TISSU NUMÉRIQUE CELLULAIRE À ROUTAGE ET CONFIGURATION DYNAMIQUES

THÈSE N^{O} 3226 (2005)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut des systèmes informatiques et multimédias

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Yann THOMA

ingénieur informaticien diplômé EPF
de nationalité suisse et originaire d'Amden (SG)

acceptée sur proposition du jury:

Prof. E. Sanchez, directeur de thèse
Prof. G. De Micheli, rapporteur
Prof. D. Lavenier, rapporteur
Dr G. Sassatelli, rapporteur

Lausanne, EPFL
2005

Version abrégée

A l'instar des premiers concepteurs d'avions, l'homme a toujours observé la nature, qui lui a fourni des pistes lors de la création de nouvelles machines ou de nouveaux concepts. Les circuits électroniques ne font pas exception, et les trois axes de la vie que sont l'évolution des espèces (Phylogenèse), le développement de l'organisme à partir d'une seule cellule (Ontogenèse), ainsi que l'apprentissage dont notre cerveau est capable (Epigenèse), ont vu nombre de réalisations s'en inspirer.

Ces trois axes, qui forment l'acronyme POE, ont été, pour la plupart des réalisations, exploités séparément : les principes de l'évolution permettent de résoudre des problèmes pour lesquels il est difficile d'obtenir une solution de façon déterministe, des circuits électroniques tirent profit des concepts d'autoréparation du vivant, et les réseaux de neurones artificiels sont capables d'effectuer efficacement un grand nombre de tâches. Leur réunion en un seul *tissu* électronique n'a en revanche pas encore vu le jour.

Concernant la réalisation matérielle de tels systèmes, l'avènement des circuits reconfigurables FPGAs, dont il est possible de redéfinir un nombre quasi infini de fois le comportement, en a facilité le prototypage. Ils permettent en effet d'accélérer l'exécution d'une tâche, par le parallélisme matériel qu'ils offrent, et ont été grandement exploités par les chercheurs. Toutefois, ils manquent de plasticité, leur comportement ne pouvant facilement se modifier lui-même sans une intervention extérieure.

Cette thèse, qui s'insère dans le cadre du projet européen POEtic, se propose de définir un nouveau circuit électronique reconfigurable, dans l'optique de fournir un nouveau substrat aux applications bio-inspirées mettant en jeu ces trois axes. Ce circuit est composé d'un microprocesseur, et d'un tableau d'éléments reconfigurables, ce dernier ayant été réalisé par nos soins. Les processus évolutifs sont exécutés par le premier, alors que l'épigenèse et l'ontogenèse prennent place dans le deuxième, sous la forme d'organismes artificiels multicellulaires. Relativement semblable aux FPGAs commerciaux actuels, ce sous-système offre cependant de nouvelles caractéristiques intéressantes. Premièrement, les éléments de base du tableau ont la capacité de reconfigurer partiellement d'autres éléments. Des mécanismes d'auto-réplication et de différenciation peuvent alors l'exploiter pour laisser un organisme artificiel croître ou modifier son comportement. Deuxièmement, un niveau de routage distribué offre la possibilité de créer des connexions entre différentes parties du circuit pendant son fonctionnement. Les cellules qui y sont implémentées, notamment les neurones artificiels, peuvent alors initier de nouvelles connexions, de manière à modifier le comportement global du système.

Ce routage distribué, qui constitue l'un des points importants de cette thèse, a vu la réalisation de plusieurs algorithmes. Basés sur une implémentation parallèle de l'algorithme de Lee, ils sont totalement distribués, c'est-à-dire qu'aucun contrôle global n'est nécessaire à la création de chemins de données. Quatre algorithmes sont ainsi définis et décrits matériellement sous la forme d'unités de routage reliées à leurs 3, 4, 6, ou 8 voisines, toutes identiques, qui gèrent les processus de routage. Une analyse de leurs propriétés nous permet de définir le meilleur algorithme à coupler au plus efficace des voisinages, en terme de congestion, par rapport au nombre de transistors nécessaires à leur réalisation. Nous terminons sur le routage en proposant un cinquième algorithme, qui, contrairement aux quatre précédents, n'est construit que sur des communications locales entre unités de routage. Il offre ainsi, au prix d'un coût matériel supplémentaire, une meilleure scalabilité au système complet.

Finalement, le circuit POEtic, sur lequel un de nos algorithmes de routage a été implémenté, a été physiquement réalisé. Nous présentons différents mécanismes POE qui tirent profit de ses caractéristiques nouvelles et qui peuvent y être embarqués. Parmi ces mécanismes, nous pouvons citer notamment l'auto-réplication, le matériel évolutif, les systèmes développementaux, et l'autoréparation, qui ont été développés grâce à un simulateur du circuit, également conçu durant cette thèse.

Abstract

In the design of new machines or in the development of new concepts, mankind has often observed nature, looking for useful ideas and sources of inspiration. The design of electronic circuits is no exception, and a considerable number of realizations have drawn inspiration from three aspects of natural systems : the evolution of species (Phylogenesis), the development of an organism starting from a single cell (Ontogenesis), and learning, as performed by our brain (Epigenesis).

These three axes, grouped under the acronym POE, have for the most part been exploited separately : evolutionary principles allow to solve problems for which it is hard to find a solution with a deterministic method, while some electronic circuits draw inspiration from healing process in living beings to achieve self-repair, and artificial neural networks have the capability to efficiently execute a wide range of tasks. At this time, no electronic *tissue* capable of bringing them together seems to exist.

The introduction of reconfigurable circuits called Field Programmable Gate Arrays (FPGAs), whose behavior can be redefined as often as desired, made prototyping such systems easier. FPGAs, by allowing a relatively simple implementation in hardware, can considerably increase the systems' performance and are thus extensively used by researchers. However, they lack plasticity, not being able to easily modify themselves without an external intervention.

This PhD thesis, developed in the framework of the European POEtic project, proposes to define a new reconfigurable electronic circuit, with the goal of supplying a new substrate for bio-inspired applications that bring all three axes into play. This circuit is mainly composed of a microprocessor and an array of reconfigurable elements, the latter having been designed during this thesis. Evolutionary processes are executed by the microprocessor, while epigenetic and ontogenetic mechanisms are applied in the reconfigurable array, to entities seen as multicellular artificial organisms. Relatively similar to current commercial FPGAs, this subsystem offers however some unique features. First, the basic elements of the array have the capability to partially reconfigure other elements. Auto-replication and differentiation mechanisms can exploit this capability to let an organism grow or to modify its behavior. Second, a distributed routing layer allows to dynamically create connections between parts of the circuit at runtime. With this feature, cells (artificial neurons, for example) implemented in the reconfigurable array can initiate new connections in order to modify the global system behavior.

This distributed routing mechanism, one of the major contributions of this thesis, induced the realization of several algorithms. Based on a parallel implementation of Lee's algorithm, these algorithms are totally distributed, no global control being necessary to create new data paths. Four algorithms have been defined implemented in hardware in the form of routing units connected to 3, 4, 6, or 8 neighbors. These units are all identical and are responsible for the routing processes. An analysis of their properties allows us to define the best algorithm, coupled with the most efficient neighborhood, in terms of congestion and of the number of transistors needed for a hardware realization. We finish the routing chapter by proposing a fifth algorithm that, unlike the previous ones, is constructed only through local interactions between routing units. It offers a better scalability, at the price of increased hardware overhead.

Finally, the POEtic chip, in which one of our algorithms has been implemented, has been physically realized. We present different POE mechanisms that take advantage of its new features. Among these mechanisms, we can notably cite auto-replication, evolvable hardware, developmental systems, and self-repair. All of these mechanisms have been developed with the help of a circuit simulator, also designed in the framework of this thesis.

Remerciements

> La vie, c’est comme une bicyclette, il faut avancer pour ne pas perdre l’équilibre.
>
> Albert EINSTEIN

Ah, les remerciements. Derniers mots posés, qui ne sont de loin pas les plus évidents, la mémoire étant un système très efficace, mais pas entièrement fiable. Je vais toutefois essayer d’éviter d’omettre les acteurs les plus importants, en m’excusant déjà auprès des pauvres oubliés qui peuvent toutefois être sûrs que leur présence a été également appréciée.

Cette thèse n’aurait pas vu le jour sans l’existence du Laboratoire de Systèmes Logiques (LSL), dirigé de main de maître par Daniel Mange. Sans lui mon destin aurait possiblement été fort différent, puisque les recherches du laboratoire dans le domaine de la bio-inspiration ont pesés dans la balance me faisant choisir l’EPFL plutôt que l’Université de Genève comme lieu d’études. Ayant abandonné l’idée de devenir éthologue, mêler les principes du vivant aux outils informatiques m’a semblé la voie à suivre. Merci donc à Daniel, pour m’avoir si chaleureusement accueilli dans son laboratoire, et pour ses qualités oratoires qui resteront pour moi un modèle à suivre.

Que serait une thèse sans directeur de thèse ? Bien peu de chose, assurément. Eduardo DJ Sanchez, qui m’a donné le goût des systèmes logiques lors de ma première année d’études EPFLiennes, s’est trouvé tout naturellement à cette place cruciale en me proposant de travailler sur le projet POEtic. La liberté qu’il m’a (trop ?) laissée m’a permis d’aiguiser mes armes scientifiques et critiques, et de mener à bien ce travail. Bien plus qu’un directeur, Eduardo restera un ami dont la culture musicale et la collection de CDs me feront toujours pâlir de jalousie, et dont j’espère pouvoir encore goûter les côtelettes grillées dont il a le secret.

Outre ses deux figures charismatiques, orateurs hors pairs, dont la rigueur du premier et l’accent du deuxième sont l’essence même du LSL, une foule de personnalités constituent le liant de ce laboratoire à l’ambiance exceptionnelle. Commençons par mes collègues de bureau. Le premier à avoir quitté la salle magique INN235 est Mathieu key-destroyer, la terreur des claviers, dont j’attends avec impatience le retour en terre vaudoise. Ce fut ensuite Professeur Gianluca qui le suivit, emportant avec lui sa musique, sujet sur lequel nos accords avaient, il faut l’avouer, une fâcheuse tendance à sonner faux. Enrico, troisième sur la liste et guitariste émérite nous a, quant à lui, laissé en héritage son tube planétaire F... Sunshine, que j’ai encore plaisir à écouter. Enfin, remplaçant les départs, Fabien l’homme de l’air prit la place de Mathieu. Toujours prêt à décoller, autant en journée qu’en soirée, j’ai eu fort plaisir à nos collaborations Biowalliennes. Enfin, dernier arrivé, Ludovic, le plus dark des metal allemands a su imposer un ton musical hors norme qui ne fut pas pour me déplaire.

Séparés par quelques murs, nous retrouvons Alessandro, l’homme dont le clavier tape plus vite que son ombre et qui est capable de résoudre le plus obscur des soucis informatiques. Dans le même bureau, Jonas est l’homme qui me fit découvrir la percolation. Je le remercie infiniment pour ses suggestions mathématiques toujours judicieuses. Andres, le dernier des Colombiens (hormis notre bien-aimé Eduardo), bien qu'ayant mis du temps à nous rejoindre aux pauses de midi, a bien fait de s'y mettre, sa bonne humeur éternelle étant plus qu'appréciable. Dernier habitant de ce bureau, Yvan y est arrivé pour renforcer encore notre présence à la cafétéria du DMX, haut lieu culinaire du site de l'EPFL, que Marie-Thérèse gère de la plus belle des manières. De l'autre côté du couloir, nous trouvons Pierre-André et Joël. Tous deux ont su amener un souffle nouveau, et une excellente ambiance au renouveau des fameuses soirées bières.

N'oublions pas les anciens doctorants. Christof, tout d'abord, modèle de travailleur acharné, qui rendrait modeste le plus bosseur d'entre nous. Grâce à lui je suis toujours au faîte de l'actualité de la mode vestimentaire de l'Association des Super Suisses. Ralph ensuite, toujours discret et rigoureux, dont l'efficacité au laser-game est impressionnante. Les valaisans Jaco et Jean-Luc, dont le premier est la seule personne que je connaisse capable de fumer un bouchon, et dont le deuxième, sous des apparence débonnaires écoute une musique des plus sauvages, ont amené au labo des breuvages douteux et goûteux qui ont agrémentés nos soirées folles. Et du côté colombien, Carlos Andrés et Fabio, dont j'apprécie particulièrement le calme et la sérénité, sont deux papas qui ont des enfants plus que chanceux.

Le laboratoire ne serait rien sans Marlyse, notre secrétaire, qui illumine ce laboratoire de son sourire radieux et de sa bonne humeur, et qui fut agréablement secondée par Natascha. Leur gentillesse, de même que leur promptitude à résoudre le moindre souci administratif, sont définitivement un des atouts du LSL. Et n'oublions pas André Chico Jeux-de-maux Badertscher, toujours à l'affût de la donzelle passant dans nos couloirs, et à qui je dois un petit coup de pouce du destin. Au bout de ce même couloir, nous trouvons Auke, professeur qui a insufflé encore plus de bio-inspiration aux recherches du LSL, tout en y amenant un niveau d'ondes positives très apprécié. Terminons ce survol par deux figures légendaires, JK et André Stauffer. JK, expert logicien, m'a fait passer l'oral le plus rapide de mon existence durant mes études à l'EPFL (avec fin heureuse, je vous rassure), et s'est mué collègue à l'humour toujours à la pointe du bon ton. André, quant à lui, est l'homme graphique du LSL, et son écriture manuscrite continue encore à m'impressionner.

Merci à tous pour ces quatre années radieuses. En sortant des études d'ingénieur je pensais savoir beaucoup de choses, et la confrontation à tant de matière grise et de générosité m'a définitivement appris l'humilité.

Je remercie vivement mes relecteurs occasionnels, Fabien, Joël et Eduardo, qui ont insufflé un peu plus de clarté dans ce document, et Gianluca pour les corrections anglaises. Une mention spéciale va au professeur Langaney, dont j'ai toujours autant de plaisir à écouter les exposés sur l'évolution des espèces, et qui a bien voulu partir à l'assaut des erreurs du troisième chapitre. Merci également à Marc pour nos quelques échanges fructueux, et à Marcel pour les discussions statistiques que nous avons eues.

Et n'oublions pas les rapporteurs qui ont lu et approuvé ce travail : le professeur Dominique Lavenier, le professeur Giovanni de Micheli, et le docteur Gilles Sassatelli, de même que le directeur du jury, le professeur Paolo Ienne.

Mes remerciements vont également aux membres du projet POEtic, avec qui j'ai eu beaucoup de plaisir à collaborer et à festoyer à Barcelone, York, Glasgow, et Lausanne. Je pense y avoir entre autre appris qu'il n'est pas des plus évident de coordonner une équipe répartie aux quatre coins de l'Europe. Heureusement que Daniel Roggen travaillait sur le site de l’EPFL, nous avons ainsi pu partager nos doutes et espoirs durant les trois ans que dura le projet.

Quittons à présent le monde du travail, pour nous approcher des personnes à qui je dois mon équilibre. Mes remerciements les plus chaleureux vont aux membres de ma famille, qui m’ont permis d’effectuer mes études à l’EPFL, et ont toujours su être présents, malgré ma fâcheuse tendance à passer un temps plus que certain hors du logis familial.

Un esprit sain dans un corps sain ? Alors que je tentais d’affûter le premier au laboratoire, le basket-ball m’a apporté un équilibre nécessaire au bon fonctionnement de mon organisme. Mes partenaires du Vermont Basket II y sont pour beaucoup, de même que les joueurs occasionnels avec qui j’ai toujours autant de plaisir à tapoter le ballon orange.

Le sport, ainsi qu’une certaine forme de raisonnement sont, à mon avis, indispensables à l’équilibre du corps. La situation se complique un peu lorsque la musique occupe une place également importante. J’ai tenté tant bien que mal de partager ma vie entre le laboratoire, le basketball, les répétitions, et les relations humaines (sans que cet ordre ne signifie une quelconque préférence, rassurez-vous). Les musiciens des Braibs et d’Exunda ont dès lors tenu une place importante dans mon équilibre, et je les remercie infiniment pour les moments musicaux que nous avons partagés.

Je tiens également à remercier ici Inès pour son soutien lors des doutes de début de thèse, alors que la direction de mes recherches tentait de se définir. Et évidemment, mes amis de toujours, Marcel, Joël et Christophe, avec qui j’ai toujours autant de plaisir à partir à la conquête du Sud de la France. Le mélange économètre-éducateur-sociologue-informaticien nous offre de forts beaux débats, auxquels j’espère pouvoir participer encore longtemps. Merci donc à vous d’être vous. Ne changez rien.

Finalement, un merci particulier à Stéphanie, qui a eu la lourde tâche de me soutenir durant les deux dernières années de mon travail. Sa patience et ses encouragements m’ont largement aidé à mener à bien cette thèse.

Table des matières

Chapitre 1

Prolégomènes

Entre
Ce que je pense,
Ce que je veux dire,
Ce que je crois dire,
Ce que je dis,
Ce que vous avez envie d'entendre,
Ce que vous croyez entendre,
Ce que vous entendez,
Ce que vous avez envie de comprendre,
Ce que vous comprenez,
Il y a dix possibilités qu'on ait des difficultés à communiquer.
Mais essayons quand même...

Bernard WERBER , *Le père de nos pères*

1.1 Réflexions pré-introductives

De tout temps l'homme s'est inspiré de la nature pour construire des machines, à la manière dont les oiseaux motivèrent les pionniers de la conquête des airs. Sur le plan des technologies de l'information, bien que les travaux de von Neumann sur les architectures des premiers processeurs visaient à créer un système capable d'imiter le comportement du cerveau, cette bio-inspiration n'y est que peu présente. Les microprocesseurs, qui sont aussi bien exploités par les ordinateurs et par les téléphones mobiles que par les machines à laver, n'embarquent pas de mécanismes semblables à ceux observés dans la nature.

Cependant, les systèmes aussi bien matériels que logiciels faisant intervenir des processus inspirés du vivant sont de plus en plus nombreux. A titre d'exemples, citons les réseaux de neurones artificiels qui analysent les adresses écrites à la main sur les lettres envoyées par la Poste, et la main artificielle de l'équipe d'Higuchi [116] qui s'adapte à son hôte : ces deux exemples ne sont qu'une infime partie de la recherche effectuée dans le domaine des systèmes bio-inspirés.

Mais pourquoi vouloir à tout prix s'inspirer du vivant ? Tout simplement parce que la vie est impressionnante de plasticité et d'adaptabilité. Depuis son apparition, il y a quelques milliards d'années, elle n'a cessé de se complexifier, jusqu'aux organismes que nous connaissons actuellement, en inventant sans cesse de nouvelles stratégies et de nouveaux systèmes. De plus, les capacités d'autoréparation du vivant sont impressionnantes. Alors qu'un ordinateur "plante" lamentablement si un seul des innombrables signaux qui le composent change subrepticement d'état, les organismes vivants peuvent survivre à des blessures ou aux maladies. Enfin, les capacités d'apprentissage des êtres dits supérieurs sont surprenantes, tandis que les machines peinent encore à tirer parti des expériences vécues, et à les généraliser.

1.2 Motivations POEtiques

Dans cette thèse, nous nous intéresserons particulièrement au modèle POE, qui vise à décrire les systèmes matériels bio-inspirés selon trois axes : Phylogénétique, relatif à l'évolution, Ontogénétique, pour ce qui est de la croissance et de l'autoréparation, et Épigénétique, en ce qui concerne l'apprentissage. Différentes applications peuvent mettre en oeuvre un, deux ou trois de ces axes, en fonction du problème à résoudre.

La phylogenèse traite, sur le plan du vivant, de l'évolution des espèces, et de la manière dont les gènes sont transmis des parents à la progéniture. Diverses théories coexistent, allant du néo-darwinisme au Lamarckisme, en passant par celle des équilibres ponctués et celle des monstres prometteurs. Elles ont inspiré des chercheurs qui en ont tiré le concept d'algorithme génétique, visant à résoudre un problème complexe pour lequel la solution ne peut être trouvée en un temps raisonnable par un algorithme déterministe. Le concept y est le suivant : une population d'individus représentés chacun par un génome est générée aléatoirement et les individus sont évalués puis sélectionnés selon un critère particulier au type de problème. La nouvelle population est alors réévaluée, et le cycle recommence jusqu'à approcher une solution acceptable.

L'ontogenèse concerne la croissance d'un organisme, d'un embryon à un individu adulte composé de dix mille milliards de cellules, dans le cas de l'être humain. Chaque cellule d'un être vivant contient un génome décrivant l'organisme entier, et servant à construire un corps, puis à le réparer en cas d'agression. Basé sur ce principe, le projet embryonique [142], développé au Laboratoire de Systèmes Logiques, a vu la naissance (si l'on puis dire) d'un nouveau concept de circuits électroniques réparables. Conçu pour implémenter des systèmes cellulaires sur la base de molécules électroniques, la mort de molécules et de cellules y est compensée par l'activation de parties de rechange du circuit.

L'épigenèse, quant à elle, traite de l'apprentissage d'un individu, qui à partir de connaissances acquises, et en fonction de l'interaction qu'il entretient avec son environnement, développe un réseau de neurones impressionnant. Dans le cas de l'homme, il s'agit d'environ 10 milliards de neurones, chacun possédant jusqu'à 10'000 connexions. Il est bien clair que notre cerveau est une phénoménale machine à apprendre, et à mémoriser. Par simple observation, puis imitation, nous sommes capables d'acquérir de nouveaux savoirs. Là encore les ingénieurs se sont inspirés de ce concept en développant les réseaux de neurones artificiels, qui sont entre autre mis à contribution dans des applications de reconnaissance de formes ou de contrôle de robot.

En comparaison des solutions logicielles, le matériel, de par son parallélisme intrinsèque, offre une accélération potentielle de tels systèmes, qui peuvent être gourmands en temps de calcul. Les circuits reconfigurables permettent d'embarquer des applications mettant en jeu ces différents axes du vivant, dans l'optique de disposer de plateformes matérielles bio-inspirées. Ils ne sont toutefois pas entièrement adaptés aux mécanismes tels que l'autoréparation et la plasticité. De là est né le projet européen dénommé "POEtic", dont le but était la réalisation d'un tissu électronique permettant l'implémentation efficace d'applications bio-inspirées incluant au moins un des trois axes présentés. Cette thèse s'est donc déroulée dans le cadre de ce projet de collaboration, où notre tâche principale fut de mener à bien la réalisation de la partie programmable du circuit POEtic, appelée sous-système organique. Nous tenons à citer ici les partenaires du projet, ainsi qu'une brève description de leurs tâches respectives :

- Le Laboratoire de Systèmes Logiques, où nous avons étudié divers systèmes ontogénétiques, et principalement défini le sous-système organique, qui se trouve être l'un des sujets principaux de la présente thèse.
- L'Université de Lausanne, où une équipe de neurobiologistes a défini un nouveau type de neurone à impulsion.
- Le Laboratoire de Systèmes Autonomes, de l'EPFL, où un système de codage morphogénétique, une alternative à la correspondance directe du génotype au phénotype, a été investigué. Une application robotique du réseau de neurones développé par l'Université de Lausanne utilisant le circuit POEtic y a également été réalisée.
- L'Université Polytechnique de Catalogne, où le microprocesseur ainsi que le layout électronique du circuit furent réalisés, sur la base des fichiers VHDL que nous leurs avons fournis concernant la partie reprogrammable. Ils réalisèrent également une implémentation matérielle du neurone défini par l'Université de Lausanne.
- L'Université de York, où un système immunotronique a été développé, de même qu'un mécanisme d'autoréparation cellulaire démontrant l'efficacité du circuit construit. Un outil d'édition schématique permettant de créer des designs pour POEtic y a également été réalisé.
- L'Université de Glasgow, sur l'axe mathématique, où le modèle de l'Université de Lausanne fut validé, grâce à une armada d'outils mathématiques.

Les circuits reconfigurables actuels, comme nous le verrons dans le chapitre 2, ne proposent pas de systèmes de routage dynamique permettant de connecter deux parties du circuit lors de l'exécution. En effet, leur structure est principalement composée d'éléments de base simples, reliés entre eux par un réseau d'interconnexions plus ou moins complexe. Ces connexions ne peuvent toutefois être programmées que par un contrôleur externe, lors de la configuration du circuit.

Les systèmes bio-inspirés, tels que les réseaux de neurones artificiels à topologie variable, peuvent nécessiter la création ou la destruction de connexions en cours de fonctionnement. De même, des systèmes montrant des propriétés de développement et d'autoréparation pourraient être réalisés, utilisant la propriété de création de connexions. Des cellules de rechange pourraient alors être dynamiquement reliées au système préexistant, en remplaçant des cellules dont le bon fonctionnement n'est plus assuré.

Les nanotechnologies, quant à elles, n'en sont qu'à leurs débuts, mais promettent de révolutionner la manière de concevoir des systèmes informatiques. Nous aurons, dans un futur proche, des circuits moléculaires contenant un nombre plus que conséquent d'éléments qu'il faudra pouvoir programmer. Dès lors, les mécanismes développementaux pourront prendre toute leur importance, afin de créer des systèmes capables de s'auto-organiser, et de s'auto-réparer[1]. La capacité de créer des chemins de données dynamiquement sera nécessaire, afin de disposer d'un maximum de flexibilité et de possibilités d'autoréparation.

Le manque de routage dynamique dans les circuits reconfigurables actuels et la nécessité d'en disposer dans les futurs nano-circuits nous amènent à explorer de nouvelles voies pour la réalisation de tels circuits. Il existe des algorithmes pour la découverte du chemin le plus court entre deux ou plusieurs points, la plupart étant séquentiels. Quelques solutions parallèles ont été proposées, mais aucune n'a encore été intégrée à un circuit reconfigurable. Cette thèse se propose donc d'explorer divers algorithmes pouvant être implémentés de façon parallèle, et d'en analyser les différentes caractéristiques.

Enfin, outre le routage dynamique, les systèmes bio-inspirés multicellulaires ont besoin, lors de phases de développement ou d'autoréparation, de pouvoir modifier le comportement des cellules qui le composent. Pour ce faire, le tissu électronique doit pouvoir s'auto-configurer, ce qui est une caractéristique inexistante dans les circuits reconfigurables commerciaux. Le sous-système organique de POEtic possède donc cette capacité, qui, couplée au routage dynamique, offre une plasticité matérielle aux applications qui y seront implémentées.

1.3 Contributions

Partant du constat que la possibilité de router dynamiquement des signaux dans les circuits reconfigurables actuels n'existe pas, la présente thèse apporte les contributions suivantes :

- Les systèmes de routage dynamique existants ont été étudiés et commentés.
- Un algorithme de routage matériel distribué a été proposé et implémenté, dans lequel aucun contrôle centralisé n'est nécessaire. Il est basé sur une implémentation parallèle de l'algorithme de Lee [137].
- Un second algorithme a été implémenté, réalisant la même fonction que l'algorithme de Mikami-Tabuchi [156], qui effectue une recherche par lignes.
- Une amélioration des deux précédents algorithmes a été réalisée, de manière à réduire non pas la longueur des chemins créés, mais plutôt les problèmes de congestion.
- Ces algorithmes, après avoir été implémentés pour un voisinage de 4, le furent pour les trois autres voisinages[2] réguliers à deux dimensions, à savoir 3, 6 et 8.
- Un dernier algorithme, n'exploitant que des liaisons locales afin d'améliorer la scalabilité du système, a également été réalisé.
- Une comparaison des différents algorithmes sur le plan du temps d'exécution, de la congestion, du nombre de transistors nécessaires et de la longueur des chemins a été effectuée, de manière à pouvoir proposer la meilleure alternative en fonction de différentes contraintes.

---

[1] Les systèmes moléculaires seront très probablement plus sujets aux fautes que les implémentations actuelles sur silicium.

[2] Le voisinage désigne le nombre de voisines à laquelle une unité est reliée.

- Concernant la congestion, nous avons proposé un estimateur capable de l'approximer, de façon à pouvoir prédire les risques de congestion en fonction de plusieurs paramètres.
- De même, une comparaison des différents voisinages a été faite, afin de pouvoir choisir, en fonction notamment de la connectique et de l'espace disponible, la meilleure alternative.
- Basée sur un des algorithmes proposés, la partie reconfigurable d'un nouveau circuit, qui fut physiquement réalisé, a été conçue. Elle offre, outre les fonctionnalités standard des circuits reconfigurables FPGAs, des possibilités de routage dynamique et d'auto-configuration partielle. Ce système reconfigurable possède de nombreuses caractéristiques qui en font une plateforme idéale pour y implémenter des applications bio-inspirées multicellulaires.
- Un logiciel permettant à un utilisateur de réaliser un design pour notre circuit a été réalisé. Il permet de configurer les éléments de base de la partie reconfigurable, de les connecter entre eux grâce à une interface graphique, et de visualiser les résultats d'une simulation du design.
- La manière de tirer avantage des capacités spécifiques du circuits a été démontrée, en proposant une solution pour l'implémentation de registres à décalage et de compteurs générateurs d'impulsions.
- Un mécanisme de développement basé sur un identifiant cellulaire a été proposé, puis implémenté sur ce nouveau circuit.
- Une démonstration a été faite sur la manière d'utiliser ce circuit pour réaliser une application de matériel évolutif au niveau des portes logiques, ainsi que dans le cadre d'évolution non-contrainte.

1.4 Contenu de la thèse

Cette thèse est principalement divisée en 6 parties, outre cette introduction et la conclusion. Nous y présentons les systèmes bio-inspirés, puis les FPGAs et leurs limitations pour de tels systèmes. Le routage est ensuite introduit, avant d'entrer plus en détail dans les algorithmes développés dans le cadre de cette thèse. Finalement, un nouveau circuit reconfigurable, appelé POEtic, utilisant un de nos algorithmes de routage dynamique, est présenté, de même que différentes applications en démontrant l'utilité. La description succincte des chapitres est donnée ici :

- Le chapitre 2 présente les circuits reconfigurables (FPGAs, pour Field Programmable Gate Array) et leur évolution depuis les premiers circuits ne proposant que quelques portes logiques, jusqu'aux plus puissants FPGAs actuels qui exhibent des systèmes d'interconnexion des plus complexes. Leur limitation en terme de routage dynamique sera exposée, justifiant une partie de la présente thèse.
- Le chapitre 3 présente ce que sont les systèmes bio-inspirés, insistant sur les trois axes de la bio-inspiration, à savoir la phylogenèse, l'ontogenèse, et l'épigenèse. Il se termine par la présentation du projet POEtic, qui a pour but le développement d'un nouveau type de FPGA, un tissu POEtique, visant à faciliter l'implémentation d'applications bio-inspirées mettant en jeu des combinaisons des trois axes de la vie susmentionnés.
- Le chapitre 4 enchaîne tout naturellement sur le routage, en présentant les travaux déjà effectués sur des problèmes tels que la recherche du plus court chemin

entre deux points. Nous y introduisons les premières solutions logicielles, puis les systèmes matériels qui ont déjà été développés.
- En se basant sur les systèmes de routage présentés, le chapitre 5 introduit nos solutions. Les différentes architectures développées sont détaillées, et des tests de vitesse, de taille et de congestion sont effectués pour chaque algorithme, ainsi que pour les différents voisinages.
- Le chapitre 6 présente en détail l'architecture du circuit réalisé. Un accent particulier est mis sur la partie reconfigurable du circuit, qui fut développée dans le cadre de cette thèse. De plus, nous explicitons la manière dont certaines de ses caractéristiques peuvent efficacement implémenter certains composants de base tels que des registres à décalage ou des compteurs générateurs d'impulsions.
- Ayant défini le circuit, nous nous penchons, dans le chapitre 7, sur son efficacité quant à l'implémentation de mécanismes phylogénétiques, ontogénétiques, et épigénétiques. Ce chapitre démontre notamment l'utilité du routage dynamique et de la reconfiguration partielle.
- Finalement, la conclusion résume nos travaux, explicite les limitations du routage distribué et du circuit POEtic, et propose des extensions aux systèmes développés, qui pourraient être utiles à une potentielle deuxième version du circuit.

Chapitre 2

De la configuration des circuits électroniques

Seul l’éphémère dure.

Eugène IONESCO

LES CIRCUITS électroniques envahissent le quotidien de manière impressionnante. Ils sont présents dans les machines à laver, les montres, les voitures, et évidemment dans les ordinateurs. Qui aurait pensé, le 23 décembre 1947, date de la création du premier transistor par William Shockley, Walter Braittain et John Bardeen, physiciens des laboratoires Bell, qu’en 55 ans l’électronique allait révolutionner le style de vie de l’humanité ?

En 1952, G. W. A. Dummer, un Anglais expert en radar, publia un article proposant d’utiliser un bloc de matériel solide pour connecter des composants électroniques, sans fils de connexions. Le concept de circuit intégré était né, et six ans plus tard, en 1958, Jack Kilby, travaillant pour Texas Instruments, en réalisa le premier spécimen, sous la forme d’un oscillateur à décalage de phase contenant cinq éléments sur une seule pièce de semi-conducteur. Depuis lors, la complexité des circuits intégrés n’a cessé de croître, pour arriver à des systèmes jusqu’à 1.7 milliard de transistors pour le dernier processeur d’Intel.

La fabrication d’un circuit intégré spécifique à une application (ASIC) est une tâche ardue et coûteuse en temps et en argent. En effet, la réalisation d’un prototype nécessite la création d’un ou plusieurs masques[1], qui sont très coûteux s’il ne sont destinés qu’à une petite quantité. De plus, une simple erreur dans le design du système implique la création d’un ou plusieurs nouveaux masques. Et finalement, l’investissement en temps de développement est important, et n’est pas forcément tolérable si un produit doit très rapidement être mis sur le marché.

Parallèlement à l’avancée des ASICs, divers circuits programmables ont fait leur apparition, afin de réduire le temps et le coût de développement des circuits électroniques, tout en restant relativement compétitifs sur le plan de la rapidité d’exécution.

---

[1] Un masque sert à apposer des couches de métal sur le silicium, dans l’optique de relier entre eux les divers éléments tels que les transistors (ce type de technique est également exploitée pour le dopage du silicium).

La question est alors de pouvoir concevoir des systèmes contenant des circuits intégrés de la manière la moins coûteuse en terme de temps et d'argent. En fonction de différents paramètres tels que la complexité du système, le nombre de pièces à fabriquer ou le temps imparti, un type d'implémentation est choisi parmi la gamme des possibilités.

Concernant les systèmes bio-inspirés, qui seront introduits dans le chapitre suivant, un nombre non négligeable d'applications ont mis en jeu des circuits programmables : à titre d'exemples, le large parallélisme des réseaux de neurones s'applique parfaitement à une implémentation matérielle [56], et certains chercheurs travaillent à un nouveau type de systèmes informatiques tolérants aux pannes [142] dont les prototypes nécessitent la possibilité de se reprogrammer.

Dans ce chapitre, nous allons évoquer les différents types de circuits intégrés non-programmables, programmables et reprogrammables, allant de la simple mémoire programmable au circuit FPGA (Field Programmable Gate Array) [33] de plus de 10 millions de portes logiques reprogrammables, en passant par les circuits logiques programmables simples et les circuits spécifiques à une application. Nous allons tout d'abord présenter brièvement les processeurs puis les différents types de circuits ASICs. Ensuite nous présenterons les technologies de programmation matérielle existantes et l'évolution des circuits programmables PLD (Programmable Logic Device), en suivant un ordre chronologique, jusqu'à arriver aux FPGAs, qui feront l'objet d'une attention toute particulière. L'optique de cette thèse étant de proposer une nouvelle FPGA possédant des possibilités de routage et d'auto-configuration non présentes dans les circuits commerciaux, la description de tous les circuits se fera en mettant un accent particulier sur leurs possibilités et limitations en terme de routage.

2.1 Processeur

Un processeur [185] est un circuit intégré permettant d'implémenter n'importe quelle fonction en exécutant de manière séquentielle un code compilé. Le grand avantage du processeur est sa généricité, puisque n'importe quel programme peut y être exécuté. De plus, il est très facile de le programmer, c'est-à-dire de transcrire dans le langage du processeur un problème particulier.

Malheureusement, cette généricité s'accompagne d'un coût, à savoir le temps d'exécution d'une tâche. En effet, les instructions sont exécutées une à une (bien que certains processeurs présentent un certain parallélisme), l'opération $a + 2 + b + d$ nécessitant au minimum trois pas de temps. Un système matériel peut, lui, effectuer cette opération en un seul pas. De plus, le processeur étant une unité chargée d'un calcul, une seule erreur compromet l'ensemble de son exécution. Les systèmes parallèles laissent quant à eux la porte ouverte à l'autoréparation, une unité pouvant potentiellement prendre la place d'une autre se trouvant dans un état défectueux.

Les applications intrinsèquement parallèles tels que les réseaux de neurones ou les automates cellulaires sont donc exécutées de manière séquentielle par un processeur, alors qu'un système matériel a la possibilité d'exploiter tout le parallélisme possible. Les systèmes cellulaires composés de plusieurs cellules effectuant des tâches en parallèle sont donc bien plus efficacement réalisés en matériel qu'en logiciel. C'est pourquoi le reste de cette thèse traite de matériel, en enchaînant directement sur les ASICs, une des solutions d'implémentation de tels systèmes.

2.2 ASIC

Le Circuit Intégré Spécifique à une Application (ASIC) [219] est une des manières de réaliser un système de calcul matériel. Pour chaque application, un circuit différent est créé, soit en le construisant entièrement, soit en configurant une grille d'éléments préconstruits. L'avantage des ASICs sur les circuits reconfigurables, pour une même technologie, est évidemment leur rapidité, puisque les connexions sont créées physiquement plutôt que programmées. Toutefois, leur force est également leur faiblesse, puisque leur conception nécessite plus de temps, étant donné qu'en tout cas une couche de métal doit être apposée. De plus, ils ne sont en aucun cas programmables, et une erreur dans le design implique un changement complet du circuit.

Quatre grandes classes d'ASICs sont présentées ici, le graphique 2.1 les exposant avec un niveau de complexité croissant de gauche à droite. Nous allons les détailler dans l'ordre chronologique de leur apparition.

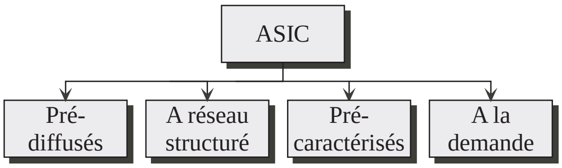


Figure 2.1 : *Les différents types de circuits ASIC.*

2.2.1 A la demande

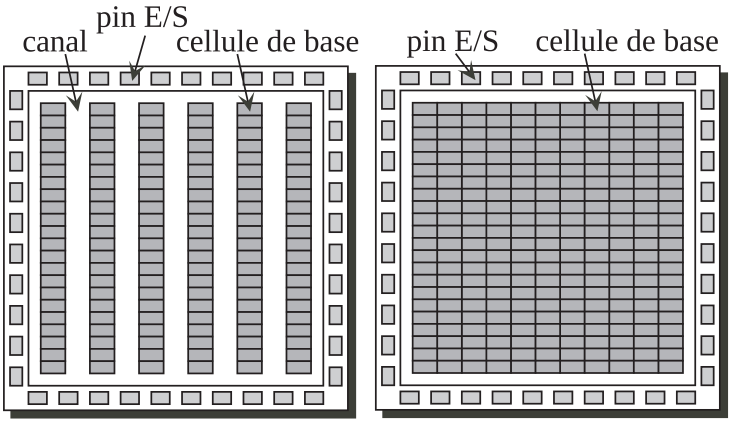
Pour la création d'un ASIC à la demande, communément appelé Full-Custom [48], les ingénieurs contrôlent chaque élément du circuit fabriqué. Ils dessinent les transistors, les placent et les connectent, sans qu'aucune contrainte prédéfinie n'existe concernant le placement des éléments. La taille de chaque transistor peut être changée afin d'optimiser le fonctionnement du système. L'avantage de cette approche est l'efficacité du circuit final en terme de quantité de silicium et de vitesse. En effet, il est possible de placer les éléments de manière optimale, sans aucune perte de place, et de le faire en tenant compte des contraintes de temps d'exécution. Par contre, si une erreur s'est subrepticement glissée dans la conception, le dessin du circuit entier doit être revu, dans un long et coûteux processus de correction. Ce type d'ASIC n'est donc pas du tout adapté au prototypage, étant donné le temps et l'investissement nécessaire à chaque prototype.

Les possibilités de routage dans un tel circuit sont quasiment infinies, étant donné que le développeur a total contrôle sur toute la conception du circuit. La seule limitation intervient par le nombre de couches de métal apposées sur le silicium. Toutefois, ce routage est fixé une fois pour toutes, ne laissant aucune possibilité de modification ultérieure.

2.2.2 Prédiffusés

Dans le milieu des années 60, Fairchild Semiconductor créa le précurseur des tableaux de portes, sous la forme d'un circuit appelé Micromatrix, composé d'une centaine de transistors. Les ingénieurs devaient dessiner à la main les interconnexions sur des feuilles de plastique, pour réaliser un circuit spécifique. Ensuite, en 1967, un nouveau circuit, appelé Micromosaic, et contenant quelques centaines de transistors, fut proposé par la même société, avec cette fois-ci un programme capable de générer automatiquement les interconnexions en fonction des spécifications des ingénieurs. Ce sont toutefois les compagnies telles qu'IBM et Fujitsu qui développèrent le concept d'ASIC prédiffusé (ou à tableau de portes)[2].

Dans le cas des mers de portes [99], les vendeurs fabriquent un circuit composé d'un tableau de cellules de base tels que des transistors. Une deuxième variante, le tableau de portes avec canaux de routage, propose plusieurs rangées de ces cellules séparées par des canaux de routage pouvant être connectés à ces éléments (Figure 2.2). La connectique interne à une rangée permet de créer des modules simples du type multiplexeurs ou bascules, et les canaux de routage offrent la possibilité de connecter ces modules entre eux. Le fabricant fournit généralement une librairie de modules, que le développeur peut utiliser, et qui sont ensuite placés sur le tableau en connectant les éléments de base grâce a des couches de métal supplémentaires.



(a) Un tableau de portes à canal

(b) Une mer de portes

Figure 2.2 : *Deux types de circuits prédiffusés.*

L'un des avantages des circuits prédiffusés est que le vendeur peut fabriquer une grande quantité de pré-circuits identiques, et qu'il n'est nécessaire, pour réaliser un circuit final, que d'ajouter des couches de métal, sans se soucier de la préparation du silicium. En contrepartie, il y a souvent un nombre non négligeable d'éléments de base inutilisés, le placement des transistors étant contraint dans le sens où ils sont déjà incrustés dans le silicium, et le routage n'est donc pas optimal. Ces différents facteurs font que le tableau de portes est nettement moins efficace qu'une implémentation full-custom en terme de consommation et performance. Il offre cependant un temps de réalisation nettement plus faible, et également un moindre coût.

---

[2] En anglais *gate array*, et *sea of gates* pour mer de portes. Connu également sous l'appelation MPGA pour Metal Programmable Gate Array ou Mask Programmable Gate Array.

2.2.3 Pré-caractérisés

Les ASIC pré-caractérisés (en anglais Standard Cell)[220] ont vu le jour au début des années 80, dans le but de combler les lacunes des circuits prédiffusés. Contrairement à ces derniers, le silicium n'est pas prédiffusé avec des éléments de base. Le vendeur propose ici une librairie de cellules de base, ainsi que des modules plus complexes tels que des RAMs, des microprocesseurs, et bien d'autres. Le développeur utilise alors ces divers éléments pour créer une liste de modules et de leurs interconnexions décrivant son système, et le vendeur se charge ensuite de créer le circuit intégré.

L'avantage de cette approche réside dans l'optimisation des modules fournis par le fabriquant. Le placement de ces modules est réalisé de manière à également optimiser les performances, tout en minimisant la place nécessaire sur le silicium. Dès lors, la solution pré-caractérisée est très proche de l'optimal obtenu grâce à l'approche full-custom, et nécessite un temps de développement nettement moindre.

2.2.4 A réseau structuré

Comme nous le verrons à la section suivante, les circuits reconfigurables ont l'avantage d'être d'excellents candidats pour les applications industrielles de petit volume ainsi que pour le prototypage, un design pouvant être immédiatement testé. Toutefois, ils ne peuvent, en terme de vitesse, rivaliser avec un circuit ASIC spécialement conçu pour une application. C'est pourquoi en 2003 le dernier-né des circuits programmable a fait son apparition[3] : l'ASIC à réseau structuré [257].

Un ASIC à réseau structuré consiste en un circuit composé de différents éléments semblables à ceux que l'on peut trouver dans un FPGA standard, tels que look-up tables[4], inverseurs, flip-flops, et bien d'autres. Toutefois, à l'inverse du FPGA, ces différents blocs ne sont pas connectés entre eux. Par analogie avec les mers de portes, on pourrait les appeler mers de macros, des modules de complexité supérieure étant prédiffusés. Le circuit de base créé, il reste donc des couches de métal disponibles qui permettent de réaliser la connectique nécessaire à la réalisation d'un design particulier.

Certains fabricants proposent des solutions mettant en jeu des look-up tables et différentes portes logiques, ainsi que des blocs de mémoire, des DLLs (Delay-Locked Loops) et des PLLs (Phase-Locked Loops). Les outils de conception, responsables de la synthèse et du placement-routage, sont petit à petit adaptés à cette nouvelle technologie. Altera, un des leaders du marché du FPGA, offre quant à lui le concept de HardCopy, un circuit contenant exactement les mêmes éléments de base que les FPGAs qu'il propose sans toutefois qu'ils soient interconnectés. A l'heure actuelle, les circuits Stratix, APEX 20Ke et APEX 20KC supportent cette solution. Un utilisateur ayant réalisé et testé un système sur un de ces FPGAs peut alors le transférer sur un ASIC structuré correspondant, en étant sûr de garder la même fonctionnalité tout en augmentant potentiellement la fréquence de fonctionnement.

A ce jour peu de tests ont été effectués dans le but de comparer cette méthodologie avec d'autres, mais les résultats préliminaires tendent à prouver que les ASICs à réseau structuré occupent trois fois plus de place que les ASICs pré-caractérisés, et consomment deux à trois fois plus [146].

---

[3] Notons toutefois que le concept est apparu au début des années 90, sans connaître grand succès.

[4] Une look-up table, ou table de correspondance en français, fait correspondre une sortie à chaque combinaison d'entrées. Dans nos systèmes digitaux, nous parlerons de k-LUT pour une look-up table à k entrées. Dans ce cas, 2^k bits définissent la sortie pour chacune des combinaisons possibles des entrées.

2.2.5 Limitations

Les quatre types d'ASIC présentés sont tous des circuits fixes, qui une fois réalisés n'offrent aucune souplesse, leur fonctionnalité étant définitivement figée. L'avantage de cette approche est la rapidité du circuit final, ainsi que la place nécessaire à l'implémentation d'un design particulier. Toutefois, la non-reprogrammabilité de tels circuits est un important obstacle au prototypage, une réalisation physique pouvant demander jusqu'à 4 mois. De plus, la fixité de tels circuits ne permet pas la conception de systèmes adaptables, capables d'évoluer en fonction d'une tâche à résoudre dans un environnement qui peut être changeant. C'est pourquoi nous allons maintenant présenter les circuits programmables, qui se proposent de pallier au manque de flexibilité des ASICs. Nous allons tout d'abord présenter les différentes technologies de programmation avant d'aborder les circuits programmables dans l'ordre chronologique de leur apparition sur le marché, correspondant également à l'ordre de complexité.

2.3 Technologies de programmation

N'ayant pas peur des truismes, commençons par constater qu'un circuit programmable doit pouvoir être programmé, de même qu'un circuit reprogrammable doit pouvoir être reprogrammé. Nous allons donc présenter ici différentes technologies servant à la réalisation de circuits programmables ou reprogrammables. Nous commencerons par les masques, les fusibles et les antifusibles, non reprogrammables, puis nous enchaînerons sur les technologies reprogrammables, à savoir l'EPROM, l'EEPROM, la FLASH et la SRAM.

2.3.1 Masque

Comme nous l'avons vu précédemment, certains circuits ASIC (prédiffusés et à réseau structuré) peuvent être programmés grâce à un masque. Ce procédé, qui est également utilisé pour la création de mémoires à lecture seule (ROM), consiste à fabriquer un circuit intégré en y plaçant les composants et quelques ou aucune couche de métal. Ensuite, l'ajout d'une ou plusieurs couches de métal supplémentaires suffit à définir exactement la fonctionnalité du circuit. Un ou des masques sont donc générés pour chaque circuit différent, et servent à imposer les dernières couches de métal.

Dans le cas d'une mémoire ROM, qui n'est accessible qu'en lecture, après sa réalisation, toutes ses connexions sont figées. Il est possible de créer une ROM spécifique à partir de rien, ou de la préconstruire et d'utiliser la technique de masquage. Dans ce deuxième cas, une cellule mémoire est semblable à celle de la figure 2.3, où l'ensemble de la mémoire est répartie en ligne/colonne, la ligne servant d'adresse, et la colonne de donnée, une seule ligne pouvant être active à un instant donné. La programmation se fait par l'ajout d'une couche de métal, connectant certains transistors à leur colonne, une résistance en pull-up forçant la valeur à '1' si la connexion n'est pas établie, ou si le transistor est ouvert. Dans le cas de l'activation d'une ligne, chaque cellule de la ligne active fournit à sa colonne la valeur mémorisée, qui dans notre exemple, est un '1' si la connexion n'est pas programmée, et '0' sinon.

Notons que la réalisation d'un circuit par masquage offre un avantage en terme de délais, ces circuits étant les plus rapides des circuits programmables. Cependant

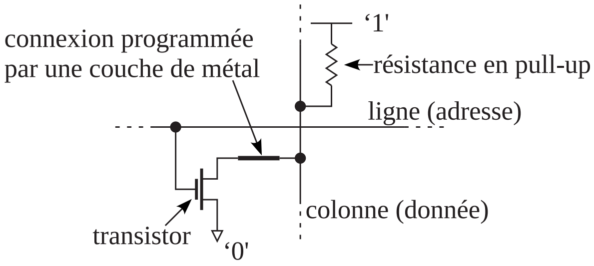


Figure 2.3 : *Une cellule de mémoire ROM programmée par masque, basée sur un transistor.*

le fait de devoir appliquer des couches de métal est un processus long et coûteux en comparaison des autres techniques.

2.3.2 Fusible

Basé sur le même principe que les fusibles domestiques, qui, pour éviter la destruction de nos appareils électroménagers, sont détruits par un trop fort courant, la technologie basée sur des fusibles fut une des premières à être utilisée. Le principe est simplement d’avoir des fusibles sur certains fils, et d’en faire brûler certains en leur appliquant un courant trop important. Sur chacun de ces fils, une résistance en pull-up ou pull-down force la valeur à 1 ou à 0 si le fusible a été détruit, comme illustré à la figure 2.4.

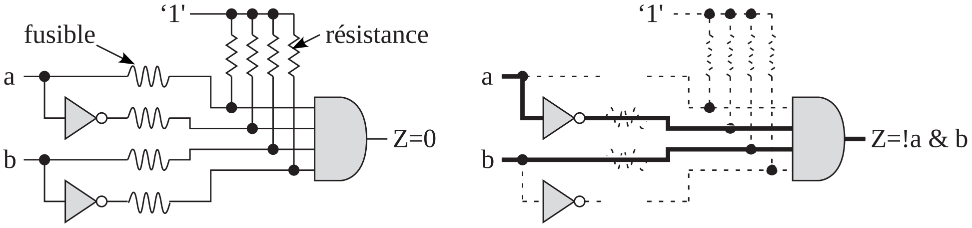


Figure 2.4 : *Un circuit contenant 4 fusibles non programmés, puis le circuit résultant après avoir brûlé le premier et le quatrième fusible.*

2.3.3 Antifusible

A ce que l’on peut considérer comme l’opposé des fusibles, se trouvent les antifusibles. De la même manière, sur certains fils du circuit se trouvent des antifusibles, mais contrairement aux fusibles, lorsqu’ils ne sont pas programmés, ils agissent comme une résistance infinie, comme si le fil était coupé. En appliquant un fort courant de grand voltage, l’antifusible est programmé, et laisse dès lors passer le courant. Une résistance est également placée après chaque antifusible, de manière à forcer la ligne à 1 ou 0 dans le cas où il n’est pas programmé (Figure 2.5).

Notons que les circuits à base de fusibles et d’antifusibles ne peuvent être programmées qu’une seule fois. Chaque nouvelle implémentation, en cas d’erreur de design par exemple, implique dès lors la programmation d’un nouveau composant et l’élimination de l’ancien. Cependant, cette programmation étant électrique, un simple appareil

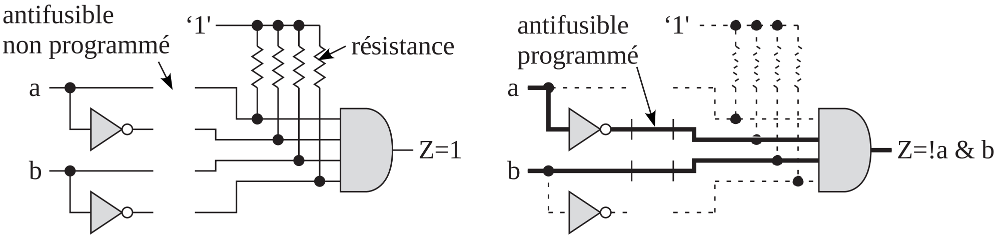


Figure 2.5 : *Un circuit contenant 4 antifusibles non programmés, puis le circuit résultant d'une programmation.*

est nécessaire, rendant le processus nettement plus simple et rapide que le masquage.

2.3.4 EPROM

Avant de décrire la technologie EPROM, passons quelques instants sur le concept de mémoire PROM. Le fonctionnement d'une mémoire PROM est identique que celui d'une ROM, si ce n'est le mode programmation. Au lieu de devoir physiquement créer des connexions grâce à une couche de métal, un fusible peut être brûlé ou non, ce qui est nettement plus rapide et moins coûteux (Figure 2.6). En effet, lors de la réalisation d'un système informatique, il n'est pas rare que des erreurs se glissent dans un design, et la destruction d'une PROM à chaque erreur n'est pas la panacée.

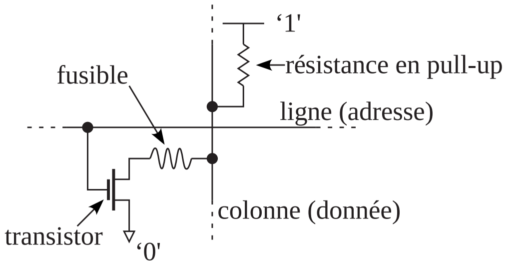


Figure 2.6 : *Une cellule de mémoire PROM, basée sur un transistor et un fusible.*

La technologie de type Erasable Programmable Read-Only Memory (EPROM), introduite en 1971 par Intel est dès lors une nette amélioration de la PROM sur le plan de la flexibilité, puisqu'elle permet une reprogrammation du circuit. Il s'agit d'un nouveau type de transistor, basé sur un transistor MOS, auquel une couche de silicium polycristallin, appelé porte flottante, a été ajoutée, isolée par des couches d'oxyde (Figure 2.7). Dans son état initial, le transistor agit normalement, tel un MOS standard. En appliquant un courant de haut voltage (typiquement 12V) entre la grille et le drain, un effet tunnel charge la porte flottante en électrons, ce qui a pour effet de bloquer le transistor en état ouvert. A la suite de la programmation, quelle que soit la tension appliquée au contrôle, aucun courant ne peut passer entre la source et le drain. La charge créée par la programmation perdure, même lorsque le circuit est hors tension, et ce n'est qu'avec une exposition du circuit à des rayons UV que la programmation est effacée. Lié à l'effacement, un des désavantages de l'EPROM est qu'une fenêtre en quartz doit être apposée sur le circuit, afin de pouvoir laisser passer les rayons UV, ce qui augmente grandement le prix du boîtier.

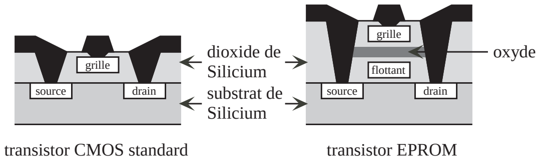


Figure 2.7 : *Un transistor CMOS standard et un transistor EPROM.*

2.3.5 EEPROM/Flash

Le principal inconvénient des EPROMs, la déprogrammation par rayons UV, est oublié dans les EEPROMs, qui sont effaçables électriquement. Il n'est dès lors plus nécessaire d'intervenir physiquement avec de la lumière UV, un système électronique pouvant reprogrammer l'EEPROM de manière autonome. Sur le plan de l'espace pris sur le silicium, une cellule mémoire d'EEPROM occupe toutefois environ 2,5 fois plus de place qu'une cellule d'EPROM, car elle est composée de deux transistors, ainsi que de l'espace nécessaire entre eux deux (Figure 2.8).

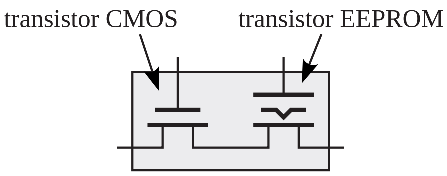


Figure 2.8 : *Une cellule mémoire EEPROM.*

Les mémoires FLASH [24] sont basées sur le même principe d'effacement électronique et de non-volatilité. De nombreuses expériences et implémentations en ont fait des composants très largement utilisés à l'heure actuelle. Vues de l'extérieur, les FLASH sont quasiment identiques aux EEPROMs, si ce n'est que l'effacement se fait par secteur, et non octet par octet. Elles sont toutefois souvent préférées, de par leur vitesse et leur taille, qui peut être plus imposante que celle des EEPROMs.

2.3.6 SRAM

La technologie SRAM, pour Static Random Access Memory, est, contrairement à celles déjà présentées, volatile, et doit donc être reprogrammée à chaque remise en marche du système. Leur second désavantage réside dans la place nécessaire à une cellule, qui est composée de quatre à six transistors formant un latch (Figure 2.9).

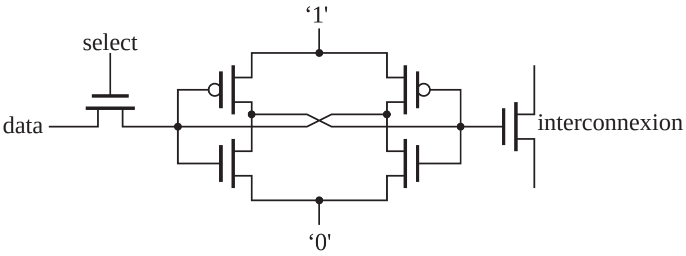
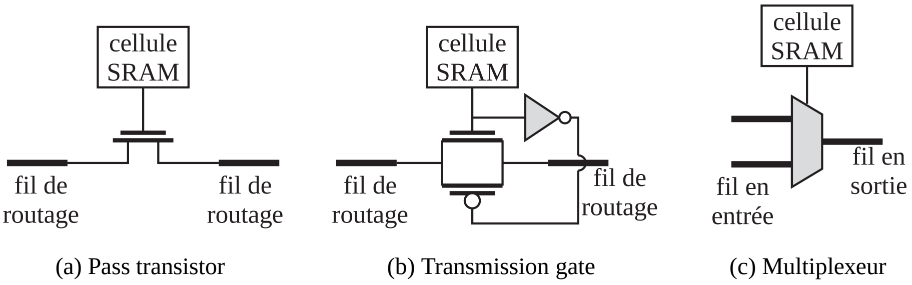


Figure 2.9 : *Une cellule mémoire SRAM.*

Toutefois, la grande facilité de programmation des circuits en technologie SRAM en a fait le choix privilégié des deux plus grands fabricants de FPGAs, à savoir Xilinx et Altera. La figure 2.10 illustre les différentes manières d'utiliser des cellules de mémoire SRAM pour la réalisation de circuits programmables. Les FPGAs commerciaux basés sur cette technologie sont le plus souvent implémentés avec des pass-transistors ou des portes de transmission. Seul le fameux XC6200, que nous décrivons à la page 23 est réalisé avec des multiplexeurs. Cette dernière technique offre l'avantage d'interdire tout court-circuit, contrairement aux deux autres. Toutefois, cet avantage se paie par une baisse de performances en terme de rapidité, un signal mettant plus de temps à passer un multiplexeur qu'un simple transistor.



(a) Pass transistor (b) Transmission gate (c) Multiplexeur

Figure 2.10 : *Trois technologies de programmation associées à la RAM statique.*

2.3.7 Résumé

Nous avons présenté différentes technologies utilisées dans le cadre des circuits programmables. Le tableau 2.1 résume les caractéristiques des trois principales techniques utilisées dans les circuits programmables. Les antifusibles sont d'excellents candidats pour les systèmes nécessitant une très haute fréquence d'horloge, mais ont le très net problème de ne pouvoir être reprogrammés. Les fusibles ne sont plus utilisés, étant donné qu'ils sont couplés à une technologie bipolaire, qui n'est plus d'actualité. Les technologies EPROM et EEPROM permettent cette reprogrammation, mais au coût d'une intervention humaine et un temps d'effacement de une à quinze minutes pour les premières, et d'un temps d'effacement non négligeable pour les deuxièmes, ainsi que d'une forte tension, de l'ordre de 12V pour la reprogrammation. Les cellules SRAM, quant à elles, sont très rapidement reprogrammables, au coût d'une plus grande place occupée et de leur volatilité.

Type	**EPROM**	**Antifusible**	**SRAM**
Rapidité	–	+	–
Densité	–	+	– –
Facilité	+	–	+
Reprogrammabilité	+	–	++

Tableau 2.1 : *Comparaison des caractéristiques des différentes technologies.*

2.4 Circuits logiques programmables

Les circuits programmables sont apparus en 1970, et depuis se sont complexifiés de manière spectaculaire. Il existe dans la littérature plusieurs manières de les classer, certains considérant que les CPLDs (Complex Programmable Logic Device) ne sont pas un sous-ensemble des PLDs (Programmable Logic Device). Nous choisirons toutefois l'approche illustrée à la figure 2.11 consistant à séparer les PLDs en deux sous-classes, à savoir les SPLDs (Simple Programmable Logic Device) et les CPLDs.

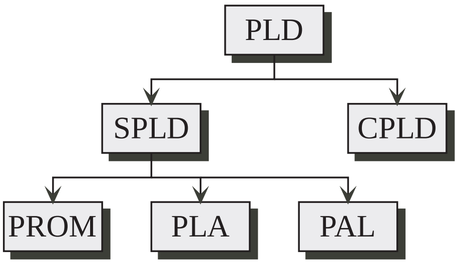


Figure 2.11 : *Une classification des circuits logiques programmables.*

Nous allons décrire les différents types de circuits programmables, dans l'ordre chronologique de leur apparition, correspondant également à leur complexité, en commençant par les SPLD, les CPLDs, puis les FPGAs.

2.4.1 SPLD

Les SPLDs (Simple Programmable Logic Device)[36], dans une description haut niveau, sont composés d'une grille de portes ET et d'une grille de portes OU, les deux étant reliées. Les entrées du système peuvent être connectées aux portes ET, et le résultat des portes OU correspond à la sortie du système (Figure 2.12). Dans ces circuits, les connexions sont préexistantes, les différentes lignes étant reliées par des fusibles, des transistors EPROM, ou des transistors EEPROM. En brûlant certains de ces fusibles, ou en programmant les transistors, il est alors possible de réaliser différentes fonctions logiques.

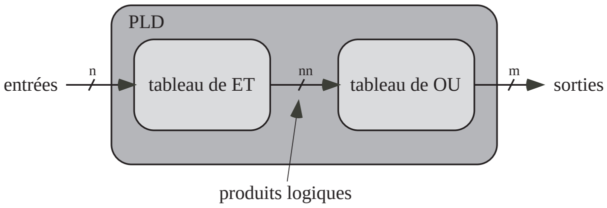


Figure 2.12 : *L'architecture d'un SPLD.*

PROM

Les premiers circuits programmable à avoir vu le jour sont les PROMs (Programmable Read-Only Memory) (Figure 2.13), le terme PROM étant introduit en 1970. Ces circuits sont des mémoires accessibles en lecture, qui contrairement aux ROMs, sont programmables. En effet, une ROM est livrées déjà configurée, et ne peut être accédée

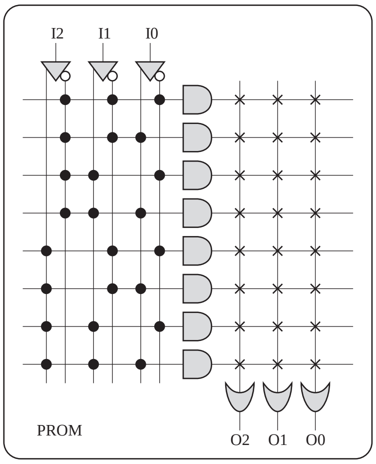


Figure 2.13 : *L'architecture fonctionnelle d'une PROM.*

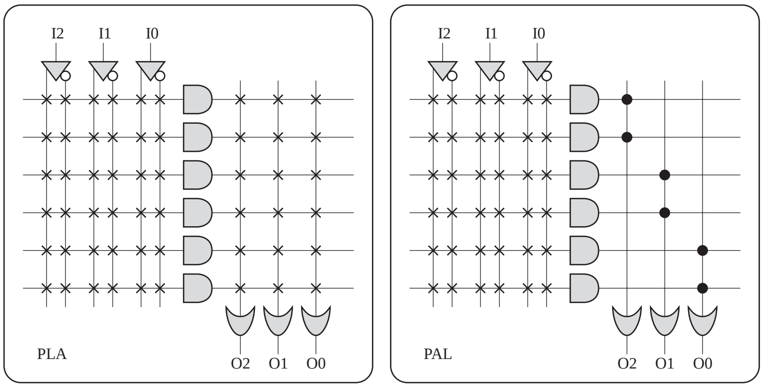
qu'en écriture, alors qu'une PROM est une mémoire vide, qui peut être écrite, une seule fois, par l'utilisateur. Après cette unique écriture, une PROM se comporte exactement comme une ROM. Ces circuits, qui ne sont pas reprogrammables, sont répartis en deux sous-classes, les Mask-Programmable Chips, programmés par le fabricant, et les Field-Programmable Chips, qui sont programmés par l'utilisateur.

Notons également qu'à l'origine les PROMs étaient vouées à faire office de mémoire d'instruction pour les ordinateurs. Leur emploi s'est toutefois généralisé puisqu'on les a utilisées pour l'implémentation de fonctions logiques simples tels que des look-up tables ou des machines d'état. Les PROMs se sont très vite imposées dans ce domaine, de par le fait qu'elles étaient plus petites, moins lourdes, moins chères, et moins sujettes aux erreurs que les systèmes composés de plusieurs circuits comportant des portes logiques. De plus, une erreur de design pouvait être rapidement modifiée en programmant une nouvelle PROM, ce qui était nettement plus rapide et simple que de modifier un circuit imprimé.

L'avantage des PROMs sur les autres PLDs est leur efficacité pour l'implémentation de fonctions logiques nécessitant un grand nombre de produits, mais elles ont le désavantage de n'accepter qu'un nombre limité d'entrées, étant donné que toutes les combinaisons possibles des entrées sont décodées (la figure 2.13 illustre ce décodage par la partie ET et la programmation de la partie OU).

PLA

Les PLAs (Programmable Logic Array) [21] apparurent aux environs de l'année 1975 dans le but de pallier les limitations des PROMs. En effet, dans un PLA (Figure 2.14(a)), contrairement à une PROM, toutes les interconnexions peuvent être programmées, ce qui en fait le PLD le plus général. Il est alors possible de définir les produits et les sommes, les rendant particulièrement efficaces lorsque plusieurs sorties utilisent les même produits. Une amélioration s'accompagnant souvent de désagréments, citons un des désavantages du PLA comparé à la PROM : étant donné que les deux tableaux (ET/OU) sont programmés, le temps de propagation d'un signal de l'entrée à la sortie est nettement plus important que lorsqu'un seul tableau l'est.



(a) *L'architecture d'un PLA* (b) *L'architecture d'un PAL*

Figure 2.14 : *Architectures d'un PLA et d'un PAL.*

PAL

Vers la fin des années 1970, les PALs (Programmable Array Logic) [133] furent introduites afin de contrer le problème de vitesse de propagation des PLAs. Dans un PAL (Figure 2.14(b)), les connexions entres les portes ET et OU sont fixes, et les connexions entre les entrées et les portes ET peuvent être programmées. L'avantage des PALs sur les PLAs est donc leur rapidité, mais elles présentent l'inconvénient de n'avoir qu'un nombre limité de produits pour chaque porte OU.

Résumé

Nous venons de présenter l'architecture générale des PLDs. Avant de continuer notre exploration des circuits programmables, notons que cette architecture a vu bon nombre de variantes voir le jour. De nombreux circuits possèdent la capacité d'inverser les sorties, de disposer de portes tri-state, ou même de faire passer les sorties par des registres. Certains autres ont même la capacité d'utiliser leurs pins comme sorties ou comme entrées supplémentaires.

Sur le plan du routage, nous pouvons noter qu'il est très limité dans le sens où il n'y a pas réellement de connectique à définir. En effet, la programmation d'un PLD agit sur la réalisation d'une fonction plutôt que sur la connexion de divers blocs de fonctions.

2.4.2 CPLD

Les SPLDs présentent deux limitations majeures, à savoir l'impossibilité de réaliser des fonctions à plusieurs niveaux et celle de ne pouvoir partager les produits de différentes fonctions. Les CPLDs (Complex Programmable Logic Device), apparus au début des années 80, sont donc le résultat de l'évolution des PLDs. Ils permettent l'implémentation de systèmes nettement plus complexes, et sont composés d'éléments de base programmables, connectés entre eux par un réseau d'interconnexions relativement simple. Ces éléments de base sont du type SPLD, et peuvent, comme dans le cas de la famille MAX3000 d'Altera [8], être composés d'un tableau de portes ET programmables et de portes OU fixes (une sorte de PAL), ainsi que d'un registre. La technologie de programmation des CPLDs dépend évidemment du constructeur, et peut être de type EPROM, EEPROM, FLASH ou SRAM.

Notons qu'un des avantages des CPLDs sur les FPGAs, que nous présenterons plus loin, est la rapidité. En effet, le réseau d'interconnexions, en étant nettement plus rudimentaire, est plus rapide que celui d'un FPGA. De plus, les connexions se font toujours avec une destination pour une source, et le temps de propagation est donc toujours le même. Le placement d'un design dans un CPLD n'est donc pas critique, et le routage peut être systématisé, sans avoir besoin de tenir compte de contraintes de temps.

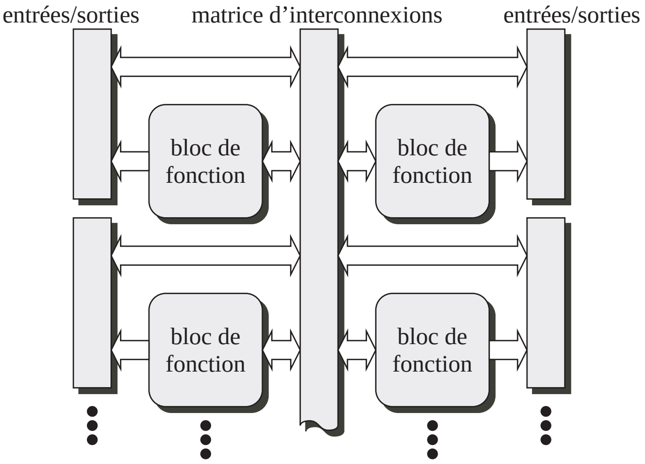


Figure 2.15 : *L'architecture d'un CPLD*

Le routage des CPLDs commence à être intéressant, puisqu'en plus de définir la fonctionnalité de blocs simples comme les SPLDs, il devient possible de les interconnecter.

2.5 FPIC

Classés hors catégorie figurent les Field Programmable Interconnect Chips ou Devices (FPIC ou FPID) [12, 39, 86]. Il s'agit de circuits d'interconnexions programmables qui font office de switchbox. Leur structure est semblable à celle du réseau de routage des FPGAs, la principale différence étant le fait qu'ils ne possèdent aucun élément logique capable d'effectuer du calcul. Placés sur un circuit imprimé, ils permettent de définir au démarrage les connexions entre différentes parties du circuit. Leur avantage est donc de permettre une plus grande souplesse dans la mise au point d'une plateforme comprenant plusieurs circuits intégrés. Par exemple, la réalisation d'un système multi-fpga peut tirer avantage des FPICs dans le but de connecter ensemble les FPGAs avec un circuit également reprogrammable.

2.6 FPGAs

Nous en arrivons au point central de ce chapitre, à savoir les circuits de type FPGA (Field Programmable Gate Array). Au début des années 80, les développeurs disposaient des circuits de type PLD, facilement configurables, mais ne pouvant contenir des designs trop complexes. Les ASICs, quant à eux, supportaient des systèmes de grande complexité, mais n'avaient pas les propriétés de configuration des PLDs. Il manquait donc un type de circuits permettant la réalisation de systèmes complexes, tout en offrant une reconfiguration rapide et peu onéreuse. C'est pourquoi en 1984, Ross Freeman, Bernie Vonderschmitt, et Jim Barnett fondent la compagnie Xilinx. En 1985, ils introduisent sur le marché le premier FPGA, le XC2064, et offrent ainsi une alternative aux précédentes approches.

Les circuits FPGAs [32, 203, 244] permettent d'implémenter des systèmes numériques aussi complexes que ceux réalisés jusqu'alors grâce aux ASICs, tout en ayant le grand avantage de pouvoir être programmés électriquement. Ils sont principalement composés d'un tableau d'éléments plus ou moins complexes pouvant être configurés, ainsi que d'un réseau complexe de connexions également configurables (Figure 2.16).

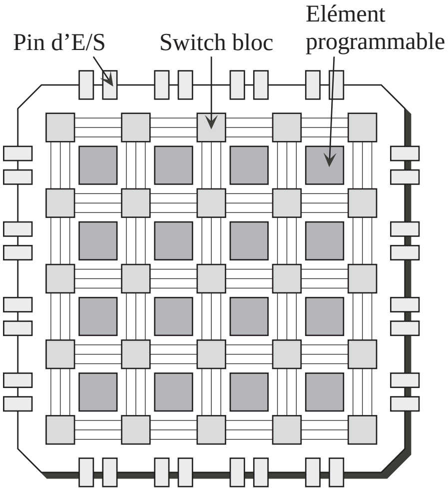


Figure 2.16 : *L'architecture générale du FPGA.*

Nous allons présenter ici deux circuits de Xilinx, les circuits d'Altera présentant le même type d'architecture. Nous aborderons le premier FPGA, le XC2064, puis nous accorderons une attention particulière à la famille XC6200, qui fut la plus utilisée dans les applications de matériel évolutif, décrites à la page 49. Nous verrons ensuite la manière dont les circuits se complexifient en embarquant de plus en plus de composants tels que RAM, multiplicateurs, processeurs, et d'autres. Afin de ne pas surcharger ce chapitre, nous ne présenterons pas les derniers-nés des FPGAs en détail, leur architecture n'étant qu'une variation évoluée du XC2064, mais nous proposerons une comparaison des circuits existant proposés par les six fabricants de FPGAs.

2.6.1 XC2000

Nous allons nous attarder quelque peu sur le premier FPGA commercial, le XC2000 de Xilinx [37]. Son architecture est relativement simple en comparaison des énormes circuits actuels, mais sensiblement identique à l'ensemble des FPGAs. Il est basé sur une technologie SRAM, de la même manière que tous les FPGAs de Xilinx, et est donc reprogrammable un nombre illimité de fois, et ce de manière très rapide.

Son élément de base, le CLB, pour Configurable Logic Bloc (Figure 2.17), est composé d'une bascule et de deux look-up tables de trois entrées qui ont également la possibilité de réaliser une look-up table à quatre entrées (suivant la dénomination introduite par Hill et Woo dans [96], il s'agit d'une look-up table à 4 entrées et 2 sorties[5]). Il est intéressant de noter que les deux sorties, X et Y, sont interchangeables, puisque leurs multiplexeurs ont exactement les mêmes entrées. Ce détail sera intéressant sur le plan du routage, comme nous le verrons plus loin.

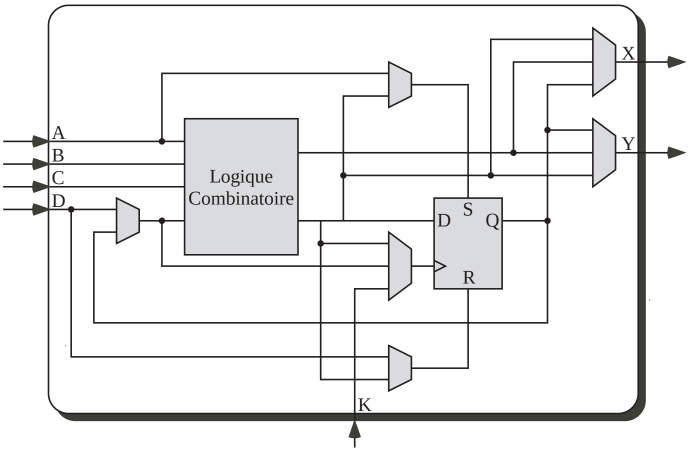


Figure 2.17 : *Le CLB d'un XC2000.*

Sur le plan des interconnexions, le XC2000 contient des switchboxs, chacun étant connecté à quatre autres switchboxs, comme présenté sur la figure 2.18. Ils sont reliés verticalement par cinq fils, et horizontalement par quatre. Ces switchboxs permettent de relier des CLBs sur de longues distances, au travers du FPGA, les problèmes liés au délai RC des noeuds routés étant évités en plaçant des buffers sur certains fils. Le choix a été fait de diviser le FPGA en neuf parties, et de placer des buffers sur toutes les frontières de ces parties. Cette structure de routage est encore présente dans les FPGAs actuels, dans lesquels nous trouvons des canaux de routage accessibles par les unités fonctionnelles, et qui sont reliés entre eux par des switchboxs.

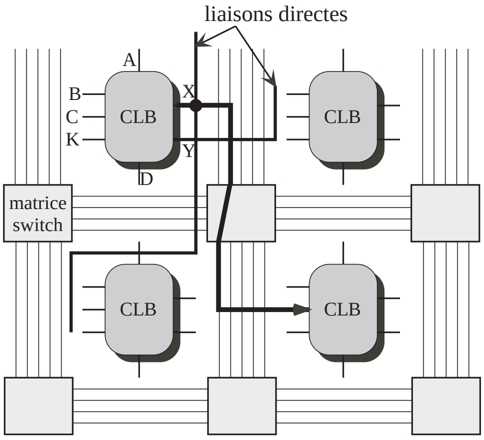


Figure 2.18 : *Le schéma d'interconnexions d'un XC2000.*

---

[5] Nous pouvons d'ores et déjà noter que l'implémentation du tissu POEtic utilisera la même base.

Afin de pouvoir implémenter des designs encore plus efficacement, des liaisons directes sont également disponibles (Figure 2.18). Elles permettent de connecter un CLB à ses quatre voisins, sans accéder aux liaisons des switchboxs, et réduisent dès lors le temps de propagation du signal, ainsi que les problèmes de congestion du routage.

La figure 2.19 montre les différents bits de configuration liés à l'interconnexion des blocs. Chaque carré correspond à un élément de mémoire SRAM, relié à un transistor qui connecte ou non les lignes verticales et horizontales. Nous pouvons noter que chacune des deux sorties X et Y est connectée à un nombre réduit de lignes. Si le routeur rencontre des difficultés à acheminer un signal, il peut simplement interchanger les deux sorties, pour avoir de nouvelles possibilités de routage. De même pour les entrées, qui ne sont pas connectées aux mêmes lignes, le routeur[6] peut les interchanger, en modifiant toutefois le contenu de la look-up table de manière à garder la même fonctionnalité.

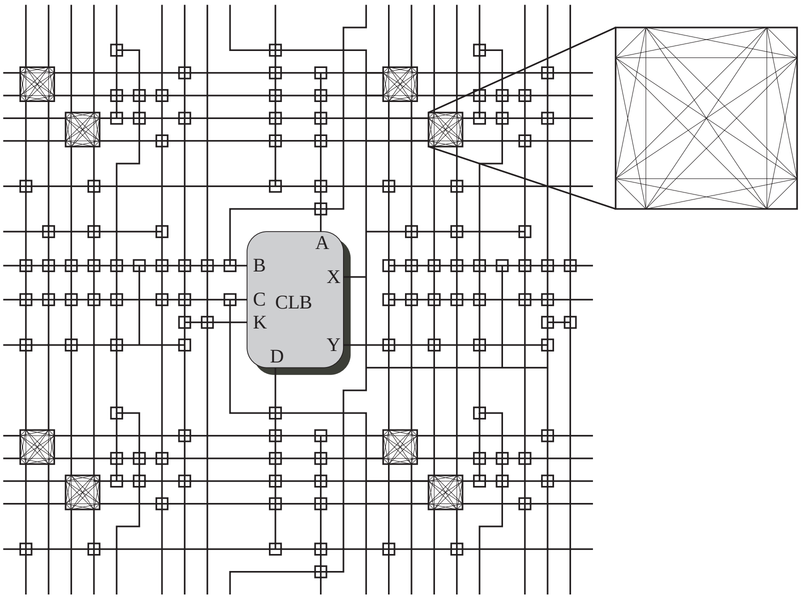


Figure 2.19 : *Les connexions d'un bloc d'un XC2000.*

Pour terminer, il est important de constater que la configuration du FPGA dans un état correct est crucial. En effet, il est très facile de créer des courts-circuits en connectant plusieurs sorties de CLBs ensemble, ce qui peut être très dommageable pour le circuit.

2.6.2 XC6200

La famille XC6200 de Xilinx [258] fut très importante pour l'ensemble de la communauté intéressée par le matériel évolutif. En effet, son architecture est encore plus simple que celle des XC2000, autant en ce qui concerne les blocs logiques que le réseau de routage. De plus, le point central de son attrait est le fait qu'elle est basée sur une technologie SRAM couplée à des multiplexeurs. Tout court-circuit est alors impossible, rendant aisée l'évolution de systèmes au niveau des portes, concept sur lequel nous reviendrons dans la section 3.3.1.

---

[6] Le concept de routeur est décrit à la section 2.6.7, page 32.

Son bloc de base, présenté par la figure 2.20, ne possède que trois entrées, et sa fonctionnalité n'est pas réalisée par une look-up table, mais par des multiplexeurs, ce qui, dans cette configuration, ne permet pas d'implémenter n'importe quelle fonction à trois variables. Une bascule est placée de la même manière que dans un XC2000, et l'unique sortie peut donc être combinatoire ou séquentielle.

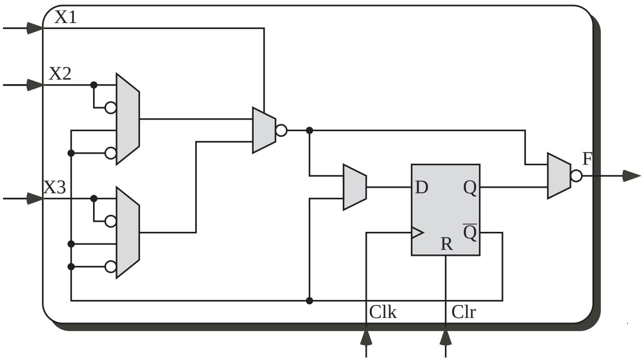


Figure 2.20 : *L'unité fonctionnelle d'un XC6200.*

Concernant les interconnexions, la figure 2.21 détaille la manière dont les entrées et la sortie de l'unité fonctionnelle sont connectées aux voisines. Nous pouvons observer que les valeurs des quatre voisines immédiates peuvent directement être récupérées par l'unité fonctionnelle. Outre ces liaisons courte-distance, il existe des lignes d'une longueur de quatre cellules, seize cellules, ou parcourant le circuit entier. Toutefois, leur nombre est très réduit en comparaison de tous les autres types de FPGAs. Un circuit de la famille XC6200 est donc efficace pour des systèmes ne demandant que peu de ressources de routage de distance plus importante que le simple voisinage.

2.6.3 Architecture MUX versus LUT

Deux types d'architectures sont utilisées par les fabricants, certains basant les éléments logiques sur des multiplexeurs (MUX), à la manière de la XC6200, et d'autres sur des look-up tables (LUT), comme dans la XC2000.

Dans l'approche MUX, la fonctionnalité est réalisée en connectant les entrées et le signal de sélection des multiplexeurs. De la logique peut également être ajoutée, les multiplexeurs sélectionnant alors une parmi plusieurs fonctions.

L'implémentation des éléments logiques grâce à une ou des LUTs permet quant à elle de réaliser n'importe quelle fonction, en programmant correctement les bits de configuration de la LUT. Cette approche a été choisie par Altera et Xilinx, les deux plus gros fabricants. Cette alternative permet d'implémenter des registres à décalage et des mémoires, en tirant parti des bits de configuration des LUTs, ce que les multiplexeurs ne peuvent faire. Son désavantage est toutefois de n'être que peu efficace pour les applications réalisées par un grand nombre de fonctions logiques simples et indépendantes. En effet, une fonction à 2 entrées nécessite la réquisition d'une LUT entière, alors qu'elle pourrait n'être implémentée que sur un multiplexeur. Concernant la taille optimale d'une LUT, les recherches ont montré que l'optimum se situait à 4 entrées [23, 96, 196, 216].

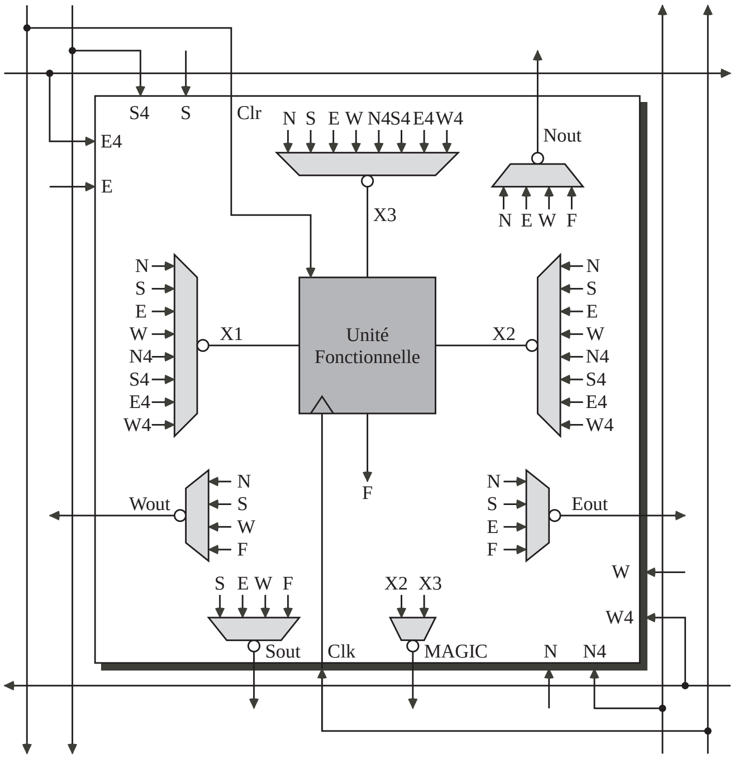


Figure 2.21 : *La cellule de base du XC6200.*

2.6.4 Technologies de programmation

Nous avons déjà présenté en détail les différentes technologies de programmation. Les FPGAs actuels ne tirent parti que des technologies anti-fusibles, EEPROM/FLASH et SRAM, le tableau 2.2, tiré de [146], résumant les caractéristiques de ces approches.

Il est intéressant de noter que les cellules anti-fusibles sont nettement plus petites que les EEPROM, elles-mêmes plus petites que les SRAM. On pourrait donc préférer utiliser des EEPROMs plutôt que des SRAM, ces premières permettant de placer plus de composants sur le même circuit. Cependant les deux plus petites technologies nécessitent l'ajout de plusieurs processus au-dessus de la technologie CMOS standard, et ont donc toujours une ou plusieurs générations de technologie de retard.

Finalement, une nouvelle technologie est peut-être en train de voir le jour. Le 12 juin 2002, Fujitsu annonçait la réalisation d'un nouveau FPGA utilisant une technologie RAM ferroélectrique (FRAM) [180]. L'avantage de cette nouvelle approche réside dans le fait que la mémoire ferroélectrique est non-volatile, et que donc le circuit est directement opérationnel au démarrage. De plus le temps de configuration du FPGA est vingt fois plus rapide qu'une technologie EEPROM ou FLASH, et ne nécessite qu'un courant de 3,3V. L'avenir nous dira si cette technologie saura s'imposer face à celles existantes.

Caractéristique	**SRAM**	**Antifusible**	**EEPROM/FLASH**
Technologie	état de l'art	une ou deux générations de retard	une ou deux générations de retard
Reprogrammable	Oui	Non	Oui
Vitesse de reprogrammation	Rapide	-	3x plus lent que SRAM
Volatile	Oui	Non	Non
Prototypage	Excellent	Non	Bon
Sécurité IP	Non	Oui	Oui
Taille des cellules	Grande (6 transistors)	Petit	Moyen (2 transistors)
Consommation	Moyenne	Basse	Moyenne
Sensible aux radiations	Oui	Non	Plutôt oui

Tableau 2.2 : *Comparaison des caractéristiques des différentes technologies de programmation appliquées aux FPGAs.*

2.6.5 Accroissement de complexité

Les deux FPGAs présentés font bien pâle figure à côté des circuits actuels. En moins de vingt ans d'existence, les circuits se sont complexifiés de manière impressionnante. A l'heure actuelle, la plupart des FPGAs comprennent, outre les blocs de logique, des blocs de mémoire RAM configurables. Dans les Virtex II Pro, un ou deux processeurs sont ajoutés, et dans les Stratix II d'Altera, ce sont des blocs multiplicateurs qui le sont.

Blocs de mémoire

Des applications toujours plus complexes en terme de ressources nécessaires sont implémentées dans les FPGAs actuels, et pour une partie d'entre elles, l'utilisation de mémoires RAM, ROM, CAM, ou FIFO est nécessaire. Les FPGAs fournissent donc la possibilité d'implémenter efficacement de tels éléments, et ceci de deux manières, en utilisant les look-up tables des éléments logiques, ou grâce à des blocs de mémoire spécialement conçus et placés dans le circuit.

Les deux plus grands fabricants de FPGAs, Xilinx et Altera, ont des architectures basées sur des LUTs. Outre la fonction standard consistant à fournir une sortie en fonction des entrées, ces LUTs peuvent servir à la réalisation de registres à décalage ou de blocs de mémoires RAM. En effet, 2^k bits de configuration sont nécessaires à l'implémentation d'une LUT à k entrées, et ces bits peuvent être utilisées à ces deux autres fins. La configuration des FPGAs se faisant en général de manière sériel, la LUT est vue comme un registre à décalage qui peut être exploité par une application. La transformer en bloc de mémoire est un peu plus compliqué, mais permet d'exploiter au maximum les cellules RAMs de configuration (dans le cas de Xilinx et Altera). Typiquement, une LUT à 4 entrées peut être vu comme une mémoire 16x1 bit et de larges mémoires peuvent être réalisées en combinant plusieurs LUTs.

Cette manière d'implémenter des mémoires n'est toutefois pas excellente en terme d'utilisation des ressources pour des mémoires de taille non négligeables. C'est pourquoi la plupart des FPGAs actuels contiennent des blocs de mémoire RAM paramétrables en ce qui concerne la profondeur et la largeur. Dans la série APEX d'Altera par exemple, ces blocs contiennent 2048 bits pouvant réaliser des mémoires de taille 2048×1, 1024×2, 512×4, 256×8, 128×16 ou 64×32. Les blocs sont soit répartis autour des éléments de logique, soit en colonnes, de manière à pouvoir minimiser la longueur des chemins de données.

Reconfiguration partielle

Pour la grande majorité des FPGAs, la configuration ne peut se faire qu'entièrement. Seuls les Virtex de Xilinx et les FPSLIC d'Atmel offrent des possibilités de reconfiguration partielle [154], où certaines parties du circuit peuvent être modifiées sans influer sur le fonctionnement du reste du circuit. La famille Virtex proposant des FPGAs imposants (jusqu'à 8 million de portes équivalentes), il est possible d'y placer de gros systèmes composés de plusieurs modules distincts. Dès lors, un nouvel axe de recherche a vu le jour concernant des systèmes d'ordonnancement de tâches matérielles [225] afin de pouvoir implémenter une application nécessitant plus de place que celle disponible sur le FPGA. Il s'agit donc d'y construire une application composée de modules n'étant pas forcément utilisés au même moment. Un ordonnancement correct permet de ne charger les modules qu'au moment de leur utilisation, grâce à la reconfiguration partielle. Nous sommes persuadés que le pourcentage de FPGAs possédant cette capacité va croître, les fabricants étant toujours à la recherche de plus de flexibilité. Notons toutefois que la reconfiguration partielle d'un circuit de type Virtex n'est pas sans risque, un court-circuit pouvant facilement être généré, avec la potentielle brûlure du FPGA.

Sur le plan des systèmes bio-inspirés, nous verrons plus loin que la reconfiguration partielle est un avantage décisif. Certaines applications mettant en jeu des systèmes évolutionnistes ne nécessitent, pour leur bon fonctionnement, de ne modifier qu'une partie du circuit. Dès lors un gain de temps est obtenu durant la configuration. De plus, alors que les circuits existants sont partiellement reconfigurés par un contrôleur, il serait utile de posséder un tel FPGA capable de s'auto-reconfigurer et où les cellules logiques auraient la possibilité de reconfigurer partiellement d'autres cellules, sans l'intervention d'un agent externe.

Blocs IP

La quantité de logique pouvant être incorporée à un circuit électronique étant à l'heure actuelle très importante, les fabricants en profitent pour intégrer à leurs FPGAs des blocs fonctionnels tels que des multiplicateurs, des MACs (multiplie et accumule), ou même des processeurs. En effet, beaucoup d'applications, notamment en traitement du signal, nécessitent des opérations particulières qui sont difficiles à implémenter grâce aux éléments logiques des FPGAs. Les multiplicateurs intégrés augmentent donc sensiblement la rapidité d'exécution de nombreuses applications.

Les FPGAs contenant un ou des processeurs permettent, quant à eux de conjuguer la flexibilité des processeurs avec la rapidité du matériel. Xilinx et Altera proposent respectivement la série Virtex II Pro et la série Excalibur, contenant chacune un processeur matériel, pouvant communiquer avec les éléments logiques. De même, ils proposent respectivement le MicroBlaze et le Nios, qui ne sont pas matériels, mais logiciels, dans le sens où il s'agit de modules pouvant être intégrés à un design qui sera ensuite programmé sur le FPGA. Ces blocs IP (Intellectual Property) sont loin d’être les seuls, chaque fabricant proposant divers modules pouvant être utilisés par les développeurs, tels qu’interface PCI, UART, et bien d’autres.

2.6.6 Fabricants

Ayant présenté les caractéristiques globales des FPGAS, nous allons rapidement passer en revue les produits proposés par les différents fabricants, à savoir, dans l’ordre alphabétique, Actel, Altera, Atmel, Lattice Semiconductor, QuickLogic et Xilinx [7].

Actel

Les principaux circuits d’Actel sont basés sur des anti-fusibles, les rendant non-volatiles, rapides, et très sûrs sur le plan de la propriété intellectuelle. De plus, les anti-fusibles sont insérés entre les couches de métal, afin d’économiser la surface de silicium. Toutefois, la taille des ces circuits n’est pas exceptionnelle, la série Axcelerator [1] contenant au plus 10’752 cellules composées d’un registre, 21’504 cellules combinatoires (quelques multiplexeurs et portes logiques), et 294’912 bits de mémoire dédiés, pour un total de 2 millions de portes équivalentes[8].

Afin de proposer une alternative à leurs précédents produits tout en maintenant la non-volatilité, Actel a récemment introduit la série ProASIC Plus [2], basée sur une technologie FLASH. Elle est également sûre, les bits de configuration étant cryptés, et va jusqu’à une densité de 1 million de portes équivalentes.

Altera

A l’heure de la rédaction de ces lignes, Altera, le deuxième plus gros fabricant, se concentre sur deux séries de base à technologie SRAM, Cyclone [7] et Stratix [10]. L’optique de la série Cyclone est de disposer d’un FPGA à bas prix. Bien que les éléments de base (LE, pour Logic Element) des deux séries soient semblables, la version Cyclone a été optimisée, et requiert 30% de moins d’espace. Il est principalement composé d’une LUT à 4 entrées, d’une bascule, et d’un système de propagation de retenue, afin d’accélérer les opérations arithmétiques. Le plus imposant des Cyclones contient 20’060 LEs et 288Kbits de RAM, pour un total de 1 million de portes équivalentes.

La série Stratix est nettement plus imposante, pouvant contenir jusqu’à 79’040 LEs, 7’427Kbits de RAM et 22 blocs DSP (88 multiplicateurs 18×18, 178 de 9×9 ou 22 de 36×36). En ajoutant jusqu’à 20 transcievers sériel à 3.125 Gbps et moyennant une baisse de densité (41’250 LEs et 3’423Kbits de RAM), nous obtenons la série Stratix GX [11]. Enfin, la dernière née des séries est la Stratix II [9]. Elle est deux fois plus dense que la première, notamment grâce à une redéfinition de son élément de base. Le LE a été remplacé par l’ALM (Adaptive Logic Module), qui contient deux bascules, deux reports de retenue, et un bloc combinatoire composé principalement de deux 4-LUTs et quatre 3-LUTs. A titre de comparaison, le plus gros Stratix II correspond à 179’400 LEs, 9Mbits de RAM et 384 multiplicateurs 18×18.

Sur le plan des processeurs, la série Excalibur propose un FPGA de type APEX20KE (avec un maximum de 38’400 LEs, soit 1.7 millions de portes équivalentes), auquel a été ajouté un processeur ARM922T tournant à 200MHz. Il

---

[7] Notez l’attrait du A comme première lettre de nom de Compagnie.

[8] Une porte équivalente correspond à la fonctionnalité d’une porte NAND, soit quatre transistors.

semblerait toutefois qu'il n'ait pas obtenu le succès escompté, et c'est plutôt sur son processeur Nios qu'Altera mise. Il s'agit d'un processeur soft, qui peut être inséré dans n'importe quel design, puis programmé sur le FPGA cible. Sa version 32 bits n'occupe que 1400 LEs, et permet donc d'en placer plusieurs sur un FPGA.

Finalement, Altera propose la solution HardCopy, qui offre à l'utilisateur la possibilité de transformer un design pour FPGA Stratix ou Apex en un ASIC. Basé sur le concept d'ASIC à réseau structuré, le circuit est prédiffusé avec les mêmes composants que ceux du FPGA, et seul l'apposition de deux couches de métal supplémentaires est nécessaire à la réalisation physique du design. L'avantage de cette approche est que la consommation est plus faible et la rapidité plus grande en comparaison de l'implémentation sur FPGA, et que le temps de réalisation est moins élevé que pour un ASIC standard.

Atmel

Atmel, fabricant de semi-conducteurs, propose deux architectures de FPGA, la série AT6000 [15] et la série AT40 [16], toutes deux de technologie SRAM. Le bloc de base de la première est composé de deux multiplexeurs à quatre entrées, et de portes logiques simples, ainsi que d'une bascule, alors que celui de l'AT40, plus conventionnel, est composé de deux 3-LUT, d'une porte ET et d'une bascule. Bien que d'architecture relativement simple, l'originalité de ces deux circuits réside dans le routage. En effet, outre un réseau de routage longue distance simple, chaque cellule est directement reliée à ces 8 voisines, alors que chez la plupart des autres fabricants les liaisons directes ne sont que 4. Cette spécificité permet entre autre d'implémenter efficacement des multiplications de matrice. Concernant la taille, nous pouvons noter que ces circuits sont plutôt petits en comparaison de leur concurrents, la série AT6000 possédant au maximum 75'000 portes équivalentes et la série AT40 1 million, dont 18Kbits de RAM.

Finalement, la série FPSLIC [17] propose un système composé d'un microcontrôleur 8-bit AVR associé à un tableau reconfigurable identique à celui de l'AT40. Ce contrôleur a priori plus simple que l'ARM d'Altera ou le PowerPC de Xilinx, a toutefois la capacité de reprogrammer le FPGA dynamiquement, avantage certain quant à de potentiels systèmes adaptatifs. Notons également que la série Secure FPSLIC offre une EEPROM de 1Mbits, permettant une programmation immédiate au démarrage, sans la nécessité d'une mémoire externe stockant la configuration du FPGA.

Lattice

Les circuits Lattice peuvent être regroupés en 3 familles de FPGA, ORCA, ispXPGA, ECP, et une famille FPSC. La première, ORCA, consiste en trois séries, à savoir les séries 2, 3 et 4. La série 2 est composée d'éléments de base appelés PFU (Programmable Functional Unit) contenant quatre 4-LUT et quatre bascules. Ces PFUs peuvent servir à implémenter des blocs de mémoire RAM et ROM, synchrone, asynchrone ou dual-port, et offrent un maximum de 99'400 portes équivalentes. Les PFUs de la série 3 et des suivantes ont une taille doublée par rapport à la série 2. La série 3 possède également une interface processeur facilitant la configuration et propose des circuits allant jusqu'à 340K portes équivalentes. Enfin, la série 4 [45] offre encore plus de complexité avec des blocs de mémoire additionnels pour un total maximal de 148Kbits, pouvant être utilisés comme RAM, ROM, FIFO, CAM ou multiplicateur, et un maximum de 899K portes équivalentes. Une interface processeur y est également proposée, qui sera exploitée dans les circuits ECP.

Alors que la famille ORCA fut récupérée lors de l'achat de Lucent Technologies, la famille ispXPGA [46] fut entièrement développée par Lattice Semiconductors. Son élément de base est composé de quatre 4-LUT et huit bascules, pour l'implémentation efficace de pipelines. Chaque PFU peut y être utilisé pour la réalisation de 6-LUT, d'une fonction logique jusqu'à 20 entrées, d'un multiplexeur à huit entrées, d'un bloc de 61 bits de RAM, ou d'un registre à décalage de 8 bits. Jusqu'à 246Kbits de RAM sont en outre disponibles, et le nombre maximal de portes équivalentes proposé est de 1'250K. La grande particularité de cette famille concerne sa programmation basée sur des cellules de mémoire E^2CMOS, offrant la non-volatilité à ces circuits. De ce fait aucune mémoire externe n'est nécessaire, et le circuit est automatiquement configuré lors du démarrage.

Suivant la série 4, Lattice propose des Field Programmable System Chip (FPSC) composés d'un tableau reconfigurable identique à celui trouvé dans la série ORCA4 ainsi que d'éléments réalisés en technologie ASIC, pouvant être des interfaces 10Gbits/s, ou backplane transceivers.

Finalement, deux familles, LatticeECP [47], pour EConomy Plus, et LatticeECP-DSP, offrent une solution à bas coûts. Leurs éléments de base sont relativement semblables à ceux de la série ORCA4, mais elles offrent plus de modules utiles à l'implémentation d'applications DSP, tels que des multiplicateurs (jusqu'à quarante 18×18), et des blocs de mémoire. La famille ECP-DSP propose en plus jusqu'à 10 blocs DSP permettant chacun l'implémentation de huit multiplicateurs 9 bits ou quatre Multiply-Accumulate sur 9 bits, par exemple. Le plus grand de ces circuits possède 40 multiplicateurs, 5120 PFUs et 645Kbits de RAM.

QuickLogic

Tous les circuits QuickLogic sont de type anti-fusibles (la technologie ViaLink leur permet d'intercaler le fusible entre deux couches de métal, afin de sauver de l'espace sur le silicium), basés sur un même élément de base, avec une légère différence pour la série Eclipse [188]. Cet élément comprend 2 portes ET à 6 entrées, 4 portes ET à 2 entrées, 6 multiplexeurs à 2 entrées, et une bascule. Cette architecture est particulière en comparaison des concurrents. En effet, elle semble moins intuitive que l'approche LUT, mais permet de créer des fonctions à grand nombre de variables d'entrées comme plusieurs fonctions à moins de variables, grâce à ses 5 sorties. Dans la série Eclipse, la principale différence réside en la présence de deux bascules au lieu d'une. Dans tous les circuits, le réseau de routage est composé de lignes et colonnes connectés par des switchboxs.

La série pASIC consiste en un simple tableau d'éléments logiques et du routage correspondant, pour un maximum de 1584 cellules logiques, soit 75'000 portes équivalentes. Augmentée de blocs de RAM, la série QuickRAM propose jusqu'à 25'344 bits, pour un total de 176'608 portes équivalentes. La série QuickPCI, toujours sur la même base, contient quant à elle un contrôleur PCI, alors que la série QuickMIPS, de loin la plus complexe, possède, outre 1152 cellules, 82'944 bits de RAM et 18 ECUs, un processeur MIPS 32 bits, un contrôleur PCI, 2 UARTs, etc. La série Eclipse, quant à elle, pour un maximum de 662'208 portes équivalentes, peut contenir jusqu'à 82'900 bits de RAM et 18 blocs ECU (Embedded Computational Units), chacun étant composé d’un multiplicateur 8×8, d’un additionneur 16 bits et d’un registre.

Xilinx

A l’heure actuelle, Xilinx, le premier fabricant de FPGAs, propose principalement deux familles, Virtex et Spartan, toutes deux de type SRAM, basées sur une architecture LUT. La différence entre les deux familles est minime, et tient principalement du nombre d’éléments proposés ainsi que du type de process utilisé, les Spartan étant positionnées bas coût en comparaison des Virtex. L’élément de base, le CLB (Configurable Logic Block), est composé de deux Slices, eux-mêmes comprenant deux 4-LUT et deux bascules. Les versions Virtex-II, Virtex-4 et Spartan-3 diffèrent toutefois, le CLB y contenant quatre Slices. Les deux familles contiennent des blocs de RAM, pouvant être utilisés en single ou dual port.

Alors que les séries Spartan-3, Virtex-II et Virtex-4 sont presque identiques en terme d’architecture, la série Virtex-II Pro introduit un ou deux processeurs de type PowerPC. Contrairement à l’échec de la solution d’Altera faisant intervenir un ARM, Xilinx a su imposer son produit, et la série Virtex-II Pro se trouve très bien positionnée dans la gamme de ses produits.

Le plus imposant de la série Spartan-3 [259] propose 5M de portes équivalentes, dont 104 multiplicateur de 18×18 bits et 1’872K bits de RAM. En comparaison, les plus gros Virtex sont le XC4VLX200, de la famille Virtex-4LX [260] et le XC4VFX140, un Virtex-FX. Le premier contient 178’176 éléments logiques, un élément logique étant une LUT à 4 entrées et une bascule. A ceci s’ajoutent 6’048 Kbits de RAM configurables, ainsi que 96 multiplicateurs 18x18 bits, pour un total de 15M de portes équivalentes (chiffre calculé). Le deuxième, le XC4VFX140, est composé de 126’336 éléments logiques, de 9’936 KBits de RAM, 192 multiplicateurs, et 2 processeurs de type PowerPC.

Nous pouvons finalement noter que Xilinx, à l’instar d’Altera, propose une solution appelée EasyPath, permettant la réalisation en ASIC d’un design implémenté sur un Virtex-II, Virtex-II Pro ou un Spartan-3.

Résumé

Le tableau 2.3, en page 34, résume les principales caractéristiques des FPGAs disponibles sur le marché. Nous pouvons observer une complexification constante de ces circuits qui se voient ajouter de plus en plus de nouvelles caractéristiques. Il devient en effet quasiment impossible d’en trouver un ne possédant pas de blocs de mémoire RAM, et la présence de multiplicateurs devient presque systématique.

Sur le plan des structures de routage, la tendance est à l’optimisation. Les technologies utilisées devenant de plus en plus petites, le délai des portes se réduit d’autant, alors que celui des fils ne diminue pas. Dès lors une nouvelle attention est donnée à cette partie du circuit, de manière à assurer un fonctionnement correct à haute fréquence. Concernant la complexité du routage, nous pouvons noter que les réseaux de routage des gros FPGAs ne se bornent pas à une simple grille de switchboxs, mais qu’ils font intervenir plusieurs niveaux. La série Startix d’Altera, par exemple, est composés de Logic Array Blocs (LABs) contenant 10 Logic Elements (LEs), chaque LE étant une LUT, une bascule et quelques portes logiques. Le LAB y a une connectique interne, et une connectique externe pour des chemins de longue distance entre les LABs. Altera voit ce réseau comme une structure à plusieurs dimensions, l’ensemble du réseau étant vu comme une grille de sous-réseaux interconnectés, eux-mêmes étant une grille de sous-réseaux, etc.

2.6.7 Placement/Routage

Le réseau de connexions des FPGAs est des plus complexes, de par le nombre d'éléments à connecter (jusqu'à plus de 150'000). Le placement-routage d'un design pour un circuit donné est donc une tâche laborieuse. Un développeur commence par décrire son système, soit dans un langage de description matériel de type VHDL, Verilog ou SystemC, soit grâce à un éditeur de schéma, tel ViewLogic ou HDLDesigner. Un logiciel s'occupe ensuite de la synthèse, transformant la création du développeur en une liste de composants de base et de leurs interconnexions. Un deuxième logiciel, est alors responsable de placer ces composants dans les éléments de base du FPGA, puis de créer le routage nécessaire au transfert des signaux entre les composants. Cette tâche est relativement simple pour de petits designs, mais peut laisser un processeur dernier cri travailler pendant quelques heures lorsque le design occupe la presque entièreté du circuit. En effet, des modules interconnectés doivent être placés les plus proches possibles, afin de limiter les délais sur les fils, et le logiciel doit trouver le moyen de tous les connecter avec l'ensemble des fils de routage existants.

Afin de réduire le temps de calcul effectué par le PC lors du routage, il serait donc intéressant de disposer d'un FPGA capable de router lui-même certains signaux. De plus, de tous les circuits actuellement existants, pas un seul n'a la capacité de modifier son routage pendant son exécution. Cette caractéristique serait pourtant bien utile à la réalisation de systèmes adaptatifs nécessitant une modification de leur fonctionnement en fonction d'un environnement changeant.

2.6.8 FPGA versus ASIC

Terminons notre survol par un bref aperçu des avantages et désavantages de l'approche FPGA en comparaison des ASICs, en commençant par les désavantages :

- Une fréquence d'horloge moins élevée, pour une même application.
- Une plus grande place nécessaire sur le silicium, beaucoup de logique additionnel étant nécessaire au bon fonctionnement de la programmation du FPGA.
- Dans la même lignée, l'ensemble des éléments programmables ne sont jamais entièrement utilisés. Leur nombre dépend de la taille du design, mais ne peut pas atteindre 100%, le routage devenant quasiment impossible lorsque le circuit est presque plein.
- Pour certains, la grande facilité de tests pousse les développeurs à appliquer une méthodologie de design “essayer-et-voir-ce-qui-se-passe”, ce qui n'encourage pas les méthodes formelles de vérification des circuits.

Et enfin les avantages des FPGAs :

- Un faible coût de développement, le prototype ne nécessitant pas de réalisation matériel, mais seulement des tests successifs sur un FPGA.
- Peu de risques, dans le sens où une erreur de design est très vite corrigée et n'implique pas la création d'un nouveau circuit.
- Une grande rapidité lors de la réalisation d'un prototype.
- La possibilité de pouvoir réaliser des applications de matériel évolutif, laissant un algorithme génétique trouver la solution à un problème donné.

2.7 Conclusion

Nous venons de présenter un survol des circuits reprogrammables, prêtant une attention particulière aux FPGAs, qui sont les circuits reconfigurables les plus flexibles et offrant le plus de fonctionnalités. Ils sont les meilleurs candidats pour la réalisation rapide de prototype, de par leur capacité à être reconfigurés un très grand nombre de fois.

Dans le chapitre suivant nous verrons qu'ils ont été largement utilisés par la communauté bio-inspirée dans la réalisation de réseaux de neurones artificiels notamment, ou dans le développement de matériel évolutif. En effet, certaines applications possédant un parallélisme inhérent peuvent tirer profit d'une implémentation matérielle efficace.

Nous avons également vu que leur configuration est lancée par un agent externe, typiquement un petit contrôleur accédant une mémoire qui contient les bits de configuration du FPGA. De plus le placement et le routage des éléments est calculé par un ordinateur, et ne peut être modifié durant l'exécution d'une application que sur peu de circuits (Virtex et FPSLIC), et de façon peu aisée, avec le risque de détruire le circuit.

Dès lors, nous allons proposer, dans le chapitre 6 un nouveau FPGA possédant des capacités d'auto-reconfiguration et d'auto-routage. Les cellules de base y auront le pouvoir de reconfigurer partiellement d'autres parties du circuit ainsi que de créer des chemins de données durant le fonctionnement du circuit.

Vendeur	**Famille**	**Technologie**	**Portes Equivalentes**	**Nb Flip-flops**	**Mémoire Bloc (Kbits)**	**Multiplicateur 18×18**	**Processeur Processeur**
Actel	Axcelerator	anti-fusible	2M	21’504	294	-	-
Actel	ProASIC Plus	FLASH	1M	56’320	198	-	-
Altera	Cyclone	SRAM	1M	20’060	295	-	-
Altera	Stratix	SRAM	4M (calcul)	79’040	7’427	176	-
Altera	Stratix II	SRAM	9M (calcul)	143’520	9’383	384	-
Altera	Excalibur	SRAM	1.7M	38’400	256	-	1 ARM922T
Atmel	AT40	SRAM	50K (Usable gates)	3’048	18.4	-	-
Atmel	AT6000	SRAM	30K (Usable gates)	6’400	-	-	-
Atmel	FPSLIC	SRAM	50K (déduit)	2’962	18.4	-	8-bit AVR
Lattice	ORCA4	SRAM	899K	18’216	148	-	-
Lattice	ispXPGA	E^2CMOS	1.25M	30’700	414	-	-
Lattice	ECP	SRAM	1M (calcul)	46’080	645	40	-
QuickLogic	Eclipse II	anti-fusible	320K	4’002	55	-	-
QuickLogic	pASIC	anti-fusible	75K	2’692	-	-	-
QuickLogic	QuickRAM	anti-fusible	176K (90K Usable PLD gates)	2’692	25	-	-
QuickLogic	QuickMIPS	anti-fusible	457K (115K Usable PLD gates)	4’032	83	18	MIPS32 4Kc
Xilinx	Virtex-II	SRAM	8M	93’184	3’024	168	-
Xilinx	Virtex-II Pro	SRAM	8M	88’192	7’992	444	2 PowerPC
Xilinx	Virtex-4LX	SRAM	15M (calcul)	178’176	6’048	96	-
Xilinx	Virtex-4FX	SRAM	11M (calcul)	126’336	9’936	192	2 PowerPC
Xilinx	Spartan-3	SRAM	5M	66’560	1’971	104	-

Tableau 2.3 : *Comparaison des caractéristiques des différentes FPGAs.*

Chapitre 3

De la bio-inspiration

> On est obligé à présent de regarder l'imposant spectacle de l'évolution de la vie comme un ensemble d'événements extraordinairement improbables, impossibles à prédire et tout à fait non reproductibles.
>
> Stephen Jay GOULD , *La vie est belle*

LA NATURE n'est-elle pas parfaite ? On pourrait le croire en observant l'incroyable diversité des espèces vivantes et leurs formidables facultés d'adaptation. Chaque animal est spécialisé, entre autre en fonction de son environnement et de son type de nourriture. Les dents du lion lui permettent de facilement déchiqueter ses proies, le vautour a une vue perçante, une sorte de loupe étant intégrée au centre de son œil, et les termites sont passées maîtres dans la construction d'édifices.

Pourtant tout ceci ne s'est pas fait en un jour (ni même en sept). L'évolution a pris le temps de générer l'éventail des espèces vivantes, en faisant disparaître les moins adaptées, tout au long d'un long processus de bricolage [110]. De plus, la faculté d'adaptation permettant au vivant de s'adapter à un environnement changeant est une des caractéristiques essentielles à la survie, aussi bien d'un individu que d'une espèce. Cette adaptation, au niveau de l'individu est caractérisée par une capacité d'apprentissage. Une modification du comportement est souvent nécessaire afin de faire face aux aléas de la vie, une non adaptation étant alors synonyme de mort certaine. Au niveau d'une espèce, l'adaptation se fait au fil des générations, au prix de nombreux essais. Par sélection naturelle, les plus aptes à se reproduire ont la chance de transmettre leurs gènes à leur descendance, et ainsi l'espèce reste adaptée à son environnement.

Aux vues de ces deux paragraphes, pouvons-nous toujours dire que la nature est parfaite ? Non, si l'on considère le nombre de morts nécessaires à l'obtention d'une solution adéquate. Mais l'on peut répondre positivement si l'on considère la vie dans son ensemble. En effet, depuis son apparition elle n'a jamais disparu. Malgré de multiples extinctions, des espèces ont toujours trouvé le moyen de s'adapter pour survivre. En comparaison, les machines créées par l'homme semblent parfaites car créées de manière exacte pour un problème particulier. Toutefois, l'adaptation n'y est pas, et une machine n’est pas capable d’effectuer une autre tâche que celle pour laquelle elle a été programmée. N’est-ce donc pas l’adaptation qui fait le succès du vivant ? Et n’est-ce donc pas cela qui manque aux systèmes artificiels ? De plus, outre ces deux niveaux d’adaptation, à l’échelle de l’individu pour l’apprentissage, et de l’espèce pour l’évolution, l’autoréparation est un autre grand atout du vivant. Alors qu’en général, une petite erreur met à mal l’entièreté d’un système artificiel (le mauvais fonctionnement d’une bascule d’un processeur compromet son utilisation), les êtres vivants sont capables de les gérer, en cicatrisant ou en adaptant l’utilisation d’autres parties du corps.

Au cours de ce chapitre, nous allons présenter différents aspects du vivant ayant inspiré les ingénieurs. Nous verrons qu’il s’agit toujours de systèmes adaptatifs, autant dans le sens d’évolution et d’apprentissage que dans le sens d’autoréparation. Après les organismes vivants, nous introduisons leurs pendants artificiels, en mettant l’accent sur les systèmes matériels. Finalement, nous terminons le chapitre par introduire l’architecture des cellules POEtic, qui seront implémentées sur le circuit que nous avons réalisé.

3.1 Le modèle POE

Comme présenté dans l’introduction, les systèmes matériels bio-inspirés peuvent être analysés selon trois axes organisationnels : phylogénétique (P), ontogénétique (O), et épigénétique (E) [204, 217, 218], qui peuvent être combinés pour obtenir des mécanismes dits POE. Sur le plan naturel, le premier traite de l’évolution des espèces au cours du temps, le deuxième du développement d’un individu à partir d’une seule cellule, et le troisième de l’apprentissage tirant profit des expériences vécues durant la vie. Sur le plan artificiel, le premier concerne les algorithmes évolutionnistes, le deuxième les systèmes de croissance et d’autoréparation, et le troisième les réseaux de neurones et les systèmes d’apprentissage.

Dans le cas des systèmes artificiels, bien que la majorité des applications courantes soient exécutées par des processeurs présents dans des ordinateurs, les implémentations matérielles ont plusieurs avantages :

- L’exécution matérielle, pour des systèmes cellulaires faisant intervenir du parallélisme, est nettement plus rapide que la logicielle.
- La tolérance aux pannes peut tirer profit d’une implémentation matérielle. En effet, dans un processeur, une faute dans un seul composant est fatale au fonctionnement du système, alors qu’un système cellulaire peut être conçu de manière à ce que des cellules en attente reprennent la tâche de cellules détruites.
- Les nanotechnologies nous promettent la faisabilité de systèmes composés d’un très grand nombre d’éléments identiques qu’il faudra être capables d’organiser. Dès lors, les implémentations matérielles actuelles permettent de mettre au point les applications futures des nanotechnologies, qui devront faire preuve de grandes facultés de tolérance aux fautes.

Le modèle POE se propose d’analyser les systèmes matériels bio-inspirés selon les trois axes susmentionnés (Figure 3.1). Alors qu’un grand nombre d’applications ne mettant en jeu que l’un de ces trois axes ont déjà vu le jour, nettement moins de systèmes combinant deux axes ont été réalisés, et aucun ne l’a été en combinaison des trois.

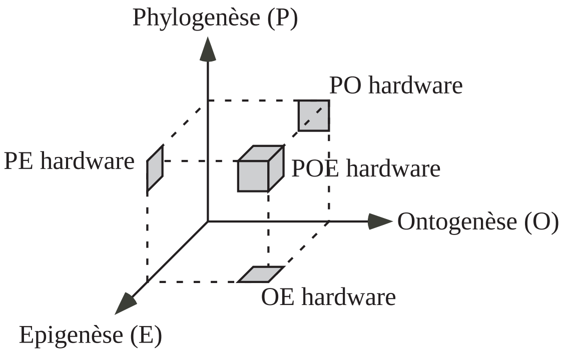


Figure 3.1 : *La représentation tridimensionnelle du modèle POE.*

3.2 La vie en 3 axes

Nous allons survoler les trois axes du vivant en présentant les principes de base de chacun, qui ont et vont inspirer les chercheurs. Notons au passage que ces trois axes représentent trois niveaux organisationnels, qui peuvent être vus comme des arbres : l'arbre phylogénétique, qui recense les liens de parenté entre les espèces vivantes, l'arbre ontogénétique[1] , qui voit une cellule se diviser en deux, puis quatre, dans un processus de complexification amenant à l'organisme entier, et l'arbre épigénétique, qui correspond à la création des neurones et de leurs interconnexions.

3.2.1 Phylogenèse

Lors d'une fécondation, chaque parent transmet à sa progéniture la moitié de ses gènes. De cette manière, l'enfant a tendance à ressembler à sa mère et à son père, puisqu'il se développe sur ce substrat génétique qu'il partage en partie avec eux. Ce génome définit en effet une sorte de programme de formation de l'être à partir de la première cellule de base (cf. section 3.2.2). Les espèces évoluent donc, de par le mélange incessant de gènes au sein d'une population d'individus génétiquement compatibles. De plus, un individu, pour pouvoir se reproduire, doit être suffisamment adapté à son environnement pour ne pas mourir avant d'avoir atteint sa maturité sexuelle. En conséquence, les parents sont des êtres aptes à la survie, qui auront tendance à donner naissance à une progéniture également capable de s'intégrer dans son environnement. Ce processus, appelé sélection naturelle, force une population à ne toujours être composée que d'individus bien adaptés.

Nous désirons ici nous étendre quelque peu sur les différentes théories de l'évolution, qui sont trop souvent ignorées au profit de la théorie de Darwin. Ces quelques considérations biologiques n'étant toutefois pas essentielles à la compréhension de la présente thèse, le lecteur peu enclin aux théories sur l'évolution des espèces peut sans sourciller passer à la section suivante.

---

[1] L'arbre ontogénétique du vers Caenorabditis Elegans [191], par exemple, est connu, et définit exactement la création des 959 cellules qui le composent.

Créationnisme

Dieu créa la Terre et tous ses habitants en six jours ! C'est en tout cas ce que laisse entendre la Bible. Tous les individus d'une espèce vivante seraient donc issus d'un seul couple originel, tel Adam et Eve pour les humains, ou plus généralement, créés par un ou plusieurs Dieux. Dans cette théorie qui n'en est pas vraiment une car non réfutable, l'évolution n'existe pas, seule la disparition d'une espèce étant possible. Bien que peu probable en tant qu'explication du vivant, nombre d'entre-nous considèrent le créationnisme comme étant la seule et unique vérité. Nous n'y accorderons toutefois que peu d'attention, le fixisme des espèces ne laissant aucune place à une quelconque adaptation, et sa non réfutabilité n'étant pas acceptée des scientifiques.

Fixisme

Vers la fin des années 1700, Georges Cuvier [35], un Français brillant par son esprit, élabore le fixisme. Entièrement en accord avec le créationnisme, il le nuance toutefois en introduisant le fait qu'il n'y aurait pas eu qu'une seule création, mais plusieurs. Suite à des catastrophes anéantissant toute vie sur Terre, le Créateur aurait créé de nouvelles espèces, dont celles issues de la dernière création auraient subsisté jusqu'à nos jours. Là encore, toute trace d'évolution est écartée, permettant à Cuvier de figurer dans les bons papiers de l'Eglise et des "bien-pensants". Cette approche n'est toutefois pas une théorie, de par les même arguments que pour le créationnisme.

Transformisme

Ce n'est qu'au début du 19$^{\text{ème}}$ siècle que Jean-Baptiste Pierre Antoine de Monet, chevalier de Lamarck, propose une nouvelle théorie. En 1800, il prononcera les mots suivants lors de d'un discours inaugural au Museum National d'Histoire Naturelle :

> "Je pourrais prouver que ce n'est point la forme du corps, soit de ses parties, qui donne lieu aux habitudes, à la manière de vivre des animaux ; mais que ce sont au contraire les habitudes, la manière de vivre et toutes les circonstances influentes qui ont avec le temps constitué la forme des animaux." [104]

Ce discours présente ce qui deviendra le transformisme, à savoir la première théorie évolutionniste. Dans sa *Philosophie Zoologique* [134] de 1809, Lamarck introduit le concept d'évolution avec hérédité des caractères acquis. Dans cette théorie, un individu ayant appris certains comportements ou dont le corps se serait modifié pourrait transmettre ces changements à sa progéniture. L'exemple le plus fréquemment cité est le cas de la girafe, dont le cou se serait allongé au fil des générations. Le cou d'un individu s'allongerait durant sa vie, afin d'attraper le plus de feuilles possibles, et cette modification physiologique serait transmise à sa descendance.

Malheureusement pour Lamarck, aucune de ses expériences ne put prouver sa théorie. Il avait introduit le concept d'évolution, mais il fallut attendre un demi-siècle pour voir le darwinisme arriver.

Notons que bien que le lamarckisme ait été abandonné depuis de nombreuses années, de récents travaux tels ceux de l'équipe de Lars Olov Bygren [112] tendent à montrer qu'il existerait une hérédité liée à des expériences vécues. De même, Yves Coppens, un éminent spécialiste français de l'évolution, cité dans Science et Vie, imagine

> “dans les caryotypes mêmes [les chromosomes, Ndlr] un mécanisme subtil qui serait capable de recevoir de l’information du milieu qui change et de s’en servir, en toute connaissance de cause, pour provoquer, dans la bonne direction, lesdites mutations.” [132]

Cette nouvelle orientation des recherches n’en est qu’à ses débuts, et d’autres expériences sont nécessaires avant d’annoncer à grand fracas le retour d’une certaine forme de Lamarckisme. Sur le plan des systèmes artificiels, le concept de Lamarckisme à été étudié par plusieurs équipes de recherche, dont certains des travaux sont résumés dans [82], [90], et [205].

Darwinisme

Revenons donc au 19ème siècle. La théorie de la sélection naturelle, ou darwinisme, est attribuée à Charles Darwin, qui la présente de manière fort complète dans son *Origine des Espèces par voie de sélection naturelle* de 1859 [52]. Toutefois, il nous semble important de souligner le fait que Alfred Russel Wallace fut co-fondateur (oublié) de la théorie. En 1858, alors que Darwin peaufine son livre, il reçoit une lettre de Wallace dans laquelle sont présentés tous les concepts sur lesquels il travaille. De peur de se faire voler la vedette, Darwin décide, sur conseils de Charles Lyell et Joseph Hooker, alors que Wallace était à l’autre bout du monde, de présenter leur vision de l’évolution des espèces. La présentation est faite devant la Linnean Society puis publiée dans leurs proceedings [54], en prenant bien garde d’exposer en premier les travaux de Darwin. Et c’est ainsi que naquit la théorie de la sélection naturelle, selon laquelle les individus les plus féconds engendrent plus de descendants, et donc imposent leurs caractères au fil des générations. L’évolution des espèces serait donc poussée dans la direction des plus aptes à la survie et à la procréation, les caractères biologiques la favorisant étant transmis à la progéniture, et ce grâce à la fécondité différentielle.

Notons que Darwin, à l’instar de Lamarck, croit à une transmission de caractères acquis, la pangenèse. Des gemmules serviraient au corps pour l’envoi d’informations aux parties génitales, et donc les informations transmises à la progéniture pourraient être affectées par les expériences vécues par l’individu [53].

Néo-darwinisme

Le néo-darwinisme voit sa source dans les travaux de Ernst Haeckel, en fin de 19ème siècle, avant la redécouverte des lois de Mendel. Le terme néo-darwinisme est introduit en 1896 par George Romanes, pour décrire la pensée d’August Weismann, qui est un fervent partisan de cette nouvelle théorie selon laquelle la sélection naturelle est toujours d’actualité, mais qui réfute la transmission des caractères acquis qu’avaient suggéré Lamarck et Darwin. Dans cette lignée, Gustav Fischer, J.B.S. Haldane et Sewall Wright proposent ensuite des modèles mathématiques de la sélection naturelle.

Théorie synthétique

En 1866, Gregor Mendel publie ses résultats [152], présentant les premiers travaux en génétique expliquant la transmission des caractères héréditaires. Ce n’est toutefois qu’à partir de 1900 que naît la génétique, après que ses travaux aient été reconnus.

Basés sur cette nouvelle science, Theodosius Dobzhansky [60], Ernst Mayr [147, 148], Georges Gaylord Simpson et Julian Huxley vont, dans les années 1940, lancer les bases de la théorie synthétique, en alliant la génétique et la paléontologie à la théorie néo-darwinienne. Le concept de sélection naturelle reste équivalent, mais la transmission du patrimoine génétique explique ici la transmission des caractères. Les chromosomes des deux parents sont mélangés par un processus appelé crossing-over, que nous expliciterons à la page 47. De plus, des mutations peuvent intervenir, en modifiant le génome d'un individu de manière aléatoire. Cette théorie explique donc la microévolution (au niveau des individus). Bien qu'adoptée par une grande majorité, elle pose quelques problèmes sur le plan de la macroévolution (au niveau des espèces). En effet, la théorie synthétique est une théorie gradualiste, où l'évolution d'une espèce vers une autre se ferait lentement, par micro changements successifs, et les paléontologues peinent à trouver des fossiles entre certaines espèces. Il existe trop de chaînons manquants que cette théorie ne peut expliquer.

Monstres prometteurs

Deux théories saltationnistes, celle des monstres prometteurs et celle des équilibres ponctués, vont à l'encontre du gradualisme. La première, due à Richard Goldschmidt, fut publiée en 1940 [78]. Goldschmidt y distingue les petites mutations de la microévolution gardant une continuité dans l'espèce, des grandes mutations de la macroévolution. Ces dernières génèreraient des monstres prometteurs, relativement distants de leurs congénères sur le plan génétique. Ces nouveaux individus seraient alors à la base d'une nouvelle espèce, sans qu'un changement graduel ne puisse être observé.

Cette théorie eut beaucoup de mal à se défendre face aux acteurs de l'approche néo-darwinienne. Ce n'est qu'à partir de 1970 que les progrès en génétique permirent de montrer que la modification d'un seul gène pouvait avoir d'énormes conséquences sur la morphologie d'un individu. Paolo Sordino, Frank van der Hoeven et Denis Duboule [61, 222], ont entre autre montré que la modification de seulement deux gènes organisateurs a pu transformer une nageoire en une patte lors du passage des poissons à la vie terrestre. Les monstres prometteurs ont donc pu jouer un rôle non négligeable dans l'évolution des espèces.

Équilibres ponctués

Dans la même optique non gradualiste, Niles Eldredge et Stephen Jay Gould publient en 1972 leurs idées sur la question [67]. Leur théorie des équilibres ponctués, habituellement attribuée à Gould, consiste en une évolution active sur de très courtes périodes, le reste du temps ne voyant que peu de variations au sein d'une même espèce. Ces courtes périodes peuvent être dues à un élément tel qu'une catastrophe naturelle, ou un brusque changement de l'environnement. Durant cette transition, soit un grand nombre d'individus ne survivraient pas, et une nouvelle population serait créée à partir d'un sous-ensemble de la population initiale, soit la population serait séparée en sous-populations par un facteur quelconque.

Cette théorie, la plus communément admise par la communauté paléontologique, explique entre autre le manque de fossiles entre deux espèce parentes. La période d'évolution étant très courte, il est effectivement très peu probable de tomber sur un fossile correspondant.

Neutralisme

Quelques temps avant la théorie de Gould, Mooto Kimura publie la *Théorie neutraliste de l'évolution* [121], jetant un pavé dans la mare de la théorie synthétique. Suite à de nombreuses expériences reprenant certaines des idées de Malécot, Wahlund et Wright, il conclut que la sélection naturelle n'a que peu d'influence quant à l'évolution moléculaire. Cette évolution ne pousserait pas dans la direction d'une meilleure adaptation, mais la plupart des mutations seraient neutres au regard de la sélection naturelle, et l'évolution correspondrait à une dérive génétique aléatoire.

Cette approche est intéressante dans le sens où ce ne sont que le hasard et les probabilités qui guident l'évolution, en créant un grand nombre de possibles, la sélection n'intervenant que dans l'élimination des inaptes.

Néo-mutationnisme

Un des défauts de la sélection naturelle est le manque de considération des contraintes structurelles, physiques, ou liées au fonctionnement de la cellule. Ces contraintes peuvent dans certains cas forcer l'évolution dans une certaine direction, sans apporter d'avantage sélectif. Un des exemples de cette théorie est la cavité cylindrique centrale présente au milieu des coquilles d'escargots, qui, selon Stephen Jay Gould [81], serait due à des modalités de développement des mollusques plus qu'à une sélection. Le néo-mutationnisme prend donc en compte les contraintes structurelles. De plus, elle réactualise la théorie des monstres prometteurs en mettant l'accent sur les mutations plus que sur la sélection, remettant le hasard au goût du jour.

Transitionisme

Finalement, une dernière théorie se développe à partir des travaux en génétique moléculaire du développement. Selon cette théorie, développée entre autres par l'équipe de Duboule [62], les mécanismes d'évolution ont eux-mêmes évolué au cours du temps, passant du gradualisme au ponctualisme. La démonstration se base sur le fonctionnement des gènes en observant que ceux-ci n'ont que peu varié tout au long de l'histoire du vivant. Nous retrouvons en effet les mêmes groupements dans les organismes ancestraux que chez l'être humain. La création d'organismes complexes n'a donc pu se faire que par l'introduction de la multifonctionnalité des gènes, ces derniers ne servant plus à définir un seul paramètre du développement, mais étant actifs dans plusieurs processus. Dès lors, les réseaux de régulation des gènes sont devenus de plus en plus complexes, et la modification d'un gène n'influant plus sur une seule partie du développement risque fort de compromettre d'autres parties, rendant ainsi l'embryon inapte à la survie. Ceci implique donc une grande résistance des gènes à la variation, peu de celles-ci étant permises. De ce fait, l'évolution d'une espèce vers une autre, qui dans les temps anciens pouvait se faire de manière gradualiste, ne dispose que de peu de possibilités, et ces possibilités représentent de grands sauts évolutifs de par le fait qu'une modification influe sur beaucoup de mécanismes.

Remarques

Nous venons de parcourir les différents courants de pensée concernant l'évolution des espèces, et pouvons remarquer la diversité des approches analytiques. De notre point de vue, un des faits importants négligé par la théorie synthétique est la distinction entre la micro et la macro évolution. Les algorithmes évolutionnistes, qui reproduisent les principes de l'évolution en se déclarant enfants du darwinisme, ne font également pas la distinction entre les deux échelles. Il serait toutefois intéressant de considérer les autres théories, une grande part des résultats de ces algorithmes étant trouvés grâce à des monstres prometteurs...

3.2.2 Ontogenèse

Le deuxième axe traite du développement de l'être à partir d'un oeuf fécondé, le zygote. Lors d'une reproduction sexuée, cette cellule contient la moitié des gènes du père et la moitié de ceux de la mère, qui ensemble décrivent la manière dont l'organisme doit se construire. La création d'un organisme entier à partir de l'oeuf est une oeuvre extraordinairement complexe et sa compréhension complète nécessitera encore de nombreuses années de travail, si toutefois nous devons en percer tous les secrets un jour.

L'information nécessaire au développement est stockée dans les chromosomes, qui sont composés d'ADN (Acide DésoxyriboNucléique, dont la structure à été découverte en 1953 par Crick et Watson [252]), et forment le patrimoine génétique d'un être vivant. Le codage de base correspond à la manière dont sont arrangés les quatre acides nucléiques constituants de l'ADN : la Guanine, l'Adénine, la Cytosine, et la Thymine. Il est intéressant de constater que le génome humain contient environ 35'000 gènes alors que le corps humain adulte est composé de cent mille milliards de cellules (10^{14}). Il est donc bien clair qu'un gène ne décrit pas une cellule, mais qu'un mécanisme de développement est nécessaire à la traduction de l'information des gènes en un corps entier. Il est également important de constater que le patrimoine génétique est présent dans toutes les cellules de l'organisme (sauf dans les cellules germinales qui n'en contiennent que la moitié). Ceci implique une duplication de l'ADN lors d'une mitose (duplication de la cellule), et permet au corps de disposer d'autoréparation. En effet, lorsque des cellules meurent ou que le corps est blessé, le corps est capable de se réparer en cicatrisant. Les organismes vivants sont donc les meilleurs exemples de systèmes auto-réparateurs, laissant les machines loin derrière et du pain sur la planche pour les ingénieurs.

Quels sont donc les mécanismes de développement de l'embryon ? Nous n'entrons pas ici dans trop de détails, laissant le lecteur trouver des réponses dans [255] et [135], mais présentons succinctement les principes directeurs connus actuellement.

Position

La fonctionnalité d'une cellule dépend entre autre de sa position dans l'organisme, qui est définie par les interactions qu'elle entretient avec son environnement. Une partie de son génome peut alors être exprimée en fonction de ces interactions. Deux systèmes influançant le comportement de la cellule semblent probables, à savoir la diffusion d'un gradient chimique et la communication intercellulaire.

Gradient chimique Un composé chimique émis depuis un emplacement précis, la zone polarisante, pourrait diffuser sur au plus un demi millimètre, ce qui correspond à environ 30 à 50 cellules. Ces gradients chimiques seraient donc utilisés lors des premières phases du développement, le nombre de cellules y étant encore petit. Un processus serait alors déclenché en fonction de la quantité de produit chimique, laissant le reste du développement s'effectuer par un autre mécanisme. A titre d'exemple, les "doigts" numéros 2 à 4 d'un membre d'embryon de poulet sont initiés par une zone polarisante. En ajoutant artificiellement une deuxième zone, il est possible de créer un poulet avec deux fois plus de doigts disposés de façons symétrique avec une organisation de type 4,3,2,2,3,4 au lieu de 2,3,4.

Communication intercellulaire La communication intercellulaire est également importante. En fonction de son état interne, une cellule envoie des signaux chimiques à son voisinage. Ces interactions modifient alors le comportement interne des autres cellules, et de ce fait, de simples communications locales peuvent aider à la création d'un organisme.

Temps

Le temps semble jouer un rôle important dans le développement d'un organisme. Comme l'a observé l'équipe bâloise de Gehring [74], la formation de l'axe de la colonne vertébrale est dirigée par un ensemble de gènes organisateurs placés dans le génome dans un ordre bien précis. Ces gènes sont exprimés dans l'ordre de leur positionnement, et durant un temps donné. De cette manière, les gènes actifs à un instant précis définissent le type de vertèbre à générer. Toujours basé sur le temps, la création du bras se fait à partir d'un bourgeon comprenant une zone proliférative. Ce bourgeon part du tronc pour créer le bras entier, en générant de nouvelles cellules. Une cellule posséderait une horloge qui s'arrêterait lorsqu'elle quitte la zone proliférative. Le temps passé dans cette zone définirait alors la position de la cellule dans le bras.

Réseau de régulation de gènes

Finalement, citons les réseaux de régulation de gènes. Il existe en effet plusieurs types de gènes participant au développement, et entre autres les gènes de régulation et les gènes de différenciation. Les premiers génèrent des protéines, qui elles-mêmes activent ou inhibent d'autres gènes, et ceci dans un réseau de dépendance pouvant être inextricable. Cette dynamique de la cellule permet son fonctionnement et dans notre cas précis, le développement de l'organisme.

Cellules souches

Les gènes de différenciation sont exprimés à des stades particuliers, lorsqu'une cellule doit se différencier. En effet, il existe, dans le corps humain, environ 350 différents types de cellules (musculaires, neuronales, ...), qui servent toutes en des endroits bien particuliers de l'organisme. Au début du développement, l'embryon est composé de cellules souches, totipotentes, qui peuvent se différencier en n'importe quel type. Ces cellules ont donc le potentiel de pouvoir effectuer n'importe quelle tâche, jusqu'à leur spécialisation. Ce processus, que l'on croyait irréversible [179], est en fait réversible, et des cellules différenciées peuvent redevenir des cellules pluripotentes.

Contrôle

Durant le développement embryologique des invertébrés, il n'existe pas d'architecte unique qui supervise les travaux, les cellules étant seules responsables de cette tâche, et ce de manière totalement distribuée. Cette distribution du développement offre l'avantage qu'il n'existe pas de maillon faible central, le dysfonctionnement d'une cellule étant souvent contrecarré par son entourage. Chez les vertébrés, en revanche, un contrôle centrale existe sous forme d'hormones, qui irradient dans le corps entier.

Remarques

La diversité des mécanismes semblant être mis en jeu lors du développement de l'organisme est importante. Les chercheurs se sont inspirés de chacun de ceux-ci pour la mise au point de nouveaux algorithmes développementaux, que nous présenterons à la page 51. Notons juste que l'extrême complexité du développement embryonnaire, basé sur la quantité d'information limitée qu'est notre génome, laisse présager une manière de résoudre le problème de scalabilité[2] des systèmes artificiels.

3.2.3 Epigenèse

Troisième axe de la vie, l'épigenèse constitue ce que nous appelons communément l'apprentissage. Notre savoir, qui définit notre comportement, est composé d'éléments innés et acquis. Les premiers nous sont "imposés" par nos gènes, qui semblent coder certaines réactions du type réflexes. Les seconds correspondent à l'ensemble des expériences accumulées durant la vie d'un individu, qui l'enrichissent et influent sur ses décisions courantes. Cet apprentissage nécessite un systèmes nerveux capable d'adaptation, ce que quasiment tous les animaux pluricellulaires possèdent. L'être humain dispose quant à lui d'un cerveau volumineux composé de quelques 10 à 30 milliards de neurones.

Mais qu'est-ce qu'un neurone ? Il s'agit principalement d'une cellule particulière ayant quatre fonctions, à savoir de transmettre de l'information extérieure ou intérieure, analyser cette information, et potentiellement la mémoriser. Ils servent à récupérer les données sensorielles tels que la lumière pénétrant les yeux ou le toucher sur la peau, à traiter et stocker cette information, et à agir sur les muscles de manière à faire se mouvoir le corps. Ils sont composés d'un corps cellulaire, d'un axone capable de transmettre de l'information sur une longue distance, et de dendrites récupérant de l'information d'autres neurones.

Il existe dans le cerveau une foule de types de neurones différents. Leur fonctionnement de base est toutefois toujours semblable, et utilise des potentiels électriques. Un neurone est capable d'émettre un signal électrique transmis à ceux lui étant connectés. Cette émission dépend de l'état interne de la cellule ainsi que des stimuli reçus, et est typiquement activée lorsqu'un seuil de potentiel est atteint.

Il est intéressant de noter que l'épigenèse est fortement liée à l'ontogenèse, le cerveau n'étant que partiellement formé à la naissance (la capacité crânienne augmente ensuite de 4,3 fois) [155]. Durant le développement, les neurones se différencient et

---

[2] La scalabilité correspond à la faculté d'un système composé d'éléments de base à être étendu à un système en contenant un grand nombre.

se disposent en couches dans le cerveau, grâce à leur capacité migratoire. Ils sont formés sur la surface cérébrale, puis migrent vers le centre en s'accrochant à des cellules gliales, qui les guident dans la bonne direction. Ils s'arrêtent à un emplacement correspondant à leur fonction, créant ainsi les différentes couches cérébrales.

Les neurones sont reliés entre eux par des synapses qui sont créées à raison de 40'000 par secondes à la naissance. Un fait important est qu'une synapse ne perdure que si elle reçoit un feed-back environnemental, c'est-à-dire participe à un traitement d'information. Et alors que nous pensions que lorsque la construction du cerveau était achevée plus aucun neurone ne pouvait être généré, de récents travaux [73] montrent qu'il existe une forme de neurogenèse, des neurones, des axones, et des dendrites se créant durant toute la vie de l'individu.

3.3 Les Systèmes artificiels en 3 axes

La vie analysée selon ces trois axes organisationnels nous a montré sa formidable capacité d'adaptation, aussi bien au niveau individuel qu'à celui de l'espèce. Les chercheurs se sont grandement inspirés de ces processus lors de la mise au point de nouvelles techniques dans divers domaines. Nous allons visiter ces champs d'investigation dans les trois domaines précédemment présentés, avant de voir la manière dont ils peuvent être combinés jusqu'à arriver à un système POE.

3.3.1 Phylogenèse

Les théories de l'évolution décrites précédemment, et principalement le néo-darwinisme, ont conduit à la création des algorithmes évolutionnistes. Ces algorithmes permettent de résoudre des problèmes d'optimisation pour lesquels une solution déterministe ne peut être trouvée en un temps raisonnable. A titre d'exemple citons celui du voyageur de commerce, qui consiste à trouver le chemin le plus court pour la visite de n villes. Le nombre de solutions possibles y est de l'ordre de $n!$ [3], qui grandit de façon non polynomiale avec le nombre de villes. L'exécution d'un algorithme testant toutes les configurations possibles d'un tel systèmes n'est dès lors pas réaliste pour de grandes valeurs de n.

Les problèmes de ce type sont appelés NP-durs et nécessitent un temps de calcul non-polynômial, qui croît donc très vite en fonction de la taille du problème. Ils sont caractérisés par un espace de recherche très grand, sur lequel on ne peut pas effectuer une simple descente de gradient, à la manière d'une escalade de montagne, où, en suivant la pente il est aisé d'arriver au sommet. En robotique, par exemple, l'optimisation de paramètres ou la tâche de contrôle peuvent également ne pas forcément être résolues de manière déterministe.

Bien qu'utilisés pour l'optimisation, il est important de noter qu'ils ne fournissent pas obligatoirement la solution optimale à un problème. Le résultat est plutôt une solution acceptable, qui approche assez l'optimum pour satisfaire le problème.

Le premier algorithme évolutionniste, appelé algorithme génétique [101], fut proposé par John Holland. De nombreuses adaptations furent développées, comme la programmation génétique [129] de John Koza, visant à laisser un programme informatique être généré automatiquement à partir d'une tâche à effectuer, la Particle Swarm

---

[3] $n! = n * (n-1) * (n-2) * ... * 2 * 1$

Optimization [119], ou les Stratégies Evolutives [190]. Un bon résumé des différentes approches peut être trouvé dans l'article de Back [18], et nous ne présenterons ici que les algorithmes génétiques.

Algorithme génétique

Le concept de base d'un algorithme génétique, dont le pseudo-code est décrit par l'algorithme 3.1, repose sur une population d'individus, chacun étant représenté par un génome. Le génome, défini sur un alphabet à choisir, code une des solutions possibles de l'ensemble des solutions, typiquement une chaîne binaire, un ensemble de nombres réels, ou une structure en forme d'arbre. Lors de la résolution d'un problème par algorithme génétique, la première question à se poser consiste en le choix de l'alphabet, qui aura une grande influence sur l'efficacité de l'algorithme. A titre d'exemple, si nous voulons trouver, pour une fonction réelle du type $f(x, y, z)$, les valeurs de x, y, et z permettant de minimiser cette fonction, il sera peut-être plus judicieux de choisir comme génome d'un individu trois valeurs réelles (x, y, z) plutôt qu'un codage binaire $(x_0, x_1, \cdots, x_n, y_0, y_1, \cdots, y_n, z_0, z_1, \cdots, z_n)$, où $x_i, y_i, z_i \in \{0, 1\}$ pour $i \in \{0, \cdots, n\}$.

Dans la phase d'initialisation, une population d'individus est générée aléatoirement. Chaque individu est ensuite évalué et son fitness calculé en fonction de la tâche à optimiser. Basée sur ces évaluations, une sélection est appliquée à la population afin de créer une nouvelle génération. Un possible crossing-over et une potentielle mutation sont appliqués aux nouveaux individus avant de les réévaluer, et tout ce processus est exécuté jusqu'à ce qu'une solution acceptable soit trouvée ou que le nombre limite d'itérations soit atteint.

Algorithme 3.1 Algorithme Génétique

```
Entrées : Un problème, et particulièrement une fonction de fitness
Résultat : Une solution acceptable
Initialiser la population g(0) aléatoirement
Evaluer chaque individu de la population g(0)
Tant que une solution acceptable n'est pas trouvée ou le nombre limite d'itérations n'est pas atteint Faire
    Générer la population g(t+1) par sélection, t=t+1
    Recombiner g(t)
    Muter g(t)
    Evaluer la population g(t)
Fin tant que
```

Nous allons maintenant détailler quelque peu les différentes opérations d'un algorithme génétique standard.

☐ **Initialisation**

Lors de l'initialisation de l'algorithme, une population de n individus est créée, en générant des génomes aléatoires. Pour une chaîne binaire, chaque bit sera fixé à '0' ou '1' avec une probabilité de 0.5, et pour une représentation en arbre, un petit arbre aléatoire pourra être créé. Bien que cette manière de faire peut sembler la plus normale, un problème de bootstrap peut toutefois surgir. En effet, si tous les individus générés ont un fitness nul, l'évolution ne pourra pas démarrer efficacement. Dans ce cas la génération de la première génération pourra être tronquée, en biaisant les probabilités, ou en se basant sur de précédents résultats.

☐ **Calcul du fitness**

Le calcul du fitness est l'opération centrale de l'algorithme, puisque c'est elle qui définit le but à atteindre. Il s'agit d'évaluer chaque individu de la population en fonction du problème posé. Toute l'efficacité de l'algorithme, outre le taux de mutation et de recombinaison, réside dans la manière de calculer le fitness à partir du génome. Pour ce faire, la façon de représenter le génome est cruciale et doit non seulement minimiser la taille de celui-ci, mais aussi favoriser une évaluation rapide de l'individu.

Dans la majorité des cas cette opération est la plus gourmande en temps de calcul (typiquement de l'ordre de 99%).

☐ **Sélection**

Une nouvelle génération est créée à partir de la génération courante, et ceci en fonction des *fitnesses* des différents individus. Pour ce faire, différents types d'algorithmes existent :

- **La sélection proportionnelle** utilise directement la valeur de fitness des individus. Après les avoir normalisés dans l'intervalle $[0, 1]$, leur valeur associée sert de probabilité de sélection. La sélection consiste donc à placer les individus dans un camembert avec une aire proportionnelle à leur fitness, et de lâcher n fois une pièce, gardant ainsi l'individu sur l'aire duquel la pièce est tombée. Un grand désavantage de cette méthode réside dans le fait qu'un individu largement supérieur aux autres sera sélectionné un nombre trop élevé de fois, empêchant une bonne exploration du champ de solutions avec le risque de voir l'algorithme converger rapidement vers un minimum local.
- **La sélection selon le rang** permet d'éviter le problème de la sélection proportionnelle. Les individus y sont listés par ordre de valeur de fitness, et une probabilité de sélection est donnée à chacun. De cette manière il est possible d'empêcher le meilleur individu d'être sur-sélectionné. Pour reprendre la visualisation du camembert, ici l'aire d'un individu ne correspond plus à son fitness, mais à une valeur dépendante de sa place dans la liste ordonnée.
- **La sélection par tournoi**, pour un tournoi de valeur k, choisit au hasard k individus, et sélectionne le meilleur pour la génération suivante, et ce n fois. Cette approche laisse une plus grande chance aux individus n'ayant pas un excellent fitness de faire partie de la génération suivante.
- **La sélection tronquée** est une approche dans laquelle les x meilleurs individus sont conservés en formant n/x descendants. Cette manière de faire peut toutefois conduire à une perte de diversité, trop peu de parents étant sélectionnés.
- **Elitisme** Jumelée à une des quatre approches précédentes, l'élitisme consiste en la sélection automatique des x meilleurs individus, qui sont placés tels quels dans la nouvelle génération. Il permet ainsi d'éviter la perte potentielle des meilleures solutions.

☐ **Recombinaison**

Après avoir été sélectionnés, les individus de la nouvelle génération peuvent se recombiner, avec une certaine probabilité. Cette opération, appelée crossing-over en anglais, mélange deux génomes de manière à potentiellement pouvoir trouver de nouvelles solutions pouvant être meilleures. La figure 3.2 présente une recombinaison de deux chaînes binaires de 8 bits avec un point de recombinaison, et la figure 3.3 montre une recombinaison de deux arbres, où deux arcs sont choisis aléatoirement, les deux sous-arbres étant alors interchangés.

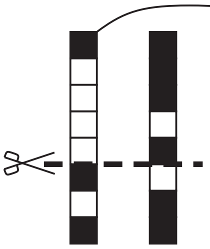


Figure 3.2 : *Exemple de recombinaison d'une chaîne binaire.*

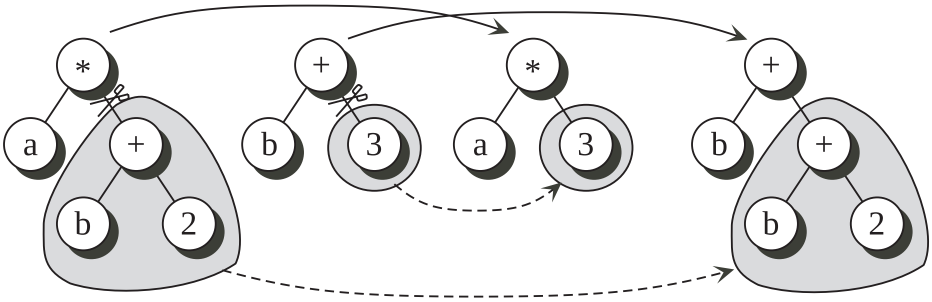
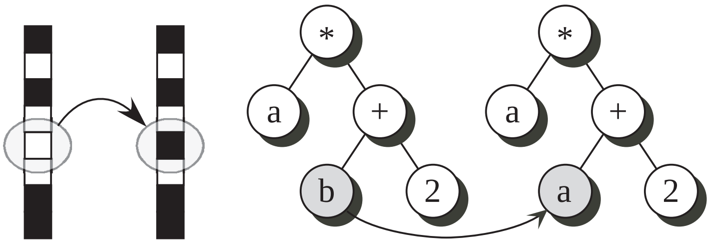


Figure 3.3 : *Exemple de recombinaison d'un arbre.*

☐ **Mutation**

Après avoir été possiblement recombiné, le génome d'un individu peut subir des mutations selon une probabilité prédéfinie. Pour une chaîne binaire il s'agira de changer la valeur d'un bit (Figure 3.4(a)), et pour un arbre il pourra s'agir de modifier un noeud (Figure 3.4(b)). Les mutations permettent de maintenir une certaine diversité au sein d'une population qui pourrait rapidement converger. De plus, elle favorise l'exploration, en proposant de nouveaux génomes qui ne pourraient être obtenus par simple recombinaison.



(a) Exemple de mutation d'une chaîne binaire

(b) Exemple de mutation d'un arbre

Figure 3.4 : *Exemples de mutations.*

Matériel évolutif

Les algorithmes évolutionnistes sont largement utilisés dans un grand nombre de domaines, allant du contrôle de robot [178] à la création d'antennes pour satellites [140], en passant par l'optimisation de la forme de lentilles [172]. Nous n'allons toutefois nous intéresser qu'à l'un de leurs champs d'application, à savoir le matériel évolutif.

La création de circuits électroniques est, de manière traditionnelle, menée à bien par des ingénieurs. Toutefois, les algorithmes génétiques ont déjà montré leur capacité à générer des designs portant sur des problèmes particuliers. Le principe y est de fournir au système des entrées et de le faire évoluer de manière à ce que les sorties correspondent au résultat désiré. L'avantage de l'approche évolutive est qu'elle peut générer des solutions que des ingénieurs n'auraient pas trouvées, l'algorithme n'étant pas cantonné aux méthodes de design traditionnelles, mais étant basé sur un système d'essais et de sélection. Elle offre également des propriétés de tolérance aux pannes. En effet, même si le matériel utilisé est défectueux, l'évolution est capable de contourner le problème et de générer des designs effectuant la tâche requise, alors qu'un ingénieur n'y parviendra que très difficilement. De plus, le système peut être réévolué dans le cas d'une nouvelle panne, de manière à le rendre à nouveau opérationnel. Finalement, cette approche pourrait, s'il n'était un problème de scalabilité [242], remplacer les développeurs dans la tâche de design. Parmi les utilités potentielles de tels systèmes, nous pouvons citer Gordon et Bentley [80] :

- Design automatique de matériel à bas coût,
- Faire face à des problèmes non-entièrement spécifiés,
- Création de systèmes adaptatifs,
- Création de systèmes tolérants aux fautes [64],
- Innovation dans des espaces de design peu compris.

Le développement du matériel évolutif ne s'est fait que grâce à l'introduction des circuits programmables. Plusieurs générations de centaines d'individus devant être testés seraient totalement irréalisables en utilisant une technologie d'application de masques et de dopage de silicium. C'est donc les PLDs, puis les FPGAs, qui ont permis l'essor de ce nouvel axe de recherche. Les PALs ont été utilisés dans des applications ne nécessitant qu'une combinatoire simple, alors que les FPGAs l'ont été pour des systèmes plus complexes.

Sur le plan des FPGAs, ce fut la famille XC6200 de Xilinx qui lança leur mise à profit. Son architecture est régulière et très simple, un élément de base ne contenant qu'une bascule et quelques multiplexeurs (cf. page 23). L'ensemble des bits de configuration d'une XC6200 est public, et donc connu, et permet aux chercheurs de définir exactement les bits à évoluer. De plus, sa configuration s'exécute de manière très rapide, grâce à un accès de type RAM. Finalement, son implémentation basée sur des multiplexeurs en ont fait le candidat idéal pour le matériel évolutif, aucune combinaison de bits de configuration ne pouvant détruire le circuit. Malheureusement, Xilinx a cessé de produire cette famille, et les chercheurs doivent à l'heure actuelle se tourner vers d'autres solutions, tels les Virtex de Xilinx, qui n'offrent pas la même facilité d'emploi pour ce type d'application.

Chaque application fait intervenir un style d'évolution différent, que ce soit par rapport au niveau d'abstraction ou à la manière dont sont évalués les individus. Nous présentons ici les paramètres d'un système de matériel évolutif permettant un classement de ceux-ci :

Niveau d’abstraction Le design de circuits électroniques peut se faire à plusieurs niveaux. Il est possible de créer directement les masques nécessaires à la réalisation du circuit, de travailler au niveau des transistors, des portes logiques, ou encore avec des langages de haut niveau. Les systèmes évolutifs peuvent également être appliqués à ces différents niveaux d’abstraction :

- Bits de configuration du circuit [139, 226, 235]
- Portes logiques [103, 115, 158, 157]
- Fonctions de type addition, multiplication [118, 171]
- Circuits analogiques, au niveau de la connectique des composants [83, 263]
- Machines d’états [95, 145, 153]
- Langage haut niveau HDL [123]

Contrainte/non contrainte Lorsque nous désirons faire évoluer un système matériel, nous lui imposons des contraintes étant au minimum le type de circuit reconfigurable que nous utilisons. Dans la plupart des applications, des blocs de base de type portes ET et OU sont utilisés pour l’évolution, la rendant contrainte. Si en revanche, aucune autre contrainte que le type de circuit n’est imposée, et que l’ensemble des bits de configuration sont évolués, nous parlons d’évolution non contrainte. Il est intéressant de noter que l’évolution non contrainte permet de générer des solutions qu’un ingénieur n’aurait pas imaginées, l’algorithme tirant parti de toutes les propriétés du matériel. Les expériences de Thompson [237] ont par ailleurs montré qu’un design évolué de cette manière sur un FPGA particulier ne fonctionnait pas forcément sur un autre circuit, les transistors et les fils n’étant pas rigoureusement identiques.

On-chip/off chip Comme nous l’avons vu, le processus d’évolution peut être séparé en deux phases, le calcul du fitness et la sélection/combinaison/mutation des individus. Cette deuxième phase est couramment effectuée par un processeur. Dans le cas où ce processeur est externe au circuit soumis à l’évolution, il s’agit d’un système off-chip. Dans le cas contraire, lorsque le processeur, spécialisé [209] ou non, est présent sur le même circuit, il s’agit d’un système on-chip.

Online/Offline Dans certains systèmes, l’évolution est appliquée avant utilisation, de manière à générer une bonne solution, et seule la meilleure est exploitée (offline). Dans d’autres, l’évolution à lieu tout au long du fonctionnement du système, des individus étant sans arrêt confrontés aux situations réelles, et sélectionnés en conséquence de leur efficacité (online).

Intrinsèque/Extrinsèque L’évaluation des individus peut se faire de deux manières : en simulation (extrinsèque), ou grâce au matériel réel (intrinsèque). Une option parmi ces deux termes, introduits par de Garis en 1993 [55], est choisie en fonction de plusieurs critères. La vitesse d’exécution de l’algorithme peut être critique, si la tâche de l’individu est complexe. Dans ce cas, si le temps de chargement du FPGA additionné au temps d’exécution en matériel est plus court que le temps de simulation, une solution intrinsèque est préférable.

Lors d'un processus d'évolution à un bas niveau d'abstraction, comme dans les travaux de Thompson [236], l'évaluation intrinsèque est capitale. S'agissant d'évolution non-contrainte, l'ensemble des bits de configuration sont évolués, laissant l'algorithme profiter des spécificités du matériel. Son application de génération d'oscillateurs sur un FPGA XC6200 a mis en avant l'efficacité de l'évolution. Les résultats étaient tellement spécialisés pour un FPGA donné qu'ils ne fonctionnaient pas sur un autre circuit, chaque FPGA ayant d'infimes différences de fabrication rendant leur comportement électronique très légèrement différent. De même, un simple changement de température du circuit peut modifier le comportement du système.

Limitations L'évaluation du fitness d'un individu, ainsi que la scalabilité, sont les deux problèmes qui limitent l'exploitation du matériel évolutif. Concernant l'évaluation du fitness, il n'est pas toujours aisé de définir correctement la manière dont elle doit s'effectuer, si les sorties ne peuvent pas directement être calculées en fonction des entrées. De plus, le temps d'exécution peut être prohibitif. S'il s'agit de faire apprendre à un robot à éviter des obstacles, il est nécessaire de le laisser rouler pendant un nombre non négligeable de secondes. Multiplié par le nombre d'individus d'une population, et par le nombre de générations nécessaire à l'obtention d'une solution acceptable, ce temps peut prendre des proportions dramatiques.

Sur le plan de la scalabilité, un système comparable à un processeur, contenant plusieurs millions de transistors, n'est à l'heure actuelle pas réalisable via cette approche. Le génome décrivant un tel système au niveau des portes serait nettement trop long, et créerait un espace de recherche quasiment infini. Afin de pallier à cette limitation majeure, plusieurs équipes ont travaillé à la création incrémentale de solutions. Des blocs de taille acceptable y sont évolués et offrent ensuite une librairie d'éléments à utiliser dans une seconde phase évolutive [117, 242, 239]. Dans la même optique, Kajitani a utilisé un génome à longueur variable [114], tandis que nombre d'approches misent sur de nouvelles techniques de mapping entre génotype et phénotype[4].

Il est intéressant de constater que, durant le siècle passé, les théories de l'évolution et la biologie moléculaire du développement étaient deux disciplines bien distinctes. A l'heure actuelle, elles sont en train de se rejoindre, car les travaux en biologie moléculaire du développement offrent de nouvelles perspectives aux théories de l'évolution, comme le montre le transitionnisme. Dans les systèmes artificiels, nous observons le même type d'interaction entre les deux mêmes axes, phylogenèse et ontogenèse, l'ontogenèse permettant de modifier le mapping entre génotype et phénotype, et donc de changer la structure du génome afin de minimiser sa taille et de réduire le problème de scalabilité. Nous verrons dans la section suivante certaines techniques de mapping qui améliorent la scalabilité, mais qui n'offrent toutefois pas la solution miracle.

3.3.2 Ontogenèse

Inspirés par la manière dont les êtres vivants se développent et cicatrisent, l'ontogenèse vise la création de systèmes capables de croître, de s'auto-organiser, et de s'auto-réparer. Les deux buts sous-jacents d'un système ontogénétique dépendent du type d'application à réaliser. Le premier est la scalabilité des algorithmes évolutionnistes, et le deuxième est l'autoréparation, ou la tolérance aux pannes, des systèmes matériels.

---

[4] Le phénotype est l'ensemble des caractères individuels correspondant à une réalisation du génotype.

Nous venons de voir que la scalabilité des algorithmes évolutionnistes pose problème lorsque le génome devient trop grand. Dans le cadre de systèmes évolutifs, une méthode d'ontogenèse (ou développement) permet alors d'avoir à disposition un mapping particulier du génotype au phénotype. Le principe est identique aux mécanismes du vivant : les cellules d'un être humain ne peuvent être décrites individuellement dans le génome, et donc un système ontogénétique est obligatoire pour la création des milliards de cellules qui nous constituent. Les systèmes artificiels multicellulaires, c'est-à-dire dont la fonctionnalité peut être décomposée en plusieurs parties relativement semblables, peuvent être conçus de la même manière.

En 1968, Lindenmayer introduit les systèmes de réécriture, appelés L-systèmes [141]. Basés sur des chaînes de caractères, et partant d'un axiome de départ, ces systèmes appliquent à chaque pas de temps des règles de production à la chaîne de caractères. Chaque caractère peut représenter une cellule, ou une liaison entre cellules, et l'évolution de la chaîne de caractères correspond donc à la création d'un organisme multicellulaire. Ce type de mécanisme développemental a souvent été utilisé en conjonction des algorithmes évolutionnistes, afin de réduire la taille du génome. A titre d'exemple, citons Kitano [122], qui a utilisé des grammaires génératives basés sur des L-systèmes pour créer des graphes représentant la structure d'un réseau de neurones. Son approche a montré une meilleure scalabilité par rapport à un encodage direct de la structure du réseau, mais il faut remarquer que cette simplification du codage implique que toutes les structures de réseau ne peuvent être générées de cette manière. Gruau [84] a également utilisé des grammaires génératives pour créer des réseaux de neurones. Cependant, au lieu de réécrire des caractères, ce sont les neurones qui sont directement affectés par les règles de réécriture. Notons finalement que l'implémentation matérielle des systèmes de réécriture n'est pas des plus évidentes, car de simples règles locales ne permettent pas d'implémenter ces mécanismes de façon efficace.

La tolérance aux pannes peut également tirer profit des systèmes ontogénétiques. Le fait de disposer de systèmes multicellulaires évite la présence d'un contrôle global, qui, s'il flanche, met en péril tout le fonctionnement du système. De plus, la communication intercellulaire, qui est utilisée lors du développement, peut également l'être lors de l'autoréparation.

Les chimies artificielles, domaine de recherche en pleine expansion, en sont un excellent exemple. Miller [159] en a créé une, améliorée ensuite par Capcarrère et Ozturkeri [181], qui permet de générer un drapeau français ou allemand dans une grille régulière de cellules. Partant d'une seule cellule, et en se basant sur des interactions locales, le drapeau se construit automatiquement. Chaque chimie particulière est générée grâce à un algorithme évolutionniste qui se charge de trouver des règles de comportement cellulaire adaptées. Le fait important est ici que l'altération de l'état d'une ou plusieurs cellules peut être contrecarrée par l'action de la chimie artificielle. Le drapeau retrouve alors sa forme initiale, ou une forme quasiment identique.

Finalement, toujours sur le plan de l'autoréparation, citons le projet Embryonique [142, 144, 230], développé au Laboratoire de Systèmes Logiques par l'équipe du professeur Mange. Il a vu la réalisation d'un circuit capable d'exécuter n'importe quelle tâche, ainsi que de s'auto-réparer et de s'auto-répliquer. Ce circuit est une sorte de FPGA dont chaque élément de base, appelé molécule, est un multiplexeur et une bascule. Un système d'autoréparation au niveau moléculaire est implémenté en matériel, et permet, en plaçant des colonnes de molécules en attente, de faire face à des erreurs du matériel. Ce substrat électronique est ensuite utilisé pour l'implémentation d'un organisme multicellulaire, une cellule étant composée d'un certain nombre de molécules. Chaque cellule y contient le génome complet de l'organisme, de manière à ce qu'au lancement du système, chaque cellule soit totipotente. Un système de différenciation basé sur des coordonnées à deux dimensions permet alors à la cellule de sélectionner une tâche à accomplir, et ce même système est utilisé lors de l'autoréparation cellulaire. En effet, lorsque les molécules ne peuvent plus supporter de nouvelles fautes, la cellule meurt, sa colonne cellulaire entière est mise hors service, et les colonnes suivantes sont toutes décalées, jusqu'à atteindre une colonne de cellules totipotentes. Toutes ces colonnes se redifférencient et le circuit peut continuer à fonctionner, sans que sa fonctionnalité n'ait été altérée.

L'approche Embryonique est fortement inspirée du vivant, de par la structure multicellulaire du système, la présence du génome dans chaque cellule, la croissance de l'organisme, et de par l'autoréparation. En ce sens, l'approche POEtic est une sorte de continuité de ce paradigme, auquel nous désirons ajouter de l'évolution et de l'apprentissage. Concernant les faiblesses d'Embryonique, nous pouvons en citer deux, qui serviront lors de la réalisation du tissu POEtic. Premièrement, la structure multicellulaire et le système de différenciation sont trop figés. Le système de coordonnées utilisé oblige à la destruction d'une colonne entière de cellules dans le cas d'une réparation cellulaire, ce qui pourrait être évité dans un système plus souple. Deuxièmement, la structure moléculaire offre une faible fonctionnalité, ce qui implique la réquisition d'un grand nombre de molécules, en comparaison des FPGAs actuels, pour l'implémentation de designs peu complexes.

3.3.3 Epigenèse

Troisième et dernier axe, l'épigenèse se propose de s'inspirer de la manière dont les neurones biologiques fonctionnent pour créer leurs cousins artificiels. L'adaptation s'y effectue durant la vie d'un seul individu, en comparaison des mécanismes phylogénétiques, qui font intervenir toute une population. De nombreuses approches à l'apprentissage sont possibles, mais nous ne traiterons ici que des réseaux de neurones, qui sont capables de modéliser nombre de phénomènes, de s'adapter à de nouvelles données, et de généraliser les exemples fournis lors de l'apprentissage.

Deux familles de neurones artificiels ont retenu l'attention des chercheurs. Premièrement, les neurones standards, qui calculent immédiatement une sortie en fonction de leurs entrées, et deuxièmement, les neurones à impulsions, qui travaillent avec une composante temporelle. Nous allons présenter ces deux types de neurones ainsi que différents modes d'apprentissages.

Modèle standard

La première définition de neurones artificiels est due à McCulloch et Pitts [150], qui proposent en 1943 la modélisation des neurones naturels par des neurones "tout ou rien", c'est-à-dire basés sur une logique binaire. Ils partent des suppositions suivantes, tirées de [150], page 118 :

1. L'activité d'un neurone est un processus "tout ou rien".
2. Un certain nombre, fixe, de synapses doivent être excitées durant la période d'addition latente pour exciter un neurone à un instant

donné, et ce nombre est indépendant de l'activité et de la position précédente du neurone.

3. Le seul délai significatif dans le système nerveux est le délai synaptique.
4. L'activité de n'importe quelle synapse inhibitrice empêche l'excitation du neurone à cet instant.
5. La structure du réseau ne change pas avec le temps.

Un tel réseau de neurones peut modéliser nombre de processus, pouvant être récursif ou non. Toutefois, le fait qu'il soit binaire ne l'a rendu que peu attractif pour des applications réelles. La deuxième moitié du siècle dernier a donc vu poindre de nouveaux types de neurones, proposant des calculs sur des nombres entiers ou à virgule. Dans le modèle standard, bien décrit par Rumelhart dans [198], un neurone est caractérisé par :

- Un niveau d'activation, qui représente la polarisation du neurone,
- Une valeur de sortie, correspondant au taux d'activation du neurone,
- Un ensemble d'entrées, qui représentent les synapses des dendrites,
- Un ensemble de poids synaptiques, chacun étant associé à une entrée, et définissant l'influence de l'entrée sur la sortie du neurone. Un poids positif signifie que la liaison est excitatrice, alors qu'un poids négatif indique que le neurone source est inhibiteur.
- Une valeur de biais, représentant le niveau de repos du neurone.

Il existe bien sûr plusieurs manières d'implémenter de tels neurones, aussi bien concernant le type de données (binaire, entier, flottant), que la fonction de seuil. Dans le cas du neurone de type Perceptron, les entrées x_j sont en général sommées après avoir été pondérées par leurs poids w_{ij}, et le biais β_i y est ajouté (Equation 3.1). Une fonction de seuil est ensuite appliquée au résultat afin de calculer la sortie qui peut être transmise aux neurones suivants. Typiquement, la fonction de seuil peut être une sigmoïde, comme l'équation 3.2, où T représente la pente de la sigmoïde. La figure 3.5 présente l'architecture global d'un tel neurone.

$$\eta_i(t) = \sum_j \omega_{ij} x_j(t) + \beta_i \tag{3.1}$$

$$y_i(t) = \frac{1}{1 + e^{\frac{-\eta_i(t)}{T}}} \tag{3.2}$$

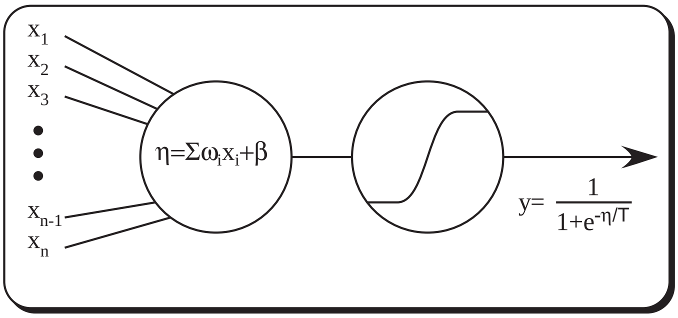


Figure 3.5 : *Architecture générale d'un neurone artificiel de type Perceptron.*

Neurones à impulsions

Les neurones que nous venons de présenter ont montré leur efficacité dans grand nombre de problèmes, mais ne correspondent pas à la réalité biologique. En effet, les neurones naturels fonctionnent sur un système d'impulsions.

Dans ce modèle, décrit en détail par Gerstner et Kistler dans [75], un neurone est caractérisé par :

- Un potentiel de membrane, qui évolue au cours du temps,
- Une valeur de sortie, qui est ici un potentiel pouvant prendre deux valeurs : repos ou activation.
- Un ensemble d'entrées, qui représentent les synapses des dendrites,
- Un ensemble de poids synaptiques, chacun étant associé à une entrée, et définissant l'influence de l'entrée sur la sortie du neurone. Un poids positif signifie que la liaison est excitatrice, alors qu'un poids négatif indique que le neurone source est inhibiteur.
- Une valeur de repos, correspondant à l'état du neurone lorsqu'aucune entrée n'a été activée depuis un certain temps,
- Un seuil d'activation, qui, lorsqu'il est atteint par le potentiel de membrane, provoque une activation de la sortie du neurone.

Le fonctionnement du neurone, illustré à la figure 3.6, est basé sur son potentiel de membrane, qui est initialement à sa valeur de repos. Lorsqu'une entrée est activée, ce potentiel grimpe rapidement d'une hauteur dépendant du poids synaptique de la dendrite correspondante. Ensuite il descend doucement tant qu'il n'y a pas de nouvelle entrée active. Lorsque plusieurs activations se trouvent assez rapprochées pour ne pas laisser le potentiel redescendre, il peut atteindre le seuil d'activation. A ce moment-là, la sortie du neurone est activée durant un court instant, et le potentiel de membrane redescend brusquement en dessous du niveau de repos, créant ainsi une période réfractaire durant laquelle aucune nouvelle activation ne peut avoir lieu.

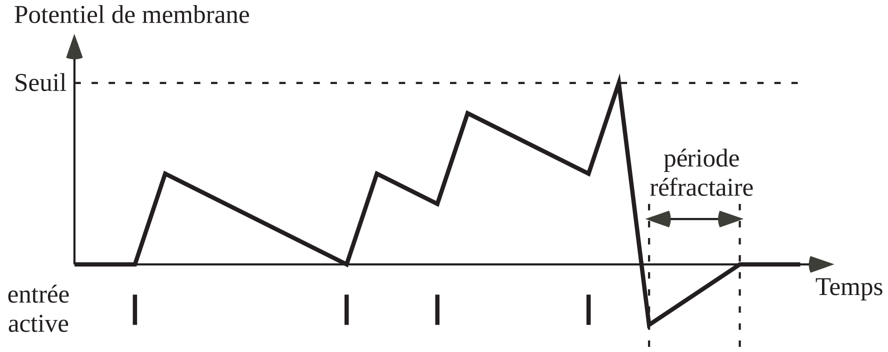


Figure 3.6 : *Fonctionnement d'un neurone à impulsions.*

Contrairement au modèle standard, où le neurone n'a pas d'état interne, le potentiel de membrane définit ici un état du neurone, qui dépend de son histoire. Le temps joue donc un rôle majeur, de par la modification incessante du potentiel de membrane.

Alors que dans le modèle standard, le codage de l'information est relativement direct (un neurone peut simplement récupérer la valeur d'une entrée), dans le cas par impulsions, la question est nettement moins triviale. Le type de codage est intimement lié au temps qui s'écoule, et peut par exemple être corrélé avec la densité moyenne d'activation de certains neurones ou avec le temps écoulé entre deux activations.

Apprentissage

L'apprentissage des réseaux de neurones artificiels sont pour la plupart basés sur l'observation de Hebb, qui, en 1949, introduit une règle d'apprentissage qui porte son nom [91], qu'il énonce ainsi :

> "Lorsqu'un axone de la cellule A est suffisamment proche pour pouvoir exciter la cellule B et qu'il prend part de manière répétitive ou persistante à cette excitation, alors on doit trouver soit un phénomène de croissance, soit un changement métabolique dans l'une ou l'autre des cellules tel que l'efficacité de la cellule A pour exciter B doit être accrue." [91]

Cette règle modifie itérativement les poids synaptiques en fonction de l'activité des neurones pré- et post-synaptiques. S'ils sont corrélés, le poids aura tendance à augmenter, afin de renforcer encore leur corrélation. Les connexions entre neurones synchronisées sont donc renforcées. Mathématiquement parlant, cette règle correspond à la simple équation 3.3.

$$w_{ij}(t+1) = w_{ij}(t) + \eta y_j(t) x_i(t) \tag{3.3}$$

Elle est biologiquement inspirée, peut être utilisée avec tous types de neurones, qu'ils soient standards ou à impulsions, et constitue un type d'apprentissage non-supervisé, qui, de par sa composante locale, et a l'avantage d'être facilement implémentable en matériel.

L'apprentissage d'un réseau de neurones se fait généralement par modification des poids synaptiques, et parfois par l'ajout ou la suppression de neurones et de synapses. Notons toutefois que la modification de la topologie du réseau peut se réduire à la modification de poids synaptiques, un poids de 0 correspondant à l'absence de liaison entre deux neurones. Suivant l'application cible, différentes règles d'apprentissage peuvent être envisagées. Avant d'en présenter les plus importantes, notons que deux types d'apprentissage existent :

Supervisé Le réseau de neurones est entraîné à effectuer une tâche précise définie par le développeur. Ce dernier peut lui donner une récompense lorsqu'un ensemble d'action a été bénéfique, comme dans le cas de l'apprentissage par renforcement [113], ou encore, lorsqu'il connaît la réponse prévue du système pour un stimulus particulier, il peut lui présenter un ensemble d'exemples, et le faire s'adapter de manière à fournir les sorties désirées pour un maximum de tests. Le grand avantage d'un réseau de ce type sur un système ne faisant que comparer les entrées avec une base de donnée d'exemples est ce que l'on appelle la généralisation. En effet, après la phase d'apprentissage, la présentation d'un exemple proche d'un de ceux utilisés précédemment aura de grandes chances de produire la même sortie. En reconnaissance de l'écriture, par exemple, cette propriété est cruciale, une lettre n'étant jamais écrite exactement de la même manière par toutes les personnes.

Non-Supervisé Le réseau n'est pas entraîné avec des échantillons dont on connaît la réponse, mais s'entraîne seul, avec les données qui lui sont fournies en entrée. Ce type d'apprentissage permet donc de classifier des échantillons selon des classes indéfinies, le réseau créant lui-même un partitionnement de l'espace des entrées.

Réseaux et apprentissage

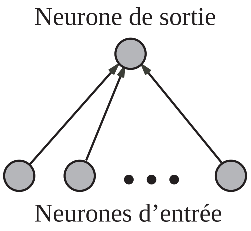


Figure 3.7 : *Perceptron simple couche.*

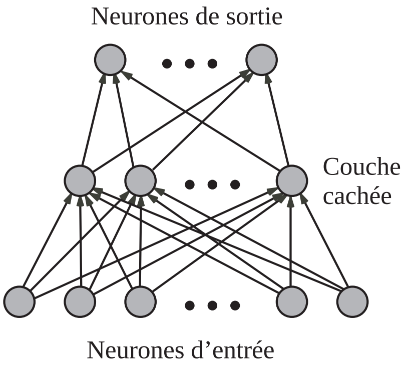


Figure 3.8 : *Architecture Perceptron multi-couche.*

Un réseau de neurones est créé en connectant les neurones entre eux selon une architecture, qui là encore, peut grandement varier. Le Perceptron simple couche[5] (Figure 3.7) est une des architectures les plus connues, où des neurones d'entrée ne font que récupérer les entrées du réseau, et les envoient au neurone de sortie, qui calcule la sortie du réseau en appliquant les équations 3.1 et 3.2. Ce type de réseau permet d'effectuer une séparation linéaire de l'espace des entrées. Plus complexe, le perceptron multi-couche (Figure 3.8) introduit une ou plusieurs couches de neurones cachés, chaque couche de neurones ayant comme entrée les sorties des neurones de la couche précédente, jusqu'à obtenir la couche de sortie. D'autres réseaux sont récursifs, une couche pouvant influencer des neurones de la même couche ou de couches inférieures (Figure 3.9), tandis que d'autres réseaux sont totalement connectés, à la manière de ceux de Hopfield (Figure 3.10).

Back-Propagation L'architecture de type Perceptron permet d'implémenter un système d'apprentissage par rétro-propagation de l'erreur [254], et donc d'utiliser

---

[5] Le lecteur attentif aura repéré que la figure 3.7 possède deux couches. La couche d'entrée n'exécutant toutefois pas de calcul, elle n'est pas comptée, et ce réseau n'est rien d'autre qu'un perceptron simple couche.

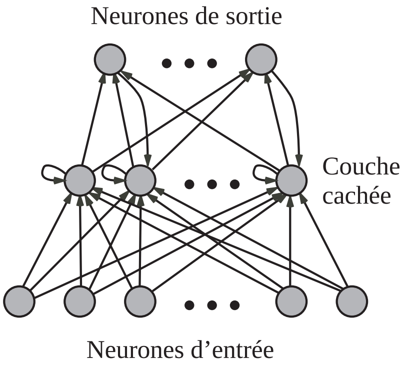


Figure 3.9 : *Réseau de type Perceptron, avec boucles de rétroaction.*

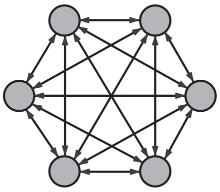


Figure 3.10 : *Réseau totalement connecté.*

le réseau dans des tâches comme la reconnaissance de formes. Un grand nombre d'exemples sont présentés au réseau, et la modification des poids synaptiques selon un calcul d'erreur permet d'augmenter ses capacités de reconnaissance. Chaque poids synaptique est modifié après la présentation d'un exemple, et ce en proportion de la dérivée de l'erreur, et en fonction du poids. Le changement est donc proportionnel à l'équation 3.4. Cet algorithme est grandement utilisé dans nombre d'applications nécessitant un apprentissage supervisé.

$$\frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial w_{ij}} \tag{3.4}$$

Kohonen Les réseaux de Kohonen [124] sont des réseaux permettant de projeter des données d'un espace à N dimensions dans un espace de dimension moindre (typiquement à 2 dimensions). Les éléments proches dans le premier espace le seront également dans le deuxième, lorsque l'apprentissage est terminé. Le réseau est constitué d'une couche d'entrée comprenant N neurones (un pour chaque dimension), d'une grille de neurones de sortie, et des connexions W qui lient les deux couches. Lors de la phase d'apprentissage, les échantillons sont présentés en entrée, et le neurone de sortie étant le plus proche de l'échantillon est sélectionné (il s'agit de celui dont l'équation 3.5 est minimale).

$$d_i = \sum_{j=1}^{M}(I_j - W_{ij})^2 \tag{3.5}$$

Ses poids synaptiques sont alors modifiés selon l'équation 3.6, où η représente le coefficient d'apprentissage, dans [0,1], qui décroît avec le temps. Les neurones voisins du neurone sélectionné sont également modifiés avec la même équation, en tenant compte de leur distance au neurone sélectionné.

$$W_{kj}(t+1) = W_{kj}(t) + \eta(t) * (I_j(t) - W_{kj}(t)) \tag{3.6}$$

Les applications des réseaux de Kohonen vont de la sélection de données représentatives dans une grande base de cas à la compression d'images, en passant par la modélisation de la cartographie des aires visuelles.

Hopfield D'autres réseaux utilisent des boucles de rétroaction, les entrées d'une couche pouvant être les sorties de n'importe quelle autre couche. Hopfield [106] en a fait des systèmes capables de mémoriser des motifs. Un réseau de neurones entièrement connecté, c'est-à-dire où chaque neurone a accès à la sortie de tous les autres, peut y servir de mémoire tolérante aux fautes, le mauvais fonctionnement d'un neurone pouvant ne pas être fatal, de par la propriété de généralisation du réseau.

Implémentation matérielle

Si l'homme veut un jour pouvoir créer des machines aussi intelligentes que lui, tel que le prévoit (avec un certain optimisme) Moravec [167] pour 2020, la réalisation de systèmes intelligents doit proposer des solutions extrêmement rapides. Un réseau de neurones, qu'il soit inspiré du modèle standard ou à impulsions, présente un parallélisme intrinsèque, de par sa structure cellulaire. Or, le fonctionnement séquentiel d'un microprocesseur n'est pas optimisé pour le traitement des tâches parallèles. Une implémentation matérielle peut alors tirer parti de cette caractéristique de manière à grandement améliorer les performances des réseaux de neurones.

Nous pouvons distinguer plusieurs types d'implémentation : analogique, digitale fixée dans le silicium, ou digitale implémentée sur FPGA.

Analogique Le monde vivant est analogique. Les potentiels électriques qui circulent dans les axones de nos neurones sont effectivement continus, et c'est donc tout naturellement que les circuits analogiques ont été choisis par certains, comme Eberhardt [63] ou Someya [221], pour l'implémentation de réseaux de neurones. Eberhardt a créé une plateforme où les neurones sont implémentés avec des composants analogiques (des capacités stockent les poids synaptiques), l'apprentissage étant laissé à un processeur relié au système. Alors que ce circuit est destiné à des neurones standards, Someya a réalisé un design à base de neurones à impulsions.

Digital ASIC De nombreux circuits spécifiques à un type de neurones particulier ont été créés. Certains, tels celui de Shibata [212, 213], réalisent physiquement chaque neurone, ainsi que leurs interconnexions. D'autres, comme Schoenauer [214], sont composés d'un seul neurone, et d'une logique servant à multiplexer l'usage du neurone. Ses poids synaptiques sont mis à jour en fonction du neurone à évaluer. Ce type d'implémentation est rendue possible pour les neurones à impulsions, de par leur faible taux d'activation[6]. En effet, un neurone n'est que rarement activé, et Schoenauer en tire parti grâce à une liste d'activations permettant de décider quel est le neurone à mettre à jour.

---

[6] A chaque pas de temps, la probabilité qu'un neurone soit actif est très faible.

Digital FPGA Pour des raisons évidentes, les FPGAs sont d'excellentes alternatives aux ASICs, le prototypage y étant rapide et peu coûteux. De nombreuses implémentations, dont bon nombre sont décrites dans [162], ont déjà vu le jour. Nous n'entrerons pas dans les détails de celles-ci, mais pouvons résumer les caractéristiques des différentes approches.

Premièrement, le type de neurone et la manière dont les données sont échangées sont de la première importance. Les neurones standards, avec lesquels Eldredge implémente un algorithme de backpropagation [65, 66], travaillent avec des données sur plusieurs bits. La communication inter-neuronale se fait donc à travers des bus ou par multiplexage des données. Les neurones à impulsions, comme utilisés par Upegui [246] ou Schäfer [208], offrent l'avantage de ne nécessiter qu'une ligne de donnée en sortie d'un neurone, une impulsion étant caractérisée par un passage à '1' de cette ligne.

Deuxièmement, nous pouvons différencier les réalisations à base d'un multiplexage de temps, qui ne contiennent pas un neurone physique par neurone du réseau. Ces implémentations nécessitent entre autre un système de contrôle et de stockage des poids synaptiques, ce qui peut être efficace en terme de temps de calcul, mais pas en ce qui concerne une implémentation cellulaire.

Troisièmement, rares sont les implémentations d'un apprentissage on-chip. La taille des neurones étant le facteur le plus limitatif pour une implémentation matérielle, beaucoup de chercheurs effectuent l'apprentissage à l'aide d'un processeur externe, n'utilisant le réseau que pour la phase d'exploitation. Il est bien évident qu'un apprentissage on-chip est nettement plus bio-inspiré, et bon candidat à une implémentation cellulaire, autonome, et auto-réparable, où un organe central n'est pas apprécié.

Finalement, l'apprentissage, à la manière du vivant, comme suggéré par Perez-Uribe dans [184], devrait ne pas agir seulement sur les poids synaptiques, mais également sur la structure du réseau, par l'ajout et la suppression de neurones et de synapses. Dès lors, un système matériel ayant cette caractéristique de création de chemins de données semble le pas suivant à franchir dans le design de réseaux de neurones matériels.

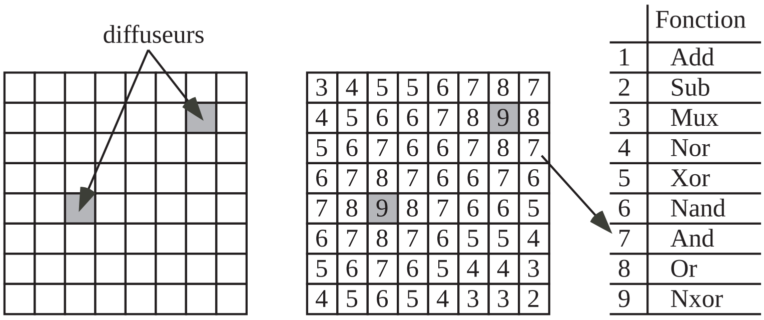
3.4 Combinaisons

Les trois axes que nous venons de survoler peuvent évidemment être combinés. Observons sans plus attendre les implications de telles combinaisons dans le cadre des implémentations matérielles.

3.4.1 PO

L'analyse des axes phylogénétiques et ontogénétiques des êtres vivants est en pleine évolution, les récentes découvertes des gènes organisateurs influant grandement sur les théories évolutionnistes [102]. Une nouvelle discipline, appelée evo-devo [79], est d'ailleurs née de la combinaison de ces deux axes.

Les systèmes artificiels se doivent de prendre le même chemin, car comme nous l'avons expliqué précédemment, la manière dont un individu est créé à partir de son génome influe grandement sur l'efficacité de l'algorithme génétique y relatif. Le codage morphogénétique, introduit par Roggen et Floreano [193] en est un exemple. Un système cellulaire sert de base à la création d'un individu. Des émetteurs, dont l'emplacement dans le substrat est défini par le génome, sont la base d'un gradient chimique qui décroît avec la distance à la source (Figure 3.11). Plusieurs gradients peuvent être considérés (typiquement quatre), et servent ensuite à la cellule lors de la phase de différenciation, pour accéder le bon gène. Le grand avantage de cette approche est la scalabilité, le génome n'étant pas (ou peu)[7] influencé par la taille du tableau de cellules, et dans plusieurs cas les performances sont meilleures qu'un codage direct.



3	4	5	5	6	7	8	7
4	5	6	6	7	8	9	8
5	6	7	6	6	7	8	7
6	7	8	7	6	6	7	6
7	8	9	8	7	6	6	5
6	7	8	7	6	5	5	4
5	6	7	6	5	4	4	3
4	5	6	5	4	3	3	2

	Fonction
1	Add
2	Sub
3	Mux
4	Nor
5	Xor
6	Nand
7	And
8	Or
9	Nxor

Figure 3.11 : *Gradient chimique propagé par le codage morphogénétique. La fonctionnalité de la cellule est ensuite définie par une table, qui constitue une partie du génome.*

La CAM-brain machine de de Garis [55, 57] est un autre exemple de système PO. Il s'agit d'un réseau de neurones implémentés sur FPGAs, dont la structure est évoluée. Le génome stocke la manière dont le réseau se crée, et un automate cellulaire est en charge de l'ontogenèse. Nous plaçons cette machine dans la combinaison PO, car, bien que basée sur des neurones, ceux-ci ne sont pas capables d'apprentissage. Haddow, Tufte, et Remortel, ont, quant à eux, utilisé des L-systèmes pour réduire la taille du génome, dans une application en matériel évolutif [87].

De manière générale, les systèmes PO matériels doivent être capables d'exécuter un algorithme évolutionniste, et présenter une forme de croissance, réduisant ainsi la taille du génome, ou d'autoréparation, offrant au système la propriété de tolérance aux pannes. Alors que de nombreuses approches liées à la réduction de la taille du génome ont vu le jour, les systèmes auto-réparables de type PO ne sont pas légion.

3.4.2 PE

Certains réseaux de neurones artificiels ne possèdent pas de capacité d'apprentissage, les chercheurs préférant utiliser des algorithmes génétiques pour la modification des poids synaptiques ou de la topologie du réseau (par exemple, les Block-Based Neural Networks de Moon [164]). Nous ne pouvons toutefois pas arguer qu'il s'agisse d'un système PE, l'apprentissage standard étant inexistant.

Pour pouvoir créer des systèmes PEs, nous devons faire appel à des réseaux de neurones permettant l'implémentation d'algorithmes d'apprentissage, et qui sont donc des systèmes adaptables. L'évolution peut y apporter un plus en modifiant la structure du réseau, les poids synaptiques initiaux d'un individu, ou les règles d'apprentissage. Un excellent résumé des différents travaux allant dans ce sens peut être trouvé dans l'article de Xin [261], et nous pouvons citer ici deux exemples, [14] et [210]. Dans le

---

[7] La position des émetteurs est stockée dans le génome, mais n'en représente qu'une faible partie.

cas de la robotique évolutive, de tels systèmes ont déjà montré leur avantage sur ceux ne mettant en jeu qu'un des deux mécanismes [71].

Parmi les systèmes tirant parti de ces deux axes, nous pouvons citer l'algorithme SOS, développé par Mesot [153]. Des machines à états finis sont adaptées grâce à un algorithme d'apprentissage par renforcement et un algorithme génétique. L'algorithme génétique y fournit des configurations de départ qui servent de base à l'apprentissage, et la combinaison des deux approches offre de meilleurs résultats que l'utilisation des méthodes séparées.

3.4.3 OE

Les réseaux de neurones sont intrinsèquement tolérants aux pannes, de par leur propriété de généralisation. Toutefois, leur structure cellulaire permet l'ajout d'autoréparation au système, de la même manière que pour une application PO.

Une deuxième manière de relier les deux axes est de considérer la neurogenèse comme faisant partie de l'axe ontogénétique. Peu de réseaux de neurones artificiels font intervenir des topologies variables par croissance ou destruction de neurones [27, 184, 231]. Toutefois, cette propriété, que nous retrouvons dans le règne animal, devrait apporter de nouvelles solutions aux systèmes d'apprentissage. Un mécanisme de gestion de croissance et destruction doit donc être mis en place au niveau matériel, afin que le réseau puisse se modifier en fonction de ses interactions avec l'environnement.

3.4.4 POE

Il n'existe à l'heure de l'écriture de ces lignes, aucun système matériel connu combinant les trois axes. La réalisation d'un tel système devra donc mettre en jeu de l'évolution, de l'apprentissage, de la croissance, et de la tolérance aux pannes, le tout dans un même circuit électronique. Il devrait sans doute s'agir d'un réseau de neurones capables d'apprentissage, et dont la topologie pourrait être modifiable par un algorithme génétique ou par les neurones eux-mêmes. Un mécanisme de croissance et de tolérance aux pannes au niveau cellulaire devrait également y être présent, pour que le réseau puisse se construire de manière autonome, sur la base d'un génome, et qu'un neurone puisse être remplacé par une cellule en attente dans le cas de la présence de matériel défectueux.

La topologie du réseau doit pouvoir varier durant son fonctionnement, et ce de manière non centralisée. Les neurones doivent être capables de créer de nouvelles cellules, et d'initier de nouvelles connexions. Pour ce faire, l'utilisation d'un circuit programmable capable de modifier sa connectique de manière autonome est nécessaire. Outre une fonctionnalité universelle, il doit offrir un système de routage géré par les cellules. De plus, les cellules doivent pouvoir y modifier leur comportement de façon autonome, afin que les cellules totipotentes puissent se différencier, en fonction des tâches à effectuer, et notamment lors de processus d'autoréparation.

3.5 Le projet POEtic

En symbiose parfaite avec ces principes de bio-inspiration, le projet européen POEtic, dans le cadre duquel la présente thèse a été développée, avait pour but principal la réalisation d’un “tissu POEtic”, électronique, capable d’implémenter des systèmes cellulaires mettant en jeu des mécanismes phylogénétiques, ontogénétiques, et épigénétiques, séparément ou non. Les trois axes n’ayant pas encore été réunis sur un même substrat électronique, le défi était donc d’y parvenir, grâce à l’implication de biologistes, mathématiciens, informaticiens et électroniciens.

3.5.1 Nomenclature

Avant de poursuivre, nous désirons tout d’abord clarifier quelques nuances dans les définitions des termes utilisant les lettres POE :

- **Le Modèle POE**, tel que nous l’avons décrit à la page 36, a pour but d’analyser les systèmes bio-inspirés selon les trois axes Phylogénétique, Ontogénétique, et Epigénétique.
- **Le Tissu POEtic** est une réalisation matérielle d’un système impliquant ces trois axes. Défini dans le cadre de notre projet, il s’agit de l’implémentation d’un système cellulaire capable de croissance, d’autoréparation, d’apprentissage, et d’évolution [8].
- **L’Architecture POEtic** correspond, quant à elle, à la structure d’une cellule d’un organisme multicellulaire impliqué dans le tissu POEtic. Cette organisation, que nous allons détailler, est composée de trois niveaux : génotypique, de configuration, et phénotypique.
- **Le Circuit POEtic** correspond au circuit électronique qui a été développé dans le cadre de cette thèse, qui est reconfigurable et capable d’accueillir des applications bio-inspirées. Lorsqu’un organisme multicellulaire organisé selon les trois couches de l’architecture POEtic y est implémenté, et que de l’évolution y prend place, nous obtenons un tissu POEtic.

3.5.2 Architecture POEtic

Notre tissu POEtic s’est développé comme un système cellulaire permettant notamment d’exprimer de la croissance et de l’autoréparation. Un organisme y est constitué de plusieurs cellules, où chacune contient un génome décrivant l’ensemble de l’organisme. Les cellules, qui, au démarrage du système, sont totipotentes, se voient attribuer une tâche particulière, en fonction de la croissance de l’organisme. Pour ce faire, une architecture cellulaire a été définie, sur trois niveaux présentés à la figure 3.12 : le Genotype, le Mapping, et le Phénotype. Ces trois niveaux correspondent aux trois axes P, O, et E.

Nous tenons ici à mettre en avant le fait que l’architecture a pour but d’être implémentée en matériel. En effet, le but de stocker le génome décrivant l’entièreté de l’organisme est à terme de fournir les bases de l’autoréparation à un tel système, où, à l’instar d’Embryonique, une cellule pourrait prendre la place d’une cellule défectueuse. De ce fait, le fonctionnement global de l’organisme pourrait perdurer, les mécanismes implémentés dans les cellules étant capables de gérer des erreurs par elles-mêmes.

L’architecture POEtic sert donc à réaliser des systèmes matériels possédant des capacités de croissance, et potentiellement d’autoréparation. L’apprentissage peut y

---

[8] Dans la littérature, le tissu POEtic fait référence au circuit POEtic, mais nous préférons faire la séparation entre les deux concepts.

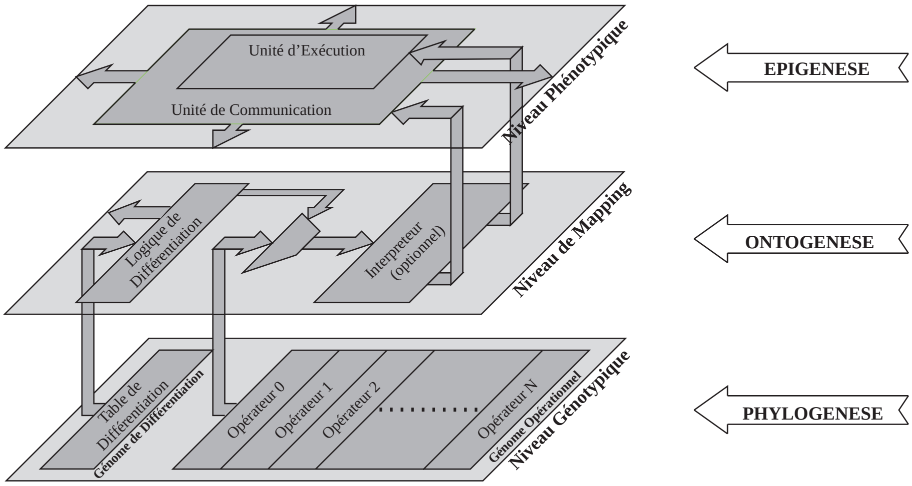


Figure 3.12 : *Les trois niveaux organisationnels d'une cellule du tissu POEtic.*

être ajouté en définissant de manière judicieuse l'unité d'exécution de la cellule, laissant le système totalement distribué dans les cellules. En revanche, les mécanismes d'évolution nécessitent un contrôle global en charge de gérer une population d'individus, de charger ces individus, et de récupérer l'évaluation de leur fitness. Dès lors, le tissu POEtic devra être composé d'un contrôleur pour la gestion de l'évolution, et d'une grille de cellules conçues en accord avec les trois niveaux que nous décrivons maintenant en détail.

Phénotype

Le phénotype correspond à la fonctionnalité d'une cellule lorsque l'organisme a terminé sa croissance. Toutes les fonctionnalités requises doivent être présentes dans chacune des cellules, de manière à ce que n'importe laquelle d'entre elles puisse exprimer n'importe laquelle des fonctions mises à disposition pour une application particulière. Le phénotype de l'organisme, outre la fonctionnalité des cellules, est également défini par les connexions intercellulaires. Pour ce faire, un mécanisme, exécuté par une unité de communication, doit être capable de connecter de manière adéquate une cellule à d'autres, et ce, soit de manière prédéfinie, soit en fonction du gène exprimé.

L'épigenèse est implémenté par le niveau phénotypique, étant donné qu'il s'agit d'un mécanisme d'apprentissage au niveau de l'individu. Un tel organisme doit donc posséder des cellules, du type neurones artificiels, ayant des capacités d'apprentissage. Cet apprentissage peut alors avoir lieu dans l'unité d'exécution, qui peut modifier son comportement en fonction de l'environnement de l'organisme, et également dans l'unité de communication, qui peut créer de nouvelles connexions durant la vie de l'organisme.

Génotype

Le phénotype est créé en fonction du développement de l'organisme, et de son génome. Ce dernier est en charge de stocker toutes les informations nécessaires à la définition d'un organisme. Les gènes qui le composent sont ensuite exprimés lors de la phase de différenciation des cellules. Le génome peut être représenté de nombreuses manières, en fonction de la structure du phénotype et du type de système de croissance et de différenciation. Citons-en deux ici.

Premièrement, il peut ne s'agir que d'un tableau de gènes, qui sont sélectionnés par le niveau de mapping, et directement chargés pour exprimer le phénotype. Chaque gène, composé d'un certain nombre de bits, est donc la définition immédiate d'une fonctionnalité. Lors des processus d'évolution, cette table est directement modifiée par l'application des mutations et des crossing-over.

Deuxièmement, il peut être réparti en deux tables, comme c'est le cas pour le codage morphogénétique (cf. page 60). La première contient l'ensemble des fonctionnalités mises à disposition pour une application particulière. Chaque ligne y contient un opérateur, comme dans la figure 3.11, page 61, qui sert ensuite au phénotype pour effectuer son traitement. Une deuxième table, dite table de différenciation, est alors nécessaire. Lorsque la cellule se différencie, elle fait correspondre son état de différenciation à un index de la table d'opérateurs. Dans le cas du codage morphogénétique, cette table fournit un index en fonction de la quantité de chaque gradient. L'évolution n'est ici appliquée qu'à la table de différenciation, celle des opérateurs étant définie une fois pour toutes lors de la mise au point du système (cf. Prototype PO, page 230).

Mapping

Entre les niveaux génotypiques et phénotypiques, le mapping est en charge d'exprimer le bon gène, en fonction de la croissance de l'organisme, et de configurer correctement la couche phénotypique. Les mécanismes de croissance et de différenciation sont donc implémentés dans ce niveau, et peuvent revêtir différentes formes. Un simple identifiant peut être calculé en fonction des cellules voisines, un organisme étant composé de cellules numérotées de 1 à n. Cet identifiant peut alors permettre d'accéder au gène correspondant et de l'exprimer. Le codage morphogénétique, en revanche, fait intervenir des mécanismes plus complexes, où la couche de mapping gère la propagation des gradients, ainsi que l'accès aux deux tables du génome. Dans ce cas, où le génome est séparé en deux tables, nous distinguons la logique de différenciation, qui sélectionne une valeur dans la table de différenciation, en fonction de la manière dont la cellule s'est différenciée, et l'interpréteur, qui charge le bon opérateur dans le niveau phénotypique.

Nous pouvons noter qu'à l'instar de l'unité de connexion, le niveau de mapping doit également être capable de communiquer avec d'autres cellules, via la logique de différenciation, de façon à construire l'organisme de manière autonome. Outre le développement d'un organisme, d'autres approches peuvent également être implémentées dans ce niveau, notamment pour y inclure des mécanismes d'autoréparation.

3.5.3 Pourquoi un nouveau circuit ?

Le tissu POEtic, défini comme un système multicellulaire, qui peut être évolué, peut apprendre, et où une cellule possède une structure à trois niveaux comme nous l'avons décrit, nécessite, pour une réalisation matérielle, un circuit électronique ayant plusieurs caractéristiques.

1. Il doit être autonome, et donc ne pas nécessiter de contrôleur externe.
2. Il doit être assez général pour implémenter diverses applications, une cellule devant être capable d'effectuer d'importe quel type de traitement. Pour ce faire,

nous devons prévoir un système matériel reconfigurable offrant assez de généralité.

3. Il doit posséder un moyen d'implémenter des mécanismes évolutionnistes. Un algorithme génétique doit donc y être réalisable sans difficulté.
4. Le processus d'évolution doit pouvoir accéder rapidement au système reconfigurable, pour y charger des individus dans le but de calculer leur fitness.
5. Les cellules, implémentées dans le système reconfigurable, doivent pouvoir créer des connexions de manière autonome. Les travaux sur le routage distribué, présentés au chapitre précédent, ont donc été intégrés au développement du circuit.
6. Le système doit être scalable, et plusieurs circuits doivent pouvoir être reliés ensemble, de manière à offrir un grand nombre de cellules potentielles.

Toutes ces contraintes nous ont guidés lors de la réalisation du circuit POEtic, et plus particulièrement pour l'élaboration de la partie reconfigurable. Nos solutions à la création de connexions seront présentées au chapitre 5, et le circuit en lui-même le sera au chapitre 6. Enfin, les différents mécanismes POE qui y ont été implémentés seront abordés au chapitre 7.

Chapitre 4

Le routage au fil des ans

> Va toujours par le chemin le plus court, et le plus court est le chemin tracé par la nature.
>
> Marcus Aurelius ANTONINUS (MARC-AURÈLE)

LE ROUTAGE tel que nous l'avons déjà évoqué dans le chapitre présentant les circuits reconfigurables consiste en la génération d'une manière de relier différents points d'un graphe. A partir d'une liste de points ou coordonnées à relier, un algorithme de routage doit pouvoir fournir un ensemble de chemins permettant de les connecter. De plus, des contraintes telles que la longueur maximale d'un chemin ou le nombre de virages maximal peuvent y être ajoutées, en fonction du problème à résoudre.

Dans ce chapitre, nous allons tout d'abord présenter quelques exemples de tous les jours (ou presque) dans lesquels le routage joue un rôle central. Nous partirons ensuite des solutions proposées par la nature puis détaillerons les approches algorithmiques qui furent développées durant les cinquante dernières années, et ce dans un ordre chronologique. Nous verrons ensuite les différents systèmes qui furent proposés dans l'optique d'accélérer le traitement du routage, en commençant par les coprocesseurs, et en finissant par des systèmes purement matériels, qui nous amèneront tout naturellement aux travaux de cette thèse qui seront présentés dans le chapitre suivant.

4.1 Pourquoi un plus court chemin ?

La question peut sembler inutile. Il est vrai qu'à part un touriste se promenant dans une ville sans but précis, ou un processus de recherche aléatoire ne visant pas de point défini, les exemples sont plutôt rares où les trajectoires ne sont pas destinées à directement relier deux points. Lorsqu'un taxi vous emmène à l'aéroport, pour autant que le chauffeur soit honnête, il empruntera le chemin le plus court afin de rendre le temps de la course (et donc le coût) le plus faible possible. De manière générale, nos déplacements sont guidés par la minimisation du chemin à parcourir, l'homme étant toujours à la recherche du moindre effort. Sur le plan technique, les réseaux de communication de type Internet sont créés sur la base de protocoles visant à réduire le temps de transfert de l'information entre deux points du globe. Des algorithmes de routage dynamique sont ici nécessaires à l'envoi de paquets au travers de routeurs, dans un réseau global tolérant aux pannes.

La réalisation de systèmes digitaux de type VLSI (Very Large Scale Integrated Circuits) ou de cartes PCB nécessite également la recherche du plus court chemin. En effet, relier deux transistors par un fil de métal ou deux circuits par un fil de cuivre implique un délai d'autant plus grand que le fil est long. De plus, la tâche de relier tous les points donnés est dite NP-dure (cf. page 45), c'est-à-dire qu'elle est non triviale et qu'une méthode déterministe ne peut lui trouver une solution en un temps polynomial. Notons donc ici la distinction entre la recherche du plus court chemin et le problème de routage : ce dernier nécessite de réaliser plusieurs connexions entre différents points, et peut dès lors utiliser la recherche du plus court chemin dans la génération de chaque liaison.

Nous allons poursuivre en présentant quelques solutions, naturelles, puis algorithmiques, au problème de la recherche du plus court chemin, puis nous continuerons avec le problème plus général du routage. Nous ne traiterons pas des systèmes de routage dynamique de type réseaux de communications, notre approche étant plutôt basée sur une grille régulière d'éléments. En revanche nous accorderons une attention particulière à toutes les solutions qui furent développées dans le cadre de l'optimisation des outils de conception de circuits.

4.2 Aparté naturel

La nature s'efforce toujours de dépenser le moins d'énergie possible, ce qui la mène tout naturellement à chercher le chemin le plus court dans diverses circonstances. Les animaux, tels les fourmis, ont, au cours de l'évolution, développé divers systèmes leur permettant d'économiser leur énergie en empruntant des chemins les plus courts possibles. Les bulles de savon optent pour des formes minimisant la pression présente sur leur membrane, et offrent un moyen de trouver la membrane de taille minimum reliant plusieurs points. Un gaz lâché en un point d'un labyrinthe s'étendra de manière régulière jusqu'à la sortie. En remontant alors le gradient chimique, il est possible de reconstituer le plus court chemin de la source à la sortie.

Ces exemples naturels sont autant de sources d'inspiration pour les chercheurs travaillant sur des problèmes nécessitant l'optimisation d'un chemin entre plusieurs points. La présente thèse ayant été grandement influencée par ces phénomènes naturels, nous les présentons brièvement ici.

4.2.1 Les fourmis

La plupart des colonies de fourmis sont en quête perpétuelle de nourriture. Des éclaireuses partent à la recherche de nouvelles sources, et lorsqu'elles en ont trouvé une, reviennent au nid, déposant au passage des phéromones. Ces phéromones, hormones olfactives, servent aux ouvrières à retrouver la nourriture fraîchement découverte. En passant sur ce chemin, elles déposeront également des phéromones, accentuant la force du chemin. En conséquence, plus un tracé est utilisé, plus il sera marqué de phéromones, et plus les fourmis auront tendance à suivre cette piste olfactive forte, et donc à suivre leurs congénères en direction de leur nourriture. De plus, lorsqu'un chemin existant se trouve bloqué par un événement quelconque, les fourmis sont capables de trouver, toujours selon la même méthode, un détour par lequel elles peuvent contourner l'obstacle.

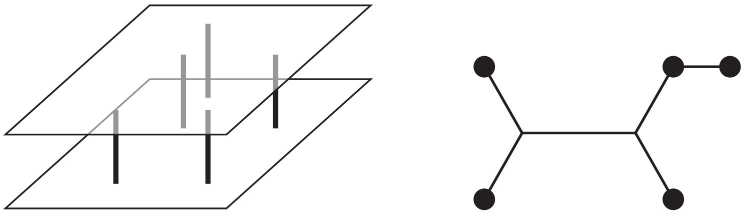
Basée sur ce principe, l'"Ant Colony Optimisation" (ACO), introduite en 1991 par Colorni, Dorigo, et Maniezzo [43], permet de résoudre des problèmes NP-complets, tels que celui du voyageur de commerce. Cependant, la méthode développée par les fourmis pour trouver le chemin le plus court semble difficilement adaptable aux systèmes digitaux, c'est pourquoi nous allons, dans les deux sections suivantes, explorer les bulles de savon, les sons, et l'expansion des gaz.

4.2.2 Les bulles de savon

Les bulles de savons sont un bel exemple de connexions de points grâce à un réseau minimal [76, 77, 109]. En 1892 déjà, Boyz présentait à un auditoire ses travaux sur les bulles de savon :

> "Quelle que soit la complication de la carcasse, on ne voit jamais plus de trois membranes se couper le long d'une crête, ou de quatre arêtes ou de six membranes se rencontrer en un point. [...] Pendant la formation des bulles, on voit parfois un nombre trop grand de membranes se rencontrer en un point ou le long d'une arête, mais l'une d'elles glisse aussitôt et l'ensemble redevient stable. Dans tous ces systèmes, les plans qui passent par une même arête s'y rencontrent sous des angles égaux." [29](pp. 47-48)

Ces quelques phrases, qui peuvent sembler quelque peu abruptes, définissent les propriétés essentielles des membranes d'eau savonneuse. Lorsque plusieurs bulles de savon sont accolées, elles s'appareillent de manière à minimiser la tension superficielle de leur membrane. Les bords extérieurs sont toujours des cercles, et les parois intérieurs séparant deux membranes se croisent toujours selon des angles égaux. Ce phénomène peut être exploité pour trouver un chemin de taille totale minimale entre plusieurs points (un arbre de Steiner, cf. page 86).



(a) Deux planches reliées par des bâtonnets

(b) La membrane d'eau savonneuse reliant les bâtonnets, vue de dessus

Figure 4.1 : *Deux planches reliées par des clous et la membrane connectant ces bâtonnets.*

La procédure est comme suit. Prenez une paroi transparente et collez-y des bâtonnets sur une de ces faces. Placez ensuite une autre paroi à l'autre extrémité des bâtonnets de manière à ce que les bâtonnets touchent ladite paroi (Figure 4.1(a)). Trempez la construction dans de l'eau savonneuse et retirez-en la. Vous verrez alors une membrane d'eau savonneuse reliant tous les bâtonnets (Figure 4.1(b)). Il est très intéressant de noter que la membrane ainsi formée correspond au chemin minimum permettant de relier les différents bâtonnets[1]. La raison en est la minimisation de l'énergie contenue dans la membrane. Un fait remarquable est qu'à chaque intersection de trois pans de membrane, les angles les séparant sont exactement de 120 degrés, toujours pour des raison d'énergie.

Dès lors il semble être prometteur de tester des architectures de type hexagonales permettant justement la création de liaisons à 120 degrés.

4.2.3 Le gaz, le son, la lumière

Un gaz lâché en un point d'un espace quelconque, s'étendra dans cet espace de manière uniforme, à la manière d'un son. Cette "exploration" est similaire à un front d'onde (sonore ou lumineuse), qui avance dans l'espace. A un temps t, tous les points sur le front d'onde sont à la même distance de l'origine (dans le cas d'un milieu homogène), et la direction opposée à l'expansion indique la direction du plus court chemin menant à la source. La figure 4.2 présente le front d'onde d'un son ou l'expansion d'un gaz, à partir d'une source (disque noir), où pour trouver le chemin le plus court entre la source et un point quelconque (disque blanc), il suffit de remonter perpendiculairement au front.

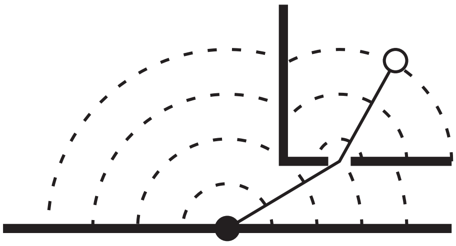


Figure 4.2 : *L'expansion d'un gaz, ou le front d'onde d'un son.*

Cette expansion est similaire à une recherche en largeur d'abord, qui sera explicitée plus loin, et servira de modèle pour la réalisation d'un système de routage dynamique réalisé en matériel. En effet, l'expansion peut être vue comme un processus parallèle, chaque point du front d'onde au temps $t + \Delta t$ étant "calculé" au même instant par les lois de la nature. Le matériel, de par ses possibilités de calcul parallèle, semble donc être en mesure de s'inspirer de ce phénomène pour étendre un "front d'onde" sur une structure cellulaire.

4.2.4 De la ficelle

Avant de passer aux bases théoriques, citons une méthode originale pour résoudre le problème du plus court chemin entre deux villes. Minty, en 1957, propose, dans une note de 12 lignes dans le journal Operations Research [161], une solution à base de ficelle et de clous. Il suffit de prendre un clou par ville, et de les relier par des bouts de ficelle dont la longueur correspond à la distance séparant chaque paire de villes

---

[1] Suivant la configuration des bâtonnets, il peut s'agir d'un minimum local, et un nouveau plongeon dans l'eau savonneuse peut modifier le résultat.

reliées par une route. En tirant ensuite sur les clous qui représentent les villes à relier, le chemin le plus court se trouve tout naturellement en suivant les bouts de ficelle tendus.

Par cette approche, il est également possible de trouver en une fois tous les chemins entre un point (la source) et tous les autres, en suspendant des poids à tous les clous. En tenant le clou de la source, tous les autres clous pendront par une suite de bouts de ficelle représentant le plus court chemin à la source.

4.3 Les Bases théoriques

Le problème du routage consiste en trouver une manière de relier plusieurs points grâce à des liaisons physiques tout en respectant certaines contraintes. Typiquement, dans un espace discrétisé composé d'un ensemble de points P, étant donné une liste $L = \{L_i\}$ de liaisons $L_i = (S_i, D_i)$ à créer avec $S_i \in P$ et $D_i \in P$, un tel algorithme doit retourner une liste de connexions $C = \{C_i\}$ où $C_i = (S_i, P_{i,1}, P_{i,2}, ..., P_{i,j-1}, P_{i,j}, D_i)$. Dans la plupart des cas, les contraintes impliquent qu'un point ne peut appartenir à deux chemins de source différente, comme c'est le cas pour les circuits imprimés. De plus, il est en général intéressant de générer des chemins les plus courts possibles, afin de minimiser les délais sur les fils, ou le temps de parcours dans le cas d'un chauffeur de taxi cherchant à amener un client à bon port.

4.3.1 Arbre de poids minimal

Avant les solutions au plus court chemin, ce fut le problème de la création de l'arbre couvrant de poids minimum (Définition 4.1) qui fut traité par les scientifiques.

Définition 4.1. *L'arbre couvrant minimal (minimum spanning tree) d'un graphe, dont un exemple est montré à la figure 4.3, est l'arbre contenant tous les nœuds (ou terminaux) du graphe, et dont la somme des poids des arêtes est minimale.*

L'arbre couvrant de poids minimum permet de relier plusieurs points par un réseau de poids minimum. Les réseaux électriques, par exemple, en tirent parti, afin de minimiser la quantité de câbles qui servent à interconnecter l'ensemble des points du réseau.

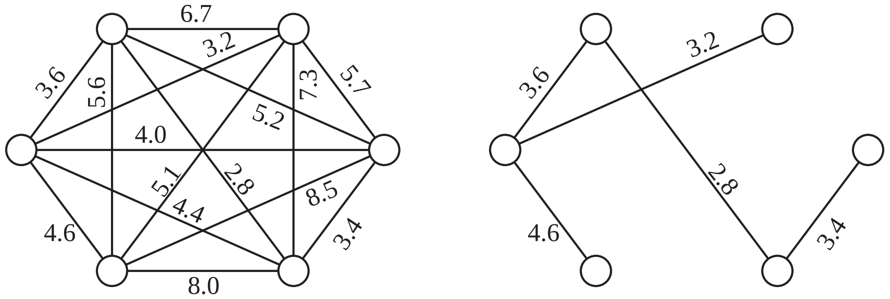


Figure 4.3 : *A droite, l'arbre couvrant minimal du graphe de gauche.*

Kruskal

Kruskal est le premier, en 1956, à proposer une méthode de création de l'arbre couvrant minimal d'un graphe non-orienté [130], et ce dans l'optique de prouver un théorème sur l'unicité de l'arbre couvrant minimal publié en tchèque en 1926 [28]. Partant d'un graphe $G = (V, E)$, où V est un ensemble de nœuds, et E un ensemble d'arêtes, associées à un poids et connectant ces nœuds, il permet de créer l'arbre de poids minimum.

Dans son article, il présente trois algorithmes donnant le même résultat. Nous n'en explicitons qu'un seul, les deux autres faisant appel à une approche très similaire. L'approche, triviale, consiste en la création d'un sous-graphe du graphe initial par ajout successif des arêtes. A chaque pas, une nouvelle arête qui ne crée pas de cycle, et dont le poids est minimum, est ajoutée à la liste finale (Algorithme 4.1). Lorsque toutes les arêtes du graphe initial ont été testées, le graphe final constitue l'arbre de poids minimum.

Algorithme 4.1 Algorithme de Kruskal

```
Entrées : Un graphe $G = (V, E)$, contenant un ensemble de nœuds $V = \{v_x\}$ et un
    ensemble d'arêtes $E$
Résultat : Un arbre $F$ de poids minimum
Classer les arêtes par ordre de longueur croissante $(e_1, e_2, \cdots, e_M)$
$F = \emptyset$
Pour $i = 1, \cdots, M$ Faire
    Si $F \cup \{e_i\}$ est un arbre alors
        poser $F = F \cup \{e_i\}$
    Fin si
Fin faire
```

Le temps d'exécution de l'algorithme de Kruskal est $O(|E|log|V|)$.

Prim

Peu de temps après, en 1957, Prim publie un article [187] où il présente une autre méthode de création de l'arbre couvrant minimal. Avant de présenter son algorithme, nous introduisons quelques termes nécessaire à sa compréhension. Un *terminal isolé* est un terminal qui, à un instant du déroulement de la construction, n'est connecté à aucun autre. Un *fragment* est un ensemble de terminaux interconnectés par des liens directs, et finalement un *fragment isolé* est un fragment qui n'est connecté à aucun terminal externe à lui-même[2].

Son algorithme est basé sur deux principes simples :

- **Principe 1 :** N'importe quel terminal isolé peut être connecté à un plus proche voisin.
- **Principe 2 :** N'importe quel fragment isolé peut être connecté à un plus proche voisin par l'arête disponible la plus courte.

En appliquant ces deux principes judicieusement, l'arbre couvrant minimal est construit très facilement. Au départ, on sélectionne un terminal du graphe, que l'on

---

[2] Si nous avons quatre nœuds a, b, c, d reliés comme ceci : $a - b - c$, avec d comme terminal isolé, alors $a - b$, $b - c$, et $a - b - c$ sont des fragments, et $a - b - c$ est un fragment isolé.

connecte avec le terminal qui lui est relié par l'arête de poids le plus faible. Ce premier lien constitue un fragment isolé. A chaque pas suivant, il suffit de choisir l'arête de poids le plus faible reliant un nœud au fragment isolé existant, et d'ajouter ce nœud au fragment.

La simplicité de l'algorithme en rend l'exécution manuelle possible, et l'auteur va même jusqu'à suggérer qu'il pourrait être utile dans le cas où un message doit être passé à tous les membres d'un organisation secrète sans qu'il ne soit intercepté par l'ennemi. Pour ce faire, une probabilité d'interception est définie entre chaque couple de membres, et sert de poids aux arêtes. L'arbre couvrant minimal permet alors de trouver le moyen de faire passer le message à tous les membres en minimisant le risque d'être pris par l'ennemi.

La formulation initiale de Prim ne fut pas des plus formalisées, mais nous retranscrivons ici sa déclaration formelle (Algorithme 4.2).

Algorithme 4.2 Algorithme de Prim

```
Entrées : Un graphe $G = (V, E)$, contenant un ensemble de nœuds $V = \{v_x\}$ et un
    ensemble d'arêtes $E = \{e_{ij}\}$
Résultat : Forêt $F$ de poids minimum
$F =$ ensemble vide
$S = \{s,$ un sommet de V$\}$
Tant que $S$ différent de $V$ Faire
    Trouver $e_{ij}$ avec un poids minimum entre $v_i \in S$ et $v_j \in V - S$
    $F = F \cup e_{ij}$
    $S = S \cup v_j$
Fin tant que
```

Nous pouvons d'ores et déjà noter que les algorithmes de Kruskal et de Prim ne se prêtent pas instinctivement à une implémentation matérielle décentralisée. En effet, si nous partons d'un système où les nœuds sont des cellules, une vision globale est nécessaire, pour trier correctement les arêtes en fonction de leur poids, ou au moins pour pouvoir comparer tous les poids des arêtes reliées à un nœud.

4.3.2 Plus court chemin

Les algorithme de Kruskal et de Prim ne permettent que de trouver l'arbre couvrant minimal d'un graphe, c'est-à-dire qu'ils connectent forcément tous les nœuds ensemble, dans un même sous-graphe. Bien qu'utiles dans certaines applications, ils ne permettent pas de trouver le chemin le plus court entre deux points du graphe.

La première méthode de résolution de ce problème que nous avons pu trouver est à attribuer à Ford et Fulkerson. En 1956 [72], ils proposent une méthode pour trouver le flot maximal entre deux points dans un graphe non orienté, et constatent que trouver le plus court chemin entre deux points dans un graphe G revient à trouver le flot maximal entre ces deux points dans le graphe dual de G.

Dantzig

Alors que le résultat de Ford et Fulkerson n'est vu que comme une dualité du problème de flot maximal, dans un article de 1957, Dantzig [51] résout directement le plus court chemin, et propose une approche basée sur la résolution, fort joliment illustrée, d'un problème de contraintes, qui peut être résolu grâce à la méthode du simplex [50]. Son algorithme débute par la création d'un arbre quelconque incluant tous les nœuds du graphe. A chaque nœud est assigné une valeur correspondant à sa distance au nœud d'origine. Les arêtes qui ne sont pas présentes dans l'arbre y sont ajoutées petit à petit, si elles font décroître la distance à l'origine des autres points. Lorsqu'une arête est ajoutée, une autre en est enlevée, afin de conserver la structure d'arbre, et le processus continue jusqu'à ce que plus aucune arête ne puisse être ajoutée. Le plus court chemin entre le point d'origine et tous les autres est alors disponible en suivant les arêtes de l'arbre.

Bellman

Une année plus tard, Bellman [22] propose une approche par la programmation dynamique, dont l'application est quasiment identique à celle de Dantzig. Les poids des arêtes sont stockés dans une matrice T, de taille $N \times N$, le poids entre le nœud i et j étant caractérisé par t_{ij}, et le but de l'algorithme est de trouver le chemin le plus court[3] entre le nœud 1 et le nœud N.

Définissons :

$$\begin{array}{rcl} f_i & = & \text{le coût requis pour aller de } i \text{ à } N\text{, pour } i = 1, 2, \cdots, N-1, \\ & & \text{en utilisant une politique optimale,} \\ \text{avec } f_N & = & 0. \end{array} \tag{4.1}$$

Il suffit, pour trouver la solution, d'appliquer itérativement les équations suivantes :

$$\begin{array}{rcl} f_i^{(k+1)} & = & \underset{j \neq i}{Min}[t_{ij} + f_i^{(k+1)}], \quad i = 1, 2, \cdots, N-1, \\ \text{avec } f_N^{(k)} & = & 0, \\ \text{et } f_i^{(0)} & = & \underset{j \neq i}{Min}\, t_{ij}, \quad i = 1, 2, \cdots, N-1 \end{array} \tag{4.2}$$

pour $k = 0, 1, 2, \cdots, N-1$

Le problème de cette méthode, qui procède par approximations successives, est le temps de calcul, qui nécessite $N-1$ pas, où chaque pas nécessite $O(N^2)$ opérations.

Dijkstra

Suivant de près Bellman, Dijkstra fut le second, en 1959, à proposer un algorithme cité par tous, dans [59]. Il y propose une manière de trouver le chemin de poids le plus faible entre deux points d'un graphe non-orienté pondéré dont les arêtes ont un poids positif ou nul. Peut de temps après, en 1960, Whiting et Hillier [253] on présenté le même algorithme, en suggérant que la méthode pouvait aisément être appliquée aux graphes orientés.

Nous avons traduit la partie de l'article de Dijkstra traitant de ce problème dans l'algorithme 4.3. Cette première version littéraire correspond à l'algorithme 4.4, qui présente une facette nettement plus mathématique. Le temps d'exécution de l'algorithme est de $O(|V|^2)$, et peut être réduit à $O(|E|log|V|)$ suivant l'implémentation.

---

[3] Nous utilisons le terme chemin le plus court comme synonyme du chemin le moins coûteux.

Algorithme 4.3 Algorithme du plus court chemin de Dijkstra (version traduite)

On considère n points (nœuds), dont certaines paires sont connectées par une arête ; la longueur de chaque arête est donnée. Nous nous restreignons au cas où il existe au moins un chemin entre n'importe quels deux nœuds.

Problème : Trouver le chemin ayant la distance totale minimale entre deux nœuds P et Q.

Nous utilisons le fait que, si R est un nœud sur le chemin minimal de P à Q, la connaissance de ce dernier implique la connaissance du chemin minimal entre P et R. Dans la solution présentée, les chemins minimaux entre P et n'importe quel autre nœud sont construits par ordre de longueur jusqu'à ce que Q soit atteint.

Les nœuds sont subdivisés en trois ensembles :

- A. Les nœuds pour lesquels le chemin minimum depuis P est connu ; les nœuds seront ajoutés à cet ensemble par ordre de longueur de chemin minimum depuis P ;
- B. Les nœuds candidats à un transfert dans l'ensemble A. Cet ensemble comprend tous les nœuds qui sont connectés à au moins un nœud de l'ensemble A, mais qui ne sont pas dans A eux-mêmes ;
- C. Les autres nœuds.

Les arêtes sont également subdivisés en trois ensembles :

- I. Les arêtes permettant de construire les chemins les plus courts entre P et les nœuds de l'ensemble A ;
- II. Les arêtes candidates à un transfert dans l'ensemble I ; Une et une seule des arêtes de cet ensemble est connectée à chaque nœud de l'ensemble B ;
- III. Les autres arêtes (rejetées ou non encore considérées).

Au départ de l'algorithme, tous les nœuds sont dans l'ensemble C et toutes les arêtes sont dans l'ensemble III. Nous transférons maintenant le nœud P dans l'ensemble A et répétons ensuite les points suivants.

Pas 1. Considérons toutes les arêtes connectées au nœud qui vient d'être transféré dans l'ensemble A avec les nœuds R dans l'ensemble B ou C. Si le nœud R est dans l'ensemble B, nous vérifions si l'arête r crée un chemin plus court de P à R que le chemin connu qui utilise les arêtes de l'ensemble II. Si ce n'est pas le cas, l'arête r est rejetée ; si toutefois l'arête r crée une connexion plus courte entre P et R que celle déjà obtenue, elle remplace la branche correspondante dans l'ensemble II et cette dernière est rejetée. Si le nœud R est dans l'ensemble C, il est ajouté à l'ensemble B et l'arête r est ajoutée à l'ensemble II.

Pas 2. Chaque nœud de l'ensemble B peut être connecté au nœud P par un seul chemin si nous nous restreignons aux arêtes de l'ensemble I et d'une de l'ensemble II. En ce sens, chaque nœud de l'ensemble B connaît sa distance au nœud P ; le nœud ayant la distance minimum à P est transféré de l'ensemble B à l'ensemble A, et l'arête correspondante est transférée de l'ensemble II au I. Nous retournons ensuite au pas 1 et répétons le processus jusqu'à ce que le nœud Q soit transféré dans l'ensemble A. La solution a ensuite été trouvée.

Algorithme 4.4 Algorithme du plus court chemin dans un graphe de Dijkstra

Entrées : Un graphe non-orienté $G = (V, E)$, des coûts c_{ij} sur les arêtes, et deux nœuds v_a et v_b

Résultat : Le plus court chemin entre v_a et v_b

```
$Val(v_j) = c_{aj}$, pour tout $v_j \in V$
$T = V; P = \{v_a\}; Val(v_a) = 0$
Répéter
  Trouver $v_t \in P$ tel que $Val(v_t) < Val(v_i)$, pour tout autre $v_i \in P$
  Si $v_t = v_b$ alors
    Remonter le chemin des prédécesseurs de $v_b$ à $v_a$, et sortir de l'algorithme
  Fin si
  $T = T - \{v_t\}$
  $P = P \cup \{v_t\}$
  Pour tout $v_j \in T$ Faire
    Si $Val(v_t) + c_{tj} < Val(v_j)$ alors
      $Val(v_j) = Val(v_t) + c_{tj}$
      Prédécesseur de $v_j = v_t$
    Fin si
  Fin faire
Jusqu'à ce que T soit vide
Il n'y a pas de chemin entre $v_a$ et $v_b$
```

La figure 4.4 présente l'exécution de l'algorithme de Dijkstra (Algorithme 4.4) pour un graphe simple dans lequel nous voulons trouver le chemin minimal entre le nœud présent en haut à gauche, et celui positionné en bas à droite.

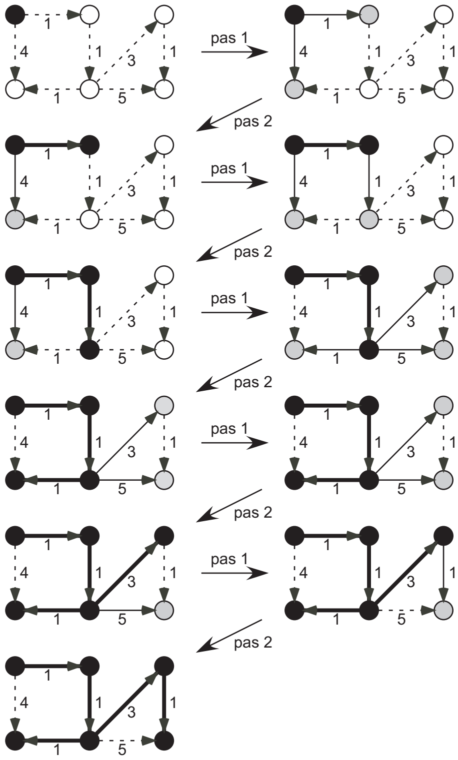


Figure 4.4 : *Exemple d'exécution de l'algorithme de Dijkstra.*

Il est intéressant de noter que Pollack et Wiebenson, dans un article de 1960 [186], attribuent la paternité d'un algorithme quasiment semblable à Minty, en ne citant comme référence qu'une "Personal Communication". Agissant cette fois sur un graphe orienté, il est présenté par l'algorithme 4.5.

Algorithme 4.5 Algorithme du plus court chemin dans un graphe de Minty

```
Entrées : Un graphe orienté $G = (V, E)$, et deux nœuds $v_a$ et $v_b$
Résultat : Le plus court chemin entre $v_a$ et $v_b$
$Val(v_a) = 0$
Tant que $v_b$ n'est pas atteint Faire
    Pour toute arête $e_{ij}$ telle que la valeur de $v_i$ a été définie et la valeur de $v_j$ ne l'a pas été Faire
        Calculer $Val(v_i) + c(e_{ij})$
    Fin faire
    Sélectionner l'arête $e_{ij}$ pour laquelle cette somme est minimale
    $Val(v_j) = Val(v_i) + c(e_{ij})$
Fin tant que
```

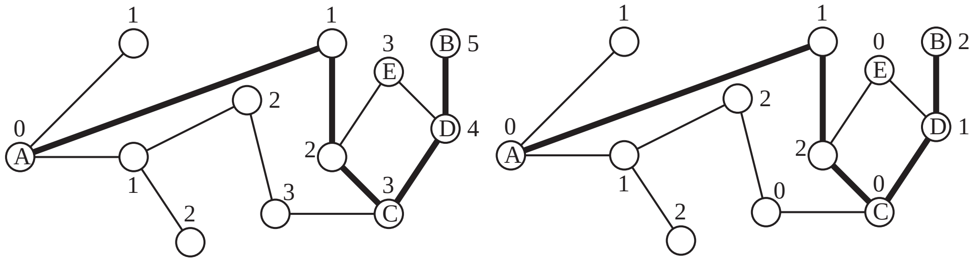
Moore

La même année que Dijkstra, Moore [165] écrit un article dans lequel il présente quatre algorithmes, dont un résolvant le même problème que Dijkstra, mais avec une efficacité moindre. Nous ne réquisitionnerons toutefois pas plus de lignes sur des algorithmes similaires à celui de Dijkstra, pour lequel d'autres optimisations ont été effectuées ultérieurement par différentes équipes [41, 183]. En effet, le routage dans les circuits électroniques, que nous allons entre autre développer dans le chapitre suivant, traite, de manière générale, du plus court chemin entre deux points, et ce dans une grille de points. Ce problème est un cas particulier des trois autres algorithmes de Moore, qui sont appliqués à des graphes non pondérés (dont les poids des arêtes sont identiques). Ils permettent donc de trouver le plus court chemin entre deux points du graphe avec différentes implémentations plus ou moins coûteuses en terme de mémoire et de temps d'exécution.

Algorithme A Pour trouver le chemin le plus court entre un point A et un point B, l'algorithme donne tout d'abord la valeur 0 au point A. Au pas suivant, la valeur 1 est assignée à tous les points reliés au point A. Au pas suivant, tous les points non assignés reliés aux points de valeur 1 se voient assigner la valeur 2. Et ainsi de suite, jusqu'à arriver au point B. La valeur du point B donne sa distance au point A, et il suffit de revenir en arrière, en suivant la décrémentation des numéros jusqu'à arriver à A, pour trouver le chemin le plus court (Algorithme 4.6). Dans le cas où plusieurs chemins sont possibles, comme à partir du point D de la figure 4.5(a), qui peut choisir entre C et E, il suffit d'en prendre un au hasard.

Algorithme 4.6 Algorithme A du plus court chemin de Moore

```
Entrées : Un graphe $G = (V, E)$, et deux nœuds $v_a$ et $v_b$
Résultat : Le plus court chemin entre $v_a$ et $v_b$
$i = 0$
Tant que $v_b$ n'est pas atteint Faire
    Pour tout nœud $v_j$ non visité ayant une arête reliée à un nœud de valeur $i$ Faire
        $Val(v_j) = i + 1$
    Fin faire
    $i = i + 1$
Fin tant que
Parcourir le chemin de $v_b$ à $v_a$
```



(a) Solution de l'algorithme A (b) Solution de l'algorithme B

Figure 4.5 : *Le chemin le plus court (trait gras) entre deux points A et B, dans un graphe non pondéré, par les algorithmes de Moore.*

Algorithme B En 1959, la puissance des ordinateurs n'était pas ce que l'on peut qualifier d'exceptionnelle. Les développeurs devaient donc faire particulièrement attention à la taille de leurs programmes, et surtout à la mémoire nécessaire à leur exécution. Dans cette optique, Moore propose une amélioration de son algorithme : au lieu de stocker un entier pour chaque nœud, il est possible de ne stocker que la valeur de cet entier modulo 3, c'est-à-dire une valeur pouvant être 0, 1, ou 2. De cette manière, seulement 2 bits sont nécessaires pour chaque nœud, et le chemin entre B et A peut être retrouvé en décrémentant les valeurs modulo 3. Sur la figure 4.5(b), nous pouvons observer, par exemple, que depuis le point C, qui a la valeur 0, le point avec valeur 2 sera choisi.

Algorithme C Toujours poussé par le souci de minimiser la taille de la mémoire nécessaire, il va jusqu'à proposer une implémentation ne nécessitant qu'un bit par nœud. Ce bit indique si le nœud a été visité ou non, et est initialement placé à 0 dans tous les nœuds. Le principe est relativement semblable aux approches précédentes, et part du point A, en y plaçant la valeur 1. A chaque pas, tous les nœuds dont la valeur est 0 et qui sont reliés à un nœud de valeur 1 voient leur valeur passer à 1, et ce jusqu'à trouver le point B. La liaison ayant permis d'arriver à B est alors mémorisée comme faisant partie du plus court chemin, et le processus est relancé, pour trouver le plus court chemin entre A et le prédécesseur de B. Au total, le processus est lancé d fois, si la distance entre A et B vaut d. Il est donc nettement plus coûteux en terme de temps d'exécution que les variantes A et B.

Les trois approches de Moore sont intéressantes dans le sens où elles offrent une possibilité de parallélisation par un système synchrone. En effet, les lignes 3-5 de l'algorithme 4.6 peuvent sans autre être exécutées en parallèle.

4.3.3 Algorithme de Lee

Le plus influant des articles sur le sujet des algorithmes de routage est sans nul doute celui de Lee [137], qui est cité par tous ses successeurs. Il y présente une méthode de spécification de problèmes de traitement de patterns, qu'il applique à la création de chemins dans un graphe régulier. Nous allons décrire son formalisme, et nous montrerons que sa solution au plus court chemin est identique à celle proposée par Moore.

Commençons par définir un ensemble de cellules (Lee est le premier à introduire le terme *cellule*) : $C = \{c^1, c^2, \cdots\}$. Pour chaque cellule $c^i \in C$, nous définissons son *1-voisinage* $N(c^i) = \{c^i_1, c^i_2, \cdots, c^i_n\}$, qui doit obéir aux deux règles suivantes :

N1) Tout 1-voisinage a exactement n cellules, où $n \geq 1$.

N2) $c^j \in N(c^i) \Leftrightarrow c^i \in N(c^j)$, qui signifie que le graphe est non-orienté.

Basé sur la fonction N, nous définissons, pour toute cellule c^i ayant un voisinage $N(c^i) = \{c^i_1, c^i_2, \cdots, c^i_n\}$, la fonction $d_k(c^i) = c^i_k$, pour $k = 1, 2, \cdots, n$. Dès lors, $d_k(c^i)$ est la k-ème cellule du 1-voisinage.

Définissons un ensemble fini de symboles $S = \{s^1, s^2, \cdots, s^m\}$.

Définissons un mapping Γ de C dans $C \times S$, qui associe un symbole $s^j \in S$ à chaque cellule $c^i \in C$. Nous devons donc, pour chaque problème particulier, définir $\Gamma(c^i) = (c^i, s(c^i))$, qui peut, dans l'exemple du routage, indiquer si une cellule est libre ou occupée.

Pour deux cellules c^i et c^j distinctes, nous définissons un chemin entre ces deux cellules par $p(c^i, c^j) = \{c^0 = c^i, c^1, c^2, \cdots, c^m = c^j\}$, de manière à ce que $c^{i+1} \in N(c^i)$ pour tout $i = 0, 1, \cdots, m-1$. $\pi(c^i, c^j)$ est alors l'ensemble de tous les chemins entre c^i et c^j.

Définissons une carte d'admission M, du domaine $\pi(c^i, c^j)$ à l'ensemble $\{0, 1\}$. Chaque chemin tel que $M(p(c^i, c^j)) = 1$ est défini comme étant admissible, les autres étant inadmissibles, et l'ensemble des chemins admissibles est appelé $\pi^*(c^i, c^j)$.

Avec toutes ces définitions, nous disposons d'un quintuple (C, S, N, Γ, M), que nous appelons un $\mathcal{C} - space$.

Dans le cas de la création de chemins, qui nous intéresse, nous devons encore définir un vecteur de fonctions $F = (f_1, f_2, \cdots, f_r)$, où chaque fonction f_i est définie de $\pi^*(c^i, c^j)$ dans $\mathbb{N}$. f_i nous donne une fonction de coût du chemin entre deux cellules, et le but de l'algorithme est alors de trouver le chemin de plus bas coût entre deux cellules données.

Pour que l'algorithme fonctionne, il faut que les fonctions f soient monotones, c'est-à-dire que si $p(c^i, c^k)$ est un sous-chemin de $p(c^i, c^j)$, alors $f(p(c^i, c^k)) \leq f(p(c^i, c^j))$.

Durant l'exécution de l'algorithme, une masse cellulaire $(m_1, m_2, \cdots, m_r)$ est associée à chaque cellule. Une masse cellulaire $m = (m_1, m_2, \cdots, m_r)$ sera plus petite que la masse $m' = (m'_1, m'_2, \cdots, m'_r)$ si $m_i = m'_i$ pour $i = 0, 1, \cdots, k$, mais que $m_{k+1} < m'_{k+1}$, avec $0 \leq k \leq r$.

Se basant sur toutes ces définitions, l'algorithme 4.7 présent la solution générale de Lee au problème du routage d'un chemin.

Création de chemin dans une grille Pour le problème de trouver le plus court chemin dans une grille régulière de cellules reliées à leurs quatre voisines, nous posons les définitions suivantes :

Les fonctions de coordonnées d_1, d_2, d_3,et d_4, pour une cellule c^i, sont définies ainsi : $d_1(c^i)$ est la cellule en haut, $d_2(c^i)$ est la cellule de droite, $d_3(c^i)$ est la cellule du bas, et $d_4(c^i)$ est la cellule de gauche.

Si une cellule peut être occupée ou inoccupée, nous pouvons définir M comme ceci : Pour un chemin $p(c^*, c^{**}) = \{c^0 = c^*, c^1, \cdots, c^{n-1}, c^n = c^{**}\}$, $p(c^*, c^{**})$ est admissible, c'est-à-dire, $M(p(c^*, c^{**}) = 1$, si $s(c^i)$ est inoccupée, pour $1 \leq i \leq n$.

Ensuite, le vecteur F doit être défini en fonction du problème à résoudre. Par exemple, pour trouver un chemin minimisant le contact avec des cellules occupées, nous pouvons définir F comme étant une seule fonction f comme ceci :

1) $f(p(c^*, c^*)) = 0$.

2) Si $s(c^i)$ est inoccupée, alors

$$f(p(c^*, c^i)) = \begin{cases} (min\{f(p(c^*, c^j))|c^j \in N(c^i)\}) + R(c^i) \\ \text{pour tout} f(p(c^*, c^j)) \text{ayant été définie,} \\ \text{et est indéfinie sinon} \end{cases} \tag{4.3}$$

Où $R(c^i) =$ le nombre de cellules $c^j \in N(c^i)$ qui sont occupées.

Pour le problème du plus court chemin, nous pouvons définir F comme étant une seule fonction f comme ceci :

1) $f(p(c^*, c^*)) = 0$.

2) Si $s(c^i)$ est inoccupée, alors

Algorithme 4.7 Algorithme de Lee

```
Entrées : $\mathcal{C} - space\ (C, S, N, \Gamma, M)$, un vecteur $F = (f_1, f_2, \cdots, f_r)$, une cellule
    initiale $c^*$ et une cellule finale $c^{**}$
Résultat : Le chemin de coût le plus faible entre $c^*$ et $c^{**}$
L = {c*}  →  $L = \{c^*\}$
Toutes les cellules de $C$ reçoivent une masse $(0, 0, \cdots, 0)$
Tant que $c^{**} \notin L$ ou $L \neq \emptyset$ Faire
    $L1 = \emptyset$
    Pour tout cellule $c \in L$ Faire
        déterminer l'ensemble des cellules admissibles $\{c^i\} \in N(c)$ dont la masse
        n'a pas été déterminée
        Ajouter cet ensemble à L1
    Fin faire
    Pour tout cellule $c^i \in L1$ Faire
        Calculer une masse cellulaire possible, en trouvant parmi les voisins
        $c^j \in N(c^i)$ le r-tuple minimal $(f_1(p(c^*, c^j, c^i)), \cdots, f_r(p(c^*, c^j, c^i)))$
    Fin faire
    Les masses des cellules ayant une masse possible minimum $\{c^j\}$ sont mises à
    jour avec celle-ci
    Pour chacune de ces cellules, la coordonnée d'origine de l'expansion est stockée
    $L = L \cup \{c^j\}$
    Enlever de $L$ les cellules dont toutes les voisines ont une masse calculée
Fin tant que
Si $L = \emptyset$ alors
    Il n'y a pas de solution
Sinon
    Partant de $c^{**}$, parcourir la chaîne de coordonnées jusqu'à atteindre $c^*$
Fin si
```

$$f(p(c^*, c^i)) = \begin{cases} (min\{f(p(c^*, c^j)) | c^j \in N(c^i)\}) + 1 \\ \text{pour tout} f(p(c^*, c^j)) \text{ayant été définie,} \\ \text{et est indéfinie sinon} \end{cases} \tag{4.4}$$

Il est intéressant de noter que Lee, dans son article, ne propose pas directement de solution pour le plus court chemin, mais une pour minimiser la distance totale ET le contact avec des cellules occupées. Pour ce faire, il utilise deux fonctions, dont une est celle de l'équation 4.4. Les lecteurs ont donc extrapolé que le plus court chemin était créé avec cette fonction-ci, et c'est cet algorithme qui est toujours cité. Or, par le mécanisme de son algorithme général, disposer d'une simple fonction $f(p(c^*, c^i)) = 0$ suffit à trouver le plus court chemin entre deux points. Ceci évite de devoir stocker la distance à l'origine de chaque point, ce qui est toujours reproché à Lee, et ne nécessite que de mémoriser le prédécesseur de chaque cellule.

Cette dernière version est cruellement semblable à celle de Moore, mais appliquée à une grille régulière d'éléments connectés à leurs n voisins. Son approche y est, comme nous l'avons vu, nettement plus mathématique, alors que Moore ne le présentait que d'une façon littéraire. L'algorithme de Lee permet en outre de placer des contraintes comme par exemple la minimisation non pas seulement de la distance mais également du nombre de croisements de chemins.

A partir de maintenant, nous nous référons au cas de la recherche du plus court chemin grâce à la fonction 4.4 comme étant l'algorithme de Lee. Appliqué à une grille d'éléments reliés à leurs quatre voisins, il fonctionne à la manière dont une vague se propage à partir de la source, créant une structure en forme de diamant. Dans ce que nous appellerons dorénavant la phase d'expansion, un front d'onde atteint les cellules les unes après les autres et lorsqu'une cellule est touchée par le front d'onde, elle stocke sa distance à la source en incrémentant de 1 la distance de la cellule par laquelle l'expansion est arrivée. La phase d'expansion se termine lorsque la destination est atteinte, signe que le chemin est trouvé. Il ne reste plus alors qu'à remonter jusqu'à la source en passant d'une cellule de distance d à une de distance $d-1$ jusqu'à arriver à la source, qui a une distance de 0 (phase de rétropropagation). La figure 4.6 montre, pour un exemple, les pas de l'algorithme lors de l'expansion, ainsi que le chemin généré.

L'aspect fondamental de l'algorithme de Lee réside en son parallélisme intrinsèque. En effet, dans sa présentation, Lee propose une approche séquentielle, où les cellules sont ajoutées au front d'onde les unes après les autres, mais il est clair qu'avec un système parallèle la phase d'expansion pourrait être plus efficace. Alors que la solution séquentielle a une complexité de $O(d^2)$ pour une distance entre la source et la destination de d, la solution parallèle offre une complexité de $O(d)$.

4.3.4 Variations sur Lee

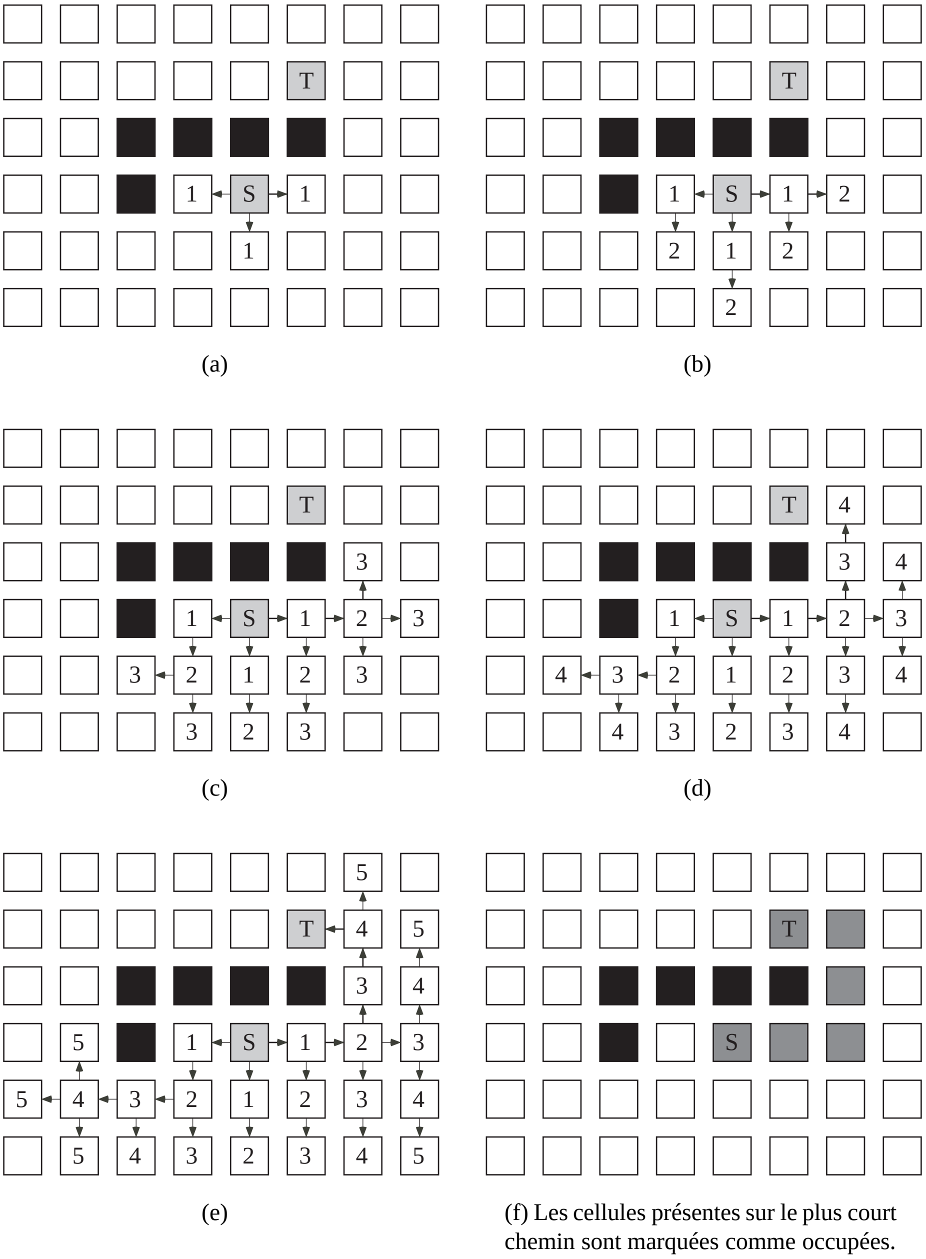
De nombreuses variations sur l'algorithme de Lee ont été proposées. Nous en présentons ici les principales, mais le lecteur pourra en trouver des secondaires dans [42, 92, 93, 94, 128, 163]. La plupart de ces variations, ainsi que l'algorithme principal, ont été implémentées sur des systèmes parallèles dans le but de les optimiser, dont notamment [25, 200, 262].

Akers

En 1967, Akers publie un article [6] où il propose une variante de l'algorithme de Lee, appliquée à une grille rectangulaire de cellules, dans laquelle sont placées des barrières. Au lieu de mémoriser, dans chaque cellule, la distance à l'origine, il lui suffit d'y placer une valeur binaire (Moore pouvait le résoudre avec une valeur ternaire pour le cas d'un graphe non régulier), réduisant ainsi la quantité de mémoire nécessaire à la bonne marche de l'algorithme.

Il part de l'observation suivante : Dans l'algorithme de Lee, une cellule qui a été atteinte par la phase d'expansion et numérotée X, a comme voisines des cellules qui sont soit vides, soit des murs, soit qui contiennent la valeur $X-1$ ou $X+1$. Il suffit donc, pour pouvoir tracer le chemin durant la phase de rétropropagation, que le prédécesseur ait une valeur différente de celle de son successeur, et la séquence $1, 1, 2, 2, 1, 1, 2, 2, \cdots$ remplit parfaitement cette condition.

L'algorithme de Lee est donc adapté de manière à placer la séquence susmentionnée au lieu de la séquence $1, 2, 3, 4, 5, \cdots$ dans les cellules visitées. Il ne reste plus qu'à la phase de rétropropagation de parcourir la séquence dans l'ordre correct, en fonction de la manière $(11, 12, 21$, ou 22) dont l'expansion a atteint le point B (Figure 4.7).



(a) (b) (c) (d) (e) (f) Les cellules présentes sur le plus court chemin sont marquées comme occupées.

Figure 4.6 : *Déroulement de l'algorithme de Lee (S est une source, et T est une destination).*

Rubin

Prenant encore plus de recul, en 1974, Rubin [197] propose des améliorations à l'algorithme de Lee dans l'optique de réduire le temps de calcul. Les deux plus importantes sont les suivantes :

Recherche bidirectionnelle Une solution de Pohl suggère de lancer la phase d'expansion depuis la source ET la destination. De cette manière, le nombre de cellules

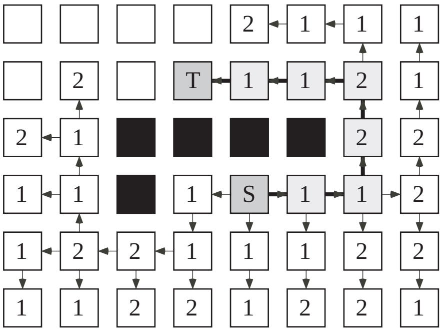


Figure 4.7 : *Le plus court chemin trouvé grâce à l'algorithme de Akers.*

explorées peut être réduit, la somme des cellules visitées par les deux vagues étant plus petite que celle d'une seule vague.

Profondeur d'abord Une autre méthode, développée par Rubin, utilise une recherche en profondeur d'abord. L'expansion se poursuit prioritairement dans la direction de la destination, tant qu'aucun obstacle n'est présent. Le temps d'exécution peut être grandement réduit par cette approche, mais une connaissance globale de la position de la destination est nécessaire, ce qui n'est pas la panacée dans le cas d'une implémentation matérielle.

Hoel

En 1976, Hoel [100] propose deux nouvelles variations à l'algorithme de Lee, pour des coûts de chemins quelconques. La première est purement algorithmique, vouée à une implémentation logicielle, et propose d'utiliser plusieurs listes pour le stockage des cellules présentes sur la frontière, afin d'améliorer la rapidité de la recherche de la prochaine cellule à étendre. La deuxième est très intéressante, et vise à réduire la mémoire nécessaire à l'exécution de l'algorithme. Alors que l'approche de Akers qui plaçait des 1 et des 2 dans les cellules atteintes par l'expansion n'est valable que dans le cas où il n'y a pas de coûts différents entre les cellules, celle de Hoel est applicable à des coûts différenciés. Seuls 2 bits par cellule sont nécessaires, et permettent de définir si la cellule est inoccupée, occupée, ou sur le front, de même que l'origine de l'expansion : occupé=tout droit=0, frontière=virage à gauche=1, inoccupé=2, virage à droite=3. Cette réduction, qui économise un maximum la mémoire, a toutefois un coût. Durant l'exécution de la création d'un chemin, la configuration initiale (cellules inoccupées ou occupées) est perdue, et une mémoire secondaire où le tableau initial (avec les cellules occupées) est stocké est nécessaire. Après chaque création d'un nouveau chemin, il doit être mis à jour avec les nouvelles cellules occupées. Finalement, il est intéressant de noter que cette approche peut être appliquée à un voisinage de 6, sans que des bits supplémentaires ne soient nécessaire (voir l'article [100] pour plus de détails).

Mikami et Tabuchi

En 1968, Mikami et Tabuchi [156] introduisent un nouvel algorithme, appelé line-search, pour le routage d'un circuit imprimé dont un des côtés peut contenir des lignes horizontales, et l'autre des verticales, les deux couches étant reliées par des trous. Leur but est d'offrir une alternative moins coûteuse en terme de temps d'exécution et de mémoire à celui de Lee et de Akers. Au lieu de travailler sur une grille de cellules, l'algorithme stocke des lignes, déterminées par seulement trois coordonnées. La recherche s'effectue alternativement depuis la source et la destination, par l'extension de lignes étendues dans les quatre directions. Les lignes originaires de la source sont comparées à celles originaires de la destination afin de détecter la fin de l'algorithme, lorsque deux lignes se croisent. En outre, l'algorithme exploite les lignes déjà connectées à la source ou à la destination, rendant le mécanisme de liaison, initialement point-à-point, point-à-chemin ou chemin-à-chemin. De cette manière, il évite de créer de nouvelles lignes qui risqueraient de surcharger la carte inutilement.

Cet algorithme garantit de trouver une solution si elle existe, mais ne fournit pas forcément le chemin le plus court entre les deux points, mais le chemin le plus simple, c'est-à-dire avec le moins de coins possibles. Bien que sa structure, telle que présentée dans [156], ne se prête pas à une implémentation parallèle, son concept peut l'être, et nous nous en inspirerons en page 128.

Backtracking

Le routage à la manière de Lee peut conduire à une impasse si trop de chemins doivent être créés. Cette congestion peut être évitée par l'utilisation de backtracking. Agrawal et Breuer, en 1977, introduisent un algorithme basé sur deux principes supplémentaires : le backtracking et la déviation. Les connexions y sont réalisées dans un ordre quelconque, mais après chaque ajout, un test détermine si les connexions restantes sont forcément bloquées par celui-ci. Si c'est le cas, l'ajout est supprimé et un autre chemin est cherché. Dans le cas où aucun des plus courts chemins n'est acceptable, une déviation de taille δ est autorisée, et agrandit le chemin de δ cellules.

Cet algorithme, dont tous les détails peuvent être trouvés dans [5], garantit de trouver une solution au problème général du routage de N chemins, si une telle solution existe. Toutefois, le prix à payer pour une telle réussite est le temps de calcul, qui peut s'avérer prohibitif, de par les nombreuses phases potentielles de backtracking.

Soukup

En 1978, Soukup [223] propose une amélioration de l'algorithme de Lee, qui exploite une recherche en profondeur d'abord, quasiment de la même manière que Rubin. L'expansion se poursuit dans le sens de la destination, tant qu'aucun obstacle n'est rencontré. Lorsque cette expansion de type line-search n'est plus possible, l'algorithme de Lee prend le pas, jusqu'à ce que l'obstacle ait été contourné. Le temps d'exécution peut être grandement réduit par cette approche, mais une connaissance globale de la position de la destination est nécessaire, ce qui n'est pas la panacée dans le cas d'une implémentation matérielle.

Les figures 4.8 et 4.9 illustrent la différence entre l'approche de Soukup et celle de Lee dans le nombre de cellules visitées lors de la recherche de la destination. Dans la figure 4.8, les points noirs ont été atteints par une expansion de type line-search, alors que les blancs l'ont été par une expansion de type Lee.

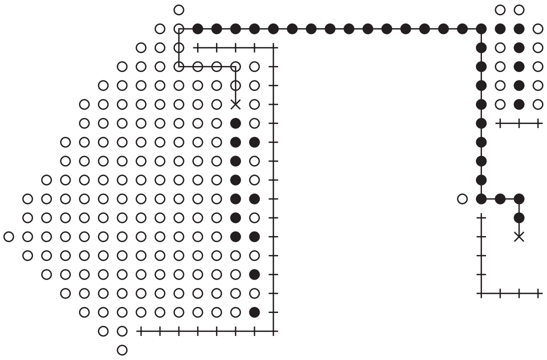


Figure 4.8 : *Nombre de cellules visitées par l'algorithme de Soukup.*

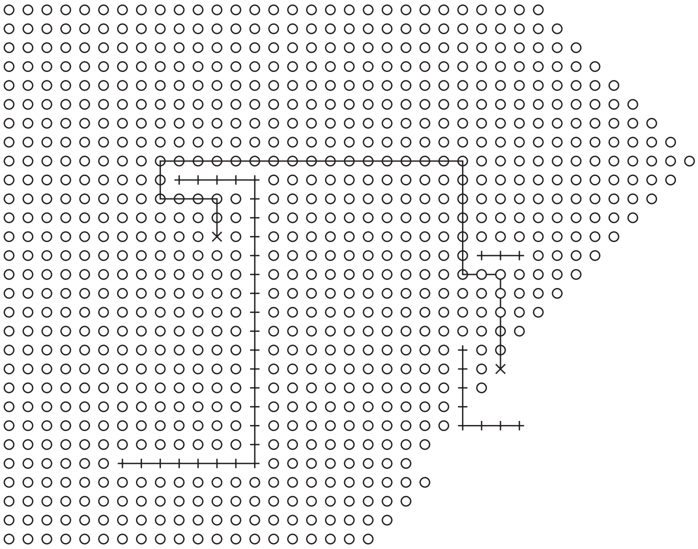


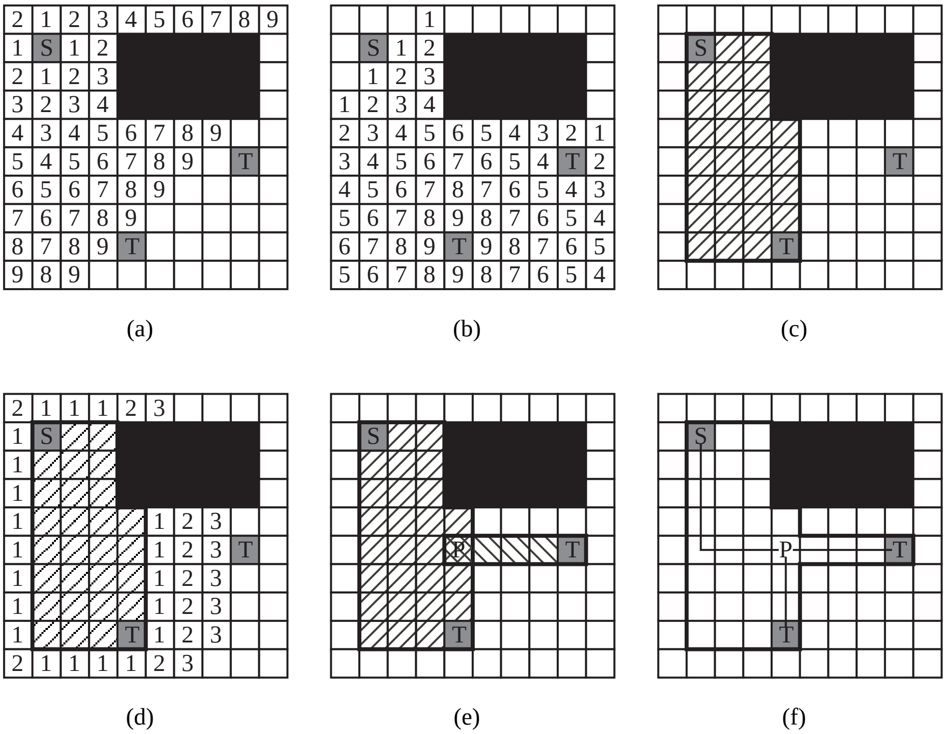
Figure 4.9 : *Nombre de cellules visitées par l'algorithme de Lee, à comparer avec l'approche de Soukup, figure 4.8.*

Watanabe

Bien plus tard, en 1986, Watanabe et Sugiyama [251], présentent un algorithme de routage parallèle, qu'ils appellent PAR, pour Parallel Adaptable Routing. Leur première variante, PAR-1, n'est qu'une adaptation de l'algorithme de Lee, avec comme paramètre supplémentaire la taille de la vague d'expansion. A chaque pas, les cellules présentes sur le front d'onde s'étendent de D_{ex} cellules. Dès lors, si $D_{ex} = 1$, leur algorithme correspond à celui de Lee, et si $D_{ex} = \infty$, il est semblable au line-search de Mikami.

Un deuxième algorithme, PAR-2, présente, lui, une intéressante solution pour les connexions multipoints, où une source doit être connectée à plusieurs destinations. Son but est de construire un arbre de Steiner quasi-minimum (Définition 4.2) entre les différents points, afin de minimiser le nombre de ressources nécessaires. Pour y parvenir, une phase d'expansion part de la source, jusqu'à atteindre une destination (Figure 4.10(a)). Ensuite, une deuxième vague part de ce point pour retrouver la source (Figure 4.10(b)), et permet de définir l'ensemble des plus courts chemins entre les deux points (Figure 4.10(c)). Pour relier le point suivant, une expansion est lancée depuis cet ensemble de points (Figure 4.10(d)), et de la même manière une nouvelle zone est créée, les deux zones étant reliées par un point P (Figure 4.10(e)). Il ne reste alors plus qu'à relier les trois points en passant par ce point P (Figure 4.10(f)). Le procédé peut être réitéré dans le cas où plus de trois points sont à connecter, suivant le même principe.

Définition 4.2. *L'arbre de Steiner de poids minimum d'un graphe connecte tous les nœuds de ce graphe par un arbre de poids minimum, qui peut inclure des noeuds supplémentaires appelés points de Steiner (Figure 4.1(b), page 69).*



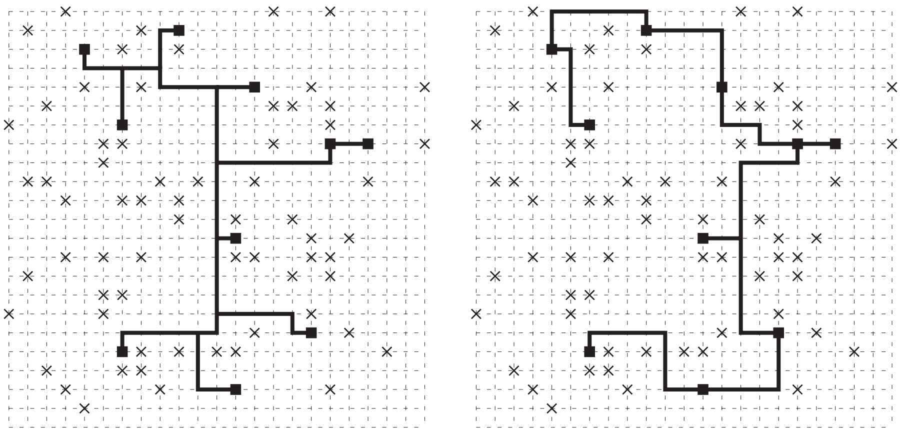
(a) (b) (c)

(d) (e) (f)

Figure 4.10 : *Principe de base de l'algorithme PAR-2 pour la construction d'un arbre de Steiner quasi-minimum.*

La figure 4.11 illustre la différence entre un réseau de connexions réalisé à l'aide de PAR-2, comparé avec celui créé par un algorithme standard.

Une technique de Rip-up a été ajoutée à leur système, et permet de détruire un chemin déjà créé s'il en bloque un nouveau en phase d'expansion. Cette approche de Rip-up a également été exploitée par d'autres équipes [42, 248], et permet de grandement améliorer le nombre de connexions qui peuvent être routées. Nous noterons toutefois que la parallélisation d'une telle technique a un prix, qui est soit celui d'un contrôle global capable de détecter un conflit entre deux chemins, soit celui d'un ajout non négligeable de complexité au niveau des cellules responsables du routage, qui doivent pouvoir détecter un conflit, et détruire un chemin déjà routé.



(a) Algorithme de Watanabe (b) Algorithme standard

Figure 4.11 : *Comparaison de l'algorithme de Watanabe et de l'approche standard.*

Automates cellulaires

Deux équipes ont présenté, en 1996, une solution au routage grâce aux automates cellulaires à deux dimensions. Contrairement aux algorithmes précédemment exposés, la structure d'un automate cellulaire offre ici un parallélisme intrinsèque, qui peut parfaitement être appliqué à l'algorithme de Lee. Ces deux solutions sont basées sur des automates synchrones, où les états de toutes les cellules sont mis à jour simultanément.

Hochberger/Hoffmann La solution proposée par Hochberger et Hoffmann [98] fait intervenir 14 états, dont 4 pour indiquer l'origine de l'expansion, et 4 pour la configuration du chemin entre la source et la destination. Ils proposent également une solution permettant la gestion de la création de plusieurs chemins en parallèle, basée sur des priorités données à ces chemins. Cette amélioration nécessite 24 états, additionnés à deux registres supplémentaires qui mémorisent une appartenance de la cellule à un chemin, ainsi que la priorité de ce chemin.

Adamatzky Adamatzky [3], quant à lui, présente une solution axée sur la recherche du plus court chemin, capable de gérer des coûts entre les cellules de l'automate. Un coût c y induit un retard d'autant de coups d'horloge dans la propagation du signal d'expansion, et le nombre d'états de cette solution dépend du coût maximal autorisé.

4.3.5 A* et algorithmes évolués

Alors que la plupart des algorithmes précédents effectuent une recherche en largeur d'abord, il est possible de chercher en profondeur d'abord, c'est-à-dire, dans un processus séquentiel, d'étendre le dernier nœud visité plutôt que le premier. Sur le plan de l'implémentation logicielle, ceci correspond à utiliser une liste LIFO (Last In First Out) plutôt que FIFO (First In First Out). La recherche best-first utilise une connaissance de la distance potentielle au but pour choisir le nœud à étendre. Il s'agit toujours du nœud qui est potentiellement le plus proche de la destination. Dans le cas d'une grille à deux dimensions, comme pour le problème qui nous intéresse, la distance à la destination est simplement la distance de Manhattan (Définition 4.3) entre le nœud courant et la destination. Cet algorithme garantit de trouver une solution si elle existe, mais elle n'est pas forcément la plus courte.

Définition 4.3. *La distance entre deux points du plan,* $A = (a_x, a_y)$ *et* $B = (b_x, b_y)$*, vaut* $|a_x - b_x| + |a_y + b_y|$.

L'algorithme de recherche A* [89, 199], qui est grandement utilisé en intelligence artificielle, est de type best-first, et tire parti d'un heuristique qui combine le coût $g(n)$ du chemin entre la source et le nœud courant avec le coût potentiel $h(n)$ entre ce nœud et la destination : $f(n) = g(n) + h(n)$. Le nœud ayant un $f(n)$ le plus faible est alors celui qui est étendu. Dans notre cas, $g(n)$ est calculé durant l'expansion, et correspond à la distance minimal entre le noœud et la source, tandi que $h(n)$ est la distance de Manhattan entre le nœud n et la destination. Le chemin trouvé par cet algorithme est toujours optimal si $h(n)$ ne surestime jamais le coût nécessaire à atteindre la destination. L'avantage de cet algorithme est que son nombre d'itérations est grandement réduit par rapport à une recherche en largeur d'abord. Toutefois, il ne se prête pas à une implémentation parallèle, car il nécessite une vision globale nécessaire à l'évaluation de $h(n)$, et que son essence même est séquentielle.

D'autres algorithmes de routage, dont notamment [34, 97, 151, 173], dédiés aux FPGAs, ont également vu le jour après les années 85, date de création des premiers FPGAs. Le problème global y est légèrement différent que pour les ASICs, de par la structure fixe des circuits reconfigurables, et le nombre limité de ressources de routage disponibles. L'optimisation y a pour but de router l'ensemble des connexions demandées, ainsi que de minimiser les délais de propagation. Nous n'étudierons pas en détail ces algorithme, car leur structure ne se prête pas à une implémentation purement distribuée. Ils nécessitent tous, à la manière de A*, une vision globale, qui aide à l'optimisation du routage.

4.4 Approches Matérielles

Un des buts de cette thèse étant d'explorer une implémentation matérielle du routage, nous allons maintenant nous intéresser aux réalisations matérielles de tels systèmes, dont la plupart ont tiré parti du parallélisme intrinsèque de l'algorithme de Lee. À part les deux solutions de Shannon et Rapaport, toutes ont été réalisées dans l'optique du routage des circuits imprimés ou électroniques.

Nous allons présenter plusieurs types de systèmes qui furent développés, pour la grande majorité, dans les années 80 :

- Les coprocesseurs développés pour accélérer certaines parties de l'algorithme de Lee, comme le stockage des listes par exemple.
- Les processeurs spécialisés.
- Les architectures MIMD (Multiple Instruction Multiple Data), où un ensemble de processeurs indépendants, et reliés entre eux, résolvent le problème.
- Les architectures SIMD (Single Instruction Multiple Data), où un ensemble de processeurs simples exécutent le même code au même instant.
- Les architectures purement matérielles, qui nous amèneront tout naturellement vers notre solution.

Avant de poursuivre, citons ici l'approche matérielle de Shannon au problème du labyrinthe, où trouver un point est identique à rechercher un chemin entre une source et une destination. En 1952 [211], il présente une machine mécanique, entre autre composée de 70 relais électriques, capable de résoudre cette tâche dans un labyrinthe de taille 5x5. Elle mémorise le chemin qu'elle a déjà parcouru pour explorer tout l'espace à la recherche du but, grâce à un senseur détectant les murs et le but.

4.4.1 Routage de FPGA

L'approche de DeHon, Huang, et Wawrzynek ne cadre pas dans les catégories que nous venons d'énumérer, mais présente une intéressante réalisation d'un FPGA auto-routable. Les FPGAs, comme nous l'avons vu au chapitre 2, ont des réseaux d'interconnexions qui deviennent de plus en plus complexes. Il serait dès lors appréciable de disposer de tels circuits capables de configurer eux-mêmes leur routage, sans faire appel à un logiciel spécialisé. Dans cette optique, DeHon, Huang, et Wawrzynek [58, 107], ont donc développé un système matériel possédant de tels capacités. Il s'agit de pouvoir router un réseau de type Hierarchical Synchronous Reconfigurable Array (Figure 4.12), qui a la caractéristique de posséder un unique ensemble de switchbox entre une source et une destination. Le routage global devient donc une simple affaire locale, qui peut être gérée par quelques transistors, présents à chaque switch point. Cette approche est très intéressante et prometteuse, mais ne peut malheureusement être appliquée qu'à un réseau de type HSRA, ce qui en limite grandement l'application à d'autres systèmes.

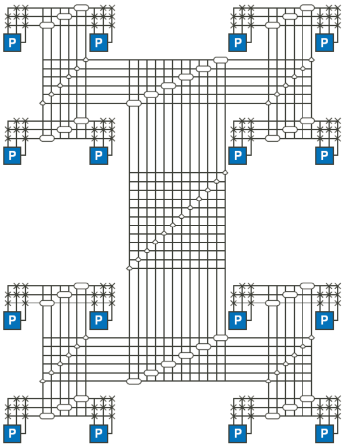


Figure 4.12 : *La topologie d'un tableau HSRA*

4.4.2 Coprocesseur

Une des façons d'accélérer un algorithme de routage est d'augmenter les performances d'un processeur standard en lui ajoutant un coprocesseur capable d'optimiser certaines tâches inhérentes à de tels algorithmes. Damm et Gethöffer [49] offrent un "noyau de routage" capable de gérer les fonctions de coût ainsi que la liste de cellules, qui peut être relié à un processeur pour accélérer le déroulement de l'algorithme. Chan et Schlag [38], quant à eux, ont créé un système de routage distribué sur plusieurs stations de travail, et dont une partie est accélérée par du matériel spécifique, notamment pour le calcul de la fonction de coût.

4.4.3 Processeur

Plus complexe qu'un coprocesseur, plusieurs approches ont proposé l'architecture d'un processeur spécialisé dans le routage. Connecté à un processeur hôte, qui ne s'occupe que de charger la liste des cellules à router, il gère seul l'entièreté du routage d'un circuit. Spiers et Edwards [224] en ont proposé une réalisation basée sur un seul processeur, et Won, Sahni, et El-Ziq [256], ont, eux, réalisé un processeur à trois niveaux de pipeline pour l'accélération de la phase d'expansion. Une autre implémentation pour résoudre le problème du routage sans grille, c'est-à-dire non cellulaire, mais dans le plan réel, a été réalisée par Sato, Kubota et Ohtsuki [206], à l'aide de Content Addressable Memory appliquées à un algorithme de type line-expansion.

4.4.4 SIMD

Alors que les accélérateurs de type coprocesseurs ou processeurs exécutent l'algorithme de routage de manière séquentielle, tout en l'accélérant, les architectures SIMD permettent de profiter du parallélisme matériel. Des éléments de calcul (PE, pour Processing Element) y sont commandés par une unité centrale, et exécutent au même moment les mêmes opérations. Cette approche peut donc grandement améliorer la phase d'expansion de l'algorithme de Lee.

Les deux algorithmes parallèles de Watanabe, par exemple, ont été développés dans le but d'être implémentés sur un réseau de processeurs, l'Adaptive Array Processor (AAP-1) [125, 126, 227]. Il s'agit d'une grille de 256×256 éléments de calcul de un bit, qui semble être la plus grande à avoir été construite. Parmi les autres implémentations que l'on peut trouver, dont [4] [26] [131] [228] [250] , peu travaillent avec un PE par cellule. En effet, la taille des circuits à router dépasse facilement 256×256 (taille de l'AAP-1), et donc le nombre de PEs pourrait devenir ingérable. Dès lors, certains utilisent une grille toroïdale, où un PE peut servir à plusieurs cellules [228], d'autres exécutent un routage grossier, qu'ils affinent ensuite [131], et d'autres encore utilisent un tableau virtuel, dont des parties sont chargées dans les PEs [26].

4.4.5 MIMD

Plus générale que l'architecture SIMD, les MIMD sont composés de petits processeurs, qui au lieu d'exécuter tous le même code au même moment, sont indépendants. De nombreuses implémentations ont vu le jour, ainsi que plusieurs manières de gérer un grand tableau de cellules [105] [120] [138] [174] [229][202][195]. Certaines ont une architecture en arbre binaire, où les processeurs sont organisés de façon hiérarchique, d'autres forment une grille. L'avantage de l'approche MIMD est qu'elle peut tirer partie des architectures déjà réalisées pour résoudre d'autres problèmes. Toutefois, la non possession d'un tel réseau d'ordinateurs peut impliquer un investissement nettement plus important que la réalisation de matériel spécialisé.

4.4.6 Analogique

Rapaport

Rapaport et Abramson [189], en 1959, proposent, quant à eux, une machine analogique capable de résoudre le problème du plus court chemin dans un graphe non-orienté dont les poids ne sont pas forcément égaux. Leur système est basé sur des timers, qui sont placés sur les arêtes du graphe, avec un seuil correspondant au poids de l'arête. Le processus démarre de la source, qui lance les timers des arêtes lui étant connectées, et lorsqu'un timer arrive à son terme, il allume une lumière et lance les timers reliés à son nœud d'arrivée. De cette manière, l'arbre des chemins les plus courts est construit depuis la source, jusqu'à arriver à la destination, qui bloque le mécanisme.

Leur approche est élégante, et permet de résoudre le problème pour un graphe de taille quelconque, en reliant entre eux plusieurs de leurs prototypes.

Cheng

Un routeur totalement analogique est proposé par Cheng, Tanaka, et Yamada [40], en 1991. L'idée est de pouvoir y router un chemin dans une grille rectangulaire de cellules, et de le faire en considérant les liaisons entre voisines comme des résistances. Au centre d'une cellule, une source de courant peut être imposée, avec un potentiel positif ou négatif. Il suffit alors de placer un haut voltage pour la source, et un faible pour la destination, de laisser le réseau se stabiliser, et de parcourir la distribution de potentiel du plus grand au plus petit.

L'implémentation physique est réalisée à l'aide de transistors, qui permettent de bloquer certaines liaisons, et de placer des sources et des destinations. Le blocage de cellules se fait en imposant un haut voltage, obligeant ainsi le potentiel à contourner l'obstacle. Ce système analogique trouve donc un chemin entre une source et une destination, mais ne trouve pas forcément le chemin le plus court, étant donné que plusieurs chemins possibles sont exploitables par une descente de gradient.

4.4.7 Tableau de cellules pour la réalisation de circuits

A notre connaissance, quatre réalisations matérielles de l'algorithme de Lee sous forme de tableau de cellules existent à ce jour, les trois premières datant du début des années 80, et la quatrième du début du millénaire. Elles sont toutes quatre basées sur une implémentation parallèle de l'algorithme de Lee, où les cellules stockent l'origine de l'expansion sur 2 ou 4 bits.

Breuer

En 1981, Breuer et Shamsa [30] proposent l'architecture d'une L-machine, une implémentation de l'algorithme de Lee basée sur des L-Cellules matérielles (Figure 4.13). Chaque cellule est composée de 7 bascules JK et de 75 portes logiques, communique directement avec ses quatre voisines, et est adressable par une unité de contrôle global. Trois des bascules codent l'état de la cellule parmi cinq, et les quatre restantes mémorisent l'origine de la phase d'expansion, en codage 1 parmi 4.

En 1981, la taille des circuits intégrés n'était pas celle d'aujourd'hui. Breuer et Shamsa présentent donc également une méthode visant à relier plusieurs circuits entre eux. Certains y ont des interfaces sur les quatre côtés, d'autres sur trois, et d'autres encore sur deux, une interface y servant à connecter deux circuits entre eux. Dans le premier cas, pour un circuit contenant $n \times n$ cellules, le nombre de pins est $2log_2(n) + 4n + 8$, et pour le cas d'un circuit fonctionnant seul, il est de $4log_2(n) + 8$.

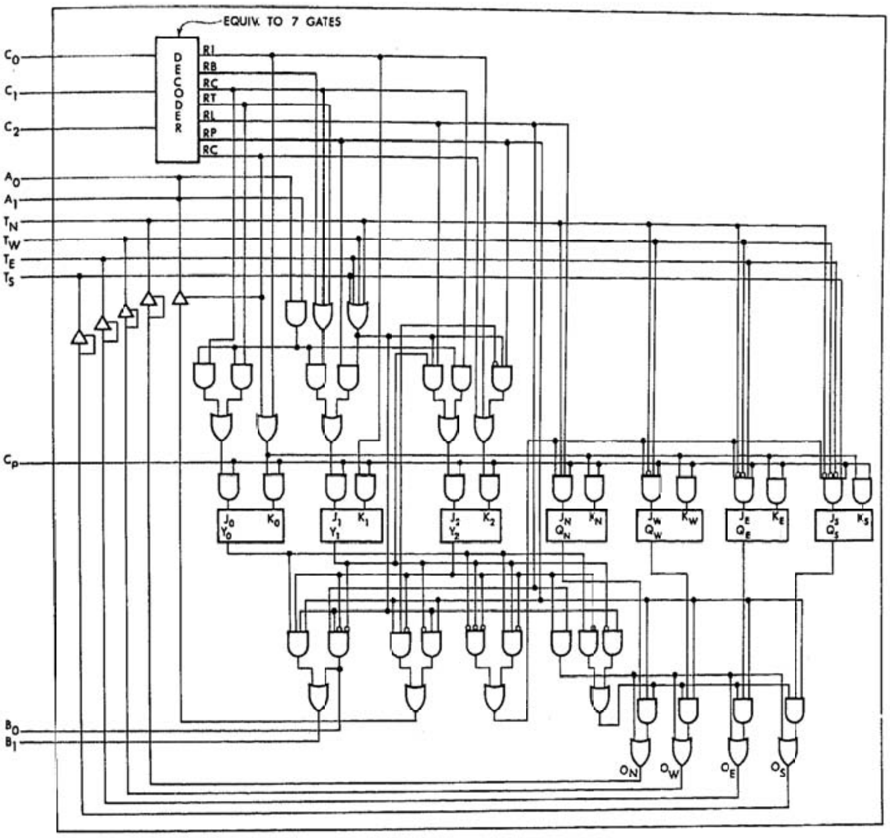


Figure 4.13 : *Implémentation d'une L-cellule de Breuer.*

Iosupovicz

Quasi simultanément, Iosupovicz [108] développe son implémentation, faite également d'une unité de contrôle, d'un tableau de cellules, et d'interfaces pour interconnecter plusieurs circuits. Sa cellule contient 6 bascules et 34 portes logiques[4] (Figure 4.14). Quatre des bascules définissent l'état de la cellule, et les deux dernières définissent l'origine de l'expansion (chez Breuer il s'agit d'un codage en 1 parmi M, qui nécessite plus de bascules, mais moins de logique additionnelle). La différence entre les deux implémentations réside dans l'interfaçage entre circuits. Iosupovicz propose une amélioration au modèle de Breuer, en diminuant le nombre de pins nécessaires. Pour que quatre circuits puissent être connectés à un autre, ce dernier ne nécessite que $4log_2(n-1)+14$ pins au lieu de $2log_2(n)+4n+8$. Ceci implique toutefois une communication plus complexe, où seuls les changements d'états des cellules de bord sont envoyés aux circuits voisins. L'exécution peut donc être ralentie, lorsque plusieurs cellules changent d'état sur le même bord.

---

[4] Ce nombre de 34 portes logiques est tiré de l'article, mais il semble clair, au vue de la figure 4.14, qu'il a été calculé en considérant qu'une porte à 5 entrée est identique à une à 2 entrées. Il faut donc revoir cette évaluation à la hausse, rendant le nombre de portes logiques relativement équivalent aux autres implémentations.

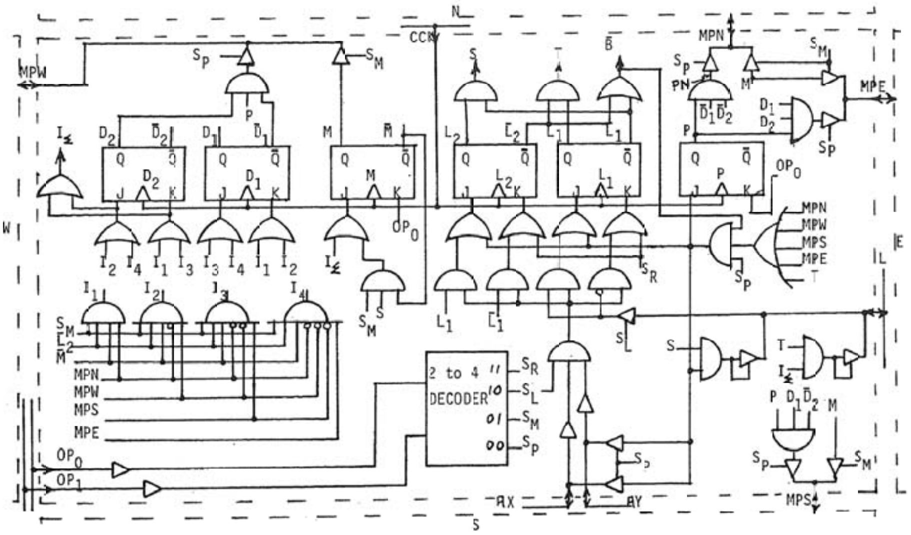


Figure 4.14 : *Implémentation d'une cellule de Iosupovicz.*

Ryan

L'approche prise par Ryan et Rodgers [201], en 1987, vise à router un système sur deux couches, où une cellule a cinq voisines, dont une est sa correspondante sur la couche opposée. Une cellule y est composée de 6 bascules et de 71 portes logiques (Figure 4.15).

La grande différence, en comparaison des autres approches, est leur manière de traiter des tableaux de grande taille. Ici, un système de fenêtre est utilisé, un processeur hôte devant envoyer les fenêtres les unes après les autres au système, tout en récupérant l'état de la fenêtre qui vient d'être calculée. Ceci se fait en décalant les données dans le tableau de cellules, qui est vu comme un grand registre à décalage.

Un routage d'un tableau de $qn \times qn$ points, dans un circuit physique de taille $n \times n$, se fait tout d'abord sur un système où une cellule physique correspond à un tableau de cellules virtuelles de taille $q \times q$. Ces cellules représentent la connectivité grossière du système, et un chemin peut y être trouvé. Ensuite, pour chaque cellule grossière sur le chemin à créer, le tableau de cellules fines correspondant est routé indépendamment, une cellule physique correspondant alors à une cellule réelle.

Nestor

Finalement, la quatrième réalisation est due à Nestor [175, 176], qui, en 2002, implémente un algorithme de Lee pour un routage multicouche, sur FPGA. Une cellule (Figure 4.16) est ici composée d'un séquenceur faisant office de machine d'états (8 états ne nécessitant pas d'autres bits de mémorisation pour l'origine), et d'un registre à décalage qui contient l'état des cellules sur les différentes couches. De cette manière, les couches sont mises à jour à tour de rôle, en multiplexant le temps d'utilisation de la cellule. Concernant l'implémentation sur un FPGA de Xilinx, le XC2S300E, une cellule nécessite 27 look-up tables à 4 entrées, en utilisant trois de ces LUTs comme des registres à décalage.

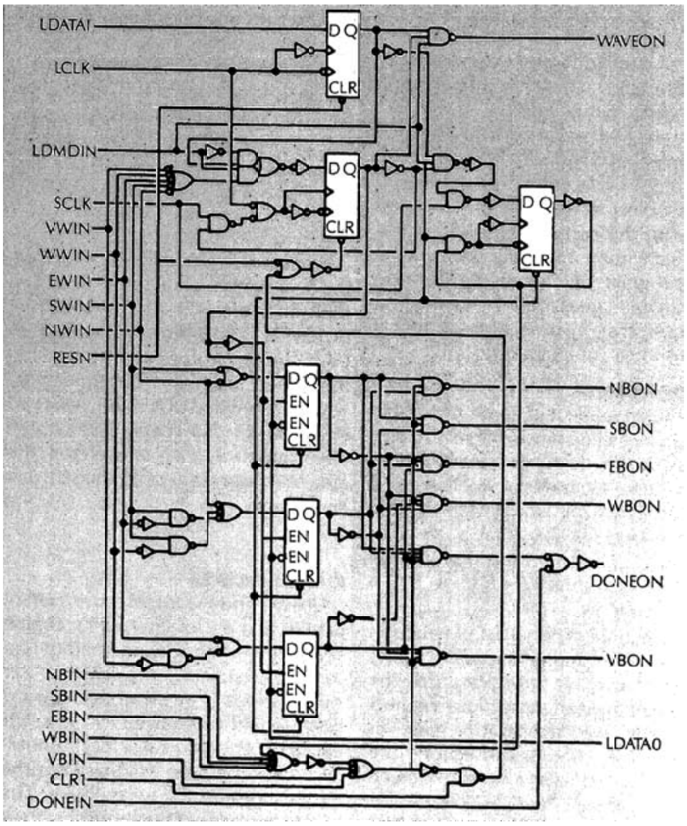


Figure 4.15 : *Implémentation d'une cellule de Ryan.*

4.4.8 Tableau de cellules pour du matériel bio-inspiré

Toutes les implémentations que nous avons passées en revue sont basées sur le fait qu'une cellule est soit occupée, soit inoccupée, et sont toutes dédiées à l'optimisation du routage de circuits imprimés ou intégrés. Les cellules sont reliées entre elles par un graphe dont les arêtes ne sont pas orientées.

Dans le cadre de systèmes bio-inspirés, l'idée est d'offrir plus d'adaptabilité aux circuits électroniques. Dans cette optique, Moreno [169] propose un mécanisme capable de créer des chemins de données, en configurant dynamiquement des blocs de multiplexeurs, dans une structure semblable à celle exploitée par Ercal et Lee [68]. Le but y est de coupler les cellules de routage avec des éléments qui doivent pouvoir se transmettre des informations, et dont les connexions ne sont pas définies a priori, mais peuvent l'être durant le fonctionnement du système.

Une cellule, que Moreno appelle unité de routage, n'est autre qu'un contrôleur qui devrait, pour une réalisation réelle, être relié à un switchbox composé de cinq multiplexeurs (Figure 4.17) : quatre pour sélectionner la valeur à envoyer à chacune de ses voisines, et un pour la valeur à transmettre à un élément lui étant connecté. L'unité de routage peut être implémentée grâce à 47 LUTs d'un FPGA Virtex-E de Xilinx, et de 5 bascules. Le mécanisme de routage étant basé sur l'algorithme de Lee, deux d'entre elles servent à stocker l'origine de l'expansion, et les trois autres permettent de gérer le processus.

Pour son bon fonctionnement, une unité de routage possède quinze entrées, qui

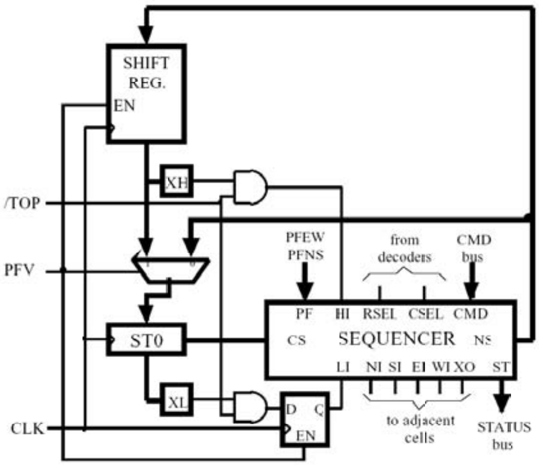
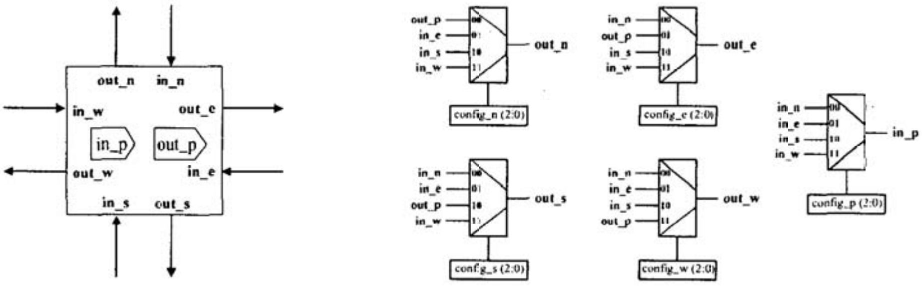


Figure 4.16 : *Implémentation d'une cellule de Nestor.*



(a) Entrées/sorties de la cellule

(b) Cinq multiplexeurs sélectionnent les valeurs à envoyer aux voisins

Figure 4.17 : *Une cellule de Moreno.*

définissent la configuration existante des multiplexeurs. Chacun d'eux possède des bits indiquant le signal à sélectionner (Figure 4.17(b)), et un autre en charge de préciser si le multiplexeur fait déjà partie d'un chemin existant où non. Le processus de routage doit effectivement prendre garde à ne pas effacer un chemin préexistant, mais peut réutiliser un tel chemin partant de la source qui cherche à être connectée.

Chaque unité de routage envoie des signaux à chacune de ses quatre voisines, via trois lignes directionnelles. Ce nombre élevé de connexions pourrait toutefois aisément être réduit, afin d'éviter des problèmes liés au nombre de pins des circuits électroniques.

4.5 Conclusion

Le problème du routage de circuits imprimés et intégrés a, nous venons de le voir, suscité le développement d'un nombre non négligeable d'algorithmes et d'implémentations matérielles. Les années 80 ont vu le nombre d'accélérateurs matériel croître de manière fulgurante, les circuits à router devenant de plus en plus gros, et la vitesse des processeurs de l'époque n'étant pas à la hauteur de telles applications.

Toutes ces implémentations, qu'elles soient prévues pour des cartes à simple, double, ou multi-couche, visent à relier des points par des chemins, sans toutefois que ceux-ci ne soient directionnels. L'espace y est divisé en cellules, qui peuvent être occupées par un chemin existant, ou inoccupées, et donc candidates potentielles au passage d'un nouveau chemin. Notre intérêt se situe, quant à lui, dans la génération de chemins de données dirigés, basés sur des multiplexeurs responsables de router les signaux d'une source à une destination, au travers d'une grille de cellules.

Seul le système présenté par Moreno, spécialement conçu pour des applications bio-inspirées, traite de chemins directionnels. Son algorithme est semblable à celui de Lee, qui fut implémenté pour la réalisation de circuits imprimés, avec pour nuance la directionnalité des liens entre cellules. Nous pouvons cependant émettre trois remarques concernant cette implémentation.

Premièrement, la cellule de routage a été simplifiée au maximum, puisqu'elle ne contient qu'un petit contrôleur auquel il faut fournir la configuration courante des multiplexeurs. Il serait plus judicieux de considérer les multiplexeurs comme faisant partie intégrante de la cellule de routage.

Deuxièmement, le nombre de liaisons inter-unités de routage est trop important. Dans le cas de la réalisation d'un système multi-circuits, pour des circuits contenant un tableau de $n \times n$ unités de routage, pas moins de $24n$ pins seraient nécessaires au bon fonctionnement de l'algorithme, ce qui est loin d'être négligeable.

Troisièmement, comme dans toutes les implémentations que nous avons passées en revue, un contrôle global doit être présent afin de définir quelles sont les sources et destinations à connecter. Dans le cas de systèmes distribués, il serait plus judicieux qu'ils ne nécessitent pas un tel contrôle.

Dans le chapitre suivant, nous allons donc proposer un système où les cellules sont maîtres de leur destin, et ont la possibilité d'initier elles-mêmes des connexions, et ce de manière totalement distribuée.

Chapitre 5

Le routage distribué

> On a tort d'apprendre aux enfants que tous les problèmes n'ont qu'une et une seule solution...
>
> Alice FERNEY , *Extrait de Libération - 20 janvier 2001*

L'ENSEMBLE des solutions de routage matériel présentées dans le chapitre précédent possèdent une même caractéristique, à savoir qu'un contrôleur global est toujours présent. Dans l'optique de réaliser des systèmes cellulaires autonomes, nous nous proposons d'introduire des algorithmes distribués autonomes, ne nécessitant aucun contrôle global.

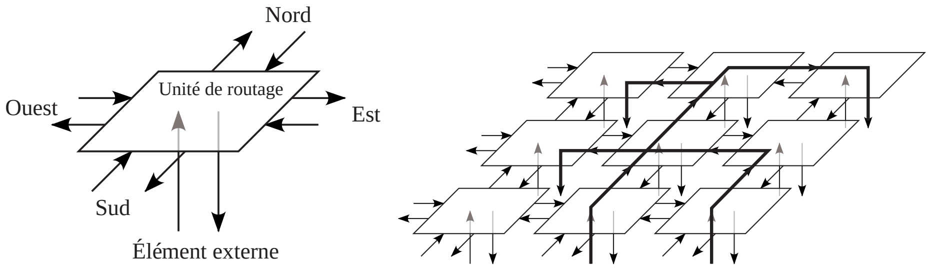
Dans ce chapitre nous présentons nos solutions au problème du routage appliqué aux circuits électroniques reconfigurables. Nous commençons par poser le concept général de notre système, les principes directeurs qui nous ont dirigés durant la mise au point de nos algorithmes, puis nous introduisons des hypothèses structurelles quant à son architecture. Nous proposons quelques solutions préliminaires, qui nous amènent à la première version de nos algorithmes, HIDRA, que nous présentons alors en détail. Suivent trois autres versions, HIDRA-RC, HIDRA-RT et HIDRA-RTC, ayant pour caractéristique respective de réduire le risque de congestion, le temps d'exécution ou une combinaison des deux. HIDRA-L, un algorithme fonctionnant uniquement avec des communications locales est alors présenté, comme alternative scalable aux quatre premiers algorithmes. Alors que ces implémentations sont basées sur des unités de routage reliées à leurs 4 voisines, nous proposons ensuite d'explorer des voisinages de 3, 6 et 8, et de les comparer sur le plan de l'efficacité en terme de congestion et de nombre de transistors requis. Finalement les expériences réalisées sur les différents algorithmes et voisinages sont présentées et les résultats analysés en détail.

5.1 Concept

Notre but est d'inclure un mécanisme de routage dynamiquement configurable dans un circuit reconfigurable. Dans les FPGAs commerciaux, le routage est calculé par un logiciel dédié, puis chargé dans le circuit lors de la configuration. Il n'est ensuite plus possible de le modifier, si ce n'est par une intervention externe sous la forme d'une reconfiguration. Les systèmes de processeurs parallèles utilisent fréquemment un mécanisme appelé “wormhole” [177], qui consiste à envoyer des paquets d'information au travers du réseau de processeurs, en connaissant la coordonnée exacte du destinataire. Chaque processeur possède donc un système de gestion du routage, qui fait transiter des parties de paquets dans la bonne direction, et évite tant que faire se peut les problèmes de congestion. L'implémentation de tels mécanismes nécessite cependant des piles capables de stocker momentanément des parties de paquets lorsque le réseau se congestionne, et donc leur réalisation purement matérielle peut être d'une certaine importance en terme de nombre de transistors (une implémentation en a été faite dans [127], qui comporte 15K transistors pour une unité de routage avec un voisinage de 4). De même, les systèmes de *paquet switching*, qui permettent d'adresser des unités de routage par le biais d'identifiants, nécessitent des unités de routage possédant des tables de routage. Or, dans le cas d'une implémentation matérielle, ces tables impliquent la réquisition d'une grande quantité de ressource.

Alors que le routage des FPGAs standards est statique, et que le type de routage wormhole ou packet switching est totalement dynamique, dans le sens où l'information est transmise dans une direction choisie dynamiquement par une unité de routage, nous proposons une solution intermédiaire, pseudo-dynamique. Des unités de routage y sont responsables de transmettre de l'information entre différents points du circuit. Ces unités peuvent être configurées grâce à un algorithme de routage, qui prend en compte les désirs des éléments connectés aux unités de routage, et lorsque le routage est effectué, les unités servent de transmetteurs statiques. Nous offrons donc la possibilité de modifier dynamiquement la structure du réseau, et la transmission d'information se fait ensuite toujours de la même manière, jusqu'à une modification ultérieure.

La figure 5.1(a) montre une unité de routage, qui dans notre cas est reliée à ses quatre voisines, ainsi qu'à un élément externe, qui n'est autre qu'une partie du circuit. Dans le cas de la réalisation du circuit POEtic, cet élément, relativement semblables aux éléments de base des FPGAs standards, est notamment composé d'une look-up table de 16 bits et d'une bascule. La figure 5.1(b) illustre le fait que la grille des unités de routage est utilisée pour faire transiter de l'information entre différents endroits du circuit.



(a) Unité de routage

(b) Les unités de routage transmettent de l'information au travers du circuit

Figure 5.1 : *Une unité de routage et un réseau de celles-ci.*

Durant tout ce chapitre nous utiliserons le terme “source” pour désigner une unité de routage chargée d'envoyer une information de la part d'un élément externe, au travers de la grille des unités de routage. Par analogie, une “destination” signifiera une unité de routage recevant cette information, et la transmettant à un élément externe. Il est également important de noter que les liaisons que nous désirons créer ne sont pas binaires. Une source peut y être connectée à plusieurs destinations, comme c'est le cas dans les réseaux de neurones, où la sortie d'une cellule est récupérée par un grand nombre d'autres cellules. De plus, un nouveau routage peut être lancé par une source ou une destination, cette dernière désignant la source à laquelle elle désire se connecter. En comparaison des techniques de wormhole, notre approche offre un temps de transfert d'information nettement plus faible, puisqu'elle se fait de façon purement combinatoire. De plus, le mécanisme à base d'identifiant que nous utilisons est nettement plus flexible que celui basé sur des adresses physiques des unités communiquantes.

5.2 Principes

Trois principes de base nous ont guidés durant la mise au point de nos algorithmes, de manière à les rendre les plus efficaces possibles dans le cas d'une implémentation matérielle.

- **Parallélisme.** Premièrement, nous nous intéressons aux systèmes parallèles, ou distribués. L'idée sous-jacente est ici de disposer d'un grand nombre d'éléments identiques capables de mener à bien la tâche de routage. Le projet encadrant cette thèse visant la réalisation d'un circuit de type FPGA, il est également clair qu'une structure régulière d'éléments identiques correspond entièrement au paradigme FPGA. Nous verrons plus loin que l'architecture retenue met en jeu une structure de type FPGA dont les éléments de base sont reliés à un deuxième niveau de modules responsables du routage.
- **Autonomie.** Deuxièmement, contrairement aux systèmes existants, nous désirons rendre le routage autonome. Aucun contrôleur global ne doit être présent, les unités de routage distribuées devant être seules maîtres de leur fonctionnement. Cette approche a été retenue dans l'optique de réaliser un système autoréparable. Un élément de contrôle global y serait un maillon faible qui, en cas de défaillance, pourrait compromettre l'ensemble du fonctionnement du système. Nous verrons que l'autoréparation n'a toutefois pas été investiguée dans cette thèse, mais que de futurs travaux pourraient être entrepris dans ce sens. Finalement, un des points forts d'un système autonome concerne les applications où des parties de circuit décident elles-mêmes de créer des connexions, tels des réseaux de neurones à topologie variable, où des neurones pourraient dynamiquement créer de nouvelles liaisons en fonction de leur taux d'activation. Dans de tels cas un contrôle global est superflu, et n'est pas en accord avec le concept de système cellulaire pour lequel un maximum d'autonomie est souhaitable.
- **Simplicité.** Troisièmement, notre optique est de travailler sur des systèmes physiquement réalisables sur du silicium et potentiellement en nanotechnologies dans le futur. Le nombre de transistors disponibles dans les circuits intégrés n'étant pas infini, une attention particulière doit être donnée à la minimisation de la taille de nos éléments de routage. Il est hors de question d'implémenter un protocole du type TCP/IP, c'est pourquoi le développement de nos algorithmes s'est effectué dans le respect du principe de simplicité. Chaque élément de routage est peu complexe, mais permet au système de fonctionner parfaitement.

Nous pouvons également noter que l'un des aboutissements de ce travail fut la réalisation physique d'un nouveau FPGA comportant un système de routage dynamique. Le peu de silicium à disposition nous a donc fortement poussé à une optimisation maximale de la taille de nos composants.

5.3 Hypothèses

Les trois principes directeurs énumérés ci-dessus ont fixé un cadre large concernant la conception matérielle d'un système de routage distribué autonome. Les hypothèses présentées dans cette section visent à donner un cadre plus étroit dans lequel tous les algorithmes développés doivent tenir.

1. Un système de routage est composé d'unités de routage identiques connectées à leurs plus proches voisines (principe de parallélisme). Nous présenterons les solutions concernant un voisinage de 4 avant de proposer d'autres alternatives.
2. Aucun contrôleur global ne doit être présent, l'entièreté de la gestion du routage étant laissée aux unités de routage (principe d'autonomie).
3. Nous tolérons des liaisons combinatoires de longue distance traversant les unités de routage présentes dans le système.
4. Une unité de routage est entre autre composée de multiplexeurs permettant de sélectionner le signal à transmettre à chacune des voisines. Le but de l'algorithme de routage est la configuration correcte de ces multiplexeurs.
5. Le routage terminé, les multiplexeurs transmettent de manière combinatoire les valeurs de proche en proche.
6. Chaque unité de routage est connectée à un élément externe, qui peut typiquement être un élément logique d'un FPGA. Cet élément externe reçoit et envoie des valeurs à l'unité de routage afin de mener à bien tout processus de routage ainsi que pour transmettre des informations à travers le circuit.
7. L'implémentation matérielle des algorithmes étant le but de notre étude, le design des unités de routage se doit de nécessiter un nombre de transistors le plus faible possible (principe de simplicité).
8. De même, le nombre de connexions entre les unités de routage doit être le plus petit possible.

La figure 5.2 illustre la grille à deux dimensions composée d'unités de routage interconnectées. Nous y observons également la présence des éléments externes reliés à ces unités. Dans ce chapitre nous ne définirons pas la structure de ces éléments, mais seulement le type d'interactions qu'ils doivent avoir avec leur unité de routage. Ce n'est que dans le chapitre 6, consacré à l'implémentation du circuit POEtic, que nous en montrerons une réalisation.

5.4 Premières solutions

Se basant sur les hypothèse énoncées, plusieurs solutions peuvent être réalisées. Nous en proposons trois dans cette section, par ordre croissant de complexité. Ces options n'ont pas été retenues pour une implémentation matérielle, de par leur manque de flexibilité et d'adaptabilité, caractéristiques indispensables quant à la réalisation de

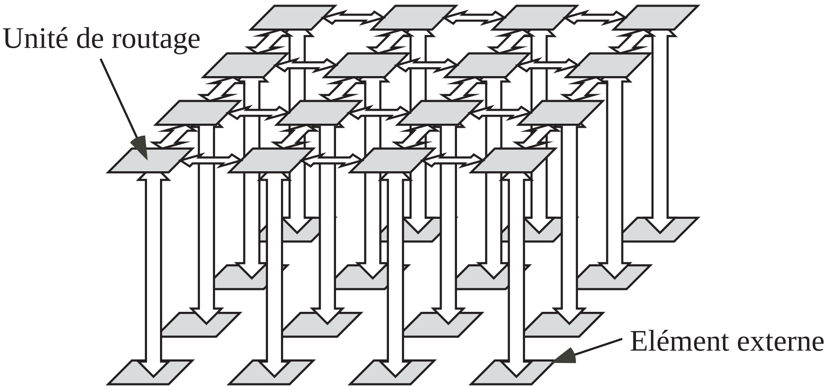


Figure 5.2 : *Le tableau d'unités de routage, connecté à un tableau d'éléments quelconques.*

systèmes bio-inspirés. Nous les présentons toutefois de manière à être le plus complet possible sur les potentielles solutions de routage distribué autonome. Ce cheminement vers une complexité croissante nous mènera tout naturellement vers les algorithmes HIDRA de la section suivante.

5.4.1 Algorithme direct

Le système de routage le plus simple consiste à envoyer une valeur dans une direction, la première destination rencontrée la récupérant (Figure 5.3). La réalisation matérielle d'une unité de routage, telle qu'explicitée à la figure 5.4 ne fait intervenir aucun contrôleur, les seuls bits envoyés par l'élément externe configurant directement les multiplexeurs.

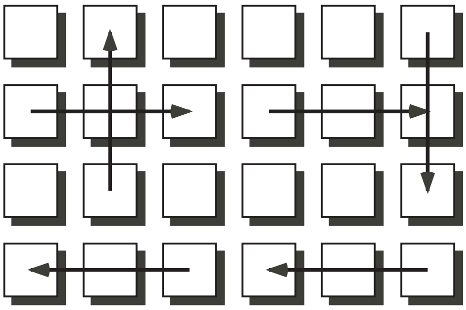


Figure 5.3 : *Dans le cas le plus simple, une connexion est une ligne droite entre la source et la destination.*

Une unité de routage transmet donc par défaut les signaux reçus dans la direction opposée, sauf si elle est une source. Dans ce cas, le signal provenant de son élément externe est transmis dans une direction particulière. Dans le cas d'une destination, la valeur reçue d'une certaine direction est transmise à l'élément externe (Figure 5.4). Sur le plan de l'implémentation matérielle, quatre multiplexeurs à deux entrées sélectionnent, pour chacune des directions, la valeur à envoyer. Par défaut, il s'agit de la valeur reçue de la voisine de direction opposée. Grâce à un démultiplexeur contrôlé par deux bits de direction, et sensible au fait que l'unité de routage est une source, au maximum un des multiplexeurs est configuré pour sélectionner la valeur envoyée par l'élément externe. Si, en revanche, l'unité de routage est une destination, les deux bits de direction permettent de récupérer la valeur venant d'une des quatre directions, via un multiplexeur à quatre entrées, et de la transmettre à l'élément externe.

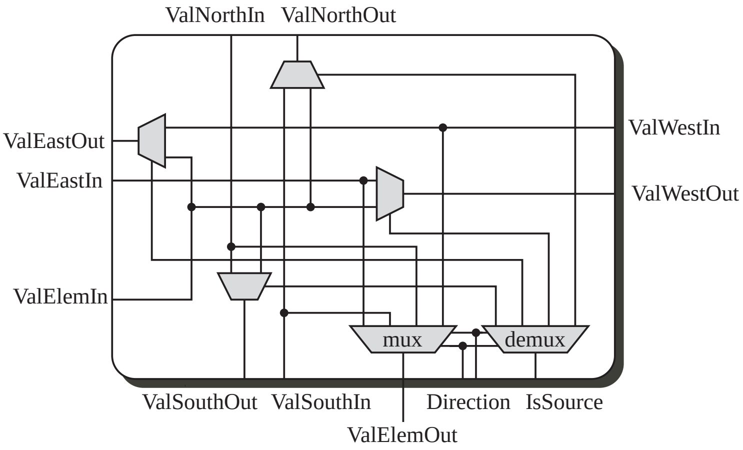


Figure 5.4 : *Une unité de routage ne permettant que de connecter des points sur la même ligne.*

Cette approche a l'avantage de ne nécessiter qu'un nombre très réduit de matériel par unité de routage. Il est également intéressant de noter qu'aucun processus de routage n'est nécessaire, puisqu'ici les signaux de configuration soumis par l'élément externe suffisent à configurer correctement les multiplexeurs. La réalisation d'un tel système serait adaptée à des applications de type automates cellulaires, où une cellule est toujours connectée à ses plus proches voisines. Toutefois, les connexions ne peuvent s'effectuer que sur des lignes verticales ou horizontales, sans qu'un seul virage ne soit possible. Une source émettant dans une direction X ne peut en effet toucher que des destinations présentes sur la même ligne, dans ladite direction. Son manque de flexibilité est donc flagrant, et elle n'a par conséquent pas été retenue pour une implémentation physique.

5.4.2 Algorithme à adressage relatif direct

Afin de pallier au faible potentiel de routage de la solution précédente, nous proposons d'introduire un système d'adressage relatif. Une source y contient deux valeurs, représentant respectivement un décalage en X et un en Y, utilisés pour joindre une destination. Nous avons ici l'introduction d'un processus de routage durant lequel une source cherche à joindre sa destination correspondante, en envoyant la valeur relative (x, y) à sa voisine présente dans la direction de la destination.

Pour l'implémentation d'un tel système, chaque unité de routage doit être composée de quatre multiplexeurs à 4 entrées pour les signaux à propager vers ses voisines, et un supplémentaire pour le signal à transmettre à l'élément externe. De plus, un contrôleur doit gérer le processus de routage afin de configurer de manière correcte les multiplexeurs. L'algorithme réalisé est relativement simple.

Premièrement, un mécanisme permettant d'établir une priorité entre les unités de routage est nécessaire. Une fois un maître élu, il envoie de manière sérielle les coordonnées (x, y) à sa voisine correspondant à la direction requise, telle décrite au tableau

5.1.

Règle	**Direction de propagation**
Si $x > 0$	Est
Sinon si $x < 0$	Ouest
Sinon si $y > 0$	Nord
Sinon si $y < 0$	Sud
Sinon	Élément externe

Tableau 5.1 : *Direction de propagation du routage, en fonction des coordonnées.*

L'unité de routage recevant ces coordonnées effectue alors une opération sur les coordonnées reçues. Cette incrémentation ou décrémentation dépend de l'origine de leur réception, et est décrite dans le tableau 5.2. Une seule opération est réalisée, et son résultat est directement utilisé afin de déterminer la direction de propagation décrite par le tableau 5.1.

Origine	**Opération**		
Nord	$x = x$	;	$y = y + 1$
Est	$x = x + 1$	;	$y = y$
Sud	$x = x$	;	$y = y - 1$
Ouest	$x = x - 1$	;	$y = y$

Tableau 5.2 : *Opération réalisée sur les coordonnées, en fonction de l'origine de celles-ci.*

Lorsque l'unité de routage obtient une coordonnée $(0, 0)$, c'est qu'elle est la destination du processus, et ce dernier se termine. La figure 5.5 présente trois chemins créés, chaque source contenant les coordonnées relatives de sa destination. Nous pouvons noter que l'algorithme à adressage relatif est plus souple que le direct, étant donné qu'il est possible d'y relier n'importe quelle unité de routage avec n'importe quelle autre. Toutefois, les chemins sont toujours construits de la même manière, en se positionnant d'abord sur l'axe X, puis en retrouvant la destination sur l'axe Y. De ce fait, une unité de routage se trouvant sur un chemin existant allant dans la même direction que la propagation ne peut être connectée. C'est pourquoi une variante à adressage relatif indirect se propose de pallier à ce manque de flexibilité.

5.4.3 Algorithme à adressage relatif indirect

L'idée de cette troisième solution est, comme pour la précédente, de connecter une source et une destination en fonction de leur adressage relatif. Le manque de flexibilité de l'adressage relatif direct vient du fait qu'un chemin possède au plus un virage. Nous proposons donc de permettre la création de chemins sinueux garantissant ainsi la réalisation du chemin s'il en existe un. Pour ce faire, deux modifications sont nécessaires. Tout d'abord, après avoir choisi un maître, celui-ci transmettra les coordonnées dans les quatre directions, et non dans une seule. De la même manière que précédemment, chaque unité de routage recevant ces coordonnées stocke l'origine de ces informations

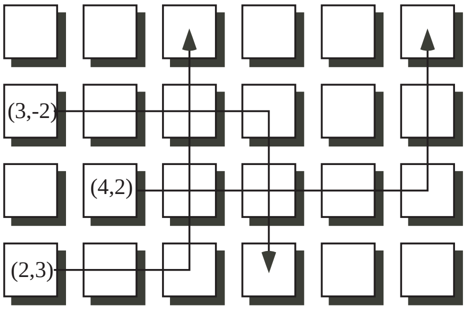


Figure 5.5 : *Trois chemins créés avec un adressage relatif direct.*

et effectue l'opération du tableau 5.2 puis compare le résultat à $(0,0)$. Si elle n'est pas la destination, elle propage les nouvelles coordonnées dans les quatre directions. Cette phase d'expansion se termine lorsque la destination est atteinte, et une phase de rétro-propagation partant de la destination permet de configurer les multiplexeurs de manière à créer le chemin désiré.

Nous n'entrons pas en détail dans la description de la phase d'expansion et de rétro-propagation, étant donné qu'elles seront décrites dans le cadre de la solution retenue. La figure 5.6 montre la création d'un chemin supplémentaire en comparaison de la figure 5.5. Ce chemin ne pouvait pas être créé avec l'algorithme relatif direct, alors que la variante indirecte le rend possible, au prix de deux virages.

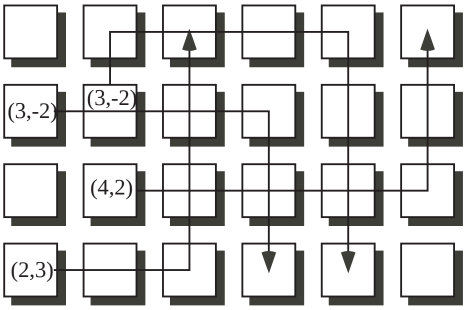


Figure 5.6 : *Quatre chemins créés avec un adressage relatif indirect.*

5.5 HIDRA

Nous venons de présenter trois solutions au problème du routage matériel distribué autonome, dont la troisième offre une bonne flexibilité dans le sens où un adressage relatif permet de relier deux unités de routage sans que le chemin créé ne doive être composé seulement de deux lignes droites. Toutefois, cette flexibilité manque encore d'adaptabilité, une destination devant connaître par avance la position exacte (même si relative), de sa source. Afin d'améliorer encore les performances de notre routage matériel, nous allons maintenant baser nos algorithmes sur un adressage par identifiants. Une source y possède un identifiant, et une destination y connaît l'identifiant de sa source. De cette manière, la position des deux unités de routage à relier n'importe pas, et n'a donc pas besoin d'être connue a priori.

Cet avantage est décisif dans l'optique d'implémenter des systèmes cellulaires adaptatifs bio-inspirés du type réseaux de neurones à topologie variable, ou du type système multicellulaire capable de croissance et d'autoréparation. En effet, si un nouveau neurone est ajouté à un réseau, il doit avoir la possibilité de se connecter à ceux déjà présents, et ce sans avoir de connaissance préalable sur leur position exacte. De plus, le projet POEtic, comme nous le verrons au chapitre 6, a vu la mise au point d'un nouveau circuit FPGA adaptatif contenant l'algorithme de routage que nous allons présenter, dont le fonctionnement a été décrit dans [234]. Dans le cas d'applications cellulaires, le placement des cellules sur le substrat électronique qu'est la partie reconfigurable du FPGA n'est donc plus important, la connexion de cellules par le réseau de routage n'étant pas dépendante de la position de celles-ci.

Le système à base d'identifiants est donc un excellent candidat pour l'implémentation distribuée autonome, chaque unité de routage étant consciente de son identifiant ou de l'identifiant de sa source. Aucun contrôle global n'est dès lors nécessaire pour déterminer deux unités de routage à connecter. Alors que dans les solutions précédemment évoquées (sections 5.4.2 et 5.4.3) les adresses correspondaient aux coordonnées relatives de la destination par rapport à la source, nous utiliserons à présent le terme “adresse” comme synonyme d'identifiant.

Une des particularités de HIDRA est que la création d'un chemin, c'est-à-dire le lancement d'un processus de routage, peut être effectué à n'importe quel instant, par une source ou par une destination. En effet, certaines applications, tels les processus ontogénétiques, nécessitent qu'une cellule cherche à se connecter à une cellule en attente, et donc qu'une source tente de trouver une destination libre. De même, pour un réseau de neurones à topologie variable, un nouveau neurone doit pouvoir choisir la provenance de ses entrées. Une destination doit donc être en mesure de partir à la recherche de sa source correspondante.

Avant de présenter l'algorithme HIDRA, nous allons brièvement expliquer son nom, composé de cinq lettres liées à des mots anglais, publication scientifique oblige :

- **H (Hardware) :** Nous nous intéressons à la réalisation matérielle (hardware) de circuits électroniques.
- **I (Incremental) :** Les algorithmes développées se doivent d'être incrémentaux, c'est-à-dire que la création de chemins de données doit pouvoir se faire à tout moment, sans qu'une connaissance de tous les chemins à créer ne soit nécessaire. De plus, la création d'un nouveau chemin ne doit en aucun cas en détruire un préexistant.
- **D (Distributed) :** Nos algorithmes sont distribués, c'est-à-dire qu'un ensemble d'unités de routage identiques est responsable de la création des chemins, sans qu'un contrôle global ne soit nécessaire.
- **R (Routing) :** Il s'agit bien évidemment de routage, où l'on désire configurer les multiplexeurs d'un switchbox qui sont ensuite utilisés pour la transmission d'information entre différents points du circuit.
- **A (Algorithm) :** Finalement, nous parlons d'un algorithme, servant à créer des chemins de données entre différents points d'un circuit électronique.

Le nom de notre algorithme étant maintenant explicité, nous le présentons dans la section suivante, avant de détailler l'architecture de l'unité de routage.

5.5.1 Algorithme

Dans le chapitre précédent, nous avons étudié l'implémentation matérielle distribuée de l'algorithme de Lee proposée par Moreno. Dans notre algorithme, la création d'un chemin d'une source à sa destination suit le même principe de fonctionnement correspondant à l'expansion d'une vague. Nous allons toutefois y ajouter la capacité des unités de routage de gérer elles-mêmes les processus de routage. HIDRA est principalement composé de cinq phases exécutées l'une à la suite de l'autre, que nous allons illustrer par l'exemple de la figure 5.7. Dans cette figure, ainsi que dans toutes les suivantes, la lettre “S” désigne une source, et la lettre “T” (Target) une destination.

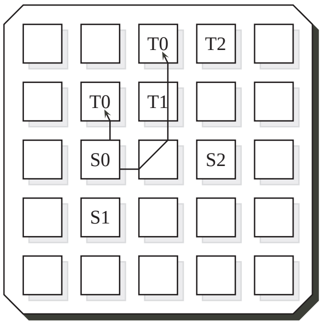


Figure 5.7 : *Exemple de routage à effectuer, avec des chemins déjà créés.*

Travaillant avec un voisinage de 4, nous utiliserons les points cardinaux Nord, Est, Sud, et Ouest pour désigner les positionnements relatifs des unités de routage. A titre d'exemple, dans la figure 5.7, la source S0 est au Nord de S1, et T2 est à l'Est de T0, et le numéro des unités de routage correspond à leur identifiant.

Phase 1 : Définition d'un maître

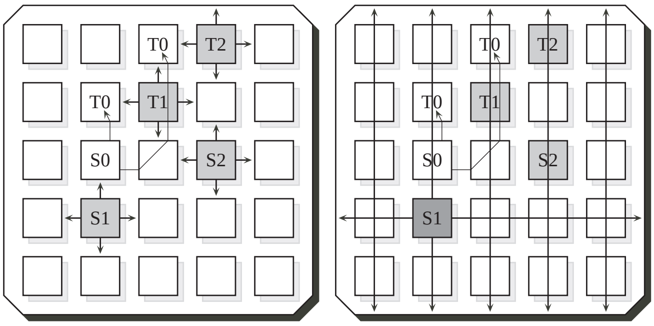
Tout d'abord, plusieurs unités de routage pouvant désirer se connecter à d'autres en même temps, il faut pouvoir régler ce litige. Pour ce faire nous utilisons une priorité donnée à l'unité de routage la plus au Sud-Ouest. Un dispositif simple a été développé, laissant plusieurs unités de routage placer un signal à ’1’, et permettant à chacune de savoir si elle a la priorité ou non. La priorité est gagnée par l'unité la plus au Sud n'ayant pas d'unité active à l'Ouest. Nous définissons la ligne implémentant ce système comme étant la ligne de propagation. Un seul coup d'horloge est nécessaire à cette première phase, la priorité étant définie de manière combinatoire.

La figure 5.8 montre la tentative de propagation de signal des quatre unités de routage à connecter, avec la victoire de celle placée le plus au Sud-Ouest.

A ce stade, nous venons d'introduire la ligne de propagation, qui permet aux unités de routage de transmettre un signal à toutes les autres, ainsi qu'à définir une priorité. Les unités de routage possèdent au total deux sorties dans chacune des directions, dont la première est cette ligne de propagation, et dont la deuxième sert à la phase d'expansion et de création du chemin.

Phase 2 : Envoi de l'identifiant

L'unité de routage ayant gagné la priorité envoie son identifiant à toutes les autres unités de routage en broadcast combinatoire, via la ligne de propagation. L'identifiant



(a) Tentative de devenir maître

(b) Une unité de routage devient maître

Figure 5.8 : 1^{re}*phase de HIDRA : Définition d'un maître.*

est envoyée de manière sérielle, en n coups d'horloge pour une adresse de n bits. Chaque unité de routage étant une source ou une destination la compare au sien, grâce à un comparateur sériel. Un trigger est chargé d'envoyer un signal à chaque unité de routage après les n coups d'horloge, période après laquelle chaque unité sait si elle possède ou non le même identifiant que le maître.

Phase 3 : Eliminations des concurrents

Après la comparaison des identifiants, plusieurs destinations ou sources peuvent se considérer comme participant au processus de routage courant, si elles possèdent le même identifiant que le maître. Deux cas peuvent se présenter :

- Si le maître est une source, il envoie un signal sur la ligne de propagation, et toutes les sources participantes sont désactivées. De cette manière, même lorsque plusieurs sources ont le même identifiant, seule celle ayant initié le processus de routage participe effectivement à celui-ci. Plusieurs destinations de même identifiant peuvent prendre part au processus de routage, et seule la première atteinte par la propagation sera effectivement connectée à la source.

  Dans le cas d'un système multicellulaire capable de croissance, par exemple, cette caractéristique est importante, et permet de disposer de plusieurs cellules totipotentes qui possèdent chacune une entrée (destination) en attente de connexion. Ayant toutes le même identifiant, il faut garantir qu'une cellule qui tente de se connecter à l'une d'elle pour y introduire un génome ne se connecte qu'à une et une seule de ces cellules.
- Si le maître est une destination, aucun signal n'est transmis via la ligne de propagation, et toutes les destinations participantes sont désactivées. Seule la destination ayant initié le routage sera effectivement connectée, permettant à d'autres destinations de même identifiant de se connecter ultérieurement. Plusieurs sources possédant le même identifiant peuvent participer au processus de routage, et seule la première à atteindre la destination lui sera connectée.

La figure 5.9 montre la propagation du signal à '1' de la part de la source.

Cette phase est cruciale afin d'assurer que seule l'unité de routage ayant lancé le processus et ayant gagné la priorité sera connectée. Plusieurs applications nécessitent

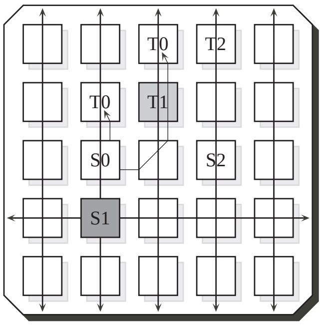


Figure 5.9 : *3^{e}phase de HIDRA : Désactivation des concurrents.*

effectivement cette caractéristique, dont notamment les processus ontogénétiques, que nous décrirons en page 224. Les sources y lancent des routages, alors que plusieurs sources ont le même identifiant. Il faut donc pouvoir garantir que seule l'unité maître ne se connecte.

Dans le cas de neurones, par exemple, l'ajout d'une nouvelle cellule implique qu'elle connecte ses entrées à la sorties des neurones auxquels elle doit être reliée. Pour ce faire, elle cherche, pour chaque entrée, la source la plus proche qui possède le même identifiant que son entrée, de manière à récupérer la sortie d'un autre neurone.

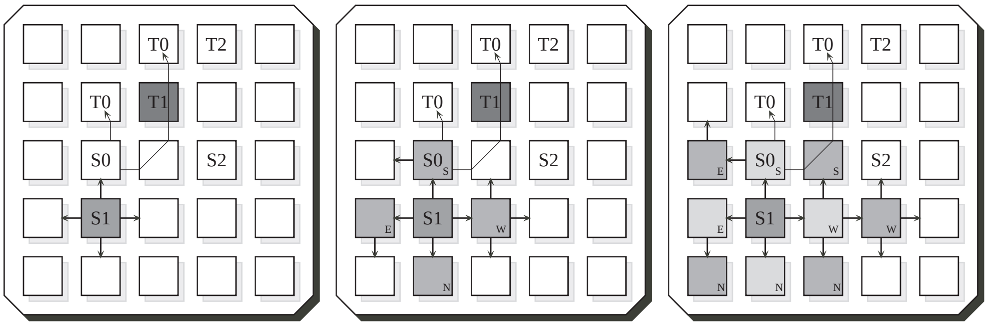
Phase 4 : Expansion

Après que les unités de routage concurrentes du maître aient été inhibées, les sources participantes lancent l'expansion. Cette phase est semblable à l'algorithme de Lee-Moore, et à l'implémentation qu'en a fait Moreno. Le front d'onde est propagé, en partant des sources actives, à chaque coup d'horloge, jusqu'à atteindre une destination. Chaque source participante envoie un signal à chacune de ses voisines, pour autant que le multiplexeur correspondant ne soit pas configuré de manière à sélectionner un signal autre que celui venant de l'élément externe. De par ce fait, un chemin existant ne peut être détruit, et la possibilité de réutiliser un chemin déjà créé est conservée.

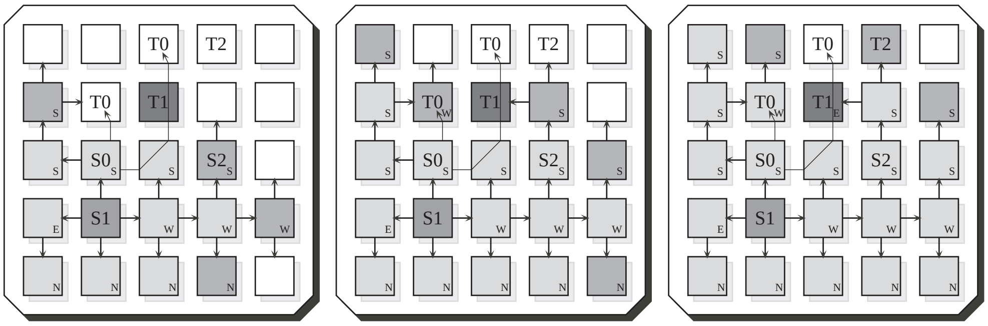
Chaque autre unité de routage attend qu'une de ses entrées soit activée, et lorsqu'une ou plusieurs entrées sont activées, une priorité est donnée au Nord, à l'Est, au Sud, puis finalement à l'Ouest, et l'origine du signal est stockée dans deux bascules. Une fois l'origine stockée, c'est-à-dire après un coup d'horloge, l'unité de routage propage l'expansion en activant les sorties vers ses voisines, pour autant que le multiplexeur correspondant ne soit pas configuré en sélectionnant une autre entrée que l'origine de l'expansion.

Cette phase nécessite un nombre de coups d'horloge égal à la plus courte distance entre la source et la destination prenant en compte les chemins déjà créés.

La figure 5.10 montre l'entièreté de la phase d'expansion permettant de relier la source 1 à la destination 1. Les nuances de gris permettent d'identifier les différentes unités de routage. La plus foncée est la destination à atteindre, la nuance suivante indique les unités de routage sur le front d'onde, c'est-à-dire qui vont étendre la vague au coup d'horloge suivant, et les plus claires sont les unités qui ont déjà mémorisé leur origine et propagé la vague. La lettre présente dans le coin inférieur droit des unités atteintes par la vague indique l'origine de l'expansion avec les premières lettres anglaises des quatre points cardinaux.



(a) Expansion 1 (b) Expansion 2 (c) Expansion 3



(d) Expansion 4 (e) Expansion 5 (f) Expansion 6

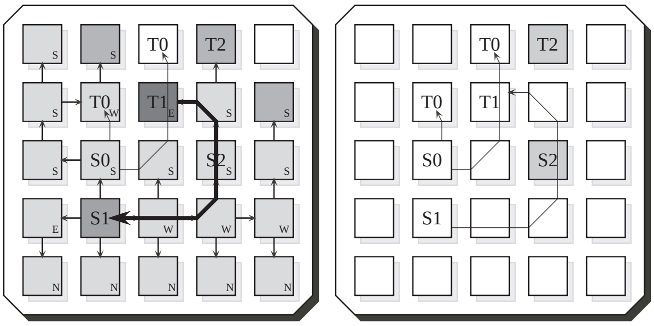
Figure 5.10 : *4e phase de HIDRA : Propagation du front d'onde.*

Phase 5 : Création du chemin

Lorsque la destination est atteinte par l'expansion, elle active la ligne de propagation au coup d'horloge suivant, signalant ainsi la fin du processus de routage. Si plusieurs destinations sont atteintes au même instant, la ligne de propagation permet d'établir une priorité, et seule la destination la plus au Sud-Ouest crée effectivement un chemin. La destination sélectionnée envoie alors un signal actif en direction de l'origine de l'expansion, qu'elle a précédemment stockée. L'unité de routage touchée par ce signal le propage également en direction de son origine, et cette rétro-propagation s'effectue jusqu'à arriver à la source. En parallèle à cette rétro-propagation, les unités de routage touchées configurent le multiplexeur correspondant au sens de leur expansion de manière à créer le chemin entre la source et la destination. Une fois le chemin créé, soit après un coup d'horloge, toutes les unités de routage se retrouvent dans leur état initial, prêtes à recommencer un nouveau processus de routage, ou à simplement œuvrer à la transmission d'information.

La figure 5.11 présente la création du chemin, où en un coup d'horloge un signal est propagé de la destination à la source, fixant ainsi les multiplexeurs présents sur le chemin.

L'algorithme 5.1 illustre les cinq phases de HIDRA, d'un point de vue global. Nous allons maintenant nous intéresser aux unités de routage, qui sont des composantes locales, sur le plan de leur fonctionnement et de leur implémentation.



(a) Rétropropagation

(b) Exploitation du routage effectué.

Figure 5.11 : 5^e*phase de HIDRA : Création du chemin.*

Algorithme 5.1 Algorithme HIDRA

```
Tant que vrai Faire
  Si au moins une unité de routage (RU) veut se connecter alors
    Le maître est la RU la plus au Sud n'ayant pas de RU désirant se connecter à l'Ouest
    Pour i=1 à taille de l'identifiant Faire
      Le maître envoie le $i^{\text{ème}}$ bit de son identifiant à tous les RUs
      Les RUs le comparent avec le $i^{\text{ème}}$ bit de leur propre identifiant
    Fin faire
    Les RUs de même identifiant participent au routage
    Si Le maître est une source alors
      Aucune autre source ne participe
    Sinon
      Aucune autre destination ne participe
    Fin si
    Les sources participantes sont sur le front d'onde
    Tant que Une destination participante n'est pas atteinte Faire
      Les RUs sur le front d'onde s'étendent
      Les RUs atteintes par le front d'onde stockent son origine
      Les RUs atteintes par le front d'onde deviennent le front d'onde
      Si Le front d'onde est vide alors
        Problème de congestion, le chemin ne peut être trouvé
      Fin si
    Fin tant que
    Parcourir les RUs de la destination à la source, en configurant les multiplexeurs
  Fin si
Fin tant que
```

5.5.2 Unité de routage

L'algorithme HIDRA ayant été explicité, nous pouvons nous intéresser à la structure d'une unité de routage permettant son implémentation distribuée. Après avoir présenté sa structure globale, nous allons détailler ses composants, et plus particulièrement son contrôleur.

Structure Globale

Une unité de routage est principalement composée d'un switchbox, d'un contrôleur, d'un comparateur d'adresse, et d'une unité de propagation, comme le montre la figure 5.12. Nous pouvons y observer la présence d'un multiplexeur sélectionnant la valeur à transmettre dans chaque direction, permettant au contrôleur d'envoyer un signal venant de lui durant les processus de routage, ou d'être utilisé de manière à laisser le switchbox sélectionner la valeur à transmettre lors de la phase d'exécution normale. Ceci implique que le niveau de routage distribué ne peut être utilisé durant un processus de routage, et fonctionne donc sur deux modes : en cours de routage, ou en transmission d'information. Il serait possible aux deux modes d'être actifs simultanément à condition d'ajouter une sortie dans chacune des directions, et de supprimer ces quatre multiplexeurs. Étant donné que la minimisation du nombre de liaisons entre unités de routage fut de la plus grande importance, nous avons préféré la solution multiplexée, mais rien n'empêcherait de modifier cette implémentation pour faire fonctionner ces deux modes simultanément. Dans le cas où le fonctionnement des éléments externes doit être inhibé pendant les processus de routage, le signal `IsRouting`, en sortie de l'unité de routage, indique si un routage est en cours ou non.

Intéressons-nous tout d'abord aux entrées de l'unité de routage :

- `Clk` : Une horloge, identique pour toutes les unités.
- `Rst` : Un reset, également identique pour toutes.
- `AddrIn` : Un bit de l'identifiant envoyé par l'élément externe.
- `IsSource` : Ce signal, envoyé par l'élément externe, indique si l'unité de routage est une source ou non.
- `IsTarget` : Ce signal, envoyé par l'élément externe, indique si l'unité de routage est une destination ou non.
- `StartRouting` : Ce signal, envoyé par l'élément externe, indique s'il désire se connecter ou non. Dans le cas d'une activation, un routage sera lancé par l'unité de routage tant qu'elle n'est pas connectée.
- `TriggerIn` : Ce signal est utilisé pour détecter la fin de comparaison d'un identifiant. Il envoie un '1' tous les n coups d'horloge pour lesquelles la sortie ReadAddr est active, et ce pour un identifiant de n bits.
- `ValInExt` : Valeur envoyée par l'élément externe lorsque l'unité de routage est une source.
- `ValInN, ValInE, ValInS, ValInW` : Un signal venant de chaque voisine sert à transmettre une valeur durant le fonctionnement normal du système, ou à gérer le déroulement de l'algorithme lors d'un processus de routage.
- `PropInN, PropInE, PropInS, PropInW` : Un signal venant de chaque voisine permet de transmettre un signal à toutes les unités de routage du circuit ainsi qu'à introduire une priorité entre les unités de routage (ligne de propagation).

Et à ses sorties :

- `ReadAddr` : Indique à l'élément externe qu'il doit fournir un nouveau bit d'identifiant et active également le trigger.

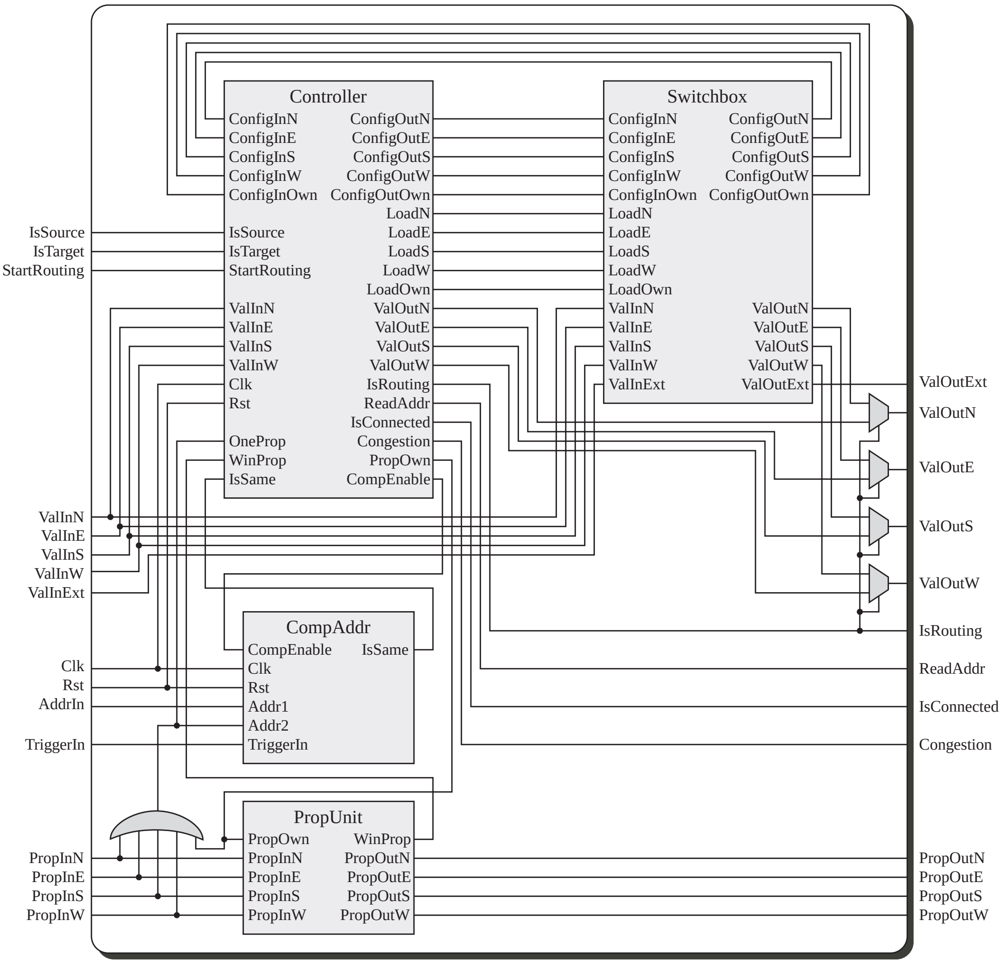


Figure 5.12 : *Schéma général d'une unité de routage, composée de quatre parties : un switchbox, un contrôleur, un comparateur d'adresse, et une unité de propagation.*

- `Congestion` : Indique une potentielle congestion. Au niveau du circuit, une combinaison de ces signaux reçus par toutes les unités de routage permet de savoir si l'algorithme est bloqué ou non.
- `IsRouting` : Indique si un processus de routage est en cours ou non.
- `IsConnected` : Indique à l'élément externe si l'unité de routage est connectée à son correspondant.
- `ValOutExt` : Valeur envoyée à l'élément externe lorsque l'unité de routage est une destination et qu'elle a été connectée.
- `ValOutN, ValOutE, ValOutS, ValOutW` : Valeurs envoyées à chacune des voisines, servant à transmettre une valeur durant le fonctionnement normal du système, ou à gérer le déroulement de l'algorithme lors d'un processus de routage.
- `PropOutN, PropOutE, PropOutS, PropOutW` : Signal envoyé à chaque voisine permettant de transmettre un signal à toutes les unités de routage du circuit ainsi qu'à introduire une priorité entre les unités de routage (ligne de propagation).

Switchbox

Commençons notre description des composants par celui permettant la transmission des signaux dans la direction adéquate. Le switchbox contient cinq multiplexeurs à quatre entrées : un dans chacune des directions, ainsi qu'un dernier dirigé vers l'élément externe. Chaque multiplexeur est contrôlé par deux bits de configuration, qui sont modifiés par le contrôleur de l'unité de routage. Un bit supplémentaire est également ajouté aux quatre premiers multiplexeurs, indiquant si le multiplexeur a été configuré ou non. Il permet au contrôleur de vérifier l'existence d'un chemin précédemment créé, de manière à ne pas le détruire.

La figure 5.13 montre la manière dont les multiplexeurs sont reliés aux entrées et sorties. Pour chacune des quatre directions, la sélection se fait parmi les trois autres voisines et l'élément externe, alors que la sortie vers l'élément externe est choisie entre les quatre voisines.

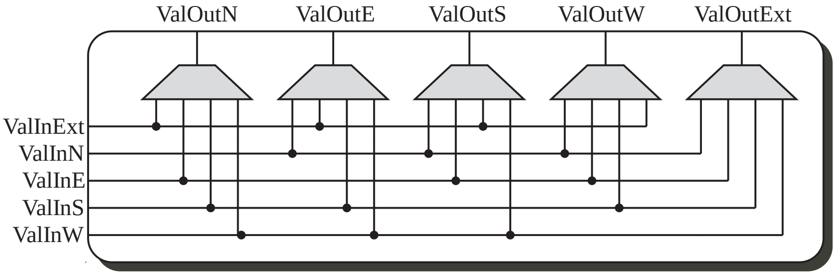


Figure 5.13 : *Schéma d'un switchbox, ne représentant que la connectique des multiplexeurs.*

La figure 5.14 donne le schéma d'un des quatre premiers multiplexeurs, sur lequel nous pouvons distinguer les bascules nécessaires au stockage de la configuration d'un multiplexeur. Cette illustration représente le multiplexeur responsable de sélectionner la valeur à envoyer au Sud. Il peut choisir parmi les valeurs reçues par les trois voisines Nord, Est, et Ouest, ou la valeur envoyée par l'élément externe. Les signaux `LoadS` et `ConfigInS` sont envoyés par le contrôleur de l'unité de routage, qui récupère `ConfigOutS`, dont il a besoin pour la bonne marche de la phase d'expansion de l'algorithme. Nous pouvons d'ores et déjà noter que les éléments nécessaires à la réalisation d'un switchbox sont 5 multiplexeurs à 4 entrées et 14 bascules[1].

Unité de propagation

L'unité de propagation, présente dans chaque unité de routage, sert à transmettre une valeur en broadcast à toutes les autres unités de routage, et à définir une priorité lors de la phase 1 de l'algorithme. L'idée est ici de pouvoir transmettre un '1' à travers tout le circuit tout en étant capable de discerner si une unité de routage se trouve la plus au Sud-Ouest.

Pour ce faire, nous disposons d'une entrée et d'une sortie par côté. Il ne reste qu'à définir la manière dont une entrée active est traitée, c'est-à-dire à quelle voisine elle doit être transmise. La figure 5.15 montre le sens de transmission des signaux dans le

---

[1] Le bon fonctionnement du contrôleur ne nécessite pas de savoir si le multiplexeur de l'élément externe est configuré ou non.

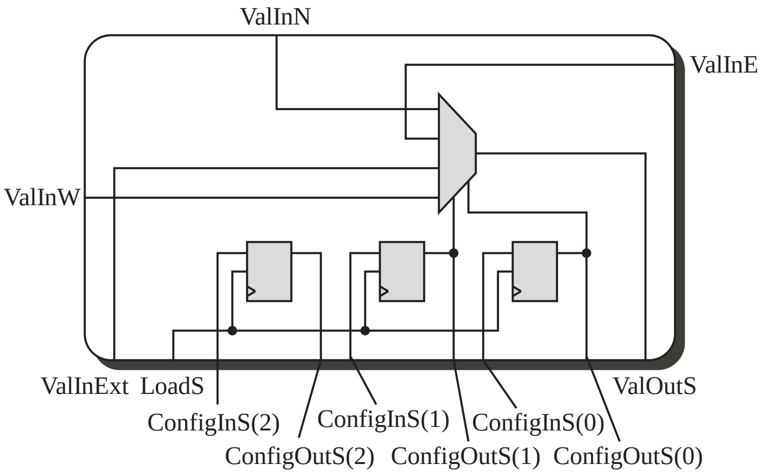


Figure 5.14 : *Schéma représentant un multiplexeur et ses trois bits de configuration.*

cas où plusieurs unités de routages placent un ’1’ sur la ligne de propagation. Nous pouvons observer que l’élément le plus au Sud-Ouest est le seul n’ayant aucune entrée à ’1’, car il a pu gagner la priorité sur ceux placés plus au Nord ou plus à l’Est de lui.

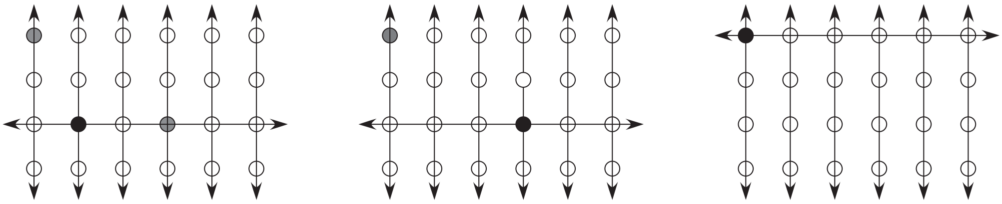


Figure 5.15 : *Ces trois figures illustrent la direction de propagation d’un signal sur la ligne de propagation. L’unité noire constate qu’elle est le maître alors que les grisées, qui tentent également de l’être, ne peuvent que constater leur échec.*

Le tableau 5.3 définit la transmission du signal de propagation permettant d’obtenir ce comportement. Le signal PropOwn est géré par le contrôleur de l’unité de routage, qui le place à ’1’ pour propager, tandis que WinProp indique si le contrôleur a gagné la priorité, c’est-à-dire est le plus au Sud-Ouest.

La figure 5.16 illustre deux schémas correspondant au comportement du tableau 5.3. Celui du haut n’est pas spécialement optimisé, mais présente bien la priorité donnée aux signaux provenant du Sud, de l’Ouest, du contrôleur, de l’Est, puis du Nord. Celui du bas, en revanche, offre un délai moins long au travers des portes logiques, et sera donc un meilleur candidat pour une réalisation matérielle, où la longueur des chemins combinatoires doit être réduite au maximum.

Comparateur d’adresse

Dans la phase 2 de l’algorithme HIDRA, le contrôleur vérifie si l’identifiant reçu correspond à sa propre adresse. Pour ce faire, un comparateur sériel a été implémenté. Outre une horloge et un reset, il possède quatre entrées : deux bits d’adresse à comparer, un *enable* autorisant la comparaison, et un trigger signifiant la fin de la comparai-

Règle	**Sorties actives**
Si `PropInS`=’1’	Nord
Sinon si `PropInW`=’1’	Nord, Sud, Est
Sinon si `PropOwn`=’1’	Nord, Est, Sud, Ouest, WinProp
Sinon si `PropInE`=’1’	Nord, Sud, Ouest
Sinon si `PropInN`=’1’	Sud

Tableau 5.3 : *Direction de transmission du signal de propagation, en fonction de son origine.*

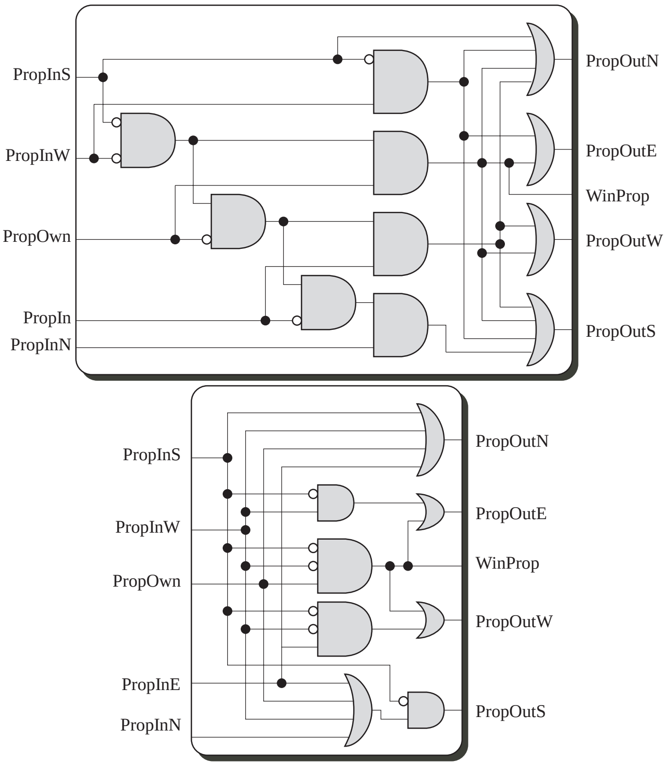


Figure 5.16 : *Deux schémas possibles d’une unité de propagation.*

son. Une seule sortie sert à indiquer si la comparaison a présenté une égalité des deux adresses ou non.

Sur le plan de l’implémentation, une seule bascule est nécessaire, de manière à mémoriser si au moins un bit comparé a été différent. Sa valeur vaut ’1’ après un reset, ou après la fin d’une comparaison d’adresse, et passe à ’0’ lorsque les deux bits d’adresse passés en entrée ne sont pas identiques, lorsque l’*enable* de la bascule est à ’1’. La sortie du comparateur est à ’1’ si la bascule est à ’1’, et que les deux bits d’adresse courants sont identiques. La figure 5.17 montre son schéma.

Contrôleur

Nous en arrivons à la partie centrale de l’unité de routage, à savoir le contrôleur. Le choix a été fait de l’implémenter à l’aide d’une machine d’états finis, de manière

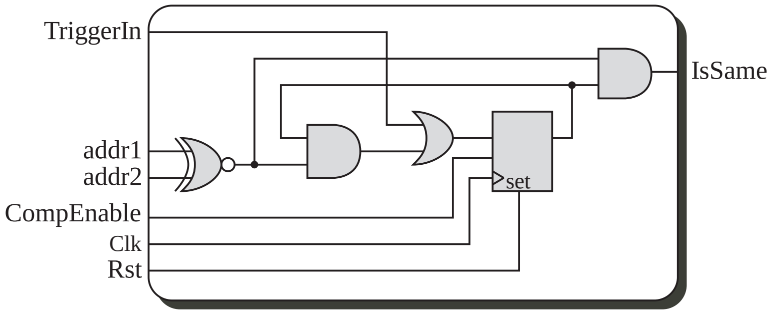


Figure 5.17 : *Schéma d'un comparateur sériel.*

à en rendre le développement et la compréhension les plus faciles possibles. Il s'agit d'une machine de Mealy (Figure 5.18), où les sorties dépendent de l'état du contrôleur et de ses entrées[2].

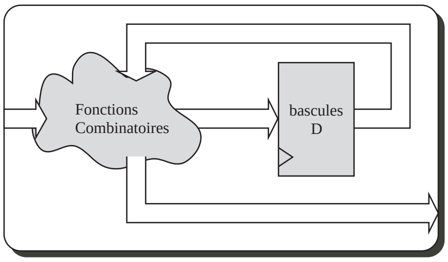


Figure 5.18 : *Le contrôleur est implémenté grâce à une machine de Mealy.*

Concernant les entrées/sorties du contrôleur, outre celles de l'unité de routage (page 111), les suivantes lui permettent de contrôler les multiplexeurs du switchbox.

En sortie :

- `LoadN`, `LoadE`, `LoadS`, `LoadW`, `LoadOwn` : forcent le chargement des bascules de contrôle de chacun des multiplexeurs.
- `ConfigOutN`, `ConfigOutE`, `ConfigOutS`, `ConfigOutW`, `ConfigOutOwn` : pour chacun des multiplexeurs, les trois bits de configuration à charger, sauf `ConfigOutOwn` qui n'en compte que deux.

Et en entrée :

- `ConfigInN`, `ConfigInE`, `ConfigInS`, `ConfigInW`, `ConfigInOwn` : pour chacun des multiplexeurs, les trois bits de configuration (sauf `ConfigOutOwn` qui n'en compte que deux), qui sont utilisés lors de la phase d'expansion de l'algorithme.

Six états, que nous appelerons `sInit`, `sAddress`, `sChooseSource`, `sWaitExp`, `sFrontExp`, et `sHasExp`, sont nécessaires à la bonne marche de l'algorithme HIDRA décrit précédemment. Le graphe de transitions de la machine d'états est illustré à la figure 5.19, et nous allons passer en revue ses six états en présentant dans les encarts le pseudo-code correspondant. Une description "françisée" est également fournie, afin de le rendre le plus clair possible.

Mais avant de commencer, précisons qu'en plus des bascules nécessaires à la réa-

---

[2] Le fait qu'il s'agisse d'une machine de Mealy implique des chemins combinatoires qui traversent potentiellement tout le circuit.

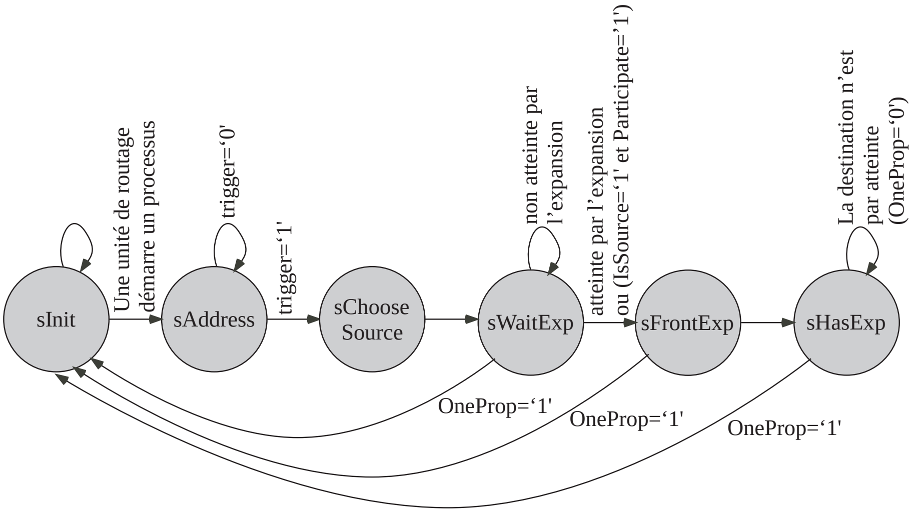


Figure 5.19 : *Machine d'états du contrôleur.*

lisation de la machine d'états, cinq autres sont indispensables à la bonne marche de l'algorithme :

- `Participate` : indique si l'unité de routage doit participer au processus de routage courant, c'est-à-dire est la source ou la destination courante.
- `IsMaster` : indique si l'unité de routage est celle qui lance le processus. Elle passe à '1' lors de la première phase de l'algorithme, lorsqu'un maître est choisi en fonction de la priorité Sud-Ouest.
- `IsConnected` : lorsque l'unité de routage est une source ou une destination, indique si elle est connectée à sa correspondante.
- `Origin` : sur 2 bits, permet de stocker l'origine de la phase d'expansion, de manière à pouvoir effectuer la phase de rétro-propagation.

De même, ces quelques signaux sont utiles à la compréhension des sections suivantes :

- `PropOwn` : signal géré par le contrôleur, relié à la ligne de propagation, et qui lui permet de l'activer.
- `WinProp` : indique si le contrôleur est prioritaire sur la ligne de propagation.
- `OneProp` : indique si au moins une des entrées de propagation ou PropOwn est active.
- `ValIn` : indique si au moins une des valeurs `ValIn`*X* est à '1'.
- `CompEnable` : Autorise la comparaison d'adresses.
- `IsSame` : indique si les deux adresses comparées sont identiques ou non.
- `WantConnect` : indique si l'unité de routage doit encore tenter de se connecter à sa correspondante.

Concernant les signaux séquentiels, nous placerons le préfixe `Next` devant le nom du signal pour indiquer la valeur qu'il prendra au coup d'horloge suivant (Exemple : `NextState`, `NextIsMaster`). Nous partons du principe que si aucune affectation n'est effectuée, la valeur de la bascule reste inchangée.

sInit Après un Reset, la machine d'états est placée dans son état initial. Les actions qui y sont prises sont décrites par la partie d'algorithme 5.2. Le contrôleur reste dans cet état tant qu’aucune unité de routage ne désire lancer un processus de routage. Le lancement est signalé par un passage à ’1’ de la ligne de propagation, et dans ce cas, tous les contrôleurs passent dans l’état sAddress, et la bascule `IsMaster` passe à ’1’ pour le contrôleur ayant gagné la priorité.

Algorithme 5.2 HIDRA : Contrôleur de l’unité de routage, état sInit

```
Si state = sInit alors
    PropOwn $\Leftarrow$ WantConnect AND StartRouting ;
    NextIsMaster $\Leftarrow$ WinProp ;
    Si OneProp alors
        NextState $\Leftarrow$ sAddress ;
        IsRouting $\Leftarrow$ ’1’ ;
    Fin si
    NextParticipate $\Leftarrow$ ’0’ ;
Fin si
```

sAddress Durant la deuxième phase de l’algorithme, tous les contrôleurs se trouvent dans l’état sAddress (partie d’algorithme 5.3). Ils y restent jusqu’à ce que le signal `TriggerIn` passe à ’1’. Le maître envoie sériellement son identifiant (lignes 12-14), et les autres la comparent au leur (ligne 15). Lorsque `TriggerIn` s’active, tous les contrôleurs passent dans l’état sChooseSource, et ceux ayant le même identifiant que le maître font passer la bascule `Participate` à ’1’ (lignes 19-21).

Algorithme 5.3 HIDRA : Contrôleur de l’unité de routage, état sAddress

```
10: Si state = sAddress alors
11:     IsRouting $\Leftarrow$ ’1’ ;
12:     Si IsMaster alors
13:         PropOwn $\Leftarrow$ AddrIn ;
14:     Fin si
15:     CompEnable $\Leftarrow$ ’1’ ;
16:     ReadAddr $\Leftarrow$ ’1’ ;
17:     Si TriggerIn alors
18:         NextState $\Leftarrow$ sChooseSource ;
19:         Si IsSame AND (WantConnect OR IsSource) alors
20:             NextParticipate $\Leftarrow$ ’1’ ;
21:         Fin si
22:     Fin si
23: Fin si
```

sChooseSource Le but de l’état sChooseSource (partie d’algorithme 5.4), correspondant à la troisième phase de l’algorithme, est de désactiver la participation de certaines unités de routage. Si le maître est une source, il place un ’1’ sur la ligne de propagation (lignes 26-28), et les autres unités de routage ayant `Participate` à ’1’, c’est-à-dire ayant le même identifiant, et étant également des sources, désactivent leur participation (lignes 29-30). De même, si le maître est une destination, la ligne de propagation reste à ’0’, et les autres destinations placent `Participate` à ’0’. Les contrôleurs ne restent qu’un coup d’horloge dans cet état, et passent directement à l’état sWaitExp.

sWaitExp La phase d’expansion de l’algorithme débute en voyant tous les contrôleurs passer dans l’état sWaitExp (partie d’algorithme 5.5). Les sources participantes n’y effectuent aucun traitement, mais passent directement à l’état sFrontExp (lignes 39-40). Tous les autres contrôleurs attendent d’avoir une de leurs entrées `ValIn`X à

Algorithme 5.4 HIDRA : Contrôleur de l'unité de routage, état sChooseSource

```
24: Si state = sChooseSource alors
25:   IsRouting ⇐ '1' ;
26:   Si IsMaster AND IsSource alors
27:     PropOwn ⇐ '1' ;
28:   Fin si
29:   Si OneProp AND not(IsMaster) AND IsSource alors
30:     NextParticipate ⇐ '0' ;
31:   Fin si
32:   Si not(OneProp) AND not(IsMaster) AND IsTarget alors
33:     NextParticipate ⇐ '0' ;
34:   Fin si
35:   NextState ⇐ sWaitExp ;
36: Fin si
```

'1'. Lorsque c'est le cas, une priorité est donnée dans l'ordre Nord, Est, Sud, Ouest pour le calcul de l'origine de l'expansion. Cette origine est stockée, et le contrôleur passe dans l'état sFrontExp.

Algorithme 5.5 HIDRA : Contrôleur de l'unité de routage, état sWaitExp

```
37: Si state = sWaitExp alors
38:   IsRouting ⇐ '1' ;
39:   Si IsSource and Participate alors
40:     NextState ⇐ sFrontExp ;
41:   Sinon
42:     Si ValInN alors
43:       NextOrigin ⇐ "00" ;
44:       NextState ⇐ sFrontExp ;
45:     Sinon si ValInE alors
46:       NextOrigin ⇐ "01" ;
47:       NextState ⇐ sFrontExp ;
48:     Sinon si ValInS alors
49:       NextOrigin ⇐ "10" ;
50:       NextState ⇐ sFrontExp ;
51:     Sinon si ValInW alors
52:       NextOrigin ⇐ "11" ;
53:       NextState ⇐ sFrontExp ;
54:     Sinon
55:       Congestion ⇐ '1' ;
56:     Fin si
57:     Si OneProp alors
58:       NextState ⇐ sInit ;
59:     Fin si
60:   Fin si
61: Fin si
```

sFrontExp Une unité de routage se retrouve dans l'état sFrontExp (partie d'algorithme 5.6) après avoir été atteinte par l'expansion, ou s'il s'agit d'une source participante. Elle n'y reste qu'un seul coup d'horloge, et passe directement à l'état sHasExp. S'il s'agit d'une source participante, le contrôleur place un '1' en valeur de sortie `ValOut`*X*, si l'une de ces deux conditions est remplie : le multiplexeur correspondant à la sortie n'est pas configuré, ou il l'est en sélectionnant la valeur de l'élément externe. De ce fait, aucun chemin existant ne peut être effaçé, et il est possible que le nouveau chemin en réutilise un déjà créé et partant de la même source (lignes 71-98). Si l'unité de routage n'est pas une source participante, elle active ses sorties `ValOut`*X* si l'une des deux conditions suivantes est remplie : le multiplexeur correspondant à la sortie n'est pas configuré, ou il l'est en sélectionnant la valeur venant de l'origine de l’expansion. Là encore, un chemin existant ne peut être effacé, mais il est possible de suivre un chemin connecté à la source participante.

Algorithme 5.6 HIDRA : Contrôleur de l’unité de routage, état sFrontExp

```
62:  Si state = sFrontExp alors
63:    IsRouting ⇐ ’1’ ;
64:    Si OneProp alors
65:      NextState ⇐ sInit ;
66:      Congestion ⇐ ’1’ ;
67:    Sinon
68:      Congestion ⇐ ’0’ ;
69:      NextState ⇐ sHasExp ;
70:      Si IsSource AND Participate alors
71:        Si ConfigInN(2) alors
72:          Si ConfigInN(1..0)="00" alors
73:            ValOutN ⇐ ’1’ ;
74:          Fin si
75:        Sinon
76:          ValOutN ⇐ ’1’ ;
77:        Fin si
78:        Si ConfigInE(2) alors
79:          Si ConfigInE(1..0)="01" alors
80:            ValOutE ⇐ ’1’ ;
81:          Fin si
82:        Sinon
83:          ValOutE ⇐ ’1’ ;
84:        Fin si
85:        Si ConfigInS(2) alors
86:          Si ConfigInS(1..0)="10" alors
87:            ValOutS ⇐ ’1’ ;
88:          Fin si
89:        Sinon
90:          ValOutS ⇐ ’1’ ;
91:        Fin si
92:        Si ConfigInW(2) alors
93:          Si ConfigInW(1..0)="11" alors
94:            ValOutW ⇐ ’1’ ;
95:          Fin si
96:        Sinon
97:          ValOutW ⇐ ’1’ ;
98:        Fin si
99:      Sinon
100:       ValOutN ⇐ (ConfigInN(2)=’0’) OR (Origin=ConfigInN(1..0)) ;
101:       ValOutE ⇐ (ConfigInN(2)=’0’) OR (Origin=ConfigInE(1..0)) ;
102:       ValOutS ⇐ (ConfigInN(2)=’0’) OR (Origin=ConfigInS(1..0)) ;
103:       ValOutW ⇐ (ConfigInN(2)=’0’) OR (Origin=ConfigInW(1..0)) ;
104:     Fin si
105:   Fin si
106: Fin si
```

sHasExp Après avoir passé dans l’état sFrontExp, un contrôleur se retrouve dans l’état sHasExp (partie d’algorithme 5.7), en attendant la création du chemin par la destination atteinte. Si l’unité de routage est une destination participante, elle place un ’1’ sur la ligne de propagation (lignes 109-111). Étant donné qu’il est possible que plusieurs destinations soient atteintes au même instant, la priorité de la ligne de propagation permet de ne créer un chemin que pour la vainqueur. Le contrôleur prioritaire place alors un ’1’ à la sortie `ValOut`*X* correspondant à l’origine qu’il a stockée (lignes 118-124), et chaque contrôleur recevant une entrée `ValIn`*X* active fait de même. De ce fait, en un coup d’horloge toutes les unités de routage sur le chemin entre la destination et la source sont touchées par ce signal de rétro-propagation. Le contrôleur touché configure alors le multiplexeur sélectionnant la valeur en direction de la destination, en fonction de l'origine qu'il a précédemment stockée (lignes 126-145). Au coup d'horloge suivant la phase de rétro-propagation, tous les contrôleurs se retrouvent dans l'état sInit, prêts à relancer un nouveau processus de routage.

Les deux bits de poids faibles chargés dans la configuration des multiplexeurs dépendent du fait que l'unité de routage est une source participante ou non. Si tel est le cas, le multiplexeur est configuré de façon à sélectionner la valeur reçue de l'élément externe, alors que dans le cas contraire, ce sont les deux bits `Origin` que le sont.

Algorithme 5.7 HIDRA : Contrôleur de l'unité de routage, état sHasExp

```
107: Si state = sHasExp alors
108:   IsRouting ⇐ '1' ;
109:   Si Participate AND IsTarget alors
110:     PropOwn ⇐ '1' ;
111:   Fin si
112:   Si WinProp alors
113:     NextIsConnected ⇐ '1' ;
114:   Fin si
115:   Si OneProp alors
116:     Si IsSource and Participate alors
117:       NextIsConnected ⇐ '1' ;
118:     Sinon si ValIn OR WinProp alors
119:       switch (origin)
120:         "00" ⇒ ValOutN ⇐ '1' ;
121:         "01" ⇒ ValOutE ⇐ '1' ;
122:         "10" ⇒ ValOutS ⇐ '1' ;
123:         "11" ⇒ ValOutW ⇐ '1' ;
124:       end switch
125:     Fin si
126:     Si ValInN alors
127:       Load0 ⇐ '1' ;
128:       ConfigOut0(2) ⇐ '1' ;
129:     Fin si
130:     Si ValInE alors
131:       Load1 ⇐ '1' ;
132:       ConfigOut1(2) ⇐ '1' ;
133:     Fin si
134:     Si ValInS alors
135:       Load2 ⇐ '1' ;
136:       ConfigOut2(2) ⇐ '1' ;
137:     Fin si
138:     Si ValInW alors
139:       Load3 ⇐ '1' ;
140:       ConfigOut3(2) ⇐ '1' ;
141:     Fin si
142:     Si WinProp alors
143:       LoadOwn ⇐ '1' ;
144:       ConfigOutOwn(2) ⇐ '1' ;
145:     Fin si
146:     NextState ⇐ sInit ;
147:   Sinon
148:     Congestion ⇐ '1' ;
149:   Fin si
150: Fin si
```

Notons, avant de poursuivre, que la congestion du réseau est détectée grâce à une combinaison des signaux `Congestion`, qui sont en sortie des unités de routage. Dans l'état sInit, ce signal est à '0', et durant la phase d'expansion et de création de chemins, il est à '1' si le contrôleur ne change pas d'état. En combinant les sorties de toutes les unités dans une grande porte ET, il est alors possible de détecter, lors de l'expansion et de la création de chemin, si au moins un contrôleur change d'état. Si tous les contrôleurs restent dans le même état, nous sommes donc en présence d'un problème de congestion, et par ce moyen il est facile de le détecter.

Les encarts 5.1 à 5.2 donnent les équations des différents signaux et bascules du contrôleur de l'unité de routage de HIDRA. Ces équations peuvent directement servir à une implémentation matérielle sur la base de portes logiques, et vont nous servir à comparer les différents algorithmes que nous allons présenter. Bien que leurs comportements ne soient absolument pas identiques, seules des modifications mineures dans quatre de ces équations permettent de passer d'un algorithme à l'autre. Il s'agit des valeurs transmises aux unités de routage voisines, de l'encart 5.2.

États du contrôleur de HIDRA

$$\begin{aligned}
sInit^{+} &= (sInit \wedge \neg OneProp) \vee (sWaitExp \wedge OneProp) \\
&\quad \vee (sFrontExp \wedge OneProp) \\
&\quad \vee (sHasExp \wedge OneProp) \\
sAddress^{+} &= (sInit \wedge OneProp) \vee (sAddress \wedge \neg TriggerIn) \\
sChooseSource^{+} &= sAddress \wedge TriggerIn \\
sWaitExp^{+} &= sChooseSource \vee (sWaitExp \wedge \neg OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate \vee ValInN \\
&\quad \vee ValInE \vee ValInS \vee ValInW)) \\
sFrontExp^{+} &= \neg OneProp \wedge (sWaitExp \wedge (IsSource \wedge \\
&\quad Participate \vee ValInN \vee ValInE \vee \\
&\quad ValInS \vee ValInW)) \\
sHasExp^{+} &= \neg OneProp \wedge (sFrontExp \vee sHasExp)
\end{aligned}$$

Bascules du contrôleur de HIDRA

$$\begin{aligned}
IsConnected^{+} &= IsConnected \vee (sHasExp \wedge WinProp) \vee \\
&\quad (sHasExp \wedge OneProp \wedge IsSource \wedge Participate) \\
Origin(0)^{+} &= ((sWaitExp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (\neg ValInN \wedge (ValInE \vee \neg ValInS \wedge ValInW))) \vee \\
&\quad Origin(0) \wedge \neg(sWaitExp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (ValInN \vee (\neg ValInE) \wedge ValInS))) \\
Origin(1)^{+} &= ((sWaitEsp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (\neg ValInN \wedge \neg ValInE \wedge (ValInS \vee ValInW))) \vee \\
&\quad Origin(1) \wedge \neg(sWaitExp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (ValInN \vee ValInE))) \\
IsMaster^{+} &= (sInit \wedge WinProp) \vee (\neg(sInit \wedge IsMaster)) \\
Participate^{+} &= (Participate \wedge \neg(sInit \vee (sChooseSource \wedge \\
&\quad ((OneProp \wedge \neg IsMaster \wedge IsSource) \vee (\neg OpeProp \wedge \\
&\quad \neg IsMaster \wedge IsTarget))))) \vee (sAddress \wedge \\
&\quad TriggerIn \wedge IsSame \wedge (WantConnect \vee IsSource))
\end{aligned}$$

Equations 5.1: Signaux séquentiels du contrôleur de HIDRA

Implémentation matérielle Le but de nos algorithmes étant d'être implémentés en matériel, nous avons évalué le nombre de transistors nécessaires à leur implémentation. Pour ce faire, nous avons utilisé le synthétiseur Leonardo Spectrum, pour lequel nous avons écrit une librairie de composants simples. Pour chacun de ces composants, décrits par le tableau 5.4, nous avons défini son comportement ainsi que le nombre de

Divers signaux du contrôleur de HIDRA

$$
\begin{aligned}
OneProp &= PropInN \vee PropInE \vee PropInS \vee PropInW \vee PropOwn \\
WantConnect &= \neg IsConnected \wedge (IsSource \vee IsTarget) \\
PropOwn &= (sInit \wedge WantConnect \wedge StartRouting) \vee \\
&\quad (sAddress \wedge IsMaster) \vee \\
&\quad (sChooseSource \wedge IsSource \wedge IsMaster) \vee \\
&\quad (sHasExp \wedge IsTarget \wedge Participate) \\
ValIn &= ValInN \vee ValInE \vee ValInS \vee ValInW \\
ReadAddr &= sAddress \\
CompEnable &= sAddress \\
IsRouting &= \neg sInit \vee OneProp \\
Congestion &= (sWaitExp \wedge \neg(IsSource \wedge Participate \vee ValIn)) \vee \\
&\quad (sFrontExp \wedge OneProp) \vee (sHasExp \wedge \neg OneProp)
\end{aligned}
$$

Valeurs de contrôle des multiplexeurs

$$
\begin{aligned}
ConfigOutN(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
ConfigOutE(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
&\quad \vee(IsSource \wedge Participate \wedge \text{”01”}) \\
ConfigOutS(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
&\quad \vee(IsSource \wedge Participate \wedge \text{”10”}) \\
ConfigOutW(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
&\quad \vee(IsSource \wedge Participate \wedge \text{”11”}) \\
ConfigOutOwn(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
LoadN &= ConfigOutN(2) = sHasExp \wedge OneProp \wedge ValInN \\
LoadE &= ConfigOutE(2) = sHasExp \wedge OneProp \wedge ValInE \\
LoadS &= ConfigOutS(2) = sHasExp \wedge OneProp \wedge ValInS \\
LoadW &= ConfigOutW(2) = sHasExp \wedge OneProp \wedge ValInW
\end{aligned}
$$

Valeurs transmises aux voisines par le contrôleur de HIDRA

$$
\begin{aligned}
ValOutN &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
&\quad \neg ConfigInN(2) \wedge ConfigInN = \text{”00”}) \vee \\
&\quad (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee \\
&\quad Origin = ConfigInN(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”00”}) \\
ValOutE &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
&\quad \neg ConfigInE(2) \wedge ConfigInE = \text{”01”}) \vee \\
&\quad (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee \\
&\quad Origin = ConfigInE(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”01”}) \\
ValOutS &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
&\quad \neg ConfigInS(2) \wedge ConfigInS = \text{”10”}) \vee \\
&\quad (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee \\
&\quad Origin = ConfigInS(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”10”}) \\
ValOutW &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
&\quad \neg ConfigInW(2) \wedge ConfigInW = \text{”11”}) \vee \\
&\quad (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee \\
&\quad Origin = ConfigInW(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”11”})
\end{aligned}
$$

Equations 5.2: Signaux combinatoires du contrôleur de HIDRA

transistors qu’il contient. La synthèse du code VHDL décrivant nos unités de routage s’est ensuite exécutée en demandant à Leonardo d’optimiser le nombre de transistors du design.

Composant	Entrées	Fonction	Transistors
Mux	sel, a, b	$sel * a + \overline{sel} * b$	10
MuxInv	$sel0, sel1, a, b$	$sel0 * a + sel1 * b$	8
Inv	a	$\overline{a}$	2
Nor2	a, b	$\overline{a + b}$	4
Nor3	a, b, c	$\overline{a + b + c}$	6
Nor4	a, b, c, d	$\overline{a + b + c + d}$	8
Nor5	a, b, c, d, e	$\overline{a + b + c + d + e}$	10
Nor6	a, b, c, d, e, f	$\overline{a + b + c + d + e + f}$	12
Nand2	a, b	$\bar{a} + \bar{b}$	4
Nand3	a, b, c	$\bar{a} + \bar{b} + \bar{c}$	6
Nand4	a, b, c, d	$\bar{a} + \bar{b} + \bar{c} + \bar{d}$	8
Nand5	a, b, c, d, e	$\bar{a} + \bar{b} + \bar{c} + \bar{d} + \bar{e}$	10
Nand6	a, b, c, d, e, f	$\bar{a} + \bar{b} + \bar{c} + \bar{d} + \bar{e} + \bar{f}$	12
And2	a, b	$a * b$	6
DF1	clk, d	latch : $Q, \bar{Q}$	18
DFC1	clk, clr, d	Bascule D : $Q, \bar{Q}$	20
DFC2	clk, set, d	Bascule D : $Q, \bar{Q}$	20

Tableau 5.4 : *Composants de la librairie de synthèse.*

Il est bien clair que la valeur fournie par le synthétiseur ne correspond pas forcément au résultat d'un layout dessiné par des ingénieurs, qui ont d'autres possibilités d'optimisation (nous n'avons notamment créé qu'un nombre limité de composants pour notre librairie). Cependant, il permet de se faire une idée sur un ordre de grandeur de la taille d'une unité de routage, et offre un moyen de comparer les unités de routage des différents algorithmes.

Le tableau 5.5[3] résume le nombre de transistors et le nombre de bascules nécessaires à l'implémentation d'une unité de routage, de son contrôleur et de son switchbox. Le nombre de transistors n'y tient pas compte des transistors des bascules, leur réalisation étant extrêmement dépendante de la technologie utilisée.

Composant	Transistors	Bascules
Contrôleur	646	12
Switchbox	292	14
Unité de routage	884	26

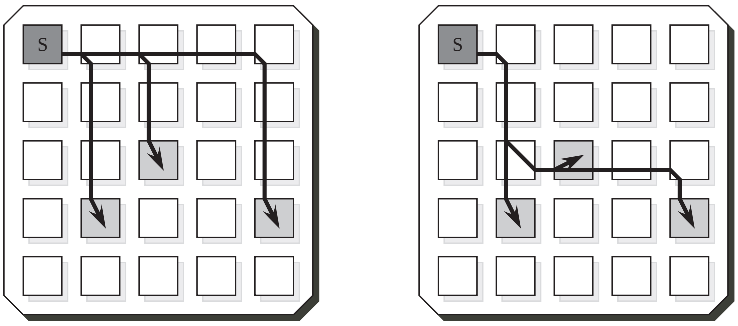
Tableau 5.5 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA.*

---

[3] Le lecteur attentif aura noté que le cumul des valeurs du contrôleur et du switchbox est supérieur au nombre total de transistors. Ce phénomène est dû aux qualités d'optimisation de Leonardo.

5.6 HIDRA-RC

L’algorithme HIDRA procède en lançant la phase d’expansion depuis la source uniquement. La création d’un chemin peut y exploiter un chemin préexistant partant depuis la même source, mais il n’y a pas de priorité donnée aux unités de routage déjà connectées à la source. De ce fait, le chemin trouvé est toujours le plus court, mais le nombre de multiplexeurs réquisitionnés n’est pas forcément optimal, comme le montre la figure 5.20(a). Une priorité étant donnée aux signaux arrivant du Nord lors de la phase d’expansion, les trois destinations de cet exemple se connectent grâce à des chemins allant à l’Ouest, puis au Sud.



(a) Trois destinations reliées à la source, avec HIDRA

(b) Trois destinations reliées à la source, avec réduction des ressources nécessaires

Figure 5.20 : *Comparaison de trois chemins créés avec HIDRA, et la version améliorée HIDRA-RC.*

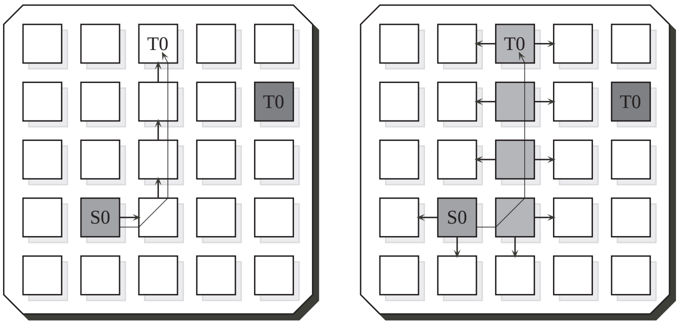
La congestion du réseau de routage, qui voit l’impossibilité de créer certaines nouvelles connexions, de par l’occupation des multiplexeurs, est un problème à éviter absolument. Il sera étudié en détail dans la section 5.11.5, où nous montrerons qu’il est possible de prévoir le nombre de chemins implémentables dans un tableau d’unités de routage de taille définie. Un algorithme capable de minimiser le nombre de multiplexeurs configurés serait donc bienvenu, comme le montre la figure 5.20(b), qui donne une façon de connecter la source à ses trois destinations en minimisant le nombre de multiplexeurs nécessaires, qui passe de 12 à 8.

L’algorithme HIDRA-RC, pour Reduced Congestion, se propose donc d’améliorer les performances de HIDRA en terme de probabilité de congestion. Plusieurs auteurs ont suggéré de lancer la phase d’expansion de l’algorithme de Lee par toutes les cellules présentes sur les chemins reliés à la source à connecter. Nous reprenons cette idée dans HIDRA-RC, en accédant en un coup d’horloge à toutes les unités de routage déjà reliées aux sources participantes. Pour ce faire, nous modifions l’état sWaitExp de la manière suivante :

> Si le contrôleur est une source active, il active `ValOut`*X* si le multiplexeur correspondant est configuré ET qu’il sélectionne la valeur de l’élément externe. S’il n’est pas une source active, et si le contrôleur reçoit une entrée `ValIn`*Y* active, il transmet de manière combinatoire le signal dans

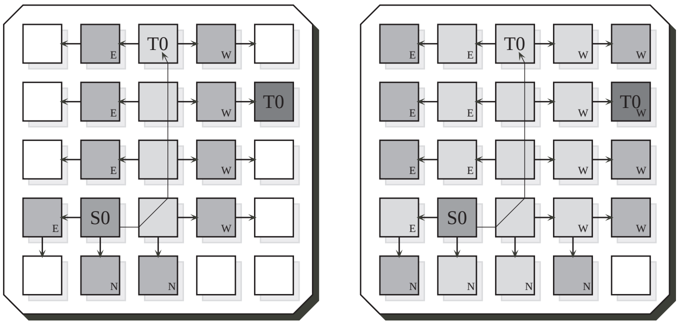
la direction *X* si le multiplexeur correspondant à la direction X est configuré ET s'il est configuré de façon à sélectionner la valeur venant de *Y*.

Cette simple modification implique un changement du comportement des unités de routage, et une propagation combinatoire d'un signal le long de tous les chemins partant de la source active. Au coup d'horloge suivant, la source et toutes les unités de routage atteintes se trouvent dans l'état sFrontExp, et sont dès lors prêtes à lancer l'expansion. La figure 5.21 montre cette phase pour un exemple simple.



(a) Les unités de routage sur un chemin existant sont activées

(b) Expansion 1



(c) Expansion 2

(d) Expansion 3

Figure 5.21 : *Phase d'expansion de l'algorithme HIDRA-RC.*

Les seules modifications à apporter au contrôleur de HIDRA, pour obtenir l'algorithme HIDRA-RC, consistent en les termes en gras de l'encart 5.3. Il s'agit uniquement d'ajouter deux cas où la sortie `ValOutX` passe à '1', comme nous venons de le décrire. Concernant le nombre de transistors nécessaires, présentés au tableau 5.6, il est légèrement plus élevé que pour l'algorithme HIDRA, les équations modifiées étant plus importantes que les originales.

Comme nous l'avons déjà mentionné, nos unités de routage utilisent des liaisons combinatoires pouvant traverser le circuit entier. Cette caractéristique est essentielle à l'implémentation de HIDRA-RC, car toutes les unités de routage présentes sur un chemin relié à la source participante doivent être atteintes en un seul coup d'horloge. En effet, dans un système ne fonctionnant qu'avec des interactions locales, pour obtenir le même résultat, il faudrait un moyen de synchroniser toutes les unités reliées à la source, afin de garantir que toutes les unités lancent simultanément une expansion,

Composant	**Transistors**	**Bascules**
Contrôleur	826	12
Switchbox	292	14
Unité de routage	1078	26

Tableau 5.6 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-RC.*

$$
\begin{aligned}
ValOutN \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInN(2) \wedge ConfigInN = \text{”00”}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee \\
& Origin = ConfigInN(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”00”}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInN} = \textbf{”100”}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInN(2)} \wedge (\mathbf{ConfingInN(1..0)} = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \textbf{”00”})))) \\
ValOutE \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInE(2) \wedge ConfigInE = \text{”01”}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee \\
& Origin = ConfigInE(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”01”}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInE} = \textbf{”101”}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInE(2)} \wedge (\mathbf{ConfingInE(1..0)} = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \textbf{”01”})))) \\
ValOutS \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInS(2) \wedge ConfigInS = \text{”10”}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee \\
& Origin = ConfigInS(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”10”}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInS} = \textbf{”110”}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInS(2)} \wedge (\mathbf{ConfingInS(1..0)} = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \textbf{”10”})))) \\
ValOutW \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInW(2) \wedge ConfigInW = \text{”11”}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee \\
& Origin = ConfigInW(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{”11”}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInW} = \textbf{”111”}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInW(2)} \wedge (\mathbf{ConfingInW(1..0)} = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \textbf{”11”}))))
\end{aligned}
$$

Equations 5.3: Valeurs transmises aux voisines par le contrôleur de HIDRA-RC

après un nombre indéfini de coups d'horloge. Ce problème, connu sous le nom de peloton d'exécution (firing squad), a largement été traité par les mathématiciens et les informaticiens dans les cas d'automates cellulaires à une ou deux dimensions, dans le plan continu ou discret [19, 149, 160, 166, 249]. La synchronisation d'un arbre n'a toutefois pas de solution à base d'automate cellulaire à états finis, et donc HIDRA-RC ne pourrait être réalisé à l'aide d'unités de routage à connexions locales.

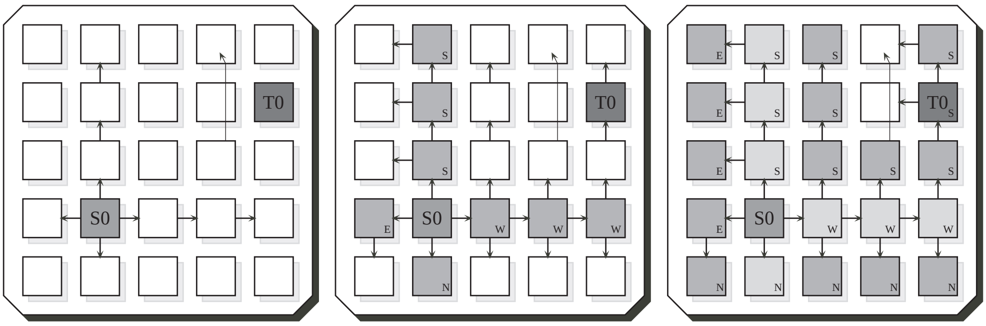
Finalement, notons que HIDRA-RC se comporte exactement de la même manière que HIDRA dans le cas où une source est connectée à au plus une destination. La première connexion à une source est effectivement identique pour les deux algorithmes, HIDRA-RC n'exploitant de nouvelles possibilités que dans le cas de connexions plurielles. Les applications de type réseaux de neurones pourraient en tirer parti, de par le fait que la sortie d'un neurone est, dans la grande majorité des cas, utilisée par plusieurs autres neurones.

5.7 HIDRA-RT

Alors que HIDRA-RC se propose d'améliorer le risque de congestion de HIDRA, une seconde variante permet de réduire le temps d'exécution de l'algorithme. HIDRA-RT, pour Reduced Time, se base sur une recherche de type line-search, inspirée par Mikami et Tabuchi [156]. L'idée est ici de minimiser le temps requis par la phase d'expansion de l'algorithme, qui, dans le cas de HIDRA, nécessite un nombre de coups d'horloge égal à la distance minimal entre la source et la destination.

Alors que dans HIDRA, l'expansion s'exécute pas à pas, où une unité de routage est atteinte après un temps égal à sa distance à la source, dans HIDRA-RT, l'expansion s'effectue par lignes entières. Un contrôleur dans l'état sFrontExp, c'est-à-dire sur le front d'onde, se comporte exactement de la même manière que précédemment, en activant la sortie `ValOut`*X* si cela est possible. Cependant, dans l'état sWaitExp, un contrôleur recevant une entrée `ValIn`*Y* active, transmet immédiatement le signal dans la direction opposée à Y, de façon combinatoire. De ce fait, toutes les unités de routage atteignables par une ligne droite depuis une des unités du front d'onde passent dans l'état sFrontExp au coup d'horloge suivant.

Dans le cas où aucun chemin précédemment routé ne bloque l'expansion en direction de la destination, cette dernière est atteinte en un maximum de deux coups d'horloge, comme illustré à la figure 5.22.



(a) Expansion 1 (b) Expansion 2 (c) Expansion 3

Figure 5.22 : *Phase d'expansion de l'algorithme HIDRA-RT.*

L'avantage de cette approche est donc le temps d'exécution, qui peut être grandement réduit. Toutefois, cette rapidité a un prix, sous la forme de la longueur des chemins. Cet algorithme, de par le fait qu'une ligne est étendue au maximum, tend à minimiser le nombre de virages entre une source et une destination, mais pas forcément la longueur du chemin. Dans le cas où des chemins préexistants bloquent l'expansion courante, il se peut très bien que le chemin trouvé ne soit pas optimal en terme de nombre de multiplexeurs traversés, comme dans l'exemple de la figure 5.23.

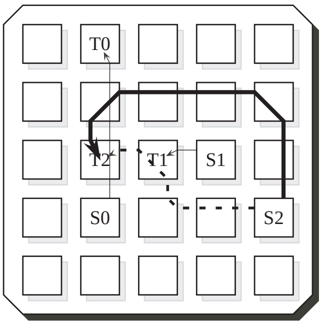


Figure 5.23 : *En gras, le chemin créé par HIDRA-RT (6 muxs), et en traitillé, la solution trouvée par HIDRA (4 muxs), qui minimise la taille du chemin, dans le cas où les sources et destinations 0 et 1 sont déjà connectées.*

Là encore, de même que pour HIDRA-RC, seul le comportement des contrôleurs dans l'état sWaitExp est modifié par rapport à HIDRA, comme ceci :

> Outre sous les conditions présentées dans HIDRA, la valeur `ValOutX` est activée si les conditions suivantes sont réunies :
>
> - l'unité de routage n'est pas une source participante
> - la valeur `ValInY` est active, avec Y étant la direction opposée à X
> - soit le multiplexeur X n'est pas configuré, soit il l'est en sélectionnant la valeur venant de Y.

L'encart 5.4 illustre les modifications apportées aux équations des signaux `ValOutX`, écrites en gras. Là encore, nous pouvons noter qu'une faible modification des unités de routage permet de grandement influencer le comportement de l'algorithme de routage. Le nombre de transistors nécessaires est alors légèrement supérieur à l'implémentation de HIDRA, mais moins importante que pour HIDRA-RC, comme le montre le tableau 5.7.

Composant	**Transistors**	**Bascules**
Contrôleur	754	12
Switchbox	292	14
Unité de routage	958	26

Tableau 5.7 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-RT.*

Finalement, à l'instar de HIDRA-RC, HIDRA-RT ne peut fonctionner qu'avec des liaisons combinatoires pouvant traverser les unités de routage. Dans un système n'acceptant que des connexions locales entre les unités de routage, il est clair que le nombre de coups d'horloge indispensables pour atteindre la destination depuis la source est

$$
\begin{array}{rcl}
ValOutN & = & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
 & & \neg ConfigInN(2) \wedge ConfigInN = ''00'') \vee \\
 & & (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee \\
 & & Origin = ConfigInN(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
 & & \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''00'') \\
 & & \vee(\mathbf{sWaitExp} \wedge (\neg \mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \neg \mathbf{ValInN} \wedge \neg \mathbf{ValInE} \\
 & & \mathbf{ValInS} \wedge (\neg \mathbf{ConfigInN(2)} \vee \mathbf{ConfigInN(1..0)} = ''\mathbf{10}'')) \\
ValOutE & = & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
 & & \neg ConfigInE(2) \wedge ConfigInE = ''01'') \vee \\
 & & (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee \\
 & & Origin = ConfigInE(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
 & & \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''01'') \\
 & & \vee(\mathbf{sWaitExp} \wedge (\neg \mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \neg \mathbf{ValInN} \wedge \neg \mathbf{ValInE} \wedge \neg \mathbf{ValInS} \\
 & & \mathbf{ValInW} \wedge (\neg \mathbf{ConfigInE(2)} \vee \mathbf{ConfigInE(1..0)} = ''\mathbf{11}'')) \\
ValOutS & = & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
 & & \neg ConfigInS(2) \wedge ConfigInS = ''10'') \vee \\
 & & (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee \\
 & & Origin = ConfigInS(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
 & & \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''10'') \\
 & & \vee(\mathbf{sWaitExp} \wedge (\neg \mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \\
 & & \mathbf{ValInN} \wedge (\neg \mathbf{ConfigInS(2)} \vee \mathbf{ConfigInS(1..0)} = ''\mathbf{00}'')) \\
ValOutW & = & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
 & & \neg ConfigInW(2) \wedge ConfigInW = ''11'') \vee \\
 & & (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee \\
 & & Origin = ConfigInW(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
 & & \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''11'') \\
 & & \vee(\mathbf{sWaitExp} \wedge (\neg \mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \neg \mathbf{ValInN} \wedge \\
 & & \mathbf{ValInE} \wedge (\neg \mathbf{ConfigInW(2)} \vee \mathbf{ConfigInW(1..0)} = ''\mathbf{01}''))
\end{array}
$$

Equations 5.4: Valeurs transmises aux voisines par le contrôleur de HIDRA-RT.

au moins aussi grand que la distance minimale les séparant. HIDRA y est donc la meilleure alternative, le principe line-search n'étant efficace que pour des solutions non parallèles ou acceptant des liaisons combinatoires longue distance.

5.8 HIDRA-RTC

Dernière variante proposée, HIDRA-RTC se propose de combiner les approches de HIDRA-RC et HIDRA-RTC afin de réduire le temps d'exécution, et potentiellement le nombre de multiplexeurs réquisitionnés. L'idée y est de faire partir la phase d'expansion de toutes les unités de routage déjà reliées à la source, comme dans HIDRA-RC, et d'effectuer ensuite l'expansion par lignes entières, à la manière de HIDRA-RT.

Les modifications des équations, par rapport à HIDRA, sont un peu plus importantes que pour les deux variantes précédentes, mais ne touchent toujours que les signaux `ValOut`*X*, comme le montre l'encart 5.5.

La taille de l'implémentation matérielle se situe entre HIDRA-RT et HIDRA-RC, avec un total de 1022 transistors et 26 bascules D, comme illustré au tableau 5.8.

Sur le plan de la congestion, nous verrons, lors de l'analyse ultérieure, que HIDRA-RTC est légèrement plus efficace que HIDRA-RT, mais inférieur à HIDRA-RC.

Composant	**Transistors**	**Bascules**
Contrôleur	828	12
Switchbox	292	14
Unité de routage	1048	26

Tableau 5.8 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-RTC.*

5.9 HIDRA-L

Les quatre algorithmes que nous venons de décrire, HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, bien qu'efficacement implémentés en terme de nombre de transistors et nombre de pins, possèdent une faiblesse : la scalabilité. En effet, la grille d'unités de routage peut être facilement étendue, en connectant simplement les unités de routage à leurs voisines, mais les liaisons combinatoires, notamment celles gérées par les unités de propagation, impliquent une réduction de la fréquence d'horloge si la taille de la grille prend des proportions trop importantes. Dans l'optique de pallier à ce problème de scalabilité, nous avons développé un algorithme totalement local, où aucune liaison combinatoire ne parcourt plus d'une unité de routage.

Nous allons décrire en détail cet algorithme, et constater que, bien que plus élégant en terme de scalabilité, il ne sera pas retenu pour la réalisation du circuit POEtic, et ce pour deux raisons. Premièrement, son implémentation nécessite la présence d'un nombre nettement plus important de logique que les quatre versions précédentes, faisant doubler la taille d'une unité de routage. Le circuit POEtic étant limité en terme de surface de silicium, nous avons dû faire le choix de sacrifier un peu de sa scalabilité, pour pouvoir y implémenter un nombre raisonnable d'éléments reconfigurables. Deuxièmement, le nombre de connexions entre deux unités de routage voisines est 2.5 fois plus important, et, dans le cas où plusieurs circuits devaient être reliés entre eux, il n'est pas possible de réduire le nombre de pins nécessaires, contrairement à HIDRA (nous le découvrirons en page 200, dans le cadre de la réalisation du circuit POEtic). Pour un tableau de taille $x \times y$, ce nombre vaut $20x + 20y$, ce qui n'est pas négligeable. Le nombre de pins à disposition pour la réalisation physique du circuit POEtic ayant également été limité, nous ne pouvions décemment garder cet algorithme comme bon candidat.

5.9.1 Algorithme

De manière globale, HIDRA-L (pour Local) fonctionne grâce à cinq mécanismes, à savoir (1) la création d'un espace de routage, (2) la propagation de l'identifiant, (3) la désactivation des concurrents, (4) la création du chemin, qui est décomposée en une phase d'expansion et une phase de rétro-propagation, et (5) la destruction d'un chemin. Ils sont gérés par trois processus concurrents, dont l'un est responsable des mécanismes 2-4, qui sont identiques à ceux décrits dans le cadre de HIDRA, si ce n'est que la propagation des signaux est séquentielle plutôt que combinatoire.

- La création d'un chemin est semblable à celle observée dans HIDRA, et pour qu'elle puisse être menée à bien, il faut que, dans un processus de routage, seules les sources et les destinations répondant à un identifiant unique soient activées.

$$
\begin{aligned}
\mathbf{Follow} &= \mathbf{ValIn} \wedge ((\mathbf{ConfigInN}(\mathbf{2}) \wedge \mathbf{ConfigInN}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{00}'')\vee \\
&\quad (\mathbf{ConfigInE}(\mathbf{2}) \wedge \mathbf{ConfigInE}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{01}'')\vee \\
&\quad (\mathbf{ConfigInS}(\mathbf{2}) \wedge \mathbf{ConfigInS}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{10}'')\vee \\
&\quad (\mathbf{ConfigInW}(\mathbf{2}) \wedge \mathbf{ConfigInW}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{11}'')\vee \\
&\quad (\mathbf{IsConnected} \wedge \mathbf{IsTarget} \wedge \mathbf{ConfigInOwn}(\mathbf{1..0}) = \mathbf{Origin}^{+})) \\
ValOutN &= \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate\wedge} \\
&\quad \cancel{\neg ConfigInN(2) \wedge ConfigInN = ''00'')\vee} \\
&\quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2)\vee} \\
&\quad \cancel{Origin = ConfigInN(1..0)))))\vee}(sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''00'') \\
&\quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInN}(\mathbf{2})) \\
&\quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInN} = ''\mathbf{100}'') \\
&\quad \vee(\neg(\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInN}(\mathbf{2})\wedge \\
&\quad \mathbf{ConfigInN}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{00}'')\vee \\
&\quad (\neg\mathbf{Follow} \wedge \neg\mathbf{ValInN} \wedge \neg\mathbf{ValInE} \wedge \mathbf{ValInS} \wedge \neg\mathbf{ConfigInN}(\mathbf{2})))))) \\
ValOutE &= \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate\wedge} \\
&\quad \cancel{\neg ConfigInE(2) \wedge ConfigInE = ''01'')\vee} \\
&\quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2)\vee} \\
&\quad \cancel{Origin = ConfigInE(1..0)))))\vee}(sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''01'') \\
&\quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInE}(\mathbf{2})) \\
&\quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInE} = ''\mathbf{101}'') \\
&\quad \vee((\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInE}(\mathbf{2})\wedge \\
&\quad \mathbf{ConfigInE}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{01}'')\vee \\
&\quad (\neg\mathbf{Follow} \wedge \neg\mathbf{ValInN} \wedge \neg\mathbf{ValInE} \wedge \neg\mathbf{ValInS} \wedge \mathbf{ValInW} \wedge \neg\mathbf{ConfigInE}(\mathbf{2})))))) \\
ValOutS &= \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate\wedge} \\
&\quad \cancel{\neg ConfigInS(2) \wedge ConfigInS = ''10'')\vee} \\
&\quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2)\vee} \\
&\quad \cancel{Origin = ConfigInS(1..0)))))\vee}(sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''10'') \\
&\quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInS}(\mathbf{2})) \\
&\quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInS} = ''\mathbf{110}'') \\
&\quad \vee((\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInS}(\mathbf{2})\wedge \\
&\quad \mathbf{ConfigInS}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{10}'')\vee \\
&\quad (\neg\mathbf{Follow} \wedge \mathbf{ValInN} \wedge \neg\mathbf{ConfigInS}(\mathbf{2})))))) \\
ValOutW &= \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate\wedge} \\
&\quad \cancel{\neg ConfigInW(2) \wedge ConfigInW = ''11'')\vee} \\
&\quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2)\vee} \\
&\quad \cancel{Origin = ConfigInW(1..0)))))\vee}(sHasExp \wedge OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = ''11'') \\
&\quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInW}(\mathbf{2})) \\
&\quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInW} = ''\mathbf{111}'') \\
&\quad \vee((\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInW}(\mathbf{2})\wedge \\
&\quad \mathbf{ConfigInW}(\mathbf{1..0}) = \mathbf{Origin}^{+} \wedge \mathbf{Origin}^{+} \neq ''\mathbf{11}'')\vee \\
&\quad (\neg\mathbf{Follow}\neg\mathbf{ValInN} \wedge \mathbf{ValInE} \wedge \neg\mathbf{ConfigInW}(\mathbf{2}))))))
\end{aligned}
$$

Equations 5.5: Valeurs transmises aux voisines par le contrôleur de HIDRA-RTC.

Nous créons donc des espaces de routage, à l’intérieur desquels seules les unités de routage répondant à un identifiant participent au processus de création de chemin.

- Dans un espace de routage, le maître désigné envoie son identifiant, puis élimine ses concurrents, de la même façon que dans HIDRA. La phase d'expansion est également identique : seule la configuration des multiplexeurs se fait de manière séquentielle, et non en un seul coup d'horloge.
- Finalement, étant donné que plusieurs unités de routage peuvent désirer se connecter au même instant, les espaces de routage sont concurrents, et un système de priorité permet à un des espaces de l'emporter sur les autres. Lorsqu'un espace est petit à petit détruit, et que la phase de configuration des multiplexeurs n'est pas achevée, il faut déconfigurer les multiplexeurs qui ne sont pas reliés à leur source.

Nous allons passer en revue le détail de fonctionnement de ces trois processus, afin de mieux cerner les subtilités de l'algorithme HIDRA-L.

Création d'un espace de routage

Un système de propagation permet à une unité de routage désirant se connecter d'émettre un signal vers ses quatre voisines, qui au coup d'horloge suivant le transmettront plus loin en suivant les règles du tableau 5.9. L'ensemble des unités atteintes par cette vague constituent ce que nous appelons un espace de routage, et participent ensuite au processus de routage de l'unité centrale. Si plusieurs unités tentent de prendre possession de l'espace de cette façon, une priorité est donnée à celle la plus au Sud-Ouest, son expansion écrasant l'espace de routage réservé par les autres, comme illustré à la figure 5.24.

Règle	**Sorties actives**
Si Sud actif	Nord
Sinon si Ouest actif	Nord, Sud, Est
Sinon si Contrôleur actif	Nord, Sud, Est, Ouest
Sinon si Est actif	Nord, Sud, Ouest
Sinon si Ouest actif	Sud

Tableau 5.9 : *Direction de transmission du signal de propagation, en fonction de son origine.*

Ce mécanisme est prioritaire sur le processus de création de chemins. Lorsqu'un espace de routage $E1$ en écrase un autre $E2$, le processus de routage de $E2$ est détruit au fur et à mesure de l'avancée de $E1$.

Lorsque la création du chemin impliquant l'unité de routage maître de l'espace de routage est terminé, la source de ce chemin relâche l'espace de routage, en propageant un signal actif de manière identique à la création d'un espace de routage. Les unités touchées transmettent ce signal, et se retrouvent à nouveau libres. Dès lors, les unités désirant se connecter peuvent recréer un nouvel espace, pour lancer un nouveau processus de routage, comme illustré à la figure 5.25.

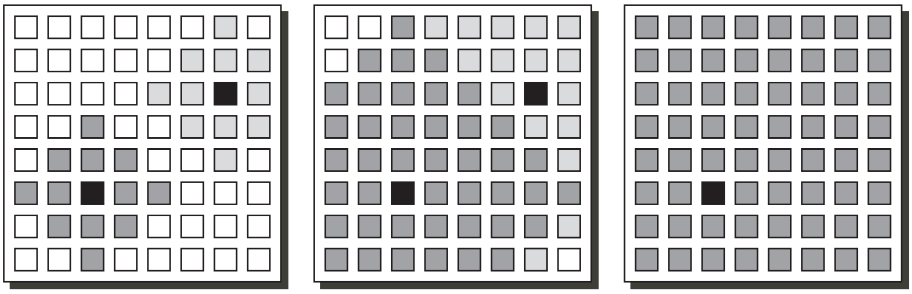


Figure 5.24 : *Trois étapes de la création des espaces de routage, aux temps* t_0+2, t_0+5 *et* t_0+10. *Les deux unités noirs désirant initier une connexion. Les deux couleurs grises indiquent à quel espace de routage une unité appartient.*

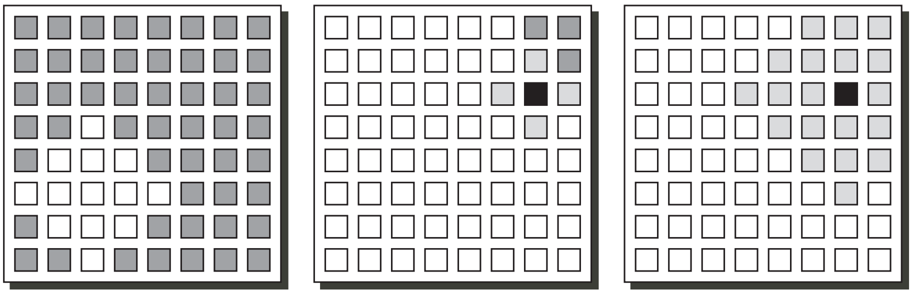


Figure 5.25 : *Destruction d'un espace de routage, aux temps* $t_1 + 2$, $t_1 + 8$ *et* $t_1 + 10$. *L'unité noir désire initier une connexion, et crée un nouvel espace de routage. Les deux couleurs grises indiquent à quel espace de routage une unité appartient.*

Propagation d'identifiant

Immédiatement après avoir émis son bit de propagation, l'unité de routage désirant se connecter envoie son identifiant de manière sérielle à ses voisines, qui le transmettent ensuite à leur voisinage suivant les règles du tableau 5.9. Pour ce faire, un compteur est inclus dans le contrôleur afin de sélectionner un bit de l'identifiant stocké dans l'élément externe. Typiquement, pour un identifiant sur 16 bits un compteur de 4 bits permet d'effectuer cette tâche. Chaque unité de routage recevant les bits de l'identifiant les compare avec son propre identifiant, afin de savoir si elle doit ou non participer au processus courant. Ici encore, le compteur est utilisé afin d'accéder au bon bit de l'identifiant de l'élément externe. A ce stade nous pouvons noter qu'un accès sériel aurait nécessité un nombre moindre de liaisons entre l'unité de routage et l'élément externe, mais qu'il était rendu impossible par l'écrasement des espaces de routage. Le coup d'horloge suivant un écrasement, le premier bit de l'identifiant est reçu et doit immédiatement être comparé avec le bit local. Il faudrait donc pouvoir replacer le registre à décalage de l'élément externe dans son état initial en un seul coup d'horloge, ce qui nécessiterait une grande quantité de logique.

Lorsque l'adresse a été envoyée, l'unité de routage désirant se connecter indique son type (source ou destination) en envoyant un signal dans la même direction que l'identifiant. Il est également transmis par les autres unités et sert à désactiver les unités de routage possédant le même identifiant et le même type. De cette manière nous garantissons qu'un seul chemin est créé à la fois.

Création du chemin

Après que l'adresse ait été envoyée, la phase d'expansion est similaire à celle de HIDRA, où un front d'onde s'étend, à partir de la source, à chaque coup d'horloge jusqu'à atteindre la destination. Lorsque la destination est atteinte, elle lance la phase de rétro-propagation, en envoyant un signal actif dans la direction de l'origine qu'elle a stockée. La voisine transmet ensuite le signal toujours en direction de son origine, tout en configurant le multiplexeur permettant de créer le chemin de données. Ce processus est séquentiel, et nécessite donc un nombre de coups d'horloge identique à la phase d'expansion, qui correspond à la distance entre la source et la destination. Lorsque la source est atteinte, à la fin de cette phase, toutes les unités de routage sur le chemin le plus court entre la source et la destination sont configurées. A ce moment-là elle utilise le même principe que lors de la création de l'espace de routage pour le détruire, laissant ainsi la place à de futurs processus de routage.

Destruction d'un chemin

Finalement, un mécanisme d'effacement de chemin est nécessaire dans le cas où un processus de routage n'a pas le temps de se terminer dans un espace de routage avant que cet espace ne soit supprimé par un autre. Le chemin partiellement créé doit alors être détruit, et il sera reconstruit lors d'un processus ultérieur. Ce cas arrive lorsqu'un chemin est en train d'être construit lors de la deuxième phase de l'algorithme de Lee. Si une unité de routage est atteinte par la rétro-propagation en même temps que par une destruction d'espace de routage, elle envoie un signal dans le sens de sa destination et ne configure pas son multiplexeur. L'unité suivante déconfigure alors son multiplexeur et le signal est transmis jusqu'à ce que la destination soit atteinte. Nous pouvons noter que ce processus est prioritaire sur tous les autres, car un chemin qui a commencé à être détruit doit impérativement l'être jusqu'au bout afin de ne pas réquisitionner inutilement des ressources.

La figure 5.26 montre un espace de routage qui en détruit un dans lequel un chemin est en train d'être créé en phase de rétro-propagation. Les multiplexeurs sont déconfigurés, et un nouveau processus de routage sera nécessaire à la réalisation de ce chemin.

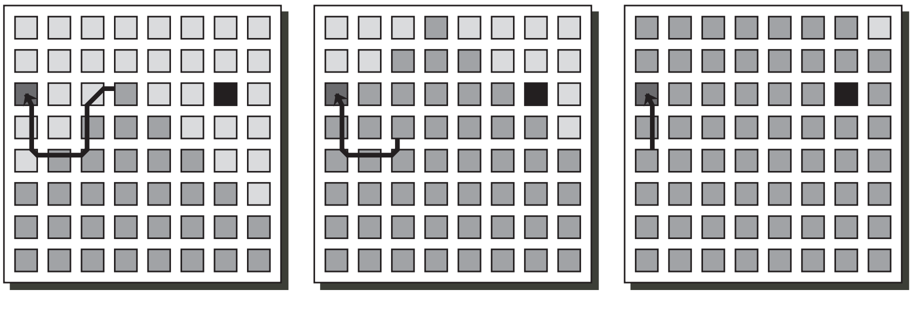


Figure 5.26 : *Destruction d'un espace de routage par un autre, aux temps t_2, t_2+2 et $t_2 + 5$. Le chemin qui était en train de se créer est détruit petit à petit. Les deux couleurs grises indiquent à quel espace de routage une unité appartient.*

Un deuxième cas de figure voit la destruction d'un chemin partiellement créé. Lorsque deux chemins, dans un même espace de routage, sont en phase de rétro-propagation, il faut garantir qu'un seul d'entre eux sera effectivement réalisé, de manière à ce qu'un processus de routage, comme dans HIDRA, ne crée qu'une seule connexion à la fois. Pour ce faire, une unité de routage qui est touchée par la rétro-propagation donne une priorité à un des chemins qui tentent de passer par elle, et détruit les autres.

5.9.2 Implémentation

L'algorithme global explicité, nous pouvons maintenant entrer dans les détails de l'unité de routage. La figure 5.27 nous montre la décomposition de l'unité en un switchbox et un contrôleur. Le switchbox est réalisé à l'aide de cinq multiplexeurs, un pour chacune des directions et un pour le signal à envoyer à l'élément externe, si l'unité est une destination. Chacun des multiplexeurs est contrôlé par deux bits de sélection, stockés dans deux bascules, et un bit supplémentaire sert à indiquer au contrôleur si le multiplexeur est déjà configuré, c'est-à-dire appartient à un chemin de données. Ce switchbox se distingue de ceux précédemment utilisés par la présence de cinq bascules supplémentaires, qui imposent une transmission locale des données, afin de garantir qu'aucun chemin combinatoire ne traverse plusieurs unités de routage.

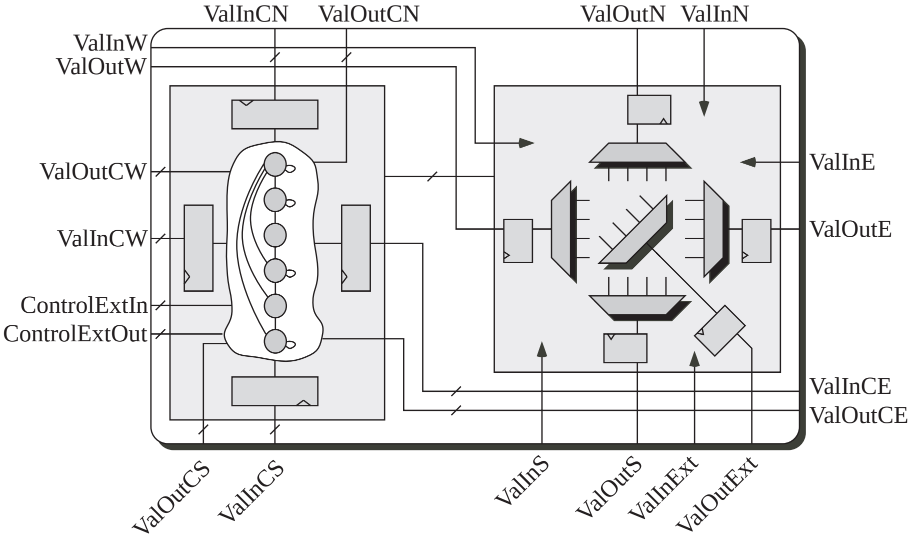


Figure 5.27 : *Une unité de routage de type HIDRA-L.*

Nous pouvons également constater que le contrôleur possède des registres, en entrée, qui en font une machine de Moore. Le reste du contrôleur est réalisé à l'aide d'une machine d'états codée en 1 parmi M. Cette implémentation n'est pas optimale en terme de ressources, mais a servi à démontrer le fonctionnement correct de l'algorithme. Dans le cas où un circuit embarquant cet algorithme devait voir le jour, nous devrions pouvoir la réduire de quelques bascules. Nous n'allons pas donner ici le code correspondant, mais un pseudo-code décrivant les trois processus concurrents intervenant dans le contrôleur. Le premier, la destruction de chemins (Processus 5.10), est prioritaire sur les deux autres. Le deuxième, gérant les espaces de routage (Processus

5.8), est quant à lui prioritaire sur le dernier, qui implémente le mécanisme d'envoi d'adresse et de construction de chemin (Processus 5.9).

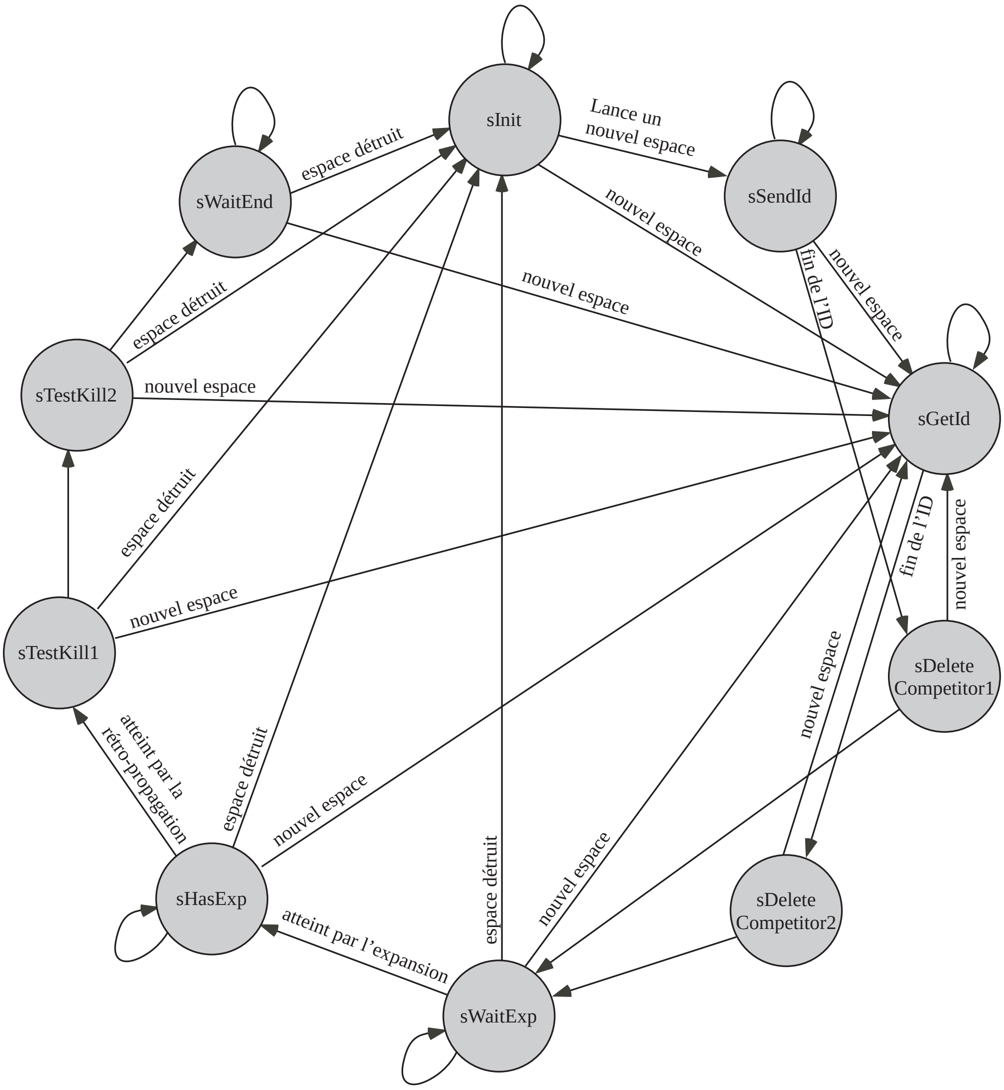


Figure 5.28 : *Machine d'états du contrôleur de HIDRA-L.*

Sur le plan de l'implémentation matérielle, la machine d'état nécessite l'utilisation de 10 bascules en codage 1 parmi M, et plusieurs bascules supplémentaires sont présentes :

- `Participate` : Indique si l'unité de routage participe au routage courant (possède la même adresse que le maître de l'espace de routage et n'est pas du même type que lui).
- `Origin` : Sur 2 bits, mémorise l'origine de la phase d'expansion.
- `PropOrigin` : Sur 2 bits, mémorise l'origine de la phase de création de l'espace de routage.
- `IsConnected` : Indique si l'unité de routage est connectée à son correspondant.
- `IsSame` : Un bit, qui indique si les deux identifiants comparés sont identiques ou non.

- `LastConf` : Sur 3 bits, mémorise quel est le dernier multiplexeur configuré.
- `Counter` : Un compteur, typiquement sur 4 bits, permettant l'accès aux bits d'adresse.

Enfin, des bascules sont ajoutées à chacune des entrées afin d'assurer qu'aucun chemin combinatoire ne traverse le circuit. Dans l'implémentation actuelle nous avons quatre entrées depuis chacune des directions, pour un total de 16 bascules, ce qui implique qu'un côntrôleur comporte 40 bascules.

Sur le plan des liaisons entre unités de routage, un signal par direction sort du switchbox, et est transmis à la voisine. Les contrôleurs nécessitent quant à eux quatre signaux dans chaque direction, qui sont définis de la manière suivante :

- `SpaceIn`*X*, `SpaceOut`*X* : Sur deux bits, est utilisé pour créer et détruire les espaces de routage, envoyer l'identifiant, désactiver les concurrents, et pour l'expansion du front d'onde :

  “11” : Création d'un espace de routage
  “01” : Destruction d'un espace de routage
  “10” : Expansion, bit d'adresse à 1, ou désactivation des sources
  “00” : bit d'adresse à 0, ou désactivation des destinations

- `KillPathIn`*X*, `KillPathOut`*X* : Ce signal s'active pour détruire un chemin qui n'a pas terminé de se former avant d'être avalé par un nouvel espace de routage.
- `RetroPropIn`*X*, `RetroPropOut`*X* : Utilisé lors de la phase de rétropropagation, qui est lancée par la destination lorsqu'elle est atteinte par l'expansion.

Nous allons à présent détailler le comportement des contrôleurs, en commençant par la création et la destruction des espaces de routage. Nous enchaînerons sur leur comportement dans chacun des états de la machine d'états, avant de terminer par la destruction de chemins.

Création/destruction des espaces de routage

Lorsqu'une unité de routage se trouve dans l'état sInit et que son élément externe lui demande de se connecter, le contrôleur lance la création d'un espace de routage, en plaçant `SpaceOut`*X* à la valeur “11” dans chacune des quatre directions, pour autant que `SpaceInS(0)`$= 0$ et `SpaceInW(0)`$= 0$. De ce fait, une priorité est donnée aux espaces étendus depuis les points situés au Sud-Ouest.

Quand une source se trouve connectée grâce au signal de rétropropagation lancé par la destination correspondante, elle place `SpaceOut`*X* à la valeur “01” dans chacune des directions, pour détruire l'espace de routage dans lequel elle se trouve.

Le processus de création d'espace est, comme explicité par la partie d'algorithme 5.8, prioritaire sur celui de destruction d'espace, et sur le plan des directions, la priorité est donnée dans l'ordre suivant : Sud, Ouest, élément externe, Est, et Nord. Les deux bascules de `PropIn` mémorisent l'origine de la précédente vague de création d'espace de routage, de façon à ce que la priorité soit toujours correctement respectée.

Machine d'état

Nous allons maintenant décrire les actions exécutées dans chacun des états du contrôleur, qui sont résumées par la partie d'algorithme 5.9. Nous partons du principe que ces actions sont prises pour autant qu'une création ou une destruction d'espace de

Processus 5.8 HIDRA-L : Contrôleur de l’unité de routage, processus des espaces de routage

```
Si SpaceInS="11" OU SpaceInW="11" alors
    Passer dans l’état sGetId
    Activer SpaceOutY comme décrit dans le tableau 5.9
Sinon si SpaceInS="01" OU SpaceInW="01" alors
    Passer dans l’état sInit
    Activer SpaceOutY comme décrit dans le tableau 5.9
Sinon si l’unité veut se connecter et la machine d’état est dans l’état sInit alors
    Passer dans l’état sSendId
    SpaceOutX="11"
Sinon si l’unité est la source et vient d’être connectée (état sHasExp) alors
    Passer dans l’état sInit
    SpaceOutX="01"
Sinon si SpaceInE="11" OU SpaceInN="11" alors
    Passer dans l’état sGetId
    Activer SpaceOutY comme décrit dans le tableau 5.9
Sinon si SpaceInE="01" OU SpaceInN="01" alors
    Passer dans l’état sInit
    Activer SpaceOutY comme décrit dans le tableau 5.9
Fin si
```

routage n’a pas cours. Si tel n’est pas le cas, le comportement de la machine d’états suit les indications de la partie d’algorithme 5.8.

sInit Le contrôleur reste par défaut dans son état initial. S’il désire initier une connexion, il passe dans l’état sSendId, et s’il n’est pas touché par un nouvel espace de routage, il passe dans l’état sGetId.

sSendId Dans cet état, le contrôleur est maître de l’espace de routage, et envoie son identifiant de manière sérielle, en un nombre de coups d’horloge correspondant à la valeur de son compteur. Il l’incrémente, de façon à accéder le bon bit d’identifiant de l’élément externe, et détecter la fin de l’envoi de l’identifiant, qui le fait passer dans l’état sDeleteCompetitor1. Avant de changer d’état, il fait passer `Participate` à ’1’.

sGetId De même que dans l’état sSendId, le compteur est activé, mais ici l’identifiant reçu est comparé avec celui de l’élément externe. Lorsque le compteur a terminé son comptage, l’état suivant est sDeleteCompetitor2, et si les identifiants sont égaux, `Participate` passe à ’1’.

sDeleteCompetitor1 Si l’unité de routage est une source, elle place `SpaceOut`*X* à "10" dans toutes les directions. L’état suivant est, pour autant qu’il n’y ait pas création d’un nouvel espace de routage, sWaitExp.

sDeleteCompetitor2 Si l’unité de routage est une source, et que `SpaceIn`*Y* vaut "10", alors elle désactive sa participation en plaçant ’0’ dans `Participate`. Si elle est une destination et que `SpaceIn`*Y* vaut "00", elle désactive également sa participation. L’état suivant est sWaitExp.

Processus 5.9 HIDRA-L : Contrôleur de l'unité de routage, processus d'envoi d'adresse et de création de chemin

```
Si (state)
  Egale sInit $\Rightarrow$
    Tous les signaux sont désactivés
  Egale sSendId $\Rightarrow$
    Envoie un bit d'adresse à la fois aux 4 voisines
    Incrémente le compteur
    Si compteur termine alors
      Aller à sDeleteCompetitor1
    Fin si
  Egale sGetId $\Rightarrow$
    Compare le bit reçu avec son adresse et le transmet plus loin
    Incrémente le compteur
    Si compteur termine alors
      Si IsSame=='1' alors
        Participate $\Leftarrow$ '1'
      Fin si
      Aller à sDeleteCompetitor2
    Fin si
  Egale sDeleteCompetitor1 $\Rightarrow$
    Si l'unité est une source alors
      SpaceOut à "10" pour les 4 voisines
    Fin si
    Aller à sWaitExp
  Egale sDeleteCompetitor2 $\Rightarrow$
    Si Participate=='1' alors
      Si (Un SpaceIn est à "10" ET l'unité est une source) OU
      (Pas de SpaceIn à "10" reçu ET l'unité est une destination) alors
        Participate $\Leftarrow$ '0'
      Fin si
    Fin si
    Aller à sWaitExp
  Egale sWaitExp $\Rightarrow$
    Si Participate=='1' et l'unité est une source alors
      Activer le signal d'Expansion pour toutes les voisines
      Aller à sHasExp
    Sinon si Expansion en entrée est actif alors
      Stocker l'origine du signal
      Transmettre le signal aux voisines
      Aller à sHasExp
    Fin si
  Egale sHasExp $\Rightarrow$
    Si (Participate='1' ET l'unité est une destination) OU un RetroPropIn est actif alors
      Configurer le multiplexeur correspondant
      Si Participate='0' alors
        Transmettre la rétropropagation dans la direction de l'origine
      Fin si
      Aller à sTestKill1
    Fin si
  Egale sTestKill1 $\Rightarrow$
    Active KillPathOut dans la direction de LastConf
  Egale sTestKill2 $\Rightarrow$
    Si KillPathIn ne s'active pas alors
      Déconfigure le multiplexeur pointé par LastConf
      Activer KillPathOut dans la direction pointée par LastConf
    Fin si
  Egale sWaitEnd $\Rightarrow$
    Tous les signaux sont désactivés
Fin si égal
```

sWaitExp Si l'unité est une source et que `Participate` est à '1', alors le contrôleur passe dans l'état sHasExp et active le signal d'expansion dans chacune des directions dont le multiplexeur n'est pas configuré de façon à sélectionner une autre valeur que celle de l'élément externe. Dans tous les autres cas, le contrôleur attend d'être atteint par l'expansion. Il stocke alors son origine, transmet le signal dans les directions autorisées, et passe dans l'état sHasExp.

sHasExp Si l'unité est une destination et que `Participate` est à '1', elle active `RetroPropOutY` dans la direction pointée par `Origin`, et configure son multiplexeur sélectionnant la valeur à envoyer à l'unité externe. Toute autre unité de routage attend que `RetroPropInX` soit activé, et si tel est le cas, elle configure le multiplexeur de la direction *X*, et active `RetroPropOutY` dans la direction correspondant à l'origine de l'expansion. Avant de passer dans l'état sTestKill1, le contrôleur stocke, dans `LastConf`, quel est le multiplexeur qui vient d'être configuré. Il est important de noter qu'il est possible, si plusieurs destinations ont été atteintes par la phase d'expansion, que plusieurs chemins tentent de se créer au même instant.

sTestKill1 Le contrôleur active le signal `KillPathOutX`, où X correspond à la direction pointée par `LastConf`, et passe dans l'état sTestKill2.

sTestKill2 S'il ne reçoit pas de signal `KillPathInX`, où X correspond à la direction pointée par `Origin`, il déconfigure le multiplexeur pointé par `LastConf`, et lance la destruction du chemin.

Les deux états sTestKill1 et sTestKill2, qui peuvent sembler surprenants, servent à ne créer qu'un seul chemin par processus de routage. En effet, si plusieurs destinations sont atteintes par la phase d'expansion, elles vont toutes commencer à créer un chemin en direction de la source, en y configurant les multiplexeurs. Il faut absolument éviter qu'ils ne soient tous créés, et donc si nous avons trois unités de routage, U1, U2, et U3, où U2 et U3 sont voisines de U1, que U1 a été atteint par l'expansion avant U2 et U3, et que U2 et U3 sont atteintes en même temps par la rétropropagation (Figure 5.29), il faut que l'une des deux soit déconfigurée. Le comportement illustré par le tableau 5.10 sera alors observé, à chaque nouveau coup d'horloge.

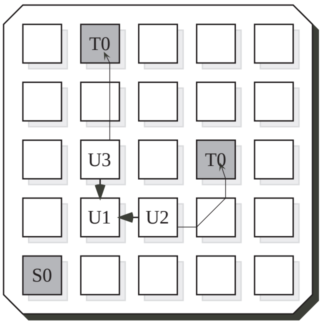


Figure 5.29 : *Exemple de rétropropagation où deux chemins entrent en concurrence.*

sWaitEnd Cet état est atteint par une unité de routage qui est présente sur un chemin qui vient de se créer. Elle y attend que l'espace de routage soit détruit, ou qu'un nouvel espace l'écrase.

U1	**U2**	**U3**	action
sHasExp	sHasExp	sHasExp	U2 et U3 activent le signal de rétropropagation en direction de U1, configurent un multiplexeur, et passent dans l'état sTestKill1.
sHasExp	sTestKill1	sTestKill1	U1 continue la rétropropagation et configure les deux multiplexeurs des directions de U2 et U3, et les trois contrôleurs changent d'état.
sTestKill1	sTestKill2	sTestKill2	U1 active `KillPathOut` dans la direction de U2. U3 ne reçoit pas de signal, déconfigure le multiplexeur qu'il vient de configurer, et active `KillPathOut` dans la direction de la destination courante.

Tableau 5.10 : *Comportement de trois unités de routage voisines, lors de la rétropropagation.*

Destruction de chemin

La destruction d'un chemin partiellement construit s'exécute avec l'aide d'un des quatre signaux de connexion inter-unité de routage, `KillPath`. L'activation de ce signal s'exécute dans trois cas, résumés par la partie d'algorithme 5.10 :

- Si le contrôleur est dans l'état `sTestKill1`, et qu'il ne change pas d'espace de routage, il active `KillPathOut`*X* dans la direction pointée par `LastConf`.
- Si le contrôleur est dans l'état `sTestKill2` et qu'il ne reçoit pas de `KillPathIn`*Y* actif, il active `KillPathOut`*X* dans la direction pointée par `LastConf`.
- Si un signal `KillPathIn`*Y* est actif en entrée et que l'état n'est pas `sTestKill2`, les multiplexeurs sélectionnant cette direction sont déconfigurés, et les `KillPathOut`*X* correspondant à ces multiplexeurs sont activés.

Processus 5.10 HIDRA-L : Contrôleur de l'unité de routage, processus de destruction de chemins

```
Si la machine d'état est dans l'état sTestKill1 alors
    Activer KillPathOut dans la direction pointée par LastConf
Sinon si KillPathIn n'est pas actif et l'état courant est sTestKill2 alors
    Déconfigurer le multiplexeur pointé par LastConf
    Activer KillPathOut dans la direction pointée par LastConf
Sinon si KillPathIn est actif en entrée et l'état n'est pas sTestKill2 alors
    Déconfigurer le multiplexeur sélectionnant cette entrée
    Activer KillPathOut dans la direction du multiplexeur
Fin si
```

Réalisation matérielle

Concernant la réalisation matérielle des unités de routage, elles ont, comme les précédentes, été décrites en VHDL synthétisable. Le tableau 5.11 résume la quantité de logique nécessaire à l'implémentation d'une unité de routage, et nous pouvons constater qu'elle double par rapport aux quatre algorithmes précédents. Bien que plus scalable, cette solution a donc le désavantage de la taille et du nombre de pins nécessaires à la potentielle mise en réseau de plusieurs circuits.

Composant	**Transistors**	**Bascules**
Contrôleur	1828	40
Switchbox	292	19
Unité de routage	2078	59

Tableau 5.11 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-L.*

Dans la suite de ce chapitre, consacré tout d'abord aux voisinages, puis à l'analyse des algorithmes, nous nous concentrons sur les quatre premiers, HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC. Concernant la phase d'analyse, les performances de HIDRA-L sont en effet comparables à celles de HIDRA, tant au niveau des longueurs de chemins et des multiplexeurs réquisitionnés, qu'au niveau des problèmes de congestion. Seul le nombre de coups d'horloge nécessaires à l'exécution de tous les processus de routage diffère, étant donné que la rétro-propagation n'est plus combinatoire, mais séquentielle.

5.10 Voisinages

Les études concernant le routage matériel présentées au chapitre 4 se sont toutes penchées sur le même type de voisinage, à savoir des systèmes à quatre voisines. De même, jusqu'à présent nous nous sommes uniquement concentrés sur des algorithmes faisant intervenir un voisinage de Von Neumann. Toutefois, rien ne prouve qu'il n'existe pas de voisinages plus performants, considérant la probabilité de congestion comparée au nombre de transistors requis pour une implémentation matérielle. Nous allons donc étudier l'impact de différents voisinages sur les performances des quatre algorithmes HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC.

Travaillant sur des systèmes matériels à deux dimensions, et désirant le faire avec des structures régulières, nous allons nous occuper des pavages réguliers du plan. Les trois pavages basés sur des polygones réguliers sont réalisés à l'aide de triangles, carrés et hexagones, correspondant respectivement à des voisinages de 3, 4 et 6. Nous ajoutons à ces possibilités le voisinage de Moore, qui peut aisément être créé avec une structure régulière d'octogones (Figure 5.30).

Il est également possible d'exploiter les unités de routage du voisinage de 8 pour créer un voisinage de 4 où deux liaisons sont établies entre chaque paire de voisines. L'implémentation de base des unités de routage reste identique, seule la connectique entre voisines étant modifiée, comme le montre la figure 5.31. Nous sommes donc en possession d'un voisinage de 4 à double liaison, et ce sans devoir modifier la description des unités. Par la suite, nous désignerons ce voisinage par le sigle 4-2.

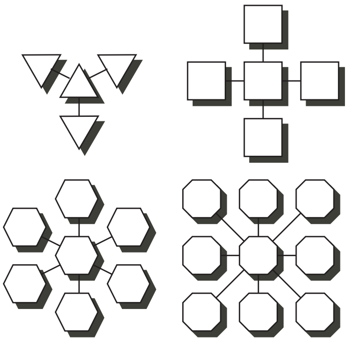


Figure 5.30 : *Les différents types de voisinages utilisés dans cette étude.*

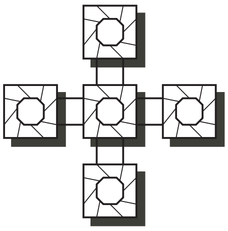


Figure 5.31 : *Un voisinage de 4 avec deux liaisons par couple d'unité de routage.*

5.10.1 Switchboxs

Un des éléments déterminants dans la taille occupée par une unité de routage est le switchbox[4]. Il doit contenir $n + 1$ multiplexeurs pour un voisinage de n : un dans chaque direction, plus un additionnel vers l'élément externe lui étant connecté. De plus, la taille de ces multiplexeurs, et donc le nombre de bascules nécessaires au stockage des configurations, dépend également du voisinage. Le nombre de bits nécessaires à la définition du comportement d'un multiplexeur est égal à $\lceil log_2(n) \rceil + 1$, le "$+1$" correspondant au bit indiquant si le multiplexeur est déjà configuré ou non. Le tableau 5.12 résume ces caractéristiques dans le cas des voisinages traités dans cette thèse. Le nombre total de bascule y est calculé comme $NombreDeMux \times BitsDeConfiguration - 1$, où le "$-1$" correspond au multiplexeur dirigé vers l'élément externe, qui ne nécessite pas d'indiquer s'il est configuré ou non. L'équation en fonction de n est donc : $(n + 1)(\lceil log_2(n) \rceil + 1) - 1$.

Voisinage	**Multiplexeurs**	**Bits de sélection (par mux)**	**Bits de configuration (par mux)**	**Transistors (total)**	**Bascules (total)**
3	4	2	3	192	11
4	5	2	3	292	14
6	7	3	4	650	27
8	9	3	4	982	35

Tableau 5.12 : *Composition d'un switchbox en fonction du type de voisinage.*

---

[4] Une description de la fontionnalité du switchbox a été faite en page 113.

5.10.2 Nombre total de transistors

Le risque de congestion, que nous allons traiter un peu plus loin, est d'autant plus faible que le nombre de voisines d'une unité de routage est élevé, ce qui semble relativement intuitif. Cependant, avant de déclarer qu'un voisinage de 8 est plus efficace qu'un de 3, ou que tel algorithme est meilleur qu'un autre, il faudra prendre en compte le nombre de transistors nécessaires à leurs implémentations respectives. En effet, si une unité à huit voisines est nettement plus efficace qu'une à trois, mais qu'elle nécessite 100 fois plus de matériel pour sa réalisation physique, nous ne pourrons pas arguer qu'elle lui est préférable.

Pour les besoins de la comparaison, nous avons implémenté en VHDL les quatre algorithmes HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, pour chacun des voisinages. Nous ne décrirons pas ici les équations des signaux des contrôleurs des unités de routage, étant donné que leur fonctionnement est semblable à celui du voisinage de 4. Le tableau 5.13 résume les caractéristiques de chacune des 16 implémentations que nous avons réalisées.

	3	4	6	8
HIDRA	736/23	884/26	1476/40	1952/48
HIDRA-RC	862/23	1078/26	1826/40	2422/48
HIDRA-RT	772/23	958/26	1592/40	2070/48
HIDRA-RTC	834/23	1048/26	1748/40	2240/48

Tableau 5.13 : *Nombre de transistors et de bascules d'une unité de routage, pour chaque algorithme, en fonction du voisinage.*

Afin d'être complète, la comparaison doit également tenir compte du nombre de pins nécessaires à l'implémentation d'une grille d'unités de routage de taille $X \times Y$ dans un circuit physique. Dans le voisinage de quatre, nous disposions de deux sorties dans chacune des directions, `ValOutX` et `PropOutX`. Alors que la première doit être présente dans chacune des directions, pour n'importe quel voisinage, la deuxième ne nécessite que d'être transmise dans les quatre points cardinaux. De plus, dans une de ces quatre directions, il est possible de regrouper les signaux `PropOutX` grâce à une porte OU, et donc de minimiser le nombre de pins nécessaires[5]. La ligne de propagation n'occupe que 8 pins : une entrée et une sortie par point cardinal. Le reste des pins ne sert qu'à transmettre les signaux `ValOutX` et `ValInX`. Le tableau 5.14 résume le nombre de pins nécessaires, en fonction de la taille de la grille et du voisinage choisi.

5.10.3 4 versus 8

Avant d'observer les résultats des expériences qui ont été menées, il est intéressant d'analyser le potentiel des voisinages 4 et 4-2. Pour ce faire, nous pouvons grouper quatre unités de routage à 4 voisines à la manière du schéma de gauche de la figure 5.32. Ce groupement correspondant à une super-unité de routage qui possède huit connexions, de la même manière qu'une unité de routage 4-2. Le schéma de droite

---

[5] Cette particularité est explicitée en page 200, dans le chapitre consacré au circuit POEtic.

Voisinage	Pins
3	$2X + 4Y + 8$
4	$4X + 4Y + 8$
6	$8X + 8Y + 8$
4-2	$8X + 8Y + 8$
8	$12X + 12Y$

Tableau 5.14 : *Nombre de pins nécessaires à l'implantation d'une grille de taille* $X \times Y$.

de la figure montre une manière d'organiser des unités à 8 connexions à la façon 4-2 que nous avons définie, en ayant deux liens avec chacune des quatre voisines.

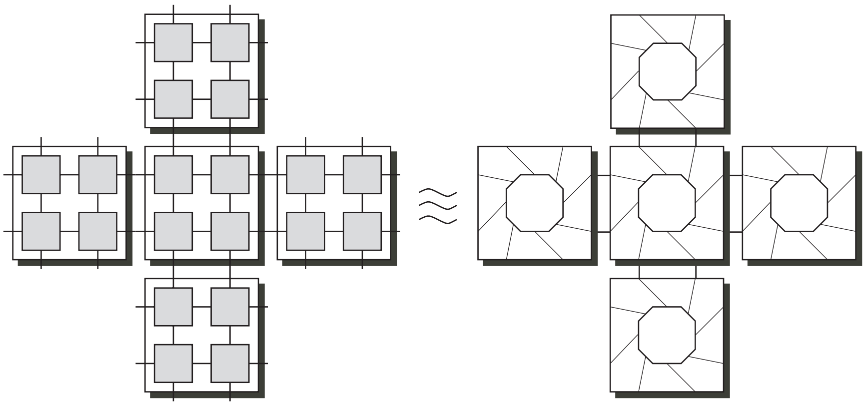


Figure 5.32 : *Regroupement par quatre d'unités de routage à quatre voisines.*

Il est alors aisé de noter qu'une unité 4-2 est plus efficace qu'un groupement de quatre unités à 4 voisines. En effet, la première permet n'importe quelle connectique, chaque sortie pouvant transmettre le signal de n'importe quelle autre entrée, sans aucune contrainte. La deuxième option, en revanche, n'offre pas la même souplesse, car certaines configurations sont impossibles, tel que le montre la figure 5.33. Sur cette figure, en nous conformant au voisinage défini à la figure 5.32, les liaisons seraient équivalentes. Toutefois, la liaison en pointillé ne peut être réalisée avec quatre unités de routage à 4 voisines groupées, alors qu'elle peut l'être avec une unité 4-2.

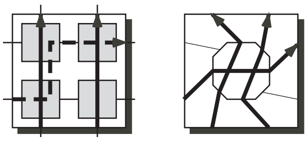


Figure 5.33 : *En suivant les connexions de la figure 5.32, les deux schémas seraient équivalents.*

Avant que les résultats des expériences ne corroborent nos dires, nous pouvons calculer le nombre de connexions impossibles à réaliser par un groupement de quatre unités à 4 voisines. Il est en effet possible, pour un nombre fixé de connexions à réaliser à l'aide des quatre unités, d'en calculer exactement le nombre réalisable. Pour ce faire, nous disposons de huit sorties, chacune d'elles pouvant ou non sélectionner une parmi les sept entrées correspondant autres directions. Une sortie est donc active si elle sélectionne une entrée, et inactive sinon. Il ne reste qu'à compter, pour chaque combinaison de sorties actives, le nombre de sélections d'entrées qui sont réalisables. Le tableau 5.15 expose ces chiffres, pour un nombre de sorties actives compris entre 1 et 8. La première colonne y indique le nombre de sorties actives, la deuxième le nombre de combinaisons réalisables, la troisième le nombre total de combinaisons possibles, et la dernière le pourcentage de succès. Le nombre total de combinaisons est calculé par la fonction suivante, qui correspond au nombre de combinaisons de sorties actives, multiplié par le nombre de combinaisons d'entrées sélectionnées : $\frac{8!}{n!(8-n)!}7^n$.

Nombre de connexions	Succès	Total	Succès/Total (%)
1	56	56	100.0
2	1372	1372	100.0
3	18568	19208	96.6681
4	145830	168070	86.7674
5	673750	941192	71.5848
6	1798392	3294172	54.5931
7	2561950	6588344	38.8861
8	1506243	5764801	26.1283

Tableau 5.15 : *Nombre de connexions possibles.*

Nous pouvons observer, dans ce tableau, que l'efficacité du groupement de quatre unités de routage, en comparaison d'une unité 4-2 baisse grandement avec le nombre de sorties actives. Pour une faible connectivité, les deux approches sont donc équivalentes, et les unités de type 4-2 montrent toute leur efficacité dans le cas d'une forte demande en connexions. Elles sont donc moins sujettes à des problèmes de congestion, comme nous le verrons empiriquement.

Finalement, le voisinage de 8 standard est encore plus performant que le 4-2, de par la possibilité de mettre à profit les diagonales. Ces dernières permettent de réduire la longueur du chemin entre deux points du circuit, et donc de diminuer d'autant le nombre de multiplexeurs réquisitionnés. Alors que dans un voisinage de 4 ou de 4-2, la distance minimale entre deux points $(x1, y1)$ et $(x2, y2)$ correspond à la distance de Manhattan $|x2 - x1| + |y2 - y1|$, le voisinage de 8 offre une distance minimale de $\max(|x2 - x1|, |y2 - y1|)$. Dès lors, comme nous le verrons de manière empirique, le voisinage de 8 est un meilleur candidat que celui de 4-2, puisque pour un même nombre de transistors, il offre un risque plus faible de congestion.

5.11 Analyse

Ayant présenté les quatre algorithmes à comparer, ainsi que les voisinages possibles, nous sommes parés pour entrer dans le vif de l'expérimentation. Nous allons tout d'abord décrire le type d'expériences que nous avons menées, puis nous observerons les différences majeures entre toutes les implémentations, dans le but de définir un optimum en terme de congestion par nombre de transistors. Nous proposerons ensuite un modèle explicatif, capable d'approximer la probabilité de congestion en fonction de divers paramètres, avant de suggérer une analyse grâce à un modèle de percolation, qui nécessiterait une thèse entière à lui seul.

5.11.1 Expérience

Les expériences que nous avons menées se sont effectuées sur la base d'un logiciel, écrit par nos soins, capable de simuler le comportement d'un tableau d'unités de routage. Une interface graphique y offre une visualisation des unités de routage, c'est-à-dire de leur état, et de la configuration de leur switchbox. Deux types de simulations peuvent y être exécutées : purement logicielle, ou basée sur une simulation matérielle des fichiers VHDL.

Dans le premier cas, le logiciel, écrit en C++, simule le comportement exact des unités de routage matériel, en étant capable de calculer le nombre de coups d'horloge nécessaires à l'exécution d'un processus de routage. Dans le deuxième cas, les unités de routage ayant été décrites en VHDL, nous avons utilisé la libraire FLI (Foreign Language Interface), de Modelsim, qui permet d'interfacer une simulation VHDL avec du code C. Une entité peut y être décrite en C plutôt qu'en langage de description matériel, et depuis ce code, il est possible d'accéder à tous les signaux de la simulation, autant en lecture qu'en écriture. Ce code est compilé sous la forme d'une librairie dynamique (DLL), et nous avons utilisé un pipe pour transmettre différentes informations entre la simulation de Modelsim et notre interface graphique, comme nous le montre la figure 5.34.

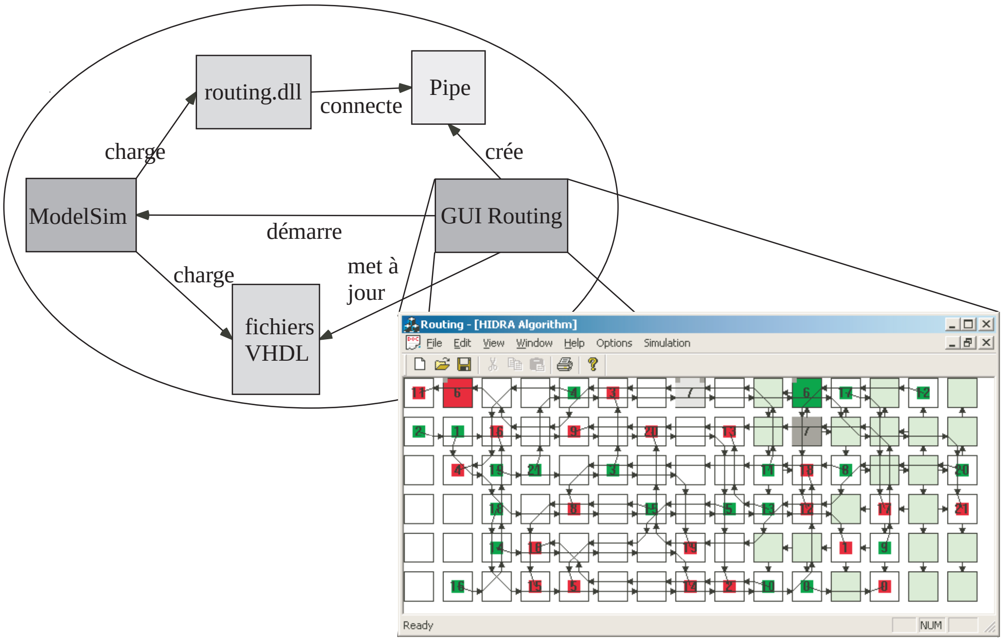


Figure 5.34 : *Lors d'une simulation, l'interface graphique communique avec Modelsim au travers d'un pipe.*

Ce logiciel nous a tout d'abord servit de debugger, lors de la mise au point de nos différents algorithmes. Il s'est ensuite transformé en outil d'expérimentation capable de lancer un grand nombre d'expériences, dans le but de comparer nos différents algorithmes.

Chacune de nos expériences est basée sur le même principe de base. Pour chaque nombre de destinations $ndest$, nous lançons 100 runs, pour chacun desquels nous plaçons à une position aléatoire des sources et des destinations, en fonction de paramètres de l'expérience. Nous effectuons ces 100 runs pour une destination, puis deux, et ainsi de suite, jusqu'à ce que, pour 10 $ndest$ consécutifs, nous obtenions 100% de congestion. Nous partons du principe que si 10 runs consécutifs congestionnent le système, les suivants n'auront pas plus de chance de succès[6]. Les paramètres d'une expérience sont les suivants :

- **Algorithme** : définit le type d'algorithme à utiliser, et peut être HIDRA, HIDRA-RC, HIDRA-RT, ou HIDRA-RTC.
- **Voisinage** : le type de voisinage, qui peut être de 3, 4, 6, 8 ou 4-2.
- **Taille de la grille** : définit la taille du tableau d'unités de routage ($n \times m$).
- **Destinations par source** : définit le nombre de destinations reliées à la même source.
- **Liaisons** : définit le type de liaisons entre sources et destinations. Elles peuvent être libres, auquel cas les sources et destinations sont placées totalement aléatoirement dans le tableau, ou de longueur fixe, la distance minimale entre une source et une destination étant toujours égale à une valeur fixée.

Chacune de nos expériences, pour chaque run, nous fournit les données suivantes :

- **Congestion** : indique si au moins un des chemins n'a pas pu être créé.
- **NbRouted** : Nombre de chemins routés.
- **Clocks** : Nombre de coups d'horloge nécessaire à la terminaison de tous les processus de routage.
- **NbMux** : Nombre de multiplexeurs réquisitionnés par l'ensemble des chemins créés.
- **MaxLength** : Longueur du plus long chemin créé.
- **Longueurs** : Nombre de chemins de chacune des longueurs de 1 à MaxLength.

Il est bien clair que le nombre de paramètres de nos expérience offre un espace de recherche des plus considérables. Nous allons donc nous contenter de comparer les algorithmes entre eux, de même que les différents voisinages. Chaque expérience nous donne, pour chaque nombre de destinations, 100 de ces données, que nous pouvons imprimer en fonction de ce nombre de destination, à la manière des figure 5.35(a) à 5.35(d).

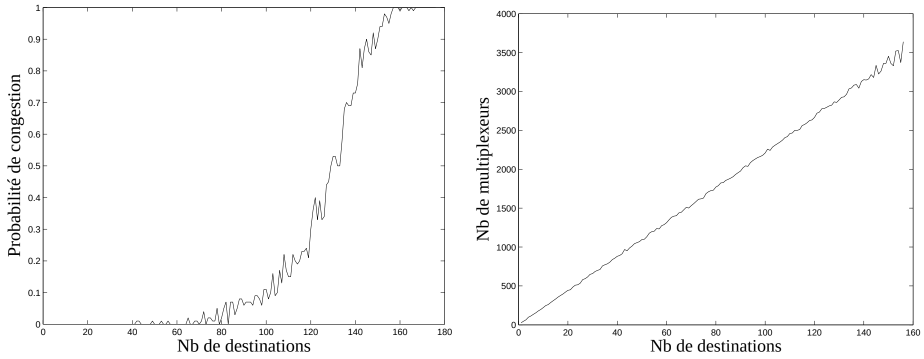
5.11.2 Temps d'exécution

Deux de nos algorithmes, HIDRA-RT et HIDRA-RTC, ont pour but de minimiser le nombre de coups d'horloge nécessaires à l'exécution des processus de routage. Les expériences menées prouvent leur efficacité, mais avant de l'observer empiriquement, passons sur quelques constats théoriques. Le nombre de coups d'horloge pris par un processus de routage, T, peut être décomposé en deux parties : la latence fixe Tf, et la phase d'expansion Te.

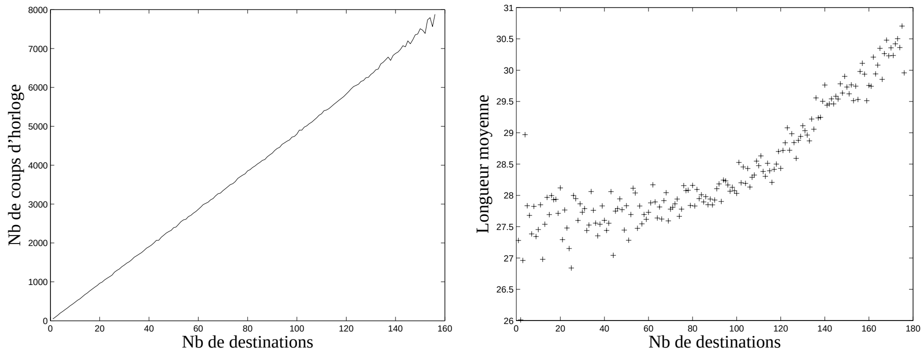
Pour ce qui est de la première, un processus de routage, c'est-à-dire la génération d'une connexion entre une source et une destination voit toujours les contrôleurs des unités de routage passer de l'état sInit à l'état sAddress, dans lequel elle reste un

---

[6] Avant de décider de bloquer les expérimentation après 10 $ndest$ congestionnés, nous avons mené plusieurs expériences afin de vérifier que ce nombre était suffisant.



(a) Probabilité de congestion (b) Nombre de multiplexeurs



(c) Nombre de coups d'horloge (d) Longueur moyenne

Figure 5.35 : *Données récupérées par une simulation de l'algorithme HIDRA, pour une taille de* 40×40*, trois destinations par source, et un placement aléatoire.*

nombre de coups d'horloge correspondant à la taille de l'adresse, puis à l'état sChoose-Source, avant que ne débute la phase d'expansion. Après la terminaison de celle-ci, un coup d'horloge supplémentaire voit la configuration des multiplexeurs lors de la phase de rétropropagation. Chaque destination à connecter nécessite donc un nombre de pas de temps fixe minimum, qui est de $taille(adresse) + 5$. Pour un nombre $ndest$ de destinations, le temps fixe Tf est donc de $ndest \times (taille(adresse) + 5)$. Ce nombre est indépendant de l'algorithme considéré, seule la phase d'expansion étant modifiée entre les quatre variantes étudiées.

Le temps d'exécution de la phase d'expansion, par contre, est influencé par l'algorithme considéré, ainsi que par le nombre de destinations reliées à une même source, dans le cas de HIDRA-RC et HIDRA-RTC.

- Dans le cas de HIDRA, ce temps d'exécution est égal à la distance minimale locale[7] entre la source et la destination. Le temps d'exécution de la phase d'ex-

---

[7] Le chemin trouvé entre la source et la destination ne correspond pas forcément au chemin minimal, étant donné que les multiplexeurs déjà configurés peuvent bloquer certains passages. Il s'agit donc d'un minimum local, et non global.

pansion, Te_{hidra}, peut alors être directement calculé à partir du nombre de destinations à connecter et de la longueur des chemins créés, comme ceci : $Te_{hidra} = \sum_{i=1}^{ndest} length_i$.

- L'algorithme HIDRA-RC propose un temps d'exécution plus faible ou égal à celui de HIDRA, pour la phase d'expansion. Si une source est reliée à au plus une destination, le temps d'exécution est identique, puisque le comportement des deux algorithmes y est similaire. En revanche, si plusieurs destinations sont reliées à une même source, le temps d'exécution peut être réduit de par le fait que toutes les unités de routage déjà reliées à la source sont atteintes en un seul coup d'horloge. Le temps total est donc borné supérieurement par Te_{hidra}, et peut être calculé en fonction du nombre de multiplexeurs utilisés par l'ensemble des chemins. En effet, lors de la première connexion d'une source, le temps de la phase d'expansion correspond à la longueur du chemin créé, et donc au nombre de multiplexeurs réquisitionnés. Les connexions successives de la même source seront exécutées en un temps égal à la longueur du chemin séparant la destination de la plus proche des unités de routage déjà reliées à la source. Et cette longueur équivaut au nombre de multiplexeurs utilisés pour la création du nouveau chemin. Nous avons donc que $Te_{hidra-rc} = \sum mux_{conf}$.
- Concernant HIDRA-RT, le temps d'exécution de la phase d'expansion est nettement inférieur aux deux premières approches. Dans le cas idéal où aucun multiplexeur ne bloque le passage entre la source et la destination, un maximum de 2 coups d'horloge sont nécessaires à la source pour atteindre n'importe quel endroit du tableau d'unités de routage. Si la destination ne peut être atteinte en 2 coups d'horloge, la seule affirmation que nous pouvons tenir est que le nombre de coups d'horloge correspond au nombre de segments droits reliant la source à la destination.
- Enfin, pour HIDRA-RTC, de même que pour HIDRA-RT, dans le cas idéal, un maximum de deux coups d'horloge sont nécessaires à la phase d'expansion. Toutefois, de manière générale, ce nombre ne peut être couplé à aucune autre mesure.

Empiriquement, nous pouvons comparer les temps d'exécution des quatre algorithmes, avec et sans Tf, et ce pour plusieurs paramètres d'expérimentation. Dans le tableau 5.16, nous observons, pour plusieurs tailles de tableau et nombres de destinations connectées à une source (DpS), le temps moyen d'exécution T_m ainsi que le temps d'expansion Te_m d'un processus de routage. Les colonnes %T_m et %Te_m comparent les algorithmes à la solution HIDRA, en terme de temps total, pour un identifiant de 16 bits, et en terme de temps d'expansion. Les sources et destinations sont ici placées au hasard dans la grille d'unités de routage.

5.11.3 Longueur de chemins

A l'instar de la rapidité d'exécution, la longueur des chemins générés par les processus de routage est fortement liée à l'algorithme exploité. Seul HIDRA garantit de trouver le chemin de taille minimale, car l'expansion est lancée par la source uniquement, et, à chaque pas de temps, le front d'onde atteint une nouvelle couche d'unités de routage. Les unités postées à une distance d de la source sont atteintes après d coups d'horloge, ce qui garantit l'optimalité de la solution. Les autres solutions ont pour but l'optimisation d'autres mesures : le nombre de multiplexeurs utilisés pour

Taille	DpS	**HIDRA**		**HIDRA-RC**				**HIDRA-RT**				**HIDRA-RTC**			
		T_m	Te_m	T_m	Te_m	$\%T_m$	$\%Te_m$	T_m	Te_m	$\%T_m$	$\%Te_m$	T_m	Te_m	$\%T_m$	$\%Te_m$
20 × 20	1	34.67	13.67	34.55	13.55	99.6	99.1	23.26	2.26	67.1	16.5	23.24	2.24	67.0	16.4
20 × 20	3	34.75	13.74	30.41	9.41	87.5	68.5	23.29	2.29	67.0	16.7	22.84	1.84	65.7	13.4
20 × 20	5	34.78	13.78	28.79	7.79	83.8	56.6	23.31	2.31	67.0	16.8	22.69	1.69	65.2	12.3
40 × 40	1	47.99	26.99	47.92	26.93	99.9	99.8	23.3	2.30	48.6	8.5	23.31	2.31	48.6	8.6
40 × 40	3	48.16	27.16	39.50	18.50	82.0	68.1	23.41	2.41	48.6	8.9	22.96	1.96	47.7	7.2
40 × 40	5	48.26	27.26	36.21	15.21	75.0	55.8	23.42	2.42	48.5	8.9	22.81	1.81	47.3	6.7
60 × 60	1	61.06	40.06	61.16	40.16	100.2	100.3	23.29	2.29	38.1	5.7	23.30	2.3	38.2	5.7
60 × 60	3	61.51	40.51	48.50	27.50	78.8	67.9	23.43	2.43	38.1	6.0	23.00	2.0	37.4	4.9
60 × 60	5	61.73	40.73	43.55	22.55	70.6	55.4	23.46	2.46	38.0	6.0	22.86	1.86	37.0	4.6
80 × 80	1	74.44	53.44	74.45	53.45	100.0	100.0	23.27	2.27	31.3	4.3	23.28	2.28	31.3	4.3
80 × 80	3	74.81	53.81	57.44	36.44	76.8	67.7	23.45	2.45	31.4	4.6	23.01	2.01	30.8	3.7
80 × 80	5	75.03	54.03	50.87	29.87	67.8	55.3	23.47	2.47	31.3	4.6	22.90	1.90	30.5	3.5

Tableau 5.16 : *Comparaison des algorithmes, en terme de coups d'horloge.*

HIDRA-RC, le temps d'exécution pour HIDRA-RT, et une conjugaison des deux pour HIDRA-RTC. La longueur moyenne des chemins peut alors y être sous-optimale.

Dans le cas de HIDRA-RC, la différence apparaît lorsque plus d'une destination sont connectées à une source. A partir de la deuxième, la longueur du chemin créé n'est pas égale à la distance à la source, mais bien à la distance minimale de la destination à l'une des unités de routage déjà reliées à la source. Ce nombre possède une borne minimale étant la distance entre la source et la destination, mais aucune borne maximale.

Aucune propriété théorique ne peut être énoncée pour les deux options HIDRA-RT et HIDRA-RTC, si ce n'est que la longueur d'un chemin est forcément aussi grande que celle d'un chemin créé avec HIDRA.

Empiriquement parlant, le tableau 5.17 présente les longueurs moyennes des chemins créés, pour les mêmes données que lors de la comparaison des temps d'exécution. Nous donnons ici la longueur moyenne, ainsi que la comparaison avec la solution de base HIDRA.

Taille	DpS	**HIDRA**	**HIDRA-RC**		**HIDRA-RT**		**HIDRA-RTC**	
		L_m	L_m	$\%L_m$	L_m	$\%L_m$	L_m	$\%L_m$
20 × 20	1	15.53	15.51	99.9	15.63	100.7	15.70	101.1
20 × 20	3	15.07	15.75	104.5	15.45	102.5	15.99	106.1
20 × 20	5	15.11	16.62	110.1	15.41	102.0	16.66	110.3
40 × 40	1	29.07	29.35	100.9	29.57	101.7	29.56	101.7
40 × 40	3	28.43	30.10	105.9	29.00	102.0	30.55	107.5
40 × 40	5	28.60	31.90	111.5	29.17	102.0	31.97	111.8
60 × 60	1	42.91	43.03	100.3	43.42	101.2	43.49	101.4
60 × 60	3	41.74	44.39	106.3	42.67	102.2	44.97	107.7
60 × 60	5	42.12	47.14	111.9	42.89	101.8	47.13	111.9
80 × 80	1	57.16	57.01	99.7	57.21	100.1	57.41	100.4
80 × 80	3	55.05	58.74	106.7	56.13	102.0	59.32	107.8
80 × 80	5	55.47	62.41	112.5	56.44	101.7	62.26	112.2

Tableau 5.17 : *Comparaison des algorithmes, en terme de longueur de chemins.*

Ce tableau présente la moyenne générale, sans qu'aucune distinction ne soit faite entre les différents nombres de destinations à connecter. La figure 5.36 montre l'évolution des longueurs, pour un tableau de taille 80×80, en fonction de l'algorithme. Nous pouvons y observer qu'à faible densité de destinations, les variantes HIDRA et HIDRA-RT sont plus efficaces que les deux autres, mais que l'écart se réduit lorsque cette densité augmente. Ceci s'explique aisément par l'arrivée des solutions congestionnées. En effet, nous ne prenons en compte que les routages n'ayant pas subi de congestion, afin de ne pas biaiser les données. La courbe traitillée de la figure 5.36 montre la probabilité de congestion de l'algorithme HIDRA, qui passe de 0 pour les petites densités, à 1 pour les grandes. Plus la densité augmente, plus les solutions HIDRA-RC et HIDRA-RTC peuvent tirer profit de la réutilisation des multiplexeurs, et donc ne pas augmenter la longueur moyenne des chemins. Les deux autres, en revanche, doivent trouver des chemins de plus en plus longs, afin d'éviter les multiplexeurs déjà configurés.

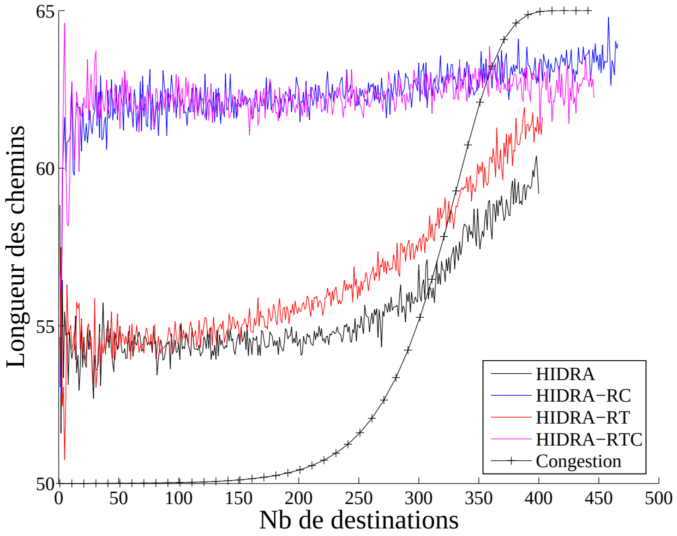


Figure 5.36 : *Comparaison de la longueur des chemins générés par les quatre algorithmes, pour un voisinage de 4, et 5 destinations par source, disposées aléatoirement. La congestion indiquée est celle de HIDRA.*

5.11.4 Nombre de multiplexeurs

Fortement lié à la longueur des chemins, le nombre de multiplexeurs réquisitionnés pour la connexion d'un ensemble de sources à un ensemble de destinations va nous mener tout naturellement vers l'analyse de la congestion. En effet, le nombre de multiplexeurs occupés influe directement sur la probabilité qu'un nouveau chemin ne trouve pas de solution.

Nous pouvons nous attendre à ce que HIDRA-RC soit le plus efficace en regard du nombre de multiplexeurs réquisitionnés, car il réutilise au maximum les chemins déjà créés à partir d'une source. En comparaison de HIDRA-RTC, qui lance également l'expansion depuis toutes les unités de routage déjà reliées à la source, il est au moins aussi efficace, puisque sa phase d'expansion trouve le chemin de longueur minimum entre la destination et l'unité la plus proche parmi toutes les unités déjà reliées à cette source. Il semble également instinctif de prédire que HIDRA-RT occupe le plus grand nombre de transistors, puisqu'il ne garantit pas le chemin minimal, ni ne réutilise les chemins déjà créés.

Le tableau 5.18 compare nos quatre algorithmes, en indiquant le nombre moyen de multiplexeurs configurés par destination, ainsi que le pourcentage en comparaison de HIDRA. Il est intéressant de noter que HIDRA-RT se comporte de façon semblable à HIDRA, probablement de par le fait que dans la plupart des processus de routage, la destination peut être trouvée en deux pas pour HIDRA-RT, et que si tel est le cas, alors la solution est optimale. Si nous revenons au tableau 5.16, nous pouvons en effet constater qu'en moyenne la phase d'expansion s'exécute en 2.5 coups d'horloge, et que donc la solution trouvée est presque toujours très proche de l'optimale.

Concernant les variantes HIDRA-RC et HIDRA-RTC, nos prédictions sont confirmées, et leur nombre de multiplexeurs configurés est nettement inférieur à celui de HIDRA, lorsque le nombre de destinations par source augmente, économisant jusqu'à 25% de multiplexeurs pour un tableau de taille 60×60 et 5 destinations par source.

Taille	DpS	**HIDRA**	**HIDRA-RC**		**HIDRA-RT**		**HIDRA-RTC**	
		L_m	L_m	$\%L_m$	L_m	$\%L_m$	L_m	$\%L_m$
20 × 20	1	13.67	13.55	99.1	13.88	101.5	13.84	101.3
20 × 20	3	11.02	9.41	85.4	11.15	101.2	10.11	91.7
20 × 20	5	9.74	7.79	80.0	9.74	100.0	8.63	88.6
40 × 40	1	26.99	26.93	99.8	27.33	101.3	27.39	101.5
40 × 40	3	22.19	18.49	83.4	22.28	100.4	20.10	90.6
40 × 40	5	19.69	15.21	77.3	19.66	99.9	17.17	87.2
60 × 60	1	40.06	40.16	100.3	40.63	101.4	40.52	101.1
60 × 60	3	33.27	27.50	82.7	33.38	100.3	29.98	90.1
60 × 60	5	29.71	22.55	75.9	29.57	99.5	25.68	86.4
80 × 80	1	53.44	53.45	100.0	53.84	100.7	53.84	100.8
80 × 80	3	44.34	36.44	82.2	44.41	100.2	39.81	89.8
80 × 80	5	39.7	29.87	75.2	39.44	99.4	34.21	86.2

Tableau 5.18 : *Comparaison des algorithmes, en terme de nombre moyen de multiplexeurs configurés.*

La figure 5.37 présente l'évolution du nombre moyen de multiplexeurs configurés par destination, en fonction du nombre de ces destinations, et ce pour 5 destinations reliées à chaque source, dans un tableau de taille 80 × 80. Nous pouvons y observer que les valeurs pour un faible nombre de destinations sont nettement au-dessus du niveau moyen des valeurs à partir de 50 destinations. Ceci est dû au fait qu'un seul chemin à créer ne peut réutiliser aucun multiplexeur. Il est également intéressant, que contrairement aux longueurs de chemins, qui ont tendance à s'allonger avec le nombre de destinations à connecter, le nombre de multiplexeurs est stable, ce qui montre bien que les algorithmes peuvent exploiter les unités de routage déjà reliées à la source du processus courant.

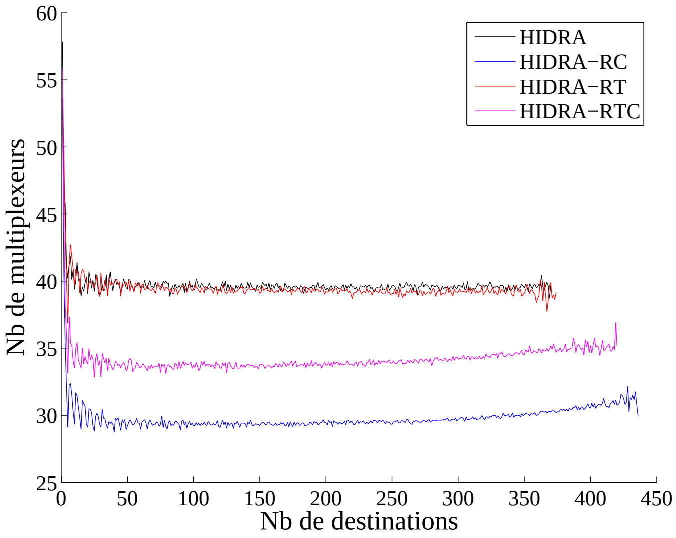


Figure 5.37 : *Comparaison des quatre algorithmes, sur le plan du nombre de multiplexeurs configurés, divisé par le nombre de destinations connectée, pour un voisinage de 4, et 5 destinations par source, disposées aléatoirement.*

Finalement, conformément à nos attentes, HIDRA-RC reste la meilleure solution en terme de nombre de multiplexeurs, et nous amène tout naturellement à analyser la congestion de nos algorithmes, où nous verrons que cet algorithme y est le plus efficace.

5.11.5 Congestion

Alors que les données que nous venons de traiter ne sont pas cruciales, la congestion constitue un point critique de nos algorithmes. Ce problème intervient lorsqu'un nouveau chemin à créer ne trouve aucune solution, c'est-à-dire qu'il n'existe pas de combinaison de multiplexeurs inoccupés, ou reliés à la source, permettant de relier la source à la destination courante. La figure 5.38 illustre un de ces cas, où la source numéro 4 n'a aucune possibilité de joindre la destination de même numéro.

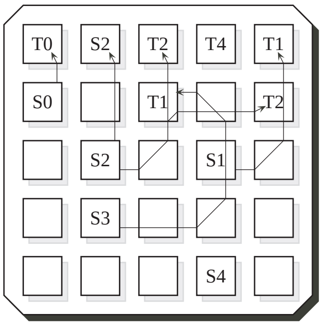


Figure 5.38 : *Exemple de congestion : la source 4 ne peut se connecter à la destination 4.*

Chaque expérience menée nous a fourni, pour chaque nombre de destinations $ndest$, 100 échantillons, caractérisés par un problème de congestion ou non. Il s'agit d'un résultat, qui vaut 1 si l'échantillon est congestionné, et 0 s'il est entièrement routé.

Nous nous intéressons ici à la probabilité de congestion en fonction de $ndest$, de manière à pouvoir prédire si une application particulière, dont on connaît la connectivité, a une chance d'être implémentée sur une grille de taille fixe. Pour un nombre de chemins à router donné, nous avons une probabilité p_c qu'un problème de congestion survienne, et une probabilité $p_{\overline{c}} = 1 - p_c$ que le routage s'effectue normalement. Nous pouvons considérer la variable aléatoire de Bernoulli définie par :

$$C = \begin{cases} 1 & \text{si l'échantillon est congestionné} \\ 0 & \text{si l'échantillon est entièrement routé} \end{cases}$$

L'espérance de cette variable aléatoire est alors :

$$E(C) \quad = \quad \sum_i p_i c_i = 0p_{\overline{c}} + 1p_c = p_c$$

La variance de cette variable aléatoire est ensuite définie comme suit :

$$\begin{aligned} V(C) &= \sum_i p_i(c_i - E(C))^2 = \sum_i p_i c_i^2 - (\sum_i p_i c_i)^2 && (5.1) \\ &= 0^2 p_{\overline{c}} + 1^2 p_c - (0p_{\overline{c}} + 1p_c)^2 = p_c - p_c^2 && (5.2) \end{aligned}$$

Réciproquement, nous avons :

$$p_c = \frac{1 \pm \sqrt{1 - 4V(C)}}{2} \tag{5.3}$$

Pour un échantillon de taille n suffisamment grande, la statistique de C, $\overline{C}$, qui est un estimateur sans biais de p_c, obéit approximativement à une loi du type $N(p_c, p_c(1 - p_c)/n)$. La variance inconnue de C peut alors être remplacée par son estimation $\bar{c}(1 - \bar{c})$, où $\bar{c}$ est une estimation ponctuelle de $p_c = E(C)$.

Nous pouvons donc estimer la probabilité de congestion p_c par la moyenne des résultats de l'échantillon, pour un $ndest$ donné. Désirant trouver une courbe approximant la probabilité de congestion, en fonction de $ndest$, nous avons testé plusieurs modèles explicatifs.

Modèles explicatifs

La courbe de congestion, telle celle illustrée à la figure 5.39, possède deux caractéristiques particulières. Premièrement, il s'agit d'une fonction de transition, semblable à une sigmoïde, ou à une tangente hyperbolique, dont la valeur est incluse dans $[0, 1]$. Deuxièmement, la courbure est plus faible lorsque la probabilité de congestion est inférieure à 0.5 que lorsqu'elle en est supérieure. Une simple sigmoïde ne se prête donc pas parfaitement à un ajustement sur les données, puisque cette fonction est totalement symétrique.

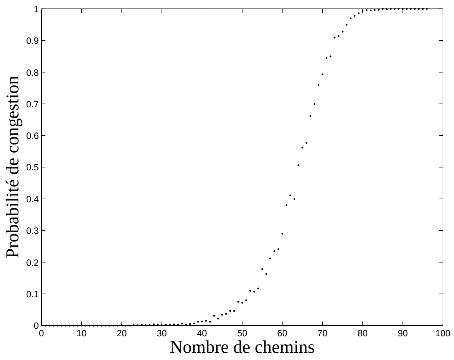


Figure 5.39 : *Probabilité de congestion en fonction du nombre de chemins.*

Dans notre recherche d'une bonne candidate, nous avons tout d'abord tenté des fonctions de type sigmoïde :

$$P_c(x|a, b) = \frac{1 + \tanh(a + bx)}{2} \tag{5.4}$$

$$\begin{aligned} P_c(x|a, b, c, d) = {} & \frac{1 + \tanh(a + bx))}{2} + \\ & (1 - \frac{1 + \tanh(a + bx))}{2})\frac{1 + \tanh(c + dx)}{2} \end{aligned} \tag{5.5}$$

La première est une fonction sigmoïde, et donc symétrique. Elle offre une bonne approximation pour les tableaux de petite taille, mais peine à être efficace lorsque le nombre d'unités de routage devient plus important et que la courbe devient de plus en plus asymétrique. La deuxième combine deux sigmoïdes, afin de pallier au manque de symétrie de la première. Elle permet de nettement mieux approcher les valeurs expérimentales, mais a le désavantage de nécessiter quatre paramètres au lieu de deux. La figure 5.40 montre les deux courbes sur une expérience de taille 80×80, avec voisinage de 4, et 3 destinations par source. La courbe traitillée correspond à l'équation 5.4, tandis que la courbe pleine, qui approxime mieux les données, correspond à l'équation 5.5.

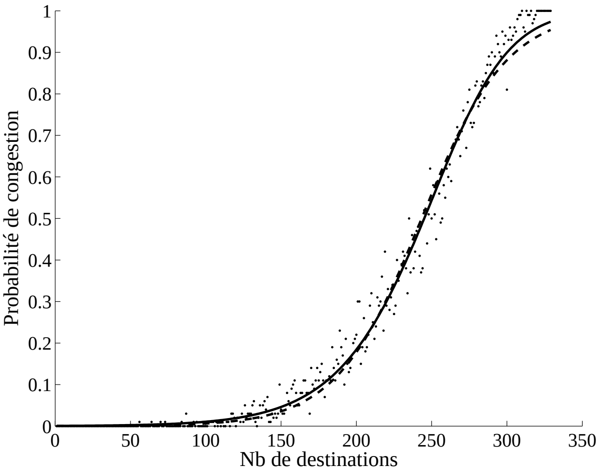


Figure 5.40 : *Comparaison des estimateurs 1 et 2, sur une taille de* 80×80, *4 voisins, 3 destinations par source.*

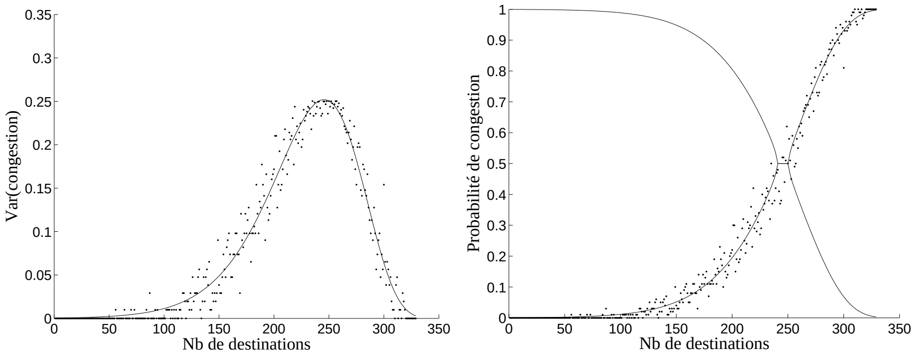
Ces deux courbes ont toutefois l'inconvénient de sous-estimer la probabilité de congestion lorsque celle-ci approche 1. Nous nous sommes alors penchés sur l'observation de la variance des échantillons, en observant qu'avec une symétrie verticale, elle ressemblait fort à une loi de distribution Lognormale. Nous avons donc tenté d'estimer la variance par l'équation 5.6, qui correspond à une loi Lognormale légèrement modifiée, et de calculer ensuite la probabilité de congestion grâce à l'équation 5.7, dérivée de l'équation 5.3.

$$V(x|a,b,\sigma,\mu) = \frac{a}{(b-x)\sigma\sqrt{2\Pi}} e^{\frac{-(\ln(b-x)-\mu)^2}{2\Pi^2}} \tag{5.6}$$

$$P_c(x|a,b,\sigma,\mu) = \frac{1 \pm \sqrt{1 - 4\frac{a}{(b-x)\sigma\sqrt{2\Pi}} e^{\frac{-(\ln(b-x)-\mu)^2}{2\Pi^2}}}}{2} \tag{5.7}$$

La figure 5.41(a) illustre la variance des échantillons, et la courbe calculée grâce à l'équation 5.6. En la transformant, nous obtenons le résultat pour la probabilité à la figure 5.41(b), qui contient deux courbes, correspondant aux deux possibilités de l'équation 5.7. L'estimation est ici très bonne aux probabilités extrêmes, mais manque de justesse lorsque la probabilité approche 0.5, et n'offre pas forcément la possibilité d'en construire une fonction continue.

Dans notre recherche d'une bonne candidate, nous avons finalement posé nos yeux



(a) Variance des échantillons, et courbe approchée

(b) Probabilité de congestion et courbe approchée

Figure 5.41 : *Approximation de la variance par une loi de distribution Lognormale.*

sur la formule de la fonction de distribution cumulative[8] de Weibull :

$$F(x|gamma) = 1 - e^{-(x^{\gamma})}, \text{ pour } x \geq 0; \gamma > 0$$

La courbe de Weibull respecte parfaitement les deux caractéristiques de celle de congestion, et nous l'avons donc choisie comme moyen d'approximer les données reçues des expériences. En la modifiant très légèrement, nous obtenons la fonction suivante :

$$P_c(x|a, b, \gamma) = 1 - e^{-(|ax+b|^{\gamma})} \tag{5.8}$$

La figure 5.42 illustre son adaptation parfaite aux données, la courbe étant continue et offrant la même efficacité que la précédente au niveau des probabilités proches de 0 ou de 1.

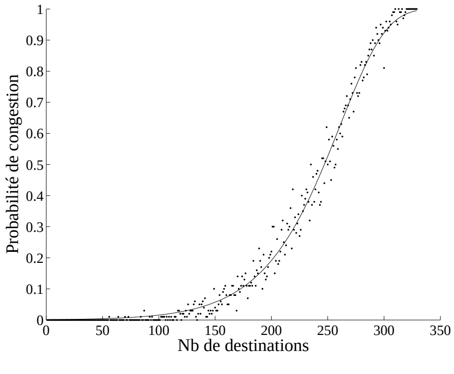


Figure 5.42 : *Probabilité de congestion, et approximation par une fonction de distribution cumulative de Weibull.*

---

[8] Une fonction de distribution cumulative est la probabilité que la variable x (en abscisse) prenne une valeur inférieure ou égale à x.

Analyse

Grâce à l'ensemble des expériences effectuées, nous pouvons à présent déterminer la probabilité de congestion d'un système plaçant des sources et $ndest$ destinations de manière aléatoire dans une grille de taille $n \times n$, pour un nombre de destinations par source de 1, 3, ou 5, en fonction de l'algorithme choisi. Les expériences ont été menées avec des tableaux de taille allant de 5×5 à 80×80. Matlab a été utilisé comme outil de régression non-linéaire capable de nous fournir les paramètres a, b, et γ, en fonction des échantillons récoltés, grâce à la méthode des moindres carrés. Concernant le nombre d'échantillons que nous avons choisi, nous avons mené quelques expériences avec 100 et 1000 échantillons, et avons comparé les différences. Pour un nombre quelconque de destinations, la différence de probabilité de congestion est de moins de 2%, ce qui est tout à fait raisonnable. Nous avons donc décidé de travailler avec 100 échantillons, de manière à accélérer les expériences. De plus, Matlab nous fournit l'intervalle de confiance de la courbe. Pour un niveau de confiance de 95%, et avec 100 échantillons, nous obtenons un intervalle de confiance inférieur à 3%, pour la grande majorité des expériences. Nous pouvons donc dire que la probabilité de congestion se trouve, avec 95% de certitude, sur un point situé à $\pm 3\%$ de la courbe calculée.

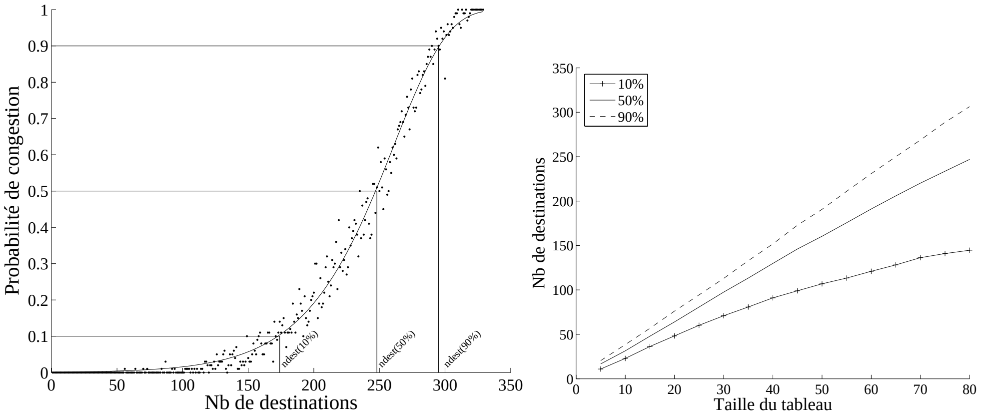
Avec nos expériences, nous disposons de toutes les courbes d'estimation de nos échantillons, et pouvons les exploiter de deux manières. Premièrement, pour une application particulière implémentée dans une grille de taille définie, nous pouvons estimer la probabilité de congestion en fonction du nombre de sources et de destinations présentes. Deuxièmement, nous pouvons établir le nombre de destinations qu'il est possible de connecter en fonction d'une probabilité donnée, grâce à la formule inverse de 5.8 :

$$x(P_c|a, b, \gamma) = \frac{\sqrt[\gamma]{\log(1 - P_c)} - b}{a} \tag{5.9}$$

Nous pouvons, grâce à cette formule, calculer à partir de quel nombre de destinations le risque de congestion devient trop élevé, et utiliser cette valeur pour comparer nos algorithmes, ainsi que les différents voisinages. La figure 5.43 illustre le nombre de destinations pour lequel la probabilité de congestion est de 10, 50 ou 90%, pour l'algorithme HIDRA, 3 destinations par source, et des tableaux de taille $5n \times 5n$, pour $n \in [1, 16]$.

Notre intérêt est plus de comparer les algorithmes et les voisinages que de réellement pouvoir évaluer la probabilité de congestion en fonction des multiples facteurs – algorithme, voisinage, taille en x, taille en y, nombre de destinations par source. Nous disposons en tout de 16 options, les quatre algorithmes combinés aux quatre voisinages, et nous pouvons effectuer des comparaisons deux à deux de la manière suivante : pour les probabilités de congestion de $]0, 1[$, nous calculons le nombre de destinations $ndest1(p_c)$ et $ndest2(p_c)$ correspondantes pour les deux options, puis traçons un graphe de $ndest1(p_c)/ndest2(p_c)$ ou de $ndest1(p_c), ndest2(p_c)$. A titre d'exemple, nous obtenons, pour le cas d'un voisinage de 4, une taille de 40×40, et 3 destinations par source, la comparaison de HIDRA et HIDRA-RC présentée à la figure 5.44. Nous pouvons y observer que HIDRA-RC est plus efficace que HIDRA, ce qui correspond à nos estimations intuitives.

Pour une comparaison générale, nous prenons HIDRA avec un voisinage de 4



(a) Trois points de congestion et leur nombre de destinations correspondant

(b) Nombre de destinations acceptables donnant un taux de congestion donné, en fonction de la taille du tableau d'unités de routage

Figure 5.43 : *Estimation du nombre de destinations acceptables pour un taux de congestion donné.*

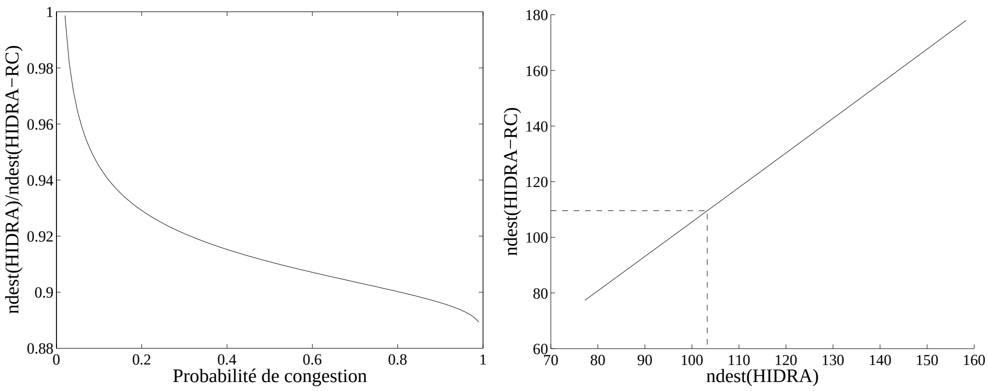


Figure 5.44 : *Comparaison de HIDRA et HIDRA-RC en terme de congestion.*

comme référence, et évaluons les points de congestion 10, 50, et 90% pour toutes les combinaisons $Comb(algorithme, voisinage)$, et transcrivons dans le tableau 5.19 la valeur $Comb(algorithme, voisinage)/Comb(HIDRA, 4)$, et ce pour 1, 3, et 5 destinations par source. Les nombres inscrits dans ce tableau représentent donc l'efficacité d'une implémentation, relativement à (HIDRA,4). Un nombre x signifie que le risque de congestion de y% est atteint pour un nombre de destinations x fois supérieur à celui voyant le même risque pour (HIDRA,4).

Ce tableau nous permet de constater l'efficacité de l'algorithme HIDRA-RC, qui, comme nous l'avions prévu, possède un risque de congestion plus faible que les autres, étant donné qu'il tire un maximum parti des multiplexeurs précédemment configurés. Cette différence est d'autant plus marquée que le nombre de destinations par source est élevé, atteignant un facteur 4.3 pour un risque de 10% et un voisinage de 8. Les trois autres algorithmes ont en revanche des performances équivalentes en terme de

V-D	HIDRA			HIDRA-RC			HIDRA-RT			HIDRA-RTC		
	10	50	90	10	50	90	10	50	90	10	50	90
3-1	0.37	0.41	0.44	0.37	0.41	0.44	0.34	0.39	0.41	0.37	0.39	0.42
3-3	0.38	0.43	0.46	0.41	0.46	0.49	0.30	0.37	0.42	0.33	0.39	0.45
3-5	0.38	0.44	0.47	0.44	0.50	0.53	0.30	0.37	0.43	0.33	0.40	0.47
4-1	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.04	0.99	1.02	1.04
4-3	1.00	1.00	1.00	1.06	1.10	1.12	0.94	0.98	0.99	1.02	1.06	1.07
4-5	1.00	1.00	1.00	1.16	1.16	1.16	0.95	0.98	1.00	1.10	1.11	1.11
6-1	2.28	2.05	1.94	2.30	2.04	1.92	2.16	1.95	1.84	2.13	1.93	1.85
6-3	2.29	2.00	1.89	2.51	2.20	2.07	1.99	1.85	1.75	2.17	1.96	1.87
6-5	2.15	1.93	1.83	2.49	2.21	2.09	1.97	1.85	1.80	2.19	2.00	1.91
8-1	4.01	3.46	3.20	3.97	3.45	3.21	3.45	3.02	2.77	3.47	3.02	2.78
8-3	3.95	3.39	3.15	4.36	3.71	3.43	3.50	3.09	2.91	3.67	3.23	3.03
8-5	3.71	3.24	3.04	4.27	3.71	3.46	3.48	3.10	2.93	3.73	3.28	3.08

Tableau 5.19 : *Comparaison des algorithmes et des voisinages. La première colonne indique le voisinage et le nombre de sources reliées à une même destination.*

congestion, ce qui va dans le sens des résultats que nous avons obtenus en analysant le nombre de multiplexeurs configurés, en page 154.

La grande importance de ces résultats tient plutôt dans la comparaison des voisinages que dans les différences entre algorithmes. Pour ce faire, et en nous référant au nombre de transistors nécessaires à l'implémentation des différentes unités de routage (Tableau 5.13, page 145), nous pouvons mettre en relation l'efficacité en terme de congestion, et le nombre de transistors nécessaires. Trois constatations vont nous donner ces résultats :

- Pour un même algorithme et une même taille de tableau, nous pouvons observer que :
  - Le voisinage de 3 est 2 fois moins performant que le voisinage de 4.
  - Celui de 6 est 2 fois plus performant que celui de 4.
  - Celui de 8 est 3.5-4 fois plus performant que celui de 4.
- Si nous reprenons la figure 5.43(b), nous notons que, pour un risque de congestion donné, le nombre de destinations correspondant est égal ou inférieur à une valeur proportionnelle à n, pour un tableau de taille $n \times n$. En effet, sur cette figure, pour les probabilités de congestion de 50 et de 90%, le nombre de destinations est directement proportionnel à n, et pour 10% de probabilité, la valeur suit une droite légèrement infléchie. Cette affirmation a été vérifiée sur les autres expériences, et nous pouvons donc affirmer, que si nous connaissons $ndest(p_c, n)$, nous pouvons calculer $ndest(p_c, 2n)$ comme étant $2ndest(p_c, n)$.
- Enfin, concernant le nombre de transistors, nous pouvons, sur la base du tableau de la page 145, calculer les valeurs relatives entre les différents voisinages. Le tableau 5.20 nous donne ces valeurs pour l'algorithme HIDRA-RC, où la proportion calculée correspond au nombre de transistors (respectivement bascules) du voisinage de la colonne divisé par celui de la ligne. Un des résultats intéressant y est que nous avons environ un facteur 2 entre les voisinages 3 et 6, et le même facteur entre les voisinages 4 et 8.

A la vue des ces trois points nous pouvons affirmer qu'un tableau de taille $2n \times 2n$

	3	4	6	8
3	1.00/1.00	1.25/1.13	2.12/1.73	2.81/2.08
4	0.80/0.88	1.00/1.00	1.69/1.53	2.25/1.85
6	0.47/0.58	0.59/0.65	1.00/1.00	1.33/1.20
8	0.36/0.50	0.45/0.54	0.75/0.83	1.00/1.00

Tableau 5.20 : *Pour HIDRA-RC, rapport entre les nombres de transistors et les nombres de bascule, en fonction du voisinage.*

avec un voisinage de 8 voit des problèmes de congestion arriver lorsque le nombre de destinations est deux fois plus grand que pour un tableau de $n \times n$. La comparaison des voisinages de 4 et 8 nous donne ceci :

$$\begin{aligned} ndest(p_c, n, voisinage = 8) &= ndest(p_c, 2n, voisinage = 8)/2 \\ &\simeq 2ndest(p_c, 2n, voisinage = 4) \end{aligned}$$

L'implication de ce résultat est grande. Il signifie qu'il est préférable de disposer d'un tableau de taille $\frac{n}{2} \times \frac{n}{2}$ avec un voisinage de 8 que d'un tableau de taille $n \times n$ avec un voisinage de 4, le premier étant 2 fois plus efficace en terme de congestion que le deuxième. De plus, sur le plan de l'implémentation physique, si le nombre de transistors requis par le deuxième est nbt, alors ce nombre n'est que de $nbt \times 2/4 = nbt/2$ pour le premier, sachant que nous avons un facteur 2 entre la taille des unités de routage, et un nombre 4 fois moindre de ces unités présentes dans le tableau. Avec une quantité de logique deux fois moindre, il est donc possible de router deux fois plus de destinations sur un tableau de voisinage 8.

Finalement, notons que nous n'avons pas intégré le voisinage 4-2 dans nos comparaisons, pour la simple raison qu'il est moins efficace que le voisinage de 8, en terme de congestion, alors qu'il nécessite la même quantité de transistors. La figure 5.45 montre la supériorité du voisinage de 8, pour un tableau de taille 40×40, l'algorithme HIDRA, et 3 destinations par source.

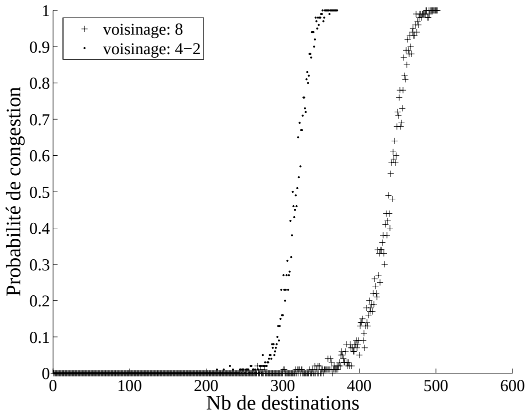


Figure 5.45 : *Comparaison entre un voisinage de 8 et de 4-2.*

5.11.6 Théorie de la percolation

La forme des courbes de congestion montre une caractéristique particulière. Lorsque le nombre de chemins à créer est faible, il est toujours possible de les trouver ; et lorsque ce nombre est très élevé, en proportion du nombre d'unités de routage présentes, aucune solution ne peut être trouvée. Entre ces deux phases, nous observons donc un passage de “tout est routable” à “rien n'est routable”, qui n'est autre qu'une transition de phase. Les transitions de phase peuvent être observées dans nombre de phénomènes physiques, tel le passage de l'eau en glace.

La théorie de la percolation, dont les premiers travaux sont dus à Broadbent et Hammesley [31], sert à analyser des phénomènes macroscopiques liés à des actions microscopiques. Son nom, introduit par Hammersley, vient du percolateur de la machine à café. Le percolateur est chargé de compresser le café moulu, au travers duquel l'eau chaude passe et se charge en caféine. Plus la pression est forte, plus l'eau est en contact avec un grand nombre de particules de café, et plus le café est fort. A partir d'un certain point, appelé point de percolation, l'eau ne peut plus circuler, l'espace entre les particules de café étant trop faible. Au delà de ce point critique, l'eau ne peut donc jamais passer, et en deça elle le peut toujours. De nombreux autres phénomènes peuvent être analysés selon ce point de vue de la percolation. A titre d'exemple, citons les feux de forêts. Les arbres peuvent être modélisés comme des points reliés à leurs proches voisins. Chaque ligne entre deux arbres possède une probabilité de propager le feux, si un des arbres est touché par les flammes. Il est alors possible de prévoir, en fonction de la probabilité de propagation, si toute la forêt risque de brûler ou non, si un incendie se déclare. Au delà d'un point critique, elle brûlera entièrement, et en deça, seule une partie de la forêt sera détruite.

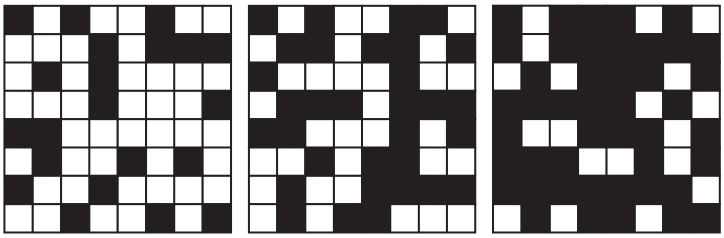
Deux modèles théoriques ont notamment émergé de cette théorie, la percolation de sites, et la percolation de liens[9]. Dans le premier, nous disposons d'une grille d'éléments pouvant être noirs ou blancs avec une probabilité p. La grille, de taille $n \times n$ est remplie d'éléments, chacun d'eux ayant la probabilité p d'être noir (Figure 5.46). Nous nous intéressons à la propriété global consistant à pouvoir relier le haut de la grille au bas en ne passant que sur des cases noires. Si la probabilité p est proche de 0, jamais un cluster noir ne sera assez grand pour relier les deux bords de la grille, et si elle est proche de 1, il existera toujours un cluster assez grand pour le faire. Entre les deux probabilités se situe une transition de phase, passant par le point critique. La caractéristique intéressante d'un tel problème apparaît lorsque n tend vers l'infini. La transition de phase devient de plus en plus petite, et pour $n = \infty$, un point de probabilité critique p_c définit la frontière entre les réseaux ne pouvant jamais relier les deux bords, et ceux qui le peuvent toujours. Un tel problème, pour une grille de voisinage 4, possède un point critique $p_c = 0.6$.

La percolation de liens est relativement semblable. Des points d'une grille carrée sont reliés par des liens, qui peuvent être actifs avec une probabilité p, et inactifs avec une probabilité $1 - p$. Le modèle percole, comme dans le cas de la figure 5.47, si le haut de la grille peut être relié au bas par une suite de liens actifs. Ici, le point critique, lorsque la taille de la grille tend vers l'infini, est de $p_c = 0.5$.

Le modèle de percolation de liens peut être étendu à d'autres voisinages plus complexes, voir même à des graphes quelconques dans le cas des applications de réseaux de télécommunications. Le parallèle avec notre problème de congestion semble évident.

---

[9] Le lecteur intéressé trouvera une excellente introduction à la théorie de la percolation dans [182].



(a) $p(noir) = 0.3$ (b) $p(noir) = 0.5$ (c) $p(noir) = 0.7$

Figure 5.46 : *Modèle de percolation de sites, avec différentes probabilités de coloriage. Les deux réseaux de droite percolent.*

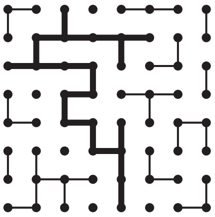


Figure 5.47 : *Modèle de percolation de liens, où nous pouvons observer un réseau percolant.*

Nous nous intéressons effectivement à savoir, si pour un nombre de liaisons source-destination donné, un chemin supplémentaire peut être créé ou non. Nous pourrions dire que le réseau percole lorsqu'une nouvelle connexion ne peut être établie.

Alors que pour le modèle de percolation de liens, une analyse mathématique rigoureuse permet de calculer exactement le point critique, la tâche semble plus ardue dans notre cas. En effet, le réseau entre unités de routage est en fait un réseau entre les multiplexeurs présents dans les switchboxs. Ce réseau directionnel, illustré à la figure 5.48, bien que régulier, montre une complexité nettement supérieure à celle d'une grille de points reliés à leurs quatre voisins. De plus, alors que dans le modèle standard, la probabilité d'activité d'un lien est indépendante des autres liens, dans notre problème de routage, les chemins relient les sources aux destinations, et donc l'occupation des liens entrant et sortant d'un multiplexeurs ne sont pas des variables indépendantes.

Nous suggérons donc une analyse plus poussée du problème de congestion de nos algorithmes, sous l'œil bienveillant de la théorie de la percolation, qui pourrait aisément conduire à une nouvelle thèse, tant les paramètres sont vastes : taille de la grille, type d'algorithme, nombre de destinations par source, longueur moyenne des chemins, ...

Nous avons, dans les pages précédentes, mené une analyse dans ce sens, en tentant de caractériser nos algorithmes en fonction d'une probabilité qu'une unité de routage soit une destination (p_{dest}), et en fonction du nombre de destinations reliées à la même source. Le nombre total de destinations, pour un tableau de taille $n \times m$ est défini par $ndest = p_{dest} \times n \times m$. Il serait alors intéressant de voir émerger une propriété de congestion indépendante de la taille de la grille, uniquement en fonction de la densité de destinations.

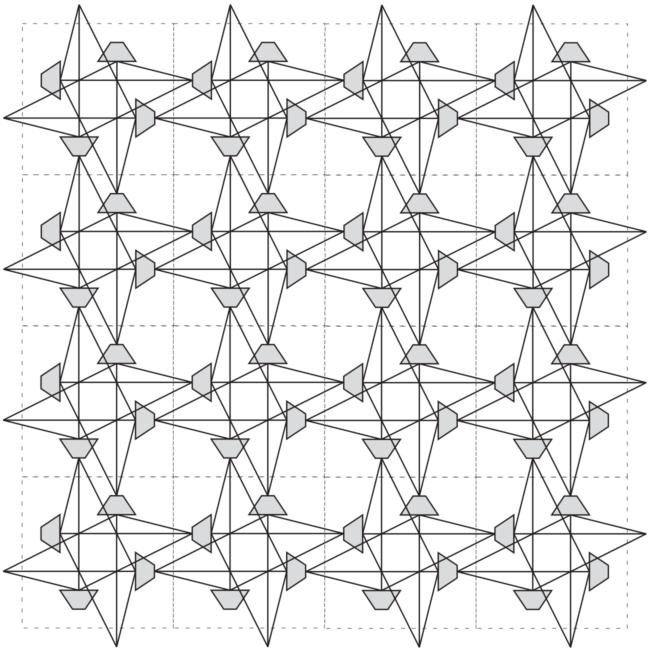


Figure 5.48 : *Réseau directionnel d'unités de routage (carrés traitillés) avec un voisinage de 4.*

Notons que le type d'expériences à mener, pour observer un phénomène de percolation mettant en jeu un point critique indépendant de la taille du réseau, devrait se faire en définissant une longueur moyenne de chemin. En effet, la pose aléatoire de sources et destinations dans le plan de taille $n \times n$ a pour effet de créer des chemins dont la longueur moyenne est proportionnelle à n, ce qui est le cas dans toutes nos expériences précédentes.

Dans le cas d'une longueur moyenne l fixe, avec $ndest$ destinations reliées à différentes sources, la probabilité qu'un lien entre deux multiplexeurs soit occupé est proportionnelle à $\frac{ndest \times l}{n^2} = \frac{p_{dest} \times n^2 \times l}{n^2} = p_{dest} \times l$. Dans le cas d'une pose aléatoire, il est en revanche proportionnel à $\frac{ndest \times n}{n^2} = \frac{p_{dest} \times n^3}{n^2} = p_{dest} \times n$, donc dépendant de la taille du tableau. Plus cette taille augmente, plus le risque de congestion est élevé.

Lors des comparaisons des probabilités de congestion entre différentes tailles de tableau, il faudrait donc prendre garde à bien traiter la probabilité qu'un lien soit occupé, de manière à pouvoir faire émerger une caractéristique indépendante de la taille du tableau d'unités de routage. Comme piste de recherche, nous pouvons nous intéresser à nos résultats précédents. La figure 5.43(b), en page 161, montre que l'apparition du problème de congestion apparaît à un nombre de destinations proportionnel à n. Ceci correspond exactement au fait que la longueur des chemins générés par nos expériences est justement proportionnelle à n. En divisant le nombre de destinations, correspondant à un niveau de congestion donné, par n, nous obtenons des droites de pente 0, qui, pour les faibles niveaux de congestion, ont tendance à baisser lorsque la taille du tableau augmente.

La figure 5.49 est identique à la figure 5.43(b), si ce n'est que le nombre de destinations a été divisé par la taille n du tableau. Pour les grandes valeurs de probabilité de congestion, il s'agit d'une droite de pente nulle, ce qui tend à prouver que nous sommes

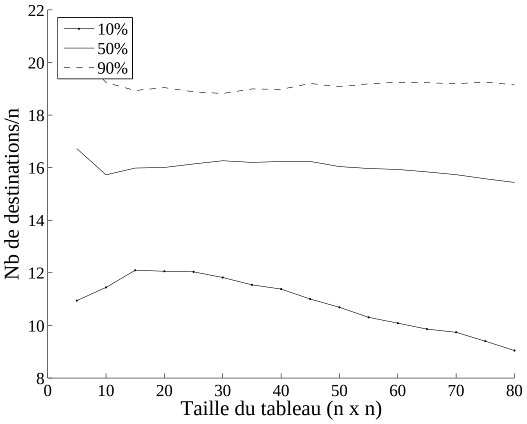


Figure 5.49 : *Nombre de destination correspondant à une probabilité de congestion donnée, divisé par la taille* n *du tableau, en fonction de la taille* $n \times n$ *du tableau (HIDRA, 3 destinations par source).*

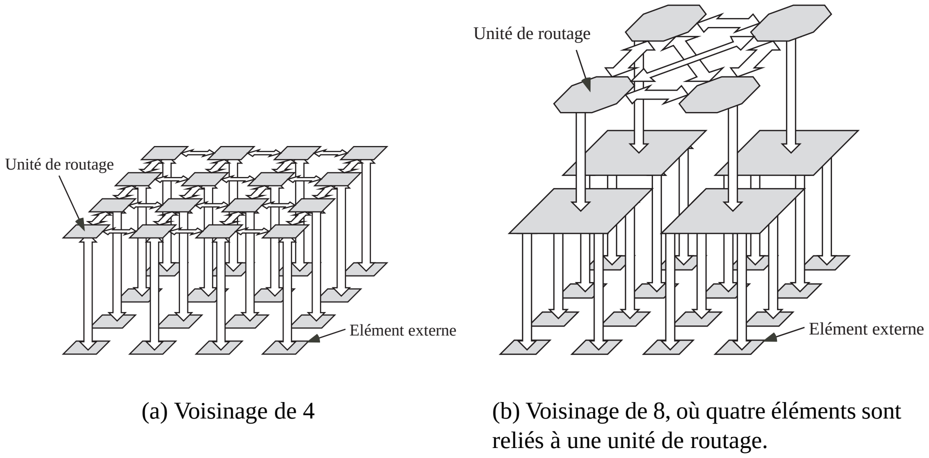
en présence d'un phénomène indépendant de la taille du tableau, pour une longueur de chemins moyenne fixe. Cependant, un nombre considérable d'expériences supplémentaires, et de recherches théoriques, seraient nécessaires pour confirmer ces dires, et pour caractériser le problème de congestion comme étant un modèle de percolation.

5.12 Conclusion

Nous venons de présenter cinq algorithmes de routage distribué, dont trois d'entre eux sont des variations sur le premier, HIDRA. Leurs caractéristiques ont été étudiées, et plus particulièrement le risque de congestion. En effet, notre mécanisme de routage est dynamique en comparaison du routage calculé par un logiciel spécialisé, qui est ensuite chargé de manière à configurer un circuit programmable. Il est toutefois statique en comparaison des méthodes de type *wormhole* [177] ou *packet switching*, où des paquets d'information transitent au travers d'un réseau de façon dynamique, leur acheminement pouvant prendre des chemins différents d'une fois à l'autre. Nos algorithmes se chargent de configurer des multiplexeurs, qui sont ensuite utilisés pour transmettre de l'information par des chemins fixes. Dès lors, il est possible, lorsqu'un nombre trop élevé de multiplexeurs sont configurés, que la création d'un nouveau chemin se trouve bloquée sans trouver de solution.

Le circuit POEtic, que nous décrivons dans le prochain chapitre, embarque un algorithme de routage, qui se trouve être HIDRA, et ce avec un voisinage de 4. Nous venons cependant de prouver qu'il aurait été préférable de baser notre réalisation sur un voisinage de 8. En effet, en connectant quatre éléments externes à une unité de routage de voisinage 8 au lieu d'un élément par une de voisinage 4, pour un nombre total de transistors deux fois moins élevé, cette option offre un nombre de connexions potentiellement routables deux fois plus élevé (Figure 5.50(b)). Cette analyse n'a malheureusement été effectuée qu'après le lancement de la fabrication du circuit, et nous ne pouvons que suggérer l'emploi de l'option à 8 voisines dans le cas où un deuxième circuit de ce type devait voir le jour.

Toujours sur la question du voisinage, il serait intéressant d'approfondir celui de



(a) Voisinage de 4

(b) Voisinage de 8, où quatre éléments sont reliés à une unité de routage.

Figure 5.50 : *Tableaux à voisinage de 4 et 8, dont celui de 8 est deux fois plus efficace en terme de congestion.*

6, en exploitant les propriétés des membranes d'eau savonneuse, décrites en page 69. Trois parois d'eau savonneuse se croisent toujours en des angles de 120 degrés, et une telle membrane reliant plusieurs points s'organise de manière à créer un arbre de Steiner minimal (ou qui correspond à un minimum local). La figure 5.51 illustre la manière dont plusieurs sources reliées à une destination pourraient l'être grâce à un chemin total minimum, en le modifiant itérativement. Le schéma de gauche correspond à la solution trouvée par HIDRA, et les deux autres sont des modifications du chemin menant au chemin de taille totale minimum, qui utilise deux multiplexeurs de moins que la solution initiale. Un tel mécanisme permettrait d'économiser le nombre de multiplexeurs réquisitionnés lorsque plusieurs destinations sont reliées à une même source, et donc de réduire substantiellement le risque de congestion.

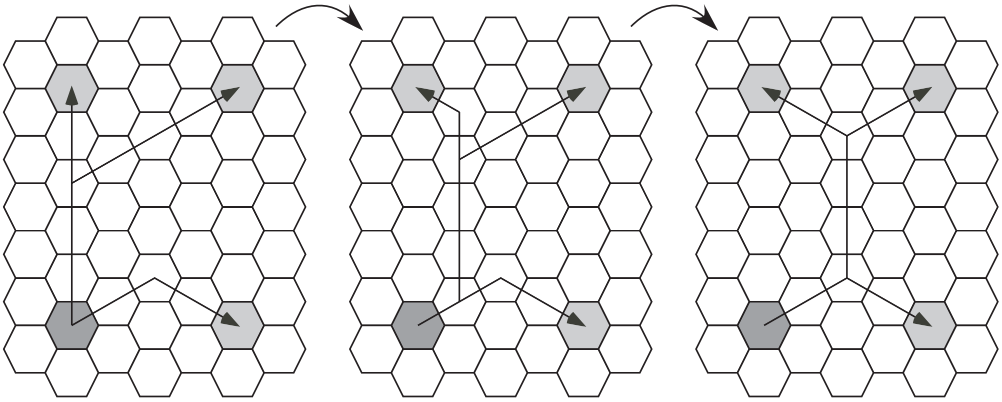


Figure 5.51 : *Evolution des chemins entre une source (en bas à gauche) et trois destinations, de manière à construire un arbre de Steiner de poids minimum.*

Concernant la scalabilité des quatre premiers algorithmes, HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, nous pouvons noter que les chemins combinatoires traversant les unités de routage obligent à baisser la fréquence d'horloge du système pour que son fonctionnement ne se trouve pas altéré. L'algorithme HIDRA-L a apporté une solution synchrone à ce problème, en transformant les unités de routage en machines de Moore. Cette implémentation possède toutefois le désavantage de nécessiter un nombre de transistors qui, pour un voisinage de 4, double par rapport à l'algorithme de base HIDRA.

Enfin, il serait possible de concevoir un algorithme semblable à HIDRA-L, basé sur des mécanismes asynchrones. Il serait alors possible de disposer d'un système totalement scalable, sans soucis de synchroniser une horloge. Bien que potentiellement élégante, cette solution se distinguerait toutefois par un nombre de transistors sans doute encore plus important que pour HIDRA-L.

Chapitre 6

Le Circuit POEtic

> C'est pas avec des idées poétiques qu'on fait une révolution.
>
> Bertrand BONELLO , *Dialogue du film Le pornographe*

DANS la section 3.5, page 62, nous avons défini une architecture de cellule POEtic. Un tissu POEtic, embarquant de telles cellules, ainsi que des capacités d'évolution, pourrait évidemment être créé pour une application particulière, sous la forme d'un circuit intégré composé d'un tableau de cellules déjà spécialisées, et d'un processeur pouvant y faire évoluer des organismes artificiels. Cependant, chaque nouvelle application impliquerait la réalisation d'un nouveau circuit, avec le temps de développement et les coûts que cela implique. Il serait dès lors nettement plus avantageux de disposer d'un substrat reconfigurable sur lequel nous pourrions implémenter n'importe quel type de cellules.

Le circuit POEtic [168], dont la partie reconfigurable a été développée dans le cadre de cette thèse, a été conçu dans ce but, et propose des mécanismes facilitant l'implémentation de ces systèmes multicellulaires bio-inspirés. Un tableau d'élément simples, que nous appelons molécules, y sert à accueillir les cellules, qui ensemble forment un organisme (Figure 6.1). Ce substrat reconfigurable dispose de capacités qui ne sont pas présentes dans les FPGAs commerciaux, et qui peuvent être exploitées par ces applications spécifiques :

- Le routage dynamique permet aux cellules de créer des connexions durant la “vie” de l'organisme.
- L'auto-configuration partielle des molécules offre de la plasticité aux cellules, qui peuvent modifier leur comportement, notamment lors de la phase de différenciation des mécanismes ontogénétiques.
- Le routage inter-moléculaire est effectué via des multiplexeurs, de manière à empêcher toute possibilité de court-circuit, ce qui fait de POEtic une plateforme idéale pour de l'évolution matérielle non-contrainte.
- La configuration des molécules est exécutée par lots de 32 bits, afin d'accélérer le chargement d'un organisme.

Nous allons à présent nous intéresser à la réalisation physique de ce circuit POEtic,

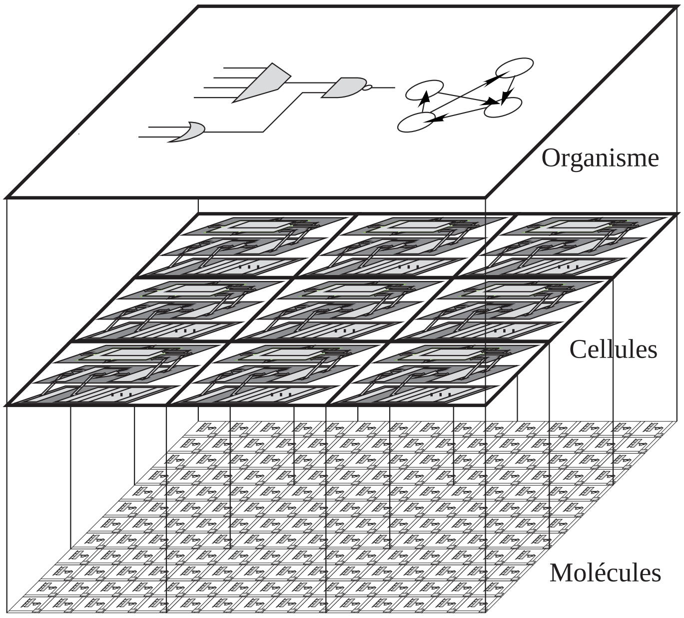


Figure 6.1 : *Les trois niveaux d'organisation d'un organisme implémenté sur le circuit POEtic.*

principalement composé d'un microprocesseur et d'un tableau d'éléments reconfigurables. Ce dernier est couplé à un système de routage distribué quasiment identique à HIDRA, que nous avons présenté en détail en page 104, et constitue donc une application directe des recherches effectuées sur le routage distribué du chapitre 5. Bien qu'il soit un System On Chip générique, nous allons présenter le circuit POEtic en partant du principe que ce sont des cellules possédant une architecture de type POEtic qui y seront implémentées, étant donné qu'il a spécialement été conçu pour accueillir des designs multicellulaires bio-inspirés.

Avant de poursuivre, il est important d'accentuer le fait que le circuit POEtic a été réalisé physiquement, et non uniquement simulé. Un premier prototype ne contenant que 12 molécules et le microprocesseur a tout d'abord été réalisé et testé avec succès avant que ne le soit le circuit final, ce dernier étant actuellement en phase de test. Ce chapitre présente donc la structure du circuit réel, qui a dû être figée dans le silicium, et n'est donc plus modifiable. Nous verrons toutefois que quelques améliorations pourraient être faites si une deuxième version de POEtic devait voir le jour.

Nous allons tout d'abord présenter la structure globale du circuit. Nous nous arrêterons ensuite sur le tableau reconfigurable, qui fut développé dans le cadre de cette thèse. Le routage distribué sera brièvement rappelé, et ses différences d'avec HIDRA explicitées, avant de définir exactement son interface le connectant au tableau reconfigurable. Nous verrons alors comment plusieurs circuits POEtic peuvent être connectés en un super-circuit. La partie reconfigurable ayant été explicitée, sous passerons rapidement sur les caractéristiques du microprocesseur embarqué, ce dernier ayant été conçu à Barcelone, ainsi que sur l'interfaçage entre ce processeur et la partie reconfigurable. Nous présenterons ensuite comment certains composants à la base de nombreux designs peuvent y être efficacement implémentés. Nous terminerons finalement par les outils de développement, qui ont été réalisés dans le but d'épauler un utilisateur final dans la conception de designs spécifiquement prévus pour notre circuit.

6.1 Structure globale

Le circuit POEtic est composé de trois parties distinctes : le sous-système environnemental, le sous-système organique, et l'interface entre les deux (Figure 6.2).

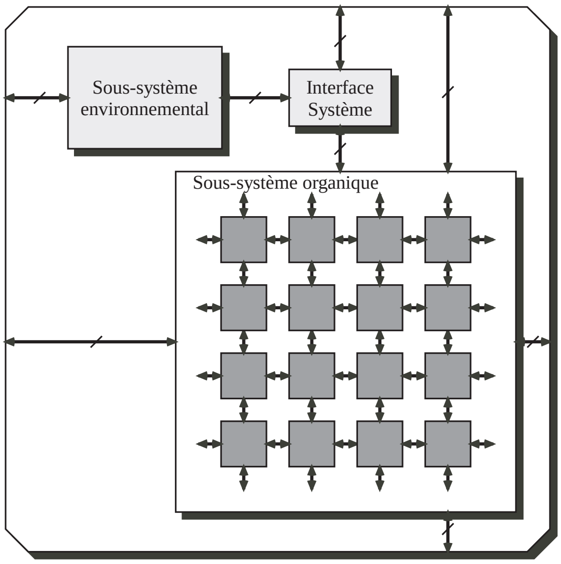


Figure 6.2 : *La structure du circuit POEtic, composé de trois parties.*

Le sous-système environnemental est chargé de gérer les entrées/sorties du circuit, en récupérant les données de l'environnement, et en les envoyant au sous-système organique, où l'organisme y étant implémenté peut les intégrer de manière à agir en conséquence. Il est également responsable des processus évolutifs et du chargement des individus dans le sous-système organique. Un microprocesseur 32 bits de type RISC constitue le cœur de ce sous-système, offrant ainsi un grande généricité au circuit, un microprocesseur étant capable d'effectuer n'importe quel traitement. En outre, des périphériques particuliers lui ont été ajoutés, tels un générateur de nombre pseudo-aléatoires, qui s'avère fort utile lors de l'exécution d'algorithmes génétiques.

Le sous-système organique est la partie dans laquelle sont chargés les individus, et où leur comportement est exprimé, et évalué. L'ontogenèse prend place dans ce sous-système, les cellules qui y sont implémentées pouvant être capables de s'auto-configurer sur la base de leur génome et des interactions qu'elles entretiennent avec les autres cellules, ainsi qu'avec leur environnement. De manière plus générale, ce sous-système permet l'implémentation de n'importe quel type de design, qu'il soit bio-inspiré ou non. Les éléments reprogrammable qui composent ce sous-système sont appelés “molécules”, par analogie avec le niveau hiérarchique trouvé dans les organismes vivants.

L'interface des systèmes est, quant à lui, responsable de la bonne communication entre les deux sous-systèmes. En outre, il gère automatiquement les systèmes multi-circuits, et offre donc une meilleure scalabilité au tissu complet.

Nous allons commencer par décrire le sous-système organique, qui a été développé par nos soins, la description des deux autres parties nécessitant de connaître la structure de celui-ci.

6.2 Le sous-système organique

Notre circuit est spécifiquement dédié à des applications bio-inspirées dont la structure est multicellulaire, et les cellules ayant une architecture POEtic doivent y être facilement implémentables (Figure 6.1). Étant donné que nous désirions offrir assez de souplesse pour permettre l'implémentation de n'importe quel type de cellules et de connectique entre ces cellules, nous avons développé un nouveau type de circuit reconfigurable. Composé de trois niveaux (Figure 6.3), un tableau de molécules, un tableau d'unités de routage, et d'une interface entre les deux, il offre des caractéristiques qui n'existent pas dans les FPGAs commerciaux : du routage autonome, de l'auto-reconfiguration partielle, et l'impossibilité de créer des courts-circuits.

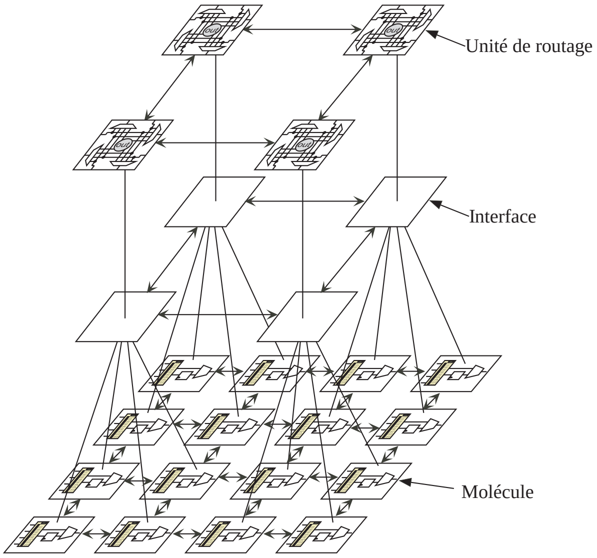


Figure 6.3 : *Les trois couches du sous-système organique.*

Le routage autonome permet aux cellules de créer de nouvelles connexions sans nécessiter l'intervention d'un contrôleur externe. De ce fait, des réseaux de neurones à topologie variable peuvent y être implémentés, de même que des mécanismes de croissance et d'autoréparation. La reconfiguration partielle, quant à elle, autorise les cellules à modifier les bits de configuration de leurs molécules, c'est-à-dire à modifier leur propre comportement, où celui d'autres cellules. Dans le cas de l'ontogenèse, il sera intéressant de pouvoir disposer de telles capacités permettant à une cellule de configurer une voisine, qui pourra ensuite la remplacer ou continuer une phase de croissance. Finalement, l'impossibilité de créer des courts-circuits va permettre l'utilisation du circuit POEtic comme base à du matériel évolutif, aucune combinaison de bits de configuration ne pouvant causer de tort au matériel. L'utilisation de bus de connexions a donc été remplacé ici par des switchboxes à base de multiplexeurs, qui ralentissent la propagation de signaux au travers du circuit, mais offrent cette garantie de sécurité.

Le premier niveau du sous-système organique est donc, comme nous l'avons déjà cité, composé de molécules. Chaque molécule (Figure 6.4) est principalement composée d'une look-up table de 16 bits, d'une bascule, et d'un switchbox lui permettant de communiquer avec d'autres molécules du même circuit.

Le niveau de routage distribué est, lui, composé d'un tableau d'unités de routage

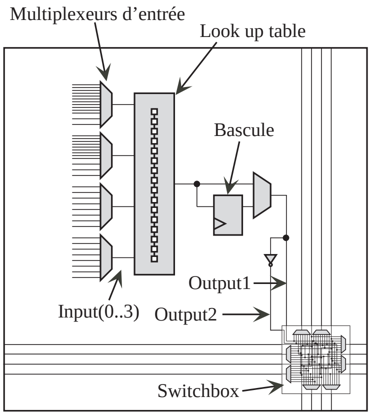


Figure 6.4 : *Structure de base d'une molécule.*

relativement semblables à celles que nous avons décrites dans l'implémentation de l'algorithme HIDRA. Chaque unité de routage est connectée à ses quatre voisines, ainsi qu'à une interface moléculaire. Ce niveau, qui fait la spécificité du circuit POEtic, permet de créer des connexions entre molécules durant le fonctionnement du système, et ce de manière totalement distribuée. Étant donné que nous allons présenter les molécules avant le routage distribué, il est important de noter une différence importante avec HIDRA. En effet, deux modes de routage sont disponibles, et sont imposés par les molécules elles-mêmes. Alors que le premier offre un comportement identique à HIDRA, le deuxième, appelé mode pseudo-statique, ne permet que de créer des communications selon un seul axe. Une source et une destination doivent y être situées sur la même ligne horizontale ou verticale, et le processus de routage complet est exécuté en 16 coups d'horloge.

Finalement, l'interface entre les molécules et les unités de routage a été conçu dans le but de pouvoir facilement paramétrer le nombre de molécules reliées à une unité de routage. Dans la réalisation physique du circuit, nous avons choisi un ratio de $4:1$, mais il serait extrêmement aisé de le modifier, par exemple pour relier neuf molécules à une unité de routage.

Nous allons maintenant décrire en détail le fonctionnement et l'implémentation des molécules. Nous continuerons ensuite avec les unités de routage, puis avec leur interface.

6.3 Les Molécules

Chaque molécule (Figure 6.4) contient une bascule D, une look-up table de 16 bits, et un switchbox. Les quatres entrées de la LUT, `Input(0..3)`, sont sélectionnées grâce à quatre multiplexeurs, qui sont détaillés en page 190, mais dont il est important de savoir qu'ils ont la possibilité de récupérer la sortie de la bascule. La sortie de la LUT peut passer par la bascule, en fonction de la configuration du multiplexeur de sortie. Enfin, deux signaux, `Output1` et `Output2`, sont transmis au switchbox, et sont, dans la configuration standard, la valeur sélectionnée par le multiplexeur de sortie et son inverse. Les molécules d'un même circuit peuvent communiquer en faisant passer des valeurs par les switchboxs moléculaires, qui offrent deux lignes unidirectionnelles dans chacune des quatre directions. Elle est donc reliée à ses quatre voisines, ainsi qu'à une unité de routage, pour les communications intercellulaires.

Alors que le projet embryonique [142] était basé sur une molécule dont la fonctionnalité était constituée d'un multiplexeur et d'une bascule, notre circuit offre une granularité plus grosse. Dans une cellule implémentée sur embryonique, une grande partie des molécules ne servent qu'à router des signaux, alors que la présence du switchbox dans les nôtres permet d'éviter le gaspillage de molécules à cette fin.

En comparaison des FPGAs dernier cri d'Altera et de Xilinx, qui contiennent plusieurs LUTs, notre molécule possède une structure moins complexe, mais offre l'avantage de disposer de mécanismes spécifiques permettant aux molécules de configurer leurs voisines, ainsi que d'accéder au niveau de routage distribué. De plus, la taille du circuit final étant limitée, nous devions prendre garde à ne pas développer de molécules trop grandes, qui n'auraient autorisé la présence que d'un petit nombre d'entre elles dans ce circuit.

Enfin, outre la fonctionnalité de base, qui consiste en une 4-LUT, la molécule peut être configurée selon huit modes opératoires, qui seront décrits en détails dans les section 6.3.1 à 6.3.8 :

- En mode **4-LUT**, la LUT fournit une sortie en fonction de ses quatre entrées, et la sortie peut passer par la bascule ou non (page 177).
- En mode **3-LUT**, la LUT est décomposée en deux LUTs à 8 bits. Elles partagent les mêmes trois entrées, et la sortie de la première peut passer par la bascule. La deuxième sortie, qui est reliée au switchbox, est également directement transmise à la molécule voisine au Sud, et permet d'implémenter efficacement des opérations arithmétiques nécessitant une retenue (page 178).
- En mode **Comm**, la LUT est décomposée en une LUT à trois entrées et un registre à décalage de 8 bits. Ce mode pourrait donc être utilisé pour comparer une adresse stockée dans le registre à décalage avec une entrée sérielle (page 178).
- En mode **Shift Memory**, les 16 bits de la LUT sont considérés comme un registre à décalage, et peuvent être utilisés pour stocker de l'information, comme un génome de cellule, par exemple. Deux entrées seulement sont utilisées dans ce mode, une pour le contrôle du décalage, et une comme donnée à insérer dans le registre (page 179).
- En mode **Input**, la molécule est considérée comme une entrée de la cellule, et a pour but de récupérer une valeur transmise via le niveau de routage distribué. Elle est connectée à une unité de routage, et peut ou non initier une nouvelle connexion. Dans tous les cas, elle se doit d'accepter toute connexion correspondant à son adresse, qui est stockée dans les 16 bits du registre à décalage. Seules deux entrées sont utilisées, l'une servant à initier une connexion, et l'autre servant à définir le mode de routage du tissu POEtic (page 180).
- En mode **Output**, la molécule est considérée comme une sortie de la cellule, et peut envoyer des valeurs via le niveau de routage distribué. De la même manière qu'une molécule Input, elle stocke son adresse dans son registre à décalage et possède une entrée pour initier une connexion. Sa deuxième entrée est alors la valeur à envoyer à sa destination correspondante (page 181).
- En mode **Trigger**, le registre à décalage de la molécule doit contenir la valeur "$000\cdots01$" pour une adresse codée sur 16 bits, et sert à synchroniser les unités de routage lors de la phase de comparaison d'adresse. L'une des entrées de la molécule est un *Chip enable*, capable de désactiver certaines molécules du tissu

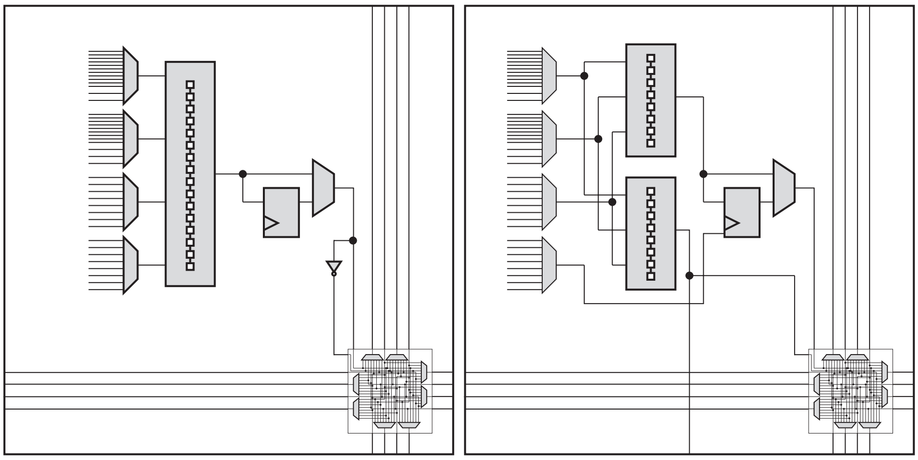
POEtic, et l'autre permet un reset du routage en détruisant toutes les connexions existantes (page 182).

- En mode **Configure**, la molécule a la capacité d'accéder aux bits de configuration d'autres molécules présentes sur le même circuit. Elle peut donc partiellement reconfigurer certaines molécules, grâce à ses deux entrées, dont une contrôle la reconfiguration tandis que l'autre correspond au bit à insérer à chaque coup d'horloge (page 184).

Nous allons à présent entrer dans le détail des modes opératoires de la molécule, avant de décrire l'implémentation des molécules et leurs spécificités.

6.3.1 Mode 4-LUT

En mode 4-LUT (Figure 6.5(a)), la molécule peut calculer n'importe quelle fonction à 4 entrées. Le signal `Input` sélectionne un bit parmi les seize du registre de la molécule. Il est donc possible d'implémenter n'importe quelle fonction à quatre variables avec une molécule 4-LUT. En ce qui concerne les entrées, les quatres valeurs de `Input` sont utilisées comme entrées de la look-up table, et les multiplexeurs sont donc séparés en 4 multiplexeurs à 8 entrées. En sortie, `Output1` correspond à la sortie de la look-up table, et peut être combinatoire ou séquentielle, tandis que `Output2` est l'inverse de `Output1`.



(a) (b)

Figure 6.5 : *A gauche, une molécule en mode 4-LUT, et à droite en mode 3-LUT.*

Entrée	**Fonction**
`Input(0)`	Bit de sélection 0 de la 4-LUT
`Input(1)`	Bit de sélection 1 de la 4-LUT
`Input(2)`	Bit de sélection 2 de la 4-LUT
`Input(3)`	Bit de sélection 3 de la 4-LUT
Sortie	**Fonction**
`Output1`	Résultat de la 4-LUT, séquentiel ou non
`Output2`	Inverse de `Output1`

6.3.2 Mode 3-LUT

En mode 3-LUT (Figure 6.5(b)), la molécule peut calculer deux fonctions utilisant les mêmes trois entrées. Le résultat de la première fonction peut passer par la bascule, alors que le deuxième est directement envoyé à la voisine au Sud, afin d'accélérer les opérations de type addition parallèle.

La LUT est séparée en deux LUTs de 8 bits. Le registre est donc coupé en deux, les 8 bits de poids faible définissant le comportement de la première LUT, et les 8 bits de poids fort définissant celui de la deuxième.

Les trois entrées de la première LUT sont identiques à ceux de la deuxième, et correspondent aux trois premiers signaux d'entrée `Input(0..2)`. La dernière entrée, `Input(3)`, agit comme un *enable* de la bascule, et n'est utilisée que si le bit de configuration de l'*enable* de la bascule est actif.

Entrée	**Fonction**
`Input(0)`	Bit de sélection 0 des deux 3-LUT
`Input(1)`	Bit de sélection 1 des deux 3-LUT
`Input(2)`	Bit de sélection 2 des deux 3-LUT
`Input(3)`	*enable* de la bascule D
Sortie	**Fonction**
`Output1`	Résultat de la première 3-LUT, séquentiel ou non
`Output2`	Résultat de la deuxième 3-LUT

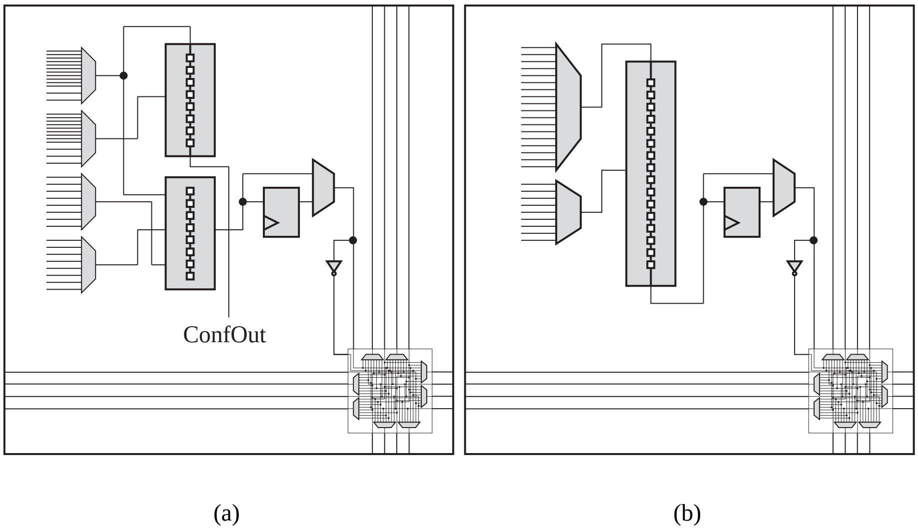
`Output1` correspond à la sortie de la première LUT, et peut être séquentielle ou non, tandis que `Output2` est la sortie de la deuxième LUT, qui peut également être directement récupérée par la molécule au Sud. Ce signal permet d'éviter de devoir passer par le switchbox moléculaire, et donc de réquisitionner inutilement des ressources, dans le cas d'application nécessitant des opérations parallèles.

6.3.3 Mode Comm

Le mode Comm a initialement été développé pour permettre l'implémentation efficace d'un mécanisme de communication par paquets, qui ne nécessiterait pas de passer par le routage distribué. Les 16 bits du registre à décalage sont coupés en un registre à décalage de 8 bits pour l'octet de poids faible, et une LUT de 8 bits pour l'octet de poids fort (Figure 6.6(a)). Une des entrées contrôle le décalage du registre, et les trois autres correspondent aux entrées de la 3-LUT. Notons toutefois que $Input(0)$ est utilisé conjointement par la 3-LUT et par le registre à décalage, dont il est le bit à insérer.

Entrée	**Fonction**
`Input(0)`	Bit de sélection 0 de la 3-LUT, et bit d'entrée du registre à décalage
`Input(1)`	Contrôle du décalage du registre
`Input(2)`	Bit de sélection 2 de la 3-LUT
`Input(3)`	Bit de sélection 1 de la 3-LUT

Sortie	**Fonction**
`Output1`	La valeur calculée par la 3-LUT
`Output2`	Inverse de `Output1`



(a) (b)

Figure 6.6 : *A gauche, une molécule en mode Comm, et à droite en mode Mémoire à décalage.*

Le bit de poids fort du registre à décalage peut être récupéré par les molécules voisines, le signal `ConfOut` de la figure 6.6(a) étant accessible via un niveau de communication normalement réservé aux bits de configuration.

Pour l'implémentation d'une communication par paquets, le registre à décalage peut contenir une adresse, sur 8 bits, qui doit être comparée avec une adresse arrivant en série. Le registre à décalage peut boucler sur lui-même, son entrée étant également utilisée par la LUT. En utilisant le bit sériel de l'adresse à comparer comme deuxième entrée de la LUT, et la bascule comme troisième, il est alors aisé de réaliser un comparateur d'adresse sériel dans une seule molécule.

6.3.4 Mode Shift Memory

Une molécule en mode Shift Memory, que nous appellerons également mode Mémoire, sert à stocker de l'information de manière sérielle. Les 16 bits du registre sont considérés comme un registre à décalage, et plusieurs de ces molécules peuvent être chaînées pour créer un registre à décalage de taille plus importante. Une entrée active à '1' gère le décalage, et l'autre correspond au bit à insérer lors d'une telle action.

Entrée	**Fonction**
`Input16(0)`	Bit d'entrée du registre à décalage
`Input16(1)`	Contrôle du décalage du registre
Sortie	**Fonction**
`Output1`	bit de poids fort du registre à décalage, où valeur de la bascule
`Output2`	Valeur inversée de Output1

Il est intéressant de noter que la sortie du registre à décalage peut passer par la bascule de la molécule avant d'être utilisée. De cette manière, il est possible de créer un registre à 17 bits dans une seule molécule.

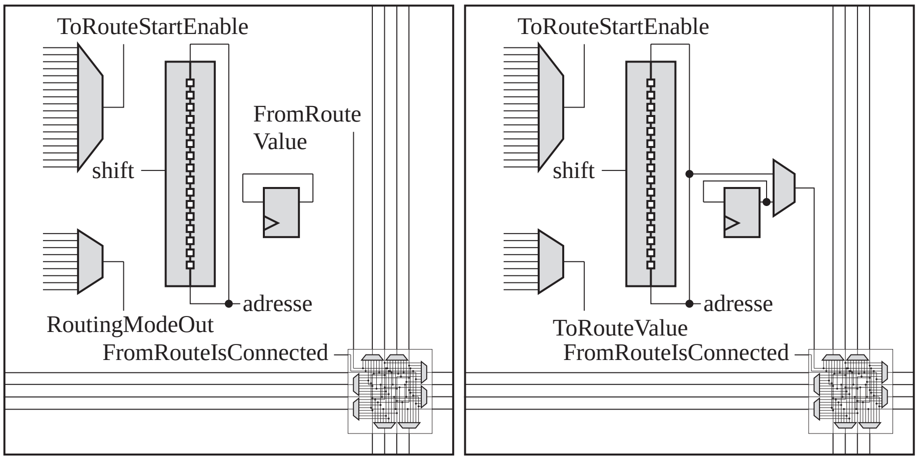
6.3.5 Mode Input

Alors que les modes précédents sont des caractéristiques qu'il est possible de retrouver dans certains FPGAs actuels (Virtex de Xilinx, par exemple), le mode Input est une particularité du circuit POEtic. Une molécule dans ce mode opératoire accède à l'unité de routage qui lui est connectée, et est considérée comme une destination. Deux entrées gèrent cette molécule : la deuxième entrée sert à indiquer le mode de routage général, tandis que la première indique si l'unité de routage doit impérativement se connecter à sa source correspondante. Si elle est inactive, l'unité de routage se doit toutefois d'accepter une connexion si une source possédant la même adresse tente de créer un nouveau chemin.

Entrée	**Fonction**
`Input16(0)`	Force la création de la connexion
`Input16(1)`	Sélection du mode de routage
Sortie	**Fonction**
`Output1`	Valeur transmise par l'unité de routage
`Output2`	Indique si la molécule est connectée à sa correspondante

Les deux sorties de la molécule sont respectivement la valeur d'entrée fournie par l'unité de routage, et une indication sur l'état de la connexion : si la deuxième sortie est à '1', cela signifie que l'unité de routage est connectée à sa correspondante.

Le registre à décalage contient ici une adresse, qui est accédée par l'unité de routage de manière sérielle, lors d'un processus de routage. L'unité de routage gère donc le décalage, et récupère le bit de poids fort du registre à décalage. Cette adresse peut être composée de 16, 8, 4, 2, ou 1 bits, en fonction du nombre de connexions potentielles du circuit. La taille est gérée par la molécule en mode Trigger, et elle définit la manière de remplir le registre de 16 bits. En effet, pour des tailles inférieures à 16 bits, l'adresse doit être répétée, comme indiqué dans le tableau 6.1, où a_i est le i-ème bit de l'adresse.



(a) (b)

Figure 6.7 : *A gauche, une molécule en mode Input, et à droite en mode Output.*

Taille de l'adresse	**Contenu du registre**
1	$a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0\ a_0$
2	$a_1a_0\ a_1a_0\ a_1a_0\ a_1a_0\ a_1a_0\ a_1a_0\ a_1a_0\ a_1a_0$
4	$a_3a_2a_1a_0\ a_3a_2a_1a_0\ a_3a_2a_1a_0\ a_3a_2a_1a_0$
8	$a_7a_6a_5a_4a_3a_2a_1a_0\ a_7a_6a_5a_4a_3a_2a_1a_0$
16	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$

Tableau 6.1 : *Contenu du registre, en fonction de la taille des adresses utilisées par le routage distribué.*

Si le mode de routage sélectionné par les molécules du circuit est pseudo-statique, le contenu du registre est directement chargé dans les bits de configuration des multiplexeurs de l'unité de routage, par un accès sériel, et ce en 16 coups d'horloge. Le tableau 6.2 définit les bits du registre responsables des multiplexeurs, et le tableau 6.3 indique le signal sélectionné par ces multiplexeurs, en fonction de leurs bits de configuration.

6.3.6 Mode Output

Le mode Output (Figure 6.7(b)) est relativement similaire au mode Input, mais correspond à une source de données qui seront transmises via le routage distribué. Le registre y est géré de la même manière, et son contenu est identique à celui d'une molécule Input. Une des deux entrées permet de forcer une connexion, tandis que la deuxième sert tout naturellement de valeur à transmettre via le routage distribué, vers la destination connectée. En sortie de la molécule, la première peut être le bit de poids fort de l'adresse (peu utile), ou la valeur contenue dans la bascule, qui peut permettre de disposer d'une valeur fixe à '0' ou à '1'. La deuxième, à l'instar du mode Input, indique si l'unité de routage est connectée ou non à sa correspondante.

bits du registre	Multiplexeur
2..0	Vers le Nord
5..3	Vers l'Est
8..6	Vers le Sud
11..9	Vers l'Ouest
14..12	Vers la molécule

Tableau 6.2 : *En mode de routage pseudo-statique, le contenu du registre est directement utilisé pour configurer les multiplexeurs de l'unité de routage.*

Bits de configuration	Vers le Nord	Vers l'Est	Vers le Sud	Vers l'Ouest	Vers la molécule
X00	Molécule	Nord	Nord	Nord	Nord
X01	Est	Molécule	Est	Est	Est
X10	Sud	Sud	Molécule	Sud	Sud
X11	Ouest	Ouest	Ouest	Molécule	Ouest

Tableau 6.3 : *Signal sélectionné par chaque multiplexeur de l'unité de routage, en fonction de ses bits de configuration.*

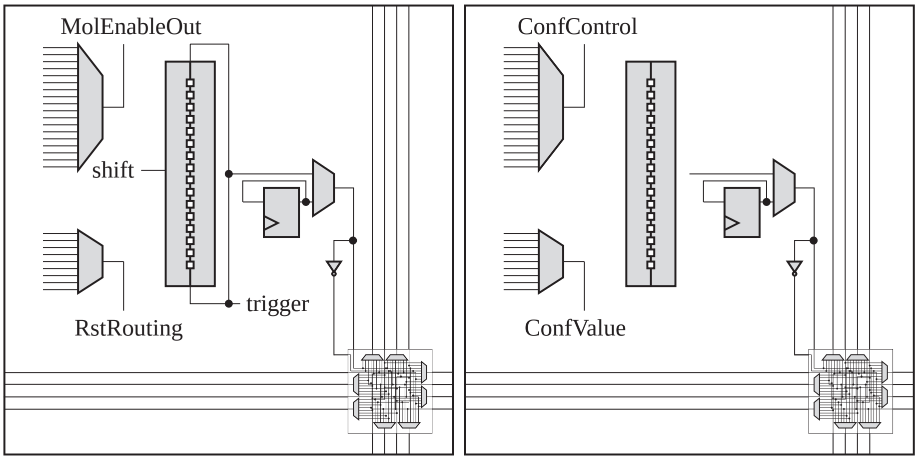
Entrée	Fonction
`Input16(0)`	Force la création de la connexion
`Input16(1)`	Valeur à transmettre à l'unité de routage

Sortie	Fonction
`Output1`	Si la sortie est combinatoire, il s'agit du bit de poids fort du registre. Sinon il s'agit de la valeur de la bascule
`Output2`	Indique si la molécule est connectée à sa correspondante

6.3.7 Mode Trigger

Une molécule en mode Trigger (Figure 6.8(a)) a un status un peu particulier. Elle n'exécute aucune tâche spécifique en relation avec d'autres molécules du circuit, mais est utile aux unités de routage pour synchroniser la phase de comparaison des adresses. Le contenu du registre est décalé de la même manière que les adresses des molécules Input et Output, et est en charge de fournir un '1' pour indiquer la fin de la comparaison. Pour une adresse de taille n, il doit donc contenir $n - 1$ '0's, puis un '1', ce motif étant répété de façon à remplir le registre, comme indiqué dans le tableau 6.4.

Les deux entrées de la molécule ont des fonctions globales. La première agit comme un *chip enable*, par lequel certaines molécules, sensibles à ce signal, peuvent être momentanément désactivées. Cette particularité offre notamment la possibilité de séparer une cellule en une partie fonctionnelle et une partie ontogénétique, cette dernière pouvant s'occuper de construire l'organisme, tout en inhibant la partie fonctionnelle. La deuxième entrée permet de réinitialiser le routage dynamique, en forçant une



(a) (b)

Figure 6.8 : *A gauche, une molécule en mode Trigger, et à droite en mode Configure.*

Taille de l'adresse	**Contenu du registre**
1	1111111111111111
2	0101010101010101
4	0001000100010001
8	0000000100000001
16	0000000000000001

Tableau 6.4 : *Le contenu d'une molécule en mode Trigger dépend de la taille des adresses utilisées par le routage distribué.*

reconstruction des chemins de données. Une cellule peut donc, si besoin s'en fait sentir, comme durant des processus d'autoréparation, réinitialiser le routage intercellulaire de manière à prendre en compte une nouvelle cellule capable de remplacer une cellule défaillante. Le reset du routage distribué est effectué après un coup d'horloge, de manière synchrone, une bascule ayant été placée après la porte ET générale réunissant les signaux de Reset de toutes les molécules.

Entrée	**Fonction**
`Input16(0)`	Désactive certaines molécules
`Input16(1)`	Force un Reset du routage distribué

Sortie	**Fonction**
`Output1`	Si la sortie est combinatoire, il s'agit du bit de poids fort du registre. Sinon il s'agit de la valeur de la bascule
`Output2`	La valeur inversée de `Output1`

La sortie `Output1` peut être soit le bit de poids fort du trigger (peu utile), soit la valeur de la bascule, et la sortie `Output2` correspond à l'inverse de `Output1`. Ces deux sorties n'ont que peu d'utilité, une molécule Trigger n'étant présente que pour fournir une valeur à une unité de routage ainsi que deux signaux globaux, mais ont le mérite d'exister.

6.3.8 Mode Configure

Une molécule en mode Configure (Figure 6.8(b)) a la capacité de partiellement reconfigurer d'autres molécules présentes sur le même circuit. Pour ce faire, deux entrées sont utilisées : la première indique aux voisines si elles doivent effectuer une configuration partielle, et la deuxième est le bit à décaler dans les bits de configuration des molécules cibles.

Entrée	**Fonction**
`Input16(0)`	Active la configuration du voisinage
`Input16(1)`	Bit de configuration envoyé au voisinage

Sortie	**Fonction**
`Output1`	Si la sortie est combinatoire, il s'agit du bit de poids fort du registre. Sinon il s'agit de la valeur de la bascule, qui n'est pas fixe après un Reset [1]
`Output2`	La valeur inversée de `Output1`

La reconfiguration partielle, expliquée en page 192, est une des spécificités du circuit POEtic. Une cellule POEtic y a donc la possibilité de modifier son comportement en reconfigurant une partie de ses molécules. Pour l'implémentation de processus ontogénétiques, nous pouvons couper la cellule en une partie fonctionnelle et une partie chargée de la division et de la différenciation. Si un génome est stocké dans les cellules, et que ce génome représente des bits de configuration de molécules, la cellule peut sélectionner, en fonction de sa différenciation, une partie du génome, et la charger dans les molécules de la partie fonctionnelle.

6.3.9 Entrées/sorties

Les entrées/sorties d'une molécule peuvent être classées dans quatre catégories : les signaux globaux, les signaux de configuration, les communications intermoléculaires avec les quatre voisines, et les communications avec l'interface de l'unité de routage. Nous allons brièvement passer en revue ces différents signaux, afin de maîtriser l'interface des molécules.

Signaux globaux

Plusieurs entrées et sorties de la molécule agissent à un niveau global (Tableau 6.5), c'est-à-dire ont un impact direct sur le comportement de toutes les molécules, ou de toutes les unités de routage.

Le `ChipEnable`, géré par le microprocesseur externe, permet de bloquer le fonctionnement du circuit entier, par exemple durant la configuration de celui-ci.

---

[1] Dans une version ultérieure du circuit, il est bien clair que la valeur de la bascule devrait être fixe, pour une molécule en mode Config.

`IsRouting` indique si un processus de routage est en cours, dans quel cas la fontionnalité des molécules est bloquée, et seules les molécules Input, Output et Trigger sont actives. Enfin, `MolEnableIn` permet de désactiver certaines molécules (cf. page 195).

Concernant les sorties, `RstRouting` permet à une molécule Trigger de forcer un reset du routage distribué, tandis que `RoutingModeOut`, contrôlé par les molécules en mode Input, sélectionne le type de routage, qui peut être pseudo-statique, ou semblable à l'algorithme HIDRA. Enfin, une molécule Trigger peut utiliser `MolEnableOut` pour désactiver certaines molécules, ce signal étant à mettre en relation avec `MolEnableIn`.

Entrée	**bits**	**Description**
`clk`	1	Horloge globale du système
`rst`	1	Reset du système, pour la bascule D
`ChipEnable`	1	Désactive la molécule
`IsRouting`	1	Indique si un routage est en cours
`MolEnableIn`	1	Désactive la molécule

Sortie	**bits**	**Description**
`RstRouting`	1	Force un reset du routage, par une molécule Trigger
`RoutingModeOut`	1	Mode de routage
`MolEnableOut`	1	Permet de désactiver certaines molécules, commandé par une molécule Trigger

Tableau 6.5 : *Entrées/Sorties globales d'une molécule.*

Signaux de configuration

La configuration des molécules se fait de manière parallèle, en accédant le tableau de molécules comme une RAM de 32 bits de données (Tableau 6.6). Les données en entrée `ValueIn`, et respectivement en sortie `ValueOut`, permettent au microprocesseur de configurer les molécules, et de lire leur contenu, en choisissant le mode d'accès grâce au signal `Write`. Et étant donné que la molécule possède 76 bits de configuration, les trois signaux `Cs`, dont un seul doit être actif à la fois, sélectionnent la partie des bits de configuration qui est accédée (cf. page 192).

Entrée	**bits**	**Description**
`ValueIn`	32	Bits de configuration à charger
`Write`	1	Force une écriture des bits de configuration
`Cs`	3	Sélectionne un des trois blocs de bits de configuration

Sortie	**bits**	**Description**
`ValueOut`	32	32 bits de configuration, lors d'une lecture

Tableau 6.6 : *Entrées/Sorties d'une molécule, au niveau de la configuration.*

Signaux de communication intermoléculaire

Les molécules sont organisées selon une grille régulière à deux dimensions, où chacune est directement connectée à ses quatre voisines. Parmi les signaux du tableau 6.7, les connexions sont ainsi :

- `ValInN` ⇐ `ValOutS` de la molécule au Nord.
- `ValInE` ⇐ `ValOutW` de la molécule à l'Est.
- `ValInS` ⇐ `ValOutN` de la molécule au Sud.
- `ValInW` ⇐ `ValOutE` de la molécule à l'Ouest.
- `ConfPartInN` ⇐ `ConfPartOut` de la molécule au Nord.
- `ConfPartInE` ⇐ `ConfPartOut` de la molécule à l'Est.
- `ConfPartInS` ⇐ `ConfPartOut` de la molécule au Sud.
- `ConfPartInW` ⇐ `ConfPartOut` de la molécule à l'Ouest.
- `ConfInN` ⇐ `ConfOut` de la molécule au Nord.
- `ConfInE` ⇐ `ConfOut` de la molécule à l'Est.
- `ConfInS` ⇐ `ConfOut` de la molécule au Sud.
- `ConfInW` ⇐ `ConfOut` de la molécule à l'Ouest.
- `ChainIn` ⇐ `ChainOut` de la molécule au Nord.

Entrée	**bits**	**Description**
`ConfPartInN`	1	Force une configuration partielle venant du Nord
`ConfPartInE`	1	Force une configuration partielle venant de l'Est
`ConfPartInS`	1	Force une configuration partielle venant du Sud
`ConfPartInW`	1	Force une configuration partielle venant de l'Ouest
`ConfInN`	1	Bit de configuration à insérer durant une reconfiguration partielle venant du Nord
`ConfInE`	1	Bit de configuration à insérer durant une reconfiguration partielle venant de l'Est
`ConfInS`	1	Bit de configuration à insérer durant une reconfiguration partielle venant du Sud
`ConfInW`	1	Bit de configuration à insérer durant une reconfiguration partielle venant de l'Ouest
`ValInN`	3	Valeurs transmises du Nord
`ValInE`	3	Valeurs transmises de l'Est
`ValInS`	3	Valeurs transmises du Sud
`ValInW`	3	Valeurs transmises de l'Ouest
`ChainIn`	1	Valeur de retenue envoyée par le Nord

Sortie	**bits**	**Description**
`ConfPartOut`	1	Propose une reconfiguration partielle, envoyée aux 4 voisines
`ConfOut`	1	Bits de configuration à décaler lors d'une reconfiguration partielle
`ChainOut`	1	Retenue envoyée au Sud, pour les opérations arithmétiques ou logiques
`ValOutN`	3	Valeurs transmises au Nord
`ValOutE`	3	Valeurs transmises à l'Est
`ValOutS`	3	Valeurs transmises au Sud
`ValOutW`	3	Valeurs transmises à l'Ouest

Tableau 6.7 : *Entrées/Sorties d'une molécule, pour la communication intermoléculaire.*

Les valeurs d'entrée de `ValOutX(0..1)` correspondent aux valeurs sélectionnées par le switchbox de la molécule, qui fournit deux signaux dans chacune des directions. `ValOutX(2)`, en revanche, est identique dans chacune des directions, et correspond à la première sortie `Output1`, calculée par l'unité fonctionnelle de la molécule. A titre d'exemple, pour une molécule en mode 3-LUT, il s'agit du résultat de la première LUT, ou de la sortie de la bascule D. Le fait qu'une molécule puisse ainsi récupérer de manière directe la sortie de n'importe laquelle de ces voisines permet d'économiser des ressources au niveau du routage intermoléculaire en groupant efficacement les molécules dont les fonctions sont intimement liées.

Le signal `Chain{In/Out}`, à l'instar du `ValOutX(2)`, permet également d'économiser du routage intermoléculaire. Il transmet à la molécule présente immédiatement au Sud la valeur de sortie de la deuxième LUT de la molécule, ce qui est utile lors d'opérations nécessitant une retenue, comme une addition parallèle, qui, pour un nombre de n bits, peut être implémentée grâce à n molécules (cf. page 211).

Les deux signaux `ConfPartOut` et `ConfOut` sont utilisés lors de la reconfiguration partielle de molécules (cf. page 192), et chacun d'eux est transmis aux quatre molécules voisines. Le premier contrôle cette reconfiguration partielle, tandis que le deuxième correspond au bit de donnée à insérer dans la configuration de la molécule destination.

Signaux de communication avec l'unité de routage

Comme nous l'avons déjà mentionné, quatre molécules sont reliées à une unité de routage via une interface (cf. page 202), et seule une de ces molécules peut être en mode Input ou Output, sans quoi un conflit empêche le bon fonctionnement du système.

En entrée de la molécule (Tableau 6.8), `FromRouteValue` est exploité par une molécule en mode Input, et consiste en la valeur transmise par la molécule Output reliée à elle via le routage distribué. `FromRouteShift`, commandé par l'unité de routage, force la molécule Input, Output, ou Trigger, à décaler le contenu de son registre 16 bits, durant un processus de routage. Enfin, `FromRouteIsConnected` indique simplement à une molécule Input ou Output si son unité de routage est connectée à une unité de routage ayant la même adresse.

La sortie `ToRouteValue` est plus ou moins l'homologue de `FromRouteValue`, et est la valeur envoyée, via le routage distribué, par une molécule en mode Output. De plus, elle sert, lors du processus de routage, à envoyer le bit de poids fort du registre 16 bits de la molécule, qui peut être l'adresse, dans le cas d'une molécule Input ou Output, ou un trigger, pour une molécule Trigger. `ToRouteStartEnable`, comme son nom l'indique, permet à une molécule Input ou Output de forcer la création d'un chemin dans le niveau de routage distribué, pour la connecter à la molécule, respectivement Output ou Input, possédant la même adresse. Et finalement, `ToRouteMode`, sur 3 bits, est le mode opératoire de la molécule, qui permet à l'interface molécule/unité de routage d'envoyer les bons signaux à l'unité de routage.

6.3.10 Communication intermoléculaire

Outre par sa fonctionnalité, une molécule est également définie par la communication qu'elle entretient avec ses voisines. Pour ce faire, chaque molécule possède un

Entrée	bits	Description
`FromRouteValue`	1	Valeur récupérée de l'unité de routage par une molécule en mode Input
`FromRouteShift`	1	Indique qu'il faut décaler le registre, pour une molécule Input, Output, ou Trigger
`FromRouteIsConnected`	1	Indique si l'unité de routage reliée à la molécule est connectée à sa correspondante

Sortie	bits	Description
`ToRouteValue`	1	Valeur transmise à l'unité de routage par une molécule Input, Output, ou Trigger
`ToRouteStartEnable`	1	Force l'unité de routage à démarrer un routage, pour une molécule Input ou Output
`ToRouteMode`	3	Mode opératoire de la molécule, transmis à l'unité de routage

Tableau 6.8 : *Entrées/Sorties d'une molécule, communiquant avec l'unité de routage.*

switchbox qui lui permet d'envoyer ou de faire transiter des valeurs vers d'autres molécules. Décision fut prise d'offrir deux lignes de communication vers chacune des quatre voisines, afin d'éviter les problèmes de congestion qui furent rencontrés dans le cadre du projet Embryonique. L'expérience a montré que ce type de communication permet d'implémenter un système multicellulaire sans problèmes de congestion, car une cellule n'est composée que d'un nombre relativement restreint de molécules – Le circuit final comportant 144 molécules, et la communication entre circuits ne se faisant qu'au niveau du routage distribué, une cellule ne doit pas dépasser une taille de 144 molécules, sans quoi elle doit être répartie sur plusieurs circuits. La communication intercellulaire doit passer par le niveau de routage distribué, et le routage intermoléculaire, confiné à l'intérieur d'une cellule, n'explose donc pas. La figure 6.9 montre la manière dont les molécules sont arrangées, ainsi que la structure du switchbox.

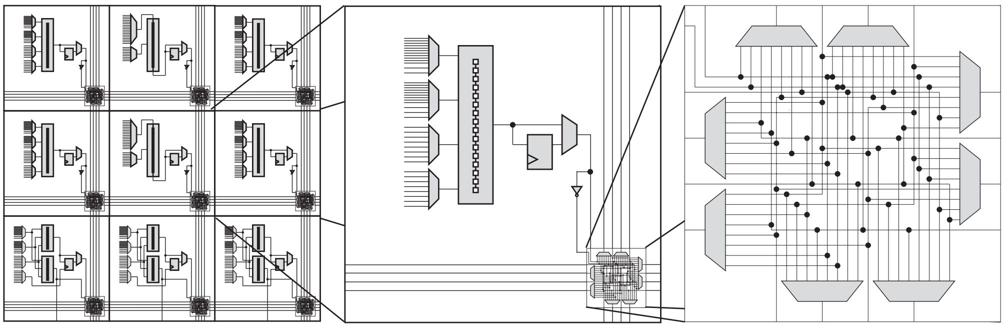


Figure 6.9 : *Un tableau de molécules, et le switchbox d'une molécule.*

En entrée du switchbox, nous trouvons huit lignes, représentant deux lignes par molécule voisine, ainsi que deux signaux résultant de l'activité de la molécule, `Output1` et `Output2`. Trois bits de configuration par multiplexeur définissent exactement la valeur à sélectionner, et donc un total de 24 bits de configuration sont nécessaires à la définition du comportement du switchbox. Le tableau 6.9 indique les signaux sélectionnés en fonction des bits de configuration.

ConfigSwitch(2..0)	ValOutN(0)	ConfigSwitch(5..3)	ValOutN(1)
000	Output1	000	Output1
001	Output2	001	Output2
010	ValInE(0)	010	ValInE(0)
011	ValInE(1)	011	ValInE(1)
100	ValInS(0)	100	ValInS(0)
101	ValInS(1)	101	ValInS(1)
110	ValInW(0)	110	ValInW(0)
111	ValInW(1)	111	ValInW(1)
ConfigSwitch(8.6)	ValOutE(0)	ConfigSwitch(11..9)	ValOutE(1)
000	ValInN(0)	000	ValInN(0)
001	ValInN(1)	001	ValInN(1)
010	Output1	010	Output1
011	Output2	011	Output2
100	ValInS(0)	100	ValInS(0)
101	ValInS(1)	101	ValInS(1)
110	ValInW(0)	110	ValInW(0)
111	ValInW(1)	111	ValInW(1)
ConfigSwitch(14..12)	ValOutS(0)	ConfigSwitch(17.15)	ValOutS(1)
000	ValInN(0)	000	ValInN(0)
001	ValInN(1)	001	ValInN(1)
010	ValInE(0)	010	ValInE(0)
011	ValInE(1)	011	ValInE(1)
100	Output1	100	Output1
101	Output2	101	Output2
110	ValInW(0)	110	ValInW(0)
111	ValInW(1)	111	ValInW(1)
ConfigSwitch(20..18)	ValOutW(0)	ConfigSwitch(23..21)	ValOutW(1)
000	ValInN(0)	000	ValInN(0)
001	ValInN(1)	001	ValInN(1)
010	ValInE(0)	010	ValInE(0)
011	ValInE(1)	011	ValInE(1)
100	ValInS(0)	100	ValInS(0)
101	ValInS(1)	101	ValInS(1)
110	Output1	110	Output1
111	Output2	111	Output2

Tableau 6.9 : *Bits de configuration du switchbox intermoléculaire.*

6.3.11 Multiplexeurs d'entrée

La molécule possède un nombre d'entrées non négligeable, dont certaines sont exploitées par la partie fonctionnelle de la molécule, comme les 8 entrées reliées au switchbox, les quatre entrées directes des voisines, la retenue venant du nord, etc. Or, au maximum quatre signaux sont nécessaires au fonctionnement de la molécule, dans les modes 4-LUT, 3-LUT, et Comm. Afin de sélectionner parmi cet ensemble de valeurs, des multiplexeurs ont été placés dans la molécule, et fournissent par exemple, les quatre signaux d'entrée de la LUT. Ce bloc de sélection des entrées fournit donc quatre valeurs, que nous nommons `Input(0..3)`.

Dans certains modes opératoires, seuls deux entrées sont nécessaires. Dans ces cas, les deux premiers multiplexeurs sont combinés pour obtenir plus de combinaisons, et les deux derniers le sont également, grâce à deux multiplexeurs visibles à droite de la figure 6.10. Les signaux `Input(0)` et `Input(2)` correspondent alors aux entrées utilisées, et nous les renommons `Input16(0)` et `Input16(1)`, pour plus de clarté.

Le bloc de sélection des entrées, observé à la figure 6.10, est donc principalement formé de quatre multiplexeurs à 8 entrées, chacun étant contrôlé par 3 bits de configuration `ConfigLutSel(3X,3X+1,3X+2)`, pour $X \in [0, 3]$. Si les deux bits de configuration `ConfigSpecialInput` et `ConfigDirectIn` sont à '0', les entrées de ces multiplexeurs correspondent plus ou moins aux entrées `ValIn`*X*`(1,0)`, qui sont les valeurs envoyées par les switchboxs des quatre molécules voisines. Pour une molécule en mode 4-LUT, n'importe quelle combinaison de 4 parmi les 8 entrées standards est réalisable. Le troisième multiplexeur a également accès à la valeur de la bascule de la molécule, tandis que le deuxième peut fournir un '1' logique, qui constitue une valeur utile à plusieurs modes opératoires. La présence de la valeur de la bascule est évidemment indispensable pour la réalisation de compteurs ou de machines d'états, tandis que le '1' permet d'éviter de gaspiller une molécule pour fournir cette valeur logique.

Le nombre de signaux disponibles aux deux premiers multiplexeurs est augmenté, grâce aux bits de configuration `ConfigSpecialInput` et `ConfigDirectIn`. Le premier bit offre au premier multiplexeur les valeurs suivantes :

- La valeur calculée par la deuxième LUT de la molécule du Nord, qui permet de faciliter la réalisation d'opération nécessitant la propagation d'une retenue.
- Le bit de poids fort du registre de 16 bits de la molécule, qui est utile lorsque la molécule est en mode Mémoire. Il est alors possible d'en faire un registre à décalage bouclant sur lui-même, en sélectionnant son dernier bit comme entrée du registre.
- Le bit de donnée sélectionné par les deux bits de configuration utiles lors d'une reconfiguration partielle.
- La sortie de la bascule de la molécule.
- Et enfin la valeur logique '0', qui est notamment utile aux modes Input et Output, pour lesquels '0' est la valeur du signal `ToRouteStartEnable` pour ne pas forcer une connexion à s'établir.

Le bit `ConfigDirectIn` permet, quant à lui, au deuxième multiplexeur d'accéder aux valeurs directement transmises par chacune des molécules voisines, correspondant à leur première sortie.

Concernant le contrôle des multiplexeurs, en mode 4-LUT, 3-LUT, et Comm, chacun est géré par trois bits de configuration. En revanche, dans tous les autres modes,

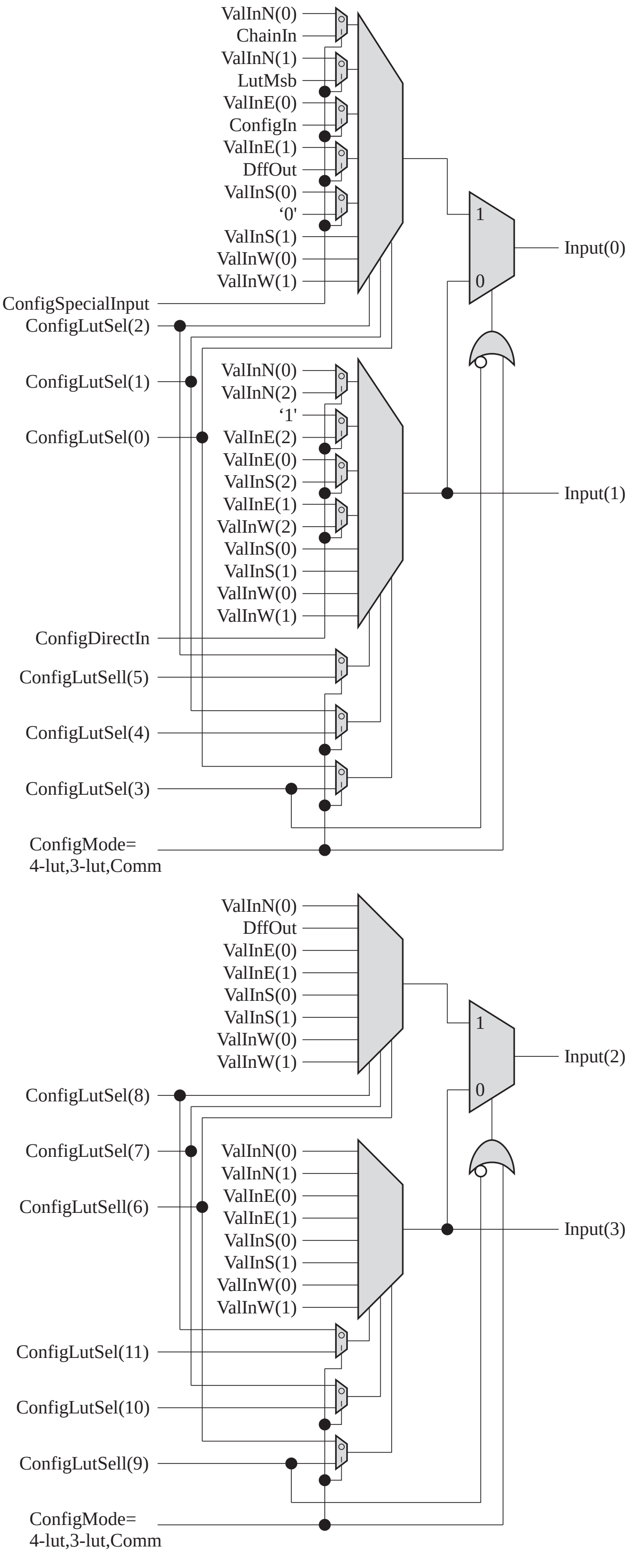


Figure 6.10 : *Multiplexeurs d’entrée.*

qui ne nécessitent l’utilisation que de deux entrées, les deux premiers multiplexeurs sont commandés par les mêmes trois bits (`ConfigLutSel(0,1,2)`), de même que

Mode	**Input(0)**	**Input(1)**	**Input(2)**	**Input(3)**
	Input16(0)		**Input16(1)**	
4-LUT	SelLut(0)	SelLut(1)	SelLut(2)	SelLut(3)
3-LUT	SelLut(0)	SelLut(1)	SelLut(2)	*Enable* de la bascule
Comm	SelLutHigh(0) et bit d'entrée	Décalage	SelLutHigh(2)	SelLutHigh(1)
Memory	Bit à insérer		Contrôle le décalage	
Input	Force la connexion		Mode de routage	
Output	Force la connexion		Valeur envoyée	
Trigger	*Enable* moléculaire		Reset du routage	
Configure	Contrôle de la configuration		Bit à insérer	

Tableau 6.10 : *Utilisation des valeurs fournies par les multiplexeurs d'entrée, en fonction du mode opératoire.*

les deux derniers le sont par `ConfigLutSel(6,7,8)`. Un petit multiplexeur à deux entrées sert ensuite à sélectionner la sortie d'un des deux multiplexeurs, en fonction d'un quatrième bit de configuration. L'avantage de ce couplage de multiplexeurs en un plus imposant réside dans le nombre d'entrées à disposition. En effet, si nous n'avions pas introduit le multiplexeur à deux entrées, les molécules n'utilisant que deux entrées n'auraient pas pu avoir le choix d'accéder les entrées directes des voisines, ou la retenue du Nord, par exemple.

En fonction du mode opératoire de la molécule, les entrées sélectionnées par nos multiplexeurs ont différentes fonctions, dont le tableau 6.10 offre un résumé.

6.3.12 Bits de configuration et reconfiguration partielle

Le comportement d'une molécule est défini non pas seulement par le contenu de son registre, mais par un total de 76 bits de configuration. Une des spécificités du circuit POEtic étant sa capacité d'autoreconfiguration partielle, nous avons séparé ces 76 bits en 5 blocs distincts, indépendamment reconfigurables par les molécules elles-mêmes. Chacun de ces blocs possède un bit supplémentaire indiquant s'il peut être reconfiguré ou non. De plus, au niveau de la molécule, trois bits de configuration spéciaux servent lors d'une reconfiguration partielle, pour indiquer son origine, et si elle doit être transmise à d'autres voisines. Au total, nous avons donc $5 + 3 = 8$ bits intouchables par une reconfiguration partielle, les autres étant modifiables par les molécules.

Le tableau 6.11 détaille les bits de configuration, séparés en cinq blocs reconfigurables (les trois premiers bits de la liste sont fixes). L'*enable* (qui correspond à un chip select) et le numéro indiqués sont utilisés lors de la configuration des molécules par le microprocesseur, présentée en page 206.

Le principe de la reconfiguration partielle est de laisser à une molécule en mode

Nombre de bits	***Enable* d'accès**	**Numéro de bit**	**Nom**	**Description**
1	cs2	17	*`ConfigPartialEnable`*	*Enable* de configuration partielle
2	cs2	15..16	*`ConfigPartialIn`*	Origine d'une configuration partielle
1	cs0	16	*`ConfigPartialLut`*	*Enable* de configuration partielle de la LUT
16	cs0	0..15	`LUT`	Contenu du registre à décalage de 16 bits
1	cs0	31	`ConfigPartialLutSel`	*Enable* de configuration partielle des entrées
12	cs0	17..28	`ConfigLutSel`	Sélection des entrées de la molécules
1	cs0	29	`ConfigSpecialInput`	Sélection d'une entrée spéciale
1	cs0	30	`ConfigDirectIn`	Sélection d'une entrée comme valeur d'une voisine
1	cs1	24	*`ConfigPartialSwitch`*	*Enable* de configuration partielle du switchbox
3	cs1	0..2	`ConfigSwitch(0..2)`	Sélection pour la ligne `ValOutN(0)`
3	cs1	3..5	`ConfigSwitch(3..5)`	Sélection pour la ligne `ValOutN(1)`
3	cs1	6..8	`ConfigSwitch(6..8)`	Sélection pour la ligne `ValOutE(0)`
3	cs1	9..11	`ConfigSwitch(9..11)`	Sélection pour la ligne `ValOutE(1)`
3	cs1	12..14	`ConfigSwitch(12..14)`	Sélection pour la ligne `ValOutS(0)`
3	cs1	15..17	`ConfigSwitch(15..17)`	Sélection pour la ligne `ValOutS(1)`
3	cs1	18..20	`ConfigSwitch(18..20)`	Sélection pour la ligne `ValOutW(0)`
3	cs1	21..23	`ConfigSwitch(21..23)`	Sélection pour la ligne `ValOutW(1)`
1	cs2	3	*`ConfigPartialMode`*	*Enable* de configuration partielle du mode
3	cs2	0..2	`ConfigMode`	Mode opératoire de la molécule
1	cs2	14	*`ConfigPartialOthers`*	*Enable* de configuration partielle des bits divers
1	cs2	0	`ConfigSequential`	Sortie séquentielle ou combinatoire
1	cs2	1	`ConfigRstValue`	Valeur de Reset de la bascule
1	cs2	2	`ConfigLocalRstEnable`	Utilisation ou non de l'*enable* local de la bascule
1	cs2	3	`ConfigClockEdge`	Flanc de fonctionnement de la bascule
3	cs2	4..6	`ConfigRstOrigin`	Origine du Reset local
1	cs2	7	`ConfigRstEnable`	*Enable* du Reset local
1	cs2	8	`ConfigSynchRst`	Reset de la bascule synchrone ou asynchrone
1	cs2	9	`ConfigMolEnable`	Autorisation de l'*enable* moléculaire
1	cs2	18	`DffOut`	Valeur de la bascule

Tableau 6.11 : *Les bits de configuration d'une molécule, classés selon l'ordre en vigueur lors d'une reconfiguration partielle. Les signaux en italique ne peuvent être modifiés durant une reconfiguration partielle.*

Configure la possibilité de forcer la modification des bits de configuration d'autres molécules du circuit. La molécule en mode Configure utilise deux signaux, l'un activant la reconfiguration, et l'autre envoyant le bit à insérer. Dans les molécules atteintes par la reconfiguration, l'accès s'effectue de façon sérielle, en traversant les cinq blocs de bits de configuration, qui peuvent ou non être touchés par la reconfiguration. Les 8 bits fixes permettent de définir une direction par laquelle la configuration partielle peut se faire, ainsi que les blocs à modifier. Si aucun des blocs ne l'est, la molécule sert simplement à transmettre l'information, de façon à ce qu'une molécule puisse influencer des parties du circuit qui ne lui sont pas directement accolées.

La figure 6.11 montre comment la molécule de coordonnées $(1, 1)$ peut reconfigurer quatre autres, aux coordonnées $(4, 2), (5, 2), (5, 0), (6, 0)$, qui ne sont pas directement contiguës. Pour ce faire, les molécules intermédiaires ont leur bit de configuration `ConfigPartialEnable` actif, et propagent donc le signal envoyé par la molécule $(1, 1)$. Il est important de noter que ces molécules peuvent continuer à effectuer leur tâche principale alors même qu'elles servent à faire transiter les bits de configuration. Les flèches en entrée des molécules correspondent à l'origine sélectionnée par `ConfigPartialIn`. Il aurait été possible de faire l'usage d'un démultiplexeur en sortie de la molécule plutôt que d'un multiplexeur en entrée, ce qui aurait impliqué qu'une molécule aurait pu choisir vers quelle direction propager la configuration, plutôt que de choisir la provenance de celle-ci. L'option retenue possède toutefois l'avantage que plusieurs molécules peuvent être reconfigurées avec les mêmes données, comme le montre la figure 6.11, ce qui ne serait pas possible si une seule sortie de configuration était activable.

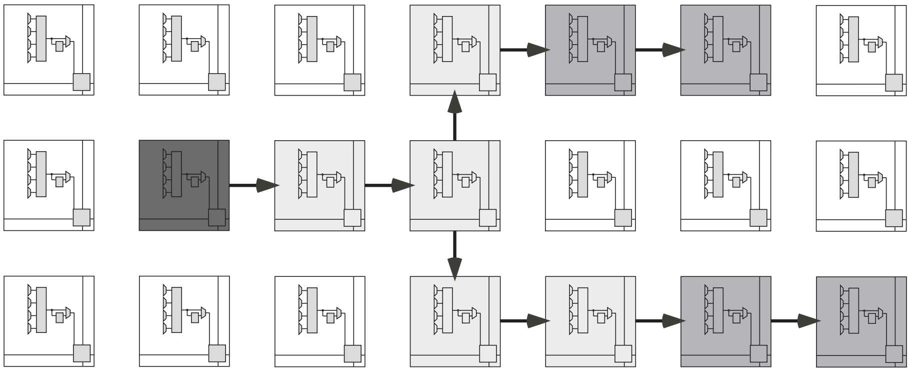


Figure 6.11 : *La molécule la plus foncée reconfigure les quatre molécules de grisé intermédiaire.*

Un bloc de bits de configuration, comme présenté à la figure 6.12, est composé d'un registre à décalage, qui est décalé lorsque le signal `conf` est à '0' et que le bit de configuration `ConfigPartial` est à '1'. Ce bit ne peut être accédé que par le sous-système environnemental, et définit donc le comportement du bloc lors d'une reconfiguration partielle. Nous pouvons observer que le bloc est *bypassé* s'il n'est pas décalé, de manière à pouvoir servir de transit à une reconfiguration devant affecter d'autres blocs ou d'autres molécules.

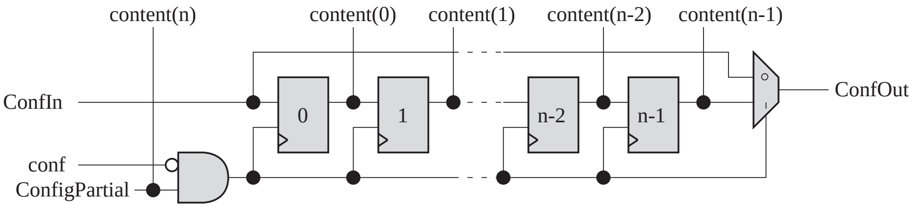


Figure 6.12 : *Un bloc de bits de configuration implémenté grâce à un registre à décalage.*

Les cinq blocs sont chaînés dans l'ordre de la figure 6.13, qui montre une reconfiguration partielle de deux blocs dont les bits de contrôle de reconfiguration sont actifs (noirs). Les deux bits de configuration `ConfigPartialIn` sélectionnent la molécule de par qui une reconfiguration partielle peut être exécutée, en sélectionnant le bit à insérer, et le signal de contrôle. Le bit de configuration `ConfigPartialEnable` indique, quant à lui, si la configuration partielle doit être propagée aux autres voisines. Ceci permet de chaîner plusieurs molécules au niveau des bits de configuration, ou de laisser une molécule reconfigurer une cousine lointaine, en traversant d'autres molécules sans les modifier.

La capacité de reconfiguration partielle, telle que nous venons de la décrire, constitue une des innovations du circuit POEtic. Elle pourra être plus qu'utile lors de l'implémentation de mécanismes ontogénétiques, et ce de plusieurs façons. Premièrement, nous verrons, en page 208, que la réalisation de registres à décalage pouvant stocker un génome peut en tirer profit, en mémorisant jusqu'à 54 bits par molécule. Deuxièmement, la phase de différenciation d'une cellule pourra se faire en reconfigurant certaines molécules de sa partie fonctionnelle.

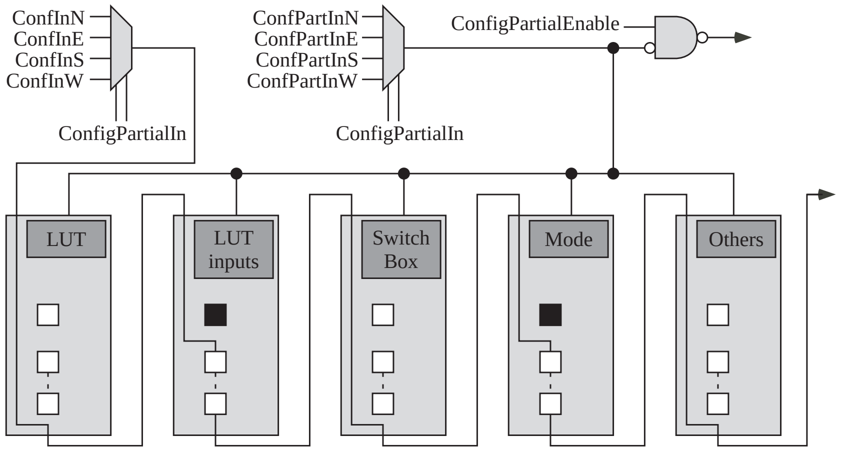


Figure 6.13 : *Les blocs de bits de configuration sont chaînés.*

6.3.13 Enable moléculaire

Une des particularités du circuit POEtic est son *enable* moléculaire, géré par les molécules en mode Trigger (page 182). Actif à '1', il est calculé en passant les sorties `MolEnableOut` de chaque molécule dans une porte ET, comme montré à la figure 6.14. De ce fait, une seule sortie à '0' force une désactivation de certaines molécules.

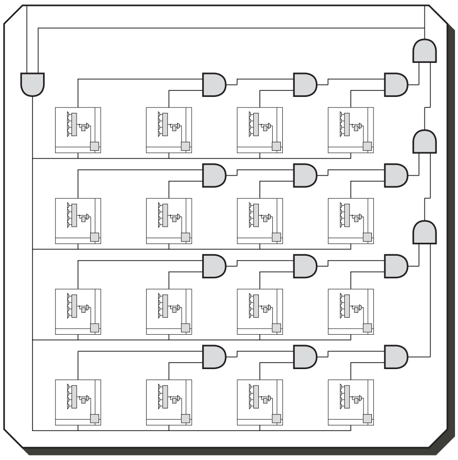


Figure 6.14 : *L'enable moléculaire global est calculé grâce à une grande porte ET.*

Dans la liste de ses bits de configuration, chaque molécule en possède un, nommé `ConfigMolEnable`, qui indique si la molécule doit être sensible ('1') ou non ('0') à l'*enable* moléculaire. Si tel est le cas, lorsque l'*enable* est à '0', les fonctions suivantes de la molécule sont affectées :

- Si la molécule est en mode Memory ou Comm, le décalage standard de la LUT est bloqué. Toutefois, la reconfiguration partielle de la molécule est autorisée.
- Si la molécule est en mode Configure, elle n'est plus autorisée à agir en tant que telle.
- La bascule de la molécule ne peut plus être chargée par un Load habituel, mais peut accepter un Reset.

Dans le cas d'un système multi-circuits, il est possible de combiner tous les signaux d'*enable* en sortie, dans une grande porte ET, et de réinjecter le résultat en entrée de chacun (signal entrant en haut à gauche de la figure 6.14). De ce fait, toutes les molécules présentes dans un tableau de circuits POEtic obéissent au même *enable* moléculaire.

Cette particularité de l'*enable* moléculaire sera notamment exploitée par les processus ontogénétiques que nous présenterons plus loin. Une cellule peut y être décomposée en une partie responsable de l'ontogenèse, et une partie fonctionnelle. Lors du processus de croissance de l'organisme, les parties fonctionnelles peuvent être inhibées, et ce dans toutes les cellules. Après la fin de la croissance, les cellules relâchent l'*enable*, et les parties fonctionnelles peuvent commencer le traitement pour lequel elles ont été configurées. De plus, des mécanismes d'autoréparation peuvent également en tirer parti, en bloquant la partie fonctionnelle des cellules, le temps que la cellule défectueuse soit remplacée par une autre.

6.3.14 Gestion de la bascule

Sur le plan fonctionnel, une molécule possède, outre une LUT, une bascule D. Elle permet d'implémenter des systèmes séquentiels, en y faisant passer la sortie de la LUT. La figure 6.15 illustre le contrôle de son fonctionnement, qui obéit notamment à plusieurs Reset.

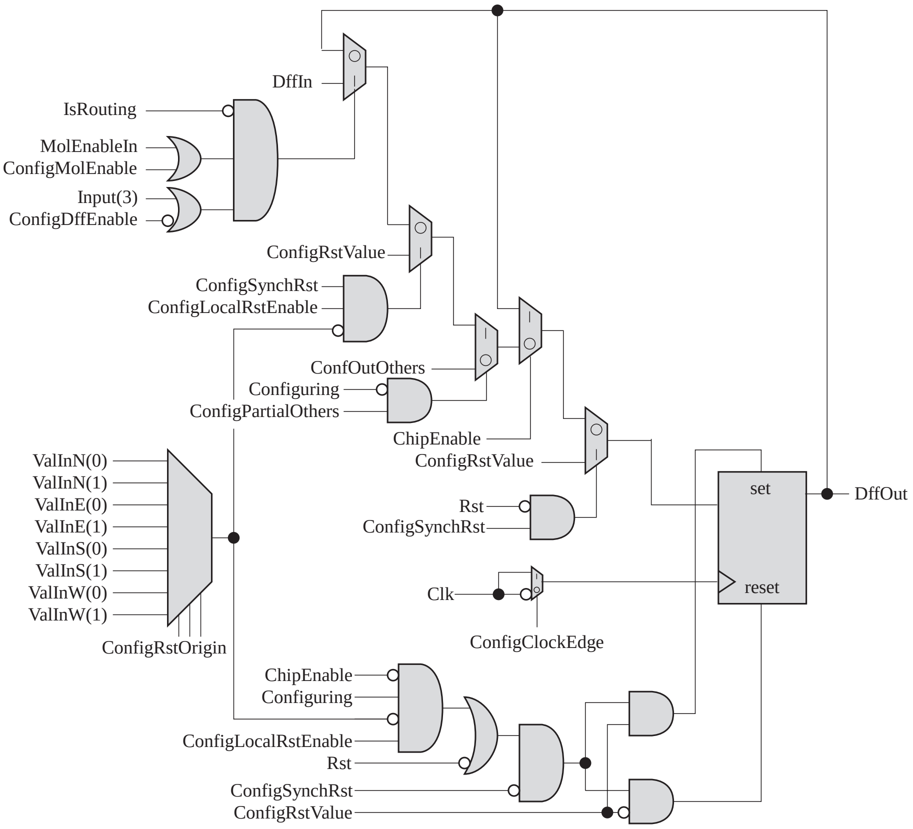


Figure 6.15 : *Gestion de la bascule d'une molécule.*

Tout d'abord, il est possible, grâce au bit de configuration `ConfigClockEdge`, de sélectionner le flanc d'horloge auquel est chargée la bascule : s'il est à '1', le flanc est montant, et s'il est à '0', le flanc est descendant.

Pour la modification du contenu de la bascule, le Reset global du circuit (`Rst`), actif à '0', a la priorité sur les autres actions. S'il n'est pas actif, toute autre action est conditionnée par le *chip enable* `ChipEnable`, qui bloque le fonctionnement des molécules s'il est inactif ('1').

Un Reset local offre aux molécules la capacité de forcer un reset de n'importe quelle autre, en faisant passer un signal par les switchboxes du routage intermoléculaire. Trois bits de configuration, `ConfigRstOrigin`, sélectionnent parmi les huit entrées du switchbox, et un Reset peut avoir lieu si ce signal est à '0' et que le bit de configuration `ConfigLocalRstEnable` est à '1'. Lors d'un reset, qu'il soit global ou local, la valeur de `ConfigRstValue` est chargée dans la bascule, de manière synchrone ou asynchrone, en fonction de la valeur de `ConfigSynchRst` ('1'=synchrone, '0'=asynchrone).

A l'instar des Resets, une configuration partielle peut modifier la valeur de la bascule. En effet, dans la chaîne des bits de configuration modifiables par les autres molécules, la bascule est placée en fin du bloc de bits de configuration divers. Dès lors, si ce bloc est soumis à une reconfiguration partielle, la bascule y participe, et elle se charge avec la valeur de `ConfigMolEnable`, qui est le dernier bit du bloc de configuration divers. La sortie de la bascule est alors récupérée par le système de configuration partielle et est envoyée aux quatre voisines, candidates potentielles à une telle configuration.

Enfin, si aucun Reset n'est actif, que le *chip enable* est actif, et qu'aucune configuration partielle n'a lieu, la bascule se trouve en fonctionnement normal, et obéit à un signal de chargement dépendant de plusieurs facteurs. Un chargement ne s'effectue que si aucun processus de routage n'est en cours, ce qui prévient le circuit de se trouver dans un état où des molécules Input ou Output seraient en cours de configuration partielle durant un processus de routage, ce qui aurait des conséquences néfastes sur le fonctionnement de l'algorithme. De plus, nous garantissons ainsi que les molécules exécutent leur tâche uniquement si toutes les connexions demandées via le routage distribué ont été créées. L'*enable* moléculaire peut également bloquer le chargement de la bascule. Le chargement n'est autorisé que si `MolEnableIn` est actif ('0'), et que `ConfigMolEnable` est également à '0'. Enfin, si `ConfigDffEnable` est à '1', la molécule doit prendre en compte un *enable* local, qui n'est autre que la sortie du quatrième multiplexeur d'entrée, `Input(3)`.

Lors d'un chargement de la bascule, le signal `DffIn` est calculé comme montré dans le schéma gauche de la figure 6.16. Si la molécule est en mode Input, Output, ou Trigger, la bascule mémorise sa valeur, et dans les autres modes, elle charge la valeur calculée par la LUT. Nous pouvons noter que le schéma de droite aurait été nettement plus judicieux, permettant à une molécule Input de faire passer la valeur reçue de l'unité de routage par la bascule.

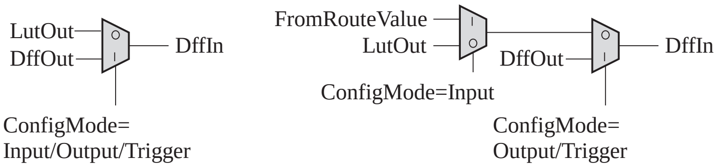


Figure 6.16 : *Valeur chargée dans la bascule, à gauche dans l'implémentation finale, et à droite telle qu'elle devrait être.*

6.3.15 Look-up table

La look-up table de la molécule est séparée en deux 3-LUTs, comme illustré à la figure 6.17. Chacune de ces LUTs possède trois entrées qui sélectionnent le bit de configuration à placer en sortie, une entrée contrôlant le décalage de ses bits de configuration, et un bit qui est inséré lors d'un décalage. En sortie, chacune fournit la valeur sélectionnée par les trois entrées, ainsi que le bit de poids fort de son registre.

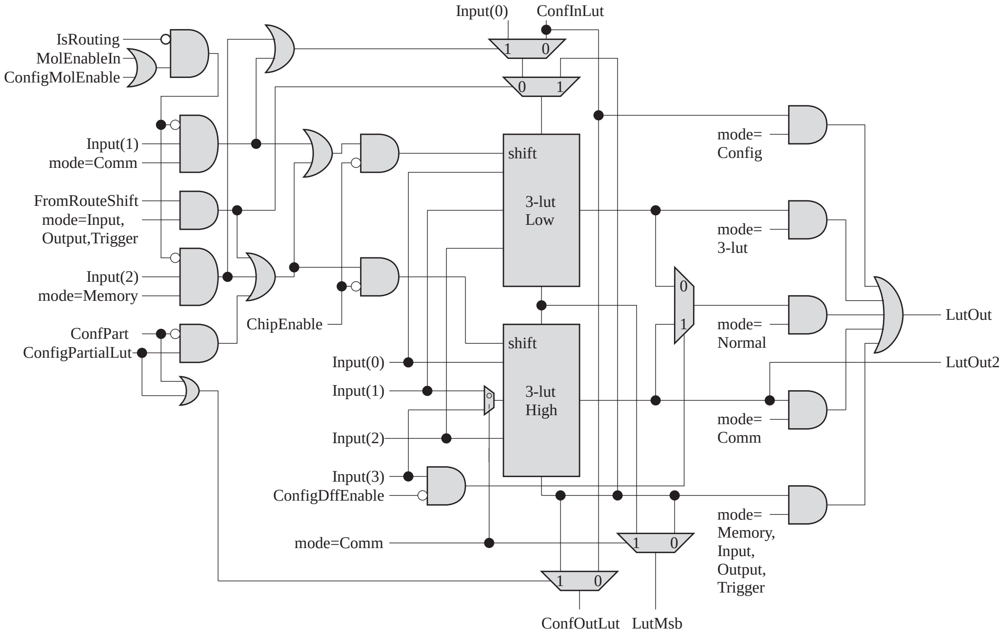


Figure 6.17 : *Mécanisme de contrôle des deux 3-LUTs.*

La première sortie dépend du mode opératoire de la molécule, et peut être la valeur calculée par la première 3-LUT, par la deuxième 3-LUT, le bit de poids fort de la deuxième 3-LUT, ou encore le bit de configuration partielle en entrée du bloc LUT[2]. La deuxième sortie n'est autre que la valeur calculée par la deuxième 3-LUT, qui est transmise à la molécule du Sud, ainsi qu'au switchbox, en mode 3-LUT. La sortie `LutMsb` est, quant à elle, le bit de poids fort du registre à décalage, qui correspond au bit de poids fort de la première 3-LUT en mode Comm, ou à celui de la deuxième dans les autres modes.

6.4 Le routage distribué

L'une des faiblesses des circuits reconfigurable existants est l'impossibilité qu'ils ont de se configurer dynamiquement, tant au niveau de la fonctionnalité des blocs de base, que du routage. Nous avons vu que nos molécules sont capables de configurer leurs voisines, ce qui offre des possibilités de configuration dynamique. Concernant le routage, nous avons couplé la grille de molécules à une grille d'unités de routage, qui implémentent l'algorithme HIDRA, décrit à la page 104. Nous n'allons pas représenter

---

[2] Cette dernière option n'est sélectionnée que dans le mode Config, et aurait dû permettre à la molécule de récupérer les bits de configuration d'une voisine pour directement les injecter dans une autre. Une erreur de design l'en empêche malheureusement.

son fonctionnement, mais uniquement mettre l’accent sur les quelques modifications que nous avons apportées aux unités de routage, qui offrent, outre le routage de type HIDRA, ce que nous appelons le routage pseudo-statique.

6.4.1 Routage pseudo-statique

Le routage selon HIDRA offre une grande souplesse, étant donné que les connexions se font sur la base d’identifiants, qui doivent être identiques pour qu’une source se connecte à une destination. Dans le cas où nous désirons réaliser un système multicellulaire où les cellules sont uniquement reliées à leurs voisines immédiates, il est nettement plus simple de pouvoir le faire avec des lignes droites (Figure 6.18) plutôt que par la connaissance des identifiants des cellules voisines.

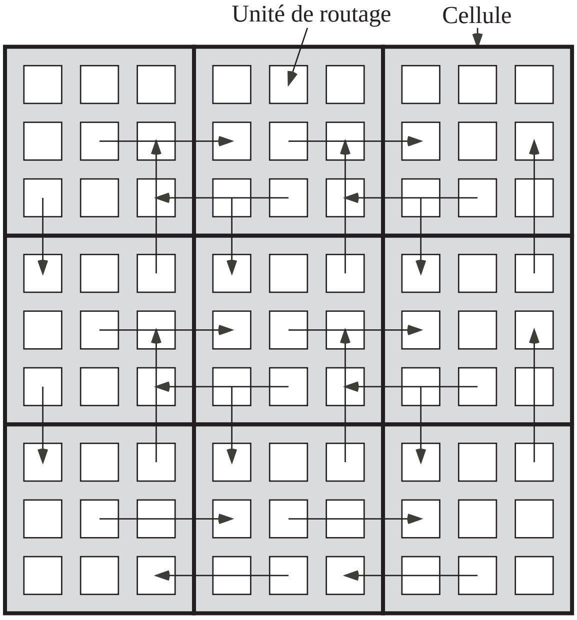


Figure 6.18 : *Connexions possibles de cellules avec leurs 4 voisines, grâce au routage pseudo-statique.*

Pour ce faire, les molécules en mode Input ont le pouvoir de forcer le routage à se faire dans ce mode, si au moins l’une d’elles place sa deuxième entrée à ’0’. Pour que l’algorithme fonctionne, il est impératif que les bits de contrôle des multiplexeurs de l’unité de routage sélectionnent la valeur de l’entrée opposée, après un Reset du routage. Lorsqu’un processus de routage est lancé, le contenu du registre de chaque molécule Input et Output est chargé dans ces bits de contrôle, de manière sérielle. La configuration se termine après 16 coups d’horloge, lorsque la sortie des molécules Trigger, dont le registre doit contenir la valeur “000...001”, s’active. Le circuit est alors opérationnel, et si le contenu des registres des molécules Input et Output ont été correctement définis, des liaisons en lignes droites directes sont créées entre les sources et les destinations, à la manière de la figure 6.18.

Sur le plan de l’implémentation, seules les commandes effectuées dans l’état sInit du contrôleur de HIDRA sont modifiées, de même que les bits de contrôle des multiplexeurs du switchbox. Au lieu de n’être que des registres chargés par le contrôleur, ils offrent également la possibilité de décaler leur contenu, à la manière d'un registre à décalage.

6.4.2 Entrées/sorties et systèmes multi-chip

Comme nous l'avons déjà mentionné, les cellules sont implémentées à l'aide de molécules. La communication intercellulaire est, quant à elle, réalisée via le niveau de routage distribué. Le circuit POEtic final n'étant pas de taille infinie, nous avons prévu de pouvoir en relier plusieurs entre eux, en un tableau de taille quelconque. Le nombre d'entrées/sorties étant également limité, il était impossible de laisser les molécules du bord du circuit communiquer directement avec leurs voisines immédiates des circuits voisins. La communication inter-circuit, pour les sous-systèmes organiques, se fait donc uniquement au travers du routage distribué, qui nécessite un nombre nettement plus réduit d'entrées/sorties.

La figure 6.19 montre l'organisation du niveau de routage distribué, et plus particulièrement la manière dont sont gérées les entrées/sorties. Chaque unité de routage envoie deux signaux vers chacune de ses voisines, comme nous pouvons l'observer au centre de la figure.

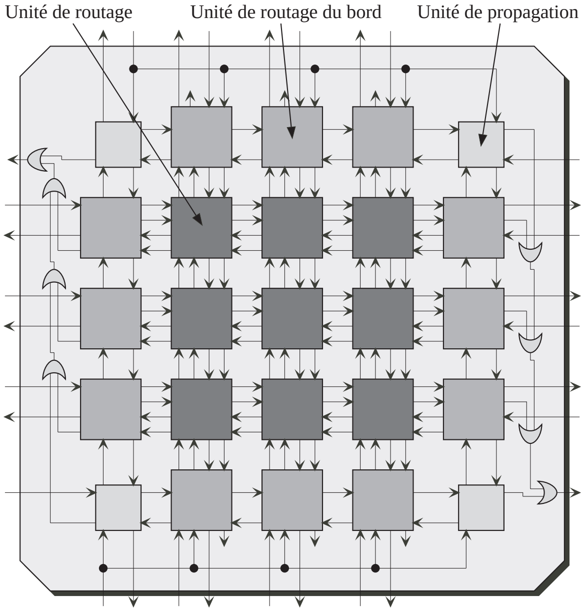


Figure 6.19 : *Les connexions entre unités de routage, ainsi que les entrées/sorties du circuit.*

Sur les bords du circuits sont disposés des unités de routage spéciales. Elles permettent de relier plusieurs circuits entre eux (Figure 6.20), et de gérer les entrées/sorties au niveau cellulaire. Elles sont composées d'un contrôleur légèrement simplifié par rapport aux unités de routage standards, ainsi que de quelques bits de configuration. Leur principe de fonctionnement dépend de ce à quoi elles sont connectées :

- Si le bord du circuit est relié à un autre circuit POEtic, les unités de routage du bord sont bypassées, et transmettent directement les signaux à l'unité standard à laquelle elles sont connectées.

- Si le bord du circuit n'est pas relié à un autre circuit POEtic, l'unité du bord peut agir comme une entrée, une sortie, ou une unité inactive.

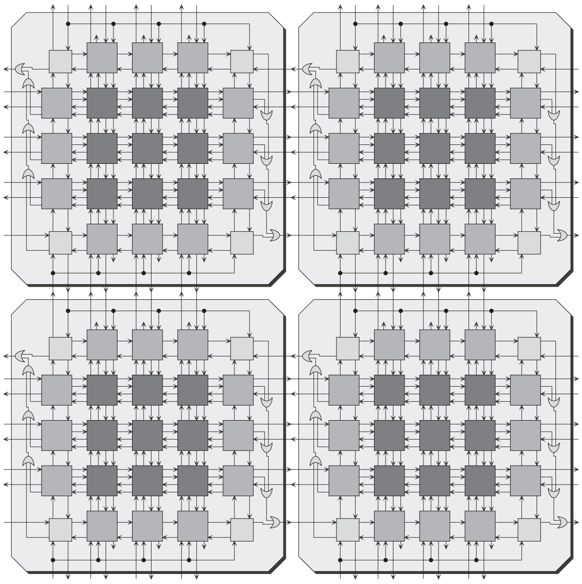


Figure 6.20 : *Connexions entre plusieurs circuits, au niveau du routage distribué.*

Le premier cas est trivial, et ne nécessite que la présence de quelques multiplexeurs sélectionnant les bons signaux à transmettre. Le deuxième, par contre, voit l'unité de routage du bord agir comme n'importe quelle autre unité de routage. Trois bits de configuration, dénommés `action(0..2)`, servent à définir exactement son comportement. Elle peut donc ne rien faire, être une sortie ou une entrée, et forcer une connexion ou non, et le tableau 6.12 définit exactement la fonctionnalité de l'unité en fonction des trois bits de configuration `action`.

`action`	**Description**
`001`	Sortie, qui ne nécessite pas forcément d'être connectée
`01X`	Entrée, qui ne nécessite pas forcément d'être connectée
`101`	Sortie, qui doit se connecter
`11X`	Entrée, qui doit se connecter
`X00`	Inutilisée, donc bypassée

Tableau 6.12 : *Bits de configuration d'une unité de routage du bord définissant son fonctionnement.*

Outre ces trois bits de configuration, une adresse est également nécessaire dans le cas ou l'unité de routage du bord est une entrée ou une sortie. Un registre à décalage de 16 bits est donc présent, et peut être chargé par le sous-système environnemental, en même temps que `action`, par l'interface parallèle.

La figure 6.21 montre le schéma d'une unité de routage du bord, avec en son centre le contrôleur simplifié. Les signaux globaux `Clk`, `Rst`, `ChipEnable`, `TriggerIn`, `RoutingMode`, `Congestion`, ainsi que ceux responsables de la configuration parallèle ont été omis, afin de ne pas surcharger le dessin. Le multiplexeur présent à droite du schéma permet de bypasser l'unité de routage, lorsqu'elle n'est pas utilisée comme entrée ou sortie du système. La valeur envoyée à l'unité de routage lui étant directement connectée à l'intérieur du circuit ne vient du contrôleur que lors d'un routage, si l'unité du bord est une entrée ou une sortie. De même, la porte ET présente en bas à gauche remplit la même fonction pour la valeur envoyée à l'extérieur du circuit.

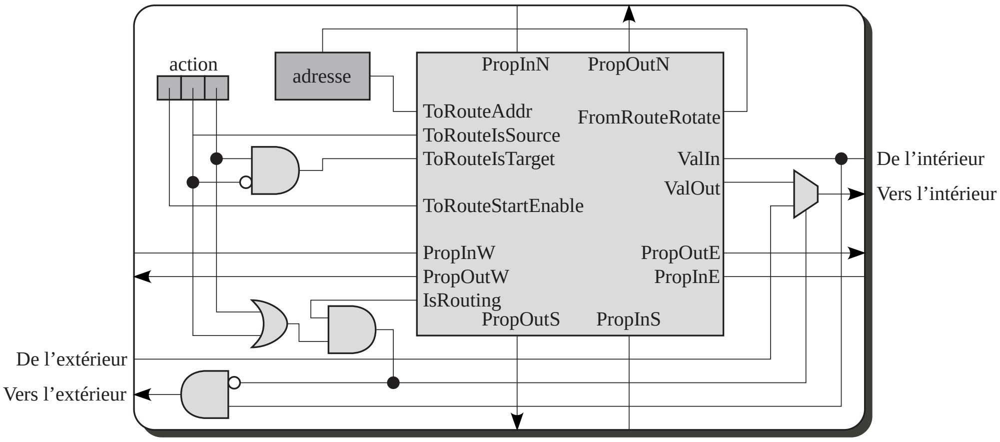


Figure 6.21 : *Une unité de routage du bord, composée d'un contrôleur, de 16 bits d'adresse, et de trois bits d'action.*

6.4.3 Interface molécule/unité de routage

Maintenant que nous avons décortiqué les molécules et les unités de routage, voyons comment les relier ensemble. Étant donné que quatre molécules ont accès à la même unité de routage, une interface est nécessaire, de façon à combiner correctement les différents signaux.

Les entrées/sorties de la molécule ont été présentées dans le tableau 6.8, à la page 188. Le tableau 6.13 montre, quant à lui, les entrées/sorties d'une unité de routage vers son élément externe.

Pour la suite de cette section, nous nommerons les signaux de la i-ème molécule $\texttt{NomDuSignal}_i$, pour plus de clarté. Les signaux en direction des molécules sont simplement identiques à ceux sortant le l'unité de routage. Il est donc du ressort de la molécules de les gérer correctement si elle n'est pas de type Input, Output, ou Trigger. Concernant ceux en sens inverse, la figure 6.22 illustre les opérations appliquées sur les sorties des molécules.

A part `InsideTrigger` et `IsTrigger`, tous les signaux calculés sont transmis à l'unité de routage reliée à l'interface. Concernant le trigger, toutes les unités de routage ont besoin d'un tel signal pour synchroniser les comparaisons d'adresse. Or, disposer d'une molécule Trigger pour chaque unité de routage serait impossible. Nous avons donc inclus, dans l'interface, un mécanisme de propagation du trigger, dans les directions Nord et Est. Chaque interface possède donc, outre des liaisons avec les molécules et les unités de routage, des connexions avec ses quatre voisin. En fonction de ses entrées Sud et Ouest, et du signal `InsideTrigger`, calculé comme indiqué en bas à droite de la figure 6.22, le trigger à transmettre à l'unité de routage ainsi qu'aux interfaces Nord et Est est calculé.

Entrée	**bits**	**Description**
`IsSource`	1	Indique si l'unité de routage est une source
`IsTarget`	1	Indique si l'unité de routage est une destination
`Address`	1	Bit d'adresse transmis à l'unité de routage durant la phase de comparaison d'adresse
`ValueIn`	1	Valeur envoyée par une unité de routage source
`StartEnable`	1	Signal forçant la création d'une connexion
`Trigger`	1	Trigger signifiant la fin de la comparaison d'adresse

Sortie	**bits**	**Description**
`ValueOut`	1	Valeur transmise à la molécule Input lorsqu'il y en a une
`Shift`	1	Indique aux molécules Input, Output, et Trigger qu'elles doivent décaler leur registre
`IsConnected`	1	Indique si l'unité de routage est connectée à sa correspondante

Tableau 6.13 : *Entrées/Sorties d'une unité de routage, communiquant avec l'interface molécule/unité de routage.*

La figure 6.23 montre le petit dispositif dédié à la propagation du trigger. Avec très peu de logique, il permet de créer des espaces d'influence des molécules Trigger. Dans la figure de gauche, les ronds noirs sont des molécules en mode Trigger, et les flèches représentent la direction de propagation de la valeur de ces triggers. L'avantage de ce système est qu'il est scalable, si l'on place des molécules Trigger de manière pas trop éloignée. Le développeur doit toutefois obligatoirement placer une molécule Trigger dans le coin Sud-Ouest de chacun de ses designs mettant en jeu le routage distribué.

Nous avons fait le choix de propager le trigger dans les interfaces plutôt que directement dans les unités de routage, ceci dans l'optique de rendre le système de routage le plus indépendant possible de l'implémentation des triggers et du registre d'adresse. De plus, la présence de cet interface entre molécules et unités de routage rend aisé la modification du nombre de molécules reliées à une unité de routage. Le code VHDL a été séparé en trois tableaux, contenant respectivement les unités de routage, les interfaces, et les molécules. De ce fait, seul le tableau des interfaces ne nécessiterait une modification dans le cas d'une réutilisation ultérieure du code.

Finalement, nous pouvons noter qu'un conflit peut naître, si plusieurs molécules Input et Output tentent d'accéder à la même unité de routage. Dans un tel cas, la valeur et l'adresse envoyées à cette unité seraient un OU logique entre les sorties des différentes molécules. Il est donc du ressort du développeur de prêter une attention particulière lors du placement des molécules accédant au niveau de routage. Ce choix a été dicté par les contraintes de place sur le silicium. Dans une autre version, il serait évidemment possible de concevoir un système de priorité, qui permettrait d'éviter les contentieux, mais qui nécessiterait un plus grand nombre de portes logiques.

6.5 Le sous-système environnemental

Le sous-système environnemental est décomposé en deux modules, un microprocesseur et un sous-système d'acquisition. Ce dernier s'occupe de récupérer les données des senseurs, et de les transmettre au sous-système organique. Il a ensuite accès aux valeurs calculées par le sous-système organique, dans les cellules, et peut les envoyer

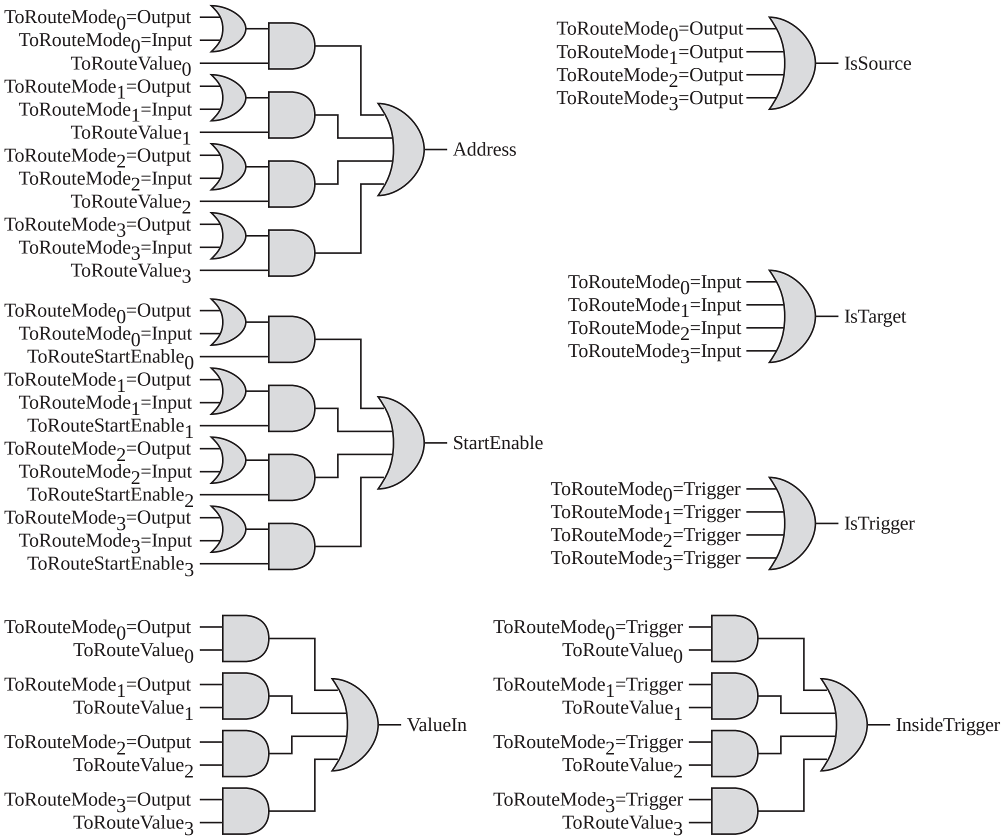


Figure 6.22 : *Les opérations appliquées aux signaux passant de quatre molécules à une unité de routage.*

sur les pins du circuit, afin d'agir sur de potentiels actuateurs. Il possède seize pins bidirectionnels, qui peuvent être utilisés indépendamment en entrée ou en sortie. Chaque ligne peut en outre passer par un registre, et est reliée à une unité de routage semblable à celles présentes dans le sous-système organique. Ces unités de routage, reliées au sous-système organique, permettent donc à ce dernier d'accéder aux entrées/sorties depuis n'importe quelle molécule.

Le microprocesseur, basé sur une architecture RISC 32 bits, possède les caractéristiques suivantes :

- Chaque instruction est exécutée en un coup d'horloge.
- Il contient un pipeline à cinq niveaux : Fetch, Decode, Execute, Memory, et Write-back.
- Les instruction sont codées sur 32 bits.
- Le code d'opération est codé sur 6 bits.
- L'architecture du processeur est basée sur une banque de registres à trois ports, de 32×32 bits, permettant trois accès simultanés, deux en lecture et un en écriture. De ce fait, un seul coup d'horloge est nécessaire à la lecture de deux opérandes et au chargement d'un résultat à une adresse différente des opérandes.
- La communication entre le processeur et la mémoire et/ou ses périphériques s'exécute selon un schéma LOAD/STORE.
- La communication entre le processeur et son environnement est réalisée via une

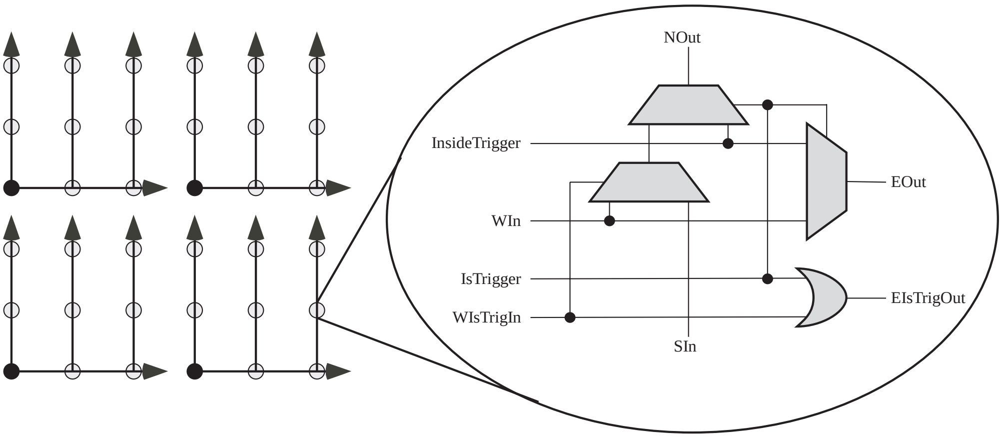


Figure 6.23 : *A droite, l’espace d’influence d’un trigger, et à gauche, l’implémentation du mécanisme de gestion du trigger.*

interface semblable au bus AMBA [13]. Le microprocesseur possède donc un contrôleur AHB interne, ainsi qu’un pont APB, qui permet l’utilisation d’un nombre indéterminé de périphériques via le bus APB.

- De la mémoire peut être ajoutée, en la connectant sur le bus AHB externe. Deux timers à 16 bits et un multiplicateur booth de taille 16 × 16 ont été intégrés comme périphériques, dans le circuit.
- Jusqu’à 5 sources d’interruption sont gérables par le microprocesseur, une priorité leur étant donnée grâce à un masque d’interruption. Un stack interne dédié aux interruptions a été inclus au microprocesseur, et permet dès lors au séquenceur d’accepter jusqu’à 32 interruptions enchaînées.
- L’ALU du microprocesseur contient un générateur de nombres pseudo-aléatoires, ces derniers étant grandement utilisés par les algorithmes évolutionnistes. Il est implémenté grâce à un Linear Feedback Shift Register de 32 bits, et deux instructions permettent de l’initialiser et de récupérer une valeur pseudo-aléatoire.

La figure 6.24 illustre la structure interne du microprocesseur.

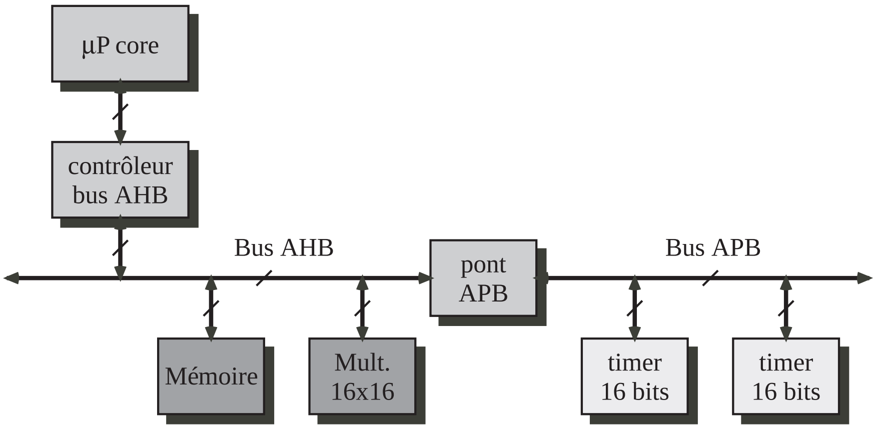


Figure 6.24 : *La structure interne du microprocesseur POEtic.*

6.6 L’interface du système

La communication entre le sous-système environnemental et le sous-système organique est gérée par une interface illustrée par la figure 6.25, que nous appellerons unité de configuration. Elle est accessible via le bus AHB, et tout accès au sous-système organique passe par elle. Elle permet également de connecter plusieurs circuits POEtic, et rend ainsi le système scalable. Cette spécificité est rendue possible par la présence de l’unité de gestion de coordonnées, qui calcule la position du circuit dans une grille de circuits interconnectés.

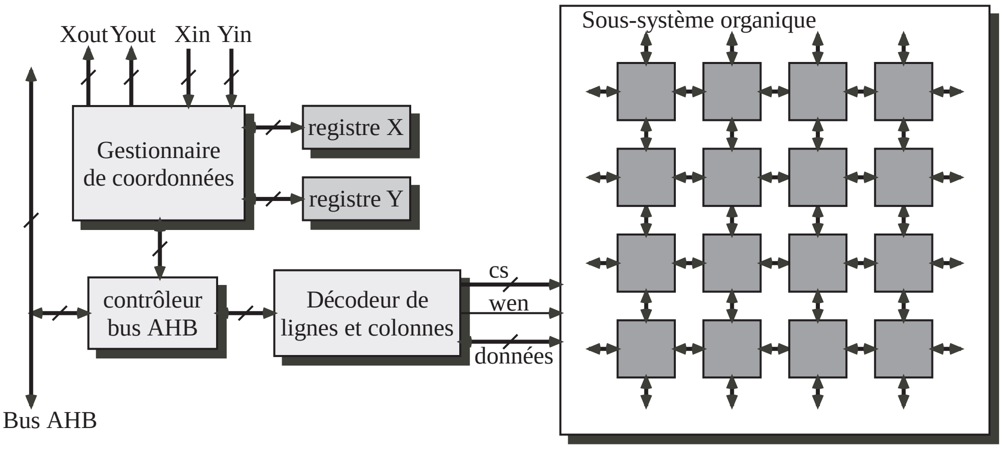


Figure 6.25 : *L’interface des systèmes.*

Le microprocesseur accède au sous-système organique grâce à plusieurs signaux : un chip select (cs), un *write enable* (wen), et les données de configuration. Étant donné qu’une molécule contient 76 bits de configuration, et que le bus de données du processeur est sur 32 bits, trois accès sont nécessaire pour définir l’entièreté d’une molécule. Le signal chip select d’une molécule est donc séparé en trois fils, qui permettent de sélectionner une des trois parties de ses bits de configuration. Le signal wen indique si l’accès se fait en lecture ou en écriture, et le bus de données contient les données à écrire dans la molécule, ou lues depuis cette molécule.

Le décodeur d’adresse permet d’effectuer de la configuration partielle d’un sous-ensemble des molécules. Le microprocesseur définit un masque de lignes et de colonnes, et de ce fait il est possible de configurer simultanément plusieurs molécules, ce qui peut accélérer le processus de configuration lors de l’utilisation d’un système cellulaire où toutes les cellules sont identiques. La figure 6.26 montre deux accès en écriture, où à chaque fois deux colonnes et deux lignes sont sélectionnées. Les quatre molécules présentes aux intersections des lignes et colonnes sélectionnées sont alors modifiées.

Concernant la scalabilité du tissu POEtic, il est nécessaire qu’un grand nombre de cellules soient implémentables. Les coûts de fabrication du circuit ne nous ont permis de réaliser qu’un système composé d’un tableau de 12 × 12 molécules, ce qui n’est évidemment pas suffisant pour la création d’un système possédant un grand nombre de cellules. L’unité de gestion de coordonnées a donc été prévue dans le but de réaliser une grille de circuits POEtic, afin de disposer d’un tableau reconfigurable plus important. Chaque circuit est connecté à ses quatre voisins, et les sous-systèmes organiques

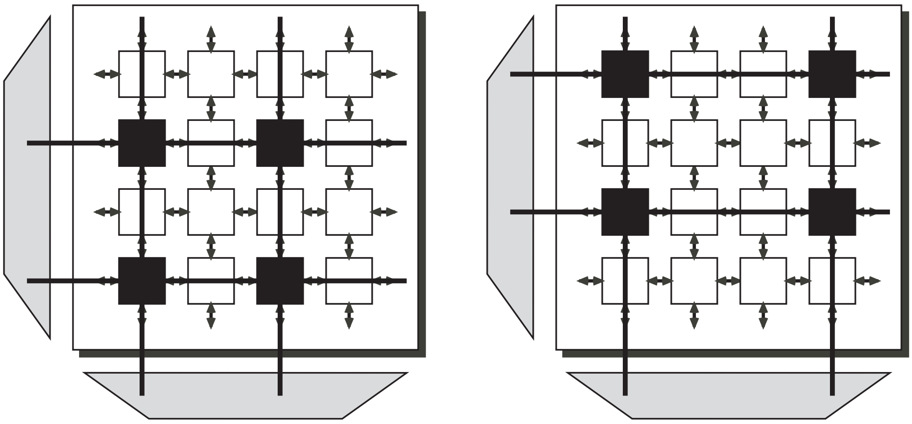


Figure 6.26 : *Configuration simultanée de plusieurs molécules.*

le sont par les unités de routage. De plus, chaque circuit est capable de calculer sa position exacte, grâce aux lignes sérielles Xout, Yout, Xin, et Yin. Lors de la mise en marche d'un tel système, un seul microprocesseur constate qu'il est le maître, et démarre le processus de propagation de coordonnées. Après un certain nombre de coups d'horloge, chaque circuit est identifié de manière unique par sa paire de coordonnées (X,Y). Ces coordonnées définissent ensuite une partie de l'adresse de mémoire réservée pour une unité de configuration donnée. Le processeur maître peut alors accéder à l'ensemble des molécules présentes.

Si nous posons que l'espace d'adressage du bus AHB/APB débute à l'adresse 0x00010000, que le tissu est composé d'un tableau de 16 $\times$ 16 circuits POEtic, et que l'espace d'adressage des molécules est placé à la fin de l'espace mémoire, l'adresse de base de l'espace mémoire d'un sous-système organique est 1_$X_3X_2X_1X_0$_$Y_3Y_2Y_1Y_0$_0000_0000_0000_0000_0000_000, où (X, Y) correspond aux coordonnées du circuit concerné. Lors d'un accès à une molécule, le microprocesseur place son adresse sur le bus AHB, et l'unité de configuration qui détecte la bonne coordonnée (X,Y) autorise l'accès à la molécule correspondante.

6.7 Fabrication du circuit

Un premier circuit de test a tout d'abord été réalisé, afin de valider le microprocesseur, les molécules et les unités de routage. Ce circuit, outre le microprocesseur, contient 12 molécules et 3 unités de routage, regroupés en 3 groupes, visibles au bas de la figure 6.27. Sa taille est de 13 mm^2, pour une technologie CMOS 0.35μm, une couche de polysilicium et 5 couches de métal. Il a servi à valider la conception des éléments principaux du circuit final, qui ont montré un fonctionnement irréprochable.

Le circuit POEtic final a été réalisé physiquement grâce à la même technologie, pour une taille plus imposante : 53.77 mm^2 (7.647 mm $\times$ 7.032 mm). Il n'a pas encore été testé, mais nous pouvons présenter le layout final, à la figure 6.28. Nous y distinguons le microprocesseur, qui occupe toute la largeur du circuit, sur le bas du schéma. Le sous-système organique représente la partie la plus importante, avec ses 144 molécules. Elles sont organisées selon des groupes de 4 molécules (Figure 6.29) reliées à une unité de routage, ces groupes étant arrangés en 9 colonnes de 4 groupes de molécules.

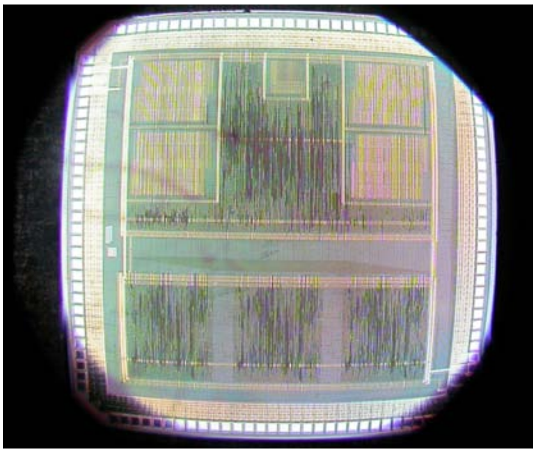


Figure 6.27 : *Photographie du circuit de test.*

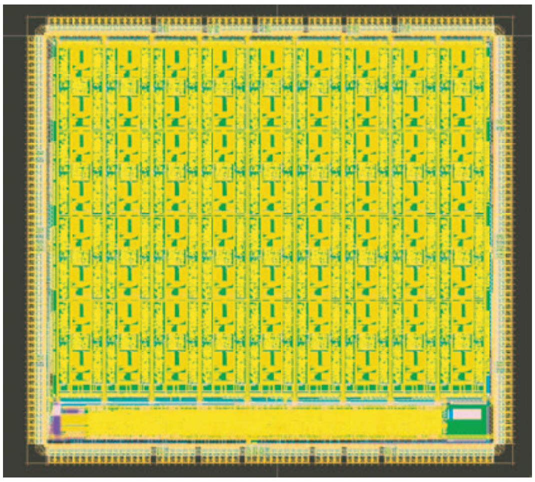


Figure 6.28 : *Le layout du circuit POEtic.*

6.8 Implémentation de composants de base

Après avoir présenté toutes les spécificités du tissu POEtic, nous nous intéressons maintenant à l'implémentation de certains composants de base tels que le registre à décalage ou le compteur. La réalisation de ces éléments peut, suivant les systèmes, être critique en terme de nombre de molécules, c'est pourquoi les caractéristiques spéciales de ces molécules aident à cette optimisation.

6.8.1 Le registre à décalage

L'importance de l'implémentation d'un registre à décalage de manière efficace n'est plus à démontrer. En effet, dans un système bio-inspiré nécessitant un génome présent dans chaque cellule, la place allouée au stockage de celui-ci peut vite prendre des proportions peu réalistes. C'est pourquoi nous présentons ici la meilleure manière d'implémenter un tel registre à décalage.

Comme nous l'avons déjà présenté, un des modes opératoires de la molécule, appelé mode Mémoire, transforme la look-up table en un registre à décalage de 16 bits.

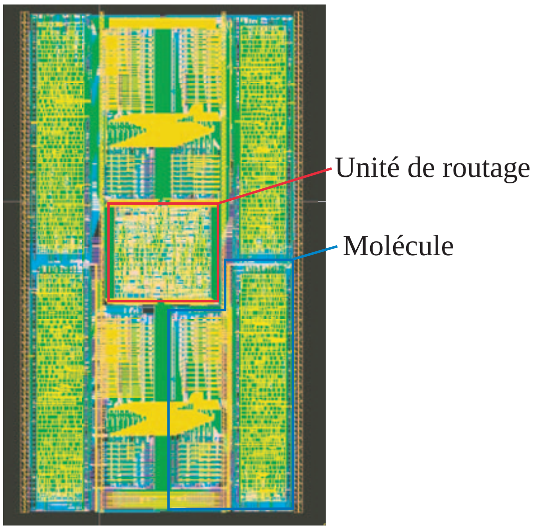


Figure 6.29 : *Un groupe de quatre molécules, et leur unité de routage.*

De plus, la bascule de la molécule peut être accolée à la look-up table, de manière à créer un registre à 17 bits. Il est ensuite possible de chaîner plusieurs molécules afin de créer un registre de $16r0 + 17r1$ bits, où $r0$ et $r1$ sont des nombres entiers positifs. Toutefois, dans le cas où le nombre de bits à stocker est considérable, le nombre de molécules nécessaires peut devenir très important. C'est pourquoi une bonne solution en terme de taille est donnée par la faculté d'auto-configuration partielle des molécules.

En effet, cette auto-configuration se fait de manière sérielle, une molécule injectant les bits de configuration un à un. Notons également qu'un des multiplexeurs d'entrées de la molécule a la possibilité de récupérer le dernier bit de la configuration partielle d'une de ces quatre voisines. Il est donc possible de chaîner plusieurs molécules sur le plan de leurs bits de configuration, en ayant une molécule supplémentaire contrôlant le décalage tout en réinjectant le génome de manière à avoir une mémoire à décalage cyclique.

Une reconfiguration partielle, comme décrite au chapitre 6.3.12, peut agir sur 5 blocs, à savoir la look-up table, les entrées de la molécule, le mode opératoire, le switchbox, et un bloc de bits divers. Il est donc possible de combiner un ou plusieurs de ces blocs afin de créer un grand registre à décalage dans la molécule. Il faut toutefois pouvoir garantir une bonne marche du circuit dans n'importe laquelle des configurations possibles. Pour ce faire, les trois points suivants doivent être respectés :

- Le bloc décrivant le mode opératoire ne peut être utilisé à cette fin, car si une molécule se trouve par hasard dans le mode trigger, input ou output, le routage dynamique a de fortes chances de se trouver paralysé par cette molécule ayant une configuration aléatoire.
- Le bloc décrivant les bits divers ne peut également pas être utilisé, car, lors d'une reconfiguration partielle, il est couplé à la valeur de la bascule D de la molécule. La limitation vient ici du fait que la valeur de cette bascule peut changer lorsque le génome n'est pas décalé. En effet, si les bits stockés dans les bits divers permettent un reset local, un tel reset peut être accidentellement exécuté, et donc changer l'état de la bascule. De plus, par défaut, la bascule se charge à chaque flanc d'horloge, et ce sont les bits divers, qui sont justement incontrôlables en

cas de stockage d'information, qui gèrent le chargement de la bascule.

- Finalement, si les bits contrôlant les multiplexeurs du switchbox servent à stocker de l'information, il est de la plus haute importance de garantir que le switchbox n'est pas impliqué dans du routage intermoléculaire. En effet, ce routage, se trouverait modifié à chaque décalage de la configuration, détruisant les chemins de données existants.

Ces trois limitations définies, nous avons donc trois blocs à dispositions :

- La look-up table, à savoir 16 bits,
- Les entrées de la molécule, soit 14 bits,
- Et le switchbox, fournissant 24 bits.

Il est dès lors possible de construire un registre à décalage de taille $16l+14e+24s$, où $l \in \mathbb{N}$, $e \in \mathbb{N}$, $s \in \mathbb{N}$. Le nombre de molécules nécessaires est alors $max(l,e,s)+1$, le "$+1$" représentant la molécule chargée de la gestion de la configuration. La figure 6.30 montre 5 molécules implémentant un registre à décalage de 160 bits, alors que 10 molécules en mode mémoire seraient nécessaires pour atteindre la même capacité.

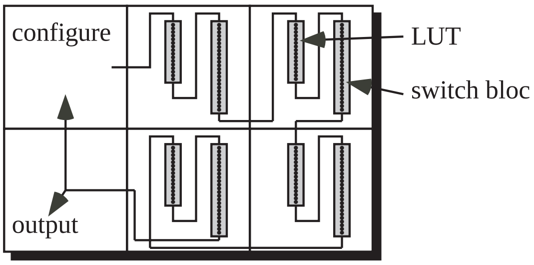


Figure 6.30 : *Un registre à décalage de 160 bits implémenté grâce à 5 molécules.*

N'oublions pas une troisième possibilité de stockage qui est tout simplement la bascule présente dans toute molécule. Elle permet de stocker un bit, ce qui est loin d'être efficace, mais permet de compléter les solutions précédentes, de manière à pouvoir construire un registre à décalage de taille quelconque.

Nous pouvons donc construire un registre à décalage en combinant les trois stockages différents, en essayant de minimiser le nombre de molécules nécessaires de la manière suivante :

- $16r0 + 17r1$ bits, dans $r0 + r1$ molécules en mode mémoire,
- b bits, dans b molécules en mode 3-LUT, où l'on n'utilise que la bascule,
- et $16l + 14e + 24s$ bits, dans $max(l,e,s) + 1$ molécules dont on utilise les bits de configuration.
- Au total, nous pouvons disposer de $16r0 + 17r1 + b + 16l + 14e + 24s$ bits, dans un nombre m de molécules défini ci-dessous :

$$m = \begin{cases} r0 + r1 + b & \text{si } l + e + s = 0 \\ r0 + r1 + b + max(l,e,s) + 1 & \text{sinon} \end{cases}$$

Pour un registre à décalage de taille n, il faut donc trouver une solution à l'équation $16r0 + 17r1 + b + 16l + 14e + 24s = n$, tout en minimisant la valeur de m telle que définie.

6.8.2 Le compteur

Dans la liste des éléments indispensables à grand nombre de designs figure le compteur. Le mode opératoire 3-LUT des molécules a été tout particulièrement conçu afin d'optimiser la réalisation de ce composant. En effet, un compteur binaire de n bits peut être construit avec n molécules, et donc un compteur à x est implémenté avec $\lceil \log_2(x) \rceil$ molécules.

Une molécule en mode 3-LUT est décomposée en deux look-up tables, la première servant au calcul du bit courant, la deuxième calculant le bit de retenue. Cette retenue est directement transmise à la molécule voisine immédiate au sud, sans le besoin de passer par un switchbox. De ce fait, la création d'un compteur en utilisant une colonne de molécules est optimale, étant donné qu'aucun switchbox n'est nécessaire. La figure 6.31 présente un compteur-trigger à 4 bits, qui, si nous modifions le contenu des LUTs, peut agir comme un compteur binaire standard.

Notons que la réalisation d'un compteur incrémenteur ou décrémenteur ne se différencie que par les valeurs présentes dans leurs LUTs.

6.8.3 Le compteur-trigger

Un compteur standard, comme présenté précédemment, permet l'accès à sa valeur en tout temps. Toutefois, certains systèmes ne nécessitent que le comptage d'un certain nombre fixe de coups d'horloge, ou d'événements. Dans ce cas il n'est pas indispensable d'avoir accès à tous les bits du compteur, mais seulement à un signal qui passe à '1' tous les n événements, en fonction d'un *enable* d'entrée. Nous appellerons un tel compteur un compteur-trigger, que nous abrégerons CT.

Les CTs, contrairement aux compteurs nécessitant un accès à la valeur courante, peuvent être optimisés, en prenant en compte les caractéristiques spéciales des molécules. En effet, puisqu'ici il s'agit d'avoir un signal qui passe à '1' durant un coup d'horloge lorsque le compteur atteint une certaine valeur, certaines optimisations peuvent se faire. Outre le compteur binaire, un registre à décalage peut implémenter un CT, de même qu'une combinaison de plusieurs CTs. Nous allons présenter les trois approches avant de les comparer, afin de pouvoir déterminer, pour n'importe quelle valeur de CT, la meilleure option.

Implémentation : compteur binaire

La première option consiste en l'implémentation d'un compteur binaire. Nous proposons de le réaliser grâce à un décompteur initialisé au reset à la valeur $CTVal - 1$. Il suffit alors de détecter le passage de '0' à '1' du bit de poids fort qui interviendra après $CTVal$ *enables*, et d'utiliser ce signal pour réinitialiser la valeur du compteur à $CTVal - 1$. La figure 6.31 montre un CT binaire de 4 bits, et le tableau 6.14 défini le contenu de chacune des LUTs.

Pour un CT de n bits, la première LUT de la molécule implémentant le bit de poids faible est de type 0, la première LUT de la molécule de poids fort est de type 3, et toutes les autres premières LUTs sont de type 2. La deuxième LUT de la molécule de poids fort est de type 4, et toutes les autres deuxièmes LUTs sont de type 1. Les valeurs de reset des différentes molécules définissent la valeur du compteur. Par exemple, pour un compteur à 13, les molécules doivent être initialisées à 12, et donc les valeurs de reset sont 1100.

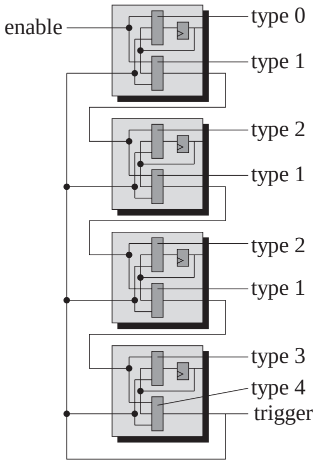


Figure 6.31 : *Un compteur-trigger binaire à 4 bits, implémenté dans 4 molécules.*

Une compteur-trigger de valeur CTVal est implémenté, conformément à l'explication de la section 6.8.2, à l'aide de $\lceil \log_2(CTVal) \rceil$ molécules.

Inputs	**Type 0**	**Type 1**	**Type 2**	**Type 3**	**Type 4**
000	1	0	1	Rst_value	1
001	1	1	0	0	0
010	Rst_value	0	Rst_value	Rst_value	1
011	Rst_value	1	Rst_value	0	0
100	0	1	0	0	0
101	0	1	1	1	0
110	Rst_value	1	Rst_value	0	0
111	Rst_value	1	Rst_value	1	0

Tableau 6.14 : *Valeurs des LUTs en fonction du type d'implémentation, pour un compteur-trigger décrémenteur*

Implémentation : registre à décalage

Comme nous l'avons vu à la section 6.8.1, les molécules permettent la réalisation efficace de registres à décalage. En plaçant dans le registre un ou plusieurs '1' séparés par un nombre égal de '0', nous pouvons construire un compteur-trigger, comme le montre la figure 6.30, où il suffit de placer un des bits à '1', tous les autres étant mis à '0'. Un compteur-trigger à 16 peut effectivement être simplement implémenté grâce à une molécule en mode mémoire, ce qui économise 3 molécules par rapport à une implémentation par compteur. De plus, un CT à 256 peut être réalisé grâce à 2 molécules en mode mémoire et une molécule en mode 3-LUT, ce qui économise 5 molécules par rapport à une implémentation par compteur.

Avec cette solution, un compteur-trigger de CTVal=$16r0+17r1+b+16l+14e+24s$ nécessite un registre à décalage de taille n, qui, comme nous l'avons déjà montré, est implémenté à l'aide du nombre suivant de molécules :

$$m = \begin{cases} r0 + r1 + b & \text{si } l + e + s = 0 \\ r0 + r1 + b + max(l, e, s) + 1 & \text{sinon} \end{cases}$$

Nous pouvons également observer qu'un CT de CTVal quelconque peut être réalisé avec un registre à décalage de taille $d{*}CTVal, d \in \mathbb{N}$, en plaçant un '1' tous les CTVal emplacements. De cette manière, un compteur-trigger de CTVal=$(16r0 + 17r1 + b + 16x + 14y + 24z)/d$ peut être implémenté avec le nombre suivant de molécules :

$$m = \begin{cases} d \cdot (r0 + r1 + b) & \text{si } l + e + s = 0 \\ d \cdot (r0 + r1 + b + max(l, e, s)) + 1 & \text{sinon} \end{cases}$$

Par exemple, un compteur-trigger à 7 est réalisé avec 2 molécules, en utilisant les bits de configuration des entrées d'une molécule (14), et une molécule gérant la configuration, ce qui est plus efficace qu'une implémentation faite avec une compteur binaire, qui nécessite 3 molécules.

Implémentation : combinaison

Les deux premières méthodes sont pour ainsi dire triviales, alors que la troisième offre d'intéressantes possibilités d'optimisation. En effet, un compteur binaire ou un registre à décalage peuvent être considérés comme des compteur-triggers, mais pour certains entiers factorisables, une combinaison des deux, ou de plusieurs registres à décalages peut s'avérer nettement plus efficace en terme d'espace.

Le compteur-trigger a, comme nous l'avons déjà mentionné, un *enable*, qui peut tout-à-fait provenir d'un autre de ces CT. Nous pouvons dès lors chaîner plusieurs CTs afin d'en créer un nouveau. Si nous chaînons n CTs, ayant chacun pour valeur $CTVal_i, i \in [1, n]$, alors nous créons un CT de valeur $CTVal = \prod_{i=1}^{n} CTVal_i$. Il faut toutefois être vigilant, et ne pas simplement chaîner sans autre les CTs. En effet, le n-ième CT ne doit être activé que si l'*enable* d'entrée, de même que les sorties de tous les CTs d'index inférieur est activé. Pour ce faire, des molécules en mode 3-LUT sont connectées aux différents CTs comme indiqué à la figure 6.32. La première LUT est chargée de détecter que les deux premières entrées sont à '1', et contient donc la valeur 10001000. La deuxième LUT, elle, est en charge de détecter que les trois entrées sont à '1', et contient donc la valeur 10000000.

Afin de calculer l'espace requis par un CT construit en combinaison, en terme de molécules, il faut tout d'abord savoir si une telle décomposition est possible. Pour ce faire, il suffit de trouver tous les facteurs premiers de notre nombre n, puis de comparer les tailles d'implémentation de toutes les combinaisons possibles de ces décompositions. En d'autre termes, il faut évaluer la taille de toutes les décompositions satisfaisant la condition suivante : $n = \prod_{i=1}^{j} CTVal_i, j \in [2, n/2], CTVal_i \in [2, n/2]$.

Conformément à la figure 6.32, le nombre de molécules nécessaires à la réalisation d'un CT de CTVal= $\prod_{i=1}^{nbcomb} CTVal_i$ est :

$$m = \sum_{i=1}^{nbcomb} nbmol(comb_i) + \lceil (nbcomb - 1)/2 \rceil$$

Où $nbcomb$ est le nombre de CT combinés, et $nbmol(comb_i)$ est le nombre de molécules nécessaires à l'implémentation du i-ème CT. Le terme $\lceil (nbcomb - 1)/2 \rceil$ se justifie de la manière suivante : Pour une combinaison de $nbcomb$ CTs, il faut

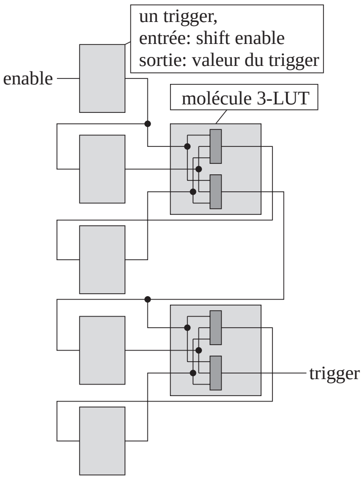


Figure 6.32 : *Un compteur-trigger obtenu par combinaison de 4 compteur-triggers.*

$nbcomb - 1$ look-up tables. Étant donné qu'une molécule en mode 3-LUT contient 2 look-up tables, $\lceil (nbcomb - 1)/2 \rceil$ molécules sont nécessaires à l'implémentation de $nbcomb - 1$ look-up tables.

Théorème 1. *Un compteur-trigger optimal obtenu par combinaison contient au maximum un compteur-trigger binaire.*

Démonstration. Supposons que nous disposons de deux compteur-triggers implémentés par des compteurs binaires de valeurs $c1$ et $c2$. Le nombre de molécules nécessaires à chacun d'eux est respectivement $\lceil \log_2(c1) \rceil$ et $\lceil \log_2(c2) \rceil$. De plus, une molécule en mode 3-LUT au moins est nécessaire à tout CT combinaison. Le nombre total de molécules est donc : $nb1 = \lceil \log_2(c1) \rceil + \lceil \log_2(c2) \rceil + 1$.

La réalisation d'un CT avec un seul compteur binaire de $c1 + c2$ nécessite $nbtot = \lceil \log_2(c1 + c2) \rceil$ molécules.

Il faut donc vérifier que $nbtot \leq nb1$.

Nous avons $c1 \geq 2$ et $c2 \geq 2$.

$\Rightarrow c1 \cdot c2 \geq c1 + c2$

$\Rightarrow 2 \cdot c1 \cdot c2 \geq 2(c1 + c2)$

$\Rightarrow 2^{\log_2(c1)+\log_2(c2)+1} \geq 2^{\log_2(c1+c2)+1}$

$\Rightarrow \log_2(c1) + \log_2(c2) + 1 \geq \log_2(c1 + c2) + 1$

Or, $\log_2(c1) + \log_2(c2) + 1 \leq \lceil \log_2(c1) \rceil + \lceil \log_2(c2) \rceil + 1$

Et $\log_2(c1 + c2) + 1 \geq \lceil \log_2(c1 + c2) \rceil$

Donc $\lceil \log_2(c1) \rceil + \lceil \log_2(c2) \rceil + 1 \geq \lceil \log_2(c1 + c2) \rceil$

La présente démonstration est similaire pour un nombre plus grand de CT compteurs. Si un CT en combinaison contient plus d'un compteur binaire, ces compteurs peuvent donc être fusionnés en un seul compteur binaire, de manière à réduire la taille du CT global. ☐

Résumé

Nous avons donc 3 façons de créer un compteur-trigger :

- Un compteur binaire.
- Un registre à décalage contenant un ou plusieurs '1'.

- Une combinaison d'un compteur binaire et d'un ou plusieurs registres à décalage.

L'encadré ci-dessous résume ces trois manières d'implémenter un compteur-trigger, et le nombre de molécules nécessaires pour chacune. Lors de la réalisation d'un de ces composants, il suffit de trouver une solution qui satisfasse la valeur du compteur, tout en minimisant le nombre de molécules nécessaire.

Compteur binaire

Valeur $= n$

Nb de molécules $= \lceil \log_2(n) \rceil$

Registre à décalage

Valeur $= (16r0 + 17r1 + b + 16l + 14e + 24s)/d$

$$\text{Nb de molécules} = \begin{cases} d \cdot (r0 + r1 + b) & \text{si } l + e + s = 0 \\ d \cdot (r0 + r1 + b + max(l, e, s)) + 1 & \text{sinon} \end{cases}$$

Combinaison

$$\text{Valeur} = \prod_{i=1}^{nbcomb} CTVal_i$$

$$\text{Nb de molécules} = \sum_{i=1}^{nbcomb} nbmol(comb_i) + \lceil (nbcomb - 1)/2 \rceil$$

6.9 Les outils de développement

Un circuit reconfigurable n'est rien s'il n'est pas possible d'y implémenter des applications. Pour ce faire, un logiciel de développement appelé POEticMol a été créé par nos soins, afin de permettre à un utilisateur de concevoir un design spécifiquement pour le tissu POEtic, et de le tester.

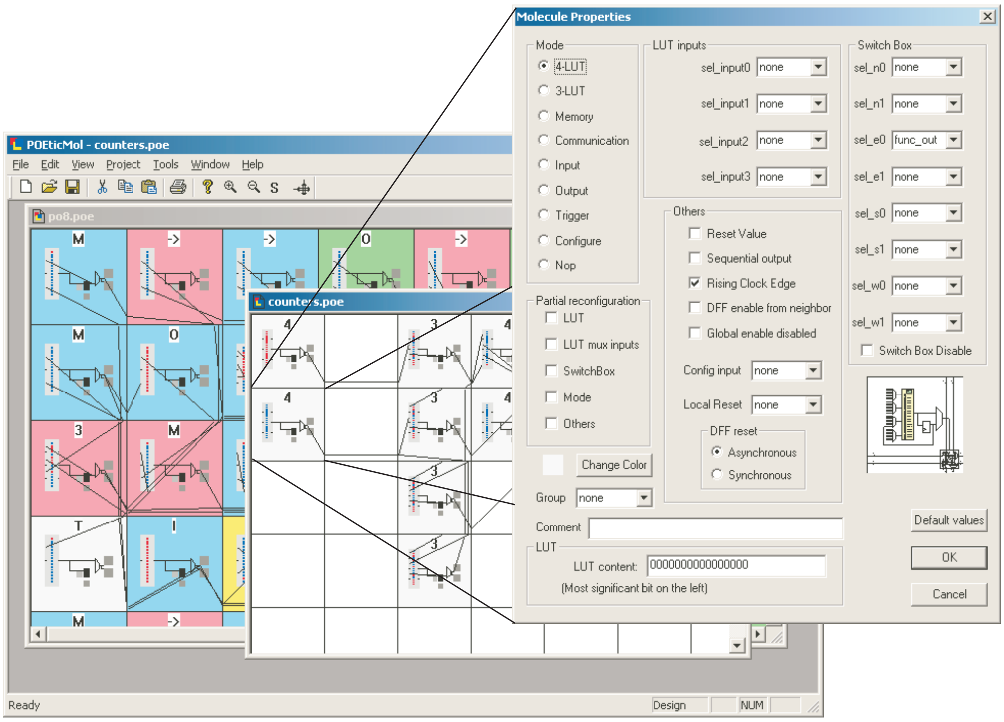
6.9.1 Design

Lors de la phase de design, POEticMol offre une interface graphique montrant la configuration des molécules du circuit (Figure 6.33). La fonctionnalité exacte d'une molécule peut être définie grâce à une fenêtre de dialogue atteignable en cliquant sur la molécule. Cette fenêtre est séparée en sections correspondant aux cinq blocs de bits de configuration, de manière à en faciliter l'usage.

Concernant le routage intermoléculaire, un algorithme de recherche du plus court chemin du type Lee a été implémenté en logiciel pour permettre à l'utilisateur de créer un chemin de donnée en cliquant simplement sur une source et une destination. L'entrée qui a été sélectionnée dans la molécule destination mémorise la sortie correspondante de la molécule source, et de cette manière un routage intermoléculaire peut être relancé après avoir déplacé des molécules, ou supprimé des chemins.

Diverses fonctionnalités ont été ajoutées au logiciel pour en faciliter l'utilisation, dont les suivantes :

- Il est possible de grouper des molécules et d'afficher les différents groupes grâce à des couleurs distinctes.
- La sélection de molécules grâce au pointeur de la souris permet de les déplacer dans le tableau, sans en écraser d'autres déjà définies.
- Le copier-coller permet de récupérer une partie de schéma et de le dupliquer ou de l'inclure dans un autre design (l'application gère plusieurs fichiers ouverts).



Mode
● 4-LUT
○ 3-LUT
○ Memory
○ Communication
○ Input
○ Output
○ Trigger
○ Configure
○ Nop

LUT inputs
sel_input0 none
sel_input1 none
sel_input2 none
sel_input3 none

Switch Box
sel_n0 none
sel_n1 none
sel_e0 func_out
sel_e1 none
sel_s0 none
sel_s1 none
sel_w0 none
sel_w1 none
☐ Switch Box Disable

Partial reconfiguration
☐ LUT
☐ LUT mux inputs
☐ SwitchBox
☐ Mode
☐ Others

Others
☐ Reset Value
☐ Sequential output
☑ Rising Clock Edge
☐ DFF enable from neighbor
☐ Global enable disabled
Config input none
Local Reset none
DFF reset
● Asynchronous
○ Synchronous

Group none
Comment

LUT
LUT content: 0000000000000000
(Most significant bit on the left)

Figure 6.33 : *L'interface graphique de POEticMol.*

Une fois un design entièrement réalisé, la traduction en un fichier de bits de configuration est immédiate, étant donné que l'utilisateur a exactement spécifié la fonctionnalité des molécules.

6.9.2 Simulation

Lorsqu'un design a été réalisé, il est possible d'en simuler le fonctionnement. Pour ce faire, le code VHDL décrivant exactement le sous-système organique est chargé dans Modelsim, un logiciel de simulation. Nous y exploitons le Foreign Language Interface de Modelsim, de par lequel il est possible d'implémenter un composant avec une DLL windows. Cette DLL est un fichier contenant du code compilé offrant à Modelsim des fonctions à appeler au démarrage et lorsque certains signaux changent d'état. Cette DLL, que nous avons développée en C++ permet dès lors de récupérer la valeur de n'importe lequel des signaux présents dans le design simulé, ainsi que de forcer leur valeur.

Notre DLL, appelée POEtic_vhdl.dll, crée un fichier pipe avec POEticMol. Il leur sert de moyen de communication, POEticMol pouvant ordonner un nombre de coups d'horloge à exécuter, et la DLL renvoyant l'état des molécules et des unités de routage. Une deuxième DLL, également chargée par Modelsim, POEtic_IO.dll permet de forcer les entrées du sous-système organique, et de récupérer ses sorties. Par défaut, elle place toutes les entrées à zéro et n'exécute aucune tâche. L'utilisateur n'a qu'à en modifier le code, grâce à des fonctions de haut niveau fournies, pour interfacer avec n'importe quel programme. Dans l'exemple de la figure 6.34, un simulateur de robot Khepera est connecté à la DLL via un pipe, et peut donc envoyer la valeur de capteurs au tissu POEtic et récupérer des données afin d'agir sur les roues du robot.

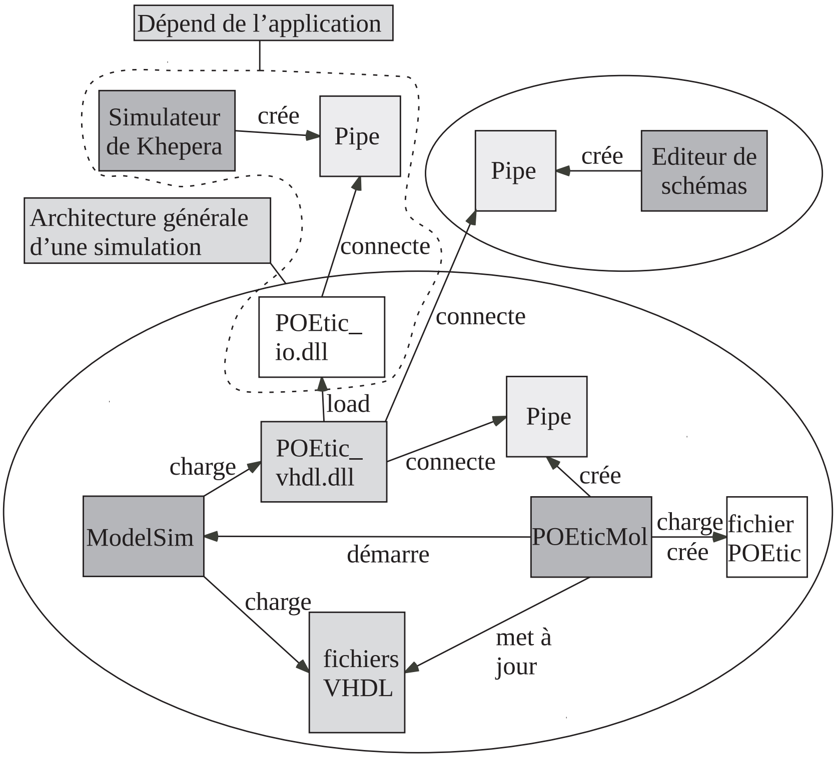


Figure 6.34 : *Réseau d'applications pour la simulation d'un design avec POEticMol.*

L'avantage de notre approche sur l'intégration de tout le système dans POEticMol réside dans la validité du VHDL. Ce dernier a été utilisé pour la réalisation du layout du circuit final, et correspond donc parfaitement à ces spécifications. De plus, durant toute la phase de développement des molécules et du routage, notre outil a été de grande utilité pour tester le VHDL.

Finalement, un outil d'édition de schémas a été développé par l'équipe de York. Il permet de créer un design au niveau des composants tels que compteurs, triggers, ou portes logiques, et de les relier entre eux. Ce logiciel peut ensuite synthétiser et placer/router le design sur les molécules et importe le résultat dans POEticMol. Lors d'une simulation il est alors possible de visualiser simultanément le résultat dans les molécules et dans l'éditeur de schémas (Figure 6.34).

Sur le plan de la programmation du processeur, un assembleur et un compilateur C, dérivé du meta-compilateur LCC [88], ont été réalisé. De plus, un émulateur a été mis au point, de manière à pouvoir tester un programme écrit pour POEtic. La suite d'outils est donc complète, est permet à un utilisateur de prendre en main aisément toutes les phases de conception d'un système à implémenter sur POEtic. Le lecteur désirant plus d'information sur ces différents outils pourra en trouver une description plus approfondie dans [232].

6.10 Conclusion

La partie reconfigurable du circuit POEtic ayant été définie, nous désirons la comparer aux approches d'autres travaux, au niveau de l'auto-configuration et du routage dynamique. Nous proposerons ensuite des améliorations possibles, qui pourraient servir lors de la réalisation d'un nouveau circuit POEtic.

6.10.1 Comparaison

Le circuit POEtic, tel que présenté, offre des caractéristiques d'auto-configuration et de routage dynamique, caractéristiques que les FPGAs commerciaux n'offrent pas encore. Concernant la première, d'autres équipes ont créé des FPGAs de ce type, basées sur des contextes multiples. Alors que dans un FPGA standard il n'existe qu'une configuration chargée, dans un FPGA multi-contexte, plusieurs configurations peuvent être présentes dans le circuit, et une seule d'entre elles est active à un instant donné. L'avantage y est de pouvoir changer de contexte de façon très rapide, en un temps situé entre 5 et 100 nanosecondes, alors que le chargement de la configuration d'un FPGA standard est de l'ordre de la milliseconde.

Dans certains de ces FPGAs multi-contextes, [170], [207], et [243], les éléments logiques ont la possibilité d'accéder aux contextes inactifs et de les modifier, en effectuant des écritures comme s'il s'agissait de RAM. Le désavantage de cette approche est toutefois la quantité de logique nécessaire au stockage et à la gestion de plusieurs contextes. De plus, l'accès aux bits de configuration des contextes n'est pas trivial, et nécessite de nombreuses ressources en terme d'éléments logiques du contexte actif, qui doit effectuer des accès mémoire parallèles. Leur utilité a toutefois été démontrée par Sidhu [215], qui tire profit des multi-contextes pour accélérer des applications de programmation génétique.

Notre approche est toute différente, nos molécules ne possédant qu'un seul contexte. L'auto-configuration s'effectue durant le fonctionnement du circuit, et l'utilisateur doit faire attention à ce qu'aucune molécule ne se trouve dans un état non désiré à la fin d'une configuration partielle, sans quoi le comportement du système peut se trouver perturbé (par exemple par un Reset du routage intercellulaire). L'avantage de notre implémentation tient en la facilité d'accès des bits de configuration. Une seule molécule, en mode Configure, peut accéder aux bits de configuration d'un nombre quelconque de voisines, et ce au travers d'un registre à décalage. Dans le cadre de systèmes POEtic, il est alors aisé, par un mécanisme ne nécessitant que peu de molécules, de recopier l'entièreté d'une cellule dans un espace libre où les bits de configuration gérant la configuration partiel ont été définis de manière identique que dans la cellule source. La reconfiguration partielle offre également des perspectives dans le contexte de l'autoréparation, où une cellule défectueuse dont la partie fonctionnelle serait tripliquée pourrait reconfigurer une cellule inutilisée avec une de ses parties fonctionnelles valides.

Concernant le routage distribué, le circuit POEtic est, à notre connaissance, le seul à proposer une approche basée sur des adresses. Une équipe japonaise a présenté en 2001 la réalisation d'un VLSI implémentant une Plastic Cell Architecture [127]. Leur circuit est également composé de deux plans, dont un contient un tableau de Plastic Part (PP), qui ne sont autre que des éléments logiques, alors que l'autre prend en charge un mécanisme de routage dynamique, par le biais de Built-in Parts (BP). La granularité des PPs y est nettement plus grosse que dans notre implémentation, puisqu'un PP contient 8 × 8 cellules, où chaque cellule est composée de quatre 2-LUTs. Aucune bascule n'est présente dans les PPs, et donc tout design y étant implémenté est asynchrone, étant donné qu'il n'existe pas d'horloge globale. Les LUTs peuvent alors servir à réaliser des latchs ou des éléments Muller (cellule Muller-C). Le niveau de routage dynamique est également asynchrone, et utilise des techniques de Wormhole [177] pour faire transiter des paquets d'information entre différentes parties du circuit. Le routage s'effectue sur la base de l'adresse de destination du message, qui correspond aux coordonnées X et Y de la cellule destinataire.

L'avantage de leur approche est évidemment la scalabilité, puisque tout y est asynchrone. De plus, les problèmes de congestion que nous pouvons rencontrer dans le circuit POEtic sont évités grâce au routage dynamique de type wormhole. En revanche, la taille de leurs BPs est nettement supérieure à celle de nos unités de routage, et ce d'un facteur 10. Et l'asynchronisme, qui est un atout en ce qui concerne la scalabilité, force la réalisation de systèmes plus complexes que les systèmes synchrones.

6.10.2 Améliorations

Comme nous l'avons déjà signalé, le circuit POEtic a été physiquement réalisé, et certaines applications ont montré quelques-unes de ses imperfections, que nous avons mentionnées au cours de ce chapitre. Nous ne reviendrons pas sur ces modifications mineures, mais allons plutôt proposer des améliorations majeures qui pourraient être faites.

Premièrement, la scalabilité est un des problèmes de notre circuit. Si plusieurs circuits doivent être reliés entre eux en un super-circuit, la fréquence d'horloge faisant fonctionner le sous-système organique doit être baissée, de manière à garantir que les signaux combinatoires aient le temps de parcourir plusieurs centimètres avant la mise-à-jour des bascules présentes aussi bien dans les molécules que dans les unités de routage. Étant donné que la communication intermoléculaire ne peut traverser les frontières du circuit, c'est à la communication intercellulaire d'offrir une meilleure scalabilité. Il serait dès lors possible d'utiliser des unités de routage de type HIDRA-L, qui ne fonctionnent que grâce à des communications locales. Une approche totalement asynchrone de ces unités de routage serait également envisageable, pour autant que le coût matériel ne soit pas trop important, en terme de nombre de transistors et de nombre de liaisons entre unités de routage.

Deuxièmement, sur le plan de la reconfiguration partielle, sur les 76 bits de configuration, 8 ne peuvent être modifiés par ce biais-là. Il sont en effet indispensables au guidage des bits de configuration, en définissant d'où une configuration est acceptée, et quels sont les bits touchés. Un mécanisme semblable à l'algorithme du petit poucet, développé au Laboratoire de Systèmes Logiques [143], pourrait autoriser la modification de tous les bits de configuration d'une molécule, et donc il serait possible d'y construire des cellules entièrement duplicables. Durant la phase d'exploration du projet une telle approche a traversé les esprits, mais la quantité de matériel qu'il aurait fallu ajouter aux molécules n'aurait pas été acceptable.

Finalement, il serait intéressant de disposer d'autoréparation au niveau moléculaire. Dans la réalisation actuelle, une cellule doit implémenter elle-même des mécanismes d'autoréparation, alors qu'il serait nettement plus élégant que les molécules soient capables d'en gérer elles-mêmes. Le problème vient ici de la complexité des molécules. Dans le projet Embryonique, les molécules possédaient de telles caractéristiques, au prix d'un ajout de logique non négligeable. Nos molécules étant nettement plus complexes que celles d'Embryonique, qui, nous le rappelons, ne sont composées que d'un multiplexeur et d'une bascule, la quantité de logique risquerait fort d'exploser.

Chapitre 7

Mécanismes POE

> La théorie, c'est quand on sait tout et que rien ne fonctionne. La pratique, c'est quand tout fonctionne et que personne ne sait pourquoi. Ici, nous avons réuni théorie et pratique : Rien ne fonctionne... et personne ne sait pourquoi !
>
> Albert EINSTEIN

LE SOUS-SYSTÈME organique, comme nous l'avons vu dans le chapitre précédent, inclut des caractéristiques non présentes dans les circuits programmables actuels, et qui ont pour but de faciliter l'implémentation matérielle de mécanismes bio-inspirés :

- Premièrement, la reconfiguration partielle des molécules permet une grande plasticité du circuit. Les molécules peuvent servir à stocker de l'information dans leurs bits de configuration, ou leur fonctionnalité peut être modifiée durant le fonctionnement du circuit, par d'autres molécules.
- Deuxièmement, le niveau de routage dynamique permet de créer des connexions entre différents endroits du circuit, typiquement entre des cellules. Cette nouvelle caractéristique va être mise à profit par des systèmes nécessitant une plasticité au niveau des connexions.
- Troisièmement, son implémentation n'offre aucun moyen de créer des courts-circuits, et rend le circuit sûr dans le cas de bits de configuration générés aléatoirement.
- Quatrièmement, nous connaissons en détail les bits de configuration du circuit POEtic, et pouvons en conséquence les faire évoluer dans le cadre d'applications de matériel évolutif.

En outre, le microprocesseur présent sur le circuit offre la possibilité d'exécuter des algorithmes évolutionnistes de façon rapide, et de pouvoir évaluer les individus en configurant le sous-système organique. Cette configuration est très rapide, puisqu'elle se fait par lot de 32 bits en parallèle.

A l'heure de l'écriture de ces quelques lignes, le circuit final n'a pas encore pu être testé, mais ce que nous présentons dans ce chapitre a été développé grâce au simulateur du sous-système organique. Seul le prototype PO a vu une réalisation sur un FPGA, 80 molécules, leurs unités de routage, ainsi que le processeur ayant été placés sur un circuit VirtexII de Xilinx. Ce prototype n'a pas été utilisé pour les autres mécanismes, car la simulation est basée sur le VHDL décrivant exactement le circuit final, et que le comportement des molécules est nettement plus facilement observé au travers d'une interface graphique qu'au travers d'un port série reliant la carte de la VirtexII à un PC.

Dans ce chapitre, nous allons donc présenter différents mécanismes, principalement ontogénétiques, qui pourront être utiles à l'implémentation de systèmes bio-inspirés sur le circuit POEtic. Nous mentionnerons brièvement les systèmes développés par nos partenaires européens, à savoir un neurone à impulsion, des mécanismes d'autoréparation, et une application de synthèse vocale. Chacun des exemples que nous allons présenter, et qui sont résumés dans le tableau 7.1, tire parti d'au moins une des caractéristiques de POEtic qui n'est pas offerte par les FPGAs commerciaux.

Type	**Exemple**	**Configuration partielle**	**Routage distribué**	**Non courts-circuits**	**Bits de configuration connus**	**Microprocesseur onchip**	**Section**
O	Autoréplication	X	X		X		7.1
O	Développement	X	X		X		7.2
PO	Prototype PO	X	X		X	X	7.3
P	Matériel évolutif			X	X	X	7.4
O	Autoréparation	X	X				7.5.1
P	Synthèse vocale		X			X	7.5.2
E	Neurone		X				7.5.3

Tableau 7.1 : *Exemples d'utilisation du circuit POEtic, indiquant les caractéristiques exploitées.*

7.1 Autoréplication

Un des axes de recherche concernant les systèmes bio-inspirés consiste en l'autoréplication, où un système évoluant sur un substrat particulier doit pouvoir y créer une copie de lui-même. Ce principe semble fondamental dans l'élaboration des mécanismes de la vie artificielle, étant donné qu'il est à la base des organismes biologiques, dans lesquels les cellules sont capables de se diviser de manière autonome. Ce furent les automates cellulaires qui servirent de premiers substrats à l'autoréplication, avec les travaux de Von Neumann sur son constructeur universel [247], puis vingt ans plus tard avec les boucles autoréplicatives de Langton [136]. Plus récemment, l'algorithme du petit Poucet [143] constitue une implémentation matérielle d'un système autoréplicatif capable d'embarquer une fonctionnalité.

Notre circuit POEtic, de par ses capacités d'auto-configuration partielle, permet d'implémenter des mécanismes approchants. Nous avons conçu une cellule composée de deux sous-systèmes : une partie fonctionnelle (PF), et une partie responsable de l'autoréplication (PA). Il est alors possible, si un autre composant PA est placé sur le substrat, de dupliquer la partie fonctionnelle de la cellule, et ce, en sauvegardant l'état de ses bascules.

Comme le montre la figure 7.1, la partie PA nécessite 15 molécules plus une molécule Trigger, tandis que la PF dépend de l'application considérée (dans l'exemple, nous avons 16 molécules). La partie fonctionnelle peut être quelconque, mais ses 8 bits de configuration fixes doivent être ajustés de manière à créer un chemin traversant toute la PF au niveau des bits de configuration, comme montré sur la figure. Le dernier bit de configuration de l'ensemble des molécules est récupéré par une des molécules de la PA, et peut être directement réinjecté par une molécule en mode Configure. En effet, pour qu'il y ait duplication, il est nécessaire qu'après cette action, la cellule de départ retrouve son état précédent. Elle va donc, durant tout le processus de transmission des bits de configuration, créer un registre à décalage bouclant sur lui-même, de façon à ce que sa partie fonctionnelle retrouve son état inchangé. La cellule qui se crée, quant à elle, va utiliser les bits reçus et directement les injecter dans sa PF.

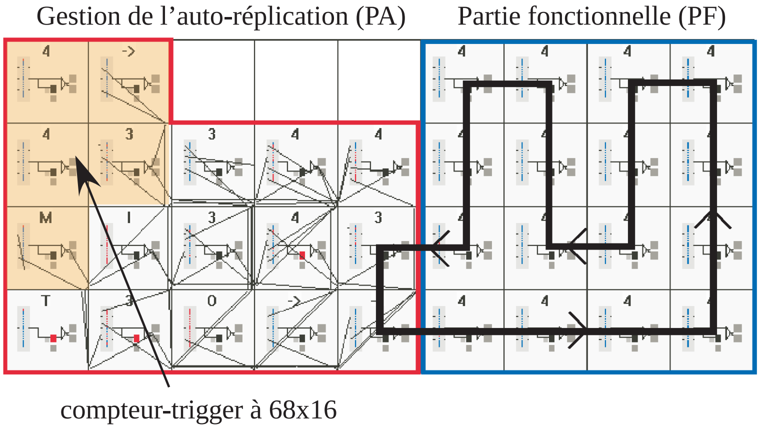


Figure 7.1 : *Une cellule capable de dupliquer sa partie fonctionnelle.*

Le mécanisme d'autoréplication est relativement simple, et exploite l'auto-configuration des molécules, ainsi que le routage distribué. Une cellule désirant s'auto-répliquer lance un routage, à partir d'une molécule Output, dont l'identifiant est 11...1. La réussite d'une duplication ne peut se faire que s'il existe quelque part sur le substrat une PA en attente, avec à ses côtés, des molécules inutilisées dont les 8 bits de configuration fixes sont identiques à ceux de la PF de la cellule de départ (gauche de la figure 7.2). Lorsque la connexion est établie, la cellule de départ décale les bits de configuration de sa partie fonctionnelle, et les envoie à la cellule d'arrivée. Cette dernière les injecte dans sa partie fonctionnelle, et ce pendant un nombre de coups d'horloge qui doit être déterminé par les PA, grâce à un compteur-trigger de valeur $68n$, pour une PF de n molécules. Dans notre exemple de 16 molécules, le compteur-trigger est réalisé grâce à 5 molécules : une molécule mémoire comptant 16, deux molécules dont on exploite les bits de configuration pour obtenir $68 = (16 + 14 + 24) + 14$, une molécule en mode Configure pour contrôler les deux précédentes, et une pour la combinaison des deux compteurs de 16 et 68.

Lorsque la configuration est terminée, la première cellule peut reprendre son acti-

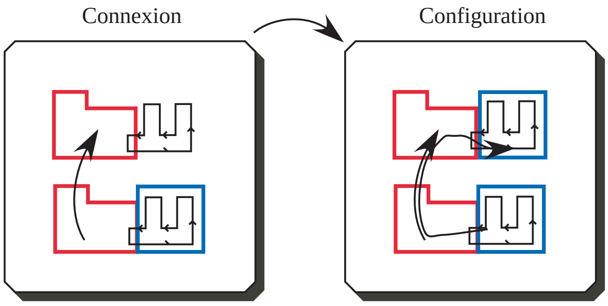


Figure 7.2 : *Duplication de la partie fonctionnelle d'une cellule, initiée par une connexion à une cellule en attente. Les bits de configuration sont ensuite envoyé à la nouvelle cellule.*

vité, et la deuxième peut, en fonction de l'application prévue, relancer une duplication ou simplement laisser travailler sa partie fonctionnelle. Il est bien clair que la partie responsable de l'autoréplication peut être modifiée en fonction du comportement désiré. Parmi les autres nombreux modes d'autoréplication qui peuvent être imaginés, citons-en deux :

- Les cellules pourraient être conçues pour s'auto-répliquer un certain nombre de fois prédéfini, en réinitialisant le routage distribué de manière à libérer leur molécule Output utilisée pour se connecter à une PA en attente.
- La duplication pourrait être lancée par la partie fonctionnelle, en envoyant un signal à la PA, par exemple lorsqu'un taux d'activation devient trop important dans le cas de réseaux de neurones à topologie variable.

Par notre exemple, nous avons montré qu'il était possible d'exploiter les capacités de POEtic pour implémenter un mécanisme d'autoréplication, sous certaines contraintes. Il est clair que la structure même de POEtic n'autorise pas une autoréplication complète dans le sens de von Neumann, puisque certains bits de configuration ne sont pas modifiables par les molécules elles-mêmes. Il serait dès lors intéressant, si la conception d'un tel circuit devait se refaire, de tenter d'y inclure des mécanismes du type de l'algorithme du petit Poucet, qui offriraient une réelle autoréplication aux systèmes cellulaires qui y seraient implémentés.

7.2 Développement

Le mécanisme que nous venons de décrire est basé sur la duplication de la partie fonctionnelle d'une cellule. Dans cette section, nous nous intéressons à des mécanismes ontogénétiques pour lesquels un génome décrivant l'organisme complet est présent dans chacune des cellules. De tels mécanismes peuvent tirer avantage des trois caractéristiques suivantes :

- Premièrement, les molécules peuvent servir à stocker un génome, en implémentant un registre à décalage.
- Deuxièmement, la propriété d'auto-configuration des molécules, où une molécule a la capacité de partiellement configurer son voisinage, permet de dynamiquement modifier le comportement d'une cellule.
- Troisièmement, le niveau de routage dynamique permet de créer des connexions entre différents endroits du circuit, typiquement entre des cellules. Un système

peut alors, à partir d'une seule cellule, s'étendre jusqu'à la création d'un organisme complet.

Un processus de développement, tel qu'il nous intéresse ici, nécessite un génome, présent dans chacune des cellules. Nous allons donc tout d'abord nous intéresser à la manière de stocker le génome dans les molécules de manière efficace. Ensuite, les deux phases du processus de développement, à savoir la croissance et la différenciation cellulaire seront explicitées.

7.2.1 Stockage de génome

Le génome décrivant un organisme est, dans tous les systèmes artificiels, à l'instar des organismes vivants, stocké dans une grande chaîne d'éléments d'informations. La vie a choisi les chaînes d'ADN, composés de 4 éléments, alors que les circuits électroniques, construits sur une base binaire, utilisent des chaînes de '0' et de '1'. Les molécules peuvent être utilisées à cette fin, en stockant le génome accessible en mode sériel. Pour ce faire nous nous basons sur l'implémentation du registre à décalage présenté à la section 6.8.1, page 208. Sur cette base, plusieurs manières de stocker le génome peuvent être développées, dont deux sont présentées ici.

Premièrement, le génome peut être simplement un long registre à décalage. Dans ce cas, et si le gène décrivant la fonctionnalité d'une cellule est de taille g et que l'organisme contient n cellules, alors l'accès au gène de la cellule c nécessitera $(c+1)g$ coups d'horloge. En effet, il faut cg coups pour atteindre le bon gène, puis g coups pour le récupérer. Ensuite, il faut replacer le génome dans son état initial, ce qui s'effectue en $(n-c)g$ coups d'horloge. La configuration de l'organisme entier s'exécute donc en ng coups d'horloge, car tous les gènes doivent pouvoir être accédés et le génome doit revenir dans son état initial. L'implémentation du génome nécessite, dans cette configuration, deux compteurs, un pour la valeur g, et un pour la valeur c.

Deuxièmement, le génome peut être décomposé en n registres à décalages de taille g, chacun contenant un gène, et l'accès peut se faire à l'aide de multiplexeurs, comme indiqué à la figure 7.3. L'accès se fait alors en g coups d'horloge, puisque n'importe quelle partie du génome peut être accédé sans aucune latence. Le temps de configuration de l'organisme entier correspond donc à celui d'une seule cellule. L'implémentation nécessite un compteur de valeur g, ainsi que les multiplexeurs nécessaires à la sélection de la bonne partie du génome.

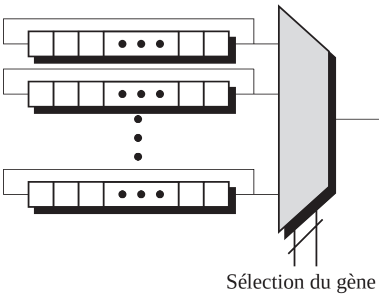


Figure 7.3 : *Un génome où un gène est directement accessible, grâce à l'utilisation d'un multiplexeur.*

Notons qu'une des différences importante entre les deux implémentations est que la première peut être implémentée avec des compteur-triggers, alors que la seconde nécessite un système contrôlant les multiplexeurs, de la forme d'un registre accessible en parallèle.

7.2.2 Croissance

Le développement d'un organisme artificielle est séparé en deux phases, la croissance et la différenciation cellulaire. La première voit le nombre de cellules croître jusqu'à atteindre le nombre nécessaire au bon fonctionnement de l'organisme, tandis que la deuxième sert à définir le comportement de chaque cellule, qui puise des informations dans le génome en fonction d'un facteur. Ce facteur dépend de la manière dont est implémenté l'ontogenèse, et peut par exemple être un identifiant unique à chaque cellule, ou des morphogènes, dans le cas d'un codage morphogénétique. Nous allons ici présenter différentes façons d'aborder la croissance, en fonction, notamment, de la manière de considérer les cellules.

Nous allons décrire deux mécanismes où chaque cellule possède un identificateur unique. Cet identificateur sert, lors de la phase de différenciation, à sélectionner la partie du génome devant être exprimée. Nous proposons les systèmes suivants :

- Un identificateur linéaire, où les cellules sont numérotées de 0 à $n - 1$.
- Un système de coordonnées à 2 dimensions, l'identifiant ayant une composante en X et une en Y, du type (x, y).

Commençons par nous intéresser aux identificateurs linéaires, et décrivons comment un tel système de croissance peut aisément être construit pour POEtic. Un organisme de n cellules contiendra des cellules ayant pour identifiant tous les entiers dans $[0, n - 1]$. L'idée de base étant de laisser le sous-système organique le plus indépendant possible d'un agent extérieur tel que le microprocesseur, qui ne sert qu'à lancer la croissance, en envoyant à une des cellules le premier identifiant, ainsi que le génome.

Rappelons tout d'abord qu'avant toute exécution d'un design dans le sous-système organique, le microprocesseur est en charge de configurer les molécules. Dans le cas d'un système ontogénétique tel que présenté, nous partons du principe qu'après configuration, des cellules totipotentes ont été placées sur le tissu. Elles contiennent, entre autre, une partie fonctionnelle ainsi que les molécules nécessaires au stockage du génome décrivant l'organisme, et sont, évidemment, toutes identiques.

Chaque cellule doit posséder une molécule en mode Input, en attente d'une connexion, sans toutefois initier une nouvelle connexion. Cette molécule sert de port d'entrée pour la phase de croissance, et doit avoir la même adresse dans toutes les cellules. Il faut évidemment garantir que cette adresse ne sera jamais utilisée par la suite lors du fonctionnement de l'organisme. Un agent externe, typiquement le microprocesseur, est alors chargé d'initier une première connexion, en utilisant une unité de routage du bord. Une fois la connexion établie avec la cellule totipotente la plus proche, l'agent externe lui envoie sont identifiant. La cellule le stocke, et calcule en même temps celui de la cellule suivante. Afin d'éviter que la cellule ne doive connaître la taille de l'organisme, il est proposé que le premier identifiant envoyé soit $n - 1$, et que chaque cellule calcule l'identifiant suivant en effectuant une décrémentation. De ce fait, la fin de la croissance est détectée par la dernière cellule, qui possède l'identifiant 0.

Lorsque l'identifiant a été totalement reçu et que l'identifiant de la cellule suivante a été calculé, la cellule reçoit le génome, qu'elle stocke dans les molécules prévues à cet effet. Ensuite, la cellule utilise une molécule en mode Output ayant le même identifiant que celle d'entrée, pour se connecter à une autre cellule totipotente. La connexion établie, elle envoie l'identifiant fraîchement calculé et son génome[1], puis attend que l'organisme entier soit ainsi construit. Elle relâche également l'*enable* moléculaire global, qui est justement utilisé pour détecter la fin de la croissance. Les molécules prenant en charge la croissance ne sont pas sensibles à cet *enable*, alors que toutes les autres le sont. De cette manière, dès que la dernière cellule a reçu son identifiant et le génome, l'*enable* global est entièrement relâché, et le reste du circuit commence à fonctionner, lançant la phase de différenciation.

Nous proposons deux manières de coder l'identifiant : en format binaire, ou en format un parmi M. La principale différence entre ces deux codages est la manière d'effectuer la décrémentation. En effet, l'identifiant étant envoyé à la cellule de manière sérielle, le format binaire peut utiliser une molécule afin de calculer sériellement l'identifiant suivant. Pour un codage en un parmi M, décrémenter signifie simplement un décalage de un bit, ce qui nécessite une molécule de moins. Dans ce cas, la cellule 0 aura l'identifiant "1000000000000000", la cellule 1 "0100000000000000", etc, de manière à ce qu'un décalage à gauche de 1 signifie une décrémentation. Notons toutefois que, si le un parmi M économise une molécule, il ne peut être réalisé de manière plus efficace que pour des organismes ayant au plus 16 cellules.

L'algorithme 7.1 explicite les actions exécutées par chaque cellule afin de mener à bien la croissance de l'organisme :

Algorithme 7.1 Traitement effectué par chaque cellule, pour une croissance basée sur un identificateur simple

```
Attendre une connexion par le port d'entrée
Recevoir son identifiant et le stocker, et calculer l'identifiant suivant
Recevoir le génome et le stocker
Si l'identifiant courant est différent de 0 alors
    Se connecter à une cellule totipotente
    Lui envoyer son identifiant
    Lui envoyer le génome
Fin si
Relâcher l'enable moléculaire global
Attendre que l'organisme soit totalement construit
```

Passons maintenant à un système de coordonnées à 2 dimensions. Dans ce cas-ci, le routage dynamique permet de créer un tableau de cellules, en connectant celles-ci à leurs 4 voisines, par exemple, et ceci de manière virtuelle. En effet, deux cellules voisines dans le tableau n'ont pas besoin d'être voisines physiques, puisque le routage dynamique permet de créer n'importe quel chemin de données. La figure 7.4 montre un organisme composé d'un tableau de 3×3 cellules, les coordonnées virtuelles des cellules ne correspondant pas aux coordonnées réelles.

Pour l'implémentation du système de coordonnées, nous partons sur le même principe que précédemment : l'agent externe lance la croissance et doit interférer le moins possible, et toutes les cellules sont, après configuration initiale, totipotentes. De même, chaque cellule possède une molécule en mode Input, et deux en mode Output, avec une adresse similaire, du type "111...111". Il est proposé que le premier identifiant envoyé,

---

[1] De la même façon que pour les mécanismes d'autoréplication, le génome de la cellule source est sauvegardé, en utilisant un principe de registre à décalage bouclant sur lui-même.

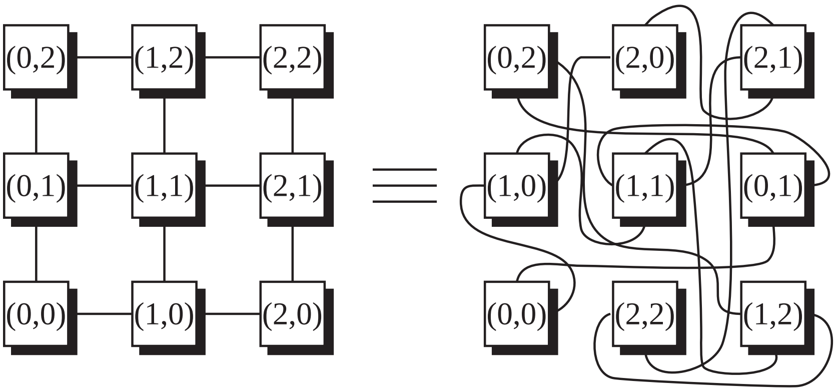


Figure 7.4 : *Le placement physique d'un tableau virtuel de* 3×3 *cellules.*

pour un tableau de $n \times m$ cellules, soit $(0, 0)$, et que chaque cellule calcule l'identifiant suivant en effectuant une incrémentation en y, de même qu'en x, pour celles ayant une coordonnée y à 0. En effet, l'argument utilisé dans le cas d'un identifiant unidimensionnel concernant le fait que la cellule n'a pas besoin de connaître la taille totale de l'organisme n'est plus valide ici, étant donné qu'une cellule doit savoir si sa coordonnée en y est 0 ou $m-1$. La fin de la croissance est donc détectée par la dernière cellule, qui possède l'identifiant $(n-1, m-1)$.

Chaque cellule est chargée de se connecter à une cellule totipotente selon les conditions suivantes, pour une cellule de coordonnée (x, y) :

- Si $y \neq m-1$, alors se connecter à une nouvelle cellule, en lui envoyant l'identifiant $(x, y+1)$.
- Si $y = 0$ et $x \neq n-1$, alors se connecter à une nouvelle cellule, en lui envoyant l'identifiant $(x+1, y)$, ou $(x+1, 0)$, ce qui est équivalent.

De cette manière, nous créons un arbre de développement tel que celui de la figure 7.5.

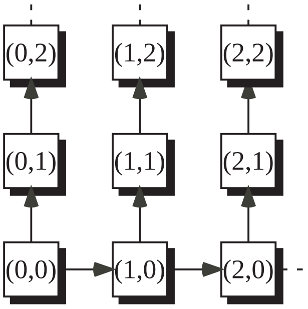


Figure 7.5 : *Arbre de développement d'un système de coordonnées.*

L'arbre de développement de la figure 7.5 place physiquement les cellules dans le tissu. Toutefois, il se peut que le placement physique diffère du placement virtuel. Afin d'obtenir un placement physique le plus proche possible du virtuel, l'entrée et les deux sorties doivent être placés tels que sur la figure 7.6. La sortie ox est alors utilisée pour connecter les cellules de la ligne 0 entre elles, et la sortie oy sert de connexion vertical. La figure 7.6 illustre bien le fait que ce placement minimise le routage nécessaire pour connecter les différentes cellules entre elles, la sortie ox, de même que la sortie oy, étant voisine immédiate de l'entrée de la cellule suivante.

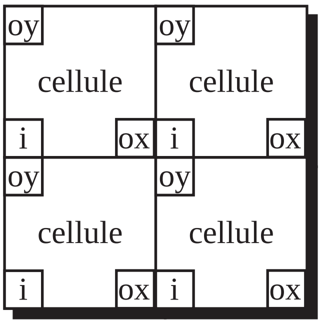


Figure 7.6 : *L'entrée et les deux sorties d'une cellule, pour obtenir un arbre de développement tel que celui de la figure 7.5.*

L'agent externe débute la phase de développement, en se connectant, via une unité de routage du bord, à la cellule totipotente la plus proche, et lui envoie ensuite son identifiant, soit la valeur $(0, 0)$ pour un tableau de $n \times m$ cellules, ainsi que le génome. La cellule récupère son identifiant, le stocke, et compare la coordonnée en Y avec 0 et $m - 1$ et la coordonnée en X avec $n - 1$. En fonction de sa coordonnée, elle se connecte à zéro, une, ou deux cellules totipotentes, et leur transmet les coordonnées calculées, de même que le génome.

De la même manière que précédemment, l'*enable* moléculaire global peut être utilisé afin de détecter la fin de la croissance. Les quelques lignes de l'algorithme 7.2 explicitent les actions exécutées par chaque cellule afin de mener à bien la croissance de l'organisme.

Algorithme 7.2 Traitement effectué par chaque cellule, pour une croissance basée sur un systèmes de coordonnées

```
Attendre une connexion par le port d'entrée
Recevoir son identifiant $(x, y)$ et le stocker
Recevoir le génome et le stocker
Si $y = 0$ et $x \neq n - 1$ alors
    Se connecter à une cellule totipotente
    Lui envoyer l'identifiant $(x + 1, y)$
    Lui envoyer le génome
Fin si
Si $y \neq m - 1$ alors
    Se connecter à une cellule totipotente
    Lui envoyer l'identifiant $(x, y + 1)$
    Lui envoyer le génome
Fin si
Relâcher l'enable moléculaire global
Attendre que l'organisme soit totalement construit
```

D'autres mécanismes de croissance peuvent évidemment être envisagés. Le système morphogénétique, développé au Laboratoire de Systèmes Autonomes dans le cadre du projet POEtic, en est un exemple. Fonctionnant sur le principe de diffusion de gradients chimiques, il a été implémenté sur le substrat moléculaire. Les détails de cette implémentation peuvent être trouvés dans [192].

7.2.3 Différenciation

Contrairement aux êtres vivants dans lesquels les cellules n'attendent pas que l'organisme entier soit terminé pour se différencier, dans notre approche, la phase de différenciation est distincte de la croissance. En effet, les organismes vivants ne pourraient terminer leur développement sans que la croissance ne soit accompagnée de différentiation, car la différenciation cellulaire influe sur la croissance des tissus. En revanche, dans notre approche, les cellules ayant un identifiant unique, il est possible de générer l'identifiant de chaque cellule sans mettre en oeuvre les mécanismes naturels complexes. De plus, la création des connexions entre cellules grâce au routage dynamique implique une séparation des deux phases. En effet, si toutes les cellules ne sont pas créées, c'est-à-dire n'ont pas d'identifiant, alors certaines connexions ne pourront se créer, bloquant ainsi le système.

La phase de différenciation débute donc lorsque toutes les cellules ont acquis leur identifiant. Elles utilisent l'information contenue dans le génome pour configurer correctement certaines de leurs molécules, en fonction de leur identificateur, qui sert d'indice pointant sur le bon gène. La propriété d'auto-configuration des molécules permet alors de partiellement configurer certaines de leurs parties, comme par exemple la look-up table. Notons également que les adresses des molécules d'entrée/sortie peuvent être reconfigurées de cette manière, laissant le génome définir la topologie du réseau cellulaire.

La configuration de la cellule s'exécute en un nombre de coups d'horloge dépendant du type de stockage du génome tel que défini à la section 7.2.1. Le génome est décalé, et les molécules concernées par la configuration sont configurées avec les nouvelles valeurs. Lorsque toutes les cellules sont ainsi correctement configurées, la création du réseau cellulaire peut débuter, mettant en jeu les adresses d'entrée/sortie fraîchement définies. Plusieurs routages s'effectuent jusqu'à ce que l'organisme entier soit opératoire, la fonctionnalité de ses cellules ainsi que leur topologie étant définies.

Dans le cas d'un système de coordonnées en 2 dimensions, et où la topologie du réseau cellulaire correspond à un voisinage de Von Neumann (4 voisins directs), la topologie n'a pas besoin d'être stockée dans le génome, mais peut être calculée en fonction de la coordonnée de la cellule. Le génome ne sert donc qu'à la fonctionnalité de la cellule, et le gène à sélectionner peut être calculé en fonction de cette coordonnée, en concaténant les coordonnées x et y.

7.3 Un exemple concret : le prototype PO

Après avoir explicité différentes manières de stocker un génome, faire croître un organisme, et différencier des cellules, nous allons maintenant illustrer ces concepts avec un exemple concret : le prototype PO. Dans le cadre du projet POEtic, ce prototype [194] visait à montrer des capacités d'évolution (P) et de développement (O), et devait être implémenté sur FPGA. Pour ce faire, nous avons réalisé, en collaboration avec Daniel Roggen, un design comprenant le microprocesseur POEtic ainsi qu'un sous-système organique de 80 molécules. Le tout fut placé sur une Xilinx Virtex XC2V3000-4FF1152.

Notons qu’une des différences importantes entre notre implémentation et le circuit POEtic final est que dans notre prototype, le sous-système organique est configuré de manière sérielle, et non parallèle. Cette modification fut obligatoire afin de réduire les problèmes de routage, et de pouvoir placer un maximum de molécules sur le FPGA.

7.3.1 Implémentation physique

La synthèse du système donne pour résultat un usage des ressources de 5% des LUTs pour le microprocesseur, 66% pour le sous-système organique, et un petit pourcentage pour la connexion des deux modules. Après placement-routage, 97% des ressources et 51% des blocs de mémoire sont réquisitionnés. En réalité, le placement s’exécute avec succès pour 100 molécules, mais le routage ne trouve aucune solution, 33 fils ne pouvant être routés. Concernant la fréquence de fonctionnement du système, les résultats après placement-routage ne sont pas significatifs, étant donné que le sous-système organique contient des boucles combinatoires, à cause des switchboxs inter-moléculaires et les switchboxs du routage dynamique. Toutefois, l’expérience montre que le système fonctionne parfaitement à 10MHz. Notons que le fonctionnement correct du système n’était pas garanti, étant donné que le placement-routage devait se faire sans contraintes de timing. En effet, les outils de Xilinx, à cause des boucles combinatoires, sont incapables de mener à bien le placement-routage en prenant en compte ces contraintes.

Il est bien clair que 80 molécules dans un FPGA contenant 14000 éléments de base est loin d’être un résultat exceptionnel. Toutefois, il faut prendre en compte le fait qu’un FPGA n’est pas fait pour émuler un autre FPGA, et ceci pour deux raisons majeures. Premièrement, le routage du FPGA émulé, qui est un des éléments critique dans un tel circuit, demande énormément de ressources. Deuxièmement, chacune de nos molécules, qui, en terme de complexité, correspond plus ou moins à un des 14000 éléments de bases du FPGA cible, contient 76 bits de configuration, et chacun de ces bits est implémenté dans un élément de la cible, ce qui implique une grande utilisation des ressources matérielles. Ces résultats montrent bien que notre système n’est qu’un prototype, et que la réalisation physique du circuit POEtic est plus que nécessaire afin d’obtenir un système efficace.

Pour ce prototype, nous avons choisi de réaliser un système cellulaire où chaque cellule est composée d’une fonction quelconque à trois entrées. L’évolution permet alors d’évoluer la fonctionnalité de chacune des cellules, ainsi que la connectivité intercellulaire. Étant donné le peu de molécules à disposition, il n’a été possible de placer que deux cellules sur les 80 molécules. La fonctionnalité de l’organisme est donc une fonction logique, dépendante de l’application, et le système a été évalué avec trois applications : un multiplexeur, un additionneur avec retenue, et une application de contrôle de robot. Nous allons décrire la structure cellulaire implémentée, ainsi que les concepts ontogénétiques utilisés, avant d’entrer plus en détail dans les différentes applications.

7.3.2 Structure cellulaire

Le principe fondamental de notre tissu PO est que chaque cellule contient le génome décrivant entièrement l’organisme. La structure cellulaire suit le principe des trois couches introduites dans [245], et que nous avons traitées en page 62. La couche génotypique contient le génome, la couche de mapping contient deux éléments, un pour la phase de croissance, et un pour la différenciation, et finalement, la couche phénotypique contient la fonctionnalité de la cellule, à savoir trois entrées, une sortie, et une look-up table à trois entrées. La figure 7.7 explicite ces trois couches, et la figure 7.8 montre l'emplacement de ces couches dans la cellule. 40 molécules sont nécessaires à l'implémentation d'une cellule, ce qui ne permet d'avoir que deux cellules dans le système. La répartition des trois couches en terme de molécules est la suivante :

- La couche génotypique occupe 16 molécules en mode mémoire. Le génome représente une organisme de 4 cellules au maximum, et donc les mêmes cellules peuvent être utilisées pour une application plus conséquente. Nous pouvons cependant noter qu'il serait possible d'exploiter les bits de configuration des molécules pour le stockage du génome, ce qui aurait pour conséquence de réduire la taille de la cellule. Ne pouvant de toute façon pas placer plus de deux cellules dans les 80 molécules, nous avons gardé les molécules en mode mémoire, ce qui permettait de simplifier le traitement effectué par le microprocesseur.
- La couche de mapping est implémentée sur 18 molécules : 8 molécules pour la croissance, et 10 pour la différenciation.
- La couche phénotypique n'est réalisée qu'avec 5 molécules : 3 pour les entrées, une pour la sortie, et une pour la partie fonctionnelle (une 3-LUT).

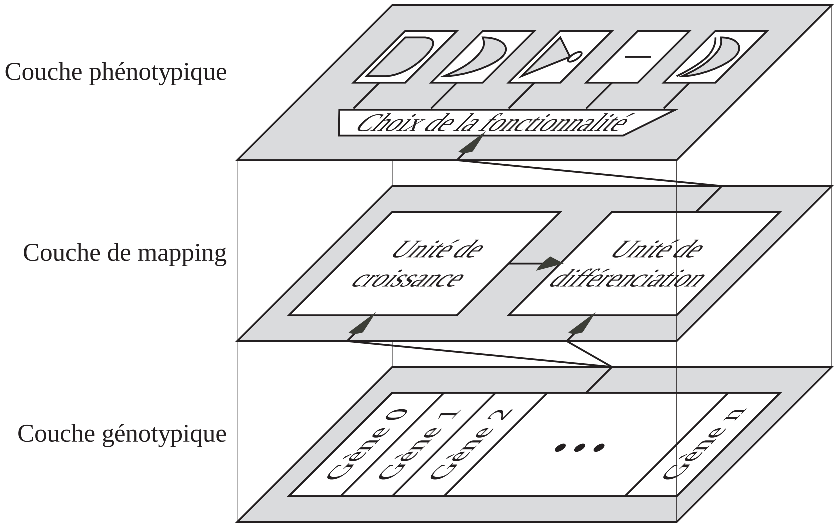


Figure 7.7 : *Les trois couches composant une cellule. La couche génotypique contient le génome, la couche de mapping s'occupe du développement, et la couche phénotypique correspond à la fonctionnalité de la cellule.*

7.3.3 Mécanisme de développement

Un des buts principaux du prototype PO étant de montrer une capacité de développement d'un organisme à partir d'une cellule, les cellules ont été spécifiquement définies dans cette optique. En effet, la partie fonctionnelle n'occupe que 12,5% de la cellule, le reste des molécules réalisant le mécanisme de développement. Le mécanisme ontogénétique est basé sur un système d'identificateur linéaire, codé en 1 parmi

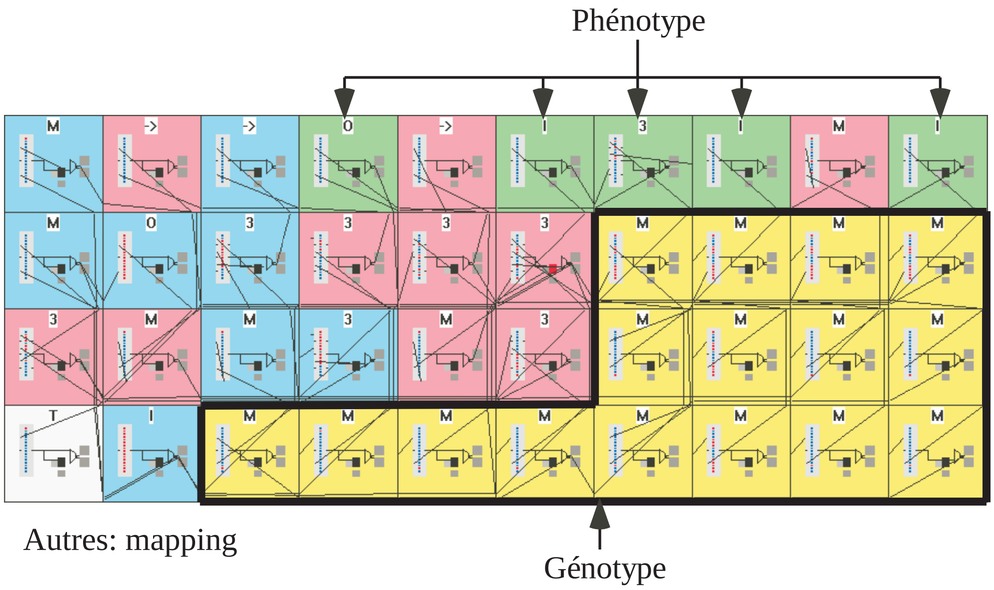


Figure 7.8 : *Implémentation d'une cellule sur les molécules, montrant les trois blocs : génome, développement, fonctionnalité.*

M, sur 16 bits. La cellule numéro 0 a un identifiant 100...0, la cellule numéro 1 a l'identifiant 010...0, etc.

Croissance Après la configuration des molécules, toutes les cellules sont identiques, totipotentes, non connectées, et non différenciées. La phase de croissance est initiée par le microprocesseur, qui utilise une unité de routage du bord pour se connecter à la cellule totipotente la plus proche. Chaque cellule possède une molécule en mode Input, avec l'adresse 11...111. Une fois la cellule connectée grâce au routage dynamique, le microprocesseur lui envoie son identifiant en mode sériel. L'opération est effectuée en 17 coups d'horloge, de manière à ce que la cellule génère l'identifiant de la cellule suivante, en décrémentant son propre identifiant. Après les 17 coups d'horloge, la cellule détecte si elle est la dernière (ID zéro) ou non, en observant le bit de poids fort de son identifiant. Si elle n'est pas la dernière, elle se connecte à une autre cellule totipotente, puis lui envoie l'identifiant qu'elle vient de calculer. En même temps, elle relâche l'*enable* global qui permet, lorsque la dernière cellule est atteinte, de lancer la différenciation (les cellules responsables de la croissance sont insensibles à cet *enable*, alors que toutes les autres le sont).

Différenciation Lorsque toutes les cellules ont reçu leur identifiant, l'*enable* global est relâché, et les molécules en charge de la différenciation prennent le relais. Le génome, composé de 16 molécules en mode mémoire, permet de décrire un organisme contenant jusqu'à quatre cellules. Chaque cellule est donc décrite par quatre molécules, soit 64 bits. Chaque adresse d'entrée de la cellule doit alors être configurée avec 16 bits, et les 16 derniers bits servent pour moitié à la définition de la fonctionnalité de la cellule.

Le génome est donc, dans chaque cellule, décalé. L'identifiant, contenu dans une LUT, est lui décalé tous les 64 coups d'horloge, et lorsqu'il vaut 0 (100...000), le gène récupéré est directement utilisé pour configurer les trois entrées de la partie fonctionnelle, ainsi que la LUT. Après 4x64 coups d'horloge, le génome est donc dans son état initial, et toutes les cellules de l'organisme sont correctement différenciées. Un routage dynamique est alors lancé jusqu'à ce que toutes les connexions intercellulaires soient créées. Le circuit est ensuite prêt à fonctionner, le microprocesseur peut forcer les entrées et récupérer la sortie, afin de calculer le fitness de l'individu testé.

7.3.4 Evolution

Alors que la majorité des circuits électroniques sont conçus par des ingénieurs, le concept de matériel évolutif délègue cette tâche à un système automatique. Le microprocesseur est ici en charge d'exécuter un algorithme génétique faisant évoluer une population d'individus de deux cellules. Le système possède six entrées et deux sorties, placés tels que sur la figure 7.9, qui sont notamment utilisées comme résultat et retenue pour l'additionneur. Une entrée d'une cellule peut donc choisir entre une des six entrées et la sortie d'une des cellules, et a donc huit choix possibles, définis par 3 bits. Comme nous l'avons vu, la cellule est définie par sa fonctionnalité, soit 8 bits, et par la connectivité de ses entrées, soit 3 bits chacune. Au total nous avons donc 17 bits par cellule, pour un total de 34 pour un individu.

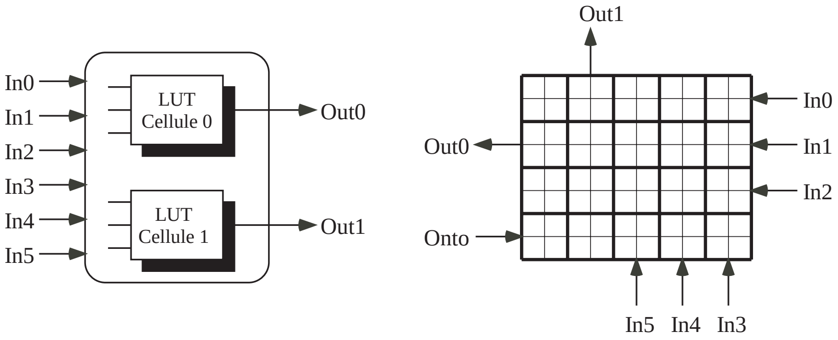


Figure 7.9 : *A gauche, la partie fonctionnelle de l'organisme, et à droite la répartition des entrées/sorties sur le circuit.*

Evolution de fonctions logiques Dans cette application, le prototype PO est utilisé pour l'implémentation de fonctions logiques, un additionneur, et un multiplexeur. Les trois premières entrées du système sont forcées par le microprocesseur, tandis que les trois suivantes sont fixées aux valeurs 0,1 et 0. L'application multiplexeur ne comporte qu'une sortie, alors que l'additionneur en compte deux, une pour le résultat, et une pour la retenue. Chaque individu est évalué en lui présentant toutes les entrées possibles, soit huit possibilités, et en vérifiant que la sortie corresponde à la fonction désirée.

Un algorithme génétique est exécuté par le microprocesseur, avec les paramètres décrits dans le tableau 7.2, afin de trouver une solution efficace. Le fitness d'un individu est calculé en comptant le nombre de sorties correctes sur les huit évaluées. Dans le cas du multiplexeur, le fitness est compris entre 0 et 8, et dans celui du multiplicateur, étant donné qu'il a deux sorties, entre 0 et 16. La figure 7.10 montre l'évolution du fitness moyen, ainsi que celui du meilleur individu, pour le multiplexeur et l'additionneur. Sur 32 exécutions de l'algorithme génétique, 31 ont réussi à trouver une

Paramètre	Valeur
taille de la population	200
taux de crossover	30%
taux de mutation	5%
sélection	au rang, dans les 20 meilleurs
élitisme	oui

Tableau 7.2 : *Paramètres de l'algorithme génétique*

solution pour le multiplexeur, en 50 générations au maximum (le seul échec a terminé avec un meilleur fitness de 7). Concernant l'additionneur, 20 exécutions sur 32 se sont terminées avec succès, les 12 échecs arrivant à un fitness de 14. Ces résultats correspondent au fait que l'additionneur, avec deux sorties, est un système plus complexe à réaliser que le multiplexeur.

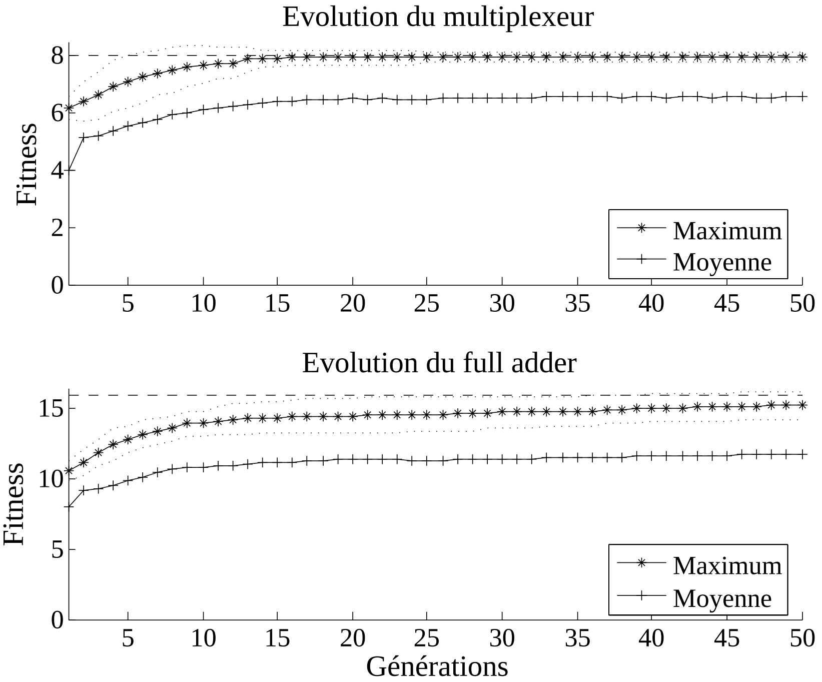


Figure 7.10 : *L'évolution du fitness moyen et maximum, pour un multiplexeur et un additionneur complet.*

Evolution d'un contrôleur de robot La deuxième application du prototype PO vise à contrôler un robot Khepera [111] dans une tâche d'évitement d'obstacle. Le Khepera est un robot qui, dans sa version de base, possède huit capteurs de proximité infrarouges, ainsi que deux roues. La figure 7.11 illustre la position des capteurs du robot, ainsi que de ses deux roues. Notons que, notre système à deux cellules ayant six entrées, les capteurs latéraux sont groupés, la valeur du capteur le plus actif étant utilisée en entrée. Une entrée de l'individu est à '1' si la valeur du capteur est supérieure à un certain seuil, et à '0' sinon. La vitesse des roues du robot est directement calculée en fonction des valeurs des capteurs de proximité infrarouge du robot, grâce aux deux cellules de notre tissu. Chacune des sorties commande une des deux roues, et les moteurs du Khepera ont une période de 100ms durant laquelle la vitesse reste constante. Le tissu POEtic met donc cette vitesse à jour à la fin de chaque période, en fonction de la sortie. Si la sortie est à ’1’, le moteur fonctionne à +80mm/s, tandis qu’une valeur ’0’ le fait fonctionner à -80mm/s. Dans notre modèle, les moteurs ne peuvent jamais s’arrêter, ce choix ayant été dicté par le petit nombre de cellules de notre tissu.

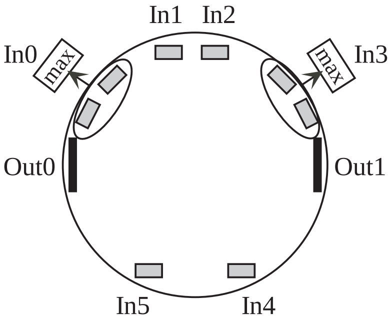


Figure 7.11 : *Les entrées (capteurs de proximité) et sorties (moteurs des roues) du robot Khepera.*

Le fitness d’un individu est évalué en fonction du comportement du robot de manière à maximiser le mouvement d’avancement, tout en minimisant les contacts avec les murs. Le microprocesseur le calcule en sommant la vitesse des moteurs lorsqu’ils tournent les deux vers l’avant [70]. Cette technique simple permet de favoriser les robots qui évitent les obstacles, car la vitesse des moteurs d’un robot bloqué contre un mur est nulle, de par le frottement avec le sol. L’algorithme génétique exécuté par le microprocesseur utilise les mêmes paramètres que pour l’application précédente ; toutefois, afin de gagner du temps, l’évolution s’est faite en simulation, seul le meilleur individu ayant été implémenté dans le robot réel. La figure 7.12 montre l’évolution du fitness moyen et maximum, et nous pouvons noter que dès la dixième génération, de très bons candidats furent obtenus.

7.3.5 Remarques conclusives

Par ce prototype du tissu POEtic, nous avons démontré que le routage dynamique ainsi que la capacité d’auto-configuration des molécules sont des caractéristiques importantes qui peuvent être exploitées par un système ontogénétique. Le système présenté, capable de développer un organisme à partir d’un tableau de cellules totipotentes sur ordre d’un agent externe, peut être une base sur laquelle construire un organisme auto-réparateur, où une cellule endommagée pourrait être remplacée par une autre. De plus, ce prototype a permis de vérifier le bon fonctionnement du microprocesseur, qui n’était alors pas encore figé dans le circuit final.

7.4 Matériel évolutif non-contraint

Le routage interne du sous-système organique de POEtic est uniquement composé de multiplexeurs. Bien qu’allongeant potentiellement les délais entre les registres, cette

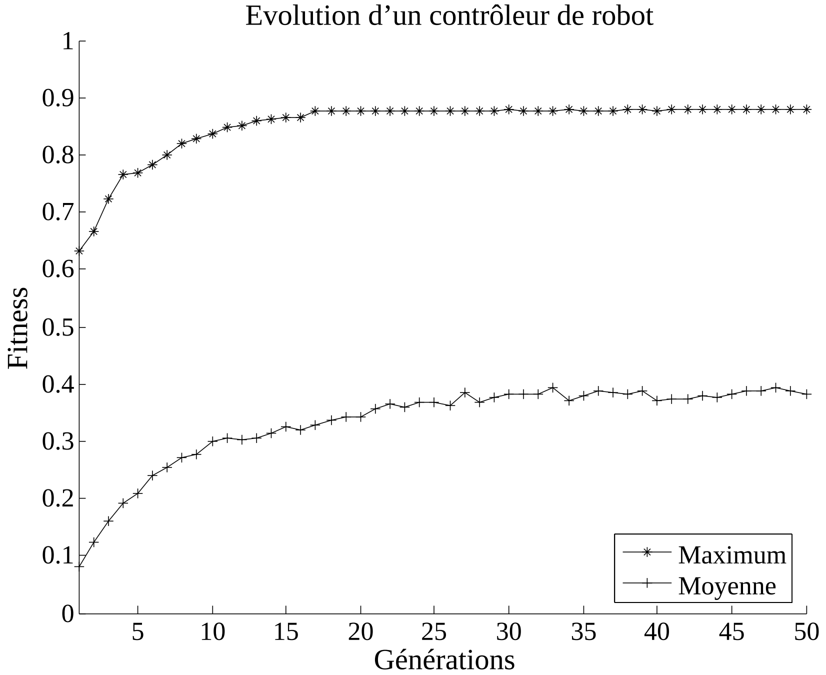


Figure 7.12 : *L'évolution du fitness moyen et maximum, pour un contrôleur de robot.*

implémentation offre l'avantage d'exclure tout court-circuit. En effet, les FGPAs actuels sont conçus sur des technologies pouvant laisser apparaître des courts-circuits si le bitstream de configuration n'est pas correct. Les outils tels que JBits [85] pour les circuits Virtex permettent de modifier les bits de configuration du FPGA à la main, mais mettent en garde l'utilisateur sur la possibilité de détruire le circuit.

La dernière FPGA conçue sans courts-circuits potentiels fut la série XC6200 de Xilinx. Depuis, aucun FPGA de ce type n'est apparu, alors que les applications de matériel évolutifs tels que celles développées par Thompson [235], ont besoin de cette qualité. Notre circuit est donc un candidat rêvé pour l'implémentation de systèmes mettant en oeuvre un évolution non-contrainte. De plus, à l'instar du XC6200, les bits de configuration sont connus, alors qu'ils ne sont pas rendus publics pour les autres FPGAs commerciaux. Il est alors trivial de ne choisir qu'une partie du bitstream à évoluer, que se soit le routage intermoléculaire, la fonctionnalité des molécules, ou d'autres. Alors que le prototype PO mettait à profit le routage distribué, dans un mécanisme d'évolution contrainte, nous ne l'exploitons aucunement dans cette section, qui vise à effectuer de l'évolution non contrainte.

Il est bien évident possible de concevoir un système faisant évoluer l'entièreté des bits de configuration d'une partie du sous-système organique. Toutefois, 76 bits définissant une molécule, un tableau de 10×10 molécules est décrit par rien moins que 7600 bits, ce qui peut poser des problèmes à un algorithme génétique qui se perdrait dans l'espace de recherche (cf. problème de scalabilité, page 51). Une solution, que nous allons développer ici, consiste en la réduction de l'espace de recherche, par la fixation de certains bits de configuration.

Nous allons expliciter la manière de fixer ces bits afin de réduire l'espace de recherche, tout en gardant une grande fonctionnalité [233]. L'élément de base à évoluer est alors tel que présenté à la figure 7.13, et contient une look-up table à trois entrées, trois multiplexeurs pour ces entrées, et un switchbox pour les sorties, chaque sortie pouvant être une des entrées des trois autres directions, ou le résultat de sortie de la molécule. Notons qu'un bit supplémentaire peut définir si la sortie est combinatoire ou séquentielle.

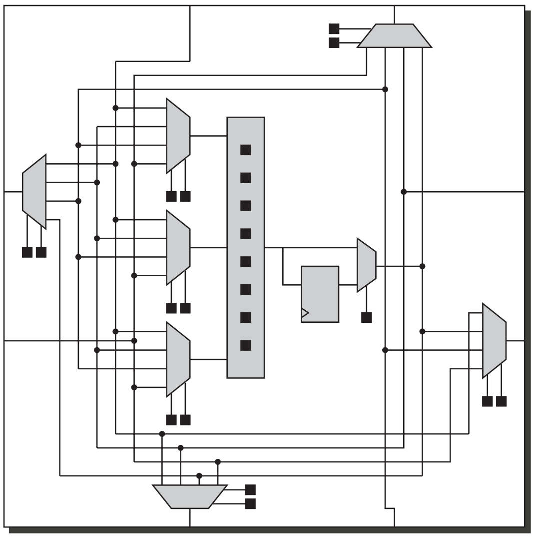


Figure 7.13 : *Le sous-ensemble de configuration d'une molécule, défini par 22 bits, ou 23 pour l'utilisation de la bascule.*

7.4.1 La look-up table

Durant la phase d'évolution, nous laissons la possibilité à la look-up table de la molécule d'évoluer. Nous pouvons choisir une 3-LUT ou une 4-LUT, la grande différence entre les deux approches étant la taille du génome. En effet, la version 3-LUT nécessite 8 bits pour la LUT, et 6 bits pour les entrées, soit 14 au total, alors que la version 4-LUT contient 16 bits pour la LUT et 8 pour les entrées, pour un total de 24. Dans l'optique de faciliter le travail de l'algorithme génétique, la version 3-LUT sera préférée, tout en n'excluant pas l'autre possibilité.

7.4.2 Le switchbox

Le switchbox d'une molécule, comme nous l'avons défini en page 187, est composé de huit multiplexeurs à huit entrées. Il y a donc deux lignes partant dans chaque direction. Nous pouvons, toujours afin de limiter l'espace de recherche, nous limiter à une seule ligne dans chaque direction. La moitié des multiplexeurs est alors inutilisée, et les quatre restants n'ont pas besoin de toutes leurs entrées, mais seulement de quatre d'entre elles. En effet, en n'utilisant que les lignes `ValInN(0)`, `ValInE(0)`, `ValInS(0)` et `ValInW(0)`, les entrées `ValIn`*`X`*`(1)` ne sont pas définies. Nous nous retrouvons en fin de compte avec un switchbox composé de quatre multiplexeurs à quatre entrées. La figure 7.14 montre le switchbox original, et les bits à fixer, ainsi que le switchbox résultant.

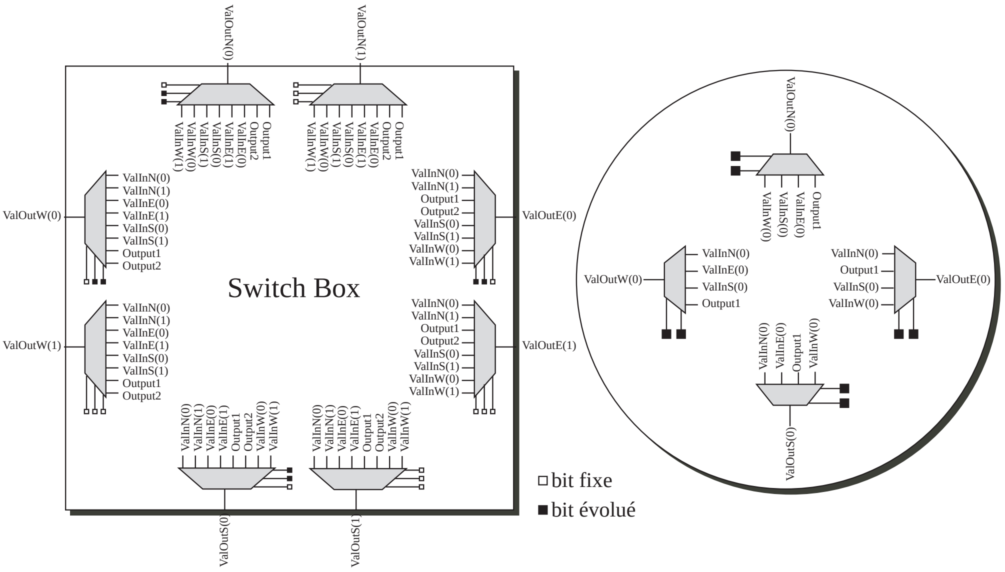


Figure 7.14 : *A gauche le switchbox d'une molécule, avec certains bits de configuration fixés. A droite, le switchbox résultant des fixations.*

7.4.3 Les entrées

Les entrées de la look-up table sont gérées par des multiplexeurs à 8 entrées, plus quelques autres. Etant donné que nous avons quatre entrées, venant chacune d'une des quatre voisines, les multiplexeurs peuvent être réduits en fixant certains bits de configuration. La figure 7.15 montre les multiplexeurs originaux, et les bits à fixer, ainsi que les multiplexeurs résultants pour les deux premières entrées.

7.4.4 Représentation du génome

Dans cette section nous présentons une manière de représenter le génome, afin d'optimiser le temps d'exécution de l'algorithme génétique.

Approche conventionnelle La première approche consiste simplement en une représentation compacte du génome. Les 22 bits sont stockés de manière optimale dans la mémoire du microprocesseur, et l'ensemble des bits de configuration d'une molécule, sur 76 bits, est reconstruit à partir de ces 22 bits. Cette méthode peut être plus efficace en terme de temps d'exécution si les opérateurs de crossover et mutation sont extrêmement rapides. Le mapping entre le génome et les bits de configuration réels est toutefois relativement complexe.

Nouvelle approche Nous proposons une nouvelle approche dans laquelle nous cherchons à optimiser la phase de mapping. En effet, passer des 22 bits d'un gène aux 76 décrivant la configuration d'une molécule est une tâche lourde en terme de temps de calcul. Nous proposons donc de travailler avec un génome composé de gènes de $3 \times 32 = 96$ bits. Sur ces 96 bits, seuls 22 représentent de l'information utile pour l'évolution, 54 sont des bits de configuration fixes, et les 20 restants ne sont tout simplement pas utilisés. L'idée est donc de voir un gène comme étant constitué d'une

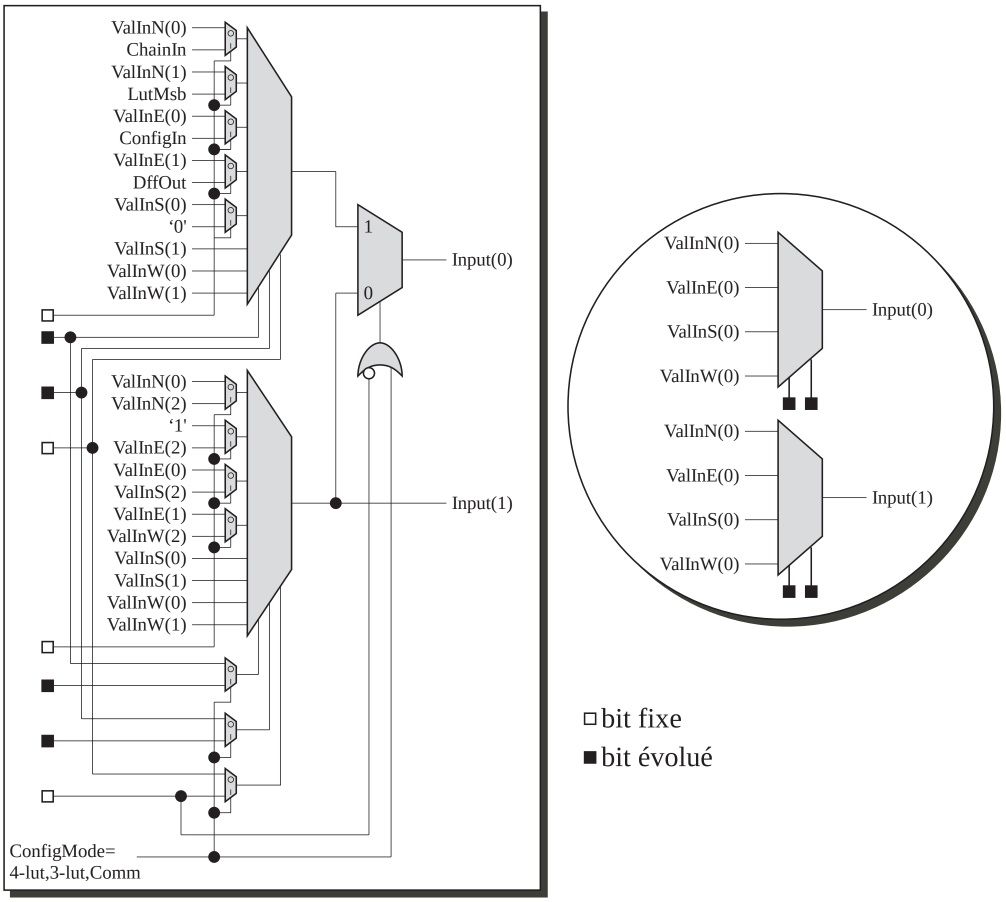


Figure 7.15 : *A gauche, les entrées d'une molécule, avec certains bits de configuration fixés. A droite, les entrées résultantes des fixations.*

partie utile, et d'une partie inutile, à la façon des organismes vivants, où une partie de l'ADN, appelée ADN poubelle, ne code aucune information (selon les recherches actuelles). Si nous reprenons le mapping mémoire des bits de configuration d'une molécule tel que défini à la page 192, nous découpons les 96 bits d'un gène de la façon décrite à la figure 7.16.

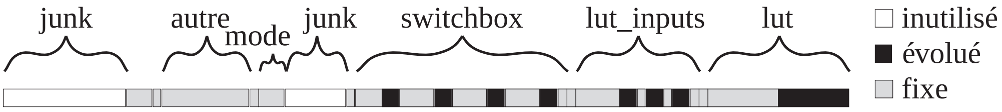


Figure 7.16 : *Un gène de 96 bits, décomposé en bits à évoluer, fixes, et inutilisés.*

Lors d'un crossover, le gène entier est considéré, de même que lors d'une mutation, et le mapping du gène aux bits de configuration peut se faire grâce à des opérations logiques simples. En effet, il suffit de garantir que les bits de configuration fixes le sont, et que les bits évolués sont bien exploités. Pour ce faire, il suffit de disposer de deux masques complémentaires, appliqués aux bits fixes et aux bits évolués, et d'opérer un OU logique entre les deux résultats. La figure 7.17 résume les opérations à effectuer sur un gène afin d'obtenir les bits de configuration de la molécule. Notons que l'application du masque aux bits fixes peut n'être effectuée qu'une seule fois, car son résultat est réutilisé pour toutes les molécules.

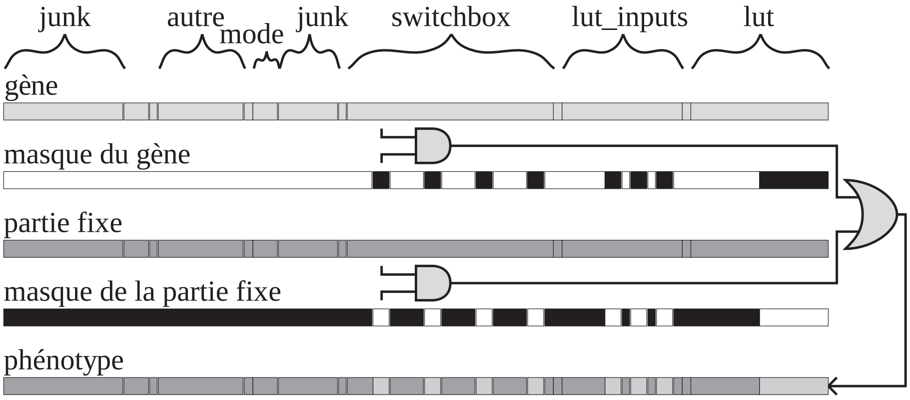


Figure 7.17 : *Les opérations logiques nécessaires au mapping d'un gène aux bits de configurations.*

Comparaison Lorsque le circuit POEtic sera opérationnel, il sera possible de comparer ces deux approches, afin de définir la plus efficace. Nous donnons ici une piste quant à leur évaluation. Le temps d'exécution de l'approche conventionnelle est dénotée par Tc, et celui de la nouvelle par Tn.

Posons :

Tx_{mut} est le temps d'exécution de l'opérateur de mutation, sur un génome, pour l'approche x.

Tx_{cross} est le temps d'exécution de l'opérateur de crossover pour deux individus, pour l'approche x.

Tx_{sel} est le temps d'exécution de la sélection de tous les individus d'une nouvelle génération, pour l'approche x.

Tx_{fit} est le temps d'exécution du calcul de fitness d'un individu, pour l'approche x.

Tx_{map} est le temps nécessaire à la création des bits de configuration de la molécule, à partir du génome, pour l'approche x.

Tx_{tot} est le temps d'exécution d'une phase de l'algorithme génétique, c'est-à-dire du calcul d'une nouvelle population.

Le temps d'exécution d'une phase de l'algorithme génétique correspond, pour une population de taille sp à ceci :

$$Tx_{tot} = sp * Tx_{mut} + sp/2 * Tx_{cross} + Tx_{sel} + sp * Tx_{map} + sp * Tx_{fit}$$

Or, nous avons que :

$$\begin{aligned} Tc_{sel} &= Tn_{sel} \\ Tc_{fit} &= Tn_{fit} \end{aligned}$$

La deuxième approche est plus efficace en terme de temps d'exécution si et seulement si $Tn_{tot} < Tc_{tot}$, ce qui, en regard des deux égalités, est équivalent à :

$$Tn_{mut} + Tn_{cross}/2 + Tn_{map} < Tc_{mut} + Tc_{cross}/2 + Tc_{map}$$

7.4.5 Caractéristiques de l'évolution

Lorsque le circuit POEtic sera opérationnel, nous serons en mesure de mettre en œuvre le mécanisme d'évolution non-contrainte que nous venons de présenter. Nous pouvons toutefois déjà caractériser le circuit POEtic, en fonction des paramètres des systèmes de matériel évolutif définis par Torresen dans [240, 241] :

- Le microprocesseur peut exécuter un *Algorithme génétique*.
- La technologie cible est *digitale*.
- L'architecture appliquée à l'évolution peut être *Complete Circuit Design*, où des blocs de base et le routage sont évolués, ou *Circuit Parameter Tuning*, ou seuls des paramètres configurables sont évolués.
- Les blocs de base peuvent être des portes logiques (les LUTs), où des fonctions (neurones, ...).
- L'évolution est exécutée *online*, car chaque individu peut être testé sur le matériel reconfigurable.
- L'évolution est *on-chip*, étant donné que le microprocesseur est incorporé au circuit reconfigurable.
- L'évolution peut y être *contrainte* (prototype PO), ou *non contrainte*, comme nous venons de le suggérer.
- Le domaine d'évolution peut être *statique* ou *dynamique*, dépendant du type d'application.

7.5 Autres exemples

Nous désirons conclure notre survol des mécanismes tirant parti des caractéristiques de POEtic par trois applications qui furent développées par les partenaires du projet. Nous n'entrerons pas dans les détails, laissant le lecteur les trouver dans les références indiquées.

7.5.1 Autoréparation

L'autoréparation est un des points de recherche important dans le cadre des systèmes bio-inspirés. La nature est plus qu'efficace pour résoudre des pannes, comme nos corps qui sont capables de cicatriser rapidement, et de faire face aux maladies. Les systèmes artificiels ne sont que rarement capables de gérer les pannes de manière efficace. Une des solutions matérielles, notamment utilisée en aéronautique, se base sur la duplication ou la *triplication* du système critique. Deux unités permettent alors de détecter une panne, et trois offrent la possibilité, via un vote, de définir quelle unité est défectueuse.

Les circuits reconfigurables ne sont pas intrinsèquement tolérants aux pannes, si ce n'est le tableau de cellules embryoniques qui furent développé au Laboratoire de Systèmes Logiques [142]. Le circuit POEtic n'offre pas de mécanisme matériel de gestion des pannes comme c'est le cas dans Embryonique, mais des travaux ont permis de montrer qu'il était possible de démontrer des mécanismes d'autoréparation en tirant parti de la reconfiguration partielle et du routage dynamique [20]. Deux approches ont été développées et implémentées grâce au simulateur du sous-système organique.

La première a vu la création d'un système de type embryonique, où une cellule contient un génome décrivant l'organisme entier. Un mécanisme de différentiation semblable à celui que nous avons décrit précédemment laisse la cellule sélectionner la partie du génome correspondant à son identifiant. Les cellules sont alors reliées entre elles en une chaîne, de la même manière que lors de la croissance basée sur un identifiant à une dimension. Deux parties fonctionnelles calculent les sorties de la cellule, et une comparaison est effectuée, afin de détecter une erreur. Lorsqu'une faute survient, la cellule concernée envoie un message à la cellule suivante, avec son identifiant. La cellule atteinte met alors à jour son identifiant et fait suivre le décalage. Toutes les cellules faisant partie de l'organisme, et étant positionnées après la cellule touchée, modifient donc leur comportement, et une nouvelle cellule totipotente est exploitée de manière à combler le manque de la cellule morte.

La deuxième réalisation se base sur la duplication d'une cellule, à la manière que nous avons déjà présentée. Aucun génome n'est présent dans la cellule, mais trois parties fonctionnelles permettent de détecter une faute et de sauvegarder l'état du système. Lorsqu'une faute survient, la cellule bloque le fonctionnement de la partie fonctionnelle de toutes les cellules de l'organisme, en utilisant l'*enable* moléculaire global. Elle se connecte ensuite à une nouvelle potentielle cellule, qui reconstruit trois nouvelles parties fonctionnelles en se basant sur les bits de configuration envoyés par la cellule endommagée. Par ce mécanisme, la fonctionnalité de l'organisme est garantie, puisque son état est sauvegardé en tout temps. La configuration initiale du circuit doit ici contenir l'ensemble des cellules utiles à l'implémentation d'un organisme complet, ainsi que des parties de cellules capables de prendre le relais lors de fautes. Ces parties de cellules sont également couplées à des molécules dont les bits de configuration forment une chaîne qui peut être reconfigurée par la cellule endommagée.

Ces deux approches ont exhibé une manière d'exploiter l'auto-reconfiguration partielle des molécules, ainsi que le routage distribué de POEtic. Il est bien clair cependant que ces deux systèmes partent du principe qu'aucune erreur ne peut survenir dans les unités de routage, ni dans les molécules responsables de la gestion de l'autoréparation, aucun mécanisme matériel n'ayant été mis en place dans POEtic pour parer à des erreurs éventuelles.

7.5.2 Synthèse vocale

Un système de synthèse vocale basée sur une grille d'éléments reliés à leurs quatre voisins a été développé par Cooper [44], à York. De l'évolution est appliquée à la forme de la grille, qui reçoit en entrée du bruit blanc, et est capable de générer un signal sonore approchant une courbe de fréquence présentée au système. L'implémentation est réalisée à l'aide de 16 molécules par élément, qui sont reliés entre eux via le routage dynamique.

Notons toutefois que l'architecture cellulaire ne tire pas parti des spécificités du sous-système organique, mais que cette réalisation exploite la présence du microprocesseur pour accélérer l'évolution et la configuration du circuit.

7.5.3 Neurone à impulsion

Finalement, sur l'axe épigénétique, un nouveau type de neurones à impulsion a été développé par les biologistes [69], puis implémenté grâce aux molécules du circuit POEtic [238]. 80 molécules sont nécessaires à la réalisation d'un seul neurone, qui est une implémentation sérielle de celui décrit par les biologistes. Cette taille non négligeable implique qu'un seul neurone peut être présent dans un circuit, qui possède 144 molécules. De ce fait, la possibilité de relier plusieurs circuits entre eux est une caractéristique essentielle de POEtic, qui offre ainsi un tableau reconfigurable de taille acceptable. Enfin, le réseau de neurones basé sur ces cellules exploite donc le routage dynamique pour la création des interconnexions cellulaires.

7.6 Conclusion

Après avoir présenté le circuit POEtic, nous avons, dans ce chapitre, tenté de mettre en avant des applications ou mécanismes tirant parti de ses caractéristiques le différenciant des FPGAs commerciaux. Cependant, toutes ces implémentations, hormis le prototype PO, ont été effectuées à l'aide du logiciel, développé par nos soins, qui permet de concevoir des systèmes pour POEtic, en simulant le comportement du sous-système organique. A l'écriture de ces lignes, le circuit final a été reçu, mais n'est pas encore opérationnel, ni même testé. Nous ne pouvons donc qu'espérer qu'il fonctionne correctement, et que toutes ces applications puissent être, dans un futur proche, chargées dans le circuit POEtic.

Concernant les caractéristiques de POEtic, nous pourrions suggérer quelques améliorations, dans le cas où une deuxième version devait voir le jour. Premièrement, il serait intéressant de disposer d'une auto-configuration complète des molécules, ce qui permettrait aux mécanismes de duplication de ne pas nécessiter que des molécules doivent être pré-configurées sur le plan des 8 bits de configuration fixes. Deuxièmement, à la manière d'Embryonique, un mécanisme d'autoréparation matériel pourrait être envisagé, afin d'aider à la réalisation de systèmes réellement tolérants aux pannes.

Chapitre 8

Conclusions

> Information is not knowledge,
> Knowledge is not wisdom,
> Wisdom is not truth,
> Truth is not beauty,
> Beauty is not love,
> Love is not music,
> Music is THE BEST...
>
> Frank ZAPPA , *Joe's Garage*

LA PRÉSENTE thèse a vu la réalisation physique d'un nouveau circuit reconfigurable, spécialement conçu pour l'implémentation d'applications bio-inspirées basées sur des systèmes cellulaires. Pour ce faire, outre la conception des éléments reconfigurables de base, nous avons défini un algorithme de routage distribué, qui offre des possibilités de création de chemins de données par le circuit lui-même. Nous allons commencer par résumer notre recherche sur le routage, et y proposer de futures explorations. Nous terminerons par l'analyse des caractéristiques du circuit POEtic, et suggérerons de potentielles améliorations, qui pourraient servir à de futurs concepteurs d'un circuit dédié aux mêmes types d'applications.

8.1 Le routage distribué

En première intention, notre algorithme de routage HIDRA fut développé spécifiquement pour le circuit POEtic. Il nous fallait un moyen de permettre aux cellules qui seraient implémentées sur ce substrat reconfigurable de modifier les connexions intercellulaires de manière autonome, ce qui n'est offert dans aucun FPGA commercial. Après nous être penchés sur un système de type anneau à jeton, nous avons opté pour la solution actuelle, qui offre plus de flexibilité.

Des unités de routage, reliées à leurs 4 voisines, ainsi qu'à un élément externe à la grille, forment une grille à deux dimensions. Elles sont composées d'un contrôleur et de cinq multiplexeurs : quatre qui sélectionnent les valeurs à envoyer dans chacune des direction, et un qui sélectionne la valeur à transmettre à l'élément externe. Le contrôleur gère les processus de routage, et est donc en charge de configurer correctement les multiplexeurs. L’algorithme est totalement distribué, et permet à des sources et destinations d’initier de nouvelles connexions à n’importe quel instant. Un identifiant leur permet alors de choisir précisément leur correspondant.

Nous avons décliné notre algorithme en trois autres variantes, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, offrant des optimisations en terme de temps d’exécution du routage ou en terme de risque de congestion. Ce dernier est un point critique de nos implémentations, car aucune technique de rip-up n’a été implémentée, par souci de minimisation de l’espace requis par une unité de routage. Nous avons mené une analyse de cette congestion, afin de pouvoir comparer nos quatre algorithmes, et affirmer qu’il serait préférable de sélectionner HIDRA-RC plutôt que HIDRA, dans le cas où un tel mécanisme devait à nouveau être exploité. HIDRA-RC possède la capacité d’exploiter de manière efficace les chemins de données déjà existants, lorsque plusieurs destinations doivent être reliées à une même source. En minimisant le nombre total de multiplexeurs réquisitionnés, il est alors possible de réduire le problème de congestion.

Nos quatre variantes ont également été implémentées pour des voisinages de 3, 6, et 8, alors que le circuit POEtic est construit sur la base d’un voisinage de 4. Le code VHDL synthétisable les décrivant a servi de base à des simulations permettant de valider les implémentations, puis d’évaluer les algorithmes. Nous nous sommes particulièrement penchés sur la congestion, et avons proposé un estimateur modélisant la probabilité de congestion en fonction du nombre de chemins à créer, pour une taille de tableau, un algorithme, un voisinage, et un nombre de destinations par source donnés. Cette fonction peut donc être exploitée pour prédire si un application particulière nécessitant tel nombre de chemins à créer est implémentable sur une grille de taille donnée.

Nous avons également utilisés ces résultats pour effectuer une analyse visant à comparer les algorithmes et les voisinages. En reprenant les courbes de congestion, et puisque ces courbes sont bijectives, nous pouvons, pour un risque de congestion donné, et pour une configuration donnée (taille, voisinage, algorithme, nombre de destinations par source), évaluer le nombre de destinations correspondant. La comparaison s’est faite sur ces bases, et notre principale conclusion est qu’un voisinage de 8 est environ quatre fois plus efficace qu’un voisinage de 4, en ce qui concerne la congestion, pour un même nombre de transistors utilisés. En effet, une unité de routage à 8 voisines ne nécessite que deux fois plus de logique qu’une à 4 voisines, mais offre 2 plus de possibilités de routage que des unités à 4 voisines groupées par 4.

Une amélioration potentielle, que nous proposerions d’explorer, est le voisinage de 6 couplé à un mécanisme de routage capable de créer un arbre de Steiner de poids minimum entre une source et plusieurs destinations. Comme nous l’avons mentionné en page 69, les membranes d’eau savonneuses s’arrangent toujours en formant des angles de 120 degrés entre trois parois, et créent ainsi une membrane de taille minimale. En reprenant ce principe, il semble prometteur d’exploiter le voisinage de 6, qui offre de tels angles, pour concevoir un nouvel algorithme où les chemins peuvent se modifier lorsque plusieurs destinations sont reliées à une même source, de façon à diminuer le nombre de multiplexeurs réquisitionnés, et par voie de fait, le risque de congestion.

Un des points faibles de nos quatre algorithmes est la présence de liaisons combinatoires traversant le circuit entier, et potentiellement plusieurs circuits, ce qui a pour désagréable conséquence de devoir réduire la fréquence d’horloge pour les applications mettant en jeu une grille de circuits. Nous avons donc développé un cinquième algorithme, HIDRA-L, où chaque unité de routage est une machine de Moore. Toutes les connexions sont absolument locales, et donc, aucune liaison longue distance n'est présente, offrant ainsi une bien meilleure scalabilité. Toutefois, leur réalisation nécessite une quantité de logique environ deux fois supérieure à celle d'une unité de routage d'un des quatre algorithmes précédents. Cette logique supplémentaire est la principale raison qui nous a fait choisir l'algorithme HIDRA, plutôt que HIDRA-L pour la réalisation de POEtic.

La scalabilité de HIDRA-L pourrait finalement être encore améliorée si nous introduisons des techniques asynchrones, ou GALS (Globalement Asynchrone Localement Synchrone), dans nos unités de routage. Dans HIDRA-L, une horloge globale doit être distribuée à toutes les unités de routage, pour les synchroniser. Dans un système multi-circuit, cette synchronisation n'est pas chose aisée, et l'asynchronisme pourrait apporter une stabilité supplémentaire au routage distribué. Le nombre de transistors nécessaires à une telle réalisation risque toutefois fort de dépasser celui de HIDRA-L, qui est déjà conséquent.

Enfin, l'arrivée des nanotechnologies promet de nouveaux défis pour les scientifiques. Nous sommes en droit d'espérer que, dans un futur plutôt proche, nous disposerons de grilles, à deux ou trois dimensions, d'éléments nanoscopiques qui devront être capables de s'auto-configurer. Gageons que le routage de signaux dans de telles structures ne se fera pas sans peine, et peut-être que nos solutions, qui ciblent actuellement les systèmes électroniques, trouveront des applications dans ce nouveau domaine de recherche.

8.2 Conclusions POEtiques

Le premier de nos algorithmes de routage a été spécifiquement conçu dans le but d'être partie intégrante du circuit POEtic. Ce circuit, qui contient un microprocesseur couplé à un tableau d'éléments reconfigurable, le sous-système organique, offre une nouvelle plateforme dédiée à l'implémentation d'applications bio-inspirées mettant en jeu des mécanismes phylogénétiques, ontogénétiques, et/ou épigénétiques. Une de nos contributions fut la conception du sous-système organique, sous forme de fichiers VHDL, écrits en respectant une description directement réalisable en matériel.

Le microprocesseur y offre la possibilité d'exécuter des algorithmes évolutionnistes, et de tester les individus sur le substrat reconfigurable qu'est le sous-système organique. Ce dernier, basé sur des éléments de premier abord semblables aux blocs de base des FPGAs commerciaux de fabricants comme Xilinx ou Altera, offre de nouvelles caractéristiques, qui permettent à nos organismes artificiels d'évoluer, de modifier leur comportement, en se reprogrammant et en changeant la topologie du réseau intercellulaire.

Les éléments clés de la plasticité du vivant sont la capacité des cellules à modifier leur comportement en fonction de leur environnement, ainsi que leur aptitude à créer de nouvelles connexions intercellulaires, notamment dans le cas des cellules neuronales. Notre sous-système organique a donc été conçu en gardant en mémoire ces deux caractéristiques fondamentales du vivant. Nos éléments de base, les molécules, peuvent reconfigurer d'autres molécules, et le routage distribué offre à ces molécules la possibilité de créer des connexions par elles-mêmes, sans qu'une intervention extérieure au sous-système organique ne soit nécessaire.

La reconfiguration partielle offre aux cellules implémentées sur le circuit POEtic la possibilité de modifier leur comportement en changeant les bits de configuration des molécules qui les composent. Cette caractéristique peut notamment être exploitée par les mécanismes de croissance, où une cellule se différencie en chargeant la configuration des molécules de sa partie fonctionnelle en fonction de sa position dans l'organisme. L'autoréparation peut également en tirer parti, en laissant une cellule partiellement endommagée être remplacée par une autre en pleine santé, qui peut être reconfigurée.

Le routage distribué permet aux cellules de créer de nouvelles connexions durant la *vie* de l'organisme. Lors de l'exécution de mécanismes ontogénétiques, une cellule peut se connecter à une cellule totipotente, et lui envoyer des informations, comme sa position, ou le génome décrivant les fonctionnalités de l'organisme. L'organisme peut ainsi croître de manière autonome, sans qu'une intervention extérieure ne soit nécessaire. Les réseaux de neurones peuvent également tirer profit du routage, en initiant de nouvelles connexions par lesquelles la structure du réseau change.

Deux autres caractéristiques du circuit POEtic sont exploitables par les applications de matériel évolutif : l'impossibilité de créer des courts-circuits, et la connaissance des bits de configuration des molécules. Depuis la série XC6200 de Xilinx, aucun circuit reconfigurable n'a offert ces deux caractéristiques, qui sont essentielles pour la réalisation d'évolution matérielle non contrainte. Couplées au microprocesseur, elles font de POEtic une nouvelle plateforme parfaitement adaptée à de tels applications.

Afin de rendre l'exploitation de POEtic possible, nous avons développé un logiciel de conception de designs pour le sous-systèmes organique. Une interface graphique laisse le développeur configurer les molécules, et les connecter entre elles par de simples manipulations. Un outil de simulation, que nous avons également réalisé, lui permet alors de vérifier le fonctionnement de son système, en visualisant l'état des molécules et des unités de routage.

Outre le circuit POEtic, nous avons élaboré différents mécanismes ontogénétiques, qui exploitent ses caractéristiques dans le but d'implémenter des systèmes multicellulaires. L'auto-réplication partielle d'une partie du circuit a été démontrée, de même que la manière d'y réaliser des systèmes tolérants aux pannes, où une cellule endommagée peut recopier une partie de sa fonctionnalité dans une cellule saine, afin que l'organisme entier puisse continuer à effectuer sa tâche. Nous avons également montré que POEtic pouvait être utilisé comme plateforme à du matériel évolutif au niveau des portes logiques.

Le tableau 8.1 compare deux FPGAs commerciaux avec POEtic par rapport à différentes caractéristiques importantes pour l'implémentation d'applications bio-inspirées.

Alors qu'un circuit de test contenant le microprocesseur, 12 molécules et 3 unités de routage a été réalisé et validé, le circuit final vient d'être reçu de la fonderie, mais n'a pas encore pu être testé. Dans un futur proche, nous pourrons vérifier que la réalisation des mécanismes que nous avons présentés fonctionne, et dans le cas où un nouveau circuit du même type devait un jour être conçu, nous pourrions suggérer les améliorations suivantes :

- Le routage intercellulaire pourrait être revu, comme suggéré à la section précédente, afin de réduire le risque de congestion. L'implémentation présente dans le circuit POEtic utilise un voisinage de 4, qui devrait être avantageusement remplacé par un voisinage de 8.

Caractéristique	**Xilinx XC6200**	**Xilinx Virtex II Pro**	**POEtic**
Impossibilité de court-circuit	Oui	Non	Oui
Processeur dans le circuit	Non	Oui	Oui
Processeur accédant les bits de configuration	Non	Non	Oui
Détails des bits de configuration accessibles	Oui	Non	Oui
Routage dynamique	Non	Non	Oui
Auto-configuration	Non	Non	Oui

Tableau 8.1 : *Comparaison des caractéristiques utiles aux systèmes bio-inspirés, entre un XC6200, un Virtex II Pro, et le circuit POEtic.*

- Concernant la scalabilité, et étant donné que la communication inter-circuits passe par le routage distribué, nous ne saurions que conseiller l'évaluation de méthodes asynchrones, qui éviteraient de baisser la fréquence de fonctionnement du système lorsque plusieurs circuits POEtic sont reliés entre eux.
- Au niveau moléculaire, un mécanisme permettant une auto-reconfiguration totale, c'est-à-dire où tous les bits de configuration peuvent être accédés par les molécules, pourrait être mis au point. Une cellule pourrait alors entièrement s'auto-répliquer, sans qu'une partie des molécules ne doive être préparée à recevoir une configuration.
- Il serait également intéressant d'ajouter un mode opératoire moléculaire autorisant l'accès au registre de la molécule à la façon d'une mémoire RAM. L'exploitation du génome par une cellule serait alors simplifié, et ne nécessiterait pas de parcourir l'entièreté du génome par un grand registre à décalage.
- Enfin, toutes ces belles suggestions se devraient de garder à l'esprit la taille des éléments, qu'il s'agisse des molécules ou des unités de routage. Notre circuit n'a pu accueillir que 144 molécules et 36 unités de routage. Il est clair qu'avec les dernières technologies de pointe, et pour un budget plus conséquent, il aurait été possible de disposer d'un tableau reconfigurable nettement plus imposant, mais il faut rester conscient que les améliorations susmentionnées sont autant de facteurs favorisant l'explosion du nombre de transistors nécessaires.

Finalement, l'avènement des nanotechnologies nécessitera, comme nous l'avons mentionné dans la section précédente, des mécanismes d'auto-configuration des systèmes matériels. Les grilles d'éléments nanoscopiques ne pourront pas forcément être totalement configurées de l'extérieur, et une forme d'ontogenèse sera nécessaire à leur bon fonctionnement. Le contenu de cette thèse pourra dès lors apporter une petite pierre à l'édifice des systèmes auto-configurables, et qui sait, peut-être nos enfants utiliseront-ils un jour des nanomachines construites sur la base de quelques éléments de notre travail...

Liste des figures

Liste des tableaux

Liste des algorithmes

Bibliographie

[1] ACTEL. « *Axcelerator Family FPGAs* », février 2004. Available from http ://www.actel.com.

[2] ACTEL. « *ProAsicPLUS Flash Family FPGAs* », avril 2004. Available from http ://www.actel.com.

[3] A. I. ADAMATZKY. « Computation of Shortest Path in Cellular Automata ». *Mathematical and Computer Modelling*, 23(4) :3415–3418, 1996.

[4] H. G. ADSHEAD. « Towards VLSI Complexity : The DA Algorithm Scaling Problem : Can Special DA Hardware Help ? ». Dans *Proceedings of the 19th Conference on Design Automation*, pages 339–344. IEEE Press, 1982.

[5] P. AGRAWAL et M. A. BREUER. « Some Theoretical Aspects of Algorithmic Routing ». Dans *Proc. 14th Design Automation Conference*, pages 23–31, Piscataway, NJ, USA, 1977. IEEE Press.

[6] S. AKERS. « A Modification of Lee's Path Connection Algorithm ». *IEEE Transactions on Electronic Computers*, EC-16(1) :97–98, février 1967.

[7] ALTERA. « *Cyclone Device Handbook* », 2003. Available from http ://www.altera.com.

[8] ALTERA. « *MAX 3000A Programmable Logic Device Family* », juin 2003. Available from http ://www.altera.com.

[9] ALTERA. « *Startix II Device Handbook* », 2004. Available from http ://www.altera.com.

[10] ALTERA. « *Stratix Device Handbook* ». Altera Corporation, avril 2004. Available from http ://www.altera.com.

[11] ALTERA. « *Stratix GX FPGA Family* ». Altera Corporation, février 2004. Available from http ://www.altera.com.

[12] APTIX, INC.. « *FPIC AX1024D. Preliminary Data Sheet* ». Aptix, Inc., San Jose, CA, 1992.

[13] ARM. « *AMBA Specification, Rev 2.0* ». Advanced RISC Machines Ltd (ARM), http ://www.arm.com/armtech/AMBA_Spec, 1999.

[14] J. C. ASTOR et C. ADAMI. « A Developmental Model for the Evolution of Artificial Neural Networks ». *Artificial Life*, 6(3) :189–218, 2000.

[15] ATMEL. « *AT6000(LV) Series* ». Atmel, octobre 1999. Available from http ://www.atmel.com.

[16] ATMEL. « *AT40K05AL Datasheet* ». Atmel, mai 2002. Available from http ://www.atmel.com.

[17] ATMEL. « *FPSLIC Datasheet* ». Atmel, novembre 2003. Available from http ://www.atmel.com.

[18] T. BACK, U. HAMMEL et H.-P. SCHWEFEL. « Evolutionary Computation : Comments on the History and Current State ». *IEEE Transactions on Evolutionary Computation*, 1(1) :3–17, avril 1997.

[19] R. BALZER. « An 8-state Minimal Time Solution to the Firing Squad Synchronization Problem ». *Information and Control*, 10 :22–42, 1967.

[20] W. BARKER, D. M. HALLIDAY, Y. THOMA, E. SANCHEZ, G. TEMPESTI, J.-M. MORENO et A. M TYRRELL. « Fault Tolerance using Dynamic Reconfiguration on the POEtic Tissue ». 2005. To be submitted.

[21] A. BARNA et D.I. PORAT. « *Integrated Circuits in Digital Electronics* », pages 413–420. John Wiley & Sons, New York, 1973.

[22] R. BELLMAN. « On a Routing Problem ». *Quarterly of Applied Mathematics*, 16 :87–90, 1958.

[23] V. BETZ et J. ROSE. « How Much Logic Should Go in an FPGA Logic Block ». *IEEE Design & Test of Computers*, 15(1) :10–15, janvier- mars 1998.

[24] R. BEZ, E. CAMERLENGHI, A. MODELLI et A. VISCONTI. « Introduction to Flash Memory ». *Proceedings of the IEEE. Flash Memory Technology*, 91(4) :489–502, avril 2003.

[25] T. BLANK. « A Survey of Hardware Accelerators used in Computer-Aided Design ». *IEEE Transactions on Design and Test*, 1(3) :21–39, août 1984.

[26] T. BLANK, M. STEFIK et W. VANCLEEMPUT. « A Parallel Bit Map Processor Architecture for DA Algorithms ». Dans *Proceedings of the 18th Conference on Design Automation*, pages 837–845. IEEE Press, 1981.

[27] S. BORNHOLDT et T. ROHLF. « Topological Evolution of Dynamical Networks : Global Criticality from Local Dynamics ». *Physical Review Letters*, 84(26) :6114–6117, juin 2000.

[28] O. BORŮVKA. « O jistém Problému Minimálním. Über ein Minimalproblem ». *Práce Moravské Přírodovědecké Spoločnosti*, 3 :37–58, janvier 1926.

[29] C.-V. BOYS. *Bulles de Savon : Quatre Conférences sur la Capillarité Faites Devant un Jeune Auditoire*. Gauthier-Villars, Paris, 1892.

[30] M. A. BREUER et K. SHAMSA. « A Hardware Router ». *Journal of Digital Systems*, 4(4) :393–408, 1981.

[31] S. R. BROADBENT et J. M. HAMMERSLEY. « Percolation Processes I. Crystals and mazes ». *Proceedings of the Cambridge Philosophical Society. Mathematical and Physical Science*, 57 :629–641, 1957.

[32] S. BROWN, R. FRANCIS, J. ROSE et Z. VRANESIC. *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[33] S. BROWN et J. ROSE. « FPGA and CPLD Architectures : a Tutorial ». *IEEE Design & Test of Computers*, 13(2) :42–57, 1996.

[34] S. Brown, J. Rose et Z.G. Vranesic. « A Detailed Router for Field-Programmable Gate Arrays ». *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(5) :620–628, 1992. TY - JOUR.

[35] E. Buffetaut. *Cuvier, le Découvreur de Mondes Disparus*. Belin - Pour la Science, Paris, 2002.

[36] V.L. Burton. *The Programmable Logic Device Handbook*. TAB Professional and Reference Books, USA, 1990.

[37] W. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo et S. L. Sze. « A User Programmable Reconfigurable Gate Array ». Dans *IEEE 1986 Proc. Custom Integrated Circuits Conference*. IEEE, 1986.

[38] P. K. Chan et M. D. F. Schlag. « Acceleration of an FPGA Router ». Dans *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, page 175. IEEE Computer Society, 1997.

[39] Y.-W. Chang, K. Zhu, G.-M. Wu, D. F. Wong et C. K. Wong. « Analysis of FPGA/FPIC Switch Modules ». *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(1) :11–37, janvier 2003.

[40] G.-X. Cheng, M. Tanaka et M. Yamada. « A Parallel Routing Technique Based on Local Current Comparison ». Dans *IEEE International Symposium on Circuits and Systems*, volume 5, pages 3114–3117. IEEE Press, 1991.

[41] B. V. Cherkassky, A. V. Goldberg et T. Radzik. « Shortest Paths Algorithms : Theory and Experimental Evaluation ». *Mathematical Programming*, 73 :129–174, juin 1996.

[42] W. Choi et G. Sobelman. « Hardware Rip-up Router with Concurrent Wavefront Propagation ». *Electronics Letter*, pages 373–374, mars 1989.

[43] A. Colorni, M. Dorigo et V. Maniezzo. « Distributed optimization by ant colonies ». Dans F. Varela et P. Bourgine, éditeurs, *Proc. First European Conference on Artificial Life*, pages 131–142. Elsevier Publishing, 1991.

[44] C. H. V. Cooper, D. M. Howard et A. M. Tyrrell. « Using GAs to Create a WaveGuide Model of the Oral Vocal Tract ». Dans G. R. Raidl et al., éditeur, *Proc. of EvoWorkshops 2004*, volume 3005 de *LNCS*, pages 280–288, Berlin Heidelberg, 2004. Springer-Verlag.

[45] Lattice Semiconductor Corporation. « *ORCA Series 4 FPGAs* », novembre 2003. Available from http ://www.latticesemi.com.

[46] Lattice Semiconductor Corporation. « *ispXPGA Family* », juillet 2004. Available from http ://www.latticesemi.com.

[47] Lattice Semiconductor Corporation. « *LatticeECP/EC Family Data Sheet Introduction* », juin 2004. Available from http ://www.latticesemi.com.

[48] W.J. Dally et A. Chang. « The Role of Custom Design in ASIC Chips ». Dans *Proc. 37th conference on Design Automation*, pages 643–647, New York, NY, USA, 2000. ACM Press.

[49] E. Damm, H. Gethöffer, K. Kaiser et E. Damm GmbH. « Hardware Support for Automatic Routing ». Dans *Proceedings of the 19th conference on Design automation*, pages 219–223. IEEE Press, 1982.

[50] G. B. DANTZIG. Maximization of a Linear Function of Variables Subject to Linear Inequalities. Dans T.C. KOOPMANS, éditeur, *Activity Analysis of Production and Allocation - Proceedings of a Conference*, volume 13 de *Cowles Commission Monograph*, pages 339–347. Wiley, New York, 1951.

[51] G. B. DANTZIG. « Discrete-Variable Extremum Problems ». *Operations Research*, 5(2) :266–277, avril 1957.

[52] C. DARWIN. *On the Origin of Species by Means of Natural Selection*. John Murray, London, 1859.

[53] C. DARWIN. *Variation of Animals and Plants under Domestication*. Appleton and Co, New York, 1868.

[54] C. DARWIN, A. WALLACE, C. LYELL et J. D. HOOKER. « On the Tendency of Species to form Varieties ; and on the Perpetuation of Varieties and Species by Natural Means of Selection ». *Journal of the Proceedings of the Linnean Society, Zoology*, 3 :45–62, août 1858.

[55] H. de GARIS. « Growing an Artificial Brain with a Million Neural Net Modules Inside a Trillion Cell Cellular Automaton Machine ». Dans *Proc. of the Fourth International Symposium on Micro Machine and Computer Science*, pages 211–214, 1993.

[56] H. de GARIS, A. BULLER, L. de PENNING, T. CHODAKOWSKI et D. DECESARE. « Initial Evolution Results on CAM-Brain Machines (CBMs) ». Dans G. DORFFNER, H. BISHOF et K. HORNIK, éditeurs, *ICANN 2001*, volume 2130 de *Lecture Notes in Computer Science*, pages 814–819, Berlin Heidelberg, 2001. Springer-Verlag.

[57] H. de GARIS, L. KANG, Q. HE, Z. PAN, M. OOTANI et E. RONALD. « Million Module Neural Systems Evolution : The Next Step in ATR's Billion Neuron Artificial Brain ». Dans *Proceedings of Evolution Artificielle 97 (EA'97)*, volume 10, pages 231–243, 1997.

[58] A. DEHON, R. HUANG et J. WAWRZYNEK. « Hardware-Assisted Fast Routing ». Dans *Proceedings of the 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, page 205. IEEE Computer Society, 2002.

[59] E. W. DIJKSTRA. « A Note on Two Problems in Connexion with Graphs. ». *Numerische Mathematik*, 1 :269–271, 1959.

[60] T. DOBZHANSKI. *L'Homme en Evolution*. Flammarion, Paris, 1966.

[61] D. DUBOULE et P. SORDINO. « L'Origine des Doigts ». *La Recherche*, 296 :66–69, mars 1997.

[62] D. DUBOULE et A.S. WILKINS. « The Evolution of 'Bricolage' ». *Trends in Genetics*, 14(2) :54–59, février 1998.

[63] S. EBERHARDT, T. DUONG et A. THAKOOR. « Design of Parallel Hardware Neural Network Systems from Custom Analog VLSI 'Building Block' Chips ». Dans *International Joint Conference on Neural Networks (IJCNN)*, volume 2, pages 183–190, 1989.

[64] K. ECHTLE et I. EUSGELD. « A Genetic Algorithm for Fault-Tolerant System Design ». Dans R. de Lemos et AL., éditeur, *First Latin American Symposium*

on Dependable Computing, volume 2847 de *LNCS*, pages 197–213, Berlin Heidelberg, 2003. Springer-Verlag.

[65] J.G. ELDREDGE et B.K. HUTCHINGS. « Density Enhancement of a Neural Network using FPGAs and Run-time Reconfiguration ». Dans *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, 1994.

[66] J.G. ELDREDGE et B.L. HUTCHINGS. « RRANN : a Hardware Implementation of the Backpropagation Algorithm using Reconfigurable FPGAs ». Dans *IEEE International Conference on Neural Networks*, volume 4, pages 2097–2102, 1994.

[67] N. ELDREDGE et S. J. GOULD. « *Models in Paleobiology : Punctuated Equilibria : An Alternative to Phyletic Gradualism* », Chapitre 5, pages 82–115. Freeman, Cooper and Co, 1972.

[68] F. ERCAL et H. C. LEE. « Time-Efficient Maze Routing Algorithms on Reconfigurable Mesh Architectures ». *Journal of Parallel and Distributed Computing*, 44(2) :133–140, août 1997.

[69] J. ERIKSSON, O. TORRES, A. MITCHELL, G. TUCKER, K. LINDSAY, D. HALLIDAY, J. ROSENBERG, J.-M. MORENO et A. E. P. VILLA. « Spiking Neural Networks for Reconfigurable POEtic Tissue ». Dans A.M. TYRRELL, P.C. HADDOW et J. TORRESEN, éditeurs, *Evolvable Systems : From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES '03)*, volume 2606 de *LNCS*, pages 165–173, Berlin, 2003. Springer-Verlag.

[70] D. FLOREANO et C. MATTIUSSI. Evolution of Spiking Neural Controllers for Autonomous Vision-Based Robots. Dans T. GOMI, éditeur, *Evolutionary Robotics IV*, pages 38–61. Springer-Verlag, Berlin Heidelberg, 2001.

[71] D. FLOREANO et J. URZELAI. Evolution and Learning in Autonomous Robotics Agents. Dans D. MANGE et M. TOMASSINI, éditeurs, *Bio-Inspired Computing Machines*, Chapitre 12, pages 317–346. PPUR, Lausanne, 1998.

[72] L. R. FORD JR et D. R. FULKERSON. « Maximal Flow Through a Network ». *Canadian Journal of Mathematics*, 8(3) :399–404, juin 1956.

[73] F.H. GAGE. « Mammalian Neural Stem Cells ». *Science*, 287(5457) :1433–1438, février 2000.

[74] W. J. GEHRING, M. AFFOLTER et T. BÜRGLIN. « Homeodomain Proteins ». *Annual Reviews*, 63 :487–526, 1994.

[75] W. GERSTNER et W.M. KISTLER. *Spiking Neuron Models : Single Neurons, Populations, Plasticity*. Cambridge University Press, Cambridge, 2002.

[76] A. GLASSNER. « Soap Bubbles. 1 ». *IEEE Computer Graphics and Applications*, 20(5) :76–84, septembre 2000.

[77] A. GLASSNER. « Soap Bubbles. 2 ». *IEEE Computer Graphics and Applications*, 20(6) :99–109, novembre 2000.

[78] R. B. GOLDSCHMIDT. *The Material Basis of Evolution*. Yale University Press, New Haven, CT, USA, 1940.

[79] C.S. GOODMAN et B.C. COUGHLIN. « The Evolution of Evo-Devo Biology ». *Proceedings of the National Academy of Sciences of the USA*, 97(9) :4424–4425, avril 2000.

[80] T.G.W. GORDON et P.J. BENTLEY. On Evolvable Hardware. Dans S. OVASKA et L. SYTANDERA, éditeurs, *Soft Computing in Industrial Electronics*, pages 279–323. Physica-Verlag, Heidelberg, Germany, 2002.

[81] S. J. GOULD. *La Vie est Belle. Les Surprises de l'Évolution*. Seuil, collection Points Sciences, 1991.

[82] John J. GREFENSTETTE. « Lamarckian Learning in Multi-agent Environments ». Dans Rick BELEW et Lashon BOOKER, éditeurs, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 303–310, San Mateo, CA, 1991. Morgan Kaufman.

[83] J.B. GRIMBLEBY. « Automatic Analogue Circuit Synthesis Using Genetic Algorithms ». *IEE Proceedings Circuits, Devices and Systems*, 147(6) :319–323, décembre 2000.

[84] F. GRUAU. Genetic Systems of Boolean Networks with a Cell Rewriting Developmental Process. Dans D. WHITLEY et S.D. SCHAFFER, éditeurs, *Combination of Genetic Algorithms and Neural Networks*, pages 55–74. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[85] S. A. GUCCIONE, D. LEVI et P. SUNDARARAJAN. « Jbits : A Java-Based Interface for Reconfigurable Computing ». Dans *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[86] R. GUO, H. NGUYEN, A. SRINAVASAN, H. VERHEYEN, H. CAI, S. LAW et A. MOHSEN. « A 1024 Pin Universal Interconnect Array With Routing Architecture ». Dans *Custom Integrated Circuits Conference 1992*, Proceedings of the IEEE, pages 4.5.1–4.5.4, mai 1992.

[87] P.C. HADDOW, G. TUFTE et P. van REMORTEL. « Shrinking the Genotype : L-systems for EHW ? ». Dans *ICES '01 : Proceedings of the 4th International Conference on Evolvable Systems : From Biology to Hardware*, pages 128–139. Springer-Verlag, 2001.

[88] D.R. HANSON et C.W. FRASER. *A Retargetable C Compiler : Design and Implementation*. Benjamin-Cummings Publishing Company, 1995.

[89] P. HART, N. NILSSON et B. RAPHAEL. « A Formal Basis for the Heuristic Determination of Minimum Cost Paths ». *IEEE Transactions on System Sciences and Cybernetics*, SSC-4(2) :100–107, 1968.

[90] B. HAYES. « Experimental Lamarckism ». *American Scientist*, 87(6) :494–498, Nov-Dec 1999.

[91] D.O. HEBB. *The Organization of Behavior*. Wiley, New York, USA, 1949.

[92] W. HEYNS, W. SANSEN et H. BEKE. « A Line-Expansion Algorithm for the General Routing Problem with a Guaranteed Solution ». Dans *Proc. seventeenth design automation conference on Design automation*, pages 243–249, New York, NY, USA, 1980. ACM Press.

[93] D. W. HIGHTOWER. « The Interconnection Problem - A Tutorial ». Dans *DAC '73 : Proceedings of the 10th workshop on Design automation*, pages 1–21. IEEE Press, 1973.

[94] D.W. HIGHTOWER. « A Solution to Line-Routing Problems on the Continuous Plane ». Dans *Proc. 6th annual conference on Design Automation*, pages 1–24, New York, NY, USA, 1969. ACM Press.

[95] T. Higuchi, H. Iba et B. Manderick. Evolvable Hardware. Dans H. Kitano et J.A. Hendler, éditeurs, *Massively parallel artificial intelligence*, pages 398–421. MIT Press, 1994.

[96] D. Hill et N.-S. Woo. « The Benefits of Flexibility in Look-up Table FPGAs ». Dans W. Moore et W. Luk, éditeurs, *Proc. Oxford 1991 International Workshop on Field Programmable Logic and Applications*, pages 127–136, Abingdon, England, 1991. Abingdon EE&CS Books.

[97] D. D. Hill. « A CAD System for the Design of Field Programmable Gate Arrays ». Dans *DAC '91 : Proceedings of the 28th conference on ACM/IEEE design automation*, pages 187–192. ACM Press, 1991.

[98] C. Hochberger et R. Hoffmann. « Solving Routing Problems with Cellular Automata ». Dans S. Bandini et G. Mauri, éditeurs, *Proc. 2nd Conference on Cellular Automata for Research and Industry, ACRI '96*, pages 89–98. Springer, 1996.

[99] D.A. Hodges et H.G. Jackson. « *Analysis and Design of Digital Integrated Circuits* », pages 401–406. MacGraw-Hill, New York, 1983.

[100] J. H. Hoel. « Some Variations of Lee's Algorithm ». *IEEE Transactions on Computers*, C-25(1) :19–24, janvier 1976.

[101] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[102] P.W.H. Holland. « The Future of Evolutionary Developmental Biology ». *Nature*, 402 :41–44, décembre 1999.

[103] G. Hollingworth, S. Smith et A. Tyrell. « Safe Intrinsic Evolution of Virtex Devices ». Dans *proceedings of 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 195–204, 2000.

[104] Hominidés.com. « http ://www.hominides.com/html/theories/theories-evolutionnisme-lamarck.html ».

[105] S. J. Hong, R. Nair et E. Shapiro. « A Physical Design Machine ». Dans J.P. Gray, éditeur, *proceedings of the first International Conference on Very Large Scale Integration (VLSI 81)*, pages 257–266, London, 1981. Academic Press.

[106] J.J. Hopfield. « Neural Networks and Physical Systems with Emergent Collective Computational Abilities ». *Proceedings of the National Academy of Sciences*, 79(8) :2554–2558, avril 1982.

[107] R. Huang, J. Wawrzynek et A. DeHon. « Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes ». Dans *FPGA '03 : Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 78–87. ACM Press, 2003.

[108] A. Iosupovicz. « Design of an Iterative Array Maze Router ». Dans *Procs. IEEE International Conference on Circuits and Computers (ICCC)*, pages 908–911. IEEE, octobre 1980.

[109] C. Isenberg. « Soap Films and Bubbles ». *Physics Education*, 16(4) :218–222, juillet 1981.

[110] F. Jacob. « Evolution and Tinkering ». *Science*, 196(4295) :1161–1166, jun 1977.

[111] K-TEAM S.A.. « *Khepera User Manual* ». Préverenges, Switzerland (http ://www.k-team.com).

[112] G. KAATI, L. O. BYGREN et S. EDVINSSON. « Cardiovascular and Diabetes Mortality Determined by Nutrition During Parents' and Grandparents' Slow Growth Period ». *European Journal of Human Genetics*, 10(11) :682–688, novembre 2002.

[113] L. Pack KAELBLING, M. L. LITTMAN et A. P. MOORE. « Reinforcement Learning : A Survey ». *Journal of Artificial Intelligence Research*, 4 :237–285, 1996.

[114] I. KAJITANI, T. HOSHINO, M. IWATA et T. HIGUCHI. « Variable Length Chromosome GA for Evolvable Hardware ». Dans *Proc. IEEE International Conference on Evolutionary Computation*, pages 443–447. IEEE, 1996.

[115] I. KAJITANI, T. HOSHINO, D. NISHIKAWA, H. YOKOI, S. NAKAYA, T. YAMAUCHI, T. INUO, N. KAJIHARA, M. IWATA, D. KEYMEULEN et T. HIGUCHI. « A Gate-Level EHW Chip : Implementing GA Operations and Reconfigurable Hardware on a Single LSI ». Dans M. SIPPER, D. MANGE et A. PÉREZ-URIBE, éditeurs, *ICES'98*, volume 1478 de *Lecture Notes in Computer Science*, pages 1–12, Berlin Heidelberg, 1998. Springer-Verlag.

[116] I. KAJITANI, M. MURAKAWA, D. NISHIKAWA, H. YOKOI, N. KAJIHARA, M. IWATA, D. KEYMEULEN, H. SAKANASHI et T. HIGUCHI. « An Evolvable Hardware Chip for Prosthetic Hand Controller ». Dans *MICRONEURO '99 : Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems*, page 179. IEEE Computer Society, 1999.

[117] T. KALGANOVA. « Bidirectional Incremental Evolution in Extrinsic Evolvable Hardware ». Dans *Proc. of The Second NASA/DoD Workshop on Evolvable Hardware (EH'2000)*, Palo Alto, California, USA, 2000. IEEE Computer Society.

[118] T. KALGANOVA, J. F. MILLER et T. C. FOGARTY. « Some Aspects of an Evolvable Hardware Approach for Multiple-Valued Combinational Circuit Design ». Dans M. SIPPER, D. MANGE et A. PÉREZ-URIBE, éditeurs, *ICES'98*, volume 1478 de *Lecture Notes in Computer Science*, pages 78–89, Berlin Heidelberg, 1998. Springer-Verlag.

[119] J. KENNEDY et R. EBERHARDT. « Particle Swarm Optimization ». Dans *Proc. IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[120] H. KESHK, S.-I. MORI, H. NAKASHIMA et S. TOMITA. « Amon : a Parallel Slice Algorithm for Wire Routing ». Dans *Proceedings of the 9th international conference on Supercomputing*, pages 200–208. ACM Press, 1995.

[121] M. KIMURA. *Théorie Neutraliste de L'Évolution*. Flammarion, 1983.

[122] H. KITANO. « Designing Neural Networks Using Genetic Algorithms with Graph Generation System ». *Complex Systems*, 4(4) :461–476, 1990.

[123] H. KITANO. « Building Complex Systems Using Developmental Process : An Engineering Approach ». Dans M. SIPPER, D. MANGE et A. PÉREZ-URIBE, éditeurs, *ICES '98 : Proceedings of the Second International Conference on Evolvable Systems*, volume 1478 de *LNCS*, pages 218–229. Springer-Verlag, 1998.

[124] T. Kohonen. « The Self-organizing Maps ». *Proceedings of the IEEE*, 78(9) :1464–1480, septembre 1990.

[125] T. Kondo, T. Nakashima, M. Aoki et T. Sudo. « An LSI Adaptive Array Processor ». *IEEE Journal of Solid-State Circuits*, 18(2) :147–156, avril 1983.

[126] T. Kondo, T. Nakashima, T. Tsuchiya, Y. Sugiyama et T. Sudo. « A Large Scale Cellular Array Processor : AAP-1 ». Dans *Proceedings of the 1985 ACM thirteenth annual conference on Computer Science*, pages 100–111, New York, NY, USA, 1985. ACM Press.

[127] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Shiozawa, M. Inamori et K. Nagami. « PCA-1 : a Fully Asynchronous, Self-Reconfigurable LSI ». Dans *ASYNC '01 : Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, pages 54–61. IEEE Computer Society, 2001.

[128] R. K. Korn. « An Efficient Variable-Cost Maze Router ». Dans *Proceedings of the 19th conference on Design automation*, pages 425–431, Piscataway, NJ, USA, 1982. IEEE Press.

[129] J. R. Koza. *Genetic Programming : On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[130] J. B. Kruskal. « On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem ». *Proceedings of the American Mathematical Society*, 7(1) :48–50, février 1956.

[131] H. Kumar, M. A. Bayoumi, A. Tyagi, N. Ling et R. Kalyan. « Parallel Implementation of a Cut and Paste Maze Routing Algorithm ». Dans *Proc. 1993 IEEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2035–2038, 1993.

[132] P. L. et I. B.. « Et si l'Homme était une Invention de... l'Homme ». *Science et Vie*, 1026 :50–57, mars 2003.

[133] P.K. Lala. « *Digital System Design Using Programmable Logic Devices* », Chapitre 5, pages 114–166. Computer Engineering. Prentice Hall, New Jersey, USA, 1990.

[134] J. B. Lamarck. *Philosophie Zoologique*. Chez Dentu, Paris, 1809.

[135] D. Lambert et R. Rezsohazy. *Comment les Pattes Viennent au Serpent. Essai sur l'étonnante Plasticité du Vivant*. Nouvelle Bibliothèque Scientifique. Flammarion, Paris, 2004.

[136] C. G. Langton. « Self-Reproduction in Cellular Automata ». *Physica D*, 10 :135–144, 1984.

[137] C. Y. Lee. « An Algorithm for Path Connections and Its Applications ». *IRE Transactions on Electronic Computers*, EC-10(3) :346–365, septembre 1961.

[138] J. Lee, Z. Won, S. Sahni et E. Shragowitz. « Parallel Algorithms for Physical Design ». Dans *IEEE International Symposium on Circuits and Systems*, volume 1, pages 325–328. IEEE Press, 1988.

[139] D. Levi et S.A. Guccione. « GeneticFPGA : Evolving Stable Circuits on Mainstream FPGA Devices ». Dans *EH '99 : Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware*, page 12. IEEE Computer Society, 1999.

[140] D.S. LINDEN. « Optimizing Signal Strength In-Situ Using an Evolvable Antenna System ». Dans A. STOICA, J. LOHN, R. KATZ, D. KEYMEULEN et R.S. ZEBULUM, éditeurs, *The 2002 NASA/DoD Conference on Evolvable Hardware*, pages 147–151, Alexandria, Virginia, juillet 2002. IEEE Computer Society.

[141] A. LINDENMAYER. « Mathematical Models for Cellular Interaction in Development, parts I and II ». *Journal of Theoretical Biology*, 18 :280–315, 1968.

[142] D. MANGE, M. SIPPER, A. STAUFFER et G. TEMPESTI. « Towards Robust Integrated Circuits : The Embryonics Approach ». *Proceedings of the IEEE*, 88(4) :516–541, avril 2000.

[143] D. MANGE, A. STAUFFER, E. PETRAGLIO et G. TEMPESTI. « Artificial Cell Division ». *Biosystems*, 76(1-3) :157–167, août-octobre 2004.

[144] D. MANGE et M. TOMASSINI, éditeurs. *Bio-inspired Computing Machines : Towards Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[145] C. MANOVIT, C. APORNTEWAN et P. CHONGSTITVATANA. « Synthesis of Synchronous Sequential Logic Circuits from Partial Input/Output Sequences ». Dans *ICES ’98 : Proceedings of the Second International Conference on Evolvable Systems*, volume 1478 de *LNCS*, pages 98–105. Springer-Verlag, 1998.

[146] C. MAXFIELD. *The Design Warrior’s Guide to FPGAs*. Elsevier, 2004.

[147] E. MAYR. *Populations, Espèces et Evolution*. Hermann, Paris, 1974.

[148] E. MAYR. « Speciation and Macroevolution ». *Evolution*, 36(6) :1119–1132, novembre 1982.

[149] J. MAZOYER. « A Six-state Minimal Time Solution to the Firing Squad Synchronization Problem ». *Theoretical Computer Science*, 50 :183–238, 1987.

[150] W.S. MCCULLOCH et W. PITTS. « A Logical Calculus of the Ideas Immanent in Nervous Activity ». *Bulletin of Mathematical Biophysics*, 5 :115–133, 1943.

[151] L. MCMURCHIE et C. EBLING. « PathFinder : A Negotiation-Based Performance-Driven Router for FPGAs ». Dans *Proc. of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM, février 1995.

[152] G. MENDEL. « Versuch über Plfanzenhybriden ». *Verhandlungen des naturforschenden Vereines in Brünn, Bd. IV für das Jahr 1865*, pages 3–47, 1866.

[153] B. MESOT, E. SANCHEZ, C.-A. PENA et A. PEREZ-URIBE. « SOS++ : Finding Smart Behaviors Using Learning and Evolution ». Dans R.K. Standish M.A. BEDAU et H.A. ABBASS, éditeurs, *Proceedings of the Eighth International Conference on Artificial Life*, pages 264–273, Cambridge, Mass., 2003. Bradford Book, The MIT Press.

[154] D. MESQUITA, F. MORAES, J. PALMA, L. MÖLLER et N. CALAZANS. « Remote and Partial Reconfiguration of FPGAs : Tools and Trends ». Dans *Proc. International Parallel and Distributed Processing Symposium (IPDPS’03)*, pages 177–185. IEEE, 2003.

[155] E. MICHEL. « Les Neurones et la Créativité ». *Lieux d’être*, 36, Automne 2003.

[156] K. MIKAMI et K. TABUCHI. « A Computer Program for Optimal Routing of Printed Circuit Conductors ». Dans *IFIP Congress*, volume 2, pages 1475–1478, 1968.

[157] J. F. MILLER et P. THOMSON. « Aspects of Digital Evolution : Evolvability and Architecture ». Dans A. E. Eiben et AL., éditeur, *PPSN V*, volume 1498 de *Lecture Notes in Computer Science*, pages 927–936, Berlin Heidelberg, 1998. Springer-Verlag.

[158] J. F. MILLER et P. THOMSON. « Aspects of Digital Evolution : Geometry and Learning ». Dans M. Sipper et AL., éditeur, *ICES'98*, volume 1478 de *Lecture Notes in Computer Science*, pages 25–35, Berlin Heidelberg, 1998. Springer-Verlag.

[159] J.F. MILLER. « Evolving a Self-Repairing, Self-Regulating, French Flag Organism ». Dans K. DEB, R. POLI, W. BANZHAF, H. BEYER, E. K. BURKE, P. J. DARWEN, D. DASGUPTA, D. FLOREANO, J. A. FOSTER, M. HARMAN, O. HOLLAND, P. Luca LANZI, L. SPECTOR, A. TETTAMANZI, D. THIERENS et A. M. TYRRELL, éditeurs, *Genetic and Evolutionary Computation - GECCO 2004*, volume 1 de *Lecture Notes in Computer Science*, pages 129–139, Berlin, Heidelberg, 2004. Springer.

[160] M. MINSKY. « *Finite and Infinite Machines* », pages 28–29 and 282–283. Prentice Hall, 1967.

[161] G. J. MINTY. « A Comment on the Shortest-Route Problem ». *Operations Research*, 5(5) :724, octobre 1957.

[162] A. MITCHELL. « A Survey of Existing Digital Artificial Neural Network. Implementations, Concerning Their Suitability for Integration into the POEtic Tissue ». Rapport Technique, University of York, 2002.

[163] F. MO, A. TABBARA et R. K. BRAYTON. « A Force-Directed Maze Router ». Dans *Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD 2001)*, pages 404–407, 2001.

[164] S.-W. MOON et S.-G. KONG. « Block-Based Neural Networks ». *IEEE Transactions on Neural Networks*, 12(2) :307–317, mars 2001.

[165] E. F. MOORE. « The Shortest Path Through a Maze ». Dans *Proc. of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[166] E. F. MOORE. « *Sequential Machines, Selected Papers* », pages 213–214. Reading. Addison Wesley, Massachussets, 1964.

[167] H. MORAVEC. « When will Computer Hardware Match the Human Brain ? ». *Journal of Evolution and Technology*, 1, mars 1998.

[168] J.-M. MORENO, Y. THOMA, E. SANCHEZ, O. TORRES et G. TEMPESTI. « Hardware Realization of a Bio-inspired POEtic Tissue ». Dans R. S. ZEBULUM, D. GALTNEY, G. HORNBY, D. KEYMEULEN, J. LOHN et A. STOICA, éditeurs, *Proc. 2004 NASA/DoD Conference on Evolvable Hardware*, pages 237–244, Los Alamitos, California, 2004. IEEE Computer Society.

[169] J. M. MORENO AROSTEGUI, E. SANCHEZ et J. CABESTANY. « An In-System Routing Strategy for Evolvable Hardware Programmable Platforms ». Dans *Proc. 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 157–166. IEEE Computer Society Press, 2001.

[170] M. MOTOMURA, Y. AIMOTO, A. SHIBAYAMA, Y. YABE et M. YAMASHINA. « An Embedded DRAM-FPGA Chip with Instantaneous Logic Reconfiguration ». Dans *FCCM '98 : Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 264–266. IEEE Computer Society, 1998.

[171] M. MURAKAWA, S. YOSHIZAWA, I. KAJITANI, T. FURUYA, M. IWATA et T. HIGUCHI. « Hardware Evolution at Function Level ». Dans *PPSN IV : Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, pages 62–71. Springer-Verlag, 1996.

[172] Y. NAGATA. « The Lens Design Using the CMA-ES Algorithm ». Dans K. Deb et AL., éditeur, *Proc. Genetic and Evolutionary Computation COnference (GECCO 2004), Part II*, volume 3103 de *LNCS*, pages 1189–1200, Berlin, Heidelberg, 2004. Springer Verlag.

[173] R. NAIR. « A Simple yet Effective Technique for Global Wiring ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(2) :165–172, mars 1987.

[174] R. NAIR, S. J. HONG, S. LILES et R. VILLANI. « Global Wiring on a Wire Routing Machine ». Dans *Proceedings of the 19th Conference on Design Automation*, pages 224–231. IEEE Press, 1982.

[175] J. A. NESTOR. « A New Look at Hardware Maze Routing ». Dans ACM, éditeur, *Proc. 12th ACM Greate Lakes Symposium on VLSI (GLSVLSI '02)*, pages 142–147, New York, USA, avril 2002.

[176] J. A. NESTOR. « FPGA Implementation of a Multilayer Maze Router ». Dans *Proc. 6th Annual Military and Aerospace Programmable Logic Device (MAPLD) International Conference*, page P52, septembre 2003.

[177] L. M. NI et P. K. MCKINLEY. « A Survey of Wormhole Routing Techniques in Direct Networks ». *Computer*, 26(2) :62–76, 1993.

[178] S. NOLFI et D. FLOREANO. *Evolutionary Robotics. The Biology, Intelligence, and Technology of Self-organizing Machines*. MIT Press, Cambridge, MA, USA, 2001.

[179] U.E. NYDEGGER. « Hématologie 2001 : La Cellule Souche - d'Ici à Là et Retour ». *Forum Medical Suisse*, 51 :1269–1270, décembre 2001.

[180] M. OURA et S. MASUI. « A Secure Dynamically Programmable Gate Array Based on Ferroelectric Memory ». *Fujitsu Scientific & Technical Journal*, 39(1) :52–61, juin 2003.

[181] C. OZTURKERI et M. CAPCARRERE. « Emergent Robustness and Self-Repair through Developmental Cellular Systems ». Dans J. POLLACK, M. BEDAU, P. HUSBANDS, T. IKEGAMI et R. A. WATSON, éditeurs, *Proc. Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, pages 31–26, Cambridge, Massachusetts, USA, 2004. The MIT Press.

[182] S. PAJOT. « *Percolation et Economie* ». PhD thesis, Université de Nantes, 2001. Accessible sur http ://percolation.free.fr.

[183] U. PAPE. « Algorithm 562 : Shortest Path Lengths [H] ». *ACM Trans. Math. Softw.*, 6(3) :450–455, 1980.

[184] A. PEREZ-URIBE. « *Structure-Adaptable Digital Neural Networks* ». PhD thesis, Swiss Federal Institute of Technologiy-Lausanne, EPFL, 1999.

[185] C. Piguet et H. Hügli. *Du Zéro à l'Ordinateur. Une Brève Histoire du Calcul.* PPUR, Lausanne, 2004.

[186] M. Pollack et W. Wiebenson. « Solutions of the Shortest-Route Problem - A Review ». *Operations Research*, 8(2) :224–230, mars 1960.

[187] R. C. Prim. « Shortest Connection Networks and Some Generalizations ». *The Bell System Technical Journal*, 3 :1389–1401, 1957.

[188] QuickLogic. « *Eclipse Family Data Sheet* », 2001. Available from http ://www.quicklogic.com.

[189] H. Rapaport et P. Abramson. « An Analog Computer for Finding an Optimum Route Through a Communication Network ». *IEEE Transactions on Communications*, 7(1) :37–42, mai 1959.

[190] I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, Stuttgart, Germany, 1973.

[191] D. R. Riddle, T. Blumenthal, B. J. Meyer et J. R. Priess. *C. Elegans II.* Cold Spring Harbor Laboratory Press, 1997.

[192] D. Roggen. « *Multi-Cellular Reconfigurable Circuits : Evolution, Morphogenesis and Learning* ». PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Lausanne, 2005.

[193] D. Roggen, D. Floreano et C. Mattiussi. « A Morphogenetic Evolutionary System : Phylogenesis of the POEtic Tissue ». Dans A. M. Tyrrell, P. C. Haddow et J. Torresen, éditeurs, *Evolvable Systems : From Biology to Hardware ; Proceedings of the Fifth International Conference on Evolvable Systems (ICES 2003)*, pages 153–164, Berlin, 2003. Springer.

[194] D. Roggen, Y. Thoma et E. Sanchez. « An Evolving and Developing Cellular Electronic Circuit ». Dans J. Pollack, M. Bedau, P. Husbands, T. Ikegami et R. A. Watson, éditeurs, *Proc. Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, pages 33–38, Cambridge, Massachusetts, USA, 2004. The MIT Press.

[195] J. Rose. « Parallel Global Routing for Standard Cells ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(10) :1085–1095, octobre 1990.

[196] J. Rose, R. Francis, D. Lewis et P. Chow. « Architecture of Field-Programmable Gate Arrays : the Effect of Logic Block Functionality on Area Efficiency ». *Solid-State Circuits, IEEE Journal of*, 25(5) :1217–1225, octobre 1990.

[197] F. Rubin. « The Lee Path Connection Algorithm ». *IEEE Transactions on Computers*, c-23(9) :907–914, septembre 1974.

[198] D.E. Rumelhart, Bernard. Widrow et M.A. Lehr. « The Basic Ideas in Neural Networks ». *Commun. ACM*, 37(3) :87–92, 1994.

[199] S. J. Russell et P. Norvig. « *Artificial Intelligence : A Modern Approach* », Chapitre 4, pages 97–101. Prentice Hall, NJ, 2 édition, 1995.

[200] R. A. Rutenbar, T. N. Mudge et D. E. Atkins. « A Class of Cellular Architectures to Support Physical Design Automation ». *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 3(4) :264–278, octobre 1984.

[201] T. RYAN et E. ROGERS. « An ISMA Lee Router Accelerator ». *IEEE Design and Test of Computers*, 4(5) :38–45, octobre 1987.

[202] V. K. SAGAR et R. E. MASSARA. « General-Purpose Parallel Hardware Approach to the Routing Problem of VLSI Layout ». *Circuits, Devices and Systems, IEE Proceedings-G*, 140(4) :294–304, août 1993.

[203] E. SANCHEZ. « Field Programmable Gate Array (FPGA) Circuits ». Dans E. SANCHEZ et M. TOMASSINI, éditeurs, *Towards Evolvable Hardware*, volume 1062 de *LNCS*, pages 1–18, Berlin, 1996. Springer-Verlag.

[204] E. SANCHEZ, D. MANGE, M. SIPPER, M. TOMASSINI, A. PEREZ-URIBE et A. STAUFFER. « Phylogeny, Ontogeny, and Epigenesis : Three Sources of Biological Inspiration for Softening Hardware ». Dans T. HIGUCHI, M. IWATA et W. LIU, éditeurs, *Evolvable Systems : From Biology to Hardware*, volume 1259 de *LCNS*, pages 33–54, Berlin, 1997. Springer-Verlag.

[205] T. SASAKI et M. TOKORO. « Comparison between Lamarckian and Darwinian Evolution on a Model Using Neural Networks and Genetic Algorithms ». *Knowledge and Information Systems*, 2(2) :201–222, juin 2000.

[206] M. SATO, K. KUBOTA et T. OHTSUKI. « A Hardware Implementation of Gridless Routing Based on Content Addressable Memory ». Dans *Proc. 27th ACM/IEEE design automation conference*, pages 646–649, New York, NY, USA, 1991. ACM Press.

[207] S. M. SCALERA et J. R. VÁZQUEZ. « The Design and Implementation of a Context Switching FPGA ». Dans *FCCM '98 : Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 78–85. IEEE Computer Society, 1998.

[208] Martin SCHÄFER et Georg HARTMANN. « A Flexible Hardware Architecture for Online Hebbian Learning in the Sender-Oriented PCNN-Neurocomputer Spike 128K ». Dans *MICRONEURO '99 : Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems*, page 316. IEEE Computer Society, 1999.

[209] S.D. SCOTT, A. SAMAL et S. SETH. « HGA : A Hardware-Based Genetic Algorithm ». Dans *FPGA*, pages 53–59, 1995.

[210] R. SEGEV et E. BEN-JACOB. « From Neurons to Brain : Adaptive Self-wiring of Neurons ». *Journal of Complex Systems*, 1 :67–78, 1998.

[211] C. E. SHANNON. « Presentation of the Maze-Solving Machine ». Dans *Transactions of the 8th Cybernetics Conference*, pages 173–180, Josiah Macy Jr. Fundation, New York, 1952. Disponible dans C.E. Shannon, N.J.A. Sloane and A.D. Wyner, *Claude Elwood Shannon : Collected papers*, pages 681-687, IEEE Computer Society Press, New York, 1993.

[212] T. SHIBATA, T. NAKAI, Y. Ning MEI, Y. YAMASHITA, M. KONDA et T. OHMI. « Advances in Neuron-MOS Applications ». Dans *IEEE International Conference on Solid-State Circuits. Digest of Technical Papers. 43rd ISSCC*, pages 304–305, 1996.

[213] T. Shibata et T. Ohmi. « Neuron MOS Binary-logic Integrated Circuits. I. Design Fundamentals and Soft-hardware-logic Circuit Implementation ». *IEEE Transactions on Electron Devices*, 40(3) :570–576, mars 1993.

[214] T. Shoenauer, S. Atasoy, N. Mehrtash et H. Klar. « NeuroPipe-Chip : A Digital Neuro-Processor for Spiking Neural Networks ». *IEEE Transactions on Neural Networks*, 13(1) :205–213, janvier 2002.

[215] R. P. S. Sidhu, A. Mei et V. K. Prasanna. « Genetic Programming Using Self-Reconfigurable FPGAs ». Dans *FPL '99 : Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 301–312. Springer-Verlag, 1999.

[216] S. Singh, J. Rose, P. Chow et D. Lewis. « The Effect of Logic Block Architecture on FPGA Performance ». *Solid-State Circuits, IEEE Journal of*, 27(3) :281–287, mars 1992.

[217] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe et A. Stauffer. « A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-inspired Hardware Systems ». *IEEE Transactions on Evolutionary Computation*, 1(1) :83–97, avril 1997.

[218] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe et A. Stauffer. An Introduction to Bio-inspired Machines. Dans D. Mange et M. Tomassini, éditeurs, *Bio-Inspired Computing Machines : Towards Novel Computational Architectures*, Chapitre 1, pages 1–12. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[219] M.J.S. Smith. *Application-Specific Integrated Circuits*. VLSI Design Series. Addison-Wesley Publishing Company, juin 1997.

[220] M.J.S. Smith. « *Application-Specific Integrated Circuits* », Chapitre 1.1.2, pages 6–11. VLSI Design Series. Addison-Wesley Publishing Company, juin 1997.

[221] K. Someya, H. Shinozaki et Y. Sekine. « Pulse-type Hardware Chaotic Neuron Model and its Bifurcation Phenomena ». *Neural Networks*, 12(1) :153–161, 1999.

[222] P. Sordino, F. Van Der Hoeven et D. Duboule. « *Hox* Gene Expression in Teleost Fins and the Origin of Vertebrate Digits ». *Nature*, 375(6533) :678–681, juin 1995.

[223] J. Soukup. « Fast Maze Router ». Dans *Proc. 15th design automation conference*, pages 100–102, Piscataway, NJ, USA, 1978. IEEE Press.

[224] T. D. Spiers et D. A. Edwards. « A High Performance Routing Engine ». Dans *Proceedings of the 24th ACM/IEEE Conference on Design Automation*, pages 793–799. ACM Press, 1987.

[225] C. Steiger, H. Walder et M. Platzner. « Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices ». Dans P. Y. K. Cheung, G. A. Constantinides et J. T. de Sousa, éditeurs, *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, volume 2778 de *LNCS*, pages 575–584, Berlin, Heidelberg, 2003. Springer Verlag.

[226] A. STOICA, D. KEYMEULEN, R. TAWEL, C. SALAZAR-LAZARO et W. LI. « Evolutionary Experiments with a Fine-Grained Reconfigurable Architecture for Analog and Digital CMOS Circuits ». Dans *EH ’99 : Proceedings of the 1st NASA/DOD workshop on Evolvable Hardware*, page 76. IEEE Computer Society, 1999.

[227] T. SUDO, T. NAKASHIMA, M. AOKI et T. KONDO. « An LSI Adaptive Array Processor ». *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, XXV :122–123, février 1982.

[228] K. SUZUKI, Y. MATSUNAGA, M. TACHIBANA et T. OHTSUKI. « A Hardware Maze Router with Application to Interactive Rip-up and Reroute ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(4) :466–476, octobre 1986.

[229] Y. TAKAHASHI et S. SASAKI. « Parallel Automated Wire-Routing with a Number of Competing Processors ». *SIGARCH Comput. Archit. News*, 18(3) :310–317, 1990.

[230] G. TEMPESTI, D. MANGE et A. STAUFFER. « A Robust Multiplexer-Based FPGA Inspired by Biological Systems ». *Journal of Systems Architecture : Special Issue on Dependable Parallel Computer Systems*, 40(10) :719–733, septembre 1997.

[231] C. TEUSCHER et E. SANCHEZ. « Self-Organizing Topology Evolution of Turing Neural Networks ». Dans G. DORFFNER, H. BISCHOF et K. HORNIK, éditeurs, *Proceedings of the International Conference on Artificial Neural Networks (ICANN2001)*, volume 2130 de *Lecture Notes in Computer Science*, pages 820–826, Berlin, Heidelberg, 2001. Springer-Verlag.

[232] Y. THOMA, D. ROGGEN, E. SANCHEZ et J.-M. MORENO. « Prototyping with a Bio-inspired Reconfigurable Chip ». Dans F. TITSWORTH, éditeur, *Proc. 15th IEEE International Workshop on Rapid System Prototyping (RSP 2004)*, pages 239–246, Los Alamitos, California, 2004. IEEE Computer Society.

[233] Y. THOMA et E. SANCHEZ. « A Reconfigurable Chip for Evolvable Hardware ». Dans K. Deb et AL., éditeur, *Proc. Genetic and Evolutionary Computation COnference (GECCO 2004), Part I*, numéro 3102 dans LNCS, pages 816–827, Berlin, Heidelberg, 2004. Springer Verlag.

[234] Y. THOMA, E. SANCHEZ, J.-M. MORENO AROSTEGUI et G. TEMPESTI. « A Dynamic Routing Algorithm for a Bio-Inspired Reconfigurable Circuit ». Dans P. Y. K. CHEUNG, G. A. CONSTANTINIDES et J. T. de SOUSA, éditeurs, *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL’03)*, volume 2778 de *LNCS*, pages 681–690, Berlin, Heidelberg, 2003. Springer Verlag.

[235] A. THOMPSON. « Silicon Evolution ». Dans John R. KOZA, David E. GOLDBERG, David B. FOGEL et Rick L. RIOLO, éditeurs, *Genetic Programming 1996 : Proceedings of the First Annual Conference*, pages 444–452, Stanford University, CA, USA, 1996. MIT Press.

[236] A. THOMPSON. « An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics ». Dans T. HIGUCHI, M. IWATA et L. WEIXIN, éditeurs, *Proc. 1st Int. Conf. on Evolvable Systems (ICES’96)*, volume 1259 de *LNCS*, pages 390–405. Springer-Verlag, 1997.

[237] A. Thompson, I. Harvey et P. Husbands. « Unconstrained Evolution and Hard Consequences ». Dans *Papers from an international workshop on Towards Evolvable Hardware, The Evolutionary Engineering Approach*, pages 136–165. Springer-Verlag, 1996.

[238] O. Torres, J. Eriksson, J. M. Moreno et A. Villa. « Hardware Optimization and Serial Implementation of a Novel Spiking Neuron Model for the POEtic Tissue ». *Biosystems*, 76(1-3) :201–208, août-octobre 2004.

[239] J. Torresen. « A Divide-and-Conquer Approach to Evolvable Hardware ». Dans M. Sipper, D. Mange et A. Pérez-Uribe, éditeurs, *ICES'98*, volume 1478 de *Lecture Notes in Computer Science*, pages 57–65, Berlin Heidelberg, 1998. Springer-Verlag.

[240] J. Torresen. « Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications ». Dans R.W. Hartenstein et H. Grünbacher, éditeurs, *FPL 2000*, volume 1896 de *Lecture Notes in Computer Science*, pages 230–239, Berlin Heidelberg, 2000. Springer-Verlag.

[241] J. Torresen. « Evolvable Hardware as a New Computer Architecture ». Dans *Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet*, 2002.

[242] J. Torresen. « A Scalable Approach to Evolvable Hardware ». *Genetic Programming and Evolvable Machines*, 3(3) :259–282, 2002.

[243] S. Trimberger, D. Carberry, A. Johnson et J. Wong. « A Time-multiplexed FPGA ». Dans *FCCM '97 : Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, pages 22–28. IEEE Computer Society, 1997.

[244] S. M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, 1994.

[245] A.M. Tyrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg et A. E.P. Villa. « POEtic Tissue : An Integrated Architecture for Bio-inspired Hardware ». Dans A.M. Tyrrell, P.C. Haddow et J. Torresen, éditeurs, *Evolvable Systems : From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES 2003)*, numéro 2606 dans LNCS, pages 129–140, Berlin, Heidelberg, 2003. Springer Verlag.

[246] A. Upegui, C.A. Pena-Reyes et E. Sanchez. « A Functional Spiking Neuron Hardware Oriented Model ». Dans *Proceedings of the International Work-conference on Artificial and Natural Neural Networks IWANN2003*, volume 2686 de *Lecture Notes in Computer Science*, pages 136–143, Berlin Heidelberg, 2003. Springer.

[247] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Illinois, 1966. Edited and completed by A. W. Burks.

[248] Jr. W. A. Dees et II Robert J. Smith. « Performance of Interconnection Rip-up and Reroute Strategies ». Dans *Proceedings of the 18th conference on Design automation*, pages 382–390. IEEE Press, 1981.

[249] A. Waksman. « An Optimal Solution to the Firing Squad Synchronization Problem ». *Information and Control*, 9 :66–78, 1966.

[250] T. WATANABE, H. KITAZAWA et Y. SUGIYAMA. « A Parallel Adaptable Routing Algorithm and its Implementation on a Two-Dimensional Array Processor ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(2) :241–250, mars 1987.

[251] T. WATANABE et Y. SUGIYAMA. « A New Routing Algorithm and its Hardware Implementation ». Dans *Proc. 23rd ACM/IEEE conference on Design automation*, pages 574–580, Piscataway, NJ, USA, 1986. IEEE Press.

[252] J. D. WATSON et F. H. C. CRICK. « A Structure for Deoxyribose Nucleic Acid ». *Nature*, 171 :737–738, 1953.

[253] P. D. WHITING et J. A. HILLIER. « A Method for Finding the Shortest Route through a Road Network ». *Operational Research Quarterly*, 11 :37–40, 1960.

[254] B. WIDROW et M.A. LEHR. « 30 Years of Adaptive Neural Networks : Perceptron, Madaline, and Backpropagation ». *Proceedings of the IEEE*, 78(9) :1415–1442, septembre 1990.

[255] L. WOLPERT. *Le Triomphe de l'Embryon*. Dunod, Paris, 1992.

[256] Y. WON, S. SAHNI et Y. EL-ZIQ. « A Hardware Accelerator for Maze Routing ». Dans *Proc. 24th ACM/IEEE Design Automation Conference*, pages 800–806. ACM Press, 1987.

[257] K.-C. WU et Y.-W. TSAI. « Structured ASIC, Evolution or Revolution ? ». Dans *Proc. 2004 international symposium on Physical design (ISPD'04)*, pages 103–106, New York, NY, USA, 2004. AMC Press.

[258] XILINX. « *XC6200 Field Programmable Gate Arrays* », avril 1997.

[259] XILINX. « *Spartan-3 FPGA Family : Introduction and Ordering Information* », juillet 2004. Available from http ://www.xilinx.com.

[260] XILINX. « *Virtex-3 User Guide* », septembre 2004. Available from http ://www.xilinx.com.

[261] X. YAO. « Evolving Artificial Neural Networks ». *Proceedings of the IEEE*, 87(9) :1423–1447, septembre 1999.

[262] I.-L. YEN, R. M. DUBASH et F. B. BASTANI. « Strategies for Mapping Lee's Maze Routing Algorithm onto Parallel Architectures ». Dans *Proc. Seventh International Parallel Processing Symposium*, pages 672–679. IEEE, 1993.

[263] R.S. ZEBULUM, M. Aurélio PACHECO et M. VELLASCO. « Analog Circuits Evolution in Extrinsic and Intrinsic Modes ». Dans M. SIPPER, D. MANGE et A. PÉREZ-URIBE, éditeurs, *ICES '98 : Proceedings of the Second International Conference on Evolvable Systems*, volume 1478 de *LNCS*, pages 154–165. Springer-Verlag, 1998.

Curriculum Vitæ

Données Personnelles

Prénom Nom	Yann Thoma
Date de Naissance	20 Septembre 1977
Lieu de Naissance	Genève
Statut	Célibataire
Nationalité	St-Gall, Suisse

Formation

2001-2005	**Thèse de doctorat au Laboratoire de Systèmes Logiques de l'École Polytechnique Fédérale de Lausanne (EPFL).** Tissu Numérique Cellulaire à Routage et Configuration Dynamiques. Directeur de Thèse : Eduardo Sanchez.
1996-2001	**Diplôme d'ingénieur informaticien à l'EPFL.** Travail de diplôme : CoDeNios : Un Outil de Codesign pour le Processeur Nios.
1992–1996	**Certificat de maturité scientifique** (type C), Collège de Saussure, Genève. **1er prix du concours d'informatique du Collège de Genève**, pour un logiciel de conception de circuits de trains électriques.

Expérience Professionnelle

2001-2005	Assistant de Recherche au Laboratoire de Systèmes Logiques, EPFL.
2000	Stage de 3 mois chez **Kongsberg Offshore AS** en Norvège. Développement d'un logiciel distribué dédié aux systèmes de mesures temps réel avec gestion d'une base de données.
2001-2004	Assistant au Laboratoire de Systèmes Logiques, pour les cours suivants : Systèmes Logiques, Architecture des Ordinateurs, Conception Avancée de Systèmes Numériques.
2003-2004	Assistant pour le cours de Programmation Java des étudiants 1ère année de Systèmes de Communication.

Projets de Recherche

2001-2002	**Tissu reconfigurable BioWall** Rôle : Développement d'un logiciel de contrôle du BioWall. Participants : D. Mange, A. Stauffer, C. Teuscher, G. Tempesti, F. Vannel Financement : Fondation de la Villa Reuge.

2001-2004 **Projet POEtic (IST-2000-28027)**
Rôle : Développement de la partie reconfigurable du tissu POEtic, ainsi que des outils logiciels d'aide à la création de designs.
Participants : Universités de York, Glasgow, Barcelone et Lausanne.
Financement : Bourse 00.0529-1 du Fond National Suisse de la Recherche.

Intérêts de Recherche

Systèmes digitaux reconfigurables

Routage matériel

Systèmes bio-inspirés

Auto-organisation

Langues

Français	Langue maternelle
Anglais	Très bon
Allemand	Connaissances de base

Compétences Particulières

Langages	C/C++, Java, Ada, Pascal, Assembleur, PHP
Langages Matériels	VHDL, SystemC
Méthode de Développement	Fusion

Liste de publications

Article de journal :

2004
- Y. Thoma, G. Tempesti, E. Sanchez et J.-M. Moreno Arostegui. « POEtic : An Electronic Tissue for Bio-Inspired Cellular Applications ». *BioSystems*, 74(1-3) :191–200, 2004.

Articles parus dans des actes de conférences :

2004
- Y. Thoma et E. Sanchez. « A Reconfigurable Chip for Evolvable Hardware ». Dans K. Deb et al., éditeurs, *Proc. Genetic and Evolutionary Computation COnference (GECCO 2004), Part I*, volume 3102 dans LNCS, pages 816–827, Berlin, Heidelberg, 2004. Springer Verlag.

- J.-M. Moreno, Y. Thoma, E. Sanchez, O. Torres , E. Sanchez et G. Tempesti. « Hardware Realization of a Bio-inspired POEtic tissue ». Dans R. S. Zebulum, D. Galtney, G. Hornby, D. Keymeulen, J. Lohn et A. Stoica, éditeurs, *Proc. 2004 NASA/DoD Conference on Evolvable Hardware*, pages 237–244, Los Alamitos, California, 2004. IEEE Computer Society.

- Y. Thoma, D. Roggen, E. Sanchez et J.-M. Moreno. « Prototyping with a Bio-inspired Reconfigurable Chip ». Dans F. Titsworth, éditeur, *Proc. 15th IEEE International Workshop on Rapid System Prototyping (RSP 2004)*, pages 239–246, Los Alamitos, California, 2004. IEEE Computer Society.

- D. Roggen, Y. Thoma et E. Sanchez. « An Evolving and Developing Cellular Electronic Circuit ». Dans J. Pollack, M. Bedau, P. Husbands, T. Ikegami et R. A. Watson, éditeurs, *Proc. Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, pages 33–38, Cambridge, Massachusetts, USA, 2004. The MIT Press.

2003

- Y. Thoma, E. Sanchez, J.-M. Moreno Arostegui et G. Tempesti. « A Dynamic Routing Algorithm for a Bio-Inspired Reconfigurable Circuit ». Dans P. Y. K. Cheung, G. A. Constantinides et J. T. de Sousa, éditeurs, *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, volume 2778 de *LNCS*, pages 681–690, Berlin, Heidelberg, 2003. Springer Verlag.

- D. Roggen, S. Hofmann, Y. Thoma et D. Floreano. « Hardware Spiking Neural Network with Run-Time Reconfigurable Connectivity in an Autonomous Robot ». Dans J. Lohn, R. Zebulum, J. Steincamp, D. Keymeulen, A. Stoica et M. I. Ferguson, éditeurs, *Proc. 2003 NASA/DoD Conference on Evolvable Hardware*, pages 189–198, Los Alamitos, California, 2003. IEEE Computer Society.

- G. Tempesti, D. Mange, E. Petraglio, A. Stauffer et Y. Thoma. « Developmental Processes in Silicon : An Engineering Perspective ». Dans J. Lohn, R. Zebulum, J. Steincamp, D. Keymeulen, A. Stoica et M. I. Ferguson, éditeurs, *Proc. 2003 NASA/DoD Conference on Evolvable Hardware*, pages 255–264, Los Alamitos, California, 2003. IEEE Computer Society.

- G. Tempesti, D. Roggen, E. Sanchez, Y. Thoma, R. Canham et A.M. Tyrell. « Ontogenetic Development and Fault Tolerance in the POEtic Tissue ». Dans A.M. Tyrrell, P.C. Haddow et J. Torresen, éditeurs, *Evolvable Systems : From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES 2003)*, volume 2606 dans LNCS, pages 141–152, Berlin, Heidelberg, 2003. Springer Verlag.

- Y. Thoma, E. Sanchez, J.-M. Moreno Arostegui et G. Tempesti. « Un Sistema de Enrutamiento Dinàmico para un Circuito Reconfigurable Bio-Inspirado ». Dans E. B. Scalvinoni, F. G. Arribas, S. Lopez-Buedo et G. S. Capristo, éditeurs, *Computacion Reconfigurable & FPGAs*, pages 27–37, Universided Autonoma de Madrid, 2003.

2002

- G. Tempesti, D. Roggen, E. Sanchez, Y. Thoma, R. Canham, A. Tyrrell et J.-M. Moreno. « A POEtic Architecture for Bio-Inspired Systems ». Dans R.K. Standish, M.A. Bedau et H.A. Abbass, éditeurs, *Proc. 8th Int. Conf. of Artificial Life VIII*, pages 111–115, Cambridge, Massachusetts, décembre 2002. MIT Press.
- Y. Thoma et E. Sanchez. « CoDeNios : A Function Level Co-Design Tool ». Dans *Workshop on Computer Architecture Education, WCAE 2002, Workshop Proceedings*, pages 73–78, Anchorage, Alaska, 2002.