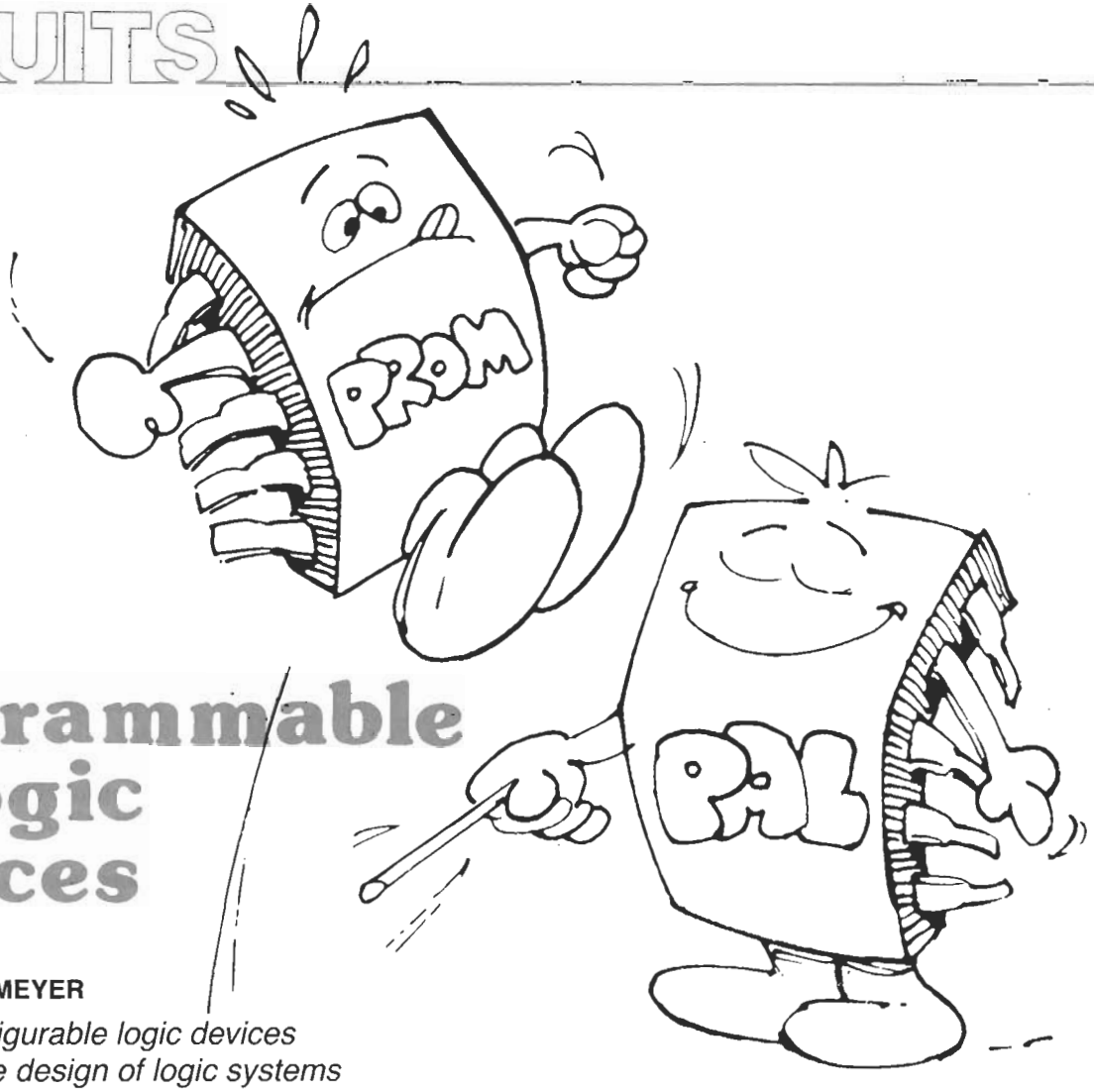


Programmable Logic Devices

ERNEST MEYER

The new reconfigurable logic devices revolutionize the design of logic systems

IF YOUR PROJECT DESIGNS USE 7400-SERIES discrete components for random-logic or state-machine control, you'll probably find that you can simplify both the design and the assembly by merely substituting *P*rogrammable *L*ogic *D*evices (PLD's) for the 7400-series hardware.

Programmable logic devices contain gates and flip-flops, just like regular 7400-series IC's. However, in the 7400 series, which is often considered to be standard parts, the gates and flip-flops are wired together in fixed configurations. On the other hand, PLD's aren't hard-wired; instead, they contain small fuses that are blown or left open to connect their internal gates and flip-flops in any needed configuration.

A PLD's fuses are similar to those used in PROM's (*P*rogrammable *R*ead-*O*nly *M*emories), EPROM's (*E*rasable *PROM*'s), and EEPROMs (*E*lectrically *E*rasable *PROM*'s). Usually, the fuse is blown by addressing the location and applying a high-voltage pulse (12–30 volts, depending on the device) across the fuse while the IC is in its *programming mode*. Most standard PROM programmers can be used to configure PLD's.

Erasable PLD's (like EPROM's and EEPROM's) can be wiped clean and reprogrammed—although the wiping is actually a way of bypassing the blown fuses. In that way, hobbyists can reuse a single PLD for a variety of projects or applications.

Circuit designers have traditionally used PLD's for logic functions that are not generally available in standard off-the-shelf components. If not for PLD's, a large number of conventional IC's would otherwise be required to perform a relatively simple non-standard logic task.

A lot for a little

Because they provide the circuit designer with an enormous array of user-programmable fuses, recent PLD designs can substitute for dozens of standard parts. Typically, a PLD can replace about a half-dozen standard parts, although the exact number of required PLD's depends on the actual circuit. Without PLD's, the increased number of parts increases the size of the circuit board, which in turn lowers the system reliability while increasing the system's cost.

Because a single PLD replaces a variety of standard parts, manufacturers also prefer them to standard parts because inventory is minimized—the number of different IC's which must be stocked is sharply reduced. Also, the integration of discrete parts into a single device puts all the wiring inside an IC rather than on the circuit board, which means that the board's design is simpler and the logic can run faster.

Programmable logic is therefore an excellent choice in systems where board size, board complexity, system reliability, inventory considerations, or speed is important. Even if no single factor is crucial, their combined weight can easily swing the balance

toward a clean and simple PLD design rather than a large composite of standard parts.

Even so, as we will show, some standard parts are so complex they don't fit very well into a programmable logic architecture. To resolve the problem of complexity, there is now under development hybrid PLD's that contain many large standard-logic functions combined with a programmable logic array.

Since the proliferation of different logic architectures will inevitably bring PLD's into common use by both the hobbyist and the professional, we will look at the extraordinary development of complex reconfigurable PLD's, and examine their diverse capabilities.

Programmable logic technology

To understand how fuse technology works, we must look at the techniques used for fabricating integrated circuits. *Silicon processing*, as it is called, is very complicated; but a quick-and-dirty description will make it a little easier to understand.

To make a silicon-based chip, wafers of silicon crystal are first processed to make transistors. The silicon crystal itself does not really conduct electricity at all. The transistors are created by doping selected regions of the silicon with phosphorus or arsenic, and metal lines deposited on the wafer connect the transistor sites. The transistors and metal lines are called features.

Simply speaking, each feature is formed on the crystal at the selected locations by spraying a light-sensitive protective chemical, called *photoresist*, in a thin, even coating on the wafer's surface. The wafer is then bombarded with light at selected locations through a precise slide projector. The slide projector (called an *optical aligner*) uses a very small slide, called a *mask*, to screen out the light where it isn't wanted. The chemical decays where the light strikes the photoresist-covered wafer, and is simply washed off, leaving bare silicon at selected locations.

The precision of the optical aligner determines how fine a feature can be made. In the early 1970's, it was difficult to make transistors smaller than 10 microns, Now, transistors can be less than a micron in size, meaning the same-size chip can hold more than 100 times as many transistors. Moreover, as features get smaller, the transistors are packed more densely; hence, the devices can work faster.

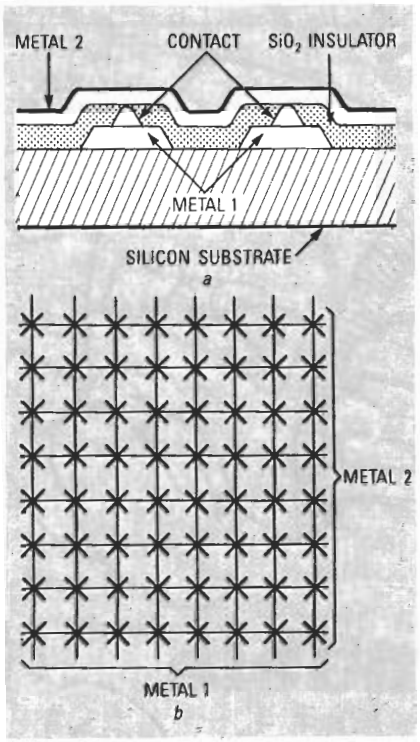


FIG. 1—THE METAL LAYERS on two-level chips are insulated by a layer of SiO_2 (*a*). The first programmable IC (*b*) provided 64 cross-points (fuses).

The bared silicon can be treated in various ways. For example, the silicon can also be treated to recast the crystalline structure into an amorphous form called *polysilicon*, which conducts electricity and therefore can be used to form wires. Single-level metal chips use polysilicon wires to cross under the metal wires at intersection points. However, polysilicon conducts electricity over a hundred times worse than metal, which substantially degrades performance.

The wafer is heated after the application of each wiring level, which causes the silicon on the surface to oxidize. Since the resultant SiO_2 is more bulky than crystalline silicon, the oxide *blooms* over any raised features (such as previously deposited metal lines) during oxidization. Since SiO_2 does not conduct electricity, it forms an insulating sheet a few microns thick over the entire wafer surface. The oxide can again be selectively etched: Photoresist is again applied to the wafer, selectively etched away again, and the wafer is exposed to chemicals that attack the SiO_2 (but not the pure Silicon) at the chosen feature locations. Further layers can be grown on top of the resultant *planarized* wafer to form a sandwich-like structure, with SiO_2 as the "bread," and metal or doped silicon as the "meat."

Several levels

The transistors on the chip are wired together with deposited strips or spots. The various layers are connected by holes (called *contacts*) through the SiO_2 layers. Although up to three levels of metal can be used, the metallization process is expensive, and so IC's are mostly made with two, or even only one, metal level. A cross section of a chip with two-level metal construction is shown in Fig. 1-*a*.

To make the first fuse-programmable chip, as shown in Fig. 1-*b*, eight 12-micron wide horizontal lines were deposited in first-level metal over the bare silicon substrate. The wafer was then heated up to cover the metal with the insulating SiO_2. After applying photoresist, sixty four very small holes (five microns in diameter) were etched into the insulator in eight rows over the first-level metal. The contacts could be much smaller than the first-level metal because they just needed to touch the edge of the metal line to provide a conducting path. (Small contacts were necessary for programmable technology, but such contacts greatly reduced the reliability of the device.)

Next, photoresist was applied again to the wafer, and aluminum was deposited in the holes (or *contacts*). Another eight metal lines were deposited in second-level metal, but those were stacked in the vertical direction, crossing over first-level metal at each contact site. The final product was therefore an 8 × 8 grid having a very small aluminum contact between each metal-1/metal-2 crosspoint.

Although the contacts—called *fuses*—are made as wide as the wires in standard silicon chips, they are narrower than the wires in programmable logic chips. The interconnecting aluminum is burned away (blown) by applying a very large current across the fuse connection. At lower voltage levels, the fuses that have not been burnt away conduct normally. That is why the metal lines were made so wide and the contacts so small—they had to be made larger than the contacts so that the contacts and not the metal lines would burn away.

Early PLD's

The earliest programmable chip is the one shown in Fig. 1-*b*; it contained nothing more than a fuse matrix and a wire grid. The crosses represent fuses that can be blown or left open.

The PLD pioneers were quick to stack transistors around the metal grid to make true *programmable devices*. Also, nichrome (nickel-chromium) fuses were used instead of contacts. In that kind of programmable technology, regular contacts are run from the lower metal to the upper metal layer; those contacts miss the actual wire intersection point by 3 microns. A 3-micron strip of nichrome compound is then deposited laterally to connect the contact site to the second-level metal strip. Because of its lower melting point, nichrome fuses out more cleanly than aluminum; in fact, the nichrome actually vaporizes during fuse programming. Therefore, a lower programming voltage is needed, which means that less power needs to be put into the chip. That simplifies programming while making the devices more reliable. Also, the metal lines are not much wider than the fuses themselves, which allows a greater density.

The PROM

The earliest fuse-programmable device was the PROM. As shown in Fig. 2, PROM input logic (the AND plane) is fixed, while the output logic (the OR plane) is programmable. The upper non-fused fixed-wire matrix feeds a set of AND gates. In the fixed AND array, the output of any one column is high only if all the connected inputs to the column are high. In PROM's, the fixed AND array provides one and only one high output for each possible condition that can exist on the inputs. The non-fused matrix is called a *fixed AND plane*.

The lower matrix is fuse-programmable. Simply speaking, each of the wiring rows feeds into a single enormous OR gate. That description is a simplification, of course, but it is adequate for our purpose.

TABLE 1

IN_1	IN_0	OUT_2	OUT_1	OUT_0
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

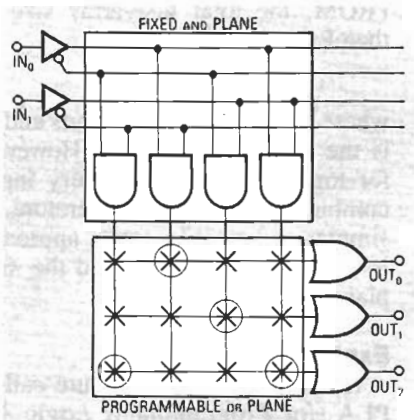


FIG. 2—IN THE EARLIEST FUSE-PROGRAMMABLE device, the AND plane was fixed; only the OR plane was programmed.

The OR gate is set so that the output is high if any connected input to the fuse row is high, a configuration that is called a *programmable OR plane*. If the only fuses left connected in Fig. 2 are those indicated by a circled-×, the ones and zeros in Table 1 constitute a truth table for a PROM programmed (blown) with the fuse pattern shown in Fig. 2

Conventionally, PROM's are regarded as programmable memories; however, the devices can obviously be used for logic as well. For example, logic equations can be written for the above device as follows:

$OUT_0 = IN_0$ and NOT-IN_1
OUT_1 = NOT-IN_0 and IN_1
OUT_2 = (NOT-IN_0 and NOT-IN_1) OR (IN_0 and IN_1)

Obviously, OUT_2 is a genuine XOR (exclusive-OR) of IN_0 and IN_1. That may seem like a very clumsy way of making a simple gate, but by expanding the size of the array very complex logic terms can be created. In particular, that kind of architecture is very good for bus-decoding functions. For example, if an 8-bit-wide bus must be decoded in six different ways to provide six enable signals in a

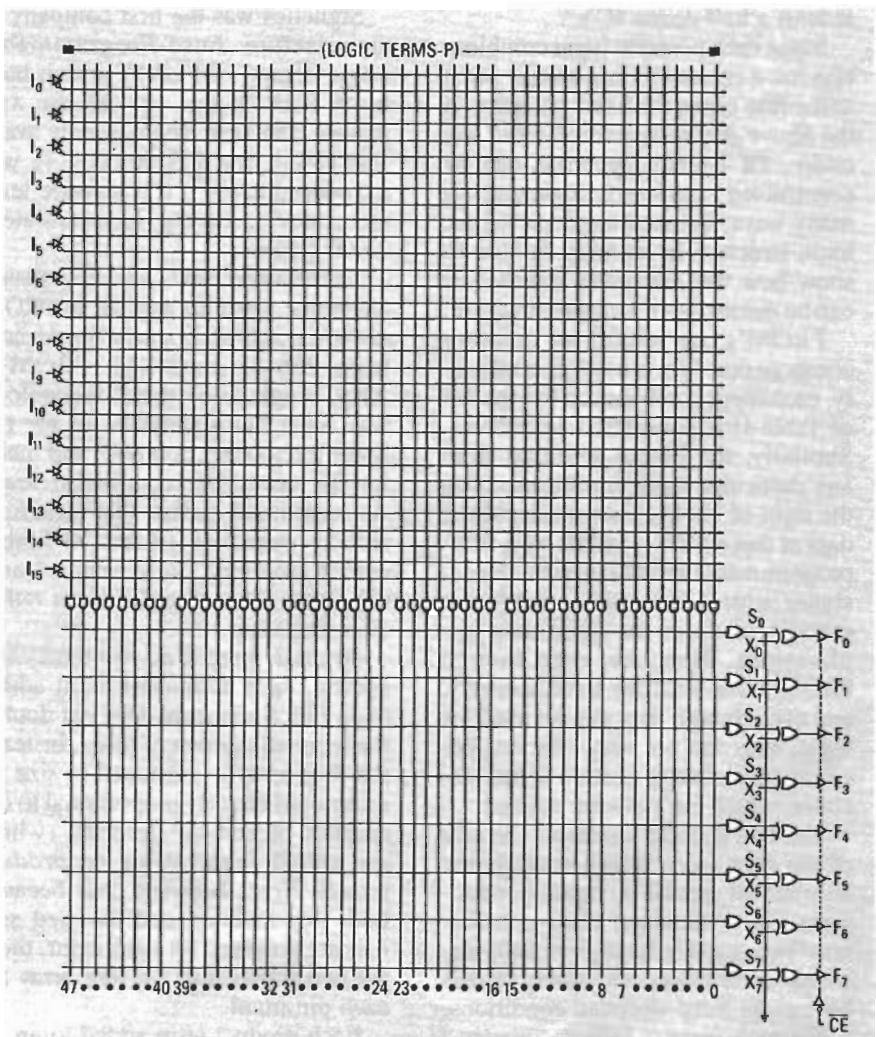


FIG. 3—THE FIRST COMMERCIALLY AVAILABLE FPLA was a single-level device having approximately 2,000 fuses.

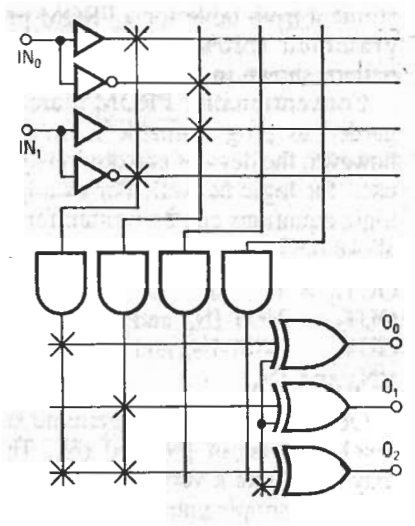


FIG. 4—IT'S EASY TO IMPLEMENT COMPLEX logic functions in an FPLA. As shown, the functions previously described for a PROM can also be mapped directly into an FPLA's architecture.

complex digital system (like a computer), one 16-pin device can replace at least a half-dozen IC's.

Since each possible input combination has a corresponding unique input to the fuse array, the fixed AND array in the above device is termed a *full decoder*. Of course, decoders can be constructed from simple logic gates in many ways, but we have depicted the logic structure in an array fashion to show how the AND gates themselves can be made part of an array structure.

PROM's are useful as memory components. When used as a memory, each input combination (on the left of Table 1) is considered an address. Similarly, the output resulting from any particular input combination (on the right of Table 1) is considered the data at that address. In that way, fuse-programmable ROM's provide the designer with a fast, static, long-term storage medium for fixed-data applications. However, even though PROM's are ideal for fixed memory, and even though they can be used for logic, they are not very efficient because they always contain a full decoder, which isn't always needed.

The full decoder increases the size of the fuse array, which restricts the number of possible input connections. To understand that, consider how two signals have four fully-decoded conditions, and three signals have eight fully- decoded conditions. Since every input is fully decoded in a PROM, each additional input doubles the number of fuses required. In a PROM, the total fuse-array size is therefore:

$$2^I \times O$$

where I is the number of inputs and O is the number of outputs. However, for logic functions, not every input combination is crucial; therefore, a simpler and more compact approach is to make both the OR and the AND planes programmable.

Exploring PLA's

IBM, using an architecture called PLA (for *P*rogrammable *L*ogic *A*rray), was the first company to use a device having both programmable OR and programmable AND planes. In the earliest PLA's, the array was metal-programmed rather than fuse-programmed. In other words, the device was customized during the actual chip fabrication rather than by the chip user, and was therefore used only by manufacturers needing a large quantity of chips.

Signetics was the first company to manufacture *F*use-*P*rogrammable *L*ogic *A*rrays" (FPLA's), which have fuses both in the OR and the AND planes. The first commercially available FPLA, the 82S100 (Fig. 3), was introduced in 1977; it is a single-level metal device having approximately 2,000 fuses.

At the same time, a similar fusing technique was also applied to PROM design, resulting in a rapid movement from 256-bit to 2K-bit PROM's. Also, single-level metal technology was used, with polysilicon for the lower interconnection level and metal for the upper interconnection level. As mentioned earlier, that technique reduces chip cost. In fact, to further reduce chip cost, some vendors now use fuses made of polysilicon rather than nichrome.

Because there is no full binary decoding, each additional input added to an FPLA structure does not double the required number of fuses. Instead, the fuse array is increased in size by an amount directly proportional to the number of vertical channels (which are called *logic terms* or *product terms*). Note, however, that because both non-inverted and inverted signals are provided for each input, there are two inputs into the fuse array for each pin input.

Each product term added to an array allows one additional set of AND products to affect an output condition. Therefore, in the 82S100, there are 48 possible unique AND products that can affect the output. Since the AND-products can be combined together in the OR plane, more than 48 unique input combinations can affect the output. Therefore, three numbers are used when measuring the size of an FPLA: the number of inputs, the number of product terms, and the number of outputs. Those numbers are combined to form the *IPO number*. To calculate the total number of fuses, the number of inputs is doubled (to provide noninverted and inverted inputs to the array), added to the number of outputs, and multiplied by the number of product terms, or:

$$((2 \times I)+O) \times P$$

For the 82S100, the IPO is 16:48:8. The total number of fuses is therefore $((16 \times 2)+8) \times 48$, or 1,920. By contrast, a PROM having 16 inputs and 8 outputs, like the 82S100, would require $2^{16} \times 8$, or 52,488 fuses—more than 25 more.

The smaller size of the FPLA architecture yields four advantages to the logic designer: 1) The array size is smaller, which makes the devices less expensive to manufacture and therefore less expensive to sell; 2) The smaller number of fuses makes programming faster; 3) The reduced array size makes the devices quicker to test; 4) The smaller size of the array makes it possible to make larger programmable devices, which increases their usefulness and power.

Additional features

Architectural enhancements increase the flexibility of the device even farther. The Signetics 82S100 FPLA, for example, adds two additional programmable features to the basic AND/OR plane.

First, the FPLA can be permanently set to give inverted or noninverted outputs. To achieve the complementary output, the outputs of the programmable OR plane feed into one input of an additional XOR gate. The other input to the XOR gate can be connected to ground. If the grounding fuse is left intact, the XOR gate inverts the output from the OR plane. If the fuse is blown, the input floats high, and the logic signal passes through the gate unaffected. Table 2 shows a truth table for the complementary-output feature.

TABLE 2

Output from array inverting fuse		Output from IC
0	0	0
1	0	1
0	X	1
1	X	0

The second feature, three-stated output, is not programmable but greatly improves the IC's usability. If many fuses are left intact in the programmable array, the increased loading slows the device's switching speed. Conversely, if there is a product term with only one or two intact fuses, the output switches much more quickly. As array size increases, the difference in switching speed between the least- and most-loaded gates can produce glitches in the IC's output. Erroneous logic states can be avoided by disabling the outputs until all the signals have settled.

Using an FPLA

It is easy to implement complex logic functions in an FPLA. For example, the functions previously described for a PROM can also be mapped into an FPLA's architecture. The connections required to do that are shown in Fig. 4.

Four fuse columns were necessary for a PROM, but note from Fig. 4 that the FPLA architecture and complementary output permits only two product terms to be used for the three functions, freeing the other inputs, outputs, and product terms for other logic operations.

Logic minimization

In Fig. 4, it is relatively easy to work out the connections to use. However, as logic complexity increases, it becomes more difficult to work out which fuse combinations use the product terms most efficiently. When large numbers of logic signals interact, the product-term columns can be tied to different combinations of inputs, and OR-ed together in respectively different ways to generate the same functions. Finding the optimal fuse map through *logic minimization* (the act of reducing logic equations to their most basic form) has two advantages: 1) The danger of running out of product terms is minimized; 2) The number of unblown fuses is minimized, which increases the speed and reliability of the device.

About the time when new software tools were developed to optimize fuse maps, a new development brought about a revolution in programmable-logic philosophy.

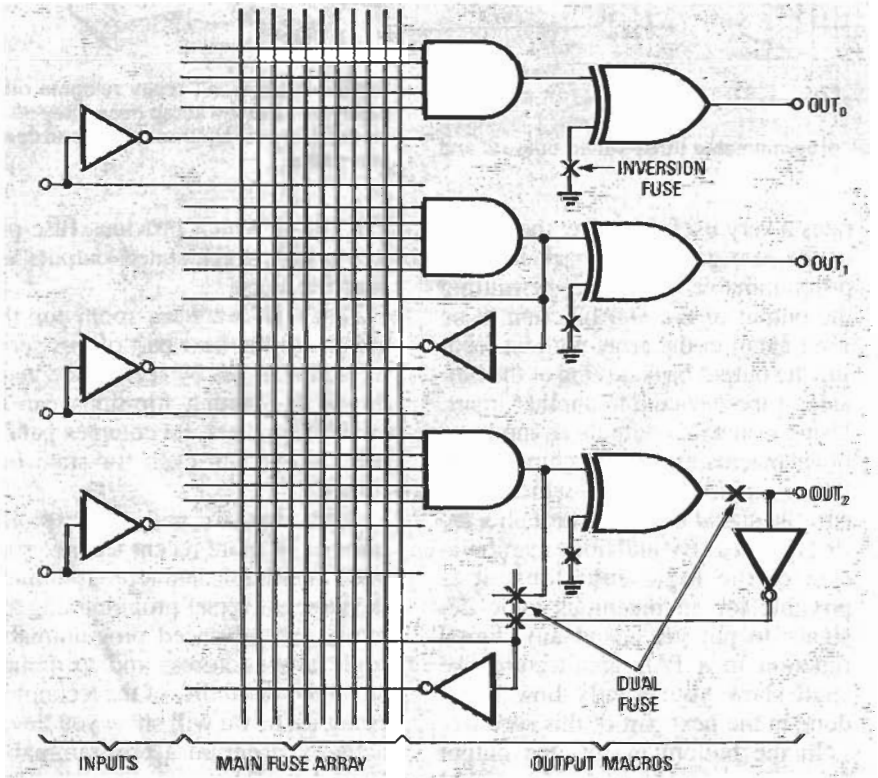


FIG. 5—PLA'S DON'T HAVE A PROGRAMMABLE OR plane at all; they have a programmable AND plane instead, making them the opposite of PROM's.

The PAL

The revolution in programmable logic was the 82S100 PAL (*P*rogrammable *A*rray *L*ogic). PAL's were marketed by Monolithic Memories as a simple alternative to standard parts—the emphasis being on *simple*. To that end, they supplied an easy-to-use programming language called PALASM, a FORTRAN IV program that translates logic equations into a fuse map suitable for use with a standard PROM programmer. At about the time that PAL's were released into the marketplace, 512 × 4-bit (2048-bit) PROM programmers were also interfaced with IBM computers, making it possible for designers to design, document, and program a simple PAL within minutes.

PALASM accepts six distinct data entries as input. First, the PAL part number is entered so that the program knows what sort of fuse map to make. Second, the pattern number is entered, so that the generated file containing a fuse map can be named. Third, the name of the device and the author's name is entered for archiving (so when someone else looks at the file, they know who made it and what it's for). Fourth, a pin list is entered, which contains the symbolic names for the pins that are used in the equations. The symbolic names can be numbers, or signal names like INIT, RESET, NMI, etc. Fifth, the actual equations are entered. A field is left open for the designer to enter notes about the design.

PALASM removes the designer from the world of fuse maps and architectures. However, to use PAL's most efficiently, it is important to choose an architecture that maps on to the type of logic you want to have. Also, PALASM allows designers to generate fuse maps (now often called *JEDEC maps*) directly. But sometimes it is actually easier to type in a fuse map than the equations. Indeed, depending on the needed functions, sometimes it is easier to draw a truth table or a state diagram.

From the architectural point of view, PAL's are easier to use than PLA's because they don't have a programmable OR plane at all: in its place they have a programmable AND plane instead, making them the opposite of PROM's. However, as we'll see very shortly, additional architectural features make PAL's just as flexible to use as FPLD's. As shown in Fig. 5, a

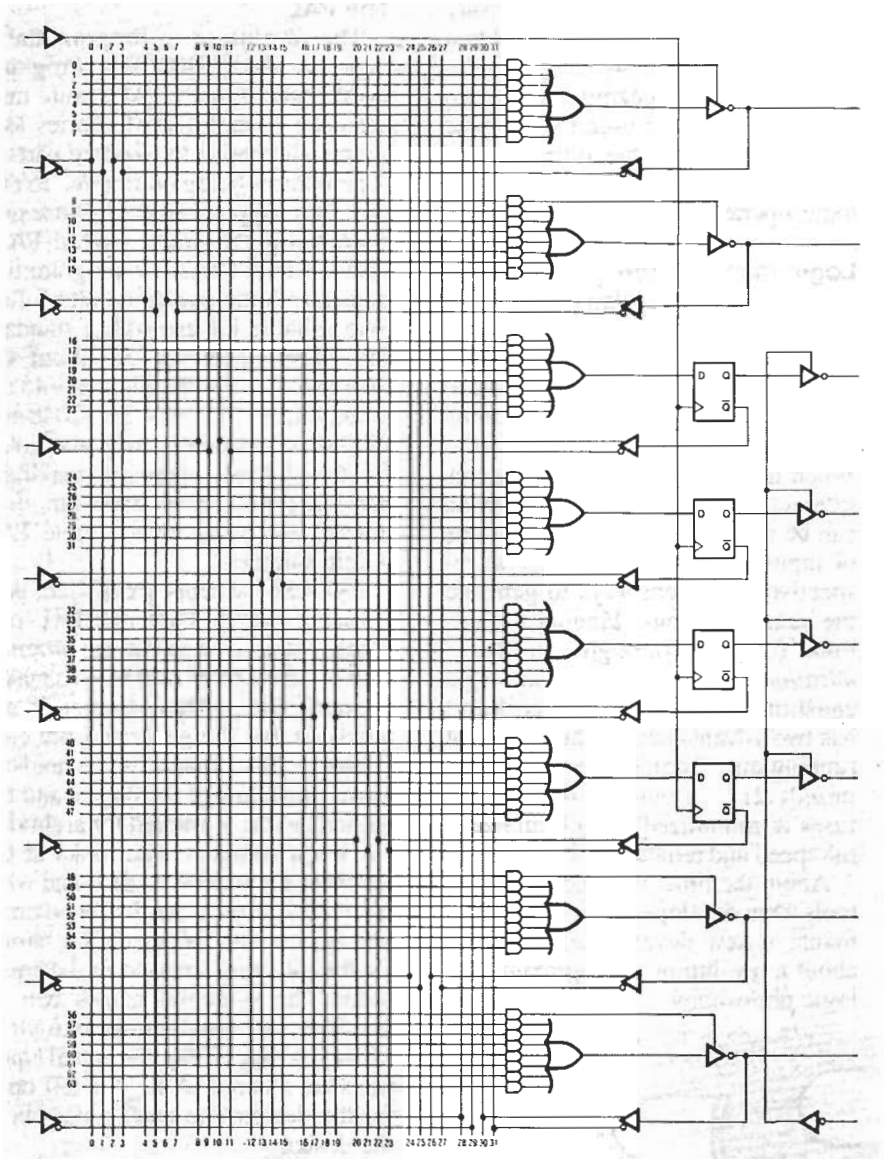


FIG. 6—THE COMMON PAL16R4 provides fuse-programmable three-stated outputs and four flip-flops.

PAL's inputs feed directly into the AND matrix.

Figure 5 also shows a new range of functions, called *macros*, at the output of the AND plane. In 1978, programmable logic components came to differ not only in their IPO ratios, but also in their macros, so that picking the best device for a particular application often hinged on choosing the one with the macros that incorporated the needed logic functions for that application.

In the top macro, the AND plane feeds a single AND gate with optional fuse-programmed inversion. That is a very simple macro; and because the inputs to the array are all inverted, we find that the output inversion is not needed at all.

The second macro down incorporates a very useful feature: the output of the AND gate is fed back into the programmable AND plane, permitting the output of the AND function to be used again in the array without feeding the output back around to the outside of the device into another input. Using a macro's outputs as inputs to other macros allows the chip to contain *multi-level* logic—which is to say, the signal can pass through a series of gates. By judicious manipulation of the logic equations, it is possible for an ingenious logic designer to put just about any digital function in a PAL architecture; we shall show you exactly how that's done in the next part of this series.

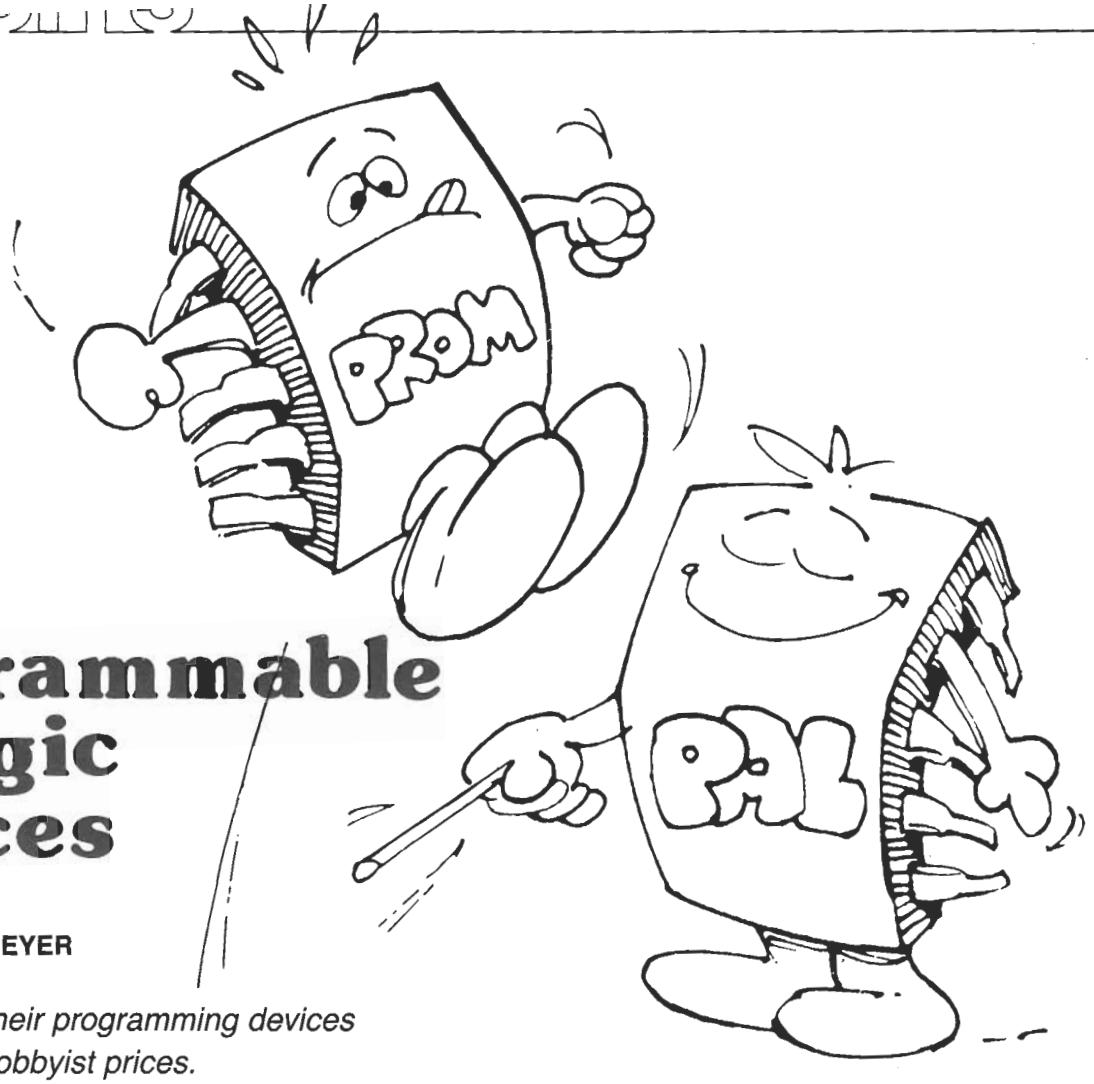
In the bottom macro, the output driver is replaced by a bidirectional driver consisting of dual fuses. Depending on how the fuses are blown, the macro can be configured as an output or as an input. If the output of the AND gate is broken, but the feedback path and the input driver are left intact, a signal can pass through the input driver and feedback path—however, the AND macro function then cannot be used at that location. On the other hand, if the output of the AND gate is broken and the input driver *is* taken out of the circuit, the AND macro functions normally, and the AND gate function is available for multi-level logic.

It should be obvious that there are a large number of possible macro constructions for combinational logic. However, that is just the beginning. Incorporating sequential logic elements in a PAL opens whole new dimensions of design. In Fig. 6 we see the architecture of the common PAL16R4, which provides fuse-programmable three-stated outputs and four flip-flops.

WHILE PAL's won't really relegate other technolgies to the scrap heap, they do offer the designer another usable and useful alternative.

That's all we have room for this month. In the next part of this series we will start off by seeing how *macrocells* containing flip-flops can be used for counters, for complex pattern generators, and even for state machines.

After that, we will also explore a number of more recent technologies, such as erasable and reprogrammable devices, universal programming systems, and advanced programmable-logic architectures; and to demonstrate the feasibility of the technology being used, we will show you how to actually program a programmable-logic device with commercially-available software. **R-E**



Programmable Logic Devices

ERNEST MEYER

Both PLD's and their programming devices are available at hobbyist prices.

Part 2 LAST TIME OUT WE looked at the early history of programmable logic because it gave us an understanding of how easily programmable IC's can be used. And to make practice conform to theory, we even designed and blew an actual PLD (*P*rogrammable *L*ogic *D*evice) just to demonstrate how it might be done by a hobbyist.

As we continue our journey through programmable-logic history, we will go a little way beyond the technology actually available to most hobbyists. However, because the field is developing so rapidly it won't be long before even the most sophisticated devices find their way into common digital projects.

Of course, while being able to program a PLD will naturally make it easier for you to use more sophisticated devices as they appear in the marketplace. Even now the PLD's available to hobbyists can replace more than a half-dozen conventional parts. Indeed, PLD's are already so complex they warrant the use of software for device design and verification, which is a fancy way of saying "creating a fusemap."

As with everything else, there is both a hard and easy way to do things. In this article we'll program PLD's the easy way by using inexpensive software that will run on any standard IBM-PC or compatible.

Software to design hardware

A considerable number of programs are presently available that will help you design a programmable IC: We looked at the earliest program, Monolithic Memories' PALASM, in the previous part of this series.

The idea of PALASM was to make programmable-logic design more accessible to digital designers working in small design houses. Previously, the large programmable-logic distributors (such as Harris and Signetics) had sought to sell only to "captive" markets—large companies like IBM and AT&T. In that way, one service-support engineer could be assigned to each large customer, thereby providing efficient service at a low cost to the distributor. (Harris and Signetics reasoned that providing adequate support for many smaller companies would be too expensive.)

With PALASM, however, even the smallest design house could easily use programmable logic, so reckoning that it would boost the company's sales of programmable IC's, Monolithic Memories made the software available free of charge to qualified businesses.

The strategy worked, and Monolithic Memories rapidly became the largest programmable-logic distributor in the world at that time. The other players in the market were quick to follow suit. For example, Sig-

netics, the company that had pioneered the commercialization of PLD's, realized it was losing market share to the newcomer, and released its own design software toolkit, called AMAZE. Signetics also provided its software free of charge to digital-design houses.

However, as we discussed in Part 1, the FPLD architecture from Signetics has both a programmable AND plane and a programmable OR plane, whereas Monolithic Memories' PAL architecture has only a programmable AND plane. As a consequence, PALASM cannot program FPLD's, and AMAZE cannot program PAL's. (Which is why MMI and Signetics gave away rather than sold their software—anyone using the software had to use the corresponding IC's.)

It is therefore hardly surprising that some independent software companies seized on the opportunity to provide a software tool for both device types. Assisted Technologies developed the first universal PLD design tool, CUPL, which is now sold by PCAD (1290 Parkmore Ave., San Jose, CA). Indeed, many of the design examples you get with CUPL still have the name Assisted Technologies on them.

The software ABEL, developed by Data I/O (10525 Willows Rd. NE, Redmond, WA 98073), does much the same thing as CUPL, but is much easier to use, although it is a great deal more expensive (around $1,000) than hobbyist-versions of CUPL.

Getting started

A starter kit version of CUPL is available for $50 from JDR Microdevices (110 Knowles Drive, Los Gatos, CA 95030). In addition to the software, the kit contains one each of four ready-to-program PAL's: the 16L8, 16R8, 16R6, and the 16R4; their characteristics are shown in Table 1. Keep in mind that the starter-kit CUPL can only program those four devices. The supplied PAL's are actually manufactured by Texas Instruments, but they are functionally identical to the same parts from Monolithic Memories.

How the software works

All the software tools for programmable logic design have a similar structure, whose basic design flow is shown in Fig. 1. As you can see, there are a number of different ways where-

TABLE 1

FUNCTION	16L8	16R4	16R6	16R8
DEDICATED INPUTS	10	8	8	8
DEDICATED REGISTERED OUTPUTS (TRI-STABLE)	—	4	6	8
BI-DIRECTIONAL COMBINATIONAL OUTPUTS (TRI-STATABLE)	8	4	2	—

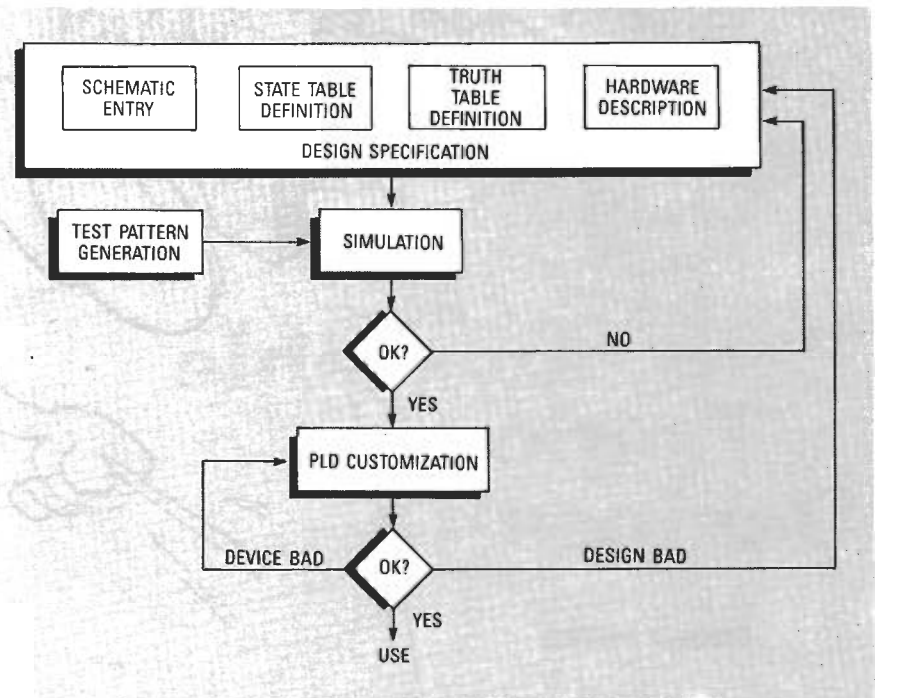


FIG. 1—THE FIRST STEP IN SPECIFICATION ENTRY can be done by schematic, state-table definition, truth-table definition, or a description of the hardware itself.

by the first step—the design specification entry—can be done: by schematic, state-table definition, truth-table definition, or a description of the hardware itself. In the commercial world, engineers like to use a schematic-entry system, whereby the logic-circuit schematic is entered in the computer and the software is then able to derive the fuse map from the schematic.

The most commonly used drafting software used to generate fuse maps is the DASH drafting software from Data I/O's subsidiary, FutureNet. Both ABEL and AMAZE accept data from DASH, while CUPL accepts schematics designed on PCAD's design system, PCAD-CAE1. Since DASH and CAE1 cost $25,000 and $7,500 respectively, they are beyond the budget of most hobbyists.

Although PALASM, AMAZE, CUPL, and ABEL do not accept schematics by themselves, hobbyists can use the design-entry systems actually built into the software and enter a high-level description of the logic in a text-form input file.

CUPL

CUPL is the highest language, hence, it's the easiest for the hobbyist to use. Using an ordinary word processor, such as WordStar or XyWrite, you can create a logic-description file. CUPL then performs both automatic logic minimization of the file and compiles a documentation file having the extension .PLD. For example, if we described a logic description of a two-bit counter in CUPL format and told CUPL to give it the name DIVIDER, CUPL would create a disk file DIVIDER.PLD.

There are separate fields in the input file: *header information* (included at the top of all the files created by CUPL, so you can make sense of your old printouts), *notes*, *pin labels*, and *logic description*. In CUPL, you don't include the part number in the

```
CUPL            2.11c  Serial# 2-99999-001
Device          p16r8  Library DLIB-f-23-10
Created         Tue Jan 01 00:37:57 1980
Name            Divider
Partno          EM0001
Revision        02
Date            8/30/87
Designer        E. Meyer
Company         VLSI
Assembly        Breadboard
Location        U2
*QP20
*QF2048
*QV13
*G0
*F0
*L0512 01111111111111111111111111111111
*L0544 11111111110111111111111111111111
*L0768 01111111111111111111111111111111
*L0800 11111111110111101111111111111111
*L0832 11111111110111101111111111111111
*C131A
*P 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
*V0001 C1XXXXXXXN0XXXXLLXXN
*V0002 C1XXXXXXXN0XXXXLLXXN
*V0003 C1XXXXXXXN0XXXXLLXXN
*V0004 C1XXXXXXXN0XXXXLLXXN
*V0005 PXXXXXXXXN0XXXX11XXN
*V0006 C0XXXXXXXN0XXXXLLXXN
*V0007 C0XXXXXXXN0XXXXLHXXN
*V0008 C0XXXXXXXN0XXXXHLXXN
*V0009 C0XXXXXXXN0XXXXHHXXN
*V0010 C0XXXXXXXN0XXXXLLXXN
*V0011 C0XXXXXXXN0XXXXLHXXN
*V0012 C0XXXXXXXN0XXXXHLXXN
*V0013 C0XXXXXXXN0XXXXHHXXN
*E797
```

FIG. 2—THIS IS THE FUSE MAP compiled by a CUPL for a 16R8 PLD.

logic-description file, although it helps to write your intended part number in a notes field so that you can remember the target device for which the pin assignments were specified. Note that the pins must be labeled before the logic equations; otherwise, CUPL cannot tell what pins the logic should be attached to.

CUPL then uses DIVIDER.PLD to create additional files: DIVIDER.DOC (a documentation file), DIVIDER.LST (a file showing all the line numbers given to the lines in A:DIVIDER.PLD—so if there's an error message reported for a particular line, you can find it), DIVIDER.ABS (a binary code file), and DIVIDER.JED (the fusemap file).

One of CUPL's advantages is that it creates an error file. If you make an error in the file used to blow the fuses, CUPL reports the error and its location. If there's an error in your .PLD file (which is likely on your first attempt), CUPL reports the error and its line number in an .LST file, whose information is used to correct the original .PLD file.

Although CUPL programming might appear to be time-consuming, it won't take very long after you've done it a couple of times. Figure 2 shows the fusemap for a 16R8 that was compiled by a CUPL .PLD file with the filename DIVIDER.

We can, if we choose, edit the fusemap directly. In fact, we can create the fusemap directly with the fusemap editor, and not bother with the logic description at all. However, if we do that we cannot then simulate the design.

A logic simulator is a program that processes two input files to create one output file. The first input file contains a logic-description. The second input is a *test pattern* file; a set of logical ones and zeros that you want to put into the circuit. The simulator puts the ones and zeros into the software model and outputs the pattern of ones and zeros that the logic simulator thinks the device will make.

PLD programmers

Simple PLD programmers are available from a number of sources, among them JDR, whose PLD programmer sells for about $300. Most modern PLD programmers are run by a control board that uses one adapter slot in an IBM-compatible computer. An umbilical cord connects the control board to an external box containing a ZIF (Zero *I*nsertion *F*orce) socket into which PLD's are inserted.

Every PLD type has its own unique internal fuse arrangement. Most PLD's use different pins for programming. The JEDEC fusemap produced by CUPL (the .JED file), includes a section that tells the programmer which pins are where. To program a PLD, one pin is raised to a high-voltage (typically 12 to 25 volts), which puts the IC in the programming mode. The high voltage enables the MODE pin to double as a normal signal pin when the device is not being programmed. The high voltage level also ensures that power-supply glitches will not set the PLD into the programming mode during normal operation, which would be disastrous.

Each fuse to be blown can then be addressed by another set of pins that act as fuse-address lines for the device in the programming mode. The addressed fuse is blown when a final pin is toggled high.

Since two pins are needed for power and ground, one further pin is needed for setting the programming mode, and yet another pin is needed to trigger the fuse-blowing operation; only 16 pins are left for addressing fuses on a 20-pin PLD, meaning there are 2^{16} fuses. Devices with a larger number of fuses can use a multiplexed address bus to define all the possible fuse locations. However, the only parts that most hobbyists will use that are multiplex-addressed are PROM's.

However, a designer doesn't need to worry about exactly how a PLD programmer works. A software *shell* supplied with the programmer on floppy disk disguises the operation of the actual hardware from the user. The designer specifies the type, manufacturer, and JEDEC file to use during programming: the programmer blows the PLD and checks that the blow is performed as expected by the user.

Unusual architectures

Although most PLD's have 20 or 24 pins, some larger PLD's have as many as 64 pins. Currently, the largest PLD device with a standard architecture contains about 2,000 equivalent

gates, making the IC big enough to contain an entire 32-bit floating-point accelerator. (Available from Intel, 1900 Prarie City Rd., Fulsom, CA, and Altera Corp. 3515 Monroe, Santa Clara, CA), Unfortunately, the sheer size of that IC really places it beyond the capabilities of standard ABEL and CUPL software. Stand-alone design systems priced at around $5000, which both include schematic entry, have been developed by Intel and Altera Corp. for that massive chip.

Even more

An even larger IC, the X3090, from Xilinx (2069 Hamilton Ave., San Jose, CA 95125), which uses multiplexers instead of fuses, contains about 9,000 equivalent gates, and the device can be configured "on the fly" with standard logic levels. Just to confuse things, people still refer to the internal connections—the multiplexers—as fuses. Design systems for the *X3090* are presently priced in excess of $20,000, although prices are destined to drop.

The Xilinx *X3090* is just one part with the new architectures that are just emerging in PLD technology. Another IC with unusual architecture is the 39V18 from Lattice Semiconductor (5555 Northeast Moore Ct., Hillsboro, OR 97124). The 39V18 is unique in that it was specially designed to emulate all the standard PLD architectures. Fuses in all the macros enable them to be configured like any of the macros in a range of PLD's; thus, one Lattice part can be a direct-pin replacement for a large number of PLD's in existing designs.

Beating the equivalent gate

We have previously alluded to *equivalent gates*. In essence, the density of customizable components is measured by the number of 2-input NAND gates that would be required to perform the same function for the largest circuit that can be configured in a programmable IC. However, that measurement can be very misleading. The logic in the macros, and their interconnection, really determines the power of a PLD.

For example, a 16R8, which contains eight flip-flops, is perfect for a complex function requiring eight counting stages. A 16L8, which contains about 50 less gates, also contains eight macros, but it does not contain any flip-flops in the macros: Two macros are needed to make a flip-flop. Therefore, at most, the 16L8 can contain a 4-bit counter—half the size of the 16R8.

But, then you must wonder, why is the equivalent gate count of the 16L8 so high? Well, the large *fan-in* into each of the macros—there are eight separate summing inputs into each macro—enables the building of very large sums-of-product terms. Functions that need gates with a large number of inputs therefore fit particularly well into a PLD. Bus decoding and state machines both fall into that category.

A 16L8 can provide eight bus-decoding functions at the same time. If a circuit has an 8-bit bus that turns on eight peripherals at eight separate and distinct addresses, then all eight addresses can be decoded by one IC. That accounts for the high equivalent-gate count. An equivalent SSI implementation could require as many as 14 quad 2-input NAND-gate devices.

In view of the particular suitability of PLD architectures to bus decoding, some manufacturers have enhanced the power of the programmable-plane structure by combining standard logic functions into the programmable IC's. Harris Corp. (Semiconductor Section, PO Box 883, Melbourne, FL 32901), the company that pioneered programmable logic, was the first to take that architectural path. Harris' *82C339* combines a multiplexed bus interface with a summing plane that acts as a programmable comparator. As a consequence, the device can produce four decoded outputs from a 16-bit bus multiplexed onto eight signal lines, as implemented by the 8088 (the microprocessor in the IBM-PC). Intel has made a special PLD called the BIC (*B*us *I*nterface *C*ontroller), which contains eight bidirectional latches, with the control-logic lines for the latches fed by a conventional programmable AND plane.

Registered PLD's are good for state machines. The output of state-machine devices depend on the previous input as well as the current input. Since a 16R8 contains eight flip-flops, the device can record eight states, with one flip-flop putting out a high logic level for each *on* state. Each flip-flop's output can be fed back into the array and logically combined with the other flip-flop states and the current inputs to switch the device into the next state.

Monolithic Memories, Altera, and Signetics have all made *programmable sequencers* containing a large number of extra "buried" flip-flops to contain "buried states" (states that do not cause any change at the outputs). .MDNM/

Altera and Monolithic Memories have taken the alternative approach of combining a programmable AND plane and a PROM into one device, in which PROM outputs can be fed back into the PLD portion. The PLD can combine that data and the inputs to produce a new address in the PROM to go to. Up to 256 states are supported by those IC's. Unfortunately, CUPL is not capable of programming those state machines as yet, although ABEL can program some of them. With time, of course, all of those devices will come within the reach of the hobbyist.

Process technology

We have already tracked the development of new PLD architectures, from the very first programmable device to the most recent innovations. To make things easier to follow, in the course of our discussion we sidestepped the advances in process technology (how the devices are manufactured), although they have also been very important in PLD development.

Originally, PLD's were all made using TTL (*T*ransistor-*T*ransistor-*L*ogic). Modern variations of TTL are very fast, but all types of TTL devices use a great deal of power. MOS technology, which uses electric fields rather than current to switch the gates, uses much less power and packs higher densities of transistors into ever smaller areas of silicon. Most modern devices are CMOS. As a matter of fact, except for the programmable sequencer from Signetics, all the advanced architectures we have discussed in this article use CMOS technology.

In the first part of this article we discussed the difference between single- and dual-level metal. Typically, densities higher than 2,000 gates, and I/O delays of less than 45 nanoseconds (speeds higher than 15 MHz) are not possible with single-level metal. In some situations, I/O delay is very important.

Different fuses

We also discussed different fuse technologies. As you might re-

member, nichrome and polysilicon fuses were the first types to be used. Nichrome fuses are still in use. Currently, tungsten, and sometimes titanium, are added to the fuse material because they burn out more cleanly. Also, the electric field across the vacated space left by a vaporized fuse can cause *metal migration*. In other words, over time the metal ions are magnetically dragged back into the fuse cavity; that is, the fuses can actually grow back. Thankfully, *grow back* is relatively rare with modern technology and standard operating conditions.

With some modern PLD's, the reverse of metal growback—called *avalanche-induced migration*—is actually used to remove the fuse. Also, lower temperatures can be used with avalanche-induced migration, which increases device reliability. Further, the fuses in some modern PLD's are fashioned in a *bow-tie* shape rather than the traditional *hourglass*. The sudden narrow taper at the point of fuse burnout in bow-tie fuses reduces the rate of metal migration.

Besides traditional fuse technologies, several newer processes are now used for manufacturing PLD's, which allow the device to be reprogrammed.

Those erasable PLD's (usually called EPLD's) use the same technology as EPROM's in that they don't use fuses at all, but rather contain *floating gates*. Those are small semiconducting regions between the two

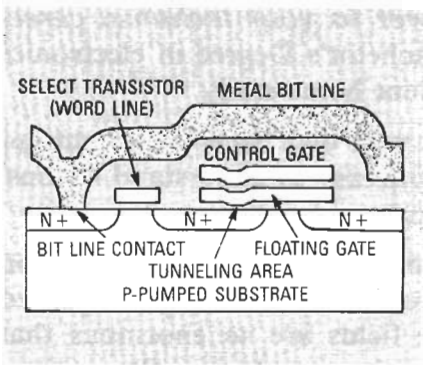


FIG. 3—THE STRUCTURE OF a floating gate. A charge on the floating gate creates a small electrical field that acts as a conducting link between the two metal levels.

metal levels. The electrical characteristics of the floating gate are very carefully selected so that its polarity is not affected by normal +5-volt conditions. But the floating gate can be charged up in the high-voltage programming mode. The charge creates a small electrical field that acts as a conducting link between the two metal levels. On the other hand, uncharged gates do not act as a connection. Figure 3 shows the structure of a floating gate.

EPLD's are usually programmed at a higher voltage level than standard PLD's (which reduces the rate the charge can leak off a charged gate), placing them beyond the reach of most inexpensive PLD programmers.

EPLD's, like EPROM's, contain small *windows* that allow them to be wiped clean by exposure to ultraviolet light. Thus EPLD's can be reprogrammed by the circuit designer.

In the most recent PLD technology developments, EPLD's have largely been replaced by EEPLD's, which are like EEPROM's. Since "E-squared" devices, as they are called, use electricity rather than ultraviolet light for the cleaning procedure, they don't need windows, thus making the package less expensive, even though the silicon is more difficult to make. Lattice Semiconductor uses E-squared for the *39V18*.

EPLD and EEPLD technologies allow the factory to check that the devices are fully functional before shipping by programming and checking a pattern. If there's an error, the PLD can be wiped clean and reprogrammed. Also, designers can reuse the same device for different circuits when prototyping. Similarly, hobbyists can use the same device over and over, in different projects.

Conclusion

PLD's are becoming commonplace in a diverse variety of applications. In computers, PLD's are particularly suitable for use in the state machines that control such system-level operations as start-up sequences, interrupt handling, control transfer, I/O arbitration, and peripheral-processor control. In the consumer world, programmable logic has found its way into video/audio control systems, washing machines, toys, automobile dashboards, and even traffic lights and elevators. Programmable logic is also common in military and aerospace applications, as well as in hospital equipment, nautical navigation systems, and telecommunications.

Quite possibly, in the near future a single PLD might provide all the circuits for any kind of device because PLD implementation avoids the process of "gluing" circuits using custom IC fabrication. By using a PLD instead of a custom IC, virtually any logic circuit can be almost instantly refabricated by simply creating a new fuse map and using it to blow a new device.

Figure 4 shows some of the materials that can be used by the average hobbyist and technician to design and create custom PLD circuits. **Radio-Electronics** will soon feature an article on how you can use that equipment to design and blow useful experimenter PLD circuits. **R-E**

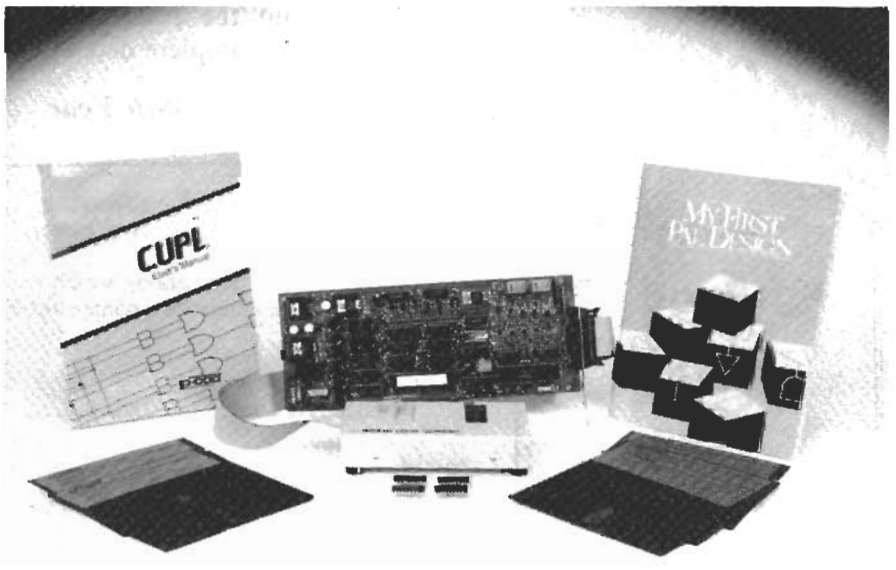


FIG. 4—A PLD PROGRAMMER CARD and an inexpensive PAL introduction kit containing four different PLD's and a simplified version of CUPL are all you need to get started in PLD design. Both are available at JDR Microdevices.