Module 4
CMOS Subsystem Design

Subsystems are building blocks to compose larger systems.

Architectural Issues

Guidelines may be
set out as follows:

l. Define the requirements (properly and carefully)'
2. Partition the overall architecture into appropriate subsystems.
3. Consider communication paths carefully in order to develop sensible interrelationships between subsystems.
4. Draw a floor plan of how the system is to map onto the silicon (and alternate between 2, 3 and 4 as necessary).
5. Aim for regular structures so that design is largely a matter of replication.
6. Draw suitable (stick or symbolic) diagrams of the leaf-cells of the subsystems.
7. Convert each cell to a layout.
8. Carefully and thoroughly carry out a design rule check on each cell.
9. Simulate the performance of each cell/subsystem'

The whole design process will be greatly assisted if considerable care is taken with:

1. **the partitioning of the system so that there are clean and clear subsystems with a minimum interdependence and complexity of interconnection between them.**

2. **the design simplification within subsystems so that architectures are adopted which allow the exploitation of a cellular design concept. This allows the system to be composed of relatively few standard cells which are replicated to form highly regular structures.**

In designing digital systems in MOS technology there are two basic ways of building logic circuits

1. **Switch Logic**

Switch logic is based on the 'pass transistor' or on transmission gates. This approach is fast for small array, and takes no static current from the supply rails' Thus, power dissipation of such arrays is small since current only flows on switching.

1. **Gate (restoring) Logic**

Gate logic is based on the general arrangement typified by the inverter circuits (the inverter being the simplest gate).

Both Nand and Nor and, with CMOS, And and Or gate arrangements are available. inverters are also employed to complement and restore logic levels that have been degraded (e.g. because they have passed through pass transistors)

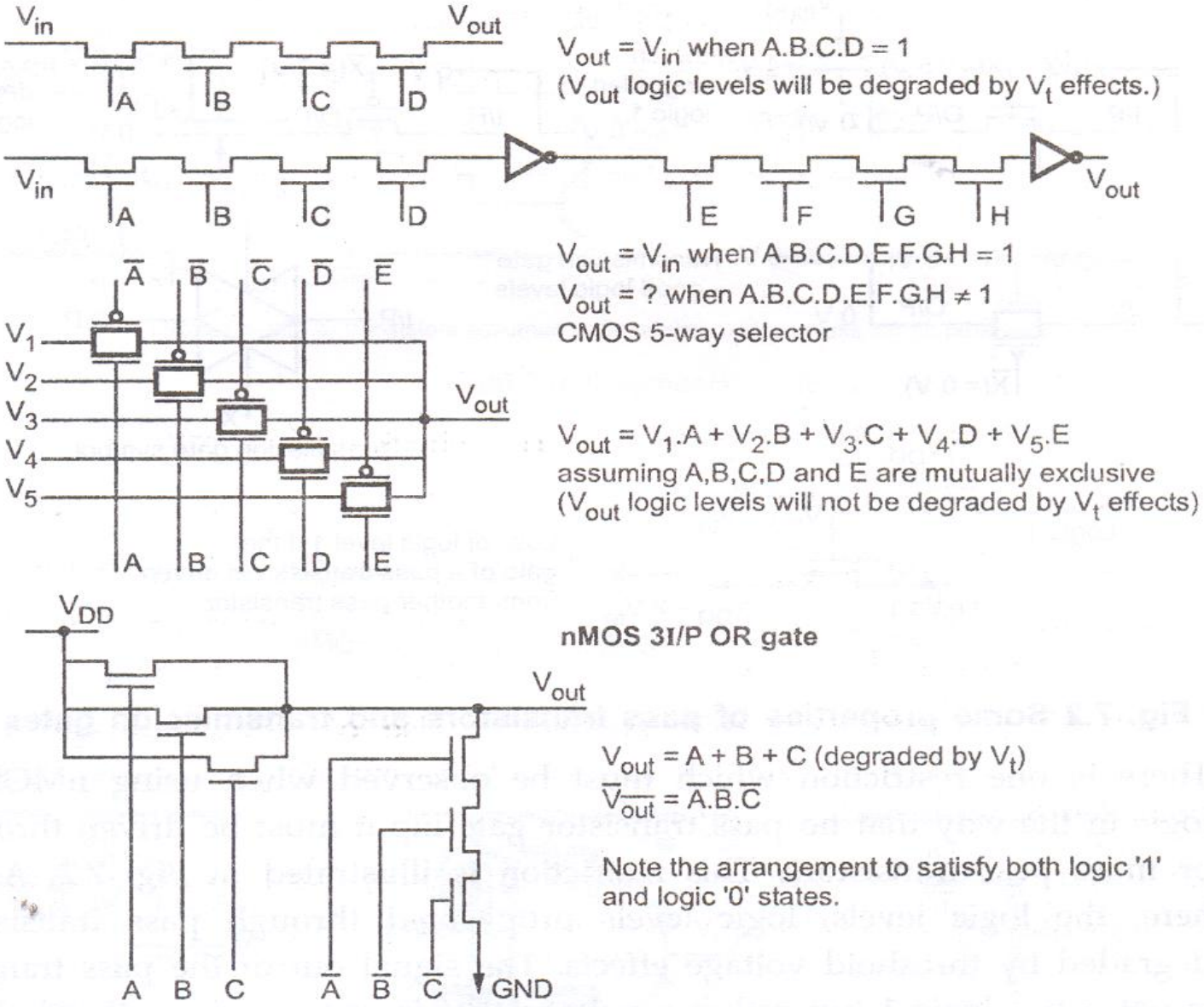


Fig. 7.1 Some switch logic arrangements

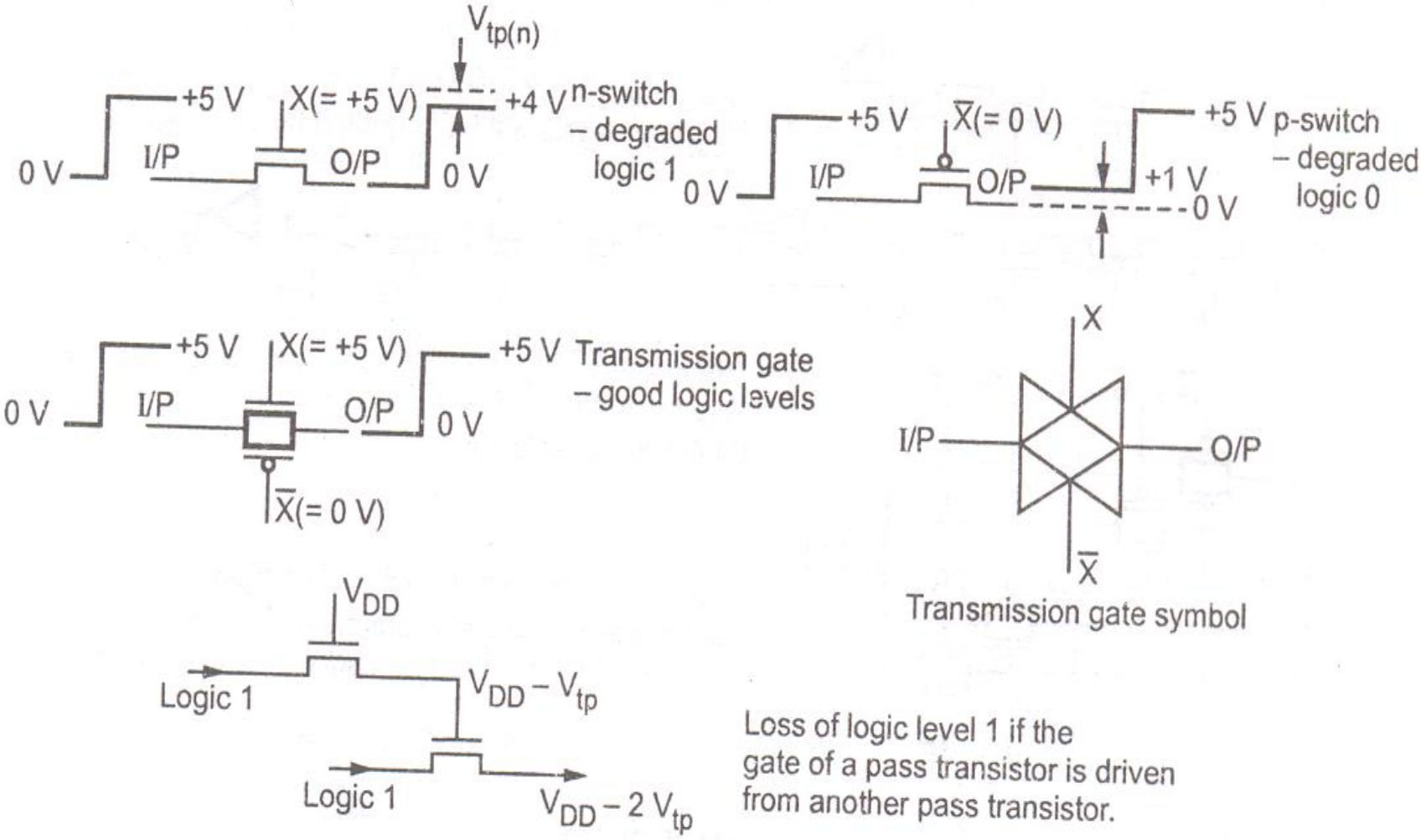
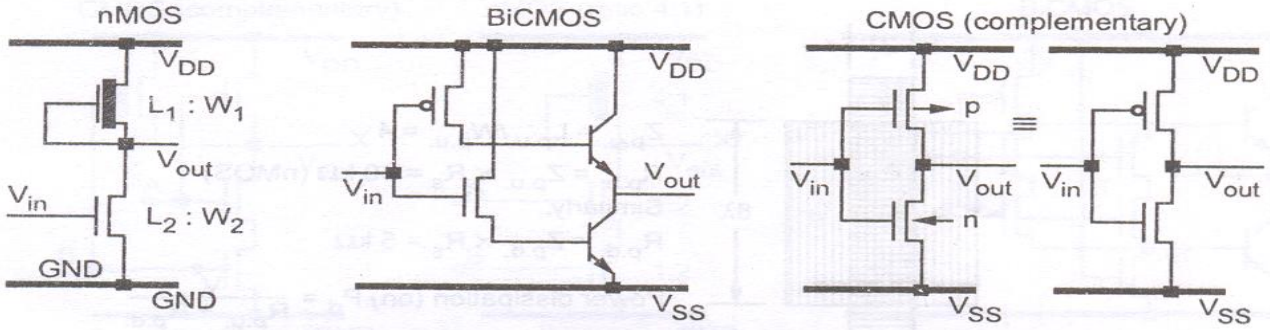
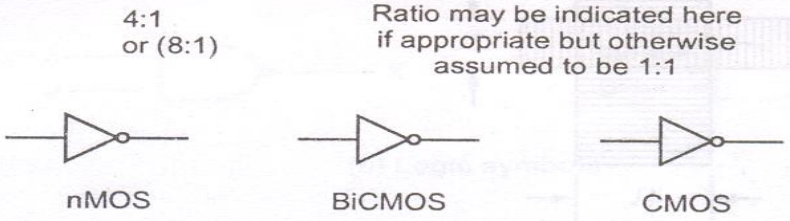


Fig. 7.2 Some properties of pass transistors and transmission gates

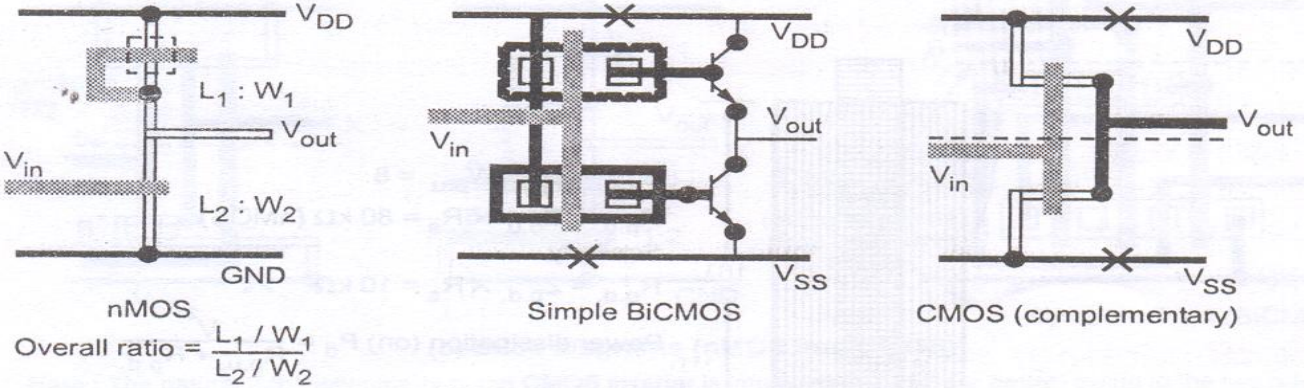


(**Note** : n- and p- transistors assumed to be minimum size unless stated otherwise)

(a) Circuit symbols

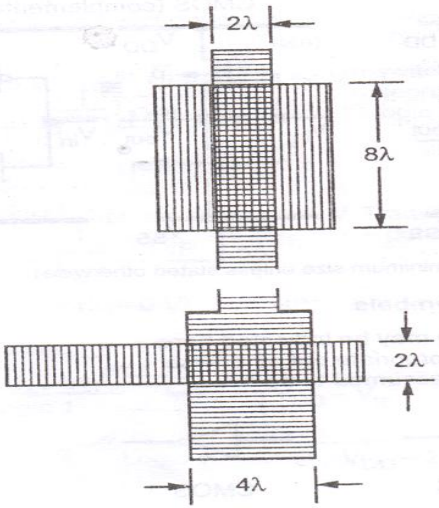


(b) Logic symbols



(c) Stick and symbolic diagrams

Fig. 7.3 nMOS, BiCMOS and CMOS inverters



$Z_{p.u.} = L_{p.u.}/W_{p.u.} = 4$

$R_{p.u.} = Z_{p.u.} \times R_s = 40\ k\Omega$ (nMOS)

Similarly,

$R_{p.d.} = Z_{p.d.} \times R_s = 5\ k\Omega$

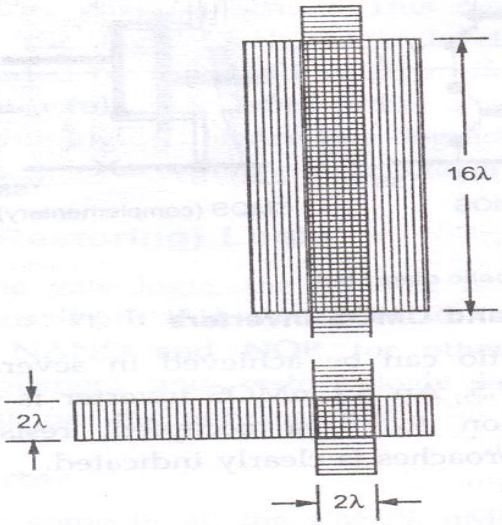
Power dissipation (on) $P_d = \dfrac{V^2}{R_{p.u.} + R_{p.d.}}$

$= 0.56$ mW

Input capacitance $= 2\square C_g$

Note : A 4:1 inverter is formed if the p.d. width is halved

Fig. 7.4 An alternative 8 : 1 nMOS inverter



$Z_{p.u.} = L_{p.u.}/W_{p.u.} = 8$

$R_{p.u.} = Z_{p.u.} \times R_s = 80\ k\Omega$ (nMOS)

Similarly,

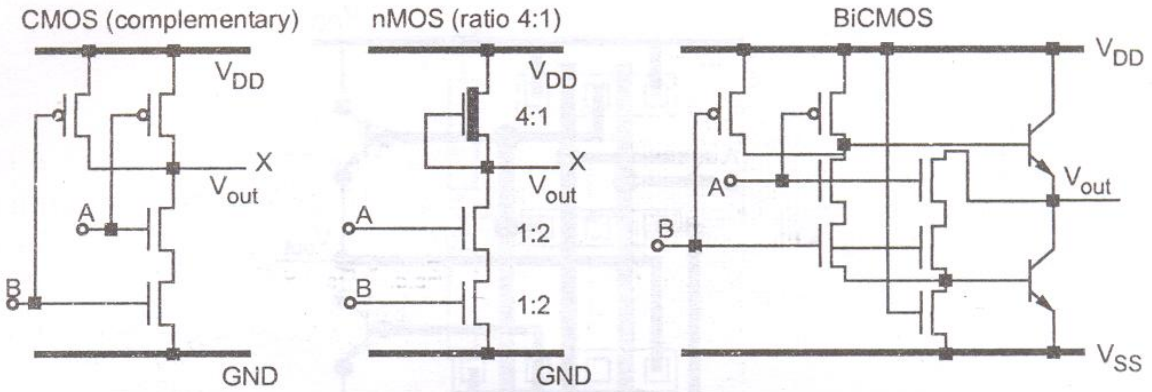
$R_{p.d.} = Z_{p.d.} \times R_s = 10\ k\Omega$

Power dissipation (on) $P_d = \dfrac{V^2}{R_{p.u.} + R_{p.d.}}$

$= 0.28$ mW

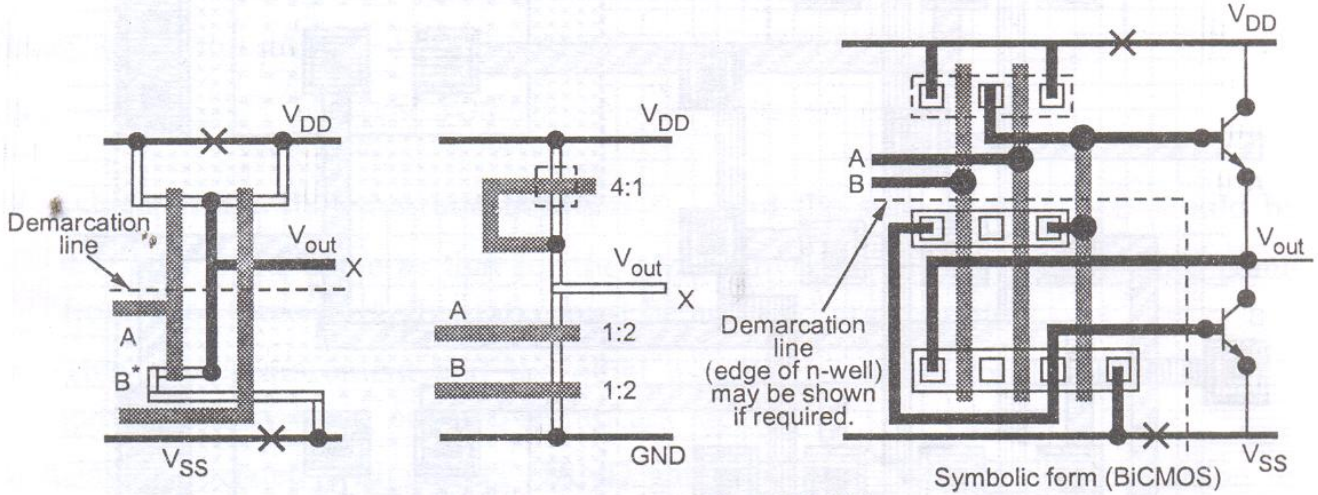
Input capacitance $= 1\square C_g$

Fig. 7.5 8 : 1 nMOS inverter (minimum size p.d.)



(a) Circuit diagrams

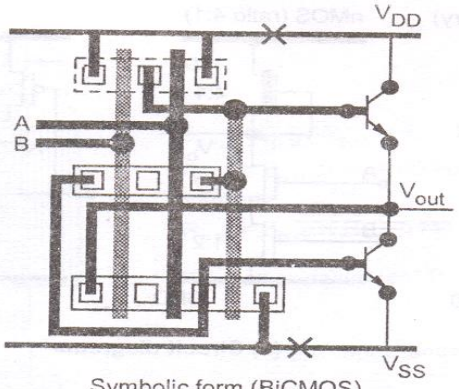
(b) Logic symbols



(c) Stick diagrams (nMOS and CMOS)

Note : The natural 2.5:1 asymmetry of the CMOS inverter is improved to 1.25:1 (or better) owing to the two n-type pull-down transistors in series for the two I/P NAND.

Fig. 7.6 CMOS, nMOS and BiCMOS 2-input NAND gates



Symbolic form (BiCMOS)

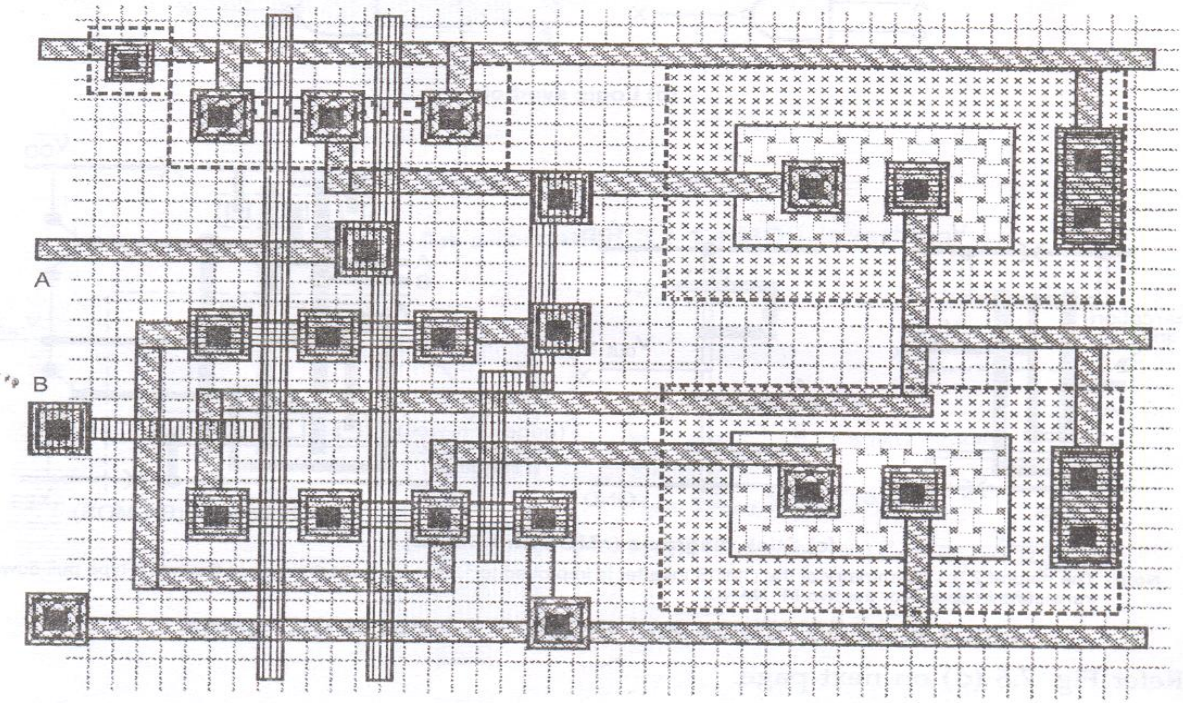


Fig. 7.6 (d) A BiCMOS two input NAND gate

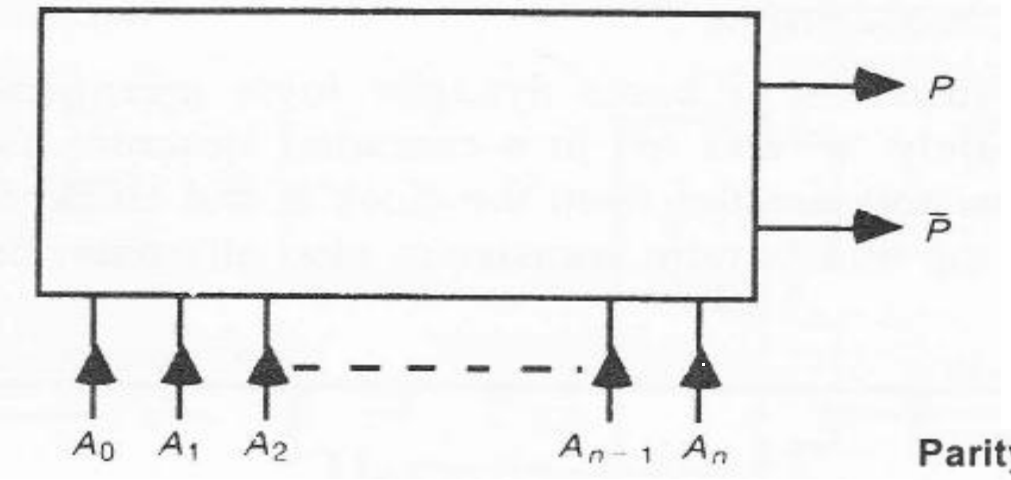
EXAMPLES OF STRUCTURED DESIGN (Combinational Logic)

Certain examples of structured of combinational logic are-

a. **Parity Generator**
b. **Multiplexers (Data Selectors)**

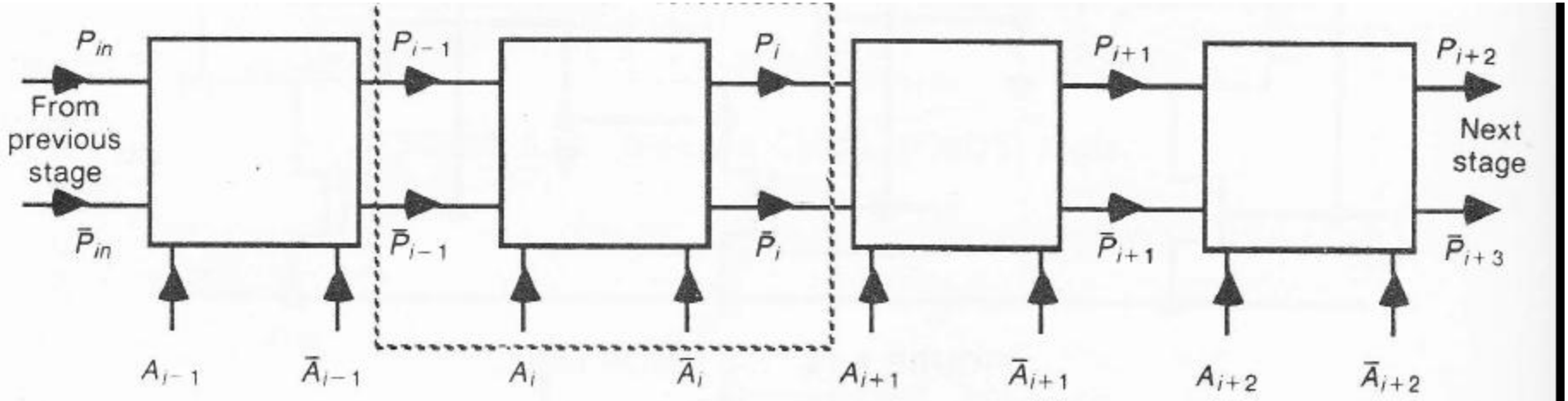
a. Parity Generator

A circuit is to be designed to indicate the parity of a binary number or word.
It will be seen that parity information is passed from one cell to the next and is modified or not by a cell, depending on the state of the input lines A_i and $\bar{A}_i$



$$P = \begin{cases} 1 & \text{Even number of 1s at input} \\ 0 & \text{Odd number of 1s at input} \end{cases}$$

Parity generator basic block diagram.



Note: Parity requirements are set at the left-most cell where $P_{in} = 1$ sets even and $P_{in} = 0$ sets odd parity.

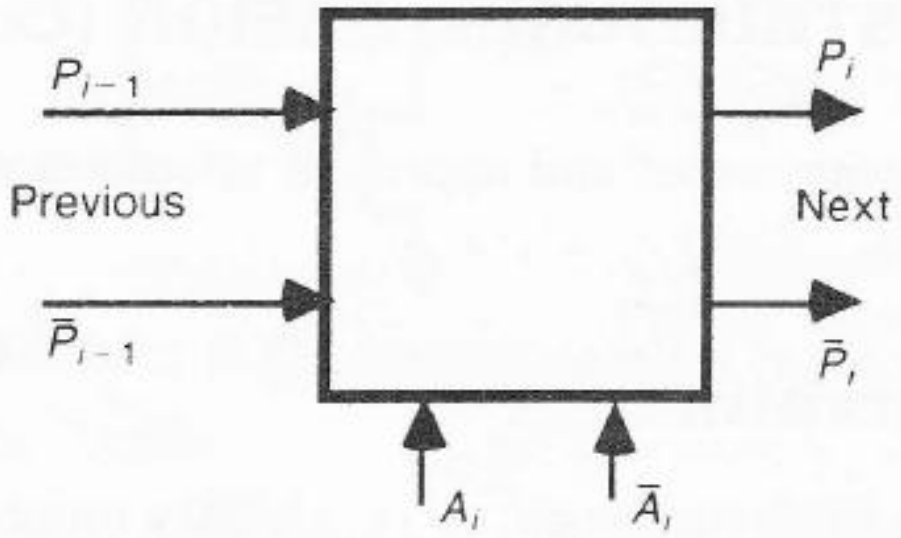


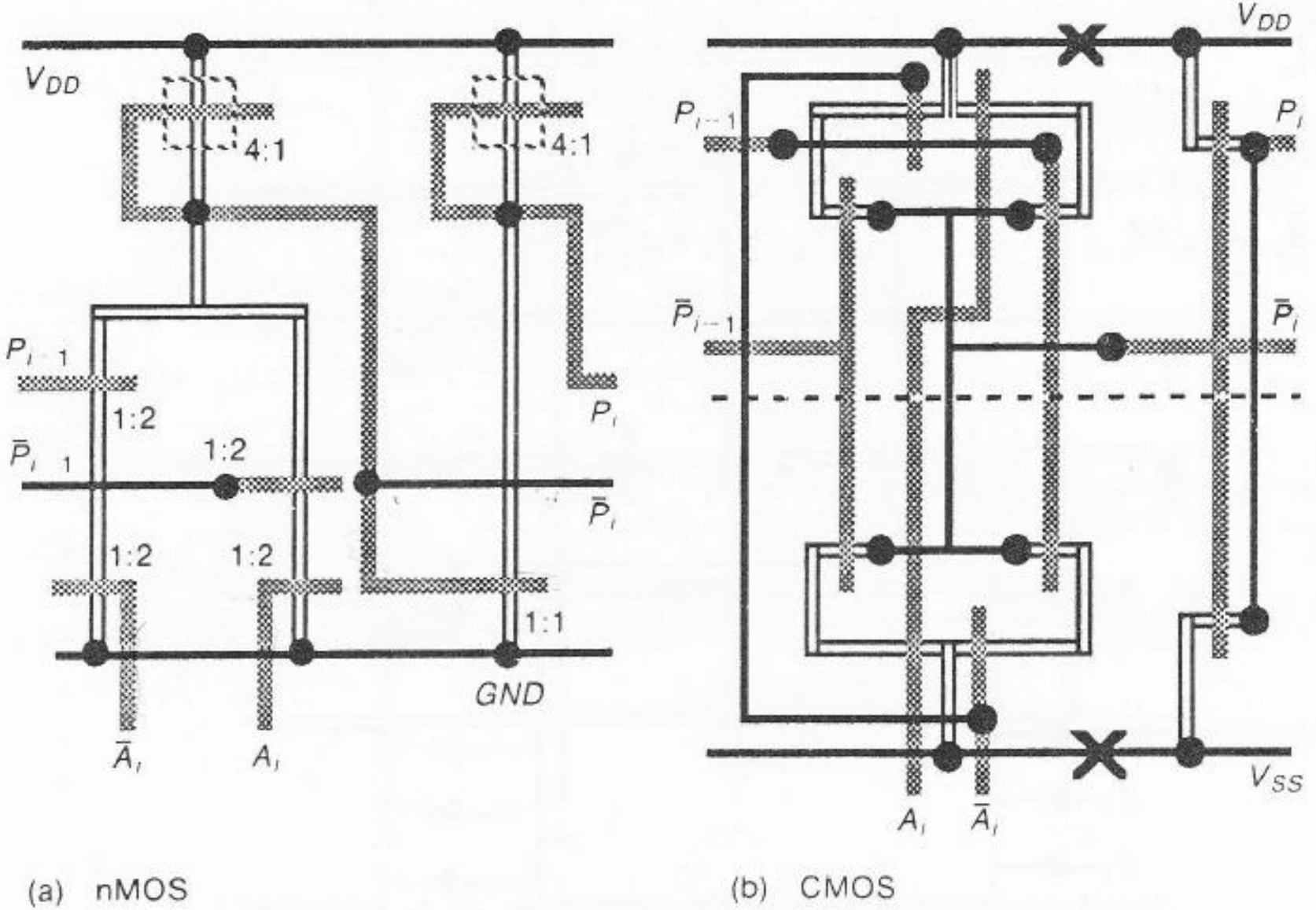
FIGURE 6.17 Parity generator—basic one-bit cell.

A little reflection will readily reveal that the requirements are:

$A_i = 1$ parity is changed, $P_i = \bar{P}_{i-1}$
$A_i = 0$ parity is unchanged, $P_i = P_{i-1}$

The circuit implements the function

$$P_i = \bar{P}_{i-1} \cdot A_i + P_{i-1} \cdot \bar{A}_i$$



(a) nMOS

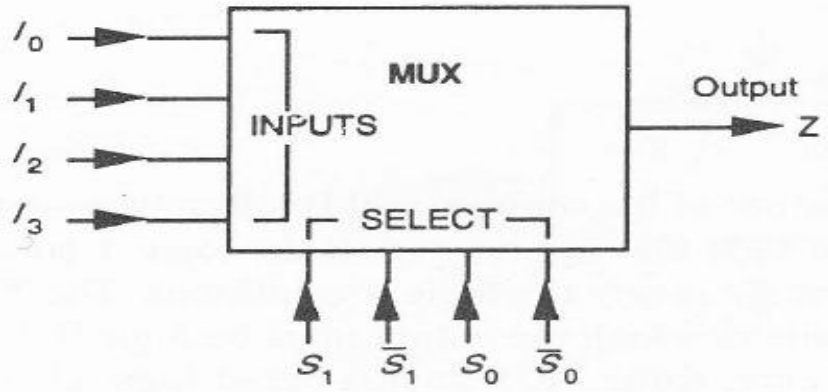
(b) CMOS

FIGURE 6.18 Stick diagrams (parity generator).

Multiplexers (Data Selectors)

The requirements and general arrangement of a four-way multiplexer a

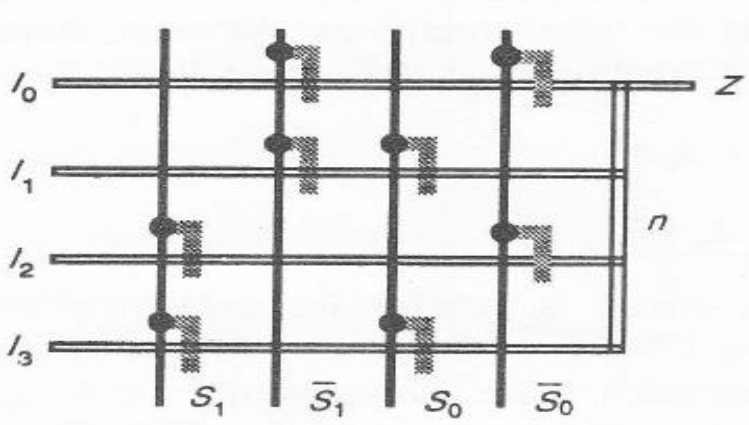
$$Z = I_0.\overline{S}_1.\overline{S}_0 + I_1.\overline{S}_1.S_0 + I_2.S_1.\overline{S}_0 + I_3.S_1.S_0$$



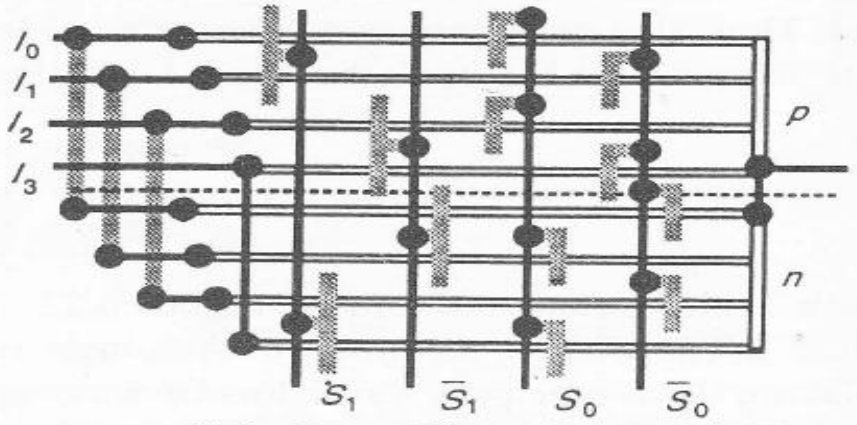
Truth table

S_1	S_0	Z
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

FIGURE 6.23 Selector logic circuit.



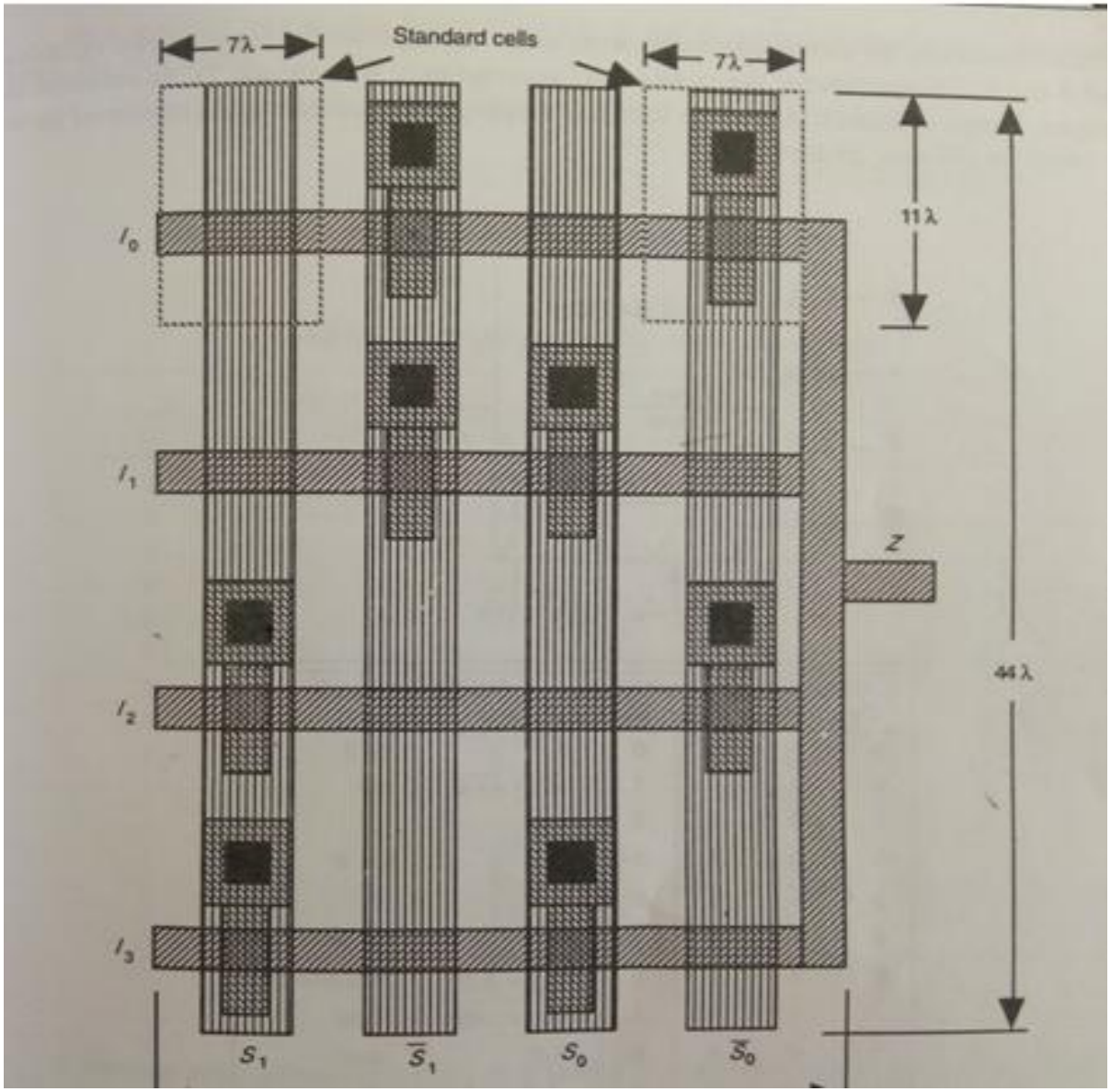
(a) nMOS switches



Note: V_{DD} and V_{SS} contacts not shown.

(b) Transmission gates (CMOS)

FIGURE 6.24 Switch logic implementations of a four-way multiplexer.


7λ
Standard cells
7λ
11λ
44λ
I_0
I_1
I_2
I_3
Z
S_1
$\overline{S}_1$
S_0
$\overline{S}_0$

FPGA

1.2 Boolean Algebra

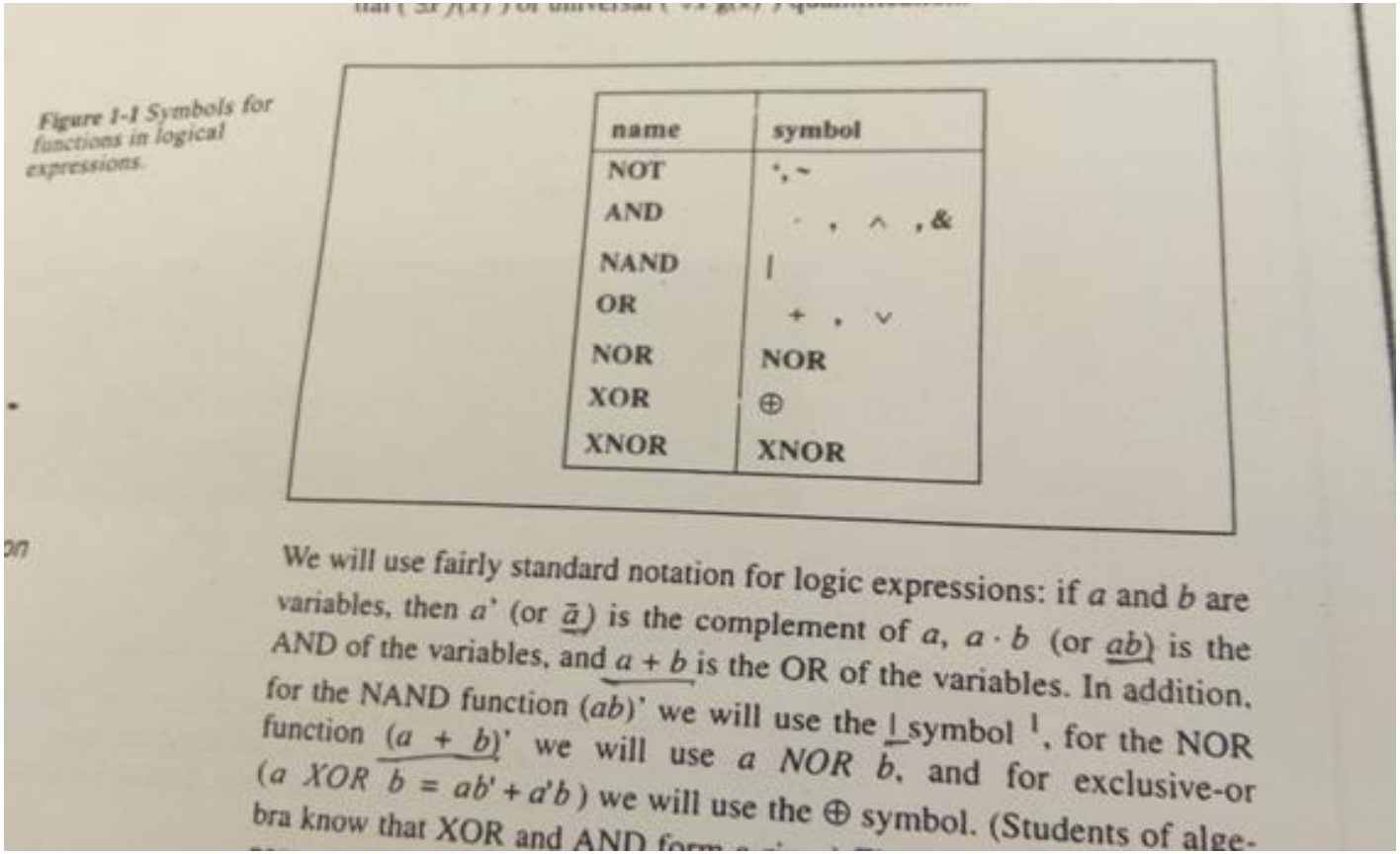


Figure 1-1 Symbols for functions in logical expressions.

name	symbol
NOT	', ~
AND	· , ^ , &
NAND	|
OR	+ , ∨
NOR	NOR
XOR	⊕
XNOR	XNOR

We will use fairly standard notation for logic expressions: if a and b are variables, then a' (or $\bar{a}$) is the complement of a, $a \cdot b$ (or ab) is the AND of the variables, and $a + b$ is the OR of the variables. In addition, for the NAND function $(ab)'$ we will use the | symbol [1], for the NOR function $(a + b)'$ we will use a *NOR* b, and for exclusive-or (a *XOR* $b = ab' + a'b$) we will use the ⊕ symbol. (Students of algebra know that XOR and AND form

Let's review some basic rules of algebra that we can use to transform expressions. Some are similar to the rules of arithmetic while others are particular to Boolean algebra:

- **idempotency**: $a \cdot a = a$, $a + a = a$.
- **inversion**: $a + a' = 1$, $a \cdot a' = 0$.
- **identity elements**: $a + 0 = a$, $a \cdot 1 = a$.
- **commutativity**: $a + b = b + a$, $a \cdot b = b \cdot a$.
- **null elements**: $a \cdot 0 = 0$, $a + 1 = 1$.
- **involution** : $(a')' = a$.
- **absorption**: $a + ab = a$.
- **associativity**: $a + (b + c) = (a + b) + c$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- **distributivity**: $a \cdot (b + c) = ab + ac$, $a + bc = (a + b)(a + c)$.
- **De Morgan's laws**: $(a + b)' = a' \cdot b'$, $(a \cdot b)' = a' + b'$.

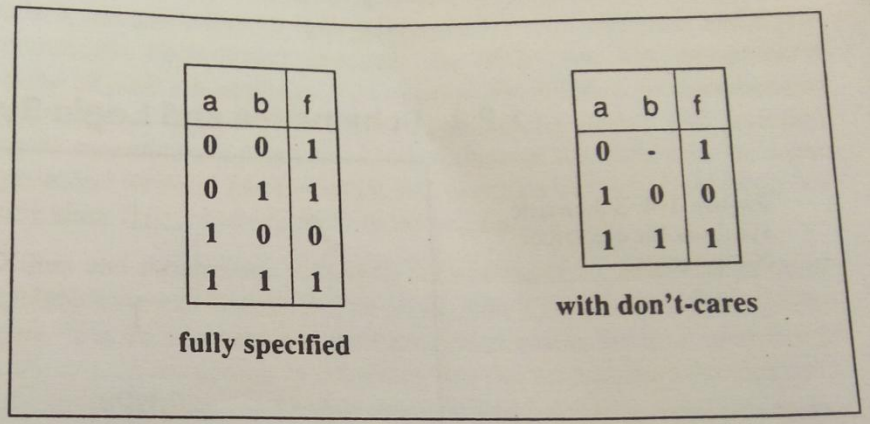
Completeness and irredundancy

A set of logical functions is **complete** if we can generate every possible Boolean expression using that set of functions—that is, if for every possible function built from arbitrary combinations of +, ·, and ', an equivalent formula exists written in terms of the functions we are trying to test. We generally test whether a set of functions is complete by inductively testing whether those functions can be used to generate all logic formulas. It is easy to show that the NAND function is complete, starting with the most basic formulas:

- $1: a|(a|a) = a|a' = 1.$
- $0: \{a|(a|a)\}|\{a|(a|a)\} = 1|1 = 0.$
- $a': a|a = a'.$
- $ab: (a|b)|(a|b) = ab.$
- $a + b: (a|a)|(b|b) = a'|b' = a + b.$

From these basic formulas we can generate all the formulas. So the set of functions {|} can be used to generate any logic function. Similarly, any formula can be written solely in terms of NORs.

Figure 1-2 An example of input don't-cares.



a	b	f
0	0	1
0	1	1
1	0	0
1	1	1

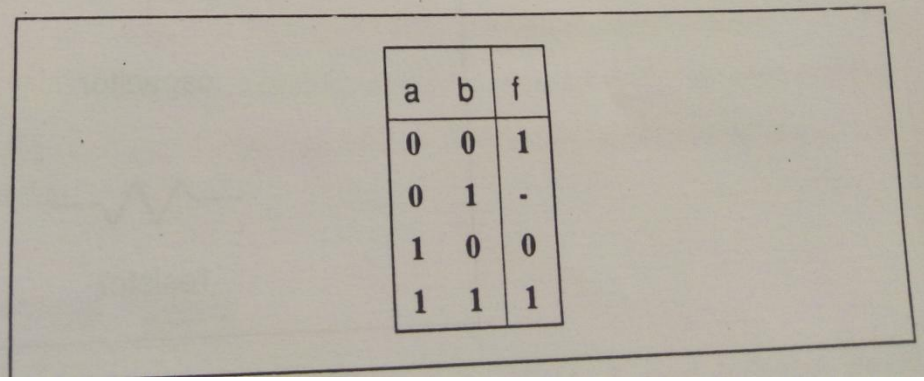
fully specified

a	b	f
0	-	1
1	0	0
1	1	1

with don't-cares

care to this term. (The complete logic function is $f = a' + ab$. You can verify this by drawing a Karnaugh map.)

Figure 1-3 An example of output don't-cares.



a	b	f
0	0	1
0	1	-
1	0	0
1	1	1

becomes a common factor. The opposite of factoring is collapsing; we collapse g into f when we rewrite f as $f = a' + ab'$.

1.2.2 Schematics and Logic Symbols

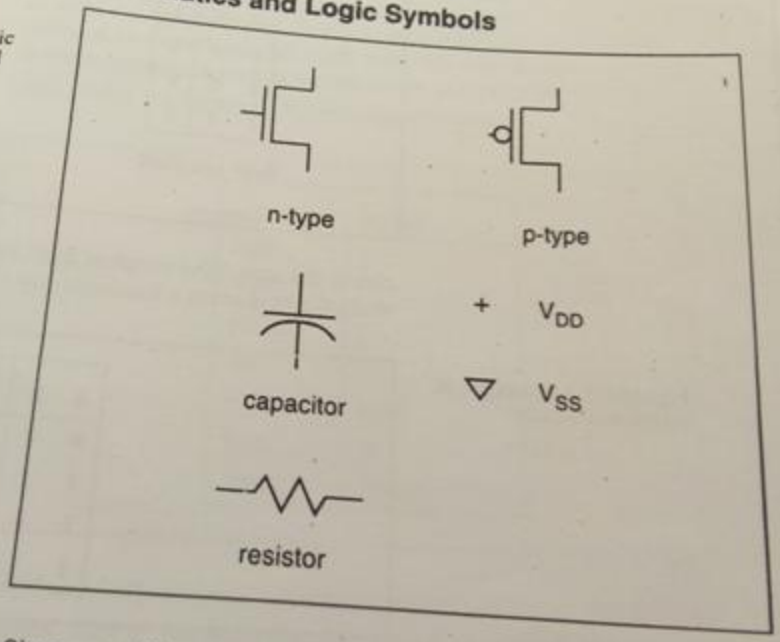


Figure 1-4 Schematic symbols for electrical components.

Since we will use a variety of schematic symbols let's briefly review them here. Figure 1-4 shows the symbols for some electrical components: n-type and p-type transistors, a capacitor, a resistor, and power supply connections (V_{DD}, the positive terminal and V_{SS}, the negative terminal).

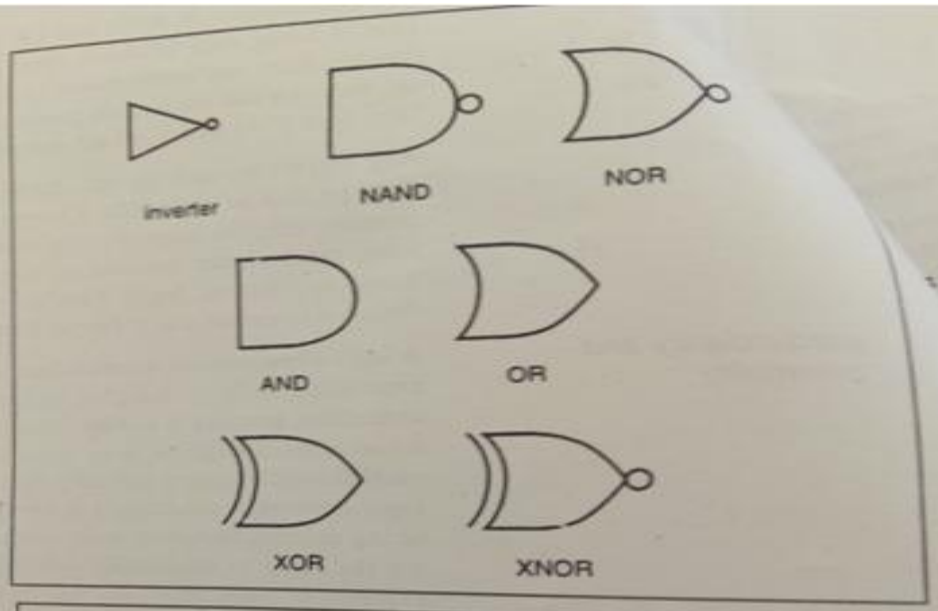


Figure 1-5 Schematic symbols for logic gates.

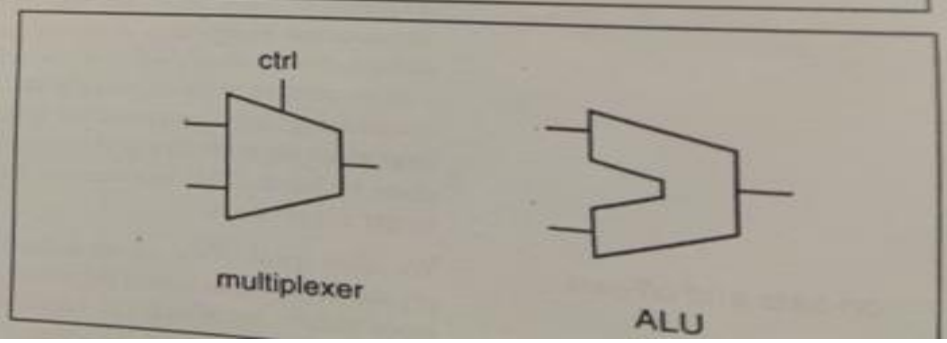


Figure 1-6 Schematic symbols for register-transfer components.

slower and more power-hungry than custom chips. Similarly, FPGAs are not custom parts, so they aren't as good at any particular function as a dedicated chip designed for that application. FPGAs are generally slower and burn more power than custom logic. FPGAs are also relatively expensive; it is often tempting to think that a custom-designed chip would be cheaper.

advantages of FPGAs

However, they have compensating advantages, largely due to the fact that they are standard parts.

- There is no wait from completing the design to obtaining a working chip. The design can be programmed into the FPGA and tested immediately.
- FPGAs are excellent prototyping vehicles. When the FPGA is used in the final design, the jump from prototype to product is much smaller and easier to negotiate.
- The same FPGA can be used in several different designs, reducing inventory costs.

The area filled by FPGAs has grown enormously in the past twenty years since their introduction. **Programmable logic devices (PLDs)** had been on the market since the early 1970s. These devices used two-level logic structures to implement programmed logic. The first level of logic, the AND plane, was generally fixed, while the second level, known as the OR plane, was programmable. PLDs are generally programmed by antifuses, which are programmed through large voltages to make connections.

They were most often used as **glue logic**—logic that was needed to connect together the major components of the system. They were often used

FPGAs have become mainstream devices for implementing digital systems.

1.3.2 FPGA Types

We have so far avoided defining the FPGA. A good definition should distinguish the FPGA on the one hand from smaller programmable devices like PLDs and on the other hand from custom chips. Here are some defining characteristics of FPGAs:

- **They are standard parts**. They are not designed for any particular function but are programmed by the customer for a particular purpose.
- **They implement multi-level logic**. The logic blocks inside FPGAs can be connected in networks of arbitrary depth. PLDs, in contrast, use two levels of NAND/NOR functions to implement all their logic.

Because FPGAs implement multi-level logic, they generally need bo programmable logic blocks and programmable interconnect. PLDs u fixed interconnect and simply change the logic functions attached to t wires. FPGAs, in contrast, require programming logic blocks and co

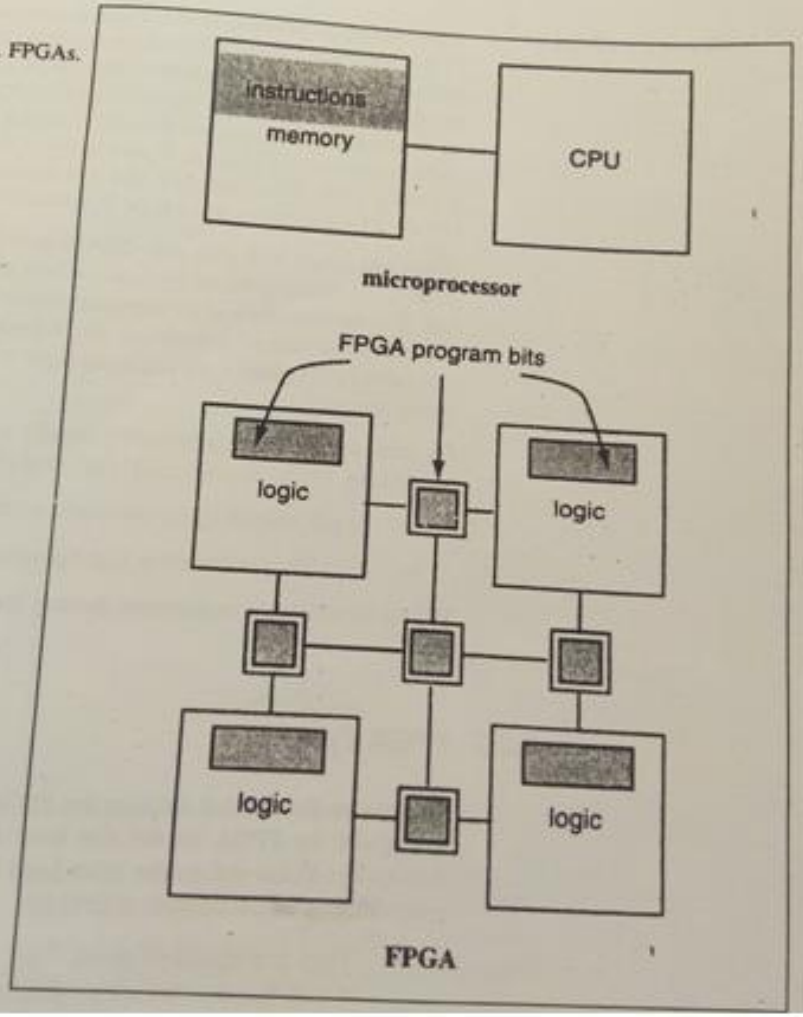


Figure 1-7 CPUs vs. FPGAs.

platform F

1.4 FPGA-Based System Design

1.4.1 Goals and Techniques

The logical function to be performed is only one of the goals that must be met by an FPGA or any digital system design. Many other attributes must be satisfied for the project to be successful:

- **Performance.** The logic must run at a required rate. Performance can be measured in several ways, such as throughput and latency. Clock rate is often used as a measure of performance.
- **Power/energy.** The chip must often run within an energy or power budget. Energy consumption is clearly critical in battery-powered systems. Even if the system is to run off the power grid, heat dissipation costs money and must be controlled.
- **Design time.** You can't take forever to design the system. FPGAs, because they are standard parts, have several advantages in design time. They can be used as prototypes, they can be programmed quickly, and they can be used as parts in the

- **Design cost.** Design time is one important component of design cost, but other factors, such as the required support tools, may be a consideration. FPGA tools are often less expensive than custom VLSI tools.
- **Manufacturing cost.** The manufacturing cost is the cost of replicating the system many times. FPGAs are generally more expensive than ASICs thanks to the overhead of programming. However, the fact that they are standard parts helps to reduce their cost.

design challenges

Design is particularly hard because we must solve several problems:

- **Multiple levels of abstraction**. FPGA design requires refining an idea through many levels of detail. Starting from a specification of what the chip must do, the designer must create an architecture which performs the required function, and then expand the architecture into a logic design.
- **Multiple and conflicting costs**. Costs may be in dollars, such as the expense of a particular piece of software needed to design some piece. Costs may also be in performance or power consumption of the final FPGA.
- **Short design time**. Electronics markets change extremely quickly. Getting a chip out faster means reducing your costs and increasing your revenue. Getting it out late may mean not making any money at all.

requirements and specifications

A design project may start out with varying amounts of information. Some projects are revisions of earlier designs; some are implementations of published standards. In these cases, the function is well-speci-

1.4.2 Hierarchical Design

Hierarchical design is a standard method for dealing with ... ital designs. It is commonly used in programming: a proce... ten not as a huge list of primitive statements but as calls ... procedures. Each procedure breaks down the task into sma... tions until each step is refined into a procedure simple enou... written directly. This technique is commonly known as **divide-an... quer**—the procedure's complexity is conquered by recursively b... it down into manageable pieces.

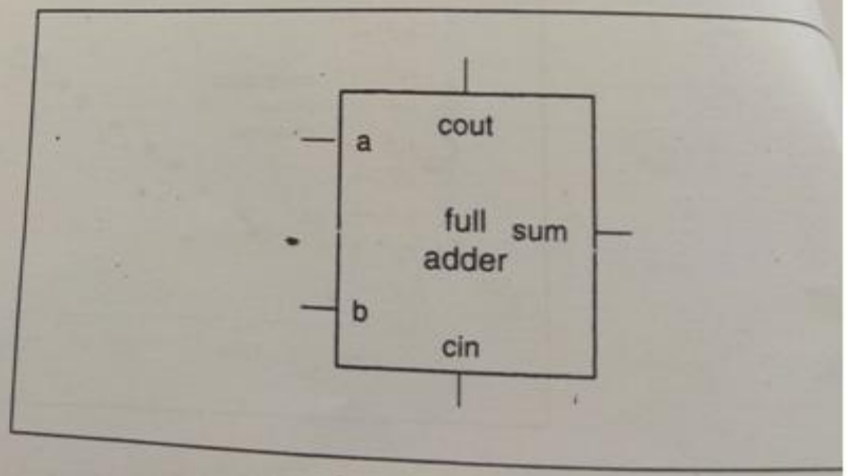


Figure 1-9 Pins on a component.

component types

Chip designers ...

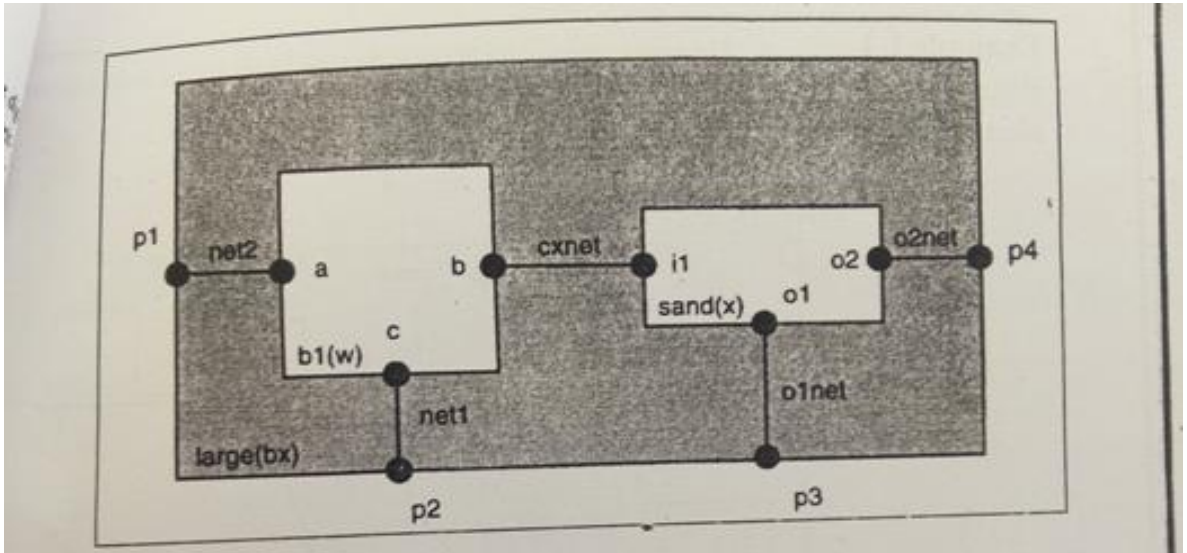
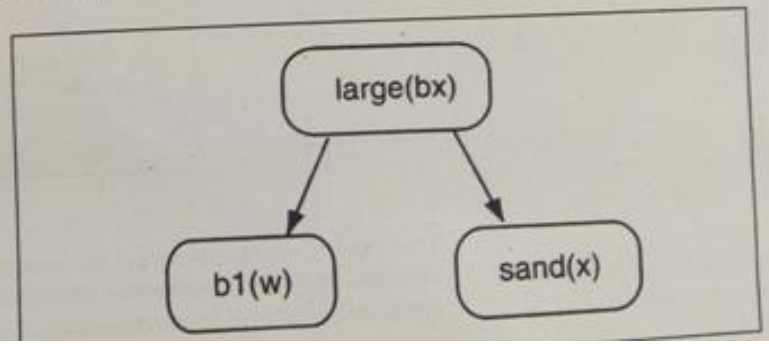


Figure 1-10 A hierarchical logic design.

Figure 1-11 A component hierarchy.



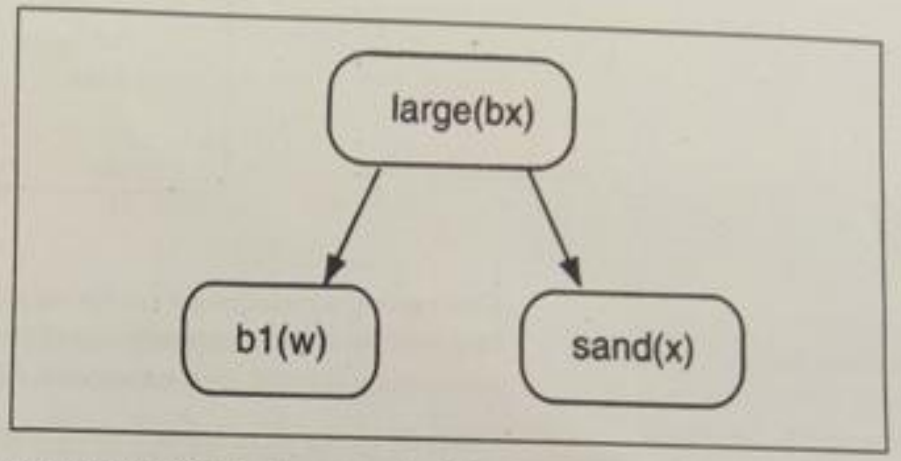


Figure 1-11 A component hierarchy.

connections which make up a circuit in either of two equivalent ways: a **net list** or a **component list**. A net list gives, for each net, the terminals connected to that net. Here is a net list for the top component of Figure 1-10:

```
net2: large.p1, b1.a;
net1: large.p2, b1.c;
cxnet: b1.b, sand.i1;
o1net: large.p3, sand.o1;
o2net: sand.o2, large.p4.
```

A component list gives, for each component, the net attached to each pin. Here is a component list version of the same circuit:

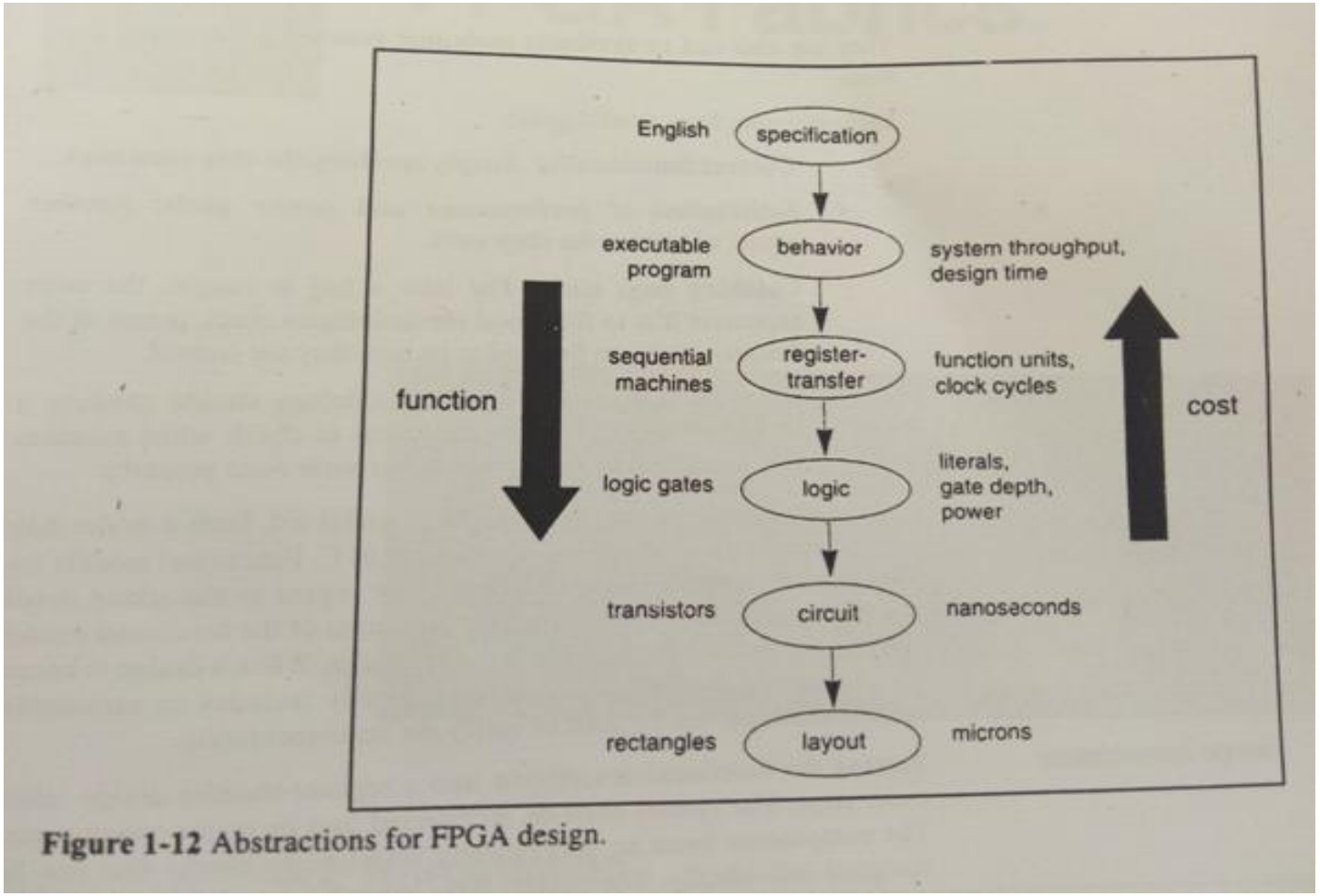


Figure 1-12 Abstractions for FPGA design.

FPGA abstractions

Figure 1-12 shows a design

- **Behavior.** A detailed, executable description of what the chip should do, but not how it should do it. A C program, for example, may be used as a behavioral description. The C program will not mimic the clock cycle-by-clock cycle behavior of the chip, but it will allow us to describe in detail what needs to be computed, error and boundary conditions, etc.
- **Register-transfer**. The system's time behavior is fully-specified—we know the allowed input and output values on every clock cycle—but the logic isn't specified as gates. The system is specified as Boolean functions stored in abstract memory elements. Only the vaguest delay and area estimates can be made from the Boolean logic functions.
- **Logic**. The system is designed in terms of Boolean logic gates, latches, and flip-flops. We know a lot about the structure of the system but still cannot make extremely accurate delay calculations.
- **Configuration**. The logic must be placed into logic elements around the FPGA and the proper connections must be made between those logic elements. Placement and routing perform these important steps.

top-down and bottom-up design

Design always requires working down from the top of the abstraction hierarchy and up from the least abstract description. Obviously, you must begin by adding detail to the abstraction—**top-down** design functional detail. But top-down

tractions for FPGA design.

1.4.4 Methodologies

Complex problems can be tackled using **methodologies**. We can use the lessons we learn on one system to help us design the next system. Digital system design methodologies have been built up over several decades of experience. A methodology provides us with a set of guidelines for what to do, when to do it, and how to know when we are done. Some aspects of FPGA design methodologies are unique to FPGAs, but many aspects are generic to all digital systems.

ion

Modern digital designers rely on **hardware description languages** (**HDLs**) to describe digital systems. Schematics are rarely used to describe logic; block diagrams are often drawn at higher levels of the system hierarchy but usually as documentation, not as design input. HDLs are tied to **simulators** that understand the semantics of hardware.

They are also tied to **synthesis tools** that generate logic implementations.

Methodologies have several goals:

- **Correct functionality**. Simply speaking, the chip must work.
- **Satisfaction of performance and power goals**. Another aspect of making the chip work.
- **Catching bugs early**. The later a bug is caught, the more expensive it is to fix. Good methodologies check pieces of the design as they are finished to be sure they are correct.
- **Good documentation**. The methodology should produce a paper trail that allows the designers to check when mistakes were made and to ensure that things were done properly.

A functional model of some sort is often a useful aid. Such a model may be written in a programming language such as C. Functional models are useful because they can be written without regard to the added detail needed to create a functional design. The output of the functional model can be used to check the register-transfer design. When a design is based on an existing standard, the standard typically includes an executable specification that can be used to verify the implementation.

3.1 Introduction

In this chapter we will study the basic structures of FPGAs, known as **fabrics**. We will start with a brief introduction to the structure of FPGA fabrics. However, there are several fundamentally different ways to build an FPGA. Therefore, we will discuss combinational logic and interconnect for the two major styles of FPGA: SRAM-based and antifuse-based. The features of I/O pins are fairly similar among these two types of FPGAs, so we will discuss pins at the end of the chapter.

3.2 FPGA Architectures

ts of FPGAs

In general, FPGAs require three major types of elements:

- combinational logic;
- interconnect;
- I/O pins.

These three elements are mixed together to form an **FPGA fabric**.

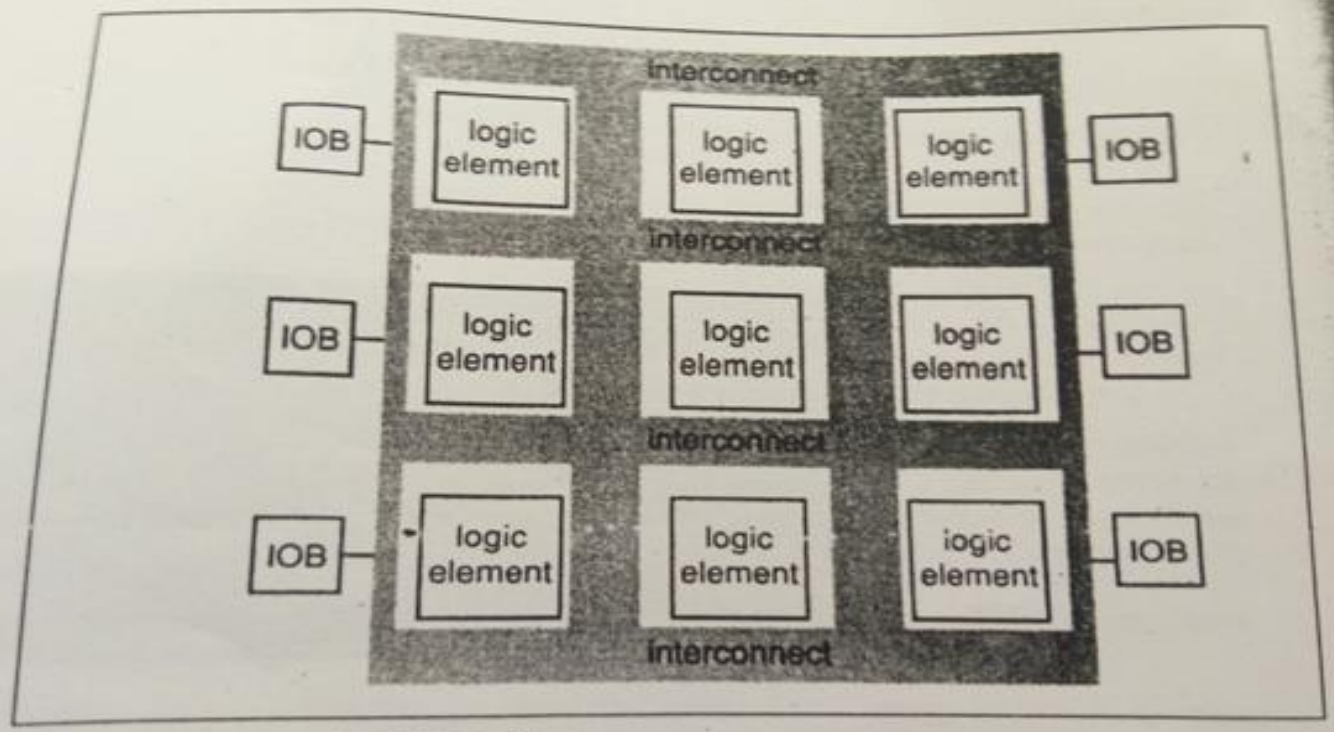


Figure 3-1 *Generic structure of an FPGA fabric.*

FPGA architectures

Figure 3-1 shows the basic structure of an FPGA that incorporates these three elements. The combinational logic is divided into relatively small units which may be known as **logic elements (LEs)** or **combinational**

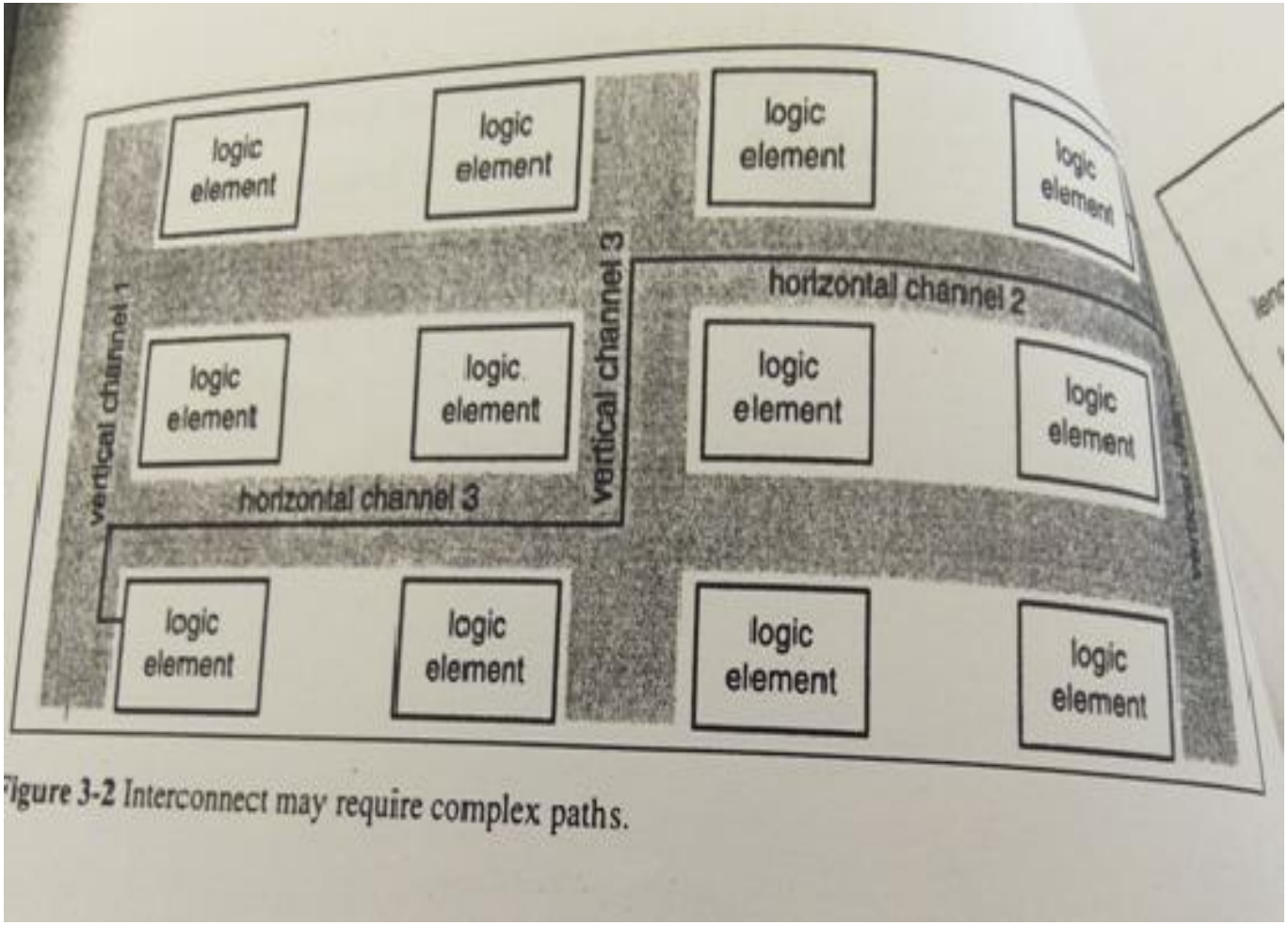


Figure 3-2 Interconnect may require complex paths.

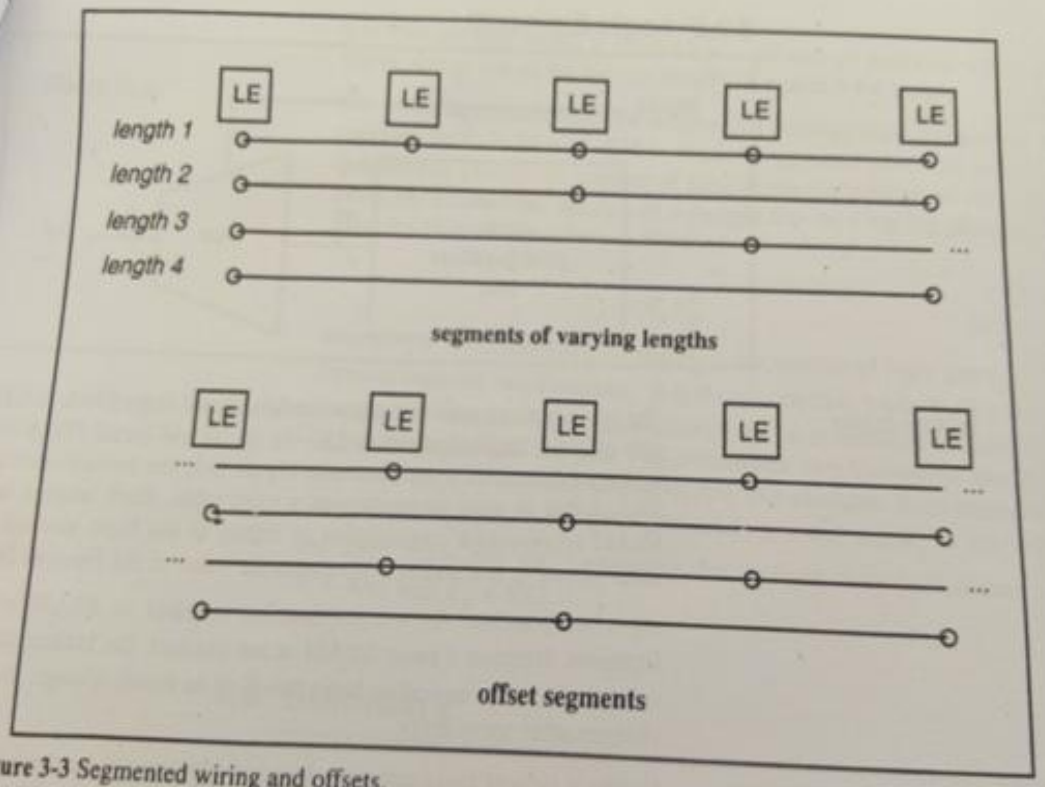


Figure 3-3 Segmented wiring and offsets.

known as a **segmented wiring**

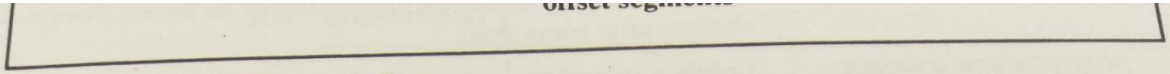


Figure 3-3 Segmented wiring and offsets.

known as a **segmented wiring** structure [ElG88] since the wiring is constructed of segments of varying lengths. The alternative to segmented wiring is to make each wire length 1. However, this would require a long connection to hop through many programmable wiring points, and as we will see in Section 3.6, that would lead to excessive delay along the connection. The segments in a group need not all end at the same point. The bottom part of Figure 3-3 shows segments of length 2 that are offset relative to each other.

FPGA configuration

All FPGAs need to be **programmed** or **configured**. There are three major circuit technologies for configuring an FPGA: SRAM, antifuse, and flash. No matter what circuits are used, all the major elements of the FPGA—the logic, the interconnect, and the I/O pins—need to be configured. The details of these elements vary greatly depending on how the FPGA elements are to be programmed. But FPGAs are very complex VLSI systems that can be characterized in many different ways.

Some of the characteristics of interest to the system designer who wants to use an FPGA include:

- How much logic can I fit into this FPGA?
- How many I/O pins does it have?
- How fast does it run?

While we can determine fairly easily how many I/O pins an FPGA has, determining how much logic can be fit into it and how fast that logic will run is not simple. As we will see in this chapter, the complex architecture of an FPGA means that we must carefully optimize the logic as we fit it into the FPGA. The amount of logic we can fit and how fast that logic runs depends on many characteristics of the FPGA architecture, the logic itself, and the logic design process. We'll look at the tools necessary to configure an FPGA in Chapter 4.

design of FPGA architectures

Some questions of interest to the person who designs the FPGA itself include:

- How many logic elements should the FPGA have?
- How large should each logic element be?
- How much interconnect should it have?
- How many types of interconnection structures should it have?
- How long should each type of interconnect be?
- How many pins should it have?

In Section 3.6 and Section 3.7 we will survey some of the extensive research results in the design of FPGA fabrics. A great deal of theory and experimentation has been developed to determine the parameters for FPGA architectures that best match the characteristics of typical logic that is destined for FPGA implementation.