COMBINATIONAL CIRCUITS

Combinational (combinatorial) circuits realize Boolean functions and deal with digitized signals, usually denoted by 0s and 1s. The behavior of a combinational circuit is memoryless; that is, given a stimulus to the input of a combinational circuit, a response appears at the output after some propagation delay, but the response is not stored or fed back. Simply put, the output depends solely on its most recent input and is independent of the circuit's past history.

Design of a combinational circuit begins with a behavioral specification and selection of the implementation technique. These are then followed by simplification, hardware synthesis, and verification.

Combinational circuits can be specified via Boolean logic expressions, structural descriptions, or truth tables. Various implementation techniques, using fixed and programmable components, are outlined in the rest of this article. Combinational circuits implemented with fixed logic tend to be more expensive in terms of design effort and hardware cost, but they are often both faster and denser and consume less power. They are thus suitable for high-speed circuits and/or high-volume production. Implementations that use memory devices or programmable logic circuits, on the other hand, are quite economical for low-volume production and rapid prototyping, but may not yield the best performance, density, or power consumption.

Simplification is the process of choosing the least costly implementation from among feasible and equivalent implementations with the targeted technology. For small combinational circuits, it might be feasible to do manual simplification based on manipulating or rewriting logic expressions in one of several equivalent forms. In most practical cases, however, automatic hardware synthesis tools are employed that have simplification capabilities built in. Such programmed simplifications are performed using a mix of algorithmic and heuristic transformations. Verification refers to the process of ascertaining, to the extent possible, that the implemented circuit does in fact behave as originally envisaged or specified.

A half adder is a simple example of a combinational circuit. The addend, augend, carry, and sum are all single binary digits or bits. If we denote the addend as A and the augend as B, the Boolean function of carry-out C_o and sum S can be written as

$$C_o = AB$$

$$S = A \oplus B = \overline{A}B + A\overline{B}$$

The carry-out and sum functions can also be specified in the form of a truth table with eight rows (corresponding to the eight possible combinations of values for the three Boolean inputs) and two columns in which the values of C_o and S are entered for each of the eight combinations. With this view, we see that C_o and S are two of the $2^{2^3} = 256$ possible functions of three variables. The foregoing is justified by noting that each of the $2^3 = 8$ rows of the truth table for a 3-variable function can be filled out in two ways, thus leading to 2^8 choices overall.

The process of designing combinational circuits involves certain levels of abstraction. For structured circuit implementation, the key is to find high-level building blocks that are sufficiently general to be used for different designs. While it is easy to identify a handful of elements (such as AND, OR, and NOT gates) from which all combinational circuits can be synthesized, the use of such simple building blocks reduces the component count by only a modest amount. A more significant reduction in component count may be obtained if each building block is equivalent to tens or hundreds of gates.

Popular combinational circuits that are more complex than single gates include the following.

- *Multiplexer*: In its simplest form, a multiplexer (mux, for short) has two data inputs x_0 and x_1, a selection signal s, and an output Y that equals x_s, that is, one of the two inputs, chosen based on the value of s. The foregoing 2-to-1 mux can be readily generalized to one with 2^L data inputs and L selection signals, as we shall see shortly.
- *Decoder*: An L-to-2^L (or L-input) decoder has 0*s* on $2^L - 1$ of its outputs and 1 on the single output whose index (in binary) is represented by the L input bits. So, if the L-bit input pattern is 01001, representing the number 9 in binary, the output Y_9 will be 1 and all other outputs will be 0*s*.
- *Encoder*: The operation of encoding is the opposite of decoding. Here, there are 2^L inputs, at most one of which (say, x_i) is 1. The L-bit output is the binary representation of the index i of the single active input line. A *priority encoder* can deal with multiple active inputs. It sets the L outputs to the index of the first such active input.

A commonly used building-block approach is based on array structures. Programmable logic devices (*PLD*s) are composed of primitive gates arranged into logic blocks whose connections can be customized for realizing specific functions. Programmable elements are used to specify what each logic block does and how they are combined to produce desired functions. This fundamental idea is used in connection with various architectures and fabrication technologies to implement a wide array of different PLDs.

IMPLEMENTATIONS WITH FIXED LOGIC

If the input-output behavior of the combinational circuit is defined by means of a logic statement, then the statement can be easily expressed in sum-of-products form using Boolean algebra. Once in this form, its implementation is a relatively straightforward task. In the following, we will consider the implementation of combinational circuits using gate networks and multiplexers. These are fixed (as opposed to programmable) logic devices in the sense that they are used by suitably interconnecting their input/output terminals, with no modification to the internal structures of the building blocks.

Using Gate Networks

Let us begin with the Boolean function D defined as

$$D = AB + B\overline{C}$$

where A, B, and C are input variables whose values can be either 0 or 1. Direct implementation based on the preceding expression would require three chips: one that contains inverters (such as 7404), one that contains two-input AND gates (such as 7408), and one that contains two-input OR gates (such as 7432). Rewriting the logic expression for D as

$$D = B(A + \overline{C})$$

reduces the number of gates from 4 to 3, but does not affect the component or chip count discussed in the preceding.

By applying DeMorgan's theorem, we can derive an equivalent logic expression for our target Boolean function that can be implemented using a single chip containing only NOR gates (such as 7402).

$$D = \overline{\overline{B} + \overline{(A + \overline{C})}}$$

Similarly, DeMorgan's theorem allows us to transform the logic expression into one whose implementation requires only NAND gates:

$$D = \overline{(\overline{AB})(\overline{B\overline{C}})}$$

Figure 1 shows the three gate network implementations of D using NOT-AND-OR, NOR, and NAND gates, as discussed in the preceding. The output D of such a combinational gate network becomes available after a certain delay following the application of its inputs. With gate-level components, the input-to-output delay, or the latency, of a combinational circuit depends on the number and types of gates located on the slowest path from an input terminal to the output. The number of gate levels is a rough indicator of the circuit's latency.

Practical combinational circuits may contain many more gates and levels than the simple examples shown in Fig. 1. As combinational circuits are often placed between synchronously clocked storage elements, or *latches,* the circuit's latency dictates the clock rate and, thus, the overall system speed. One way to improve the computation rate, or *throughput,* is to partition the gates into narrow slices, each consisting of only a few levels, and buffer the signals going from one slice to the next in latches. In this way, the clock rate can be made faster and a new set of inputs processed in each clock cycle. Thus, the throughput improves while both latency and cost deteriorate due to the insertion of latches (see Fig. 2).

Today, digital implementation technologies are quite sophisticated and neither cost nor latency can be easily predicted based on simple notions such as number of gates, gate inputs, or gate levels. Thus, the task of logic circuit implementation is often relegated to automatic synthesis or CAD tools. As an added benefit, such tools can take many other factors, besides cost and latency, into account. Examples of such factors include power consumption, avoidance of hazards, and ease of testing (testability).

Realizing combinational circuits by means of specially designed gate networks constitutes the *ASIC* (application-specific integrated circuit) approach. The circuit is tailored to computing a particular function and cannot be used for any other function, unless its components are modified or augmented. In the rest of this article, we focus on realizations based on general or flexible circuits that can be molded (programmed) to fit the needs of a multitude of functions.

Using Multiplexers

The application of discrete logic circuits becomes impractical as our Boolean expression grows in complexity. An alternative solution might be the use of a multiplexer. To implement the Boolean function with a multiplexer, we first expand it into unique *minterms;* each of which is a product term of all the variables in either true or comple-

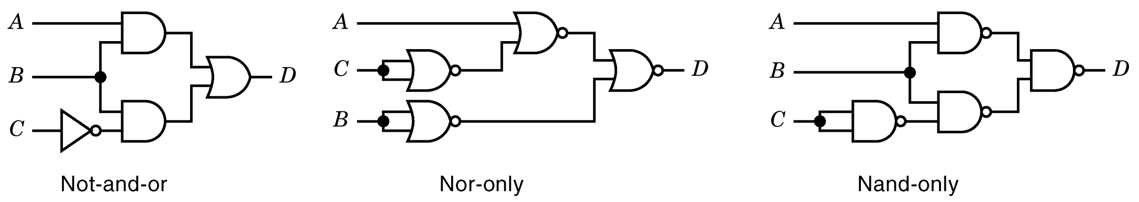

Figure 1. Realizing the Boolean function $D = AB + B\overline{C}$ by gate networks.

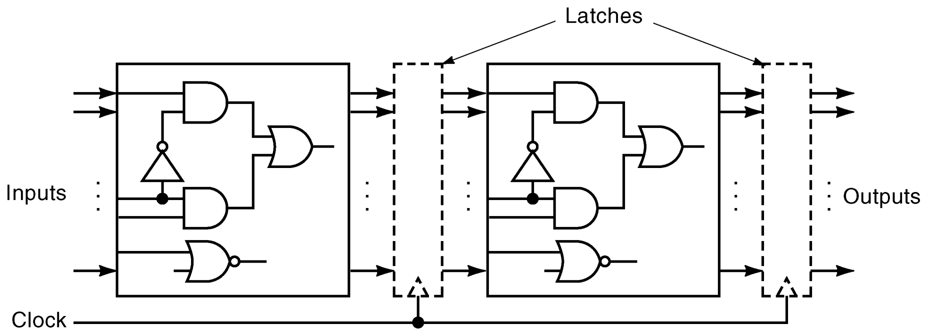

Figure 2. Schematic of a pipelined combinational circuit.

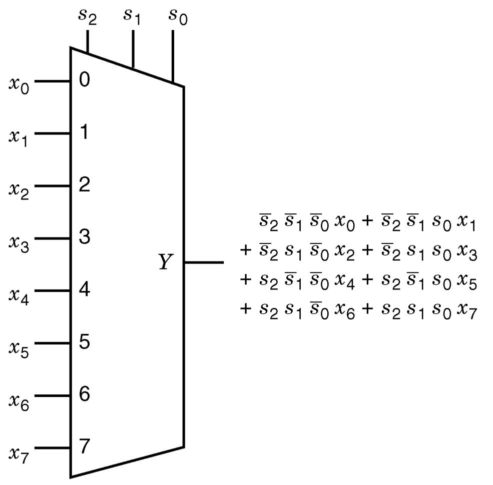
s_2
s_1
s_0
x_0
x_1
x_2
x_3
x_4
x_5
x_6
x_7
0
1
2
3
4
5
6
7
Y
$\bar{s}_2\,\bar{s}_1\,\bar{s}_0\,x_0 + \bar{s}_2\,\bar{s}_1\,s_0\,x_1 + \bar{s}_2\,s_1\,\bar{s}_0\,x_2 + \bar{s}_2\,s_1\,s_0\,x_3 + s_2\,\bar{s}_1\,\bar{s}_0\,x_4 + s_2\,\bar{s}_1\,s_0\,x_5 + s_2\,s_1\,\bar{s}_0\,x_6 + s_2\,s_1\,s_0\,x_7$

Figure 3. A multiplexer or selector transfers one of its "data" inputs to its output depending on the values applied to its "select" inputs.

ment form

$$D = AB(C + \overline{C}) + (A + \overline{A})B\overline{C} = ABC + AB\overline{C} + \overline{A}B\overline{C}$$

With L input variables, there are 2^L possible minterms, each corresponding to one data line of a 2^L-to-1 multiplexer. Figure 3 shows an 8-to-1 multiplexer and the logic expression for its output. A 2^L-to-1 multiplexer can be used to implement any desired L-variable Boolean function by simply connecting the input variables to its select lines, logic 1 to the data lines corresponding to the minterms, and logic 0 to the remaining data lines. The select inputs s_2, s_1, and s_0, when viewed as a 3-bit binary number, represent an index i in the 0 to 7 range. The value on data line x_i is then chosen as the output.

To implement a Boolean function with more variables than can be accommodated by a single multiplexer, we can connect other multiplexers to the x_i inputs of Fig. 3 to obtain a multilevel multiplexer realization. For example, to implement a 6-variable function, we can expand it in terms of three of the variables to obtain an expression similar to the one shown on the output in Fig. 3, where the x_i are *residual functions* in terms of the remaining variables.

Figure 4 shows how the function D can be implemented by an 8-to-1 multiplexer. We can view the single line entering each AND gate as representing multiple inputs. In effect, we have an 8-bit memory whose hardwired data are interrogated by the input variables; the latter information filters through the decoder, which finds the corresponding data line and selects it as the output.

With a multiplexer that can supply both the output D and its complement $\overline{D}$, we can choose to tie the minterms to logic 1 and the remaining data lines to logic 0, or vice versa. This, again, is an application of DeMorgan's theorem.

A 2^L-to-1 multiplexer can be implemented as an L-level network of 2-to-1 multiplexers. This becomes clear by noting that a 2-to-1 multiplexer is characterized by the equation

$$Y = \bar{s}x_0 + sx_1$$

and that the output logic expression for the 8-to-1 multiplexer of Fig. 3, say, can be written as:

$$Y = \bar{s}_2(\bar{s}_1(\bar{s}_0x_0 + s_1x_1)) + s_1(\bar{s}_0x_2 + s_0x_3)) + s_2(\bar{s}_1(\bar{s}_0x_4 + s_0x_5)) + s_1(\bar{s}_0x_6 + s_0x_7))$$

Another way to justify the preceding is to note that a 2-to-1 multiplexer can act as a NOT, AND, or OR gate:

NOT:	$x_0 = 1,\ x_1 = 0$	yields	$Y = \bar{s}$
AND:	$x_0 = 0$	yields	$Y = sx_1$
OR:	$x_1 = 1$	yields	$Y = s + x_0$

We have just concluded our examination of a simple programmable logic device. The basic elements include a means to store data, a decoding function to retrieve data, and an association of data with logic values. In the case of a multiplexer, the programmability is provided by manual wiring. Slightly more complicated schemes use fuse or

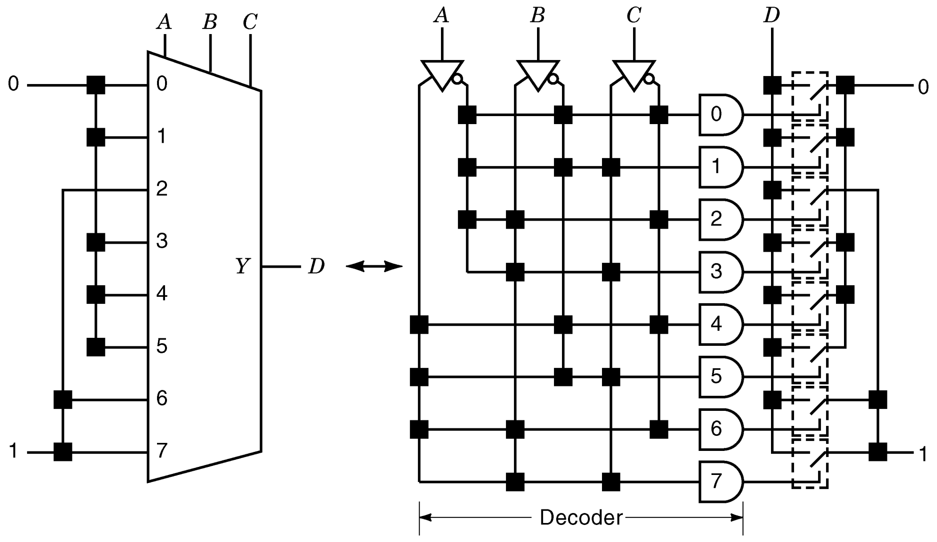

Figure 4. Realizing the Boolean function $D = AB + B\overline{C}$ by an 8-to-1 multiplexer.

antifuse elements. A fuse is a low-resistance circuit element that can be opened permanently by a relatively high surging current, thus disconnecting its endpoints. An antifuse is the opposite of a fuse; it is an open circuit element that can be made permanently low resistance. Both fuse and antifuse offer one-time programmability (*OTP*). Once programmed, they cannot be modified.

IMPLEMENTATIONS WITH MEMORY DEVICES

Multioutput Boolean functions can be implemented by several multiplexers connected in parallel. However, it seems wasteful to have multiple decoders, especially when the number of variables is large. Removing all but one of the replicated decoders in the multiplexers and making the hardwiring changeable lead to a memory structure, as shown in Fig. 5. This approach of logic being embodied in the memory content is the well-known table-lookup method for implementing Boolean functions.

Table lookup is attractive for function evaluation as it allows the replacement of irregular random logic structures with much denser memory arrays. The input variables constitute an address that sensitizes a word select line and leads to the stored data in that particular word being gated out. As in the case of the multiplexer, the values to be stored are related to the minterms of the Boolean function. Thus, the content of each memory column in Fig. 5 is the truth table of the associated output function.

Figure 6 shows the use of an 8×2 bit memory device to implement a full adder. The full adder is a half adder augmented with a single-bit carry-in C_i and is specified by the Boolean functions

$$C_o = AB + AC_i + BC_i$$
$$S = A \oplus B \oplus C_i = ABC_i + A\overline{B}\overline{C_i} + \overline{A}B\overline{C_i} + \overline{A}\overline{B}C_i$$

In general, memory cells can be classified in two major categories: read-only memory (*ROM*) (in some cases, read-mostly), which is nonvolatile, and random-access memory (*RAM*) (read-write memory is a better designation), which is usually volatile. They are distinguished by: (1) the length of write/erase cycle time compared with the read cycle time; and (2) whether the data are retained after power-off. Programmability refers to the ability to write either a logic 0 or 1 to each memory cell, which in some cases must be preceded by a full or partial erasure of the memory content (such as in EPROM and EEPROM). In this respect, PLDs are no different and actually use some form of memory in their structures.

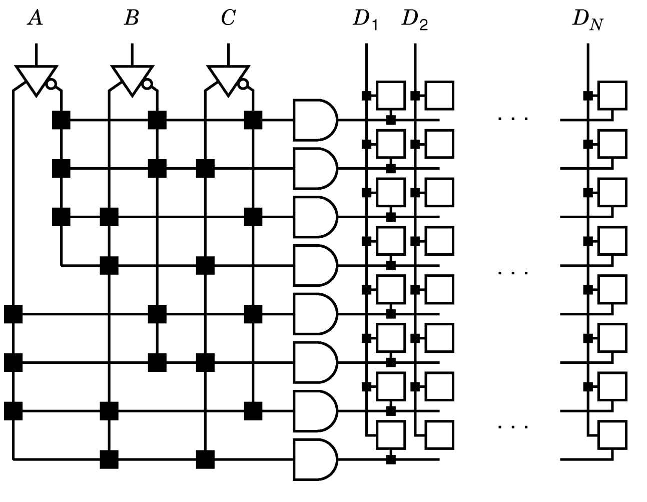

Figure 5. The read path of a memory device goes through the address decoder and the memory array. Such a device can be viewed as a multiplexer with multiple outputs.

Strictly speaking, implementations of Boolean functions based on such memory devices cannot be viewed as combinational. Many PLDs are in fact *sequential* in nature. They become combinational only because the clocked latches are bypassed. However, the programming will never occur in operation and, in some cases, is limited to a certain maximum number of times during the life of the device. Thus, between programming actions, even such *latched* or *registered* PLDs behave as truly combinational circuits.

It is noteworthy that in Fig. 5, the programmable elements (memory cells) along each column are wire-ORed together. Intuitively, the programmable elements can also be placed in the decoder so they are wired-ANDed together along each column. These and other variations lead to different building blocks. Programmable logic array (*PLA*) and programmable array logic (*PAL*) are two types of building blocks that are universally used for implementing combinational circuits in PLDs.

IMPLEMENTATIONS WITH PROGRAMMABLE LOGIC

The memory-based implementation of Fig. 5 has the essential feature of array logic, that is, a regular array that is programmable. Array logic operates by presenting an address in the data path to the memorylike structure. Decoding of this address starts the process whereby a predetermined result is extracted from the array. Because the result generated by such an array depends on the content of the array, the Boolean function can, in principle, be changed in the same way as writing into a memory.

Using Programmable Logic Arrays

Instead of expanding the product terms into minterms exhaustively, we take advantage of "don't care" conditions to let the decoder select more than one row simultaneously. Programmable logic devices are organized into an AND array and an OR array, with multiple inputs and multiple outputs. The AND array maps the inputs into particular product terms; the OR array takes these product terms together to produce the final expression. Figure 7 shows a block diagram for the array component.

Figure 8 shows a commonly used scheme for representing the topologies of PLAs. The input variables $x_1, x_2, \ldots, x_L$ and their complements $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_L$ constitute the columns of the AND array. The rows correspond to the product terms $z_1, z_2, \ldots, z_M$ in both the AND and OR arrays. The columns of the OR array represent the Boolean functions y_1, y_2, $\ldots$, y_N in sum-of-products form. The complexity of PLA is

Inputs			Minterms	Outputs	
A	B	C_i		C_o	S
0	0	0	$\bar{A}\bar{B}\bar{C}_i$	0	0
0	0	1	$\bar{A}\bar{B}C_i$	0	1
0	1	0	$\bar{A}B\bar{C}_i$	0	1
0	1	1	$\bar{A}BC_i$	1	0
1	0	0	$A\bar{B}\bar{C}_i$	0	1
1	0	1	$A\bar{B}C_i$	1	0
1	1	0	$AB\bar{C}_i$	1	0
1	1	1	ABC_i	1	1

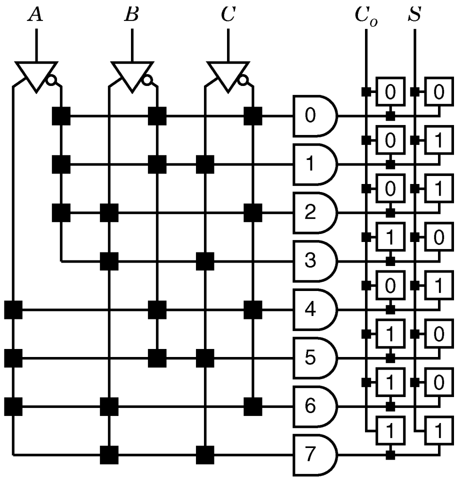

Figure 6. Using memory to realize a full adder. The memory content on the right is in one-to-one correspondence with the truth table on the left.

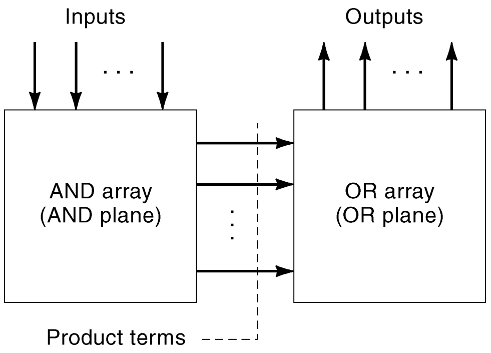

Figure 7. The basic logic array component consists of an AND array and an OR array.

determined by the number L of inputs, the number M of product terms, and the number N of outputs. An L-input, M-product-term, N-output PLA is sometimes referred to as an $L \times M \times N$ device.

The number of product terms is often selected to be much smaller than 2^L (for example, $M = 4L$). There is a penalty for this tremendous compression. Whereas a memory device with its full decoder can generate any function of the input variables, the partial decoder of the PLA device generates a very limited number of product terms.

Because of the severe limitation on the number of available product terms, an aggressive two-level logic minimization method is critical for effective utilization of PLAs. A convenient way to describe a function for PLA realization is through a personality matrix, which is a minor reformulation of the truth table. Figure 9 shows an example for a full adder and the corresponding PLA realization.

For the realization of Boolean functions PLAs are widely used within integrated circuit designs. A distinct advantage is that their regular structures simplify the automatic generation of physical layouts. Except for a handful of stand-alone PLA parts to be used in combinational circuit implementations, PLAs are generally not field-programmable. Rather, they are programmed by using appropriate masks at the time of circuit manufacture.

Multilevel logic structures can be realized with PLAs either by interconnecting several PLAs or by connecting certain of the outputs to the inputs in a single PLA. As an example, an $18 \times 42 \times 10$ PLA can implement the parity or XOR function in two-level AND-OR form for no more than six inputs. The reason is that the seven-input XOR function has 64 minterms which is beyond the capacity of the preceding PLA. Consider the problem of implementing the nine-input XOR function. One way is to divide the inputs into three groups of three and separately realize 3

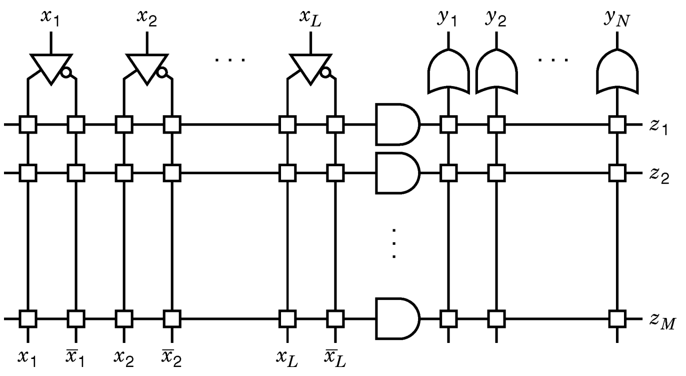

Figure 8. A commonly used scheme for representing the topology of array logic explicitly shows its columns and rows. The cross-points mark the locations of programmable elements whose states may be changed through programming.

Inputs A	B	C_i	Product terms	Outputs[a] C_o	S
1	1	–	AB	1	0
1	–	1	AC_i	1	0
–	1	1	BC_i	1	0
1	1	1	ABC_i	0	1
1	0	0	$A\overline{B}\overline{C}_i$	0	1
0	1	0	$\overline{A}B\overline{C}_i$	0	1
0	0	1	$\overline{A}\overline{B}C_i$	0	1

[a] These specify the connections in the OR array rather than output logic values.

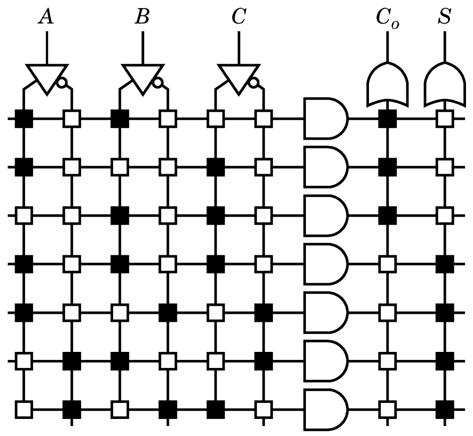

Figure 9. A personality matrix defines the inputs, product terms, and outputs of a PLA.

three-input parity functions using 9 of the inputs, 12 of the product terms, and 3 of the outputs. The preceding three outputs can then be fed back to three of the unused inputs and their XOR formed on one of the available outputs by utilizing four more product terms.

Using Programmable Array Logic

A more common programmable solution is to use PALs. There is a key difference between PLAs and PALs: PLAs have the generality that both the AND and OR arrays can be programmed; PALs maintain the programmable AND array, but simplify the OR array by hardwiring a fixed number of product terms to each OR gate.

For example, the commercial PAL device 16L8 (which means that the device has 16 inputs and 8 outputs, and it is active low combinational) arranges the AND array in 32 columns and 64 rows. Each AND gate has programmable connections to 32 inputs to accommodate the 16 variables and their complements. The 64 AND gates are evenly divided into 8 groups, each group associated with an OR gate. However, there are only 7 AND gates connected to each OR gate and, thus, each Boolean function is allowed to have at most 7 product terms. The remaining one AND gate from each group is connected to a tri-state inverter right after the OR gate, as depicted in Fig. 10. The device shown in Fig. 10 actually has 10 inputs, 2 outputs, and 6 bidirectional pins that can be used as either inputs or outputs.

There exists a fundamental trade-off between speed and capacity in PLDs. It is fair to say that for devices with comparable internal resources, a PLA should be able to implement more complex functions than a PAL. The reason is that the PLA allows more product terms per output as well as product-term sharing; that is, outputs of the AND array can be shared among a number of different OR gates. On the other hand, the PLA will be slower because of the inherent resistance and capacitance of extra programmable elements on the signal paths.

In reality, NOR-NOR arrays may be used, instead of AND-OR arrays, to achieve higher speed and density. (Transistors are complementary, but the N-type is more robust than the P-type and is often the preferred choice.)

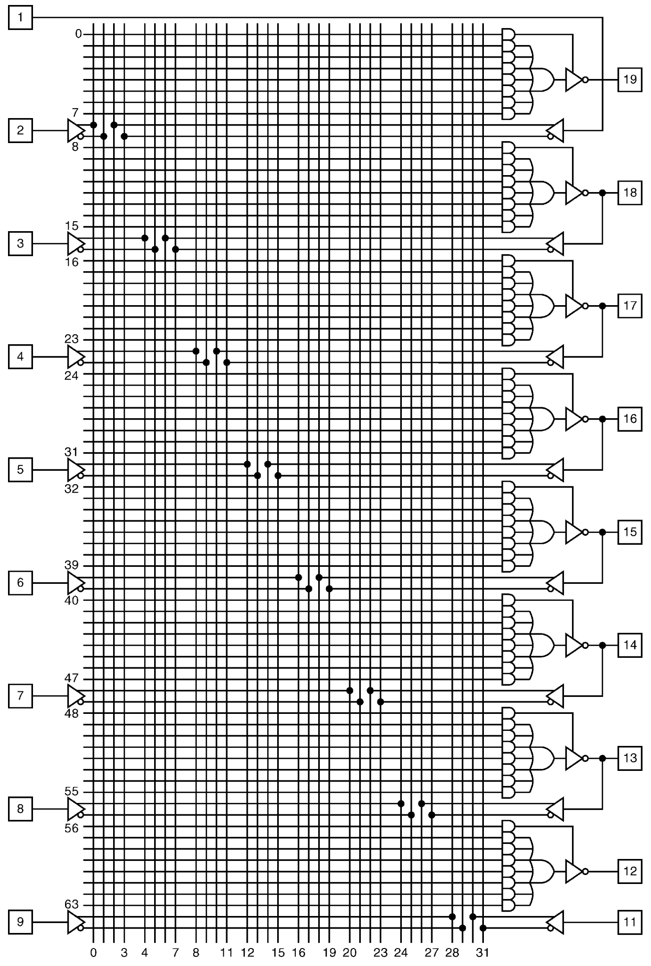

Figure 10. Schematic diagram of the PAL device 16L8 known as its programming map. Locations to be programmed are specified by their numbers (11-bit integers in the range 0 to 2047, composed of a 6-bit row number and a 5-bit column number).

Consider the full adder example. We can rewrite the Boolean functions as follows:

$$\overline{C_o} = \overline{A}\,\overline{B} + \overline{A}\,\overline{C_i} + \overline{B}\,\overline{C_i}$$

$$\overline{S} = \overline{A} \oplus \overline{B} \oplus \overline{C_i} = \overline{A}\,\overline{B}\,\overline{C_i} + \overline{A}BC_i + A\overline{B}C_i + AB\overline{C_i}$$

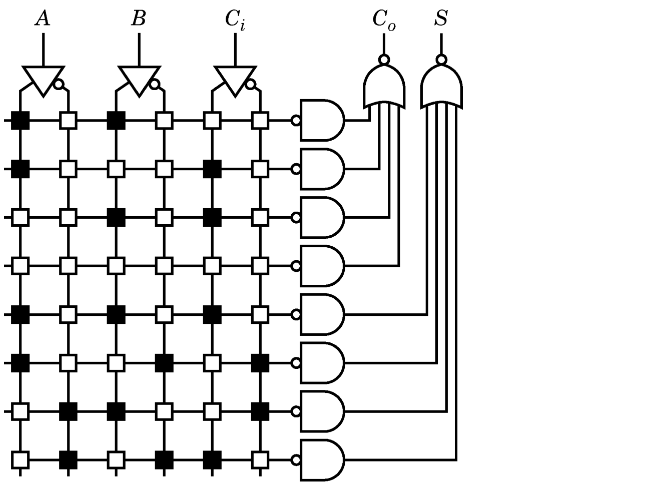

Figure 11. The implementation of a full adder in PAL using NOR gates is equivalent to that using AND and OR gates. The figure assumes that four product terms are tied to each sum term.

The inverted inputs and outputs preserve the original AND-OR structure so the realization is equivalent, as shown in Fig. 11.

As in the case of PLAs, we can use several PALs to implement logic functions that are too complex for the capabilities of a single device. Feeding back the outputs into the array in order to realize multilevel circuits is facilitated by the built-in feedback paths (see Fig. 10). As an example, to implement the 9-input XOR function using the PAL device 16L8 shown in Fig. 10, we can divide the inputs into three groups of 3 and proceed as we did for the PLA implementation. The only difference is that the feedback paths are internal and no external wiring is needed.

Other PLD Variants

Generic array logic (GAL) is a slight enhancement of PAL that includes an XOR gate after each OR gate. The XOR gate can be viewed as a controlled inverter that changes the output polarity if desired. Given that $y \oplus 0 = y$ and $y \oplus 1 = \overline{y}$, we can choose to implement a Boolean function directly or generate its complement and then invert it. As an extreme example, $y = x_1 + x_2 + \cdots + x_{16}$ cannot be implemented by PAL16L8, but it can be easily realized by a similar device that includes the aforementioned XOR gates through implementing $\overline{y} = \overline{x}_1\overline{x}_2, \ldots, \overline{x}_{16}$ and then complementing the result. It is therefore not surprising that most PALs now allow one to control their output polarity through an XOR gate or with a multiplexer that chooses the true or complement result.

The ultimate in flexibility is provided by field-programmable gate arrays (FPGAs) which consist of a regular array of logic blocks with programmable functionalities and interconnections. Figure 12 shows part of a generic FPGA component. Each block can implement one or more simple logic functions, say of four or five logic variables. The inputs to the block can be taken from its adjacent horizontal or vertical signal tracks (channels) and its output(s) can be routed to other blocks via the same channels.

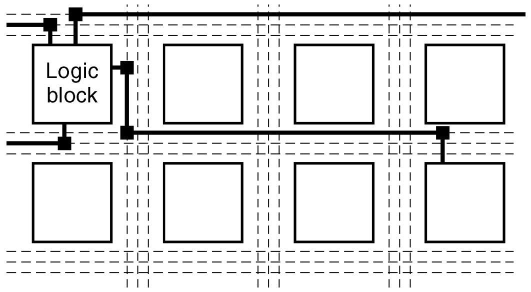

Figure 12. Part of an FPGA, consisting of four rows and two columns of logic blocks and their associated programmable interconnections (channels). The upper left logic block has been configured to receive three inputs from its upper and lower horizontal channels and to send its output to the logic block at the lower right via a vertical and a horizontal channel segment.

The logic blocks of an FPGA store their outputs in storage elements, thus making the result a sequential circuit. Combinational circuits can be implemented by programmed bypassing of the storage elements.

Many FPGAs have two types of blocks: ordinary logic blocks, as described above, and special input/output blocks (typically placed along the chip boundary for direct connection to its pins) that facilitate the interconnection of an FPGA to external circuits and devices. Many modern FPGAs use table lookup to realize the simple function(s) of each logic block. For example, if a 4-variable function is to be realized, a 16×1 table can be embedded into the logic block. This table can then be preloaded with the truth table of any desired function at set-up time. It is possible to combine these small tables with simple arithmetic circuits, such as adders, into a highly efficient *distributed arithmetic* scheme for computing functions of interest.

In order to cover most designs, PLDs are organized to balance speed and capacity within the constraints of fabrication technologies. Because the assemblages of logic blocks are positioned where they are anticipated to be useful to each other, such an approach is necessarily wasteful. On the other hand, the flexibility, short development time, and low cost of PLDs makes them ideal for rapid prototyping of digital circuits and their realization when the expected production volume is low or else the need for debugging and upgrading in the field is envisaged.

BIBLIOGRAPHY

Advanced Micro Devices, *PAL*Ⓡ *Device Data Book*, 1996.

J. W. Carter, *Digital Designing with Programmable Logic Devices*, Englewood Cliffs, NJ: Prentice-Hall, 1997.

H. Flesher, L. I. Maissel, An introduction to array logic, *IBM J. Res. Develop.*, **19** (2): 98–109, 1975.

R. H. Katz, *Contemporary Logic Design*, Redwood City, CA: Benjamin/Cummings, 1994.

Lattice Semiconductor, Introduction to GALⓇ device architectures, *ISP Encyclopedia*, 1996.

B. Parhami, *Computer Arithmetic*: Algorithms and Hardware Designs, New York: Oxford, 2nd ed., 2010.

Philips Semiconductors, *Programmable Logic Devices Data Handbook*, 1993. See also: http://www.datasheetcatalog.com/

K. Tsuchiya, Y. Takefuji, A neural network approach to PLA folding problems, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **15**: 1299–1305, 1996.

BEHROOZPARHAMI
University of California,
Santa Barbara, CA