CHAPTER 7

Physical Implementation on ICs

7.1 INTRODUCTION

A digital circuit design that has been created but just drawn, perhaps with pencil on paper or as a figure in this book, is merely a drawing. Designers must eventually implement that circuit on a real physical device, so that the device can be placed in an electronic product to carry out the desired function. In other words, how do designers get from Figure 7.1(a), the seat belt warning light circuit that was designed in Chapter 2, to Figure 7.1(b), a physical implementation?

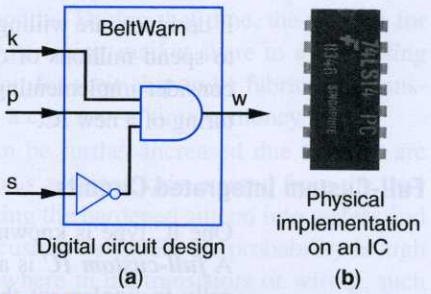


Figure 7.1 How do we get from (a) to (b)?

Most digital circuits today are physically implemented using an integrated circuit device. An ***integrated circuit***, or ***IC***, refers to a piece of semiconductor material (typically silicon) existing inside of a package (typically plastic) like the package of Figure 7.1(b), with all the components of the circuit being integrated on the surface of the silicon. Such integration is in contrast to those components existing as separate components on a board—hence the term "integrated circuit." Because the piece of silicon is cut ("chipped") from a larger wafer of silicon, an IC is commonly referred to as a ***chip***.

Designers can implement a circuit using a variety of available IC types. An ***IC type*** is a category of IC having specific features. Important features that distinguish IC types are the time and cost required to implement the circuit using that IC type, or in the speed, size, power consumption, and cost of the resulting IC. An analogy can be made with the various available car types, such as a sports car versus a family sedan; a sports car is faster but more expensive. Importantly, IC types differ in the steps required to convert a circuit into an implementation—those steps are known as the ***design flow***. For example, Figure 7.2 shows that designers might use an IC type like a full-custom IC or an ASIC (soon to be described) whose design flow requires that the designers spend millions of dollars and several months to manufacture a chip optimized for their circuit, or designers might use an IC type that is premade and that they can program in minutes. This chapter describes and compares several popular IC types for implementing digital circuits, including full-custom ICs, standard cell ICs, gate array ICs, FPGAs, PLDs, and logic ICs.

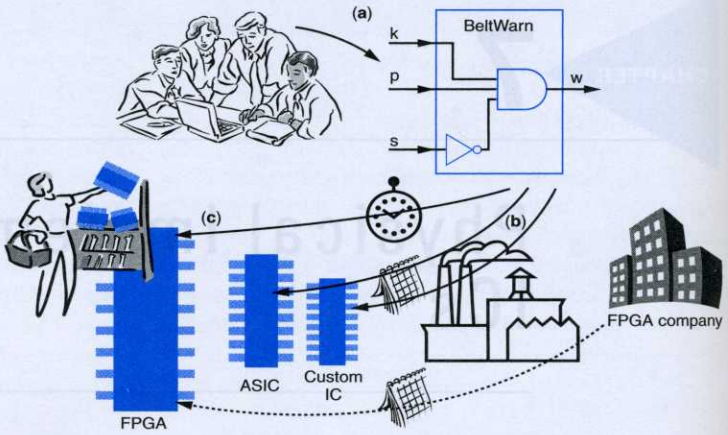


Figure 7.2 Common design flows for different IC types: (a) digital designers first create a desired circuit, (b) some IC types, like full-custom ICs or ASICs, involve manufacturing a new IC, requiring millions of dollars and many months, (c) a programmable IC type is premade and available off-the-shelf, costing tens of dollars and requiring just minutes, but resulting in bigger and slower circuit implementations.

▶ 7.2 MANUFACTURED IC TYPES

If designers are willing to wait months for a physical implementation of a digital circuit, and to spend millions of dollars for that physical implementation, then those designers might consider implementing the circuit using one of several IC types that require the manufacturing of a new IC.

Full-Custom Integrated Circuits

One IC type is known as a full-custom IC. A ***full-custom IC*** is a chip created specifically to implement the gates (actually, the transistors) of the desired circuit. The design flow for a full-custom IC is shown in Figure 7.3. Digital designers don't usually build full-custom ICs themselves, but rather they send the desired circuit to a group or company whose engineers specialize in transforming digital circuits into full-custom ICs.

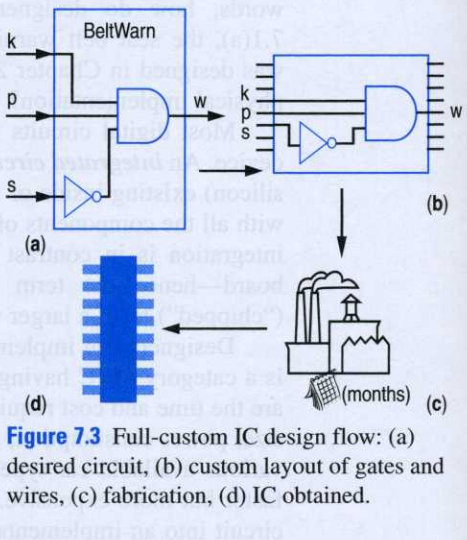
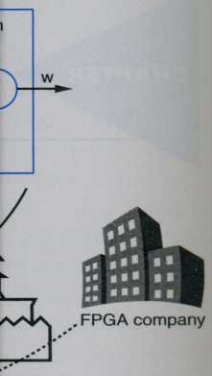
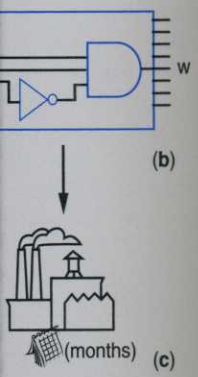


Figure 7.3 Full-custom IC design flow: (a) desired circuit, (b) custom layout of gates and wires, (c) fabrication, (d) IC obtained.

Those engineers, assisted by computer-aided design (CAD) tools, convert the desired digital circuit design into a circuit of transistors, and decide where to place each transistor on the surface of the chip, how to orient each transistor (e.g., left to right, right to left, top to bottom, etc.), how big to make each transistor, how to place the wires that connect the transistors, and so on. All that information about how the transistors and wires should be layed out on a chip's surface is known as a ***layout***. The full-custom IC engineers then send that layout information to a factory that specializes in



digital circuit, and
se designers might
quire the manufac-



esign flow: (a)
yout of gates and
obtained.

transistor, how to
tion about how the
n as a ***layout***. The
that specializes in

A 2006 Samsung fab in Texas cost $3.5 billion; fabs typically cost from $1 billion to over $10 billion.

fabricating ICs, known as a fabrication plant, or ***fab*** for short. Fabricating an IC is a complex error-prone process utilizing state-of-the-art photographic, laser, and chemical equipment that each can cost hundreds of millions of dollars. The transistors and wires are formed as dozens of ***layers*** on the surface of a chip; the lower layers define the transistors and connections within logic gates, while the upper layers define the wires that exist between gates. An IC may have tens of layers. Each layer requires a set of ***masks*** that allow light to reach specific regions of the chip to modify chemicals on the chip surface during the formation of layers. This cost for creating the masks required for IC fabrication is known as ***nonrecurring engineering (NRE) cost***. The cost is called nonrecurring because it occurs once, before chips are made, and then does not recur, no matter how many chips are subsequently manufactured. Because a full-custom IC type requires masks to be made for every layer of the chip, the NRE cost for a full-custom IC type is very high, typically tens of millions dollars. The NRE cost can be recouped by adding some amount to the selling price of each chip manufactured. Thus, if a particular IC will have 10,000,000 copies made and sold, and the NRE costs totalled $100,000,000, then $10 could be added to the selling price of each chip to recoup the NRE cost.

The time spent waiting for an IC to be manufactured also contributes to cost. In particular, the setup of the layout and masks takes months. During that time, the product for which designers are manufacturing the chip may be losing market share to a competing product already being sold while the designers wait for their chip to be fabricated, translating to lost revenue, which can be thought of as a cost too—"time is money."

The high cost of the full-custom IC type can be further increased due to what are known as respins. Fabricating an IC is referred to as a silicon ***spin***, named from the way that the silicon is melted and then spun before slicing the hardened silicon into wafers and then into chips. Because all the chip's layers are custom designed, the probability is high that engineers or tools will make a mistake somewhere in the transistors or wiring, such as placing two transistors too close together, resulting in interference. Therefore, after fabricating a full-custom IC, designers commonly find errors that necessitate refabricating the IC, known as a ***respin*** (pronounced as "ree-spin"). Respins may happen two or three times or more, each time incuring some additional NRE cost and requiring weeks or months per respin, possibly losing more market share and thus costing even more.

According to one survey, only about 10% of 2002 digital circuits were implemented as custom ICs.

Needless to say, full-custom IC fabrication is uncommon. Designers choose to implement a digital circuit on a full-custom IC when they absolutely require the IC type's small size, low power, or high performance, and when they can recoup the NRE costs due to high volumes, such as chips found inside calculators, wristwatches, mobile phones, or desktop PCs. Alternatively, designers may use a full-custom IC if cost is not tightly constrained but maximum performance is a high priority, as might be the case in military or space applications.

Semicustom (Application-Specific) Integrated Circuits—ASICs

Because physical implementation on full-custom ICs is so costly and time-consuming, semicustom IC types evolved during the 1980s and 1990s to reduce the costs and the time of fabricating a chip. These types are known as ***application-specific integrated circuits***, or ***ASICs***. An ASIC is somewhat customized, hence the term *semicustom*, and the customization is specific for a particular circuit application, hence the term *application-*

specific. The distinction is akin to that between full-custom and semicustom houses: The former involves custom-designing every aspect of a house and is thus expensive, time-consuming, and rare, while the latter involves adjusting a basic design (e.g., choosing between a larger family room or an additional bedroom) and is thus less costly, faster to build, and far more common. Two common ASIC types are *standard cell* and *gate array*.

Standard Cell ASIC

Standard cell is an ASIC type that reduces physical implementation NRE cost and manufacturing time compared to full-custom ICs. ***Standard cell*** ASICs, sometimes called ***cell-based ASICs***, use a collection (known as a ***library***) of previously layed-out gates or pieces of logic, called *cells*, that must be instantiated and connected with wires to implement a circuit.

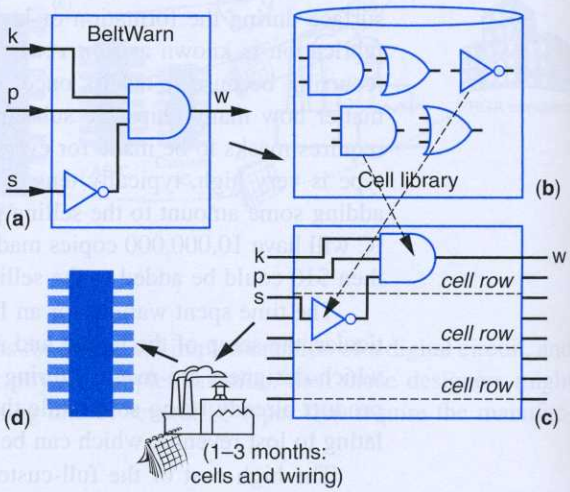


Figure 7.4 Standard cell ASIC design flow: (a) desired circuit, (b) cell library, (c) standard cell layout, (d) IC created by fabricating the cells and wires.

A cell might be a 2-input AND gate, a 2x1 mux, or a combination of gates like two 2-input AND gates connected to an OR gate connected to an inverter (called an AND-OR-INVERT, or AOI, cell). All cells are typically the same *standard* height (hence the term *standard cells*), meaning they occupy the same vertical space in Figure 7.4(c), so cells can be placed in standard-height rows on a chip. A standard cell ASIC company predesigns the layout for each cell, resulting in a collection or "library" of cell designs (the library exists in computer files). The seat belt warning light circuit in Figure 7.4(a) can be implemented as a standard cell ASIC by choosing appropriate cells from a cell library as in Figure 7.4(b), instantiating and wiring those cells as in Figure 7.4(c), and fabricating an IC as in Figure 7.4(d).

Designers themselves don't typically choose standard cells and map their circuits onto those cells. Rather, automated tools convert the desired digital circuits into standard cells and output results in data files that are processed by fabs to control the fabrication process.

A typical cell library might contain hundreds or thousands of cells; the cell library shown in Figure 7.4(b), having just five cells, is *trivially small and for illustrative purposes only*. A typical standard cell ASIC may have millions of cells, not just a few as in the figure.

Compared to the full-custom IC type, the standard cell IC type is less optimized, because the cells are restricted in their size and variety, and their placement is restricted to predetermined standard-height rows. But standard cell ASICs require less NRE cost and time, because no transistor-level design or layout is necessary, as those were done

beforehand by the standard cell company. Furthermore, respins, though still occuring, are less frequent than with full-custom ICs.

The task of converting a desired digital circuit into a circuit using only components from a particular library (e.g., a particular standard cell library) is known as ***technology mapping***. The task of determining where to place those components on a chip is known as ***placement***, and the task of connecting those components by wires is known as ***routing***. All three tasks are collectively known as ***physical design***, and are typically carried out today by automated tools.

Example 7.1 Implementing a half-adder using standard cells

Using the standard cell library in Figure 7.4, this example implements a half-adder on a standard cell ASIC. Recall that the equations for a half-adder are: `co = ab`, and `s = a'b + ab'`. Thus, the half-adder can be implemented using two inverter cells, three 2-input AND cells, and one 2-input OR cell from the library.

The half-adder can be implemented using cells as shown in Figure 7.5, assuming each cell row can hold at most three cells.

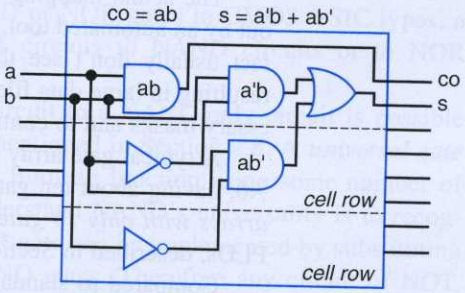


Figure 7.5 Half-adder on standard cells.

Gate Arrays (Structured ASICs)

A standard cell ASIC, while having lower NRE and involving less time than full-custom ICs, must still have all its layers fabricated. A ***gate array*** ASIC involves a chip whose transistors are *predesigned* to form rows ("arrays") of logic gates on the chip, as shown in Figure 7.6(b), meaning that only the wires remain to be fabricated. Creating the wires represents just the last steps of fabrication, and thus gate array technology eliminates much of the time and cost of fabricating a chip for a particular circuit. A gate array company predesigns and mass-produces the gate array chip. When a client wants a circuit on an IC, the company then customizes some of those

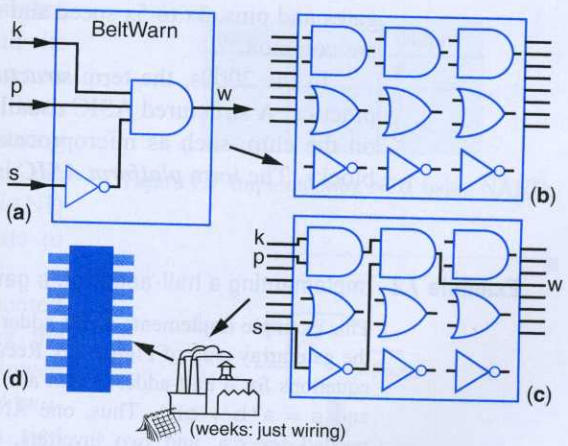


Figure 7.6 Gate array ASIC design flow: (a) desired circuit, (b) gate array before wires are added, (c) gate array after wires are added, thus implementing the desired circuit, (d) IC created by fabricating the wires. Note: real gate array ICs have millions of gates.

chips for the client's circuit by fabricating the metal layers. Figure 7.6 illustrates how the seat belt warning light desired circuit in Figure 7.6(a) might be implemented using the gate array chip in Figure 7.6(b) consisting of a row of 2-input AND gates, a row of 2-input OR gates, and a row of inverters. Figure 7.6(c) shows how to map the desired circuit's 3-input AND gate using two of the gate array's available 2-input AND gates, and how to map the desired inverter to one of the gate array inverters. The figure also shows how the desired wiring among the gate array's pins, the gate array AND gate, and the gate array inverter, might be implemented. The remaining gates and pins on the gate array chip would be unutilized. Fabricating those wires would result in the IC being customized to the seat belt application (Figure 7.6(d)).

The actual mapping of the desired circuit to a gate array would typically be carried out by an automated tool. Designers almost never carry out that mapping manually, and in fact usually don't see that mapping in any form—the mapping is all done by tools, resulting in large data files that can be processed by other tools at a fab to create the necessary masks and to control the fabrication process.

A typical gate array chip may hold *millions of gates*; the gate array shown in Figure 7.6, having about ten gates, is trivially small and is for illustration purposes only—*gate arrays with only 10 gates do not exist.* For designs with only a few gates, logic ICs or PLDs, described in Section 7.4, might be used instead.

Compared to standard cell ASICs, gate array ASICs involve less NRE and require less time. However, gate array ASICs are less optimized. Comparing Figure 7.4(c) and Figure 7.6(c) illustrates that standard cell ASIC flexibility of choosing and placing cells results in a more compact design, with fewer gates and fewer wires, than gate array ASICs. Notice that the gate array implementation in Figure 7.6 utilizes an extra level of logic than the original circuit, has longer wires, and has wasted area due to unused gates and pins. 3x to 5x speed and size differences between standard cells and gate arrays are common.

In the 2000s, the term ***structured ASIC*** began to replace the term "gate array" in practice. A structured ASIC usually supplements arrays of gates with other components on the chip, such as microprocessors, RAMs, and other higher-level system building blocks. The term ***platform ASIC*** is also sometimes used.

Example 7.2 Implementing a half-adder on a gate array

This example implements a half-adder circuit on the gate array chip of Figure 7.6. Recall that the equations for a half-adder circuit are: `co = ab`, and `s = a'b + ab'`. Thus, one AND gate is needed for `co`, and two inverters, two AND gates, and one OR gate for `s`. The gate array chip in Figure 7.6 has three AND gates, three OR gates, and three inverters, so the chip has enough gates to implement the desired circuit.

The implementation of the half-adder circuit on the gate array chip as shown in Figure 7.7.

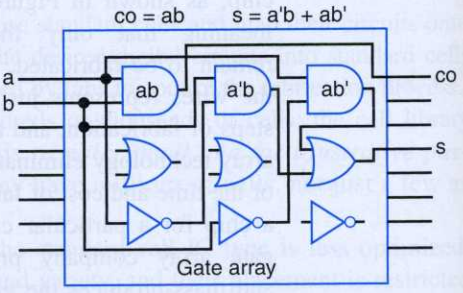


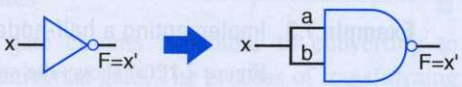
Figure 7.7 Half-adder on a gate array ASIC.

Implementing Circuits Using Only NAND Gates

Recall from Chapter 2 that CMOS transistors more efficiently implement NAND and NOR gates rather than AND and OR gates. The underlying reason was that pMOS transistors conduct `1`s well but not `0`s, while nMOS transistors conduct `0`s well but not `1`s. Therefore, gate arrays typically contain NAND or NOR gates rather than AND and OR gates, and standard cell circuits are more efficient if implemented using NAND or NOR cells rather than AND and OR cells. Furthermore, creating a gate array is easier using just one type of gate, like just NANDs, or just NORs, rather than having to decide how many AND gates, OR gates, and NOT gates to pre-instantiate in the arrays. Thus, given the preference for NAND or NOR gates in CMOS ASIC types, a method is needed for converting AND/OR circuits to NAND circuits or to NOR circuits.

Fortunately, converting any AND/OR circuit to a NAND-only circuit is possible because NAND is a universal gate, as was mentioned in Section 2.8. A ***universal gate*** is a logic gate type that can implement any Boolean function using some number of gates of that one type only. One way to understand NAND's universality is to recognize that a NOT gate, an AND gate, or an OR gate can be implemented by substituting each gate with an equivalent circuit of NAND gates. Therefore any circuit of NOT, AND, or OR gates can be implemented using just NAND gates.

To implement a NOT gate using NAND gates, we can substitute the NOT gate with a two-input NAND gate having its two inputs tied together, as shown in Figure 7.8. The truth table in the figure shows that the NAND gate with its inputs tied together acts the same as an inverter. When the input `x` is `0`, both inputs of the NAND gate are `0`, causing the NAND gate to output `1`. When the input `x` is `1`, both inputs of the NAND gate are `1`, causing the NAND gate to output `0`.



Inputs			Output
x	a	b	F
0	0	0	1
1	1	1	0

Figure 7.8 Implementing NOT using NAND.

An AND gate can be implemented using NAND gates by substituting the AND gate with a NAND gate followed by a NOT gate (implemented as a two-input NAND gate with its inputs tied together), as in Figure 7.9. This works because given inputs `a` and `b`, the first NAND computes `(ab)'`, and then the NOT gate computes `(ab)'' = ab`, which is AND.

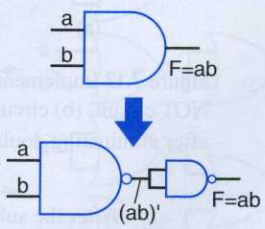


Figure 7.9 Implementing AND using NANDs.

An OR gate can be implemented using NAND gates by substituting the OR gate with a NAND gate having each input inverted (by 2-input NAND gates with inputs tied together), as in Figure 7.10. This works because given inputs `a` and `b`, the circuit of NAND gates in the figure computes `(a'b')'`. By DeMorgan's Law, that expression equals `a'' + b''`, which simplifies to `a + b`, which is OR.

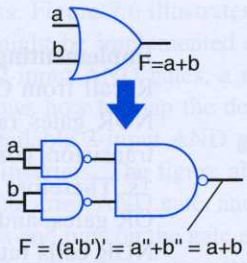


Figure 7.10 Implementing OR using NANDs.

When replacing a circuit originally having AND/OR/NOT gates by a circuit having only NAND gates using the above substitutions, some signals may get double-inverted—a signal feeds into an inverter and then immediately into another inverter. Double-inverting a signal yields the original signal, so double inversions can be replaced by a wire, as in Figure 7.11. Such elimination reduces the number of transistors and reduces a circuit's delay, without changing the circuit's function.

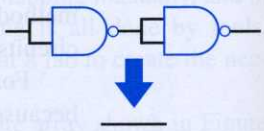


Figure 7.11 Double inversion becomes a wire.

Example 7.3 Implementing a half-adder's sum circuit using only NAND gates

Figure 7.12(a) shows the sum circuit for a half-adder (see Section 4.3), using AND, OR, and NOT gates. We can implement that circuit using only NAND gates by substituting each gate with an equivalent NAND circuit, as shown in Figure 7.12(b).

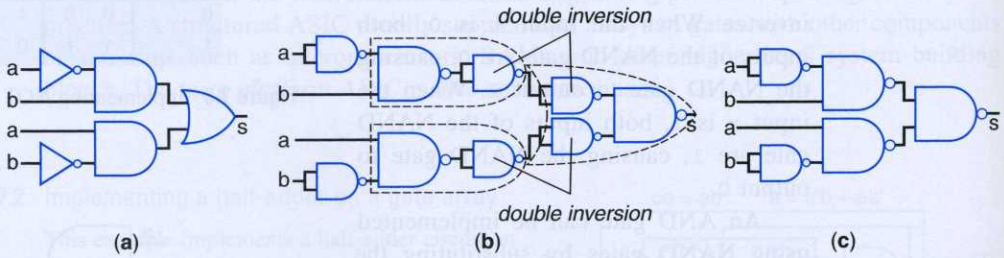


Figure 7.12 Implementing a half-adder's sum circuit using NAND gates only: (a) original AND/OR/NOT circuit, (b) circuit obtained after substituting equivalent NAND circuits for each gate, (c) circuit after eliminating double inversions.

After the substitutions, note that there are two signals that are double-inverted. Eliminating the double inversions results in the circuit shown in Figure 7.12(c).

When converting AND/OR/NOT circuits to NAND circuits by hand, some people find it easier to simply draw inversion bubbles rather than the NAND-based inverters, as shown in Figure 7.13. Then, double inversion bubbles on a signal cancel each other. Any remaining isolated inversion bubbles become a NAND-based NOT gate. Thus, the circuit in Figure 7.13, which uses inversion bubbles, would end up identical to the circuit in Figure 7.12(c).

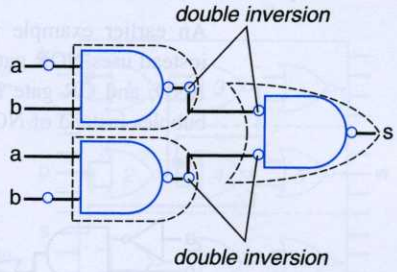


Figure 7.13 Drawing inverters as inversion bubbles during conversion to NAND.

If NAND gates with only a specific number of inputs are available, such as NAND gates with 2 inputs being the only gates available, then we can first modify the AND/OR circuit to use only 2-input AND/OR gates. Such conversion is done by composing larger AND and OR gates from smaller AND and OR gates, respectively, as discussed in Section 5.10. After the conversion to smaller gates, the conversion to NAND gates can proceed as described earlier.

Implementing Circuits Using Only NOR Gates

Converting AND/OR/NOT circuits to NOR gate circuits is similar to converting to NAND circuits, because a NOR gate is also a universal gate. The process of transforming a circuit into NOR gates replaces each AND, OR, and NOT gate with the equivalent NOR-based circuits shown in Figure 7.14. A NOT gate can be replaced with a two-input NOR gate with the inputs tied together. An OR gate is replaced with a NOR gate followed by an inverter, yielding `(a+b)'' = a+b`. An AND gate is replaced with a NOR gate having inverted inputs, yielding `(a'+b')' = a''*b'' = ab` (notice the use of DeMorgan's Law).

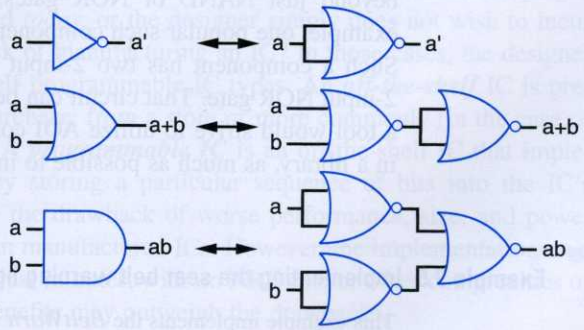


Figure 7.14 NOR gate equivalencies.

Example 7.4 Implementing a half-adder's sum circuit using only NOR gates

An earlier example implemented a half-adder's sum output using NAND gates; this example instead uses NOR gates. The half-adder's sum circuit is shown again in Figure 7.15(a). Each NOT, AND, and OR gate is replaced by its equivalent NOR circuit in Figure 7.15(b), using inversion bubbles instead of NOR-based NOT gates for convenience.

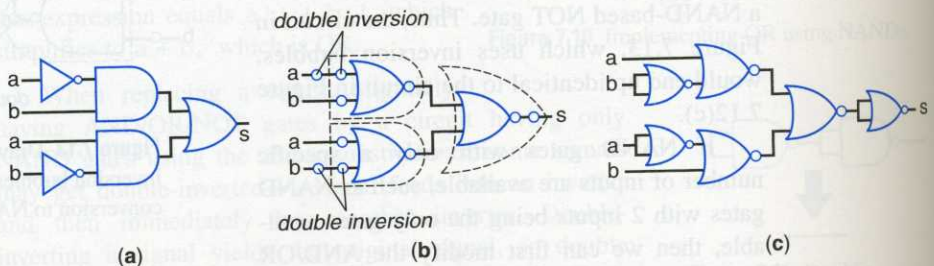


Figure 7.15 Implementing an AND/OR/NOT circuit using NORs only: (a) original circuit, (b) circuit obtained by substituting AND/OR/NOT gates with equivalent NOR circuits, using inversion bubbles for ease of drawing, (c) final circuit after eliminating double inversions and replacing standalone inversion bubbles with NOR-based NOT gates.

We eliminate double inversions, and replace stand-alone inversion bubbles by NOR-based NOT gates, as shown in Figure 7.15(c)

The half-adder's sum circuit was implemented with fewer NAND gates than NOR gates. Depending on the original circuit, the reverse could be true. NAND gates are well-suited to circuits in the sum-of-products form. NOR gates are well-suited to circuits in product-of-sums form (a level of OR gates feeding into a single AND gate). ▶ 7.3

Gate array and standard cell libraries typically include additional components, beyond just NAND or NOR gates, that have efficient CMOS implementations. For example, one popular such component is known as AND-OR-INVERT, or ***AOI*** for short. Such a component has two 2-input AND gates (thus four inputs total), feeding into a 2-input NOR gate. That circuit can be efficiently designed using CMOS transistors. Thus, a tool would strive to utilize AOI components, and other compact available components in a library, as much as possible to improve implementation performance and size.

Example 7.5 Implementing the seat belt warning light on a NOR-based gate array

This example implements the *BeltWarn* circuit of Figure 7.1(a) using the NOR-based gate array of Figure 7.16(a). Noticing that the gate array has only 2-input NOR gates, we first convert the *BeltWarn* circuit to use AND/OR gates with 2 inputs only, as shown in Figure 7.16(b). We then convert the AND/OR circuit to the NOR-only circuit in Figure 7.16(c), using the equivalencies in Figure

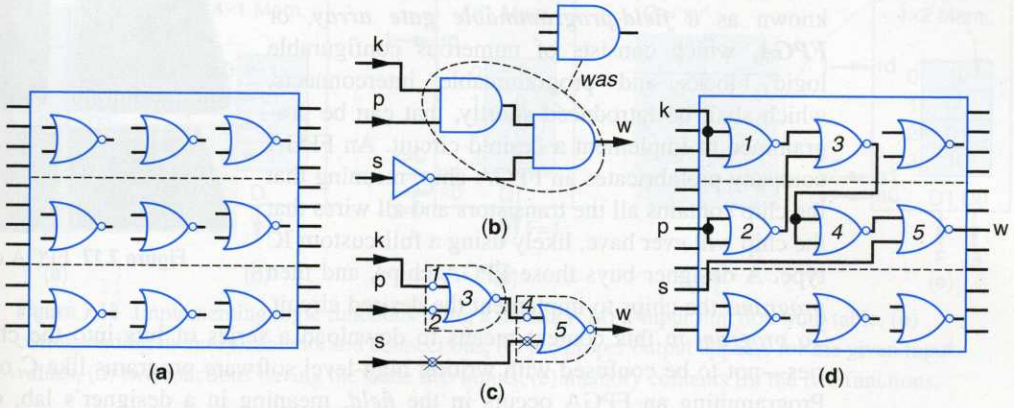


Figure 7.16 Implementing the *BeltWarn* circuit on a NOR-based gate array IC: (a) original gate array, (b)–(c) converting the desired circuit to two-input NOR gates only, (d) final gate array with wires.

7.14, and using inversion bubbles rather than NOR-based inverters. A double inversion exists on the wire from input s, so we eliminate those two inversions. Note that we do not eliminate the double inversion between points 3 and 4 in Figure 7.16(c), because the first inversion is part of a NOR gate—eliminating that first inversion would convert the NOR gate to an OR, defeating our goal of having NOR gates only.

After converting remaining standalone inversions to NOR-based inverters, we map the circuit to the gate array's 2-input NOR gates as in Figure 7.16(d)—we numbered the NOR gates of Figure 7.16(c) and (d) to show the correspondence between the two circuits.

7.3 OFF-THE-SHELF PROGRAMMABLE IC TYPE—FPGA

Manufactured IC types require at least a few weeks, and often several months, to convert a desired digital circuit design into a physical IC. What if a designer is developing a circuit that should be implemented *today*, or the designer simply does not wish to incur the NRE cost, complexity, and risk of manufacturing an IC? In those cases, the designer can use one of several off-the-shelf programmable IC types. An ***off-the-shelf*** IC is pre-manufactured and available for purchase, from a store or more commonly (in the case of ICs) by shipment from a vendor. A ***programmable IC*** is an off-the-shelf IC that implements a desired circuit simply by storing a particular sequence of bits into the IC's memory. Programmable ICs have the drawback of worse performance, size, and power compared to custom or semicustom manufactured ICs. However, the implementation on a programmable IC can be ready in just minutes, with no NRE cost or other complexities of manufacturing an IC, and these benefits may outweigh the drawbacks.

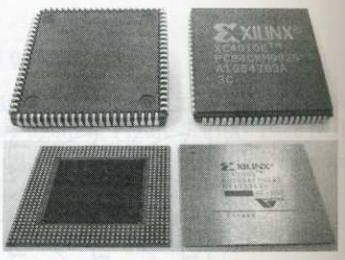


Figure 7.17 FPGA chips.

The most popular type of programmable IC is known as a ***field-programmable gate array***, or ***FPGA,*** which consists of numerous configurable logic blocks and programmable interconnects, which shall be introduced shortly, that can be programmed to implement a desired circuit. An FPGA company prefabricates an FPGA chip, meaning that the chip contains all the transistors and all wires that the chip will ever have, likely using a full-custom IC type. A designer buys those FPGA chips, and then *programs* the chips to implement the desired circuit. To *program* in this context means to download a series of bits into the chip's memories—not to be confused with writing high-level software programs like C or C++ code. Programming an FPGA occurs in the *field*, meaning in a designer's lab, or office, or home, as opposed to in a fabrication plant. Hence the words "field-programmable" in the name. Furthermore, programming typically takes only seconds, or perhaps minutes at most. Figure 7.17 shows some FPGA chips. The chip at the top, with its front and back shown, measures about three quarters of an inch on each side. The chip on the bottom is about 1 inch on each side.

Field-programmable gate arrays (FPGAs) have no "gate arrays" inside them—the name is there due to historical reasons.

The term "gate array" is there in the name because, when FPGAs first became popular in the mid-1980s, they were marketed as an alternative to the gate array IC type, which was very popular at that time. Thus, an FPGA was a semicustom IC (nearly synomous with "gate array" at that time) that could be programmed in the field instead of at a fabrication plant. However, be forewarned that the internal design of an FPGA chip looks nothing like a gate array—the naming is somewhat unfortunate.

The two basic types of components inside an FPGA are lookup tables and switch matrices. Those components are replicated thousands of times in regular patterns inside an FPGA. We now describe each type of component.

Lookup Tables

A basic idea underlying FPGAs is that *a memory can implement combinational logic.* More specifically, a 1-bit-wide memory with N address lines, and hence 2^N words, can implement any Boolean combinational function of N variables.

Recall that a memory configured to be read will output the contents of the word corresponding to the present address at the memory's address lines. So if a 4x1 memory's address lines `a1a0` are `00`, the memory will output the contents of word 0. If the address lines are `01`, the memory outputs the contents of word 1. Likewise, `10` outputs word 2, and `11` outputs word 3.

The key idea underlying FPGAs is that a memory with N address lines can implement any combinational function with N inputs.

Implementing a Boolean function with a memory can therefore be done simply by connecting the function's inputs to the memory address lines, and storing a `0` or `1` in each memory word to match the desired function output value for each combination of input values. For example, consider the function `F(x,y) = x'y' + xy`. The truth table for the function is shown in Figure 7.18(a). To implement the example function, we can connect `x` and `y` to a 4x1 memory's address lines `a1` and `a0`, respectively, and based on the truth table, we store a `1` in word 0, a `0` in word 1, a `0` in word 2, and a `1` in word 3—specifically, we

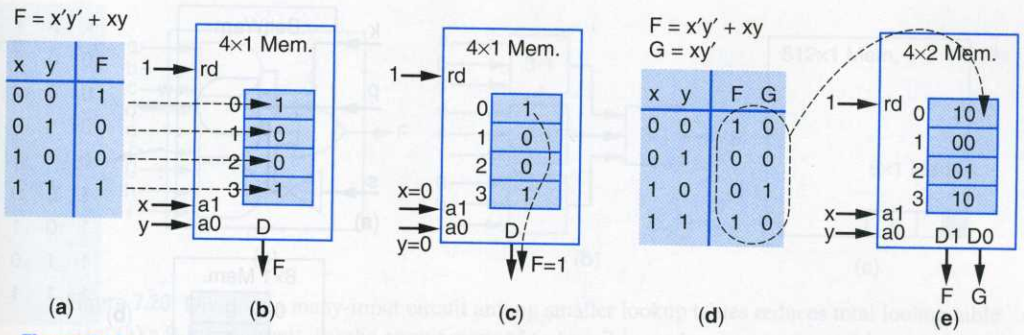


Figure 7.18 Implementing logic functions using a memory: (a) 2-input function truth table, (b) corresponding memory contents and connections, (c) the proper output appears for the given input values, (d) two functions having the same two inputs, (e) memory contents for the two functions.

store the truth table output values in the memory. The memory then implements the desired function, as shown in Figure 7.18(b). For example, when `xy=00`, we want the output to be `1`. Figure 7.18(c) shows that when `xy=00`, the memory's address lines will be `00`, and thus the memory will output the contents of word 0, which is the value `1`, as desired.

A memory with M bits per word, rather than just 1 bit per word, can implement M functions that have the same inputs. For example, consider the two functions `F(x,y) = x'y' + xy` and `G(x,y) = xy'`. The truth table for these two functions is shown in Figure 7.17(d). A 4x2 memory, which has 2 bits per word, can implement those two functions, as shown in Figure 7.18(e).

A memory used to implement a combinational circuit is known in FPGA terminology as a ***lookup table***, or LUT. When used as a lookup table, a memory is typically referred to by the number of *inputs* (address lines) and the number of outputs (bits per word), rather than by the number of *words* and the number of outputs. For example, an 8x2 memory being used as a lookup table is referred to as a "3-input 2-output lookup table," rather than as an 8x2 lookup table. An FPGA typically consists of large numbers of same-sized LUTs.

From this point forward, we will assume the memory is configured for read, and thus we won't show the read line set to `1`.

Example 7.6 Implementing the seat belt warning light with a lookup table

This example uses a lookup table to implement the seat belt warning light circuit from Figure 7.1, whose circuit appears in Figure 7.19(a) and whose equation is

```
w = kps'
```

We first generate the truth table for the function, which is shown in Figure 7.19(b). Because the circuit has three inputs, the circuit will require a 3-input 1-output lookup table (memory).

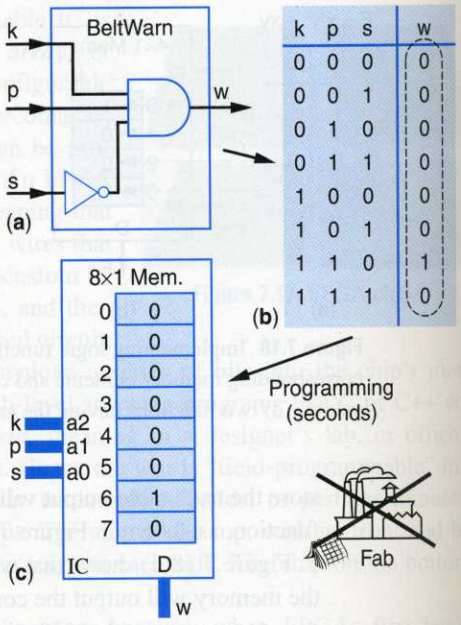


Figure 7.19 FPGA design flow: (a) desired circuit, (b) circuit's truth table, (c) implementing the truth table in an FPGA lookup table.

We connect the inputs to the memory's address lines, and store the truth table in the memory, as in Figure 7.19(c), thus implementing the desired function.

If the 3-input 1-output memory is an IC, then the implementation is complete, and we can insert the IC into the electronic system with which the IC should interact.

You've just seen an example of a very simple programmable IC—a memory. A memory chip with N address lines and hence 2^N words, and with M bits per word, can implement M different Boolean functions of the same N inputs. A designer can purchase a memory chip before it is needed in a design, and then the designer can "program" the memory chip to implement a desired Boolean function.

Mapping a Circuit among Multiple Lookup Tables

Unfortunately, using a memory to implement a Boolean function is inefficient for functions with numerous inputs. For example, while a 4-input function would need only a 16-word memory, a 12-input function would require a 4-Kword memory; a 32-input function would require a 4-billion-word memory. The needed memory size grows the same as the size of the function's truth table, which grows as 2^N, where N is the number of function inputs. In short, a truth table is *not* an efficient Boolean function representation for functions with numerous inputs, and thus a lookup table is not an efficient implementation for functions with numerous inputs. For example, the function `F = abc + def + ghi`, shown in Figure 7.20(a), has 9 inputs. Implementing the

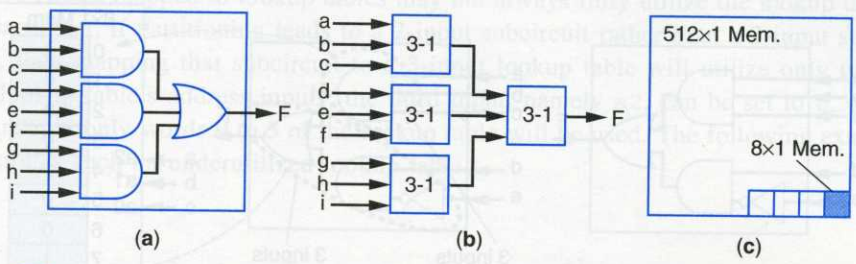


Figure 7.20 Dividing a many-input circuit among smaller lookup tables reduces total lookup table size: (a) a 9-input circuit, (b) the circuit mapped to four 3-input 1-output lookup tables, (c) the four 3-input 1-output lookup tables are much smaller than a 9-input 1-output lookup table.

function on a single lookup table would require a table with $2^9 = 512$ words. However, one can partition the circuit into subcircuits such that each subcircuit has 3 inputs and 1 output—the first subcircuit computes `abc`, the second `def`, the third `ghi`, and the fourth ORs the outputs of the first three subcircuits to generate the output `F`. Each subcircuit could be implemented using a 3-input 1-output lookup table (i.e., an 8x1 memory). The resulting implementation would have four 3-input 1-output lookup tables, as shown in Figure 7.20(b). The total words for that four-LUT implementation would be a mere 8 + 8 + 8 + 8 = 32 words—far less than the 512 words required for a single 9-input lookup table. Figure 7.20(c) shows the relative sizes of one 512-word memory and four 8-word memories; the memory sizes are drawn to scale. Notice the reduction in size obtained by using multiple small lookup tables.

As a result, FPGAs typically contain large numbers of small lookup tables, rather than a small number of large lookup tables. Researchers have conducted numerous studies on thousands of typical circuits, and found that lookup tables with 3 to 6 inputs seem to be most efficient for most circuits. An FPGA typically has one size of LUT that is replicated thousands of times.

Therefore, a circuit being mapped to an FPGA must be partitioned into subcircuits such that each subcircuit can be mapped to one of the small lookup tables in an FPGA. Such partitioning is handled by tools and forms part of the *technology mapping* task for FPGAs; we'll consider such partitioning here to gain insight into the behavior of such tools. If an FPGA uses 3-input 1-output lookup tables, then the circuit must be partitioned into subcircuits each having 3 inputs (or less) and 1 output. For example, consider the circuit shown in Figure 7.21(a). That circuit can't be mapped to a 3-input 1-output lookup table because the circuit has 4 inputs. The circuit must therefore be partitioned into two sub-circuits, as shown by the dashed circles in Figure 7.21(b). The first subcircuit's output, labeled as `t`, computes `t = abc`. The second subcircuit's output computes `F = t + de'`. The circuits are mapped to two lookup tables as shown in Figure 7.21(c). The first lookup table is programmed to implement the first subcircuit, namely `t = abc`. Likewise, the second lookup table is programmed to implement the second subcircuit, namely `F = t + de'`. The two lookup tables with the shown connections thus implement the desired circuit.

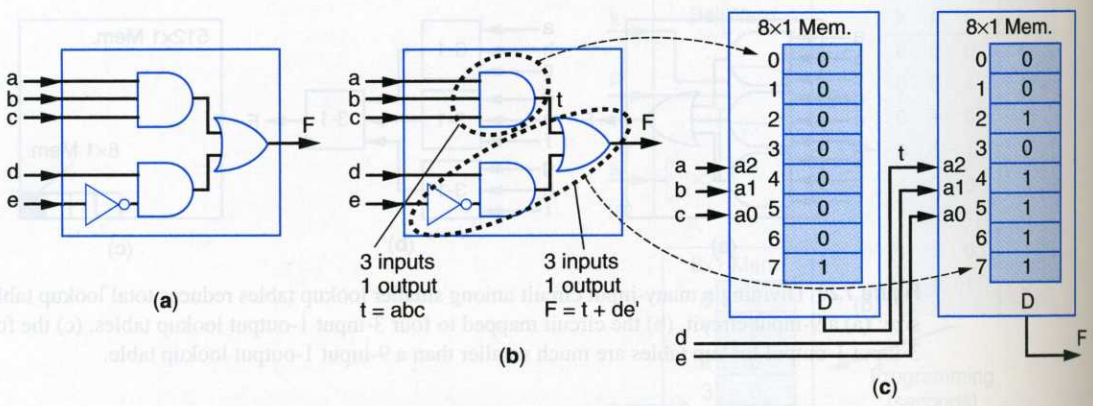


Figure 7.21 Partitioning a circuit onto two lookup tables: (a) desired circuit, (b) circuit partitioned into subcircuits with at most 3 inputs and 1 output, (c) subcircuits mapped to two 3-input 1-output lookup tables.

Example 7.7 Mapping a circuit to 3-input 1-output lookup tables

This example maps the circuit shown in Figure 7.22(a) onto a minimum number of 3-input 1-output lookup tables. The first step is to partition the circuit into subcircuits having three (or fewer) inputs and one output. Such a partitioning is shown in Figure 7.22(b). Subcircuit 1 has three inputs `a`, `b`, and `c`, and one output labeled `t`. Subcircuit 2 has three inputs `c`, `e`, and `f`, and one output labeled `u`. Subcircuit 3 has three inputs `t`, `d`, and `u`, and one output `Y`.

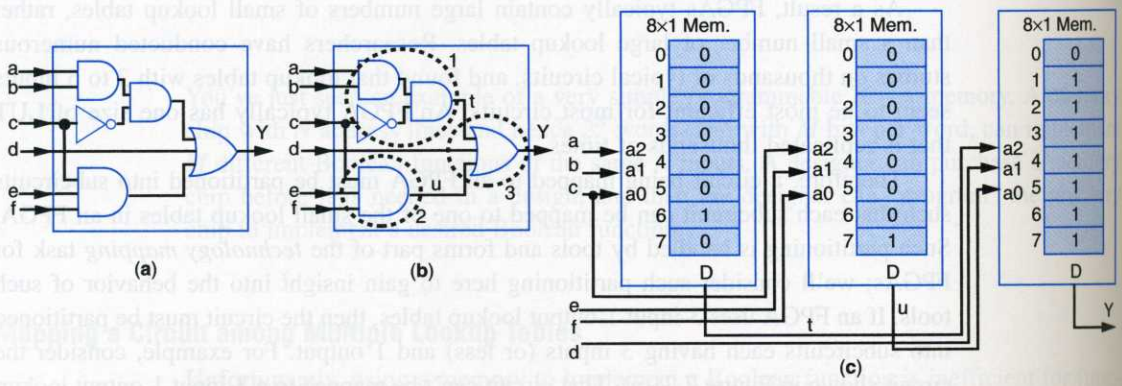


Figure 7.22 Mapping a circuit onto a minimum number of 3-input 1-output lookup tables: (a) desired circuit, (b) circuit partitioned into subcircuits with at most 3 inputs and 1 output, (c) subcircuits mapped to three 3-input 1-output lookup tables.

Figure 7.22(c) shows three lookup tables that implement those three subcircuits. The first lookup table implements `t = ab * c'` (the `1` in word 6 corresponds to `abc'`). The second lookup table implements `u = cef`. The third lookup table implements `Y = t+d+u`, thus completing the circuit implementation.

A circuit mapped to lookup tables may not always fully utilize the lookup tables. In particular, if partitioning leads to a 2-input subcircuit rather than a 3-input subcircuit, then mapping that subcircuit to a 3-input lookup table will utilize only two of the lookup table's address inputs; the third input, namely `a2`, can be set to `0`, which means that only words 0 to 3 of the lookup table will be used. The following example illustrates such an underutilized lookup table.

Example 7.8 Mapping that results in an underutilized lookup table—Extended seat belt warning light

This example maps the circuit of Figure 7.23(a), which is the extended seat belt example from Example 2.8, onto a minimum number of 3-input 1-output lookup tables. Because the circuit has four inputs, mapping the circuit to 3-input 1-output lookup tables requires first partitioning the circuit into 3-input subcircuits; such a partitioning is shown in Figure 7.23(b). While the first subcircuit has 3 inputs, the second subcircuit has only 2 inputs.

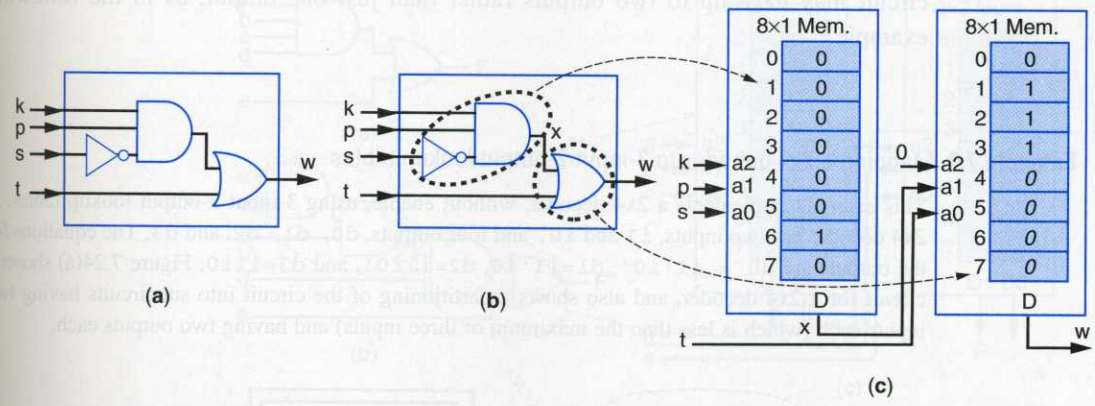


Figure 7.23 Mapping a circuit onto lookup tables sometimes yields underutilized lookup tables: (a) desired circuit, (b) circuit partitioned into two subcircuits, one subcircuit having only two inputs, (c) subcircuits mapped to two 3-input 1-output lookup tables; the second lookup table has its `a2` address line set to `0`, so only the first four words are used.

Figure 7.23(c) shows the mapping of those subcircuits to lookup tables. The first lookup table implements the first subcircuit, `x = kps'`. The second lookup table implements the second subcircuit, `w = x + t`. Because the second subcircuit has only two inputs, we set address line `a2` of the lookup table to `0`, and connect the two inputs `x` and `t` to the lower two address lines. Because `a2` is always `0`, words 4 to 7 of the lookup table will never be accessed. We've programmed those words as `0`s, but they are shown in italics to indicate that those words will *never be accessed*. The values of `x` and `t` will cause either word 0, 1, 2, or 3 to be read. Those words have been programmed to implement the OR function to achieve `x + t`.

Sometimes a circuit has a gate with four or more inputs. Clearly, such a circuit cannot be directly partitioned into subcircuits with three or fewer inputs. The solution is to replace the 4-input gate by an equivalent set of gates having fewer inputs. Technology mapping tools commonly first modify a circuit into a functionality equivalent circuit by decomoposing gates having three or more inputs into an equivalent set of 2-input gates, before trying to partition the circuit into subcircuits. Example 7.9 will perform some initial decomposition before partitioning.

Recall that a memory with M bits per word can implement M functions of the same inputs. Researchers have investigated how many bits per word a lookup table should have to accomodate most circuits efficiently, and have found that two bits per word is efficient for many circuits. The remainder of this chapter will thus use 3-input 2-output lookup tables.

Mapping a circuit onto 3-input 2-output lookup tables is similar to mapping to 3-input 1-output lookup tables, except that when partitioning into subcircuits, each subcircuit may have up to two outputs rather than just one output, as in the following example.

Example 7.9 Mapping a 2x4 decoder to 3-input 2-output lookup tables

This example implements a 2x4 decoder, without enable, using 3-input 2-output lookup tables. A 2x4 decoder has two inputs, `i1` and `i0`, and four outputs, `d0`, `d1`, `d2`, and `d3`. The equations for the outputs are `d0 = i1'i0'`, `d1=i1'i0`, `d2=i1i0'`, and `d3=i1i0`. Figure 7.24(a) shows a circuit for a 2x4 decoder, and also shows a partitioning of the circuit into subcircuits having two inputs each (which is less than the maximum of three inputs) and having two outputs each.

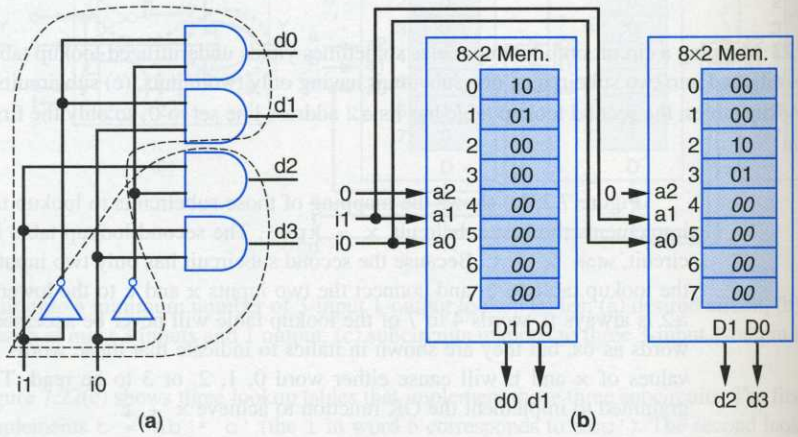


Figure 7.24 Mapping a 2x4 decoder to two 3-input 2-output lookup tables: (a) desired circuit, (b) mapping to two lookup tables. Italicized bits are unused.

Because the two subcircuits in Figure 7.24(a) have only two inputs each, the lookup tables implement those subcircuits using the top halves of the lookup tables' words; the bottom halves are unused, as shown in Figure 7.24(b). For the used words, both bits of each word are used.

Just as earlier examples showed that a subcircuit with fewer than three inputs results in unused lookup table words, likewise a subcircuit with fewer than two outputs will result in an unused lookup table column (meaning an unused LUT output), as in the following example.

Example 7.10 Mapping problem that decomposes a large gate, and that has unused LUT outputs

This example implements the circuit shown in Figure 7.25(a) using two 3-input 2-output lookup tables. The first step is to try to partition the circuit into subcircuits such that each group has at most 3 inputs and 2 outputs. However, the 4-input AND gate prevents such partitioning, because whatever subcircuit that gate is in will have at least four inputs. That problem can be remedied by first decomposing that gate into two smaller gates, while maintaining the same functionality, as shown in Figure 7.25(b). The circuit can then be partitioned into two subcircuits, each with 3 inputs and 1 output, as shown in the figure.

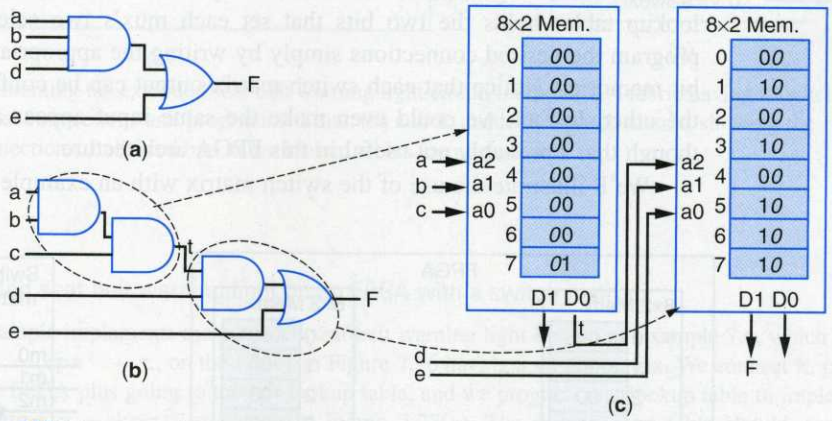


Figure 7.25 Mapping a circuit onto 3-input 2-output lookup tables: (a) original circuit, (b) transformed circuit that decomposes the 4-input AND gate into two-input gates, with the partitioning into 3-input 1-output subcircuits shown, (c) mapping of each subcircuit to a lookup table, with each subcircuit's function converted to programmed bits in the lookup table. Italicized bits are unused.

The subcircuits can then be mapped onto two 3-input 2-output lookup tables as shown in Figure 7.25(c). Notice that the first lookup table's `D1` output is unused, and the second lookup table's `D0` output is also unused; those columns have been programmed with `0`s, and are shown in italics to indicate that those bits will never be accessed. The first table's `D0` column implements `t = abc`. The second table's D1 column implements `F = td + e`.

An FPGA may have hundreds or thousands of lookup tables, and thus can implement large amounts of combinational logic.

Programmable Interconnects (Switch Matrices)

Earlier examples used custom connections between lookup tables, but the point of FPGAs is that the entire chip is prefabricated, even the wires. FPGAs thus come with ***programmable interconnects***, often called ***switch matrices***, which can be *programmed* to create the connections among lookup tables. Figure 7.26(a) shows a simple FPGA chip with five inputs (*P0–P4*), two 3-input 2-output LUTs, one 4-input 3-output switch matrix, and two outputs (*Q0, Q1*). All three of the left LUT's inputs come from the external inputs *P0*, *P1*, and *P2*—that LUT's inputs can't be changed. However, the right LUT's inputs may come from either the left LUT's outputs, or from the external inputs *P3* and *P4*. The switch matrix determines which of those connections will be made.

The switch matrix's internal design appears in Figure 7.26(b). It consists of three 4x1 multiplexers (muxes)—the bottom multiplexor is omitted from the drawing to save space. The top mux connects the switch matrix output *o0* to one of the matrix's four inputs. The second mux connects the output *o1* to one of the matrix's four inputs. The bottom mux (not drawn) connects *o2* to one of the matrix's four inputs. A two-bit memory (which is actually a 2-bit register, but called a memory for consistency with the memory inside a lookup table) holds the two bits that set each mux's two select lines. Thus, we can program the desired connections simply by writing the appropriate bits into those two 2-bit memories. Notice that each switch matrix output can be configured independently of the other. In fact, we could even make the same input appear at two or three outputs, though that's probably not useful in this FPGA architecture.

We'll illustrate the use of the switch matrix with an example.

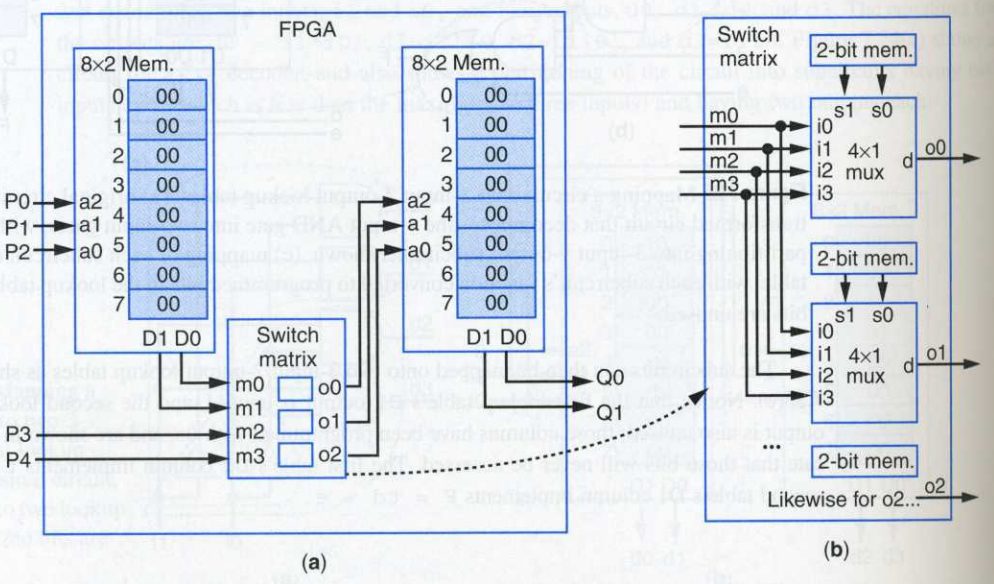


Figure 7.26 A simple FPGA architecture: (a) an FPGA that includes a switch matrix, and (b) the switch matrix's internals showing two 4x1 muxes controlled by two 2-bit registers. Note: real FPGAs have hundreds of lookup tables and switch matrices, not just a few.

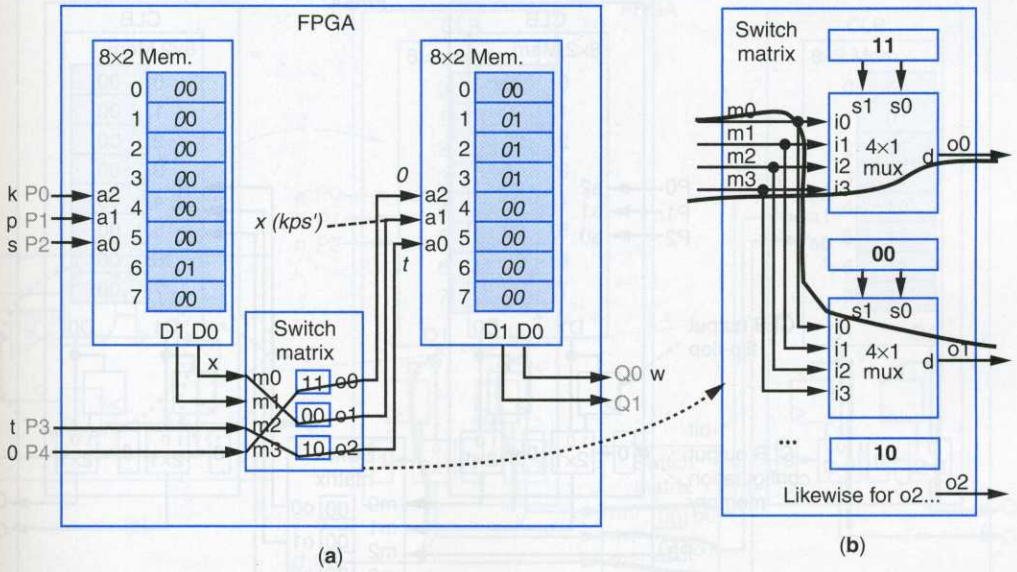


Figure 7.27 Implementing the extended seat belt warning light circuit on the FPGA fabric having a switch matrix: (a) external connections and programmed bits, (b) a look inside the switch matrix, showing the programmed connections. Italicized bits in the lookup tables are unused.

Example 7.11 Extended seat belt warning light on an FPGA with a switch matrix

This example implements the extended seat belt warning light system of Example 7.6, which computes `w = kps' + t`, on the FPGA in Figure 7.26 having a switch matrix. We connect `k`, `p`, and `s` to the FPGA pins going to the left lookup table, and we program that lookup table to implement the function `x = kps'`, as shown in Figure 7.27(a). The right lookup table should compute `w = x + t`. This subcircuit has only two inputs, so we need to set the right lookup table's `a2` input to `0`. We do so by setting pin `P4` to `0`, and then by passing that `0` to switch matrix output `o0`, by programming `11` into `o0`'s two-bit memory. `o0` is connected to `a2`, thus causing `a2` to have the value `0`, as desired.

Likewise, we need to set the right lookup table's `a1` input to `x`, which appears at the `D0` output of the left lookup table. We do this by programming `00` into the 2-bit memory for switch matrix output `o1`. We set the right lookup table's `a0` input to `t` by programming `10` for switch matrix output `o2`. Figure 7.27(b) shows how the programming of the 2-bit memories inside the switch matrix creates the desired connections. We then program the right lookup table to implement the function `x + t`, as shown in Figure 7.27(a).

Notice that, in the last few examples, to implement a desired circuit, we merely had to program different bits into the lookup tables and switch matrices. That's the appeal of FPGAs—they implement our circuit just by programming.

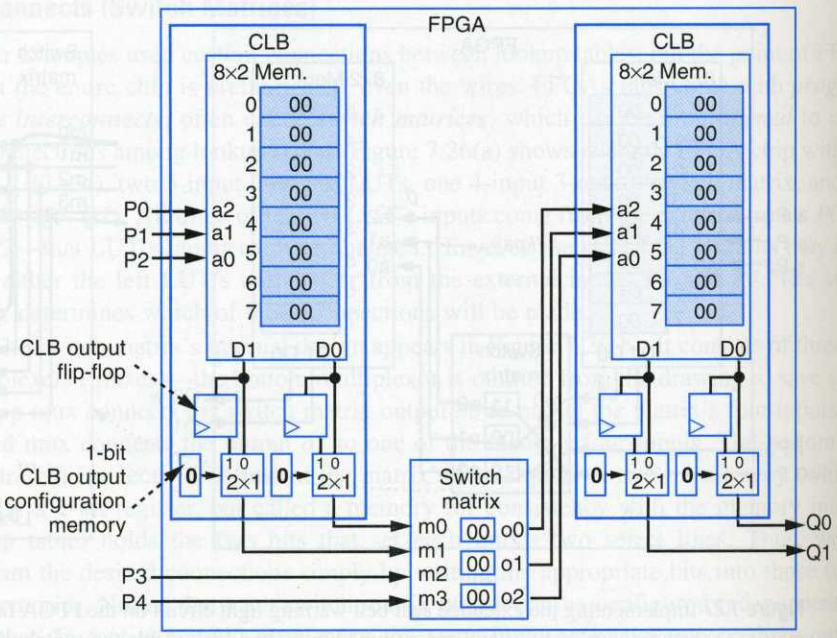


Figure 7.28 An FPGA with configurable logic blocks (CLBs), which contain flip-flops along with a lookup table.The configuration memory bit cells in the figure all contain 0s.

Configurable Logic Block

In the previous section, the illustrated FPGAs were missing a critical element needed to implement general circuits, namely, *flip-flops*. Flip-flops enable implementing sequential circuits on FPGAs.

FPGAs may include a flip-flop with each output of a lookup table—two flip-flops in the case of a 2-output lookup table. The lookup table and its flip-flops together are known as a ***configurable logic block***, or ***CLB***. A simple CLB is shown in Figure 7.28. Each configurable logic block has a 3-input 2-output lookup table, and has two outputs and two flip-flops. Each flip-flop is loaded every clock cycle with the corresponding lookup table output. Each output of the CLB can be configured to come either from the output's flip-flop, or directly from the corresponding lookup table output. That configuration is done by programming a 1-bit memory (which itself is a flip-flop, but we'll call it a memory to avoid confusion), shown in Figure 7.28, that controls a 2x1 mux for each CLB output.

The output flip-flops enable implementation of sequential circuits, such as circuits having registers, on the FPGA. Mapping a circuit onto 3-input 2-output CLBs involves partitioning the circuit into subcircuits having three or fewer inputs and two or fewer outputs, as for LUTs, but with the option that the outputs may come from a flip-flop in the circuit. The partitioning must ensure that the only place that a flip-flop appears in a subcircuit is immediately before the output of the output, because that is where the flip-flops exist in the CLBs.

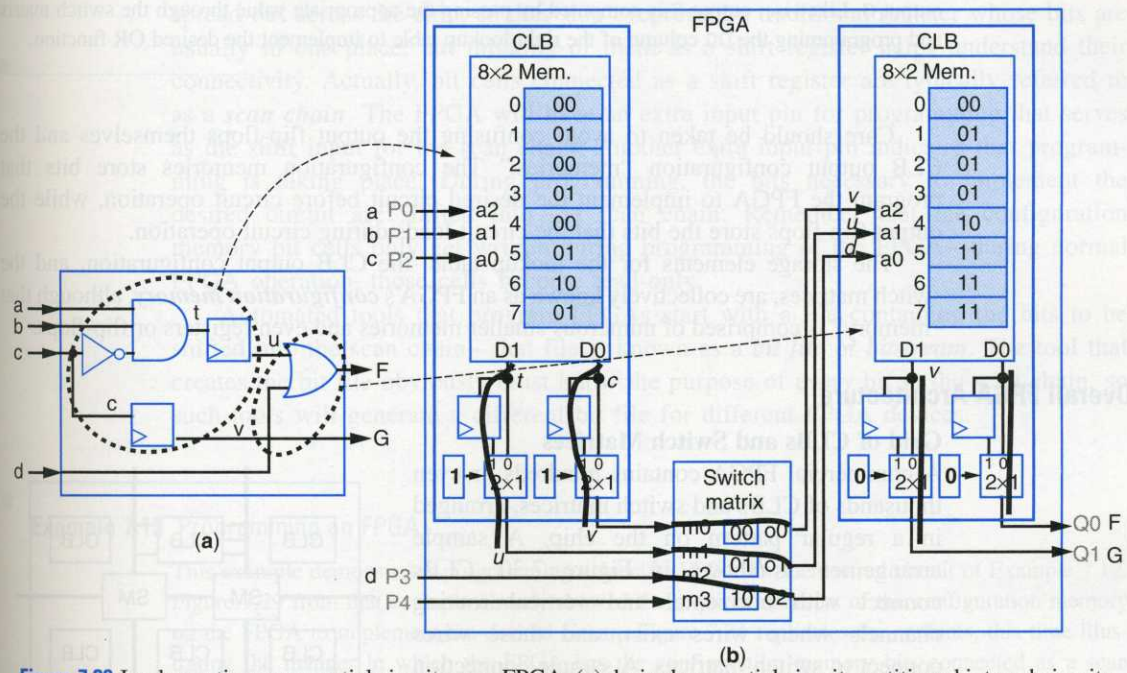


Figure 7.29 Implementing a sequential circuit on an FPGA: (a) desired sequential circuit, partitioned into subcircuits suitable for mapping onto CLBs, (b) programmed FPGA.

Example 7.12 Implementing a sequential circuit on an FPGA

This example implements the sequential circuit shown in Figure 7.29(a), having two flip-flops in the circuit, on the FPGA of Figure 7.28. The first step is to partition the circuit into subcircuits having three or fewer inputs and two or fewer outputs each, ensuring that the circuit's flip-flops only appear at the outputs of subcircuits, as shown in Figure 7.29(a). Based on that partitioning and the shown mapping to CLBs, we connect `a`, `b`, and `c` to the left lookup table in Figure 7.29(b). The left lookup table's `D1` output computes `abc'`, labeled as `t` in the figure. In the desired circuit of Figure 7.29(a), that value `t` feeds into a flip-flop (whose output is labeled `u`), and thus in Figure 7.29(b) we program the CLB's `D1` output to come from the CLB's `D1` flip-flop rather than from `D1` directly. The left lookup table's `D0` output computes `c`—note that the wire for `c` represents a simple "pass through" function, which can be programmed into the LUT just like a more complex function. In this case, because `c` is connected to address line `a0`, a `1` is programmed into the `D0` column for any word whose address has `a0 = 1`. In the desired circuit, that value `c` then feeds into a flip-flop (whose output is labeled `v`), and thus we program the CLB's `D0` output to come from the flip-flop, as shown.

The desired circuit of Figure 7.29(a) shows `v` connecting directly to external output `G`. However, the FPGA has no means for directly connecting the left CLB's `D0` output to an external output pin. Instead, we can create the desired connection by passing `v` to the switch matrix output `o0`, which connects to the right lookup table's `a2` input, and we then program that lookup table's `D1` column to pass `v` through, by programming a `1` into any word whose address has `a2 = 1`. Pin `Q1` thus represents

output G. Likewise, output F is computed by passing the appropriate value through the switch matrix and programming the D0 column of the right lookup table to implement the desired OR function.

Care should be taken to avoid confusing the output flip-flops themselves and the CLB output configuration "memories." The configuration memories store bits that program the FPGA to implement the desired circuit before circuit operation, while the output flip-flops store the bits that the circuit loads during circuit operation.

The storage elements for the lookup table, the CLB output configuration, and the switch matrices, are collectively known as an FPGA's ***configuration memory***, although that "memory" is comprised of numerous smaller memories and even registers or flip-flops.

Overall FPGA Architecture

Grid of CLBs and Switch Matrices

A commercial FPGA contains hundreds or even thousands of CLBs and switch matrices, arranged in a regular pattern on the chip. A sample arrangement is shown in Figure 7.30. CLBs connect with horizontal and vertical routing channels where wires exist, and those wires connect to switch matrices. A sample connection of a CLB to the wires in channels is shown for the top center CLB. The channels consist of tens of wires, represented in the figure just as single bolded wires.

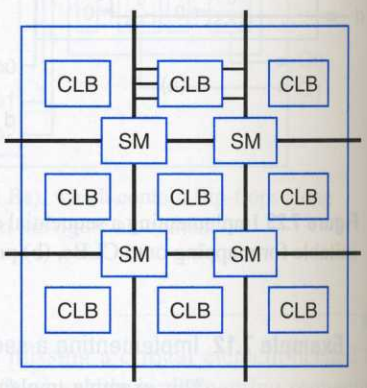


Figure 7.30 FPGA architecture.

CLBs and switch matrices in commercial FPGAs are more complex than described in this chapter. For example, CLBs may contain two lookup tables, or direct connections to adjacent CLBs to support carry chains of carry-ripple adders. Switch matrices may contain more inputs and outputs and more flexible switching options.

Furthermore, commercial FPGAs commonly include large embedded RAM memories for data storage, and embedded multipliers or multiply-accumulate units for fast multiplications. Memory and multiplcation operations are common in digital circuits, and so including RAM and multipliers results in faster and more compact implementation of those operations, and avoids occupying large numbers of CLBs and switch matrices that otherwise would be required to implement those operations. The RAMs and multipliers would be distributed throughout the FPGA fabric of Figure 7.30; there may be tens or hundreds of each in a single FPGA.

Programming an FPGA

One may wonder how to get the program bits into the configuration memories of an FPGA. The configuration memories are all the lookup table memories, the switch matrix memories, and the CLB-output configuration memories. Conceptually, programming is enabled by the FPGA having all the configuration memory bit cells connected as one big shift register (see Chapter 4). That shift register's bit cells are

spread out across the chip, so they don't represent a traditional register whose bits are usually in one place, but thinking of them as a shift register helps understand their connectivity. Actually, bit cells connected as a shift register are typically referred to as a ***scan chain***. The FPGA will have an extra input pin for programming that serves as the shift input for the scan chain. Another extra input pin indicates that programming is taking place. During programming, the bits necessary to implement the desired circuit are shifted into the scan chain. Remember that the configuration memory bit cells only get written during programming of the FPGA—during normal FPGA operation, those cells become read-only.

Automated tools that program FPGAs start with a file containing the bits to be shifted into the scan chain—that file is known as a ***bit file*** or ***bitstream***. The tool that creates the bit file obviously must know the purpose of every bit in the scan chain, so such tools will generate a different bit file for different FPGA devices.

Example 7.13 Programming an FPGA

This example demonstrates programming a specific FPGA for the desired circuit of Example 7.12. Figure 7.29 from that example already showed the required contents of the configuration memory on the FPGA to implement the desired circuit. Figure 7.31 replicates the contents, this time illustrating the manner in which the FPGA has the configuration memory bits connected as a scan chain, using a thick dotted line.

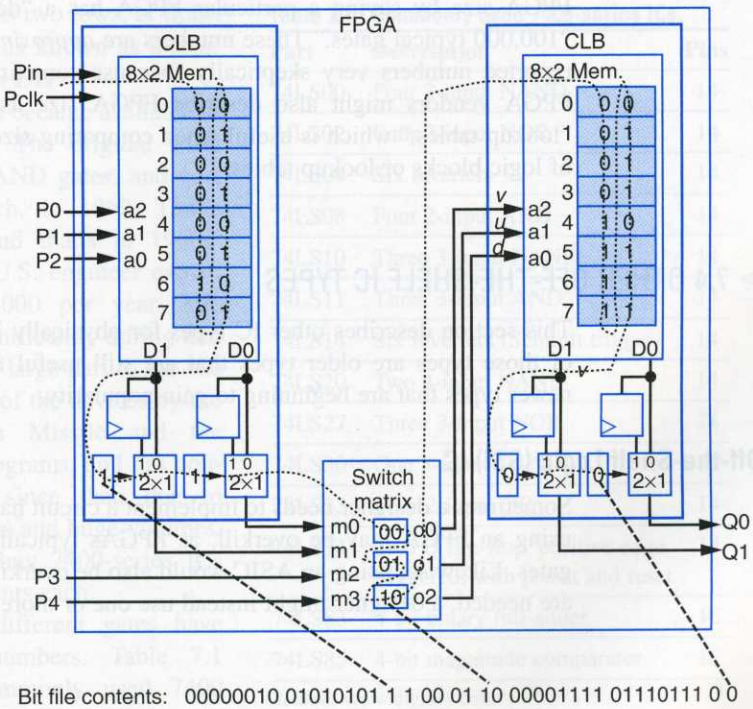


Figure 7.31 Programming an FPGA: all configuration memory bit cells exist in a scan chain. The bottom shows a bit file's contents that would be shifted in during programming—some relationships between the file's bits and configuration memory bit cells are shown.

The bottom of the figure shows the contents of a bit file that could be used to program the FPGA to implement the desired circuit. The bit file is determined simply by following the dashed line that represents the scan chain, placing `1`s and `0`s into the bit file as they appear in the figure. The spaces in the bit file are for readability of the figure, and would not actually exist in a bit file.

How Many Gates Does an FPGA Implement?

We usually think of a digital circuit's size using the notion of "gates" to represent design size. A design with 3000 gates is likely bigger than a design with 2000 gates. Of course, whether that statement is true depends on the type of gates used in each design (e.g., because XOR gates are bigger than NAND gates, 2000 XOR gates may actually be bigger than 3000 NAND gates), as well as the number of inputs to each gate (a 20-input gate is bigger than a 2-input gate). Thus, a common method of indicating design size for a circuit *approximates the number of 2-input NAND gates* that would be required to implement the circuit. So when we say that a circuit consists of 3000 gates or 2000 gates, we typically mean that if those circuits were implemented using 2-input NAND gates, they would require 3000 2-input NAND gates and 2000 2-input NAND gates, respectively.

FPGAs have lookup tables and switch matrices inside, not gates. FPGA sizes are therefore typically reported by considering how large a circuit made up of 2-input NAND gates could be implemented using the FPGA architecture. FPGA vendors may report FPGA size by saying a particular FPGA has a "density of 100,000 system gates" or "100,000 typical gates." These numbers are *approximations*, and many people view such reported numbers very skeptically (because sometimes companies like to exaggerate). FPGA vendors might also describe FPGA size as the number of "logic blocks" or "lookup tables," which is useful when comparing sizes of FPGAs having the same types of logic blocks or lookup tables.

▶ 7.4 OTHER OFF-THE-SHELF IC TYPES

This section describes other IC types for physically implementing digital circuits. Some of those types are older types that are still useful for particular situations. Others are newer types that are beginning to gain popularity.

Off-the-Shelf Logic (SSI) IC

Sometimes a designer needs to implement a circuit having just a few gates. In these cases, using an FPGA may be overkill, as FPGAs typically support thousands or millions of gates. Likewise, using an ASIC would also be overkill. For cases where only a few gates are needed, a designer might instead use one or more off-the-shelf logic ICs.

An Appolo rocket carrie astronauts to the moon. Neil Armstrong's famou words upon stepping on the moon, "One small step for man, one giant leap for mankind," were supposed to be "for a man" (NASA claimed th static obscured the word "a"). Most people understood Neil's meaning anyway.

A ***logic IC*** typically contains a few, perhaps ten or less, gates connected directly to the IC's pins, as shown in Figure 7.32. The IC shown has four AND gates and 14 pins. One pin is for power to the IC (known as *VCC*), the other for ground (*GND*) (see Chapter 2). The remaining pins connect to the four AND gates in the IC, as shown in the figure. Different logic ICs have gate types other than AND, such as OR, NAND, NOR, or NOT. To build a small circuit from these off-the-shelf logic ICs, we would simply place the ICs on a board and connect the appropriate pins. ICs with only a few gates are known as ***small-scale integration*** chips, or ***SSI*** chips.

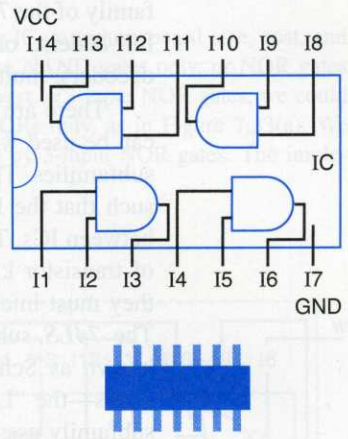


Figure 7.32 Example logic IC.

7400 ICs

The most popular off-the-shelf SSI ICs are known generally as ***7400-series*** ICs. A 7400 IC typically contains four to six logic gates, and about 14 pins. A particular 7400 IC is shown in Figure 7.31. The IC measures about 1/2 inch across. The IC package shown has two rows, or lines, of pins, and is thus known as a ***dual inline package***, or ***DIP***.

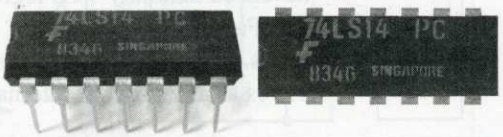


Figure 7.31 7400-series IC.

7400 ICs first became available in the early 1960s. The original 7400 chip had four NAND gates, and cost about $1000 each, in 1962. That's right—$1000. And that's in 1960s' dollars, when a U.S. engineer earned only about $10,000 per year. The price dropped significantly during that decade, thanks in large part to the use of huge numbers of the devices by the U.S. Minuteman Missile and the Apollo rocket programs, and has continued to drop since then due to cheaper transistors and huge volumes. Today, you can buy 7400-series ICs for just tens of cents each.

An Appolo rocket carried astronauts to the moon. Neil Armstrong's famous words upon stepping on the moon, "One small step for man, one giant leap for mankind," were supposed to be "for a man" (NASA claimed that static obscured the word "a"). Most people understood Neil's meaning anyway.

Parts with different gates have different part numbers. Table 7.1 shows some commonly used 7400 parts from Fairchild's 74LS00 sub-

Table 7.1: Commonly used 7400-series ICs.

Part	Description	Pins
74LS00	Four 2-input NAND	14
74LS02	Four 2-input NOR	14
74LS04	Six inverters	14
74LS08	Four 2-input AND	14
74LS10	Three 3-input NAND	14
74LS11	Three 3-input AND	14
74LS14	Six inverters (Schmitt trigger)	14
74LS20	Two 4-input NAND	14
74LS27	Three 3-input NOR	14
74LS30	One 8-input NAND	14
74LS32	Four 2-input OR	14
74LS74	Two D flip-flop, positive edge triggered, with preset and reset	14
74LS83	4-bit binary full-adder	16
74LS85	4-bit magnitude comparator	16

Source: www.digikey.com

family of the 7400 series. In addition to basic gates, the table shows ICs with D flip-flops, full-adders, or a magnitude comparator. Parts also exist for XOR, XNOR, buffers, decoders, multiplexers, up-counters, up-down-counters, and more.

There are several different subfamilies of 7400-series parts—parts from a subfamily can be used with other parts from the subfamily, but generally not with parts from other subfamilies. The reason is that the voltage and current setting of a subfamily are designed such that the ICs can be connected without requiring adjusting the voltage and current between ICs. The ***74*** series (e.g., 7400, 7402, etc.), is the basic subfamily, based on a type of transistor known as TTL—designers using logic ICs today only use 74-series ICs if they must integrate with old designs, and typically don't use the series for new designs. The ***74LS*** subfamily (e.g., 74LS00, 74LS02) uses a special type of TTL technology known as Schottky that results in lower power and slightly higher speed than the 74 series—the "L" in the name means "low-power," the "S" means "Schottky." The ***74HC*** subfamily uses high-speed (denoted by the "H") CMOS (denoted by the "C") transistors. The ***74F*** subfamily was introduced by Fairchild, consisting of fast (hence the "F") advanced Schottky TTL logic. Numerous other 7400 subfamilies exist.

Furthermore, additional series of off-the-shelf SSI ICs exist in addition to the 7400 series. Another popular series is the ***4000 series*** of ICs, a CMOS series that evolved in the 1970s as a low-power alternative to the TTL-based 7000 series. More series exist too.

Example 7.14 Seat belt warning implementation using off-the-shelf 7400 ICs

Using 74LS-series ICs shown in Table 7.1, physically implement the seat belt warning light circuit of Figure 7.1, shown again in Figure 7.32(a). We could implement the inverter using a 74LS04. The 74LS08 has 2-input AND gates, and we need a 3-input AND gate. A simple solution is to decompose the 3-input AND into two 2-input ANDs, as shown in Figure 7.32(b). The final implementation is shown in Figure 7.32(c).

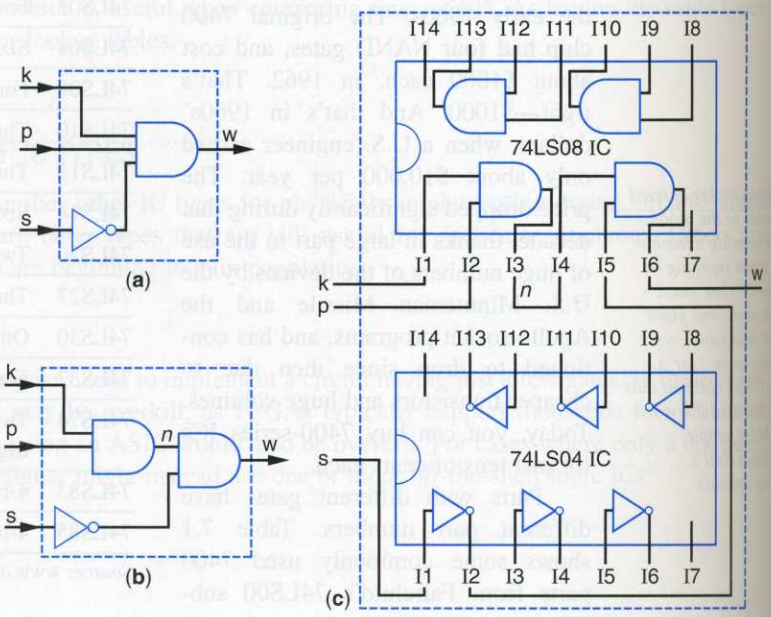


Figure 7.32 Implementing the seat belt warning circuit with 74LS-series ICs: (a) desired circuit, (b) circuit transformed to use 2-input AND gates, (c) circuit mapped to two 74LS ICs. Additional connections not shown would be power to the *I14* pins and ground to the *I7* pins on each IC.

Preferably, we would implement the circuit using just one IC, to reduce board size, cost, and power. Converting the circuit to use only one type of gate, like NAND gates only, or NOR gates only, could result in just one IC. For example, if we could convert to 3-input NOR gates, we could use the 74LS27 chip. We start by converting the circuit to NORs only, as in Figure 7.33(a). We remove the double inversion, and replace the single inversions by 3-input NOR gates. The implementation using a single 74LS27 IC is shown in Figure 7.33(c).

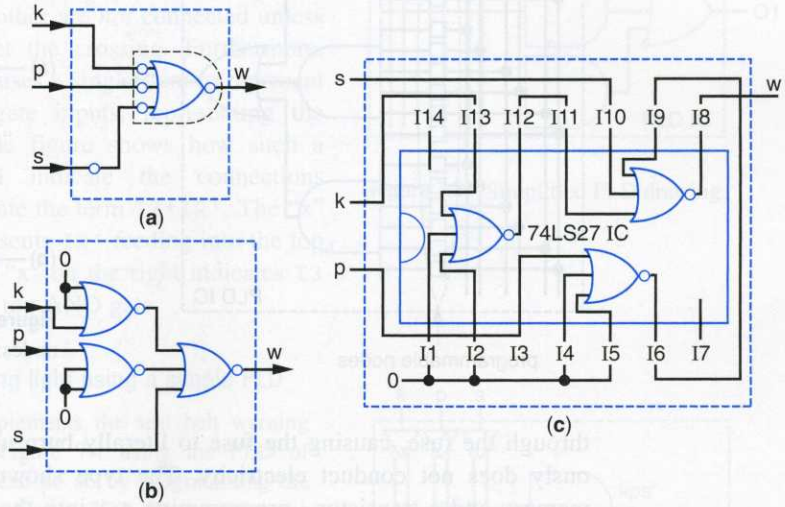


Figure 7.33 Implementing the seat belt warning circuit with one 74LS IC, namely, the 74LS27 consisting of three 3-input NOR gates: (a) desired circuit transformed to NOR gates with inversion bubbles, (b) circuit with double inversions eliminated and single inversions replaced by 1-input NOR gates, (c) circuit mapped to a 74LS27 chip. Additional connections not shown would be power to the *I14* pin, and ground to the *I7* pin.

Simple Programmable Logic Device (SPLD)

A ***programmable logic device***, or ***PLD***, is an IC that can be configured to implement a variety of logic functions, ranging from tens of gates to thousands of gates. PLDs became popular in the 1970s (thus predating FPGAs), because PLDs could implement far more functionality in a single IC than was possible using SSI ICs.

A PLD device contains a prefabricated circuit with a set of external inputs feeding into a large AND-OR circuit structure, with the special feature of allowing the user to configure (via "programming") which external inputs connect to the AND gates. For example, Figure 7.34 shows a basic PLD with three inputs feeding into three AND gates followed by an OR gate. The inputs feed into the AND gates in both true and complemented forms. Each wire feeding into each AND gate passes through a programmable node, which can either pass the node's input to the node's output, or disconnect the node's input from the node's output. Thus, by programming the nodes, a PLD can implement *any* 3-term function of three inputs.

The programmable node design varies among types of PLDs. Figure 7.35 shows two types. The type shown in Figure 7.35(a) is fuse-based PLD. A fuse conducts like a wire, unless the fuse is "blown," meaning a higher-than-normal current is passed

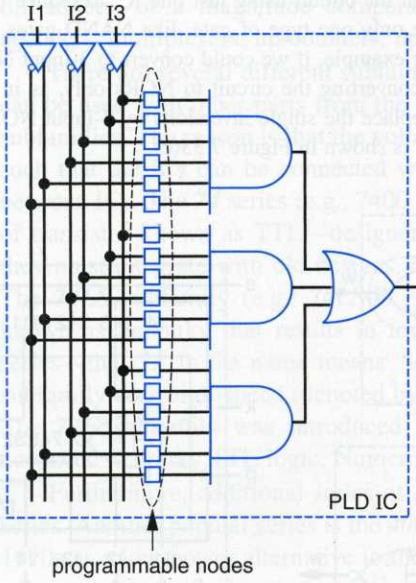


Figure 7.34 A basic example of a programmable logic device. (AND gates are wired-AND.)

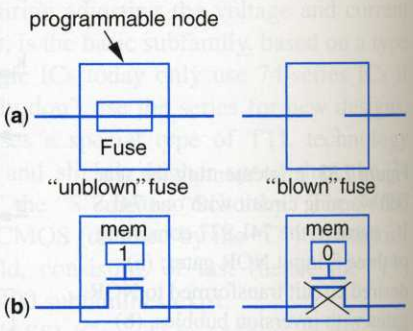


Figure 7.35 Two types of programmable nodes: (a) fuse based, (b) memory based.

through the fuse, causing the fuse to literally burn up and break. A blown fuse obviously does not conduct electricity. The type shown in Figure 7.35(b) is based on memory and a transistor—programming a `1` into the memory causes the transistor to conduct, while programming a `0` causes the transistor to not conduct. We omit the details of how to program the fuses or program the memories themselves. Memory-based PLDs can usually be reprogrammed, in contrast to fuse-based PLDs that can only be programmed once, and that are known as ***one-time programmable*** (***OTP***) devices. Fuse-based PLDs are popular in electrically noisy applications, like space applications, since memories can have their contents changed from radiation in space. They are also popular in applications demanding high security, since malicious enemies can't reprogram the device. Memory-based devices are more common, however, since they can be reprogrammed and thus reduce costs when we make design changes. The memories used are almost always nonvolatile, meaning the memories don't need power to retain their stored bits. (See Section 5.7 for more information on nonvolatile memories.)

You might be wondering how those AND gates work when the programmable node is programmed to disconnect an input. In other words, how does the AND gate treat an input with no connection—as a `0`, a `1`, or something else? Actually, PLDs don't use normal AND gates. Instead, PLDs typically use what is known as "wired-AND." Explaining how wired-AND works is beyond the scope of this book, and instead the subject of a course on transistor-level circuits. For our purposes, we can think of a wired-AND gate as an AND gate that simply ignores unconnected inputs.

Real PLDs have more inputs, gates, and outputs than shown in Figure 7.34. PLD structure drawings thus benefit from a more concise way of drawing the circuits. A concise method of drawing PLDs is shown in Figure 7.36. Such a drawing doesn't show the programmable nodes, and simply utilizes an "x" to indicate a connection. In the drawing, wires that cross each other are *not* connected unless an "x" exists at the crossing. Furthermore, such a drawing uses a single wire to represent all the AND gate inputs, representing the wired-AND. The figure shows how such a drawing would indicate the connections needed to generate the term `I3*I2'`. The "x" on the left represents `I2'` feeding into the top AND gate. The "x" on the right indicates `I3` feeding into the top AND gate.

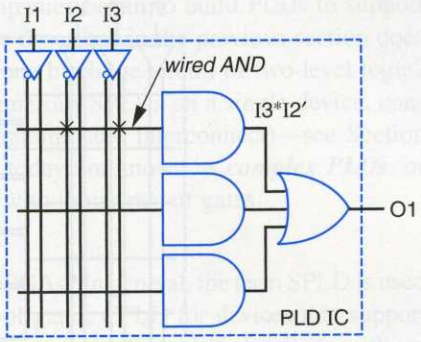


Figure 7.36 Simplified PLD drawing.

Example 7.15 Seat belt warning light using a simple PLD

This example implements the seat belt warning light system of Figure 7.1 using the PLD of Figure 7.36. We can do so by programming the PLD as shown in Figure 7.37. We generate the desired term `kps'` by programming the connections for the top AND gate as shown.

We want the bottom two AND gates to output `0`s so that the OR gate's output equals the top AND gate's output. We can achieve `0`s by ANDing an input with its complement—the result of `a*a'` is always `0`. The figure shows two ways of achieving a `0`, with the middle gate using just one of the inputs, and the bottom gate using all three inputs.

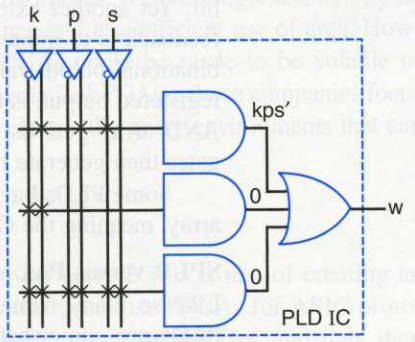


Figure 7.37 Seat belt warning system on a simple PLD.

PLDs typically have more than just one output. Figure 7.38(a) shows a PLD with two outputs instead of just one. Each output is an OR of up to three terms.

Many PLDs have a D flip-flop that stores each output's bit, and the PLD's output pin can be programmed to connect either from the OR gate output or from the flip-flop output, known as combinational or registered output, respectively. A PLD supporting combinational/registered output is shown in Figure 7.38(b).

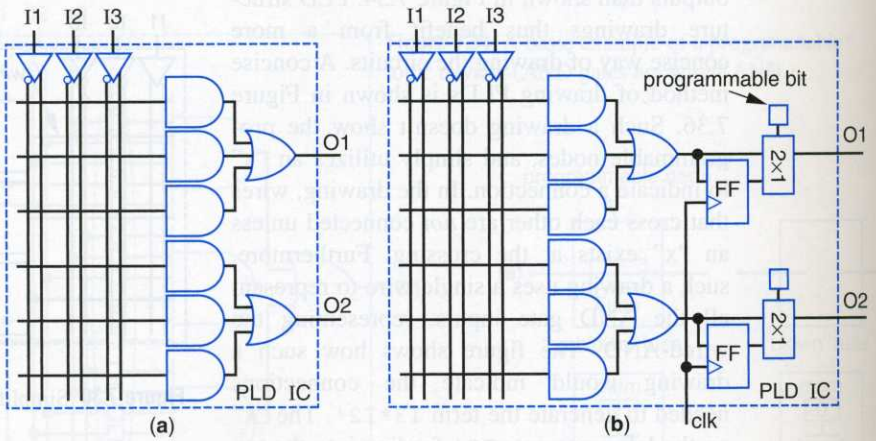


Figure 7.38 PLD: (a) with two outputs, (b) with programmable registered outputs.

Another extension is to allow the PLD output to be either the true or complemented value of the OR gate or flip-flop output, using a 2x1 mux controlled by a programmable bit. Yet another extension is for the output to feed back to the input array. One use of feedback is to implement functions with more terms, achieved by feeding back the combinational output value. Another common use of feedback, achieved by feeding back the registered output, is to implement a state register and control logic (i.e., a controller)—the AND array gets its inputs from the registered outputs and external inputs, and the OR gates then generate the external outputs and the next values for the state register.

Some PLDs have not only a programmable AND array, but also a programmable OR array, meaning the OR gate can get its inputs from any of the AND gates.

SPLD versus PAL versus GAL versus PLA

Like so many names in the rapidly evolving field of computer technology, names for PLDs are somewhat confusing. Originally, in the 1970s, PLDs consisted of programmable AND arrays and programmable OR arrays, and were known as ***programmable logic arrays***, or ***PLAs***. In the mid-1970s, a company named AMD (Applied Micro Devices, Inc.) developed PLDs that instead had OR gates with fixed rather than programmable inputs, as in Figure 7.38 and the other PLD figures in this chapter, and referred to such devices as ***programmable array logic***, or ***PALs*** ("PAL" is a registered trademark of AMD). PALs were originally fuse-based and hence one-time-programmable. A company named Lattice Semiconductor Corporation developed a PLD using a memory-based programming approach rather than fuses, resulting in reprogrammability, and referred to such devices as ***generic array logic***, or ***GAL*** (which are registered trademarks of Lattice Semiconductor Corporation). As PLDs became more complex (as the next section describes), PLDs based on PAL or GAL architectures (PLA architectures seem to be rare) became known as ***simple PLDs***, or ***SPLDs***, to contrast them with the more complex PLD varieties. Today, numerous companies manufacture SPLDs, and often state that their SPLD architecture is based on "PAL" or "PAL/GAL" architectures, with the distinction between PAL and GAL not seemingly relevant in that context. SPLDs typically support tens of logic gates to hundreds of logic gates.

Complex Programmable Logic Device (CPLD)

As IC transistor densities grew in the 1980s, companies began to build PLDs to support thousands of gates. However, the PLD architecture described in the previous section does not scale well to thousands of gates—who needs one big huge circuit of two-level logic? Instead, architectures evolved that consisted of numerous SPLDs on a single device, connected using switch matrices (also known as programmable interconnect)—see Section 7.3 for details on switch matrices. These devices today are known as ***complex PLDs***, or ***CPLDs***. CPLDs can typically implement designs with thousands of gates.

SPLDs versus CPLDs versus FPGAs

What's the difference among SPLDs, CPLDs, and FPGAs? In general, the term SPLD is used for devices that support tens of gates to hundreds of gates, CPLD for devices that support thousands of gates, and FPGAs for devices that support tens of thousands of gates to millions of gates.

Furthermore, today's SPLDs and CPLDs are almost always nonvolatile, meaning they can store their program even after power is removed, whereas FPGAs are almost always volatile, meaning they lose their program when power is removed—and thus must include external circuitry that stores the program in nonvolatile memory and that programs the FPGA from that memory on power up of the FPGA. FPGAs today are volatile in part because nonvolatile memory technology is hard to incorporate with fast logic technology on the same chip, resulting in slower circuit performance and less efficient use of area. However, conceptually, any of SPLDs, CPLDs, and FPGAs could be made to be volatile or nonvolatile, and today some companies do offer nonvolatile FPGAs—those companies focus on low power, and on space applications and other electrically noisy environments that can cause undesired changes in volatile memory.

FPGA-to-Structured-ASIC Flows

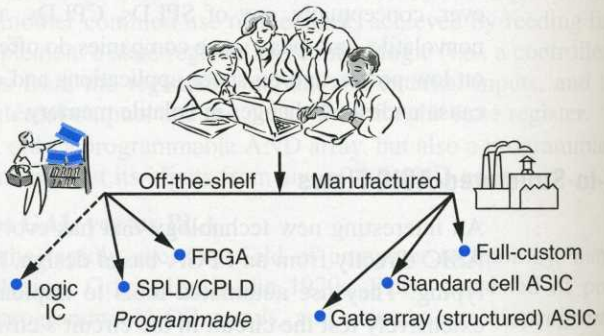
An interesting new technology that has evolved in the early 2000s is that of creating an ASIC directly from an FPGA-based design. Many designers use FPGAs for ASIC prototyping. They use automated tools to implement their circuit on FPGAs, and they then extensively test the circuit in the circuit's environment, for example, in a prototype digital video player or a prototype satellite communication chip. The FPGA-based prototype may be larger, costlier, and more power-hungry than an ASIC-based implementation, but can be useful for detecting and correcting errors in the circuit, for creating other components and software that interact with the circuit, and for demonstrating the eventual product. Once satisfied with the circuit, automated tools could be used to reimplement the circuit on an ASIC. The ASIC implementation traditionally did not utilize any information from the FPGA implementation.

Implementing large circuits on ASICs is a difficult task, even with automated tools. Nonrecurring engineering costs may exceed millions of dollars, and fabricating the IC may take months. Furthermore, any problem with the fabricated ASIC may require a second fabrication cycle, requiring additional weeks or months. Problems may arise in the ASIC that didn't appear in the FPGA due to the new implementation of the circuit as an ASIC—perhaps timing problems might arise, for example, due to the circuit being placed and routed in a completely different fashion than was done in the FPGA.

To ease the migration of a circuit from FPGA to ASIC, some FPGA vendors offer a structured ASIC approach. In this particular structured ASIC approach, a tool converts the *FPGA implementation* to an ASIC implementation, in contrast to converting the *original circuit* to an ASIC implementation. In other words, the structured ASIC will reflect the lookup table and switch matrix structure of the original FPGA. However, the structured ASIC will not be programmable, and thus will have faster lookup tables and faster switch matrices, because their contents will have been "hardwired" into the ASIC. The structured ASIC's lower layers can be prefabricated, with only wires left to be completed to implement a particular circuit. The result is less NRE cost (tens of thousands of dollars rather than millions) and shorter time-to-silicon (weeks rather than months), as well as less chance of unforeseen problems. The drawback is that the structured ASIC will be larger, slower, and more power-hungry than a traditional cell-based ASIC, but still better than an FPGA, generally about 2x smaller, faster, lower-power, and cheaper than an FPGA.

▶ 7.5 IC TRADEOFFS, TRENDS, AND COMPARISONS

As is the case when designing a digital circuit (as discussed in Chapter 6), physically implementing a circuit on an IC presents designers with numerous tradeoffs among design metrics. Common metrics include performance, size, power, cost, and time to availability. Figure 7.39 illustrates some of the tradeoffs among various IC types described in this chapter.



Easier design ← Quicker availability	0	0	1	6	12	(months)
← Lower NRE cost	0	0	1	50	150	(M$)
Faster performance →	0.05	0.5	1	3	5	(GHz)
Smaller size →	200	100	10	4	1	(sq mm)
Lower power →	10	5	0.1	0.05	0.01	(W)
Lower unit cost →	20	20	3	1	0.5	($)
More capacity → More optimized	0.001	0.1	0.5	1	2	(B gates)
			Sample values			

Figure 7.39 Tradeoffs among various IC types. Sample values for various metrics are also shown. For example, performance for an SPLD/CPLD might be 0.05 GHz, but 3 GHz for a standard cell ASIC. Actual values can vary tremendously from those shown.

Tradeoffs Among IC Types

Generally, a circuit implemented on IC types towards the right of the figure will have faster performance, smaller size, lower power, and lower unit cost (meaning lower cost per chip). For example, for a given circuit, a standard cell ASIC will be faster, smaller, and lower power than a gate array ASIC, because the cells can be chosen and placed to match the circuit, meaning there may be fewer cells and shorter wires. Likewise, a gate array ASIC uses gates rather than slower/larger/higher-power lookup tables, and wires rather than slower/larger/higher-power switch matrices. The circuit of Figure 7.24(a) could be implemented in a gate array ASIC with performance involving just a few gate delays from input to output, but when mapped to the FPGA of Figure 7.28, that circuit would have a longer delay—the inputs would pass through the left CLB's lookup table (which may have a delay of two gate-delays), through the left CLB's output muxes (another two gate-delays), through the switch matrix (another two gate-delays), through the right CLB's lookup table (another two gate-delays), and finally through the right CLB's output muxes, resulting in a total of ten gate-delays. In terms of size, a gate array implementation of the circuit of Figure 7.24(a) would require about 20 transistors, whereas the FPGA implementation using two CLBs and a switch matrix would require several hundred transistors. Some studies report that FPGAs are approximately 10 times slower, are 10–30 times bigger, and consume about 10 times more power, than ASIC implementations of the same circuit. However, these overheads are decreased compared to the previous decade, and the overheads are decreasing further as commercial FPGAs continue to mature.

Unit cost (which does not include NRE cost) is reduced towards the right of the figure, in part because IC cost is closely related to silicon size. Furthermore, for a given size chip, IC types to the right can implement larger circuits (i.e., the IC has more capacity) because the chip is optimized for the given circuit.

However, a circuit implemented on IC types towards the left of the figure will generally have quicker availability and lower NRE cost. For example, a gate array ASIC only requires wires to be fabricated and may thus be available in a few weeks and involve NRE costs of perhaps one million dollars, whereas a standard cell ASIC requires all layers to be fabricated, which may require several months and incur NRE costs of tens of millions of dollars. Likewise, an FPGA is prefabricated and thus may be immediately available if already stocked in a lab or may require only a few days to order from a vendor, and has no NRE costs, whereas a gate array ASIC requires a few weeks and NRE costs of perhaps one million dollars.

Figure 7.39 also makes a clear distinction between manufactured versus off-the-shelf IC types, because the difference in metric values between those two categories can be enormous, much like the difference between vehicles in the categories of aircraft versus automobiles. For example, while NRE costs are in the range of millions of dollars in the manufactured IC category, namely full-custom costing perhaps $150 million, standard cell $50 million, and gate array $1 million, they reduce to $0 for the off-the-shelf IC category. Likewise, while time to availability for the manufactured IC category is measured in months or weeks, the off-the-shelf IC types reduce it to just minutes or seconds—effectively zero time.

Example 7.16 Choosing an IC type

Consider a company that has a text encrypter circuit that will be used in three different projects A, B, and C.

- Project A involves putting the circuit into 100 million mobile phones; encryption speed must be 2.5 GHz, and each chip can be priced up to $5.
- Project B involves putting the circuit into 10,000 medical devices; encryption speed must be 1 MHz, and each chip can be priced up to $50.
- Project C involves putting the circuit into 100,000 automobiles; encryption speed must be 10 MHz, and each chip can be priced up to $10.

Suppose that all other factors are ignored, and that the company must choose from among standard cell ASIC, gate array ASIC, or FPGA IC types only. Considering the sample metric values shown in Figure 7.39, which IC type is best for each project?

For project A, the only IC type with at least 2.5 GHz speed is standard cell ASIC. The $50 million in NRE cost can be amortized over the 100 million chips by adding just $0.50 to the price of each chip, which when added to the $1 unit cost results in a price of $1.50 per chip, much less than the limit of $5. Thus, project A should use standard cell ASICs.

For project B, any of the three IC types meets the speed requirement of 1 MHz. The $50 million of NRE for a standard cell ASIC amortized over 10,000 chips would involve adding $5,000 to the price of each chip, which clearly exceeds the limit of $50 per chip. Even the $1 million of NRE for a gate array ASIC would require adding $100 to the price of each chip, which is still too much. Fortunately, the FPGA has no NRE cost, and a unit cost of $20, which is less than the $50 limit per chip. Thus, project B should use FPGAs.

For project C, all three IC types meet the speed requirement of 10 MHz. Amortizing standard cell NRE would result in too high a chip price. Amortizing the gate array ASIC NRE of $1 million over 100,000 chips would add $10 per chip, which when added to the $1 unit cost would result in $11 per chip, slightly exceeding the $10 per chip limit. However, the unit cost per FPGA chip is $20. Thus, none of the three IC types meets project C's price per chip requirement, but the gate array IC type comes very close, and is thus the best implementation choice.

IC Technology Trend—Moore's Law

Understanding the trends of IC technologies requires knowledge of Moore's Law. ***Moore's Law*** roughly states that IC capacity doubles every 18 months. Figure 7.40 plots such doubling, beginning with about 10 million transistors per IC in 1997. The plot uses a logarithmic scale for the *y*-axis—each tick mark on the *y*-axis represents 10 times more than the previous tick mark. The growth rate is astounding—ICs

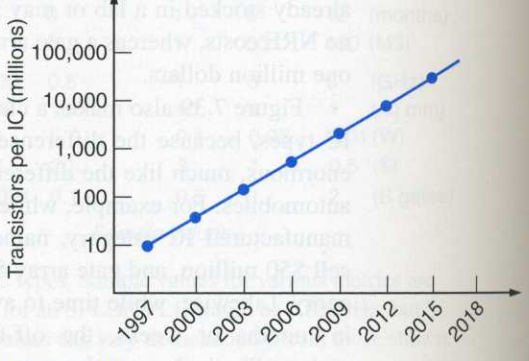


Figure 7.40 The trend of increasing transistors per IC.

In a 2004 speech, an Intel vice-president suggested that we might now consider transistors as essentially free.

are predicted to increase from 10 million transistors in 1997 to over 10 *billion* transistors in 2015. That means that the 2015 IC would hold 1000 times more transistors than the 1997 IC. In other words, the 2015 IC would be as powerful as about 1000 1997 ICs. This increasing capacity trend has also resulted in the cost per transistor dropping at nearly the same astounding rate. The increasing capacity is made possible by decreasing the smallest size of the individual parts within a chip, like the size of a single wire or of a transistor's gate, known as a chip's ***feature size***. Feature sizes in the 1980s were on the order of 1 micrometer (known as a micron), shrinking to 0.35 microns by 1995, and 0.18 microns by 2000 (around which time people began referring to feature size by nanometers rather than micrometers). Feature size shrinking continued to 90 nanometers by around 2003, then 65 nm by 2005, and 45 nm by 2007. Further shrinking may occur down to 32 nm by 2010, and 22 nm by 2012.

The IC capacity trend has many implications. One implication is that digital designers can create massively parallel designs that use huge numbers of functional units and registers, to create high-performance systems not previously practical. The number of required transistors for such designs might have been considered absurd just a decade earlier. Another implication is that the size overhead of FPGAs compared to ASICs (about 10x) becomes less relevant, making FPGAs an increasingly popular choice in more systems. Yet another implication is that designers increasingly need automated tools to help build these multimillion transistor circuits, and may increasingly wish to use RTL and even higher levels of design (e.g., C-based design) as the method for describing circuits, leaving the remaining design steps to tools. Smaller feature sizes also make layout more challenging (the closer items are, the more that can go wrong), which has increased standard cell ASIC NRE costs from tens of thousands of dollars in the 1980s to tens of millions of dollars in the 2010s.

In the early 1980s, many people predicted that feature sizes could not shrink below 1 micron. 2010 feature sizes are about 0.020 microns. As Neils Bohr said, "Prediction is very difficult, especially about the future."

At some point, Moore's Law must come to an end, because transistors cannot shrink to an infinitely small size. When that end will occur has been a subject of debate for many years. Some people claim Moore's Law is already slowing down and will perhaps end a couple decades into the 2000s.

▶ HOW DOES IT WORK?—INCREMENTALLY SCALING DOWN CHIP FEATURES

Moore's Law is enabled by chip feature sizes being made smaller every few years. A common question is: "Why don't chip makers just jump forward and make the smallest features now, rather than reducing feature sizes incrementally each year?" Part of the answer is that each incremental reduction creates a new set of problems that must be solved before the next incremental reduction can be considered, such as problems related to reliably creating smaller wires, updating tools to consider more and stricter layout rules when items are placed closer together on a chip, and testing chips for correctness even though there are more components on each chip and those components are less accessible from the chip's pins. These problems are tackled by thousands of researchers and engineers around the world, and solutions evolve slowly, often by improvements built on previous solutions. The solutions also require the more powerful computers enabled by a current generation of chips. An analogy is that of building a pyramid—only by standing on what is already built can one proceed to build the next higher level.

"If I have seen further it is by standing on the shoulders of giants." Isaac Newton.

The advent of ICs containing a billion transistors has led to ICs that contain what used to exist on multiple ICs. Thus, a single IC may contain dozens or hundreds of microprocessors, custom digital circuits, memories, buses, etc. An IC with numerous processors, custom circuits, and memories is known as a ***system-on-a-chip***, or ***SOC***.

Relative Popularity of IC Types

One may wonder about the relative popularity of IC types. Several ways exist to measure the popularity of an IC type.

One measure is each type's annual revenues. One 2007 study of IC sales reported $15.3 billion in total revenues with the following revenue percentages for different IC types: Full-custom—19%, standard cell—54%, gate array—3%, FPGA/PLD—24%. Another measure is the number of ***design starts*** for each IC type, which is the number of unique circuits implemented in each IC type, regardless of how many copies are made. A 2008 study reported about 3,000 ASIC design starts, versus about 100,000 FPGA design starts. Numbers from different studies vary, and there are many other measures that could be considered; these numbers are provided just to give a general feel for the popularity of the various IC types.

In 2002 alone, nearly 80 billion ICs (of all types) were produced. (Source: *IC Insights McClean Report, 2003.*)

The tools used to map digital designs to physical implementations, collectively known as ***electronic design automation*** tools, or ***EDA***, themselves form a market with revenues of $4 billion in 2008, employing about 30,000 people worldwide (source: EDA Consortium, 2009).

ASSPs

Rising ASIC NRE costs (from tens of thousands of dollars in the 1980s to tens of millions of dollars in the 2000s) have lead to the increasing popularity of ASSPs. An ***application specific standard product***, or ***ASSP***, is an off-the-shelf IC that targets a particular application domain, such as the video processing domain or the network processing domain, but that is intended to be sold to a variety of different companies ("users") who each will program and configure the device for their specific products. In contrast, ASICs are typically created by one company ("user") for a single device, and FPGAs are not focused on a particular application domain. For example, an ASSP for video processing might contain custom digital circuits optimized for high-speed low-power video compression and decompression (known as *codecs*)—such ASSPs often contain codecs for a wide variety of protocols (e.g., MPEG 2, MPEG 4, H.264, etc.) because the platform could be used in different products supporting different standards. An example is the Nexperia platform from Philips. One user may program the ASSP for use in a TV set-top box, while another user may place the ASSP in a security camera. As another example, an ASSP may focus on network processing, such as Intel's IXP device, which is programmed by different users to implement network gateways, routers, switches, wireless access points, and more. ASSPs typically include microprocessors and other programmable items (even FPGA fabrics) that can be programmed to customize the ASSP for a particular product.

Thus, the high NRE cost of building the ASSP IC, which may itself be created using full-custom, standard-cell ASIC, or some other IC type, can be amortized by the company that builds the ASSP over larger quantities of ICs due to being used by numerous

user companies in numerous products. Furthermore, ASSP users (distinct from the ASSP builder) obtain quicker availability of the IC and incur less risk, with drawbacks including a less optimized design. A 2008 study reported ASSP annual revenue to be about $60 billion, compared to about $20 billion for ASICs.

IC Types versus Processor Varieties

IC types and processor varieties are orthogonal implementation features. Two implementation features are ***orthogonal*** if we can select each independently (in mathematics, orthogonal means forming a right angle). Several processor varieties can each implement a desired system function, including a custom processor (i.e., a circuit created by a designer to implement a particular function, such as video compression) or a programmable processor (often called a microprocessor).

Figure 7.41 illustrates that the choice of processor variety is independent of the choice of IC type. Point *1* represents the choice of implementing desired system functionality using a custom processor circuit with a full-custom IC type. That choice results in a highly optimized design. Point *2* represents the choice of implementing a custom processor circuit on an FPGA. While the circuit may be optimized, the FPGA IC type results in a less-optimized implementation (compared to full-custom) but easier design overall.

Point *3* represents the choice of implementing system functionality as software executing on a programmable processor, where the programmable processor is implemented in standard cells. Point *4* represents the choice of implementing software on a programmable processor that is implemented on an FPGA. While that concept may seem strange, a programmable processor is just another circuit, so that circuit can be mapped to an FPGA just like any other circuit. Programmable processors mapped to FPGAs, known as ***soft core*** processors, are in fact becoming quite popular, because a designer can choose how many processors to put on a single IC (perhaps the designer wants 9 programmable processors on one IC), and because a designer can put single-purpose processors alongside programmable processors—all without having to fabricate a new IC.

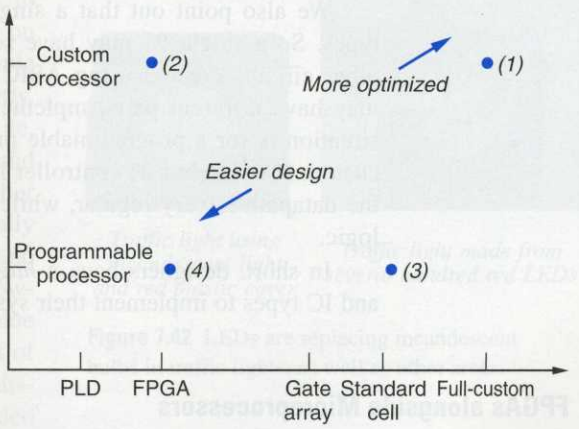


Figure 7.41 IC types and processor varieties are orthogonal implementation features. Four of the above ten possible choices are shown.

Of course, programmable processors can often be purchased as off-the-shelf ICs, so a designer using a programmable processor may not have to worry about the processor's IC type. But increasingly, designers must place a programmable processor

within their own IC, coexisting with other processors. When a programmable processor coexists on an IC along with other processors (programmable or custom), that programmable processor is called a ***core***. A ***hard core*** is built into the chip's hardware, while a soft core (mentioned above) is programmed onto the existing hardware (typically FPGA hardware).

The rise of cores in the 1990s and 2000s has led to a new processor type known as customized programmable processors or ***application-specific instruction-set processors*** (***ASIPs***), wherein a designer can extend a base programmable processor to have custom datapath components and custom instructions that provide improved performance for a particular application or application domain. The newly-created ASIP can then be implemented alongside other circuits or cores on an ASIC or FPGA.

Our discussion of IC types and processor varieties has thus far assumed just one type of each item (e.g., one type of FPGA). In reality, each item itself has many types. For example, dozens of different types of FPGAs are available, varying in their size, speed, power, cost, etc. Likewise, dozens of different types of programmable processors are available, also varying in those features. And we know that we can create different types of custom processors, varying also in their size, speed, power, etc. Thus, each point in Figure 7.39 and Figure 7.41 is actually a large collection of points that spread out in different directions on the plots, and may even overlap with other types. Furthermore, other IC types as well as processor varieties exist and continue to evolve.

We also point out that a single IC may actually incorporate several different IC types. So a single IC may have some circuits created using full-custom IC type, and other circuits created using ASIC or even FPGA types. Likewise, a single processor may have different parts implemented in different IC types. For example, a common situation is for a programmable processor to have its datapath implemented as a full-custom IC type, but its controller implemented in an ASIC type—the reason being that the datapath is very regular, while the controller is mostly unstructured combinational logic.

In short, designers have a *huge* number of choices in choosing processor varieties and IC types to implement their systems.

FPGAs alongside Microprocessors

This chapter has introduced FPGAs primarily as an alternative to manufactured ICs such as ASICs. Today, FPGAs are also viewed as an alternative to microprocessors in computing platforms. In particular, some computations can be performed faster on an FPGA than on a microprocessor. For example, multiplying 20 pairs of 4-bit numbers might require about 200 clock cycles on a microprocessor, as each multiplication requires instructions to fetch, multiply, and store the data and those instructions execute (mostly) sequentially. However, multiplying those 20 pairs of numbers could require just one clock cycle on an FPGA if the FPGA had capacity for 20 multipliers—all 20 multiplications could be done in parallel. Even if the FPGA clock cycle were 10 times slower than the microprocessor's clock cycle, the net result would still be a 20x speedup ((200/1)/10) on the FPGA. Many computing domains, such as biological computing, financial computing, and video processing, process streams of data that lend themselves well to computing on

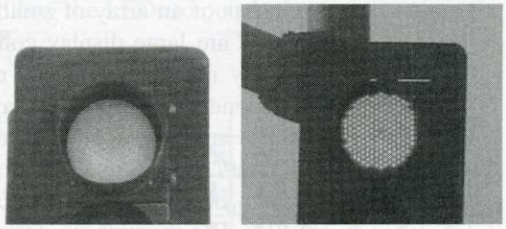
FPGAs. Thus, computer makers today increasingly provide support for adding FPGAs alongside microprocessors, and thus many desktop computers, server computers, and supercomputers today come with hardware and software support for FPGAs. Furthermore, new compilers exist that can translate high-level program code like C++ code into circuits on FPGAs. Thus, creating digital circuits is no longer just the domain of "hardware" engineers; it is becoming part of the domain of "software" engineers too.

▶ 7.6 PRODUCT PROFILE: GIANT LED-BASED VIDEO DISPLAY WITH FPGAS

In the late 1990s and 2000s, giant color video displays became popular at sport stadiums, car dealerships, casinos, freeway billboards, and various other locations. Most such video displays utilize a huge grid of light-emitting diodes (LEDs) driven by digital circuits.

A ***light-emitting diode*** (***LED***) is a semiconductor device that emits light when current passes through the device. In contrast, a traditional "incandescent" light bulb emits light when current passes through the bulb's internal filament, which is a high-resistance wire that heats up and glows when current flows through the wire—the wire, however, doesn't burn because it is enclosed in a vaccum or inert gas within the bulb. Because LED light comes from a semiconductor material and not from a hot glowing filament in a bulb, LEDs use less power, last longer, and can handle vibrations that would break a regular light bulb.

LEDs have long been used to display simple device status (e.g., on or off), text messages, or even simple graphics. However, until recently, LEDs were only available in white, yellow, red, and green colors, and were not very bright. Thus, earlier LED video displays were typically small, used only a single color, and were designed for indoor use. However, with the development of the blue LED in 1993, and the development of brighter LEDs, full-color LED displays evolved that can display video in much the same way as a computer monitor or television, even in sunny outdoor environments. In fact, LEDs, being a semiconductor technology, have been improving at a rate similar to transistors (which also use semiconductor technology). The improvement has followed what is known as ***Haitz's Law*** (the LED equivalent of Moore's Law), stating that the LED "flux per package" doubles every 18–24 months, which has been the case for several decades. Due to this improvement, many people predict that LEDs will replace incandescent light bulbs for home and office lighting. LEDs have already begun to replace incandescent bulbs in traffic lights, as illustrated in Figure 7.42. LED-based flashlights are now commonplace.



Traffic light using incandescent light and red plastic cover

Traffic light made from several hundred red LEDs

Figure 7.42 LEDs are replacing incandescent bulbs in traffic lights, as well as other areas.

Figure 7.43(a) shows a large LED video display capable of displaying full-color video on a 15-yard-by-8-yard screen. Because each LED is relatively large (1/8th of an inch wide, for example) in comparison to the pixels of a computer monitor, one has to stand several feet away from the LED display to view the image without noticing the individual LEDs. If we look closer at the LED display, as seen in Figure 7.43(b), we can see the individual lines of the displays. If we look even closer at the display, we can finally see the individual LEDs within the display, as shown in Figure 7.43(c). That figure shows that the LEDs are clustered into groups of red, green, and blue LEDs—each cluster represents one pixel. For the LED video display shown in Figure 7.43, each cluster of LEDs consists of five LEDs: two red, two green, and one blue LED. Giant video displays are indeed intended to be viewed from a distance, so most viewers don't see the individual LEDs.

Assume we want to create an LED video display capable of displaying a 720x480 pixel video, where each pixel simply consists of one red, one green, and one blue LED. If each LED cluster has a width of just over 3/8 inch (10 millimeters) and a height of 3/8 inch, our display will be roughly 24 feet wide and 16 feet high. Furthermore, our display will contain over one million individual LEDs, because 720 * 480 = 345,600 pixels, and the three LEDs per pixel results in 1,036,800 LEDs.

Controlling every LED using a single digital circuit would require millions of output pins and miles of wire to connect all of the LEDs. Instead, as depicted in Figure 7.44, an LED video display is constructed of smaller and smaller components. The LED display consists of an array of smaller components called *panels*, shown in Figure 7.44(a). The panels are large display components typically designed in a modular fashion such that display manufacturers can easily create custom-size video displays and repair broken components within a display simply by replacing individual panels. The LED display panels are further divided into LED *modules* that control the physical LEDs, shown in Figure 7.44(b). An LED module is the basic display component and, depending on the design of the module, can control anywhere from a few hundred to a couple thousand LEDs. For example, in designing a 720x480 pixel display, we may want to use an array of 6x6 panels, where each panel consists of an array of 5x5 LED modules. Each LED

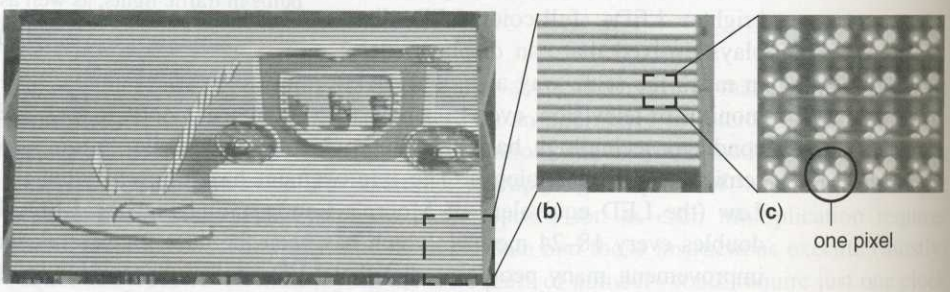


Figure 7.43 LED video display: (a) a large LED display (about 10 yards wide and 5 yards tall), (b) a closer view showing about 1 square yard, (c) a very close view showing about 1 square inch—16 "pixels" can be seen, each pixel having 2 red (upper-left and lower-right of pixel), 2 green (upper-right and lower-left of pixel), and 1 blue LED (center of pixel).

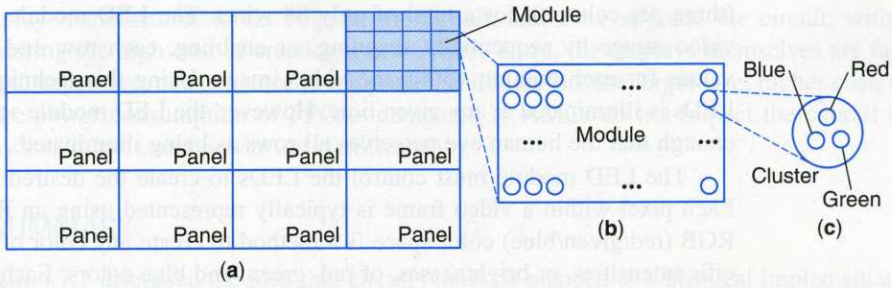


Figure 7.44 LED video displays are designed hierarchically: (a) the LED display consists of several larger panels, which can be composed to create different-sized displays, and which can be individually replaced to repair broken panels, (b) each panel consists of several smaller LED modules, responsible for controlling the individual pixels, and (c) each pixel consists of a cluster of red, green, and blue LEDs.

module would then need to control an array of 24x16 pixels, where each pixel is composed of three LEDs.

The LED video display operates by dividing the incoming video stream into separate streams for each panel. The panels further process the video stream by dividing the incoming video stream into even smaller streams for the LED modules. Finally, the LEDs modules display the video frames by controlling the LEDs to output the correct colors for each pixel, or LED cluster.

LED Module

The LED module controls the individual LEDs within the video display by turning the LEDs *on* and *off* at the proper times to create the final color images. Because each LED module can consist of thousands of LEDs, directly controlling each LED would require too many wires. Instead, as shown on Figure 7.45, the LEDs within the LED module are connected in a matrix with a single control wire for each row and three control wires for each column (one wire for each colored LED within the LED clusters). In the figure, the LED module controller controls an array of 2x3 pixels, where each pixel consists of three individual LEDs, for a total of 18 LEDs. But as shown, the controller uses only 9 wires to control those 18 LEDs. The wire saving using this row and column approach becomes even more significant for more pixels. An LED module with 24x16 pixels and three LEDs per pixel would have 24*16*3 = 1152 LEDs, but the controller would require only 16 wires (one per row) plus 24*3 wires

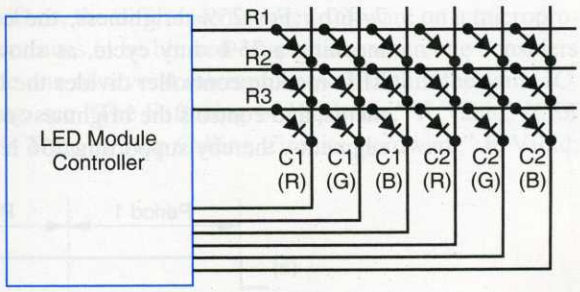


Figure 7.45 LED module circuit consisting of a matrix of red (R), green (G), and blue (B) LEDs controlled by the LED module controller. R1/R2/R3 are rows 1 through 3, and C1/C2 are columns 1 and 2—thus the matrix shown is 2x3 pixels, or 6 pixels total, with 18 LEDs total (3 LEDs per pixel).

(three per column), for a total of only 88 wires. The LED module controller displays a video image by sequentially scanning, or enabling, each row and displaying the pixel values for each column within the video image. Using this technique, only one row of LEDs is illuminated at any given time. However, the LED module scans the rows quickly enough that the human eye perceives all rows as being illuminated.

The LED module must control the LEDs to create the desired color for each pixel. Each pixel within a video frame is typically represented using an RGB color space. An RGB (red/green/blue) color space is a method to create any color of light by adding specific intensities, or brightnesses, of red, green, and blue colors. Each pixel within a video frame may be represented as three 8-bit binary numbers, where each 8-bit number specifies the intensity of the red, green, or blue colors. Thus, for each color, the LED module must be able to provide 256 distinct brightness levels. However, an LED by itself only supports two values: *on* and *off*, or full intensity and no intensity.

The largest LED display in 2004 was 135 feet wide by 26 feet tall, built using 10 large FPGAs, 323 moderate-size FPGAs, 333 flash memories, and 3800 PLDs. (Source: Xcell Journal, *Winter 2004.*)

To support 256 brightness levels, the LED module controller uses pulse width modulation. In ***pulse-width modulation*** (also known as ***PWM***), a controller drives a wire with a `1` value for a specific percentage of a time period—the signal being `1` is known as a pulse, the duration of the `1` is known as the pulse's width, and the percentage of the period spent at `1` is known as the *duty cycle*. When that pulse drives an LED, a wider pulse causes the LED to appear brighter to the human eye. Figure 7.46 illustrates how the LED module controller uses pulse width modulation to support various brightness levels for the LEDs. To illuminate an LED at full brightness, the controller simply drives the LED with 1 for the entire period, as shown in Figure 7.46(a). To illuminate the LED at half brightness, the controller uses a pulse with a 50% duty cycle, as shown in Figure 7.46(b). For 25% brightness, the controller sets the pulse to 1 for 25% of the period, meaning a 25% duty cycle, as shown in Figure 7.46(c). For an LED video display, the LED module controller divides the length of time each row is scanned into 255 time segments, and controls the brightness of the LEDs by turning each LED *on* for 0 to 255 time segments, thereby supporting 256 levels of intensity.

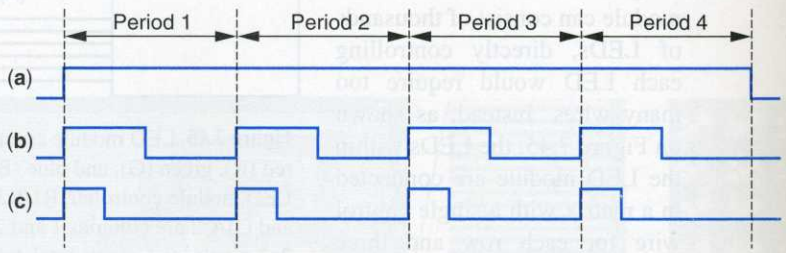


Figure 7.46 Pulse width modulation can be used to create various LED brightness levels: (a) for full brightness, the LED is always on, (b) for half brightness, the LED is turned on 50% of the time, and (c) for quarter brightness, the LED is turned on 25% of the time.

Because an LED module controller must provide precisely timed signals at a fast rate, custom processors are commonly used rather than just microprocessors. FPGAs are a common choice for implementing those custom processor circuits in LED video displays, for several reasons. First, FPGAs are fast enough to support the required scan rates. Second, the circuit on the FPGAs can be easily changed, making it possible for the

display manufacturer to fix bugs in the circuit, and even upgrade the circuit, without requiring the high cost of creating a new ASIC. Third, the displays themselves are fairly large, expensive, and consume much power, and therefore the larger size, higher cost, and more power consumption of FPGAs compared to ASICs do not impact the overall display's size, cost, and power too significantly.

▶ 7.7 CHAPTER SUMMARY

Section 7.1 discussed the idea that circuits must be mapped to a physical implementation so that those circuits can be inserted into a real system. Section 7.2 introduced some IC types that require that a new chip be fabricated to implement our circuit. A full-custom IC type gives the most optimized implementation, but is expensive and time-consuming to design. Semicustom IC types give very good implementations while costing less and taking less time to design, through the predesigning of the gates or cells that will be used on the IC. Section 7.3 described the increasingly popular IC type of FPGAs, and showed how a circuit could be mapped onto a set of programmable lookup tables and switch matrices. Section 7.4 highlighted several other IC types, including off-the-shelf SSI/MSI ICs, and programmable logic devices. Section 7.5 provided some data showing the relative popularity of the IC types described in the chapter.

An interesting trend in physical implementation is the trend toward programmable ICs (FPGAs in particular). Implementing functionality on an FPGA involves the task of downloading a bitstream into the FPGA IC device. One might notice the similarity of that task with the task of implementing functionality on a microprocessor, which also involves downloading bits into an IC device. Thus, the difference between software on a microprocessor and custom digital circuits continues to be blurred—especially when one considers that modern FPGAs can also include one or several microprocessors within the same IC. For more information on the blurring, see "The Softening of Hardware," F. Vahid, *IEEE Computer*, April 2003, and also "It's Time to Stop Calling Circuits Hardware," F. Vahid, *IEEE Computer*, September 2007.

▶ 7.8 EXERCISES

SECTION 7.2: MANUFACTURED IC TYPES

7.1 Explain why a gate array IC type has a shorter production time than a full-custom IC type.

7.2 Explain why the use of NAND or NOR gates in a CMOS gate array circuit implementation is typically preferred over an AND/OR/NOT implementation of a circuit.

7.3 Draw a gate array IC having three rows, the first row having four 2-input AND gates, the second row having four 2-input OR gates, and the third row having four NOT gates. Show how to instantiate wires to the gate array to implement the function `F(a,b,c) = abc + a'b'c'`.

7.4 Assume that a standard cell library has a 2-input AND gate, a 2-input OR gate, and a NOT gate. Use a drawing to show how to instantiate and place standard cells on an IC and wire them together to implement the function in Exercise 7.3. Draw your cells the same size as the gates in Exercise 7.3, and be sure your rows are of equal size.

7.5 Draw a gate array IC having three rows, the first row having four 2-input AND gates, the second row having four 2-input OR gates, and the third row having four NOT gates. Show