

# **Cellot**

## ***FPCL***

***Field Programmable Cell Logic***

***From Idea to Product & ASIC***

**A Breakthrough Technology To  
Replace the Traditional PLD  
Marketplace  
July 2003**

**NOTE**

The information is subject to change without notice and does not represent a commitment on the part of Cellot Inc.

## Table of contents

Abstract.....	3
FPCL Architecture .....	5
Hardware and Software: “The line is blurring!” .....	5
Significant ‘From Idea to Product’ Time Saving.....	7
FPCL Applications Development tools .....	7
Using Schematic Capture.....	7
Using Script Language.....	8
Using Both Script and Schematic Capture.....	8
Using Code to Manipulate the Results.....	8
Using High Level Language for FPCL Applications Development.....	9
Using all styles .....	9
PCI Card.....	10
Reconfigurable / Adaptive Computing Systems, Super Computers.....	10
Time Sharing Applications .....	11
Full Custom Implementation .....	12
An Implementation Example: FFT .....	12
Significantly Increasing the Application per Die Size Ratio .....	14
The FPCL as a Programmable Logic Device (PLD) .....	14
The Interconnect and the Cell Size .....	14
The Cell as a Memory .....	15
Conclusion .....	15
Fault Tolerance and Failure Immunity .....	16
Migration to ASIC .....	16
Evergreen Design.....	17

Meged Ofer  
Geva 30 St.  
Netanya, Israel, 42384  
972-9-8842554  
[Meged@Cellot.com](mailto:Meged@Cellot.com)  
[www.cellot.com](http://www.cellot.com)

## Abstract

For many years there has been an evolutionary development and improvement in the performance and complexity of Programmable Logic Devices, and recently launched FPGAs consist of millions of Gates. The basics, however, have not changed. The basic cell is a Logical Gate in which logic is implemented. Each cell is capable of doing a tiny portion of the application. To create any portion of an application, several cells are connected to each other causing differences in the implementation delay. Although each cell, normally, has the ability to be synchronized to other part of the application, there are different delays for different application parts. For example: creating a 4 to 1 decoder would probably be created by one cell with 4 inputs and one output. A 5 to 1 decoder would probably be created by two such cells in line. The propagation delay for such a decoder is twice the propagation delay of 4 to 1 decoder.

Cellot innovative Field Programmable Cell Logic (FPCL) technology is based on the concept in which the device architecture is a collection of registered memory cells interconnected via a programmable matrix that creates a recursive hierarchical (fractal like) structure. That means that all of the cells are similar and any number of cells may be combined to create similar larger cells. Each cell can be used for logic when memory is used as a look-up table, or as storage, or both. The matrix can route the output of each cell to the input of each cell and each I/O in a fixed and predictable delay.

As the device is basically a collection of resizable memory cells which implements full fixed delay connectivity it can easily and accurately be emulated on a regular PC. This emulation leads to a PC based development environment which uses the chip emulation thus enabling the One-To-One extremely fast simulation tools as well as the ability to use high level language engineering tools (e.g. C++, Verilog, Schematic capture, etc.) in creating applications. Each part of the application being developed can be in different development stages. Part can be “downloadable”, part using integer numbers, part using float numbers and so on. This allows the application engineer to be on the same time in “hardware state” and in “software state”.

One feature of the new architecture and tools enables an automatic procedure to create an RTL file ready for ASIC.

The resizable read / write cells are equivalent and built out of Atomic Cells having the same propagation delay. Every Atomic Cell can be redundant to any other Atomic Cell whether if it functions as an independent cell or, if it is part of a bigger cell. This leads to failure immunity. A simple algorithm is applied to any running application to find and replace any faulty cell even when the application is running.

The ability to use cells for logic and at the same time have other parts of the application write into such a cell thereby changing its logic functionality is ideally suited for Reconfigurable / Adaptive Computing Systems or Evolvable Hardware.

Our innovative FPCL technology has been designed to support “complex multi-million gate functionality” in a single chip. The Article will explain the ease of developing and the resulting efficiency of Fast Fourier Transform (FFT), which is common in DSP applications. It will show that the FPCL logic part is more than 10 times larger than the size of the current FPGA’s logic part for equivalent die size. As a result, and for the ability of any cell’s read, write or both and for other abilities, the Article will demonstrate that the application per die size is exceptionally high.

## FPCL Architecture

The FPCL is based on a collection of cells in a recursive, fractal like structure, which enables several different architectures. The first chosen architecture is a collection of registered memory cells and a matrix, which can route the output of each cell to the input of each cell and each I/O. Every cell is similar to any other cell. Any number of cells may be combined to create a larger similar cell. Each cell can be used for logic, implementing its memory as a look-up table, as storage or as both. The smallest (Atomic) Cell cannot be broken into smaller cells. Every Atomic Cell can be redundant to any other Atomic Cell whether it functions as a stand-alone application Cell or is part of a bigger cell.

Choosing different Atomic Cell sizes and their number on a chip allows the control of the chip's dimensions and other parameters. As an example: A device with 1M Bytes of memory (created by ~8000 1K varied data sized cells), 600 general-purpose I/O pins, 16 Global Clock inputs (8 of them having PLL for minimum skew) will provide a typical clock frequency of 250MHz (350 MHz max) and a die size of 1.7x1.7 Cm<sup>2</sup> (TSMC Library) in 0.13u (Standard Cells). Full custom design (in a slightly different architecture, which is presented below) will provide smaller size, and may double the frequency. The use of 1K memory Atomic Cells (varied in byte size to enable 1 bit data resolution) has been found as a good engineering trade-off. Small Atomic Cells increase the cross-connect wasting of "application area". The use of larger cells may become advantageous in future chip geometries and improved memories technology.

For full details, please refer to the patent publication (see [www.cellot.com](http://www.cellot.com)).

## Hardware and Software: "The line is blurring!"

As the FPCL device is basically a collection of resizable memory cells with full connectivity, fixed and predictable propagation delay it can be easily and accurately emulated on a regular PC. Each cell in the device, which is basically a registered memory, can be represented by a block of the same size memory in the PC. As such, the operation in the PC is exactly the same operation as in the device:

Applications that are implemented utilizing an FPCL device are operating by the overall integrated activities of its cells; each is responsible for doing a small portion of the application. The FPCL topology is mirrored on the PC memory so each cell in the FPCL device is mapped into a mirrored PC memory cell. Therefore, the PC Developer Environment can implement the exact same application as the FPCL does.

The PC Developer Environment has the ability of the FPCL device to change topology of the mirrored device. During application development each cell in the PC mirrored device is configured to implement one portion of the application. Once the application is broken into these portions, each can be implemented by an FPCL cell – the application is “downloadable”. Each part in the memory located in the PC can be downloaded into the same size cell in the device. Nevertheless, the application portions in the PC do not have to be implemented by a block of memory but can be implemented by a function simulating the memory block functionality.

Any such function representing a look-up table or storage has a binary input (or number of binary inputs) representing the address of the memory block (cell) and binary output representing the data output of that memory block (cell). While simulating a cell, two parameters can differ:

- Although the cell size in the device is limited by the device size, the cell size in the PC is practically unlimited.
- The device cell input and output must be binary but the cell simulation input and output can be of any type such as integer, float, double, string, etc. (for storage the input must be binary or integer)

As a matter of fact, any idea can be implemented using the concept of collections of functions each may represent a table. These “tables” are not obliged to be downloadable.

Developing a hardware application to be implemented on the FPCL device is nothing but developing a software application to implement the functionality. Once the result is satisfied, each subroutine input and output is independently changed to have binary representations for the input and the output, and then is broken into the needed “sizes”. During development, one can choose to convert part to integer numbers, leave part as double numbers and convert part to binary numbers. At the end of this process the application is running on tables (blocks of memories in the PC) and if loaded into the FPCL device – is performing the exact same functionality. In this manner, the application developed can be at the same time in “hardware state” and in “software state”. The results (tables, graphs, vectors, histograms, etc.) represent at any time the application results for any needed input.

### ***1.1 Significant 'From Idea to Product' Time Saving***

Using the above development method enables the user to develop the application ideas and the application implementation on the same tool and saves a lot of time from the idea to the product which becomes shorter and shorter as more applications are created (see the next section).

As there is no practical limitation on connectivity using the internal cross connection, the I/O pins can be set before the application is designed. Once the interface is determined, the electronics (such as the card which the device is placed on) can be built while the application is realized with the PC development tools. Such approach saves significant manufacturing time.

## **FPCL Applications Development tools**

Various interfaces are available for the FPCL Applications Developer. The idea of each interface is to provide the user the look and feel he likes while the application activity is the same: breaking the overall application into smaller portions each can be broken into smaller portion too and so on until the portions are “downloadable”. For this functionality the Development Environment has a “container”. Each container can hold other containers and / or processors (gates).

### ***1.2 Using Schematic Capture***

There is a collection of basic (atomic) processors provided to the user by libraries. Each processor represents a generic function. The user uses these processors in order to create a container which represents a downloadable cell or a non-downloadable cell. A few containers are gathered into another container to represent a small functionality. The containers and the functionalities may be added into the library so the next application design is built of bigger foundation blocks. Collection of these bigger containers, (resides in one container) may represent the needed functionality is known as Intellectual Property (IP) when the containers are built out of downloadable cells.

This kind of developing process is available as the propagation delay between each cell to each other cell is fixed and predictable, and the connectivity is full, therefore causing no synthesis issues.

### ***1.3 Using Script Language***

The above usage of integrating predefined containers (and processors) to a whole application can be done utilizing a proprietary script language. The code formatting is quite similar to the C++ formatting. The script also provides the means to create processors.

### ***1.4 Using Both Script and Schematic Capture***

The user can choose to create containers in the same manner a software programmer divides his job into block diagrams. Each can be divided again into other containers. At any point, the user may decide to implement the container utilizing the script language.

The basic (atomic) processors are provided to the user by libraries, which have been created by Cellot programmers and integrated into the Developer Environment C++ code. Nevertheless, other processors can be developed by the user and integrated into the same libraries. These user-defined processors can be written using the script language. As a matter of fact, there is no different between implementation procedure of a generic processor and a complicated IP utilizing the script language.

### ***1.5 Using Code to Manipulate the Results***

Results of running an application can be presented to the user in several ways such as tables (HEX, Binary, PCM, etc.), graphs (dots, lines, logic style, histograms, etc.) and more. The user can manipulate the presented result by the code. For example: A PCM signal is to be presented. The code result can be presented as is. If needed (A Law version of the PCM), any second bit can be inverted before being presented as hex / binary tables or logic style view. The number can be converted into integer number representing the linear PCM value and presented by a graph (scope style). Any conversion is available.

The code to manipulate the results can be the same script language but can also be non-downloadable schematic capture containers, which convert the signal into the needed presentation.

## ***1.6 Using High Level Language for FPCL Applications Development***

A few classes, an appropriate library and a simple interface to the FPCL Developer Environment are used to support any software engineer (no need for expertise in hardware) and any hardware engineer who is familiar with C++ utilizing a C++ development environment in order to:

- Simulate any hardware application,
- Create any hardware application using the FPCL emulation,
- Create any signal generation,
- Manipulate result (e.g. see graphic results in logarithmic view),
- Add commands to the FPCL Compiler,
- Manipulate the schematic diagram, and
- Add features such as wizard buttons to the FPCL Developer Environment.

Features that are developed utilizing the C++ Support can be easily integrated into the basic FPCL Developer Environment (which is written in C++ also). For example: The Compiler has been opened to support languages such as Verilog. This support can be implemented both by enhancing the FPCL Developer Environment code or using the C++ Support and then integrating (or not) the code into the FPCL Developer environment. It is seamless to the Verilog user if the code is integrated into the FPCL Developer environment or not (nevertheless, now it is integrated).

## ***1.7 Using all styles***

The script language has commands to:

- Import external functionality, and
- Define the language to be used for one command or in a block (currently only C++ and Verilog).

As the script can be used along with the schematic capture, the above commands let the user use any mixture of code and integrate the result into the same libraries. It is useful when a few programmers are working on the same job and each prefers a different interface style. Although, normally, one who writes in C++ will prefer the C++ developing environment (such as Microsoft® development environment – MFC), sometimes this mixture is useful for commands such as Logarithm. Using such commands from the C++ language makes the look and feel such that the script language itself has the Log command. Obviously, such commands can be integrated into the script language if considered valuable by Cellot's programmer team.

## 1.8 PCI Card

The PCI card is used to accelerate the emulation. It has at least two FPCL devices: one for the implementation and one to trace the logic. Using this card in the same Developer Environment may speed the emulation to near real time.

## **Reconfigurable / Adaptive Computing Systems, Super Computers**

The FPCL device can implement not only common hardware, but also complicated applications / complicated functions normally implemented utilizing a processor type of hardware (e.g. CPUs, DSPs). The FPCL execution time is expected to be faster than that utilizing a very fast processor by an order of magnitude as the processor activates its commands one by one while the FPCL hardware is working in parallel on thousands of commands. For example: if the FPCL implements 2000 commands in a single clock and its frequency is 250 MHz, the equivalent command rate is 500 GHz (or 500,000 MIPS). Combining several FPCL devices to work in parallel, will lead to enormous computing power.

The application can be downloaded into the FPCL devices and the needed super real-time computing power is achieved. Nevertheless, even a small number of FPCL devices can convert a regular computer into a super computer.

The fractal like structure enables the FPCL device to be converted into a single cell, which is nothing but a Random Access Memory that can be mapped into a CPU memory.

The base for such a supercomputer implementation is a collection of several FPCL devices each of which is to be programmed to carry out a required operation. A host computer, which has a memory, stores the code that must be loaded into each device so as to allow it to operate. While one device is executing the application, a second is connected to the host as normal memory allowing previous results to be read there from or for new code to be loaded thereto, and at the same time the code for effecting the required connections and loading the required data may be loaded into the third device. Upon completion of an application by the currently running device, the states of each device changes, and the host is connected to the device that has just finished its task for reading the result and loading a new implementation.

By such means, each implementation may be effected in hardware which is much faster than can be done in a normal CPU using software, whilst the downloading of new data to the devices, which is time-consuming, is done in parallel with the operation of a different device and therefore represents a transparent operation not demanding any overhead in real-time.

This technique can be useful when a very fast real-time computing is needed (as in configurable computers). The host computer will change the task to be implemented as needed on one device, while in the other device it fetches the result of the previous instruction.

The FPCL Development Environment already has the ability to create FPCL objects - the foundation stones for an FPCL application – in runtime. These objects may be integrated into high-level language (such as C++) enabling the engineer to create, manipulate and use the FPCL components in this high-level language so as to enable automation of the described technique.

The same concept of changing the application by the running application itself can be implemented not only using a controller, but as part of an application implemented on a single device. To simplify the explanation – imagine the FPCL device is built out of several devices (the fractal-like structure enables this) use one as a controller and implement the above procedure. Such technique may be applied to evolvable hardware.

## **Time Sharing Applications**

FPCL is also exceptionally suited for time-sharing applications. A general counter is used to present the current active channel. Once an application implementation design is finished, a new cell is assigned to each application cell to store the application cell output for each clock so as to remember this channel state for the next time it is active. In such manner we have shown an example of more than 20,000 FIR Filters implemented on a single (medium size) FPCL device.

## Full Custom Implementation

For full details, please refer to the patent publication (see [www.cellot.com](http://www.cellot.com)).

The idea is to separate the chip into smaller devices. A 64M Bytes device is proposed here. As the amount of connections needed for connecting any point to any point in such a device is very small, a smaller delayed matrix will be used. Each device is implemented in a closed small area. Integrating the two parameters: smaller area and smaller delay in the cross connect, results in increased (about double) speed.

To connect the devices without degrading performance (compare to a non separated chip) another smaller delayed matrix is required. Therefore, the overall delay – compared to a non-separated chip – is increased (about 1:1.25), as the delay of two smaller delay matrixes is bigger than the delay of one original matrix.

Please note that only the original frequency and the doubled frequency are permitted, therefore none of the FPCL features is compromised in this implementation including the emulator, which retains the One-To-One feature. The Developer Environment is designed in such a way that its simulation speed decreases only to the parts for which the clock is doubled! As the work on the doubled speed device is not completed, it is safe to assume that the typical speed will be 2.5 nSec rather than the calculated speed (2.3 nSec).

## An Implementation Example: FFT

Several FFT implementations have been investigated for different sizes and different lengths, for speed optimization and for size optimization, which shows the great advantage of the FPCL technology. Below is the result for a 1024 points, 16 bit complex (16 bits real, 16 bits imaginary) FFT implementation. Investigating some algorithms, it has been found that the Cooley-Tukey radix-4 algorithm is best for speed, while the Cooley-Tukey radix-2 algorithm is best for size. The latest could implement the whole FFT in a few cells, but would take a long time for the transform. If time-sharing is executed, dozens of thousands of transforms could be performed, but the time for any transform would be counted in seconds.

The implementation presented below has been designed for speed optimization therefore the radix-4 algorithm has been chosen. This implementation is a “straightforward” implementation result and we may implement the FFT in less real-estate (same transform time) or faster (same real-estate).

Please note that the transform time presented IS NOT the best transform time the FPCL can achieve even with this “straightforward” implementation, but is the transform time achieved for the number of cells mentioned. The transform time is divided by two each time the real estate is duplicated.

Based upon the Radix 4, Cooley-Tukey Algorithm and no accuracy compromise, a sample of the indicative results for the 1M Bytes chip is presented below. For other chip sizes the Transform Time is directly linked to path delay.

As the FFT can be implemented utilizing core only, the results are provided for such case. Anyway, as normally common PLD do not implement the FFT using core only but need the assistance of memories and DSP blocks (or multiplier or multiplier-accumulators) which may be added into such PLDs, some results are shown in case the FPCL utilized such accessories. Again, the transform time is divided by two each time the resources are duplicated.

FFT TRANSFORM TIME (Micro-Seconds)	NUMBER OF K Bytes USED	REMARKS
4.0	450	FPCL Standard Cells Implementation. One butterfly.
2.5	450	FPCL Full Custom Implementation. One butterfly.
24.5	80	FPCL Standard Cells Implementation.
15	80	FPCL Full Custom Implementation.
16.4	20	Using 4 Hardwired Multiplier-Accumulators and Standard Cells.
10.2	20	Using 4 Hardwired Multiplier-Accumulators and Full Custom Cells.

Better transform times may be achieved. For example, using 24 multiplier-accumulators and 150 K Bytes (or 900 K Bytes, utilizing core only) Full Custom implementation will lead to 1.25 microseconds transform time. This transform time is divided by two each time the resources are duplicated.

There is room to achieve better results, which have not yet been investigated:

- Compromising accuracy in the same manner other providers do,
- Using 16 bits Floating Point so the multiplication is a lot simpler,
- Optimizing the indicated implementation,
- Investigating other algorithms possibly better suited for FPCL (WFFT),
- Using LNS (Logarithmic Number System).

## Significantly Increasing the Application per Die Size Ratio

### *1.9 The FPCL as a Programmable Logic Device (PLD)*

A common Programmable Logic Device (PLD) is built of thousands of similar parts each of which can implement a tiny portion of an application. This part is called a Cell or Logic Element (LE) or Look Up Table (LUT). When the LE is capable of implementing any gate logic (and / or / xor etc.) the PLD is called a Gate Array. When the PLD has the ability to be programmed 'on board' (in the field) it is called "Field Programmable". Therefore common PLDs are called Field Programmable Gate Array (FPGA) even if their basic logic element is not a gate at all but a small piece of memory. Applications that are implemented utilizing the PLD are broken into these tiny portions. Each cell is configured to implement one portion.

### *1.10 The Interconnect and the Cell Size*

Each cell can be connected to each other cell via a Matrix. The smaller the cell is - the larger the matrix is. For example: If an FPGA having 100,000 Logic Elements (LEs) each with 4 inputs and 1 output, then 400,000 connections are needed to connect each LE to each LE. As a programmable device each connection must have a control to enable the user to choose the needed connections. A rough estimation may lead to an interconnect size area which is about the size needed to implement 400,000 bits of memory. As a 4 to 1 LE is equivalent to 16 bits of memory, 100,000 LEs are equivalent to 1,600,000 bits of memory (=200K Bytes). Compare the two numbers to realize that the area for logic should be less than 1/50,000 of the die size.

To solve this problem, such a PLD may have a few levels of routing hierarchies, giving up a lot of connections to save die area. Because of these routing hierarchies the propagation delay for a signal routed from one LE to the other now vary and depend on the route chosen. Another aspect is data carriers: as there are only 100,000 outputs (in the upper example), only 100,000 signals are really needed to be routed. On the other hand, the user may want to route a signal to each one of the inputs of a LE. Therefore, fewer connections are implemented and complex synthesis software programs are used to decide which LE is used for each portion of the application. Nevertheless, the major part of such an FPGA die area is dedicated to interconnect resources.

Another aspect of the cell size is the utilization. There is an assumption that the smaller the LE is - the better the granularity and the application can be broken more efficiently into the device. It is easy to show many applications where this assumption is incorrect. As it is complicated to divide an application into so many parts, special Hardware Design Languages are used in order to compile the user commands into the PLD code. One may compare using the LE directly to writing in assembler language.

The FPCL is based on a collection of cells in a recursive, fractal like structure. Such a structure is similar to a cauliflower structure: Every flower having the same structure of the whole and every part of it having the same structure also. In the FPCL, each cell size may vary according to the application needs. Each cell is similar to any other cell although its size may differ. This structure utilizes a lot bigger cells, which enable savings of a lot of interconnection space while increasing the logic space. For example, the larger size FPCL device is designed to have **2.5M Bytes** for logic cells. In the FPCL, most of the die area is used for logic although all connections are allowed. To learn more about the FPCL interconnection please refer to the "FPCL technology white paper" of Cellot.

It is not only that the FPCL structure enables one level of interconnect, therefore fixed and predictable propagation delay, but it is quite simple to emulate the FPCL device on a regular PC. These unique facts enable most of the features described in this Article.

### ***1.11 The Cell as a Memory***

In order to specify the application portion, each cell is programmed. The FPCL cells can be programmed to create the needed application, and can be programmed as result of the running application by the application itself. In other words, each cell (in any reprogrammable size) can be used as an application portion (Look Up Table or LUT), as a regular memory or as both. This is the basis for Evolvable Hardware features.

### ***1.12 Conclusion***

As the FPCL logic part is more than 10 times larger than size of the current FPGA's logic part (compare 2.5M Byte to 200K Bytes), as the cells can be combined to form larger cells thereby adapting the cell size to the application needs, and as each cell (in each size) can be used as a regular Look Up Table, as storage (Memory) or as both, the application per die size is higher the more complex the application.

## **Fault Tolerance and Failure Immunity**

FPCL technology can “map itself” and define faulty cells. Hence, what seems to be a faulty chip in other methods can be used seamlessly. Having said that, significant price reduction is achieved due to the usability of “faulty” dies and hence wafers’ higher production yields. The manufacturing cost of FPCL chips is expected to be low and grow linearly (rather than exponentially) with die size. Since Cells and Switches are replaceable, most of the chip has redundancy.

Any cell or few cells left unused intentionally, may restore the device which otherwise would be considered faulty. This increases the yield significantly and reduces the device-manufacturing price. Moreover, the FPCL technology can “map itself” and define faulty cells even while in active application. This enables an on-the-fly failure discovery and repair which is not related to the running application.

In order to activate the above feature in application time, storage for the downloaded information is held in parallel to the data, which has already been downloaded to the FPCL Cells. For example, for a 1M Byte FPCL Device, 1M Byte of memory should store the data. The immunity of such storage is independently considered.

## **Migration to ASIC**

The migration from FPCL to ASIC entry is automatic, quick, and will not require a new technology that frequently creates new “vicious circles” of re-design and debugging. Nevertheless, our solution meets the trend in the market of extended usage in Programmable Devices as a replacement to ASIC. Our low cost and high performance chips, will give us the advantage in competing on certain ASIC market share.

An ASIC – FPCL combination that will enable Application Specific with programmable capabilities can be customized to clients’ needs. As seen in the FFT example, applications can be implemented using core only, which provide advantages over technologies that must have accessories to implement complex applications.

## **Evergreen Design**

The FPCL technology offers an “Ever Green” solution. Unlike most of the existing Electronic Designs, our hardware architecture, the core-IPs and the user developed applications are chip geometry independent. Along with future industry technological evolution, and with the introduction of new geometries, a new design will not be required. An FPCL complex design of today may be “as is” ported and implemented on future FPCL devices, years from now. This feature will provide customers with incredible savings and incentive to use the FPCL devices. This feature results from the fractal like architecture: when smaller cells are created in one geometry, these smaller cells can be combined to create larger cells. On the other hand – larger cells can always seamlessly function as smaller cells.