



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Laboratoire de Systèmes Logiques

Outil de simulation de réseaux de molécules

Edouard FORLER

Assistants

Gianluca TEMPESTI

Dominik MADON

Projet de 7^{ème} semestre
Professeur Daniel MANGE

Lausanne, février 1999

Chapitre 1

Prolégomènes

1.1 Contexte

Alors même que les circuits à haute intégration deviennent de plus en plus petits et de plus en plus complexes, ceux-ci sont de plus en plus sensibles à l'apparition de fautes de fabrication et de fonctionnement. Comment résoudre ces problèmes ?

C'est ainsi qu'est né au Laboratoire de Systèmes Logiques de l'EPFL le projet *Embryonics* dont le but est de développer un nouveau type de FPGA à haute complexité, dont les concepts sont inspirés de mécanismes que la Nature utilise, elle, depuis toujours.

La première phase d'*Embryonics* a ainsi permis l'élaboration de deux circuits doués d'autoréplication et d'autoréparation, la "cellule" 601 (MicTree) et la "molécule" 603 (MuxTree).

La deuxième phase du projet *Embryonics* consiste à synthétiser la cellule MicTree à l'aide des molécules MuxTree. C'est dans ce cadre que se situe ce travail de semestre, qui vise à réaliser un simulateur de MuxTrees de manière à faciliter la synthèse du MicTree.

La cellule MicTree une fois reconstruite, il devrait être possible, dans une troisième phase, de mettre au point de véritables "organismes" composés de MicTrees, et également doués d'autoréplication et d'autoréparation. Le premier de ces organismes reposant sur la couche moléculaire sera la Biowatch 2001, une montre complète et extrêmement tolérante aux pannes de toutes sortes.

1.2 Enoncé du projet de semestre

On cherche à développer un outil capable de simuler un réseau de molécules artificielles ; cet outil permettra la simulation en pas-à-pas, la pose de points d'arrêt et l'examen des variables d'état des molécules.

Par molécule artificielle, on entend la molécule 603 MuxTree et son extension RamMuxTree, développés au Laboratoire de Systèmes Logiques de l'École Polytechnique Fédérale de Lausanne.

1.3 Contraintes

Certaines contraintes ont été émises quant à la réalisation du projet de semestre. Le développement doit en particulier se faire en C++, pour son aspect orienté objet d'une part, et sa portabilité sur les systèmes Unix d'autre part. On souhaite que l'approche orienté objet

permette d'obtenir une certaine flexibilité du moteur de simulation ; en effet, les molécules artificielles sont soumises à des modifications et des améliorations futures. Il s'agit de pouvoir réutiliser le moteur en tenant compte de ces éventuelles modifications, sans devoir refaire un développement lourd et coûteux du système. L'accent devra ainsi être porté sur le respect des mécanismes mis en oeuvre dans la réalisation matérielle des molécules, dans la mesure où cette implémentation ne représente pas un coût prohibitif en temps de calcul. Il ne faudra également pas perdre de vue que, si l'actuel prototype matériel est composé de 18 molécules, le but final est de développer des réseaux pouvant contenir plusieurs centaines de molécules dans les cas les plus simples.

Dans l'état actuel des choses, il n'est pas spécifiquement demandé de réaliser l'interface utilisateur. Le moteur devra bien évidemment être suffisamment ouvert pour en permettre l'intégration dans un logiciel plus complet comportant notamment ladite interface. Par ailleurs, même s'il s'agit à terme de simuler des réseaux de taille élevée, la performance tant en consommation mémoire qu'en temps de calcul n'est pas un critère prioritaire.

1.4 Cadre administratif

Ce projet est réalisé lors du 7ème semestre (hiver 1998-99) au Laboratoire de Systèmes Logiques du Département d'Informatique de l'EPFL, sous la conduite du professeur Daniel Mange (responsable du développement *Embryonics*). Les assistants responsables du suivi du projet sont Gianluca Tempesti et Dominik Madon.

1.5 Remerciements

Je tiens à remercier plus particulièrement Gianluca Tempesti et Lucian Prodan pour l'accès qui m'a été donné à la riche documentation sur le projet *Embryonics*, pour leur conseils et leurs explications sur le fonctionnement des mécanismes critiques de la molécule 603. Je remercie également le professeur Daniel Mange, pour m'avoir proposé de participer aux réunions de travail du groupe *Embryonics*, ce qui m'a permis de placer mon travail dans un cadre beaucoup plus large que celui d'un simple projet de semestre.

Chapitre 2

Molécules, cellules, réseaux

Ce chapitre a pour but de tenir à disposition du lecteur les principaux éléments nécessaires à la compréhension des concepts et architectures développés au cours du projet *Embryonics*. Ces éléments ont été extraits des publications [4] et [1] et sont présentés dans leur forme originale, en langue anglaise.

2.1 Molécule MuxTree-SR (biodule 603)

Cet extrait tiré de [4] décrit les principales caractéristiques de la molécule MuxTree-SR.

2.1.1 An FPGA for the Embryonics Project : MuxTree

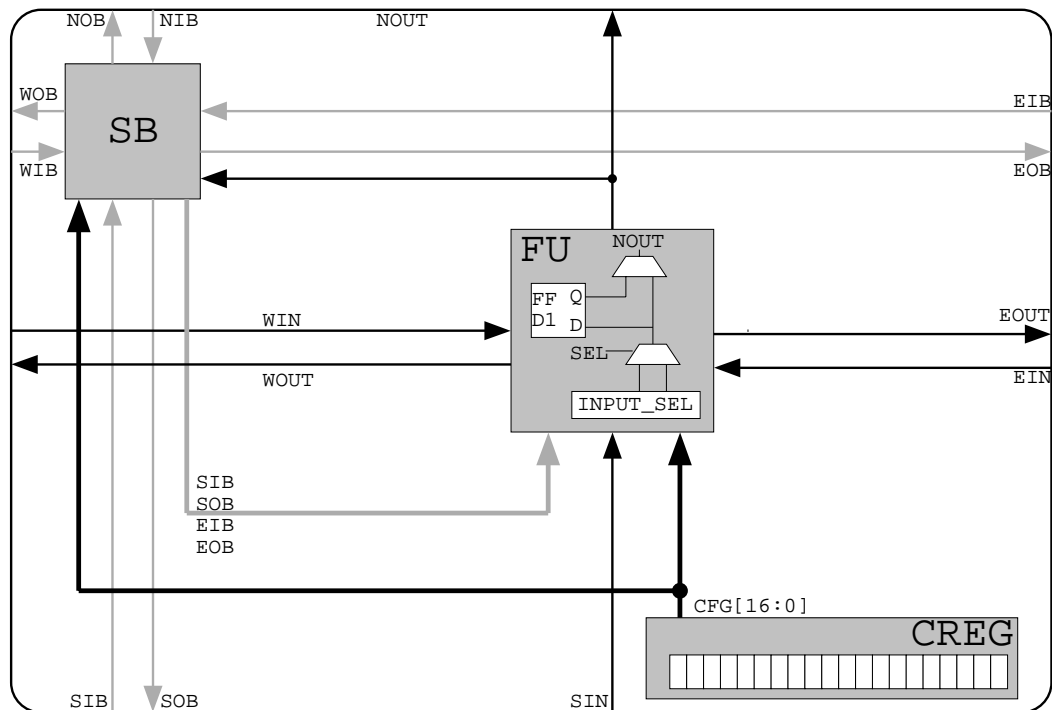


FIG. 2.1: The structure of a MuxTree element

[..] The basic element of our FPGA (Figure 2.1) is composed of three separate subsystems : the programmable function (FU), the programmable connections (SB), and the configuration register (CREG).

The programmable function is realized using a single two-input multiplexer (the name MuxTree stands for *tree of multiplexers*). The multiplexer being a universal gate (i.e., it is possible to realize any function given a sufficient number of multiplexers), the first requirement for an FPGA element is respected. In addition to the multiplexer, each element is also capable of storing a single bit of information in a D-type flip-flop, thus fulfilling the second requirement.

As for the programmable connection network, a MuxTree element contains two separate sets of connections : a fixed short-distance network for communication between immediate neighbors, and a programmable long-distance network for distant elements. The latter is controlled by a *switch box* (SB) which can route the output NOUT of an element to its four neighbors, as well as propagate signals in the four cardinal directions.

The element's function and connections are determined by a 17-bit configuration string, stored in the shift register CREG. These bits are sufficient to configure both the programmable function and the connection networks. All the configuration registers of all the elements are chained together to form a long shift register, and the configuration bitstream enters the array at the lower left corner and *propagates* from there to all the elements in the circuit.

On the surface, MuxTree is not necessarily more adapted to the implementation of bio-inspired systems than any other FPGA. However, designing a dedicated FPGA allows us to alter every detail of its architecture in order to meet our requirements (something which would not be possible with commercial FPGAs).

These requirements are fairly straightforward : we require an FPGA that can easily be configured as an array of identical processors (our artificial cells, which have an identical hardware structure). In other words, it must support self-replication, that is, the creation of multiple identical copies of our cells. In addition, since the repair mechanism at the cellular level is costly (the death of an entire column of processors represents a considerable loss of hardware resources), it would be extremely interesting for our molecules to be able to survive at least some faults (defects in the silicon substrate).

2.1.2 Self-Replication

The self-replication of the organism, achieved through the cycling of the cell's coordinates, is an immediate consequence of the architecture of our artificial cells. In order to obtain the self-replication of the cells, we analogously decided to include dedicated hardware in the architecture of our artificial molecules. This hardware will then allow us to obtain multiple copies of our cell without excessive difficulty. Unfortunately, the approach to follow in the implementation of such a mechanism was not immediately obvious, as research in the field of self-replicating hardware is relatively scarce.

Von Neumann's universal constructor was probably the first example of self-replicating computer hardware. Unfortunately, electronic technology in the fifties did not allow the development of so complex a machine. As a consequence, research on self-replicating hardware waned for several years. In the eighties, bio-inspiration gained new momentum under the label of *artificial life*, a research field pioneered by Christopher Langton, and is attracting more and more interest in the engineering community.

Langton approached the phenomenon of self-replication from a slightly different angle : he tried to define the smallest machine capable exclusively of self-replication (leaving aside von

Neumann's concept of universal computation and construction). The result was a fairly simple cellular automaton known as *Langton's loop* [10], which was the basis for our own attempts to develop self-replicating structures.

The first, theoretical phase in our research was therefore, to follow in the tradition of our predecessors, based on the use of cellular automata, and resulted in the development of a series of novel self-replicating loops considerably more versatile and powerful than Langton's [12] [15] [18].

The transition from cellular automata to hardware, however, required a careful process of synthesis, since cellular automata are very inefficient from the point of view of a hardware realization. We tried to identify the mechanisms at the core of our cellular automata, in view of a possible simplification and adaptation to hardware.

The key observation of this process was that the self-replication of our loops occurred through two distinct phases : a *structural phase*, where the "skeleton" of the offspring is created in the empty CA space, and a *configuration phase*, where the functionality of the parent (i.e., the operational information) is copied into the offspring.



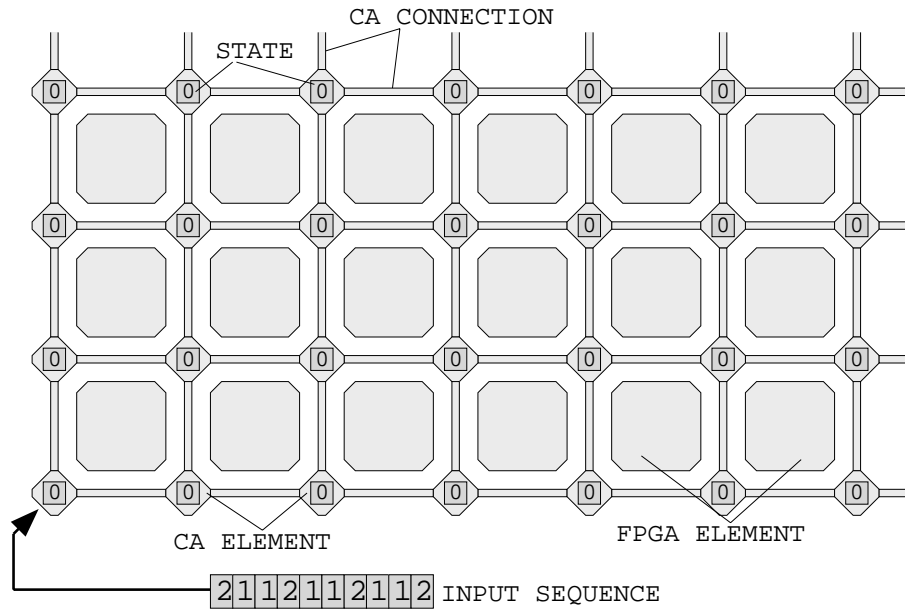


FIG. 2.3: The membrane builder is inserted among the MuxTree elements

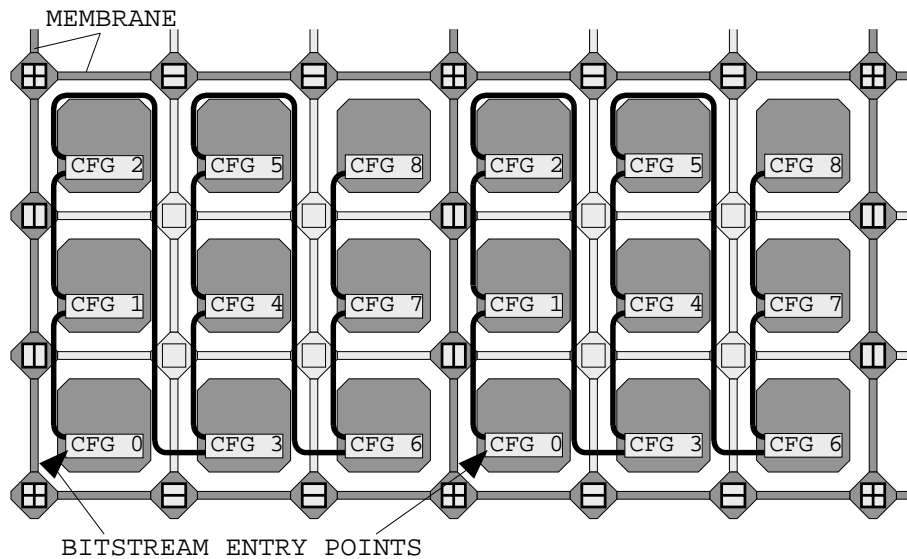


FIG. 2.4: The membrane is used to direct the configuration bitstream

The automaton can thus be seen as creating a *cellular membrane* that will surround each of our cells. Once the membrane is in place, we can in fact use it to direct the propagation of the FPGA's configuration (Figure 2.4) : since the configuration of each cell is identical, we can send the bitstream in all the blocks in parallel, thus automatically creating multiple copies of our cells, our requirement for self-replication.

By exploiting the experience accumulated in designing our self-replicating loops, we were thus able to create a very simple self-replication mechanism for our FPGA. At this stage, we

turned our attention to the second bio-inspired feature of our FPGA : self-repair.

2.1.3 Self-Repair

The development of a mechanism allowing MuxTree to self-repair is somewhat different from that of self-replication : as we mentioned, self-repair is a much more “conventional” property of digital circuits, and a considerable body of knowledge exists. Its conception therefore did not require quite as much original research on our part. On the other hand, the constraints of our project imposed considerable technical difficulties, particularly where self-test (i.e., the ability to detect the presence and the location of faults, a necessary prerequisite for any self-repair system) is concerned.

Self-Test in MuxTree

Any literature search, however superficial, on the subject of testing will reveal the existence of a considerable variety of approaches to implementing self-test in digital circuits [5], including some which can be applied to FPGAs [6] [8] [13]. Although we exploited to the greatest possible extent this existing knowledge base in our system, we found that the special requirements of our bio-inspired systems prevented the use of off-the-shelf approaches.

Among the most rigid constraints we had to respect, we will mention the need for *fault location* (that is, the need to determine not only that a defect is present, but also its exact position, so that it can be repaired), the requirement that the system be completely *homogeneous* (preventing the use of centralized control systems), and particularly our desire that the test occur, for the most part, *on-line* (that is, while the circuit is operating, preventing the use of a separate test phase).

Analyzing MuxTree’s three subsystems separately, we developed the following approach [16] [17] [18] :

- Considering its relatively small size (it occupies approximately 10% of the total silicon area of an element), we decided to test the *programmable function* through duplication (Figure 2.5), a technique as common in computer design as it is in nature (see, for example, the DNA’s double-helix structure). A simple comparison of the two outputs will then reveal the presence of a fault. In addition, a third copy of the flip-flop is necessary to guarantee that the correct value will be preserved for self-repair.
- Solutions to the problem of testing the *programmable connections* in an FPGA do indeed exist, but invariably require a considerable amount of redundancy through duplication. While not excluding the possibility of introducing it in the future, we deemed that the advantages to be gained from the test of the connections did not justify the considerable hardware overhead, at least in our current implementation.
- Testing the *configuration register* poses similar problems, but its size (about 80% of the surface of an element) makes testing imperative and prevents duplication-based approaches. The mechanism we finally settled on is based on the use of a special *test pattern*, sent to all the registers ahead of the configuration bitstream. Without describing the mechanism in detail, we will mention that it is capable of detecting any fault in the register with a remarkably small amount of additional hardware.

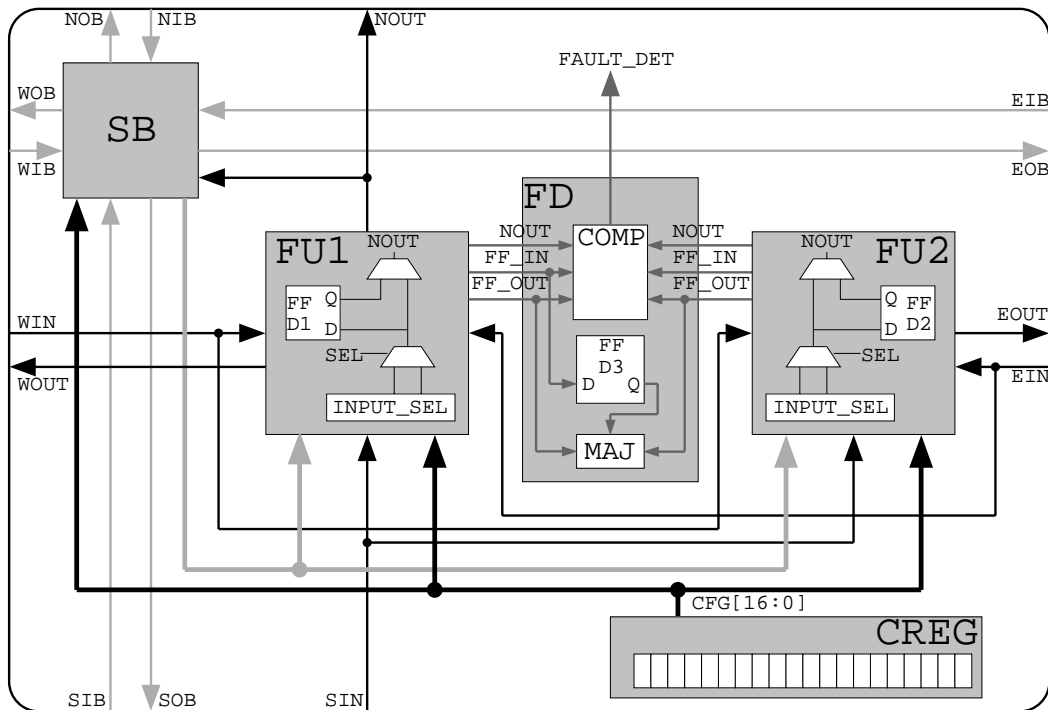


FIG. 2.5: The self-test logic for the programmable function in a MuxTree element

In conclusion, even if we could not quite meet our goal of assuring on-line fault detection (the test of the register occurs off-line during configuration), we were able to design an extremely simple fault detection system (the hardware overhead is less than 20% of an element's silicon surface) which, as we will see, is perfectly compatible with self-repair.

Self-Repair in MuxTree

As was the case for self-test, there exist a number of well-known approaches to implementing self-repair in two-dimensional arrays of identical elements [7] [9] [11]. Most rely on two mechanisms : since physically repairing a hardware fault is impossible, we must provide a set of spare elements (*redundancy*) and a way to let them replace faulty elements in the array, that is, to reroute the connections between the elements (*reconfiguration*). The self-repair system we developed for MuxTree is no exception, even if it had to satisfy a set of relatively non-standard constraints imposed by the unique features of our FPGA.

To find an efficient mechanism to implement redundancy, we turned our attention back to the self-replication mechanism. It is in fact fairly simple to modify the automaton to use the membrane itself to define which of the columns of the array will contain spare elements (Figure 2.6). Simply by adding one additional state to the automaton, we obtain a very powerful system : this approach allows us not only to limit reconfiguration to the interior of a block (a desirable feature), but also to program the robustness of the system. In fact, by adding or removing these special states to or from the CA input sequence, we can modify the frequency of spare columns, and thus the fault tolerance of the system. Without altering the configuration bitstream of the MuxTree elements (an important advantage, since the generation of a bitstream is a necessarily time-consuming process), we can introduce varying

degrees of robustness, from zero fault tolerance (no spare columns) to 100% redundancy (one spare column per active column).

To take advantage of the spare elements, we also require a mechanism to transfer the information stored in a faulty element (its configuration plus the value stored in its flip-flops) to one of the spare elements.

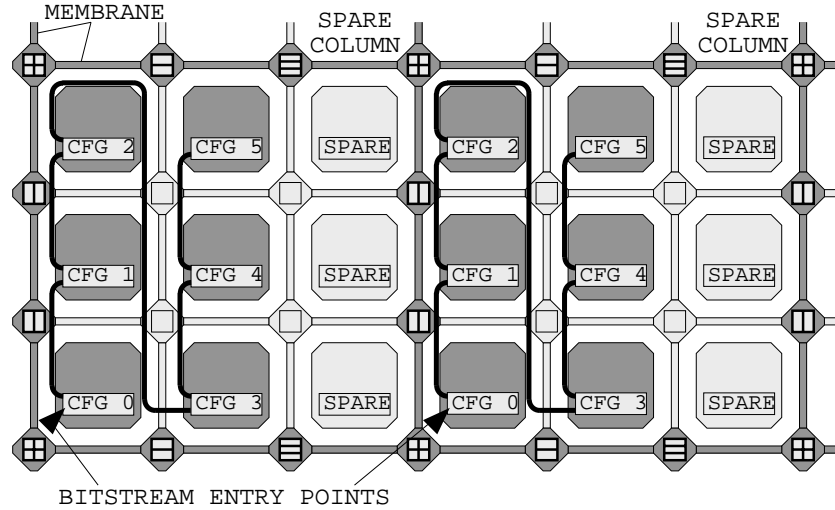


FIG. 2.6: The membrane can define the frequency and placement of spare columns

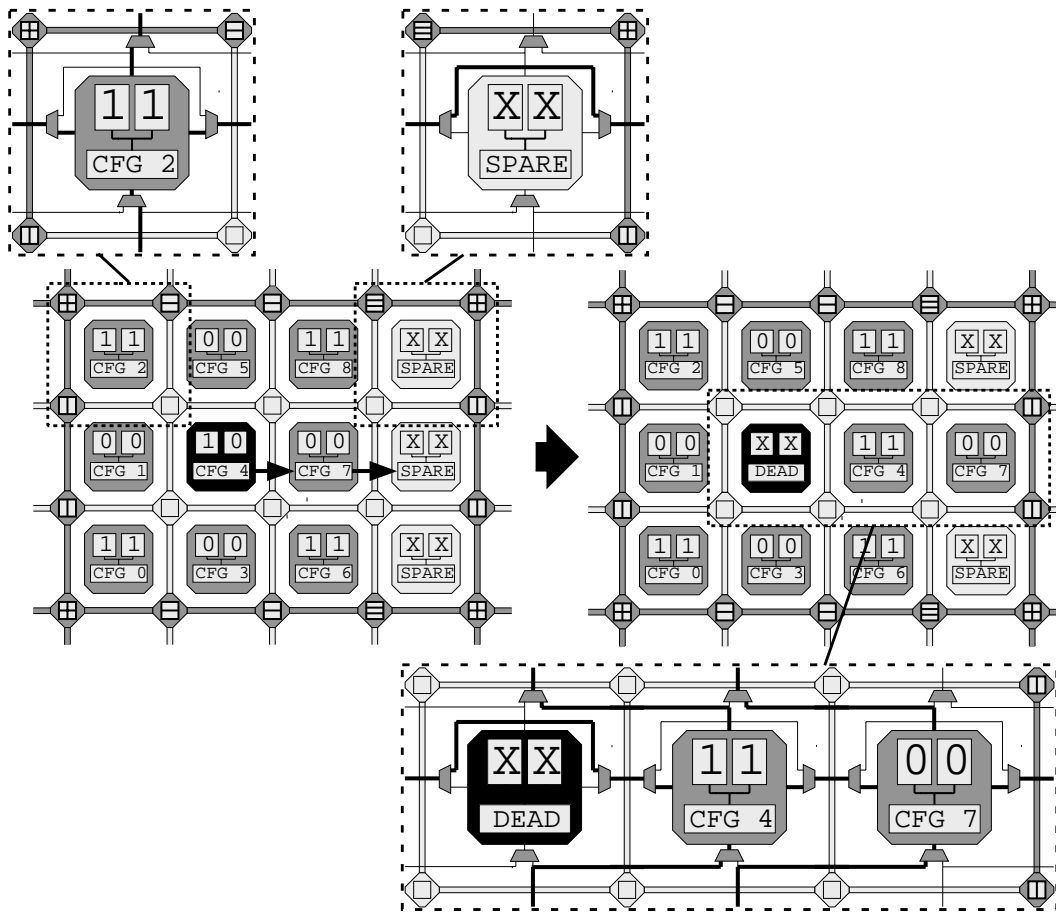


FIG. 2.7: The information stored in a faulty element and in its right-hand neighbors is shifted until a spare column is reached

Our mechanism for repairing faults (Figure 2.7) relies on the reconfiguration of the network through the replacement of the faulty element by its right-hand neighbor : the configuration of the faulty element, together with the value stored in its flip-flop, are shifted into the neighbor. The configuration of the neighbor will itself be shifted to the right, and so on until a spare element is reached.

Once the shift is completed, the faulty element “dies” with respect to the network : the connections are rerouted to avoid it, an operation which can be effected very simply by diverting the north-south connections to the right and by rendering the element transparent to the east-west connections. The array, thus reconfigured and rerouted, can then resume executing the application from the same state it held when the fault was detected. When a fault is detected, the FPGA therefore goes off-line for the time required by the reconfiguration, somewhat like an organism becoming incapacitated during an illness.

2.2 Molécule RamMuxTree-SR (biodule 603 avec mémoire)

Une fois les concepts de base en place, il s’est avéré nécessaire de doter les molécules d’une mémoire. Ceci est détaillé dans l’extrait qui suit, tiré de [1].

2.2.1 Memory Organization

The Global Memory

The memory designed for the Embryonics project has neither the features of a conventional random access memory, nor the actual structure of a RAM, it is more like a ROM which provides access to the data stored through a cyclic mechanism. For instance, storing a memory word that is 4-bit wide, the memory array will be an abstract entity, consisting out of 4 basic memory areas. This is where the memory area actually differs in structure when compared with conventional memories. For the following topics we will refer to the overall memory structure as to the global memory and we will use for the actual storing region the term of basic memory area, since this is the fundamental storage area in our organism. These terms are necessary since the global memory might consist of basic memory areas that are actually distributed inside the cell and not placed together like in the example shown in figure 2.8.

The global memory is shown inside the electronic organism in figure 2.8 inside the red rectangle as an example of a global memory consisting of four basic memory areas. This means that the genetic program's words are 4-bit wide. The blue rectangle delimits the basic memory area, described in the next paragraph.

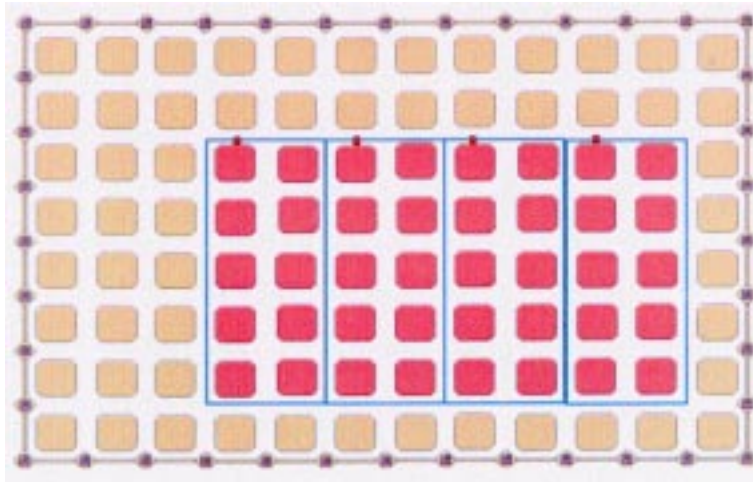


FIG. 2.8: A global memory inside the cell

Because the basic memory area continuously shifts its stored data, the output data being available at the top left element of each basic memory area, from the 4 data outputs coming from the 4 basic memory areas we will get a 4-bit wide information. This is why our global memory area also differs in functionality from conventional memories. For the reason presented above, the organization of the information inside the basic memory area is following a vertical direction rather than the more usual horizontal one. Details about the implementation are given in chapter 2.2.2. The memory areas shown in figure 2.8 have the same number of molecules because they are supposed, in this example, to store words from the same genetic program. Otherwise, the number of molecules may differ.

The way the memory is organized in the Embryonics project resembles to the biological way in the manner that instead of having a memory storing all the needed information, our cell has several memory areas and the molecules, in turn, concurrently do the same activity :

they allow accessing the stored data by continuously shifting the information. This is strongly similar with the biological parallelism that is present in any given cell.

A very poetic sight, using a top-down point of view for our electronic organism, would be that our memory architecture is behaving like the DNA : the information is coded -inside DNA by chemical components, inside our memory bit by bit - and the DNA has the structure of a chain while our memory shows the same similarity through the manner the memory molecules connect with each other. The DNA (our memory) provides the genetic information - the genome (or, in our case, the genetic program) - and ensures, this way, the living process (the functionality) of the cell - biological or electronic one, part of which process is shown in figure 2.9.

As we can see in the figure 2.9, the information which flows from one MuxTree memory molecule to another one is following a circle-like path, starting with the bottom-left element, going upside to the top and then repeating the process until the right-most column. Here information is routed so as to reach again the bottom-left element.

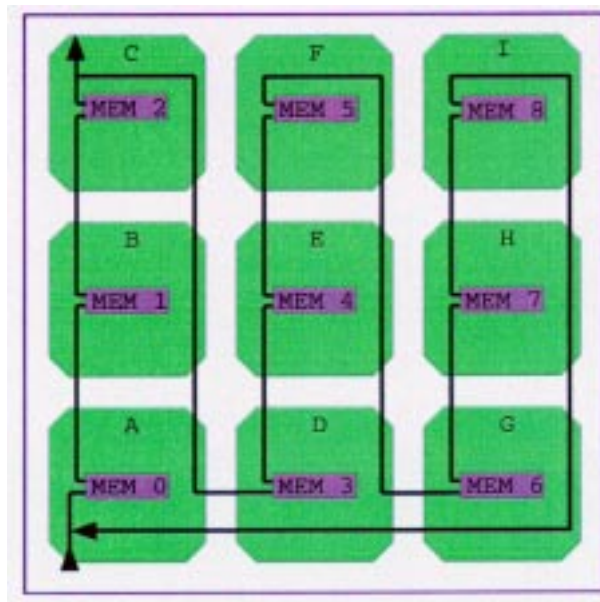


FIG. 2.9: Shifting information within the basic memory

We already now the mystery of life also includes the massive parallelism of processes inside a being, together with connectivity and extreme redundancy. In the following topics we will elaborate on the memory architecture and on the overall flexibility achieved in the design.

The Basic Memory

As we can see in the figure 2.10, the basic memory area is, from a certain point of view, actually very similar- in structure - with the cell itself, it being a rectangular area inside the another rectangular area denoting the cell. Because of the similarity between them, the ground for re-using the connections that are used for a normal, non-memory region, is already established. All that is needed is a mechanism, in a way similar with the cellular automata which builds the cellular membrane. This mechanism will be described in chapter 2.2.2.

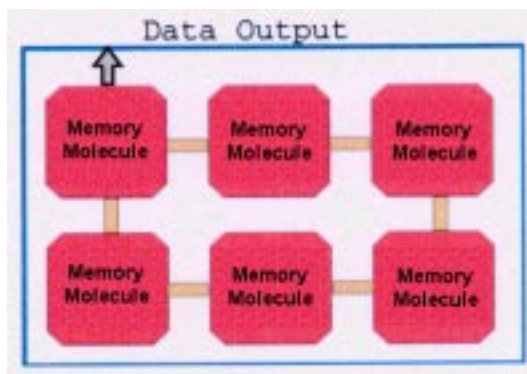


FIG. 2.10: A basic memory area. Zoomed view from figure 2.8

The information stored in the memory has to be continuously shifted. This ensures that at every clock cycle the memory array has at its data outputs one complete memory word. The words stored inside the memory are therefore output at the data-out connections and can be rerouted wherever they are needed using the normal, non-memory operating molecules.

At this moment the reader should have the basic background into the theoretical concepts and similarities over the memory design incorporated into the MuxTree cells. The conceptual background was covered; the next chapter will deal with the implementation issues.

2.2.2 The configuration register

Data Routing

The configuration register, together with the additional logic which controls the routing of data, has different behavior for different configuration patterns. In figure 2.11 we take a closer look at the configurations necessary for a memory area. They are shown together with the routing logic so the data path can easily be followed.

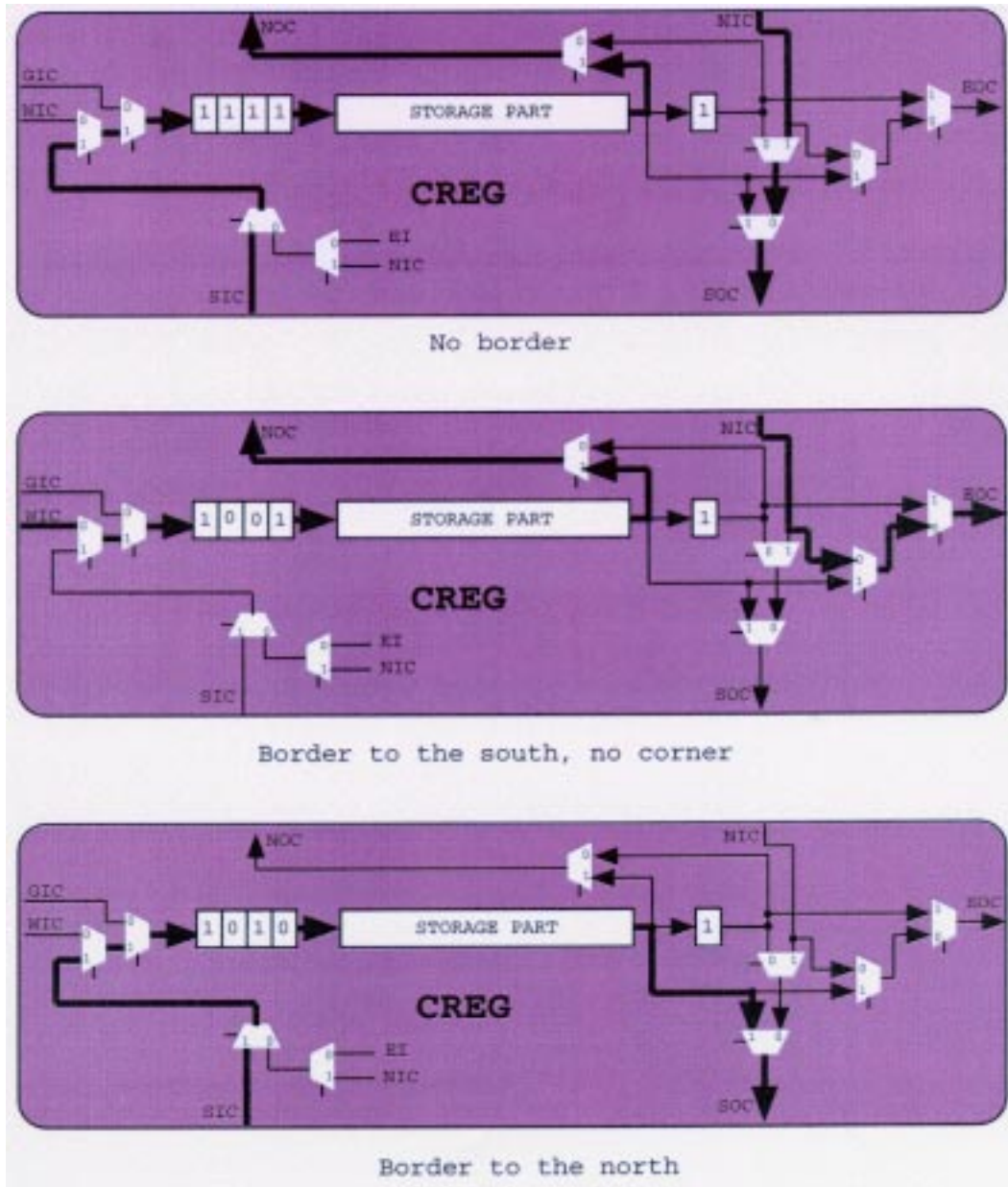


FIG. 2.11: The register configurations

In general, we distinguish three cases :

1. The element is at the northern border. In this case, information enters through the south input connection, it is shifted through the register and exits through the south output connection. In case the element provides also a data output, then data is available to the north output.
2. The element is at the southern border. In this case, information enters through the west input connection, it is shifted through the register and exits through the north output connection. The secondary data path is present here, leading information coming from

the north connection to the east output connection. The cases when the element is a corner in our memory structure are discussed below.

3. The element has no border behavior. In this case, information enters through the south input connection, it is shifted through the register and exits through the north output connection. The secondary data path is present here, leading information coming from the north connection to the south output connection.

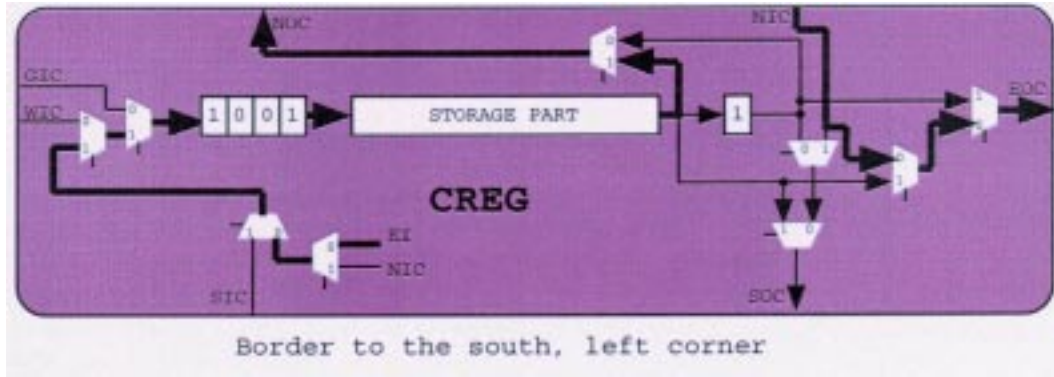


FIG. 2.12: The left corner in the memory structure

In figure 2.12 we can see how information flows inside the element when being the bottom left corner in the memory structure. In this case, information enters through the east input connection, it is shifted through the register and exits through the north output connection. The secondary data path is also present here, and it is identical to that in the case of border to the south.

When the element is a right corner, the difference from the left corner is that data is taken from the north input connection rather from the east input connection as shown in figure 2.13 below.

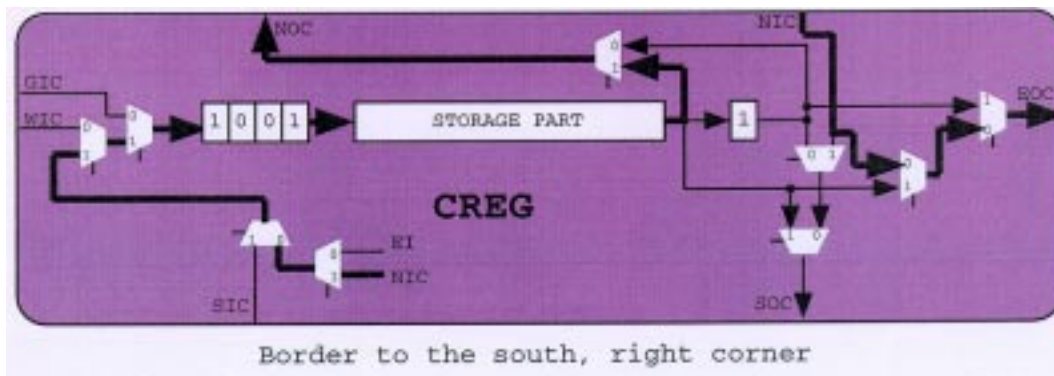


FIG. 2.13: The right corner in the memory structure

A global view for the whole memory structure is provided in figure 2.14. There we can see why the bottom row, corresponding to the southern border, has to have different configurations for the left corner, the right corner and for the regular southern border element. Having all the configurations stored inside the register, the memory structure is now functional.

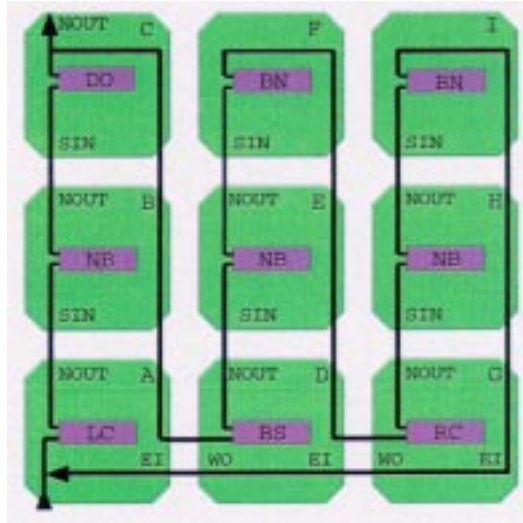


FIG. 2.14: A 3x3 memory area

The memory configuration bits, as they are defined now, only cover 6 possibilities out of possible 8, so there are two spare positions that could be exploited later on. For instance, if the possibility of having memory columns, i.e. memory areas consisting of only one single column, this could be achieved with a minimum of additional logic (the additional logic involved in this modification is just a few more logical gates).

The Register's Operating Modes

The previous design used the entire register to store the configuration. The amount of logic used for routing the data during configuration time was also smaller than in the current design, all of these being shown in figure 2.15 below :

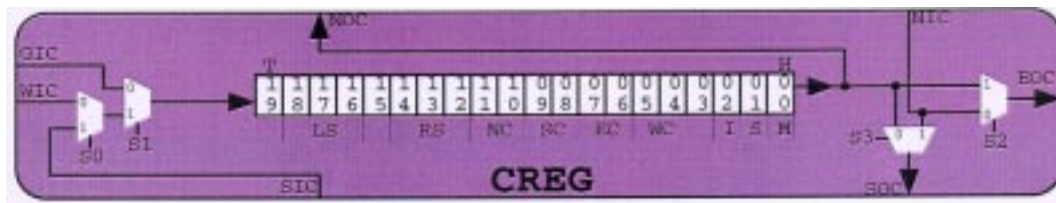


FIG. 2.15: The previous register

The modifications that were made in this project enabled the more flexible using of the configuration register. With the price of adding one bit in the register and a small amount of additional logic, we are now able to use the register in two operating modes for a total of three different possibilities :

- non-memory register
- memory register, 16-bit-wide
- memory register, 8-bit-storage + 8-bit-configuration for the switch block

2.3 Réseaux de molécules et Biodule 601 MicTree

La molécule 603 étant capable de reproduire plusieurs comportements différents en fonction d'une chaîne de configuration, il devient possible de construire des circuits plus complexes, en assemblant plusieurs de ces molécules en réseau. Nous obtenons ainsi un circuit doué d'auto-réplication et d'auto-réparation.

L'étape suivante consiste à concevoir un véritable microprocesseur sur ce principe, capable d'exécuter des programmes élaborés. Un prototype fonctionnel de ce circuit existe, il s'agit du biodule 601 (MicTree).

Les biodules 601 peuvent à nouveau être assemblés en réseaux de processeurs, qui peuvent coopérer et exécuter des tâches en parallèle. Le fonctionnement de la cellule 601 est décrit dans [4].

De tels réseaux, dont la taille moléculaire globale pourrait atteindre plusieurs milliers voire dizaine de milliers d'éléments, seraient à même d'exécuter des tâches de haut niveau tout en étant extrêmement tolérants aux pannes et capable de s'autotester, s'auto-réparer et s'auto-répliquer.

Le premier de ces réseaux complexes, que l'on appelle volontiers "organismes", pourrait être la *Biowatch 2001*, une "simple" montre tolérante aux pannes.

Chapitre 3

Développement

3.1 Approche et architecture

3.1.1 Simulons le matériel!

La ligne directrice du projet est de simuler le plus fidèlement possible la version matérielle de la molécule 603 RamMuxTree-SR. Cependant, certaines concessions sont nécessaires.

Afin de simplifier le design général du moteur de simulation et d'augmenter les performances de celui-ci, nous avons décidé de ne pas tenir compte des parties autotest et autoréparation du système, qui ne présentent de toute manière que peu d'intérêt, puisque le but est de simuler le fonctionnement de grands réseaux fonctionnels avant tout.

Dans cette optique, et selon les références présentées dans le chapitre 2, ce sont les éléments suivants qui devront être reproduits, débarrassés de leurs parties de test et de réparation :

- l'automate cellulaire, responsable de la croissance de la membrane et donc de la structure du réseau ;
- le registre de configuration, dans ses deux versions, fonctionnelle et mémoire ;
- le réseau de connexion "longue distance" et les *switch blocks* ;
- une seule unité fonctionnelle par molécule.

Quelques éléments annexes sont encore nécessaires :

- un équivalent logiciel de la ROM contenant la configuration du réseau ;
- un équivalent logiciel des signaux d'horloge (deux) et de réinitialisation.

A ce niveau, il y a deux manières de résoudre les mécanismes de simulation ; soit on décide de réaliser une simulation extérieurement conforme, mais dont le fonctionnement interne ne suit pas la conception originale (par exemple en mémorisant directement le contenu des registres de configuration, sans tenir compte du mouvement des données à travers le réseaux), soit on simule fidèlement les mécanismes internes.

La première approche a l'avantage d'améliorer considérablement les performances de l'ensemble. Cependant, survienne une modification dans la conception de la molécule 603 et il devient très difficile de reporter celle-ci dans le moteur de simulation, puisqu'il n'est fidèle qu'extérieurement.

Cette contrainte énorme suffit à nous convaincre d'opter pour la seconde solution, dans laquelle une modification du concept se traduira sans doute en grande partie par l'équivalent logiciel d'un "recâblage".

3.1.2 La puissance du concept "orienté objet"

La développement se faisant en C++, nous avons largement tiré profit des possibilités "orienté objet" du langage. En fait, la structure même du réseau homogène de molécules se prête bien à une telle approche :

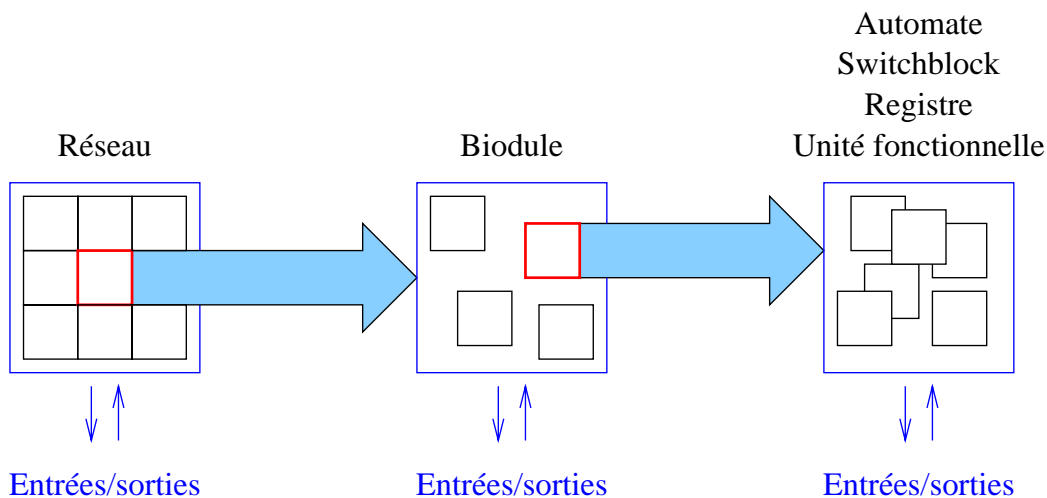


FIG. 3.1: Structure d'un réseau de molécules 603

On peut considérer le réseau, la molécule et ses éléments comme des boîtes noires avec leurs entrées/sorties, à des échelles différentes. Tout est dit dans cette phrase, puisque nous avons effectivement développé deux types d'objets de base, qui permettent la manipulation de "boîtes noires" et d' "entrées/sorties". Toutes les autres classes utiles au simulateur sont dérivées de ces classes élémentaires.

La première classe simple, celle qui gère les entrées/sorties, s'appelle `TBit`. Son rôle est d'une part de mémoriser un bit d'information (0 ou 1) et d'autre part, de gérer un nombre quelconque de liens (ou connexions) avec d'autres objets de la même classe. Nous pouvons ainsi réaliser une sorte de câblage virtuel. Cette classe a été voulue puissante : la modification de la valeur de bit associée à un objet de type `TBit` entrainera la modification des valeurs associées aux autres objets `TBit` liés à celui-ci. Nous verrons également que combiné à un objet de type "boîte noire", un `TBit` peut entraîner une action particulière au sein de celle-ci lors d'une modification de sa valeur de bit associée.

Afin de faciliter la gestion des connexions entre objets de type `TBit`, une classe `TBitList` a été conçue ; celle-ci implémente simplement une liste à chaînage double d'éléments `TBit`, avec possibilités de recherche d'un élément selon certains critères.

La deuxième classe simple importante est celle qui gère les "boîtes noires". Il s'agit de la classe `TBaseElement`. Celle-ci possède essentiellement trois composantes : deux éléments de type `TBitList` qui permettent la manipulation d'entrées et de sorties, et deux méthodes virtuelles (donc entièrement redéfinissables par une classe fille) ; La méthode `Compute()` peut être appelée de l'extérieur pour réaliser une action propre, par exemple recalculer l'état interne en fonction des entrées et produire le résultat sur les sorties. La méthode `ExRout()` est exécutée systématiquement lorsque l'une des entrées de la "boîte noire" change d'état.

Ceci en place, nous allons aisément construire les classes dont nous avons besoin pour

réaliser le simulateur proprement dit. Elles sont au nombre de neuf :

- TConfReg : le registre de configuration simple (sans mode mémoire) ;
- TRamReg : le registre de configuration avec mode mémoire ;
- TFonctUnit : l'unité fonctionnelle ;
- TSwitchBlock : le *switch block* ;
- TAutomaton : élément de la membrane cellulaire ;
- T603a : molécule 603 simple ;
- T603b : molécule 603 avec mémoire ;
- Teeeprom : élément contenant la configuration du réseau ;
- TMuxnet : réseau de molécules.

Toutes ces classes sont dérivées de la classe TBaseElement.

Les classes T603a et T603b contiennent tout naturellement des éléments de type TConfReg, TRamReg, TFonctUnit, TSwitchBlock et TAutomaton (quatre).

La classe Teeeprom reçoit la configuration du réseau tant au niveau de la membrane cellulaire que des registres. Elle est munie à cet effet des deux méthodes LoadCaSeq() (Chargement de la séquence de configuration de la membrane cellulaire) et LoadCfgSeq() (chargement de la séquence de configuration des registres). Pendant la simulation, une machine d'états, implémentée dans la méthode Compute(), va successivement transmettre les séquences au réseau, puis réaliser un *reset* de manière à ce que les valeurs par défaut des bascules (dans les unités fonctionnelles des molécules) soient chargées avec leur valeur initiale. La machine passe ensuite en attente d'un éventuel signal de *reset* global (le *reset* global réinitialise tout le réseau, registres de configuration et élément ttfamily Teeeprom compris, alors qu'un *reset* simple ne réinitialise que les unités fonctionnelles). Elle est contrôlée par le signal d'horloge rapide (CCK).

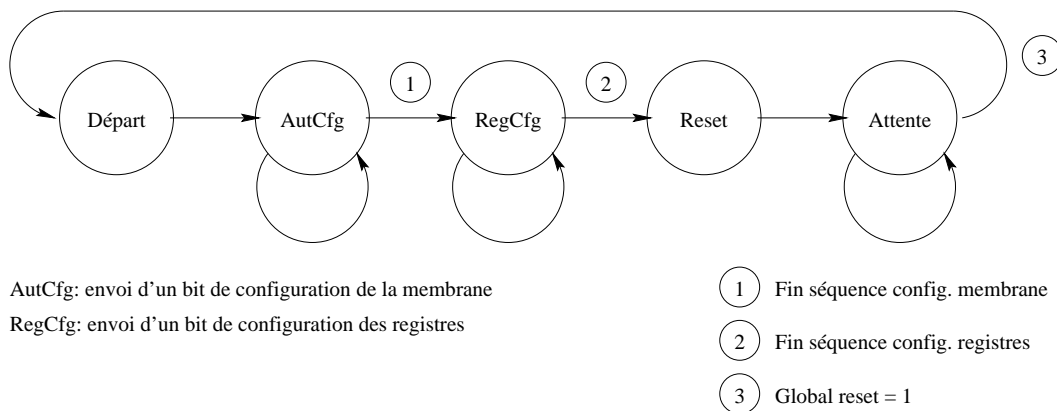


FIG. 3.2: Automate contenu dans Teeeprom

La classe TMuxnet encapsule une matrice d'éléments de type t603a ou t603b de taille totalement paramétrable à l'initialisation ; l'allocation en mémoire des molécules se fait dynamiquement. Les connexions entre molécules voisines sont automatiquement réalisées, ainsi

que les connexions à certains signaux au niveau global, tel que les signaux de *reset*, les deux signaux d'horloge FCK et CCK, le point d'entrée global de la séquence de configuration des registres. *TMuxnet* encapsule également un élément *Teeprom*.

Enfin, *TMuxnet* est responsable de la gestion des points d'arrêts. Ceci est exposé en détails dans le paragraphe 3.1.3.

Nous possédons maintenant une modélisation statique détaillée et flexible du réseau de molécules.

3.1.3 Aspects dynamique

Comment donner vie à ce réseau ? Les problèmes qui se posent sont loin d'être triviaux ; en effet :

- il s'agit d'un système massivement parallèle ; or, à moins d'envisager un noyau multitâche à temps partagé très coûteux tant en puissance qu'en développement, il est impossible de simuler parallèlement les éléments du réseau ;
- le réseau réel est contrôlé par deux signaux d'horloge indépendants de fréquences différentes. Le point précédent implique qu'il ne sera pas possible de gérer ces signaux "parallèlement" ;
- certaines molécules réalisent des fonctions séquentielles (tant mieux pour nous, car contrôlables à intervalle régulier) et d'autres des fonctions combinatoires asynchrones (donc incontrôlables à intervalle régulier) ;
- comme si cela ne suffisait pas, le système de routage à longue distance permet des rebouchements de signaux et le passage de signaux à travers des molécules qui n'en ont pas l'utilité mais n'en assurent que le routage. Comment simuler la transmission asynchrone d'un signal qui va passer à travers plusieurs, voire plusieurs dizaines de molécules, avant d'atteindre un but donné ?

La première approche envisagée pour résoudre ces problèmes de parallélisme est du type de celles qui sont utilisées très classiquement dans les simulateurs d'automates cellulaires dont l'état courant des cellules dépend de leur état précédent. Il s'agirait ainsi de calculer l'état futur de chaque molécule en fonction de son état courant et de celui de ses voisins, ceci pour chaque coup d'horloge la plus rapide (CCK), puis de basculer tous les états futurs d'un coup.

On voit tout de suite que cette méthode n'est pas applicable telle quelle, puisque d'une part, elle ne résoud pas le problème des signaux asynchrones : ceux-ci traversent en réalité tout le réseau instantanément, alors qu'avec le simulateur, ils ne traverseraient qu'une molécule à chaque coup d'horloge. D'autre part, à cause des rebouchements et du routage, cette méthode n'assure pas que l'on dispose de toutes les informations pour une mise à jour correcte, c'est-à-dire que certains signaux peuvent être dans un état erroné (pas encore mis à jour) et être utilisés pour la mise à jour d'un élément.

Nous nous sommes alors demandé s'il ne serait pas possible d'exploiter la faculté de la classe *TBit* à provoquer l'exécution d'une routine lors du changement de la valeur d'une connexion. Comme la modification se propage de connexion en connexion, on a affaire à une diffusion asynchrone de tous les signaux. Le problème est de savoir alors à quel moment de la diffusion les signaux sont tous à jour et l'état futur atteint. De plus, avec les rebouchements, les routines de calcul peuvent être exécutées plusieurs fois de suite inutilement. L'algorithme

serait donc le suivant : l'envoi d'un flanc montant de l'horloge provoque des modifications qui sont progressivement répercutées à travers tout le réseau. Le prochain état global stable sera l'état futur.

Bien que viable théoriquement, car résolvant tous les problèmes posés, cette méthode n'est de nouveau pas directement applicable ; il faut beaucoup trop de puissance de calcul pour atteindre l'état futur, et ceci inutilement, car chaque molécule est remise à jour plusieurs fois de suite avant d'atteindre un état stable (Sommes-nous d'ailleurs sûr qu'un état stable va être atteint ?)

La solution finalement retenue sera un mélange des deux méthodes déjà explorées. On peut l'illustrer avec un exemple, celui du compteur par quatre, abondamment commenté dans [18] :

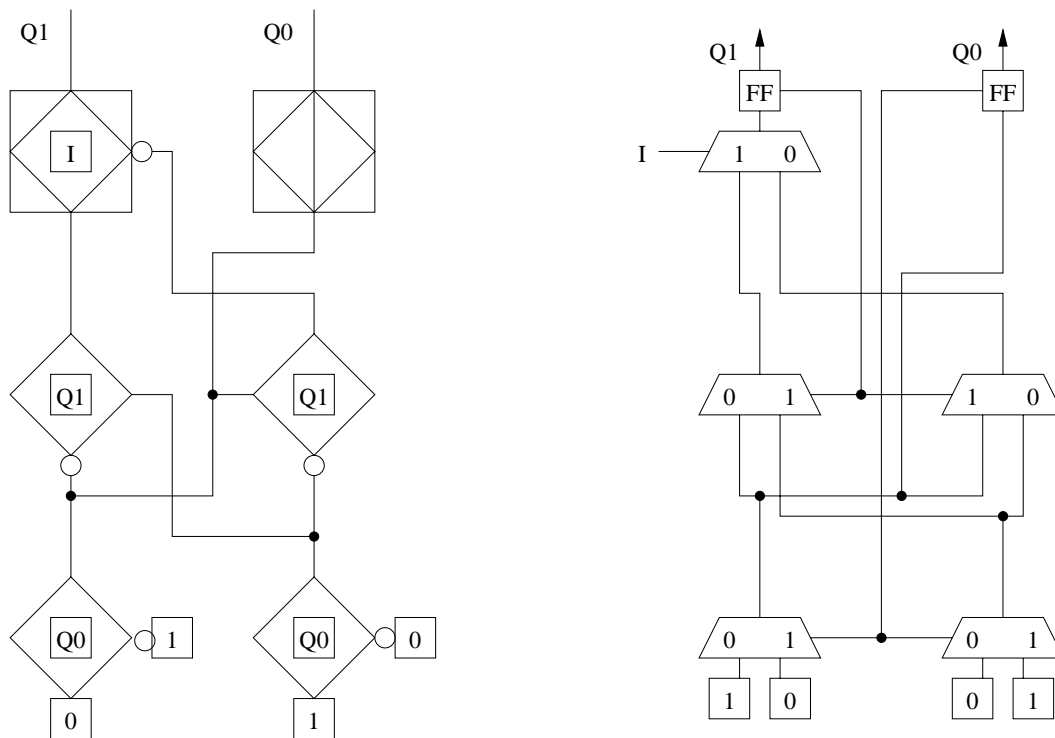


FIG. 3.3: Compteur par quatre

L'idée est de combiner une mise à jour périodique des molécules qui sont configurées de manière séquentielle (c'est-à-dire qui exploitent la bascule) avec une mise à jour asynchrone des autres molécules en essayant de déterminer le cheminement des informations à travers le réseau.

Concrètement, à chaque coup d'horloge, les premières molécules que l'on va mettre à jour sont celles qui exploitent leur bascule. Puis, en suivant les connexions entre molécules, on va suivre le chemin suivi par une information en mettant à jour chaque molécule sur son passage n'exploitant pas leur bascule. Le processus s'arrête lorsqu'on arrive sur une molécule à bascule, puisque celle-ci a déjà été mise à jour au début de l'algorithme :

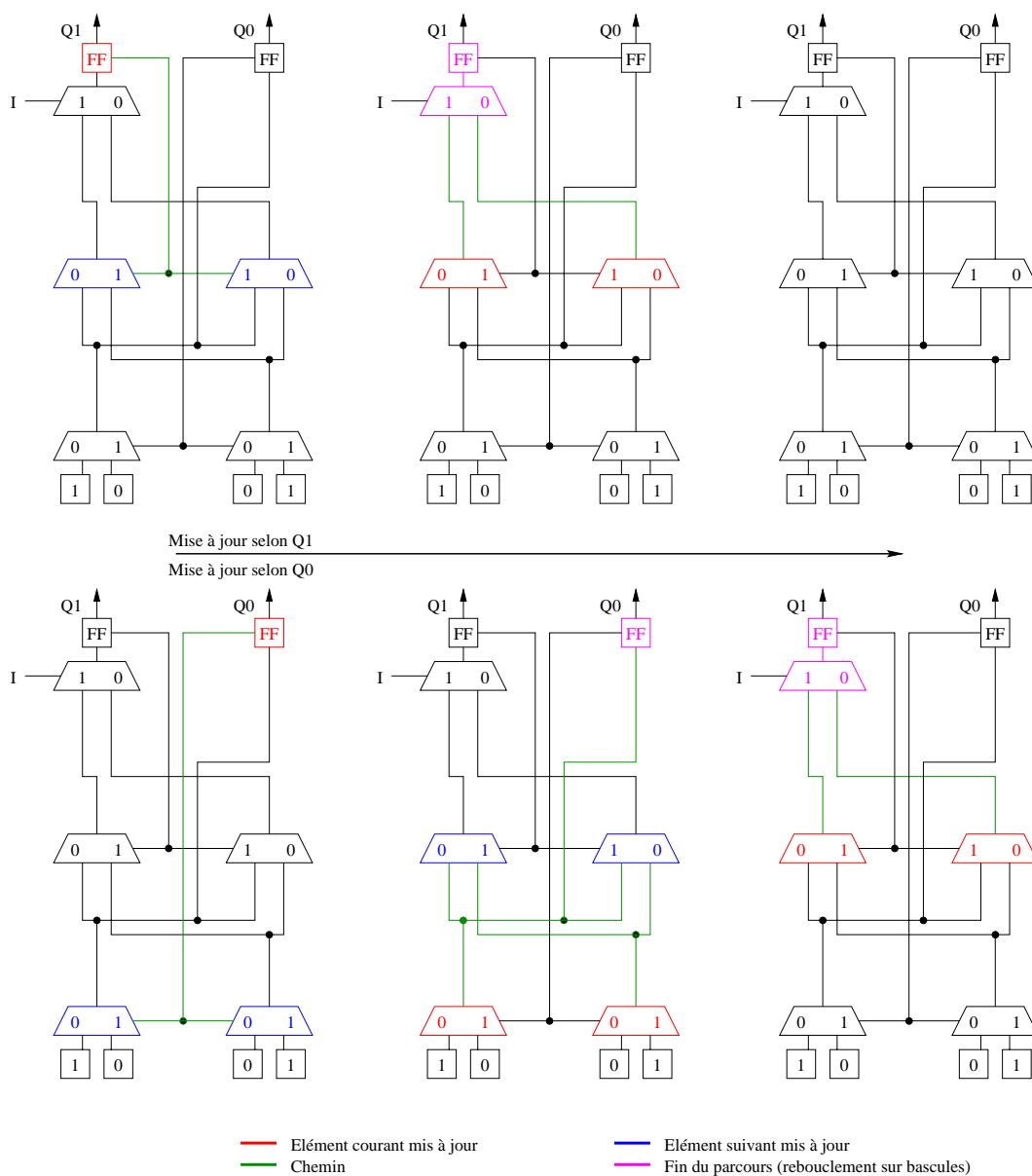
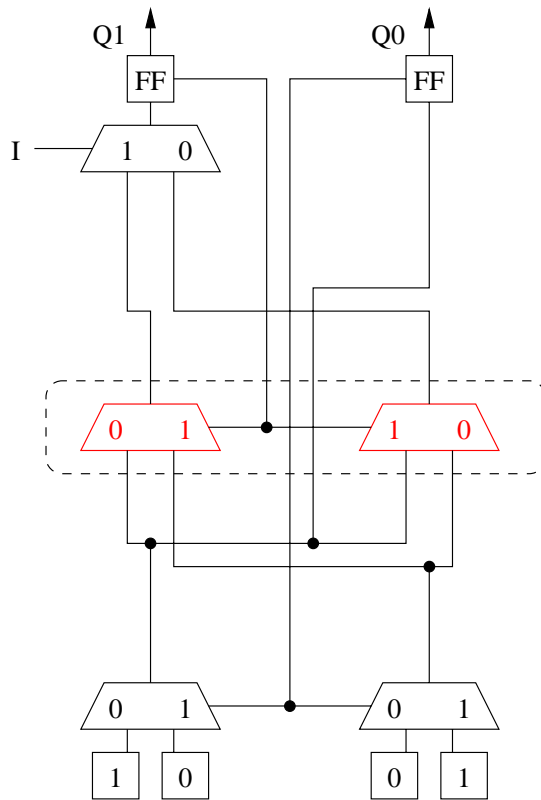


FIG. 3.4: Détermination de l'ordre de mise à jour

Cependant, la plupart des molécules reçoivent des signaux multiples de différentes provenances, qui ont suivi des chemins de longueur différente et il faut en tenir compte si on ne veut pas voir survenir des erreurs de mise à jour :



Si on met à jour les multiplexeurs rouges à l'étape 2, selon le chemin déterminé depuis Q1, ceux-ci ne tiendront pas compte des signaux venant des multiplexeurs inférieurs. Dans le chemin déterminé depuis Q0, les multiplexeurs rouges sont mis à jour dans l'étape 3. C'est donc ce chemin qui doit être retenu.

FIG. 3.5: Cas des molécules recevant des signaux de sources disparates

Cette particularité nous conduit à générer un classement des molécules suivant le moment auquel chacune doit être mise à jour. Dans le cas du compteur par quatre, nous obtenons le classement suivant :

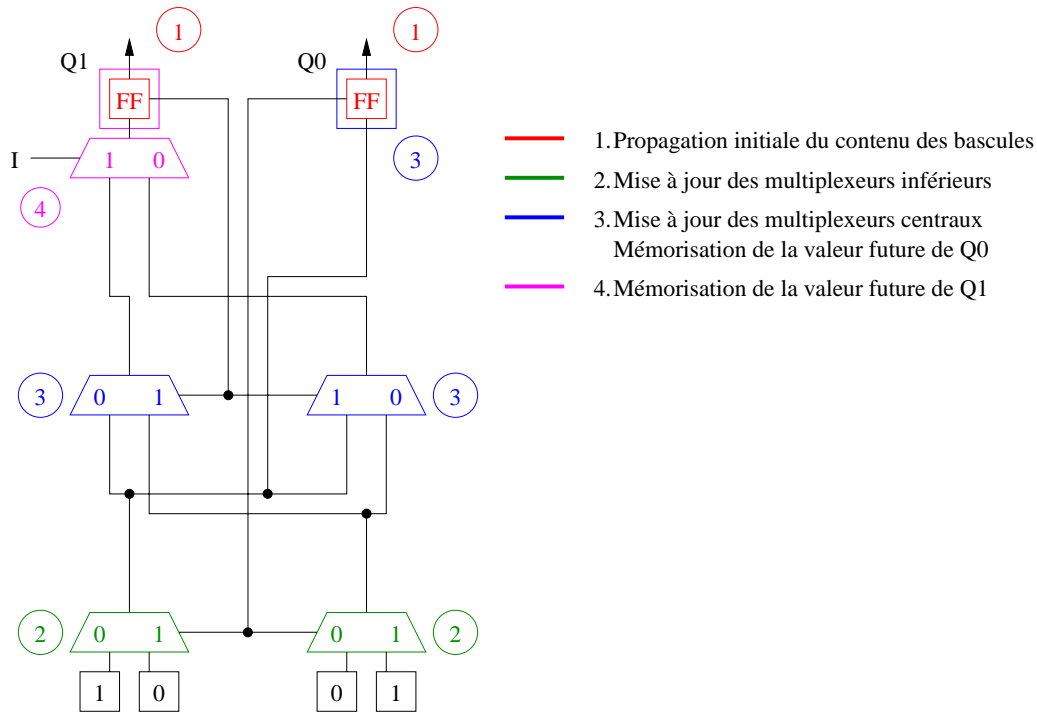


FIG. 3.6: Ordre définitif de mise à jour

Pour obtenir ce résultat, on ajoute dans la classe `t603a` ou `t603b` un index. cet index vaut zéro avant le parcours. Les molécules qui exploitent la bascule prennent une valeur d'index égale à un. A chaque passage par une molécule, la valeur d'index est incremented. Si on passe par une molécule dont l'index est non nul, cela signifie qu'un autre signal passe par cette molécule. Dans ce cas, on compare la valeur courante d'index avec celle mémorisée dans la molécule ; si la valeur courante est plus grande (cela traduit que le chemin suivi est plus long que celui-ci suivi par l'autre signal), on modifie l'index de la molécule. Celle-ci sera donc mise à jour plus tard dans le processus. On prendra bien sûr soin de mettre à jour également les index de toutes les molécules qui suivent sur le même chemin. On obtient ainsi une sorte de classement par couche avec lequel on peut garantir qu'au moment où la molécule sera mise à jour, tous les signaux entrant sont corrects, puisque provenant de molécules elles-mêmes déjà mises à jour.

Etant donné que le routage du réseau est figé à partir du moment où tous les registres de configuration sont chargés, on constate que cette étape d'optimisation peut-être réalisée une fois pour toutes au début de la simulation, au moment où le signal de *reset* est envoyé aux unités fonctionnelles.

Le moteur fonctionne. Il faut y rajouter le concept de points d'arrêt, toujours très utile en simulation. Afin d'uniformiser les mécanismes de point d'arrêt, nous avons rajouté un niveau d'héritage, de manière à ce que les classes `TBit` et `TBaseElement` héritent d'une super-classe commune `TObject`. Celle-ci définit les éléments de base nécessaires pour réaliser les points d'arrêts, et ceux-ci sont gérés dans la classe `TMuxnet`.

Les points d'arrêt sont gérés sous la forme d'une liste dynamique dans laquelle on peut ajouter ou retirer à tout moment un élément. Ces opérations sont effectuées à l'aide des

méthodes `AddBkpt(TObject *tobj, int val)` et `RemBkpt(int no)`. Chaque objet dérivé de `TObject` fournit, sous la forme de la fonction `BkTstElmnt()` virtuelle, l'accès à une de ses variables internes; par exemple, il pourrait s'agir de la valeur du registre de configuration dans la classe `TConfReg`. Chaque élément de la liste contient un pointeur `source` sur l'objet à tester et la valeur de test `value`. Il suffit donc, à chaque mise à jour du réseau, de parcourir cette liste, d'appeler la fonction `BkTstElmnt()` de chaque objet et d'y confronter la valeur de test :

```
reached=0;
for (i=1;i<bkpoints->NbBkpts()+1;i++)
    if (bkpoints->Breakpoint(i)->source->BkTstElmnt()==
        bkpoints->Breakpoint(i)->value)
        reached=i;
```

Cet extrait de la méthode `Compute()` de la classe `TMuxnet` montre la mécanique des points d'arrêts. La variable interne `reached` repère le numéro du point d'arrêt qui a été activé. Le premier porte le numéro un, de manière à ce que le numéro zéro code l'absence de point d'arrêt.

3.2 Résultats et performances

3.2.1 Conforme jusqu'à quel point ?

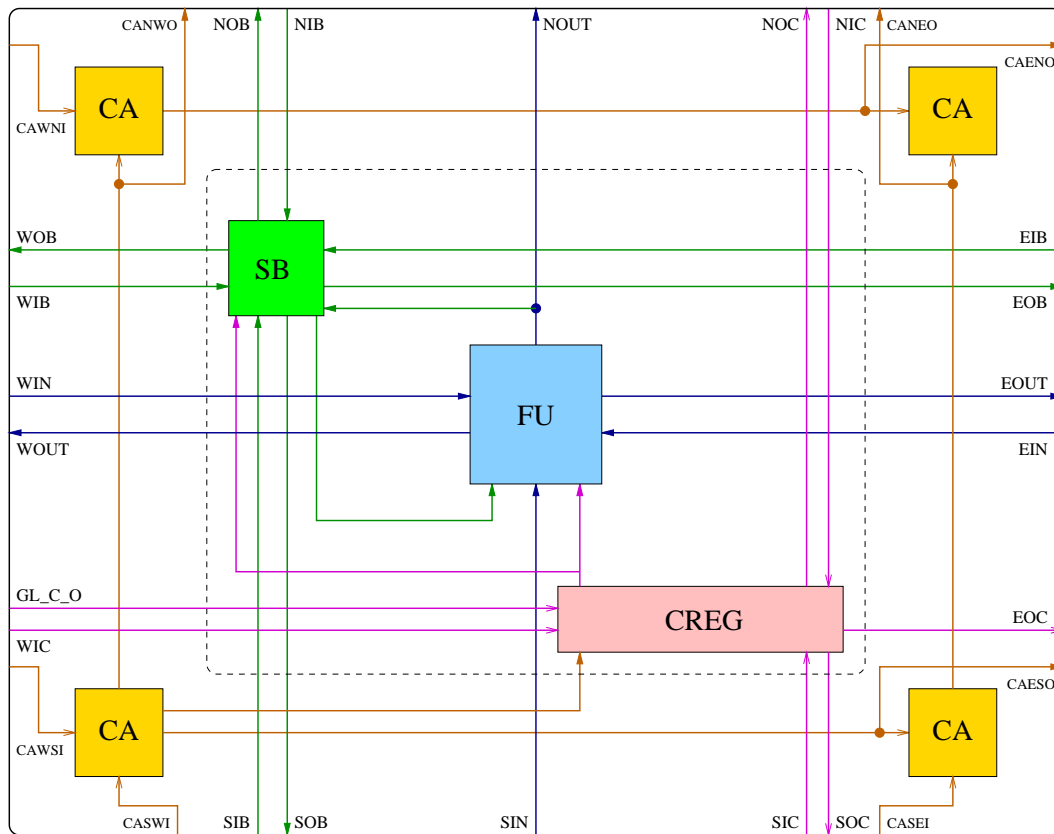


FIG. 3.7: Eléments fidèlement reconstitués

On constate sur la figure 3.7 que ce qui a été reproduit en détails est une version simplifiée de la véritable molécule 603. Par exemple, il n'y a qu'une seule unité fonctionnelle alors qu'en réalité, pour les besoins de l'autotest, la molécule 603 en possède deux. Cependant, la version logicielle du biodule 603 est parfaitement identique sur le plan fonctionnel à la molécule 603 réelle : le comportement de la membrane cellulaire est le même, le chargement du registre de configuration régit le fonctionnement de l'unité fonctionnelle et du *switch block*.

3.2.2 Quelques chiffres

Quelques mesures de performance ont été faites sur la machine qui a servi au développement ; cette machine est le serveur du Laboratoire de Systèmes Logiques, LLSUN, qui est une station de travail Sun UltraSPARC 2 Enterprise, équipée du processeur UltraSPARC-I à 200MHz et de 256Mb de mémoire vive.

L'occupation mémoire est relativement faible et augmente linéairement avec le nombre de molécules :

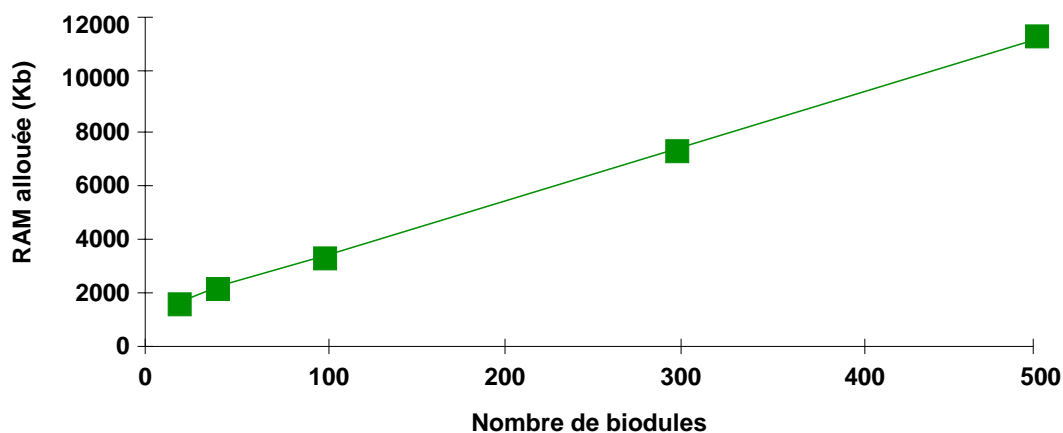


FIG. 3.8: Occupation mémoire en fonction de la taille du réseau

Avec une occupation à vide (sans aucune molécule dans le réseau) de 1200 Kb et l'allocation d'environ 20 Kb par molécule, le moteur permet, sur le type de station envisagé, de simuler facilement des réseaux de plusieurs centaines de molécules : un réseau de 500 molécules par exemple nécessite $1200 + 500 \times 20 = 11.2$ Mb de mémoire, ce qui représente moins de 5% de la mémoire vive totale.

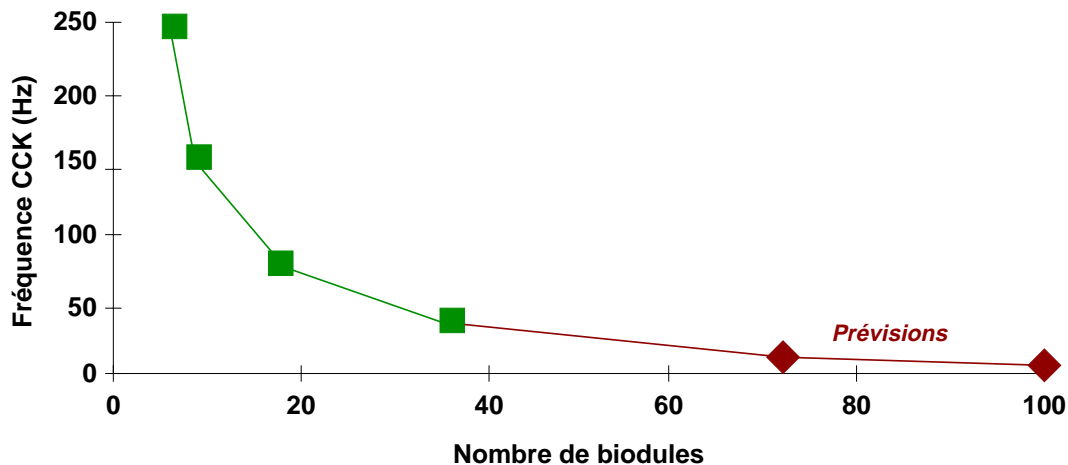


FIG. 3.9: Vitesse d'exécution en fonction de la taille du réseau

Les calculs de performance en vitesse ont été réalisés par chronométrage d'une simulation de mille coups d'horloge CCK. Ceci a permis de déterminer la fréquence approximative de ce signal d'horloge.

Malheureusement, les primitives de mesure de temps ne sont pas très précises ; De plus, il s'agit uniquement du temps de calcul, celui d'affichage n'étant pas compté. Les résultats présentés sont par conséquent optimistes. Enfin, il faut savoir que ces tests n'ont été effectués que pour des réseaux comptant entre 6 et 36 molécules. Au-delà, il s'agit de prévisions obtenues par interpolation.

Il semblerait qu'une fréquence CCK de l'ordre du Hertz soit tout de même atteignable pour des réseaux de plusieurs centaines de molécules.

3.3 Simuler une version future

Nous présentons ici les grandes lignes directrices pour la modification d'une classe existante ou la création d'une nouvelle classe héritée. Les personnes intéressées par un exemple détaillé peuvent se reporter à l'annexe C.

Les modifications ayant lieu sur les fonctionnalités de la molécule et non sur les mécanismes internes au moteur de simulation, il s'agit de porter son attention essentiellement sur le constructeur de l'objet et sur les méthodes `Compute()` et `ExRout()` de l'objet.

En effet, c'est dans le constructeur que l'on va spécifier les interconnexions avec l'extérieur et initialiser d'éventuelles variables internes, comme dans l'exemple ci-dessous, qui n'est rien d'autre que le constructeur de l'unité fonctionnelle :

```
TFunctUnit::TFunctUnit(char *id, TObject *prnt):TBaseElement(id,prnt)
{
    /* Interconnexions en entrée */
    /* Routage */
    AddInput("SIB");
    AddInput("SOB");
    AddInput("EIB");
    AddInput("EOB");
    AddInput("EIN");
    AddInput("WIN");
    AddInput("SIN");

    /* Signaux globaux: reset, horloges */
    AddInput("INIT");
    AddInput("FCK");
    AddInput("CCK");

    /* Bits de configuration */
    AddInput("EB");
    AddInput("PR");
    AddInput("REG");
    AddInput("LSO");
    AddInput("LS1");
    AddInput("LS2");
    AddInput("RSO");
    AddInput("RS1");
    AddInput("RS2");

    /* Interconnexions en sortie */
    /* Routage */
    AddOutput("EOUT");
    AddOutput("WOUT");
    AddOutput("NOUT");

    /* Certaines connexions sont reliees directement en interne */
```

```

Bit("SIN")->AddConnection(Bit("WOUT"));
Bit("SIN")->AddConnection(Bit("EOUT"));

/* Variable utilisee pour la reinitialisation synchrone */
ResetNeeded=0;
}

```

Une fois que l'on a défini les différentes entrées et sorties du module, il faut définir son comportement en interne, en fonction des entrées. Ce comportement, dont la description peut se faire de manière assez empirique, prend place dans les méthodes `Compute()` et `ExRout()` :

```

void TFuncUnit::Compute()
{
/* Valeurs de selection des multiplexeurs a 3 bits */
char ls = Bitv("LS2")*4 + Bitv("LS1")*2 + Bitv("LS0");
char rs = Bitv("RS2")*4 + Bitv("RS1")*2 + Bitv("RS0");

char m0,m1,m2,m3,m4;

/* Si INIT=1, reset synchrone a realiser */
if (Bitv("INIT")==1) { ResetNeeded++; }

/* Selection des signaux d'entree */
switch (ls)
{
case 0: m3=0; break;
case 1: m3=1; break;
case 2: m3=Bitv("SIN"); break;
case 3: m3=Bitv("EIN"); break;
case 4: m3=Bitv("WIN"); break;
case 5: m3=ff_out; break;
case 6: m3=Bitv("SIB"); break;
case 7: m3=Bitv("SOB");
}

switch (rs)
{
case 0: m4=0; break;
case 1: m4=1; break;
case 2: m4=Bitv("SIN"); break;
case 3: m4=Bitv("EIN"); break;
case 4: m4=Bitv("WIN"); break;
case 5: m4=ff_out; break;
case 6: m4=Bitv("SIB"); break;
case 7: m4=Bitv("SOB");
}
}

```

```
if (Bitv("EB")) m1=Bitv("EIB"); else m1=Bitv("EOB");

if (m1) m0=m3; else m0=m4;

/* Utilisation ou non du flip-flop */
if (Bitv("REG"))
{
  if (Bit("FCK")->Toggled()==HLP)
  {
    if (ResetNeeded)
    {
      ff_out=Bitv("PR");
      ResetNeeded=0;
    }
    else ff_out=m0;
  }

  m2 = ff_out;
}
else
  m2=m0;

/* Valeur en sortie */
Bit("NOUT")->Change(m2);
}
```

La méthode `ExRout()` dans ce cas précis consiste en un simple appel à `Compute()`, ce qui signifie que quelque soit le type de modification en entrée (synchrone ou asynchrone), c'est le contenu de `Compute()` qui sera systématiquement exécuté.

Chapitre 4

Limites et extensions possibles

4.1 Limites

L'accent ayant été d'emblée mis sur une programmation modularisée et dynamique au possible, les limites du moteur actuel sont surtout dictées par les capacités en puissance de calcul et en mémoire de la machine hôte. Ces considérations sont détaillées dans le paragraphe 3.2.2.

Certaines limites de réalisme ont été posées afin de ne pas pénaliser inutilement les performances sur la partie fonctionnelle, celle qui nous intéresse le plus. Améliorer le réalisme signifierait en particulier implémenter la machine d'états qui gère l'ensemble de la molécule et ses phases d'autotest et d'auto-réparation ; cela signifierait également réaliser un mécanisme plus fin que celui existant actuellement pour la génération des signaux d'horloge.

Enfin, il faut savoir que l'extension d'une classe existante telle que la classe `T603a` par exemple, ainsi que son utilisation dans le réseau, ne sont pas triviales. Ceci est dû aux mécanismes intrinsèques du langage C++. Un langage comme JAVA est sur ce plan beaucoup plus souple, puisqu'il permet de charger dynamiquement des classes dans un programme. Alors qu'en C++, l'intégration d'une nouvelle extension nécessite une recompilation du programme, JAVA intègre directement les nouvelles classes, immédiatement prêtes à l'emploi.

4.2 Extensions

Ce projet de semestre porte sur le développement d'un moteur de simulation. Pour les besoins de mise au point, une interface extrêmement sommaire en mode texte a été créée (voir annexe B), mais cette interface ne saurait en aucun cas permettre l'exploitation optimale du moteur. Une interface utilisateur conviviale reste à développer.

D'autre part, il serait fort intéressant de combiner le moteur de simulation avec certains outils déjà existant ou en développement. Par exemple, il existe un outil (description dans [3]), permettant de générer la configuration des registres à partir d'un diagramme composés d'éléments pris dans la bibliothèque des symboles usuels pour les différentes fonctions réalisables par une molécule (ces symboles sont présentés dans [18]). Cet outil, en fusionnant avec le moteur, permettrait le développement et la validation de structures sur réseaux de molécules. On pourrait aussi envisager un outil similaire, mais qui parte d'un diagramme de décision binaire pour générer la configuration du réseau et le simuler.

Chapitre 5

Conclusion

La tradition au Laboratoire de Systèmes Logiques a toujours voulu que l'on réalise un prototype matériel des concepts élaborés lors de projets de recherche. Cet objectif a été brillamment atteint lors de la réalisation des biodules 601 et 603.

Cependant, une nouvelle étape doit être franchie en reconstruisant le biodule 601 à l'aide de biodules 603 uniquement, ce qui est l'un des principaux buts fixés par le projet Embryonics. Certains calculs permettent d'affirmer que quelques centaines de biodules 603 sont nécessaires pour réaliser le biodule 601, mais un prototype n'est pas réalisable actuellement à cause de la taille importante des biodules 603.

Peut-être que le moteur de simulation développé lors de ce projet de semestre contribuera à lever cet obstacle en permettant d'abord l'élaboration d'un modèle simulé du biodule 601, ce qui faciliterait la fabrication d'un circuit à haute intégration ayant le comportement cherché, à savoir l'interprétation de micro-programmes avec des facultés d'autotest et d'auto-réparation.

Annexe A

Listes des classes et méthodes

Cette liste est susceptible de modifications ultérieures.

Contenu des fichiers `bkpts.*` :

```
class TObject
{
    TObject(char *id, TObject *prnt);

    /* GetParent: renvoie le contenant */
    TObject *GetParent();

    /* Name: renvoie l'identificateur */
    char *Name();

    /* BkValue: Point a tester dans l'objet */
    virtual int BkTstElmnt();

    /* Rename: renomme l'objet */
    void Rename(char *myname);

    /* ExRout: appelee lors d'une modification d'un bit contenu par l'objet */
    virtual void ExRout();
}

class TBkptList
{
    TBkptList();
    ~TBkptList();

    /* Add: ajoute un element dans la liste */
    int Add(TObject *obj, int val);

    /* Remove: retire un element designe par son nom de la liste */
    int Remove(int no);
}
```

```
/* NbBkpts: renvoie le nb d'elements dans la liste */
int NbBkpts();

/* Breakpoint: renvoie le bkpt numero no */
TBkpt *Breakpoint(int no);
}

/* Structure pour gerer les points d'arrets */
struct TBkpt
{
    TBkpt *previous, *next;
    TObjet *source;
    int value;
}

Contenu des fichiers tbit.* :
class TBit: public TObjet
{
    TBit(TObjet *prnt);
    TBit(TObjet *prnt, char val, char *nm, char dir);
    TBit(TObjet *prnt, char val, char *nm, char dir, TBitList *conns);
    TBit(TObjet *prnt, char val, char *nm, char dir, TBitList *conns, int exec);

    /* Change: change la valeur du bit, propage
       aux connexions, et appelle la fonction compute du parent associe */
    int Change(char val);
    int Change(char val, TBit *prt);

    /* Value: renvoie la valeur courante du bit */
    char Value();

    /* BValue: renvoie la valeur tampon du bit */
    char BValue();

    /* Toggled: renvoie type de flanc provoque */
    int Toggled();

    /* Direction: renvoie le sens du bit (entree, sortie) */
    char direction();

    /* GetConnections: renvoie la liste des connexions a ce bit */
    TBitList *GetConnections();

    /* AddConnection: ajoute une connexion */
    int AddConnection(TBit *bit);
}
```

```
/* RemConnection: enleve une connexion */
int RemConnection(char *name);

int RemConnection(int no);

int RemConnection(TBit *ptr);

/* GetConnNb: renvoie le nombre de connexions a ce bit */
int GetConnNb();

/* Flip: bascule le buffer */
void Flip();
}

/* Structure pour gerer les listes de bits */
struct TBitElement
{
    TBitElement *previous, *next;
    TBit        *thisone;
}

class TBitList
{
    TBitList();
    TBitList(TBit *bit);
    ~TBitList();

    /* Add: ajoute un element dans la liste */
    int Add(TBit *bit);

    /* Remove: retire un element designe par son nom de la liste */
    int Remove(char *name);

    int Remove(TBit *ptr);

    /* GetBit: renvoie un objet TBit designe par son nom */
    TBit *GetBit(char *name);

    /* GetBit: renvoie un objet TBit designe par sa position */
    TBit *GetBit(int no);

    TBit *GetBit(TBit *ptr);

    /* NbBits: renvoie le nb de bits dans la liste */
    int NbBits() { return nbbits; }

    /* Connected: TRUE si b est dans la liste */
```

```
int Connected(TBit *b);

/* Clone: cree une copie integrale de la liste */
TBitList *Clone();

private:

/* Methodes de recherche utilisee en interne */
TBitElement *getbyname(char *name);
TBitElement *getbyptr(TBit *ptr);
}

Contenu des fichiers tbase.* :
class TBaseElement: public TObject
{
public:
    TBaseElement(char *id, TObject *prnt);
    TBaseElement(char *id, TObject *prnt, TBitList *inps, TBitList *outps);
    ~TBaseElement();

/* ExRout: a definir. Fonction combinatoire des entrees */
virtual void ExRout() {}

/* Compute: a definir. Fonction combinatoire des entrees */
virtual void Compute() {}

/* Flip: bascule les sorties a l'etat futur */
virtual void Flip();

/* Recompute: A appeler lorsqu'une modif a lieu sur les entrees */
void Recompute();

/* GetInputs: Renvoie la liste des bits d'entree */
TBitList *GetInputs();

/* GetOutputs: Renvoie la liste des bits de sortie */
TBitList *GetOutputs();

/* Bit: renvoie les infos sur le bit de nom name */
TBit *Bit(char *name);

/* Bitv: renvoie la valeur du bit de nom name */
char Bitv(char *name);

/* Mise-a-jour: non utilisee actuellement */
void Refresh();
```

protected:

```
/* AddInput: ajoute une entree */  
void AddInput(char *name);
```

```
void AddEInput(char *name);
```

```
/* AddOutput: ajoute une sortie */  
void AddOutput(char *name);
```

```
/* AddBOutput: ajoute une sortie avec tampon */  
/* Va sans doute disparaître dans le futur */  
void AddBOutput(char *name)
```

```
/* BBitv: renvoie la valeur du bit de nom name (bufoutput) */  
char BBitv(char *name);
```

```
}
```

Contenu du fichier tflipflop.h :

```
class TFlipflop
```

```
{
```

```
    TFlipflop()
```

```
    TFlipflop(char st)
```

```
/* Reset: reset asynchrone */  
Reset();
```

```
/* SyncReset: reset synchrone */  
SyncReset();
```

```
/* Set: bascule a 1 */  
Set();
```

```
/* SyncSet: bascule a 1 synchrone */  
SyncSet();
```

```
/* LoadNS: memorise etat future */  
LoadNS();
```

```
/* Toggle: coup d'horloge */  
Toggle();
```

```
/* SetD: change l'entree de la bascule */  
SetD(char st);
```

```
/* GetQ: renvoie l'etat de la bascule */  
char GetQ();
```

```
}
Contenu des fichiers 603class.* :
class Teeprom: public TBaseElement
{
    Teeprom(char *id, TObject *prnt);
    ~Teeprom();

    /* LoadCaSeq: charge une sequence de config de la membrane */
    void LoadCaSeq(char *seq);

    /* LoadCfgSeq: charge une sequence de config des registres */
    void LoadCfgSeq(char *seq);

    /* GetCaSeq: renvoie la sequence de config membrane en memoire */
    char *GetCaSeq();

    /* GetCfgSeq: renvoie la sequence de config registres en memoire */
    char *GetCfgSeq();

    /* Compute: implemente une machine d'etat pour la diffusion */
    virtual void Compute();

    /* BkTstElmnt: vide, ne permet aucun point d'arret sur cette classe */
    virtual int BkTstElmnt();
}

class TSwitchBlock: public TBaseElement
{
    TSwitchBlock(char *id, TObject *prnt);

    /* Ces methodes implementent le comportement du switchblock */
    virtual void ExRout();
    virtual void Compute();

    /* Ces methodes ne font rien ici */
    virtual void Flip();
    virtual int BkTstElmnt();
}

class TFuncUnit: public TBaseElement
{
    TFuncUnit(char *id, TObject *prnt);

    /* Implemente le comportement de la FU */
    virtual void Compute();
```

```
/* BkTstElmnt: renvoie la valeur en sortie de la FU pour pt d'arret */
virtual int BkTstElmnt();

/* FFEnabled: Renvoie TRUE si la FU utilise sa bascule */
int FFEnabled();
}

class TConfReg: public TBaseElement
{
    TConfReg(char *id, TObject *prnt);

    virtual void Compute();

    /* BkTstElmnt: renvoie la valeur du registre pour pt d'arret */
    virtual int BkTstElmnt();

    /* Registr: renvoie le bit a la position pos du registre */
    int Registr(char pos);

    /* Registr: renvoie la valeur du registre */
    int Registr();
}

class TAutomaton: public TBaseElement
{
    TAutomaton(char *id, TObject *prnt);

    /* Compute: implemente le comportement de l'automate */
    virtual void Compute();

    /* BkTstElmnt: renvoie l'etat de l'automate pour pt d'arret */
    virtual int BkTstElmnt();

    /* State: renvoie l'etat de l'automate */
    int State();
}

class T603a: public TBaseElement
{
    T603a(char *id, TObject *prnt);
    ~T603a();

    /* Registr: renvoie la valeur contenue par le registre du biodule */
    int Registr();

    /* AState: renvoie l'etat de l'automate no (0..3) du biodule */
    char AState(int no);
}
```

```
/* Ces methodes implementent le comportement de l' "enveloppe" biodule */
virtual void Compute();
virtual void Flip();

/* BkTstElmnt: vide, aucun test possible sur l' "enveloppe" biodule */
virtual int BkTstElmnt();
}
```

Contenu des fichiers tmuxnet.* :

```
class TMuxnet: public TBaseElement
{
    TMuxnet(char *id, TObject *prnt, int sx, int sy, int init);
    ~TMuxnet();

    /* Cell: renvoie le biodule aux coordonnees (x,y) */
    T603a * Cell(int x, int y);

    /* Add: ajoute un biodule dans le montage */
    void Add(int x, int y);

    /* Remove: retire un biodule du montage */
    void Remove (int x, int y);

    /* Sizex: renvoie la longueur du montage */
    int Sizex();

    /* Sizey: renvoie la largeur du montage */
    int Sizey();

    /* LoadCaSeq: chargement sequence config membrane dans "ROM" */
    void LoadCaSeq(char *seq);

    /* LoadCfgSeq: chargement sequence config registres dans "ROM" */
    void LoadCfgSeq(char *seq);

    /* Clocks21: Signaux d'horloge a 1 */
    void Clocks21();

    /* Clocks20: Signaux d'horloge a 0 */
    void Clocks20();

    /* SetPace: regle le rapport FCK/CCK. par defaut FCK=1/5*CCK */
    void SetPace(int p);

    /* Reset: Reset global du montage */
    void Reset();
}
```

```
/* Methodes qui implementent le comportement du montage */
virtual void Compute();
virtual void Flip();

/* AddBkpt: ajoute un point d'arret sur obj pour la valeur val */
AddBkpt(TObject *obj, int val);

/* RemBkpt: retire le point d'arret numero no */
RemBkpt(int no);

/* BkReached: renvoie un numero != 0 si un point d'arret a ete atteint */
int BkReached();

/* GetBkpt: renvoie les caracteristiques du point d'arret numero no */
TBkpt *GetBkpt(int no);
}
```

Annexe B

Description de l'interface basique

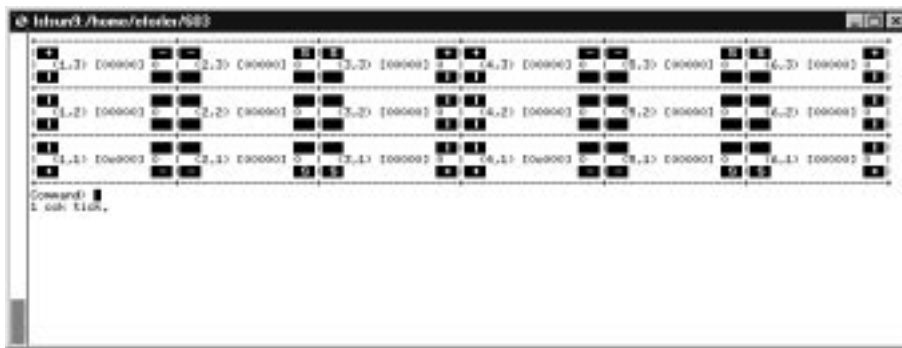


FIG. B.1: Aspect de l'interface

Une mini-interface de commande et de contrôle a été développée pour faciliter les tests du moteur. Au lancement du programme, l'utilisateur entre tout d'abord la taille du réseau en spécifiant un nombre de lignes et de colonnes. Ceci fait, l'interface de la figure B.1 apparaît à l'écran. Cette interface est réalisée en mode texte en utilisant les codes de contrôle VT100.

L'interface comporte deux parties :

- la première partie contient une représentation simplifiée du réseau de molécules ; pour chaque molécule, il est possible de voir l'état de la membrane cellulaire, le contenu du registre de configuration, l'état de sa bascule. Sont également affichées les coordonnées dans le réseau, utiles pour intervenir sur l'état d'une molécule ;
- la deuxième partie est une simple invite de commande.

Un certain nombre de commandes sont reconnues, dont voici la liste :

- commande **add** : ajoute une molécule aux coordonnées spécifiées. Equivaut à une insertion d'un élément dans un vrai réseau ;
- commande **cfg** : affiche le contenu d'un registre de configuration spécifié par les coordonnées de la molécule concernée ;
- commande **out** : affiche la valeur mémorisée par la bascule d'une molécule spécifiée par ses coordonnées dans le réseau ;

- commande **rem** : retire une molécule du réseau, spécifié par ses coordonnées. Equivaut à un retrait pur et simple d'un élément dans un vrai réseau ;
- commande **bit** : affiche les informations sur un bit de donnée d'après son nom et les coordonnées de la molécule rattachée. Il est également possible de modifier la valeur du bit ;
- commande **biv** : affiche la valeur d'un bit de donnée d'après son nom et les coordonnées de la molécule rattachée ;
- commande **see** : réaffiche le réseau de molécules ;
- commande **q** : quitte le programme ;
- commande **new** : reconstruit le réseau en entier ;
- commande **cck** : provoque un flanc montant sur le signal d'horloge CCK (horloge rapide) ;
- commande **fck** : provoque un flanc montant sur le signal d'horloge FCK (horloge lente). Réalise également un appel multiple (cinq par défaut) à la commande **cck** ;
- commande **tst** : force la mise à jour d'une molécule spécifiée par ses coordonnées ;
- commande **stp** : équivalente à la commande **cck** avec un décompte de temps pour les mesures de performance ;
- commande **run** : lance la simulation du réseau. Cette simulation ne peut être interrompue que par un point d'arrêt ou par un *control-break* au clavier.

Il n'est pas possible de placer des points d'arrêts avec cette interface. Les points d'arrêts utilisés lors des tests ont été placés directement par programmation dans le fichier source `test.cc`.

Annexe C

Nouvelle molécule, mode d'emploi

C.1 Structure globale

Pour réaliser un nouvel élément, il faut partir de la classe `TBaseElement`, qui contient les mécanismes de base permettant l'interconnexion et l'évolution de l'élément. Notre nouvelle classe sera donc définie en C++ par l'entête suivant :

```
class NouvelElement: public TBaseElement
```

Le constructeur de la classe est un élément primordial. C'est dans cette partie en effet que l'on va définir les entrées/sorties, d'éventuelles interconnexions entre bits et initialiser les variables internes. L'entête du constructeur doit prendre la forme suivante :

```
NouvelElement::NouvelElement(char *id, TObject *prnt):TBaseElement(id,prnt)
```

En effet, tout objet dans le réseau doit être identifié par une chaîne de caractères `id`. Attention cependant, cette chaîne ne fait pas office d'identificateur absolu (deux objets peuvent porter le même nom); cette possibilité a été prévue essentiellement pour faciliter la vie de l'utilisateur (le simulateur utilise des pointeurs sur les objets pour l'identification, mais il est plus simple de retenir un nom).

Par ailleurs, tout objet du réseau doit être rattaché à un objet parent `prnt` dont il dépend; par exemple, une unité fonctionnelle incluse dans une molécule dépend de celle-ci. Au niveau le plus élevé, les molécules dépendent du réseau, et le réseau ne dépend d'aucun élément (`NULL`).

Dans le constructeur même, la déclaration des entrées/sorties prend les formes :

```
AddInput(ident);
```

et

```
AddOutput(ident);
```

où `ident` est une chaîne de caractère identifiante répondant aux mêmes critères que précédemment. De plus, il est possible de connecter directement des entrées/sorties entre elles :

```
Bit(ident1)->AddConnection(Bit(ident2));
```

Dans l'exemple ci-dessus, `ident1` et `ident2` sont connectés, ce qui signifie que une modification d'état sur l'un des deux bits provoque automatiquement la même modification sur l'autre. Attention, **aucun test n'a été prévu pour empêcher la connexion de deux entrées entre elles**, ce qui dans la réalité, peut conduire à des catastrophes.

La connexion est valable dans les deux sens, c'est-à-dire que le simulateur comprend que la liaison `ident2->ident1` existe. Enfin, il faut savoir que **rien n'a été prévu pour détecter les connexions cycliques du type `ident1->ident2->ident3->ident1`, qui sont invalides**. Il faudra donc y prendre garde ; ainsi :

```
Bit(ident1)->AddConnection(Bit(ident2));
Bit(ident2)->AddConnection(Bit(ident3));
Bit(ident1)->AddConnection(Bit(ident3));
```

est totalement faux. La troisième ligne provoque une récurrence infinie et n'est pas nécessaire, car si `ident1` est connecté à `ident2` et `ident2` est connecté à `ident3`, le simulateur comprend automatiquement que `ident1` est connecté à `ident3`.

Ayant mis en place les entrées/sorties, nous devons encore définir le comportement de l'élément. Il y a trois cas à distinguer : le comportement synchrone, le comportement asynchrone et les valeurs de test pour un point d'arrêt.

La dernière partie est la plus simple à mettre en place. Ceci se réalise via la méthode virtuelle `BkTstElmnt()` qui renvoie une valeur de type `int` :

```
virtual int BkTstElmnt()
{
    return registr;
}
```

Cet exemple est la méthode de test du registre de configuration `TConfReg` : elle renvoie simplement la valeur courante du registre contenue dans la variable interne `registr`. Ainsi, si on place un point d'arrêt sur le registre de configuration, ce sera toujours la valeur du registre qui sera testée.

Le comportement proprement dit de l'élément est défini dans les méthodes `Compute()` (comportement synchrone) et `ExRout()` (comportement asynchrone). Ceci signifie en fait que la méthode `Compute()` est visible de l'extérieur, donc appellable par exemple dans une boucle qui simulerait des coups d'horloge successifs. La méthode `ExRout()` n'est pas appellable de l'extérieur, mais dépend des entrées définies dans le constructeur de l'élément ; chaque fois qu'un bit d'entrée est modifié, la méthode `ExRout()` est automatiquement exécutée. Dans la pratique, on utilisera surtout l'appel à `ExRout()`, car une modification synchrone peut très bien être simulée en définissant un bit d'entrée "clock" et en modifiant son état à intervalle régulier.

C.2 Manipulation des entrées/sorties

Deux méthodes permettent de consulter et modifier l'état des bits d'entrée/sortie :

- la méthode `char Bitv(char *name)` renvoie la valeur du bit d'identifiant `name` ;
- la méthode `TBit *Bit(char *name)` renvoie le bit d'identifiant `name`.

De plus, pour un objet de type `TBit`, nous disposons également des méthodes :

- `int Change(char val)` qui change la valeur du bit à `val` ;
- `int Toggled()` qui renvoie le type d'impulsion lors du dernier changement (flanc montant HLP, descendant LLP, ou aucun changement NOP).

Reprenons l'exemple du registre de configuration simple :

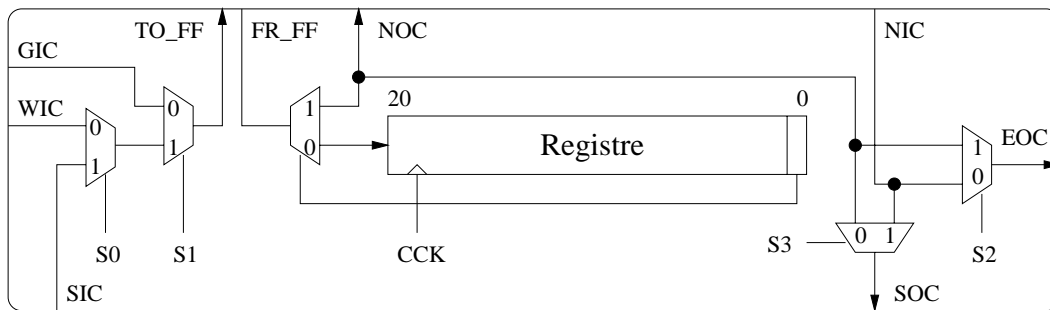


FIG. C.1: Implémentation du registre de configuration

Nous obtenons le code C++ suivant :

```
void TConfReg::Compute()
{
```

Tout d'abord, on détermine quel bit, GIC, WIC ou SIC doit être pris en compte en entrée, en fonction des valeurs de S0 et de S1 :

```
char s=Bitv("S0")+Bitv("S1")*2,so1,so2,bit,r0;
```

```
switch (s)
{
  case 0:
  case 1: so1=Bitv("GIC"); break;
  case 2: so1=Bitv("WIC"); break;
  case 3: so1=Bitv("SIC");
}
```

Cette valeur est transmise en sortie sur la ligne TO_FF, et la ligne FR_FF est récupérée en entrée :

```
Bit("TO_FF")->Change(so1);
```

```
bit=Bitv("FR_FF");
```

Tant que le bit 0 du registre vaut 0, le registre est rempli par la valeur en entrée, à chaque coup d'horloge :

```
if (!Registr(0))
{
  if (Bit("CCK")->Toggled()==HLP) Shift(bit);
}
```

Sinon, le bit d'entrée est redirigé vers les sorties adéquates, en fonction des valeurs de S2 et S3 :

```
else
{
  r0=bit;
```

```
    Bit("NOC")->Change(r0);

    if (Bitv("S3"))
        Bit("SOC")->Change(Bitv("NIC"));
    else
        Bit("SOC")->Change(r0);

    if (Bitv("S2"))
        Bit("EOC")->Change(r0);
    else
        Bit("EOC")->Change(Bitv("NIC"));
}
}
```

La description en C++ du comportement devient relativement intuitive. On se rapproche de la description que l'on pourrait faire en VHDL.

Le cœur du registre de configuration est prêt, il ne reste que quelques détails à rajouter, comme la méthode `shift(int inp)` qui implémente le registre à décalage, dont nous faisons grâce au lecteur.

Table des figures

2.1	The structure of a MuxTree element	3
2.2	The membrane builder is a very simple cellular automaton capable of partitioning the CA space into blocks of identical size	5
2.3	The membrane builder is inserted among the MuxTree elements	6
2.4	The membrane is used to direct the configuration bitstream	6
2.5	The self-test logic for the programmable function in a MuxTree element	8
2.6	The membrane can define the frequency and placement of spare columns	9
2.7	The information stored in a faulty element and in its right-hand neighbors is shifted until a spare column is reached	10
2.8	A global memory inside the cell	11
2.9	Shifting information within the basic memory	12
2.10	A basic memory area. Zoomed view from figure 2.8	13
2.11	The register configurations	14
2.12	The left corner in the memory structure	15
2.13	The right corner in the memory structure	15
2.14	A 3x3 memory area	16
2.15	The previous register	16
2.16	The new configuration register	17
3.1	Structure d'un réseau de molécules 603	20
3.2	Automate contenu dans Teeprom	21
3.3	Compteur par quatre	23
3.4	Détermination de l'ordre de mise à jour	24
3.5	Cas des molécules recevant des signaux de sources disparates	25
3.6	Ordre définitif de mise à jour	26
3.7	Eléments fidèlement reconstitués	28
3.8	Occupation mémoire en fonction de la taille du réseau	29
3.9	Vitesse d'exécution en fonction de la taille du réseau	29
B.1	Aspect de l'interface	44
C.1	Implémentation du registre de configuration	48

Bibliographie

- [1] Lucian Prodan. *Embryonics : development of the Biowatch 2001, final report*, rapport interne. Laboratoire de Systèmes Logiques, EPFL, 1998.
- [2] Lucian Prodan. *RamMuxTree-SR : a tutorial*, rapport interne. Laboratoire de Systèmes Logiques, EPFL, 1999.
- [3] Pascal Felber. *BDD-oriented FPGA editor*, Technical report. Laboratoire de Systèmes Logiques, EPFL, 1994.
- [4] Gianluca Tempesti, Daniel Mange, André Stauffer. *The Embryonics project : a machine made of artificial cells*. Laboratoire de Systèmes Logiques, EPFL, 1998.
- [5] M. Abramovici, M. A. Breuer, A. D. Friedman [1990]. *Digital Systems Testing and Testable Design*. Computer Science Press, New York, 1990.
- [6] M. Abramovici, C. Stroud [1995]. *No-overhead BIST for FPGAs*. In Proc. 1st IEEE International On-Line Testing Workshop, pp. 90-92, 1995.
- [7] F. Hanchek, S. Dutt [1998]. *Methodologies for Tolerating Cell and Interconnect Faults in FPGAs*. IEEE Transactions on Computers, v. 47, n. 1, January 1998.
- [8] W.K. Huang, F. Lombardi [1996]. *An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays*. IEEE VLSI Test Symposium, 1996.
- [9] J. Lach, W.H. Mangione-Smith, M. Potkonjak [1998]. *Efficiently Supporting Fault-Tolerance in FPGAs*. Proc. FPGA 98, Monterey, CA, pp. 105-115, February 1998.
- [10] C. G. Langton [1984]. *Self-Reproduction in Cellular Automata*. Physica 10D, pp.135-144, 1984.
- [11] R. Negrini, M. G. Sami, R. Stefanelli [1989]. *Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays*. The MIT Press, Cambridge, MA, 1989.
- [12] J.-Y. Perrier, M. Sipper, J. Zahnd [1996]. *Toward a Viable, Self-Reproducing Universal Computer*. Physica 97D, pp.335-352, 1996.
- [13] C. Stroud, S. Konala, M. Abramovici [1996]. *Using ILA testing for BIST in FPGAs*. Proc. 2nd IEEE International On-Line Testing Workshop, Biarritz, July 1996.
- [14] A. Stauffer [1997]. *Membrane building and binary decision machine implementation*. Technical Report 247, Computer Science Department, EPFL, Lausanne, 1997.
- [15] G. Tempesti [1995]. *A New Self-Reproducing Cellular Automaton Capable of Construction and Computation*. Proc. 3rd European Conference on Artificial Life, Lecture Notes in Artificial Intelligence, 929, Springer Verlag, Berlin, pp. 555-563, 1995.
- [16] G. Tempesti, D. Mange, A. Stauffer [1997]. *A Robust Multiplexer-Based FPGA Inspired by Biological Systems*. Journal of Systems Architecture : Special Issue on Dependable Parallel Computer Systems, EUROMICRO, 43(10), 1997.

- [17] G. Tempesti, D. Mange, A. Stauffer [1998]. *Self-Replicating and Self-repairing Multicellular Automata*. Artificial Life. Accepted.
- [18] G. Tempesti [1998]. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. Ph.D. Thesis, Swiss Federal Institute of Technology, Lausanne, 1998.

Index

A	
Approche	12
B	
Boîte noire	13
C	
Câblage virtuel	13
Chargement dynamique	26
Classe T603a	14, 33
Classe T603b	14
Classe TAutomaton	33
Classe TBaseElement	13, 31
Classe TBitList	13, 29
Classe TBit	13, 29
Classe TBkptList	28
Classe TConfReg	33
Classe Teeeprom	14, 33
Classe TFlipFlop	32
Classe TFonctUnit	33
Classe TMuxNet	14
Classe TMuxnet	35
Classe TObject	28
Classe TSwitchBlock	33
Compteur par quatre	16
D	
Diagramme de décision binaire	26
E	
Embryonics	1
Entrées/sorties	13
M	
Méthodes virtuelles	13
Machine d'états	14
Mise à jour	19
O	
Orienté objet	13
P	
Points d'arrêts	19
R	
Reboucement	15
Routage	15
S	
Structure TBitElement	29
Structure TBkpt	28

Table des matières

1	Prolégomènes	1
1.1	Contexte	1
1.2	Enoncé du projet de semestre	1
1.3	Contraintes	1
1.4	Cadre administratif	2
1.5	Remerciements	2
2	Molécules, cellules, réseaux	3
2.1	Molécule MuxTree-SR (biodule 603)	3
2.1.1	An FPGA for the Embryonics Project : MuxTree	3
2.1.2	Self-Replication	4
2.1.3	Self-Repair	7
2.2	Molécule RamMuxTree-SR (biodule 603 avec mémoire)	10
2.2.1	Memory Organization	11
2.2.2	The configuration register	13
2.3	Réseaux de molécules et Biodule 601 MicTree	18
3	Développement	19
3.1	Approche et architecture	19
3.1.1	Simulons le matériel!	19
3.1.2	La puissance du concept "orienté objet"	20
3.1.3	Aspects dynamique	22
3.2	Résultats et performances	28
3.2.1	Conforme jusqu'à quel point ?	28
3.2.2	Quelques chiffres	28
3.3	Simuler une version future	30
4	Limites et extensions possibles	33
4.1	Limites	33
4.2	Extensions	33
5	Conclusion	34
A	Listes des classes et méthodes	35
B	Description de l'interface basique	44

C Nouvelle molécule, mode d'emploi	46
C.1 Structure globale	46
C.2 Manipulation des entrées/sorties	47
Table des figures	49
Bibliographie	50
Index	52