

# CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

## LEARNING THE ROPES

Ingo Cyliax

### The FPGA Tour

With FPGAs becoming more commonplace, we decided it was time to introduce a bimonthly column to cover the ins and outs of working with FPGAs and CPLDs. Who better to provide this information than *Circuit Cellar* columnist Ingo Cyliax, who discovered the benefits of FPGAs long before they were considered mainstream?



If you've followed my articles in *Circuit Cellar*, you might have noticed my affinity for using field-programmable gate arrays (FPGAs). I was using FPGAs well before they were considered mainstream. More and more people are designing with FPGAs these days, mostly because of the lower costs of the design software required to use them.

We are also seeing more *Circuit Cellar* articles that use, and are based on, FPGAs. We decided it was time to introduce a new bimonthly column to cover FPGAs and the closely related CPLD. This month, I kick off the column with the first half of a two-part tutorial on what FPGAs and complex program-

mable logic devices (CPLDs) are.

FPGAs were primarily used to prototype designs for application-specific ICs (ASIC). ASICs require large lead times from design to actual implementation and are produced in large volumes. If a design error was made in a complex design, you would have to spend a lot of money and time to re-spin an ASIC.

FPGAs enable designers to implement designs in hardware for testing and prototyping with much less lead time. The turn-around time for FPGAs is less than a day, and even shorter for small designs. However, because the designs were implemented in CAD tools that are also used to design ASICs, it was generally expensive to get started with FPGAs. Nowadays, the costs for entry-level design tools is low enough for almost anyone to get started using these parts.

In this article, I go over some of the architectural features of FPGAs and CPLDs. Next month, I'll discuss the design flow and techniques. In the months after that, I'll cover many exciting FPGA and CPLD applications.

#### LET'S GO

FPGAs and CPLDs are in the family of programmable logic devices. The

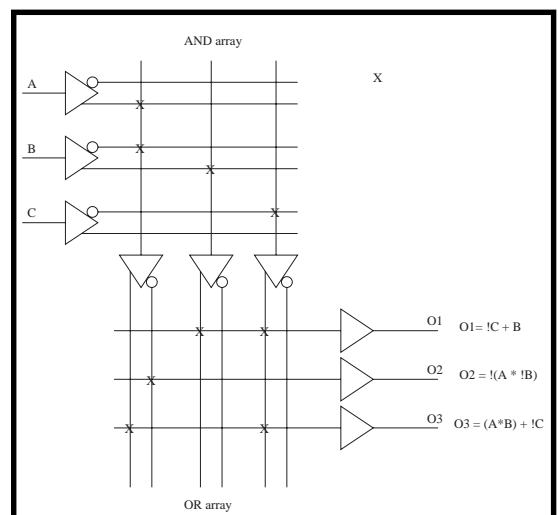
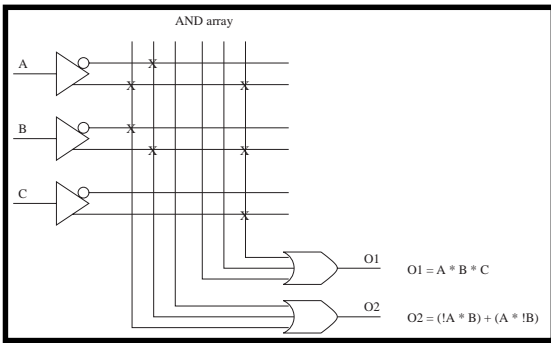


Figure 1—In the basic programmable logic array (PLA), you program each array, one for AND and one for OR, by blowing fuseable links to implement logic functions.



**Figure 2**—In contrast to a PLA, the programmable array logic (PAL) only has a programmable AND array. This makes the chip less complex (and less expensive), but limits the kinds of expressions that can be implemented efficiently.

first examples of these devices were programmable logic arrays (PLAs). As you can see in Figure 1, PLAs had two arrays of connections that enabled the designer to program which pins were ANDed and which of the AND terms were ORed together to implement an output function. This procedure was done by burning out little fuse wires on the chip so that only the desired connections remained.

After the PLA, the programmable array logic (PAL) shown in Figure 2 was introduced. Here, only the AND terms were programmable, which made the chip less complicated and enabled you to implement wide AND-OR product terms. These chips were popular for implementing decoders.

One refinement to the PAL was the addition of a programmable register at the output of the AND-OR terms. This made it possible to implement state machines and so the chips are called

programmable logic devices (PLD). PLDs have been refined so the output cell can be configured to implement a pass-through function, registered outputs, inverted output, and an internal feedback. One of the most popular PLDs is generically known as the 22V10. The basic design of the 22V10 can be seen in Figure 3.

This device is a 22-pin IC that has 10 general-purpose (V) output cells. Any of the 22 pins can be an input to the AND-OR terms and 10 of the pins are wired to the output cells. It's so popular that it's still manufactured by different vendors in several versions (low-power, high-speed, low-voltage, in-circuit programmable, etc.).

## EVOLVING TECHNOLOGY

A CPLD, as the name implies, supercedes the PLD. Because PLDs like the 22V10 are fairly small devices, designers typically had to wire several of them on a PCB to implement large designs. When you increase the component count, you reduce reliability and increase cost. Also, for high-speed designs, the chip interconnects slow down the signals. A CPLD combines several PLD structures into a single chip and adds a global programmable interconnect to connect the inputs and outputs of the PLD cells to each other. Figure 4 shows an example of a CPLD.

CPLDs are a great innovation because they make it possible to add a single chip that is totally programmable. The more PLD cells and interconnects you can get in a chip, the bigger the single-chip design can be.

On the other hand, FPGAs have a different background. Although CPLDs have their roots in PLDs, which were primarily used to reduce the component counts on PCBs, FPGAs are a programmable version of gate arrays.

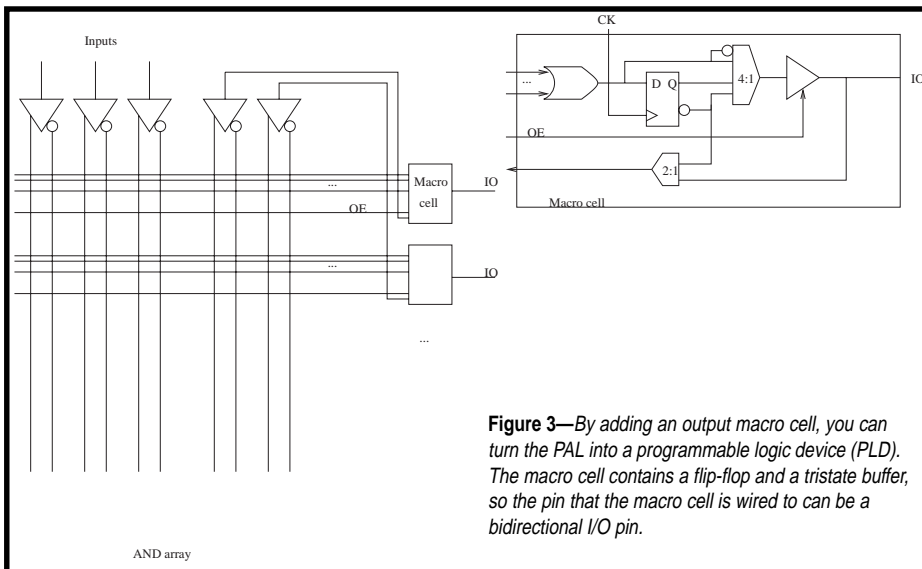
A gate array is an ASIC with a simple gate replicated in a large array. The gate is typically a two-input NAND gate. To implement a design, these silicon gates were connected with metal traces. So, in a sense, the function is implemented solely by routing. We can implement all of the basic functions (OR, AND, INVERT, PASS) with a NAND gate depending on the voltage encoding of the inputs and outputs.

With gate arrays, the actual design can be implemented fairly late in the manufacturing process (when the metal layers get deposited) so the lead time is short compared to a truly custom ASIC. These types of gate arrays are called mask-programmed gate arrays.

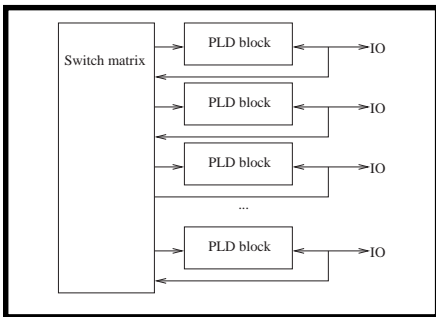
An adaptation of the mask-programmed gate arrays is the programmable gate array. These devices had predefined metal layers that connected traces to all of the gates. To program the function, fuseable links would be burned off either with a large current or with a laser.

It turns out that the metal traces actually take up much of the chip area in a programmed gate array and much of the metal layer isn't used. One way to enhance the density of the chip, is to increase the logic function of the gate in a gate array. This was first done by using 4:1 MUXs as the basic logic element. By programming the input levels of a 4:1 MUX, it can be used to generate any function of two variables, independent of the voltage levels. We can see that this is much denser than a two-input NAND gate. A single 4:1 MUX can implement a half-adder.

Instead of programming gate arrays by burning, fusing, or cutting metal traces, you can use a small programmable routing matrix to



**Figure 3**—By adding an output macro cell, you can turn the PAL into a programmable logic device (PLD). The macro cell contains a flip-flop and a tristate buffer, so the pin that the macro cell is wired to can be a bidirectional I/O pin.



**Figure 4**—Complex PLDs (CPLD) contain several PLD structures and a global interconnect matrix that can wire the inputs and I/O signals from each PLD block to each other and to external pins.

implement the routing of the chip. This matrix could, for example, connect the input and outputs of each logic block to the nearest neighbor or to a global interconnect. If this device can be programmed by the user, then you have a basic FPGA architecture. Figure 5 shows an example of a generic FPGA architecture.

Let's look at programming these devices. I mentioned that early chips were programmed burning out fusible links or similar features. Of course, these chips are not reprogrammable and are called one-time programmable (OTP) devices. There are applications for OTP CPLDs and FPGAs. For example, Actel makes a line of OTP FPGAs that are robust in the presence of radiation and thus are used in military and space applications where you don't want your chips to get reprogrammed. Also, fusible links propagate signals fast because they are essentially just wires on the chip.

Although some OTP applications are interesting, I'll primarily focus on reprogrammable architecture in this column. There are two types of reprogrammable FPGA/CPLD technologies—flash-memory/EPROM based and SRAM based. Flash-memory/EPROM-based CPLDs are easy to understand. A small pass gate is wired to a flash memory or EPROM cell and that enables us to program the terms, the macro cells, and the large interconnect.

Just like EPROMs, EPROM-based CPLDs have pretty much been surpassed by flash memory-based devices. Flash-memory

devices are reliable and don't require expensive windowed packaging to erase. Also, just like flash-memory devices, flash memory-based CPLDs are in-circuit programmable, usually via a JTAG or other serial interface.

On the other hand, reprogrammable FPGAs tend to be primarily SRAM based. Same idea with the small routing matrix, which is implemented using pass gates driven by the value of the SRAM memory cell assigned to it. However, instead of using pass gates to program the function of the basic logic block, most SRAM-based FPGAs use look-up table (LUT) function generators, which are small SRAM cells with four or five inputs. The inputs are the address lines of the SRAM cell and the output of the SRAM cell is the output of the function generator. Of course, the programmable flip-flop after the logic block is programmed via pass gates. To program the function of the logic block, you load up the contents of the SRAM cell and configure the logic block to be either registered or combinatorial.

SRAM-based FPGAs tend to be much denser than flash memory-based CPLDs, but they lose their configuration once the power is turned off. Because they lose their configuration, you need some sort of external memory to store the configuration information. Most FPGAs can read programming information from a serial or parallel EPROM or flash memory. This mode is called the master mode. The FPGA will provide all signals and addressing to read the data on its own. No components other than the serial/parallel PROM are needed.

SRAM-based FPGAs can also be programmed via an external source. In slave mode, the FPGA accepts a serial

or parallel data stream that represents the configuration data. The source of the data can be a processor, computer, or an FPGA that is acting as a master. Using this technique, it's possible for several FPGAs to be programmed from a single memory. A master FPGA is wired to a daisy chain of slave FPGAs. When the master FPGA has been programmed, it will keep reading the data from the memory and pass it on to the slave devices until all of the FPGAs are configured.

Configuring SRAM-based FPGAs is faster than programming a flash memory-based CPLD, but takes some time when the system is started. It's important to take the FPGA configuration time (at startup) into account when designing your system. If you need instant power-on performance, you probably want to use a flash memory-based device or an OTP device.

To recap, a CPLD is a device that has several PLD-like blocks connected via a large connection matrix, and an FPGA has a large number of logic blocks that are usually simple lookup tables followed by a configurable register connected with smaller routing matrices in an array. This is the basic structure idea, but of course, not everyone is happy, so let's look at some architectural enhancements and features that can be found in many modern FPGAs and CPLDs.

One of the features in a SRAM-based FPGA is the SRAM-based LUT. Because they usually have four or five inputs, they are essentially  $16 \times 1$  or  $32 \times 1$  memory blocks. Early FPGAs would only let you use these LUTs as ROM cells. If you wanted to implement registers or memory, you had to use the flip-flop in each logic block as a single register bit. By making the LUT writable, you can now use a LUT as a general-purpose memory or register block in our design. So, you get 16 or 32 registers for each logic block.

Occasionally it would be nice to have larger memory blocks. Maybe you want a FIFO that can buffer up data. Newer FPGAs now include dedicated large memory blocks that can be used in this way. This is only one trend to

Voltage level			Function implemented with NAND	
A	B	O		
L=0	L=0	L=0	$!(A * B)$	-> A NAND B
L=0	L=0	L=1	$A * B$	-> A AND B
L=0	L=1	L=0	$!(A * !B)$	-> invert A, when B = L
L=0	L=1	L=1	$(A * !B)$	-> pass A, when B = L
L=1	L=0	L=0	$!(!A * B)$	-> invert B, when A = L
L=1	L=0	L=1	$(!A * B)$	-> pass B, when A = L
L=1	L=1	L=0	$A + B$	-> A OR B
L=1	L=1	L=1	$!(A + B)$	-> A NOR B

**Table 1**—It's a brain teaser, but the basic NAND gate can be used to implement all basic logic functions depending on the input and output voltage conventions.

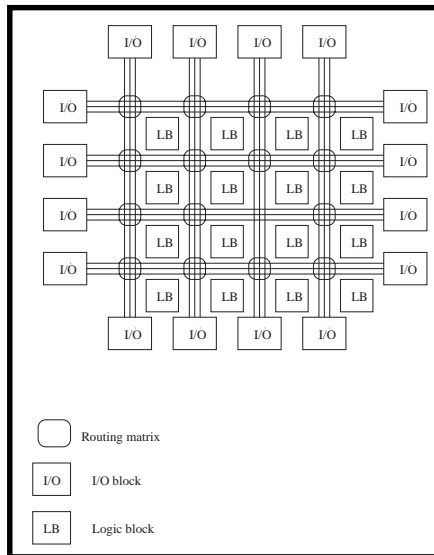
combine more complex functions with a general-purpose FPGA. There are FPGA architectures from Lucent that include a whole PCI bus interface as dedicated logic on the chip. Also, Triscend has an interesting architecture that adds a processor core with an FPGA. Check out the links in the sources sections to get more information on some of the chips available.

Besides registers and memory blocks, math is important. The most common operation is the add. A full adder can be implemented in a four-input LUT (or two blocks if you need a carry out). This setup is ideal for implementing ripple-carry adders. However, ripple-carry adders are slow when the word size increases and you generally want to use carry-lookahead adders, which take more logic to implement.

Because adders are so prolific (think counters), current FPGAs and some CPLDs also include hardwired carry chains in the logic blocks. These carry chains are dedicated carry generators. If the adjacent bits of the adder or subtraction are connected to adjacent logic blocks, you can use the carry logic to implement a fast ripple-carry adder without using additional logic. The hardware carry chain is so fast it can be used to efficiently implement 16- or even 32-bit adders.

FPGAs and CPLDs are register rich. Each logic block has a flip-flop. FPGAs are good for implementing synchronous circuits and have efficient dedicated global clock networks that distribute clock signals to all of the flip-flops on the chip. Most FPGAs have multiple global clock networks, making it suited for implementing multiclock domain circuits. These global clock networks can also be used for logic signals that need to go every place on the chip. Clock-enable signals or strobe signals can use these networks.

Some FPGAs also have tristate drivers at each logic block that can drive bus-like networks running along the rows or column of the device. These drivers can be used just like tristate buffers and buses in board-level designs. One common trick is to use buses and tristate buffers to implement wide MUXs. Implementing MUXs using tristate buffers is essentially free in



**Figure 5**—The basic field programmable gate array (FPGA) contains configurable logic blocks, small routing matrices, and I/O blocks that can configure each I/O pin for different functions.

FPGAs that have this feature because there is a tristate buffer in every logic cell output.

I mentioned earlier that gate arrays can be implemented with simple NAND gates, but the density is not as high. However, it's much easier to synthesize high-level descriptions of circuits into simple gates, than trying to take advantage of high-level functional blocks. This debate is similar to the argument about a C compiler being able to optimize more easily to a RISC processor than to a CISC processor. However, just as C compilers have gotten more sophisticated and can target CISC processors better, high-level logic synthesizers have gotten much better at targeting complex logic functions. For example, the VHDL compiler that comes with Xilinx's Foundation toolset can figure out when to use carry chains to implement an adder automatically.

Because the tools are getting better and complex logic function can be implemented more densely, the trend seems to be to implement more complex logic block functions.

Initially, FPGA had only local and global routing resources (i.e., a logic block could only connect to adjacent logic blocks or to global networks). Newer FPGAs have multilevel routing hierarchies, so logic blocks can connect to different levels in the routing hierarchy. These FPGAs are complex, but

luckily the design software takes care of these issues for you.

Incidentally, routing performance is one area in which CPLDs are more predictable because they have fewer routing matrices than FPGAs. Each routing matrix adds a little delay to the signal, so the fewer routing matrices a signal has to traverse, the faster it gets there. FPGAs have many matrices and the software has to route the signals around the chip. So, depending on where the logic blocks end up on the chip, the signals can be delayed significantly.

I mentioned that the logic complexity is going up in FPGAs. Also, FPGAs tend to have higher logic densities per chip than CPLDs. But all of this is changing. CPLDs are becoming more dense, with more PLD blocks and more routing matrices so, in a sense, CPLDs are becoming more like FPGAs and FPGAs more like CPLDs. Also, with more system-on-a-chip functionality (e.g., dedicated CPUs, bus interfaces, and memory blocks), it will be interesting to see where all this is going.

Lucky for us, the new advances and features in the high-end devices have made last year's basic low-density FPGA and CPLD architectures more economical to use in embedded systems. Many production-volume products now ship with FPGAs and CPLDs in them, never bothering to implement the function in an ASIC.

## JUST THE BEGINNING

Now you know about LUTs, product terms, routing matrices, carry chains, and dedicated system functions. If you think this would be hard to use and design for, think again. FPGA and CPLD vendors have gone through great pains to make the software hide all the architectural details. In many cases you can implement a design without worrying about whether you are targeting a CPLD or FPGA and let the tools handle all of the details.

Of course, if you really want to twiddle the bits yourself, some vendors give you the tools. Next time, I'll take a look at some of the design software and how to design some simple examples.

*Ingo Cyliax has written for Circuit Cellar on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego-based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. You may reach him at [cyliax@derivation.com](mailto:cyliax@derivation.com).*

## SOURCES

### **FPGAs/CPLDs**

Xilinx, Inc.

(408) 559-7778

Fax: (408) 559-7114

[www.xilinx.com](http://www.xilinx.com)

Altera Corp.

(800) SOS-EPLD

(408) 544-7000

Fax: (408) 544-6403

[www.altera.com](http://www.altera.com)

### **OTP FPGAs**

Actel Corp.

(888) 99-ACTEL

(408) 739-1010

[www.actel.com](http://www.actel.com)

### **FPGA+CPU**

Triscend Corporation

(650) 968-8668

Fax: (650) 934-9393

[www.triscend.com](http://www.triscend.com)

### **FPGA+PCI**

Lucent Technologies

[www.lucent.com](http://www.lucent.com)