Ecole polytechnique de Louvain (UCL)

Etude et implémentation d'un NoC multiprocesseur à basse consommation selon la topologie de De Bruijn

Mémoire présenté en vue de l'obtention

du grade d'ingénieur civil électro-mécanicien (orientation mécatronique)

par STAS François

Auteur

Stas François

Jury :

Bol David (promoteur)

Legat Jean-Didier

Kuti Lusala Angelo

Juin 2013

Remerciements

Réaliser ce travail est le fruit du labeur de toute une année. Il constitue un défi qui permet à son auteur de montrer ce qu'il est capable de réaliser mais aussi de repousser les limites de ses capacités.

Je tiens particulièrement à remercier David Bol, Professeur, pour la motivation qu'il a su m'inculquer ainsi que pour la patience dont il a fait preuve. Il m'a permis de tirer le meilleur de moi-même afin que je puisse réaliser un travail de qualité. Je tiens également à le remercier pour l'opportunité qu'il m'a donnée, de pouvoir publier mon travail à la conférence FTFC 2013. Cette opportunité a été pour moi, un gage de confiance en mes capacités et en la qualité de mon travail.

Je remercie également Angelo Kuti Lusala, Docteur, pour ses encouragements et ses conseils précieux. Je tiens à le remercier pour sa rigueur terminologique et ses relectures indispensables.

Pour terminer, je remercie tous mes proches, pour leur soutien, leur patience et leurs relectures minutieuses malgré le peu d'affinité dans le domaine étudié.

Merci.

François

Abstract

Dans le cadre de l'évolution de l'électronique digital vers les Networks-On-Chip (NoC), ce travail propose l'étude d'un réseau dont les différents *processing elements* sont reliés entre eux selon la topologie de De Bruijn. Nous expliquerons pourquoi cette topologie est avantageuse par rapport au mesh. Nous étudierons notamment la relation entre la latence et la taille du réseau du point de vue théorique. Durant cette étude, un gain théorique de 30% sur la latence sera observé en comparaison avec le mesh pour un réseau de 64 *processing elements*. Ensuite nous proposerons plusieurs possibilités d'algorithmes de routage de paquet à travers le réseau afin d'obtenir les meilleures latences de transmissions en fonction de la contrainte d'injection et de la taille du réseau. Nous effectuerons ensuite une optimisation en fusionnant deux des algorithmes proposés pour implémenter le FIFO-based GSRAD avec Priority-deviation. Le but est de tirer parti des avantages des deux algorithmes de base tout en diminuant leurs désavantages respectifs. Nous observerons notamment une division de la latence moyenne par un facteur 4 par rapport au mesh XY baseline pour une contrainte d'injection de 10%. Par la suite, il sera aussi question de la consommation du réseau et en particulier du coût de connexion au réseau afin d'adapter le réseau de De Bruijn et son routage à des applications à basses consommations. Nous essayerons, notamment, de parvenir à un optimum dans le but de concilier latence et consommation. Enfin nous essayerons de relever le challenge du placement et routage d'un réseau de De Bruijn et nous proposerons des pistes de recherches afin de permettre l'implémentation d'un réseau d'ordre très élevé.

Abstract

In the context of digital electronic evolution towards Networks-on-Chip(NoC), this work propose a study of a network-on-chip (NoC) whose processing element are connected to each other in function of the De Bruijn topology. We will explain why this topology is interesting in comparaison to the mesh topology. Thus, we will study, theoritycally, the raltion between latency and network size. In this study, a 30% latency theoritical profit will be observed in comparaison with the mesh topology for a network with 64 processing elements. Then, we will propose several possibility for the packet routing algorithm in the network to obtain the best transmission latency performances in function of the injection contraint and the network size. We will also make a optimization in merging two studied algorithms to implemente the FIFO-based GSRAd with Priority-deviation. The goal is to have the advantages of the both without the disadvantages. We will observe which the latency is divided by 4 with this algorithm in comparison of the XY baseline mesh for 10% injection contraint. Then, it will be question of the network consumption and, in particular, the cost to connect a processing element to the network. The goal is to adapt the De Bruijn network and its routing algorithm for the low-power applications. We will try to obtain a optinum to concilier conciliate latency performances and consumption. Finally, we will try to solve the place and route challenge for a De Bruijn NoC and we will propose some possibility to allow place and route for high-order De Bruijn Network.

Table des matières

Chapitre 1

Introduction

L'ITRS *"International Technology Raodmap for Semiconductor"*, l'organisme chargé de donner les différentes orientations futures dans le domaine de l'électronique prévoit une évolution, dans ces projections de 2005[1], de la demande en puissance de calcul des systèmes digitaux au cours des quinze prochaines années (Fig 1.1).

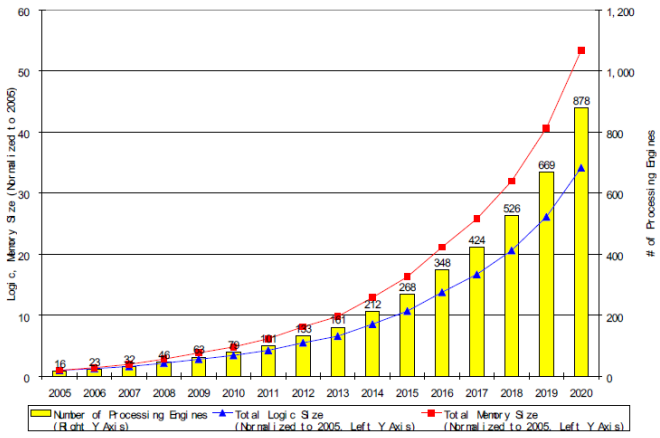


FIGURE 1.1 – Prévision de l'ITRS sur l'évolution du nombre de *processing elements* dans les systèmes digitaux[1]

Cette évolution est notamment le reflet de la loi de Moore. Cette loi éponyme de son auteur est une loi empirique énoncée dans "Electronics magazine" en 1965 par Gordon E. Moore, fondateur de la société Intel[2]. A l'origine, elle prédisait que l'évolution du nombre de transistors présents sur une puce de silicium doublerait tous les deux ans. Après sa vérification au cours des années 70, cette loi est devenue une perspective pour le secteur qui tend à y faire correspondre ses prévisions de développement.

Entre 1970 et 2005, l'évolution du nombre de transistors a ainsi suivi la loi de Moore en allant de 2300 transistors du processeur 4004 d'Intel jusqu'à plus de cent millions de transistors pour le Pentium4 HT (Fig. 1.2).

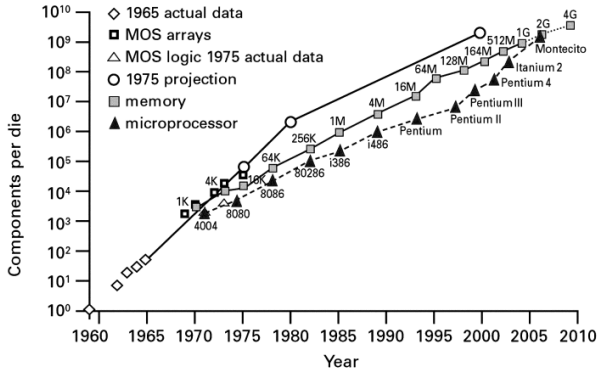


FIGURE 1.2 – L'évolution du nombre de transistors au sein des processeurs selon la loi de moore[3]

Le suivi de la loi de Moore a notamment été possible grâce à l'évolution en parallèle de la taille des transistors (Fig. 1.3). En effet, depuis l'énoncé de cette loi, la taille des transistors a suivi une évolution inverse diminuant systématiquement au cours des années afin de permettre l'implémentation d'un plus grand nombre de transistors sur une même surface[3].

A l'heure actuelle, afin de pouvoir poursuivre les objectifs de la loi de Moore, l'électronique a évolué vers les systèmes digitaux. En effet, la taille des processeurs ne pouvant augmenté indéfiniment, les recherches se sont dirigées vers les systèmes digitaux en implémentant plusieurs processeurs sur une même puce de silicium[4]. C'est ainsi que des systèmes dual-cores et quad-cores apparaissent sur le marché notamment en téléphonie mobile avec les smartphones.

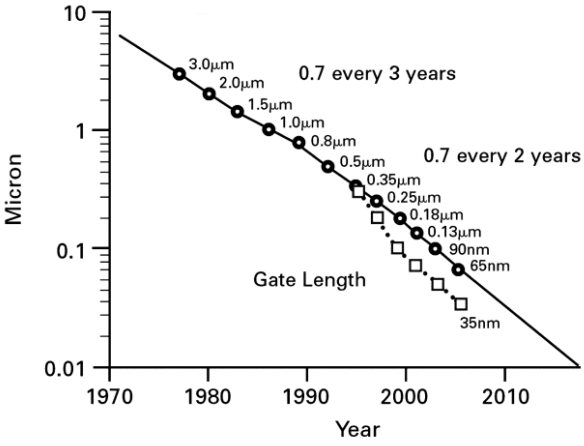


FIGURE 1.3 – L'évolution de la taille des transistors conçus par Intel[3]

C'est ainsi que les recherches en électronique digitale se sont consacrées à l'étude des systèmes multi-cores sous forme de réseau avec comme objectif d'atteindre les perspectives définies par l'ITRS en nombre de *processing elements* implémentés sur une seule puce de silicium. En effet les possibilités des différents aspects des réseaux (topologie, routage, consommation) offrent un vaste champ de recherches et des perspectives prometteuses.

Le cadre de ce travail se situe donc dans l'étude des réseaux sur puce ou *Networks-on-Chip* (NoCs). Il propose une étude de l'implémentation d'un réseau sur puce connecté selon la topologie de De Bruijn afin d'obtenir des bonnes performances en terme de transmission.

Ce travail se situe également dans l'évolution vers une électronique à basse consommation. Cet aspect est notamment du à la contrainte sur l'énergie disponible dans les systèmes embarqués notamment en téléphonie mobile. De plus depuis quelques années, sous la pression des politiques et de consortiums tels que GreenTouch et CoolSilicon, le monde de l'électronique tente une évolution vers un plus grand respect de l'environnement. Ce travail portera donc vers l'optimisation de la consommation des NoCs étudiés en fonction de leurs performances.

L'organisation de ce travail suivra le flux de conception des systèmes digitaux. Nous commencerons par passer en revue les avancées déjà effectuées dans le domaine des Networks-on-Chip. Nous parlerons notamment des différentes topologies qui ont été étudiées jusqu'à présent ainsi que des différents algorithmes de routage qui ont été mis en place afin de permettre une gestion efficace des communications entre les différents blocs.

Ensuite, nous introduirons la topologie de De Bruijn via son historique avant d'étudier les avantages que représente l'utilisation de cette topologie au sein des réseaux sur puce. Nous comparerons ses performances en terme de parcours moyen à la topologie mesh qui est la topologie dominante à l'heure actuelle. Nous introduirons également les routages spécifiques à la topologie de De Bruijn(GSRA et SSRA).

Au Chapitre 4, nous identifierons les conflits qui peuvent se présenter lors des transmissions de paquets au sein du réseau. Dès lors, nous proposerons plusieurs possibilités d'implémentation afin de réaliser un routage efficace via l'utilisation de FIFO (FIFO-based GSRA) ou par déviation de paquets (Priority-deviation-based GSRAD et Priority-proximity-based GSRAD). Nous proposerons également une optimisation de l'algorithme (FIFO-based GSRAD avec Priority-deviation).

Nous analyserons, ensuite au Chapitre 5, les comportements du réseau en terme de performances de transmission, en fonction des conditions auxquelles il est soumis notamment en fonction du taux d'injection de paquets et de la taille du réseau. Cette analyse qui a fait l'objet d'une publication à la conférence FTFC 2013, nous permettra notamment de comparer les différents algorithmes pour définir lequel est le plus performant(Annexe B). Nous soutiendrons également la comparaison avec un mesh baseline possédant un routage XY déterministe. Nous constaterons une diminution d'un facteur 4x de la latence moyenne de transmission dans le réseau de De Bruijn avec un FIFO-based GSARD avec Priority-deviation par rapport au mesh.

Le Chapitre 6 est consacré à la synthèse et à l'étude de la consommation des différents algorithmes de routage. Nous confirmerons l'hypothèse de la consommation importante des FIFOs et nous tenterons, à cette occasion, d'en optimiser la taille afin d'obtenir une faible consommation tout en conservant leurs performances de transmissions.

Le dernier Chapitre porte, quant à lui, sur le placement et routage d'un réseau de 16 nœuds. A cause de la taille des réseaux sur puce, cette étape est un challenge lors du flow de conception. Elle l'est d'autant plus que les graphes de De Bruijn sont non-planaires et leur scaling est difficile. Nous étudierons les options choisies dans la littérature scientifiques et nous nous en inspirons pour réaliser cette étape à l'aide de guides.

Nous conclurons, enfin, en faisant la synthèse des données collectées et nous établirons des conclusions quant à l'implémentation d'un réseau sur puce selon la topologie de De Bruijn pour applications à basse consommation. Nous conclurons notamment sur le fait la topologie de De Bruijn est un candidat intéressant pour les NoCs et que le FIFO-based GSRAD avec Priority-deviation et des FIFO composées de trois registres, est le routage le plus adapté parmi ceux proposés. Néanmoins le challenge du placement et routage reste une difficulté et une étage critique même s'il existe des pistes afin d'y parvenir.

Chapitre 2

Networks-on-Chip

A l'instar des réseaux macroscopiques de télécommunication, les réseaux sur puce sont des ensembles d'éléments qui sont connectés entre eux afin d'échanger des données dans le but d'accomplir une tache complexe. La particularité des réseaux sur puce se situe dans le fait que l'ensemble du réseau est disposé sur la même plaque, la même puce de silicium en opposition aux réseaux tels que Internet où les éléments connectés sont éloignés géographiquement les uns des autres. Comme décrit dans l'introduction, les réseaux sur puce sont une solution attractive pour la crise des communications et deviennent omniprésents dans les systèmes digitaux[5]. Ils visent à remplacer les architectures utilisant un bus de communication dont les performances se dégradent lorsque le nombre de processing element augmente[4][6]. A terme, le but est de pouvoir augmenter exponentiellement la taille des réseaux afin d'obtenir plus de puissance de calcul.

Ce chapitre vise à introduire les Networks-on-Chip (NoC). Tout d'abord, nous exposerons le principe de bus que les NoCs tendent à remplacer. Puis nous explorerons les différentes catégories de réseaux qui existent dans la littérature scientifique. Enfin nous introduirons le concept d'*algorithme* de *routage* de paquets au sein d'un réseau de processeurs et nous exposerons les caractéristiques principales de ces derniers. Enfin nous introduirons les modes de commutation ainsi que la métrique et les concepts liés aux réseaux sur puce.

2.1 Topologies des Networks-on-Chip

La topologie, du grec "$\tau o \pi o \varsigma$" et "$\lambda o \gamma o \varsigma$", est la science qui étudie les lieux, les situations. Dans le cas d'une étude sur les réseaux, la topologie est la branche qui étudie l'ensemble de la disposition des éléments d'un réseau et la manière dont ils sont connectés entre eux. En effet, il existe plusieurs possibilités de relier plusieurs objets pour former un réseau avec des propriétés, des avantages et des contraintes différents.

2.1.1 Architecture Bus

Avant de présenter les NoCs, nous allons, tout d'abord, présenter les architectures bus[7],[8]. L'architecture bus est une topologie qui connecte l'ensemble des modules sur une connexion unique, "le bus" (Fig. 2.1). Cette topologie est généralement utilisée pour connecter un processeur avec sa mémoire ou ses mémoires ainsi qu'avec ses modules extérieurs. A la Figure 2.1, nous pouvons observer une architecture bus de type AVALON développée par l'ALTERA sur lequel est connecté une caméra et un processeur NIOS II[9].

L'architecture bus de base comprend généralement trois sous-bus[5] ; le bus d'adresse, le bus de données, et le bus de contrôle. La gestion des transmissions à travers le bus est opérée par le *bus master* qui arbitre les transmissions. Dans beaucoup de cas, ce rôle est attribué au processeur. Pour effectuer une transmission, le bus master envoie une requête à travers le signal de contrôle *Req* vers un module *bus slave*. Le module qui correspond à l'adresse *Addr* capte la donnée et lève un signal de contrôle *Ack* pour confirmer la réception de la requête[5]. Parmi les bus standards, nous pouvons citer en plus du bus AVALON d'ALTERA[8], le bus AMBA de l'entreprise ARM[7].

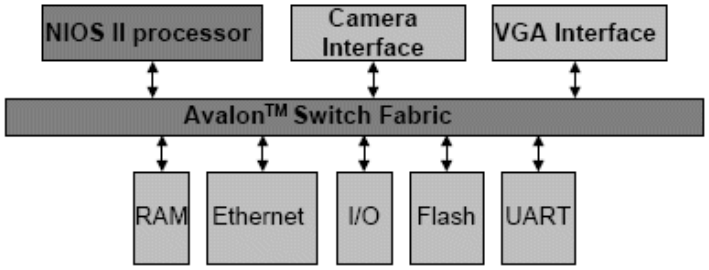


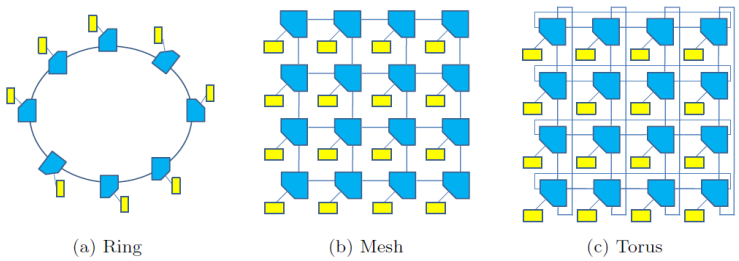
FIGURE 2.1 – Architecture bus AVALON développée par ALTERA[9]

L'architecture bus possède, cependant, ses limites[4][5]. En effet, il est difficile d'effectuer des opérations à hautes vitesses lorsqu'un grand nombre de modules sont connectés au bus. En effet, afin de relier tous les modules aux bus, la taille de ce dernier s'accroît, augmentant ainsi la capacité du bus. Le temps de propagation au sein du bus étant inversement proportionnel à celle-ci, la transmission à travers le bus a donc de plus en plus de mal à satisfaire les contraintes de timing. De plus, une seule transmission à la fois peut être effectuée sur le bus. Pour ces deux raisons, cette architecture est donc inadaptée pour les réseaux de processeurs.

2.1.2 Caractéristiques des Networks-on-Chip

On peut distinguer trois types de topologies dans la littérature scientifique : les topologies directes, indirectes et irrégulières[4][6][10]. Cette distinction se base sur la connexion ou la non-connexion de chaque nœud du réseau avec un terminal appelé *processing element* ou un *storage element*. Un terminal peut, notamment, être un processeur, une mémoire ou un accélérateur par exemple.

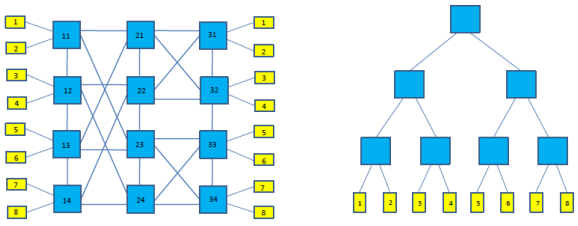
Dans une topologie directe, chaque nœud est relié à un terminal. Dans le cas d'un réseau de processeurs, tous les nœuds du réseau possèdent leur processeur qui peut émettre et recevoir des informations. Les topologies directes(Fig. 2.2) les plus répandues sont le mesh[11][12] et le torus[13]. Chaque point du réseau (en bleu) est relié à un processeur (en gris). De par leur ordonnancement, ces deux topologies permettent un placement et un routage très facile et efficace des différents modules du réseau. De plus la proximité d'un nœud avec ses voisins auxquels il est connecté, ne génère pas d'interconnexions traversant l'ensemble du réseau, sources de consommation et de contraintes sur le timing du circuit. Le dernier avantage du mesh et du torus est leur scaling qui peut être implémenté très facilement en fonction du nombre d'éléments désirés dans le réseau.



(a) Ring (b) Mesh (c) Torus

FIGURE 2.2 – Topologies directes[10] : (a) Mesh (b) Torus

La deuxième catégorie regroupe les topologies indirectes(Fig. 2.3). En opposition aux topologies directes, chaque nœud ne possède pas un *processing element*. Certains points du graphe ne sont utilisés que pour transférer les informations aux nœuds suivants. Les topologies en arbre ou en papillon[14][15] entrent dans cette catégorie. La topologie indirecte permet, notamment, de connecter des sous-ensembles d'éléments[15]. Cette disposition permet de regrouper en *cluster* des éléments qui doivent communiquer de manière intensive sans perturber le reste du réseau(Fig. 2.4). L'ensemble des clusters sont ensuite reliés les uns aux autres pour échanger les données traitées par exemple. Ceux-ci sont également utilisés dans le cadre de réseaux à topologie mesh[16].



(a) Butterfly (b) Binary Tree

FIGURE 2.3 – Topologies indirectes[10] : (a) Papillon (b) Arbre

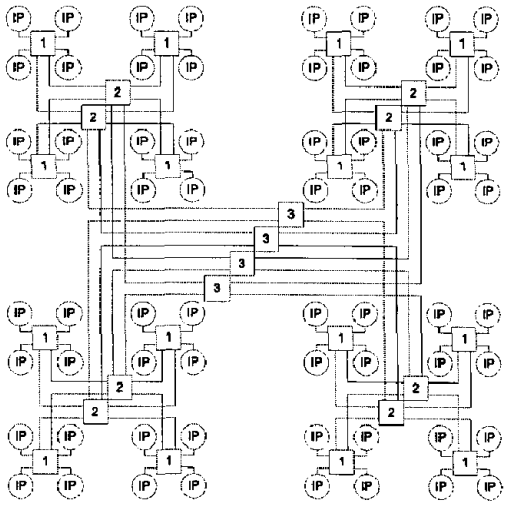


FIGURE 2.4 – Topologie papillon constitué de clusters reliés entre eux[15]

La dernière catégorie contient toutes les topologies hybrides. Ces topologies sont souvent des structures qui sont optimisées afin de tirer avantage des différentes topologies dont elles sont issues (Fig. 2.5). Lors de la réalisation d'un circuit de traitement de signal selon les normes MPEG-4[17][18], la topologie proposée relie les différentes parties du circuit de manière désordonnée selon les besoins de l'implémentation.

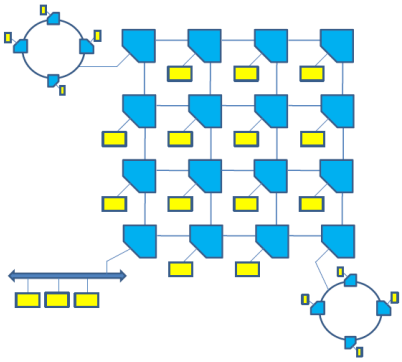


FIGURE 2.5 – Topologies hybrides[10]

Comme nous pouvons le constater, il existe plusieurs topologies possédant chacune des avantages et des inconvénients. Dans le cadre de ce travail, nous allons nous intéresser aux réseaux directs de processeurs selon la topologie de De Bruijn qui sera introduite au Chapitre 3. L'ensemble des nœuds du réseau possèderont des terminaux similaires comprenant un processeur ARM Cortex M0[19] et une mémoire de 1kB. Afin de pouvoir comparer la topologie de De Bruijn, nous avons choisi la topologie mesh car cette topologie est la topologie la plus répandue dans la littérature scientifique actuelle et elle constitue une solide base de comparaison.

2.2 Algorithme de routage

L'algorithme de routage permet la gestion des transmissions de paquets au sein du réseau. Il régule le trafic et détermine le trajet que doit emprunter un paquet pour arriver à destination. Cette section explore les différentes caractéristiques des algorithmes de routage qui sont étayés par différents exemples de networks-on-chip qui exploitent leurs capacités.

Il existe plusieurs types d'algorithmes de routage en fonction des spécifications du réseau et des performances à atteindre[6][20]. Ils existent deux grandes caractéristiques des routages au sein de NoCs. La première porte sur le caractère distribué ou de source. La deuxième permet de distinguer un algorithme déterministe ou adaptatif.

2.2.1 Déterministe ou adaptatif

Un algorithme de routage peut être déterministe ou adaptatif[11][12][13]. Les algorithmes déterministes fonctionnent sur le principe de lois fixes. Un paquet généré au nœud A à destination du nœud B suivra toujours le même parcours quelque soit l'état du réseau.

Les algorithmes déterministes sont, généralement, soumis à certaines conditions de fonctionnement. Par exemple, il ne peut avoir *fault* dans le réseau, c'est-à-dire, un endommagement des interconnexions entre les éléments du réseau. En effet, ces changements n'étant pas pris en compte par le routage, il est possible que l'algorithme ne soit plus en adéquation avec la topologie du réseau. Dans le cas de Network-On-Chip, la perte de connexion entre deux nœuds, suite à un défaut ou à un endommagement, pourrait entraîner des pertes de paquets lors des transmissions. L'avantage des algorithmes déterministes réside dans leur simplicité. Dans le cas de règles de routage simples, l'implémentation est d'une grande facilité et leur consommation est faible. Les algorithmes déterministes sont, par exemple, très bien adaptés au tri dans une arborescence. A chaque nœud, une question détermine si la donnée doit continuer dans la branche de droite ou de gauche. Un autre exemple est celui de l'algorithme déterministe XY pour un réseau connecté selon la topologie mesh[5] qui sera utilisé à des fins de comparaison dans ce travail (Sec. 5.8).

Les algorithmes adaptatifs permettent une plus grande liberté. En effet, selon leurs spécifications, ils peuvent prendre en compte l'état de congestion du réseau ou les états de leurs voisins les plus proches. Lors de l'envoi d'un paquet, une demande d'état aux nœuds adjacents peut être envoyée et la prise de décision est conditionnée par les réponses fournies. Les algorithmes adaptatifs ont l'avantage d'être plus performants en cas de changement dans l'encombrement du réseau ou en cas d'endommagement[12][21][22]. Néanmoins ils demandent une plus grande complexité et leur consommation est plus importante.

2.2.2 Distribué ou routage de source

Dans le routage de source, le nœud à partir duquel est généré le paquet, choisit le chemin que celui-ci prendra pour arriver à destination sans nouvelle évaluation durant sa transmission. A l'inverse, le nœud par nœud oblige chaque nœud à évaluer la prochaine destination du paquet en pouvant adapter le trajet de celui-ci en fonction de l'état du réseau dans le cas de routage adaptatif.

Le routage de source[23] est efficace dans le cas du réseau à topologie stable où chaque nœud connaît les chemins les plus performants. Pour une topologie donnée dans laquelle peu de données circulent et où chaque nœud ne communique qu'avec une petite quantité de nœuds déterminés, le routage par source est parfaitement adapté. En effet, les nœuds n'ont qu'à envoyer les données par des chemins prédéfinis. Par exemple, dans le cas d'un système de processeur où un seul nœud donné est une mémoire, chaque processeur peut envoyer un paquet au nœud mémoire via un chemin prédéfini et déjà encodé. L'avantage est que le bloc de routage est quasi inexistant puisque sa seule fonctionnalité est de transmettre un paquet en fonction de son adresse sans devoir définir son chemin. Le désavantage est qu'en cas d'encombrement à un endroit du réseau, une déviation des paquets n'est pas possible. Le nœud est obligé de stocker le paquet avant de pouvoir le transmettre une fois l'encombrement résolu.

Le routage "nœud par nœud", au contraire, évalue à chaque nœud quel chemin est le plus approprié pour transmettre le paquet. Le point de départ n'est pas obligé de connaître la totalité du chemin qu'empruntera le paquet tant qu'il connaît le nœud suivant. Ce type de routage est approprié dans le cas d'encombrement puisque un nœud peut modifier l'itinéraire d'un paquet en fonction des données sur l'état du réseau qu'il connaît. Le désavantage peut être une perte de temps par rapport au routage de source. Si l'évaluation du chemin le plus approprié prend du temps, la vitesse de transmission se réduira puisque chaque nœud va devoir réévaluer ce chemin.

2.3 Modes de commutation

Les modes de commutation définissent le degré de réservation utilisé pour acheminer un paquet dans le réseau. Il existe trois types de modes de commutation[4][5].

Le premier est le *wormhole*[24]. Pour envoyer une série de paquets vers un nœud "B", le nœud "A" envoie un paquet de tête (*Header*) pour réserver le chemin vers le nœud "B". Par la suite, l'ensemble des paquets suivront ce chemin, le dernier paquet libérant le chemin. L'avantage de ce mode de commutation est qu'une série de paquets arrive dans l'ordre dans lequel il est parti[4].

Le deuxième mode de commutation est le *cut-through*. Ce mode de commutation est plus libre car il se concentre sur chaque paquet séparément sans se soucier du lien entre les paquets. Ainsi quand un nœud "A" désire envoyer une série de paquets, il est possible que ces paquets suivent des chemins différents et n'arrivent pas dans l'ordre initial. Cependant, il permet de ne pas contraindre le réseau avec des réservations de chemin, évitant ainsi les problèmes de deadlock[4].

Le dernier mode est le *store-and-forward*. Ce mode oblige chaque nœud à attendre tous les paquets qui sont liés entre eux avant qu'ils puissent continuer leur chemin dans le réseau.

2.4 Métriques d'un réseau sur puce

Dans l'étude d'un réseau, nous allons définir plusieurs métriques intéressantes à étudier[5] dans le cadre de ce travail.

Le premier volet concerne les métriques quantifiant les performances de la transmission des paquets. La métrique la plus importante est la latence moyenne de transmission qui est définie comme étant le nombre de cycles d'horloge nécessaires en moyenne pour effectuer la transmission d'un nœud quelconque vers un autre. A la différence de la latence, le nombre moyen de hops mesure le nombre

de sauts qu'un paquet doit effectuer en moyenne pour atteindre sa destination. Le nombre de hops ne prend donc pas en compte les éventuels temps d'attente que le paquet peut subir entre deux hops. Nous pouvons aussi introduire la contrainte d'injection qui est le nombre de paquets qu'un nœud désire injecter sur le réseau durant un délai de cent cycles. De plus, le taux effectif d'injection est le nombre de paquets que le nœud a réellement pu injecter sur le réseau durant ce délai.

Le deuxième volet reprend les métriques en lien avec la consommation et la surface du réseau. La première métrique est la puissance consommée par le réseau. Dans le cadre de ce travail, nous travaillerons avec la techonologie ST65 de *STMicroelectronics* à des fréquences supérieures à 20MHz. La puissance consommée est donc majoritairement dominée par la puissance dynamique du réseau donc l'expression dépend des capacités des cellules et des fils de connexions (Eq. 2.1)[26].

$$P_{dyn} = \frac{1}{2} C V_{dd}^2 f_{clk} \tag{2.1}$$

La surface du réseau est le métrique qui mesure la place que requiert le placement des cellules physiques sur une puce. Cette métrique dépend de la densité souhaitée qui définit le taux d'occupation de l'espace et qui est fournie comme paramètre aux outils de placement et routage (Chap. 7).

Chapitre 3

Introduction de la topologie de De Bruijn

La topologie présentée lors de ce travail est basée sur les graphes dits "De Bruijn" définis en 1946 par Nicolaas De Bruijn[27], professeur néerlandais de mathématiques à l'université d'Eindhoven. Les graphes de De Bruijn permettent de représenter le chevauchement de l'ensemble des mots d'une longueur donnée par un graphe orienté.

Depuis lors, des recherches ont été effectuées sur les propriétés des graphes de Bruijn notamment en mathématique[28][29] et en biologie[30]. Depuis une vingtaine d'années, des recherches ont également proposé cette topologie comme une alternative au mesh dans le domaine des NoCs[31][32][33]. De par ses interconnexions, cette topologie permet, en effet, d'obtenir des performances en terme de latence moyenne très intéressantes notamment lorsque la taille du réseau augmente comme nous allons pouvoir l'observer au Chapitre 5.

Dans ce chapitre, nous allons, tout d'abord, introduire les graphes de De Bruijn et en définir les propriétés. Nous exposerons aussi les différentes variantes qui ont été développées dans la littérature[23][34]. Ensuite, nous expliciterons les algorithmes de routage adaptés à cette topologie. Nous terminerons, enfin, par une comparaison théorique entre la topologie mesh et la topologie de De Bruijn.

3.1 Historique

Les graphes dits "De Bruijn" ont été définis par Nicolaas De Bruijn dans "A combinatorial problem" [27]. Dans cet article, l'auteur fournit la solution au problème dit des "SuperString".

Quel est le SuperString le plus court possible qui contient toutes les possibilités de chaînes de longueur ***k*** *parmi un alphabet donné ?*

Dans l'alphabet digital, les seuls caractères sont le 1 et le 0. L'ensemble des mots de longueur **k** est simplement l'ensemble des chiffres binaires de longueur **k**. En d'autres termes, pour une longueur k égale à 3, quel est le plus court des mots(binaires dans le cas d'un alphabet digital) contenant l'ensemble des chiffres binaires de longueur 3 {000, 001, 010, 011, 100, 101, 110, 111} ? Dans ce cas-ci, la solution est 0001110100.

Afin d'obtenir la solution mathématique à ce problème, De Bruijn s'inspire de la solution de L. Euler à propos du célèbre problème des sept ponts de la ville prussienne de Königsberg(Fig. 3.1) :

La ville de Königsberg étant située sur deux îles et reliée à la berge par plusieurs ponts, existe-il un parcours qui permet de se promener sur l'intégralité des ponts en ne passant qu'une seule fois sur chaque pont ?

L. Euler résout ce problème en 1736 grâce à la théorie des graphes[35]. De nos jours, un graphe est dit eulérien si il est possible, en partant d'un nœud quelconque, de parcourir l'ensemble de ses arêtes de manière unique.

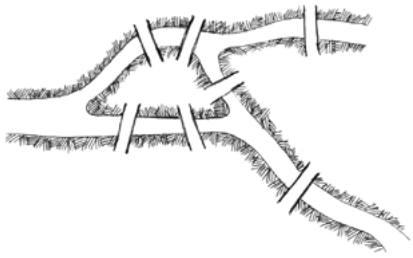


FIGURE 3.1 – Problème des ponts de la ville Königsberg[35]

L'idée de De Bruijn est de construire un graphe dont chaque nœud correspond à un mot de longueur $k-1$ de l'alphabet. Un nœud est relié à un autre si son suffixe correspond au préfixe du nœud auquel il est relié. Par exemple, à la Figure 3.2, une arête part du nœud 0**00** vers le nœud **00**1 car le suffixe de l'un correspond au préfixe de l'autre. De plus, à chaque arête correspond un mot de longueur k dont le suffixe est le nœud de départ et le préfixe correspond au nœud d'arrivée. En considérant un cycle eulérien du graphe (**0**000, **0**001, **0**011, **0**110, **1**100, **1**001, **0**010, **0**101, **1**011, **0**111, **1**111, **1**110, **1**101, **1**010, **0**100, **1**000), la SuperString cyclique la plus courte correspond au mot formé par le premier caractère de chaque arête du cycle (0000110010111101). En reportant les k-1 premiers caractères de la SuperString cyclique à la fin du mot, on trouve la SuperString la plus courte pour une longueur de mot et un alphabet donné : 0000110010111101000.

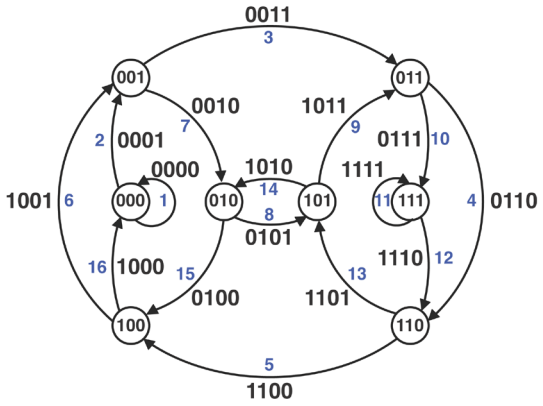


FIGURE 3.2 – Graphe de De Bruijn pour un k égale à 4[30]

3.2 Définition mathématique

Historiquement[27] comme expliqué dans le paragraphe 3.1, un graphe De Bruijn est un graphe d'ordre $k = 2^n$ construit à partir de la SuperString λ dont

- les sommets sont les facteurs de longueur k de λ,
- et les arêtes σ_i sont des facteurs de λ de longueur $k+1$ entre les 2 facteurs longueur k de σ_i.

Cette définition est extrêmement difficile à utiliser à des fins d'implémentation car il nécessite la connaissance des SuperString. Il existe un moyen de contourner ce problème en utilisant la propriété qui lie deux nœuds entre eux. Cette solution sera développée à la Section 4.1.

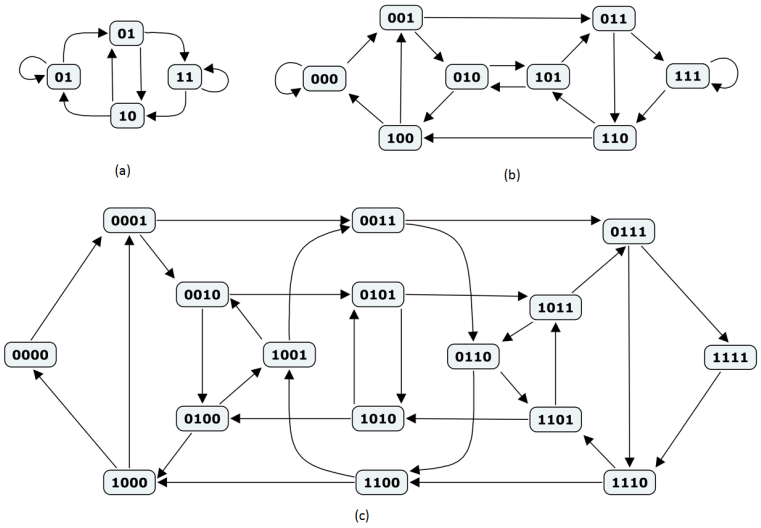


FIGURE 3.3 – Graphe de De Bruijn (a) ordre 4, (b) ordre 8, (c) ordre 16

3.3 Propriétés de la topologie de De Bruijn

3.3.1 Rappel des notions principales de la théorie des graphes

Afin de caractériser les performances des réseaux, il est utile de faire appel à la théorie des graphes. Un graphe est une abstraction théorique d'un réseau qui donne lieu à une modélisation mathématique de celui-ci. Dans beaucoup de domaines, la théorie des graphes est une part importante de l'aspect théorique notamment dans le transport routier ou dans notre cas, dans l'étude des NoC. L'étude d'un réseau via son graphe équivalent permet d'en observer les propriétés mathématiques afin d'en tirer des conclusions algorithmiques intéressantes. Dans cette optique, la transposition d'un réseau de processeurs dans la théorie des graphes est un aspect important de l'étude de ce dernier. Il donne une validation mathématique aux propriétés du réseau.

Un graphe G est représenté par un doublet (N,R) de trois sous-ensembles. Le sous-ensemble N est un ensemble fini dont les éléments sont les nœuds du graphe et R, le sous-ensemble d'arêtes reliant les différents nœuds de N du graphe G.

Sur la Figure 3.4, G_0 est un graphe possédant un sous-ensemble N de quatre nœuds : $\{A_0, A_1, A_2, A_3\}$ et un sous-ensemble R de six arêtes : $\{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$.

L'ordre n d'un graphe représente le nombre de nœuds que possède le graphe. Dans ce sens , le degré i d'un graphe est le nombre d'arêtes qui constituent ce dernier. Dans notre exemple, l'ordre du graphe G_0 équivaut à 4 tandis de son degré est de 6.

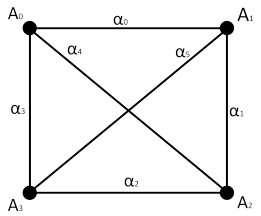


FIGURE 3.4 – Schéma du graphe G_0 à 4 sommets et 6 arêtes

Le parcours moyen (P_{mean}) d'un graphe est la moyenne des chemins les plus directs entre les différents nœuds du graphe. De plus, le parcours maximum (P_{max}) d'un graphe est le trajet qui, parmi les chemins directs entre deux nœuds, passe par le plus grand nombre d'arêtes. Si le parcours maximum est fortement supérieur au parcours moyen, cela signifie que le réseau est déséquilibré et donc, que certains éléments du graphe sont plus isolés que d'autres.

3.3.2 Parcours moyen et maximum d'un réseau de De Bruijn

Ordre et degré : Comme déjà mentionné, un graphe De Bruijn est de l'ordre $k = 2^n$ avec $n \in \mathbb{N}$. Pour la suite, n sera le niveau du graphe. Un graphe de De Bruijn de niveau 3 est donc un graphe possédant 8 nœuds. Le degré d'un graphe De Bruijn de niveau n équivaut à son ordre multiplié par 2 soit 2^{n+1} puisque de chaque nœud part deux arêtes.

Parcours moyen et parcours maximum Le parcours maximum équivaut au niveau du graphe De Bruijn et est donc fonction du logarithme en base 2 de l'ordre du graphe de De Bruijn. La démonstration de cette propriété sera effectuée lors de l'explication de l'algorithme de routage GSRA à la Section 3.6.

$$P_{max} = n = log_2(k)$$

Le parcours moyen d'un graphe de De Bruijn est, quant à lui, extrêmement difficile à évaluer mathématiquement à cause de sa non-linéarité[29]. Il est cependant possible de connaître une borne supérieure du parcours moyen d'un graphe de De Bruijn en fonction de son niveau. En effet, en partant d'un nœud quelconque, on considère qu'à chaque étape de propagation, chaque nœud atteint permet d'en atteindre deux autres, le chemin moyen est équivalent au logarithme en base deux du niveau du graphe diminué de un ($P_{mean} = log_2(n-1)$). Par exemple, pour un graphe de De Bruijn de niveau 3, en partant de 101, la propagation dans le réseau se déroule comme sur la Figure 3.5. Nous pouvons constater qu'après trois sauts, l'ensemble des nœuds ont été atteints. Cependant plusieurs nœuds ont été atteints différemment à des étapes de propagation différentes comme le nœud 011. Cela a pour conséquence d'augmenter la valeur du chemin moyen par rapport à sa valeur réelle.

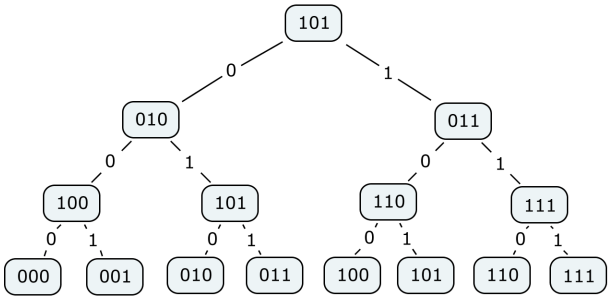


FIGURE 3.5 – Arbre binaire équivalent

3.4 Variantes des graphes de De Bruijn

Dans cette section, nous allons maintenant présenter les variantes des graphes de De Bruijn existantes dans la littérature scientifique. La conception permet notamment d'obtenir de meilleures performances en terme de parcours moyen ou encore, de créer un graphe de De Bruijn d'ordre quelconque.

3.4.1 Graphe De Bruijn direct ou bidirectionnel

Un graphe de De Bruijn est historiquement orienté. L'une des variantes est donc de le transformer en graphe bidirectionnel[23] en supprimant la direction des flèches du graphe. Cette transformation n'est pas sans conséquence. En effet, dans la pratique, supprimer le caractère directionnel d'une interconnexion, signifie dédoubler ce fil afin de permettre l'envoi et la réception d'un paquet. Cette modification permet de meilleures performances en terme de parcours moyen car la bidirectionnalité permet de raccourcir ces chemins. Mais, d'autre part, elle augmente le nombre de fils d'interconnexion ce qui présentera un problème lors du placement et routage (Chap. 7).

3.4.2 Graphe de De Bruijn généralisé

Dans ses travaux[23][32], M. Hosseinabady généralise la topologie de Bruijn en modifiant les lois de connexions pour construire un graphe de Bruijn possédant un nombre de nœuds quelconques (Fig. 3.6) contrairement à topologie de base où le nombre de nœuds varie selon la puissance de deux.

Dans un graphe de Bruijn généralisé, les lois d'interconnexions entre les nœuds se définissent selon les règles 3.1. Il existe une connection d'un nœud i vers un nœud j s'ils respectent une des deux équations où n est le nombre de nœuds et r, le degré d'un nœud soit 2 pour un alphabet digital. La Figure 3.6 représente un réseau généralisé de De Bruijn pour 14 nœuds. De plus, il s'agit d'un graphe bidirectionnel puisque l'orientation des connexions a été supprimée.

$$\begin{aligned} \mathrm{i} &= 2 * \mathrm{j} + \mathrm{r} (\mathrm{mod}\ \mathrm{n}), \mathrm{r} = 0 \text{ or } 1 \\ \mathrm{j} &= 2 * \mathrm{i} + \mathrm{r} (\mathrm{mod}\ \mathrm{n}), \mathrm{r} = 0 \text{ or } 1 \end{aligned} \tag{3.1}$$

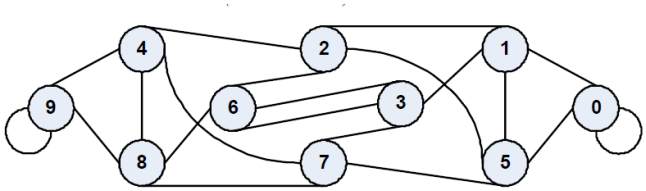


FIGURE 3.6 – Exemple de graphe de De Bruijn généralisé à 14 nœuds[23]

3.4.3 Graphe De Bruijn à alphabet complexe

Une autre variante est la construction basée sur un alphabet complexe de m lettres (Fig. 3.7). Chaque nœud est relié à m autres nœuds. Cette complexification du réseau permet une plus grande performance en ce qui concerne la vitesse de transmission car le nombre d'interconnexions augmente. Le prix à payer pour ces performances est la complexification de l'algorithme de routage ce qui augmentera la consommation du réseau. De plus, l'augmentation de la longueur des fils d'interconnexion complexifie l'étape de placement et routage qui devient un challenge de plus en plus difficile à réaliser (Chap. 7).

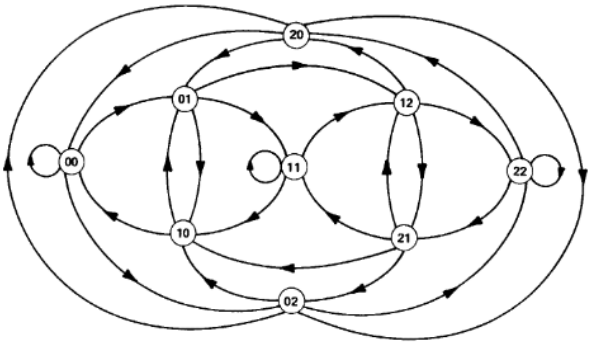


FIGURE 3.7 – Graphe de De Bruijn[34] avec un alphabet de trois lettres : 0,1,2.

3.5 Comparaison entre la topologie De Bruijn et la topologie Mesh

Comme exposé à la Section 2.1.2, il existe plusieurs topologies intéressantes dans le domaine du réseau sur puce. Afin de pouvoir comparer les performances de la topologie de De Bruijn, nous avons retenu la topologie mesh. Ce choix se justifie par la notoriété du mesh qui en fait une excellente référence.

L'une des propriétés les plus importantes dans l'étude de réseaux est le parcours moyen et maximum. Ces deux métriques permettent d'évaluer la vitesse de transmission au sein du réseau ainsi que l'équilibre de celui-ci.

Dans le cas du mesh (Fig. 2.2), l'évolution des parcours moyens et maximaux suit une courbe en racine carrée du nombre de nœuds connectés au réseau. Sur la Figure 3.8, nous constatons que les délais de transmission explosent lorsque le nombre de nœuds augmente selon les prévisions de l'ITRS (Fig. 1.1).

De plus, le parcours maximal évoluant beaucoup plus rapidement que le parcours moyen, le déséquilibre de la topologie se fait de plus en plus ressentir au fur et à mesure que la taille du réseau augmente. Certains éléments du réseau ont de plus en plus de mal à communiquer avec les éléments situés à l'autre bout du réseau.

Enfin, de par sa topologie, nous observons que le centre d'un réseau mesh devra gérer beaucoup plus de passages d'informations que les bords, vu que la majorité des chemins entre deux nœuds passe par la zone centrale du réseau. Cela sera entre autre une justification des moins bonnes performances face à la congestion, qui seront observées dans l'analyse comportementale.

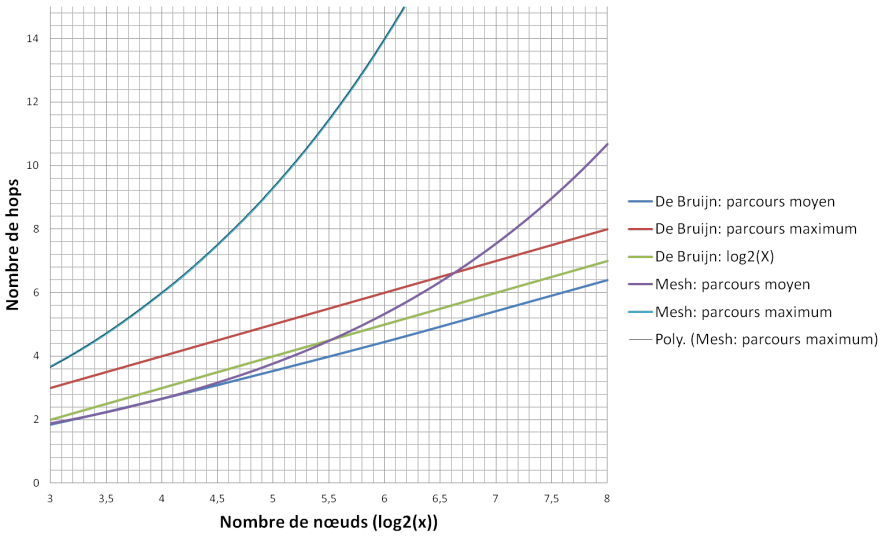


FIGURE 3.8 – Évolution en fonction du logarithme en base 2 du parcours moyen et maximal pour les topologies mesh et de De Bruijn

Afin de comparer le parcours moyen dans un mesh et dans un graphe de De Bruijn, nous avons réalisé un programme informatique pour déterminer le parcours moyen d'un réseau de De Bruijn d'un

niveau donné en sommant les longueurs des chemins de toutes les combinaisons de 2 nœuds, divisées par le nombre de combinaisons (Fig. 3.8).

Nous observons que l'évolution du parcours moyen et celle du parcours maximum sont linéaires en fonction du niveau du De Bruijn, c'est-à-dire, qu'elles évoluent de façon logarithmique par rapport au nombre d'éléments du réseau.

De plus, les courbes moyennes et maximales tendant à être parallèles, le déséquilibre du réseau reste, donc, constant ce qui permet d'espérer un comportement face à la congestion indépendant de la taille du réseau.

En conclusion, nous constatons, tout de suite, que, du point de vue théorique, la topologie de De Bruijn possède des propriétés intéressantes en terme de parcours moyen. Ces propriétés sont d'autant meilleures lorsque la taille du réseau devient importante.

3.6 Algorithmes de routage dans un réseau de De Bruijn

Il existe deux possibilités d'algorithmes[36] de routage totalement adaptés pour la topologie de De Bruijn : le GSRA et le SSRA. Le principe de ces deux algorithmes est basé sur la propriété de la topologie de De Bruijn qui relie deux nœuds dont l'identifiant du premier est un décalage de l'identifiant du second. Le GSRA profite de cette propriété dans un graphe de De Bruijn orienté. Le SSRA est, quant à lui, utilisé pour le graphe de De Bruijn non orienté et tire parti de la bidirectionnalité du lien entre deux nœuds.

Le **G**eneral **S**hifted **R**outing **A**lgorithm ou GSRA, en français l'algorithme de routage général par décalage est un routage spécifique à la topologie de De Bruijn. Il est basé sur le principe que chaque nœud est relié à un voisin dont le nom est un décalage de son propre identifiant. Il permet de déterminer par simple décalage un chemin vers la destination du paquet.

Par exemple, dans un réseau De Bruijn de niveau 3, un paquet dont la destination est le nœud "Z" correspondant à l'adresse 100, est généré au nœud "A" possédant l'identifiant 000. Dans un graphe de De Bruijn, le nœud "A" est relié à un nœud "B" qui possède le même identifiant mais décalé de 0 ou 1 vers la gauche. Le paquet n'a qu'à choisir son chemin en décalant l'identifiant du nœud où il se trouve par le premier de sa destination à savoir 1 pour arriver au nœud "B" dont l'identifiant est 001. Puis sa destination suivante sera le nœud avec l'identifiant de "B" décalé par le deuxième bit du paquet soit 0. Le paquet arrivera au nœud "C" dont l'identifiant est 010. Enfin, il effectuera la même opération avec le troisième et dernier bit de son adresse de destination pour arriver au nœud "Z" (Fig 3.9). Via cet exemple, nous pouvons démontrer que le parcours maximum dans un graphe de De Bruijn est égal au niveau de celui-ci. En effet, un nœud quelconque n'a qu'à choisir son chemin en décalant l'adresse de sa position par chaque bit de son adresse pour arriver à sa destination. Le parcours maximum est donc égal au nombre de bits des adresses, lui-même équivalent au niveau du graphe de De Bruijn.

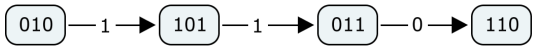


FIGURE 3.9 – Exemple de parcours via GSRA

L'algorithme SSRA pour **S**orthest **S**hifted **R**outing **A**lgorithm est l'équivalent du GSRA dans un réseau de De Bruijn bidirectionnel. La différence est que le paquet peut choisir de décaler l'adresse de

sa position soit par la droite, soit par la gauche. Dans notre exemple, le choix le plus judicieux aurait été de décaler l'adresse du nœud "A" de 1 vers la droite, pour arriver immédiatement au nœud "Z". Bien que facile pour un graphe de niveau 3, le SSRA devient plus complexe pour le niveau plus élevé car il doit prendre en compte beaucoup plus de combinaisons que le GSRA.

En conclusion, ce chapitre nous a permis d'introduire les graphes de De Bruijn qui présentent une interconnexion originale avec des propriétés intéressantes qui en fait un candidat dans le cadre d'une implémentation des réseaux sur puce. Cette topologie possède des qualités en terme de parcours moyen par rapport à la topologie mesh et un certain équilibre dans ses connexions évitant une centralisation des transmissions vers certains endroits du réseau. La suite de ce travail est donc basée sur l'étude d'un réseau connecté selon la topologie de De Bruijn unidirectionnel. Nous tenterons notamment d'implémenter un algorithme de routage performant basé sur le GSRA pour étudier le comportement du réseau. Enfin, nous essayerons de procéder au placement et routage de ce dernier et donnerons des pistes pour des recherches futures.

Chapitre 4

Conception d'un réseau de De Bruijn et de son algorithme de routage

Ce chapitre est consacré à la conception d'un réseau de De Bruijn. En effet, de par la complexité de la définition d'un réseau de De Bruijn, il est nécessaire de concevoir une méthodologie efficace afin de pouvoir réaliser les interconnexions entre les nœuds du réseau de manière automatique. De plus, nous verrons que la conception de l'algorithme de routage GSRA au sein d'un réseau de De Bruijn, fait face à des conflits. Au Chapitre 3, l'analyse des propriétés se base sur l'hypothèse selon laquelle un paquet voyageant dans le réseau est seul et ne rencontre aucun autre paquet. Nous n'avons donc pas encore pris en compte les conflits pouvant apparaître lorsque deux paquets se rencontrent dans le réseau. C'est notamment le cas lorsque deux paquets se trouvent à un nœud simultanément et désirent tous les deux partir dans la même direction.

Dans la première partie de ce chapitre, nous exposerons une définition plus intuitive des interconnexions dans un réseau de De Bruijn que la définition mathématique de la Section 3.2. A partir de cette définition, nous expliquerons une méthodologie de génération automatique de réseau de De Bruijn de niveau quelconque. Nous exposerons, aussi, les différentes possibilités d'implémentation du routage GSRA au sein d'un réseau de De Bruijn.

La seconde partie du chapitre traitera des possibilités de conception de l'algorithme de routage et nous y justifierons nos choix de conception. Ensuite nous exposerons les différents cas de conflits au sein du réseau et proposerons trois algorithmes de routage afin de les résoudre pour éviter les pertes de paquets. Les chapitres suivants détermineront ensuite lequel est le plus approprié pour effectuer le routage dans un réseau de De Bruijn à faible consommation.

4.1 Implémentation du réseau

La définition mathématique d'un graphe de De Bruijn (Sec. 3.2) demande la connaissance de la superString la plus courte d'un alphabet ainsi que de l'ensemble de ses facteurs. De ce fait, la construction sur base de cette définition est longue et fastidieuse. De plus, une topologie de niveau "k" de De Bruijn n'étant pas un ensemble de De Bruijn de niveau "k-1", il est impossible de construire un réseau d'un niveau donné à partir d'un assemblage de réseaux de niveau inférieur. Il est donc nécessaire d'automatiser la génération du réseau de la manière la plus simple possible.

Une construction plus intuitive consiste à réaliser un graphe de De Bruijn à partir de l'ensemble des nœuds correspondant aux nombres binaires de longueur k. Chaque nœud est ensuite relié à quatre autres sommets. Les deux premiers sommets sont ceux dont le nom correspond au nom du nœud de départ shifté de 1 ou 0 vers la gauche. Par exemple, le nœud 011 sera relié aux nœuds 110 et 111. Ces deux arêtes sont celles qui sortent de notre nœud. Les deux autres sommets sont ceux dont le nom correspond au nom du nœud de départ shifté de 1 ou 0 mais cette fois-ci vers la droite. Les deux arêtes sont celles qui entrent dans notre nœud. Le nœud 010 sera donc relié à 001 et 101.

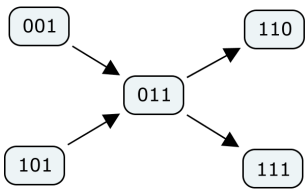


FIGURE 4.1 – Construction d'un nœud et de ses arêtes

Une fois le travail effectué pour chaque nœud, l'ensemble du réseau est connu puisque tous les nœuds ainsi que toutes les arêtes auxquelles ils sont reliés sont connus. Il est facile de générer l'ensemble du réseau en SystemC ou en Verilog. Cette construction intuitive est facile à réaliser en pratique. Le code des deux classes JAVA qui implémente cette solution se trouve sur la Figure 4.2. La première classe crée un tableau avec l'ensemble des nœuds implémentés sous forme d'objet JNode. Chaque objet JNode créé possède quatre String qui contiennent les noms des nœuds auxquels ils sont reliés en entrée ou en sortie. Une partie du code générant les fichiers SystemC et Verilog d'un réseau De Bruijn de niveau quelconque est présente dans l'annexe C.

4.2 Implémentation du routage GSRA

Comme décrit dans la caractérisation des algorithmes de routage, il existe plusieurs possibilités d'implémenter le GSRA. La Section 2.2 a permis de dégager plusieurs philosophies dans l'implémentation des algorithmes de routage. Il faut notamment déterminer si nous implémentons un routage par source ou distribué. De plus, il faut choisir si l'algorithme sera déterminé ou adaptatif.

Dans le cadre de ce travail, nous allons poser des choix quant à ces différentes possibilités. En effet, en fonction des objectifs et buts à atteindre, certaines caractéristiques semblent plus adaptées que d'autres.

Le premier choix porte sur l'implémentation d'un algorithme de source ou nœud par nœud. Il est possible d'implémenter le GSRA selon ces deux philosophes.

```
public JNode [] genNode()
{
  JNode [] jn = new JNode[pow(2,level)];
  int j;
  for(j=0; j<pow(2,level); j++)
  jn[j] = new JNode(decToBin(j, level));

  return jn;
}
public String decToBin(int nbr, int length)
{
  int temp = nbr;
  String str = "";

  int i;
  for(i=31; i>-1; i--)
 {
    int sub = pow(2,i);
    if((temp - sub) < 0)
    {
      if(str.compareTo("")!=0 | i<length)
      str = str + "0";
    }
    else
    {
      str       = str + "1";
      temp      = temp - sub;
    }
  }
  return str;
}
```

```
public JNode(String name)
{
  this.name = name;
  in1  = "0" + shiftRight(name);
  in2  = "1" + shiftRight(name);
  out1 = shiftLeft(name) + "0";
  out2 = shiftLeft(name) + "1";
}
public String shiftRight(String str)
{
  String shift = "";
  String [] str_split = str.split("");

  int i;
  for(i=1; i<(str_split.length-1); i++)
  shift = shift + str_split[i];

  return shift;
}

public String shiftLeft(String str)
{
  String shift = "";
  String [] str_split = str.split("");

  int i;
  for(i=2; i<str_split.length; i++)
  shift = shift + str_split[i];

  return shift;
}
```

FIGURE 4.2 – Code JAVA générant les nœuds et les arêtes d'un réseau De Bruijn

Dans le routage de source (Fig. 4.3), le nœud de départ fournit l'adresse de destination et envoie, au premier nœud, un paquet contenant la donnée, l'adresse et un compteur initialisé à 0. Chaque nœud doit ensuite envoyer le paquet au nœud suivant en fonction de la valeur du bit d'adresse désigné par le compteur. Ainsi si un paquet avec l'adresse 0101 et un compteur de 2 arrive au nœud "A" dont l'identifiant est 1001, ce dernier prend le troisième bit de l'adresse de destination du paquet à savoir 0. Le nœud "A" envoie ensuite le paquet à l'un des deux nœud auxquels il est relié et dont l'identifiant est son propre identifiant (1001) décalée de 0 c'est-à-dire 0010. Le nœud "B" 0010 recevra donc un paquet avec l'adresse 0101 et un compteur de 3.

L'autre possibilité est l'implémentation du routage nœud par nœud où chaque point du réseau recevant un paquet réévalue le chemin du paquet. Afin de connaître le prochain point du réseau qui permettra d'acheminer le paquet le plus rapidement possible, le nœud va comparer l'ensemble des préfixes de l'adresse de destination du paquet avec les suffixes de sa propre adresse. Dans un réseau de niveau 5, la donnée dont l'adresse de destination est 11010 arrive au nœud 11101 (Fig. 4.4). L'ensemble des suffixes du nœud sont 11101, 1101, 101, 01, 1 et l'ensemble des préfixes de l'adresse du paquet sont 11010, 1101, 110, 11, 1. La comparaison deux à deux des deux ensembles donne deux possibilités 1101 et 1. Le premier (1101) est préféré au deuxième car il possède le plus grand nombre de bits. Dans l'adresse du paquet, le bit suivant le préfixe 1101 est 0. Le nœud envoie donc le paquet au nœud dont le nom est son identifiant shifté de 0 auquel il est relié (11101 <- 0 <- 11010) à savoir le nœud 11010.

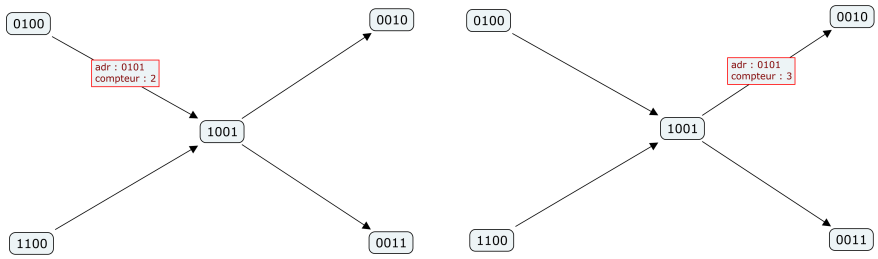


FIGURE 4.3 – Implémentation du GSRA via un routage de source

Dans le cadre d'une implémentation sur puce, le routage par nœud s'impose pour trois raisons. Premièrement, le routage de source nécessite l'envoi des bits de comptage au sein du paquet ce qui augmente la taille du paquet transmis et donc la consommation du réseau (Chap. 7). Deuxièmement, le routage de source impose un compteur tandis que le routage nœud par nœud ne demande que des comparaisons qui sont plus rapides et moins grandes consommatrices d'énergie. Troisièmement, en cas des conflits, les possibilités de résolution du routage par source sont restreintes car le paquet ne peut être dévié de sa trajectoire. Pour ces différentes raisons, notre choix se porte sur l'implémentation d'un routage nœud par nœud.

Le deuxième choix de conception se porte sur le caractère déterministe ou adaptatif de l'algorithme. Cette question se pose, notamment, du point de vue de la congestion du réseau. L'algorithme du routage doit-il s'adapter à l'état de congestion du réseau ou pas ? Dans la section 4.3, nous allons voir qu'il est possible d'envisager les deux possibilités notamment pour résoudre les conflits au sein du réseau.

En conclusion, après analyse des différentes possibilités, certaines caractéristiques nous semblent ressortir plus que d'autres. Dans le cadre de ce travail, nous effectuerons donc une implémentation du GSRA qui sera distribuée et qui se basera sur la comparaison entre les suffixes du nœud et les préfixes de l'adresse du paquet.

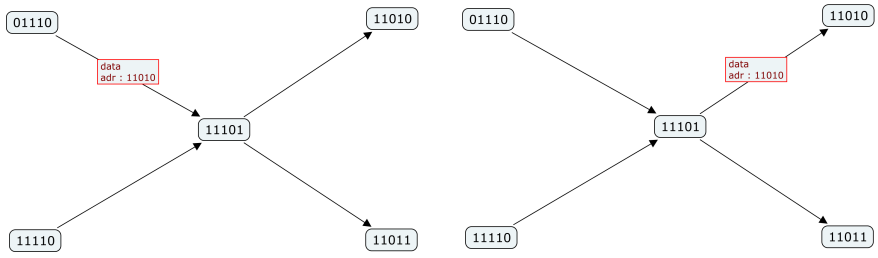


FIGURE 4.4 – Implémentation du GSRA via un routage nœud par nœud

4.3 Conflits au sein d'un nœud

Nous allons maintenant étudier les conflits qui peuvent exister lors de transmissions de paquets dans un réseau. Comme énoncé dans l'introduction de ce chapitre, il existe des possibilités de conflits. En effet, dans un graphe de De Bruijn, un nœud peut recevoir deux paquets en même temps et générer lui-même un paquet dans le réseau (Fig. 4.5). Il est donc possible qu'un nœud soit contraint de devoir gérer la transmission de 2 ou 3 paquets en même temps ce qui est source de conflits. Comme une seule transmission de paquet par sortie peut être effectuée, les conflits apparaissent lorsque deux paquets désirent être envoyés dans la même direction.

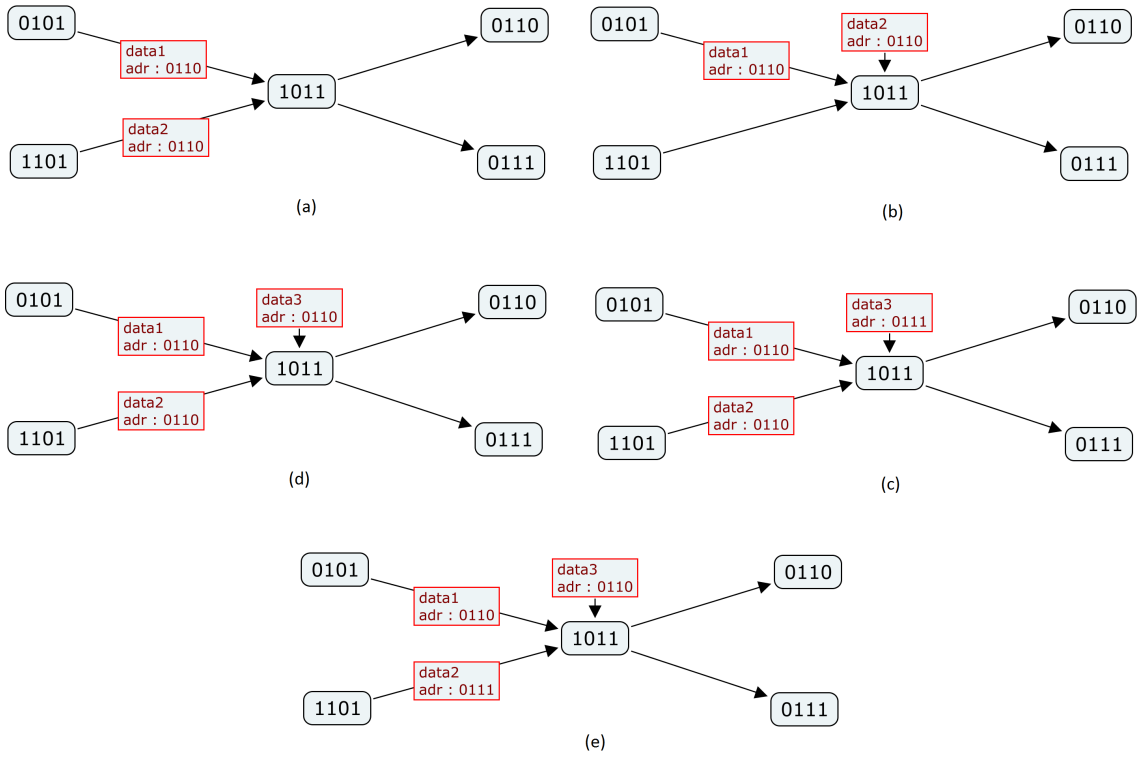


FIGURE 4.5 – Possibilités de conflits au sein d'un nœud

Il existe cinq possibilités de conflits.

- Cas 1 :Deux paquets arrivent par les deux entrées du nœud et désirent être transmis dans la même direction (Fig. 4.5(a)).
- Cas 2 :Un conflit apparaît si un paquet entrant désire être transmis dans la même direction qu'un paquet généré par le nœud (Fig. 4.5(b)).
- Il est aussi possible que deux paquets arrivent au nœud et qu'un troisième paquet soit généré par le nœud. Dans ce cas, il existe trois possibilités :
  - Cas 3 :Les trois paquets désirent aller dans la même direction (Fig. 4.5(c)).
  - Cas 4 :Les deux paquets entrants désirent aller dans la même direction et le paquet généré dans l'autre direction (Fig. 4.5(d)).
  - Cas 5 :Les deux paquets entrants désirent aller dans des directions différentes et le paquet généré dans l'une de celles-ci (Fig. 4.5(e)).

Ces cinq cas posent un réel problème lors de l'implémentation du module du routage. Ils sont au coeur de l'analyse comportementale du réseau. En effet, il est impératif que l'ensemble des conflits soient résolus sans perte de données et avec un ralentissement minimum du réseau. La solution apportée déterminera les performances du réseau lorsque le taux de paquets générés dans le réseau sera important. Plus il y aura de paquets générés en même temps dans le réseau, plus le nombre de conflits sera élevé. La rapidité et l'efficacité de la solution apportée est donc prépondérante. Il est donc nécessaire d'implémenter des stratégies efficaces afin de résoudre ces conflits. Nous allons envisager les stratégies. La première se base sur l'implémentation d'un algorithme déterministe utilisant des FIFO. La deuxième solution est un algorithme adaptatif qui permet de trouver des chemins alternatifs lorsqu'un conflit se produit.

4.4 Résolution des conflits par intégration de FIFO

La première possibilité est, donc, l'implémentation d'un algorithme déterministe utilisant des FIFO (First-In, First-Out). La FIFO est un ensemble de registres qui permet de stocker des paquets avant qu'ils puissent être envoyés. Elle est très rapide mais se compose de registres qui consomment beaucoup et prennent de la place[23]. Le nombre de registres que possèdent une FIFO est donc un paramètre à minimiser comme nous le verrons dans le Chapitre 6. Par la suite, cet algorithme de routage sera appelé *FIFO-based GSRA* par la suite.

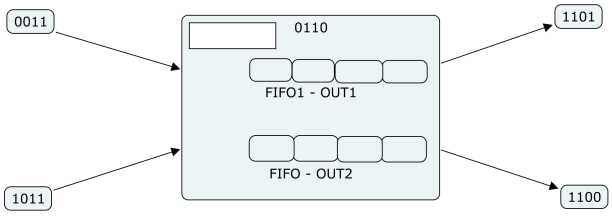


FIGURE 4.6 – Module de routage avec intégration de FIFO aux sorties

L'intégration d'une FIFO aux sorties du nœud (Fig. 4.6) est donc une première possibilité. En référence à son nom "First-In, First-Out", la solution de la FIFO peut se résumer à premier arrivé, premier sorti. Lorsqu'un ou plusieurs paquets arrivent aux nœuds à un instant donné, ils sont enregistrés dans la FIFO de leur sortie et attendent d'être envoyés un à un. Le désavantage de cette solution réside dans la capacité de la FIFO. Si le nombre de conflits augmente, il est possible que la FIFO soit remplie, saturée et qu'elle ne puisse plus enregistrer de données.

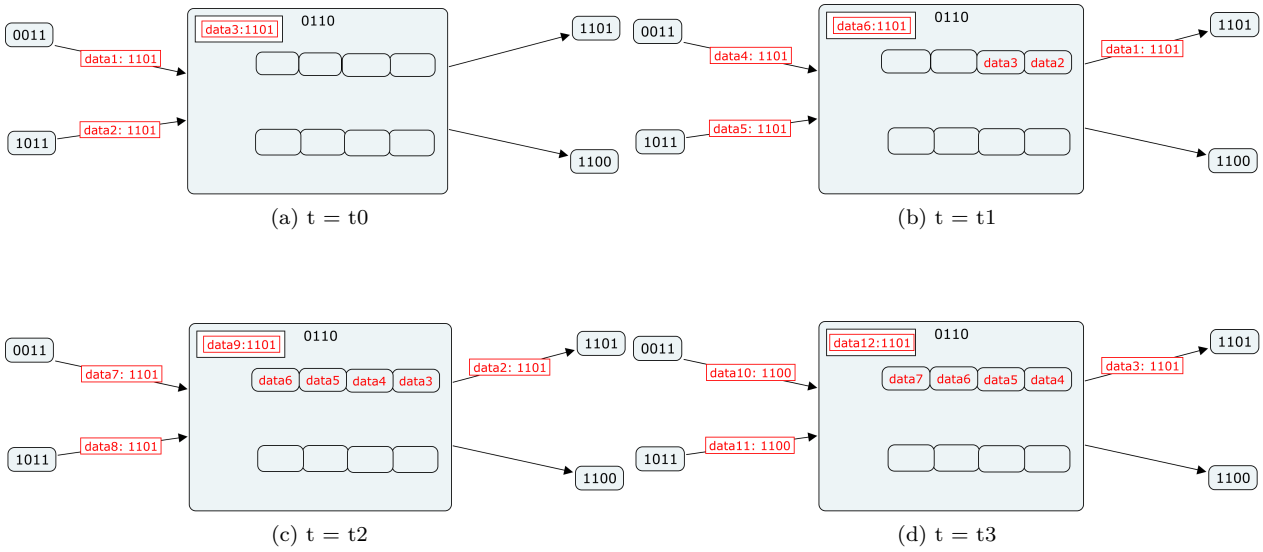


FIGURE 4.7 – Gestion de la transmission des données dans un module de routage avec FIFO

En effet, dans le cas, d'une FIFO avec 4 registres, si le nœud rencontre trois conflits de suite du type (Case 3), le nœud se comportera comme sur la Figure 4.7. Après les trois coups d'horloge, la FIFO n'a plus de place pour enregistrer les données 8 et 9 qui seront perdues (Fig. 4.7d) ce qui n'est pas tolérable dans les NoCs actuels[5]. Il est donc nécessaire de prévoir un mécanisme qui empêche ces pertes comme par exemple un mécanisme de *control flow* avec un mode de commutation wormhole(Sec. 2.3).

4.5 Résolution des conflits par déviation de données : GSRAD

4.5.1 Principe

La deuxième possibilité implémentée pour gérer les conflits au sein du réseau, est de dévier un des paquets ou de mettre en attente le processeur du nœud lorsqu'un conflit existe. Ainsi dans le cas de la Figure 4.8, le paquet 2 est envoyé dans la bonne direction tandis que le paquet 1 est dévié et que le *processing element* du nœud 0110 est mis en attente tant que le paquet 3 ne peut pas être injecté sur le réseau.

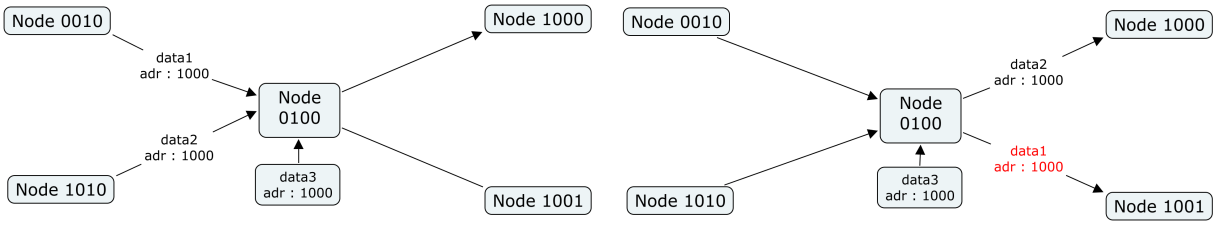


FIGURE 4.8 – Gestion de la transmission des données dans un module de routage avec déviation

Cette solution possède l'avantage de ne pas fournir au réseau plus de données qu'il ne saurait en gérer puisqu'il ne peut y avoir que 2n données [1] en même temps sur le réseau contrairement à la solution avec FIFO où le nombre de données présentes dans le réseau peut croître en fonction de la taille des FIFO. Le réseau ne peut donc, en aucun cas, perdre des paquets de données par manque de capacités. Cette solution possède, néanmoins, deux désavantages. D'une part, les déviations de données ne sont pas contrôlées ce qui peut amener un paquet à être dévié à l'infini sans pouvoir atteindre sa destination. D'autre part, vu que, dans certains cas, le processeur attend avant de pouvoir envoyer au réseau un paquet de données, il existe une perte de puissance de calcul due à la gestion des conflits.

4.5.2 Types de déviation

Comme déjà mentionné, la solution de déviation peut s'avérer dangereuse car il existe une probabilité non nulle qu'un paquet soit en permanence dévié. De plus il est nécessaire d'imposer des critères sur le choix du paquet qui sera dévié. Pour cela, nous proposons d'analyser trois types de critères sur lesquels se basera ce choix.

Déviation aléatoire

La première solution est basée sur l'absence de critère. Le choix du paquet dévié se fait par pur hasard. Cette implémentation n'est pas réalisable en pratique car elle nécessite l'intégration du module de génération d'une variable totale aléatoire. Néanmoins il est intéressant de pouvoir comparer les autres solutions à celle-ci. Cela permettra de déterminer si la solution apporte une plus-value par rapport à un système non-contraint. Il est à noter que bien que difficile à mettre en oeuvre dans la pratique, la solution est facile à implémenter en SystemC grâce à la fonction rand(). Nous appellerons cet algorithme le *Random GSRAD*.

1. n : le nombre de nœud du réseau

Contrainte sur le nombre de déviations

La deuxième solution consiste à ajouter un poids à un paquet en fonction du nombre de déviations qu'il a subies(*Deviation-priority-based GSRAD*). Ainsi un paquet déjà dévié sera prioritaire par rapport à un autre. Cette solution permet de supprimer la possibilité d'un paquet voyageant sans fin dans le réseau puisqu'à mesure qu'il est dévié, il devient prioritaire. Sur la Figure 4.9, nous constatons que le paquet 2 est prioritaire par rapport au paquet 1 car le paquet 2 a déjà été dévié par deux fois tandis que le paquet ne l'a pas été. Au coup d'horloge suivant, le paquet 1 est donc dévié et son compteur de déviations s'est incrémenté d'une unité.

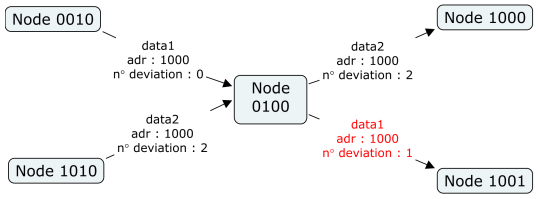


FIGURE 4.9 – Ajout d'un critère de choix sur le nombre de déviations d'un paquet

Contrainte sur la proximité

La dernière solution proposée est d'ajouter une priorité au paquet à mesure qu'il est à proximité de sa destination (*Proximity-priority-based GSRAD*). Cette approche permet de ne pas dévier un paquet qui serait arrivé à sa destination au jump suivant. L'objectif est de limiter l'impact des transmissions qui se déroulent entre deux nœuds adjacents. En effet, dans les deux premières solutions, un paquet qui doit être transmis au nœud voisin possède une probabilité d'être dévié et de générer à son tour des conflits. Ainsi sur la Figure 4.10, le paquet 1 doit d'abord passer par le nœud 1000 avant d'arriver au nœud 0001 deux coups d'horloge plus tard tandis que le paquet 2 a pour destination le nœud 1000. Le paquet 2 est donc prioritaire car il atteint sa destination le coup d'horloge suivant. Si le paquet 2 avait été dévié à la place du paquet 1, les deux paquets seraient encore présents sur le réseau au coup d'horloge suivant. Le nombre de paquets sur le réseau ne diminuant pas, le risque de congestion est plus important.

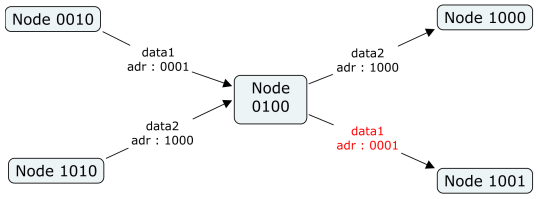


FIGURE 4.10 – Ajout d'un critère de choix sur la proximité de la destination

En conclusion de ce chapitre, nous avons maintenant conceptualisé quatre possibilités de routage (FIFO-based GSRA, Random-based GSRAD, Priority-proximity-based GSRAD et le Priority-deviation-based GSRAD) et déterminé une méthodologie afin de réaliser un réseau de De Bruijn automatiquement. L'objectif de la suite de ce travail est de déterminer quel algorithme est le mieux adapté au point de vue des performances de transmission (Chap. 5) et de la consommation (Chap. 6) afin de pouvoir proposer un algorithme performant dans le cas d'applications basses consommations utilisant un réseau de De Bruijn.

Chapitre 5

Étude comportementale

Cette section est consacrée à l'analyse comportementale d'un réseau selon la topologie de De Bruijn. Elle permet l'évaluation des performances et des capacités de transmission. Dans un premier temps, nous introduirons les paramètres des simulations en SystemC et la métrique statistique nécessaire à l'étude comportementale de tels réseaux. Par la suite, nous évaluerons l'impact des différentes solutions implémentées pour résoudre les conflits dans le réseau notamment du point de vue de la latence. Ces simulations seront effectuées sur base d'un réseau de De Bruijn de niveau 6 possédant, donc, 64 nœuds. Ensuite, nous réaliserons une étude sur l'impact de la taille du réseau pour observer les effets du nombre de nœuds sur la congestion.

Nous observerons également l'effet de la contrainte d'injection de paquets sur le taux d'injection effectif pour les algorithmes GSRAD et sur la taille nécessaire des FIFO pour le FIFO-based GSRA. Nous étudierons aussi, les effets d'une diminution de la taille des FIFO afin de minimiser la consommation du module de routage qui sera étudiée au Chapitre 6.

A la fin de ce chapitre, sur base des résultats observés, nous proposerons de combiner le FIFO-based GSRA avec le Priority-deviation-based GSRAD afin d'obtenir un algorithme de routage performant tant au point de vue de la consommation que de la latence. Dans la suite de ce travail, cet algorithme sera désigné sous le nom de FIFO-based GSRAD avec Priority-deviation. Nous le comparerons ensuite avec un mesh baseline possédant un routage déterministe XY.

5.1 Étude statistique

Dans le cadre d'études de tels systèmes, il n'est pas possible de vérifier le comportement du réseau pour l'ensemble des cas de Figure par manque de ressources et de temps. En effet, dans le cas d'un réseau de De Bruijn de niveau 3, les 8 nœuds peuvent envoyer un paquet aux 7 autres nœuds. Ainsi il existe 56 trajets différents comprenant en moyenne 3 nœuds intermédiaires. En admettant que le paquet est seul sur le réseau, le paquet peut se trouver à un nœud dans un de ces 168 états. Une analyse complète du réseau demanderait une étude de l'ensemble des possibilités avec "n" paquets sur le réseau, c'est-à-dire $\sum_1^8 168^n$ possibilités. De plus, dans le cas de déviation, il faut y ajouter la possibilité que le paquet soit dévié et donc la probabilité de rencontrer un paquet avec un des 168 états. Dans le cas de la solution utilisant des FIFO, il faut ajouter les possibilités dues aux nombres de données présentes dans les FIFO. Au total, le nombre de possibilités est extrêmement important.

Dans ces conditions, l'analyse complète d'un réseau demanderait une méthodologie complexe et des ressources très importantes. Face à ce problème, notre choix s'est porté sur une analyse statistique du problème. Elle permet de développer un modèle probabiliste des performances moyennes de notre réseau.

5.1.1 Paramètres de simulation

L'une des possibilités pour une étude statistique de tels réseaux, est de générer aléatoirement des paquets au sein d'un nœud du réseau à destination d'un autre nœud choisi au hasard. A cette fin, chaque nœud est composé d'un module de routage relié à un module de génération de paquets (Fig.5.1). Ces modules sont chargés de produire à chaque nœud du réseau, des paquets possédant une destination aléatoire afin qu'ils soient transmis. De plus le module est implémenté pour que la génération envoie en moyenne un taux de paquets déterminé pour contrôler l'activité moyenne du réseau.

```
if((rand() % 100) >89) // condition verifiee 1 fois sur dix en moyenne
{
        // Generation aleatoire de l'adresse differente de celle du noeud
        int rand = rand() % NBR_NODE;
        while(rand==noma) rand = rand() % NBR_NODE;
        // Envoie de la donnee 42 a l'adresse rand
        adr = rand; data = 42;
        memwrite = 1;
}
else memwrite = 0;
```

FIGURE 5.1 – Module de génération aléatoire de paquets

Dès lors que la génération aléatoire de données est implémentée, il s'agit de rassembler un nombre de résultats suffisants pour une analyse statistique. Dans ce but, l'information véhiculée dans le paquet sert de collecteur d'informations. En effet, une partie de la donnée est utilisée comme compteur qui s'incrémente durant la transmission afin de connaître la latence du paquet. De plus, une partie de la donnée comptabilise le nombre de déviations subies par le paquet (Fig. 5.2). L'information est ensuite collectée dans un fichier texte lors de son arrivée à destination.

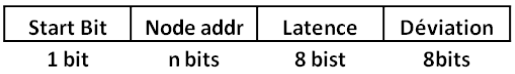


FIGURE 5.2 – Découpage d'un paquet de données : (a)Start Bit, (b)Adresse de destination, (c)Compteur de la latence et (d)Nombre de déviations

Pour atteindre un nombre suffisant de données afin de valider l'étude, le nombre de paquets transmis au sein du réseau suit l'équation 5.1 où n représente le niveau du réseau de De Bruijn et r, le taux de génération en pour cent.

$$P = 2^n.r.10^7 \tag{5.1}$$

Ainsi, dans un réseau de niveau 3 pour un taux de transmission d'un paquet tous les 100 coups, chaque nœud transmet 10.000 paquets soit un total de 80.000 transmissions pour l'ensemble du réseau.

5.1.2 Étude de la distribution statistique

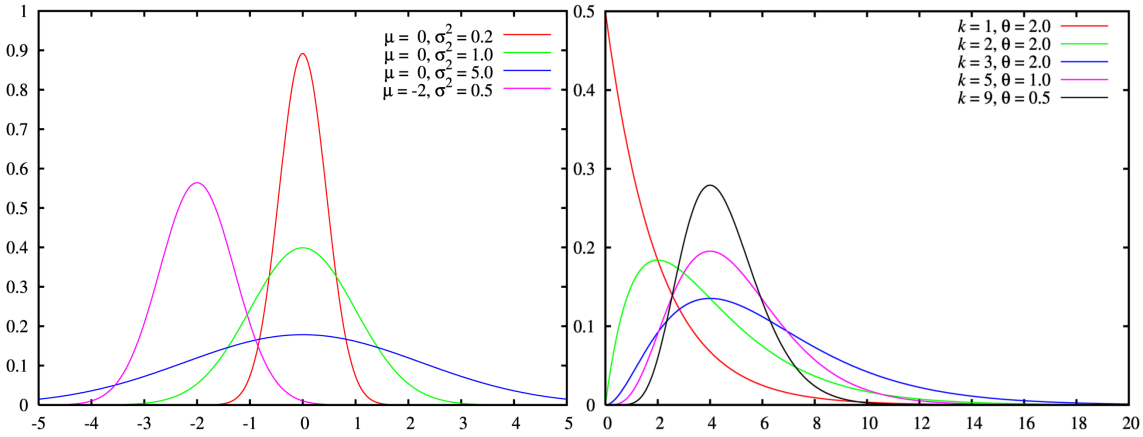
Dans le cadre d'une étude statistique[37], il est intéressant de relier les données obtenues à une courbe de distribution statistique possédant des métriques dont les variations représentent une évolution concrète de l'échantillon. Par exemple, dans le cas d'une distribution selon la loi normale(Fig. 5.3a), la valeur de la moyenne, μ, et de l'écart-type, σ, permettent de quantifier respectivement où se trouve en moyenne la population et quelle est la distance qui sépare un individu de cette moyenne.

Il existe plusieurs lois de distributions statistiques qui peuvent être utilisées dans des conditions précises. Dans le cas de l'étude de la latence au sein d'un réseau, il est difficile d'interpréter les données suivant une distribution statistique. En effet, la latence étant toujours strictement positive, il nous faut trouver une distribution unilatérale ce qui n'est déjà pas le cas de la loi normale. De plus, comme ce sera confirmé plus tard, il est possible que la distribution soit dissymétrique. Il existe bien sûr diverses distributions unilatérales dissymétriques comme par exemple la loi gamma (Fig. 5.3b) qui possède deux paramètres (k et θ) représentatifs de sa forme. Malheureusement elles n'offrent pas des degrés de liberté suffisants pour être utilisées dans notre cas.

Dans le cas où aucune distribution statistique cohérente ne peut être trouvée, la solution est de revenir aux métriques de base de la statistique à savoir la moyenne, la médiane et l'écart-type (μ, m, σ). Ces trois indicateurs primaires peuvent déjà donner des conclusions intéressantes sur l'évolution de la latence au sein du réseau.

Dans le cadre d'une distribution unilatérale dissymétrique, la moyenne permet de connaître comme son nom l'indique où se trouve en moyenne un individu. C'est le paramètre le plus important car c'est lui qu'il faut minimiser afin d'augmenter les performances du réseau. Néanmoins la moyenne ne permet pas d'obtenir une vision d'ensemble de l'évolution de la population.

L'écart-type et la médiane permettent de remédier à ce problème. Ainsi l'écart-type donne une image de la distance qui sépare, en moyenne, un individu de la moyenne. Il mesure donc l'affaissement de la courbe. Plus la courbe est affaissée, plus l'écart-type est grand. L'écart-type quantifie l'étalement des données.



(a) Loi normale (b) Loi gamma

FIGURE 5.3 – Exemples de loi de distributions statistiques

Dans une distribution dissymétrique, les données pouvant se propager de zéro jusqu'à l'infini, il est possible que quelques données très éloignées fassent augmenter la moyenne et l'écart-type. La médiane est alors très importante car elle mesure où se situe le pic de données et n'est donc pas influencée par les données extrêmes. S'il existe quelques données très extrêmes, la moyenne et l'écart-type augmenteront mais la médiane restera fixe. Elle permet donc de quantifier la dissymétrie d'une distribution unilatérale. Sur la Figure 5.4, l'écart entre la moyenne et la médiane est plus important à gauche et nous observons bien que la dissymétrie y est plus marquée.

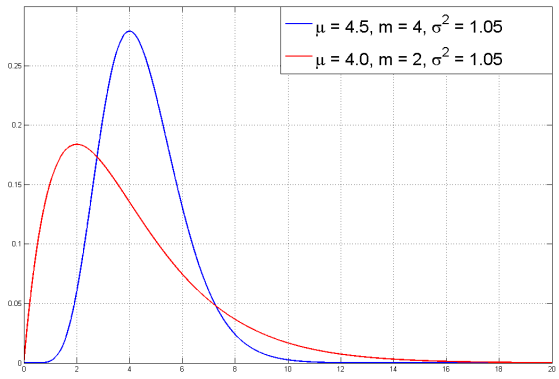


FIGURE 5.4 – Distributions statistiques possédant une moyenne et un écart-type semblables mais des médianes différentes.

Dans notre cas, le but est d'atteindre trois objectifs. Le premier est d'obtenir une distribution avec une moyenne la plus faible possible pour minimiser l'énergie moyenne nécessaire pour une transmission. Le deuxième est de parvenir à combiner cette moyenne à un faible écart-type pour minimiser le délai dans lequel la plupart des paquets sont transmis. Le troisième objectif est de rapprocher la médiane le plus près possible de la moyenne, c'est-à-dire, de minimiser le nombre de paquets qui voyagent pendant un temps très long dans le réseau.

5.2 Analyse comportementale des GSRAD

5.2.1 Distribution de la latence

Cette section permet d'évaluer si les différentes solutions proposées ont un impact concret par rapport au random GSRAD(Fig. 5.5a) qui dévie les paquets aléatoirement.

le Tableau 5.1 rapporte, immédiatement, une vision plus claire de l'impact des différentes implémentations. En effet, nous pouvons constater que le Proximity-priority-based GSRAD permet à plus grand nombres de paquets d'arriver plus vite car sa médiane est plus petite que celle du random GSRAD. Néanmoins, nous pouvons observer que leur moyenne et leur écart-type sont assez similaires ce qui signifie que la diminution de la médiane s'est faite au détriment des valeurs extrêmes qui ont été repoussées plus loin. En d'autres termes, un plus grand nombre de paquets sont arrivés sans être déviés mais ceux qui ont été déviés, l'ont été plusieurs fois avant d'arriver à leur destination. Cela s'observe également sur la Figure 5.5 où on voit bien que la médiane est à 5 tandis que les valeurs tendant vers de grands nombres possèdent un pourcentage plus élevé, ce pourcentage ne tombant en dessous de 0.001% qu' à partir de 38 cycles.

Dans le cas du Deviation-priority-based GSRAD, nous pouvons observer l'inverse. Ce sont la moyenne et l'écart-type qui diminuent par rapport au random GSRAD tandis que la médiane reste inchangée. Cette observation signifie que les valeurs extrêmes ont été réduites. Nous pouvons expliquer ce constat par le fait que le Deviation-priority-based GSRAD empêche un paquet d'être dévié plusieurs fois mais qu'en contre partie, il dévie plus de paquets au moins une fois. On constate nettement cette tendance si on regarde le nombre de cycles à partir duquel le pourcentage descend en dessous de 0.001% qui chute à 25 cycles.

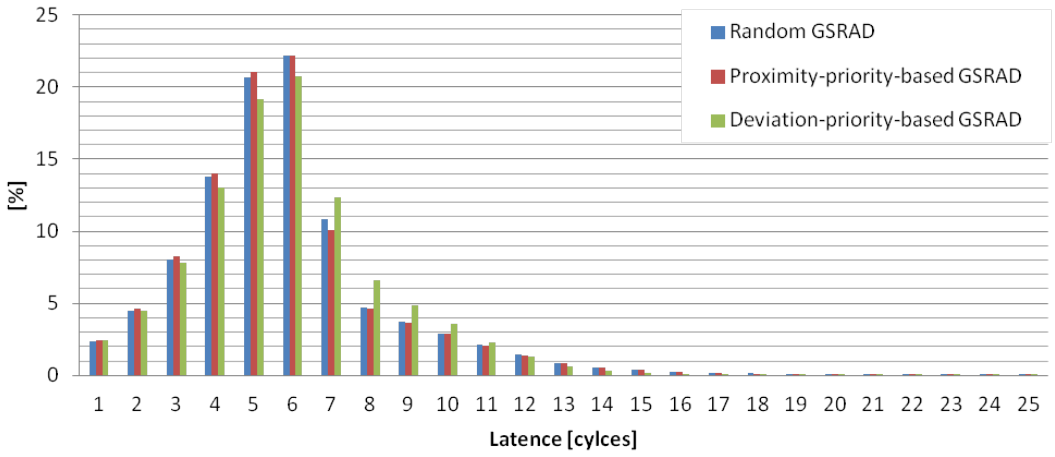
5.3 Evolution du taux effectif d'injection

Le taux effectif d'injection est le taux de paquets réellement injectés dans le réseau. En effet, dans le cas des implémentations utilisant la déviation de paquets(GSRAD), le *processing element* attend avant d'injecter le paquet sur le réseau lorsqu'un conflit du cas 3 ou 5 se produit. Ce processus de stall fausse le taux réel d'injection. Il est donc important de pouvoir l'étudier car il produit une perte de puissance de calcul dans les *processing element*.

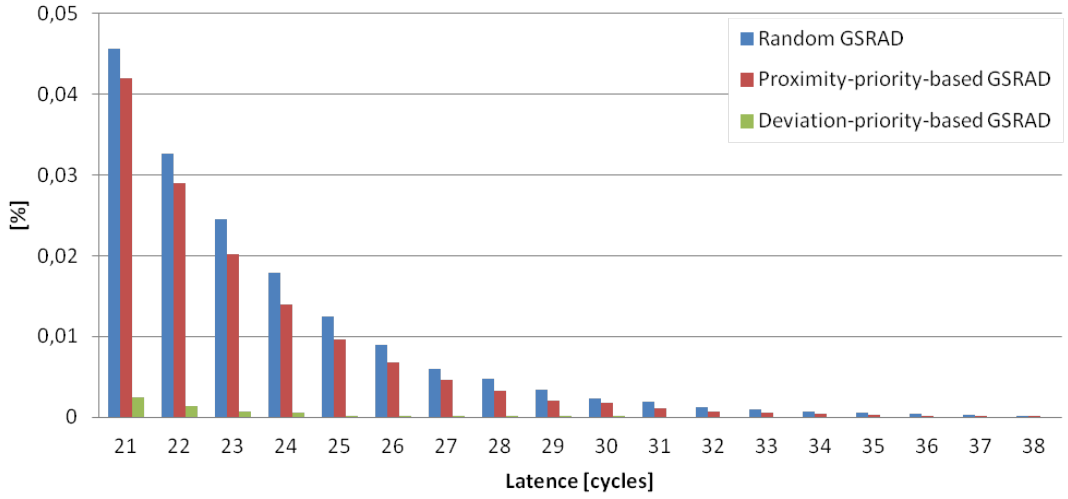
Sur la Figure 5.6, nous distinguons clairement cette perte de puissance lorsque la contrainte d'injection dépasse les 10%. A partir de là, le réseau n'arrive plus à résorber le trafic et le taux d'injection effectif s'effondre. A 20% de contrainte d'injection, le taux réel descend à 11% soit la moitié de la contrainte exigée. Par ailleurs, l'extrapolation de la courbe montre que le taux d'injection effectif du réseau a tendance à se stabiliser autour de cette valeur. Cette perte est inhérente à l'algorithme de routage GSRAD et l'ajout de priorité ne permet pas d'en diminuer l'effet. En effet, sur la Figure 5.6, nous ne distinguons qu'une très faible différence entre les 3 priorités.

TABLE 5.1 – Paramètres statistiques des distributions de la latence pour les GSRAD

	μ [cycles]	m [cycles]	σ^2 [cycles]	$< 0.001\%$ [cycles]
Random GSRAD	5.85	6	2.75	35
Proximity-priority-based GSRAD	5.8	5	2.72	38
Deviation-priority-based GSRAD	5.35	6	2.51	25



(a)



(b)

FIGURE 5.5 – Histogramme de la latence pour le routage GSRAD avec un taux d'injection de 10%

De plus sur la Figure 5.7, nous pouvons observer que la latence avant injection sur le réseau a tendance à augmenter avec la contrainte d'injection. Dans les cercles intérieurs (1% et 50% de contrainte d'injection), la quasi totalité des paquets demandant un stall du *processing element*, ne dure qu'un seul cycle. Néanmoins, lorsque la contrainte d'injection augmente de plus en plus de paquets demandent une mise en attente du *processing element* durant un nombre de cycles plus grand avant de pouvoir être injectés. A 20% de contrainte d'injection, 4% des paquets prennent plus de 5 cycles avant de pouvoir être injectés sur le réseau. Les valeurs maximales observées atteignent jusqu'à 50 cycles pour un nombre infimes paquets.

Ces observations montrent clairement que pour des taux d'injections élevés, le routage d'un réseau de De Bruijn avec déviation et un *stalling processing*, a tendance à diminuer l'activité des *processing element*s afin de pouvoir résorber le trafic présent sur le réseau entrainant une perte de puissance de calcul des *processing element* conséquente. Il est cependant important de noter que le réseau reste stable même si la contrainte d'injection sature le trafic. L'ensemble des paquets sont ainsi délivrés sans aucune perte lors des transmissions. Il n'est donc pas nécessaire d'implémenter un mécanisme de *control flow* du réseau pour d'éventuels cas de saturation.

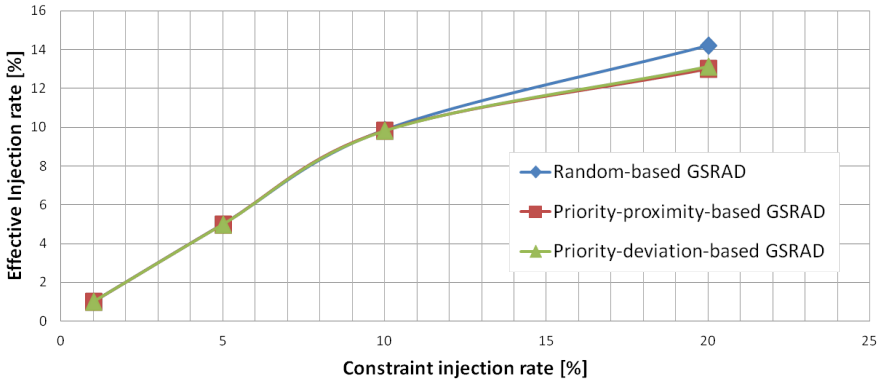
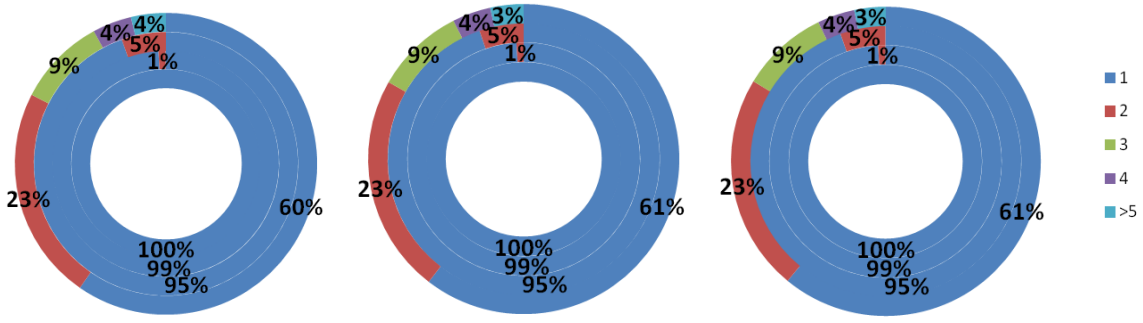


FIGURE 5.6 – Taux d'injection effectif en fonction de la contrainte d'injection pour les routages GSRAD



(a) Random-based (b) Priority-proximity-based (c) Priority-deviation-based

FIGURE 5.7 – Histogramme des délais de mise en attente du processeur avant injection d'un paquet en fonction de contrainte sur le taux d'injection (du centre vers l'extérieure : 1%, 5%, 10%, 20%)

5.4 Analyse comportementale du FIFO-based GSRA

5.4.1 Distribution de la latence

Dans le Tableau 5.2, nous pouvons observer les paramètres de distribution de la latence pour le FIFO-based GSRA. Le premier constat est que ces paramètres sont nettement meilleurs que ceux des GSRAD avec une latence moyenne de 4.68 cycles contre 5.35 pour le meilleur des GSARD. De même, l'écart-type est divisé pour un facteur 2.2x. Enfin moins de un millième des paquets sont transmis avec plus 15 cycles contre 25 pour le Priority-deviation-based GSARD. Ce dernier constat est le fait que la moyenne est plus faible que la médiane et que leur écart est réduit. Le nombre de paquets possédant une latence de transmission importante, est donc limité dans le cas du FIFO-based GSRA.

Toutefois, le nombre de registres implémentés dans les FIFO lors des simulations est très élevé de telle sorte qu'aucune FIFO n'est en état de saturation ce qui n'est pas réaliste. Il est donc nécessaire de relativiser ces chiffres comme nous allons le montrer dans les sections suivantes.

	μ [cycles]	m [cycles]	σ^2 [cycles]	$< 0.001\%$ [cycles]
FIFO-based GSRA	4.68	5	1.2	15

TABLE 5.2 – Paramètres statistiques de la distribution de la latence pour le FIFO-based GSRA

5.4.2 Impact de la taille des FIFO

La métrique intéressante dans le cadre du routage FIFO-based GSRA est, donc, le nombre maximum de registres au sein des FIFO qui ont été nécessaires lors des simulations pour assurer toutes les transmissions. En effet, comme mentionné dans la Section 4.6, lors des simulations comportementales, le nombre de registres disponibles est quasi infini. Il n'est, cependant, pas judicieux d'implémenter un aussi grand nombre de registres par la suite pour deux raisons. Premièrement, un registre consomme beaucoup d'énergie[23] et leur multiplication entrainerait un accroissement conséquent de la consommation. Deuxièmement ceux-ci prennent beaucoup de place, ce qui compliquera encore plus les prochaines étapes de conception(Section 7.4.2). D'un autre coté, les FIFO doivent posséder un nombre suffisant de registres pour assurer les transmissions. En effet, dans le cas où une FIFO remplie devrait accueillir un paquet supplémentaire, ce paquet serait perdu. Il y a donc un trade-off à trouver entre performances et coût énergétique.

Taux d'injection	1%	5%	10%	20%	30%
Nombre de registres	3	5	7	12	-

TABLE 5.3 – Nombres maximum de registres utilisés simultanément dans une FIFO pour le routage FIFO-based GSRA en fonction de la contrainte d'injection

Dans le Tableau 5.3, nous pouvons observer, en fonction de contrainte sur le taux d'injection, le nombre maximum de registres qui ont été utilisés simultanément dans une FIFO. Cela signifie, qu'à aucun moment de la simulation, une FIFO n'a contenu simultanément plus de paquets que le nombre indiqué dans le Tableau.

On remarque immédiatement que plus la contrainte d'injection est élevée plus le nombre de registres nécessaires augmente. Cela est logique dans le sens où il n'y a pas de mise en attente du *processing element*. Chaque nouveau paquet injecté dans le réseau, est envoyé directement dans les FIFO. Cependant ce tableau n'assure pas que ce nombre est le nombre suffisant de registres qu'une FIFO doit posséder pour assurer toutes les transmissions à un taux d'injection donné. En effet, ces chiffres sont statistiques et non absolus. Ils assurent que pour la plupart des cas concernés, ce nombre de registres est suffisant mais il est possible que ce nombre de registres soit insuffisant dans certains cas non traités lors des simulations.

De plus, une autre observation vient de la dernière colonne. En effet pour une contrainte d'injection de 30%, le nombre maximum de registres nécessaires a dépassé la limite fixée lors des simulations qui était de 512 registres. Cette évolution exponentielle est un signe de deadlock. Le réseau ne parvient plus à résorber le trafic et il se retrouve bloqué. Cette observation est importante car dans l'état actuel, l'utilisation de l'algorithme FIFO-based GSRA ne peut pas être utilisé sans un système de *control flow* au cas où les FIFO satureraient.

En conclusion, nous pouvons effectuer l'optimisation de la taille de FIFO sur base du Tableau 5.3, mais cela ne donne pas l'assurance que tous les paquets arrivent à destination. De plus l'utilisation de l'algorithme FIFO-based GSRA comporte le risque de créer un deadlock du réseau et son implémentation sans système de sécurité risqué si le taux d'injection augmente temporairement.

5.5 Comparaison du FIFO-based GSRA et des GSRAD

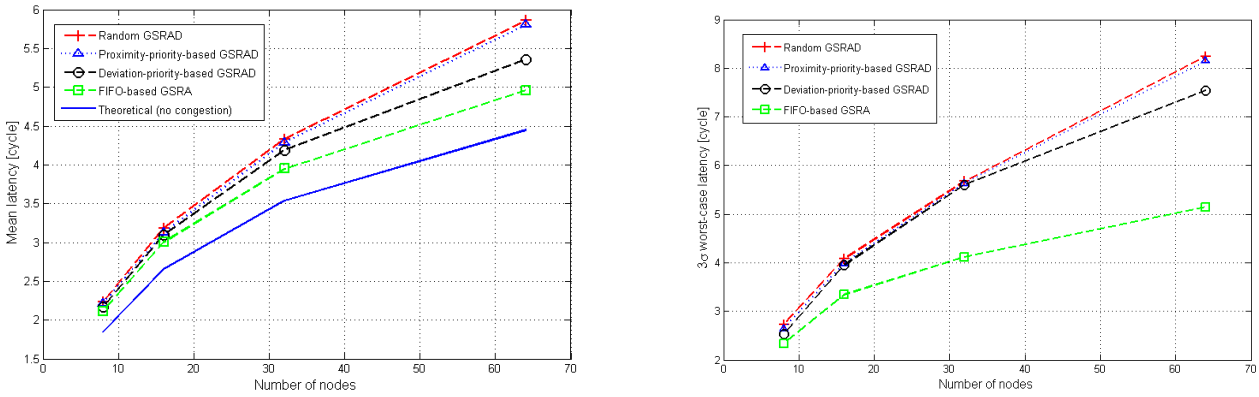
De point de vue des algorithmes GSRAD, les analyses ont montré que les GSRAD possédaient de bonnes performances en terme de latence moyenne et de variance. Mais seule la Priority-Deviation-based GSARD montrait un gain de performance par rapport à la déviation aléatoire. Sa latence moyenne diminue de 10% par rapport à la déviation aléatoire. Nous avons aussi constaté que le nombre de valeurs extrêmes pour la latence diminuait fortement. Moins de un paquet sur cent mille arrivait après 25 cycles tandis que cette valeur atteignait 35 cycles pour la déviation aléatoire et 38 pour la Priority-proximity-based GSRAD.

Néanmoins, nous avons constaté que l'algorithme qui obtient les meilleures performances en terme de latence était le FIFO-based GSRA. Avec une latence de 4.78 et un écart-type de 1.2, le FIFO-based GSRA obtient les meilleures performances avec une diminution de 20% de la latence moyenne par rapport à la déviation aléatoire. Malgré ses performances, le FIFO-based GSRA possède deux désavantages très importants. D'une part, il est difficile d'adapter la taille des FIFO sans connaître le taux d'injection auquel le réseau devra faire face. De plus, l'optimisation de la taille des FIFO ne peut s'effectuer sans compromettre l'assurance qu'un paquet soit transmis. D'autre part, un risque de saturation du réseau existe notamment quand le taux d'injection augmente.

En conclusion, le FIFO-based GSRA est l'algorithme le plus prometteur en terme de latence mais n'est pas utilisable à l'état brut car il est sujet à des pertes de paquets. D'autre part, Le Priority-deviation-based GSRAD ne possède pas les problèmes du FIFO-based GSRA mais sa latence moyenne est plus élevée. Il est, cependant, possible de combiner les deux afin d'obtenir de meilleurs résultats.

5.6 Impact de la taille du réseau sur la congestion

Sur la Figure 5.8a et la Figure 5.8b, nous pouvons observer, respectivement, l'évolution de la latence moyenne et de l'écart-type en fonction de la taille du réseau. Nous constatons que cette évolution de la latence moyenne suit un tracé similaire à la courbe théorique [1]. Cela signifie que quelque soit le GSRA utilisé, la latence évolue de manière logarithmique par rapport au nombre de nœuds. Nous pouvons aussi confirmer que le FIFO-based GSRA obtient les meilleures performances en terme de latence et cela indépendamment de la taille du réseau. Ces performances en sont même d'autant meilleures à mesure que le nombre de nœuds augmente.



(a) (b)

FIGURE 5.8 – Evolution des performances du réseau en fonction de la taille du réseau de Bruijn

5.7 FIFO-based GSRAD avec Priority-deviation

Dans cette section, nous allons donc proposer un algorithme de routage optimisé en combinant le FIFO-based GSRA avec le Priority-deviation-based GSRAD. Le principe de cet algorithme est basé sur le FIFO-based GSRA avec un nombre de registres réduit qui dévie les paquets et met en attente le *processing element* en cas de surcharge des FIFO.

L'utilisation des FIFO avec un nombre de registres réduit a pour but de se rapprocher des performances du FIFO-based GSRA sans augmenter la consommation de celles-ci de manière exagérée. De plus, l'ajout de la déviation et stalling du *processing element* permet non seulement de ne pas perdre de paquets mais aussi de s'assurer que le réseau ne sature pas.

Cet algorithme possède donc une dualité qui le pousse d'un coté vers les performances en latence en moyenne consommatrice d'énergie et de l'autre vers la faible consommation aux performances plus faibles en terme de latence et de puissance de calcul. Dans la suite de ce chapitre, nous allons étudier ces performances en terme de latence moyenne afin de déterminer quel est le nombre optimum de registres que doit posséder les FIFO. Dans le Chapitre 6, une étude de la consommation sera effectuée afin de trouver un trade-off entre consommation et performances avec comme paramètre la profondeur des FIFO.

1. Comme défini à la Section 3.3, la courbe théorique de latence correspond à la latence moyenne d'une transmission pour un paquet se trouvant seul sur le réseau.

Sur la Figure 5.9, lorsque le nombre de registres passe de 8 à 3, nous pouvons constater une diminution très légère de la latence. Cela est induit par le principe de déviation lorsque les FIFO sont pleines. Plus la taille des FIFO est réduite, plus la file d'attente est réduite à son tour. Les paquets en surplus sont alors déviés. Cette manipulation tend à diminuer la latence tant que le gain produit par la réduction de la file d'attente est supérieur au coût des déviations. La latence diminue donc jusqu'à 3 registres par FIFO. A ce moment là, le gain et le coût s'équilibrent puis basculent vers un coût plus important pour un gain moindre lorsque les FIFO ne possèdent plus que un ou deux registres.

De plus, nous pouvons constater que cet effet a tendance à devenir plus extrême lorsque la contrainte d'injection augmente. En effet, nous pouvons constater qu'à 20%, le gain produit est un peu plus prononcé lorsque le nombre de registres passe de 8 à 3. Nous observons surtout que lorsque l'équilibre est dépassé, les pertes en terme de performance sont plus importantes.

Parallèlement à la latence, nous pouvons aussi observer dans le Tableau 5.4 qu'il faut attendre beaucoup plus de cycles avant que le pourcentage de paquets non encore transmis descend en-dessous du millième de pour-cent. Cela est du au fait que lorsque la taille des FIFO diminue, un paquet a de plus en plus de chance de passer par un nœud où les FIFO sont pleines. Et donc, la probabilité d'être dévié plusieurs fois augmente malgré la priorité ce qui entraine une augmentation de la latence de transmission de certains paquets.

Enfin, nous pouvons constater que les *processing elements* ont tendance à être mis de plus en plus en attente lorsque le nombre de registres diminue (Fig. 5.10). Cette tendance est en lien direct avec la probabilité élevée de saturation des FIFO. Toutefois, comme nous pouvons l'observer à la Figure 5.11, la plupart des paquets sont envoyés instantanément et lorsque que cela n'est pas possible, le temps attendu avant l'injection du paquet se résume à quelques cycles. Le *processing element* n'a donc pas de longues périodes d'attente avant de pouvoir injecter un paquet.

Nombre de registres	Taux d'inj.	μ [cycles]	m [cycles]	$< 0.001\%$ [cycles]
8->4 registres	10[%]	4.78	6	16
	20[%]	4.98	6	20
3 registres	10[%]	4.71	6	16
	20[%]	4.79	6	20
2 registres	10[%]	4.82	6	17
	20[%]	6.01	6	26
GSRAD	10[%]	5.30	6	25
	20[%]	8.98	6	38

TABLE 5.4 – Evolution des paramètres statistiques des distributions en fonction du nombre de registres des FIFO

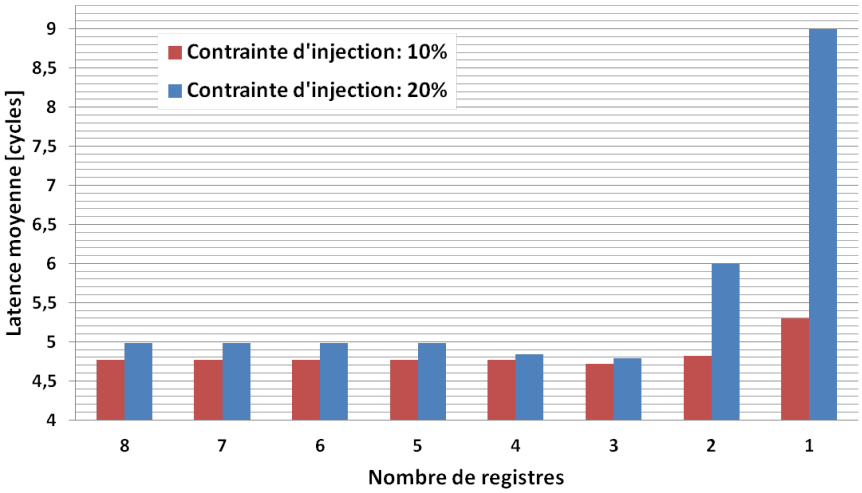


FIGURE 5.9 – Evolution de la latence moyenne en fonction du nombre de registres dans les FIFO

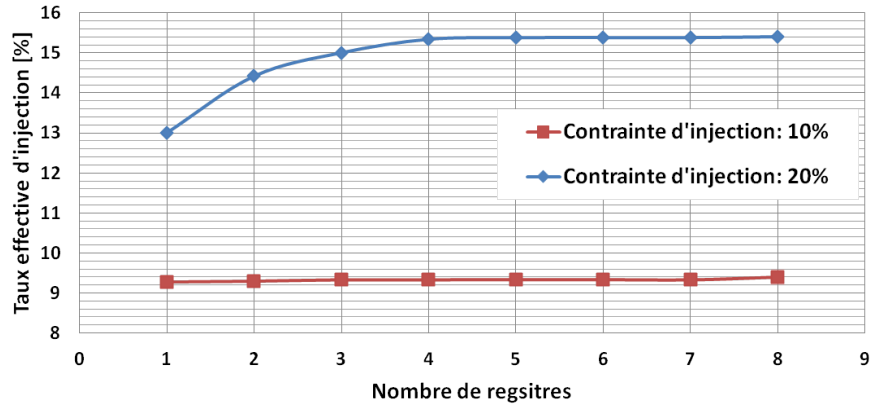


FIGURE 5.10 – Evolution du taux effectif d'injection en fonction du nombre de registres dans les FIFO

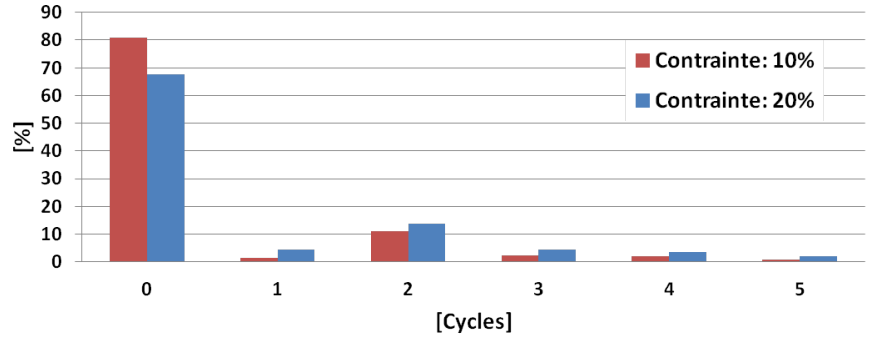


FIGURE 5.11 – Histogramme des paquets en fonction du cycles attendu avant injection

5.8 Comparaison avec un XY baseline mesh

Dans cette section, nous allons maintenant comparer notre solution optimisée à la topologie mesh baseline implémentée avec un routage XY déterministe. Dans la topologie mesh, chaque nœud possède deux identifiants désignant la colonne et la ligne dans lesquels le nœud se trouve. De même, un paquet possède une adresse composée de deux nombres indiquant la colonne et la ligne de sa destination. Le routage XY utilisé est un algorithme qui va tout d'abord envoyer le paquet vers la colonne de destination puis le fera parcourir cette colonne jusqu'à atteindre sa destination. Afin de régler les conflits au sien du réseau, des FIFO aux entrées des nœuds ont été implémentées. De plus, un *control flow* est mis en place pour empêcher la saturation des FIFO et la perte de paquets.

Topologie	Routage	Taux eff. d'inj.	Latence moyenne[cycles]	Nombre moyen de hops
Mesh	XY baseline déterministe	5.39[%]	18.58	5.15
De Bruijn	FIFO-based GSRAD	9.18[%]	4.71	4.65

TABLE 5.5 – Tableau comparatif des performances d'un réseau de De Bruijn avec un routage FIFO-based GSRAD avec Priotity-Deviation et un réseau mesh avec un routage XY baseline déterministe pour une contrainte d'injection de 10 pour-cent pour un réseau de 64 nœuds

Dans le Table 5.5, nous pouvons voir très clairement l'avantage du De Bruijn par rapport au mesh. Tout d'abord, le taux effectif d'injection y est nettement plus élevé. La perte de puissance de calcul est inférieure à 10% dans le réseau de De Bruin alors qu'elle avoisine les 50% dans le mesh baseline. De plus, la latence moyenne est près de 4 fois plus petite (3.95x) dans le réseau de De Bruijn. Cela permet d'implémenter de plus petits *processing elements* moins consommateurs d'énergie mais qui échangent plus d'informations. Le nombre moyen de hops effectués par un paquet est lui aussi plus petit avec un gain de 10%. Cette métrique est importante dans le cas où le coût énergétique de la transmission serait principalement du au chargement des capacités des longs fils d'interconnexions (Sec. 7.1).

En conclusion, du point de vue comportemental, l'analyse d'un réseau de De Bruijn avec le routage optimisé (FIFO-based GSRAD avec Priority-deviation) montre que la topologie de De Bruijn possède des propriétés très intéressantes. La comparaison avec le mesh est nettement à l'avantage de la topologie de De Bruijn avec une diminution d'un facteur 4 sur la latence moyenne. Il est, toutefois, important de relativiser ces chiffres en l'absence de données claires sur la consommation et la surface des FIFO qui constituent un point critique[23].

Chapitre 6

Implémentation logique et synthèse d'un nœud

Ce chapitre traite de la consommation des blocs de routage des différents algorithmes implémentés dans le Chapitre 4. La consommation de la transmission d'un paquet dans un réseau se divise en deux parties[26]. La première est l'énergie attachée aux interconnexions entre les nœuds. Souvent de longueur plus importante, ces interconnexions possèdent une consommation importante due à leur capacité de charge et aux cellules qui les *drive*(Chap. 7). L'autre partie de la consommation est imputée au module de routage lui-même. A longueur d'interconnexions constantes, plus cette composante énergétique du module de routage sera faible, plus le coût énergétique d'une transmission diminuera. Dans cette optique, nous allons étudier l'implémentation et la consommation des trois algorithmes de routage proposés (Proximity-priority-based GSRAD, Deviation-priority-based GSRAD et FIFO-based GSRA). Comme mentionné dans la Section 4.5, le Random-based GSRAD est difficilement implémentable et ne sert que de base d'étude comportementale dans ce travail. Il ne sera donc pas étudié dans ce chapitre.

Le langage SystemC RTL ne pouvant pas être directement traduit en cellules logiques de manière correcte à l'heure actuelle, il est nécessaire d'implémenter les différents modules de routage en langage Verilog afin qu'ils puissent être traduits par les outils de synthèse. Cette implémentation comporte une part d'optimisation car un module bien implémenté, est un module qui consomme moins. La première partie de ce chapitre porte donc sur l'implémentation en Verilog des routeurs.

Par la suite, nous exposerons les résultats de la consommation et de la surface qui ressortent de la synthèse. Afin de pouvoir avoir un ordre de grandeur, nous avons choisi d'attacher à chaque nœud une mémoire de 1kB et un processeur "Cortex M0" 32-bits conçu par l'entreprise *ARM*[19]. Ce sont ces deux éléments qui jouent le rôle de *processing element*. Il nous sera, ainsi, permis de comparer la consommation et la surface des routeurs par rapport à celles du *processing element*. Le choix du Cortex M0 se justifie par le fait qu'il a été optimisé dans le but de servir pour des applications à basse consommation. Il est donc tout indiqué dans le cadre de cette étude.

6.1 Implémentation d'un nœud

Avant d'effectuer la synthèse, il est nécessaire de clarifier les connexions internes d'un nœud et l'interfaçage entre le routeur et le processing element. En effet, l'ajout du *processing element* en Verilog demande une compréhension précise de la communication avec le routeur avant de pouvoir être implémenté.

6.1.1 Architecture interne d'un nœud

Dans une topologie de De Bruijn directionnelle et sans interconnexion de feedback entre les nœuds, les entrées ou *inputs* d'un nœud sont composées des deux interconnexions (in1, in2) provenant de deux autres nœuds. Les sorties ou *outputs*, quant à elles, sont composées aux deux interconnexions se connectant à deux autres nœuds (out1, out2). Le datapath d'un réseau se résume donc à connecter les interconnexions sortantes out1 et out2 aux interconnexions entrantes in1 et in2 des différents nœuds. De plus, comme décrit dans l'introduction du chapitre, un nœud est constitué d'un *processing element* et de son routeur (Fig. 6.1). Dans notre cas, le *processing element* se résume à un Cortex M0 et à sa mémoire de 1kB (à savoir 256 mots de 32-bits).

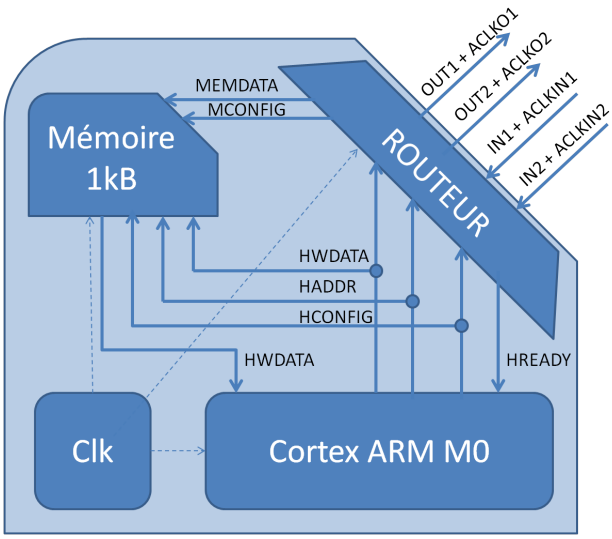


FIGURE 6.1 – Représentation des interconnexions des modules au sein d'un nœud

6.1.2 Générateur de clock et réseau asynchrone

Dans un circuit digital, l'horloge est l'élément qui permet de synchroniser tout le circuit[5]. La plupart des circuits digitaux ne travaillent qu'avec une seule horloge afin de simplifier la synchronisation. Lors de l'étape du placement et routage (Chap. 7), l'horloge va être propagée dans tout le circuit pour atteindre l'ensemble des Flip-Flop afin qu'ils reçoivent le signal d'horloge simultanément. Afin de réaliser cette opération, les outils de placement et routage vont, donc, créer un *clock tree* composé d'inverseurs pour jouer les timings des fils.

Dans le cas de circuits tels que les NoCs, la synchronisation de l'ensemble du circuit peut être un inconvénient[6]. En effet, lors du placement et routage, les outils vont essayer de propager l'horloge dans tout le réseau. La taille importante du circuit augmente la difficulté de l'opération car les outils sont obligés d'ajouter beaucoup d'inverseurs pour propager le signal jusqu'au bout du circuit. Les cellules du *clock tree* viennent, ensuite s'ajouter aux cellules du circuit lors du placement et routage.

La synchronisation d'un réseau sur une seule clock augmente aussi les contraintes de timing[6]. En effet, les NoCs possèdent des interconnexions entre les nœuds de très grandes longueurs car ces fils doivent connecter des portes éloignées les unes des autres. L'utilisation d'une seule horloge impose que la contrainte de timing sur ces connexions soit respectée ce qui augmente la difficulté du placement et routage.

Afin de contourner ce problème, il est possible d'implémenter un réseau asynchrone dans lequel chaque nœud possède sa propre clock locale mais de fréquence similaire à tout le réseau(Fig. 6.1)[39]. La propagation des horloges s'effectue seulement localement limitant ainsi la taille du *clock tree*.

La difficulté de cette solution se situe au niveau des transferts entre régions possédant des clocks différentes. Pour effectuer cette transmission, nous avons choisi d'ajouter l'inverse du signal de l'horloge locale "ACLK" à chaque sortie du nœud. Ce signal permet la synchronisation du paquet une fois transmise au nœud suivant (Fig. 6.2). Lorsqu'un paquet est transmis du nœud "Y" au nœud "Z", ACLK est également transmis. Une fois arrivé au nœud "Z", le paquet rentre dans un Flip-Flop trigguant sur le flanc montant de ACLK. Le signal du paquet est donc maintenu pendant un cycle entier de l'horloge ce qui permet au nœud "Z" de capter le signal avec sa clock locale.

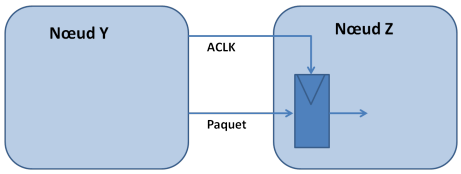


FIGURE 6.2 – Synchronisation de transfert dans un réseau asynchrone

6.1.3 Interface entre le routeur et le *processing element*

La communication entre le processeur et sa mémoire s'effectue grâce à cinq I/O (HADDR, HRDATA, HWDATA, HCONFIG). HRDATA est un input du Cortex M0 qui fournit en permanence au processeur, la donnée enregistrée dans la mémoire à l'adresse HADDR. HWDATA est la donnée que le processeur désire enregistrer à l'adresse HADDR quand HCONFIG prend une valeur spécifique. [1]

Afin de pouvoir communiquer avec le réseau, le *processing element* doit être capable d'envoyer et de recevoir un paquet. La réception d'un paquet s'effectue en implémentant une mémoire dual-port. Ainsi le routeur est capable d'écrire une donnée (MEMDATA) à l'adresse (MEMADDR) selon les informations contenues dans le paquet reçu. L'envoi de paquets s'effectue par le processeur via les I/O HADDR, HWDATA et HCONFIG comme pour la mémoire. Les commandes destinées à la mémoire et celles destinées au routeur se distinguent grâce à la valeur prise par HADDR.

En effet, la mémoire utilisée dans ce travail est de 1kB, soit 256 registres. Huit bits d'adresse sont donc suffisants pour distinguer l'ensemble des registres. Comme HADDR est une connexion de 32 bits,

1. HCONFIG n'est pas défini en tant que tel dans l'instanciation du processeur. Il est la concaténation des inputs HSIZE, HTRANS et HWRITE.

il est donc possible d'utiliser les autres bits pour différencier les commandes destinées à la mémoire et celles destinées au routeur selon la topologie de la Figure 6.3.

Le premier bit de l'adresse détermine si le processeur parle à sa mémoire ou au routeur. Les "n" bits suivants sont consacrés à l'adresse du nœud vers lequel la donnée doit être envoyée si le premier bit est à un. Comme la longueur de l'adresse des nœuds dans le réseau dépend du niveau de celui-ci, le nombre de bits y consacrés, change également. Enfin, les 8 derniers bits représentent l'adresse du registre de la mémoire (du nœud ou d'un autre) dans lequel la donnée doit être enregistrée.

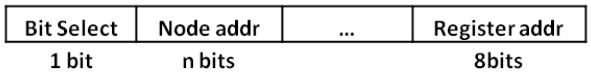


FIGURE 6.3 – Structure du signal HADDR

Il est aussi à noter que, dans le cas où le processeur peut attendre avant qu'un de ses paquets soit injecté sur le réseau, il faut également relier l'input HREADY au routeur. Le passage de celui-ci à un, permet de mettre instantanément le processeur en attente. Pour finir, il existe des I/O supplémentaires qui ne sont pas utilisés ici mais qui permettent l'ajout de fonctionnalités supplémentaires si besoin est.

6.1.4 Architecture des paquets

L'architecture des paquets est importante car minimiser la taille de ceux-ci permet de diminuer la consommation d'un hop. Sur la Figure 6.4, nous pouvons observer la structure d'un paquet.

Le premier bit sert de *bit de start*. Ce dernier est toujours à un dans un paquet. En effet, comme nous pourrons le constater, il est possible qu'un paquet soit composé uniquement de zéros. Dans ce cas-là, sans bit de start, le routeur est incapable de déterminer s'il est confronté un paquet ou à une absence de paquet. Les "n" bits suivants sont consacrés à l'adresse du nœud de destination du paquet. Dans un réseau de De Bruijn, le nombre de bits consacré à l'adressage des nœuds varie linéairement avec le niveau du réseau. Les 8 bits suivants définissent dans quel registre de la mémoire du nœud de destination doit être enregistrée la donnée. Les 32 derniers bits constituent la donnée en tant que telle que les *processing elements* désirent s'échanger.

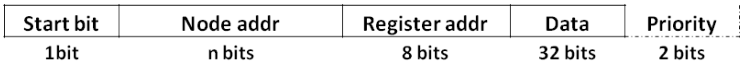


FIGURE 6.4 – Structure d'un paquet dans un réseau de De Bruijn

On remarque, tout de suite, qu'il est possible d'avoir un paquet composé uniquement de zéros si un nœud quelconque désire enregistrer la donnée zéro dans le registre zéro de la mémoire du nœud zéro. Cependant, dans le cadre d'une optimisation agressive de la consommation, il est possible de supprimer le bit de start en empêchant les processing elements d'écrire dans le registre zéro des mémoires des autres nœuds.

Il est à noter que dans le cas des GSARD, un certain nombre de bits doit être ajouté afin de communiquer le degré de priorité du paquet. Dans notre cas, nous nous contenterons de 2 bits permettant une hiérarchisation des paquets sur 4 niveaux.

6.2 Implémentation du routeur

Après avoir clarifié le fonctionnement interne du nœud, nous allons maintenant nous pencher sur l'implémentation en Verilog des différents routages proposés afin de pouvoir étudier leur consommation. Une partie des codes Verilog utilisé pour le routage d'un réseau de De Bruijn de niveau 4 sont présents à l'annexe D.

6.2.1 Implémentation du module GSRA

Comme expliqué dans la Section 4.2, le GSRA utilise une comparaison entre les suffixes du nœud et les préfixes de l'adresse de destination du paquet. Le nombre de préfixes et de suffixes étant une fonction du niveau du réseau, la taille de ce module varie en fonction de celui-ci. La génération du Verilog en fonction du niveau du réseau s'effectue de la même manière qu'en SystemC de la Section 4.1. Le code Verilog pour le module GSRA pour un réseau de De Bruijn de niveau 4 est présent à l'annexe D.2.

6.2.2 Implémentation des GSRAD

L'implémentation des priorités se traduit par l'ajout d'un module de contrôle chargé de résoudre les conflits. Le module de routage se divise donc en deux modules GSRA liés aux 2 entrées du nœud et au module de contrôle. Ce dernier choisit quel paquet est dévié en fonction du nombre de déviations subies par les paquets ou de la proximité de leur destination selon que le routage implémenté soit le Priority-deviation-based GSRAD ou le Priority-proximity-based GSRAD. Comme nous pouvons le constater sur la Figure 6.5a, l'avantage de ces solutions est la grande simplicité d'implémentation.

6.2.3 Implémentation du FIFO-based GSRA

L'implémentation du FIFO-based GSRA se complique par rapport au GSRAD. En effet, il faut ajouter des FIFO devant posséder trois entrées et une sortie. Les entrées étant obligatoires dans le cas où les deux paquets entrants (in1, in2) et un paquet du *processing element* désirent partir dans la même direction au même moment. Chaque FIFO doit donc être capable d'enregistrer un, deux ou trois paquets simultanément. Le module de contrôle a donc pour fonction d'indiquer combien de paquets se présentent aux entrées des FIFO pour être enregistrés. Le choix d'implémenter les FIFO en sortie du nœud plutôt qu'en entrée est justifié. En effet, si les FIFO avaient été placées aux deux entrées, une troisième FIFO aurait été nécessaire sur la connexion HWDATA entre le *processing element* et le routeur augmentant d'autant plus la consommation.

Afin de faciliter l'implémentation de FIFO, l'enregistrement des données se déroule toujours à partir de l'entrée 1 vers l'entrée 3 de la FIFO. Ainsi lorsque deux paquets désirent être enregistrés, les multiplexeurs placés devant les FIFO et commandés par le module de contrôle, placent les deux paquets aux entrées 1 et 2 de la FIFO (Fig. 6.5b).

6.2.4 Implémentation du FIFO-based GSRAD with Priority-deviation

L'implémentation du FIFO-based GSRAD avec Priority-deviation est plus complexe qu'il n'y paraît. En effet, l'ajout de la priorité sur le nombre de déviations entraîne une complexification importante de la gestion des entrées des FIFO. Cette complexification est déjà perceptible lors de l'implémentation du FIFO-based GSRA (Fig. 6.5b) avec l'ajout d'un module de contrôle et de multiplexeurs devant les FIFO.

Afin de contourner la difficulté, nous avons choisi de placer les FIFO aux entrées du nœud. De ce fait, chaque FIFO n'a plus qu'une entrée et qu'une sortie ce qui est plus facile à implémenter. L'idéal aurait été de placer également une FIFO entre le routeur et le *processing element*. Néanmoins, afin de diminuer la consommation, nous avons choisi de ne pas ajouter de FIFO entre le module de routage et le *processing element*. Dès lors, en cas de conflit avec un paquet présent dans les FIFO des entrées in1 et in2, le *processing element* est mis en attente jusqu'à injection de son paquet.

Le schéma d'implémentation est celui présenté à la Figure 6.5c. Lorsqu'un paquet arrive via une entrée (in1, in2) du nœud, celui-ci est envoyé dans la FIFO. Pendant ce temps, le contrôleur reçoit la direction désirée pour les deux paquets situés à la sortie des FIFO. Si aucun conflit n'existe, chaque paquet est envoyé dans sa direction. Si un conflit existe, seul un paquet est envoyé. Dans le cas où une des deux FIFO est pleine, cette dernière informe le contrôleur. Celui-ci envoie donc les deux paquets situés en sortie des FIFO en déviant l'un des deux selon le nombre de déviations déjà subies par chaque paquet.

6.3 Résultats de la synthèse d'un nœud

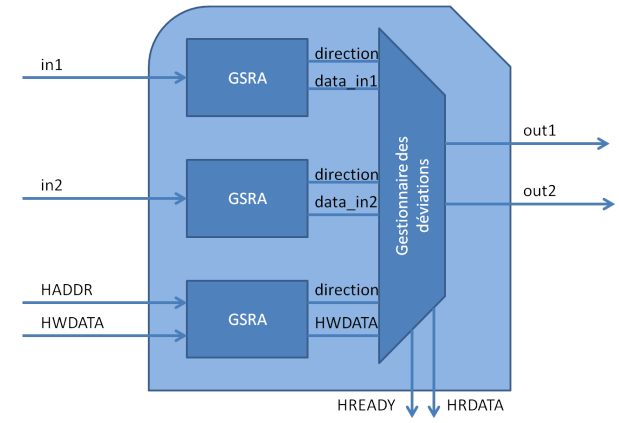
Nous allons, maintenant, comparer la consommation requise pour les différents modules de routage post-synthèse sur base de l'énergie par rapport à la consommation d'un Cortex M0. Nous en profiterons également pour observer la surface utilisée par les différents modules de routage. Cette information sera un élément important lors de la phase de placement et routage (Chap. 7) car plus la surface sera petite, plus la distance entre le nœud sera réduite.

La synthèse d'un nœud pour chaque algorithme de routage proposé est effectuée grâce à l'outil de synthèse Design Vision de Synopsys[38]. La technologie utilisée est la technologie ST65 de STMicroelectronics [2] qui permet de réaliser des transistors de 65nm de longueur de canal. Du point de vue des contraintes, la fréquence de fonctionnement a été fixée à 50MHz.

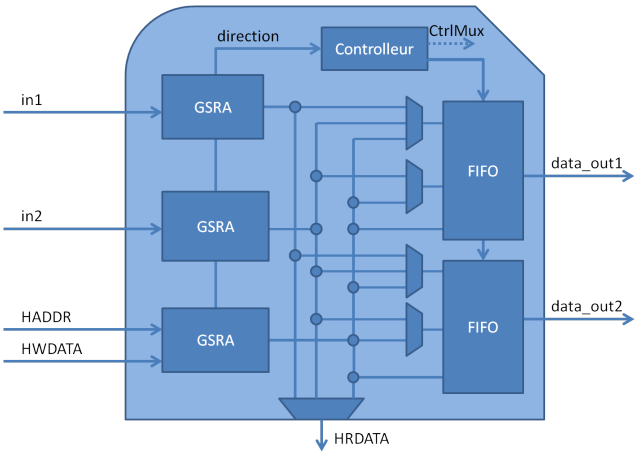
le Tableau 6.1 donne les données relatives à la consommation des algorithmes de routage(FIFO-based GSRA, Priority GSRAD, FIFO-based GSRAD avec priorité). Nous pouvons y observer la répartition de celle-ci entre le module de routage proprement dit, le switch fabric et les FIFO. Dans ce tableau, l'analyse combinatoire du GSRA ainsi que les compteurs incrémentant les paquets déviés(Priority-deviation) et la comparaison des proximités(Priority-proximity) sont considérés faisant partie du module de routage. Le *switch fabric* englobe la matrice de commutation reliant les inputs aux registres des outputs (ou aux FIFO)[5]. La consommation des FIFO se calcule sur la base de la consommation des FIFO proprement dites et du module de contrôle qui leur est dédié.

Nous pouvons directement observer que les FIFO sont une source de consommation importante. Cette observation confirme celles effectuées par M. Hosseinabady[23] dans le cadre de son implémen-

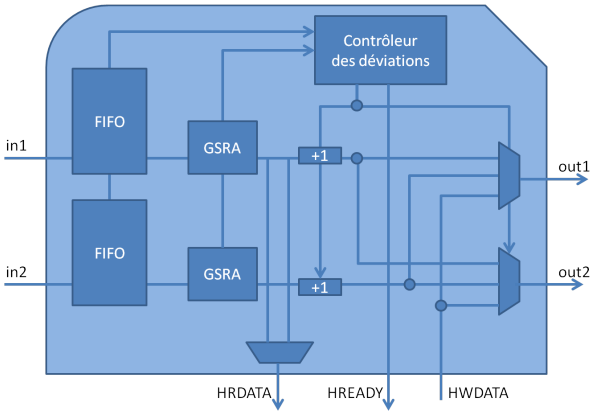
2. STMicroelectronics : http :/www.st.com



(a) GSRAD



(b) FIFO-based GSRA



(c) FIFO-based GSRAD avec Priority-deviation

FIGURE 6.5 – Schéma d'implémentation des routeurs proposés

tation de réseau de De Bruijn bidirectionnel. Le pourcentage de l'énergie attribuée aux FIFO est plus élevé dans notre cas puisque notre module de routage se résume à de l'analyse combinatoire de base. A la Figure 6.7b, nous pouvons constater que la consommation des FIFO représente 40% de la consommation du nœud. Le coût de la connexion au réseau est donc très importante dans le cas d'un routage FIFO-based GSRA. Nous pouvons constater que cette consommation diminue logiquement avec le nombre de registres que possède une FIFO (Fig. 6.6). Néanmoins même pour 3 registres, la consommation des FIFO ne descend pas en dessous des 20% de la consommation du nœud (Fig. 6.7c).

A l'inverse, la consommation totale des GSRAD est très faible vu l'absence des FIFO. On remarque cependant une augmentation de la consommation du module GSRA en tant que tel pour des GSRAD par rapport à celui du FIFO-based GSRAD avec Priority-deviation. Cela se justifie par la présence des compteurs de déviations qui incrémentent le paquet en cas de déviation. Au Tableau 6.1, nous observons également que la Swicth fabric possède une consommation plus importante que le module de routage pour le GSRAD. Cela s'explique par le fait que le Switch Fabric contient les registres de sorties alors que le module GSAR n'est composé que d'analyse combinatoire. Enfin, lorsque nous comparons la consommation du routage GSRAD et de la Switch Fabric à celle du Cortex M0, nous constatons que le coût de connexion au réseau est faible(<10% de la consommation de l'ensemble du nœud).

Du point de vue de la consommation du module de routage dans sa globalité, les algorithmes GSRAD sont de très faibles consommateurs par rapport au FIFO-based GSRAD avec Priority-deviation. Les performances en terme de latence moyenne ont donc un prix énergétique élevé. Néanmoins il est possible d'atténuer cette consommation en diminuant le nombre de registres implémentés (FIFO-based GSRAD avec Priority-deviation). En effet, si nous considérons que 3 registres est l'optimum au point de vue des performances de routage, la consommation du routeur s'approche des 20% de la consommation globale et représente plus de la moitié de la consommation du routeur (Tableau 6.1).

Si nous observons la surface à la Figure 6.8, nous pouvons constater que la surface consacrée aux FIFO est dans les mêmes proportions que la consommation tandis que le reste du routeur ne demande qu'une surface limitée. Nous remarquons néanmoins que la diminution du nombre de registres à trois unités permet de restreindre la surface à 13% de la surface totale du nœud dans le cadre du FIFO-based GSRAD avec Priority-deviation.

En conclusion, il est clair que le FIFO-based GSRA est l'algorithme routage le plus performant en terme de latence. En diminuant la taille des FIFO au nombre optimum de trois regsitres et en intégrant une priorité aux paquets déviées (FIFO-based GSRAD avec Priority-deviation), nous avons donc pu maintenir les performances du FIFO-based GSRA tout en diminuant la consommation et la surface du routeur afin d'obtenir un algorithme performant pour les applications à basse consommation.

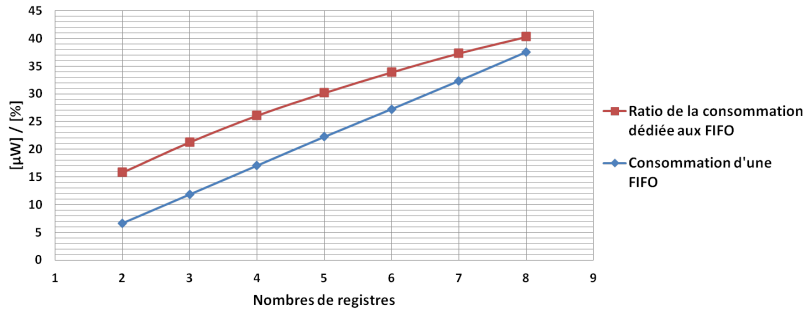
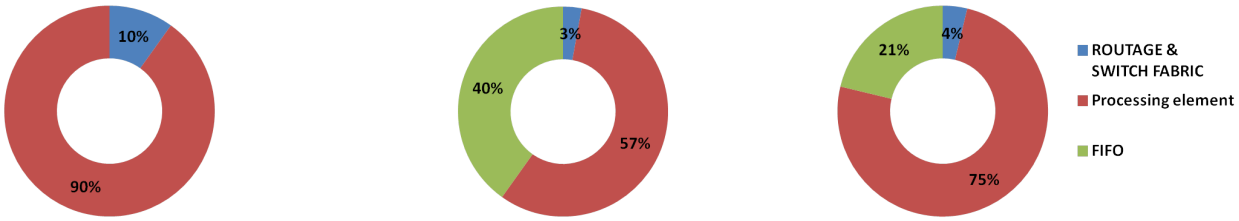


FIGURE 6.6 – Evolution de la consommation en fonction du nombre de registres dans les FIFO

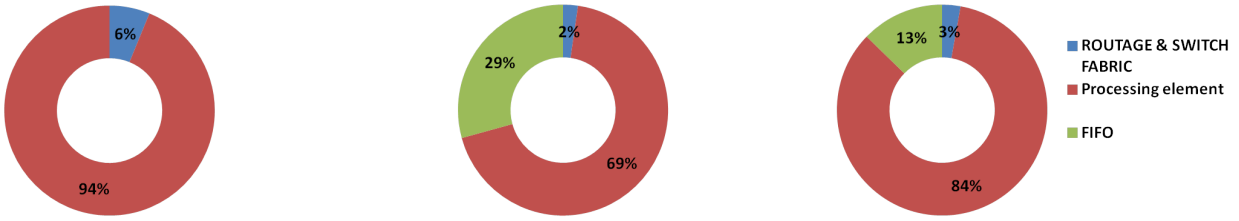
Type de routage	Unité	Routage	Switch fabric	FIFO	Total
FIFO-based GSRA (8 registres)	[μW]	1.39	1.07	42.2	**45.1**
		3%	2.4%	94.6%	100%
FIFO-based GSRAD(3 registres)	[μW]	3.30	3.39	8.5	**15.2**
		21.7%	22.3%	56%	100%
Priority-deviation-based GSRAD	[μW]	3.27	3.43	-	**6.69**
		48.8%	51.2%	-	100%
Priority-proximity-based GSRAD	[μW]	3.17	3.3	-	**6.47**
		49%	51%	-	100%

TABLE 6.1 – Tableau comparatif de la répartition de la consommation entre les modules du routage



(a) GSRAD (b) FIFO-based GSRA (c) FIFO-based GSRAD

FIGURE 6.7 – Répartition de la consommation au sein d'un nœud pour les algorithmes FIFO-based GSRA et GSRAD



(a) GSRAD (b) FIFO-based GSRA (c) FIFO-based GSRAD

FIGURE 6.8 – Répartition de la surface au sein d'un nœud pour les algorithmes GSARD, FIFO-based GSRA et FIFO-based GSRAD

Chapitre 7

Le placement et routage d'un NoC selon la topologie de De Bruijn

Le placement et routage est l'une des dernières étapes dans le flow de conception de circuits digitaux. C'est durant cette dernière étape que les outils d'implémentation prennent en compte la disposition physique de la totalité des cellules digitales en traduisant le code Verilog implémenté. Parmi les paramètres pris en compte, nous trouvons notamment la distance physique qui sépare deux cellules reliées entre elles par une connexion dont la longueur est non-nulle et donc possédant une capacité et un certain timing. Cette étape d'implémentation demande beaucoup de ressources de calcul pour minimiser les longueurs des interconnexions. Elle doit, donc, être prise en considération dès le début de l'implémentation d'un circuit digital par le designer afin d'éviter des difficultés voire des impasses lors de cette étape dans les cas où l'agencement des blocs serait compliqué à réaliser.

Le placement et routage est un challenge majeur lors de la réalisation d'un réseau basé sur la topologie de De Bruijn surtout lorsque le niveau d'un réseau de De Bruijn et donc le nombre de cellules augmente. Contrairement à la topologie mesh, le scaling de la topologie de De Bruijn n'est pas évident. En effet, son irrégularité induite par sa nature non-planaire est un obstacle pour l'obtention d'une représentation 2D claire et lisible. L'agencement d'un réseau de De Bruijn est plus chaotique et des règles de scaling sont difficiles à mettre en place. De plus, comme nous allons le montrer, une représentation claire d'un réseau de De Bruijn n'assure pas une minimisation des longueurs d'interconnexions qui sont un élément clé lors du placement et routage. Dans ce chapitre, nous exposerons les différentes possibilités déjà envisagées dans la littérature[23][33] pour réaliser un agencement des PE dans la topologie de De Bruijn afin d'effectuer une placement et routage le plus efficace possible. Par la suite, nous tenterons de proposer des pistes permettant de faciliter le placement et routage pour des réseaux de De Bruijn d'ordre élevé.

7.1 Le placement et routage de NoCs

Comme mentionné, l'étape de placement et routage consiste à simuler le placement des cellules physiques sur le chip de silicium et de les relier entre elles lors du routage. Cette étape permet notamment de prendre en compte l'agencement des cellules et la longueur des connexions entre les cellules.

La prise en compte de la longueur des cellules induit deux problématiques. La première problématique concerne la capacité d'une cellule à pouvoir *driver* la connexion. En effet, plus les connexions sont longues plus les cellules connectées au début de ces connexions doivent pouvoir injecter un courant important. Les outils d'implémentation vont donc augmenter la taille des cellules afin qu'elles puissent fournir un courant plus important. cette opération s'effectue au détriment de la surface et de la consommation. En effet, la surface et la consommation d'une cellule augmentent avec la taille de celle-ci.

La deuxième problématique lors de la réalisation de cette étape pour un NoC, concerne les contraintes de timing qui deviennent critiques à cause des connexions qui peuvent atteindre des longueurs importantes. En effet, si nous considérons une connexion "k" possédant une longueur "L", la contrainte de timing impose que le temps de propagation le long de "k" soit plus petit que la période de l'horloge du circuit. Le but étant qu'un signal envoyé au flanc montant du signal d'horloge puisse être réceptionné au flanc suivant. Le temps de propagation étant fonction de la capacité, elle-même évoluant dans la longueur de la connexion, plus un fil sera long, plus le temps de propagation sera élevé.

Les outils auront donc tendance à minimiser la longueur des connexions afin de diminuer la taille des cellules et la contrainte sur le timing. Il en résulte une optimisation importante du placement et du routage. Dans le cas des NoC, ce sont les longues interconnexions entre les nœuds qui sont critiques[4]. En effet, les outils vont avoir tendance à placer toutes les cellules d'un même nœud proche les unes des autres à cause de leurs nombreuses interconnexions. Il s'agit, ensuite, de pouvoir connecter les nœuds entre eux sans entrainer des interconnexions trop longues.

C'est à la lumière de ce problème que se justifie le succès des NoC implémentés en mesh. En effet, de par leur ordonnancement, chaque nœud est relié à son voisin et donc les fils d'interconnexions ont une longueur égale au coté de la surface du *processing element*. Si cette longueur n'est pas trop importante, le scaling du réseau ne pose aucun problème.

Dans le cas d'un réseau implémenté selon la topologie de De Bruijn, l'agencement des nœuds n'est pas évident et dépend du nombre de nœuds que possède le réseau. La longueur des interconnexions n'est donc pas une constante et peut même devenir très importante voire trop importante rendant le placement et routage impossible.

Dans la suite de ce chapitre, nous allons donc nous attarder sur les méthodes d'agencement des réseaux de De Bruijn existantes dans la littérature. Par la suite, nous essayerons d'en dégager une méthodologie afin de procéder à un placement et routage facilement adaptable en fonction de la taille du réseau de De Bruijn.

7.2 Représentation d'un graphe de De Bruijn de faible niveau

L'agencement des différents nœuds au sein d'un réseau de De Bruijn est au coeur du challenge lors du placement et routage. Il existe plusieurs représentations des graphes de De Bruijn pour les ordres allant de 3 à 6 dans la littérature scientifique(Fig. 3.3 & Fig. 7.1). Ces représentations bien que claires et lisibles ne sont que des vues artistiques d'un réseau de De Bruijn. Les références d'où elles proviennent[30][36][29], étudient les propriétés mathématiques des graphes dans des domaines autres que les NoC. Ces Figures ne sont donc pas adaptées au placement et routage car les longueurs des interconnexions et la proximité des nœuds n'ont pas été minimisées. Nous pouvons, notamment, constater que la représentation du réseau de niveau 6 comporte des interconnexions importantes(Fig. 7.1). Néanmoins, bien que non adaptées, ces représentations ont le mérite de disposer les nœuds de manière claire et lisible. Comme nous le verrons dans la Section 7.4.2, elles peuvent, donc, être utilisées comme guides afin de faciliter le travail de l'outil de placement et routage.

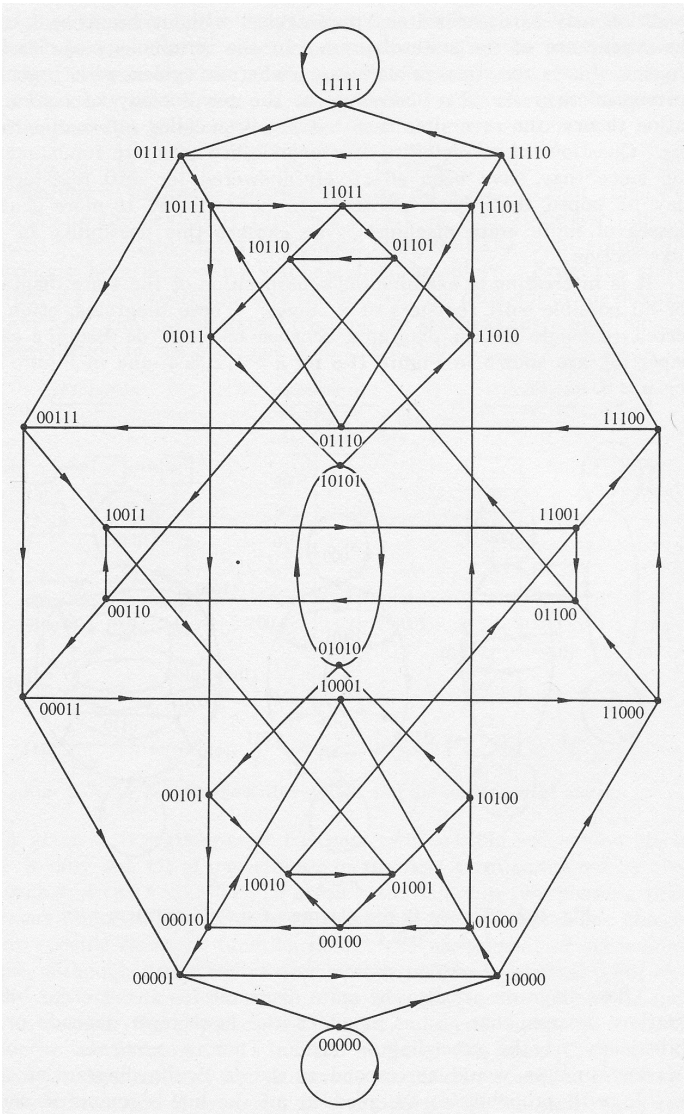


FIGURE 7.1 – Représentation d'un graphe de De Bruijn de niveau 5[29]

7.3 Agencement d'un graphe de De Bruijn selon l'agencement d'un mesh

Dans la littérature, plusieurs travaux[23][33][40] ont essayé de résoudre le problème que représente le placement et routage d'un réseau de De Bruijn. Dans [23][33], l'une des possibilités avancées est de parvenir à ordonner les nœuds d'un De Bruijn de la même manière que pour mesh (Fig. 7.3 & Fig. 7.4).

7.3.1 Agencement en mesh avec ByPass

Dans [23], l'agencement d'un réseau de Bruijn en mesh est réalisé grâce à des Bypass link (Fig. 7.4). Un Bypasslink est un registre qui est placé sur un long fil de connexion entre deux nœuds pour le diviser en deux parties. Cette modification a pour conséquence de relacher la contrainte de timing en octroyant plusieurs cycles pour transférer le paquet d'un nœud à un autre. Le désavantage est que le hop du nœud de départ au nœud d'arrivée, s'effectue en plusieurs cycles au lieu d'un seul ce qui augmente la latence d'un hop de un à deux.

Du point de vue mathématique, la mise en place de ByPass ajoute un poids aux arcs du graphe. Dans le cas où chaque hop s'effectue en un seul cycle comme c'était le cas jusqu'à présent, tous les arcs avaient un poids similaire de un. Le chemin entre deux nœuds correspondait au nombre de hops. Maintenant, si on ajoute des poids différents aux arcs (p_i), la longueur du parcours entre deux nœuds sera égale au nombre de hops multipliés par leur poids respectif ($P = \sum_i^n p_i$). Si on reprend le graphe exemple du Chapitre 3 en y ajoutant des poids(Fig. 7.2), la longueur du parcours entre A_1 et A_3 via A_2 est de $1 + 2 = 3$.

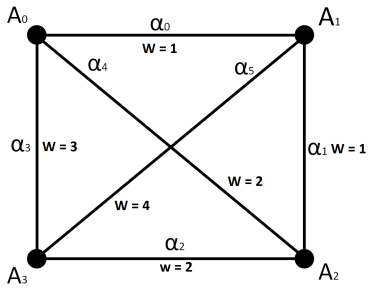


FIGURE 7.2 – Exemple de graphe à poids d'interconnexion variables

Du point de vue pratique, les ByPass ont pour effet d'augmenter la latence mais de diminuer la consommation puisque la longueur des connexions est plus petite (Sec. 7.1). L'ajout de ByPass fait donc face à une optimisation entre la consommation d'un hop d'une part et sa latence de l'autre. De plus, dans la cas où plusieurs ByPass sont placés sur les interconnexions de manière irrégulière, le chemin qui minimise le nombre de hops n'est pas nécessairement celui qui minimise la latence. Ce dernier constat est problématique du point de vue du routage. En effet, le GSRA ne prend en aucun cas en compte le poids des interconnexions entre les nœuds. Pour un réseau de De Bruijn avec un routage GSRA, l'ajout de ByPass doit donc s'effectuer de manière prudente et régulière de façon à ne pas déséquilibrer le graphe.

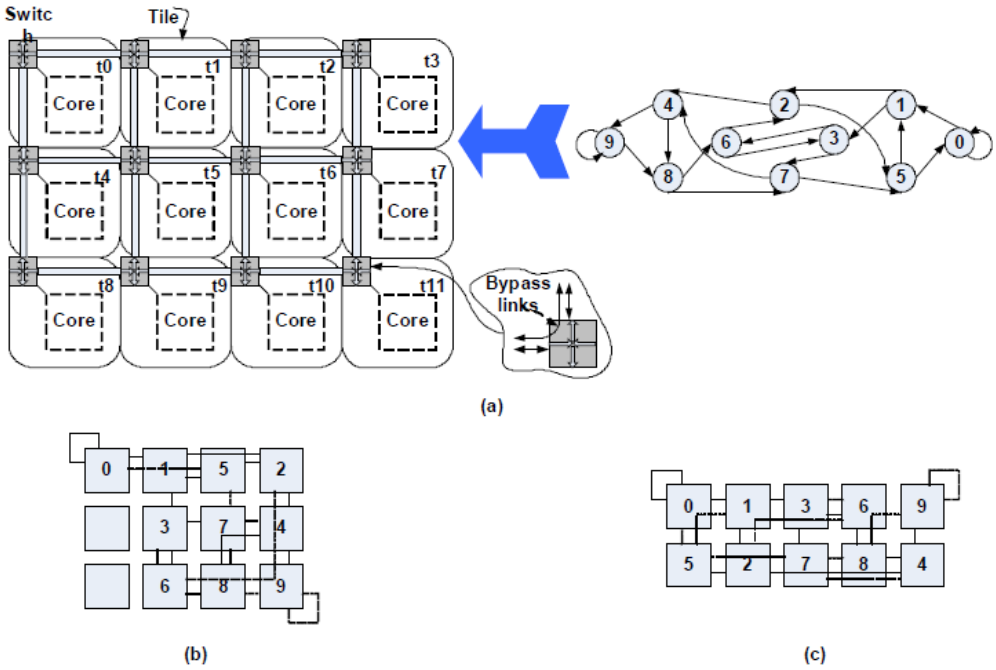


FIGURE 7.3 – Agencement d'un graphe de De Bruijn généralisé selon ordonnancement d'un mesh[23]

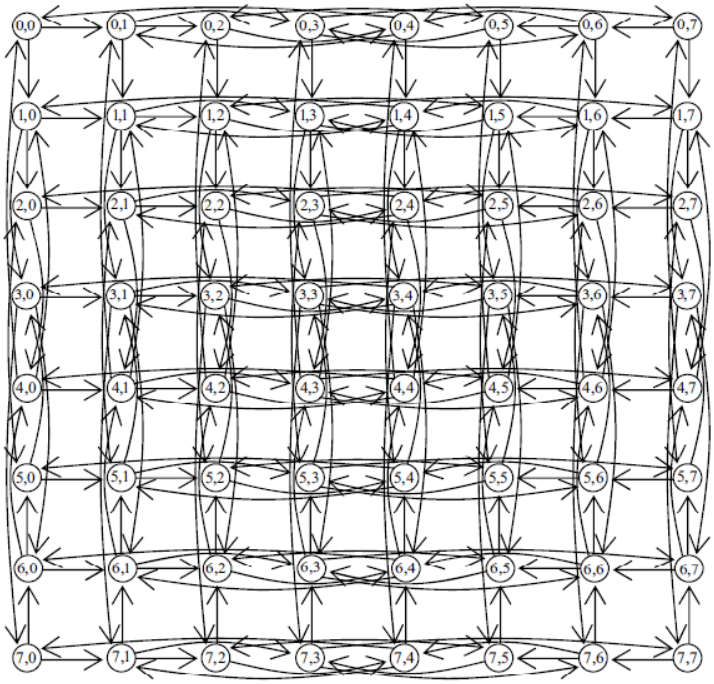


FIGURE 7.4 – Agencement d'un ensemble de graphes de De Bruijn de niveau 3 selon ordonnancement d'un mesh[33]

Dans ses travaux[23], M. Hosseinabady prend comme contrainte de base que chaque nœud possède un ByPass (Fig. 7.4) et que les interconnexions entre les nœuds passent par ces ByPass comme sur la Figure 7.4.b. Le problème se résume donc à la manière de répartir les nœuds sous forme de mesh en minimisant le nombre de fois qu'un fil d'interconnexion passe par un ByPass. Afin de résoudre cette optimisation, il modélise le coût total des interconnexions comme la somme des coûts de chaque fil d'interconnexion(Eq. 7.1). Le coût entre deux nœuds j et l du mesh est appelé C_{jl} et représente le nombre de sauts entre deux ByPass à effectuer pour relier j et l.

De plus, il existe un lien entre la position j et la position l dans le mesh si les nœuds i et k du Bruijn qui y ont été disposés, sont liés dans un graphe de De Bruijn. Par exemple, les deux positions sur la gauche de la première ligne (Fig. 7.4.b) possèdent un lien entre eux car les nœuds d'adresse 0 et 1 du réseau De Bruijn y ont été disposés. Ces deux nœuds sont reliés entre eux dans le graphe de De Bruijn comme nous pouvons le constater sur la Figure 7.4.a. Pour modéliser cette condition, il faut la décomposer en trois sous-conditions. Premièrement $x_{ij} = 1$ si le nœud i du De Bruijn a été disposé à la position j dans le mesh sinon $x_{ij} = 0$. Deuxièmement, $x_{kl} = 1$ si le nœud k du De Bruijn a été disposé à la position l du mesh. Enfin $\alpha_{ik} = 1$, si les nœuds i et k sont connectés dans un graphe de De Bruijn.

Le coût total des interconnexions se traduit par l'Eq. 7.1. Une fois modélisé, le problème peut ensuite être optimisé par un algorithme d'optimisation qui testera l'ensemble des positions afin de déterminer le coût minimal des interconnexions.

$$\sum_{i,j,k,l} \alpha_{ik} x_{ij} x_{kl} C_{jl} \tag{7.1}$$

En conclusion, grâce à cette formulation, M. Hosseinabady a réussi à déterminer quelles sont les positions les plus appropriées dans le mesh. Néanmoins, cette formulation possède deux inconvénients. Le premier déjà exprimé, est qu'elle n'assure pas l'homogénéité des poids au sein du graphe et donc la latence de certains hops peut dépasser la somme des latences de deux hops. Dans ce cas, un détour peut être un choix avisé dont il faut tenir compte dans le routage. Le deuxième inconvénient est le contrôle du nombre d'interconnexions passant par un ByPass. En aucun cas, cette information n'est prise en compte dans l'optimisation et il est possible que la taille de certains ByPass devienne importante, augmentant la consommation du réseau de façon importante.

7.3.2 Agencement en mesh par subdivision

Une autre possibilité d'agencement d'un graphe de Bruijn en mesh est de subdiviser celui-ci en une série de graphes de De Bruijn de niveau 3[33]. En effet, la linéarisation d'un graphe de De Bruijn de niveau 3 (Fig. 7.5) permet de disposer sur une seule ligne l'ensemble des éléments avec une longueur maximum d'interconnexions entre deux nœuds bornée à trois intervalles. Si on considère qu'une longueur de trois intervalles n'est pas en dehors des contraintes lors du placement et routage, il est possible de connecter tous les nœuds d'une ligne ou d'une colonne de 8 nœuds selon la topologie de Bruijn afin d'obtenir un réseau de 64 nœuds (Fig. 7.3).

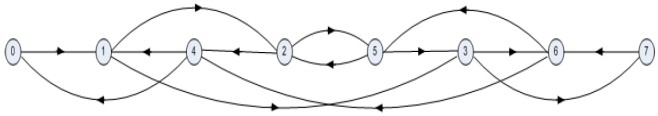


FIGURE 7.5 – Linéarisation d'un graphe de De Bruijn de niveau 3

Cette approche est séduisante mais présente quelques inconvénients. Premièrement, le réseau n'est pas connecté en tant que tel selon la topologie de De Bruijn et donc un routage particulier autre que le GSRA doit être implémenté. De plus, la latence moyenne s'en trouve affectée car elle subit les effets de centralisation des transmissions vers le centre du réseau. Deuxièmement, le passage à un réseau de taille supérieure s'avère compliqué. Le premier graphe de Bruijn de niveau supérieur contient 16 nœuds et sa linéarisation entraine des longueurs d'interconnexions nettement supérieures à trois intervalles. Il est, cependant, possible de généraliser le graphe de De Bruijn selon les règles définies à la Section 3.4.2 afin d'obtenir des réseaux possédant 9 ou 10 nœuds sur une seule colonne mais cette possibilité n'est pas viable au delà d'une certaine taille puisque la longueur d'interconnexion augmente avec le nombre de nœuds.

En conclusion, bien que ces solutions paraissent intéressantes du point de vue du placement, elles possèdent des inconvénients importants en termes de latence moyenne et de routage des interconnexions qui les rendent peu compétitives. Néanmoins, à la Section suivante, nous tenterons de diminuer la difficulté du placement et routage en agençant les noeuds du réseau de De Bruijn selon l'ordonnancement d'un mesh afin de déterminer si cette solution permet de faciliter cette étape.

7.4 Placement et routage d'un NoC de De Bruijn de niveau 4

Dans cette section, nous allons effectuer le placement et routage pour un réseau de De Bruijn de niveau 4 et évaluer les ressources nécessaires à l'outil informatique. Pour cela, nous avons utilisé le programme *Encounter* de Cadence[41] dans sa version de 2013. Dans la Section 7.4.1, nous présenterons les différentes méthodes qui peuvent être mises en place afin de faciliter le travail de l'outil d'implémentation. Puis nous expliciterons la méthodologie que nous avons employée et enfin, nous présenterons les résultats obtenus.

7.4.1 Méthodes de placement

Outre le fait que le placement et routage soit un challenge pour les réseaux de De Bruijn, la méthode utilisée influence aussi les performances des outils informatiques lors de cette étape. En effet, le placement et routage d'un très grand nombre de cellules comme c'est le cas dans un réseau, demande des ressources physiques importantes aux outils notamment au point de vue de la mémoire et du temps utilisés. Cependant il existe plusieurs méthodes qui peuvent influer sur ces paramètres.

Il existe plusieurs méthodes de placement et routage qui peuvent être répertoriées en fonction de la liberté que l'utilisateur désire laisser à l'outil. En effet, il est possible d'imposer à l'outil des contraintes géographiques dans le positionnement des blocs. Dans cette section, nous allons détailler ces méthodes en partant de la moins contraignante à la plus contraignante.

La première méthode est appelée le *full flat*. Elle consiste à donner l'ensemble des cellules à l'outil sans lui imposer aucune contrainte. Cette méthode est habituellement utilisée par des petits designs entièrement digitaux où l'on ne désire pas imposer de contraintes structurelles et pour lesquels l'outil n'a pas besoin d'aide. L'outil est donc libre de placer les cellules n'importe où dans la surface affectée appelée le *floorplan*.

Ensuite, la contrainte la moins restrictive est l'utilisation de *guides*. Il est possible de définir un guide pour un module quelconque du design en définissant une surface dans le floorplan. Définir un guide, impose à l'outil de placer la majorité des cellules du module au sein de la surface définie mais il est libre de placer certaines cellules du module en dehors la surface et ou de placer des cellules d'autres modules dans cette zone. Le guide comme son nom l'indique permet donc de guider l'outil lors du premier placement avant l'optimisation. Les guides sont notamment utilisés lorsque l'utilisateur a déjà une idée de l'agencement idéal de son circuit.

On peut contraindre un peu plus l'outil en utilisant des *régions*. Une région est une surface dans laquelle nous indiquons que l'outil doit placer toutes les cellules d'un module sans exception. Toutefois, lorsque toute les cellules du module sont placées, des cellules d'autres modules peuvent y être placées s'il reste de la place. Les régions sont idéales quand l'utilisateur veut placer certains modules à des endroits spécifiques et venir combler ensuite les trous avec les autres modules.

La méthode la plus contraignante est l'utilisation de *fence*. Un fence est une surface dans laquelle toutes les cellules d'un module sont placées sans exception et seulement les cellules de ce module. L'utilisateur utilise, notamment, les fences lorsqu'il désire partitionner son design pour que, par la suite, le remplacement d'un module par un autre module de surface équivalente soit très aisé.

7.4.2 Paramètres du placement et routage

Nous avons donc choisi d'effectuer le placement et routage d'un réseau de De Bruijn de niveau 4. Ce choix se justifie par le fait que le placement de 16 nœuds contenant chacun un *processing element* est un premier challenge suffisant pour détecter les problèmes qui peuvent survenir lors du placement et routage. L'ensemble des implémentations s'effectuera sur base d'une clock de 10MHz comme pour la synthèse d'un nœud (Chap. 6). Il est à noter que nous avons indiqué à l'Encounter de ne pas effectuer la conception du clock tree. Ce choix se justifie par le fait que nous désirons d'abord savoir si le placement et routage du réseau est possible. L'ajout d'un clock tree augmentant cette étape, nous avons opté pour sa suppression de l'implémentation.

Dans un premier temps, nous effectuerons le placement et routage d'un nœud afin de déterminer que les éventuels problèmes se situent bien au niveau du réseau et non au sein des nœuds.

Ensuite, nous effectuerons deux implémentations d'un réseau de De Bruijn de niveau 4. La première sera effectuée selon la méthode *full flat*. Le but est de déterminer si Encounter est capable d'effectuer le placement et routage sans aide extérieure. La deuxième implémentation sera effectuée en créant des *fences*. L'objectif est de comparer si le placement manuel des nœuds permet à Encounter d'obtenir de meilleures performances par rapport à la méthodologie *full flat*.

Afin d'effectuer l'implémentation, il a été nécessaire d'ouvrir le réseau afin que celui ne soit pas refermé sur lui-même. Pour cela, nous avons supprimé le nœud 1111 et avons implémenté les interconnexions 11111110 et 01111111 comme des I/O du réseau.

De plus, la répartition des *fences* utilisés pour le placement et routage s'est effectuée selon l'agen-

cement d'un réseau mesh. Le placement des différents nœuds a été réalisé afin de minimiser les interconnexions entre les nœuds.

7.4.3 Résultats du placement et routage

Premièrement le placement et routage d'un seul noeud a permis de déterminer que l'Encounter est capable de réaliser cette étape pour un noeud. En effet, à l'issue de l'implémentation, aucun *violation path* n'a été répertorié et aucune erreur de géométrie ou de connectivité n'a été détectée. L'outil a réussi à placer les 6437 cellules reliées par 6564 connections. L'estimation de la consommation du circuit est 129μW.

Il est à noter que, du point de vue des ressources utilisées, l'Encounter a utilisé 500Mb de mémoire et l'implémentation a duré 11min.

Deuxièmement, nous avons effectué l'implémentation du réseau selon la méthodologie *full flat*. Cependant l'outil n'a pas réussi à effectuer cette dernière jusqu'à bout. En effet, l'outil a atteint la capacité limite de la mémoire fixée à 4Gb lors du routage du réseau. A la Figure 7.6, nous pouvons observer l'évolution de la mémoire utilisée pour l'implémentation avec l'augmentation lors du routage. C'est en particulier lors de l'étape de *Detail routing* qu'Encounter a du augmenter ses ressources en mémoire afin de résoudre les 10 millions de violations qu'il avait enregistrées à ce moment-là après 60 heures de simulation.

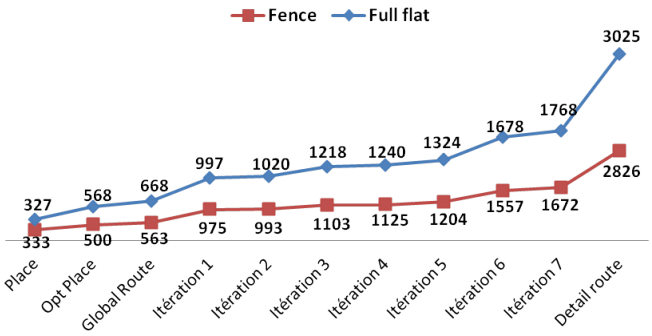


FIGURE 7.6 – Quantité de la mémoire utilisée durant l'implémentation (en Mb)

Enfin, lors de l'implémentation selon la méthodologie utilisant des *fences*, nous constatons un nouvel échec de la part de l'outil. En effet, ce dernier a, encore une fois, atteint la capacité de la mémoire lors de *Detail routing*.

En conclusion, nous ne sommes malheureusement pas parvenu à effectuer le routage d'un réseau de De Bruijn complètement car les ressources requises par l'outil dépassent les capacités de la mémoire disponible à cause du nombre de violations trop élevé lors de la résolution de celles-ci.

Conclusion

Dans le cadre de ce travail, nous avons pu étudier le réseau sur puce connecté selon la topologie de De Bruijn. Cette étude se situe dans la tendance de l'électronique digitale actuelle à évoluer vers des systèmes plus complexes, comprenant un nombre important d'éléments communiquant les uns avec les autres et implémentés sur une seule puce de silicium. Le but de ce travail était de pouvoir situer la topologie de De Bruijn parmi la liste des candidats potentiels pour ce type de système.

Nous avons pu ainsi établir que la topologie de De Bruijn est un candidat intéressant dans le domaine des NoC. De par ses propriétés en terme de latence moyenne de transmission évoluant de façon logarithmique avec le nombre de noeuds, il supplante la topologie mesh qui est la topologie dominante à l'heure actuelle.

Nous avons ensuite proposé quatre algorithmes de routage. Deux d'entre eux utilisaient la déviation d'un des deux paquets en cas de conflits dans le réseau en basant leur choix sur la priorité que possédaient les paquets. Le premier donnait une priorité aux paquets déjà déviés (Priority-deviation-based GSRAD). Le deuxième regardait la proximité des paquets avec leur destination, donnant la priorité à celui qui en était le plus proche (Priority-proximity-based GSRAD).

Les deux autres algorithmes proposés implémentaient des FIFO à la sortie des noeuds afin de résoudre les conflits. Le premier implémentait suffisamment de registres afin de ne jamais saturer les FIFO (FIFO-based GSRA). Le deuxième réduisait le nombre de registres dans les FIFO et se combinait avec le Priority-deviation-based GSRAD pour dévier les paquets avec une priorité aux paquets déviés lorsque les FIFO saturaient pour former le FIFO-based GSRA avec Priority-deviation.

Après l'analyse comportementale, nous avons pu observer que les deux algorithmes implémentant des FIFO possédaient les meilleures performances en terme de latence moyenne de transmission. De plus, le nombre de paquets voyageant durant un temps important dans le réseau était plus limité pour ces deux algorithmes. Nous avons pu aussi déterminer que le nombre de registres nécessaires dans les FIFO pour le FIFO-based GSRA dépendait du taux d'injection de paquets dans le réseau. De plus, ce nombre se basant sur une analyse statistique, aucune garantie quant à d'éventuelles pertes de paquets n'était assurée. Lors de l'analyse comportementale, nous avons également conclu que la réduction à trois registres de la taille des FIFO pour le FIFO-based GSRA avec Priority-deviation était le nombre optimal en terme de latence moyenne lorsque le taux d'injection ne dépassait pas 20%.

Nous avons ensuite procéder à la synthèse des différents algorithmes proposés. Cette analyse nous a permis de conclure que les algorithmes n'implémentant pas de FIFO étaient ceux qui consommaient le moins tandis que les algorithmes avec FIFO consommaient énormément d'énergie à cause de celles-ci. Cependant en observant la consommation du FIFO-based GSRA avec Priority-deviation avec un

nombre de registres réduits, nous avons pu observer que la consommation du routeur atteignait un niveau acceptable. Dès lors, nous avons pu conclure que le FIFO-based GSRA avec Priority-deviation est l'algorithme le plus adapté au routage d'un réseau de De Bruijn en terme de consommation et de performance.

Pour finir, nous avons étudié le placement et routage. La nature non-planaire et la difficulté du scaling de cette topologie font de l'étape de placement et routage, un challenge important. Afin de parvenir à résoudre cette difficulté, nous avons testé deux méthodologie d'implémentation. La première dite de *full flat* laissait entière liberté à l'outil d'implémentation lors du placement et routage. La deuxième se basait sur l'utilisation de *fences* aidant l'outil dans le placement des noeuds. Dans les deux cas, l'outil a été dans l'incapacité de résoudre le placement et routage d'un réseau de De Bruijn composé de seize noeuds atteignant la limite de la taille disponible.

En conclusion à ce travail, nous avons pu observer que la topologie de De Bruijn est un candidat intéressant dans le cadre des NoCs. Il permet d'obtenir des performances en terme de latence moyenne de transmission supérieures à la topologie mesh notamment lorsque le réseau de De Bruijn est routé avec le FIFO-based GSRA avec Priority-deviation. Néanmoins, nous avons pu constater que l'étape de placement et routage est une étape critique pour cette topologie à laquelle nous n'avons pas pu trouver de solution.

Bibliographie

[1] Semiconductor Industry Association (SIA), *"International Technology Roadmap for Semiconductors 2005"*, Proc. Austin, 2005. (Available on the Internet with the URL : http ://public.itrs.net)

[2] G. E. Moore, *"Cramming more components onto integrated circuits"*, Proc. Electronics, vol. 38, pp.114-117, 1965.

[3] D. Brock, *"Understanding Moore's Law : Four Decades of Innovation"*, Proc. Chemical Heritage Press, 2006.

[4] T. Bjerregaard, S. Mahadevan, *"A survey of research and practices of network-on-chip"*, Proc. ACM Comput. Surv., vol. 38, pp. 1-51, 2006.

[5] W. J. Dally, B. Towles, *"Principles and Practices of Interconnection Networks"*, Proc. Morgan Kaufmann Publishers, 2004.

[6] L. Benini, G. DeMicheli, *"Networks on Chips : A New SoC Paradigm"*, Proc. Computer, vol. 35, pp. 70-78, 2002.

[7] ARM, *"AMBA specification"*, 1999. (Available on the Internet with the URL : http ://www.arm.com)

[8] ALTERA, *"Avalon interface specification"*, 2013. (Available on the Internet with the URL : http ://www.altera.com)

[9] A. B. Atitallah, P. Kadionik, F. Ghozzi, P. Nouel, N. Masmoudi, H. Levi, *" An FPGA implementation of HW/SW codesign architecture for H.263 video coding"*, Proc. Int. Journal of Electronics and Communications, vol. 61, pp. 605-620, 2007.

[10] Kuti Lusala A., *"Hybrid Network-on-Chip Architectures for Real-Time Applications on Multiprocessor System-on-Chip Platforms"*, Proc. University of Louvain-la-neuve, 2012.

[11] G. M. Chiu, *"The odd-even turn model for adaptive routing"*, Proc. IEEE Trans. on Parallel and Distributed Systems, vol. 11, pp. 729-738, 2000.

[12] R. Yu, L. Leibo, Y. Shouyi, H. Jie, W. Qinghua, W. Shaojun, *"A fault tolerant NoC architecture using quad-spare mesh topology and dynamic reconfiguration"*, Proc. Journal of Systems Architecture, 2013.

[13] R. Dara, S.A. Hamid, H. Shaahin, E.K. Abbas, *"Power-efficient deterministic and adaptive routing in torus networks-on-chip"*, Proc. Microprocessors and Microsystems, vol. 36, pp. 571-585, 2012.

[14] A. DeHon, *"Compact, multilayer layout for butterfly fat-tree"*, Proc. 12th ACM, pp. 206–215, 2000.

[15] P.P. Pande, C. Grecu, A. Ivanov and R. Saleh, *"Design of a Switch for Network on Chip Applications"*, Proc. Circuits and Systems, vol. 5, pp. 217-220, 2003.

[16] M. Winter, S. Prusseit, and G.-P. Fettweis, *"Hierarchical routing architectures in clustered 2D-mesh Networks-on-Chip"*, Proc. International SoC Design Conference, pp. 388-391, 2010.

[17] M. Kim, D. Kim, and G. E. Sobelman, *"Mpeg-4 performance analysis for CDMA network on chip"*, Proc. International Conference on Communications Circuits and Systems, vol. 1, pp. 493-496 ,2005.

[18] Xiaohang Wang, Mei Yang, Yingtao Jiang, Peng Liu, *"On an efficient NoC multicasting scheme in support of multiple applications running on irregular sub-networks"*, Proc. Microprocessors and Microsystems, vol. 35, pp. 119-129, 2011.

[19] ARM, *"Cortex M0, Technical reference manual"*, 2009. (Available on the Internet with the URL : http ://www.arm.com)

[20] J. Duato, S. Yalamanchili, L. Ni, *"Interconnection Networks : An Engineering Approach"*, Proc. IEEE CS Press, 1997.

[21] C. Grecu, P. Pande, A. Ivanov, R. Saleh, *"BIST for Network-on-Chip Interconnect Infrastructures"*, Proc. 24th IEEE VLSI Test Symposium, pp. 30-35, 2006.

[22] C. Concatto, M. Lubaszewski, *"Improving the yield of NoC-based systems through fault diagnosis and adaptive routing, Journal of Parallel and Distributed Computing"*, vol. 71, pp. 664-674, 2011.

[23] M. Hosseinabady, M.R. Kakoee, J. Mathew, D.K. Pradhan, *"De Bruijn Graph as a Low Latency Scalable Architecture for Energy Efficient Massive NoCs"*, Proc. IEEE Transactions, vol.19, pp.1469-1480, 2011.

[24] M. Horchani, M. Atri, R. Tourki, *"A SystemC QoS router design with virtual channels reservation in a wormhole-switched NoC"*, Proc. Design and Test Workshop, pp.335-340, 2008.

[25] E. Ganesan, D.K. Pradhan, *"Wormhole routing in de Bruijn networks and hyper-de Bruijn networks"*, Proc. Circuits and Systems , vol.3, pp. 870-873, 2003.

[26] L. Benini, G. De Micheli, *"Powering networks on chips : energy-efficient and reliable interconnect design for SoCs"*, Proc. 14th international symposium on Systems synthesis, pp.33-38, 2001.

[27] G. De Bruijn, *"A combinatorial problem"*, Proc. Akademie van Wetenschappen, vol. 49, pp. 758-764, 1946.

[28] B. Arazi, *"Method of constructing de-Bruijn sequences"*, Proc. Electronics Letters , vol.12, pp.658-659, 1976.

[29] W.S. Golomb, *"Shift register sequences"*, Proc. Holden-Day, 1967.

[30] P.E. Compeau, P.A Pevzner, G. Tesler, *"How to apply de Bruijn graphs to genome assembly"*, Proc. Nat. Biotechnol, vol. 29, pp. 987-991, 2011.

[31] L. Zhen, *"Optimal routing in the de Bruijn networks"*, Proc. Distributed Computing Systems, pp. 537-544, 1990.

[32] M. Hosseinabady, M.R. Kakoee, J. Mathew, D.K. Pradhan, *"Reliable network-on-chip based on generalized de Bruijn graph"*, Proc. High Level Design Validation and Test Workshop, pp.3-10, 2007.

[33] R. Sabbaghi-Nadooshan, M. Modarressi, H. Sarbazi-Azad, *"The 2D DBM : An attractive alternative to the simple 2D mesh topology for on-chip networks"*, Proc. Computer Design, pp.486-490, 2008.

[34] S. Wolfram, *"Computation theory of cellular automata"*, Proc. Springer, 1984.

[35] Euler L., *"Solutio Problematis ad geometriam situs pertinentis, Commentarii Academiae Scientarium Imperalis Petropolitanae"*, vol. 8, pp.128-140, 1736.

[36] Z. Aamir, *"3D NOC for many-core processors"*, Proc. Microelectronics Journal, vol. 42, pp. 1380-1390, 2011.

[37] D.D. Wackerly, W. Mendenhall, R.L. Scheaffer, *"Mathematical Statistics with Application"*, Proc. Thomson, 2008.

[38] Synopsys, *"Design Vision, User Guide"*, 2009. (Available on the Internet with the URL : http ://www.sypnosys.com)

[39] J. Muttersbach, T. Villiger, *"Practical design of globally-asynchronous locally-synchronous systems"*, Proc. 6th International Symposium on Circuits and Systems, pp.52-59, 2000.

[40] J. D'Hania, *"Constructing Higher-Order De Bruijn Graphs"*, Proc. Naval School Monterey, 2002.

[41] Cadence, *"Encounter, User Guide"*, 2007. (Available on the Internet with the URL : http ://www.cadence.com)

Appendices

Annexe A

Glossaire

Un **paquet** est l'élément unitaire routé dans le réseau qui contient une donnée ou une partie de celle-ci.

Le **nœud** d'un réseau désigne un élément du réseau dans sa généralité. Il est donc constitué du routeur et d'un processing element. Il est à noter que dans les réseaux indirects, certains nœuds sont dépourvus de processing element.

Une **interconnexion** est un fil de connexion entre deux nœuds d'un réseau. Ces fils sont généralement de longueur importante car ils relient des points qui peuvent très éloignés les uns des autres.

Le **routeur** est un module appartenant à un nœud du réseau qui est chargé de la gestion de la transmission des paquets dans le réseau.

Le ***processing element*** est le terminal qui est attaché à un routeur au sein d'un nœud. Un processing element peut être un processeur avec sa mémoire, ou une mémoire seule, etc. C'est lui qui envoie des paquets au routeur afin qu'ils soient injectés dans le réseau.

Le **hop** est le saut d'un paquet d'un nœud du réseau vers un autre.

La **latence** est le nombre de cycles entre deux événements. Elle peut donc définir la latence moyenne de transmission comme étant le nombre moyenne de cycles d'horloge que prend le paquet pour arriver à sa destination.

Le ***control flow*** est un mécanisme qui commande l'allocation des ressources au sein d'un réseau telles que la bande passante du canal ou la mémoire tampon.

Le **deadlock** est une situation d'impasse dans le réseau lorsque des processus du réseau attendent une ressource telle sorte qu'ils forment une chaine de mise en attente fermée sur elle-même.

La **congestion** est un phénomène de saturation du réseau qui se produit lorsque le nombre de paquet transmis simultanèment est trop important pour le réseau entrainant un blocage de celui-ci.

Annexe B

FTFC publication 2013

Annexe C

Code JAVA

C.1 Génération du main.cpp

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.io.BufferedWriter;

public class JMainC {

        public JDebruijn        jd;

        public JMainC(JDebruijn jd)
        {
                this.jd         = jd;
                genMainC();
        }

        public void genMainC()
        {
                String str = genHeadC();
                str = str + genInstC();
                str = str + genWireC();
                int i;
                for(i=0; i<jd.table.length; i++)
                {
                        str = str + ""+genNodeC(jd.table[i]);
                }

                str = str + genEndC();

                File file = new File("main.cpp");
                try
                {
                        Writer output = new BufferedWriter(new FileWriter(file));
                        output.write(str);
```

```
                output.close();
        }
        catch (IOException e){}
}

public String genHeadC()
{
        String str = "//*********************************\n\n";
        str  = str + "//Stas Francois production 2013\n\n";
        str  = str + "//Debruijn network\n\n";
        str  = str + "//*********************************\n\n";
        str  = str + "#include \"systemc.h\"\n";
        str  = str + "#include \"Node.h\"\n";
        str  = str + "#include \"avance.h\"\n";
        str  = str + "#include \"MyTestbench.h\"\n\n";
        str  = str + "sc_trace_file *tf;\n\n";
        str  = str + "int main(int argc , const char *argv[])\n{\n";
        str  = str + "std::cout << \"Start System Debruijn network!\";\n\n";
        str  = str + "sc_set_time_resolution(100, SC_PS);\n\n";
        str  = str + "sc_clock clk(\"clock\", 1, SC_US);\n";
        str  = str + "sc_signal<bool> reset;\n\n";
        str  = str + "tf = sc_create_vcd_trace_file(\"MyDeBruijnNetwork\");\n";
        str  = str + "tf->set_time_unit(1,SC_NS);\n\n";
        str  = str + "sc_trace(tf, clk, \"clk\");\n\n";
        str  = str + "avance cnt(\"cnt\");\n";
        str  = str + "cnt.clk(clk);\n";
        str  = str + "cnt.reset(reset);\n\n";
        str  = str + "MyTestbench Testbench_1(\"Testbench_1\");\n";
        str  = str + "Testbench_1.reset(reset);\n\n";

        return str;
}

public String genInstC()
{
        String str = "";
        int i;
        for(i=0; i<jd.table.length; i++)
        {
                str = str + "Node Node"+jd.table[i].name+"_inst(\"MyNode"
                +jd.table[i].name+"\");\n";
        }
        for(i=0; i<jd.table.length; i++)
        {
                str = str + "sc_signal<sc_uint<"+jd.level+">> NOMA"
                +jd.table[i].name+";\n";
                str = str + "NOMA"+jd.table[i].name+" = "+i+";\n";
        }
        return str;
}

public String genWireC()
{
        String str = "\n sc_signal<sc_uint<"+(jd.width+jd.level+1)+">>";

        int count = 0;
```

```
                int i;
                for(i=0; i<jd.wire.length; i++)
                {
                        if(i==jd.wire.length-1)
                        {
                                str = str + "N"+ jd.wire[i] +";\n\n";
                        }
                        else
                        {
                                if(count == 3)
                                {
                                count = 0;
                                str = str + "N"+ jd.wire[i] + ";\nsc_signal<sc_uint<"
                                +(jd.width+jd.level+1)+">>␣␣␣␣␣␣";
                                }
                                else
                                {
                                        count ++;
                                        str = str + "N"+ jd.wire[i] +",␣";
                                }
                        }
                }
                count = 0;

                return str;
        }

        public String genNodeC(JNode jn)
        {
                String str = "";
                str  = str + "Node"+jn.name+"_inst.clk(clk);\n";
                str  = str + "Node"+jn.name+"_inst.reset(reset);\n";
                str  = str + "Node"+jn.name+"_inst.in1(N"+jn.in1+");\n";
                str  = str + "Node"+jn.name+"_inst.in2(N"+jn.in2+");\n";
                str  = str + "Node"+jn.name+"_inst.out1(N"+jn.out1+");\n";
                str  = str + "Node"+jn.name+"_inst.out2(N"+jn.out2+");\n";
                str  = str + "Node"+jn.name+"_inst.noma(NOMA"+jn.name+");\n\n";
                return str;
        }

        public String genEndC()
        {
                String str = "␣␣cout␣<<␣endl␣<<␣\"Begin␣simulation␣at␣time\"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<<␣sc_time_stamp()␣<<␣endl␣<<␣endl;␣\n\n";
                str  = str + "␣␣sc_start(10000000,␣SC_US);\n\n";
                str  = str + "␣␣cout␣<<␣endl␣<<␣\"End␣simulation␣at␣time\"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<<␣sc_time_stamp()␣<<␣endl;␣\n\n";
                str  = str + "␣␣system(\"pause\");\n\n";
                str  = str + "␣␣return␣0;\n\n}";
                return str;
        }
}
```

C.2 Génération du TopLevel.v

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.io.BufferedWriter;

public class JMainV {

        public JDebruijn        jd;
        public JMainV(JDebruijn jd)
        {
                this.jd         = jd;
                genMainV();
        }

        public void genMainV()
        {
                String str = genHeadV();
                str = str + genWireV();
                str = str + genInstV();
                str = str + genEndV();

                File file = new File(jd.level+"-level.v");
                try
                {
                        Writer output = new BufferedWriter(new FileWriter(file));
                        output.write(str);
                        output.close();
                }
                catch (IOException e){}
        }

        public String genHeadV()
        {
                String str = "//*********************************\n\n";
                str  = str + "//Stas␣Francois␣production␣2013\n\n";
                str  = str + "//Debruijn␣network\n\n";
                str  = str + "//*********************************\n\n";
                str  = str + "module␣network(input␣";

                int i;
                for(i = 0; i<(pow2(jd.level)-1); i++)
                str = str + "clk"+i + ",reset"+i +",␣";

                str = str + "clk"+(pow2(jd.level)-1) + ",reset"+(pow2(jd.level)-1) +");\n\n";
                return str;
        }

        public String genInstV()
        {
                String str = "";
                int i;
                for(i=0; i<jd.table.length; i++)
                {
```

```
                str = str + "␣␣␣Node␣Node"+jd.table[i].name+"_inst(.clk(clk"+i+"),
.reset(reset"+i+"),␣.in1(N"+jd.table[i].in1+"),␣.in2(N"+jd.table[i].in2+"),
.out1(N"+jd.table[i].out1+"),␣.out2(N"+jd.table[i].out2+"),
.noma("+jd.level+"'b"+jd.table[i].name+"));\n";
                }
                return str;
        }

        public String genWireV()
        {
                String str = "wire␣["+(jd.length-1)+":0]␣";
                int count = 0;
                int i;
                for(i=0; i<jd.wire.length; i++)
                {
                        if(i==jd.wire.length-1)
                        {
                                str = str + "N"+ jd.wire[i] +";\n\n";
                        }
                        else
                        {
                                if(count == 5)
                                {
                                count = 0;
                                str = str + "N"+ jd.wire[i] + ";\n␣wire␣["+(jd.length-1)+":0]␣
                                }
                                else
                                {
                                count ++;
                                str = str + "N"+ jd.wire[i] +",␣";
                                }
                        }
                }
                count = 0;
                return str;
        }

        public String genEndV()
        {
                String str = "\n\nendmodule␣";
                return str;
        }

        public int pow2(int a)
        {
                int result = 1;
                if(a == 0) return result;
                else
                {
                        int i;
                        for(i=0; i<a; i++) result = result*2;

                        return result;
                }
        }
}
```

Annexe D

Code Verilog

D.1 TopLevel.v

```
//*********************************
//  Stas Francois production 2013
//  Top level
//  Debruijn network
//*********************************

module network(input clk0,reset0, clk1,reset1, clk2,reset2, clk3,reset3,
               input clk4,reset4, clk5,reset5, clk6,reset6,
               input clk7,reset7, clk8,reset8, clk9,reset9,
               input clk10,reset10, clk11,reset11, clk12,reset12,
               input clk13,reset13, clk14,reset14,
               input [44:0] N11111110,
               output [44:0] N01111111);


wire [44:0] N00000000, N10000000, N00000001, N10000001, N00010010, N10010010;
wire [44:0] N00010011, N10010011, N00100100, N10100100, N00100101, N10100101;
wire [44:0] N00110110, N10110110, N00110111, N10110111, N01001000, N11001000;
wire [44:0] N01001001, N11001001, N01011010, N11011010, N01011011, N11011011;
wire [44:0] N01101100, N11101100, N01101101, N11101101, N01111110;

Node Node0000_inst(.clk(clk0), .reset(reset0), .in1(N00000000), .in2(N10000000),
                   .out1(N00000000), .out2(N00000001), .noma(4b0000));
Node Node0001_inst(.clk(clk1), .reset(reset1), .in1(N00000001), .in2(N10000001),
                   .out1(N00010010), .out2(N00010011), .noma(4b0001));
Node Node0010_inst(.clk(clk2), .reset(reset2), .in1(N00010010), .in2(N10010010),
                   .out1(N00100100), .out2(N00100101), .noma(4b0010));
Node Node0011_inst(.clk(clk3), .reset(reset3), .in1(N00010011), .in2(N10010011),
                   .out1(N00110110), .out2(N00110111), .noma(4b0011));
Node Node0100_inst(.clk(clk4), .reset(reset4), .in1(N00100100), .in2(N10100100),
                   .out1(N01001000), .out2(N01001001), .noma(4b0100));
Node Node0101_inst(.clk(clk5), .reset(reset5), .in1(N00100101), .in2(N10100101),
                   .out1(N01011010), .out2(N01011011), .noma(4b0101));
Node Node0110_inst(.clk(clk6), .reset(reset6), .in1(N00110110), .in2(N10110110),
```

```
                    .out1(N01101100), .out2(N01101101), .noma(4b0110));
Node Node0111_inst(.clk(clk7), .reset(reset7), .in1(N00110111), .in2(N10110111),
                    .out1(N01111110), .out2(N01111111), .noma(4b0111));
Node Node1000_inst(.clk(clk8), .reset(reset8), .in1(N01001000), .in2(N11001000),
                    .out1(N10000000), .out2(N10000001), .noma(4b1000));
Node Node1001_inst(.clk(clk9), .reset(reset9), .in1(N01001001), .in2(N11001001),
                    .out1(N10010010), .out2(N10010011), .noma(4b1001));
Node Node1010_inst(.clk(clk10), .reset(reset10), .in1(N01011010), .in2(N11011010),
                    .out1(N10100100), .out2(N10100101), .noma(4b1010));
Node Node1011_inst(.clk(clk11), .reset(reset11), .in1(N01011011), .in2(N11011011),
                    .out1(N10110110), .out2(N10110111), .noma(4b1011));
Node Node1100_inst(.clk(clk12), .reset(reset12), .in1(N01101100), .in2(N11101100),
                    .out1(N11001000), .out2(N11001001), .noma(4b1100));
Node Node1101_inst(.clk(clk13), .reset(reset13), .in1(N01101101), .in2(N11101101),
                    .out1(N11011010), .out2(N11011011), .noma(4b1101));
Node Node1110_inst(.clk(clk14), .reset(reset14), .in1(N01111110), .in2(N11111110),
                    .out1(N11101100), .out2(N11101101), .noma(4b1110));

endmodule
```

D.2 GSRA.v (4-level)

```
module GSRA (data_sc, noma, data_adr, state);

parameter WIDTH_NOMA = 4;

input data_sc;
input [WIDTH_NOMA-1:0] noma, data_adr;
output reg [1:0] state;


always @(*)  // VOIR RAPPORT: SECTION IMPLEMENTATION GSRA NOEUD PAR NOEUD
begin
        if (data_sc)
                begin
                if(data_adr[WIDTH_NOMA-1:WIDTH_NOMA-4] == noma[WIDTH_NOMA-1:0])
                        begin
                                                        state[1:0] = 0;
                        end
                else if(data_adr[WIDTH_NOMA-1:WIDTH_NOMA-3] == noma[WIDTH_NOMA-2:0])
                        begin
                        if(!data_adr[WIDTH_NOMA-4])     state[1:0] = 1;
                        else                            state[1:0] = 2;
                        end
                else if(data_adr[WIDTH_NOMA-1:WIDTH_NOMA-2] == noma[WIDTH_NOMA-3:0])
                        begin
                        if(!data_adr[WIDTH_NOMA-3])     state[1:0] = 1;
                        else                            state[1:0] = 2;
                        end
                else if(data_adr[WIDTH_NOMA-1] == noma[0])
                        begin
                        if(!data_adr[WIDTH_NOMA-2])     state[1:0] = 1;
                        else                            state[1:0] = 2;
                        end
                else
                        begin
                        if(!data_adr[WIDTH_NOMA-1])     state[1:0] = 1;
                        else                            state[1:0] = 2;
                        end
                end
        else
                begin
                                                        state[1:0] = 0;
                end
end

endmodule
```

D.3 FIFO.v (4 registres)

```
module FIFO( clk, reset,next, data1_fifo,data_read_fifo, full);

parameter WIDTH_NOMA = 4;
parameter WIDTH_DATA = 32;
parameter WIDTH_FIFO = 3;
parameter WIDTH_DEV = 3;

input clk, reset;
input next;
input [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] data1_fifo;
output reg [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] data_read_fifo;
output full;

reg [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] REGISTER_BANK0, REGISTER_BANK1, REGISTER_BANK2;
reg [3:0] adr0, adr0_next;

always @(*)
begin
        if(data1_fifo[0] & next) adr0_next = adr0;
        else if(data1_fifo[0] & !next) adr0_next = adr0+1b1;
        else if(next) adr0_next = adr0-1b1;
        else  adr0_next = adr0;

end

assign full = (adr0_next == 4d3) ? 1b1 : 1b0;

always@(posedge clk)
begin
        if(reset) adr0 <= 0;
        else adr0 <=  adr0_next;
end

always@(posedge clk)
begin
        if(reset) data_read_fifo <= 0;
        else data_read_fifo <=  REGISTER_BANK0;
end

always @(posedge clk)
begin
        if(reset)
                begin
                REGISTER_BANK0 <= 0;
                REGISTER_BANK1 <= 0;
                REGISTER_BANK2 <= 0;
                end
        else if(next & data1_fifo[0])
                begin
                if(adr0 == 3b000)
                        begin
                        REGISTER_BANK0 <= data1_fifo;
                        REGISTER_BANK1 <= REGISTER_BANK1;
                        REGISTER_BANK2 <= REGISTER_BANK2;
```

```
                        end
                else if(adr0 == 3b001)
                        begin
                        REGISTER_BANK0 <= REGISTER_BANK1;
                        REGISTER_BANK1 <= data1_fifo;
                        REGISTER_BANK2 <= REGISTER_BANK2;
                        end
                else if(adr0 == 3b010)
                        begin
                        REGISTER_BANK0 <= REGISTER_BANK1;
                        REGISTER_BANK1 <= REGISTER_BANK2;
                        REGISTER_BANK2 <= data1_fifo;
                        end
                end
        else if(!next & data1_fifo[0])
                begin
                if(adr0 == 3b000)
                        begin
                        REGISTER_BANK0 <= REGISTER_BANK0;
                        REGISTER_BANK1 <= data1_fifo;
                        REGISTER_BANK2 <= REGISTER_BANK2;
                        end
                else if(adr0 == 3b001)
                        begin
                        REGISTER_BANK0 <= REGISTER_BANK0;
                        REGISTER_BANK1 <= REGISTER_BANK1;
                        REGISTER_BANK2 <= data1_fifo;
                        end
                end
        else
                        begin
                        REGISTER_BANK0 <= REGISTER_BANK0;
                        REGISTER_BANK1 <= REGISTER_BANK1;
                        REGISTER_BANK2 <= REGISTER_BANK2;
                        end
end

endmodule
```

D.4 routage.v

```
module routage( clk, reset, noma, memwrite, data_write, adr_write, in1, in2, out1, out2);

parameter WIDTH_NOMA = 4;
parameter WIDTH_DATA = 32;
parameter WIDTH_DEV = 3;
parameter WIDTH_FIFO = 3;

input   clk, reset;
input   memwrite;
input   [WIDTH_NOMA-1:0] noma;
input   [WIDTH_DATA-1:0] data_write, adr_write;
input   [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] in1, in2;
output  [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0]out1, out2;


wire next1, next2, next1_par, next2_par, full1, full2;
wire [1:0] state_conf1_par, state_conf2_par, state_conf1, state_conf2,
           state_conf1_tot, state_conf2_tot;
wire stall, stall_tot, stall_par;
wire [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] data_fifo1, data_fifo2;
wire [1:0] compt;

wire sc_write_tmp;
wire [WIDTH_NOMA-1:0] adr_write_tmp;
wire [1:0] state1, state2, state3;
GSRA #( .WIDTH_NOMA(WIDTH_NOMA)) GSRA_inst2(.data_sc(sc_write_tmp), .data_adr(adr_write_tmp),
        .noma(noma), .state(state3));  // internal input

wire [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] data_write_tmp;
Internal_interface #(.WIDTH_NOMA(WIDTH_NOMA), .WIDTH_DATA(WIDTH_DATA), .WIDTH_DEV(WIDTH_DEV))
                    ii_inst(.clk(clk), .reset(reset), .stall(stall), .memwrite(memwrite),
                            .data_write(data_write), .adr_write(adr_write[WIDTH_NOMA-1:0]),
                            .sc_write_tmp(sc_write_tmp), .data_write_tmp(data_write_tmp),
                            .adr_write_tmp(adr_write_tmp));    // Internal interface


FIFO fifo_inst0( .clk(clk), .reset(reset), .next(next1), .data1_fifo(in1),
                 .data_read_fifo(data_fifo1), .full(full1));
FIFO fifo_inst1( .clk(clk), .reset(reset), .next(next2), .data1_fifo(in2),
                 .data_read_fifo(data_fifo2), .full(full2));

assign state_conf1 = full1|full2 ? state_conf1_tot : state_conf1_par;
assign state_conf2 = full1|full2 ? state_conf2_tot : state_conf2_par;
assign stall = full1|full2 ? stall_tot : stall_par;
assign next1 = full1|full2 ? 1b1 : next1_par;
assign next2 = full1|full2 ? 1b1 : next2_par;



GSRA #( .WIDTH_NOMA(WIDTH_NOMA)) GSRA_inst0(.data_sc(data_fifo1[0]),
        .data_adr(data_fifo1[WIDTH_NOMA:1]), .noma(noma), .state(state1));  // input 1
GSRA #( .WIDTH_NOMA(WIDTH_NOMA)) GSRA_inst1(.data_sc(data_fifo2[0]),
        .data_adr(data_fifo2[WIDTH_NOMA:1]), .noma(noma), .state(state2));  // input 2
```

```
GESTION_CONF_TOT GEST_inst0(.state1(state1), .state2(state2), .state3(state3),
                            .state_conf1(state_conf1_tot),
                            .state_conf2(state_conf2_tot), .stall(stall_tot), .compt(compt));

GESTION_CONF_PART GEST_inst1(.state1(state1), .state2(state2), .state3(state3),
                             .state_conf1(state_conf1_par),
                             .state_conf2(state_conf2_par), .stall(stall_par),
                             .next1(next1_par), .next2(next2_par));

wire [WIDTH_DATA+WIDTH_NOMA+WIDTH_DEV:0] in_next1, in_next2;
COMPTEUR #( .WIDTH_NOMA(WIDTH_NOMA), .WIDTH_DATA(WIDTH_DATA), .WIDTH_DEV(WIDTH_DEV))
            COMPT_inst0(.in(data_fifo1), .compt(compt[0]), .out(in_next1));
COMPTEUR #( .WIDTH_NOMA(WIDTH_NOMA), .WIDTH_DATA(WIDTH_DATA), .WIDTH_DEV(WIDTH_DEV))
            COMPT_inst1(.in(data_fifo2), .compt(compt[1]), .out(in_next2));

MUX4 MUX4_inst01(.in1(in_next1), .in2(in_next2), .in3(data_write_tmp),
                 .cdt(state_conf1), .out(out1));
MUX4 MUX4_inst11(.in1(in_next1), .in2(in_next2), .in3(data_write_tmp),
                 .cdt(state_conf2), .out(out2));

endmodule
```