AN ARCHITECTURE FOR ELECTRICALLY CONFIGURABLE GATE ARRAYS

Abbas El Gamal, Jonathan Greene, Justin Reyneri,
Eric Rogoyski, Khaled El-Ayat and Amr Mohsen

Actel Corporation
320 Soquel Way
Sunnyvale, California 94086

ABSTRACT: A new architecture for electrically configurable gate arrays using a two-terminal anti-fuse element is described. The architecture is extensible, and can provide a level of integration comparable to mask-programmable gate arrays. This is accomplished by using a conventional gate array organization with rows of logic modules separated by wiring channels. Each channel contains segmented wiring tracks. The overhead needed to program the anti-fuses is minimized by an addressing scheme that utilizes the wiring segments, pass transistors between adjacent segments, shared control lines, and serial addressing circuitry at the periphery of the array. By providing sufficient wiring tracks segmented into carefully chosen lengths and a logic module with a high degree of symmetry, fully automated placement and routing is facilitated.

1. Introduction

Mask programmable gate arrays offer the architectural flexibility and efficiency to integrate thousands of gates, but require long development time and high non-recurring engineering costs. On the other hand, the convenience of field programming is available with programmable logic device (PLD) technologies, but their architectures have not allowed integration of a wide variety of applications exceeding a few hundred gates [1,2].

We describe a novel gate array architecture [3] which combines the flexibility of mask-programmable arrays with the convenience of field-programmability. Its implementation is made possible by a two-terminal electrically programmable anti-fuse offering low "on" resistance and small area. The anti-fuse is so called because it irreversibly changes from high to low resistance when "blown" by applying a programming voltage across it. The anti-fuse, or fuse for short, is further described in [4].

The architecture supports a design style similar to conventional gate arrays, including fully automatic placement and routing algorithms attaining 85 to 95 per cent utilization. This required considerable emphasis on symmetry and routability, which we touch on below.

2. Programmable Interconnect Architecture

The general architecture, shown in Figure 1, exhibits the familiar gate array organization: rows of logic cells interspersed with routing channels. There are, of course, several key differences.

The tracks in the channels are not simply empty areas in which metal lines can be arranged for a specific design. Rather, they contain pre-defined wiring "segments" of various lengths. Other wiring segments pass through the channels vertically. Each input and output of a logic module is connected to a dedicated vertical segment. Other vertical segments just pass through the modules, serving as feed-throughs between channels. (The number and lengths of segments in Figure 1 are only suggestive and can be varied).

A fuse is located at each crossing of a horizontal and vertical segment. Programming one of these "cross fuses" provides a low resistance bi-directional connection between the segments. Other fuses are located between adjacent horizontal segments within a track. When blown, these "horizontal fuses" connect the two segments to form a longer one. (Although not shown in the diagram, fuses may also be placed to connect adjacent vertical segments).

In order to program a fuse, we need to apply high voltage across it. This is accomplished by an efficient addressing scheme that uses the wiring segments themselves, pass transistors connecting adjacent segments, and control logic at the periphery of the array. Fuse addresses are shifted into the chip serially.

As shown in Figure 1, each column of "horizontal pass transistors" connecting horizontal tracks is controlled by a shared "horizontal control" line running across the array. Each row of "vertical pass transistors" is controlled by a "vertical control" line. The peripheral circuitry can drive the control lines and the segments at the end of each track.

Horizontal fuse programming is quite simple. In the example of Figure 2, we apply programming voltage V_{PP} across the fuse F_1. All horizontal control lines except the one in the column containing F_1 are turned on by connecting them to V_{PP}, and the appropriate track segments are driven to *GND* and V_{PP} as shown. (Vertical fuses, if present, are programmed similarly.) Cross fuse programming uses both horizontal and vertical control lines as shown in Figure 3. Segments not driven to either *GND* or V_{PP} are left precharged to $V_{PP}/2$. Thus the voltage across fuses not being

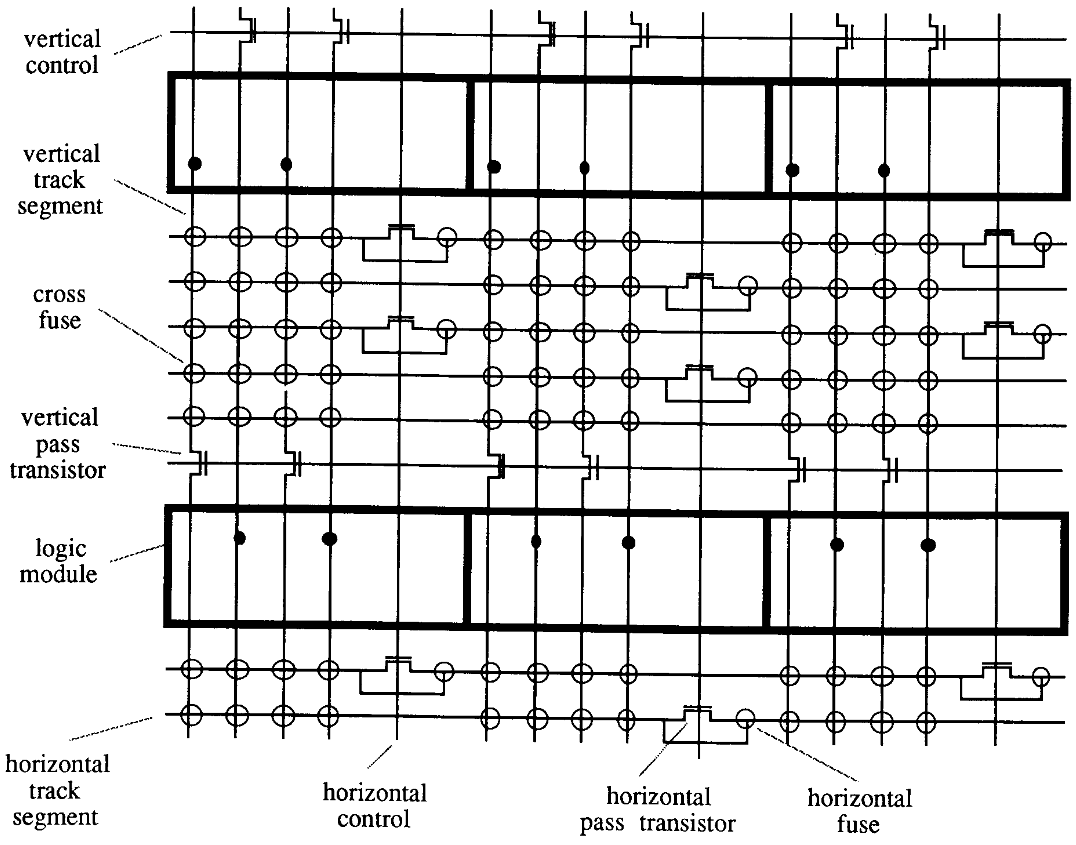

Fig. 1: Interconnect Architecture

programmed is either zero or $V_{PP}/2$.

Some care is required to assure that a unique fuse is addressed. Figure 4 shows how incorrect addressing allows diversion of current along a "sneak path", in this case programming fuse F_5 through previously blown fuses F_3 and F_4 instead of programming F_2. Sneak paths are eliminated by blowing the fuses in proper order. The order is dependent on the set of fuses to be blown.

The pass transistors and control logic are also used to test the chip prior to programming. For instance, the continuity of the segments in a track can be verified by turning on all horizontal control lines. All cross fuses can be stressed in parallel to check for shorts before programming by turning on all control lines, grounding all horizontal segments and driving all vertical segments to the stress voltage. The vertical segments and pass transistors can also be used to apply test patterns to the logic modules.

3. Choice of the Logic Module

As outlined so far, the programmable interconnection architecture could be used with a variety of logic modules. Which would be best? This turned out to be a very difficult question, involving subtle tradeoffs among routability, the logical capabilit) of the module as perceived by the user, and delays due to capacitive loading in the routing segments.

The complexity of the module must be balanced with the routing overhead. Mask-programmed gate arrays provide very flexible and efficient routing. They therefore use a simple four transistor cell. On the other hand, routing is very expensive in both area and delay with present programmable logic arrays. These generally use a module capable of implementing more complex functions [2]. The architecture outlined here has a cost of routing closer to a conventional gate array, suggesting a logic module of intermediate size. Because this is about the same complexity as conventional gate array hard macros, the designer can use a library like the familiar gate array cell libraries; there is no need to map logic into a more complex module. The table below lists several typical gate array macros and the numbers of four

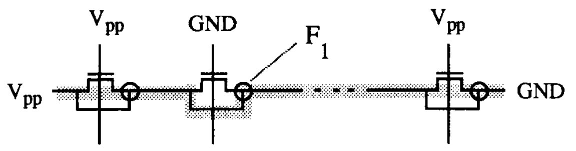

Fig. 2: Horizontal Fuse Programming

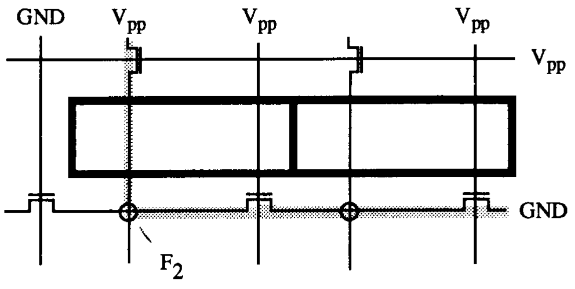

Fig. 3: Cross Fuse Programming

transistor cells and logic modules required to implement them.

macro	4 transistor cells	modules
3 input NOR	2	1
4:1 mux, non-inverting	6	1
D latch with clear	4	1
D flip-flop with clear/set	7	2

Our chosen module has eight inputs and a single output. Various macros, such as those in the table, are implemented by using an appropriate subset of the inputs and tying the remaining inputs high or low. Thus the module can implement all macros with two inputs, most with three inputs, many with four inputs, etc.

The module's output is connected to a vertical segment spanning several channels. Each input is connected to a short vertical segment spanning one channel. Four of these span the channel above the module, four the channel below. The use of short segments for the inputs reduces parasitic capacitance and hence delay.

Note that each input is accessible from either the channel above or below but not both. At first, this would appear to limit routability compared to a conventional "double-entry" gate array cell, which wires may enter from either direction. However, there is nearly always more than one way to implement a macro. By letting the router choose the implementation that uses inputs accessed from convenient channels, the benefits of full double-entry symmetry are approached or sometimes attained. The degree of symmetry

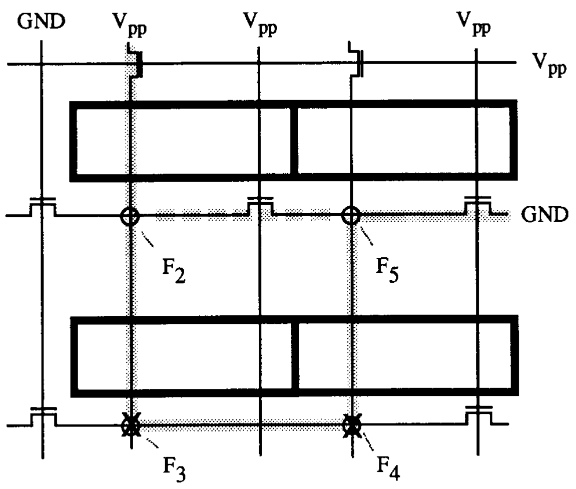

Fig. 4: A Sneak Path

depends on the mix of macros and the details of the module, and is an important criterion for choosing a module.

4. Routing

Figure 5 illustrates the routing of a net. The long vertical segment connected to the driving module's output is connected by cross fuses to horizontal segments, which in turn connect to the segments associated with module inputs. In the top channel, a horizontal fuse has been used to link two segments into a longer one.

The resistance of the blown fuses and the parasitic capacitance of the segments used form an R-C tree, with the driver of the net as the root. Note that each input is driven through a maximum of three and generally two fuses to limit the delay. The maximum number of fuses and the segment lengths (hence capacitances) can be altered to suit the chip dimensions and the resistance of the fuse technology.

To minimize clock skew due to differential routing delay, one entire track (or more if needed) in each channel is set aside for clock distribution. These tracks are connected directly to buffers, so that each input presents a similar load driven through exactly one fuse.

An interesting theoretical question is whether more horizontal tracks are needed in each channel here (where the lengths of the wiring segments must be predetermined) than in mask-programmed routing (where the wiring is customized for the design). Surprisingly, a high probability of routability is obtained with only a few tracks above channel density. This requires careful choice of the lengths of the segments according to statistics from a suite of designs, and taking advantage of the symmetry of the macros where possible. For example, observe that if macro 4 in Figure 5 permits its input to be routed from either the upper or lower channel, there is a better chance of finding a free horizontal segment to connect it.

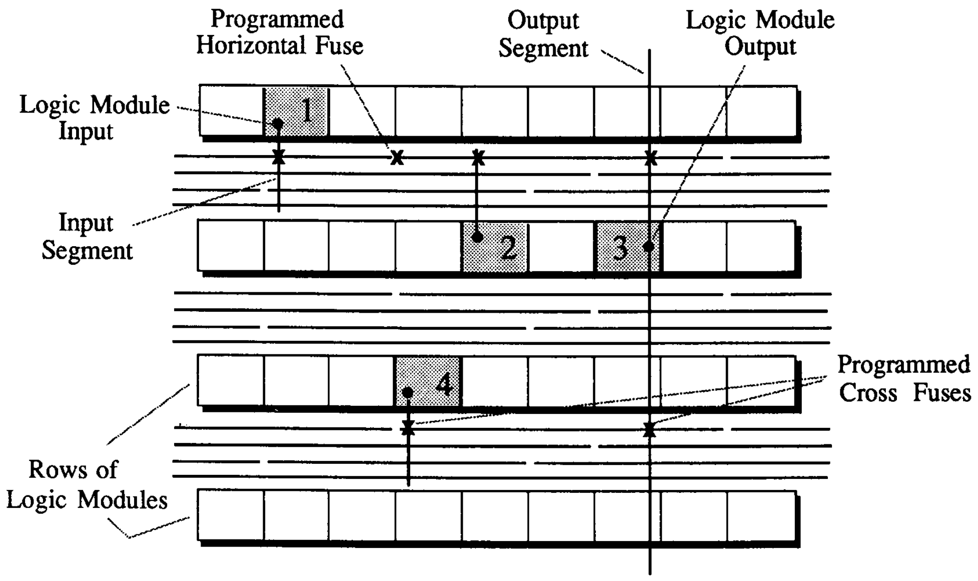

Fig. 5: A Routing Path

5. Implementation: Silicon and Software

The architecture has been implemented in a CMOS device described in [4].

Computer-aided design tools have been developed to support the architecture. Designs are entered as schematics or netlists using a cell library.

The placement and routing algorithms are specific to the architecture. As usual these are time consuming, taking up to a few hours on a low-cost workstation. They achieve 100% routing completion. (Even expert users have never been able to improve manually on the automatic router). The probability of successful routing can be predicted by analyzing some statistics of the design.

Because the nets are R-C trees, delays are not a simple function of capacitive load as with mask-programmed gate arrays. Nevertheless, we are able to quickly calculate precise delays at each input for post-layout simulation and timing verification.

Acknowledgement

The authors gratefully acknowledge the technical contributions of John Chang, David Gluss, Richard Guo, Dana How, and Faysal Sohail.

References

[1] S. Wong, H. So, C. Hung, J. Ou, "CMOS Erasable Programmable Logic with Zero Standby Power," *Int'l Solid State Circuits Conf. Digest of Technical Papers*, Feb. 1986, pp. 242-243.

[2] H. Hsieh, K. Duog, J. Ja, R. Kanazawa, L. Ngo, L. Tinkey, W. Carter, R. Freeman, "A Second Generation User Programmable Gate Array," *Proc. of the Custom Integrated Circuits Conf.*, May 1987, pp. 515-521.

[3] A. El Gamal, K. El-Ayat, A. Mohsen. "Programmable Interconnect Architecture", pending U.S. patent.

[4] K. El-Ayat, A. El Gamal, R. Guo, J. Chang, E. Hamdy, J. McCollum, A. Mohsen, "A CMOS Electrically Configurable Gate Array," *Int'l Solid State Circuits Conf. Digest of Technical Papers*, Feb. 1988.