الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne démocratique et populaire

وزارة التعليم العالي و البحث العلمي
Ministère de l'enseignement supérieur et de la recherche scientifique

جامعة سعد دحلب البليدة
Université SAAD DAHLAB de BLIDA

كلية التكنولوجيا
Faculté de Technologie

قسم الإلكترونيك
Département d'Électronique

Mémoire de projet de fin d'Etude

Présenté par :

Melle. Amimer Chahinez

&

Melle. Raouraoua Mounia

Pour l'obtention du diplôme master en télécommunication option : **système des télécommunications**

Thème

Automatisation des interconnexions variables-parité pour un émulateur de décodeurs LDPC sur FPGA

Proposé par : Mr. M.MAAMOUN

Année universitaire : 2019-2020

Remerciement

Avant tout, Nous tenons particulièrement à remercier Allah le tout puissant, et le créateur de nous avoir facilité le chemin et qui nous a donné durant toutes ces années la santé, le courage à la fois en nous même pour arriver à ce jour. Ce mémoire n'aurait jamais été réalisé sans sa bénédiction.

Nous tenons à remercier vivement mon promoteur Mr M. MAMOUNE pour ses conseils et ses suggestions qui nous ont permis de mener à bonne fin notre bon travail.

Un grand merci aux membres du jury, qui ont accepté d'évaluer notre travail de fin d'études.

Nous tenons aussi à remercier tout ce qui nous a apporté leur assistance et leur encouragement.

Dédicace

Je dédie ce modeste travail aux être qui me sont les plus chers, A mes parents les plus chers de monde, Papa et maman que dieu les garde et les protège pour moi et j'espère que vous êtes fière de moi, je suis là grâce à vous.

A mon deuxième père mon oncle « Meddah Salah », merci beaucoup pour tous de mes premiers pas jusqu'à maintenant.

A ma jolie sœur « Safaa »

A ma charmante « Chahinaz », très reconnaissante à notre amitié, que ses années de mes études sont passées merveilleux avec toi.

Mes chères amies, spécialement : Sara, Fella, et Asma .Avec eux j'ai partagé des moments inoubliables De bonheur et de chagrin qui resteront toujours gravés dans ma mémoire.

A la personne qui mérite mes profonds amours et respect, mon cher ami « Mohamed » qui a été toujours mon appui et mon aide.

A tous ceux que j'aime Merci.

MOUNIA

À celui qui s'est toujours tenu à mes côtés et a travaillé dur pour me plaire et me donner tout l'amour et la tendresse, mon cher père Mohammed, je prie Allah qu'il soit un d'Ahl eljennah.

À celle qui était la raison de ce que je suis aujourd'hui, la source de mon inspiration et de ma force, la femme idéale dans ma vie, ma chère mère Dalila, qu'Allah te protège.

À ma grande sœur Ahlem, son mari Ridha et leur fils, mon petit neveu, Adem.

À ma petite sœur khitem et mon frère mon poussin Youcef.

À ma chère grand-mère, ainsi qu'à tous mes oncles, tantes, cousins et cousines sur tous mes belles Sophie et Melissa.

Je tiens à exprimer mes sincères remerciements envers celle qui m'a partagé ce merveilleux voyage avec tous ses arrêts difficiles et ses beaux moments, l'adorable Mounia.

Je dédie ce modeste travail aussi à ma meilleure amie depuis de nombreuses années Touati Ahlem, à tous mes amis/amies : Hanane et Sarah.

Qu'Allah vous garde pour moi !

CHAHINAZ

الملخص:

الموضوع الرئيسي لأطروحتنا النهائية هو: أتمتة الترابط المتغير لمحاكي وحدة فك ترميز LDPC على FPGA. ركز البحث المقدم في هذه الرسالة على تحقيق RAMs على FPGAs من ملف Alist، درسنا ثم برمجنا تحت بيئة مولد نظام Matlab.

الكلمات المفتاحية:

مصفوفة تحقق التكافؤ،Alist، أتمتة على FPGA، أكواد LDPC، وحدة فك ترميز للمحاكي.

Résumé :

Le sujet principal de notre mémoire de fin d'étude est : Automatisation des interconnexions variables-parité pour un émulateur de décodeurs LDPC sur FPGA. Les travaux de recherche présentés dans ce mémoire ont porté sur la réalisation des RAM sur FPGA à partir d'un fichier Alist, nous avons étudié puis programmé sous l'environnement Matlab system generator.

.

Mots clés :

Matrice de contrôle de parité, Alist, Automatisation sur FPGA, codes LDPC, décodeur pour émulateur.

Abstract:

The main subject of our final thesis is: Automation of variable-parity interconnections for an LDPC decoder emulator on FPGA. The research presented in this thesis focused on the realization of RAMs on FPGAs from an Alist file, we studied and then programmed in the MATLAB system generator environment.

Key words:

parity check matrix, Alist, Automation on FPGA, LDPC codes, decoder for emulator.

Abréviations:

ASIC: Application Specific Integrated Circuit.

FPGA: Field programmable gate array.

HDL: Hardware Description Language.

VHDL: VHSIC Hardware Description Language.

CLB: Configurable Logic Blocks.

IOB: I / O blocks.

SRAM: Static Random-Access Memory.

ROM: Read-Only Memory.

RAM: Random Access Memory.

MOS: Metal Oxide Semiconductor

DSP: Digital Signal Processing.

LUT: Look-Up-Tables.

EBR: Embedded Block RAM.

DRAM: Dynamic Random Access Memory.

EPROM: Erasable Programmable Read-Only Memory.

BRAM: Block Random Access Memory.

FPU: Floating-Point Unit.

DPRAM: Dual Ported Random-Access Memory.

ADC: Analog-to-Digital Converter.

FIFO: First In First Out.

LDPC: Low-Density Parity-Check.

BF: Bit Flipping.

SPA: Sum-Product Algorithm.

DVBS2: Digital Video Broadcasting - Satellite 2.

WIMAX: Worldwide Interoperability for Microwave Access.

WIFI: Wireless Fidelity.

BER: Bit error ratio.

OFDM: Orthogonal Frequency-Division Multiplexing.

DVB: Digital Video Broadcasting.

WLAN: Wireless Local Area Network.

MLG: Majority Logic.

APP:A Posteriori Probability.

LLR: Log-Likelihood Ratios.

LLR-SPA: Log-Likelihood Ratios Sum-Product Algorithm.

MAP: Maximum A Posteriori Probability.

VNU : Variable Node Unit.

CNU : CheckNode Unit.

QC-LDPC: Quasi-Cyclic Low-Density Parity-Check.

SNR: Signal-to-Noise Ratio.

RS-LDPC: Reed–Solomon.

VHSIC: Very High-Speed Integrated Circuit.

PLD: Programmable Logic Device.

WRAN: Wireless Regional Area Networks.

LAN*:* Local Area Network.

MAN: Metropolitan Area Network.

Liste des figures :

Fig.1.1-Architecture FPGA retenue par Xilinx, la première est la couche appelée circuit configurable et la deuxième est une couche de réseau mémoire SRAM.

Fig.1.2 -Architecture interne d'un FPGA.

Fig.1.3-Interconnexion interne d'un FPGA.

Fig.1.4-Architecture d'un slice Virtex 4.

Fig.1.5-Architecture d'un slice Virtex 5.

Fig.1.6-Les tailles d'une RAM de blocs.

Fig.1.7-BRAM à port unique.

Fig.1.8- BRAM à deux ports.

Fig. 2.1-Représentation sous forme pseudo-triangulaire inférieure de la matrice H.

Fig. 2.2-(a) : Matrice de contrôle de parité H (b) : Graphe de Tanner correspondant.

Fig. 2.3- Architecture matérielle générale d'un décodeur LDPC.

Fig. 2.4-Illustration (a) d'une architecture de décodeur parallèle et (b) d'un décodeur série architecture.

Fig. 2.5-Flux de conception pour l'émulation matérielle.

Fig. 2.6-Une architecture canonique du (2048,1723) décodeur RS-LDPC composé de 32unités de traitement.

Fig. 2.7-Une architecture en couches du (2048,1723) décodeur RS-LDPC composé de 32unités de traitement.

Fig. 2.8- Matrice de contrôle de parité H et Graphe de Tanner correspondant.

Fig. 3.1-Conversion la forme Alist en VHDL BRAM.

Fig. 3.2 –Le code VHDL.

Fig. 3.3 -Le premier résultat obtenu d'un langage VHDL.

Fig. 3.4 -Le résultat suivant obtenu d'un langage VHDL.

Fig. 3.5-Concept architectural de base des FPGAs.

Fig. 3.6- Les points d'interconnexion.

Fig. 3.7- Les différentes méthodes d'interconnexion entre blocs logiques.

Fig. 4.1- Organigramme de notre projet

Fig. 4.2-Programmation de alist vers un fichier txt.

Fig. 4.3- Un schéma explicatif pour cette étape

Fig. 4.4- La programmation pour la conversion binaire

Fig. 4.5- Schéma explicatif pour l'étape de la conversion binaire

Fig. 4.6- Le code VHDL

Fig. 4.7- Les blocs de Xilinx pour Simulink

Fig. 4.8 -Etape de configuration de Matlab pour utiliser le system générator

Fig. 4.9- La fenêtre de configuration Matlab

Fig. 4.10- Xilinx System Generator - Constant Block Symbol

Fig. 4.11- Xilinx System Generator - Gateway In Block Symbol

Fig. 4.12- Xilinx System Generator - Gateway Out Block Symbol

Fig. 4.13 - Le bloc System Generator

Fig. 4.14 - Le bloc black box

Fig. 4.15 - Le bloc To file

Fig. 4.16- Le bloc counter limited

Fig. 4.17 - Le modèle réalisé

Fig. 4.18 -"Alist Matrix" de la norme WIFI (802.11)

Fig. 4.19 - Le code VHDL pour la norme WIFI (802.11)

Fig. 4.20 - Simulink system generator modèl pour la norme WIFI (802.11)

Fig. 4.21- La programmation Matlab

Fig. 4.22- Programme Matlab

Fig. 4.23- Programmation Matlab

Fig .4.24-"Alist Matrix" de la norme WIMAX (802.16)

Fig. 4.25-Le code VHDL pour la norme WIMAX (802.16)

Fig. 4.26-Simulink system generator modèle pour la norme WIMAX (802.16)

Fig. 4.27-Programme Matlab

Fig. 4.28-Programme Matlab

Fig. 4.29-Programme Matlab

Fig. 4.30-"Alist Matrix" de la norme WRAN(802.22)

Fig. 4.31-Le code VHDL pour la norme WRAN (802.22)

Fig. 4.32-Simulink system generator modèle pour la norme WRAN (802.22)

Fig. 4.33-Le programme Matlab correspond

Fig. 4.34-Le programme Matlab correspond

Fig. 4.35-Le programme Matlab pour la vérification des 2 fichiers

Fig. 4.36-"Alist Matrix" de la norme LAN /MAN (802.3an)

Fig. 4.37-Le code VHDL pour la norme LAN/MAN (802.3an)

Fig. 4.38- Simulink system generator modèle pour la norme LAN/MAN (802.3an)

Fig. 4.39- Programmation Matlab

Fig. 4.40- Programme Matlab

Fig. 4.41-Un programme Matlab pour la verification des 2 fichiers

Liste des tableaux :

Table.2.1-Principales caractéristiques des architectures de décodeur

Table des matières :

Introduction générale ……………………………………………1

Chapitre 1 : Calcul en virgule fixe sur FPGA

1.1. Introduction ………………………………………………………….3

1.2. Architectures des FPGAs ……………………………………………3

1.2.1. Introduction aux technologies FPGA ……………………………3

1.2.2. Architecture des FPGAs …………………………………………4

1.2.3. Architecture interne d'un FPGA ………………………………...6

1.3. Les DSPs ……………………………………………………………8

1.3.1. Types des DSPs …………………………………………………9

1.4. Les CLBs……………………………………………………………. 9

1.5. Les BRAMs…………………………………………………………11

1.6. Conclusion …………………………………………………………14

Chapitre 2 : Émulateurs des décodeurs LDPC sur FPGA

2.1. Introduction ………………………………………………………..15

2.2. Codage et décodage LDPC ………………………………………..15

2.2.1. Bref historique …………………………………………………15

2.2.2. Les Codes LDPC ………………………………………………16

2.2.2.1. Codes blocs linéaires ……………………………………...16

2.2.3. Les classes des codes LDPC …………………………………..17

2.2.4. Caractéristiques et avantages des codes LDPC ……………….19

2.2.5. Codage des codes LDPC ………………………………………19

2.2.5.1. Codage conventionnel basé sur l'élimination de Gauss-Jordan ……………………………………………………………....20

2.2.5.2. Codage par approximation triangulaire inférieure …………21

2.2.6. Décodage des codes LDPC ………………………………...22

2.2.6.1. Décodage MLG des codes LDPC …………………………..24

2.2.6.2. Décodage BF ……………………………………………….24

2.3. Algorithme SPA …………………………………………………25

2.4. Architectures des décodeurs pour émulateurs ……………………..27

2.4.1. Architectures de décodeur LDPC ……………………………27

2.4.2. Architectures de décodeur pour émulateur …………………..30

2.5. Interconnexions entre nœuds de variables et nœuds de parité ……..35

2.5.1. Graphe de Tanner ……………………………………………35

2.6. Conclusion ………………………………………………………37

Chapitre 3 : Automatisation des interconnexions pour émulateur sur FPGA

3.1. Introduction ……………………………………………………….38

3.2. Les matrices de contrôle de parité ………………………………...38

3.2.1. Définition ………………………………………………….38

3.2.2. Matrice G de génération de code…………………………..38

3.2.3. Choix de la matrice de contrôle de parité ……………………39

3.2.4. Exemples de matrice de contrôle de parité …………………..40

3.3. La forme ALIST ……………………………………………………42

3.4. Conversion ALIST en VHDL BRAM …………………………...43

3.4.1. Définition du language VHDL…………………………………43

3.4.2. Conversion ALIST en VHDL BRAM…………………………43

3.5. Architecture des interconnexions sur FPGA ……………………50

3.6. Conclusion ………………………………………………………52

Chapitre 4 : Simulations et résultats

4.1. Introduction ……………………………………………..53

4.2. Plateforme de test sur MATLAB System Generator …...55

4.3. Simulations et résultats …………………………………69

4.3.1. WIFI (802.11) ……………………………………69

4.3.2. WIMAX (802.16) ………………………………..74

4.3.3. WRAN (802.22) …………………………………79

4.3.4. LAN/MAN (802.3an) ……………………………84

4.4. Conclusion ……………………………………………...89

Conclusion générale ……………………………………90

Introduction générale

Durant ces dernières années l'évolution de l'industrie a connu une évolution gigante que grâce à la concurrence entre les fabricants.

Actuellement, la logique programmable est la plus utilisés par les microcontrôleurs (µc) et microprocesseurs(µp) mais l'inconvénient de ces circuit et c'est qu'on peut seulement les programmer selon le programme existant dans une mémoire, l'architecture interne est celle proposé par le fabriquant, tout comme les entrées/sorties, et aussi sans oublie le nombre de ces circuit nécessaire qui peut être important, ce qui avait pour conséquence un prix important, une mise en œuvre complexe et un circuit imprimé de taille.

Pour diminuer le cout de fabrication, de développement et de maintenance Les PLD ont subi une évolution technologique au fil du temps depuis l'apparition de premier PAL (programmable array logique réseau logique programmable) jusque à l'aboutissement des premiers circuits intégrés reconfigurables de type FPGA (Field Programmable Gate Arrayou réseau de cellules logiques programmables) par la société XILINX en 1985.

Ces circuits FPGAs sont capables de réaliser plusieurs fonctions logiques complexes avec la souplesse et la flexibilité apportée par la logique programmable est ça dans un seul circuit, et ils sont les circuits logiques programmables les plus performants qui existent en ce moment.

Auparavant pour décrire le fonctionnement d'un circuit électronique programmable les techniciens et les ingénieurs utilisaient des langages de difficile (ABEL, PALASM, ORCAD/PLD,) Ou plus simplement un outil de saisie de schémas

Les sociétés de développement ainsi que les ingénieurs ont voulu s'affranchir des contraintes technologiques des circuits PLD, en créant des langages plus faciles qui sont VHDL et VERILOG.

Ces langages permettent au code écrit d'être portable, de façon qu'une description écrite pour un circuit puisse être facilement utilisée pour un autre circuit. Ceci permet de matérialiser les structures électroniques d'un circuit. En effet les instructions écrites dans ces langages se

traduisent par une configuration logique de portes et de bascules qui est intégrée à l'intérieur des circuits PLDs.

Les codes Low Density Check LDPC ont été proposé par Gallager en 1963, Ils sont utilisés dans de nombreuses normes de communication à grande vitesse telles que la diffusion des vidéos numériques par voie satellitaire DVB-S2 (Digital Video Broadcasting Satellite-Second Generation), WiMax (Worldwide Interoperability for Microwave Access), et les systèmes sans fil 4G et 5G. **[14]**

David Mackay, Matthew Davey et John Lafferty à partir des codes LDPC ont tous écrit des matrices de contrôle de parité à faible densité dans un format appelé alist.

Le but de notre projet de fin d'étude est de d'écrire un fichier au format Alist sur des RAMs sur FPGA.

Ce mémoire est constitué de quatre chapitres :

Le premier chapitre est introductif nous avons présenté brièvement l'un des circuits logiques programmables FPGAs et ses architectures.

Le second chapitre aborde quelques concepts généraux concernant les codes LDPC, les principes de base du codage et décodage, dans un second lieu nous avons défini les architectures des décodeurs pour émulateurs. Finalement nous avons présenté les interconnections entre nœuds de variables et nœuds de parité.

Le troisième chapitre nous avons incorporé dans un premier lieu les matrices de contrôle de parité. En deuxième lieu le format Alist ensuite nous avons expliqué sa conversion vers VHDL BRAM. Finalement l'architecture des interconnexions sur FPGAs.

Le dernier chapitre a débuté par une introduction. A la fin de ce manuscrit, une conclusion générale récapitule les apports essentiels de notre travail.

CHAPITRE 1

Calculs en virgule fixe

1.1. Introduction

L'implantation d'applications de traitement numérique du signal dans un système
Embarqué requiert l'usage de l'arithmétique virgule fixe et de minimiser le nombre de bits pour représenter les données. Cette action permet de réduire la surface et la consommation mais entraine une perte de précision des calculs ; il est donc nécessaire de spécifier une contrainte de précision au niveau applicatif. Contrairement aux processeurs, les ASIC et les FPGA permettent une liberté totale sur les choix du nombre des unités arithmétiques et du nombre de bits pour les entrées/sorties de chacune. Ceci constitue un potentiel important pour l'optimisation d'architectures. Ce chapitre introduit les principales caractéristiques des FPGAs et des architectures de système à base de FPGA.

1.2. Architecture des FPGAs

1.2.1. Introduction aux technologies FPGA

Les FPGAs (Field programmable gate array) sont inventés par la société Xilinx en 1985. Xilinx fut précurseur du domaine en lançant le premier circuit FPGA commercial, le XC2000. Ce composant avait une capacité maximum de 1500 portes logiques. La technologie utilisée était alors une technologie aluminium à 2 micromètres avec 2 niveaux de métallisation. Xilinx ne sera suivi qu'un peu plus tard, et jamais lâchée, par son plus sérieux concurrent Altera qui lança en 1992 la famille de FPGA FLEX 8000 dont la capacité maximum atteignait 15 000 portes logiques. Les FPGAs se situent entre les réseaux logiques programmables et les circuits logiques près diffusés. Les réseaux logiques programmables sont des composants qui ne nécessitent aucune étape technologique supplémentaire pour être personnalisés, ce sont des circuits standards, programmables par l'utilisateur grâce aux différents outils de développement et qui incluent un grand nombre de solutions basées sur les variantes de l'architecture des portes ET et OU. Les prés diffusés sont des circuits intégrés basés sur l'utilisation des réseaux de cellules dont les blocs ont été préalablement diffusés, il faut créer les connexions entre ces blocs. Les FPGA combinent donc à la fois la souplesse de la programmation des réseaux logiques programmables et les performances des circuits prés

diffuses. En d'autres termes le FPGA permet d'avoir une architecture conçue sur mesure à haute densité dans un circuit électronique, avec la possibilité de modifier cette architecture

quand des nouvelles applications apparaissent. Les langages HDL comme le VHDL ou le Verilog sont utilisés pour décrire les fonctionnalités qui seront implémentées sur le composant, puis la description matérielle est traduite dans un fichier de configuration pour le FPGA cible. La description matérielle peut être aussi utilisée pour la description d'un composant ASIC. En 2000 et 2001, les deux concurrents Xilinx et Altera ont franchi une nouvelle étape au niveau de la densité d'intégration en proposant respectivement leurs circuits Virtex et Apex-IIdont les capacités maximums avoisinaient les 4 millions de portes logiques équivalentes avec de plus l'introduction de larges bancs de mémoires embarquées. Aujourd'hui, les fréquences de fonctionnement de ces circuits sont de l'ordre de quelques centaines de Méga Hertz (ces dernières sont en réalité très dépendantes de l'application). Bien que ces valeurs soient relativement réduites par rapport aux ASICs, elles sont suffisantes pour une très large majorité d'applications actuelles. À partir des années 2000, les capacités des FPGA ont permis d'offrir aux concepteurs une solution supplémentaire de réalisation pour une majorité d'applications. De plus, les outils de mise en œuvre des FPGA ont évolué et bien qu'encore pénalisants lors de la conception, ils permettent la réalisation rapide d'applications complexes. La flexibilité du FPGA permet de changer sa description matérielle en fonction de l'application, c'est le facteur principal qui a déterminé la popularité des composants FPGA. De plus, les FPGAs peuvent être couplés avec un processeur à usage général. Ainsi la section la plus exigeante du logiciel peut être traduite sur la partie matérielle qui permet d'accélérer l'exécution du programme. Ce qui donne des accélérations notables dans l'exécution, en particulier lorsque le programme exécuté en série sur un processeur peut être traduire en matériel pour exploiter le parallélisme de l'algorithme. Par conséquent, les FPGA permettent d'obtenir des performances de calcul importantes. De plus, le nombre de portes logiques sur les FPGA les plus récents permet l'implémentation complète d'un système sur puce (SoC - System On Chip). **[1]**

1.2.2. Architecture des FPGAs

Cette partie présente les caractéristiques des FPGA, en particulier pour clarifier la façon dont ils peuvent mettre en œuvre la personnalisation du matériel via leur reconfiguration. Bien qu'il existe actuellement plusieurs fabricants de circuits FPGA, dont Xilinx [Xila] et Altera [Alt] qui sont les plus connus. L'architecture, retenue par Xilinx (figure 1.1), se présente sous

la forme de deux couches distinctes, la première est la couche appelée circuit configurable et la deuxième est une couche de réseau mémoire SRAM.

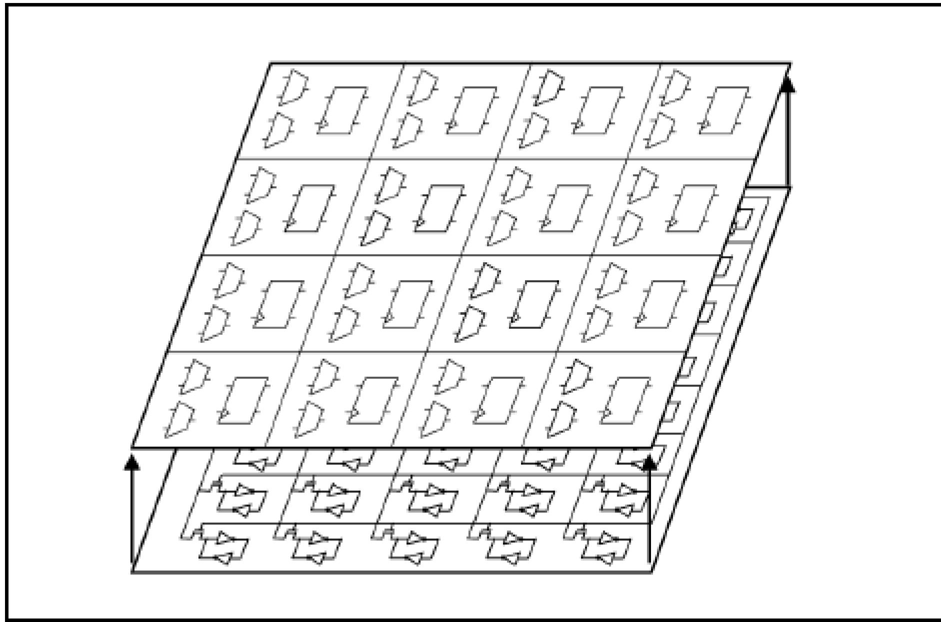


Fig.1.1-Architecture FPGA retenue par Xilinx, la première est la couche appelée circuit configurable et la deuxième est une couche de réseau mémoire SRAM. [1]

La première couche (circuit configurable) est constituée d'une matrice de blocs logiques configurables (CLB - Configurable Logic Bloc). Les CLB permettent de réaliser des fonctions séquentielles et combinatoires. Autour de ces blocs logiques configurables, nous trouvons les blocs entrées/sorties (IOB - Input Output Bloc)). Ils permettent de gérer les entrées-sorties pour réaliser l'interface avec les modules extérieurs.

La seconde couche est un réseau de mémoire SRAM qui permet la programmation du circuit FPGA. La programmation est réalisée en appliquant les potentiels adéquats sur la grille de certains transistors à effet de champ pour interconnecter les éléments des CLB et des IOB afin

de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont mémorisés dans le réseau de mémoire SRAM. Un dispositif interne au FPGA permet à chaque mise sous tension de charger les SRAM internes à partir de la ROM externe où est stockée la configuration du FPGA. **[1]**

1.2.3. Architecture interne d'un FPGA

Les circuits FPGA du fabricant Xilinx utilisent deux types de cellules de base, les cellules d'entrées/sorties appelées IOB et les cellules logiques appelées CLB. Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable. Donc les trois blocs principaux (Fig1.2) sont les blocs logiques configurables, les blocs d'entrées/sorties et les ressources de communications.

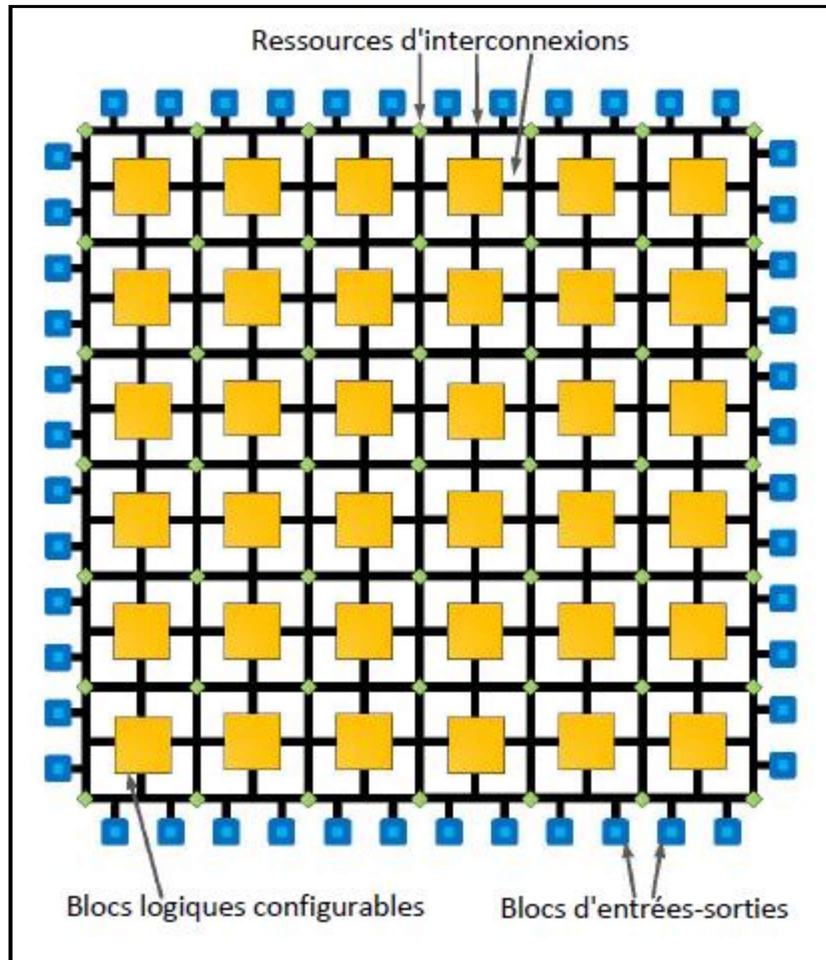


Fig.1.2 -Architecture interne d'un FPGA. [1]

✓ Les blocs d'entrée-sortie (IOB) :

Les blocs d'entrée-sortie permettent l'interconnexion de la logique interne aux ports d'entrées et de sorties du FPGA. Les IOB ont leur propre mémoire de configuration, elle stocke les standards de tension et la direction des ports. Ces blocs sont présents sur toute la périphérie du

circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé (haute impédance).

✓ Les ressources d'interconnexions :

Les ressources d'interconnexion au sein d'un FPGA permettent la connexion arbitraire des CLB et des IOB. Les connexions internes dans les circuits FPGA sont composées de segments métallisés. Parallèlement à ces lignes, nous trouvons des matrices programmables réparties sur la totalité du circuit, horizontalement et verticalement entre les divers CLB. Elles permettent les connexions entre les diverses lignes, celles-ci sont assurées par des transistors MOS dont l'état est contrôlé par des cellules de mémoire vive ou RAM. Le rôle de ces interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les entrées/sorties afin que le taux d'utilisation dans un circuit donné soit le plus élevé possible. Pour parvenir à cet objectif, Xilinx propose trois sortes d'interconnexions selon la longueur et la destination des liaisons comme la montre la figure (1.3).

Fig. 1.3 -Interconnexion interne d'un FPGA. [3]

Les trois blocs présentés jusqu'ici sont interconnectés ensemble dans le dispositif pour créer une infrastructure de communication composée d'un réseau de communication et d'IOBs autour des CLBs. Des cellules de mémoire liées à chaque bloc détiennent les caractéristiques principales, de telle sorte que les interconnexions entre l'infrastructure de communication, les normes de tension des entrées-sorties d'un IOB, et les équations soient commandées par des valeurs particulières stockées dans une mémoire. Toutes ces configurations sont stockées dans des SRAM qui sont volatiles : lorsque le composant est mis sous tension, toute sa configuration est perdue et il doit être redémarré avec une nouvelle configuration. Habituellement, une machine externe se charge de télécharger la configuration sur le FPGA via une de ses interfaces de configuration, et envoie une commande de démarrage pour signaler que la configuration a eu lieu. Certaines cartes sont une mémoire ROM d'intégrée. Elle permet de stocker la configuration, de sorte qu'elle puisse ensuite être téléchargée sur le FPGA. Dans ce cas, les données de configuration sont copiées sur la mémoire SRAM de configuration du FPGA au démarrage de celui-ci. **[1]**

1.3. Les DSPs

Les DSPs « Digital Signal Processing » sont des blocs qui permettent des conceptions plus complexes, qui peuvent consister soit en traitement numérique du signal ou seulement certains assortiments de multiplication, addition et soustraction. Comme pour les BRAM, il est possible de mettre en œuvre ces blocs grâce au CLB, mais il est plus efficace en termes de performances, et de consommation d'énergie d'intégrer plusieurs de ces composants au sein du FPGA. Un bloc DSP permet de réaliser un multiplicateur, un accumulateur, un additionneur, et des opérations logiques (AND, OR, NOT, et NAND) sur un bit. Il est possible de combiner les blocs DSP pour effectuer des opérations plus importantes, telles que l'addition avec virgule flottante simple, la soustraction, la multiplication, la division, et la racine carrée. Le nombre de blocs DSP est dépendant du dispositif.

Les DSPs ont révolutionné les systèmes électroniques embarqués et cela grâce à leur architecture particulière et à leurs périphériques intégrés qui leur procurent puissance et rapidité. **[1]**

1.3.1. Types des DSPs

- Les DSPs à virgule fixe :

On retrouve dans la plupart des applications où le coût est un facteur important (ils sont moins chers que les DSP à virgule flottante), par contre ils sont plus compliqués à programmer.

- Les DSPs à virgule flottante :

Ils sont plus faciles et plus souples à programmer que les précédents. Dans ce cas on fait intervenir une mantisse et un exposant et on bénéficie d'une dynamique plus importante. **[W1]**

1.4. Les CLBs

Les blocs logiques configurables sont les principaux éléments d'un FPGA. Ils peuvent avoir un ou plusieurs générateurs de fonctions réalisées avec des tables de correspondance LUT (look-up-tables) qui peuvent mettre en œuvre une logique arbitraire en fonction de leur configuration. Autour d'une LUT, il y a une logique d'interconnexion qui permet les liaisons à destination et vers la LUT. Elle est mise en œuvre à l'aide de portes logiques et de multiplexeurs. Pendant le processus de configuration d'un FPGA, les mémoires des LUT sont écrites pour y implémenter une fonction, et la logique qui l'entoure est configurée pour router correctement les signaux afin de construire un système complexe. Il existe différents types de CLB en fonction du FPGA utilisé. Pour Xilinx le bloc logique configurable FPGA est appelé slice. L'architecture du slice (avec des modifications mineures) est utilisée dans toutes les familles FPGA Virtex et toutes les familles de FPGA Spartan. Un slice Virtex-4 (figure 1.4) est composé de deux LUT4à 4 entrées (Look-Up Tables). Ces LUT4 peuvent mettre en œuvre une fonction booléenne (une LUT est marquée "F", l'autre est marqué "G"). Il y aussi deux multiplexeurs contrôlés par l'utilisateur (MUXF5 et MUXFX). MUXF5 peut être utilisé pour combiner les sorties des LUT4.MUXFX est utilisé pour combiner les sorties de MUXF5 et MUXFX à partir d'un autre slice. Le bloc arithmétique et logique est composé de deux additionneurs sur 1 bit, d'une chaine de retenue et deux portes ET dédiées à la multiplication. Pour finir, nous avons deux registres sur1 bit. L'entrée de ces registres est sélectionnée par les multiplexeurs YMUX et XMUX. Notez que ces multiplexeurs ne sont pas contrôlés par

l'utilisateur le chemin est sélectionné lors de la configuration du FPGA. Le schéma simplifié d'un slice Virtex-4 est présenté ci-dessous. Les slices pour les Virtex 5 et 6 sont légèrement différentes (figure 1.5), mais leurs principes de fonctionnement restent similaires au slice du Virtex 4. Il est composé de 4 LUT et 4 registres par slice. Les LUT peuvent être configurés en

mode 6 entrées avec 1 bit de sortie ou 5 entrées avec 2 bits de sortie ou encore 4 entrées avec 1 bit de sortie. **[1]**

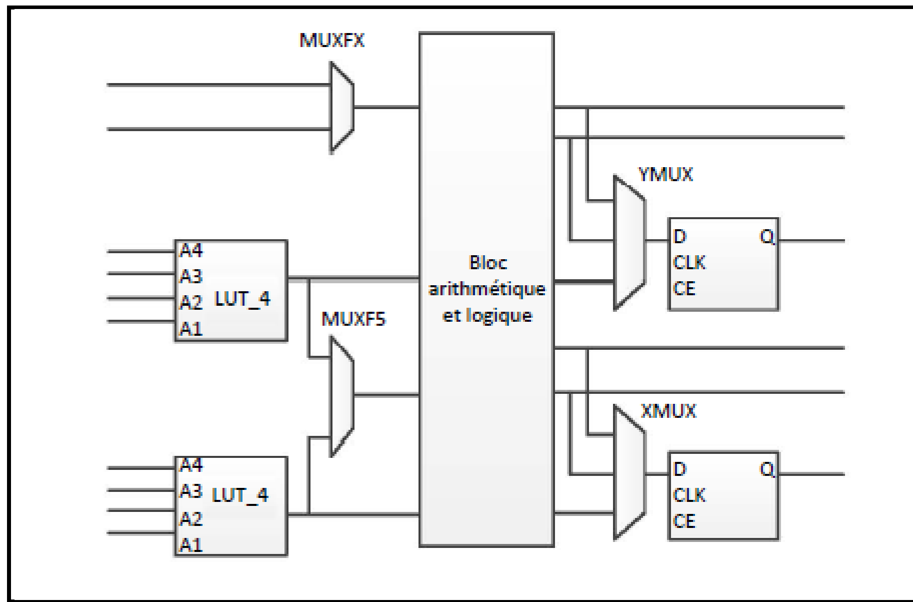


Fig. 1.4 -Architecture d'un slice Virtex 4. [1]

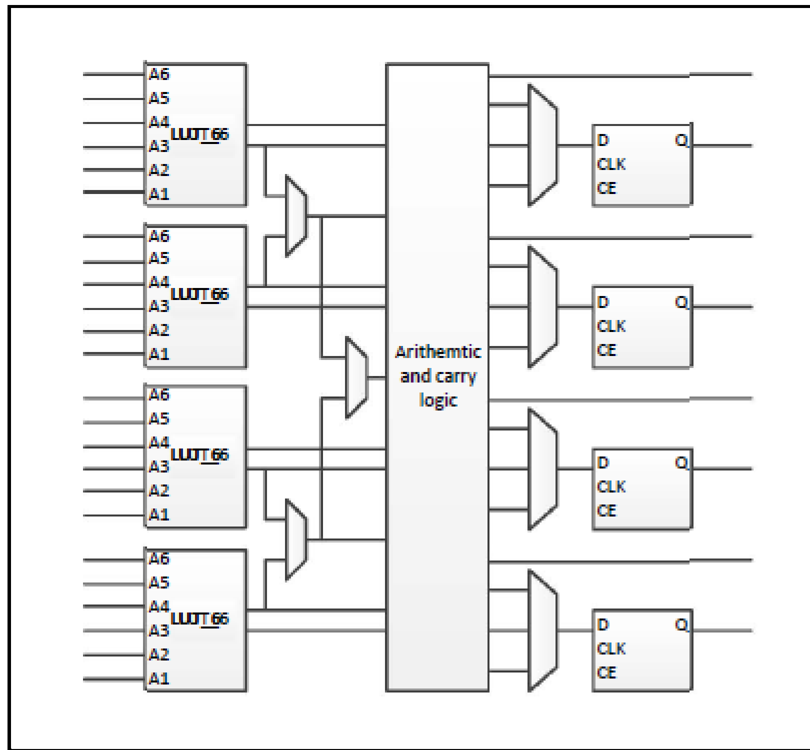


Fig. 1.5 -Architecture d'un slice Virtex 5. [1]

1.5. Les BRAMs

Block RAMs (ou BRAM) signifie Block Random Access Memory. Les blocs RAM sont utilisés pour stocker de grandes quantités de données à l'intérieur de votre FPGA. Ils sont l'un des quatre composants couramment identifiés sur une fiche technique FPGA. Les trois autres sont des bascules, des tables de consultation (LUT) et des processeurs de signaux numériques (DSP). Habituellement, plus le FPGA est gros et cher, plus il y aura de RAM de bloc. Étant donné que cela se trouve tout en haut d'un aperçu des produits FPGA

Une RAM de bloc (parfois appelée mémoire intégrée, ou RAM de bloc intégrée (EBR)), est une partie discrète d'un FPGA, ce qui signifie qu'il n'y en a que beaucoup disponibles sur la puce. Chaque FPGA a une quantité différente, donc en fonction de votre application, vous aurez peut-être besoin de plus ou moins de Block RAM. Savoir de combien vous aurez besoin devient plus facile à mesure que vous devenez un meilleur concepteur numérique. Comme je l'ai déjà dit, il est utilisé pour stocker de "grandes" quantités de données à l'intérieur de votre

FPGA. Il est également possible de stocker des données en dehors de votre FPGA, mais cela se ferait avec un appareil comme une SRAM, une DRAM, une EPROM, une carte SD, etc.
Les RAM de blocs sont de taille finie, 4/8/16/32 kb (kilobits) sont courants. Ils ont une largeur et une profondeur personnalisables. **[W2]**

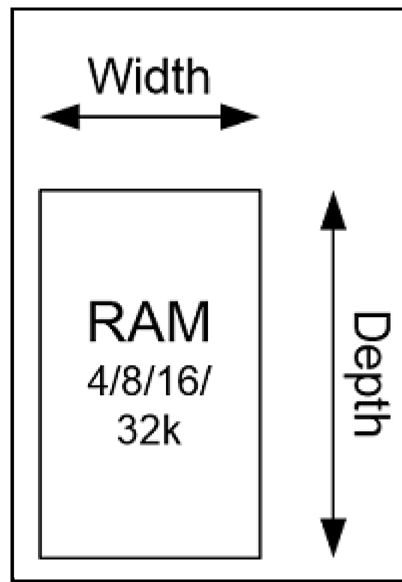


Fig. 1.6 -Les tailles d'une RAM de blocs. [W2]

- ## Configuration BRAM à port unique :

La configuration de la RAM à bloc de port unique est utile lorsqu'il n'y a qu'une seule interface qui doit récupérer des données. Il s'agit également de la configuration la plus simple et elle est utile pour certaines applications. Un exemple serait de stocker des données en lecture seule qui sont écrites sur une valeur fixe lorsque le FPGA est programmé. C'est une chose à propos de Block RAM, c'est qu'ils peuvent tous être initialisés (à l'exception des micros FPGA, qui ne peuvent pas être initialisés car ils sont d'une architecture différente.

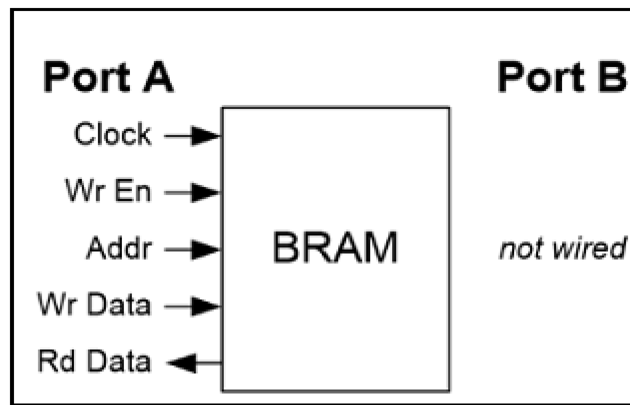


Fig. 1.7-BRAM a port unique. [W2]

- **Configuration BRAM à deux ports :**

La configuration RAM à deux ports (ou DPRAM) se comporte exactement de la même manière que la configuration à port unique, sauf que vous avez un autre port disponible pour lire et écrire des données. Le port A et le port B se comportent exactement de la même manière. Le port A peut effectuer une lecture sur l'adresse 0 sur le même cycle d'horloge que le port B écrit à l'adresse 200. Par conséquent, une DPRAM est capable d'effectuer une écriture sur une adresse tout en lisant à partir d'une adresse complètement différente. Personnellement, je trouve que j'ai plus de cas d'utilisation pour les DPRAM que pour les RAM à port unique.

Un cas d'utilisation possible serait de stocker des données hors d'un périphérique externe. Par exemple, vous voulez lire des données sur une carte SD, vous pouvez les stocker dans une RAM à double port, puis les lire plus tard. Ou peut-être voulez-vous vous connecter à un convertisseur analogique-numérique (ADC) et aurez besoin d'un endroit pour stocker les valeurs converties de l'ADC. Un DPRAM serait idéal pour cela. De plus, les RAM à double port sont généralement transformées en FIFO, qui sont probablement l'un des cas d'utilisation les plus courants pour la RAM de bloc sur un FPGA. **[W2]**

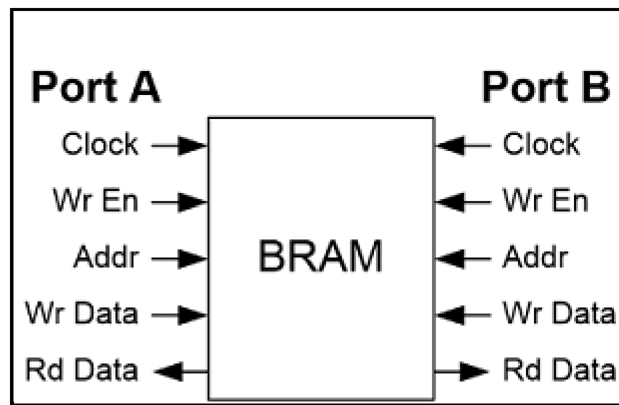


Fig .1.8 -BRAM a deux ports. [W2]

1.6. Conclusion

Pourquoi on fait du calcul en virgule fixe est pas du calcul en virgule flottante ?

- Le calcule en virgule flottante n'est pas possible, pas de FPU ! FPGA trop petit pour instancier un processeur complet avec FPU (plus vrai de nos jours)
- Pas de FPU sur processeur / microcontrôleur / DSP.
- Le calcul en virgule flottante coûte trop cher.
- Prend trop de place sur FPGA.
- Prend trop de temps sur processeur (latence des instructions).

Et donc l'implantation des algorithmes de traitement numérique du signal nécessite l'utilisation de l'algorithme virgule fixe. En effet, les opérateurs en virgule fixe ne travaillent que sur des entiers. En conséquence, la largeur des opérateurs arithmétiques, des bus et des mémoires est plus faible dans les architectures virgule fixe, ce qui permet de diminuer le coût en surface, le temps d'exécution des opérations et la consommation d'énergie.

CHAPITRE 2

Emulateur d'un décodeur LDPC sur FPGA

2.1. Introduction

Au travers de ce chapitre, nous avons pu identifier l'évolution des codes LDPC, tous d'abord nous avons décrit les notions de base de ces derniers, quelques méthodes de codage et de décodage. Concernant le décodage des codes LDPC, on a présenté deux algorithmes : le décodage de basculement BF et le décodage SPA. Ensuite, nous abordons plusieurs architectures matérielles de décodeur LDPC, en particulier de décodeur pour émulateur. Enfin On a présenté le graphe de Tanner, qui est une représentation utile des codes LDPC et ses interconnections entre nœuds de variables et nœud de contrôles.

2.2. Codage et décodage LDPC

2.2.1. Bref historique

Les codes LDPC composent une classe des codes en bloc qui sont caractérisés par une matrice de contrôle de parité H creuse. Ils ont été décrits pour la première fois dans la thèse de Gallager au début des années 60. Ce travail a été délaissé pendant 30 années. Seulement quelques études rares se sont rapportées au cours de cette période. En particulier, Tanner a proposé une généralisation des codes de Gallager et représentations en graphe bipartite. En 1993, l'algorithme de décodage itératif des turbo-codes a été inventé par Berrou, Glavieux et Thitimajshima. Les performances remarquables obtenues par les turbo-codes ont suscité plusieurs questions et beaucoup d'intérêt vers les techniques itératifs. Au milieu des années 90, la redécouverte des codes de Gallager a été par Mackay et autres, Wilberg et Sipser et autres. Par conséquent, les codes LDPC sont à la confluence de deux grandes révolutions dans la communauté du codage de canal : la représentation du Tanner basée sur les graphes et les techniques de décodage itératif. La communauté scientifique a été motivée à suivre les travaux sur les codes LDPC grâce à la simplicité de l'analyse théorique de ces codes. En 2004, et pour la première fois, un code LDPC a été normalisé par le satellite DVBS2 et après ces codes ont été utilisés par les standards IEEE 802.16e (WiMax Mobile) et IEEE 802.11n(WIFI). **[2]**

2.2.2. Les Codes LDPC

Les codes LDPC (Low-Density Parity-Check) sont des codes en bloc linéaire qu'on peut noter (n, k) ou bien (n, dc, dr), où n est la longueur du mot de code ; k la longueur du mot d'information ; dc le poids de la colonne (c'est à dire les nombres des éléments non nuls dans une colonne de la matrice de parité) et dr Le poids de la ligne (c'est-à-dire le nombre des éléments non nuls dans une ligne de la matrice de contrôle de parité). Les codes LDPC possèdent deux caractéristiques fondamentales :

- Parity-Check : Les codes LDPC sont représentés par une matrice de contrôle de parité H binaire.
- Low- Density : la matrice de contrôle de parité H est une matrice creuse c'est-à-dire de faible densité (La faible densité signifie qu'il y a plus de « 0 » que de « 1 » dans la matrice H). **[3]**

  L'avantage principal des codes LDPC est d'être capables d'atteindre des performances très proches de la capacité limite de Shannon, pour une grande diversité de canaux, avec des algorithmes à temps de décodage linéaire (mais complexes toutefois). Leur structure est adaptée aux traitements parallèles intensifs que les utilisations temps-réel imposent. **[4]**

2.2.2.1. Codes blocs linéaires

Dans un code C (n, k) en bloc les messages transmis sont groupés en bloc (mots code) de longueur n. Les symboles de l'information (symboles émis par la source) et les mots code sont des alphabets q -aires. Si q = 2 on parle des codes en blocs binaires alors que si q > 2 on parle des codes en blocs non binaires.

Dans le cas d'un code en bloc binaire, un mot code composé de n symboles est considéré comme un vecteur dont les composantes appartiennent à l'alphabet binaire du corps fini {0 ,1}.

Les mots de longueur n sont des éléments de $\{0,1\}^n$ qui sont écrits sous la forme de vecteurs lignes. Chacun des 2^K messages différents possibles correspond de manière unique à un des mots code.

Le code en bloc C (n, k) sera linéaire si les 2^K mots code possibles forment un sous espace vectoriel de dimension K de l'espace vectoriel $GF(2)^n$ ou GF (2) est le corps de Galois dont les composantes sont des éléments binaires. Par conséquent, la somme de deux mots code et le produit d'un élément de GF (2) par un mot code sont aussi des mots code. On appelle distance de Hamming le nombre de symboles différents entre deux mots code. La distance minimale d'un code C (n, k), notée d_{min} est alors la distance de Hamming minimale entre deux mots code quelconques. Le nombre n-k, appelé le nombre de symboles de redondance, est le nombre de symboles de longueur k ajoutés à l'information pour générer le mot code. Le code linéaire en bloc sera alors défini par trois paramètres :

- La longueur du mot n
- La longueur de l'information k
- La distance minimale $dmin$

Le code ainsi défini est capable de corriger $e = Ent\left(\frac{d_{min}-1}{2}\right)$ erreurs où Ent est la partie entière. Le rapport R $= \frac{k}{n}$ est appelé taux ou rendement du code. Si le nombre de bits de la redondance appliquée à une information croit, le rendement diminue. Plus il est proche de zéro, meilleure est la protection contre les erreurs de transmission. Cependant, le temps de transmission est important car il faut transmettre n bits dans une durée nT_b alors que nous n'avons besoin à la fin, de décodage, que des k bits d'informations qui nécessitent seulement une durée de kT_b (T_b: temps que va durer un bit, son unité est la seconde). **[5]**

2.2.3. Les classes des codes LDPC

La classe des codes LDPC engendre un très grand nombre de codes. Il est commode de les distinguer en deux classes :

- Les codes LDPC réguliers.
- Les codes LDPC irréguliers.

Un code LDPC est dit régulier, lorsque la matrice de contrôle H contient un nombre constant de '1' dans chaque ligne et un nombre constant de '1'' dans chaque colonne. Ainsi, chaque bit du mot de code contribue à un même nombre d'équations de parité et chacune des équations de parité utilise le même nombre de bits. Ce code est noté code LDPC (n, d_c, d_r).

Exemple :

Soit un code LDPC (8, 2, 2)

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

C'est un code LDPC régulier.

Un code LDPC est dit irrégulier si le nombre de '1' par ligne et le nombre de '1' par colonne varie et le principe d'irrégularité des variables d'un code LDPC irrégulier est défini par le polynôme suivant :

$\lambda(x) = \sum_{i\geq 1}^{w_{c_{max}}} \tilde{\lambda}_i \, x^{i-1}$.....(2.1)

Le coefficient λ_i est égal au rapport entre le nombre cumulé de '1' par colonne de poids i et le nombre total de '1' de la matrice H, par exemple :

$\lambda(x) = 0.2x^4 + 0.8x^3$ …... (2.2)

Cela est équivalent un code où 20% de ‘1’ sont associés à des nœuds de variables de degré 5 et 80% de ‘1’ sont associés à des nœuds variables de degré 4.

Autrement dit, le profil d’irrégularité peut être représenté par le polynôme suivant :

$\rho(x) = \sum_{j=2}^{W_{rmax}} \tilde{\rho}_j \, x_{j-1}$ ….. (2.3)

Le coefficient ρ_j est égal au rapport entre le nombre cumulé de ‘1’ des lignes de poids j et le nombre total de ‘1’ de la matrice H.**[2]**

Les codes LDPC réguliers sont plus faciles à réduire la complexité dans les implémentations matérielles (hardware) et logicielles (software). D'autre part, les codes LDPC irréguliers ont une meilleure performance que les réguliers. **[6]**

2.2.4. Caractéristiques et avantages des codes LDPC

Les codes LDPC forment une famille de codes linéaires obtenus à partir de matrices de parité creuses (Ils sont caractérisés par le faible nombre de « 1 » (densité de « uns ») de leurs matrices de contrôle de parité). Bien que leur définition ne l'impose nullement, les codes LDPC utilisés (et les seuls que nous considérerons pour le moment) sont binaires. Les codes LDPC ont par définition une matrice de parité creuse (le poids de chaque ligne vaut quelques unités, alors que la longueur peut atteindre plusieurs milliers). La performance de l'algorithme de décodage tient essentiellement à la faible densité de la matrice.

Les codes LDPC sont en grande compétition avec les codes turbo dans les systèmes de communications numériques qui demandent une fiabilité élevée. Aussi, les codes LDPC ont quelques avantages par rapport aux codes turbo :

- Ils ne nécessitent pas d'entrelacer pour réaliser une bonne performance d’erreur.
- Ils ont une meilleure performance par trame.
- Leur plancher d'erreur se produit à un niveau de BER de beaucoup inférieur.

- (d) leur décodage n'est pas basé sur un treillis et peut être réalisé par un processus parallèle.

Pour des grands taux et longueurs, les codes LDPC ont des meilleures performances que les codes turbo. Ils sont appliqués aux systèmes OFDM (Orthogonal Frequency Division Multiplexing) et aux systèmes de codage espace-temps. Récemment, ils ont été sélectionnés pour la norme de transmission vidéo numérique (DVB) et pour des applications en temps réel comme le stockage magnétique, l'Ethernet à 10 GB et les réseaux locaux sans fil avec débit élevé (WLAN). **[7]**

2.2.5. Codage des codes LDPC

Quelles que soient leurs nombreux avantages, le codage des codes LDPC peut constituer un obstacle pour les applications commerciales, car ils ont une grande complexité de codage et du retard de codage. Le codage pour les codes LDPC comprend essentiellement deux tâches :

- Construire une matrice de contrôle de parité creuse (sparce).
- Générer les mots de code à l'aide de cette matrice. **[8]**

2.2.5.1. Codage conventionnel basé sur l'élimination de Gauss-Jordan

L'algorithme de codage classique est basé sur l'élimination de Gauss-Jordan et la réorganisation des colonnes pour calculer le mot de code.

Semblable à la méthode générale de codage des codes en blocs linéaires, Neal a proposé un schéma simple [30]. Pour un mot de code Cdonné et une matrice de contrôle de parité Hirrégulière de taille $(m \times n)$, on partitionne le mot de code C en bits de message x ,et des bits de contrôle de parité p.

$C = [x/p]$ …… (2.4)

Après l'élimination de Gauss-Jordan, la matrice de contrôle de parité Hest convertie en forme systématique et ensuite divisé en une matrice A de taille $m \times (n-m)$sur la gauche et une matrice Bde taille $m \times m$ sur la droite.

$H = [A/B]$ …… (2.5)

Pour détecter les erreurs, on utilise le fait que tout mot de code valide doit obéir à la condition **:**

$C \times H^t = 0$……. (2.6)

De la condition précédente, nous avons :

$A.x^T + B\,p^T = 0$ …… (2.7)

Par conséquent,

$p^T = B^{-1}A.x^T$…… (2.8)

Donc, (2.8) peut être utilisé pour calculer les bits de contrôle sous la condition (2.6) que B soit non singulière.

D'une manière générale, la matrice de contrôle de parité H ne sera pas une matrice sparce après le prétraitement. Ainsi, la complexité des procédés classiques pour le codage de ce code LDPC est élevée. **[8]**

2.2.5.2. Codage par approximation triangulaire inférieure

La complexité des algorithmes de codage classiques est essentiellement proportionnelle au carré de la longueur de code et devient un problème important pour de grandes longueurs de code. Pour résoudre ce problème, Richardson et Urbanke proposent un algorithme de codage efficace pour des codes LDPC. Nous allons donner une description détaillée de cet algorithme de codage dans ce qui suit.

L'idée est de faire une transformation de la matrice de contrôle de parité en utilisant seulement la permutation des lignes et des colonnes de manière à maintenir creuses. Toute matrice creuse arbitraire peut être converti en une matrice de contrôle de parité souhaitée، avec une forme triangulaire inférieure approximative.

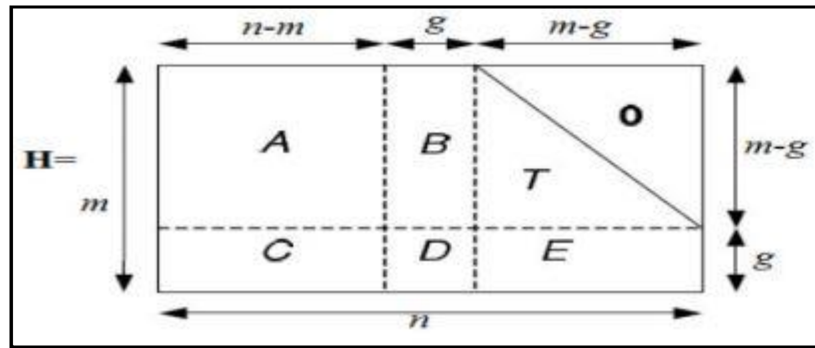


Fig. 2.1-Représentation sous forme pseudo-triangulaire inférieure de la matrice H. [8]

a) Algorithme de Richardson-Urbanke :

Les étapes de cet algorithme se résument comme suite :

1. Effectuer la permutation de ligne et de colonne pour mettre ،sous une forme triangulaire inférieure approximative :

$$H=\begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}$$

Où A est de taille (n$-m$)$\times (m-g)$, B est de taille $g \times (m-g)$, T est une matrice triangulaire inférieur de taille (m$-g$)$\times (m-g)$,C est de taille (n$-m$) $\times g, D$ est de taille $g \times g$ et enfin E est de taille $(m-g) \times g$. Les$g$ lignes de H sont appelés l'écart de la représentation approximative.

2. Une fois le format triangulaire supérieure T est obtenu, nous utilisons l'élimination de Gauss pour vider E, ce-qui est équivalent à la pré-multiplication suivante :

$$\begin{pmatrix} I & 0 \\ -ET^{-1} & I \end{pmatrix}\begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix} = \begin{pmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{pmatrix}$$

$$= \begin{pmatrix} A & B & T \\ C' & D' & 0 \end{pmatrix}$$

Où l'on note

$D' = -ET^{-1}B + D$…... (2.9)

3. Le codage

Considérons le mot de codeCconstitué d'une partie x systématique et de deux parties de parité P_1etP_2, avec les longueurs g et $(m-g)$, respectivement. On applique la condition $H.x^T = 0$ au code $C = [x\ P1\ P2]$ nous obtenons :

$Ax^T + Bp_1^T + Tp_2^T = 0$……. (2.10)

$\hat{C}x^T + \hat{D}P_1^T + 0P_2^T = \hat{C}x^T + \hat{D}P_1^T = 0$ …… (2.11)

Supposons que D est inversible, $p1$ peut être trouvé à partir de (2.10) :

$\boldsymbol{P_1^T} = -\hat{D}^{-1}\hat{C}x^T = -\hat{D}^{-1}(-ET^{-1} + A + C)x^T$ …… (2.12)

Où la faible densité de A, B et T peut être utilisé pour maintenir la complexité de cette opération faible ; puisque T est triangulaire supérieure, P_2peut-être trouvé par :

$\boldsymbol{P_2^T} = -T^{-1}(Ax^T + B\boldsymbol{P_1^T})$ …… (2.13)

Cette méthode est la plus populaire pour le codage des codes LDPC et elle a été adoptée par les IEEE802.11n et les normes IEEE 802.16e. L'avantage de ces codes est leurs constructions qui sont réalisée d'une manière systématique qui diminue la complexité de codage et abaisse la mémoire requise. **[8]**

2.2.6. Décodage des codes LDPC

Le décodage est le moyen qui permet de reconstruire le mot C transmis à partir du message X (X étant la sortie du canal de transmission). L'opération de décodage utilise la matrice de contrôle de parité H. La condition $\hat{X} \circ H^T = 0$ ($\hat{X}$ étant le message binaire estimé par le

décodeur à partir du message reçu Y) définie les contraintes de parité que doit vérifier le mot estimé $\hat{X}$ pour qu'il puisse s'agir d'un mot code. En considérant la matrice H de la figure (2.2)

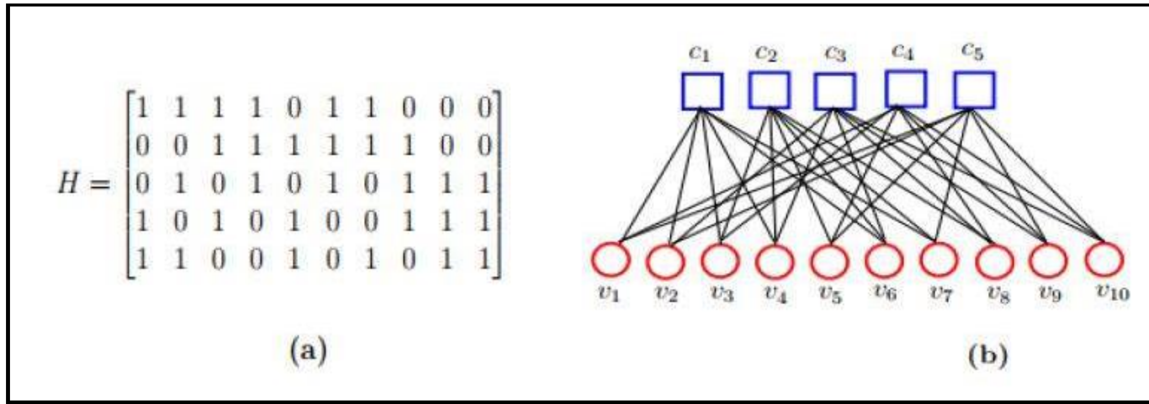


Fig. 2.2-(a) -Matrice de contrôle de parité H (b) : Graphe de Tanner correspondant. [5]

Et $\hat{X} = (\hat{x}_0, \hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4, \hat{x}_5, \hat{x}_6, \hat{x}_7, \hat{x}_8, \hat{x}_9)$, les 5 contraintes de parité seront alors :

$\hat{x}_0 \oplus \hat{x}_1 \oplus \hat{x}_2 \oplus \hat{x}_3 \oplus \hat{x}_5 \oplus \hat{x}_6 = 0$

$\hat{x}_2 \oplus \hat{x}_3 \oplus \hat{x}_4 \oplus \hat{x}_5 \oplus \hat{x}_6 \oplus \hat{x}_7 = 0$

$\hat{x}_1 \oplus \hat{x}_3 \oplus \hat{x}_5 \oplus \hat{x}_7 \oplus \hat{x}_8 \oplus \hat{x}_9 = 0$

$\hat{x}_0 \oplus \hat{x}_2 \oplus \hat{x}_4 \oplus \hat{x}_7 \oplus \hat{x}_8 \oplus \hat{x}_9 = 0$

$\hat{x}_0 \oplus \hat{x}_1 \oplus \hat{x}_4 \oplus \hat{x}_6 \oplus \hat{x}_8 \oplus \hat{x}_9 = 0$

Si le mot binaire estimé $\hat{X}$ vérifie $\hat{X} \circ H^T = 0$ alors celui-ci est un vrai mot code, sinon le message binaire estimé par le décodeur contient encore des erreurs de transmission. **[5]**

Par rapport aux autres types de codes, le décodage des codes LDPC ne pose pas autant de problèmes pour les chercheurs que leur construction. Le travail le plus difficile est de trouver les meilleures méthodes pour construire des codes LDPC efficaces. Un code LDPC peut être décodé par plusieurs méthodes, telles que :

1-Décodage avec des décisions fermes

- Décodage avec la logique majoritaire (MLG)
- Décodage avec basculement de bit (BF)

2-Décodage avec des décisions pondérées

- Décodage basé sur la probabilité a posteriori (APP)
- Décodage itératif basé sur la propagation de la confiance (somme-produit SPA)

3-Décodage mixte (ferme et pondéré)

Décodage BF pondéré

La méthode MLG est la plus simple du point de vue de la complexité du circuit. La méthode BF demande un peu plus de complexité du circuit, mais elle donne des meilleures performances d'erreur que la méthode MLG. Les méthodes APP et SPA donnent des meilleures performances d'erreur, mais nécessitent aussi une plus grande complexité du circuit. Le décodage BF pondéré représente un bon compromis entre les deux caractéristiques. La méthode SPA donne la meilleure performance d'erreur entre les 7 types de décodage. **[9]**

2.2.6.1. Décodage MLG des codes LDPC

La méthode MLG à une seule étape peut être appliquée au décodage des 4 types de codes LDPC. On calcule les syndromes :

$$s_j^{(l)} = e * h_j^{(l)} = \sum_{i=0}^{n-1} e_i \, h_{j,i}^{l} \quad \ldots\ldots (2.14)$$

Où $h_j^{(l)}$ $(1 < j < \gamma)$ sont les lignes de **H** qui sont orthogonales sur le bit de la l-ème position. L'ensemble des sommes de contrôle $s_j^{(l)}$ sont orthogonales sur le bit d'erreur e_l et on peut les utiliser pour l'estimation de e_l . Le bit d'erreur est bien corrigé si dans le vecteur d'erreur il y a moins de $\gamma/2$ erreurs. **[9]**

2.2.6.2. Décodage BF

Le décodage Bit Flipping BF est un décodage itératif de décision hard permet de comparer les résultats binaires des messages reçus avec celles du mot code pour voir si le message est bien reçu. Cet algorithme détecte les erreurs puis les corriger dans les étapes suivantes :

1) Un nœud de bit (variable nœud) envoie un message déclarant s'il est un ou zéro.

2) Les messages de nœud de contrôle sont calculés à partir de l'équation de parité où est $E_{i,j}$ égale à l'exor du message reçu $M_i$correspond le i-ième bit de w_c dans la matrice de parité puis les envoyer vers les variables nœuds.

3) chaque variable nœud reçoit un message envoyé par les nœuds de contrôle, si le message égal à un donc ce bit est correct si non le bit change (flip) sa valeur actuelle. Ces bits sont envoyés à nouveau aux nœuds de contrôles. Chaque nœud de contrôle détermine son équation de contrôle de parité. Cette équation est satisfaite si la somme modulo-2 des valeurs de bits entrants est nulle, si non ce processus est répété jusqu'à ce que toutes les équations de contrôle de parité soient satisfaites, ou jusqu'à ce qu'un nombre maximum d'itérations de décodeur soit passé et que le décodeur soit abandonné. **[6]**

2.3. L'algorithme de décodage SPA (Sum-ProductAlgorithm)

L'algorithme SPA dit aussi LLR-SPA (Log-Likelihood Ratios Sum-Product Algorithm), est un algorithme à décision soft et généralement qualifié d'optimal puisque le décodage SPA converge vers le maximum de vraisemblance à condition que le graphe biparti associé au code LDPC ne contienne pas des cycles courts. Cet algorithme atteint les meilleures performances dans les applications pratiques, parce qu'il est très proche de la capacité du canal (limite de Shannon) et qu'il offre le meilleur accord possible entre le rendement et la performance (proches de la limite de Gilbert -Varshamov)

Les probabilités de bit d'entrée sont appelées les probabilités a priori pour les bits reçus car ils étaient connus à l'avance avant d'exécuter le décodeur LDPC. Les probabilités de bits retournées par le décodeur sont appelées les probabilités a posteriori. Dans le cas du décodage SPA, ces probabilités sont exprimées en rapports log-vraisemblance LLR (log-likelihood ratio)

Le but du décodage SPA est de calculer la probabilité maximale a posteriori (MAP) pour chaque bit de mot de code, Pi = P {ci = 1 | n}, qui est la probabilité $i_{ème}$ bit de mot de code soit 1 conditionnel à n que toutes les contraintes de contrôle de parité sont satisfaites. L'information supplémentaire sur le bit i reçu des contrôles de parité est appelée information extrinsèque pour le bit i.

On pose $L_{1_{i \to k}}$ la fonction LLR correspond au message envoyé par le $i_{ème}$ variable nœud au nœud de contrôle. Et $L_{2_{k \to i}}$ est la fonction LLR du message envoyé (x_i) au variable nœud par le nœud de contrôle. Les principales étapes du LLR-SPA sont les suivantes :

1) Initialisation : chaque variable nœud envoie un message x_i au nœud de contrôle

$$L_{1_{i \to k}}(x_i) = \text{LLR}\ (x_i) = \ln \frac{p\ (x_i = \ 0)}{p\ (x_i = 0)} \ \ldots\ldots.\ (2.15)$$

Où $p(x_i \ = x)$et $x_i \in \{0{,}1\}$ est la probabilité du mot code au $i_{\ ème}$ peut avoir le bit x.

2) Dans la deuxième étape de LLR-SPA les nœuds de contrôles envoient des messages aux nœuds de variable.

$$L_{2_{k \to i}}(x_i) = 2.\tanh^{-1}(\prod_{j \in A(k) \backslash i} \ \tanh(\tfrac{1}{2} L_{1_{i \to k}}\ (x_j)) \ \ldots\ldots.(2.16)$$

3) La troisième étape de l'algorithme LLR-SPA : Les messages envoyés par les nœuds variables aux nœuds de contrôles sont calculés au moyen de la formule suivante :

$$L_{1_{i \to k}}(x_i) = \text{LLR}\ (x_i\ ) + \sum_{j \in B(i) \backslash k} L_{2_{k \to i}}(x_i) \ldots\ldots\ (2.17)$$

Cette équation est évaluée sou la forme suivante :

$$L_{1_i}(x_i) = \text{LLR}\ (x_i) + \sum_{j \in B(i)} L_{2_{k \to i}}(x_i) \qquad \ldots\ldots\ (2.18)$$

Cette équation va être utilisée dans l'étape suivante.

4) Décision : Dans cette étape, la valeur de fiabilité calculée par l'équation (2.18) est utilisée pour obtenir une valeur estimée x_i' de mot code recevé x_i à partir de la loi suivante :

$$x_i' = \begin{cases} 0 & L_{1_i}(x_i) \geq 0 \\ 1 & L_{1_i}(x_i) < 0 \end{cases} \quad \text{……………..} \ (2.19)$$

Ensuite, le syndrome du mot de code x_i' estimé par la matrice H est calculé. Si le syndrome est nul donc l'équation de parité est satisfaite, le décodage s'arrête et donne le mot de code x_i' comme résultat.

Si non, l'algorithme va être répéter, en revenant à l'étape 2 avec les valeurs actualisées de $L_{1_{i \to k}}(x_i)$. Dans ce dernier cas, une nouvelle vérification est fait sur le nombre d'itérations : Lorsqu'un nombre maximal d'itérations préfixé est atteint, le décodeur arrête d'itérer et émet le mot de code estimé comme résultat de traitement. Cependant, si le décodage échoue, un message d'erreur est détecté. **[6]**

2.4. Architectures des décodeurs pour émulateurs

2.4.1. Architectures de décodeur LDPC

Une architecture matérielle typique d'un décodeur LDPC se compose de plusieurs unités à nœuds variables (VNU) et unités de nœuds de contrôle (CNU), des mémoires optionnelles pour les LLR a priori / postérieur et les messages échangés, et un réseau d'interconnexion (routage) comme le montre la figure (2.3). Dans le cas de décodeurs basés sur MS, le principal des VNU l'opération est représenté par l'addition, tandis que les CNU calculent le premier et la seconde valeur minimale parmi les valeurs d'entrée (par exemple, en utilisant un arbre de comparateurs).

Notez que les VNU et CNU peuvent être interconnectés de différentes manières (en fonction de l'architecture matérielle du décodeur) et parfois ils peuvent être fusionnés en une seule unité de traitement.

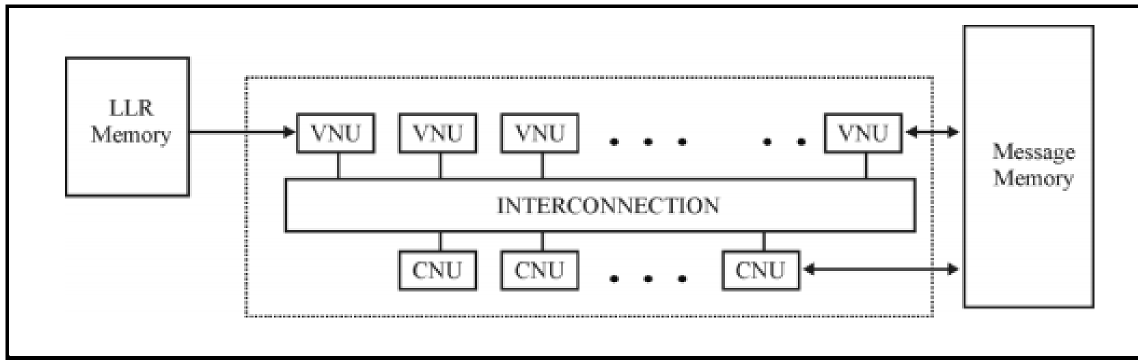


Fig. 2.3 -Architecture matérielle générale d'un décodeur LDPC. [10]

Généralement, les architectures de décodeur LDPC peuvent être classées en trois catégories principales, à savoir les architectures entièrement parallèles, série et partiellement parallèles.

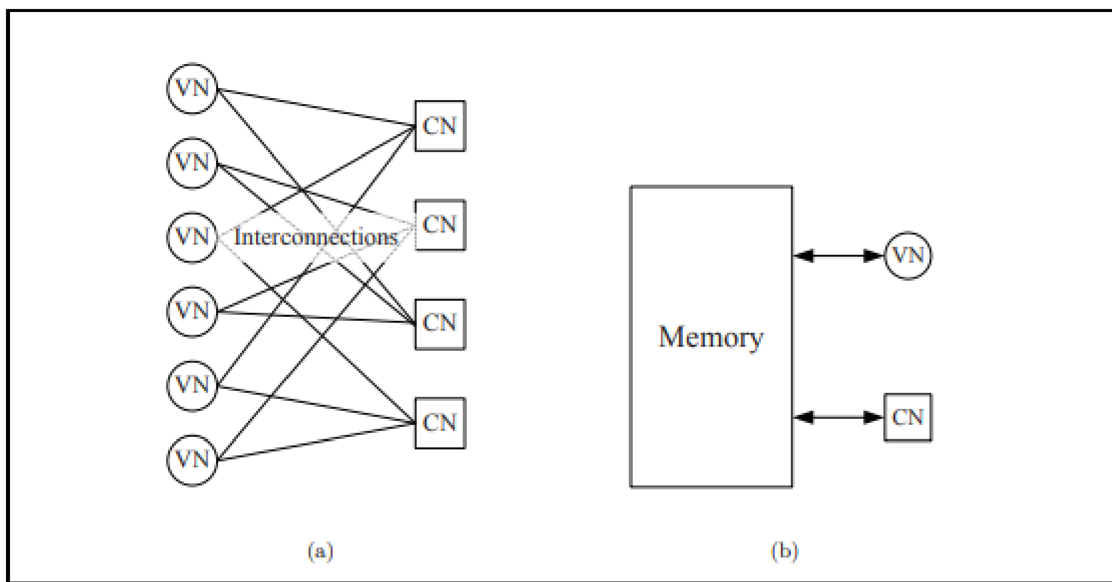


Fig. 2.4-Illustration (a) d'une architecture de décodeur parallèle et (b) d'un décodeur série architecture. [10]

Entièrement parallèle les architectures implémentent naturellement un ordonnancement inondé. De telles implémentations sont caractérisées par un très grand nombre de CNU et les VNU et un réseau d'interconnexion dense. En raison de la grande, et généralement très réseau d'interconnexion irrégulier, les décodeurs entièrement parallèles posent des problèmes importants dans le lieu-ET-routage ou le câblage du décodeur. Par conséquent, les implémentations entièrement parallèles ont rarement été adoptées dans les implémentations pratiques.

Dans le spectre de conception opposé, les architectures série n'utilisent qu'un seul VNU et un CNU, qui sont ensuite réutilisés pour traiter tous les nœuds de variable et de contrôle du graphe bipartite. Le passage de message est implémenté en stockant les messages calculés dans une mémoire dédiée, et les lire à partir de la mémoire chaque fois qu'ils sont nécessaires à une autre unité de traitement. Les architectures série présentent de nombreux avantages :

- ✓ Très faible coût,
- ✓ Pas de problèmes de congestion des itinéraires, peut mettre en œuvre l'un des planifications inondées ou en couches, qui peut être simplement contrôlée par l'unité de contrôle
- ✓ Une grande flexibilité, car ils prennent en charge une grande variété de codes. Mais ils ont aussi un inconvénient majeur : ils ont un débit très faible, qui peut être bien trop
- ✓ Faible pour des applications pratiques dans des systèmes de communication modernes, caractérisés par un besoin accru de débits de données de plus en plus élevés.

L'architecture partiellement parallèle hérite des principales caractéristiques et avantages des deux architectures ci-dessus. Dans l'architecture partiellement parallèle, le nombre des VNU et CNU instanciés dans le matériel sont inférieurs au nombre de variables et les nœuds de contrôle et plusieurs nœuds partagent la même unité de traitement. Différents compromis entre la surface, le débit et la flexibilité peuvent être obtenus en ajustant le nombre d'unités de traitement. Bien que le concept d'architecture partiellement parallèle ne soit pas nécessairement lié à un ordonnancement de décodage spécifique, il est naturellement adapté aux stratégies de planification en couches. Par conséquent, la plupart des architectures partiellement parallèles sont en fait des architectures en couches partiellement parallèles,

utilisant des codes QC-LDPC structurés, afin de réduire encore la complexité du réseau d'interconnexion.

Les caractéristiques des architectures entièrement parallèles, sérielles et partiellement parallèles sont résumées dans la table (2.1). **[10]**

Architecture du décodeur	Zone	Débit	La flexibilité
Entièrement parallèle	Large	Très faible	✗
En série	Petite	Élevé	✓
Partiellement Parallèle	Moyen	Haute	✓

Table. 2.1-Principales caractéristiques des architectures de décodeur. [10]

2.4.2. Architectures de décodeur pour émulateur

La conception d'un émulateur de décodeur sur FPGA doit être différenciée de la conception d'un décodeur pour des implémentations pratiques. Les implémentations pratiques visent des performances élevées (Fonction et débit) et efficacité (surface et puissance) tout en satisfaisant une particulière exigence de l'application, tandis que l'émulateur de décodeur est conçu avec une efficacité des ressources et la configurabilité comme objectifs principaux. La plateforme FPGA est utilisée comme une enquête plate-forme, et en tant que telle une grande quantité de ressources sur FPGA sont dédiées à la capture des traces d'événements pour l'analyse, laissant un niveau limité de parallélisme disponible pour la conception du décodeur.

L'architecture du décodeur doit également être très reconfigurable, afin de pouvoir être programmé pour différents codes, algorithmes de décodage et capable de fonctionner avec un nombre variable d'itérations à différents niveaux de SNR.

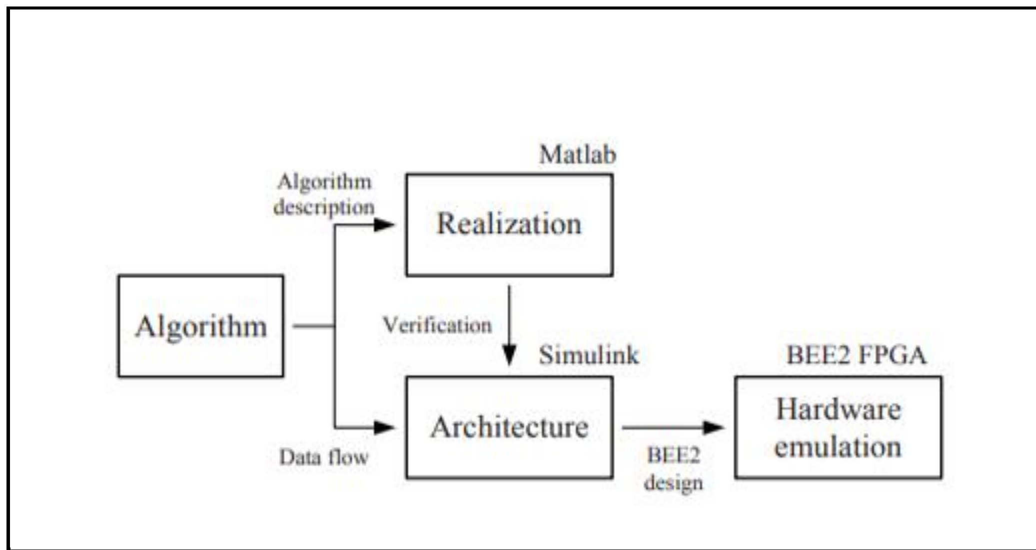


Fig. 2.5 -Flux de conception pour l'émulation matérielle. [11]

Deux architectures ont été utilisées pour mapper les émulateurs de décodeur, une architecture canonique et une architecture en couches. Les deux architectures sont basées sur la parallélisation partielle de l'architecture série, qui ressemble aux conceptions proposées dans, mais le degré de parallélisme est volontairement limité en partitionnant la matrice H en seulement une direction (c.-à-d. paralléliser entre les partitions de colonne et traiter les lignes en série) pour réduire complexité. Chacune des partitions est configurable en fonction de la structure de la matrice H.

Par rapport à une architecture entièrement parallèle, qui n'est pas reconfigurable, ou à une architecture entièrement en série architecture, qui n'a pas le débit, ces architectures représentent un compromis pour le but de l'émulation de code.

Un code RS-LDPC (6, 32) régulier (2048, 1723) est sélectionné pour l'illustration de ces architectures. Ce code LDPC particulier a été adopté comme correction d'erreur directe dans la norme IEEE 802.3an 10GBASE-T, qui régit le fonctionnement de 10 Gigabit

Ethernet sur jusqu'à 100 m de câble à paires torsadées non blindées CAT-6a (UTP). La matrice H de ce code contient M = 384 lignes et N = 2048 colonnes. Cette matrice peut être partitionnée en $\gamma = 6$ groupes de lignes et $\rho = 32$ groupes de colonnes de $\delta \times \delta = 64 \times 64$ sous-matrices de permutation. Dans l'architecture canonique, la partition de colonne est appliquée pour diviser le décodeur en 32 unités parallèles, où chaque unité traite un groupe de 64 bits. La figure 2.9 illustre L'architecture du décodeur de produit somme RS-LDPC. Deux jeux de mémoires, M0 etM1, sont conçus pour être accessibles en alternance. M0 stocke les messages de variable à vérifier et M1 stocke les messages de vérification des variables. Chaque ensemble de mémoires est divisé en 32 banques. Chaque banque est affectée à une unité de traitement qui peut y accéder indépendamment. Dans une opération check-to-variable définie dans (2.20), les 32 messages de variable to check passent par le log-tanh, puis le nœud de contrôle calcule la somme de ces messages. La somme est marginalisée localement dans l'unité de traitement et stockée dans M1. Les messages stockés passé par la transformation inverse log-tanh pour générer des messages de vérification en variable. Dans l'opération de variable à vérifier définie dans (2.19), le nœud de variable à l'intérieur de chaque traitement

L'unité accumule les messages de vérification des variables en série. La somme est marginalisée localement et stocké dans M0. Cette architecture minimise le nombre d'interconnexions globales en effectuant marginalisation au sein de l'unité de traitement locale. **[11]**

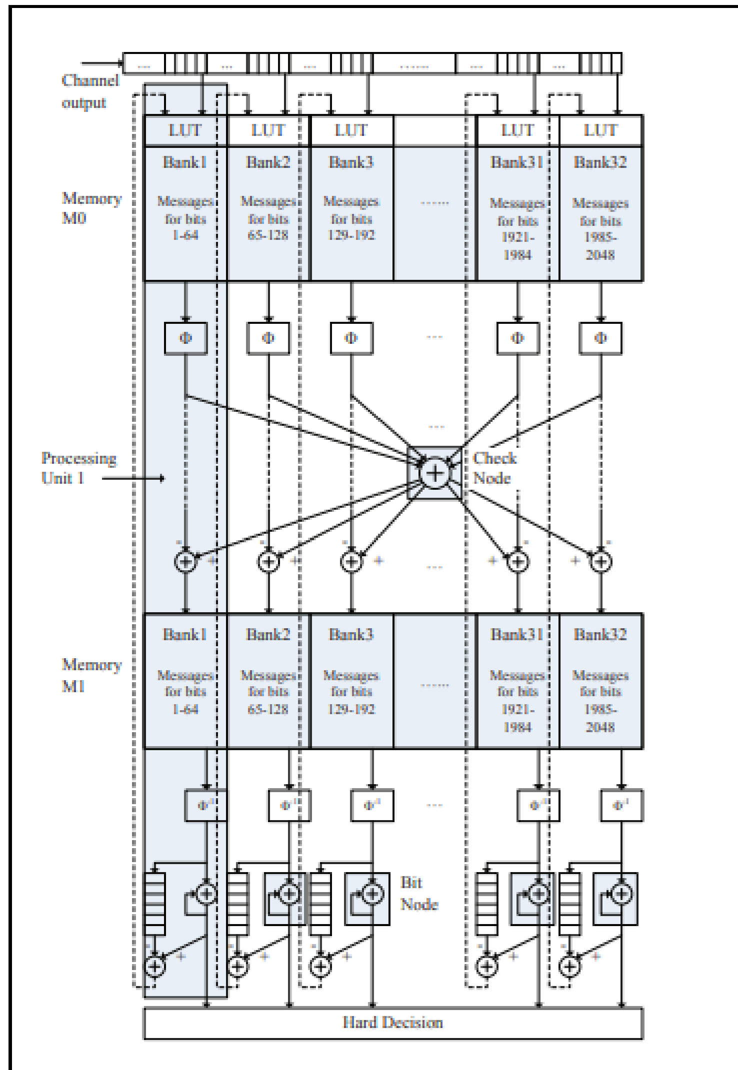


Fig.2.6-Une architecture canonique du (2048,1723) décodeur RS-LDPC composé de 32unités de traitement. [11]

L'architecture canonique réalise la forme canonique de l'algorithme somme-produit illustré dans (2.19), (2.20), (2.21) et (2.22). Ces équations peuvent également être réorganisées par en tenant compte de la relation entre les itérations de décodage consécutives. Un message variable-to-check d'itération n peut être calculé en soustrayant le message check-to-variable correspondant du postérieur de l'itération n - 1 comme dans (2.23), tandis que le postérieur de l'itération n peut être calculé en mettant à jour le postérieur de l'itération précédente avec le message check-to-variable de l'itération n, comme dans (2.25).

$$L(q_{ij}) = \sum_{j' \in Col[i] \backslash j} L(r_{ij'}) + L^{pr}(x_i) \quad \text{.......} \quad (2.19)$$

$$L(r_{ij}) = \Phi^{-1}\left(\sum_{i' \in Row[j] \backslash i} \Phi(|L(q_{i'j})|)\right)\left(\prod_{i' \in Row[j] \backslash i} sgn(L(q_{i'j}))\right) \quad \text{.......} \quad (2.20)$$

$$L^{ext}(x_i) = \sum_{j' \in Col[i]} L(r_{ij'}) \quad \text{.......} \quad (2.21)$$

$$L^{ps}(x_i) = L^{ext}(x_i) + L^{pr}(x_i) \quad \text{.......} \quad (2.22)$$

$$L_n(q_{ij}) = L_{n-1}^{ps}(x_i) - L_{n-1}(r_{ij}) \quad \text{.......} \quad (2.23)$$

$$L_n(r_{ij}) = \Phi^{-1}\left(\sum_{i' \in Row[j] \backslash i} \Phi(|L_n(q_{i'j})|)\right)\left(\prod_{i' \in Row[j] \backslash i} sgn(L_n(q_{i'j}))\right) \quad \text{.....} \quad (2.24)$$

$$L_n^{ps}(x_i) = L_{n-1}^{ps}(x_i) - L_{n-1}(r_{ij}) + L_n(r_{ij}),\ j \in Col[i] \quad \text{.......} \quad (2.25)$$

L'algorithme de produit de somme reformulé conduit à une planification de passage de message centrée sur le nœud de contrôle sans opération de nœud variable explicite. Lorsqu'il est interprété à l'aide du H matrice, les opérations sont effectuées en couches horizontales, c'est ce qu'on appelle l'architecture en couches. Le schéma fonctionnel de l'architecture en couches est illustré à la Fig.2.7 pour le (2048, 1723) Code RS-LDPC. Un seul jeu de mémoire M0 est nécessaire pour stocker les messages check-to-variable et le postérieur. Dans chaque itération sauf la première, le message de vérification de variable de l'itération précédente est soustrait du postérieur pour produire le message de variable à vérifier comme dans (2.23). Un message de variable à vérifier de chacun des groupes de colonnes est traité par le nœud de contrôle, et le nouveau message de contrôle de variable est calculé conformément à (2.24). Le

nouveau message check-to-variable remplace l'ancien check-to-variable message pour mettre à jour le postérieur comme dans (2.25). Par rapport à l'architecture canonique, le l'opération de variable à vérifier est entrelacée avec l'opération de vérification à variable dans la couche architecture. Les deux types d'architectures permettent une cartographie efficace d'un décodeur pratique. Pour exemple, un code RS-LDPC d'une longueur de bloc allant jusqu'à 8 Ko peut être pris en charge sur un Xilinx Virtex-II Pro XC2VP70 FPGA. Ces architectures sont également reconfigurables, de sorte que tout membre de la famille de codes LDPC décrite dans la section (2.20) peut être accepté. Les tables de recherche d'adresses peuvent être reconfigurées en fonction de la matrice H. Les unités de traitement peuvent être allouée en fonction des partitions de colonne, et la taille de la mémoire peut être ajustée à autoriser des taux de code variables.

Le débit de décodage des deux types d'architectures est déterminé par les dimensions de la matrice H du code LDPC. Dans un régime de SNR élevé, la majorité des trames reçues peuvent être décodées en une seule itération. Par conséquent, le maximum réalisable le débit est d'environ $f_{clk}\ N\ /\ (2M)$ pour l'architecture canonique et $f_{clk}\ N\ /M$ pour l'architecture en couches, où f_{clk} représente la fréquence d'horloge. Depuis le blocage du pipeline doivent être insérés entre les opérations de variable à vérifier et de vérification à variable dans une architecture canonique, et entre les couches horizontales dans une architecture en couches pour résoudre les données dépendances, le débit maximal pouvant être atteint en pratique est légèrement inférieur à ce qui est cité ci-dessus. Notez qu'une caractéristique des deux types d'architectures est que le décodage le débit dépend de $N\ /\ M$, qui est lié au débit du code - plus le code est élevé débit, plus le débit de décodage est élevé. **[11]**

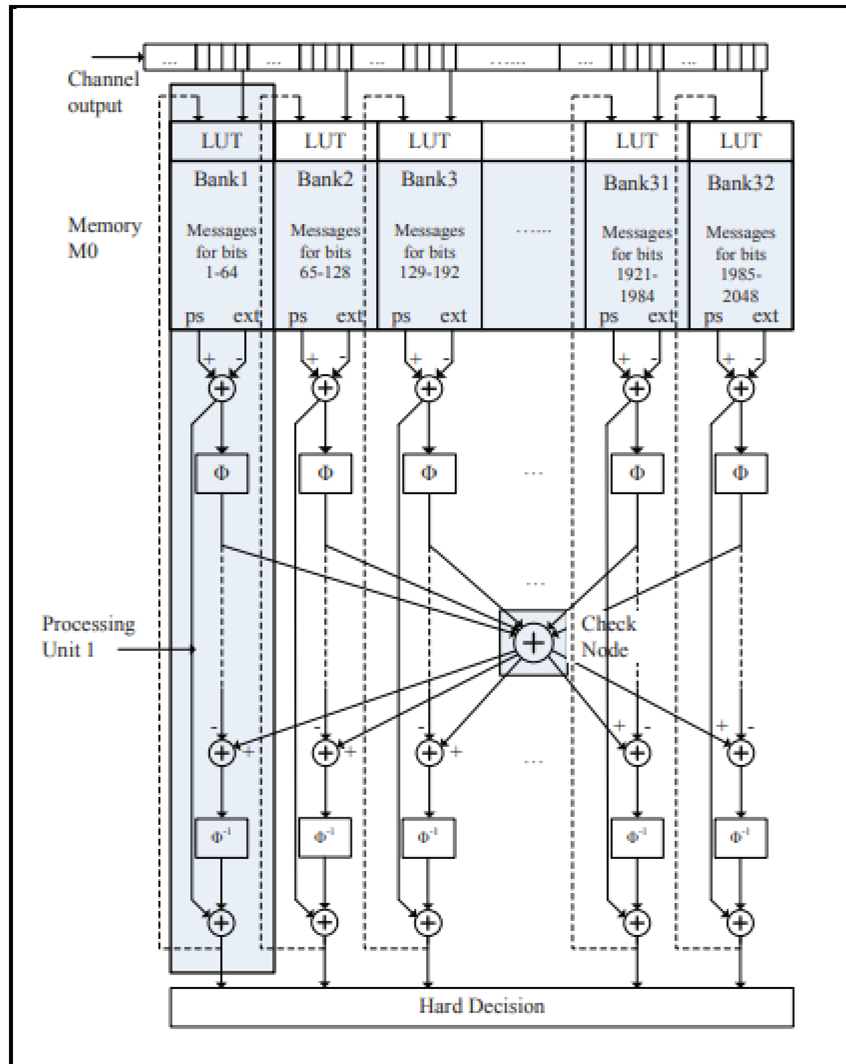


Fig.2.7-Une architecture en couches du (2048,1723) décodeur RS-LDPC composé de 32unités de traitement. [11]

2.5. Interconnexions entre nœuds de variables et nœuds de parité

2.5.1. Graphe de Tanner

Un code LDPC peut également être représenté sous une forme graphique. Cette représentation est appelée graphe biparti ou graphe de Tanner composer de deux groupes de nœuds, reliés par des branches :

- Des nœuds de variable, représentés par des cercles, et représentant les symboles du mot de code. Chacune des colonnes de H est donc associée à un nœud de variable. L'ensemble de ces nœuds est noté $E_v$et est de cardinal *N*.
- Des nœuds de contrainte, représentés par des carrés, et représentant les contraintes de parité. De même, chacune des lignes de H est associée à un nœud de contrainte. L'ensemble de ces nœuds est noté $E_c$et est de cardinal *M*.
- Une branche relie un nœud de variable à un nœud de contrainte si l'intersection de la ligne et de la colonne leur correspondant est non nulle. Le nombre de branches liées à un nœud n s'appelle degré de ce nœud et est noté d_n.

L'ensemble des nœuds directement liés à un nœud n s'appelle son Voisinage. Il est de cardinal d_n et sera noté V_n: V_n= $\{x_i \in$ graphe $|x_i$ lié à n $\}$. Lorsque n est un nœud de variable, les x_i sont des nœuds de contrainte. A l'inverse, lorsque n est un nœud de contrainte, les x_i sont des nœuds de variable. Afin d'avoir un ordre dans les indices des éléments d'un voisinage donné, nous adopterons la notation $V_n = \{x_{\mu_n(1)} \ldots x_{\mu_n(d_n)}\cdot)\}$. L'application μ_n numérote les voisins du nœud n et est notée ϕ dans le cas des nœuds de contrainte et ψ dans le cas des nœuds de variable : le $i^{ème}$voisin du nœud de variable v est le $(\psi(i))^{ème}$ nœud de contrainte dans E_C. De même, le $i^{ème}$ voisin du nœud de contrainte c'est le $(\phi\ (i))^{ème}$ nœud de variable dans E_V. Nous omettrons dans la suite l'indice n dans µn afin d'alléger les notations.

Exemple : Matrice de contrôle de parité H et Graphe de Tanner correspondant

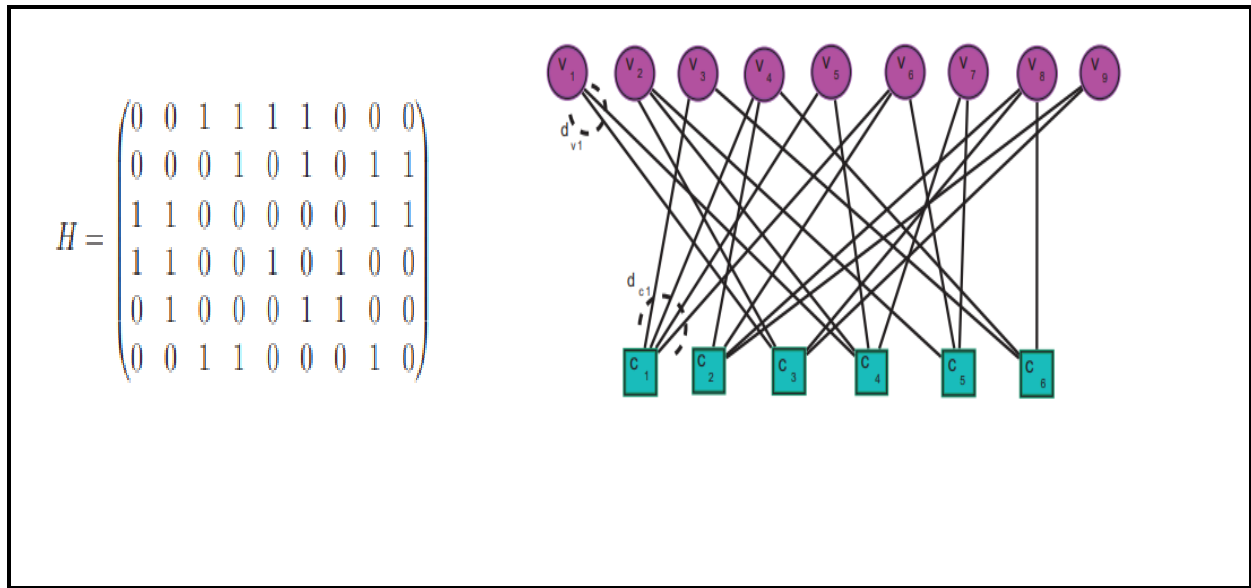


Fig. 2.8 -Matrice de contrôle de parité H et Graphe de Tanner correspondant. [12]

Dans l'exemple de la figure 2.8, la matrice est irrégulière. Le poids de chaque ligne ou colonne correspond à son degré. Le degré du nœud de variable $v_1$est d_{v1}= 2, et le degré du nœud de contrainte $c_1$est d_{c1} = 4. Le voisinage du nœud de contrainte c_1 est V_{c1}= { v_3, v_4, v_5, v_6}. Dans ce cas, {φ (1) = 3, φ (2) = 4, φ (3) = 5 et φ (4) = 6}. **[12]**

2.6. Conclusion

Dans ce chapitre on conclure :

- ✓ Les codes LDPC se sont les plus performants et les plus utilisés à ce jour.
- ✓ Le décodage SPA représente le meilleur algorithme de décodage pour les codes LDPC et le plus utilisé.
- ✓ Les plates-formes FPGA ont été couramment utilisées dans l'étude des conceptions de décodeurs, principalement à deux fins : vérifier l'architecture matérielle et accélérer la simulation de code.**[11]**
- ✓ L'émulation matérielle accélère également la simulation de code. **[11]**

CHAPITRE 3

Automatisation des interconnexions pour émulateur sur FPGA

3.1. Introduction

Les codes LDPC sont représentés par une matrice de contrôle de parité H. Cette matrice H était écrite dans un format appelé alist par David Mackay, Matthew Davey et John Lafferty. Dans ce chapitre nous avons défini ses derniers. D'autre part, on a traité aussi la conversion de Alist format vers VHDL Bram et dernièrement les architectures des interconnections sur FPGAs.

3.2. Les matrices de contrôle de parité

3.2.1. Définition

- Une matrice de contrôle est un concept de théorie des codes utilisé dans le cas des codes correcteurs linéaires.

Elle correspond à la matrice d'une application linéaire ayant pour noyau le code.

La notion de matrice de contrôle possède à la fois un intérêt théorique dans le cadre de l'étude des codes correcteurs, par exemple pour offrir des critères sur la distance minimale du code ou une condition nécessaire et suffisante pour qu'un code soit parfait et un intérêt pratique pour un décodage efficace. **[W3]**

- Soit H la matrice appelée matrice de contrôle de parité, destinée à permettre le contrôle d'erreur. Pour toute matrice G de taille $(k \times n)$, il existe une matrice H, de taille $(n - k, n)$, telle que les lignes de H soient orthogonales aux lignes de la matrice G, autrement dit, $GH^T = \underline{0}$, où H^T est la transposée de la matrice H et $= \underline{0}$ une matrice $(k, n - k)$ dont tous les éléments sont nuls. **[W4]**

3.2.2. Matrice G de génération du code

Considérons le message d de longueur k défini par la relation suivante :

$d = [d_1, \cdot\cdot\cdot, d_k]$ …………………... (3.3)

Une fois ce dernier codé, nous obtenons le vecteur c de longueur n.

$c = [c_1, \cdot\cdot\cdot, c_n]$ …………………... (3.4)

Pour générer le mot codé c, on applique l'opération binaire suivante à d :

$c = d \cdot G$ …………………….. (3.5)

Où la matrice G est appelée matrice de codage et a pour dimensions $k \times n$.

Afin d'être en mesure de décoder correctement le message initial au niveau du décodeur, deux messages d^i et d^j différents doivent engendrer des mots codes c^i et c^j in C différents. L'analyse de l'équation 3.5 montre qu'un mot de code c'est une combinaison linéaire des lignes de G. Pour cette raison, et pour garantir que chaque mot de code c'est unique, toutes les lignes de G doivent être linéairement indépendantes (c'est-à-dire, G est de plein rang).

Certains codes en blocs sont dits systématiques, ce qui signifie que dans la représentation d'un mot de code c, les bits constitutifs du message d peuvent être directement isolés des bits de contrôle de parité. La représentation de c peut ainsi être décomposée comme indiqué dans l'équation 3.9.

$c = [c_1, \cdots, c_n] = [p_1, \cdots, p_{n-k}, d_1, \cdots, d_k] = [p/d]$ ………… (3.6)

Où $p = (p_1, \cdots, p_{n-k})$ est le vecteur des bits de contrôle de parité et $d = (d_1, \cdots, d_k)$ le vecteur des bits de données. La matrice G peut alors être vue comme étant la juxtaposition d'une matrice de parité P de dimensions $k \times (n - k)$ et d'une matrice identité I_k de dimensions $k \times k$.

G = [P | Ik] …………………. (3.7) **[4]**

3.2.3. Choix de la matrice de contrôle de parité

Le choix de la matrice de contrôle de parité H a un impact majeur sur les performances et la complexité du codeur/décodeur LDPC. Les matrices de faible densité sont toujours souhaitables. La construction de la matrice H peut se faire d'une manière aléatoire ou structurée. Les matrices aléatoires sont généralement utilisées sur les processeurs décrits en software par contre les matrices structurées sont souhaitables pour l'implémentation du codeur/décodeur sur les circuits programmables de type FPGAs ou ASICs ou DSPs.

La structure dans une matrice de contrôle de parité assure une flexibilité quant à la conception des unités de fonctionnements effectuant le traitement requis par les nœuds VNs et CNs. Ce type de construction requiert aussi moins de ressources hardware, en termes de mémoire pour le stockage de la matrice de contrôle de parité. C'est le cas par exemple des matrices utilisées dans le standard de la norme DVB-S2, pour laquelle la longueur du mot code (nombre de

colonne de la matrice) est fixée à 64800 bits pour la trame normale et 16200 bits pour la trame courte. Quelques constructions des matrices aléatoires sont disponibles online sur la base de données de Mackay.

La matrice H peut se mettre sous sa forme systématique $new\ H = \mathrm{I}_{n-k}\mathrm{P}^{\mathrm{T}}$ par la méthode d'élimination de Gauss-Jordan qui consiste à effectuer un ensemble d'opérations telles que : la permutation des lignes, le remplacement d'une ligne par le résultat de l'addition modulo 2 de deux ou plusieurs lignes. Parfois quelques permutations des colonnes sont aussi effectuées. **[5]**

3.2.4. Exemples de matrice de contrôle de parité

Impossibles à mettre en œuvre lors de la première mise au point par Gallager en 1963, les codes LDPC ont été oubliés jusqu'à ce qu'ils soient "redécouverts" en 1996. Le terme « faible densité » signifie que le nombre de 1 dans chaque rangée et chaque colonne de la matrice de contrôle de parité est faible par rapport à la longueur du bloc. La matrice suivante montre un exemple la matrice de contrôle de parité de code LDPC :

Exemple 1 :

$$\mathbf{H_1} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}_{4\times 6} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3.8)$$

Où $\widehat{m}$= 4 et n = 6 représentent le nombre d'équations de contrôle de parité et le nombre de bits dans le mot de code. En arrangeant la matrice de contrôle de parité en forme systématique en utilisant des opérations de rangées et de colonnes, nous pouvons obtenir la matrice

génératrice $\mathbf{G} = [\mathbf{P}^T_{(n-\hat{m})\times\hat{m}} \,|\mathbf{I}_{(n-\hat{m})}]$. Maintenant, le mot de code peut être généré en multipliant le message m par la matrice génératrice G.

$$\mathbf{c} = \mathbf{mG} \quad \text{………………….} \ (3.9)$$

À la sortie de décodage, le mot de code est dit valide si nous avons l'équation suivante (équation de syndrome) :

$$\mathbf{cH_1^T} = \mathbf{0} \quad \text{…………………..} \ (3.10)$$

Les codes LDPC sont des codes binaires construits sur la base des transformations linéaires définies par les matrices de contrôle de parité clairsemées. Connus comme étant quelques-uns des meilleurs codes correcteurs d'erreurs de nos jours, ces codes peuvent fournir un compromis raisonnable entre la simplicité et l'efficacité. Ces codes ont été proposés par Gallager, tandis que leurs performances et leurs capacités ont été analysées récemment par Richardson et Urbanke. En raison de la longueur des blocs et des calculs nécessaires pour le décodeur de codes LDPC, il était trop difficile pour les ordinateurs de l'époque d'explorer pleinement les performances de ces algorithmes. Enfin, il a été démontré par Mackay et Neal que les codes LDPC sont des codes qui fonctionnent près de la limite de Shannon. La matrice de contrôle de parité du code LDPC est appelée (w_c,w_r) régulière si chaque bit de code comprend un nombre fixe de bits de parité (w_c) et chaque équation de contrôle de parité contient un nombre fixe de bits de code (w_r). Les codes LDPC réguliers correspondent en fait à la variante étudiée à l'origine par Gallager, que l'on retrouve également dans les travaux de Mackay et Neal et Sipser et Spielman. La matrice $\mathbf{H_1}$ précédente montre un exemple de matrice de contrôle de parité régulière pour le code LDPC avec w_c= 2, w_r = 3.

Plus tard, les généralisations des codes LDPC de Gallager produisirent les codes LDPC irréguliers qui offrent certains avantages pratiques. Si les valeurs de w_r et w_r sont différentes entre les rangées et les colonnes de la matrice de contrôle de parité, respectivement, le code LDPC est dit irrégulier. La matrice $\mathbf{H_2}$ suivante montre une matrice de contrôle de parité irrégulière pour le code LDPC :

Exemple 2 :

$$\mathbf{H_2} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}_{4\times 6} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(3.11)}$$

Dans les codes LDPC irréguliers, les poids des rangées ou des colonnes de la matrice de parité sont identifiés par la distribution des degrés de nœuds de variable et des nœuds de parité. L'étude des graphes irréguliers par Luby et al a constitué l'une des grandes avancées conceptuelles. En particulier, les auteurs ont introduit la notion d'ensembles irréguliers en termes de ces distributions de degrés. L'avantage des codes LDPC irréguliers sur les codes Turbo est que la terminaison précoce du processus de décodage des codes LDPC peut être vérifiée à la sortie du décodeur. Cela signifie que si un mot de code valide est obtenu soit à la terminaison précoce du processus de décodage, soit à l'échec du décodage du code, il peut être connu. **[13]**

3.3. La forme ALIST

David Mackay, Matthew Davey et John Lafferty ont tous écrit des matrices de contrôle de parité à faible densité dans un format appelé alist.

La forme AList est basée sur une énumération explicite d'indices non nuls dans la matrice. Il contient de manière redondante des index basés sur des colonnes et des lignes. Tous les indices matriciels sont basés sur 1. Plus précisément, les quatres premières lignes sont les suivantes :

Avec :

n nombre de nœuds de variables (colonnes)

m nombre de nœuds de contrôle (lignes)

d^{v}_{max} degré de nœud de variable maximum (c'est-à-dire poids de colonne maximum)

d^c_{max} degré de nœud de contrôle maximum (c'est-à-dire poids de ligne maximum)

d^v_i ***i-ième*** degré de nœud de variable (poids de colonne)

d^c_j ***j-ième*** vérifier le degré du nœud (poids de la ligne)

Les *n* lignes suivantes contiennent une liste d'entrées non nulles des colonnes, qui est complétée à droite avec des zéros de telle sorte que le nombre d'entrées est d^v_{max} pour chacune de ces lignes. Ensuite, *m* lignes sont créées de la même manière pour les indices de ligne non nuls. **[W5]**

3.4. Conversion ALIST en VHDL BRAM

3.4.1 Définition du language VHDL

Le VHDL est un langage de description matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique indépendamment d'un fournisseur d'outils. Ainsi, techniquement, il est incontournable car c'est un langage puissant, moderne et qui permet une excellente lisibilité, une haute modularité et une meilleure productivité des descriptions. Il permet de mettre en œuvre les nouvelles méthodes de conception.

En outre, le développement du l'ensemble synthétisable du langage VHDL est de mieux en mieux défini. Cependant, la majorité des outils de synthèse sont compatibles avec cette norme, ce qui reflète un véritable standard pour la synthèse automatique avec le langage VHDL. **[14]**

3.4.2 Conversion ALIST en VHDL BRAM

Cette conversion est constituée par 3 étapes essentielles sont :

Alist vers Fichier txt, Fichier txt vers forme binaire et la dernière étape est la description VHDL. Comme il a montré dans la figure (3.1).

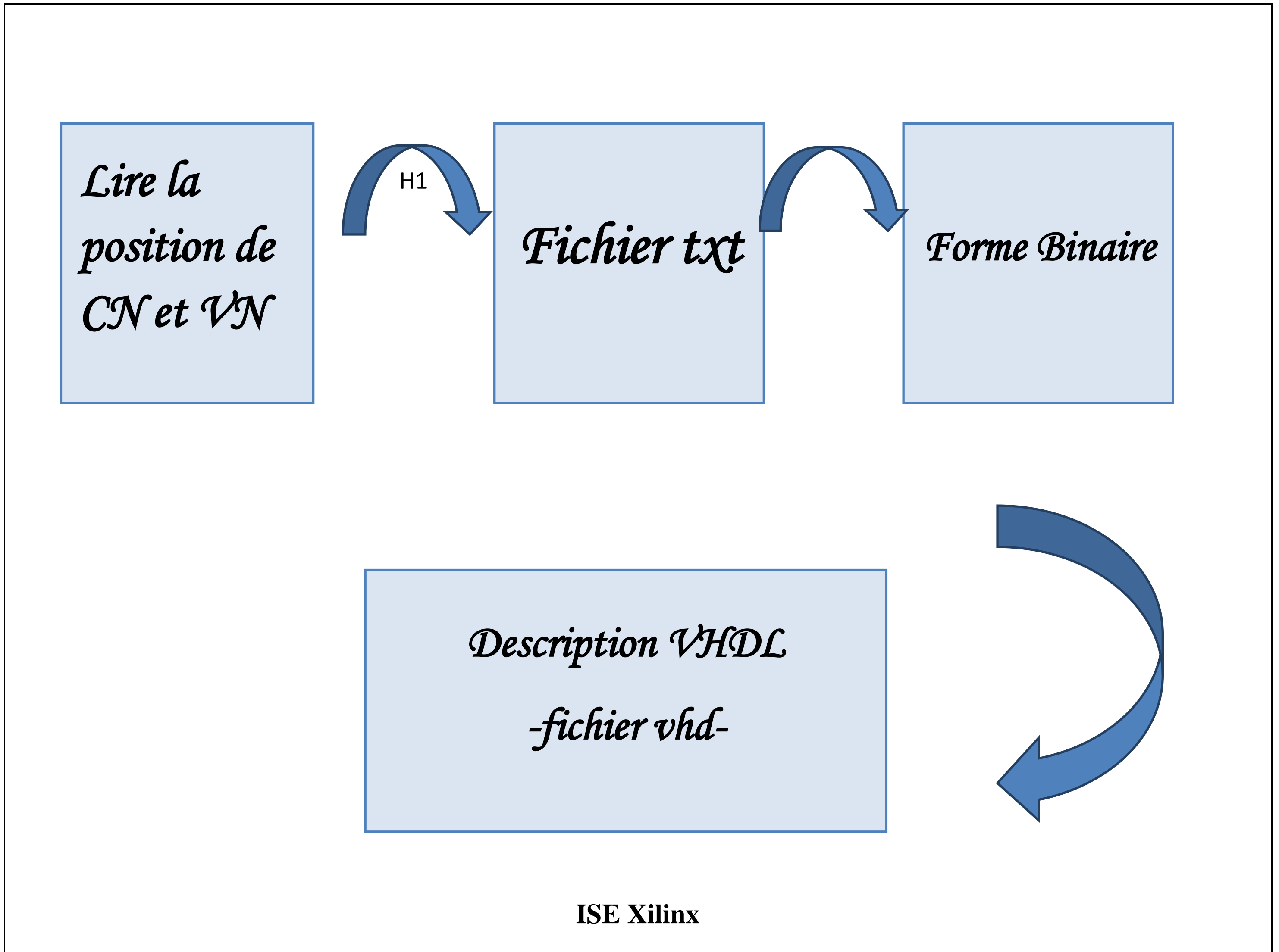


Fig. 3.1 -conversion la forme Alist en VHDL BRAM

1. Lire la position de CN et VN:

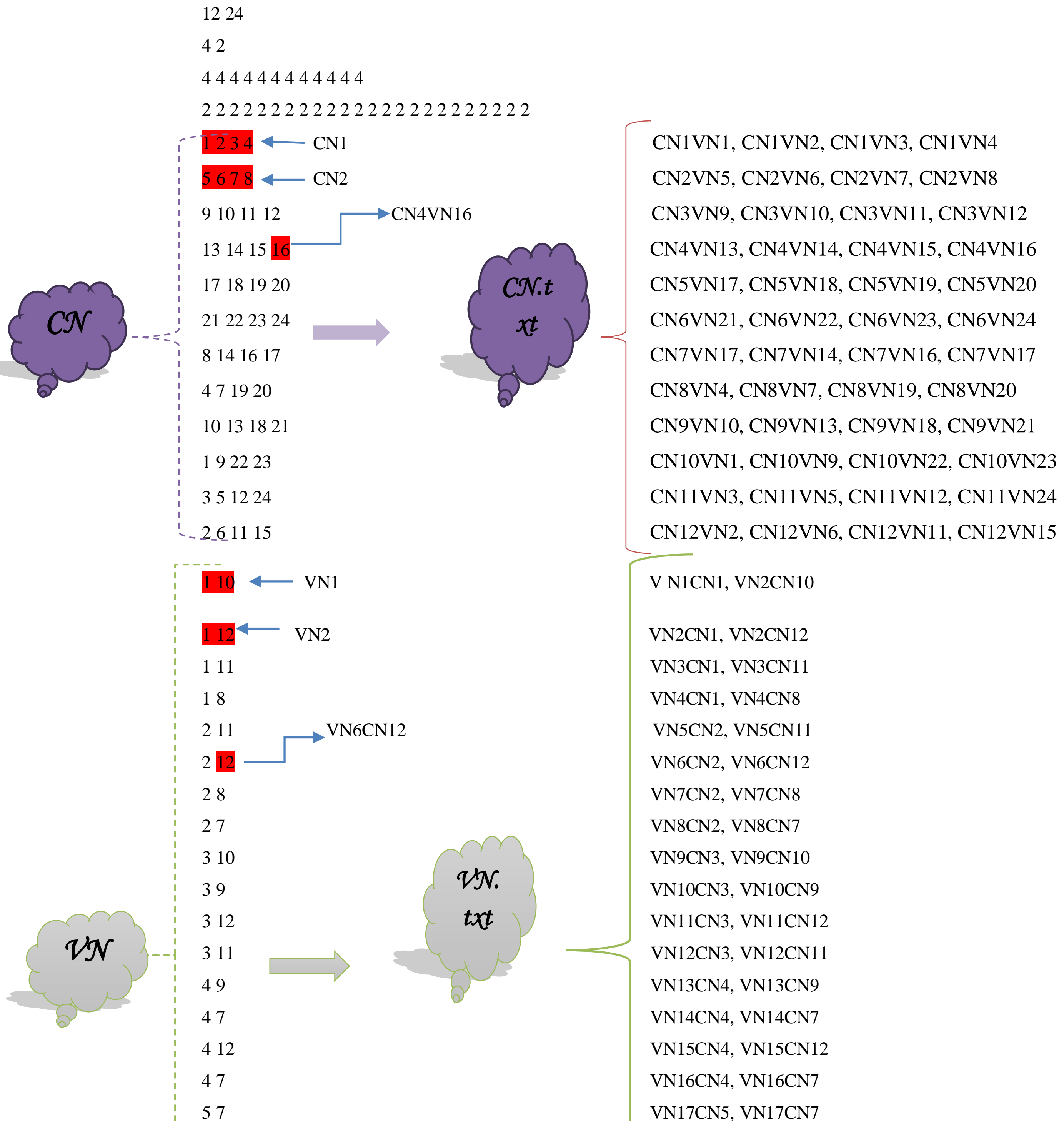
Nous avons choisi une matrice de parité sous forme « Alist » et on va lire les positions de CN et VN basé sous une programmation Matlab.

Exemple :

Par exemple, en considérant la matrice en (3.1), un fichier au format Alist pour un code LDPC (4, 2) régulier. Dans ce fichier, nous avons n = 12 et m = 24, puis d^{v}_{max} = 4 et d^{c}_{max} = 2, dans ses 3ème et 4ème lignes les degrés des n nœuds-variables et les m nœuds de contrôle, après la 1ére partie est CN matrix et la 2éme est VN matrix

- Dans CN matrix on doit fixer la ligne et lire chaque position de VN
- Dans VN matrix on doit fixer la ligne et lire chaque position de CN

Comme il a montré dans le schéma suivant :

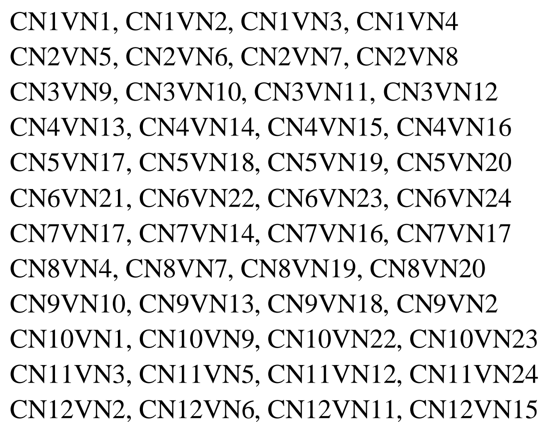


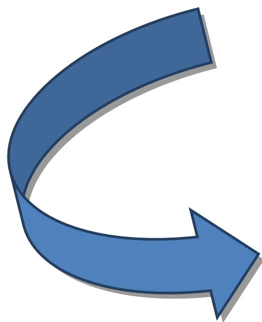
5 9	VN18CN5, VN18CN9
5 8	VN19CN5, VN19CN8
5 8	VN20CN5, VN20CN8
6 9	VN21CN6, VN21CN9
6 10	VN22CN6, VN22CN10
6 10	VN23CN6, VN23CN10
6 11	VN24CN6, VN24CN11

2. Conversion binaire

Notre but dans cette étape est de transformer les résultats précédents (CN.txt et VN.txt) vers une forme binaire (des 0 et 1), on obtient (bin.txt et bin1.txt)

CN.txt


CN1VN1, CN1VN2, CN1VN3, CN1VN4
CN2VN5, CN2VN6, CN2VN7, CN2VN8
CN3VN9, CN3VN10, CN3VN11, CN3VN12
CN4VN13, CN4VN14, CN4VN15, CN4VN16
CN5VN17, CN5VN18, CN5VN19, CN5VN20
CN6VN21, CN6VN22, CN6VN23, CN6VN24
CN7VN17, CN7VN14, CN7VN16, CN7VN17
CN8VN4, CN8VN7, CN8VN19, CN8VN20
CN9VN10, CN9VN13, CN9VN18, CN9VN2
CN10VN1, CN10VN9, CN10VN22, CN10VN23
CN11VN3, CN11VN5, CN11VN12, CN11VN24
CN12VN2, CN12VN6, CN12VN11, CN12VN15



'000001','000010','000011','000100'
'00101','000110','000111','001000'
'001001','001010','001011','001100'
'001101','001110','001111','010000'
'010001','010010','010011','010100'
'010101','010110','010111','011000'
‘001000','001110','010000','010001'
'000100','000111','010011','010100'
'001010','001101','010010','010101'
'000001','001001','010110','010111'
'000011','000101','001100','011000'
'000010','000110','001011','001111'

VN.txt

VN1CN1, VN2CN10

VN2CN1, VN2CN12
VN3CN1, VN3CN11
VN4CN1, VN4CN8
VN5CN2, VN5CN11
VN6CN2, VN6CN12
VN7CN2, VN7CN8
VN8CN2, VN8CN7
VN9CN3, VN9CN10
VN10CN3, VN10CN9
VN11CN3, VN11CN12
VN12CN3, VN12CN11
VN13CN4, VN13CN9
VN14CN4, VN14CN7
VN15CN4, VN15CN12
VN16CN4, VN16CN7
VN17CN5, VN17CN7
VN18CN5, VN18CN9
VN19CN5, VN19CN8
VN20CN5, VN20CN8
VN21CN6, VN21CN9
VN22CN6, VN22CN10
VN23CN6, VN23CN10
VN24CN6, VN24CN11

000001','001010','
000001','001100','
000001','001011','
000001','001000','
000010','001011','
000010','001100','
000010','001000','
000010','000111','
000011','001010','
000011','001001','
000011','001100','
000011','001011','
000100','001001','
000100','000111','
000100','001100','
000100','000111','
000101','000111','
000101','001001','
000101','001000','
000101','001000','
000110','001001','
000110','001010','
000110','001010','
000110','001011','

bin1.txt

3. Description VHDL BRAM

On va créer des mémoires RAMs sur FPGA qui contiennent des nombres binaires précédents basé sur le langage VHDL, on obtient les 2 résultats suivants :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;

entity rommlog is
    Port (  CLK : in  STD_LOGIC;
            CE : in  STD_LOGIC;
            EN : in  STD_LOGIC;
            ADDR1 : in  STD_LOGIC_VECTOR  ( 5 downto 0 );
            ADDR2 : in  STD_LOGIC_VECTOR  ( 5 downto 0 );
            DATA1 : out  STD_LOGIC_VECTOR ( 5 downto 0 );
            DATA2 : out  STD_LOGIC_VECTOR ( 5 downto 0 ));
end rommlog;

architecture Behavioral of rommlog is
type rom1_type is array ( 47 downto 0 ) of std_logic_vector ( 5 downto 0 );
type rom2_type is array ( 47 downto 0 ) of std_logic_vector ( 5 downto 0 );

signal ROM1 : rom1_type:= ( "000001" , "000010" ,"000011","000100","000101","000110","000111","001000","001001","001010","
signal rdata1 : std_logic_vector( 5 downto 0 );
signal ROM2 : rom2_type:= ( "000001","001010","000001","001100","000001","001011","000001","001000","000010","001011","000

signal rdata2 : std_logic_vector( 5 downto 0 );
```

Fig. 3.2 –Le code VHDL

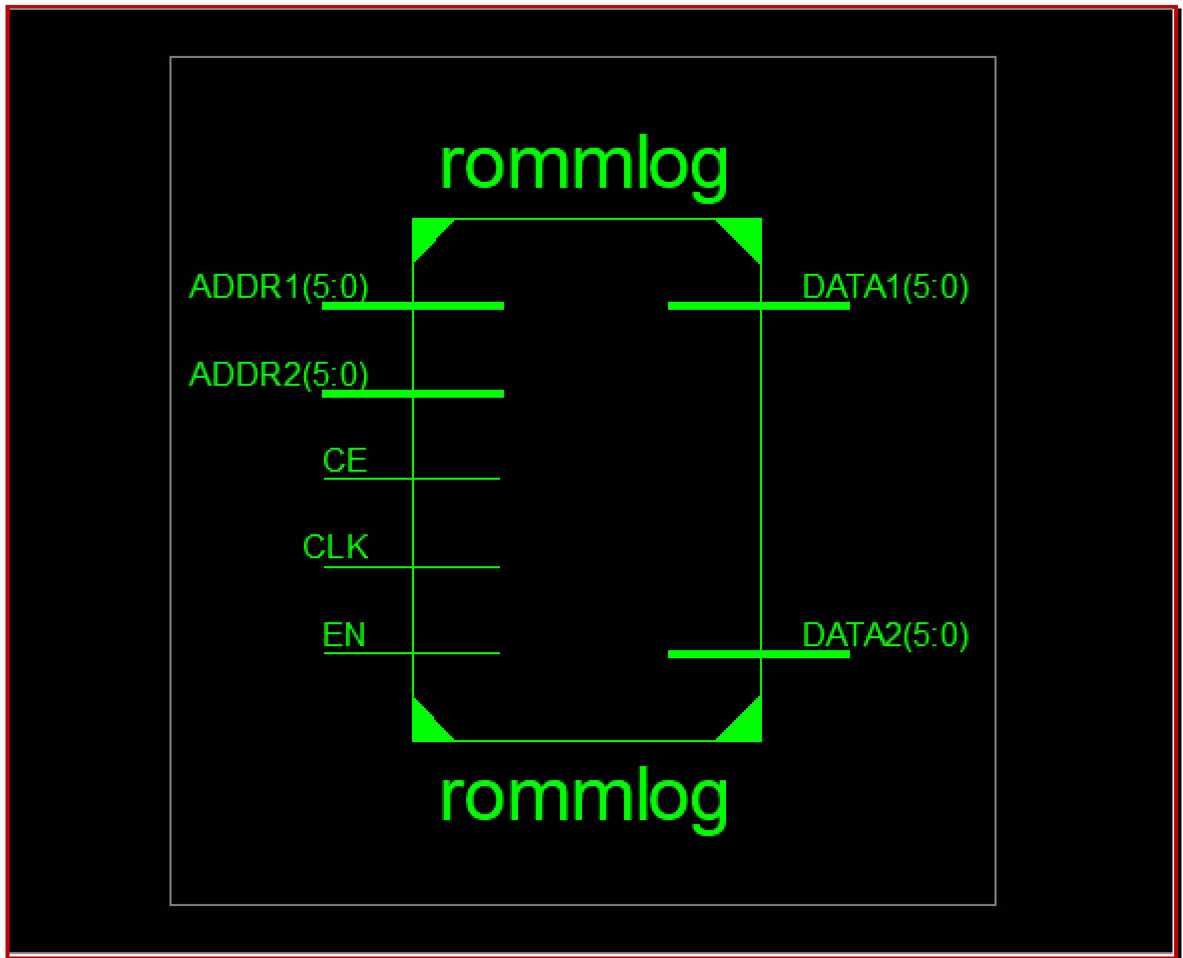


Fig. 3.3 –Le schéma RTL montre les entrées et sorties du circuit

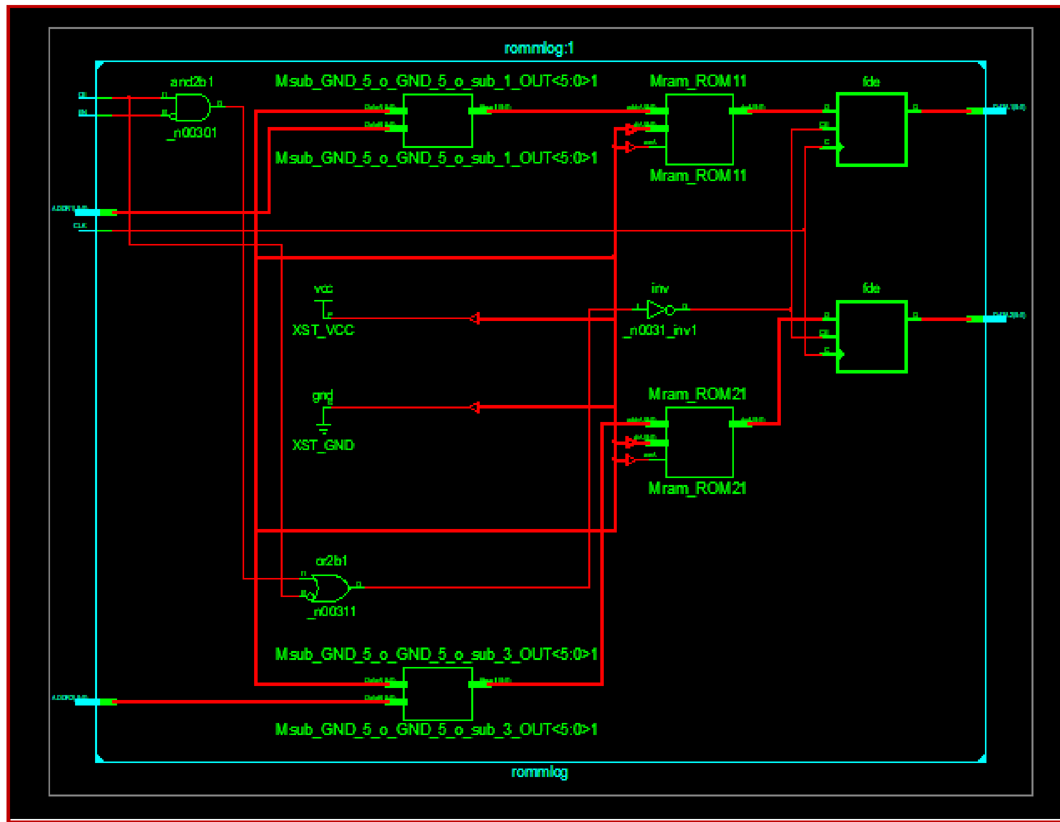


Fig. 3.4 –Le schéma RTL détaillé

3.5. Architecture des interconnexions sur FPGA

Ce réseau d'interconnexion permet de connecter une CLB avec une autre CLB ou avec une cellule d'entrée/sortie, et pour cela il existe un ensemble de lignes horizontales et verticales et un ensemble de points de connexion. On distingue plusieurs types de ligue qui sont définies par leur longueur relative et qui sont :

- Les interconnexions ou lignes segmentées à usage générale, de longueur la plus courte
- Les lignes directes ou interconnexions directe, de longueur double des lignes courtes.
- Les lignes longues.

Chaque CLB est entourée de ces lignes et des points de connexion et tout ça s'est détaillé sur les figures suivantes :

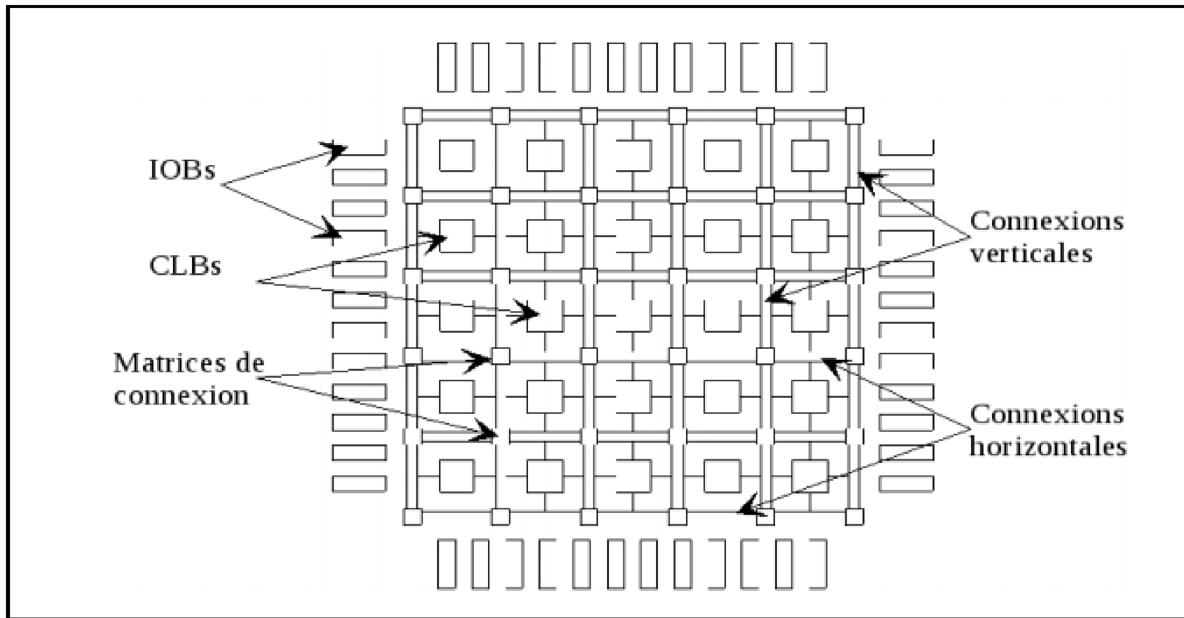


Fig. 3.5 -Concept architectural de base des FPGAs

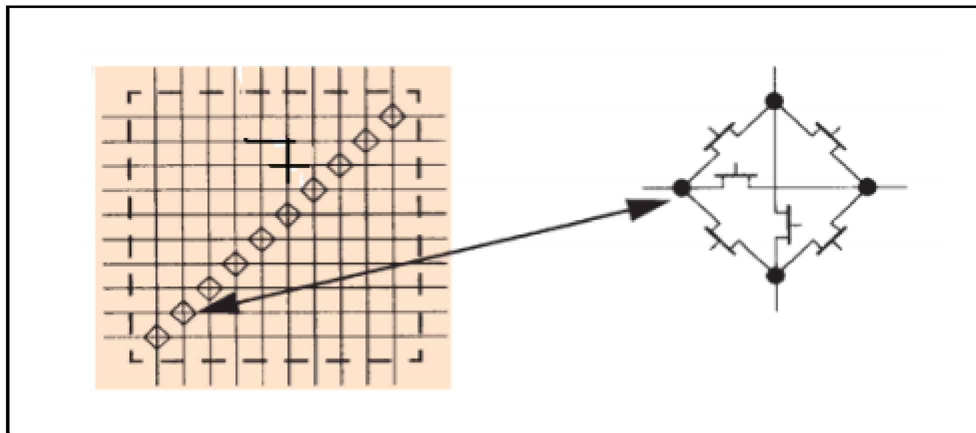


Fig. 3.6 -Les points d'interconnexion

- **Les interconnexions à usage générale :**

Sont des verticaux et horizontaux qui encadre chaque CLB et qui peuvent être reliés entre eux par une matrice de commutation car son rôle est de raccorder les segments entre eux selon diverse configuration.il assure aussi la commutation des signaux d'une voie sur l'autre.

- **Les interconnexions directes :**

Ces interconnexions permettent d'établir des chemins entre les CLBs adjacents et les cellules entrées/sorties avec un maximum d'efficacité en termes de vitesse et d'occupation de circuit.

- **Les longues lignes :**

Sont des longues lignes verticales et horizontales qui n'utilisent pas de matrice de commutation. Elles parcourent toutes la longueur et la largeur du circuit. Elles permettent aussi de transporter les signaux qui parcourent un long trajet. Elles égalisent les délais entre les signaux de façon à permettre un décalage minimum entre deux points distants de la ligne. Ces lignes conviennent pour transporter les signaux d'horloge. **[14]**

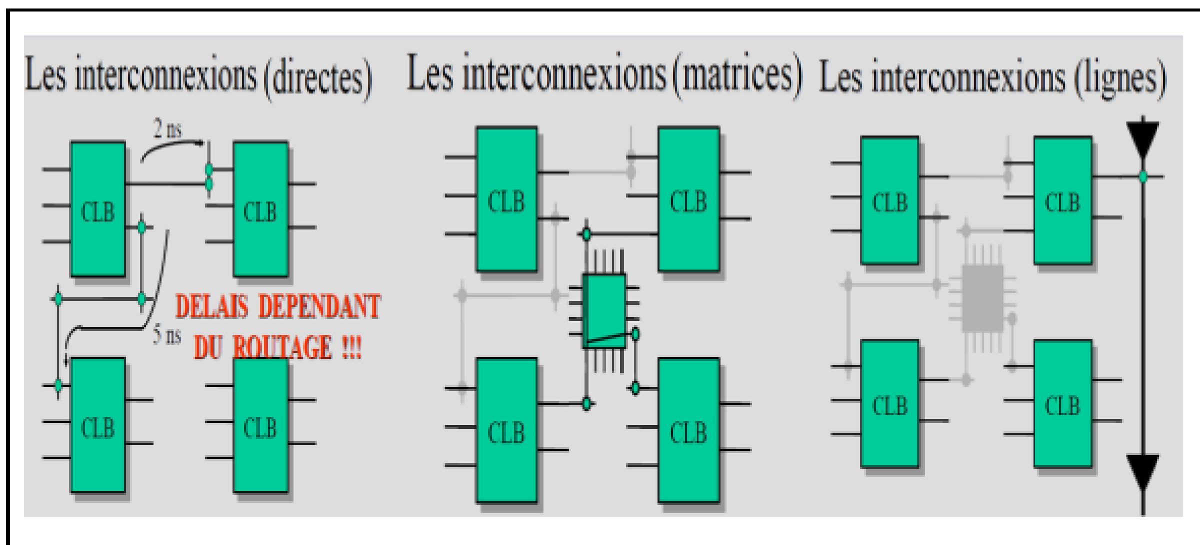


Fig. 3.7- Les différentes méthodes d'interconnexion entre blocs logiques

3.6. Conclusion

La fonction de « Alist format » est de stocker les matrices de contrôle de parité de Codes LDPC. Dans cette section, On a un but de dire qu'on peut créer des mémoires sur FPGA à

partir d'une description VHDL, commençant par un fichier au format alist, on a expliqué les étapes de cette conversion.

CHAPITRE 4

Simulations et résultats

4.1. Introduction

Au début des années 80, le département de la défense américaine (DOD) désire un système de description formelle et standard des circuits dans le cadre de son programme VHSIC, car à cette époque, les fournisseurs du DOD avaient chacun son propre HDL ce qui limitait l'échange des designs. C'est pourquoi, le DOD a décidé de définir un langage de spécification. Il a ainsi mandaté des sociétés pour établir un langage. Parmi ces langages proposés, le DOD a retenu le langage VHDL qui fut ensuite normalisé par IEEE (Institute of Electical and Electronics Engineers) en vue de satisfaire les objectifs suivants :

- La spécification par la description de circuits et de systèmes.

- La simulation afin de vérifier la fonctionnalité du système.

- La conception afin de tester une fonctionnalité identique mais décrite avec des solutions d'implémentations de différents niveaux d'abstraction.

En 1993, une nouvelle normalisation par l'IEEE du VHDL a permis d'étendre le domaine d'utilisation du VHDL vers :

- La synthèse automatique de circuit à partir des descriptions.

- La vérification des contraintes temporelles.

- La preuve formelle d'équivalence de circuits.

En 2001 nouvelle révision (VHDL 2001 ou IEEE 1076 – 2001).

En 2006 nouvelle version (VHDL 2006 ou IEEE 1076 – 2006).

Abbreviation VHDL signifier VHSIC Hardware Description Language (VHSIC: Very High-Speed Integrated Circuit). Ce langage a été écrit durant les années 70, par le département de la défense américaine destiné à modéliser les circuits intégrés complexes. Au début, ce langage était uniquement destiné à décrire les circuits intégrés déjà conçus et devait permettre de réaliser des documentations techniques facilement interprétables par certaines personnes. Aujourd'hui, la finalité de ce langage a bien changé, puisqu'il est essentiellement utilisé à

concevoir et modéliser les circuits, non plus dans un but de documentation, mais de simulation. Car on l'a étendu en lui rajoutant des extensions pour permettre la conception (synthèse) de circuits logiques programmables (P.L.D. Programmable Logic Device). **[14]**

Dans ce chapitre nous intéressont de convertir alist vers VHDL a but de réaliser des mémoires RAMs sur FPGA, en utilisant un fichier au format alist au début, Avant ça nous allons faire un rappel sur Matlab, Simulink,ISE Design Suite et System Generator. Nous terminons ce chapitre par les résultats de simulation sous Matlab et system generator.
Nous avons utilisé pour notre programmation la version **MATLAB2013a** et **XILINX** en particulièrement **ISE 14.7**.

➢ Description des logiciels de test :

▪ MATLAB/Simulink:

MATLAB qui développé par MathWorks est le logiciel. Les utilisateurs peuvent analyser données, peut développer des algorithmes et créer des modèles mathématiques. Il offre une large gamme de domaines d'utilisation. MATLAB est l'outil le plus pratique pour analyser les signaux numériques.
Simulink a également été développé par MathWorks et intégré à MATLAB. Il est logiciel de programmation graphique qui offre une conception au niveau du système, une simulation, génération automatique de code et test et vérification continus des systèmes intégrés. Il prend également en charge la conception matérielle basée sur un modèle à l'aide du générateur de système (Xilinx). **[15]**

▪ Xilinx :

Xilinx (nom complet Xilinx, Inc.) est une entreprise américaine de semi-conducteurs.
Inventeur du FPGA, Xilinx fait partie des plus grandes entreprises spécialisées dans le développement et la commercialisation de composants logiques programmables, et des services associés tels que les logiciels de CAO électroniques ; blocs de propriété intellectuelle réutilisables et formation. **[W6]**

▪ System generator :

Le générateur de système est un outil de conception DSP de Xilinx pour faire un lien entre ISE Design System et matlab qui permet d'utiliser MathWorks Model –Based Simulink pour la conception FPGA.

Xilinx System Generator, qui facilite la conception du matériel FPGA, a été le pionnier de l'idée de compiler un programme FPGA à partir de MATLAB et du modèle Simulink. Il offre modélisation de système et génération automatique de code à partir de MATLAB et Simulink. Lors de la conception du modèle Simulink, les blocs de générateurs du système peuvent être utilisés comme d'autres blocks Simulink. En outre, le concepteur peut également utiliser les deux blocs natifs Simulink et Xilinx System Generator bloque en même temps. Les blocs fournissent des abstractions de fonctions mathématiques, logiques, de mémoire et DSP pouvant être utilisées pour systèmes sophistiqués de traitement du signal. L'utilisateur peut concevoir du matériel en utilisant ces Blocs générateurs de système. **[15]**

- **Un fichier au format Alist :**

Un fichier au format Alist pour un code LDPC régulier (dv, dc) contient dans sa 1ère ligne la dimension de H, ensa 2ème ligne les valeurs dv et dc, dans ses 3ème et 4ème lignes les degrés des N nœuds-variables etles M nœuds de contrôle, respectivement. Ensuite, les lignes suivantes indiquent les positions des éléments non nulsdans chaque rangée de H. **[16]**

4.2. Plateforme de test sur MATLAB System Generator

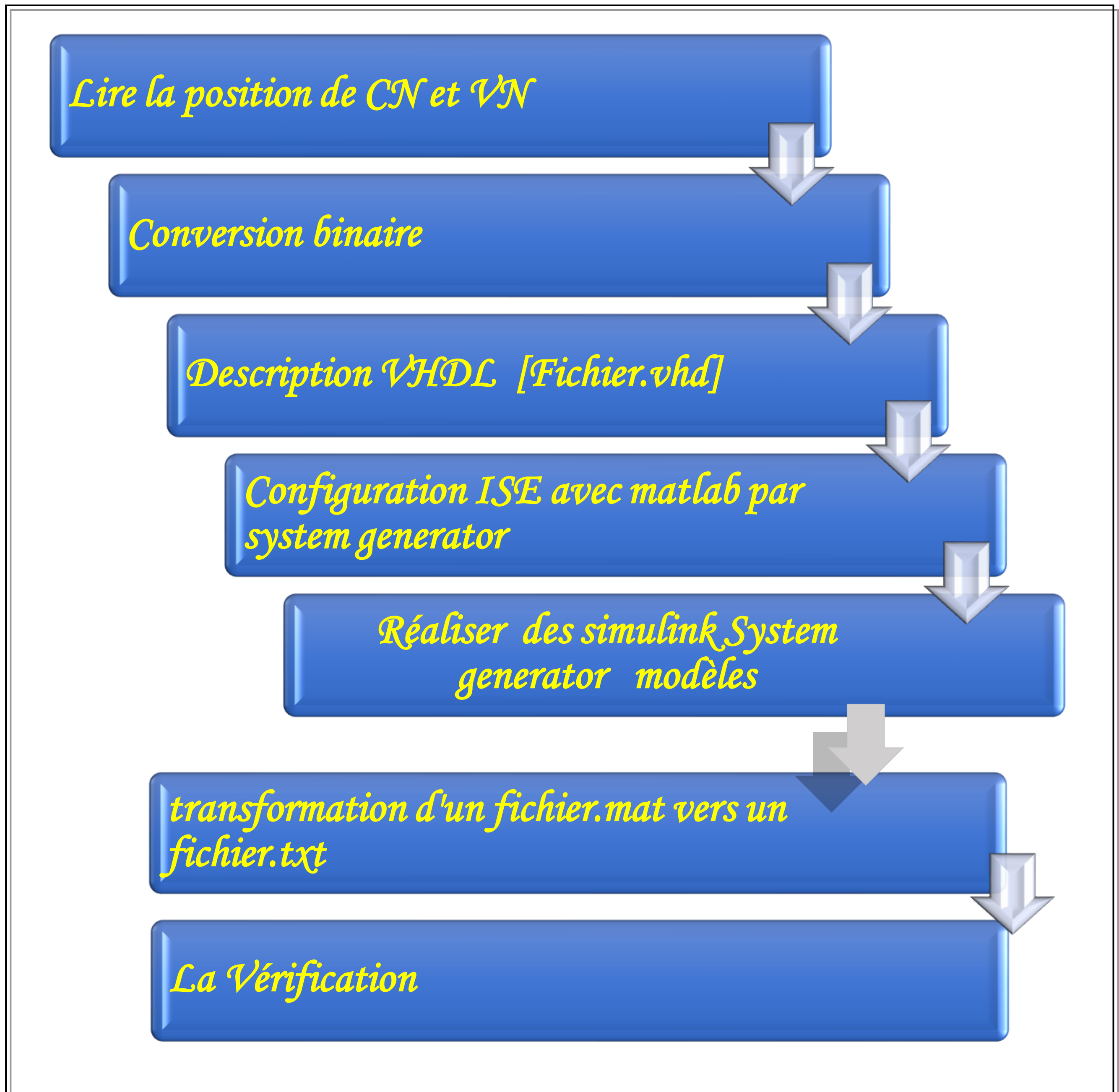


Fig.4.1 -Organigramme de notre projet

- Ses étapes présentes notre pratique dans ce mémoire et on a fait la même chose pour les 4 Normes (WIFI, WIMAX, WRAN, LAN/MAN)

- **Lire la position de CN et VN :**

Au début, nous avons choisi un fichier de forme "Alist", qui contient deux parties (check nodes CN et variables nodes VN), et on a nommé chacune d'elles (CN. matrix et VN. matrix). Après cela, nous avons créé un programme sous Matlab et le résultat sera deux fichiers (matrix.txt pour CN et matrix1.txt pour VN). Cette étape s'inscrit dans le cadre de la lecture des positions des nœuds de contrôle et nœuds de variables.

```
clc
clear all
matrix_CN = 'matrix_CN.txt';
matrix_VN = 'matrix_VN.txt';
CN=dlmread(matrix_CN);
VN= dlmread(matrix_VN);
[C,R] = size(CN);
[C1,R1] = size(VN);
file =fopen('matrix.txt','w');
for i=1:C
    for j=1:R
       if CN(i,j)==0

        else

              text=int2str(CN(i,j));
              ii = int2str(i);
              s = strcat('CN',ii,'VN',text,',');
              fprintf(file,s);
        end
    end
    fprintf(file,'\n')
end
file =fopen('matrix1.txt','w');
for i=1:C1
    for j=1:R1
        if VN(i,j)==0


        else
              text=int2str(VN(i,j));
              ii = int2str(i);
              s = strcat('VN',ii,'CN',text,',');
              fprintf(file,s);
        end



    end
    fprintf(file,'\n')
end

fclose(file);
```

Fig. 4.2 -Programmation de alist vers un fichier txt.

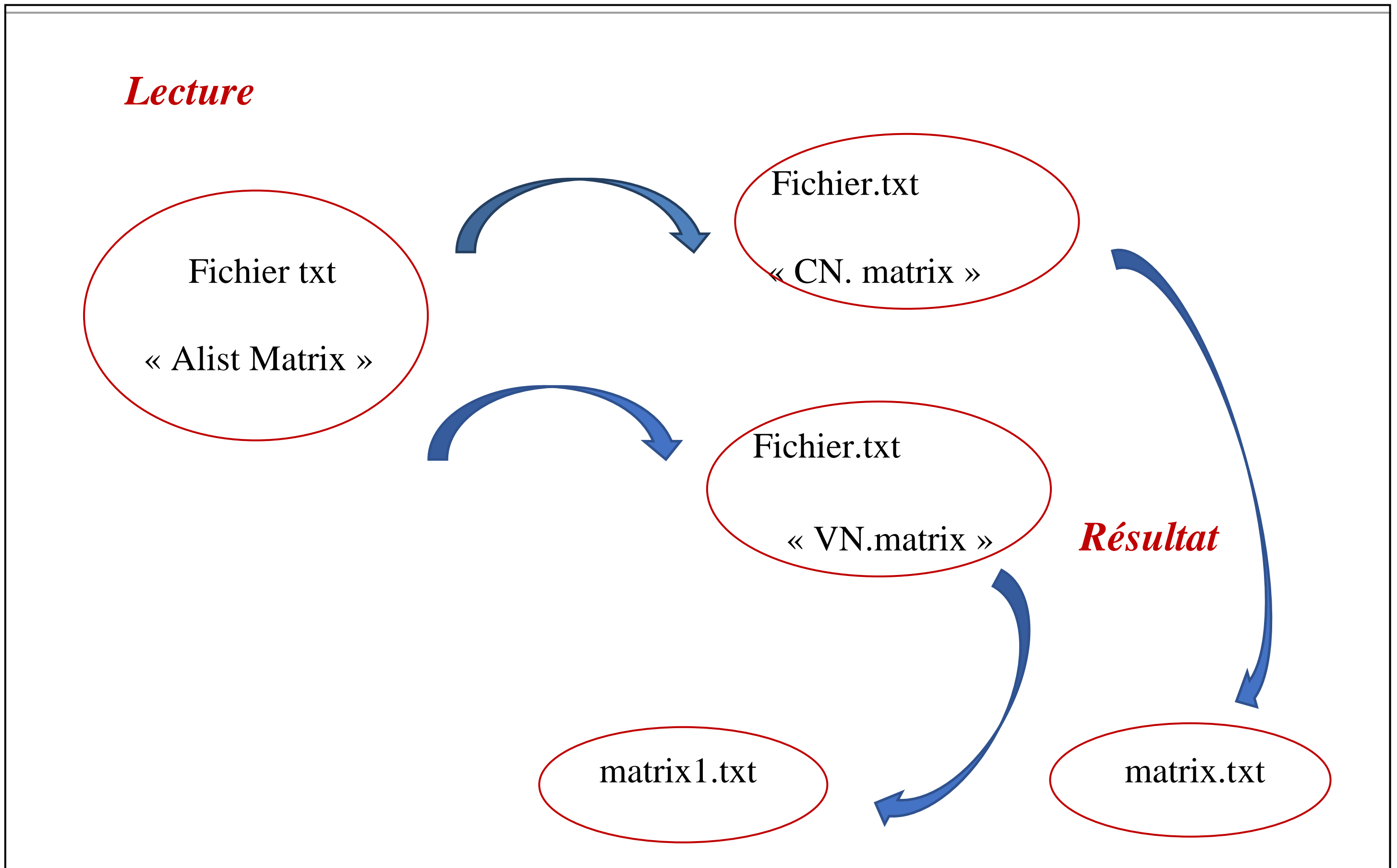


Fig. 4.3 -Un schéma explicatif pour cette étape.

- **Conversion Binaire :**

À la suite, avec un programme Matlab aussi. On va transformer le résultat obtenu précédent en conversion binaire pour obtenir deux fichiers txt (un fichier txt pour la partie de nœud de contrôle (CN) et un fichier txt pour la partie de nœud de variable (VN)), cette conversion binaire contient des 0 et des 1, et le nombre de ces derniers est lié au nombre de bit de notre "alist format.

```
clear all
matrix_CN = 'matrix_CN.txt';
matrix_VN = 'matrix_VN.txt';
CN=dlmread(matrix_CN);
VN= dlmread(matrix_VN);
[C,R] = size(CN);
[C1,R1] = size(VN);
file =fopen('bin.txt','w');
for i=1:C
    for j=1:R

              text=dec2bin(CN(i,j),nombre de bit);
              ii = dec2bin(i);
              s = strcat(text,'","');
              fprintf(file,s);

    end

end
file =fopen('bin1.txt','w');
for i=1:C1
    for j=1:R1

           text=dec2bin(VN(i,j),nombre de bit);
              ii = int2str(i);
              s = strcat(text,'","');
              fprintf(file,s);
        end
    end
fclose(file);
```

Fig. 4.4 -La programmation pour la conversion binaire.

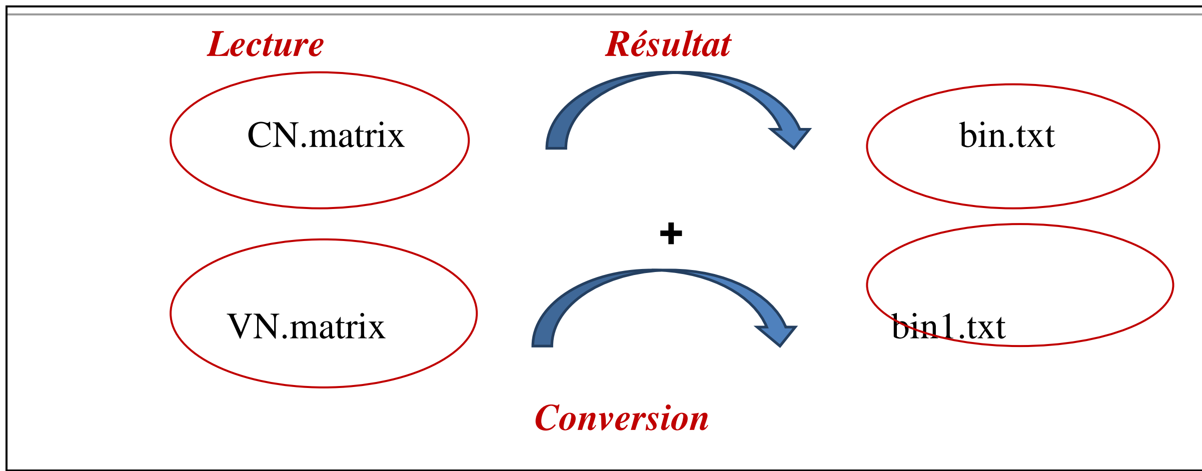


Fig. 4.5 -Schéma explicatif pour l'étape de la conversion binaire.

- **Description VHDL :**

A cette étape, on va créer deux RAM initialisées (RAM1 et RAM2) une pour CN et l'autre pour VN, la RAM initialisée c'est une RAM qui a des valeurs au début, veut dire que on va décrit les valeurs initiales en description VHDL ainsi que les entrées "CLK, CE, EN, ADDR1et ADDR2) et les sorties" DATA1, DATA2" utilisées.

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use ieee.std_logic_unsigned.all;
```

→ *Library*

```
entity rommlog is
 Port (CLK:in STD_LOGIC;
CE:in STD_LOGIC;
EN:in STD_LOGIC;
          ADDR1:in STD_LOGIC_VECTOR ();
          ADDR2:in STD_LOGIC_VECTOR ();
          DATA1:out STD_LOGIC_VECTOR ();
  DATA2:out STD_LOGIC_VECTOR ());
end rommlog;
```

```
architecture Behavioral of rommlog is
type rom1_type is array () of std_logic_vector ();
type rom2_type is array () of std_logic_vector();
signal ROM1: rom1_type: = ()
signal rdata1: std_logic_vector ();
signal ROM2: rom2_type: = ()
signal rdata2: std_logic_vector ();
begin
rdata1 <= ROM1(conv_integer(n-ADDR1));
rdata2 <= ROM2(conv_integer(n-ADDR2));
process (CLK)
   begin
```

```
if (CLK'event and CLK = '1') then
        if (CE = '1') then
  if (EN = '1') then
    DATA1 <= rdata1;
DATA2 <= rdata2;
  end if;
end if;
end if;
 end process;
end Behavioral;
```

Fig. 4.6 -Le code VHDL.

- **Configuration ISE avec Matlab par system generator**

Pour utiliser system generator il faut installer les logiciels Matlab et ISE Design Suite compatible , MATLAB devrait être installé en premier, puis Xilinx doit être installé(notre cas ISE design suite 14.7 compatible avec Matlab 2013a) puis les configurer on suivant les étapes de configurations montrées ci –dessous.

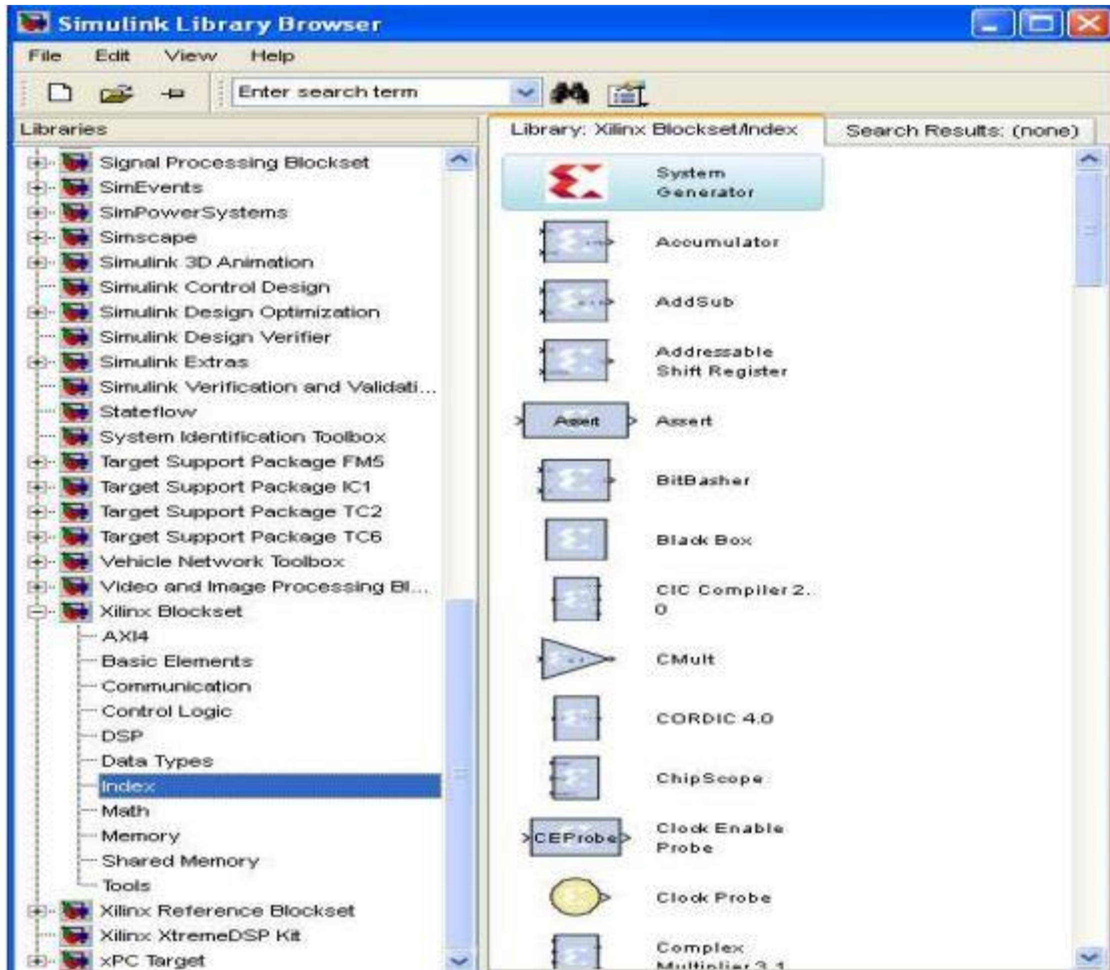
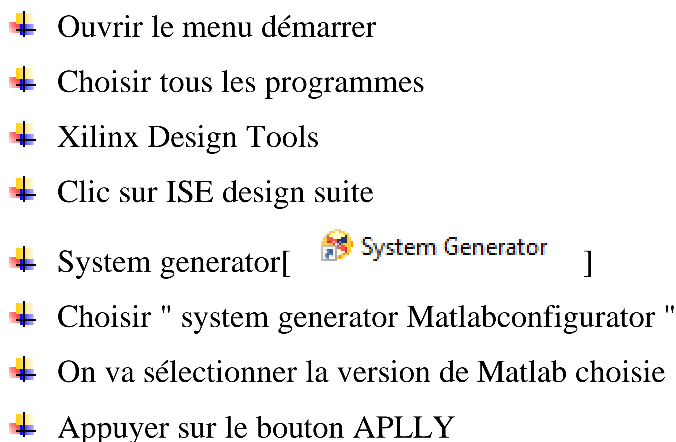


Fig. 4.7 -Les blocs de Xilinx pour Simulink

Pour trouver ces blocs on va configurer notre Matlab à l'aide de ISE Design Suite retraçant les étapes suivantes :



- Ouvrir le menu démarrer
- Choisir tous les programmes
- Xilinx Design Tools
- Clic sur ISE design suite
- System generator[System Generator]
- Choisir " system generator Matlabconfigurator "
- On va sélectionner la version de Matlab choisie
- Appuyer sur le bouton APLLY

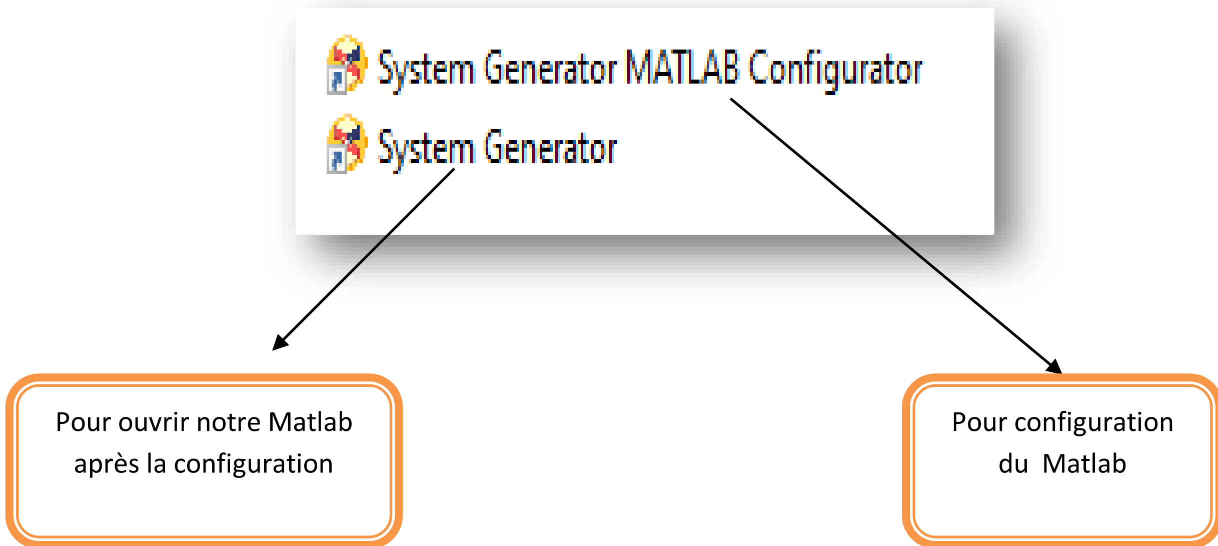
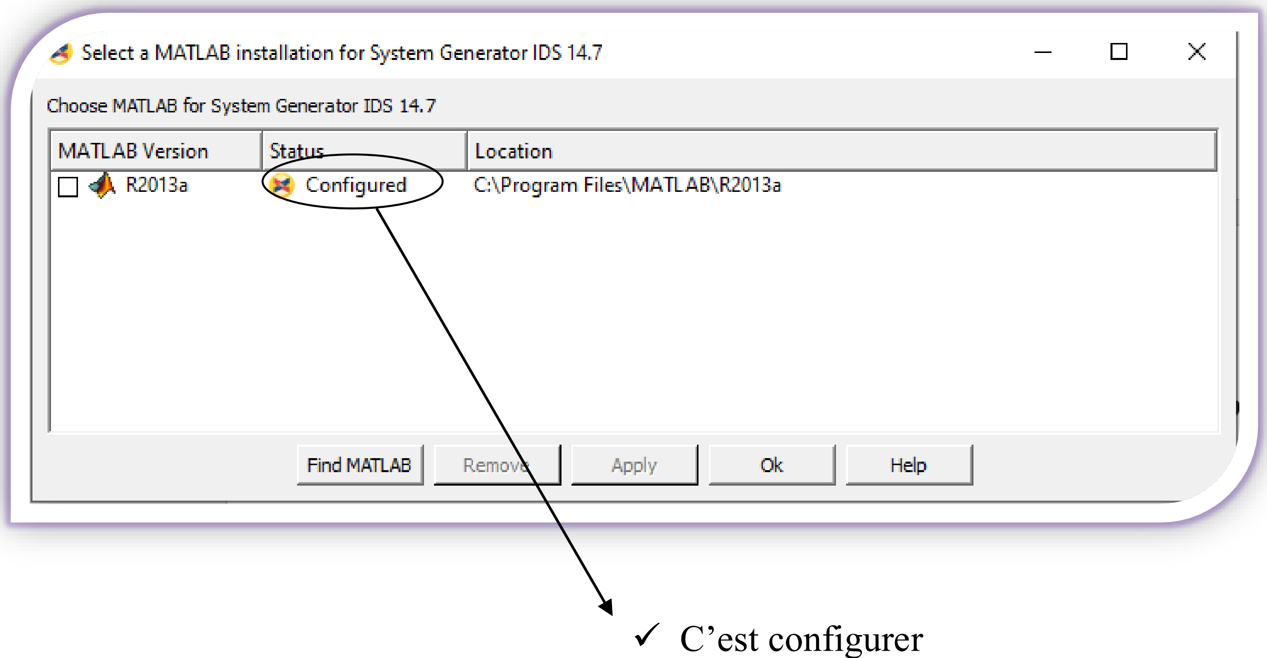


Fig. 4.8 –Etape de configuration de Matlab pour utiliser le system générator



✓ C'est configurer

Fig. 4.9 -La fenêtre de configuration Matlab.

Après la configuration en utilisant le system generator pour ouvrir notre Matlab configuré

- ***Simulink system generator modèles:***

Après la configuration on a réalisé des modèles qui lisent notre Alist et affiche des résultats sous forme fichier dans Workspace.

On a utilisé :

- System generator
- Counter limited
- Gateway in
- Constant
- Gateway out
- To file

Blocs de générateur de système Xilinx :

- **Constant**

Le bloc Xilinx Constant génère une valeur constante. Ces valeurs peuvent être une valeur fixe, une valeur booléenne. Si l'on compare avec le bloc constant Simulink, il est similaire à ce bloc, mais peut être utilisé pour piloter directement les entrées. **[15]**

Fig. 4.10 -Xilinx System Generator - Constant Block Symbol

- **Gateway In**

Le bloc Xilinx Gateway In est l'entrée dans la partie Xilinx FPGA de votre conception Simulink. Il convertit l'entrée double précision Simulink en le type de point fixe System Generator et définit un port d'entrée pour le niveau supérieur de la conception HDL générée par System Generator. **[15]**

Fig. 4.11 -Xilinx System Generator - Gateway In Block Symbol

- **Gateway Out**

Les blocs Xilinx Gateway Out sont les sorties de la partie Xilinx de votre conception Simulink. Ce bloc convertit le type de données en virgule fixe ou en virgule flottante de System Generator en un type de données Simulink entier, simple, double ou en virgule fixe. **[15]**

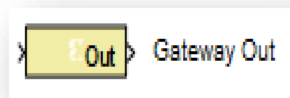


Fig. 4.12 -Xilinx System Generator - Gateway Out Block Symbol

- **System Generator**

Le jeton du générateur de système sert de panneau de commande pour contrôlerles paramètres du système et les paramètres de simulation. Il est également utilisé pour la génération de code. Chaque modèle Simulink contenant Xilinx Blockset doit contenir un bloc System Generator. Une fois ce dernier ajouté à un modèle, il est possible de spécifier la manière dont la génération de code et de simulation doit être manipulée. **[15] [17]**

Fig. 4.13 -Le bloc System Generator

- **Black box**

Le bloc black box permet d'incorporer du matériel modèle de langage de description (HDL) dans System Generator. Ce bloc est utilisé pour spécifier à la fois le comportement de simulation dans Simulink et le fichier d'implémentation à utiliser lors de la génération de code avec System generator. Les ports de black box produisent et consomment les mêmes types de signaux comme d'autres blocs de générateur de système. Lorsque le bloc black est traduite en matériel, le l'entité HDL associée est automatiquement incorporée et câblée à d'autres blocs dans la conception résultante.

Black box peut être utilisé pour incorporer VHDL ou Verilog dans un modèle Simulink.

Il peut être Co-simulé avec Simulink en utilisant l'interface du générateur de système pour le simulateur Vivado.En plus d'incorporer le HDL dans un modèle de générateur de système, il peut être utilisé pour définir l'implémentation associée à un modèle de simulation externe.

[W7]

Fig 4.14 -Le bloc black box

- **To file**

Le bloc To File écrit les données de signal d'entrée dans un fichier MAT. Le bloc écrit dans le fichier de sortie de manière incrémentielle, avec une surcharge de mémoire minimale pendant la simulation. Si le fichier de sortie existe au démarrage de la simulation, le bloc écrase le fichier. Le fichier se ferme automatiquement lorsque vous interrompez la simulation ou que la simulation est terminée. Si la simulation se termine anormalement, le bloc To File enregistre les données qu'il a enregistrées jusqu'au moment de l'arrêt anormal.

L'icône de bloc Vers fichier affiche le nom du fichier de sortie. **[W8]**

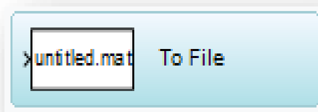


Fig. 4.15 -Le bloc To file

- **Counter limited**

Le bloc Counter Limited compte jusqu'à ce que la limite supérieure spécifiée soit atteinte. Ensuite, le compteur revient à zéro et redémarre le comptage. Le compteur s'initialise toujours à zéro. **[W9]**

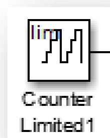


Fig.4.16 -Le bloc counter limited

La figure 4.16 montre la réalisation de notre modèle et les blocs utilisés.

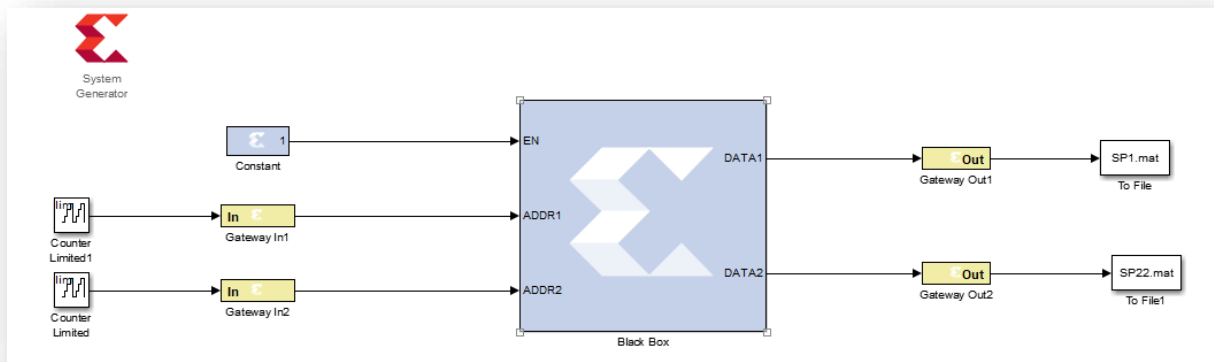


Fig. 4.17 -Le modèle réalisé

- **Transformation d'un fichier.mat vers un fichier.txt**

Par la suite, pour l'obtention du fichier .txt .il faut transformer le résultat obtenu par le Simulink modèle précédent (est sous forme d'un fichier.mat) avec une programmation Matlab. Puis on a fait un prg pour affiche ce fichier.txt avec la même forme de notre alist.

- **La vérification :**

Cette dernière étape introduit notre but de ce mémoire. Finalement, on va créer un prg pour vérifier alist et notre résultat obtenu. Une fois les résultats confirmés et les deux derniers sont identiques donc nous avons atteint notre objectif de travail.

4.3. Simulations et résultats

4.3.1. WIFI (802.11)

Dans cette norme on a choisi «Alist Matrix » avec ces dimensions :

- n=648
- m=108
- dvmax=4

- dcmax=22

```
648 108
4 22
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 2
22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 2
11 52 60 102
12 53 61 103
13 54 62 104
14 28 63 105
15 29 64 106
16 30 65 107
17 31 66 108
18 32 67 82
19 33 68 83
20 34 69 84
21 35 70 85
22 36 71 86
23 37 72 87
24 38 73 88
25 39 74 89
26 40 75 90
27 41 76 91
1 42 77 92
2 43 78 93
3 44 79 94
4 45 80 95
5 46 81 96
6 47 55 97
7 48 56 98
8 49 57 99
9 50 58 100
10 51 59 101
15 43 66 102
16 44 67 103
17 45 68 104
18 46 69 105
```

Fig. 4.18 –"Alist Matrix" de la norme WIFI (802.11)

- Au début on lit la position de CN et VN et affiche deux fichier txt comme on a expliqué précédemment puis dans la conversion binaire et à partir de cette alist notre nombre de bit est 12 bits dans les deux parties CN et VN.
- La description VHDL : on a nommé le fichier « **PFEF.vhd** », la figure suivante présente notre code vhdl dans cette norme :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;

entity PFEF is
    Port ( CLK : in  STD_LOGIC;
           CE : in  STD_LOGIC;
           EN : in  STD_LOGIC;
           ADDR1 : in  STD_LOGIC_VECTOR (11 downto 0);
           ADDR2 : in  STD_LOGIC_VECTOR (11 downto 0);
           DATA1 : out  STD_LOGIC_VECTOR(11 downto 0);
           DATA2 : out  STD_LOGIC_VECTOR(11 downto 0));
end PFEF;

architecture Behavioral of PFEF is
type rom1_type is array (2591 downto 0 ) of std_logic_vector ( 11 downto 0 );
type rom2_type is array (2375 downto 0 ) of std_logic_vector ( 11 downto 0 );
signal ROM1 : rom1_type:=("000000001011","000000110100","000000111100","000001100110","00000000110
signal rdata1 : std_logic_vector( 11 downto 0 );
signal ROM2 : rom2_type:=("000000010010","000000101001","000000111111","000001100111","00000111011

signal rdata2 : std_logic_vector( 11 downto 0 );
begin
rdata1 <= ROM1(conv_integer(2591-ADDR1));
rdata2 <= ROM2(conv_integer(2375-ADDR2));
process (CLK)
    begin
        if (CLK'event and CLK = '1') then
        if (CE = '1') then
            if (EN = '1') then
                DATA1 <= rdata1;
   DATA2 <= rdata2;
            end if;
        end if;
        end if;
          end process;

end Behavioral;
```

Fig. 4.19 -Le code VHDL pour la norme WIFI (802.11)

Dans ce code on a 2 RAMs

- RAM1 qui avait une entrée ADDR1 et sortie DATA1
- RAM2 qui avait une entrée ADDR2 et sortie DATA2

Le nombre de bits est extrait de la manière suivante :

- ✓ CN (648*4=2592= 2^12)
- ✓ VN (108*22=2576=2^12)

✓ Ensuit on a réalisé le modèle avec simulink system generator (Fig. 4.18), on a nommé le fichier : « **VV1** ».

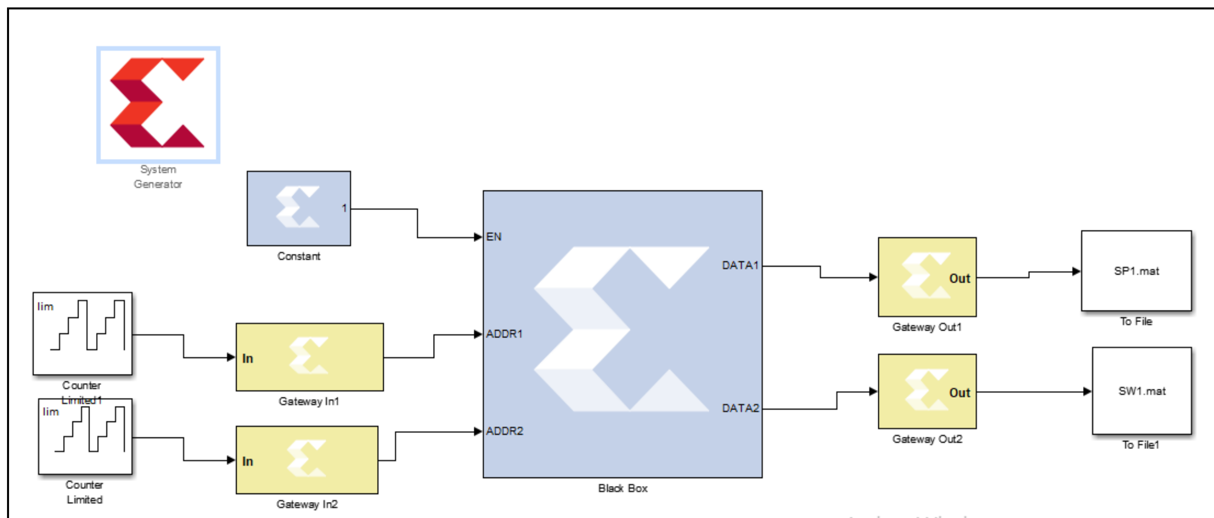


Fig. 4.20 -Simulink system generator modèl pour la norme WIFI (802.11)

✓ **Résultat :**

On obtient 2 fichiers [SP1.mat] pour CN et [SW1.mat] pour VN.

➢ D'après le résultat obtenu on va transformer les deux fichiers précédents vers un fichier txt, la figure suivante montre le programme Matlab de transformation :

```
clc
clear all
load SP1.mat;
dlmwrite('SP11.txt',VAR);
load SW1.mat
dlmwrite('SW22.txt',VAR1);
```

Fig 4.21 -La programmation Matlab

✓ **Résultat :**

On obtient deux fichiers txt nommé [SP11.txt] et [SW22.tx

➢ Ensuite on va les modifiés :

```
clc
clear all
z1='SP11.txt';
z2='SW22.txt';
z=dlmread(z1);
a=dlmread(z2);
file =fopen('SP112.txt','w');
for i=1:648
    for j=1:4
        k=((i-1)*4)+j+1;

       text=int2str(z(2,k));

       fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
file =fopen('SW221.txt','w');
for i=1:108
    for j=1:22
        k=((i-1)*22)+j+1;

       text=int2str(a(2,k));

       fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
```

Fig. 4.22- Programme Matlab

✓ **Résultat :**

Après cette modification on avoirs deux fichiers qui nommées [SP112.txt] et [SW221.txt] avec la forme « Alist »

- Enfin on va faire la vérification entres les deux fichiers précédents obtenus avec notre alist

```
clc
clear all
matrix_CN = 'wifiCN.txt';
matrix_VN='wifiVN.txt';
CN=dlmread(matrix_CN);
VN=dlmread(matrix_VN);
z1='SP112.txt';
z2='SW221.txt';
z=dlmread(z1);
a=dlmread(z2);
if z==CN
    disp('right')
else
    disp('wrong')
end
 if   a==VN
    disp('right')
else
    disp('wrong')
end
```

Fig. 4.23 -Programmation Matlab

- ✓ **Résultat :**

Les deux fichiers sont identiques donc on a réalisé notre but.

4.3.2. WIMAX (802.16)

Les dimensions de « Alist Matrix » sont :

- n=576
- m=288
- dvmax=6
- dcmax=7

```
576 288
6 7
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6
82 214 279 0 0 0
83 215 280 0 0 0
84 216 281 0 0 0
85 193 282 0 0 0
86 194 283 0 0 0
87 195 284 0 0 0
88 196 285 0 0 0
89 197 286 0 0 0
90 198 287 0 0 0
91 199 288 0 0 0
92 200 265 0 0 0
93 201 266 0 0 0
94 202 267 0 0 0
95 203 268 0 0 0
96 204 269 0 0 0
73 205 270 0 0 0
74 206 271 0 0 0
75 207 272 0 0 0
76 208 273 0 0 0
```

Fig .4.24 –"Alist Matrix" de la norme WIMAX (802.16)

- On va lit la position de CN et VN et affiche deux fichiers txt comme on a expliqué précédemment puis dans la conversion binaire et à partir de cette alist, le nombre de bits est 12 bits pour CN et 11 bits pour VN.
- La description VHDL : on a nommé le fichier « **PFEX.vhd»**, la figure suivante présente le code vhdl dans cette norme :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;

entity PFEF is
    Port ( CLK : in  STD_LOGIC;
           CE : in  STD_LOGIC;
           EN : in  STD_LOGIC;
           ADDR1 : in  STD_LOGIC_VECTOR (11 downto 0);
           ADDR2 : in  STD_LOGIC_VECTOR (11 downto 0);
           DATA1 : out  STD_LOGIC_VECTOR(11 downto 0);
           DATA2 : out  STD_LOGIC_VECTOR(11 downto 0));
end PFEF;

architecture Behavioral of PFEF is
type rom1_type is array (2591 downto 0 ) of std_logic_vector ( 11 downto 0 );
type rom2_type is array (2375 downto 0 ) of std_logic_vector ( 11 downto 0 );
signal ROM1 : rom1_type:=("000000001011","000000110100","000000111100","000001100110","000
signal rdata1 : std_logic_vector( 11 downto 0 );
signal ROM2 : rom2_type:=("000000010010","000000101001","000000111111","000001100111","000

signal rdata2 : std_logic_vector( 11 downto 0 );
begin
rdata1 <= ROM1(conv_integer(2591-ADDR1));
rdata2 <= ROM2(conv_integer(2375-ADDR2));
process (CLK)
    begin
        if (CLK'event and CLK = '1') then
        if (CE = '1') then
            if (EN = '1') then
                DATA1 <= rdata1;
   DATA2 <= rdata2;
            end if;
        end if;
        end if;
          end process;

end Behavioral;
```

Fig. 4. 25 -Le code VHDL pour la norme WIMAX (802.16)

On a 2 RAMs :

- RAM1 qui avait une entrée ADDR1 et sortie DATA1
- RAM2 qui avait une entrée ADDR2 et sortie DATA2

Le nombre de bitest extrait de la manière suivante :

✓ CN (576*6=3456= 2^12)

✓ VN (288*7=2016=2^11)

- Ensuite, la figure suivante montre le modèle avec simulink system generator .on a nommé le fichier : « **CC1** ».

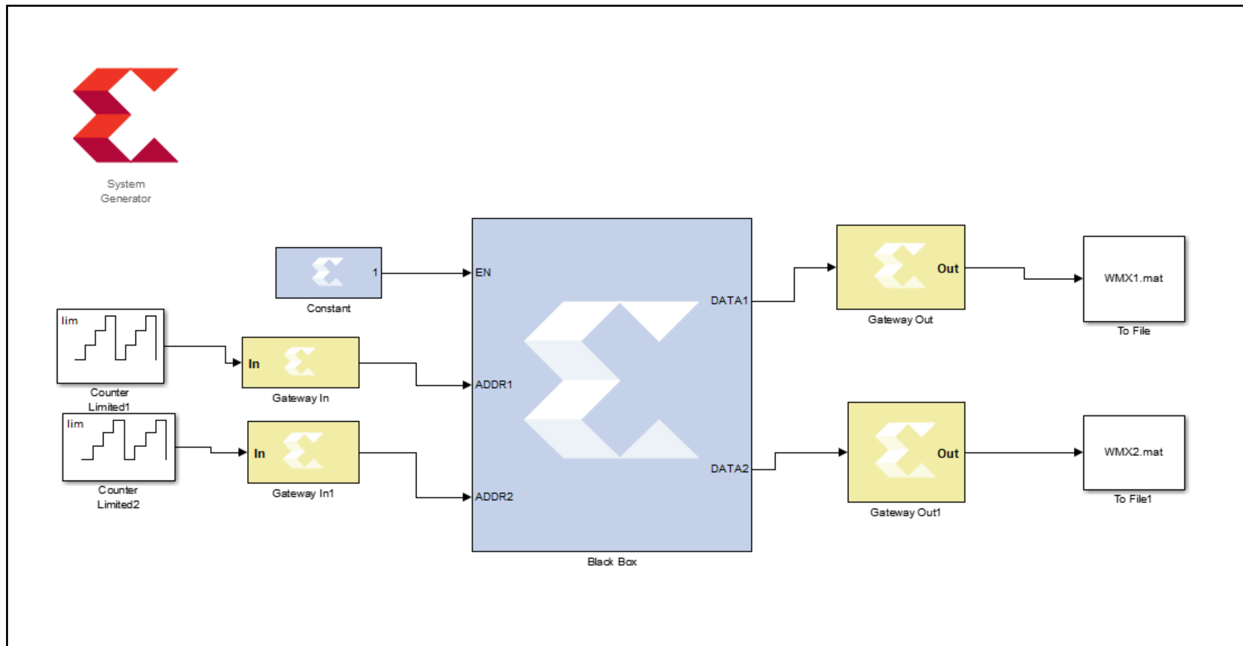


Fig. 4.26 -Simulink system generator modèle pour la norme WIMAX (802.16)

✓ Résultat :

On obtient 2 fichiers [WMX1.mat] pour CN et [WMX2.mat] pour VN.

- D'après le résultat obtenu on va transformer les deux fichiers précédents vers un fichier txt, la figure suivante montre le programme Matlab de transformation :

```
clc
clear all
load WMX1.mat;
dlmwrite('WMX11.txt',VAR4);
load WMX2.mat
dlmwrite('WMX22.txt',VAR5);
```

Fig. 4.27 -Programme Matlab

✓ **Résultat :**

On obtient deux fichiers txt nommé [WMX11.txt] et [WMX22.txt]

➢ Ensuite on va les modifiés :

```
clc
clear all
z1='WMX11.txt';
z2='WMX22.txt';
z=dlmread(z1);
a=dlmread(z2);
file =fopen('WMX112.txt','w');
for i=1:576
    for j=1:6
        k=((i-1)*6)+j+1;

       text=int2str(z(2,k));

       fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
file =fopen('WMX221.txt','w');
for i=1:288
    for j=1:7
        k=((i-1)*7)+j+1;

       text=int2str(a(2,k));

       fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
```

Fig. 4.28 -Programme Matlab

✓ **Résultat :**

Après cette modification on acquiert deux fichiers qui nommées [WMX112.txt] et [WMX221.txt] avec la forme alist.

➢ Enfin on va faire la vérification entres les deux fichiers précédents et notre alist

```
clc
clear all
matrix_CN = 'wimaxCN.txt';
matrix_VN='wimaxVN.txt';
CN=dlmread(matrix_CN);
VN=dlmread(matrix_VN);
z1='WMX112.txt';
z2='WMX221.txt';
z=dlmread(z1);
a=dlmread(z2);
if z==CN
    disp('right')
else
    disp('wrong')
end
 if   a==VN
    disp('right')
else
    disp('wrong')
end
```

Fig. 4.29 -Programme Matlab

✓ **Résultat :**

Les deux fichiers sont identiques on a réalisé notre but.

4.3.3. WRAN (802.22)

Dans la norme (802.22) notre alist utilisée est avec ses dimensions :

- n=384
- m=192

- dvmax=6
- dcmax=7

```
384 192
6 7
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 3
6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3
6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 6
7 7 7 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7
6 6 6 6 6 6
55 143 186 0 0 0
56 144 187 0 0 0
57 129 188 0 0 0
58 130 189 0 0 0
59 131 190 0 0 0
60 132 191 0 0 0
61 133 192 0 0 0
62 134 177 0 0 0
63 135 178 0 0 0
64 136 179 0 0 0
49 137 180 0 0 0
50 138 181 0 0 0
51 139 182 0 0 0
52 140 183 0 0 0
53 141 184 0 0 0
54 142 185 0 0 0
2 29 128 0 0 0
3 30 113 0 0 0
4 31 114 0 0 0
5 32 115 0 0 0
6 17 116 0 0 0
7 18 117 0 0 0
8 19 118 0 0 0
9 20 119 0 0 0
10 21 120 0 0 0
11 22 121 0 0 0
12 23 122 0 0 0
13 24 123 0 0 0
14 25 124 0 0 0
15 26 125 0 0 0
16 27 126 0 0 0
1 28 127 0 0 0
5 58 75 98 117 176
```

Fig .4.30 –"Alist Matrix" de la norme WRAN (802.22)

- ✓ Premièrement on va lit la position de CN et VN et affiche deux fichiers txt comme on a expliqué précédemment puis dans la conversion binaire et à partir de cette alist notre nombre de bits est 12 bits pour CN et VN
- ✓ La description VHDL : on a nommé le fichier « **PFES.vhd** », la figure suivante présente notre code vhdl pour cette norme :
- ✓

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;

entity PFES is
    Port ( CLK : in  STD_LOGIC;
           CE : in  STD_LOGIC;
           EN : in  STD_LOGIC;
           ADDR1 : in  STD_LOGIC_VECTOR (11 downto 0);
           ADDR2 : in  STD_LOGIC_VECTOR (10 downto 0);
           DATA1 : out  STD_LOGIC_VECTOR(11 downto 0);
           DATA2 : out  STD_LOGIC_VECTOR(10 downto 0));
end PFES;

architecture Behavioral of PFES is
type rom1_type is array (2303 downto 0 ) of std_logic_vector ( 11 downto 0 );
type rom2_type is array (1343 downto 0 ) of std_logic_vector ( 10 downto 0 );
signal ROM1 : rom1_type:=("000000110111","000010001111","000010111010","000000000000","000000000000","
signal rdata1 : std_logic_vector( 11 downto 0 );
signal ROM2 : rom2_type:=("00000100000","00000101101","00010001010","00010011110","00011000010","00011
signal rdata2 : std_logic_vector( 10 downto 0 );
begin
rdata1 <= ROM1(conv_integer(2303-ADDR1));
rdata2 <= ROM2(conv_integer(1343-ADDR2));
process (CLK)
    begin
        if (CLK'event and CLK = '1') then
        if (CE = '1') then
            if (EN = '1') then
                DATA1 <= rdata1;
    DATA2 <= rdata2;
            end if;
        end if;
        end if;
          end process;

end Behavioral;
```

Fig. 4.31- Le code VHDL pour la norme WRAN (802.22)

On a 2 RAMs :

- RAM1 qui avait une entrée ADDR1 et sortie DATA1
- RAM2 qui avait une entrée ADDR2 et sortie DATA2

Le nombre de bit est extrait de la manière suivante :

- ✓ CN (384*6=2304= 2^12)
- ✓ VN (192*7=1344=2^11)

- ✓ Ensuite, on a réalisé le modèle avec simulink system generator (Fig. 4.28) .on a nommé le fichier : « **SS1** ».

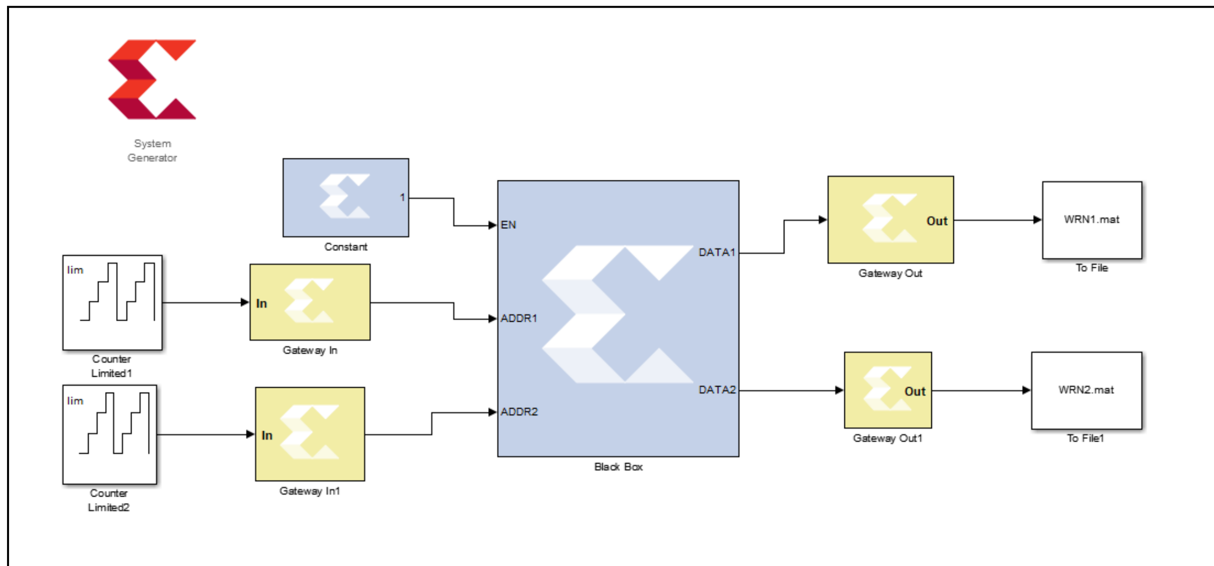


Fig. 4.32 -Simulink system generator modèl pour la norme WRAN (802.22)

✓ **Résultat :**

On obtient 2 fichiers [WRN1.mat] pour CN et [WRN2.mat] pour VN.

- ➢ D'après le résultat obtenu on va transformer les deux fichiers précédents vers un fichier txt on utilise une programmation Matlab.

```
clc
clear all
load WRN1.mat;
dlmwrite('WRN11.txt',VAR2);
load WRN2.mat
dlmwrite('WRN22.txt',VAR3);
```

Fig. 4.33 -Le programme Matlab correspond.

- ✓ **Résultat :**

  On obtient deux fichiers txt nommé [WRN11.txt] et [WRN22.txt]

- ➢ Ensuite on va les modifiés :

```
clc
clear all
z1='WRN11.txt';
z2='WRN22.txt';
z=dlmread(z1);
a=dlmread(z2);
file =fopen('WRN112.txt','w');
for i=1:384
    for j=1:6
        k=((i-1)*6)+j+1;

       text=int2str(z(2,k));

       fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
file =fopen('WRN221.txt','w');
for i=1:192
    for j=1:7
        k=((i-1)*7)+j+1;

       text=int2str(a(2,k));

       fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
```

Fig. 4.34 -Le programme Matlab correspond

- ✓ **Résultat :**

  Après cette modification on acquiert deux fichiers qui nommées [WRN112.txt] et [WRN221.txt] avec la forme Alist.

- Enfin on va faire la vérification entres les deux fichiers précédents et notre Alist

```
clc
clear all
matrix_CN = 'wranCN.txt';
matrix_VN='wranVN.txt';
CN=dlmread(matrix_CN);
VN=dlmread(matrix_VN);
z1='WRN112.txt';
z2='WRN221.txt';
z=dlmread(z1);
a=dlmread(z2);
if z==CN
    disp('right')
else
    disp('wrong')
end
 if   a==VN
    disp('right')
else
    disp('wrong')
end
```

Fig .4.35 -Le programme Matlab pour la vérification des 2 fichiers

- ✓ **Résultat :**

Les deux fichiers sont identiques. donc on a réalisé notre but.

4.3.4. LAN/MAN (802.3an)

Dans cette norme nous avons une seule « Alist Matrix » avec ses dimensions :

- n=384
- m=2048
- dvmax=32
- dcmax=6

```
384 2048
32 6
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
54 111 171 225 280 333 448 506 617 669 1497 1498 1499 1500 1501 1502 1503 1504 1506 1510 1515 1520 1526 1533 1537 1553 1576 1608 1649 1688 1719 1725
52 140 224 242 308 357 409 477 516 601 618 690 763 798 855 928 991 1024 1060 1078 1139 1201 1293 1443 1489 1523 1536 1551 1586 1682 1995 1997
51 93 139 223 241 307 356 408 476 600 689 762 854 990 1004 1023 1077 1138 1200 1292 1442 1522 1535 1550 1585 1708 1809 1838 1905 1949 2004 2008
92 138 222 240 306 475 515 599 688 761 797 853 927 989 1022 1076 1137 1291 1441 1521 1534 1549 1648 1681 1802 1825 1834 1846 1851 1920 1941 1943
50 91 221 239 305 355 407 474 514 598 668 687 760 796 852 888 926 988 1075 1136 1199 1290 1584 1647 1680 1748 1893 1902 1908 1945 1978 1979
49 90 137 220 238 304 354 406 473 513 597 667 686 759 795 831 851 925 987 1021 1074 1135 1198 1276 1289 1384 1440 1548 1583 1646 1679 1726
48 89 136 219 237 303 353 405 472 512 596 666 685 758 778 794 850 924 986 1020 1073 1134 1197 1275 1288 1383 1439 1547 1582 1645 1707 1727
47 88 135 218 236 302 352 404 471 511 595 665 684 723 757 793 849 923 985 1019 1072 1133 1196 1274 1287 1382 1438 1546 1581 1678 1706 1728
46 134 217 403 470 594 683 756 848 984 1018 1071 1132 1195 1273 1381 1437 1545 1644 1677 1789 1793 1860 1887 1892 1903 1925 1966 1998 2020 2038 2039
45 87 133 216 235 301 351 392 402 469 510 593 664 682 755 792 847 922 983 1017 1070 1131 1194 1272 1286 1380 1436 1544 1580 1643 1676 1729
44 86 132 234 350 468 509 663 754 791 846 921 982 1016 1271 1379 1456 1543 1579 1642 1675 1705 1831 1896 1919 1972 1982 1983 1994 2010 2014 2015
43 85 131 215 233 349 401 592 662 681 753 790 845 920 1015 1069 1130 1270 1435 1455 1542 1578 1641 1674 1765 1830 1844 1861 1868 1915 1947 1952
42 84 130 214 232 300 348 400 467 508 591 661 680 752 789 844 919 981 1068 1129 1193 1269 1285 1378 1434 1454 1488 1541 1640 1673 1704 1730
83 129 213 231 299 347 399 466 507 590 660 679 751 788 843 918 980 1014 1067 1128 1192 1268 1284 1377 1433 1453 1577 1639 1672 1703 1751 1811
41 82 128 212 230 298 346 398 465 561 589 659 678 750 787 842 917 979 1013 1066 1127 1191 1267 1376 1432 1452 1487 1532 1540 1638 1779 1812
40 81 127 211 229 297 345 397 560 658 786 841 916 978 1012 1065 1126 1190 1266 1375 1451 1484 1531 1702 1754 1810 1843 1911 1926 1938 1954 1962
39 126 210 228 296 344 396 464 588 657 677 749 840 977 1011 1064 1125 1189 1265 1374 1431 1483 1530 1539 1637 1671 1701 1901 2007 2024 2035 2037
38 80 125 209 227 295 343 395 463 559 587 656 676 748 785 839 915 976 1010 1063 1124 1188 1264 1373 1430 1450 1482 1529 1700 1787 1808 1813
37 79 124 208 226 294 342 394 462 558 586 655 675 747 784 838 914 975 1009 1062 1123 1187 1263 1372 1429 1449 1481 1528 1538 1636 1699 1731
36 78 123 279 341 461 557 585 746 783 837 913 974 1262 1371 1448 1480 1527 1635 1670 1698 1773 1850 1885 1886 1958 1965 1987 1990 2029 2042 2043
35 77 278 293 340 460 556 584 674 782 912 973 1008 1122 1186 1261 1329 1428 1447 1479 1509 1634 1697 1771 1774 1788 1845 1853 1855 1858 1859 1877
```

Fig .4.36 –"Alist Matrix" pour la norme LAN /MAN (802.3an)

- ✓ Comme les autres normes au début, on va lit la position de CN et VN et affiche deux fichiers txt comme on a expliqué précédemment puis dans la conversion binaire et à partir de cette alist notre nombre de bits est 14 bits pour CN et VN.
- ✓ La description VHDL : on a nommé le fichier « **MLNN.vhd »**, la figure suivante présente notre code vhdl dans cette norme :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.std_logic_unsigned.all;

entity MLNN is
    Port ( CLK : in  STD_LOGIC;
           CE : in  STD_LOGIC;
           EN  : in  STD_LOGIC;
           ADDR1 : in  STD_LOGIC_VECTOR(13 downto 0);
           ADDR2 : in  STD_LOGIC_VECTOR(13 downto 0);
           DATA1 : out  STD_LOGIC_VECTOR(13 downto 0);
           DATA2 : out  STD_LOGIC_VECTOR(13 downto 0));
end MLNN;

architecture Behavioral of MLNN is
type rom1_type is array (12287 downto 0 ) of std_logic_vector ( 13 downto 0 );
type rom2_type is array (12287 downto 0 ) of std_logic_vector ( 13 downto 0 );

 signal ROM1 : rom1_type:=("00000000110110","00000001101111","00000010101011","00000011100001",

 signal rdata1 :std_logic_vector(13 downto 0);

signal ROM2: rom2_type:=( "00000000111001","00000001111001","00000010111001","00000011111001","

signal rdata2 : std_logic_vector( 13 downto 0 );
begin

rdata1 <= ROM1(conv_integer(12287-ADDR1));
rdata2 <= ROM2(conv_integer(12287-ADDR2));
process (CLK)
    begin
        if (CLK'event and CLK = '1') then
        if (CE = '1') then
            if (EN = '1') then
                DATA1 <= rdata1;
   DATA2 <= rdata2;
            end if;
        end if;
        end if;
          end process;




end Behavioral;
```

Fig. 4.37 -Le code VHDL pour la norme LAN/MAN (802.3an)

On a 2 RAMs :

- RAM1 qui avait une entrée ADDR1 et sortie DATA1

- RAM2 qui avait une entrée ADDR2 et sortie DATA2

Le nombre de bit est extrait de la manière suivante :

✓ CN (384*32=12288= 2^14)

✓ VN (2048*6=12288=2^14)

✓ Ensuit on a réalisé le modèle avec simulink system generator on a nommé le fichier : «**MLWW** », la figure suivante présente ce modèle :

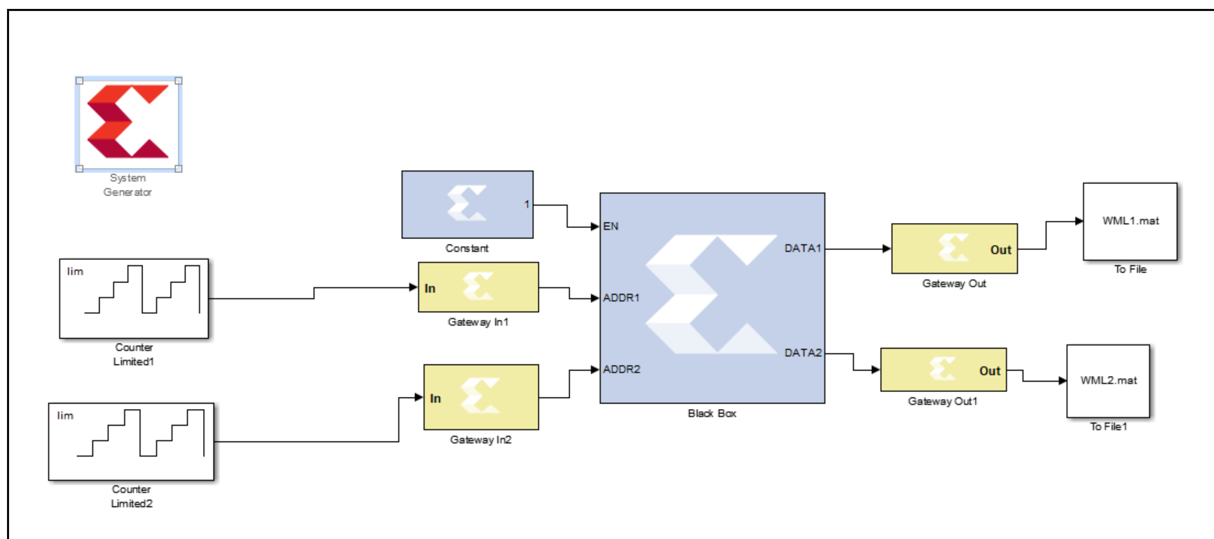


Fig. 4.38- Simulink system generator modèle pour la norme LAN/MAN (802.3an)

✓ **Résultat :**

On obtient 2 fichiers [WML1.mat] pour CN et [WML2.mat] pour VN.

➢ D'après le résultat obtenu on va transformer les deux fichiers précédents vers un fichier txt, la figure suivante montre notre programme Matlab de transformation :

```
clc
clear all
load WML1.mat;
dlmwrite('WML11.txt',VAR6);
load WML2.mat
dlmwrite('WML22.txt',VAR7);
```

Fig. 4.39 -Programmation Matlab

✓ **Résultat :**

On obtient deux fichiers txt nommé [WML11.txt] et [WML22.txt]

- Ensuite on va les modifiés :

```
clc
clear all
z1='WML11.txt';
z2='WML22.txt';
z=dlmread(z1);
a=dlmread(z2);
file =fopen('WML112.txt','w');
for i=1:384
    for j=1:32
        k=((i-1)*32)+j+1;

        text=int2str(z(2,k));

        fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
file =fopen('WML221.txt','w');
for i=1:2048
    for j=1:6
        k=((i-1)*6)+j+1;

        text=int2str(a(2,k));

        fprintf(file,'%s ', text);

    end
    fprintf(file,'\n');
end
```

Fig. 4.40- Programme Matlab

- ✓ **Résultat :**

  Après cette modification on obtient deux fichiers qui nommées [WML112.txt] et [WML221.txt] avec la forme Alist.

- ➢ Enfin on va faire la vérification entres les deux fichiers précédents et notre alist

```
clc
clear all
matrix_CN = 'lanmanCN.txt';
matrix_VN='lanmanVN.txt';
CN=dlmread(matrix_CN);
VN=dlmread(matrix_VN);
z1='WML112.txt';
z2='WML221.txt';
z=dlmread(z1);
a=dlmread(z2);
if z==CN
    disp('right')
else
    disp('wrong')
end
 if   a==VN
    disp('right')
else
    disp('wrong')
end
```

Fig. 4.41- Un programme Matlab pour la vérification des 2 fichiers

- ✓ **Résultat :**

Les deux fichiers sont identiques donc on a réalisé notre but.

4.4. Conclusion

"Alist Matrix" a été implémentée sous VHDL BRAM, généré par Matlab et simulé sous ISE Design Suite. Et d'autre part, cette matrice a été conçue avec succès sous le system generator, et à partir de ce dernier on peut trouver ce format de début. Notre objectif est donc réussi.

Conclusion générale

En conclusion, nous avons démontré dans ce mémoire un passage très important à partir de la forme alist qui a été inventé par David Mackay, Matthew Davey et John Lafferty, cette recherche est proposée en 2020 par Monsieur MAAMOUN. On a fait ce travail qui commence par cette invention « alist format » qui est basé au début du transit la forme précédente vers un VHDL BRAM , puis implémenté sous system generator ,afin de reprendre nos infos par Simulink modèle system generator on a appliqué cela sur plusieurs normes : WIFI (802.16), WiMax (802.11), WRAN (802.22) et LAN/MAN (802.3an)] dans ce cadre nous avons élaboré une plateforme permettant de générer la description VHDL du alist d'une norme automatiquement par MATLAB, cette description est interprétable par ISE Xilinx et System generator.

Bibliographie :

[1] : Nicolas Marques, "Méthodologie et architecture adaptative pour le placement efficace de tâches matérielles de tailles variables sur des partitions reconfigurables", Thèse de doctorat, Université de Lorraine, 2012

[2] : Tehami Mohammed Amine, "Codes LDPC: Construction, analyse et performances", Thèse de doctorat, 2018

[3] : MAHI Sarra. "Etude et implémentation d'algorithmes innovants de précodage MIMO associés aux codes LDPC". Thèse de doctorat, université Abou Bekr Belkaid

[4] : AL HARIRI, Alaa Aldin, "Architectures numériques configurables pour le traitement rapide sur FPGA de codes correcteurs d'erreurs de la famille QC-LDPC" , Thèse de doctorat. Université de Lorraine, 2015

[5] : Mahmoudy Iman ," Implémentation d'un décodeur LDPC sur une plateforme à base de DSP" , Université Sidi Mohamed Ben Abdellah ,2018

[6] : Guerrab Nassima,"Evaluation des performances des codes QC-LDPC en environnement multi-trajets",Thèse de doctorat, université Abou Bekr Belkaid, 2017

[7] : Derouiche Imane et Mihoubi Roumaissa,"Codes correcteurs d'erreurs LDPC de communication mobiles de la quatrième génération", Thèse de doctorat, université de bouira, 2017

[8] : Boughazi Imane, et Behri Fatima zohra, "Analyse et optimisation d'un code LDPC dans une transmission MIMO", Thèse de doctorat, université Abou Bekr Belkaid, 2016

[9] : IRINA ADJUDEANU , "Codes correcteurs d'erreurs LDPC structurés", Université Laval Québec, Mémoire de maitre en science , 2010

[10]: Thien Truong Nguyen Ly, "Efficient hardware implementations of ldpc decoders, through exploiting impreciseness in message-passing decoding algorithms", Thèse de doctorat, Université de Cergy Pontoise,2017

[11] : Zhengya Zhang," Design of LDPC Decoders for Improved Low Error Rate Performance", université de californie , berkeley , 2009

[12] : El hassani Sanae, "Optimisation du décodage des codes LDPC non-linéaires pour leur implémentation", Thèse de doctorat, 2011

[13] : AMIRZADEH, Ahmadreza. "Amélioration de la sécurité et de la fiabilité des systèmes de communication sans fil", université Laval Québec, 2017

[14] : Afettouche Malik, "Commande d'un robot en position à base d'une carte FPGA", université de Mouloud Mammeri, 2013

[15]: Hocini Ilyes et Dergaoui Mohamed, " Réalisation d'un modulateur et démodulateur OFDM sur FPGA ZYNQ", Thèse de doctorat, 2019

[16]: COCHACHIN, Franklin, DECLERCQ, David, KESSAL, Lounis, et al. Noisy Density Evolution for NAND Decoders, 2016

[17]: Djemoui Mohamed et Benaouadi Arezki," Filtrage des séquences vidéos sur FPGA avec l'outil X.S.G (xilinx system generator) , université de Saad Dahleb ,2012

Webographie :

[W1] : https://www.cder.dz/spip.php?article1003

[W2]: https://www.nandland.com/articles/block-ram-in-fpga.html

[W3] : Fr.wikipedia.org/wiki/matrice de contrôle

[W4]:http://www.telecom.ulg.ac.be/teaching/notes/total0/elen036/node129_mn.html

[W5]: uni-kl.de/Channel-codes/matrix-file-Formats/

[W6]: http://dictionnaire.sensagent.leparisien.fr/Xilinx/fr-fr/

[W7]:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug958-vivado-sysgen-ref.pdf

[W8]: https://www.mathworks.com/help/simulink/slref/tofile.html

[W9]: https://www.mathworks.com/help/simulink/slref/counterlimited.html