Programmable Cellular Logic: Past, Present, and Future

Russell Tessier*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

September 10, 1994

Abstract

In recent years there has been renewed interest in the use of flexible arrays of fine-grained computational elements to perform logical functions. This interest has been sparked by the technological advancement of very large-scale integrated circuits in the form of field-programmable gate arrays (FPGAs). Although logic mapping to regular arrays of gates in integrated chip form has only been practical for the past ten years or so, a number of reconfigurable architectures were designed and analyzed during the 1960's and early 1970's.

This paper makes a historical analysis of these structures and discusses changes in technology and architectural trends as they pertain to cellular arrays since the 1960's. Direct comparisons of various cellular array architectural features with contemporary programmable logic features are made as well as an analysis of potential future directions for FPGA development based in part on knowledge gained from the study of cellular arrays.

1 Introduction

Since the inception of integrated circuit technology in the late 1950's there has been interest in mapping reconfigurable implementations of switching functions to a silicon substrate. A simple, reconfigurable logic block in the form of an integrated circuit is recognized as a straightforward mechanism for enhancing system testability and flexibility. The introduction of batch fabrication technology in the 1960's introduced limited feasibility for mapping these functions to regular arrays of logic cells implemented as integrated circuits. These structures, called *cellular arrays*, typically consisted of a fixed interconnection of cellular elements of a limited logical variety. While a great deal of theoretical research was done in determining cell parameters such as cell size, interconnect, and logic mapping, integrated circuit technology was not yet advanced enough to support these ideas.

Interest in logical arrays of cellular elements was renewed with the introduction of the first field-programmable gate array (FPGA) in 1986 [26]. To this point in time, FPGAs have most typically been used as reconfigurable "glue" logic chips in digital systems. Only recently have these devices been recognized as computation elements.

*Area Exam. February, 1994.

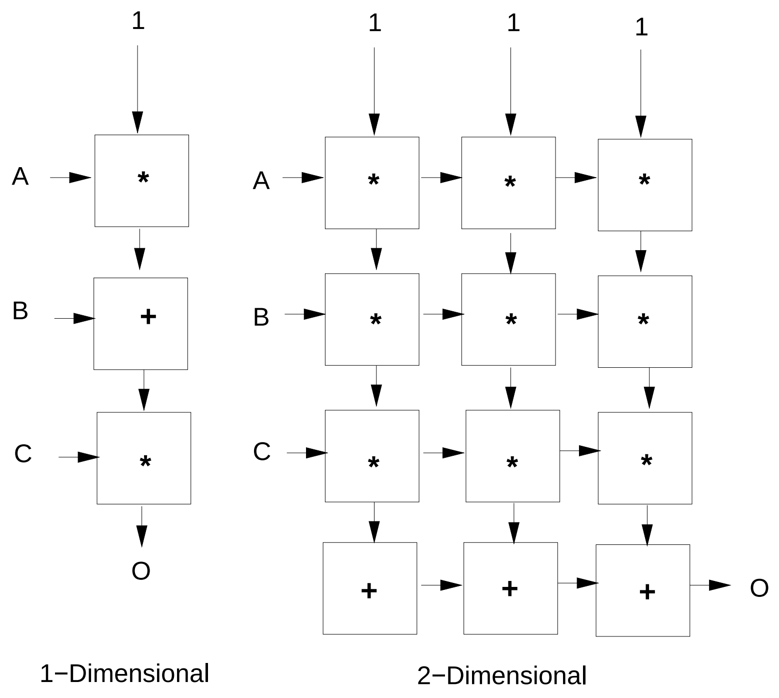

Figure 1: Typical Cellular Array Structures

The goal of this paper is to constrast cellular array architectures and implementations from the 1960's and early 1970's with FPGA architectures of today and indicate potential future directions of FPGA development based in part on prior work done with cellular arrays. It will be seen that while technology has changed dramatically in the past thirty years, some of the features that made cellular arrays attractive are the same ones that fuel interest in today's reprogrammable logic devices. After the issues of reconfigurable logic have been examined, a proposal for a potential integrated system based on this logic will be presented.

2 Background

The largest motivating factor behind cellular array development in the 1960's was a desire to reduce circuit costs and increase chip testability by increasing the amount of available logic per chip when compared to discrete MSI solutions [8]. Rather than partitioning a wafer into discrete logic packages, the goal was to locate a number of dies in a single integrated package.

As seen in Figure 1, most cellular arrays consisted of either a one-dimensional cascade or two-dimensional mesh of discrete logic cells. Arrays of cells consisting of several logic gates and perhaps a storage element were classified as microcellular arrays [8] while arrays with cells of large numbers of gates (typically more than five) were known as macrocellular arrays. This report focuses primarily on microcellular arrays due to their similarity to many contemporary reprogrammable architectures. Two-input cell Maitra cascades [12] and variants [6] are examples of one-dimensional linear cellular arrays. These cascades supported decomposition of switching functions to cells of one of six possible functions of two input variables such as and, or, and xor. Cascades with two intercell leads were found to implement every switching function at a worst case growth rate of O(n2^n), where n is the number of input variables.

Linear cascades of fixed cell types could be aligned vertically to form a two-dimensional mesh. The result of each vertical cascade could be summed together in either sum-of-products [11], product-of-sums [20], or Reed-Muller form [13] to generate a final result. Virtually all arrays and cascades used unidirectional logic flow (e.g top to bottom or left to right). Outputs along the bottom and left edge of the array could be used as feedback inputs for synchronous arrays or as inputs for neighboring arrays. Although logic minimization techniques could be applied to some functions mapped to these array styles [14], array growth rates of at least O(2^n), where n is the number of input variables, were found in these types of array structures.

Most research in cellular arrays was aimed toward determining efficient cell sizes, interconnection, and mapping heuristics. Not only were these parameters constrained by theoretical limits, they were also constrained by technological limitations of the period. This may explain a tendency toward very fine grain cell structures. Only a handful of designs were actually implemented in silicon [11], [8]. Some of these architectures used mechanical switches or light controlled photocells to program interconnections between cells or the function of the cells themselves.

Specialized configurations of reprogrammable logic were recognized as appropriate for certain types of applications. For example, a programmable array of elements was constructed for sorting a set of binary numbers [15] and determining threshold values of input sequences [16]. Some forms of content-addressed memory may also be considered specialized arrays of reconfigurable cells.

In many respects cellular arrays may be considered the stepping stone for modern technologies other than FPGAs such as programmable logic arrays (PLAs), systolic arrays [24], and cellular processors for special purpose applications.

Contemporary FPGAs are logic devices capable of holding thousands of mapped gates of logic. Like cellular arrays, FPGAs are characterized by discrete logic cells embedded in an interconnection structure. Look-up table (LUT) based FPGAs [26] have cells which contain universal logic functions of a number of inputs that produce a single output. Look-up tables are typically configured by SRAM storage loaded when power is applied to the device. Other programmable devices contain interconnected blocks of and-or logic.

While cellular arrays and FPGAs are similar in nature in many respects there are some distinct differences in design and structure. The following sections examine some of the assumptions made by cellular array researchers in the 1960's and evaluates them from a historical perspective. Where appropriate, comparisons will be made to contemporary FPGA architectures. For each issue, comments will be made as to possible future research directions as FPGA architectures evolve.

3 Interconnection and Logic Mapping

Some assumptions made by cellular array designers are not consistent with today's view of technology. Contemporary FPGA interconnection structures differ from most cellular array counterparts in that FPGA interconnect is generally programmable while cellular array interconnect was primarily fixed. Due to technological constraints, virtually all cellular arrays used point-to-point nearest-neighbor interconnect (usually unidirectional) while most FPGAs mix an assortment of local interconnect between neighboring cells and global busses that interconnect cells some distance away. In fact, one could claim that the largest difference between cellular array designers and FPGA designers is that while cellular array designers were concerned with optimizing the functionality of cells to reduce logic within a fixed

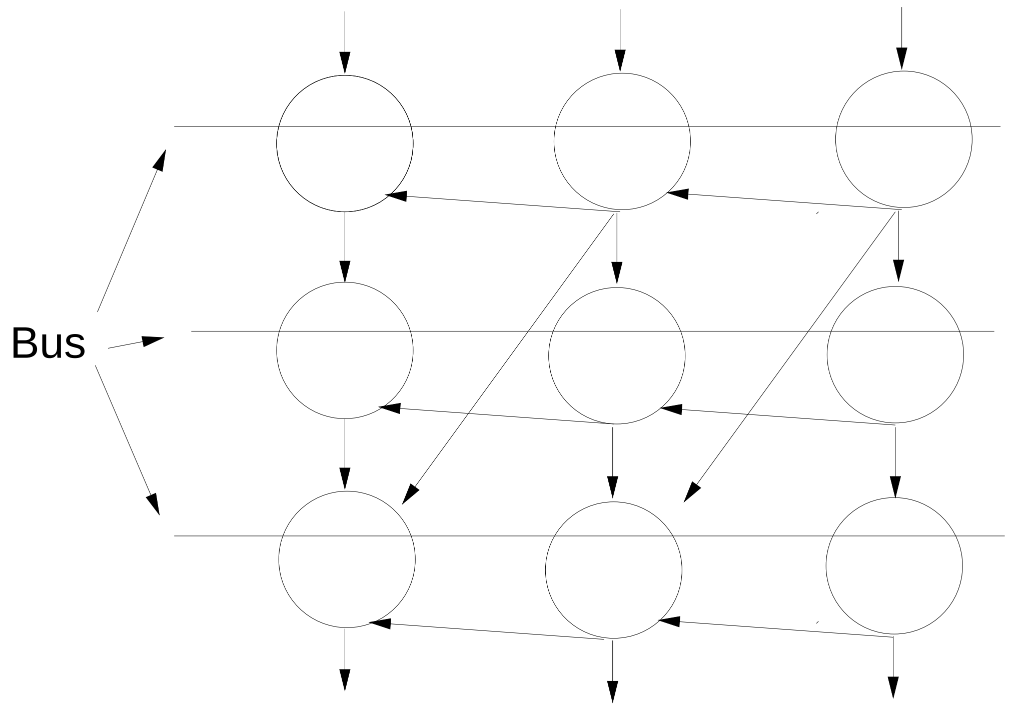

Figure 2: Cobweb Cellular Array Interconnect

interconnection structure, FPGA designers are concerned with optimizing the cell structure itself to reduce routing interconnection.

Due to advancements in logic minimization methods and VLSI technology it is not suprising that most cellular array architectures are not scalable by today's standards. While linear cascades of two-input elements are interesting from a theoretical point of view, it would make little sense to implement them in today's technology. Structures such as horizontal arrays of minterms were chosen from a ease of implementation standpoint rather than any theoretical rationale. Although such an implementation grows at O(n2^n) with respect to the number of input variables, it was easy to implement in a regular array.

While the circuit implementation of logic is largely dependent on the nature of the logic to be implemented, some generalizations can be made. Many cellular array mappings can today be recognized as mappings of the results of two-level logic minimization techniques [27]. Such results are classified as two-level logic because they can be rearranged into a form equivalent to sum-of-products format (eg. ab+cdf+ac). While this type of representation is generally efficient for a small numbers of input variables (¡ 10) larger input counts can lead to area inefficiency.

While nearest-neighbor only cell interconnect was easy to implement, it did lead to large numbers of cells being used as pass-through interconnect thus wasting logic area. Early research in cellular arrays showed that adding additional routing capacity to nearest-neighbor interconnect could reduce cell count. Minnick [7] showed that nearest neighbor array cell counts could be reduced by 30% by augmenting cells with connections to neighbors and adding horizontal busses as shown in Figure 2. While adding additional lines reduced cell count, it also made the problem of mapping logic to the array untenable without manual intervention.

.

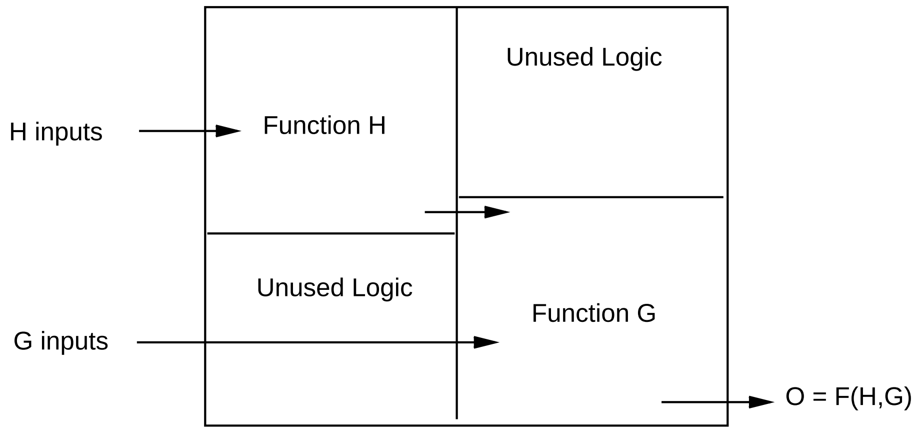

Figure 3: Increased Functional Complexity

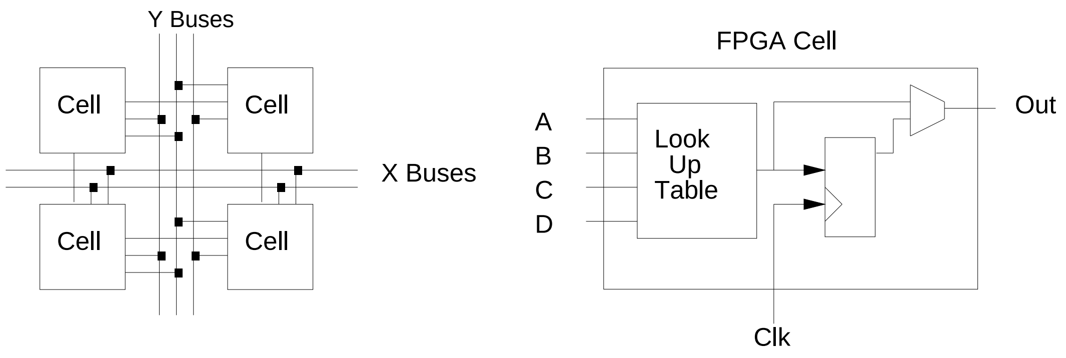

Figure 4: Typical FPGA Cell and Interconnect

The method of implementation described above breaks down for large circuits with many inputs. As seen in Figure 3, complex functions could be decomposed into less complicated switching functions in neighboring array “tiles” [5]. Due to the unidirectional nature of signal flow in the array this technique could leave large portions of the array unused. Some designers suggested feeding synchronous array outputs back into the array [11] along array edges although this technique is clearly not scalable.

The mapping techniques used for large cellular arrays are ineffiecint due to properties of logic locality. Minimal representations of logic circuits can typically be represented by tree-like clumps of logic rather than fixed uni-directional grids. While small circuits have enough locality to fit efficiently into such representations, larger circuit representations become more sparse and necessitate more interconnection resources to connect to neighboring logical bundles. Thus, by allowing a choice between localized and more global interconnect FPGAs can represent logic more efficiently.

The mapping of logic to a fixed cellular array interconnection is in direct contrast to logic mapping in today’s programmable FPGA architectures. A typical example of an interconnection of cells for an FPGA is shown in Figure 4. Each cell may consist one or more look-up tables and an associated D flip flop. Each cell output may be connected to signals existing between nearest neighbors and

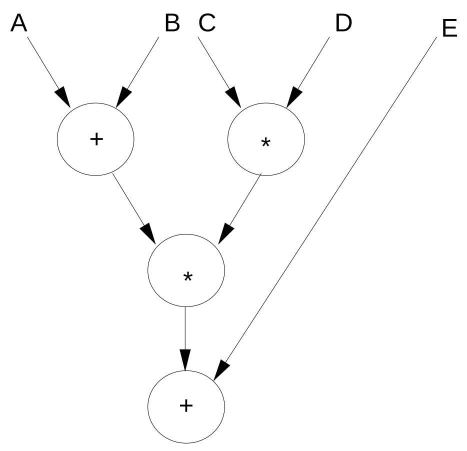

Figure 5: Decomposed Logic Tree

a variety of global busses interconnecting distant cells. This satisfies the need for both local and global interconnect. Typically, today's architectures separate the translation of logic to discrete cell arrays into three stages: the mapping of logic to cells, the placement of cells in the array, and the routing of interconnection between the cells. Switching functions are first decomposed into "trees" of logic (Figure 5). Trees with appropriate numbers of inputs may be mapped to a look-up table. This technique, referred to as multi-level logic minimization, typically creates a reduced gate count version of the circuit when compared to two-level implementations. For look-up table based architectures a mapping is made to cover all sub-trees containing the appropriate number of inputs and outputs. Once logic has been mapped to blocks, algorithms such as simulated annealing are used to find a placement requiring minimal routing. Routing of the interconnection network is performed as a final step.

Recent research [10] has shown that the use of very fine-grained cell sizes, such as those consisting of one or two gates, are actually poor choices for reducing chip area and delay in reprogrammable devices targetted toward random types of logic (non-pipelined, etc). This is due to an increased number of cells necessary to implement logic and their associated intermediate routing resources. For example, a four input look-up table is considered to be equivalent to approximately 100 NAND gates of logic. Thus the NAND cell would have to be more than 100 times smaller in area than the SRAM-based look-up table to overcome this inefficiency and make up for additional required routing between the gates. Additional routing wires add more capacitance to the connections thus adding to circuit delay. In this case small scale locality can be used as a benefit to clump associated circuitry together.

It is interesting to note that some non-reprogrammable FPGA architectures have cell structures that implement two-level logic in the form of a sum-of-product plane with a moderate number of inputs [4]. This type of structure takes advantage of the locality of limited two-level implementations while allowing the building of larger structures through a more general-purpose interconnection array.

Much of the future of reconfigurable logic depends on the improvement of high level synthesis tools for mapping logic. Work in the area of compiling hardware description languages has enjoyed moderate success although a number of improvements remain. Rather than providing minimization at a low level, the compilation of high level structures to reconfigurable devices should provide for identification of recognizable circuit constructs in high-level form and subsequently map these constructs to previously optimized versions of the structure created for a specific target device. This higher-level view is analagous to a software compiler generating optimized code for a specific construct in a high-level language. This technique creates structures already optimized for a specific architecture and can be used to reduce the need for additional logic resources and routing overhead.

Structures amenable to direct compilation include adders, counters, and comparators. Hardware compilers such as Synopsys [3] are just beginning to use libraries of optimized structures appropriate under different timing and logic area consideration.

FPGA devices using only local communication have been designed with mixed results in terms of logic mapping efficiency [21] indicating that at least some non-nearest neighbor interconnection is advantageous. High-level recognition of specific functional structures may lead to the embedding of special-purpose functional blocks within the cellular interconnection structure. The cost of placing these blocks as non-reconfigurable logic should be balanced with their overall usefulness from a system perspective. For example, the embedding of a full adder that is smaller and faster than an adder implementation in cellular blocks is not efficient if it is rarely used.

4 Testability

One of the largest motivating factors in the development of cellular arrays was a desire to overcome batch fabrication defects by configuring around damaged cells on a silicon substrate. Logic mapping software could be informed of silicon wafer defects and map logic to unaffected cells [8].

Proposals [17] for adding additional logic to cells to allow for in-circuit testability compensated for frequent chip failures. IC test capabilities have progressed to the point that test patterns of all possible SRAM-based FPGA configurations can be generated following FPGA manufacture without the need for special circuitry in cells [22]. It is currently more cost effective to discard faulty dies and chips rather than selling the parts with software to configure around errors. As reconfigurable devices become more complex and costly to produce it may make sense to reinvestigate this option. Before programming the user could use software to analyze the device for faults. Device functionality could then configure the device to ignore non-functional cells and interconnect while allowing the remaining functional cells to be used to the best possible capacity.

Although some FPGAs allow for limited testability of devices through hardware support such as JTAG boundary scan, no current devices contain special circuitry for in-circuit test of all cells. It seems unlikely at this time that there is need for portions of a reconfigurable device to check other the viability of neighboring cells. Upon detection of a chip failure external to the chip, a test of the entire chip contents should be made instead.

5 Programmability

Suprisingly, even though the technology supporting cellular arrays has changed radically in the past thirty years, most of the techniques for programming the devices have stayed the same.

The technique Maitra [12] proposed of using a final layer of metalization for programming the functionality of cells is directly analagous to mask-programmable gate array (MPGA) techniques supported today [22]. While MPGA designers select an interconnection pattern for a fixed set of gates, cellular cascade designers could select a cell function for a fixed interconnect cell interconnect.

In an alternate approach, cutpoint cellular logic [11] was programmed by passing a larger-than-normal current through configuration fuses in each cell through external connections. This is practically the same method used by Actel for programming discrete anti-fuses that configure cell functionality [1].

Several cellular array designers proposed a programming “arm” technique for configuring devices. This mechanism involves passing information from a source external to the chip through cells that have already been programmed to unprogrammed cells. [23] [17]. Essentially, each cell in the array was initially configured to set a load path to its neighbor on which configuration data could pass. After the last cell in the array had been configured, the “arm” could be retracted leaving cells programmed with its final functionality. This was also considered an effective way to test cell functionality and route around it if necessary. A similar approach suggested in [18] and used by Xilinx and others today is the formation of a dedicated shift chain of configuration bits for all cellular storage elements. This approach has the drawback of requiring logic to be reconfigured simultaneously.

For the future one could consider having multiple storage bits associated with each programmable location. When an external stimulus is applied all or part of the configuration is switched to the alternate one. Additionally, it should be possible to reconfigure a portion of the device without affecting the remaining configuration. This might be done by treating groups of configuration bits as addressable byte memory locations accessible from outside the chip.

6 FPGA Computational Extensions

Recently, reconfigurable logic has been recognized as being useful not only as combinational “glue” logic in digital systems but also as computational elements in their own right. Just as special purpose cellular array systems for sorting and arithmetic were proposed in the 1960’s, a number of special purpose systems for performing tasks such as recognizing genetic patterns [9] have been developed as special purpose coprocessors for microprocessor environments [19]. Virtually all of these systems require the user to hand develop algorithms in either hardware description languages or using schematic capture programs.

Before reconfigurable hardware can be widely used in microprocesor systems a great deal of work must be done to develop better software support. Note the microprocessor system shown in Figure 6. System performance might be accelerated by the development of software to configure a special purpose coprocessor to perform tasks frequently performed in software and costly to implement on a microprocessor. These tasks may be identified by a software profiler with a group of instructions in the instruction stream replaced by a bus access to the reconfigurable hardware configured to perform the same functionality.

Consider, for example, a microprocessor system without any special purpose coprocessors that occasionally is required to perform special purpose operations such as signal processing applications. A configuration compiler could be used to identify tasks in sequential code that require long computation time on a sequential processor and then generate a configuration for the associated reconfigurable hardware to perform the task instead. This requires the compiler to perform the following steps:

1. Search through code attempting to identify sections that are performed frequently and exhibit parallelism that may be accelerated by reconfigurable hardware.

2. Transform the selected software code into a reconfigurable hardware configuration that performs the desired task.

3. Replace the appropriate lines of code in the program with a call to the reconfigurable hardware.

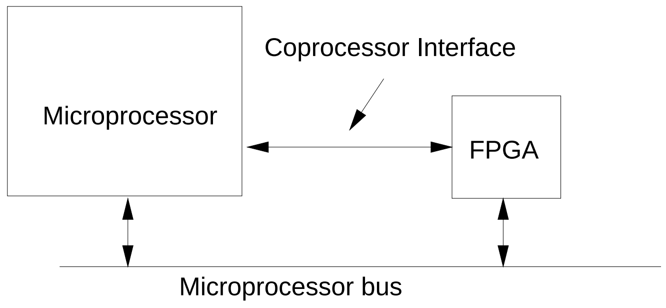

Figure 6: Microprocessor System with Reconfigurable Coprocessor

Of the tasks listed above, item 2 is by far the hardest. Certain types of tasks such as pipelined systolic operations are relatively straightforward to implement as a sequence of operations on reconfigurable hardware while operations such as loops may require the addition of extra control logic to coordinate data flow. It is possible to implement a sequence of operations in many different ways in reconfigurable logic. One could consider creating a small microcoded engine with functional units and a control unit more closely resembling a processor architecture or, alternatively, one could implement an operation as a linear sequence of functions with less control overhead.

Choices are further blurred by the capability to trade function evaluation time for space inside reconfigurable devices. For example, arithmetic operations inside reconfigurable hardware may be serialized to use fewer functional resources but require greater computation time.

The use of reconfigurable logic for computation will be greatly enhanced with the introduction of devices containing multiple configuration bits for each configurable point in the device. This feature will allow a processor to generate several configurations to accelerate computation and select between them rapidly as they are needed by the software.

7 Conclusion

Since the introduction of integrated circuits, engineers have searched for ways to implement and use reconfigurable logic. While initial efforts focused primarily on the development of hardware technology, more recent efforts have focused on improving software techniques for mapping logic to reconfigurable devices. While hardware features of reconfigurable logic such as grain size and interconnection structure have remained fairly stable for the past ten years, software techniques of mapping to this logic have changed radically and will continue to do so.

As software techniques improve reconfigurable devices will become more viable resources for special purpose computation. While today designers must hand code and optimize special purpose operations targetted for reconfigurable hardware, it is possible to envision a sophisticated compiler performing a similiar task. As outlined in this paper, it is clear that many tradeoffs exist in the construction of such a tool.

References

[1] Actel Corporation *ACT Family Field Programmable Gate Array Data Book*, April 1992.

[2] Concurrent Logic, Inc. *CLi6000 Series Field-Programmable Gate Arrays*, May 1992. Revision 1c.

[3] Synopsys, Inc. *Synopsys User's Manual*, June 1993

[4] Altera, Inc. *Altera Data Book*, December 1993

[5] S.B. Akers A Rectangular Logic Array In *Proceedings - IEEE 1971 Switching and Automata Theory Symposium*, October 1971.

[6] R.A. Short Two-Rail Cellular Cascades In *Proceedings - AFIPS Fall Joint Computer Conference* , Fall 1965.

[7] R.C. Minnick Cobweb Cellular Arrays In *Proceedings - AFIPS Fall Joint Computer Conference* , Fall 1965.

[8] R.C. Minnick A Survey of Microcellular Research. Journal of the Association of Computing Machinery. 14, 203, April 1967

[9] M. Gokhale, et.al Building and Using a Highly Programmable Logic Array Computer, January 1991.

[10] S. Singh, et al. The Effect of Logic Block Architecture o FPGA Performance IEEE Journal of Solid State Circuits. 27, 3, March 1992.

[11] R.C. Minnick Cutpoint Cellular Logic IEEE Trans. Electronic Computing. EC-13, December 1964.

[12] K.K. Maitra Cascaded switching networks of two-input flexible cells. IEEE Trans. Electronic Computing. EC-11, April 1962.

[13] K. Saluja and S. Reddy Easily Testable Two-Dimensional Cellular Logic Arrays IEEE Transactions on Computers. C-23, November 1974.

[14] A. Mukhopadhyay Unate Cellular Logic. IEEE Transactions on Computers. C-19, February 1969.

[15] W.H. Kautz Cellular Logic-in-Memory Arrays IEEE Transactions on Computers. C-19, August 1969.

[16] W.H. Kautz A Cellular Threshold Array IEEE Transactions on Computers. C-16, August 1967.

[17] F. Manning. Automatic Test, Configuration, and Repair of Cellular Arrays *MIT/LCS Technical Report-151*, June 1975.

[18] R. Shoup. Programmable Cellular Logic Arrays *Ph.D. thesis, Carnegie-Mellon University*, March 1970.

[19] P. Athanas An Adaptive Machine Architecture and Compiler for Dynamic Processor Configuration *Ph.D. thesis, Brown University*, 1992

[20] W.H. Kautz. Programmable Cellular Logic. In *Recent Developments in Switching Theory*, Academic Press, 1971.

[21] J.F. Beetem. Simultaneous Placement and Routing of the Labyrinth Reconfigurable Logic Array. In *FPGAs*, W. Moore and W. Luk, eds, Abingdon EE&CS Books, 1991.

[22] S. Trimberger. SRAM Programmable FPGAs In *Field Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

[23] S.E. Wahlstrom. Programmable Arrays and Networks. *Electronics*, 40, December 11, 1967.

[24] H.T. Kung. Why Systolic Architectures? *Computer*, 15:1, 1982

[25] J. Rose,et al. Field Programmable Gate Arrays, Kluwer Academic Publishers, 1992.

[26] Xilinx, Inc. *The XC4000 Data Book*, Aug. 1992.

[27] T. Sasao. *Logic Synthesis and Optimization*. Kluwer Academic Press, 1993.

[28] F. Hennie. *Finite-State Models for Logical Machines*. John Wiley and Sons, 1968.