THE EMBRYONICS PROJECT: SPECIFICATIONS OF THE MUXTREE® FIELD PROGRAMMABLE GATE ARRAY

Lucian Prodan, Daniel Mange, Gianluca Tempesti

1. Introduction

As new techniques for applying biological processes to the development of computer hardware reach maturity, the EMBRYONICS (for *embryonic electronics*) project [4] gains in complexity and flexibility of use. New features, such as the capability to more efficiently store data, [2] have lately been added to MUXTREE® (for *multiplexer tree*) – our electronic molecule.

A side effect to the flexibility added to the new molecule is the need for different binary configurations in order to differentiate the possible operating modes. This document is intended as a summary of the final specifications of MUXTREE® necessary to be able to effectively use the molecules as a programmable circuit. A much more complete description of the circuit is presented in references [1] and [3].

The configuration of a molecule, the *molecular code* or *MolCode*, determines its behavior inside our artificial cells. Section 2 briefly describes the structure of the molecular code. In Sections 3 and 4 we describe all the possible operating modes for a molecule. In Section 5, we illustrate how to actually configure a cellular structure. Section 6 is devoted to a discussion of the latest self-repair features of our system. Finally, Section 7 describes the first physical implementation of our MUXTREE® molecules: the BIODULE 603.

2. The Molecular Code

Each MUXTREE® molecule is configured via a binary sequence, the molecular code (MolCode). This code determines the mode of operation of the molecule, which can function in either the *logic mode* or the *memory mode*.

At configuration time, the molecular code (MolCode) is loaded into the molecule. The MolCode `MC21:0` is stored into two separate components, the flip-flop `FF` (`Q`) and the configuration register (`CREG20:0`), as shown in Figure 1 and Figure 3. The operating mode is selected by the value of bit `M=CREG20`:

- ♦ `M=0` designates the logic mode, in which the user has access to all the combinational logic resources of the molecule;
- ♦ `M=1` designates the memory mode, in which no combinational logic resources are available, with the exception of the long-distance connections (handled through the switch block `SB`) in the short memory mode.

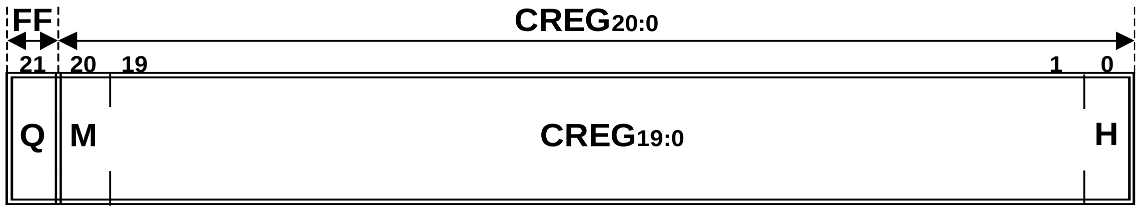


Figure 1 The molecular code (MolCode) `MC21:0`.

3. The Logic Mode (M=0)

3.1. General Description

The logic mode is defined by M=CREG20=0. In this mode, the user has access to all the combinational logic resources of the MUXTREE® molecule.

The structure of the molecular code in logic mode is shown in Figure 2. Let us examine in detail the meaning of each of the MolCode bits, from right to left:

- ♦ Bit H (CREG0) is always set to logic value "1". This bit is used exclusively at configuration time to indicate that the molecule has been completely configured (i.e., that the MolCode is in place within the configuration register).
- ♦ The memory and test bits (MT or CREG3:1) define the internal configuration of the molecule (Figure 3):
  - bit EB (CREG1) is the control variable for multiplexer M1; it selects EOBUS (EB=0) or EIBUS (EB=1) as the test variable;
  - bit R (CREG2) is the control variable for multiplexer M2; it toggles between sequential (R=1) and combinational (R=0) operation;
  - bit P (CREG3) contains the value for the asynchronous preset of the flip-flop FF; at the rising edge of INIT (a global signal), flip-flop FF will perform an asynchronous set (P=1) or reset (P=0).
- ♦ Bits N1:0, S1:0, E1:0, and W1:0 (SB or CREG11:4) control the switch block SB, as shown in Figure 4. These bits store the control variables of the multiplexers charged with propagating signals in all four cardinal directions (north, south, east, and west) through the long-distance connection network. The switch block allows all combinations of interconnections between the inputs and the outputs from the four directions.
- ♦ The connection block bits (CB or CREG19:12) drive the multiplexers which select the inputs to the molecule, as shown in Figure 3:
  - bits LEFT2:0 (CREG18:16) select the input of multiplexer M3;
  - bits RIGHT2:0 (CREG14:12) select the input of multiplexer M4;
  - bits LEFT3 (CREG19) and RIGHT3 (CREG15) are not used in the current implementation and are reserved for further developments.
- ♦ Bit M=0 (CREG20) indicates that the molecule is operating in logic mode.
- ♦ Bit Q (flip-flop FF) is, in practice, not used when in logic mode since, after the molecule has been configured, bit P will override the value stored in flip-flop FF on each high level of the signal INIT.

We will not discuss further the operation of the molecule in logic mode, which has been described in detail in reference [3] (pp. 135-143).

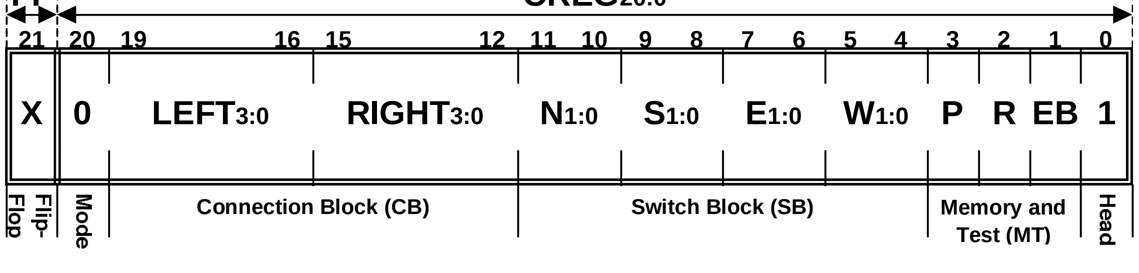


Figure 2 The MolCode (MC21:0) in logic mode.

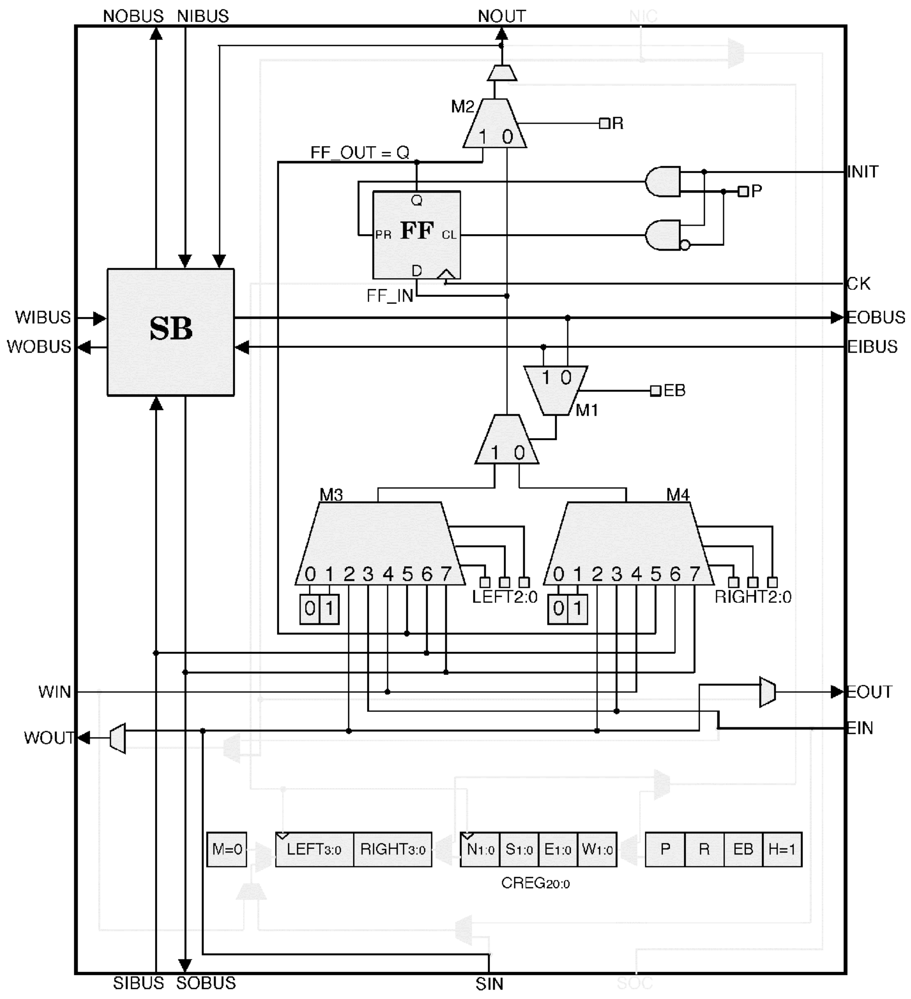


Figure 3 Layout of a MUXTREE® molecule in logic mode.

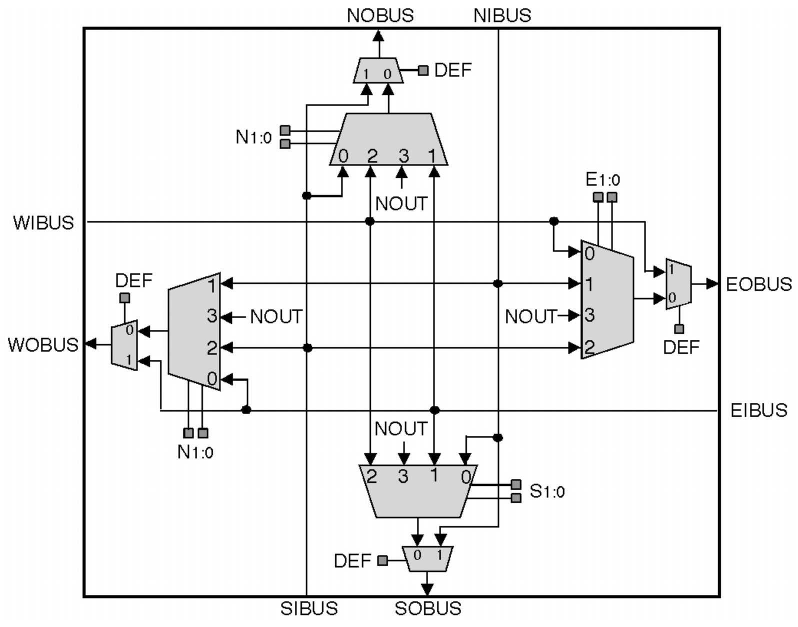


Figure 4 Internal architecture of the switch block SB. [DEF=M•Q=CREG20•Q]

3.2. An Example

Let us consider a simple example of artificial organism, a single cell (Figure 5) realizing a modulo-4 up-down counter, defined by the following sequences:

- for $C=0$ (counting up):

  $Q_1Q_0 = 00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00 \rightarrow \ldots$

- for $C=1$ (counting down):

  $Q_1Q_0 = 00 \rightarrow 11 \rightarrow 10 \rightarrow 01 \rightarrow 00 \rightarrow \ldots$

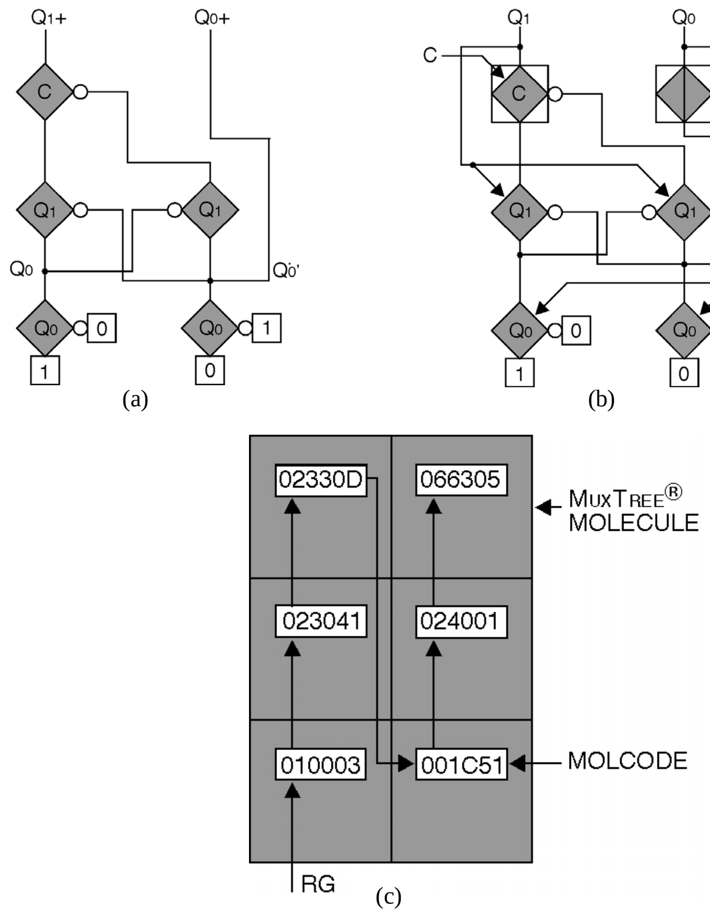


Figure 5 Modulo-4 up-down counter. (a) Ordered binary decision diagrams for Q_1+ and Q_0+. (b) Multiplexer diagram using MUXTREE® molecules. (c) 6-molecule cell. [RG: ribosomic genome]

It can be verified that the two ordered binary decision diagrams Q_1+ and Q_0+ of Figure 5a (where each test element is represented by a diamond with a single output, a “true” input, and a “complemented” input identified by a small circle) represent a possible realization of the counter [3][4]. The leaf elements, represented as squares, define the output values of the given functions (Q_1+ and Q_0+ in the example) defined by the following equations:

$$Q_1+ = C\cdot(Q_1\cdot Q_0+Q_1'\cdot Q_0')+C'\cdot(Q_1\cdot Q_0'+Q_1'\cdot Q_0)$$

$$Q_0+ = Q_0'$$

The direct implementation of the ordered binary decision diagrams on silicon follows the layout from Figure 5b. Each test element is implemented with one MUXTREE® molecule, keeping the same interconnection diagram and assigning the values of the leaf elements to the appropriate multiplexer inputs. The two state functions Q_1 and Q_0 are the outputs of the D flip-flops of the top row of the MUXTREE® molecules.

The cell implementing the counter has therefore 3 rows and 2 columns of MUXTREE® molecules. From the multiplexer diagram of Figure 5b and from the description of the MUXTREE® molecule in logic mode (Figure 3 and Figure 4) we can compute the control bits of each molecular code, finally generating the MolCodes of Figure 5c, each MolCode being a word of six hexadecimal digits (00QM, CREG19:16, CREG15:12, CREG11:8, CREG7:4, and CREG3:0). This string of MolCodes is our *ribosomic genome* RG [3].

4. The Memory Mode (M=1)

4.1. Generalities

The memory mode is defined by M=CREG20=1. In this mode, the active internal resources of the MUXTREE® molecule are the configuration register (CREG), the flip-flop (FF) and, possibly, the switch block (SB). Part of the configuration register is used as a read-only memory, implemented not as a conventional addressable memory, but rather as a *cyclic* or *shift memory*, consisting of a simple storage structure that synchronously shifts its data in a closed circle (Figure 6).

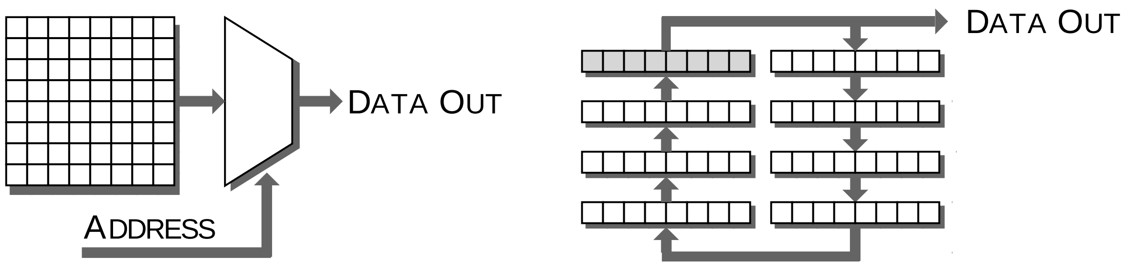


Figure 6 Comparison between addressable (left) and shift (right) memories

Depending on the value of Q=CREG21, we have two memory sub-modes:

- Q=0: *short memory*, storing 8 bits of information, allowing the use of the switch block SB;
- Q=1: *long memory*, storing 16 bits, disabling the switch block SB.

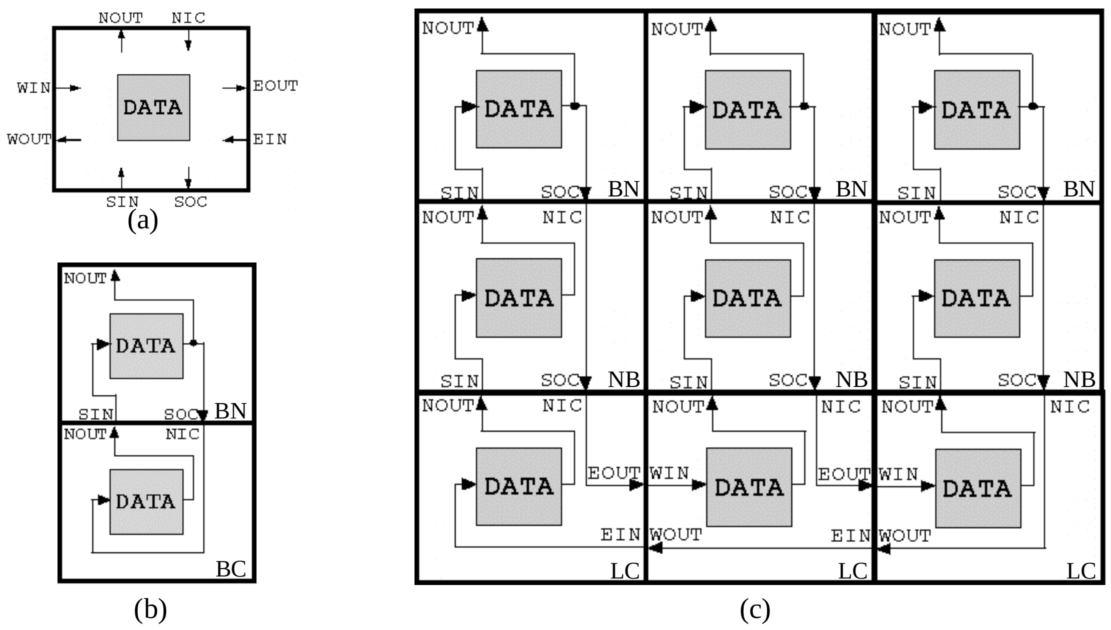


Figure 7 The memory mode. (a) Block schematic of a molecule in memory mode. (b) Macroscopic view of a 2x1 shift-memory structure. (c) Macroscopic view of a 3x3 shift-memory structure.

MEM2	MEM1	MEM0	MOLECULE'S POSITION	
0	0	0	Border to the south	BS
0	0	1	Lower right corner	RC
0	1	0	Lower left corner	LC
0	1	1	Bottom of a single column	BC
1	0	F	Border to the north with data output	BN
1	1	F	No border	NB

Table 1 The possible positions of a molecule within a memory structure and the configuration bits (F = don't care condition).

A MUXTREE® memory molecule is presented as a block schematic in Figure 7a. It uses input and output connections (I/O connections) in the four cardinal directions. The I/O connections are internally routed at configuration time and define the memory structure. Each molecule has to be loaded with a specific molecular code which will strictly determine its behavior, and which is related to its position inside the structure. This code is stored in bits `MEM2:0` (`CREG3:1`), which are special configuration bits common to both memory sub-modes and are shown in Table 1.

The shape of a memory structure is that of a rectangle, containing exclusively MUXTREE® molecules operating in memory mode, whose horizontal and vertical dimensions are defined by the user through the above-mentioned configuration bits. The data stored in the registers are continuously shifted (one bit per period of the functional clock `FCK`) and can be accessed through the `NOUT` output of any molecule situated in the upper row of the structure.

Figure 7b shows the routing of the I/O connections in a 3x3 shift-memory structure. The stored data are shifted as follows: from the bottom left molecule, the data enter serially the molecules situated immediately above (north) in the same column. Once the top molecule is reached, the data are routed to the bottom molecule in the next column to the right (east). Data are then shifted upwards, towards the top molecule, then go to the bottom molecule of the next column and thus the process repeats itself. When the data reach the top right molecule, they are routed to the bottom right molecule and from there to the leftmost bottom molecule, closing the circle.

We implemented the MUXTREE® molecule so as to allow a large choice for the user in specifying the dimensions of a memory structure. The minimal structure is a single column two molecules high (a 2x1 rectangle, as shown in Figure 8a), but there are no upper limits defined for the dimensions of the memory rectangle: they are set by simply configuring the memory molecules with the appropriate MolCodes.

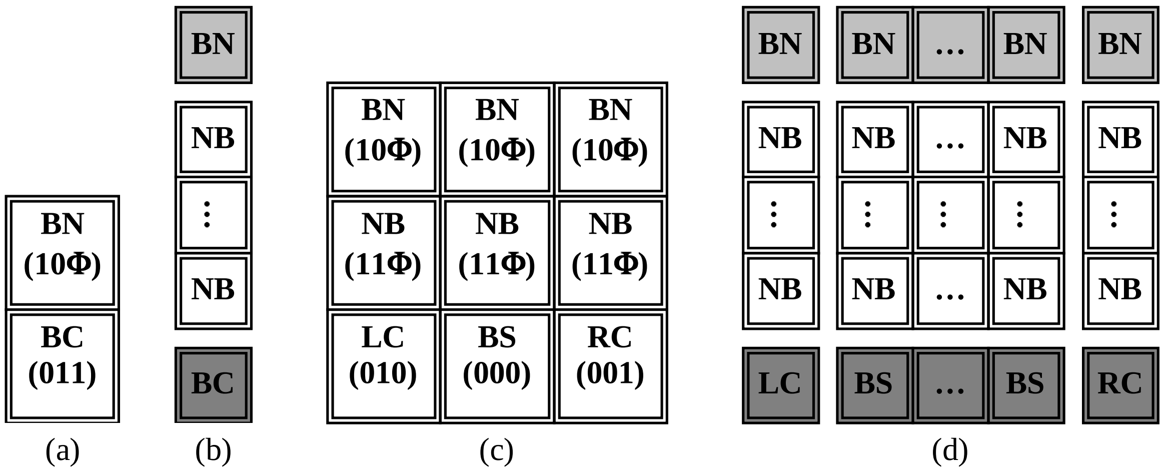


Figure 8 Memory structures and their molecular configurations MEM2:0. (a) The minimal shift-memory structure (2x1). (b) The general structure of a single column shift-memory. (c) A 3x3 shift-memory structure. (d) The general structure of a shift-memory.

The general structure of a single column memory is presented in Figure 8b and the general memory rectangle is shown in Figure 8d. The memory configuration bits (`MEM2:0`) for the structure presented in Figure 7b, respectively Figure 7c, are shown in Figure 8a, respectively Figure 8c. A memory structure cannot be implemented without using the `LC`, `RC`, `BS` and `BN`-type molecules or `BC` and `BN`-type molecules, depending to the type of structure (rectangle or single column). Therefore these molecules appear with a darker color in Figure 8.

The memory structure has data output ports in all molecules marked as `BN`. This means the user has access to the stored data through the `NOUT` output of all `BN`-type molecules. The bit-stream coming out through a data output port starts with the least significant bit stored inside the corresponding molecule (`CREG12` in short memory mode or `CREG4` in long memory mode) and is shifted from the left (most significant bit) to the right (least significant bit) every cycle of the functional clock `FCK`.

4.2. The Short Memory Mode (Q=0)

One of the two memory sub-modes is the short memory with switch block. In this mode, each molecule can store 8 bits of user data while keeping the functionality of the switch block.

In this sub-mode, the MolCode bits are structured as follows (Figure 9):

- ♦ Bit `H` (`CREG0`) is always set to logic value "`1`". This indicates that the current molecule was loaded with the MolCode.
- ♦ The molecule's position bits (`MEM2:0=CREG3:1`) were described in Subsection 4.1. and shown in Table 1.
- ♦ Bits `N1:0`, `S1:0`, `E1:0` and `W1:0` (`CREG11:4`) have the same attributes as in logic mode (Subsection 3.1).
- ♦ Bits `DATA7:0` (`CREG19:12`) store the user data.
- ♦ Bit `M=1` (`CREG20`) indicates the memory mode.
- ♦ Bit `Q=0` (flip-flop `FF`) indicates the short memory sub-mode.

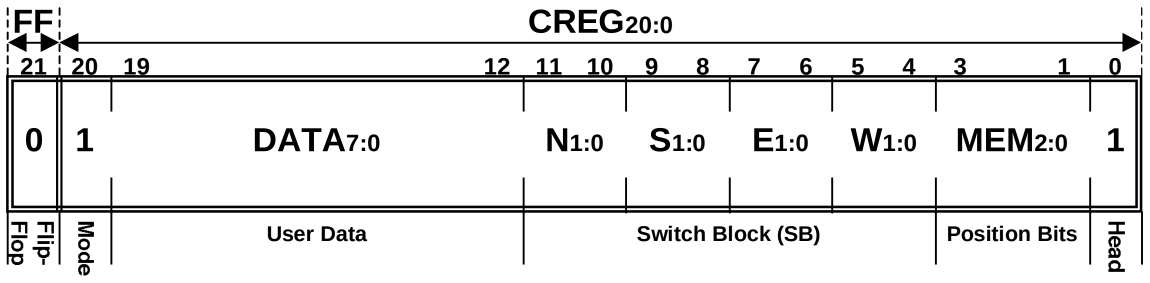


Figure 9 The short memory MolCode $MC_{21:0}$.

The possibility of storing 8-bit-wide words of user data, while keeping the routing features of the switch block, can be an advantage in connection-heavy implementations. The active parts of the molecule in this case are the switch block `SB`, the configuration register `CREG` and the flip-flop `FF`, all shown in Figure 10 (the parts that are not available to the user are drawn in light gray).

The data path is shown in Figure 10. The configuration register includes the storage part of the molecule (`CREG19:12=DATA7:0`). Bits are shifted at each `FCK` clock cycle: the user data from the previous molecule enter through the `SIN`, `EIN` or `WIN` connections, are shifted through `CREG19:12`, and then exit the molecule through the `NOUT` and/or `SOC` connections, or can be routed through any of the long-distance buses via the switch block (by setting the switch block's configuration bits to the appropriate value). From a macroscopic view, the data flow is shown in Figure 7b, depending on the molecule's position, i.e. the value of `MEM2:0`.

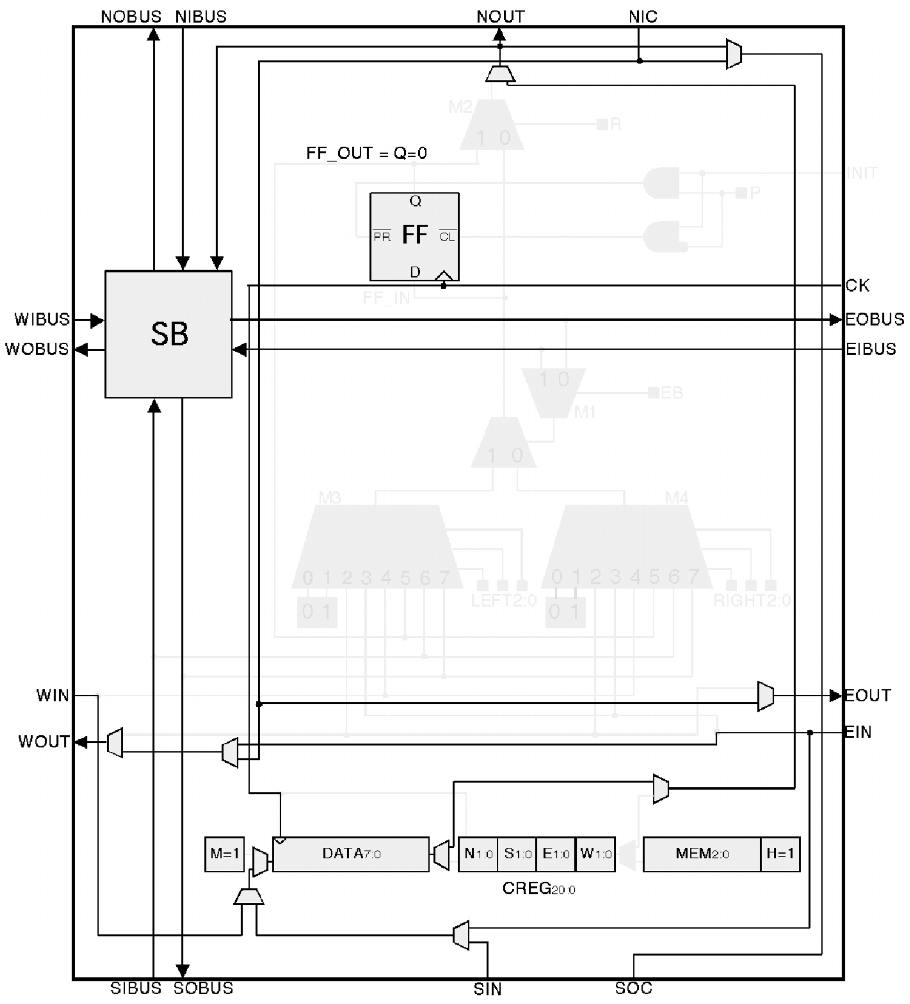


Figure 10 Layout of a MUXTREE® molecule in short memory mode.

4.3. The Long Memory Mode (`Q=1`)

The other memory sub-mode is the long memory mode. In this mode, each molecule can store 16 bits of user data, at the price of losing the functionality of the switch block (that is, no external signals can be routed through a memory structure, except, as we will see, straight horizontal and vertical connections.

In this sub-mode, the MolCode bits are structured as follows (Figure 11):

- Bit `H` (`CREG0`) is always set to logic value “`1`”. This indicates that the current molecule was loaded with the MolCode.
- The molecule’s position bits (`MEM2:0=CREG3:1`) were described in Subsection 4.1 and shown in Table 1.
- Bits `DATA15:0` (`CREG19:4`) store the user data.
- Bit `M=1` (`CREG20`) indicates the memory mode.
- Bit `Q=1` (flip-flop `FF`) indicates the long memory sub-mode.

The possibility of storing 16-bit-wide words of user data doubles the storage capacity of the short memory mode. There is, however, a disadvantage: because bits `CREG11:4` that normally define the operation of the switch block are now used to store user data, the switch block is disabled. To avoid losing all of the bus routing facilities, we designed the `SB` to operate also in long memory mode, where it simply implements a pass-through: `SIBUS` is directly connected to `NOBUS`, `NIBUS` to `SOBUS`, `EIBUS` to `WOBUS`, and `WIBUS` to `EOBUS`.

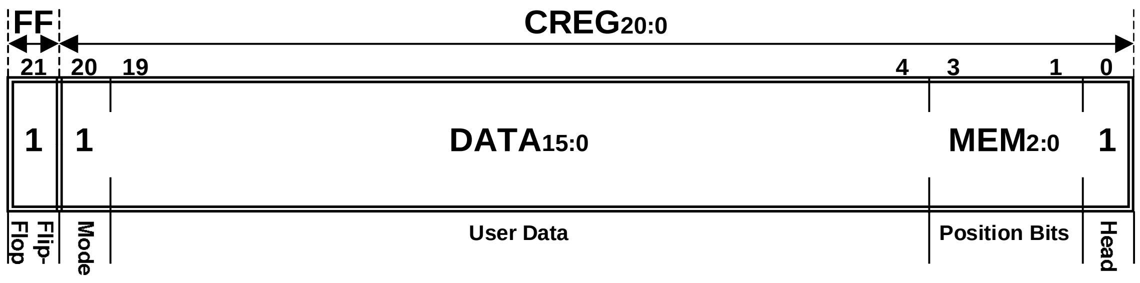


Figure 11 The long memory MolCode $MC_{21:0}$.

This sub-mode allows greater storage capacity at the expense of flexibility. The active parts of the molecule in this sub-mode are the configuration register `CREG` and the flip-flop `FF`, all shown in Figure 12 (the parts that are not available to the user are drawn in light gray).

The data path is shown in Figure 12. The configuration register includes the memory (`CREG19:4=DATA15:0`). Bits are shifted each `FCK` clock cycle: the user data from the previous molecule enter through the `SIN`, `EIN` or `WIN` connections, are shifted through `CREG19:4` and then exit through the `NOUT` and/or `SOC` connections to enter the next molecule. Macroscopically, the data flow is shown in Figure 7b, depending on the molecule's position, i.e. the value of `MEM2:0`.

4.4. An Example

A modulo-6 synchronous counter can be implemented with the microprogram shown in Table 2. The microprogram was written in the PICOPASCAL language [3].

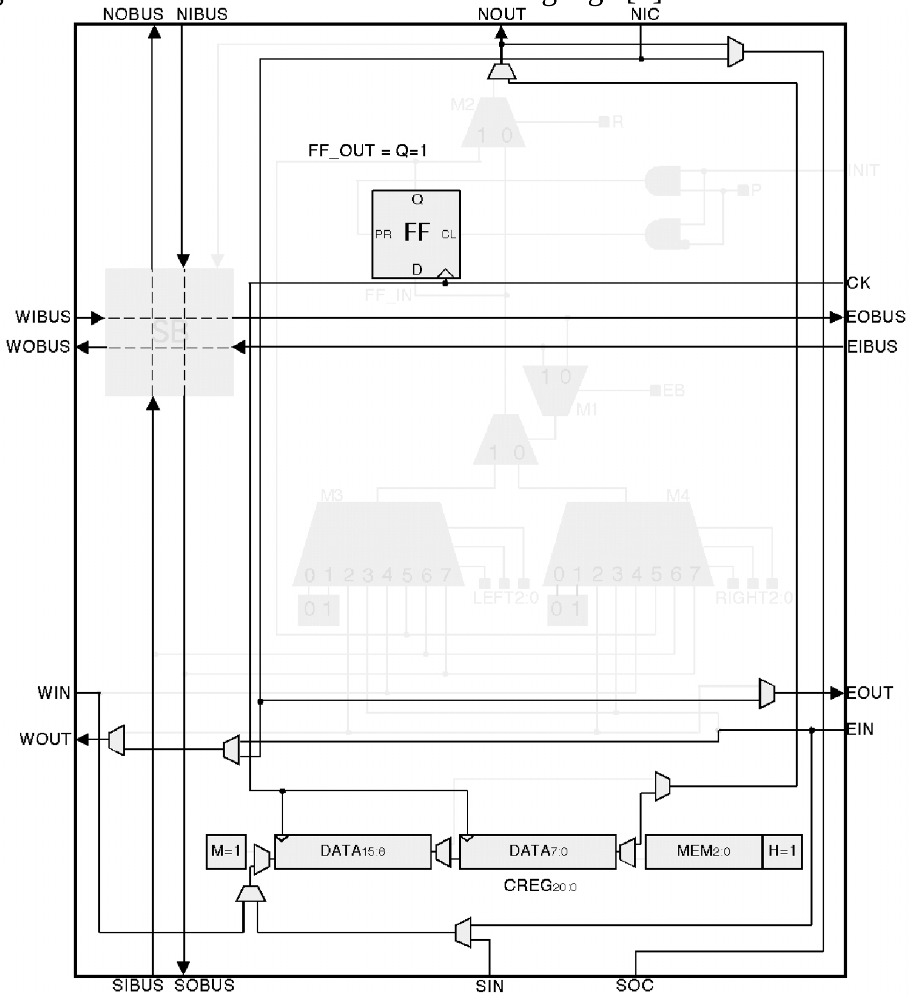


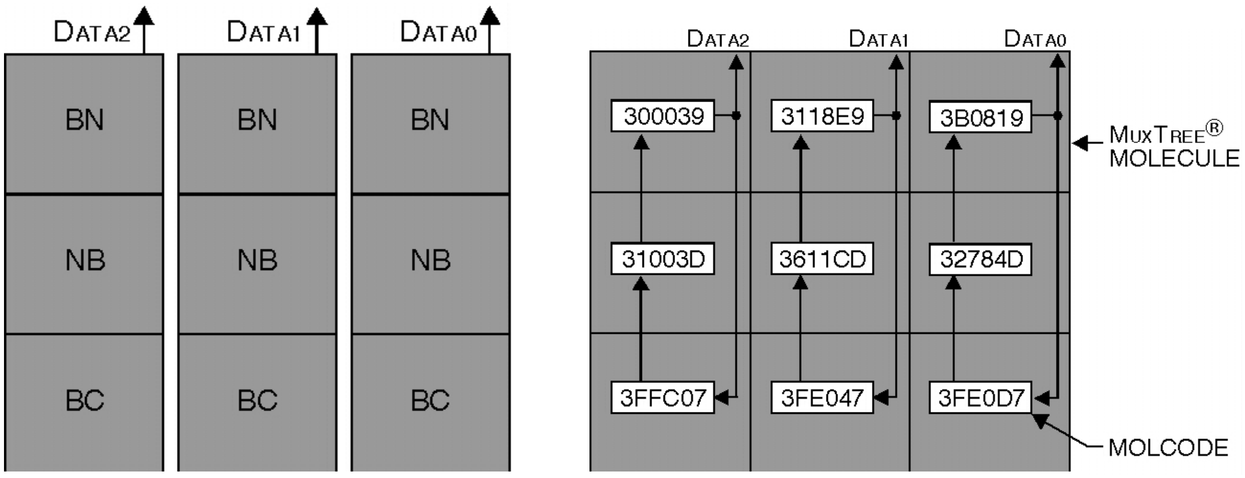
Figure 12 Layout of a MUXTREE® molecule in long memory mode.

ADDRESS	DATA	DATA$_{2:0}$	PICOPASCAL
00	5	101	while H
01	6	110	while H′
02	2	010	if [Q_2]
03	2	010	if [Q_1]
04	0	000	do 0
05	0	000	do 0
06	0	000	do 0
07	3	011	else
08	2	010	if [Q_0]
09	0	000	do 0
0A	0	000	do 0
0B	0	000	do 0
0C	3	011	else
0D	1	001	do 1
0E	0	000	do 0
0F	1	001	do 1
10	4	100	endif
11	4	100	endif
12	3	011	else
13	2	010	if [Q_1]
14	2	010	if [Q_0]
15	0	000	do 0
16	0	000	do 0
17	1	001	do 1
18	3	011	else
19	1	001	do 1
1A	1	001	do 1
1B	0	000	do 0
1C	4	100	endif
1D	3	011	else
1E	2	010	if [Q_0]
1F	0	000	do 0
20	1	001	do 1
21	0	000	do 0
22	3	011	else
23	1	001	do 1
24	0	000	do 0
25	0	000	do 0
26	4	100	endif
27	4	100	endif
28	4	100	endif
29	7	111	end
2A	7	111	end
2B	7	111	end
2C	7	111	end
2D	7	111	end
2E	7	111	end
2F	7	111	end

Table 2 The microprogram for a modulo-6 counter.

The microprogram shown in Table 2 consists of 42 memory words (addresses from 00 to 29 in hexadecimal), each word being 3 bits long (DATA in decimal, DATA2:0 in binary). To store this program, we could use a memory (Figure 13) structured as three single-column shift memories, each column consisting of 3 molecules operating in long memory mode (Figure 13a). Each column stores one of the DATA bits (DATA2, DATA1 and DATA0).

Each shift memory column stores 48 bits of data (3 molecules per column, each molecule storing 16 bits of data). Since the microprogram needs only 42 words, the last memory entries, from address 2A to address 2F, repeat the last instruction (DATA=7=end).



(a) (b)

Figure 13 A microprogrammed modulo-6 counter. (a) The memory structure using 3 single-column shift-memories. (b) The corresponding molecular codes.

At configuration time, the string of MolCodes, defined as the ribosomic genome RG, enters the molecules following the path shown to Figure 5c (the memory mode molecules are configured as any other molecule in the array). At operating time, because of the values of the MEM2:0 bits, the actual connections within the memory structure are those shown in Figure 13b. The data stored in the memory structure constitute the program of a binary decision machine (not shown here) and are finally defined as the *operative genome* OG [3].

4.5. The Hold Mechanism

Our approach to the design of the memory mode led us to build a continuously shifting memory (Figure 13b). We can store a program inside the memory and execute it cyclically for an indefinite period of time, one instruction per clock cycle. In order to stop the shifting process (a feature that might be useful for certain kinds of applications), we need some kind of *memory hold* mechanism. Essentially, this mechanism allows the memory to shift only as long as the HOLD signal of the memory structure is low (logic “0”). As soon as this signal becomes high (logic “1”) the shifting is blocked and the program halts on the address it is accessing.

The HOLD signal is connected to the input signal SIN of the bottom left molecule (the bottom left corner) of the memory structure (the SIN signal, in this particular molecule, is not used after configuration time in any of the two memory modes, as can be seen Figure 7). The HOLD signal propagates to all memory molecules of the structure, along the path shown in Figure 14.

In the case where K several distinct memory areas are simultaneously used (for example, K=3 in Figure 13), it is possible to define K different HOLD signals HOLD1, HOLD2, ..., HOLDk. Independently of the number of HOLD signals, the user must obviously implement the correct routing of these signals or fix the value HOLD=0 if there is no need to stop the shifting process (that is, the user must force the NOUT output of the molecules below the bottom left corners of all memory structures to logic “0” when no memory hold is required).

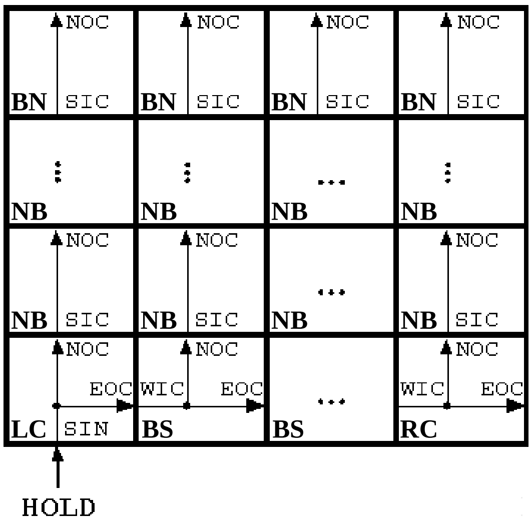


Figure 14 Propagation of the HOLD signal within a shift-memory structure.

5. The Space Divider

5.1. Generalities

The information contained in the MolCode defines the logic function, the connections, and/or the memory data of each molecule. To obtain a functional cell, i.e. an assembly of MUXTREE® molecules, we require two additional pieces of information, defining the physical position of each molecule within a cell and the presence and position of the spare columns required by the self-repair mechanism (described in detail in [1] and briefly resumed in Section 6).

The mechanism we adopted to introduce this information in the array consists of introducing in the FPGA a regular network of automata (state machines) called *space divider*. Each vertical or horizontal band of the example of Figure 15 is an instance of this automaton. Using the space divider, one can divide the entire space of the FPGA into cells of identical size and to specify the position of the spare columns. Figure 15 shows an FPGA divided into cells of height 3 and width 3, with the third column of each cell being spare. The information needed for the molecular configuration is called the *polymerase genome* PG. It can be inferred from Figure 15 and consists of a cycle of the following states:

```
PG = C,V,V,H,S,C,…
```

where C represents a corner, V a vertical band, H an horizontal band, and S an horizontal band associated with a spare column. More generally, let us use the notation {X}·[k] to represent the state (or the sequence of states) X repeated k times. Then, an organism consisting of an array of m cells horizontally and n cells vertically, each cell of height h and width w, will be defined by the following polymerase genome:

```
PG = {C,{V}·[h-1],{H}·[w-1]}·[m+n],C
```

where the presence of spare columns will be indicated by replacing one or more occurrences of H by S. The details of the design of the space divider are described elsewhere [1][3].

The components of the space divider are then the following:

- vertical band (V): the membrane grows vertically, from bottom to top;
- horizontal band (H): the membrane grows horizontally, from left to right;
- corner (C): the membrane growing splits in two directions, horizontally to the right and vertically upwards;
- spare (S): a special case of horizontal band, defining a column of spare molecules.

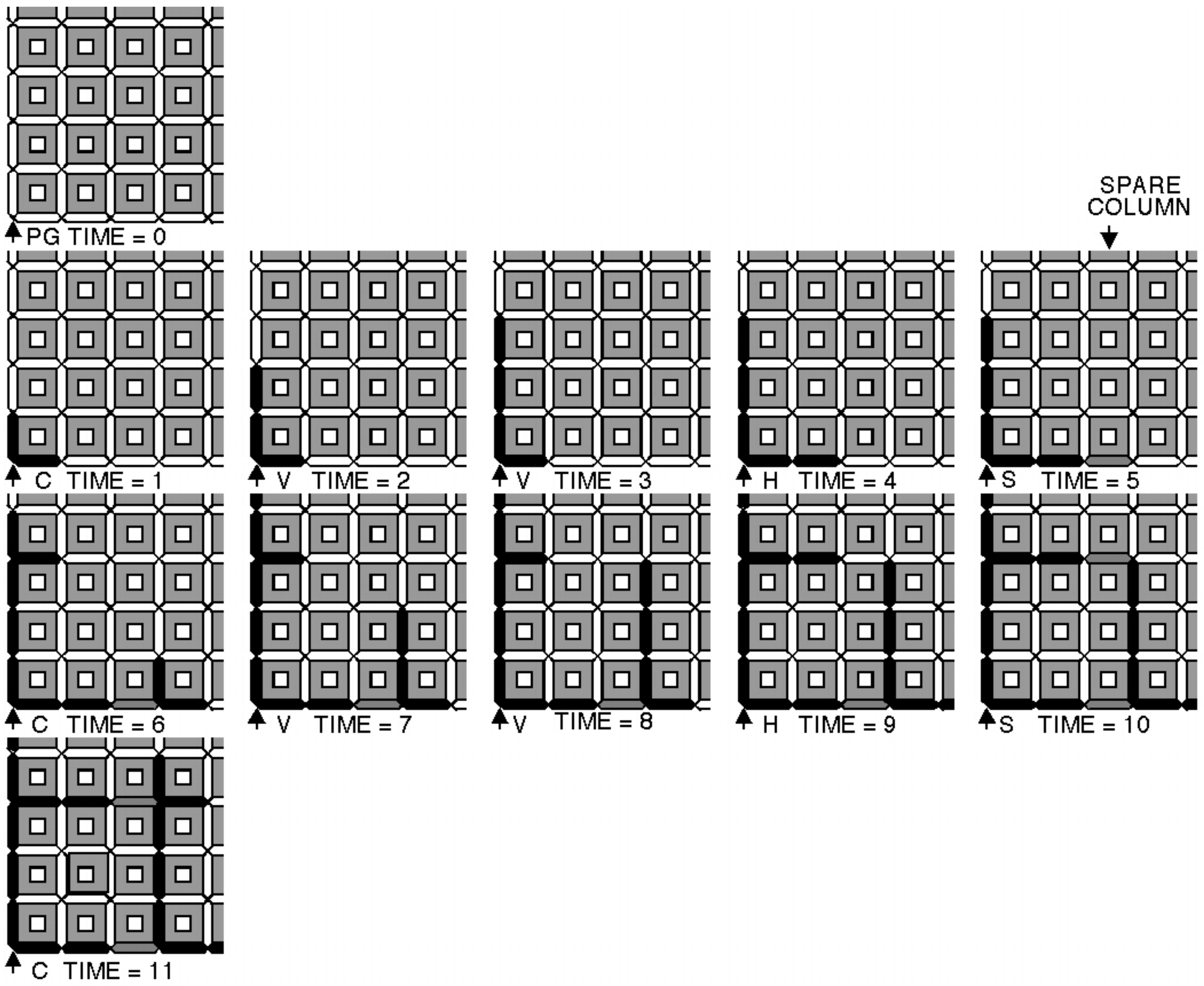


Figure 15 Example of space divider operation (cell height=3, cell width=3, 1 spare column out of 3).
[polymerase genome `PG=C,V,V,H,S,C,...`]

The various components of the space divider are binary-coded using 3 bits (`COMP2:0`) and presented in Table 3.

5.2. An Example

Let us consider the case of two identical unicellular organisms (Figure 16), where the specifications of each organism are those of the modulo-4 up-down counter. In this particular case, m signifies the number of organisms along the horizontal coordinate, and n signifies the number of organisms along the vertical coordinate (i.e., `m=2` and `n=1`). The ribosomic genome `RG` is described in Subsection 3.2 (Figure 5c). In the actual implementation, we add a spare column at the right of each organism.

The mechanism of inferring the polymerase genome was described in Subsection 5.1. In our case we have two organisms (`m=2`, `n=1`), each being constructed with a single cell (`w=3`, `h=3`). The polymerase genome `PG` has the following form:

```
PG = {C,{V}·[2],H,S}·[3],C
```

which is exactly the example shown in Figure 15.

After coding each component of the polymerase genome `PG` as shown in Table 3, the binary coded `PG` results as `PG1`:

```
PG1 = {111,{101}·[2],100,110}·[3],111
```

COMP2:0	SPACE DIVIDER STATE	
100	Horizontal band	H
101	Vertical band	V
110	Spare horizontal band	S
111	Corner	C

Table 3 The possible states of the space divider's elements.

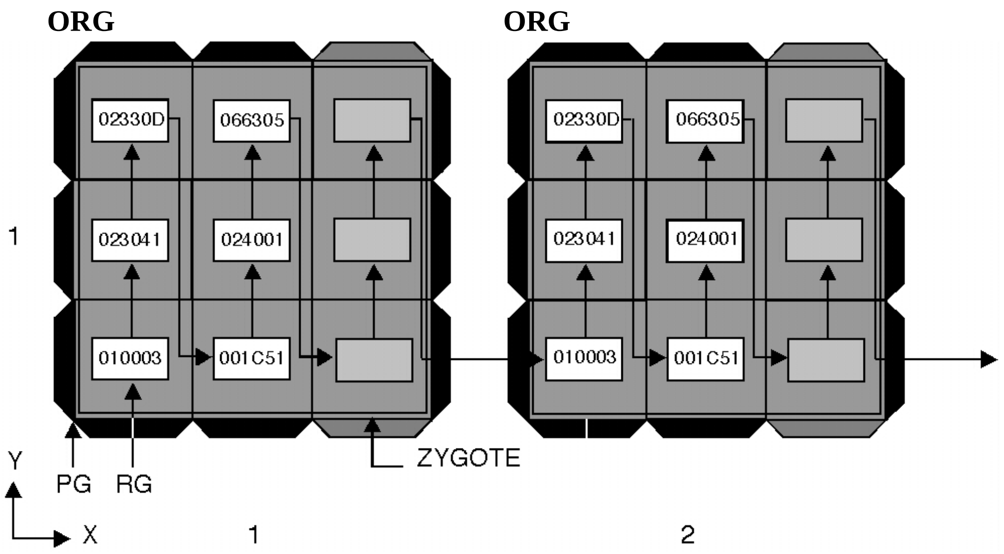


Figure 16 Two identical, unicellular organisms, each implementing the modulo-4 up-down counter.

6. Fault Detection and Self-Repair

6.1. Generalities

To provide the level of reliability and to integrate all the features required by the EMBRYONICS project, the specifications of the molecular self-repair system (as opposed to the *cellular* self-repair system, detailed elsewhere [3]) must include the following features [1]:

- it must operate in real time;
- it must preserve the state of the array, that is, the state of the flip-flop in each molecule;
- it must assure the automatic detection of a fault (self-test), its localization, and its repair (self-repair) at the molecular level;
- it must involve an acceptable overhead;
- finally, in case of multiple faults (too many faulty molecules), it must generate a global signal `KILL=1` which activates the suppression of the cell and starts the self-repair process of the complete organism.

To meet these requirements with acceptable overhead, we adopted the following solutions [1]:

- The functional part of the molecule (the multiplexers and the flip-flop) is tested through space-redundancy: the logic is duplicated and the outputs of the two copies are compared to detect a fault (Figure 17). A third copy of the flip-flop allows self-repair.
- The configuration register (`CREG`) is tested every time the configuration is entered.
- Faults on the connections (and in the switch block `SB`) can be detected, but cannot be repaired.

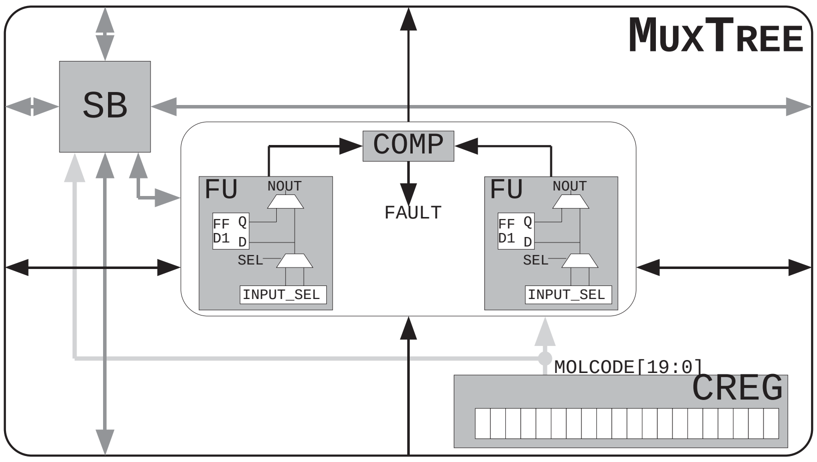


Figure 17 Test through duplication in a MUXTREE® molecule.

To minimize the hardware overhead, our self-repair system exploits the distribution of spare columns by limiting the reconfiguration of the array to a single molecule per line between two spare columns (Figure 18). This allows a reasonable amount of logic while keeping a more than acceptable level of robustness.

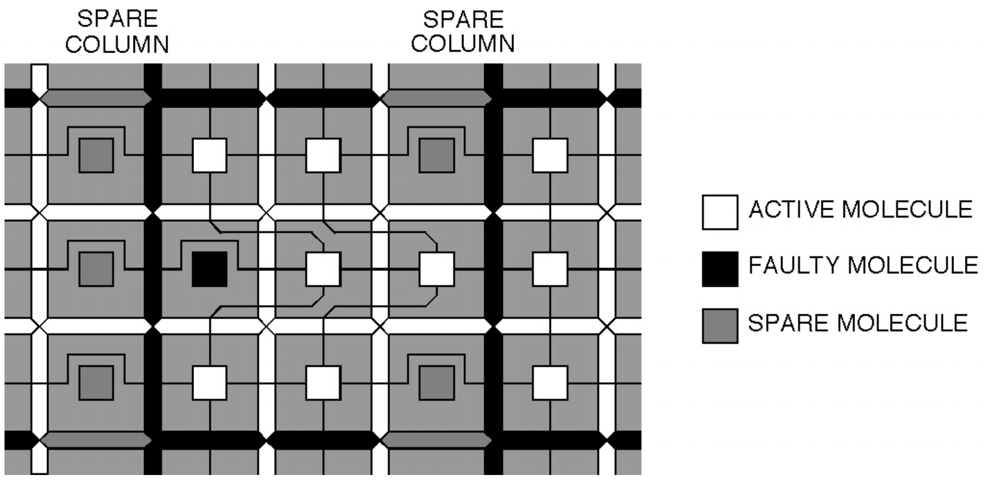


Figure 18 The self-repair mechanism for an array of MUXTREE® molecules.

6.2. The Life Cycle of an Organism

The reliability of an EMBRYONICS system depends not on a single level of self-repair, but rather on a two-level mechanism [3]: should the self-repair capabilities of the MUXTREE® molecular level be exceeded, a global `KILL` signal is generated (Figure 19) and the system will attempt to reconfigure at the higher (cellular) level. It is the combination of these two self-repair mechanisms that maximizes the reliability of EMBRYONICS systems.

At the cellular level we assume that organisms are grown from the left to the right in a rectangular array of cells, that the cells' functionality is determined by their spatial position (X,Y coordinates) within the array, and that there are spare columns (made of cells) in excess at the right side of the array. When the `KILL` signal is generated, it will propagate through the whole column containing the faulty cell. The process will "destroy" the entire column (i.e., reset the configuration registers of all its molecules, making them transparent to all horizontal connections) and force all the columns to its right to recalculate their coordinates, recovering the lost functionality using the spare columns (Figure 20). The reconfiguration at the cellular level is therefore a very expensive process, one entire column of cells being disabled because of just one faulty cell.

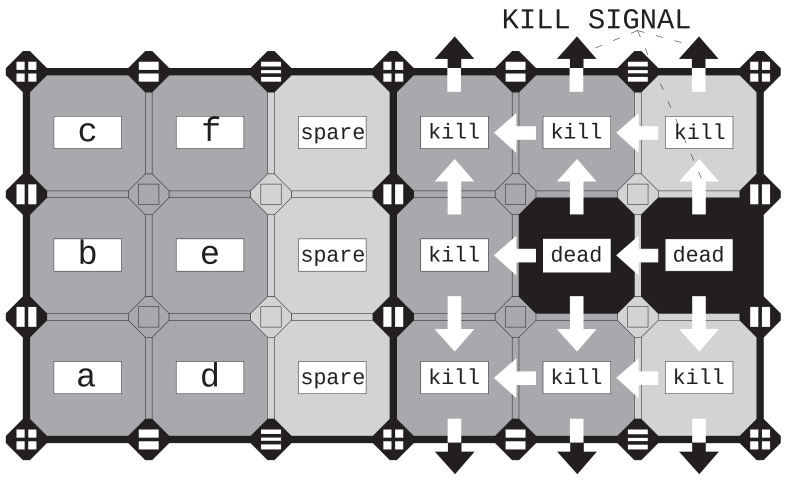


Figure 19 Faults in adjacent molecules activate the `KILL` signal.

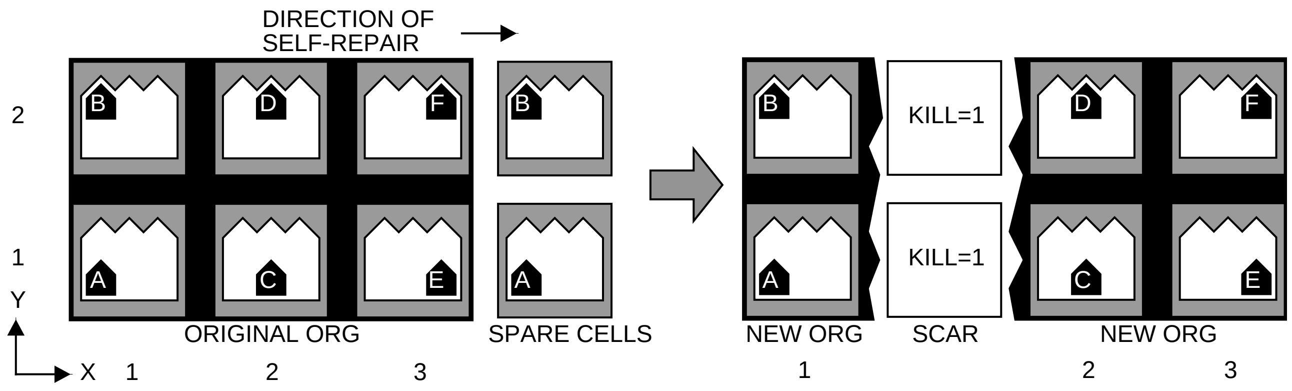


Figure 20 Self-repair at the cellular level through coordinate re-computation.

Since permanently disabling an entire column of cells is expensive, it is important to closely analyze the conditions that can lead to such a traumatic event.

The most effective approach to this kind of analysis is through a detailed description of the behavior of our circuits throughout their operation. What follows is a review of an artificial organism's life cycle, starting from power-up (Figure 21):

1) Space dividing phase: the polymerase genome `PG` propagates through the array and delimits the cells (Figure 15). The next phase is 2.
2) `CREG` registers testing phase: the `CREG` register of each molecule is tested using a dedicated *test pattern* [1]. If no faulty registers are detected, the next phase is 3. If faults are detected, the next phase is 5a.
3) Configuration phase: each molecule receives its ribosomic (`RG`) and/or operative (`OG`) genomes. The order in which the molecules are loaded with data is shown in Figure 16. The next phase is 4.
4) Normal operating phase: cells are functioning normally. A fault-detection mechanism will signal the errors that might occur inside each molecule. When errors are detected, the next phase is 5b, otherwise 4.
5) One or several faulty molecules are detected. If there are enough spare molecules, the next phase is 6 (a or b). If not, the next phase is 7 (a or b).
6) The self-repair process takes place at the molecular level, as shown in Figure 18:
   - 6a) The next phase is 3, i.e. the configuration phase.
   - 6b) The next phase is 4, i.e. the operating phase.

7) The self-repair process takes place at the higher, cellular level. There are two possible causes for activating this process:

- a faulty molecule is detected in a row and there is no spare molecule available;
- a faulty molecule is detected and there is only one spare molecule left in the row, but the spare molecule is itself faulty.

In both cases the self-repair mechanism is overwhelmed and the cell will die, generating a `KILL` signal that activates the reconfiguration at the higher, cellular level.

7a) The next phase is 3, i.e. the configuration phase.

7b) The next phase is 4, i.e. the operating phase.

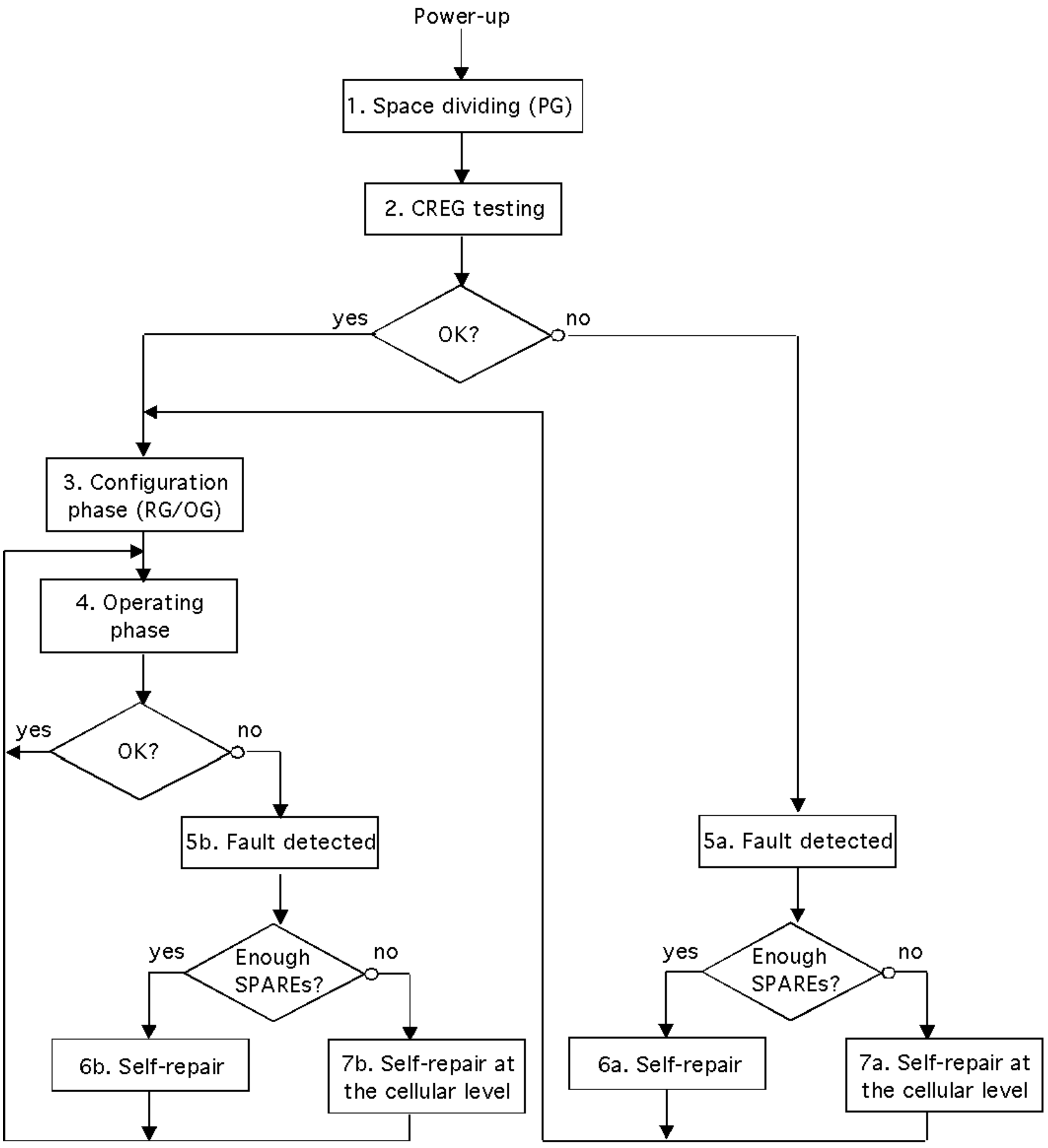


Figure 21 Flowchart of an organism's life cycle.

6.3. The `UNKILL` mechanism

In digital electronic systems, the majority of hardware faults occurring in the silicon substrate are in fact *transient*, that is, disappear after a short span of time. This observation is an important issue when designing self-repairing hardware: the parts of the circuit that have been “killed” because of the detection of a fault could potentially come back to “life” after a brief delay.

Detecting the disappearance of a fault and handling the “unkilling”, however, usually requires a relatively complex circuit. This complexity prevents us from implementing this feature at the molecular level (the molecules being very small and simple components). At the cellular level, on the other hand, it is not only possible, but also quite simple to implement this feature.

We have seen that self-repair at the cellular level consists of “destroying” (i.e., resetting) the configuration of all the molecules within a column of cells, with the effect of making the column “invisible” to the array. As it has been reset, however, nothing prevents us from sending once more the configuration bitstream to the deactivated molecules (in Figure 22, the fault in one of the dead molecules in Figure 19 has vanished, allowing the cell to come back to life, even if the other molecule contains a permanent fault): if some or all the faults have vanished, then the configuration will restore the functionality of the cells, and the reappearance of the column within the array will engender a new re-computation of the array's coordinates (in the direction opposite to self-repair) and restore the entire array to its original size and functionality.

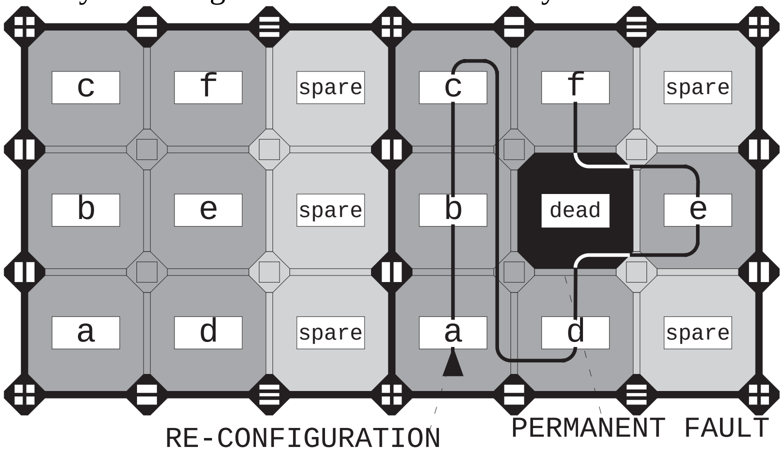


Figure 22 While the left-hand side cell keeps operating, the right-hand side cell is “unkilled”.

As far as the user is concerned, the presence of the `UNKILL` mechanism has two direct consequences, both concerning the format of the bitstream that configures the MuxTree® array:

- ♦ Since the death of a cell resets the configuration registers of all the molecules, and the `UNKILL` mechanism requires that the molecules re-acquire their functionality should one or more faults disappear, it becomes necessary to send the configuration bitstream into the array more than once (not only when the circuits first configured, but also whenever a cell has to be re-configured to be “unkilled”). Since signaling to the exterior when such a re-configuration is required would imply the need of a centralized control, which goes against the philosophy of the project, the simplest solution is to have the bitstream sent to the array continuously, in a loop.
- ♦ If the bitstream is sent continuously, the array must be capable of detecting the start of the configuration (the configuration of the bottom left molecule in the cell), in order to correctly configure the cell. To do so, a hardware mechanism was put in place within the molecule. This mechanism requires that, in the bitstream, a *minimum* of 24 logic “`0`”s must separate each instance of the 22 configuration bits of the `CREG` of the molecules.

In the next section, describing the first physical implementation of a MuxTree®, we will see exactly how these requirements affect the configuration bitstream.

7. Implementation 1: The BIODULE 603

7.1. Hardware

Our laboratory's teaching activities include a number of laboratory sessions aimed at introducing the principles of logic design. In order to accomplish this task, we use a set of *logidules* (logic modules), plastic cubes containing standard logic circuits (AND gates, RAMs, etc.). These modules can be connected together (not unlike a LEGO brick), automatically providing the circuits' power supply as well as the minimal connections between modules.

Aside from being an invaluable teaching tool, the logidules are also an interesting platform for the development of prototypes, and we decided to exploit the same approach in the design of the prototypes related to the Embryonics project, which we call BIODULES (for *biologically-inspired logic modules*). The prototype of MUXTREE®, our molecule, is no exception, and was embedded in a module called BIODULE 603 (Figure 23).

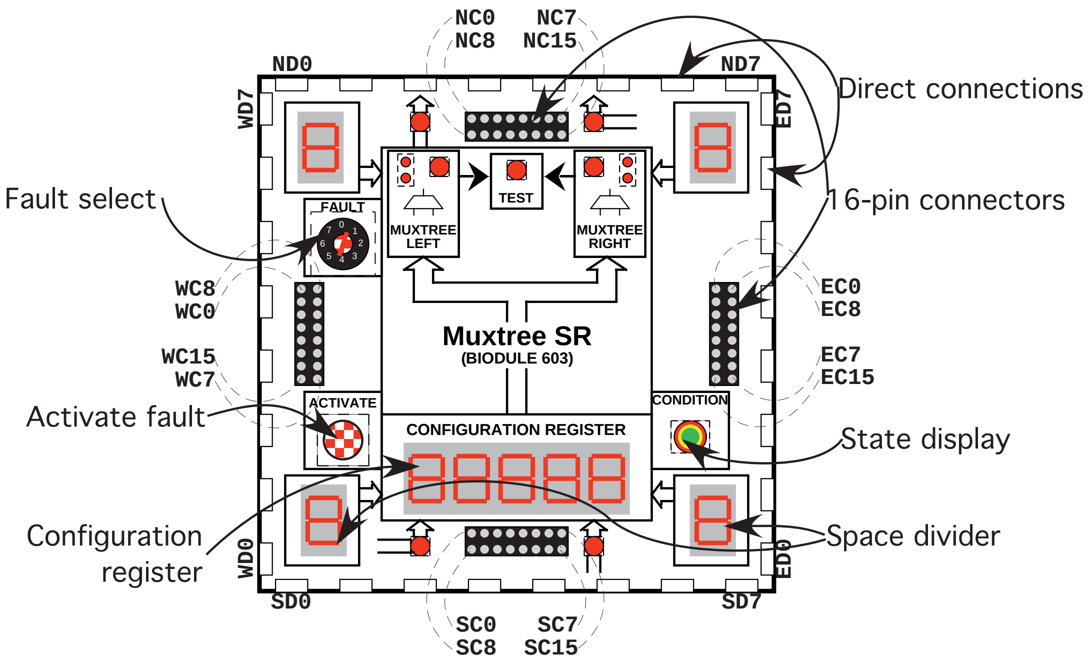


Figure 23 Top view of the BIODULE 603 module.

An exhaustive description of the hardware structure of the BIODULE 603 is available in [1]. In this section, only the main components will be mentioned, together with the information necessary for the actual use of the circuits.

Each BIODULE 603 is designed to contain a single MUXTREE® molecule, along with four copies of the space divider, the cellular automaton used to define the blocks' membranes (the need for the four copies will be explained below). The BIODULE 603's displays include:

- four 7-segment displays (in the four corners) which show the state of the corresponding space divider elements;
- a set of five 7-segment displays which show the value of the element's configuration;
- a set of four LEDs (top and bottom) used to indicate the source of the signals (in case of reconfiguration);
- two sets of LEDs which show the value of the functional outputs and the input and output values of the flip-flops of each of the two copies of the element's programmable function;
- a single LED which lights up whenever a fault is detected in the element;
- a three-color LED which denotes the state of the element (green for active, red for dead, yellow for spare or during the configuration and repair processes).

In addition to the displays, each BIODULE 603 also contains a rotary encoder used to select one of ten possible faults to introduce in the element, and a push-button which activates the selected fault: a single push will introduce a temporary fault (which will disappear whenever the FPGA is reset), while a double push will introduce a permanent fault.

Invisible to the user, the BIODULE 603 contains a small printed circuit board on which all the required circuitry is mounted. Aside from the displays mentioned above and the components (such as resistor nets) required for their control, the only “active” circuits are a Xilinx XC4013PQ240-4 FPGA, used to implement the MUXTREE® element, and a 256k-bit E^2PROM used to store the Xilinx’s configuration bitstream.

Like all logidules, the BIODULES 603 can be joined together to form a two-dimensional array (Figure 24), with the connections implemented either through the automatic contacts on the perimeter of the box (8 per side) or through four 16-bit-wide connectors. As of now, 18 modules are available, allowing us to build a 6x3 array of MUXTREE® elements (from this figure, the need for 4 copies of the space divider in each BIODULE should be obvious, as it allows all of the molecules to be used by providing cell borders on each side of the array).

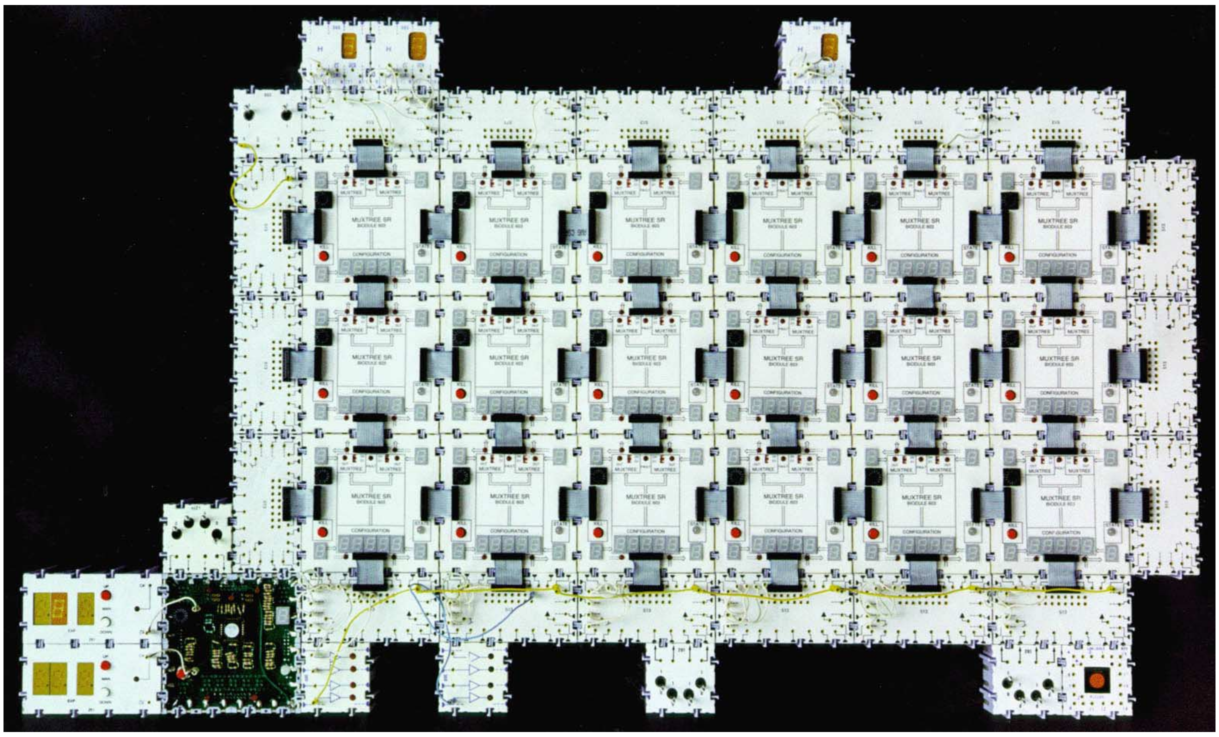


Figure 24 A 3x6 array of BIODULE 603 modules.

In addition to the BIODULES themselves, an array such as the one shown in Figure 24 requires the following additional components:

- A set of dedicated I/O logidules, placed on the four sides of the array, allowing it to be interfaced with other standard logidules. The layout of the pins necessary for the use of the array are provided in Table 4 and Table 5. For each pin, the tables define a LABEL, which follows the naming convention used throughout this document. All the pins that are not defined in the table are used for internal signals, and, if inputs, often require a particular value on the edges of the array. The latter values are pre-defined by a set of jumpers inside the plastic box of the I/O logidules, and should not be of concern to the user for a normal utilization of the BIODULES. For the data and control pins specified in the tables, the user can connect the desired pin to a conventional logidule (for example, a 7-segment display or on a set of switches) using standard wires.

LABEL	PIN	DESCRIPTION
SIN	SD4	Direct connection from south neighbor (Input)
SIBUS	SD6	Long-distance connection from south neighbor (Input)
SOBUS	SD5	Long-distance connection from south neighbor (Output)
NOUT	ND4	Direct connection to south neighbor (Output)
NIBUS	ND5	Long-distance connection from north neighbor (Input)
NOBUS	ND6	Long-distance connection from north neighbor (Output)
WIN	WD5	Direct connection from southeast neighbor (Input)
WOUT	WD4	Direct connection to southeast neighbor (Output)
WIBUS	WD7	Long-distance connection from west neighbor (Input)
WOBUS	WD6	Long-distance connection from west neighbor (Output)
EIN	ED4	Direct connection from southwest neighbor (Input)
EOUT	ED5	Direct connection to southwest neighbor (Output)
EIBUS	ED6	Long-distance connection from east neighbor (Input)
EOBUS	ED7	Long-distance connection from east neighbor (Output)

Table 4 Data pins on a BIODULE 603 (refer to Figure 23 for pin numbering).

LABEL	PIN	DESCRIPTION
CCK	SD0	Configuration (fast) clock for the configuration registers (CREG) in the MUXTREE® molecules, for the space divider, and for the self-repair process.
FCK	SD1	Functional (slow) clock for the flip-flops in the MUXTREE® molecules.
XRST	SD7	Reset of the Xilinx circuits in the BIODULES.
CRST	SD2	Reset of the MUXTREE® configuration.
INIT	SD3	Set/reset of the flip-flops in the MUXTREE® molecule.
MEMB	SC2	Input for the membrane automaton (space divider)
CFG_IN	SC4	Input for the MUXTREE® configuration bitstream

Table 5 Control pins on a BIODULE 603 (refer to Figure 23 for pin numbering).

♦ A *configuration loader*, based itself on a Xilinx XC4013PQ240-4 FPGA, containing all the logic and all the information necessary to charge the configuration into an array of BIODULES 603. The Xilinx embeds a ROM (Read-Only Memory) containing up to 16 configurations for the array (the configurations need to be entered by hand on a schematic editor, and then the new configuration bitstream for the loader must be created with the tools provided by Xilinx, and stored on an E^{2}PROM). The user selects the desired configuration with a rotary switch. In addition, the Xilinx contains all the logic necessary for the synchronization and propagation of all global signals required by the array. The user must provide the two basic clocks (the configuration clock `CCK` and the functional clock `FCK`) and the three resets (the reset of the flip-flops within the molecules `INIT`, the reset of the configuration registers of the molecules `CRST`, and the reset of the Xilinx FPGAs in the BIODULES 603 `XRST`) using conventional logidules (see bottom left of Figure 24).

A description of the contents of the ROM in the loader, the last information required to be able to use an array of BIODULES 603, is provided in the next subsection.

7.2. Software

The software side of the machine is very limited. In fact, since the loader handles the interface between the user and the array of molecules, the only software required is the configuration bitstream to be burned into the loader's E^{2}PROM. What follows is a description of the precise format of a complete configuration for an array of BIODULES, applied to the 3x3 cell example described in Subsections 3.2 and 5.2. The configuration can then be stored, in this precise format, in the loader's ROM.

The polymerase genome PG and the ribosomic genome RG configure the array of MUXTREE® molecules at configuration time. For this reason, PG and RG are binary-coded inside the ROM memory of a state machine (the *configuration loader*) whose only role is to send them toward the array. For implementation reasons, the polymerase genome PG has to be expressed in hexadecimal code. The entire process of properly coding PG is described as follows, starting from the coding of the polymerase genome PG as shown in Table 3. The binary-coded PG results in:

```
PG1 = {111,{101}·[2],100,110}·[3],111
```

First, PG1 must be turned around, becoming PG2:

```
PG2 = 111,{011,001,{101}·[2],111}·[3]
```

To transform PG2 into hexadecimal ROM-ready code, due to implementation reasons, we have to add a zero to the rightmost position, which is also the least significant bit. Unrolling PG2 and adding the necessary zero to the least significant position gives us PG3:

```
PG3 =111 011 001 101 101 111 011 001 101 101 111 011 001 101
     101 111 0
```

Keeping in mind the direction from the least significant bit to the most significant bit is from right to left, the hexadecimal coded and E^2PROM-ready polymerase genome (PG4) becomes:

```
PG4 = 1D9B7 B36F66DE
```

E^2PROM ADDRESS	PG DATA
0	B36F66DE
1	0001D9B7

Table 6 The E^2PROM-ready polymerase genome PG.

The ROM stores words worth 8 hexadecimal characters of data, so we need to split PG4 in words of this length. The least significant word is stored at the lower address inside the ROM. In our case, the ROM containing PG4 is presented in Table 6. The way of calculating RG was shown in Subsection 3.2. Moreover, as mentioned in Subsection 6.3, a *minimum* of 24 logic "0"s must separate the configurations of the molecules. Thus, the complete ROM for our example in Figure 16, containing both the polymerase genome PG and the ribosomic genome RG, is shown in Table 7 (where more that the minimum amount of logic "0"s was added to preserve readability).

E^2PROM ADDRESS	DATA	MEANING
0	B36F66DE	Polymerase genome PG (Table 6)
1	0001D9B7	
2	00000000	Nil pattern
3	30000080	CREG register test pattern
4	00000000	Nil pattern
5	01000300	Ribosomic RG and/or operative OG genomes interleaved with nil patterns (Figure 5c)
6	00000000	
7	02304100	
8	00000000	
9	02330D00	
A	00000000	
B	001C5100	
C	00000000	
D	02400100	
E	00000000	
F	06630500	

Table 7 E^2PROM contents for configuring the two identical unicellular organisms.

For practical reasons, the E^{2}PROM is in fact implemented as a 16x512 bit ROM in the Xilinx XC4013PQ240-4 FPGA of the configuration loader (as described in the preceding subsection). In order to create new bitstreams for the array, the user will need to define the contents of the bitstream in the format specified in Table 7, insert them (by hand) into the schematic diagram of the configuration loader, use the Xilinx tools to derive the configuration of the Xilinx, and burn this configuration into the E^{2}PROM of the loader. The configuration of the MUXTREE® molecules thus becomes hard-wired into the configuration of the loader's Xilinx (16 configurations can be stored in parallel, reducing the frequency of this rather cumbersome process).

8. Conclusions

This document defines the final specifications for the molecular level of our EMBRYONICS systems. It is not meant as a detailed description of a given implementation, but rather as a user's manual, providing the instructions necessary to be able to use an array of MUXTREE® molecules.

It contains, among other things, a description of the double mode of operation of the MUXTREE® molecule (logic or memory), which allows some configurations that were not possible before or they were very limited by inefficiently using the logic inside the molecule. Now the molecules can operate in two different modes (the logic mode and the memory mode, which features two different sub-modes) and cells may be built using molecules in any mixture of operating modes.

The flexibility that was gained by implementing our final design brought the MUXTREE® molecule to a much more mature status. We believe that the process of building an electronic molecule powerful and flexible enough so as to permit a comparison with its biological correspondent has now reached a local maximum. The development of the MUXTREE® molecule will not be pushed further: the next generation of molecules, already under development, will build upon the experience gained through MUXTREE® to push well beyond its capabilities.

The MUXTREE® molecule is nevertheless powerful and versatile enough, in this final implementation, to allow us to focus on developing new cells and multicellular organisms, thus bringing the realms of artificial life further from a dream and closer to reality.

References

[1] G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. Ph.D. Thesis No. 1827, The Swiss Federal Institute of Technology, Lausanne, 1998.

[2] L. Prodan, G. Tempesti, D. Mange, A. Stauffer "Biology Meets Electronics: The Path to a Bio-Inspired FPGA". In J. Miller, A. Thompson, P. Thomson, T.C. Fogarty (Eds.), *Evolvable Systems: From Biology to Hardware*, volume 1801 of *Lecture Notes in Computer Science*, pp. 187-196, Springer, Berlin, 2000.

[3] D. Mange, M. Tomassini, eds. *Bio-Inspired Computing Machines: Towards Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[4] D. Mange, M. Sipper, A. Stauffer, G. Tempesti. "Towards Robust Integrated Circuits: The Embryonics Approach". *Proceedings of the IEEE*, vol. 88, no. 4, April 2000, pp. 516-541.