net which this method is applied to is the command of the shifter, it's the one which defines the number of bits to shift. The net given as parameter is the input net. The other arguments set the different patameters. The method returns a net : the output net.

Note that it is possible to change the generator instanciated with the `SetShift` method.

Parameters

- `netIn` : the net which is going to be shifted
- `direction` : this string represents the direction of the shift :
  - "left"
  - "right"
- `type` : this string represents the type of the shift :

- "logical" : only "zeros" are put in the net
- "arith" : meaningful for "right" shift, the values put in the nets are an extension of the MSB
- "circular" : the values put in the nets are the ones which have just been taken off

Example

```
class essai ( Model ) :

 def Interface ( self ) :
  self.A = SignalIn ( "a", 4 )

  self.Cmd = SignalIn ( "cmd", 2 )

  self.S1 = SignalOut ( "s1", 4 )
  self.S2 = SignalOut ( "s2", 4 )
  self.S3 = SignalOut ( "s3", 4 )

  self.Vdd = VddIn ( "vdd" )
  self.Vss = VssIn ( "vss" )

 def Netlist ( self ) :

  self.S1 <= self.Cmd.Shift ( self.A, "right", "logical" )
  self.S2 <= self.Cmd.Shift ( self.A, "right", "arith" )

  self.S3 <= self.Cmd.Shift ( self.A, "left", "circular" )
```

If the value of "a" is "0b1001" and the value of "cmd" is "0b10", we will have :

- "s1" : "0b0010"
- "s2" : "0b1110"
- "s3" : "0b0110"

Errors

Some errors may occur :

- `[Stratus ERROR] Shift :`
  `The input net does not have a positive arity.`
  The net which is going to be shifted must have a positive arity.
- `[Stratus ERROR] Shift :`
  `The direction parameter must be "left" or "right".`
  The "direction" argument is not correct.
- `[Stratus ERROR] Shift :`
  `The type parameter must be "logical" or "arith" or "circular".`
  The "type" argument is not correct.

D.5.4 Register

Name

Reg – Easy way to instantiate a register

Synopsys

```
netOut <= netCk.Reg ( netIn )
```

Description

This method is a method of net. The net which this method is applied to is the clock of the register. The net given as parameter is the input net. The method returns a net : the output net.

Note that it is possible to change the generator instanciated with the `SetReg` method.

Example

```
class essai ( Model ) :

 def Interface ( self ) :
  self.A = SignalIn ( "a", 4 )
  self.S = SignalOut ( "s", 4 )

  self.Ck = CkIn ( "ck" )

  self.Vdd = VddIn ( "vdd" )
  self.Vss = VssIn ( "vss" )

 def Netlist ( self ) :

  self.S <= self.Ck.Reg ( self.A )
```

Errors

Some errors may occur :

- `[Stratus ERROR] Reg : The input net does not have a positive arity.`
  The input net must have a positive arity.
- `[Stratus ERROR] Reg : The clock does not have a positive arity.`
  The clock must have a positive arity.

D.5.5 Constants

Name

Constant – Easy way to instantiate constants

Synopsys

```
netOne <= One ( 2 )

net8 <= "8"
```

Description

These functions simplify the way to instanciate constants.

- The functions `One` and`Zero` permits to initialise all the bits of a net to 'one' or 'zero'.
- The instanciation of a constant thanks to a string can be done in decimal, hecadecimal or binary.

Parameters

- For `One` and `Zero` :
  - `n` : the arity of the net
- For the instanciation of a constant :
  - the constant given must be a string representing :
    - A decimal number
    - A binary number : the string must begin with "0b"
    - An hexadecimal number : the string must begin with "0x"

Example

```
class essai ( Model ) :

 def Interface ( self ) :
  self.Ones = SignalOut ( "ones", 2 )
  self.Zeros = SignalOut ( "zeros", 4 )

  self.Eight = SignalOut ( "eight", 4 )
  self.Twentu = SignalOut ( "twenty", 5 )
  self.Two = SignalOut ( "two", 5 )

  self.Vdd = VddIn ( "vdd" )
  self.Vss = VssIn ( "vss" )

 def Netlist ( self ) :

  self.Ones <= One ( 2 )
  self.Zero <= Zero ( 4 )
```

```
  self.Eight <= "8"
  self.Twenty <= "0x14"
  self.Two <= "0b10"
```

Errors

Some errors may occur :
- `[Stratus ERROR] Const :`
  `the argument must be a string representing a number in decimal, binary (0b) or hexa (0x).`
  The string given as argument does not have the right form.

D.5.6 Boolean operations

Description

Most common boolean operators can be instantiated without the `Inst` constructor.

List

Boolean operators are listed below :
- `And2:q <= i0 & i1`
- `Or2:q <= i0 | i1`
- `Xor2:q <= i0 ^ i1`
- `Inv:q <= ~i0`

Generators to instantiate

One can choose the generator to be used. Some methods are applied to the cell and set the generator used when using `&`, `|`, `^` and `~`. The generators used by default are the ones from the virtual library.

Methods are :
- `SetAnd`
- `SetOr`
- `SetXor`
- `SetNot`

Example

```
class essai ( Model ) :

 def Interface ( self ) :
  self.A = SignalIn ( "a", 4 )
  self.B = SignalIn ( "b", 4 )
  self.B = SignalIn ( "c", 4 )
```

```
  self.S = SignalOut ( "s", 4 )

  self.vdd = VddIn ( "vdd" )
  self.vss = VssIn ( "vss" )

def Netlist ( self ) :

  self.S <= ( ~self.A & self.B ) | self.C
```

Errors

Some errors may occur :

- `[Stratus ERROR] & : the nets must have the same lenght.` When one uses boolean expressions, one has to check that the sizes of both nets are equivalent.
- `[Stratus ERROR] : there is no alim.` The cell being created does not have the alimentation nets. The instanciation is impossible.

D.5.7 Arithmetical operations

Description

Most common arithmetic operators can be instantiated without the `Inst` constructor.

List

Arithmetical operators are listed below :

- `Addition:q <= i0 + i1`
- `Substraction:q <= i0-i1`
- `Multiplication:q <= i0 * i1`
- `Division:q <= i0 / i1`

Generators to instantiate

One can choose the generator to be used. Some methods are applied to the cell and set the generator used when using overloard. Methods are :

- `SetAdd` (for addition and substraction)
- `SetMult`
- `SetDiv`

The generators used by default are :

- `Addition` : Sklansky adder
- `Substraction` : Sklansky adder + inversor + cin = '1'
- `Multiplication` : CA2 multiplier (signed, modified booth/Wallace tree)

– `Division` : not available yet

Example

```
class essai ( Model ) :

 def Interface ( self ) :
  self.A = SignalIn ( "a", 4 )
  self.B = SignalIn ( "b", 4 )

  self.S = SignalOut ( "s", 4 )
  self.T = SignalOut ( "t", 8 )

  self.vdd = VddIn ( "vdd" )
  self.vss = VssIn ( "vss" )

 def Netlist ( self ) :
  self.S <= self.A + self.B
  self.T <= self.A * self.B
```

Errors

Some errors may occur :

– `[Stratus ERROR] + : the nets must have the same lenght.`
When one uses arithmetic expressions, one has to check that the sizes of both nets are equivalent.

– `[Stratus ERROR] : there is no alim.`
The cell being created does not have the alimentation nets. The instanciation is impossible.

D.5.8 Comparison operations

Name

Eq/Ne : Easy way to test the value of the nets

Synopsys

```
netOut <= net.Eq ( "n" )
```

Description

Comparaison functions are listed below :

– `Eq` : returns `true` if the value of the net is equal to `n`.

– `Ne` : returns `true` if the value of the net is different from `n`.

Note that it is possible to change the generator instanciated with the `SetComp` method.

Parameters

The constant given as argument must be a string representing :
- A decimal number
- A binary number : the string must begin with "0b"
- An hexadecimal number : the string must begin with "0x"

Example

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn ( "a", 4 )

    self.S = SignalOut ( "s", 1 )
    self.T = SignalOut ( "t", 1 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    self.S <= self.A.Eq ( "4" )
    self.T <= self.A.Ne ( "1" )
```

Errors

Some errors may occur :
- `[Stratus ERROR] Eq : the number does not match with the net's lenght.`
  When one uses comparaison functions on one net, one has to check that the number corresponds to the size of the net.
- `[Stratus ERROR] Eq : the argument must be a string representing a number in decimal, binary (0b) or hexa (0x).`
  The string given as argument does not have the right form.

D.6 Virtual library

Description

The virtual library permits to create a cell and map it to different libraries without having to change it.

List of the generators provided

- `a2:q <= i0 & i1`
- `a3:q <= i0 & i1 & i2`

- a4:q <= i0 & i1 & i2 & i3
- na2:nq <= ~ (i0 & i1)
- na3:nq <= ~ (i0 & i1 & i2)
- na4:nq <= ~ (i0 & i1 & i2 & i3)
- o2:q <= i0 & i1
- o3:q <= i0 & i1 & i2
- o4:q <= i0 & i1 & i2 & i3
- no2:nq <= ~ (i0 & i1)
- no3:nq <= ~ (i0 & i1 & i2)
- no4:nq <= ~ (i0 & i1 & i2 & i3)
- inv:nq <= ~ i
- buf:q <= i
- xr2:q <= i0 ^ i1
- nxr2:nq <= ~ (i0 ^ i1)
- zero:nq <= '0'
- one:q <= '1'
- halfadder:sout <= a ^ b and cout <= a & b
- fulladder:sout <= a ^ b ^ cin
  and cout <= (a & b) | (a & cin) | (b & cin)
- mx2:q <= (i0 & ~cmd) | (i1 & cmd)
- nmx2:nq <= ~((i0 & ~cmd) | (i1 & cmd))
- sff:if RISE (ck) : q <= i
- sff2:if RISE (ck) : q <= (i0 & ~cmd) | (i1 & cmd)
- sff3:if RISE (ck) :
  q <= (i0 & ~cmd0) | (((i1 & cmd1)|(i2&~cmd1)) & cmd0)
- ts:if cmd : q <= i
- nts:if cmd : nq <= ~i

Mapping file

The virtual library is mapped to the sxlib library. A piece of the corresponding mapping file is shown below.

In order to map the virtual library to another library, on has to write a .xml file which makes correspond models and interfaces.

Note that the interfaces of the cells must be the same (except for the names of the ports). Otherwise, one has to create .vst file in order to make the interfaces match.

The environment variable used to point the right file is `STRATUS_MAPPING_NAME`.

Generators

Some generators are also provided in order to use the cells of the library with nets of more than 1 bit. One has to upper the first letter of the model name in order to

```
<?xml version="1.0" encoding='us-ascii'?>

<technology name="sxlib">
 <model name="a2"   realcell="a2_x2"   i0="i0" i1="i1"  q="q"  vdd="vdd" vss="vss"></model>
 <model name="na2"  realcell="na2_x1"  i0="i0" i1="i1" nq="nq" vdd="vdd" vss="vss"></model>
 <model name="o2"   realcell="o2_x2"   i0="i0" i1="i1"  q="q"  vdd="vdd" vss="vss"></model>
 <model name="no2"  realcell="no2_x1"  i0="i0" i1="i1" nq="nq" vdd="vdd" vss="vss"></model>
 <model name="xr2"  realcell="xr2_x1"  i0="i0" i1="i1"  q="q"  vdd="vdd" vss="vss"></model>
 <model name="nxr2" realcell="nxr2_x1" i0="i0" i1="i1" nq="nq" vdd="vdd" vss="vss"></model>
 <model name="inv"  realcell="inv_x1"   i="i"          nq="nq" vdd="vdd" vss="vss"></model>
 <model name="buf"  realcell="buf_x2"   i="i"           q="q"  vdd="vdd" vss="vss"></model>
</technology>
```

user those generators. What is simply done is a for loop with the bits of the nets. The parameter `'nbit'` gives the size of the generator.

Example

– Direct instanciation of a cell

```
for i in range ( 4 ) :
  Inst ( 'a2'
      , map = { 'i0' : neti0[i]
              , 'i1' : neti1[i]
              , 'q' : netq[i]
              , 'vdd' : netvdd
              , 'vss' : netvss
              }
      )
```

– Instanciation of a generator

```
Generate ( 'A2', "my_and2_4bits", param = { 'nbit' : 4 } )
Inst ( 'my_and2_4bits'
     , map = { 'i0' : neti0
             , 'i1' : neti1
             , 'q' : netq
             , 'vdd' : vdd
             , 'vss' : vss
             }
     )
```

Errors

Some errors may occur :

– `[Stratus ERROR] Inst : the model ... does not exist. Check CRL_CATA_LIB.`
The model of the cell has not been found. One has to check the environment variable.

- `[Stratus ERROR] Virtual library : No file found in order to parse. Check STRATUS_MAPPING_NAME.`
  The mapping file is not given in the environment variable.

D.7 Data Base

D.7.1 Model

Synopsys

```
class myClass ( Model ) :
   ...

exemple = myClass ( name, param )
```

Description

Every cell made is a class herited from class `Model`. Some methods have to be created, like `Interface`, `Netlist` ... Some methods are inherited from the class `Model`.

Parameters

- `name` : The name of the cell (which is the name of the files which will be created)
- `param` : A dictionnary which gives all the parameters useful in order to create the cell

Attributes

- `_name` : Name of the cell
- `_st_insts` : List of all the instances of the cell
- `_st_ports` : List of all the external nets of the cell (except for alimentations and clock)
- `_st_sigs` : List of all the internal nets of the cell
- `_st_vdds`, `_st_vsss` : Two tabs of the nets which are instancied as `VddIn` and `VssIn`
- `_st_cks` : List of all the nets which are instancied as `CkIn`
- `_st_merge` : List of all the internal nets which have to be merged
- `_param` : The map given as argument at the creation of the cell
- `_underCells` : List of all the instances which are cells that have to be created
- `_and`, `_or`, `_xor`, `_not`, `_buff`, `_mux`, `_reg`, `_shift`, `_comp`, `_add`, `_mult`, `_div` : tells which generator to use when using overloard
- `_NB_INST` : The number of instances of the cell (useful in order to automatically give a name to the instances)
- `_TAB_NETS_OUT` and `_TAB_NETS_CAT` : Lists of all the nets automatically created

– `_insref` : The reference instance (for placement)

And, in connection with Hurricane :
– `_hur_cell` : The hurricane cell (None by default)
– `_db` : The database
– `_lib0 : self._db.Get_CATA_LIB ( 0 )`
– `_nb_alims_verticales, _nb_pins, _nb_vdd_pins, _nb_vss_pins, standard_instances_list, pad_north, pad_south, pad_east, pad_west` : all place and route stuffs ...

Methods

Methods of class `Model` are listed below :
– `HurricanePlug` : Creates the Hurricane cell thanks to the stratus cell.
Before calling this method, only the stratus cell is created, after this method, both cells are created. This method has to be called before View and Save, and before Layout.
– `View` : Opens/Refreshes the editor in order to see the created layout
– `Quit` : Finishes a cell without saving
– `Save` : Saves the created cell
If several cells have been created, they are all going to be saved in separated files

Some of those methods have to be defined in order to create a new cell :
– `Interface` : Description of the external ports of the cell
– `Netlist` : Description of the netlist of the cell
– `Layout` : Description of the layout of the cell
– `Vbe` : Description of the behavior of the cell
– `Pattern` : Description of the patterns in order to test the cell

D.7.2 Nets

Synopsys

```
netInput = SignalIn ( name, arity )
```

Description

Instanciation of net. Differents kind of nets are listed below :
– `SignalIn` : Creation of an input port
– `SignalOut` : Creation of an output port
– `SignalInOut` : Creation of an inout port
– `SignalUnknown` : Creation of an input/output port which direction is not defined
– `TriState` : Creation of a tristate port

- `CkIn` : Creation of a clock port
- `VddIn` : Creation of the vdd alimentation
- `VssIn` : Creation of the vss alimentation
- `Signal` : Creation of an internal net

Parameters

- `name` : Name of the net (mandatory argument)
- `arity` : Arity of the net (mandatory argument)
- `indice` : For buses only : the LSB bit (optional argument : set to 0 by default)

Only `CkIn`, `VddIn` and `VssIn` do not have the same parameters : there is only the `name` parameter (they are 1 bit nets).

Attributes

- `_name` : Name of the net
- `_arity` : Arity of the net (by default set to 0)
- `_ind` : LSB of the net
- `_ext` : Tells if the net is external or not (True/False)
- `_direct` : If the net is external, tells the direction ("IN", "OUT", "INOUT", "TRI-STATE", "UNKNOWN")
- `_h_type` : If the net is an alimentation or a clock, tells the type ("POWER", "GROUND", "CLOCK")
- `_type` : The arithmetic type of the net ("nr")
- `_st_cell` : The stratus cell which the net is instanciated in
- `_real_net` : If the net is a part of a net (Sig) it is the real net corresponding
- `_alias` : [] by default. When the net has an alias, it's a tab. Each element of the tab correspond to a bit of the net (from the LSB to the MSB), it'a a dictionnary : the only key is the net which this net is an alias from, the value is the bit of the net
- `_to_merge` : [] by default. The same as _alias
- `_to_cat` : [] by default. The same as _alias

And, in connection with Hurricane :
- `_hur_net` : A tab with all the hurricane nets corresponding to the stratus net ; From the LSB to the MSB (for example, with a 1 bit net, one gets the hurricane net by doing : `net._hur_net[0]`).

Methods

- `Buffer` : Instanciation of a Buffer
- `Shift` : Instanciation of a shifter
- `Mux` : Instanciation of a multiplexor

- `Reg` : Instanciation of a register
- `Eq/Ne` : Instanciation of comparison generator
- `Extend` : A net is extended
- `Alias` : A net is an alias of another net
- `Delete` : Deletion of the Hurricane nets

And the overloards :
- `__init__` : Initialisation of nets
- `__le__` : initialisation of a net thanks to <= notation
- `__getitem__`, `__geslice__` : Creation of "Sig" nets : which are part of nets (use of `[]` and `[:]`)
- `__and__`, `__or__`, `__xor__`, `__invert__` : boolean operation with &, |, ~
- `__add__`, `__mul__`, `__div__` : arithmetic operators with +, * and /

D.7.3 Instances

Synopsys

```
Inst ( model
     , name
     , map = myMap
     )
```

Description

Instantiation of an instance. The type of the instance is given by the `model` parameter. The connexions are made thanks to the `map` parameters.

Parameters

- `model` : Name of the mastercell of the instance to create (mandatory argument)
- `name` : Name of the instance (optional)
  When this argument is not defined, the instance has a name created by default. This argument is usefull when one wants to create a layout as well. Indeed, the placement of the instances is much easier when the conceptor has chosen himself the name f the instances.
- `map` : Dictionnary for connexions in order to make the netlist

Attributes

- `_name` : Name of the instance (the name given as parameter if there's one, a name created otherwise)
- `_model` : Name of the model given as argument

- `_real_model` : Name of the model created thanks to `_model` and all the parameters
- `_map` : Dictionnary `map` given at the instanciation
- `_param` : Dictionnary `param` given at the instanciation
- `_st_cell` : The stratus cell which the instance is instanciated in
- `_st_masterCell` : The stratus master cell of the instance

For placement :
- `_plac` : tells if the instance is placed or not (UNPLACED by default)
- `_x`, `_y` : the coordinates of the instance (only for placed instances)
- `_sym` : the symetry of the instance (only for placed instances)

And, in connection with Hurricane :
- `_hur_instance` : The hurricane instance (None by default)
- `_hur_masterCell` : The Hurricane master cell of the instance (None by default)

Methods

- `Delete` : Deletion of the Hurricane instance

Annexe E ArithLib

Sommaire

E.1 Adders 205
E.1.1 lib.fixed.adder.ripple.Ripple Class Reference 205
E.1.2 lib.fixed.adder.sklansky.Sklansky Class Reference 206
E.1.3 lib.fixed.adder.multi_add.MultiAdd Class Reference 208
E.1.4 lib.fixed.adder.mixed.Mixed Class Reference 209
E.1.5 lib.fixed.adder.redundant.Redundant Class Reference 211
E.1.6 lib.fixed.adder.derived.complementer.c2ripple.C2Ripple Class Reference 212
E.1.7 lib.fixed.adder.derived.distance.distance.Distance Class Reference 213
E.1.8 lib.fixed.adder.derived.comparator.sklansky.Sklansky Class Reference 215
E.1.9 lib.fixed.adder.derived.incrementer.sklansky.Sklansky Class Reference 216
E.1.10 lib.fixed.adder.derived.increment_decrement.ripple.Ripple Class Reference 218
E.2 Summation 219
E.2.1 lib.fixed.sum.dadda.Dadda Class Reference 219
E.3 Multipliers 221
E.3.1 lib.fixed.multiplier.nr.Booth Class Reference 221
E.3.2 lib.fixed.multiplier.mixed.Direct Class Reference 222
E.3.3 lib.fixed.multiplier.redundant.Direct Class Reference 224
E.3.4 lib.fixed.multiplier.constant.multcst.MultCst Class Reference . 225
E.4 Converters 226
E.4.1 lib.fixed.adder.converter.BsCs Class Reference 226
E.4.2 lib.fixed.adder.converter.CsBs Class Reference 228
E.5 Other 229
E.5.1 lib.fixed.other.bool.Bool Class Reference 229
E.5.2 lib.fixed.other.bool.Boolx Class Reference 230
E.5.3 lib.fixed.other.shifter.Shifter Class Reference 231
E.5.4 lib.fixed.other.coding.Encoder Class Reference 232

E.1 Adders

E.1.1 lib.fixed.adder.ripple.Ripple Class Reference

Stratus generator : Ripple adder
Construct a Ripple adder to obtain i0 + i1

Interface

- *i0* : first entry (input)
- *i1* : second entry (input)
- *o* : value of i0 + i1 (output)
- *cin* : input carry (input, optionnal)
- *cout* : output carry (input, optionnal)

Architecture

The architecture is the standard Ripple adder architecture :
- the carry ripples down each adder stage until it reaches the last stage

Parameters

- *nbit* is the adder operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *cin* indicates if there is a carry input connector (default value : False)
- *cout* indicates if there is a carry output connector (default value : False)
- *signed* indicates if the operands are signed or not (default value : True)
- *extended* indicates if the output width is greater that input to take account of overflow (default value : False)

Remarks : When extend is set, the output word length is one bit greater than larger input. It is impossible to put cout and extend, it is wether the MSB of the output, wether the output carry.

Exceptions

- Raise `SizeError` if `max(nbit1,nbit2)` < 2

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
- different values for nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin, cout, signed and extended

Reference

– [Zim]

Member Function Documentation

```
def lib.fixed.adder.ripple.Ripple.Pattern ( cls, param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.ripple.Ripple.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Model name string

```
def lib.fixed.adder.ripple.Ripple.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.adder.ripple.Ripple.EvalArea ( cls, param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Area evalutation

```
def lib.fixed.adder.ripple.Ripple.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Propagation time evalutation

E.1.2 lib.fixed.adder.sklansky.Sklansky Class Reference

Stratus generator : Sklansky adder
Construct a Sklansky adder to obtain i0 + i1

Interface

– *i0* : first entry (input)
– *i1* : second entry (input)
– *o* : value of a + b (output)
– *cin* : value of the input carry (optionnal)
– *cout* : value of the output carry (optionnal)

Architecture

The architecture is the standard Sklansky adder architecture :
– a first step of preprocessing to obtain the (pi,gi)
– a second step of processing the Sklansky algorithm
– a third step of postprocessing that compute the effective sum

Parameters

- *nbit* is the adder operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *cin* indicates if there is a carry input connector (default value : False)
- *cout* indicates if there is a carry output connector (default value : False)
- *ovf* indicates if there is an overflow output connector (default value : False)
- *pg* indicates if there are p and g output connectors (default value : False)
- *signed* indicates if the operands are signed or not (default value : True)
- *extended* indicates if the output width is greater that input to take account of overflow (default value : False)

Exceptions

- Raise `SizeError` if `max(nbit1,nbit2)` < 2

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
- different values for nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin, cout, signed and extended

Reference

- [Zim]

Member Function Documentation

```
def lib.fixed.adder.sklansky.Sklansky.Pattern ( cls, param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.sklansky.Sklansky.GetModelName ( cls, param )
```

Generate a valid model name (class method).
Parameters : *param* parameters used for Sklansky instanciation
Returns : Model name string

```
def lib.fixed.adder.sklansky.Sklansky.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.adder.sklansky.Sklansky.EvalArea ( cls, param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Area evalutation

```
def lib.fixed.adder.sklansky.Sklansky.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Propagation time evalutation

E.1.3 lib.fixed.adder.multi_add.MultiAdd Class Reference

Stratus generator : multiple outputs adder
Construct a Sklansky adder to obtain a + b, a + b + 1 and a + b + 2

Interface

- *i0* : first entry (input)
- *i1* : second entry (input)
- *o0* : value of a + b (output)
- *o1* : value of a + b + 1 (output)
- *o2* : value of a + b + 2 (output)

Architecture

The architecture is based on a standard Sklansky algorithm :
- a first step of preprocessing to obtain the half-sum
- a second step of preprocessing to obtain the (pi,gi)
- a third step of processing the Sklansky algorithm
- a 4th step of postprocessing that compute the effective sums

Parameters

- *nbit* is the adder operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)

Exceptions

- Raise `SizeError` if `max(nbit1,nbit2)` < 2

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit0 varying [2 :64] and a MaskNumber of 3
- different value of nbit0 and nbit0 : (8,16) and (16,8)

Reference

- [Zim]

Member Function Documentation

```
def lib.fixed.adder.multi_add.MultiAdd.Pattern ( cls, param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.multi_add.MultiAdd.GetModelName ( cls, param )
```

Parameters : *param* parameters used for MultiAdd instanciation
Returns : Model name string

```
def lib.fixed.adder.multi_add.MultiAdd.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.1.4 lib.fixed.adder.mixed.Mixed Class Reference

Stratus generator : Mixed adder
Construct a Mixed adder : NR + R -> R

Interface

- *i0_0, i0_1* : first entry (input, redundant)
- *i1* : second entry (input, non redundant)
- *o_0, o_1* resp. *o* : value of i0 + i1 (output, redundant, resp. non redundant)
- *cin* : input carry (input, optionnal)

Architecture

The architecture is the Mixed adder architecture :
- all of the columns can be added in parallel without relying on the result of the previous column, creating a two outputs "adder" with a time delay that is independent of the size of its inputs

Parameters

- *nbit* is the adder operand word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *i0* is the type of the redundant input : "cs" or "bs" (default value : "cs")
- *o* is the type of the output : "ca2" or "cs" or "bs"
- *cin* indicates if there is a carry input connector (default value : False)
- *signed* indicates if the values are signed or not (default value : True)
- *extended* indicates if the result is on n+1 bits (resp. n+2) if redundant (resp. non redondant) or n bits (default value : False)

Exceptions

– Raise `SizeError` if `max(nbit1,nbit2)` < 2

Validation

The validation is done from simulation with :
– a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
– different values for nbit0 and nbit1 : (8,16) and (16,8)
– all configurations of having or not cin, signed and extended

Reference

– [Dum01]

Member Function Documentation

```
def lib.fixed.adder.mixed.Mixed.Pattern ( cls, param )
```

Parameters : *param* parameters used for Mixed instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.mixed.Mixed.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Mixed instanciation
Returns : Model name string

```
def lib.fixed.adder.mixed.Mixed.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.adder.mixed.Mixed.EvalArea ( cls, param )
```

Parameters : *param* parameters used for Mixed instanciation
Returns : Area evalutation

```
def lib.fixed.adder.mixed.Mixed.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Mixed instanciation
Returns : Propagation time evalutation

E.1.5 lib.fixed.adder.redundant.Redundant Class Reference

Stratus generator : Redundant adder
Construct a redundant adder : R + R -> R

Interface

- *i0_0, i0_1* : first entry (input, redundant)
- *i1_0, i1_1* : second entry (input, redundant)
- *cin0*, cin1 : carry entries (input)
- *o_0, o_1* resp. *o* : value of i0 + i1 (output, redundant, resp. non redundant)
- *cin0* : input carry (input, optionnal)
- *cin1* : input carry (input, optionnal)

Architecture

The architecture is the Redundant adder architecture :

- all of the columns can be added in parallel without relying on the result of the previous column, creating a two outputs "adder" with a time delay that is independent of the size of its inputs

Parameters

- *nbit* is the adder operand word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *i0* is the type of the first redundant input : "cs" or "bs" (default value : "cs")
- *i1* is the type of the second redundant input : "cs" or "bs" (default value : "cs")
- *o* is the type of the output : "ca2" or "cs" or "bs"
- *cin0* indicates if the cin0 carry input connector exists or not (default value : False)
- *cin1* indicates if the cin1 carry input connector exists or not (default value : False)
- *signed* indicates if the values are signed or not (default value : True)
- *extended* indicates if the result is on n+1 bits (resp. n+2) if redondant (resp. non redondant) or n bits (default value : False)

Exceptions

- Raise `SizeError` if `max(nbit1,nbit2)` < 2

Validation

The validation is done from simulation with :

- a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
- different values for nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin0, cin1, signed and extended

Reference

- [Dum01]

Member Function Documentation

```
def lib.fixed.adder.redundant.Redundant.Pattern ( cls, param )
```

Parameters : *param* parameters used for Redundant instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.redundant.Redundant.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Redundant instanciation
Returns : Model name string

```
def lib.fixed.adder.redundant.Redundant.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.adder.redundant.Redundant.EvalArea ( cls, param )
```

Parameters : *param* parameters used for Redundant instanciation
Returns : Area evalutation

```
def lib.fixed.adder.redundant.Redundant.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Redundant instanciation
Returns : Propagation time evalutation

E.1.6 lib.fixed.adder.derived.complementer.c2ripple.C2Ripple Class Reference

Stratus generator : two's complementer (ripple architecture)
Construct a two's complementer

Interface

- *i* : number to complement (input)
- *q* : i two's complement (output)

Architecture

The architecture is derived from ripple adder. It uses only xor and or logic gates except for the buffer used to connect *i[0]* and *q[0]*.

Parameters

- *wl* is the complementer word length in bits

Exceptions

- Raise `SizeError` if `wl)` < 2

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
- different value of nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin, cout, signed and extended

Member Function Documentation

```
def lib.fixed.adder.derived.complementer.c2ripple.C2Ripple.GetParam ( cls,
param )
```

Parameters : *param* parameters used for C2Ripple instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.derived.complementer.c2ripple.C2Ripple.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.1.7 lib.fixed.adder.derived.distance.distance.Distance Class Reference

Stratus generator : Distance (Sklansky algorithm)
Construct a distance operator to obtain abs (i0 - i1)

Interface

- *i0* : first entry (input)
- *i1* : second entry (input)
- *o* : value of the distance (output)
- *ge* : optionnal indicator if a >= b (output)

Architecture

The architecture is based on a multiple output adder proposed by Alain Guyot with a Sklansky algorithm

Parameters

- *nbit* is the distance operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *hasGe* indicates if there is a connector for ge output (optional, default value : False)

Exceptions

– Raise `SizeError` if $nbit < 2$

Validation

The validation is done from simulation with a varying of nbit in [2 :64] and a Mask-Number of 3

Reference

– [Guy]

Remarks

– the inputs and output are unsigned numbers

Member Function Documentation

```
def lib.fixed.adder.derived.distance.distance.Distance.Pattern ( cls,
param )
```

Parameters : *param* parameters used for Distance instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.derived.distance.distance.Distance.GetModelName ( cls,
param )
```

Parameters : *param* parameters used for Distance instanciation
Returns : Model name string

E.1.8 lib.fixed.adder.derived.comparator.sklansky.Sklansky Class Reference

Stratus generator : Comparator
Construct a comparator that compares 2 given signals

Interface

– *i0* : first entry (input)
– *i1* : second entry (input)
– *eq* : indicator if $i0 == i1$ (optionnal output)
– *ne* : indicator if $i0! = i1$ (optionnal output)
– *ge* : indicator if $i0 >= i1$ (optionnal output)
– *le* : indicator if $i0 <= i1$ (optionnal output)
– *lt* : indicator if $i0 < i1$ (optionnal output)
– *gt* : indicator if $i0 > i1$ (optionnal output)

Architecture

The architecture is based on the standard Sklansky prefix adder architecture :
- a first step of preprocessing to obtain the (pi,gi)
- a second step of processing the PG tree
- the ge output correspond to cout and eq to Pn-1 :0, the others outputs are boolean equation of ge and eq

Parameters

- *nbit* is the adder operands word length in bits when a and b have the same width
- *nbit0* is the adder 1st operand word length in bits
- *nbit1* is the adder 2nd operand word length in bits
- *type* select output connectors to create by setting the corresponding bits as following :

```
  | bit weight | 5 | 4 | 3 | 2 | 1 | 0 |
  | connector enabled | eq | ne | ge | le | lt | gt |
```

For example type = 0b100100 corresponds to a comparator with the *eq* and *le* flags. To simplify constants has been defined, with the same example we get :

```
from fixed.adder.derived.comparator.sklansky import Sklansky
 ...
 ...
type = Sklansky.EQ & Sklansky.LE
```

Exceptions

- Raise `SizeError` if `max(nbit1,nbit2)` < 2
- Raise `ValueError` if `type` == 0

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
- different value of nbit0 and nbit1 : (8,16) and (16,8)
- all the value for type for a same width of 8 bits

Reference

- [Zim]

Remarks

- At the moment, this comparator uses only unsigned numbers

Member Function Documentation

```
def lib.fixed.adder.derived.comparator.sklansky.Sklansky.Pattern ( cls,
param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.derived.comparator.sklansky.Sklansky.GetModelName ( cls,
param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Model name string

```
def lib.fixed.adder.derived.comparator.sklansky.Sklansky.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.1.9 lib.fixed.adder.derived.incrementer.sklansky.Sklansky Class Reference

Stratus generator : Sklansky incrementer
Construct a sklansky incrementer to obtain i + 1

Interface

- *i* : entry (input)
- *o* : value of i + 1 (output)
- *cin* : value of the input carry (optionnal)
- *cout* : value of the output carry (optionnal)

Architecture

The architecture is the standard sklansky incrementer architecture :
- a step of processing the PG tree
- a last step of postprocessing that compute the effective sum

Parameters

- *nbit* is the incrementer operand word length in bits
- *algo* is the used algorithm
- *cin* indicates if there is a carry input connector (default value : False)
- *cout* indicates if there is a carry output connector (default value : False)

Exceptions

- Raise `SizeError` if $nbit < 2$
- Raise `ValueError` if `algo` is not sklansky (for the moment, only sklansky is implemented)

Validation

The validation is done from simulation with :
- a width varying [2 :64] and a MaskNumber of 3
- all configurations of having or not cin and cout

Reference

- [Zim]

Member Function Documentation

```
def lib.fixed.adder.derived.incrementer.sklansky.Sklansky.Pattern ( cls,
param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.derived.incrementer.sklansky.Sklansky.GetModelName ( cls,
param )
```

Parameters : *param* parameters used for Sklansky instanciation
Returns : Model name string

```
def lib.fixed.adder.derived.incrementer.sklansky.Sklansky.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.1.10 lib.fixed.adder.derived.increment_decrement.ripple.Ripple Class Reference

Stratus generator : Ripple increment/decrement generator.
Construct a ripple increment/decrement to obtain (i+1) or (i-1)

Interface

- *i* : entry (input)
- *o* : value of i + 1 (output)
- *dec* : select increment (0) or decrement (1) (optionnal)
- *cin* : value of the input carry (optionnal)
- *cout* : value of the output carry (optionnal)

Architecture

The architecture is the standard ripple increment/decrement architecture

Parameters

- *nbit* is the generator operand word length in bits
- *type* indicates the type of operator (0 : incrementer, 1 : decrementer, 2 : both)
- *cin* indicates if there is a carry input connector (default value : False)
- *cout* indicates if there is a carry output connector (default value : False)

Exceptions

- Raise `SizeError` if $nbit < 2$
- Raise `ValueError` if `type` is not 0, 1 or 2

Validation

The validation is done from simulation with :
- a width varying [2 :64] and a MaskNumber of 3
- all types of operators
- all configurations of having or not cin, cout and dec

Reference

- [Zim]

Member Function Documentation

```
def lib.fixed.adder.derived.increment_decrement.ripple.Ripple.Pattern ( cls,
param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.derived.increment_decrement.ripple.Ripple.GetModelName
( cls, param )
```

Parameters : *param* parameters used for Ripple instanciation
Returns : Model name string

```
def lib.fixed.adder.derived.increment_decrement.ripple.Ripple.GetParam
( cls )
```

Returns : Dictionary of all the parameters with their types

E.2 Summation

E.2.1 lib.fixed.sum.dadda.Dadda Class Reference

Stratus generator : Dadda's summation operator generator

Construct a summation operator with Dadda's improved Wallace tree structure

If N is the number of summands, lsb_{a_i} the lsb positions corresponding to the input signals and s0, s1 the result then :

$$s0 + s1 = \sum_{i=1}^{N} a_i.2^{lsb_{a_i}}$$

Representations

Inputs are either using simple position numeration or two's complement

If two's complement representation is used, then sign extension must be done up to the greatest possible output and internal growth limitation must be set to this value (Fadavi-Ardekani algorithm can also be used to avoid sign extension)

Output is in carry save representation (sum = s0+s1)

Complexity

If N is the number of summands then the delay is in : $O(log_{3/2}N)$

Parameters

- signals names and length, mandatory (example : `'sig1' : 8`)
- signals lsb position, optional (default value : 0) (example : `'sig1_lsb' : 2`)
- maximum output length, optional (default value : set by the tree growth, else the tree growth is limited to max_wl)

Interface

- input signal connectors : name and length of these connectors are determined by the generator parameters
- s0, s1 : summation result in carry-save representation

Validation

The validation is done in tree steps :

- firstly a pattern test vectors is created ; for each patterns input signals values are randomly set
- then the pattern a simulated without output signal values check and with zero delay

– finaly the output pattern is analyzed, for each test vector the output result is checked against the input values

Exceptions

– Raise ValueError if *max_wl* is too small

References

– [Dad65, Mul89]

Member Function Documentation

```
def lib.fixed.sum.dadda.Dadda.Pattern ( cls, param )
```

Parameters : *param* parameters used for Dadda instanciation
Returns : Pattern file for simulation

```
def lib.fixed.sum.dadda.Dadda.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Dadda instanciation
Returns : Model name string

```
def lib.fixed.sum.dadda.Dadda.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.3 Multipliers

E.3.1 lib.fixed.multiplier.nr.Booth Class Reference

Stratus generator : Booth multiplier
Construct a multiplier to obtain i0 ∗ i1

Interface

– *i0* : first entry (input, non redondant)
– *i1* : second entry (input, non redondant)
– *o* resp. *o_0*, *o_1* : value of [(i0+cin_i0) ∗ (i1+cin_i1)] + cin (output, non redundant resp. redundant)
– *cin* : input carry (input, optionnal)
– *cin_i0* : input carry (input, optionnal)
– *cin_i1* : input carry (input, optionnal)

Architecture

The architecture is :
- Booth modified algorithm
- Wallace tree
- Sklansky adder

Parameters

- *nbit* is the multiplier operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *o* is the type of the output of the multiplier : "ca2" or "cs" (default value : "ca2")
- *cin* indicates if there is a carry input connector (default value : False)
- *cin_i0* indicates if there is a carry input connector (default value : False)
- *cin_i1* indicates if there is a carry input connector (default value : False)

Exceptions

- Raise `SizeError` if $nbit < 3$

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :32] and a MaskNumber of 3
- different values for nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin, cin_0 and cin_1

Reference

- [Dum01]

Member Function Documentation

```
def lib.fixed.multiplier.nr.Booth.Pattern ( cls, param )
```

Parameters : *param* parameters used for Booth instanciation
Returns : Pattern file for simulation

```
def lib.fixed.multiplier.nr.Booth.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Booth instanciation
Returns : Model name string

```
def lib.fixed.multiplier.nr.Booth.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.multiplier.nr.Booth.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Booth instanciation
Returns : Propagation time evalutation

E.3.2 lib.fixed.multiplier.mixed.Direct Class Reference

Stratus generator : Multiplier
Construct a multiplier to obtain i0 $*$ i1

Interface

- *i0_0, i0_1* : first entry (input, redundant)
- *i1* : second entry (input, non redondant)
- *o* resp. *o_0, o_1* : value of [(i0+cin_i0) $*$ (i1+cin_i1)] + cin (output, non redundant resp. redundant)
- *cin* : input carry (input, optionnal)
- *cin_i0* : input carry (input, optionnal)
- *cin_i1* : input carry (input, optionnal)

Architecture

The architecture is composed of :
- the direct algorithm
- a Wallace tree
- a Sklansky adder

Parameters

- *nbit* is the adder operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *i0* is the type of the redundant input of the multiplier : "cs" or "bs" (default value : "cs")
- *o* is the type of the output of the multiplier : "ca2" or "cs" (default value : "cs")
- *cin* indicates if there is a carry input connector (default value : False)
- *cin_i0* indicates if there is a carry input connector (default value : False)
- *cin_i1* indicates if there is a carry input connector (default value : False)
- *extended* indicates if the result is on nbit0+nbit1+1 bits (resp. nbit0+nbit1+2) if redundant (resp. non redundant) or nbit0+nbit1 bits (default value : False)

Exceptions

- Raise `SizeError` if $nbit < 3$

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :32] and a MaskNumber of 3
- different values for nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin, cin_0 and cin_1

Reference

- [Dum01]

Member Function Documentation

```
def lib.fixed.multiplier.mixed.Direct.Pattern ( cls, param )
```

Parameters : *param* parameters used for Direct instanciation
Returns : Pattern file for simulation

```
def lib.fixed.multiplier.mixed.Direct.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Direct instanciation
Returns : Model name string

```
def lib.fixed.multiplier.mixed.Direct.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.multiplier.mixed.Direct.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Direct instanciation
Returns : Propagation time evalutation

E.3.3 lib.fixed.multiplier.redundant.Direct Class Reference

Stratus generator : Multiplier
Construct a multiplier to obtain i0 $*$ i1

Interface

- *i0_0, i0_1* : first entry (input, redundant)
- *i1_0, i1_1* : second entry (input, redundant),
- *o* resp. (*o_0, o_1* : value of (i0+cin_i0) $*$ (i1+cin_i1) + cin (output, non redundant resp. redundant)
- *cin* : input carry (input, optionnal)
- *cin_i0* : input carry (input, optionnal)
- *cin_i1* : input carry (input, optionnal)

Architecture

The architecture is :
- Direct algorithm
- Wallace tree
- Sklansky adder

Parameters

- *nbit* is the adder operands word length in bits when i0 and i1 have the same width
- *nbit0* is i0's lenght in bits (if *nbit* not specified)
- *nbit1* is i1's lenght in bits (if *nbit* not specified)
- *i0* is the i0's type : "cs" or "bs" (default value : "cs")
- *i0* is the i1's type : "cs" or "bs" (default value : "cs")
- *o* is the type of the output of the multiplier : "ca2" or "cs" (default value : "cs")
- *cin* indicates if there is a carry input connector (default value : False)
- *cin_i0* indicates if there is a carry input connector (default value : False)
- *cin_i1* indicates if there is a carry input connector (default value : False)
- *extended* indicates if the result is on nbit0+nbit1+1 bits (resp. nbit0+nbit1+2) if redundant (resp. non redundant) or nbit0+nbit1 bits (default value : False)

Exceptions

- Raise `SizeError` if $nbit < 3$

Validation

The validation is done from simulation with :
- a same width for nbit0 and nbit1 varying [2 :32] and a MaskNumber of 3
- different values for nbit0 and nbit1 : (8,16) and (16,8)
- all configurations of having or not cin, cin_0 and cin_1

Reference

- [Dum01]

Member Function Documentation

```
def lib.fixed.multiplier.redundant.Direct.Pattern ( cls, param )
```

Parameters : *param* parameters used for Direct instanciation
Returns : Pattern file for simulation

```
def lib.fixed.multiplier.redundant.Direct.GetModelName ( cls, param )
```

Parameters : *param* parameters used for Direct instanciation

Returns : Model name string

```
def lib.fixed.multiplier.redundant.Direct.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

```
def lib.fixed.multiplier.redundant.Direct.EvalTime ( cls, param )
```

Parameters : *param* parameters used for Direct instanciation
Returns : Propagation time evalutation

E.3.4 lib.fixed.multiplier.constant.multcst.MultCst Class Reference

Stratus generator : multiplier by a constant
Construct a multiplier by a constant operator, the generator has the fallowing specificities :

- partial products number is minimized by using canonical constant recoding
- efficient partials products summation is done using Dadda summation operator
- optional CS (Carry Save) output conversion to two's complement with a Sklansky adder
- internal sign extension is avoided using Fadavi-Ardekani algorithm
- output word length is wl+recoded_cst_wl

Parameters

- *a_wl* : input variable operand word length
- *a_iwl* : if the constant is a float, please indicate input integer word length (used only by Pattern())
- *cst* : constant value (float or integer)
- *cst_wl* : constant word length
- *cst_iwl* : constant integer word length (number of bits of the integer part) :
  - integer constant : must be set to -1
  - float constant : if set to -1, then the minimal value for iwl is used else set the iwl to use
- *ca2* : if true convert the CS output to CA2 representation by adding a Sklansky adder to the output

Interface

- *a* : input connector
- *s0, s1* : output connector if CS output
- *s* : output connector if CA2 output

Validation

The Pattern() method validates the generated model using mask if 'a' world length is greater than 12

The method works with CS and CA2 output
Depending on the constant :
– integer constant : exact matches are tried and number of differences is repported
– fixed point constant, results are saved into a file and the RSM (root-square-mean) error is computed and displayed

Member Function Documentation

```
def lib.fixed.multiplier.constant.multcst.MultCst.Pattern ( cls, param )
```

Parameters : *param* parameters used for MultCst instanciation
Returns : Pattern file for simulation

```
def lib.fixed.multiplier.constant.multcst.MultCst.GetModelName ( cls,
param )
```

Parameters : *param* parameters used for MultCst instanciation
Returns : Model name string

```
def lib.fixed.multiplier.constant.multcst.MultCst.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.4 Converters

E.4.1 lib.fixed.adder.converter.BsCs Class Reference

Stratus generator : converter
Construct a converter BS -> CS

Interface

– *i0_0, i0_1* : entry (input, borrow-save),
– *o_0, o_1* : conversion of i0 (output, carry-save)

Architecture

The architecture is the Mixed adder architecture :
– all of the columns can be added in parallel without relying on the result of the previous column, creating a two outputs "adder" with a time delay that is independent of the size of its inputs

Parameters

– *nbit* is the adder operand word length in bits

Exceptions

– Raise `SizeError` if `nbit` < 2

Validation

The validation is done from simulation with :
– a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
– different values for nbit0 and nbit1 : (8,16) and (16,8)
– all configurations of having or not cin, signed and extended

Reference

– [Dum01]

Member Function Documentation

```
def lib.fixed.adder.converter.BsCs.Pattern ( cls, param )
```

Parameters : *param* parameters used for BsCs instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.converter.BsCs.GetModelName ( cls, param )
```

Parameters : *param* parameters used for BsCs instanciation
Returns : Model name string

```
def lib.fixed.adder.converter.BsCs.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.4.2 lib.fixed.adder.converter.CsBs Class Reference

Stratus generator : converter
Construct a converter CS -> BS

Interface

– *i0_0, i0_1* : entry (input, carry-save)
– *o_0, o_1* : conversion of i0 (output, borrow-save)

Architecture

The architecture is the Mixed adder architecture :
– all of the columns can be added in parallel without relying on the result of the previous column, creating a two outputs "adder" with a time delay that is independent of the size of its inputs

Parameters

– *nbit* is the adder operand word length in bits

Exceptions

– Raise `SizeError` if `nbit` < 2

Validation

The validation is done from simulation with :
– a same width for nbit0 and nbit1 varying [2 :64] and a MaskNumber of 3
– different values for nbit0 and nbit1 : (8,16) and (16,8)
– all configurations of having or not cin, signed and extended

Reference

– [Dum01]

Member Function Documentation

```
def lib.fixed.adder.converter.CsBS.Pattern ( cls, param )
```

Parameters : *param* parameters used for CsBS instanciation
Returns : Pattern file for simulation

```
def lib.fixed.adder.converter.CsBS.GetModelName ( cls, param )
```

Parameters : *param* parameters used for CsBS instanciation
Returns : Model name string

```
def lib.fixed.adder.converter.CsBS.GetParam ( cls )
```

Returns : Dictionary of all the parameters with their types

E.5 Other

E.5.1 lib.fixed.other.bool.Bool Class Reference

Stratus generator : Bool
Construct a given boolean function between all of the input bits

Interface

– *i* : the entry (input)
– *o* : the corresponding logical value (output)

Architecture

The architecture is a binary tree of 2-bits logical gates corresponding to the given boolean function

Parameters

- *nbit* is the input word length in bits
- *op* is the boolean function to be used on the input : *'and'*, *'nand'*, *'or'*, *'nor'*, 'xor' or *'nxor'*

Exceptions

- Raise `SizeError` if `nbit` < 1
- Raise `KeyError` if `param`['op'] is not one of : *'and'*, *'nand'*, *'or'*, *'nor'*, 'xor' or *'nxor'*

Validation

The validation is done from simulation with :
- all the boolean functions
- a width varying in [1 :64] and a MaskNumber of 4

Member Function Documentation

```
def def lib.fixed.other.bool.Bool.Param ( cls,  param )
```

Parameters : *param* parameters used for Bool instanciation
Returns : Pattern file for simulation

```
def def lib.fixed.other.bool.Bool.GetModelName ( cls,  param )
```

Parameters : *param* parameters used for Bool instanciation
Returns : Model name string

E.5.2 lib.fixed.other.bool.Boolx Class Reference

Stratus generator : Boolx
Construct a given boolean function between all of the input bits and another one bit signal

Interface

- *i0* : the entry (input)
- *i1* : the one bit signal (input)
- *o* : the corresponding logical value (output)

Architecture

The architecture is a 2-bits logical gates corresponding to the given boolean function

Parameters

- *nbit* is the input word length in bits
- *op* is the boolean function to be used on the input

Exceptions

- Raise `SizeError` if `nbit` < 2
- Raise `KeyError` if `param`['op'] is not one of : and, nand, or, nor, xor or nxor

Validation

The validation is done from simulation with :
- all the boolean functions
- all the values for the one bit signal
- a width varying in [1 :64] and a MaskNumber of 4

Member Function Documentation

```
def def lib.fixed.other.bool.Boolx.Param ( cls, param )
```

Parameters : *param* parameters used for Boolx instanciation
Returns : Pattern file for simulation

E.5.3 lib.fixed.other.shifter.Shifter Class Reference

Stratus generator : Shifter
Construct a shifter operator

Interface

- *cmd* : the command of the shifter (input)
- *i* : the signal to shift (input)
- *o* : the obtained result (output)
- *sticky* : value of the sticky bit (optionnal output)
- *type* : the type of shift circular (1), arithmetical (2), logical (4) (optionnal input)
- *dir* : the direction of the shift left (0) or right (1) (optionnal input)

Architecture

The architecture is based on a standard 3 bit mux matrix for shifter

Parameters

- *nbit* is the shifter word length in bits
- *cmd_width* is the shifter command word length in bits (default log2(nbit))
- *type* corresponds to the kind of shift by setting corresponding bits as the following : sticky | log | arith | cir | left | right. For example : type = 0b001011 corresponds to an arithmetical left and right shifter without sticky bit

Exceptions

- Raise `SizeError` if `param`['nbit'] < 2

Validation

The validation is done from simulation :
- with a varying of nbit in [2 :64] and a MaskNumber of 4
- with all kind of shift
- with all width of cmd between 0 and log2(nbit)

Other simulations are done for saturation with a 8 bits shifter and a cmd width of 8 bits and all kind of shift

Reference

- Inspired from the DPGEN_SHIFT of Alliance [All]

Remarks

- When saturation (cmd $>$ 2^log2(nbit)) the value of the shift is 2^log2(nbit)-1

Member Function Documentation

```
def lib.fixed.other.shifter.Shifter.Pattern ( cls, param )
```

Parameters : *param* parameters used for Shifter instanciation
Returns : Pattern file for simulation

E.5.4 lib.fixed.other.coding.Encoder Class Reference

Stratus generator : Encoder
Construct an encoder operator, that gives the position of the only true bit in a signal

Interface

- *i* : the signal to encode (input)
- *o* : the resulting value (output)

Architecture

The architecture is based on the standard Sklansky parallel-prefix algorithm with OR gates instead classical PG

Parameters

– *nbit* is the input word length in bits

Exceptions

– Raise `SizeError` if `param`['nbit'] < 2

Validation

The validation is done from simulation :
– with a width varying in [1 :64] and all the 2^i values (i $<$ width)

Reference

– [Zim]

Member Function Documentation

```
def lib.fixed.other.coding.Encoder.Pattern ( cls, param )
```

Parameters : *param* parameters used for Encoder instanciation
Returns : Pattern file for simulation