Logical Effort:
Designing Fast CMOS Circuits

Ivan E. Sutherland
Bob F. Sproull
David L. Harris

DRAFT of May 19, 1998

Contents

Foreword

This monograph presents a method for designing MOS circuits to achieve high speed. The method is based on a simple approximation that treats MOS circuits as networks of resistance and capacitance. This RC model is formulated in a new way to simplify the mathematics and to determine quickly a circuit's maximum possible speed and how to achieve it.

The method of logical effort is a way of thinking about delay in MOS circuits, which introduces two new names for concepts that are not new: "logical effort" and "electrical effort." Logical effort describes the cost of computation due to the circuit topology required to implement a logic function, while electrical effort describes the cost of driving large capacitive loads. We have chosen related names for these two ideas to invite a comparison between them, because the two forms of effort present identical and interchangeable sources of delay. Providing new names for these concepts leads to a formulation that simplifies circuit analysis and allows a designer to analyze alternative circuit designs quickly.

Critics of this method observe that it achieves no more than conventional RC analysis and that experienced designers know how to optimize circuits for speed. Indeed, the best designers, whether by intuition or experience, design circuits that match closely those derived by the method of logical effort. But we have seen many instances where experienced designers devise poor circuits. They often become mired in detailed circuit simulations and transistor sizing and fail to study structural changes to a circuit that lead to greater performance improvements. Because of its simplicity, the method of logical effort bridges the gap between structural design and detailed simulation.

This monograph is intended for anyone who designs MOS integrated circuits. It assumes a knowledge of static CMOS digital circuits, elementary electronics, and algebra. (Some of the derivations use calculus, but only algebra is required to apply the method.) The novice designer will find simple techniques for designing high-speed circuits. The experienced designer will find new ways to think about old design techniques and rules of thumb that lead to high-speed circuits. He or she will find that the techniques of logical effort help to analyze and optimize large circuits quickly.

The method of logical effort developed in three stages. It began with research on fast asynchronous circuits by Ivan and Bob. The circuits were sufficiently large and complex that the conventional RC model and our intuition did not lead to the best designs. However, the symmetry of CMOS circuits, and especially the forms

that occur in asynchronous designs, led us to compare them to simple inverters, which are also symmetric: the equations of logical effort followed naturally. The method, of course, applies more broadly; the examples in this monograph are mostly combinational or synchronous. The research resulted in a paper on logical effort [7] and a set of course notes that form the core of this monograph.

Years later, David faced the problem of teaching junior circuit designers and graduate students at Stanford University how to design paths and size gates. Teaching is often the best way to learn; he was forced to develop coherent explanations for an intuitive approach to sizing. The explanations proved to be a rediscovery of logical effort, suggesting that logical effort is a very natural way to think about delays. He gradually discovered more properties of circuits, especially regarding the logical effort of other circuit families such as domino, and applied the principles of logical effort to design various arithmetic units. Finally he met Ivan and found that many of the results he had obtained were already in the unpublished course notes. Moreover, he and his students wanted a good reference text for logical effort, so he undertook the task of polishing the course notes into this form.

There are many ways to use this text. Chapter 1 stands alone as an introduction to logical effort. A course on VLSI design may use the first four chapters to provide examples of applying logical effort and to develop the basic theory behind the method. Experienced circuit designers and students in advanced circuit design classes will be interested in the later chapters, which apply logical effort to common circuit design problems. The conclusion includes a concise review of the method of logical effort and of important insights from the method.

Many people have helped us to develop the method of logical effort and to prepare this monograph. We wish to thank five companies that sponsored the original research: Austek Microsystems, Digital Equipment Corporation, Evans and Sutherland Computer Corporation, Floating Point Systems, and Schlumberger. We are grateful to Apple Computer for support for beginning to edit our course notes. We also thank the engineers and designers from these firms who served as students during our early attempts to teach this material and whose penetrating questions contributed to a clearer presentation of the ideas. We thank Carnegie Mellon University, Stanford University, and the Imperial College of London University for the office space, computing support, and collegial thinking they have provided. Our colleagues Ian W. Jones, Erik L. Brunvand, Bob Proebsting, Mark Horowitz, and Peter Single contributed in several ways to the work. More recently, we thank our students at Stanford University, HAL Computer, and Intel Corporation for bringing fresh life and interest to logical effort. Sally Harris worked tirelessly to prepare illustrations and text. Finally, we offer special thanks to our

friends and colleagues Bob Spence and the late Charles Molnar for ideas, encouragement, and moral support.

Ivan E. Sutherland
Bob F. Sproull
David Harris
February 1998

Chapter 1

The Method of Logical Effort

Designing a circuit to achieve the greatest speed or to meet a delay constraint presents a bewildering array of choices. Which of several circuits that produce the same logic function will be fastest? How large should a logic gate's transistors be to achieve least delay? And how many stages of logic should be used to obtain least delay? Sometimes, adding stages to a path reduces its delay!

The *method of logical effort* is an easy way to estimate the delay in an MOS circuit. By comparing delay estimates of different logic structures, the fastest candidate can be selected. The method also specifies the proper number of logic stages on a path and the best transistor sizes for the logic gates. Because the method is easy to use, it is ideal for evaluating alternatives in the early stages of a design and provides a good starting point for more intricate optimizations.

This chapter describes the method of logical effort and applies it to simple examples. Chapter 2 explores more complex examples. These two chapters together provide all you need to know to apply the method of logical effort to a wide class of circuits. The remainder of this monograph is devoted to derivations that show why the method of logical effort works, to some detailed optimization techniques, and to the analysis of special circuits such as domino logic and multiplexers.

1.1 Delay in a logic gate

The method of logical effort is founded on a simple model of the delay through a single MOS logic gate.[1] The model describes delays caused by the capacitive load that the logic gate drives and by the topology of the logic gate. Clearly, as the load increases, the delay increases, but delay also depends on the logic function of the gate. Inverters, the simplest logic gates, drive loads best and are often used as amplifiers to drive large capacitances. Logic gates that compute other functions require more transistors, some of which are connected in series, making them poorer than inverters at driving current. Thus a NAND gate must have more delay than an inverter with similar transistor sizes that drives the same load. The method of logical effort quantifies these effects to simplify delay analysis for individual logic gates and multi-stage logic networks.

The first step in modeling delays is to isolate the effects of a particular integrated-circuit fabrication process by expressing all delays in terms of a basic *delay unit*[2], τ. Thus we express absolute delay as the product of a unitless delay of the gate, d, and the delay unit that characterizes a given process:

$$d_{abs} = d\tau \tag{1.1}$$

Unless otherwise indicated, we will measure all times in units of τ. τ is about 50 ps in a typical 0.6μ process.

The delay incurred by a logic gate is comprised of two components, a fixed part called the *parasitic delay*, p, and a part that is proportional to the load on the gate's output, called the *effort delay* or *stage effort*, f. The total delay, measured in units of τ, is the sum of the effort and parasitic delays:

$$d = f + p \tag{1.2}$$

The effort delay depends on the load and on properties of the logic gate driving the load. We introduce two related terms for these effects: the *logical effort*, g, captures properties of the logic gate, while the *electrical effort*, h, characterizes the load. The effort delay of the logic gate is the product of these two factors:

$$f = gh \tag{1.3}$$

[1] The term "gate" is ambiguous in integrated-circuit design, signifying either a circuit that implements a logic function such as NAND or the gate of an MOS transistor. We hope to avoid confusion by referring to "logic gate" or "transistor gate" unless the meaning is clear from context.

[2] This definition of τ differs from that used by Mead & Conway [6].

The logical effort, g, captures the effect of the logic gate's topology on its ability to produce output current. It is independent of the size of the transistors in the circuit. The electrical effort, h, describes how the electrical environment of the logic gate affects performance and how the size of the transistors in the gate determines its load-driving capability. The electrical effort is defined by:

$$h = C_{out}/C_{in} \tag{1.4}$$

where C_{out} is the capacitance that loads the logic gate and C_{in} is the capacitance presented by the logic gate at one of its input terminals. Electrical effort is also called fanout by many CMOS designers.[3]

Combining Equations 1.2 and 1.3, we obtain the basic equation that models the delay through a single logic gate, in units of τ:

$$d = gh + p \tag{1.5}$$

This equation shows that logical effort g and electrical effort h both contribute to delay in the same way. This formulation separates τ, g, h, and p, the four contributions to delay. The process parameter τ represents the speed of the basic transistors. The parasitic delay, p, expresses the intrinsic delay of the gate due to its own internal capacitance, which is largely independent of the size of the transistors in the logic gate. The electrical effort, h, combines the effects of external load, which establishes C_{out}, with the sizes of the transistors in the logic gate, which establish C_{in}. The logical effort, g, expresses the effects of circuit topology on the delay free of considerations of loading or transistor size. Logical effort is useful because it depends only on circuit topology.

Logical effort values for a few CMOS logic gates are shown in Table 1.1. Logical effort is defined so that an inverter has a logical effort of one. This unitless form means that all delays are measured relative to the delay of a simple inverter. An inverter driving an exact copy of itself experiences an electrical effort of one. Because the logical effort of an inverter is defined to be one, an inverter driving an exact copy of itself will therefore have an effort delay of one, according to Equation 1.3.

The logical effort of a logic gate tells how much worse it is at producing output current than is an inverter, given that each of its inputs may contain only the same input capacitance as the inverter. Reduced output current means slower operation,

[3]Fanout, in this context, depends on the load capacitance, not just the number of gates being driven.

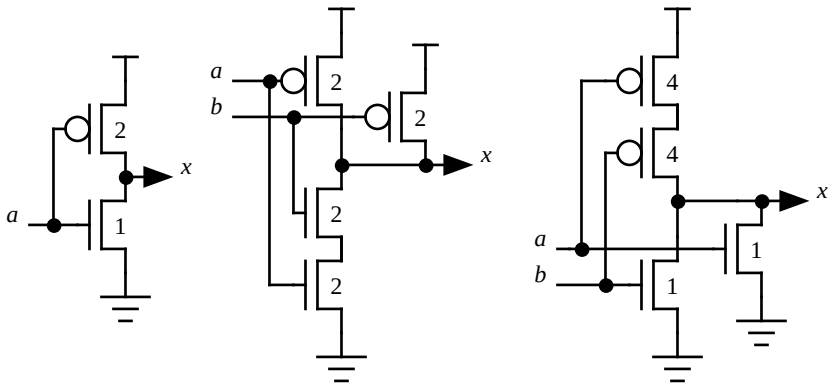
Gate type	Number of inputs					
	1	2	3	4	5	n
inverter	1					
NAND		$4/3$	$5/3$	$6/3$	$7/3$	$(n+2)/3$
NOR		$5/3$	$7/3$	$9/3$	$11/3$	$(2n+1)/3$
multiplexer		2	2	2	2	2
XOR (parity)		4	12	32		

Table 1.1: Logical effort for inputs of static CMOS gates, assuming $\gamma = 2$. γ is the ratio of an inverter's pullup transistor width to pulldown transistor width. Chapter 4 explains how to calculate the logical effort of these and other logic gates.

and thus the logical effort number for a logic gate tells how much more slowly it will drive a load than an inverter would. Equivalently, logical effort is how much more input capacitance a gate presents to deliver the same output current as an inverter. Figure 1.1 illustrates simple gates sized for roughly equal output currents. From the ratio of input capacitances, one can see that the NAND gate has logical effort $g = 4/3$ and the NOR gate has logical effort $g = 5/3$. Chapter 4 estimates the logical effort of other gates, while Chapter 5 shows how to extract logical effort from circuit simulations.

It is interesting but not surprising to note from Table 1.1 that more complex logic functions have larger logical effort. Moreover, the logical effort of most logic gates grows with the number of inputs to the gate. Larger or more complex logic gates will thus exhibit greater delay. As we shall see later on, these properties make it worthwhile to contrast different choices of logical structure. Designs that minimize the number of stages of logic will require more inputs for each logic gate and thus have larger logical effort. Designs with fewer inputs and thus less logical effort per stage may require more stages of logic. In Section 1.3, we will see how the method of logical effort expresses these tradeoffs.

The electrical effort, h, is just a ratio of two capacitances. The load driven by a logic gate is the capacitance of whatever is connected to its output; any such load will slow down the circuit. The input capacitance of the circuit is a measure of the size of its transistors. The input capacitance term appears in the denominator of Equation 1.4 because bigger transistors in a logic gate will drive a given load faster. Usually most of the load on a stage of logic is the capacitance of the input or inputs of the next stage or stages of logic that it drives. Of course, the load also includes the stray capacitance of wires, drain regions of transistors, etc. We shall



(*a*) (*b*) (*c*)

Figure 1.1: Simple gates. (a) Inverter. (b) Two-input NAND gate. (c) Two-input NOR gate.

Gate type	Parasitic delay
inverter	p_{inv}
n-input NAND	np_{inv}
n-input NOR	np_{inv}
n-way multiplexer	$2np_{inv}$
XOR, XNOR	$4p_{inv}$

Table 1.2: Estimates of parasitic delay of various logic gate types, assuming simple layout styles. A typical value of p_{inv}, the parasitic delay of an inverter, is 1.0.

see later how to include stray load capacitances in our calculations.

Electrical effort is usually expressed as a ratio of transistor widths rather than actual capacitances. We know that the capacitance of a transistor gate is proportional to its area; if we assume that all transistors have the same minimum length, then the capacitance of a transistor gate is proportional to its width. Because most logic gates drive other logic gates, we can express both C_{in} and C_{out} in terms of transistor widths. If the load capacitance includes stray capacitance due to wiring or external loads, we shall convert this capacitance into an equivalent transistor width. If you prefer, you can think of the unit of capacitance as the capacitance of a transistor gate of minimum length and 1 unit width.

The parasitic delay of a logic gate is fixed, independent of the size of the logic

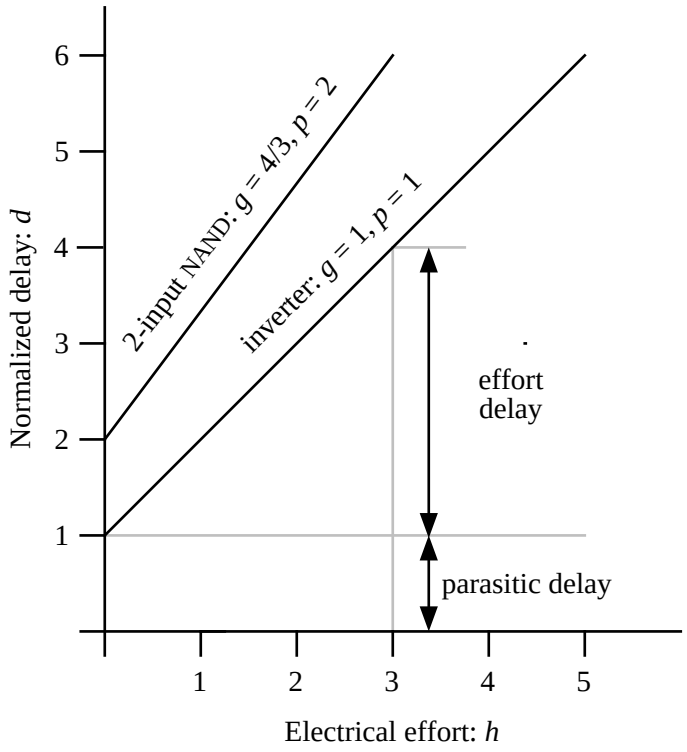


Figure 1.2: Plots of the delay equation for an inverter and aa two-input NAND gate.

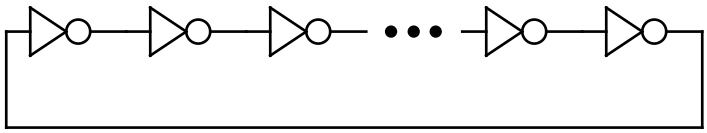


Figure 1.3: A ring oscillator of N identical inverters.

gate and of the load capacitance it drives, because wider transistors have correspondingly greater diffusion capacitance. This delay is a form of overhead that accompanies any gate. The principal contribution to parasitic delay is the capacitance of the source/drain regions of the transistors that drive the gate's output. Table 1.2 presents estimates of parasitic delay for a few logic gate types; note that parasitic delays are given as multiples of the parasitic delay of an inverter, denoted as p_{inv}. A typical value for p_{inv} is 1.0 delay units[4], which is used in most of the examples in this book. Parasitic delay is covered in more detail in Chapters 3 and 5.

The delay model of a single logic gate, as represented in Equation 1.5, is a simple linear relationship. Figure 1.2 shows this relationship graphically: delay is plotted as a function of electrical effort for an inverter and for a 2-input NAND gate. The slope of the line is the logical effort of the gate; its intercept is the parasitic delay. The graph shows that we can adjust the delay by adjusting the electrical effort or by choosing a logic gate with a different logical effort. Once we have chosen a gate type, however, the parasitic delay is fixed, and our optimization procedure can do nothing to reduce it.

Example 1.1 *Estimate the delay of an inverter driving an identical inverter, as in the ring oscillator shown in Figure 1.3.*

Because the inverter's output is connected to the input of an identical inverter, the load capacitance, C_{out}, is the same as the input capacitance. Therefore the electrical effort is $h = C_{out}/C_{in} = 1$. Because the logical effort of an inverter is 1, we have, from Equation 1.5, $d = gh + p = 1 \times 1 + p_{inv} = 2.0$. This result expresses the delay in *delay units*; it can be scaled by τ to obtain the absolute delay, $d_{abs} = 2.0\tau$. In a 0.6μ process with $\tau = 50$ps, $d_{abs} = 100$ps.

[4] p_{inv} is a strong function of process-dependent diffusion capacitances, but 1.0 is representative and is convenient for hand analysis.

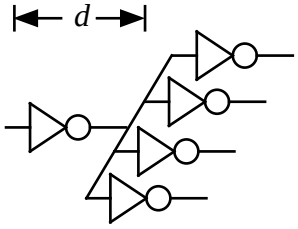


Figure 1.4: An inverter driving four identical inverters.

The ring oscillator shown in Figure 1.3 can be used to measure the value of τ. Because N, the number of stages in the ring, is odd, the circuit is unstable and will oscillate. The delay of each stage of the ring oscillator is expressed by:

$$\frac{1}{2NF} = d\tau = (1 + p_{inv})\tau \tag{1.6}$$

where N is the number of inverters, F is the oscillation frequency, and the 2 appears because a transition must pass twice around the ring to complete a single cycle of the oscillation. If a value for p_{inv} is known, this equation can be used to determine τ from measurements of the frequency of the ring oscillator. Chapter 5 shows a method in which both τ and p_{inv} can be measured.

Example 1.2 *Estimate the delay of a fanout-of-4 (FO4) inverter, as shown in Figure 1.4.*

Because each inverter is identical, $C_{out} = 4C_{in}$, so $h = 4$. The logical effort $g = 1$ for an inverter. Thus the FO4 delay, according to Equation 1.5, is $d = gh + p = 1 \times 4 + p_{inv} = 4 + 1 = 5$. It is sometimes convenient to express times in terms of FO4 delays because most designers know the FO4 delay in their process and can use it to estimate the absolute performance of your circuit in their process.

Example 1.3 *A four-input* NOR *gate drives ten identical gates, as shown in Figure 1.5. What is the delay in the driving* NOR *gate?*

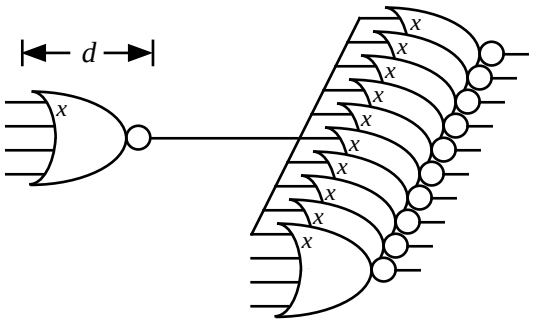


Figure 1.5: A four-input NOR gate driving ten identical gates.

If the capacitance of one input of each NOR gate is x, then the driving NOR has $C_{in} = x$ and $C_{out} = 10x$, and thus the electrical effort is $h = 10$. The logical effort of the four-input NOR gate is $9/3 = 3$, obtained from Table 1.1. Thus the delay is $d = gh + p = 3 \times 10 + 4 \times 1$ or 34 delay units. Note that when the load is large, as in this example, the parasitic delay is insignificant compared to the effort delay.

1.2 Multi-stage logic networks

The method of logical effort is applied in two ways to design fast multi-stage logic networks. It reveals the best number of stages to use in the network and it shows how to get least overall delay by balancing the delay among the stages. The notions of logical and electrical effort generalize easily to multi-stage networks.

The logical effort along a path compounds by multiplying the logical efforts of all the logic gates along the path. We use the upper-case symbol G to denote the *path logical effort*, so that it is distinguished from g, the logical effort of a single gate in the path.

$$G = \prod g_i \tag{1.7}$$

The electrical effort along a path through a network is simply the ratio of the capacitance that loads the last logic gate in the path to the input capacitance of the first gate in the path. We use an upper-case symbol, H, to indicate the electrical effort along a path.

$$H = C_{out}/C_{in} \tag{1.8}$$

In this case, C_{in} and C_{out} refer to the input and output capacitances of the path as a whole, as may be inferred from context.

We need to introduce a new kind of effort, named *branching effort*, to account for fanout within a network. So far we have treated fanout as a form of electrical effort: when a logic gate drives several loads, we sum their capacitances, as in Example 1.3, to obtain an electrical effort. Treating fanout as a form of electrical effort is easy when the fanout occurs at the final output of a network. This method is less suitable when the fanout occurs within a logic network because we know that the electrical effort for the network depends only on the ratio of its output capacitance to its input capacitance.

When fanout occurs within a logic network, some of the available drive current is directed along the path we are analyzing, and some is directed off the path. We define the branching effort b at the output of a logic gate to be:

$$b = \frac{C_{on-path} + C_{off-path}}{C_{on-path}} \tag{1.9}$$

where $C_{on-path}$ is the load capacitance along the path we are analyzing and $C_{off-path}$ is the capacitance of connections that lead off the path. Note that if the path does not branch, the branching effort is one. The branching effort along an entire path, B, is the product of the branching effort at each of the stages along the path.

$$B = \prod b_i \tag{1.10}$$

Armed with definitions of logical, electrical, and branching effort along a path, we can define the *path effort*, F. Again, we use an upper-case symbol to distinguish the path effort from the stage effort, f, associated with a single logic stage. The equation that defines path effort is reminiscent of Equation 1.3, which defines the effort for a single logic gate:

$$F = GBH \tag{1.11}$$

Note that the path branching and electrical efforts are related to the electrical effort of each stage:

$$BH = \frac{C_{out}}{C_{in}} \prod b_i = \prod h_i \tag{1.12}$$

The designer knows C_{in}, C_{out}, and branching efforts b_i from the path specification. Sizing the path consists of choosing appropriate electrical efforts h_i for each stage to match the total BH product.

Although it is not a direct measure of delay along the path, the path effort holds the key to minimizing the delay. Observe that the path effort depends only on the circuit topology and loading and not upon the sizes of the transistors used in logic gates embedded within the network. Moreover, the effort is unchanged if inverters are added to or removed from the path, because the logical effort of an inverter is one. The path effort is related to the minimum achievable delay along the path, and permits us to calculate that delay easily. Only a little more work yields the best number of stages and the proper transistor sizes to realize the minimum delay.

The path delay, D, is the sum of the delays of each of the stages of logic in the path. As in the expression for delay in a single stage (Equation 1.5), we shall distinguish the *path effort delay,* D_F, and the *path parasitic delay,* P:

$$D = \sum d_i = D_F + P \tag{1.13}$$

where the subscripts index the logic stages along the path. The path effort delay is simply

$$D_F = \sum g_i h_i \tag{1.14}$$

and the path parasitic delay is

$$P = \sum p_i \tag{1.15}$$

Optimizing the design of an N-stage logic network proceeds from a very simple principle which we will prove in Chapter 3: *The path delay is least when each stage in the path bears the same stage effort.* This minimum delay is achieved when the stage effort is

$$\hat{f} = g_i h_i = F^{1/N} \tag{1.16}$$

We use a hat over a symbol to indicate an expression that achieves minimum delay.

Combining these equations, we obtain the principal result of the method of logical effort, which is an expression for the minimum delay achievable along a path:

$$\hat{D} = NF^{1/N} + P \tag{1.17}$$

From a simple computation of its logical, branching, and electrical efforts we can obtain an estimate of the minimum delay of a logic network. Observe that when $N = 1$, this equation reduces to Equation 1.5.

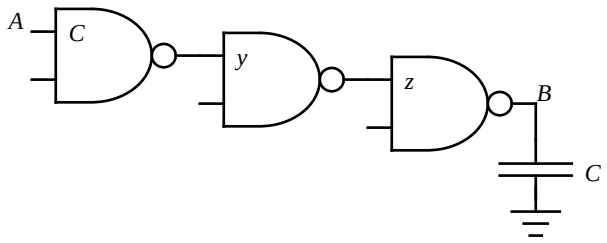


Figure 1.6: A logic network consisting of three two-input NAND gates.

To equalize the effort borne by each stage on a path, and therefore achieve the minimum delay along the path, we must choose appropriate transistor sizes for each stage of logic along the path. Equations 1.16 and 1.3 combine to require that each logic stage be designed so that

$$\hat{h}_i = F^{1/N}/g_i \tag{1.18}$$

From this relationship, we can determine the transistor sizes of gates along a path. Start at the end of the path and work backward, applying the capacitance transformation:

$$C_{in-i} = \frac{C_{out-i} \times g_i}{\hat{f}} \tag{1.19}$$

This determines the input capacitance of each gate, which can then be distributed appropriately among the transistors connected to the input. The mechanics of this process will become clear in the following examples.

Example 1.4 *Consider the path from A to B involving three two-input* NAND *gates shown in Figure 1.6. The input capacitance of the first gate is C and the load capacitance is also C. What is the least delay of this path and how should the transistors be sized to achieve least delay? (The next example will use the same circuit with a different electrical effort.)*

To compute the path effort, we must compute the logical, branching, and electrical efforts along the path. The path logical effort is the product of the logical efforts of the three NAND gates, $G = g_0 g_1 g_2 = 4/3 \times 4/3 \times 4/3 = (4/3)^3 = 2.37$. The branching effort is $B = 1$, because all of the fanouts along the path are one, i.e., there is no branching. The electrical effort is $H = C/C = 1$. Hence, the path effort is $F = GBH = 2.37$. Using Equation 1.17, we find the least delay

achievable along the path to be $\hat{D} = 3(2.37^{1/3}) + 3(2p_{inv}) = 10.0$ delay units.

This minimum delay can be realized if the transistor sizes in each logic gate are chosen properly. First compute the stage effort $\hat{f} = 2.37^{1/3} = 4/3$. Starting with the output load C, apply the capacitance transformation of Equation 1.19 to compute input capacitance $z = C \times (4/3)/(4/3) = C$. Similarly, $y = z \times (4/3)/(4/3) = z = C$. Hence we find that all three NAND gates should have the same input capacitance, C. In other words, the transistor sizes in the three gates will be the same. This is not a surprising result: all stages have the same load and the same logical effort, and hence bear equal effort, which is the condition for minimizing path delay.

Example 1.5 *Using the same network as in the previous example, Figure 1.6, find the least delay achievable along the path from A to B when the output capacitance is $8C$.*

Using the result from Example 1.4 that $G = (4/3)^3$ and the new electrical effort $H = 8C/C = 8$, we compute $F = GBH = (4/3)^3 \times 8 = 18.96$, so the least path delay is $\hat{D} = 3(18.96)^{1/3} + 3(2p_{inv}) = 14.0$ delay units. Observe that although the electrical effort in this example is eight times the electrical effort in the earlier example, the delay is increased by only 40%.

Now let us compute the transistor sizes that achieve minimum delay. The stage effort $\hat{f} = 18.96^{1/3} = 8/3$. Starting with the output load $8C$, apply the capacitance transformation of Equation 1.19 to compute input capacitance $z = 8C \times (4/3)/(8/3) = 4C$. Similarly, $y = z \times (4/3)/(8/3) = z/2 = 2C$. To verify the calculation, calculate the capacitance of the first gate $y \times (4/3)/(8/3) = z/2 = C$, matching the design specification. Each successive logic gate has twice the input capacitance of its predecessor. This is achieved by making the transistors in a gate twice as wide as the corresponding transistors in its predecessor. The wider transistors in successive stages are better able to drive current into the larger loads.

Example 1.6 *Optimize the circuit in Figure 1.7 to obtain the least delay along the path from A to B when the electrical effort is 4.5.*

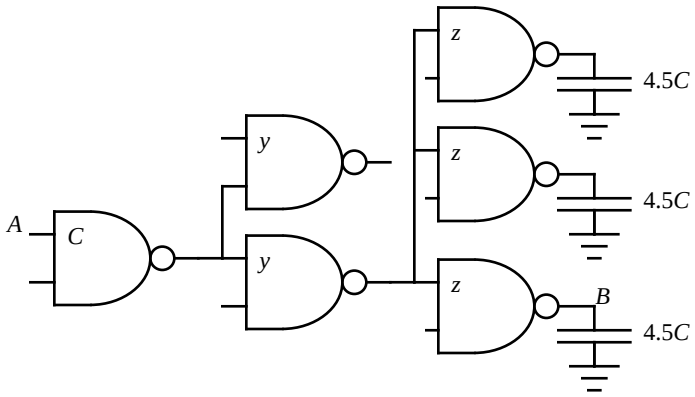


Figure 1.7: A multi-stage logic network with internal fanout.

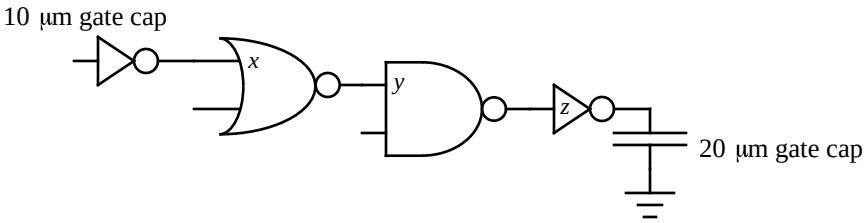


Figure 1.8: A multi-stage logic network with a variety of gates.

The path logical effort is $G = (4/3)^3$. The branching effort at the output of the first stage is $(y + y)/y = 2$, and at the output of the second stage it is $(z + z + z)/z = 3$. The path branching effort is therefore $B = 2 \times 3 = 6$. The electrical effort along the path is specified to be $H = 4.5$. Thus $F = GBH = 64$, and $\hat{D} = 3(64)^{1/3} + 3(2p_{inv}) = 18.0$ delay units.

To achieve this minimum delay, we must equalize the effort in each stage. Since the path effort is 64, the stage effort should be $(64)^{1/3} = 4$. Starting from the output, $z = 4.5C \times (4/3)/4 = 1.5$. The second stage drives three copies of the third stage, so $y = 3z \times (4/3)/4 = z = 1.5C$. We can check the math by finding the size of the first stage $2y \times (4/3)/4 = (2/3)y = C$, as given in the design spec.

Example 1.7 *Size the circuit in Figure 1.8 for minimum delay. Suppose the load*

is 20 microns of gate capacitance and that the inverter has 10 microns of gate capacitance.

Assuming minimum length transistors, gate capacitance is proportional to gate width. Hence, it is convenient to express capacitance in terms of microns of gate width, as given in this problem.

The path has logical effort $G = 1 \times (5/3) \times (4/3) \times 1 = 20/9$. The electrical effort is $H = 20/10 = 2$ and the branching effort is 1. Thus, $F = GBH = 40/9$, and $\hat{f} = (40/9)^{1/4} = 1.45$.

Start from the output and work backward to compute sizes. $z = 20 \times 1/1.45 = 14$. $y = 14 \times (4/3)/1.45 = 13$. $x = 13 \times (5/3)/1.45 = 15$. These input gate widths are divided among the transistors in each gate. Notice that the inverters are assigned larger electrical efforts than the more complex gates because they are better at driving loads. Also note that these calculations do not have to be very precise. We will see in Section 3.6 that sizing a gate too large or too small by a factor of 1.5 still result in circuits within 5% of minimum delay. Therefore, it is easy to use "back of the envelope" hand calculations to find gate sizes to one or two significant figures.

Note that the parasitic delay does not enter into the procedure for calculating transistor sizes to obtain minimum delay. Because the parasitic delay is fixed, independent of the size of the logic gate, adjustments to the size of logic gates cannot alter parasitic delay. In fact, we can ignore parasitic delay entirely unless we want to obtain an accurate estimate of the time required for a signal to propagate through a logic network, or if we are comparing two logic networks that contain different types of logic gates or different numbers of stages and therefore exhibit different parasitic delays.

Example 1.8 *Consider three alternative circuits for driving a load 25 times the input capacitance of the circuit. The first design uses one inverter, the second uses three inverters in series, and the third uses five in series. All three designs compute the same logic function. Which is best, and what is the minimum delay?*

In all three cases, the path logical effort is one, the branching effort is one, and the electrical effort is 25. Equation 1.17 gives the path delay $D = N(25)^{1/N} + Np_{inv}$ where $N = 1$, 3, or 5. For $N = 1$,

we have $\hat{D} = 26$ delay units; for $N = 3$, $\hat{D} = 11.8$; and for $N = 5$, $\hat{D} = 14.5$. The best choice is $N = 3$. In this design, each stage will bear an effort of $(25)^{1/3} = 2.9$, so each inverter will be 2.9 times larger than its predecessor. This is the familiar geometric progression of sizes that is found in many textbooks.

This example shows that the fastest speed obtainable depends on the number of stages in the circuit. Since the path delay varies markedly for different values of N, it is clear we need a method for choosing N to yield the least delay; this is the topic of the next section.

1.3 Choosing the best number of stages

The delay equations of logical effort, such as Equation 1.17, can be solved to determine the number of stages, $\hat{N}$, that achieves the minimum delay. Although we will defer the solution technique until Chapter 3, Table 1.3 presents some results. The table shows, for example, that a single stage is fastest only if the path effort, F, is 5.83 or less. If the path effort lies between 5.83 and 22.3, a two-stage design is best. If it lies between 22.3 and 82.2, three stages are best. The table confirms that the right number of stages to use in Example 1.8, which has $F = 25$, is three. As the effort gets very large, the stage effort approaches 3.59.

If we use Table 1.3 to select the number of stages that gives the least delay, we may find that we must add stages to a network. We can always add an even number of stages by attaching pairs of inverters to the end of the path. Since we can't add an odd number of inverters without changing the logic function of the network, we may have to settle for a somewhat slower design or alter the logic network to accommodate an inverted signal. If a path uses a number of stages that is not quite optimal, the overall delay is usually not increased very much; what is disastrous is a design with half or twice the best number of stages.

The table is accurate only when we are considering increasing or decreasing the number of stages in a path by adding or removing inverters, because the table assumes that stages being added or removed have a parasitic delay equal to that of an inverter. Chapter 3 explains how other similar tables can be produced. When we are comparing logic networks that use different logic gate types or different numbers of stages of logic, it is necessary to evaluate the delay equations to determine which design is best.

Path effort F	Best number of stages, $\hat{N}$	Minimum delay $\hat{D}$	Stage effort, f, range
0		1.0	
	1		0 – 5.8
5.83		6.8	
	2		2.4 – 4.7
22.3		11.4	
	3		2.8 – 4.4
82.2		16.0	
	4		3.0 – 4.2
300		20.7	
	5		3.1 – 4.1
1090		25.3	
	6		3.2 – 4.0
3920		29.8	
	7		3.3 – 3.9
14200		34.4	
	8		3.3 – 3.9
51000		39.0	
	9		3.3 – 3.9
184000		43.6	
	10		3.4 – 3.8
661000		48.2	
	11		3.4 – 3.8
2380000		52.8	
	12		3.4 – 3.8
8560000		57.4	

Table 1.3: Best number of stages to use for various path efforts. For example, for path efforts between 3920 and 14200, 7 stages should be used; the stage effort will lie in the range 3.3 – 3.9 delay units. The table assumes $p_{inv} = 1.0$.

Example 1.9 *A string of inverters is used to drive a signal that goes off-chip through a pad. The capacitance of the pad and its load is 35 pF, which is equivalent to about 20,000 microns of gate capacitance. Assuming the load on the input should be a unit-sized inverter in a 0.6μ process with 7.2 microns of input capacitance, how should the inverter string be designed?*

> As in Example 1.8, the logical and branching efforts are both 1, but the electrical effort is $20000/7.2 = 2777$. Table 1.3 specifies a six-stage design. The stage effort will be $\hat{f} = (2777)^{1/6} = 3.75$. Thus the input capacitance of each inverter in the string will be 3.75 times that of its predecessor. The path delay will be $\hat{D} = 6 \times 3.75 + 6 \times p_{inv} = 28.5$ delay units. This corresponds to an absolute delay of $28.5\tau = 1.43$ ns, assuming $\tau = 50$ ps.

This example finds the best ratio of the sizes of succeeding stages to be 3.75. Many texts teach us to use a ratio of $e = 2.718$, but the reasoning behind the smaller value fails to account for parasitic delay. As the parasitic delay increases, the size ratio that achieves least delay rises above e, and the best number of stages to use decreases. Chapter 3 explores these issues further and presents a formula for the best stage effort

In general, the best stage effort $\hat{f}$ is between 3 and 4. Targeting a stage effort of 4 is convenient during design and gives delays within 1% of minimum delay for typical parasitics. Thus, the number of stages $\hat{N}$ is about $\log_4 F$. We will find that stage efforts between 2 and 8 give delays within 35% of minimum and efforts between 2.4 and 6 give delays within 15% of minimum. Therefore, choosing the right stage effort is not critical.

We will also see in Chapter 3 that an easy way to estimate the delay of a path is to approximate the delay of a stage with effort of 4 as that of a fanout-of-4 (FO4) inverter. We found in Example 1.2 that a FO4 inverter has a delay of 5 units. Therefore, the delay of a circuit with path effort F is about $5 \log_4 F$, or about $\log_4 F$ FO4 delays. This is somewhat optimistic because it neglects the larger parasitic delay of complex gates.

1.4 Summary

The method of logical effort is a design procedure for achieving the least delay along a path of a logic network. It combines into one calculation the effort

Term	Stage expression	Path expression
Logical effort	g (Table 1.1)	$G = \prod g_i$
Electrical effort	$h = C_{out}/C_{in}$	$H = C_{path-out}/C_{path-in}$
Branching effort	—	$B = \prod b_i$
Effort	$f = gh$	$F = GBH = \prod f_i$
Effort delay	f	$D_F = \sum f_i$
		minimized when $f_i = F^{1/\hat{N}}$
Number of stages	1	$\hat{N}$ (Table 1.3)
Parasitic delay	p (Table 1.2)	$P = \sum p_i$
Delay	$d = f + p$	$D = D_F + P$

Table 1.4: Summary of terms and equations for concepts in the method of logical effort.

required to drive large electrical loads and to perform logic functions. The procedure is, in summary:

1. Compute the path effort, $F = GBH$, along the path of the network you are analyzing. The path logical effort, G, is the product of the logical efforts of the logic gates along the path; use Table 1.1 to obtain the logical effort of each individual logic gate. The branching effort, B, is the product of the branching effort at each stage along the path. The electrical effort, H, is the ratio of the capacitance loading the last stage of the network to the input capacitance of the first stage of the network.

2. Use Table 1.3 or estimate $\hat{N} = \log_4 F$ to find out how many stages, $\hat{N}$, will yield the least delay.

3. Estimate the minimum delay, $\hat{D} = \hat{N}F^{1/\hat{N}} + \sum p_i$, using values of parasitic delay obtained from Table 1.2. If you are comparing different architectural approaches to a design problem, you may choose to stop the analysis here.

4. Add or remove stages to your circuit if necessary until N, the number of stages in the path, is approximately $\hat{N}$.

5. Compute the effort to be borne by each stage: $\hat{f} = F^{1/N}$

6. Starting at the last logic stage in the path, work backward to compute transistor sizes for each of the logic gates by applying the equation $C_{in} =$

$(g_i/\hat{f})C_{out}$ for each stage. The value of C_{in} for a stage becomes C_{out} for the previous stage, perhaps modified to account for branching effort.

This design procedure finds the circuit with the least delay, without regard to area, power, or other limitations that may be as important as delay. In some cases, compromises will be necessary to obtain practical designs. For example, if this procedure is used to design drivers for a high-capacitance bus, the drivers may be too big to be practical. You may compromise by using a larger stage delay than the design procedure calls for, or even by making the delay in the last stage much greater than in the other stages; both of these approaches reduce the size of the final driver and increase delay.

The method of logical effort achieves an *approximate* optimum. Because it ignores a number of second-order effects, such as stray capacitances between series transistors within logic gates, a circuit designed with the procedure given above can sometimes be improved by careful simulation with a circuit simulator and subsequent adjustment of transistor sizes. However, we have evidence that the method of logical effort alone obtains designs that are within 10% of the minimum.

One of the strengths of the method of logical effort is that it combines into one framework the effects on performance of capacitive load, of the complexity of the logic function being computed, and of the number of stages in the network. For example, if you redesign a logic network to use high fan-in logic gates in order to reduce the number of stages, the logical effort increases, thus blunting the improvement. Although many designers recognize that large capacitive loads must be driven with strings of drivers that increase in size geometrically, they are not sure what happens when logic is mixed in, as occurs often in tri-state drivers. The method of logical effort addresses all of these design problems.

The information presented in this chapter is sufficient to attack almost any design. The next chapter applies the method to a variety of circuits of practical importance. Chapter 4 exposes the model behind the method and derives the equations presented in this chapter. Chapter 4 shows how to compute the logical effort of a logic gate and exhibits a catalog of logic gate types. Chapter 5 describes how to measure various parameters required by the method, such as p_{inv} and τ. The remaining chapters explore refinements to the method and more intricate design problems.

(*a*) (*b*)

Figure 1.9: Two circuits for computing the AND function of two inputs.

1.5 Exercises

1-1 [20] Consider the circuits shown in Figure 1.9. Both have a fanout of 6, i.e., they must drive a load six times the capacitance of each of the inputs. What is the path effort of each design? Which will be fastest? Compute the sizes x and y of the logic gates required to achieve least delay.

1-2 [20] Design the fastest circuit that computes the NAND of four inputs with a fanout of 6. Consider a 4-input NAND gate by itself, a 4-input NAND gate followed by two additional inverters, and a tree formed by two 2-input NAND gates whose outputs are connected to a 2-input NOR gate followed by an inverter. Estimate the shortest delay achievable for each circuit. If the fanout were larger, would other circuits be better?

1-3 [10] A 3-stage logic path is designed so that the effort borne by each stage is 10, 9, and 7 delay units, respectively. Can this design be improved? Why? What is the best number of stages for this path? What changes do you recommend to the existing design?

1-4 [10] A clock driver must drive 500 minimum-size inverters. If its input must be a single minimum-size inverter, how many stages of amplification should be used? If the input to the clock driver comes from outside the integrated circuit via an input pad, could fewer stages be used? Why?

1-5 [15] A particular system design of interest will have eight levels of logic between latches. Assuming that the most complex circuits involve 4-input NAND gates in all eight levels of logic, estimate a useful clock period.

1-6 [20] A long metal wire carries a signal from one part of a chip to another.

Only a single unit load may be imposed on the signal source. At its destination the signal must drive 20 unit loads. The wire capacitance is equivalent to 100 unit loads; assume the wire has no resistance. Design a suitable amplifier. You may invert the signal if necessary. Should the amplifier be placed at the beginning, middle, or end of the wire?

Chapter 2

Design Examples

This chapter presents a number of design examples worked out in detail. To simplify the presentation, some of the designs are simpler than cases that are likely to arise in practice. The last design, however, is taken from an actual problem confronted by designers.

As you read through the examples, focus not only on how the mechanics of the method of logical effort are applied, but also on the insights into circuit structure that the concepts of logical effort permit. Perhaps the greatest strength of the method of logical effort is in simplifying analysis of structural variants.

All of these examples assume we are using CMOS logic gates with $p_{inv} = 1.0$. Values for the logical effort and parasitic delay of logic gates are obtained from Tables 1.1 and 1.2 respectively. The best number of stages to accommodate a given path effort is obtained from Table 1.3.

2.1 The AND function of eight inputs

Ben Bitdiddle is developing the ALPHANOT microprocessor and needs an 8-input AND gate. He is considering three options for the structure of the circuit shown in Figure 2.1. Which one is best?

Before beginning the analysis of these three circuits, let us pause to introduce a notation that we will use in this book. To describe a path through a network, it suffices to list the logic gates that lie along the path. The circuit shown in Figure 2.1a can be described by the path (8-NAND, inverter). Similarly, the second

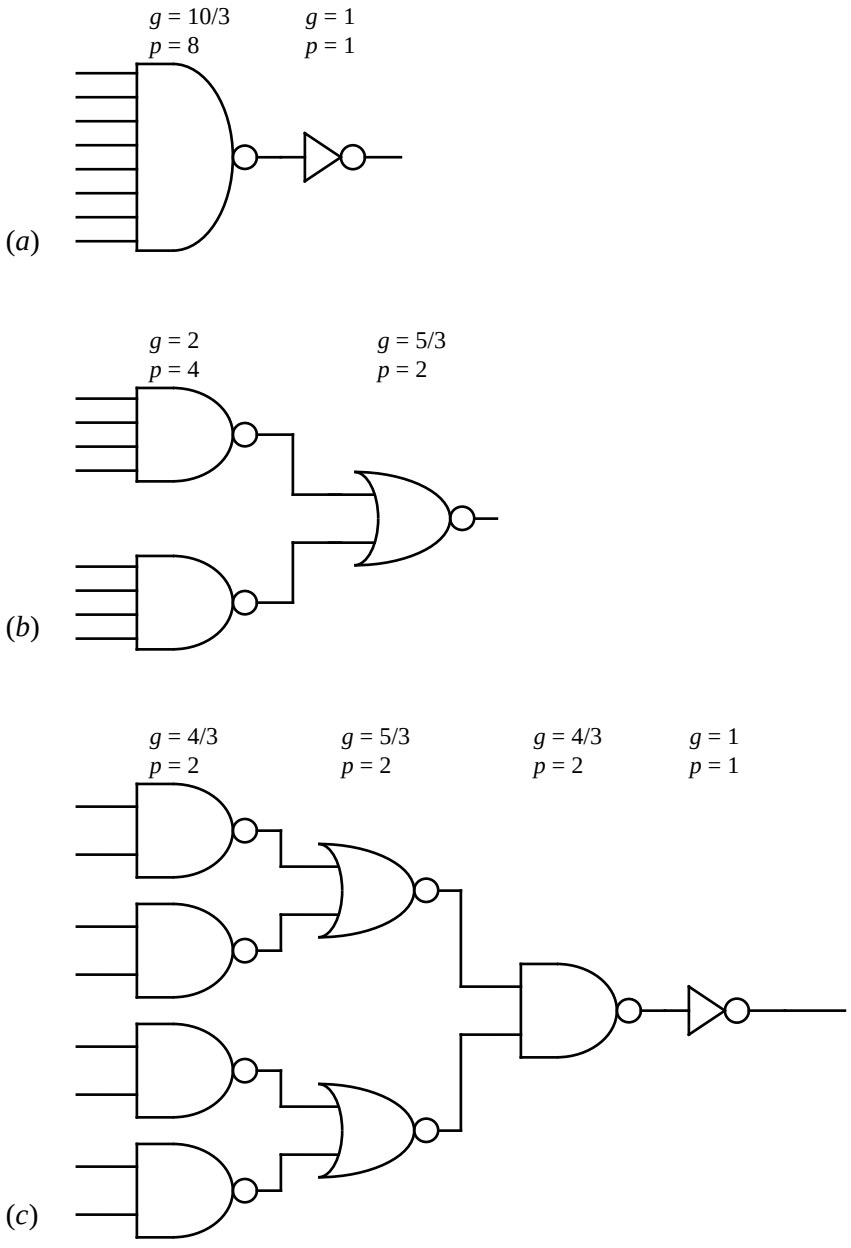


Figure 2.1: Three circuits for computing the AND of eight inputs.

circuit is (4-NAND, 2-NOR) and the third is (2-NAND, 2-NOR, 2-NAND, inverter). Often the networks are symmetric, so that all paths through the network have the same description, as is the case with all three circuits in Figure 2.1.

Let us start the analysis by computing the logical effort of each of the three alternatives. In case a, the path logical effort is the product of the logical effort of an 8-input NAND gate, which is $10/3$, and that of an inverter, which is 1, so $G = 10/3 \times 1 = 3.33$. In case b, the logical effort is the product of $6/3$, the logical effort of a 4-input NAND gate, and $5/3$, the logical effort of a 2-input NOR gate, for a total of $10/3 = 3.33$—the same as case a. The logical effort in the last case is computed as $(4/3) \times (5/3) \times (4/3) \times 1 = 2.96$. Since we know that logical effort is related to delay, we might conclude that the last case is the fastest because it yields the lowest logical effort.

Logical effort is not the only aspect to consider, however, because the load to be driven will also influence the speed of the circuit. In particular, the circuits do not all have the same number of stages, and the method of logical effort shows that minimum delay is obtained only when the number of stages is chosen to accommodate the effort, both logical and electrical. So we can't decide which circuit will achieve the least delay until we know the electrical effort and can determine the best number of stages.

The delay equation, Equation 1.17, tells us how the minimum delay that can be obtained from each circuit is related to the electrical effort H the circuit bears. These equations also include the effect of the parasitic delays, obtained by summing the parasitic delays of each of the logic gates along the path:

$$\hat{D} = N(GBH)^{1/N} + P$$

$$\text{Case } a \qquad \hat{D} = 2(3.33H)^{1/2} + 9.0 \tag{2.1}$$

$$\text{Case } b \qquad \hat{D} = 2(3.33H)^{1/2} + 6.0 \tag{2.2}$$

$$\text{Case } c \qquad \hat{D} = 4(2.96H)^{1/4} + 7.0 \tag{2.3}$$

Let us illustrate the effect of electrical effort on circuit choice by solving two problems, one with $H = 1$, and one with $H = 12$. Table 2.1 shows the results of evaluating the delay equations for the three circuits with different electrical efforts. The table shows that for $H = 1$, the designs with two stages (cases a and b) have less effort delay than the design with four stages (case c). Of the two-stage designs, case b is faster because it has less parasitic delay. When the electrical effort increases to $H = 12$, the design with the larger number of stages is best.

These results agree with the predictions for the best number of stages to use for a given path effort. Since the logical effort of all three circuits is approximately

Case	$H = 1$			$H = 12$		
	$NF^{1/N}$	P	$\hat{D} = NF^{1/N} + P$	$NF^{1/N}$	P	$\hat{D} = NF^{1/N} + P$
a	3.65	9.0	12.65	12.64	9.0	21.64
b	3.65	6.0	9.65	12.64	6.0	18.64
c	5.25	7.0	12.25	9.77	7.0	16.77

Table 2.1: Delays for computing the AND of eight inputs for two different values of electrical effort.

three, we find that the path effort when $H = 1$ is $F = GBH \approx 3$, while when $H = 12$, $F \approx 36$. Table 1.3 shows that when $F = 3$, a one-stage design will be best, while when $F = 36$, a three-stage design will be best. Clearly, cases a and b best approximate a one-stage design. It is not immediately obvious whether a two-stage or four-stage path is closest to the three-stage design recommended by the table, but usually it is better to err by one stage too many, as happens in this example where case c is the fastest. Note that this reasoning ignores the effects of parasitic delay when the logic gate types in the competing circuits are different, as they are in this case. While this method yields approximate answers, a precise answer requires comparing the delay equations for each circuit.

This example shows that the choice of circuit to use depends on the load to be driven. Because there is a relationship between the load and the best number of stages, one must know the size of the load capacitance in relation to the size of the input capacitance in order to make the proper choice of circuit structure.

2.1.1 Calculating gate sizes

The different circuits for computing the AND of eight inputs can illustrate the calculation of gate sizes along a path. Let us start with electrical effort, H, of 12, which calls for design 2.1c. Let us assume that the input capacitance is 4 units, so the load capacitance is $4H = 48$ units. From our earlier analysis, we know that each stage should bear an effort $\hat{f} = F^{1/4} = (2.96 \times 12)^{1/4} = 2.44$. Let us work backward along the path, starting with the inverter at the right. At each gate, we apply the capacitance transformation of Equation 1.19 to find the input capacitance given the output load.

The inverter at the right should have $C_{in} = 48 \times 1/2.44 = 19.66$. This becomes the load for the third stage, which therefore should have $C_{in} = 19.66 \times (4/3)/2.44 = 10.73$. This in turn becomes the load for the NOR in the second stage, which should have $C_{in} = 10.73 \times (5/3)/2.44 = 7.33$. Finally, we can

use this as the load on the NAND gate in the first stage, which should have $C_{in} = 7.33 \times (4/3)/2.44 = 4.0$. This agrees with the specified input capacitance, so our calculation checks.

If Ben Bitdiddle were building a full-custom chip, he could select transistor sizes for each gate to match the input capacitances we have just computed. This will be discussed further in Section 4.3. If Ben were using an existing cell library, he could simply select the gates from the library which have input capacitances closest to the computed values. We will see in Section 3.6 that modest deviation from the computed sizes still gives excellent performance, so he should not be concerned if his library does not have a cell of exactly the desired size. Even for a full-custom design, it is necessary to adjust transistor sizes to the nearest available size, such as an integer.

Since rounding will occur anyway and precision in sizing is not very important, experienced designers often perform logical effort calculations mentally, keeping results to only one or two significant figures.

Now let us consider electrical effort of unity, which calls for design 2.1b. We will again assume that the input capacitance is 4 units, so now the output capacitance is also 4 units. To obtain the fastest operation, each stage should bear an effort $\hat{f} = F^{1/2} = (3.33 \times 1)^{1/2} = 1.83$.

Working backward, the NOR gate in the second stage should have $C_{in} = 4 \times (5/3)/1.83 = 3.64$. This is the load on the first stage NAND gate, which must have input capacitance of 4. Notice that the NAND has an electrical effort $3.64/4 = 0.91$ less than one! This result may seem somewhat alarming at first, but it simply means that the load on the gate's output must be less than the load presented at its input, in order that the gate be sufficiently lightly loaded that it can operate in the required time. In other words, since we're equalizing effort in each stage, a stage with large logical effort g must have small electrical effort h.

2.2 Decoder

Ben Bitdiddle is now responsible for memory design on the Motoroil 68W86, an embedded processor targeting automotive applications. He must design a decoder for a 16 word register file. Each register is 32 bits wide and each bit cell presents a total load, gate and wire, equal to 3 unit-sized transistors. True and complementary versions of the four address bits are available and can each drive 10 unit-sized transistors.

The decoder could be designed with a few stages of high fan-in gates or with

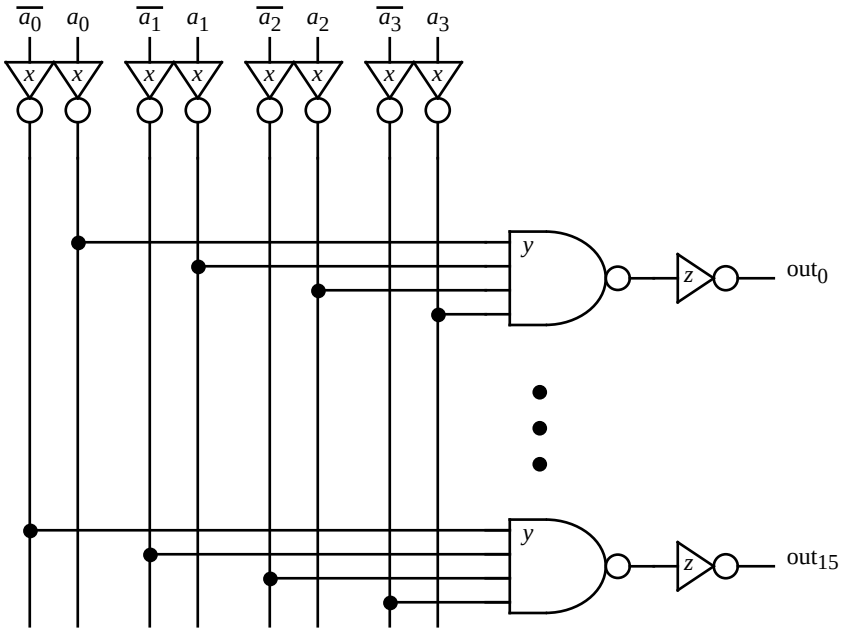


Figure 2.2: 3-stage 4:16 decoder circuit.

many stages of simple gates. The best topology depends on the effort of the path. Unfortunately, the path effort depends on the logical effort, which depends in turn on the topology!

Because a decoder is a relatively simple structure, we can make an initial estimate of the path effort by assuming the logical effort is unity. The electrical effort is $32 \times 3/10 = 9.6$. The branching effort is 8 because the true and complementary address inputs each control half of the outputs. Path effort is $9.6 \times 8 = 76.8$. Hence, we should use about $\log_4 76.8 = 3.1$ stages. Since we neglected logical effort, the actual number of stages will be slightly higher than the number we have estimated. A 3-stage circuit is shown in Figure 2.2 while a 4-stage circuit is considered in Exercise 2-3.

The circuit uses sixteen 4-input NAND gates. Since each address input must drive eight of the NAND gates, yet can handle only a relatively small input capacitance, we use an inverter to power up the signal. How do we size the decoder and what is its delay?

Because the logical effort is $1 \times 2 \times 1 = 2$, the actual path effort is 154 and the stage effort is $f = (154)^{1/3} = 5.36$. Working from the output, the final inverter must have input capacitance $z = (32 \times 3) \times 1/5.36 = 18$ and the NAND gate

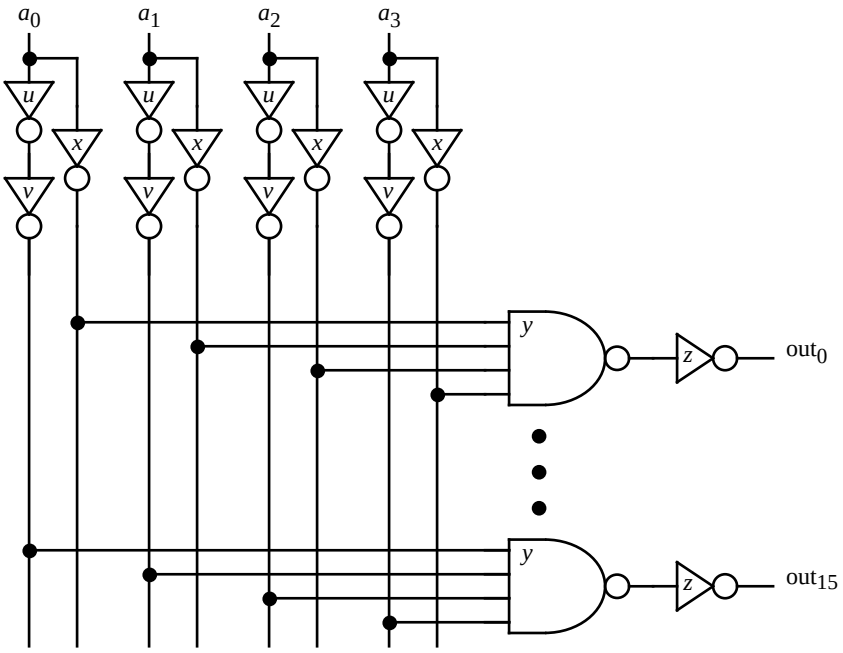


Figure 2.3: 4:16 decoder with one polarity of input.

must have input capacitance $y = 18 \times 2/5.36 = 6.7$. The delay is $3f + P = 3 \times 5.36 + 1 + 4 + 1 = 22.1$. These results are summarized in Case 1 of Table 2.2.

2.2.1 Generating complementary inputs

Now suppose the inputs were available only in true polarity and that Ben must produce his own complementary versions. To be fair, let the true signals drive a load of 20 unit-sized transistors. The new decoder is shown in Figure 2.3.

The inverter strings used to compute true and complementary versions of the input are called *forks* and are discussed further in Chapter 6. The 2-inverter and 1-inverter legs of the fork must drive the same load, a NAND gate, in the same amount of time. Computing the best sizes for circuits that fork can require iteration. Fortunately, we can make simple approximations that produce good results.

Suppose we keep all sizes the same except to choose a size v for the extra inverter. We recall that the stage efforts of inverters u and v should be equal and are therefore $\sqrt{5.36} = 2.32$ because they must together bear the same effort as the

Case	x	y	z	u	v	P	D
1	10	6.7	18			6	22.1
2	10	6.7	18	10	23.2	7	22.4
3	11.2	9.8	21.6	8.8	26.2	7	21.8

Table 2.2: Sizes and delays of decoder designs.

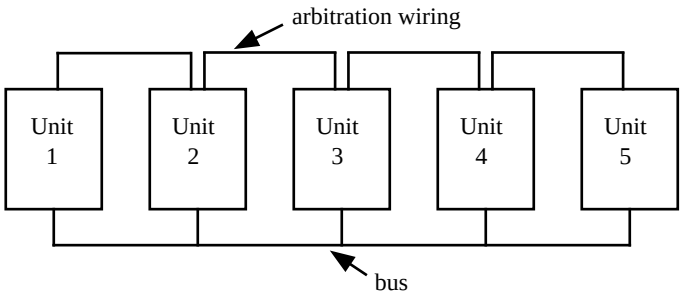


Figure 2.4: The physical arrangement of five units connected by a common bus and arbitration circuitry. The units are sufficiently large that the wires between them have significant stray capacitance.

one-inverter path. Therefore, we can size $v = 10\sqrt{5.36} = 23.2$. The delay of the decoder via the 2-inverter leg is now $2.32 \times 2 + 5.36 \times 2 + 1 + 1 + 4 + 1 = 22.4$. These results are summarized in Case 2 of Table 2.2. This topology is less than 2% slower than the original design, so the approximation worked well.

If we were concerned about every picosecond of delay, we could try tweaking some of the sizes. For example, the circuit may be improved by dedicating more than half of the address input capacitance to one leg of the fork. Also, the circuit may be improved by choosing a stage effort for the second two stages between the efforts used for the 1-inverter and 2-inverter legs of the fork. We found the best sizes by writing the delay equations in a spreadsheet and letting it solve for minimum delay. The results are summarized in Case 3 of Table 2.2. The delay improvement is tiny and was probably not worth the effort.

Ben Bitdiddle, faced with bizillions of transistors to design, would rather not waste time tweaking sizes for tiny speedups. How could he have found in advance that his design was good enough? We will show in Section 3.4 that the best possible delay is $\rho \log_\rho F + P$, where the best stage effort ρ is about 4. Therefore, a lower bound on the delay of the circuit in Figure 2.2 is $4 \log_4 154 + 1 + 4 + 1 = 20.5$.

2.3 Synchronous arbitration

Ben Bitdiddle transferred yet again to the Pentagram Processor project. The Pentagram has five separate function units that share a single on-chip bus mediated by a sinister arbitration circuit that determines which function unit may use the bus on each cycle (Figure 2.4). The operation of the bus and the arbitration circuits is synchronous: during one clock cycle, each function unit presents to the arbitration circuit its *request* signal R_i, and the arbitration circuit computes a *grant* signal G_i indicating which function unit may use the bus on the next clock cycle. While the bus is being used during one cycle, the arbitration circuit determines which function unit may use the bus during the next cycle. The five function units have fixed priority, with unit 1 having the highest priority and unit 5 the lowest.

The speed of the arbitration circuit is critical, because each unit requires a portion of the clock cycle to compute the request signals, and the remainder of the clock cycle must be sufficient to compute the arbitration results. Moreover, because the function units are physically large, the capacitance of the wiring between the units will retard the circuit. The critical delay for the circuit will be the time from the arrival of the last request signal until delivery of the last grant signal.

This example explores the proper number of stages in the path and the effect of fixed wire loading. It is somewhat complex and may be skipped on a first reading.

2.3.1 The original circuit

A designer proposed the circuit shown in Figure 2.5 for arbitration. It relies on a *daisy chain* to compute which unit should be granted access to the bus. The signal C_i represents the chain, and is interpreted as "C_i is true exactly when unit i and all higher-priority units are *not* requesting service." The designer then formulated the following boolean equations to express what each function unit must compute:

$$C_0 = \mathbf{true} \tag{2.4}$$

$$C_i = C_{i-1} \wedge \overline{R_i} \tag{2.5}$$

$$G_i = C_{i-1} \wedge R_i \tag{2.6}$$

The designer manipulated these equations so that only one gate would be required for each stage of the daisy chain:

$$\overline{C_i} = \overline{C_{i-1} \wedge \overline{R_i}} \tag{2.7}$$

$$C_i = \overline{\overline{C_{i-1}} \vee R_i} \tag{2.8}$$

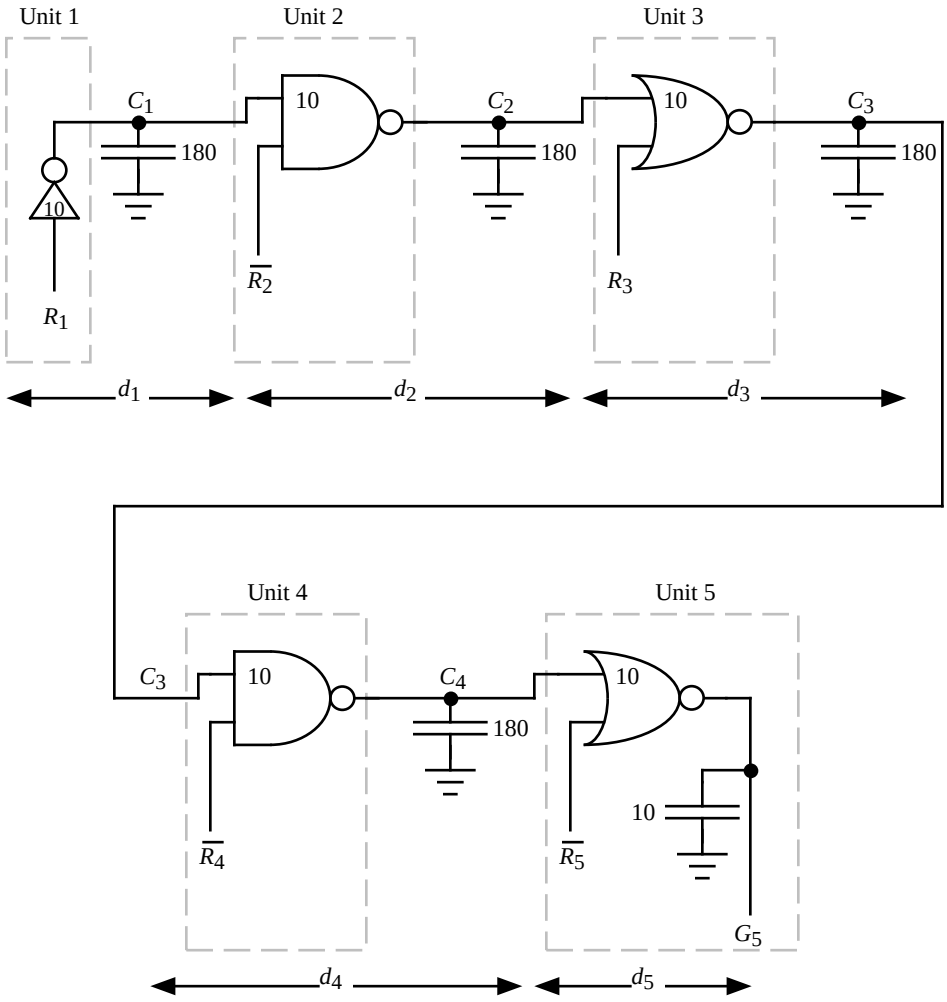


Figure 2.5: Arbitration circuit for five units, using a daisy-chain method. Unit 1 has highest priority, and unit 5 lowest. Only the critical path is shown; additional circuitry is required to compute grant signals for units 1 through 4.

Stage	C_{in}	C_{out}	$h = C_{out}/C_{in}$	g	$f = gh$	p
1	10	190	19	1	19	1.0
2	10	190	19	4/3	25.3	2.0
3	10	190	19	5/3	31.7	2.0
4	10	190	19	4/3	25.3	2.0
5	10	10	1	5/3	1.7	2.0
Total delay					103	9.0

Table 2.3: Delay computations for the circuit in Figure 2.5.

Thus the gates on the daisy chain alternate between NAND and NOR gates, and the daisy chain signal alternates between true and complement forms. Figure 2.5 shows all of the circuitry on the critical path from R_1 to G_5, but omits much of the rest. We assume that the request signals are available in true or complement form, that the grant signal can be computed in complement form, that each R_i and G_i is loaded with 10 units of capacitance, and that the daisy-chain wire leading from one function unit to the next has a stray capacitance of 180 units.

Let us start by estimating the speed of the circuit shown in the figure. We will analyze the stage delay d_i in each of the five stages, as shown in Figure 2.5. For each stage, we determine the electrical and logical effort, which we multiply to obtain the effort delay. The results are shown in Table 2.3: the overall effort delay is 103, and parasitic delay is 9, for a total delay of 112.

Table 2.3 illustrates some of the defects in the circuit design of Figure 2.5. We know that overall delay is least when the effort delay is the same in every stage, but in this design the delays vary between 1.7 and 32. This observation suggests that we have used the wrong number of stages in the design.

Let us compute the effort along the path. The electrical effort is 1, because both the input capacitance of R_1 and the output capacitance of G_5 are 10. There are four sites along the path at which the branching effort is $(180 + 10)/10 = 19$, due to the stray capacitance of the wiring; thus the branching effort is 19^4. The logical effort is the product of the logical efforts of the gates, or $1 \times (4/3) \times (5/3) \times (4/3) \times (5/3) = 4.94$. The path effort is therefore $F = GBH = 4.94 \times 19^4 \times 1 = 643785$. Table 1.3 shows that we should be using 10 stages, rather than the five in the present design. This is a big error, which suggests there is room for dramatic improvement.

A simple improvement is to enlarge the NAND gates along the daisy chain. If the input capacitance of each gate input were 90 rather than 10, the branching effort would be reduced to 3^4 and the total effort becomes $F = 4.94 \times 81 \times 1 =$

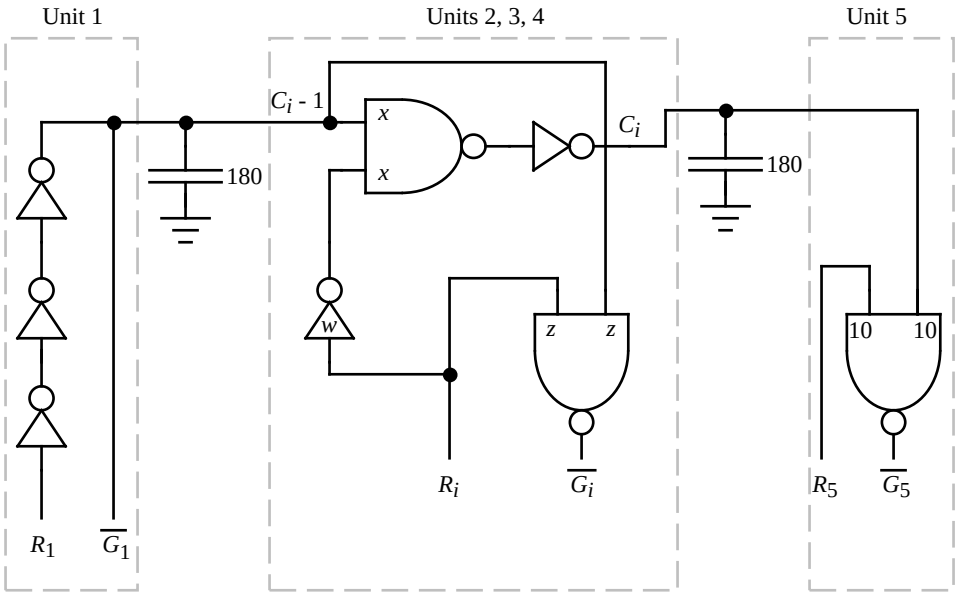


Figure 2.6: An improved arbitration circuit, using two stages of logic for each unit.

400. This calls for a 5-stage design, with an estimated delay of $5(400)^{1/5}+9p_{inv} = 25.6$, which is a vast improvement over the estimate of 112 for the original design. However, this change increases the load on each of the request signals, which will add more delay as well as more area.

2.3.2 Improving the design

Because the best design would use 10 stages of logic, an improved circuit should use two stages of logic for each function unit, rather than one. Each unit should contain a logic gate and an inverter, which permits the daisy-chain signal to have a constant polarity and makes all arbitration units identical. Figure 2.6 shows the new structure: the logic in the box at the center of the figure is the logic associated with stages 2, 3, and 4 of the arbitration. The logic of the arbitration problem allows the first and last units to differ, because C_0 is always true and C_6 is unnecessary.

The transistor sizes shown in the figure as variables $w, x, y,$ and z are all determined by the method of logical effort. Let us start by analyzing the critical path in the middle units, namely the path from C_{i-1} to C_i. The load capacitance

on this path is the stray capacitance, 180, plus $x + z$, the input capacitance of the two NAND gates in the next unit. For the critical path, $H = C_{out}/C_{in} = (180 + x + z)/x$. The logical effort along this path is the logical effort of the NAND gate, which is $4/3 \times 1 = 4/3$. For the design to be fast, we know that we should target a stage effort of about 4, as discussed in Section 1.3. Because we are using a two-stage design, the two stages should bear an effort of $4 \times 4 = 16$. So we have the equation:

$$F = GH \tag{2.9}$$

$$16 = \frac{4}{3}\frac{(180 + x + z)}{x} \tag{2.10}$$

To solve this equation, we will assume that z is small compared to $180 + x$, and can be neglected. Solving, we obtain $x = 16.4$.

We can now calculate y in two ways. The NAND gate stage should have an effort delay of 4, so:

$$f = gh \tag{2.11}$$

$$4 = (4/3)(y/x) \tag{2.12}$$

Since $x = 16.4$, we can solve for y to obtain $y = 49$. Alternatively, we can consider the delay in the inverter stage, which has electrical effort approximately $(180 + x)/y$, so we obtain a delay equation $4 = (180 + x)/y$. Solving for y, we obtain a value of 49, the same answer.

Now let us turn to calculating z and w. Even though paths leading from R_i or to G_i are not on the critical path of the entire arbitration chain, let us try to give them reasonable performance as well. For the inverter to have a stage delay of 4, we must have $x/w = 4$, so $w = 4.1$. Given the stipulation that R_i offer a load of 10 units of capacitance, we must have $z = 10 - w = 5.9$. This will mean that the effort delay in the generation of $\overline{G_i}$ will be $gh = (4/3)(10/5.9) = 2.3$. Is this delay reasonable? If it were much greater than 4, the gate would have very slow rise/fall times and could suffer hot electron problems. If it were much less than 4, the gate would probably be presenting too much load on its inputs. 2.3 is acceptable, so we are done.

Let us now turn to the first and last units of the design. The last unit need only generate $\overline{G_5}$. As a consequence, we make the NAND gate as fast as possible by making it as large as allowed, given the constraint on the load capacitance of R_5. The first unit needs to compute $C_1 = \overline{R_1}$, but must drive a considerable load. The load is 10 units for the connection to $\overline{G_1}$, 180 units for the wiring capacitance, and

Unit	Number of stages	Stage effort delay	Path effort delay	Path parasitic delay
1	3	2.8	8.4	1×3
2	2	4	8.0	$2 + 1$
3	2	4	8.0	$2 + 1$
4	2	4	8.0	$2 + 1$
5	1	1.3	1.3	2
Total delay			33.7	14

Table 2.4: Delay computations for the circuit in Figure 2.6.

$x + z = 22.3$ units for the input capacitance of unit 2, for a total of 212. Thus the electrical effort is $H = 212/10 = 21$. Since the logical effort of the inverters is 1, the path effort F is also 21. Table 1.3 tells us that two stages of logic are required to bear this effort, but we need an odd number of inversions. Shall we use one or three inverters? The effort is closer to the range for three inverters than one, so we use three. Another way of choosing the number of stages is to compute $N = \log_4 F = 2.2$, then rounding N to 3, the nearest odd number of stages.

The stage effort delay will be $H^{1/N} = 21.2^{1/3} = 2.8$. We know that the input capacitance of the first inverter is 10 units, so the input capacitance of the second will be $10 \times 2.8 = 28$, and that of the third will be $10 \times 2.8 \times 2.8 = 78$.

Now that the design is finished, let us compute the delay we expect along the critical path from R_1 to $\overline{G_5}$. This calculation is largely a matter of recalling the stage delays used to obtain the transistor sizes. The calculation appears in Table 2.4. The path effort delay is 33.7 and the parasitic delay is 14, for a total of 47.7. The improved circuit is better than twice as fast as the original. The designer of the original tried to achieve speed by minimizing the number of logic gates in the circuit, but a far faster circuit uses twice the number of gates!

Also notice that in this circuit the fixed wiring capacitance still dominates the loading. Therefore, larger gates could have been used in the daisy chain, only slightly increasing total loading on the C_i signals while significantly reducing stage effort. Finding exact solutions to problems with fixed loading usually requires iteration, but the essential idea is to enlarge gates on the node with fixed capacitance until their input capacitance becomes a non-negligible portion of the node capacitance.

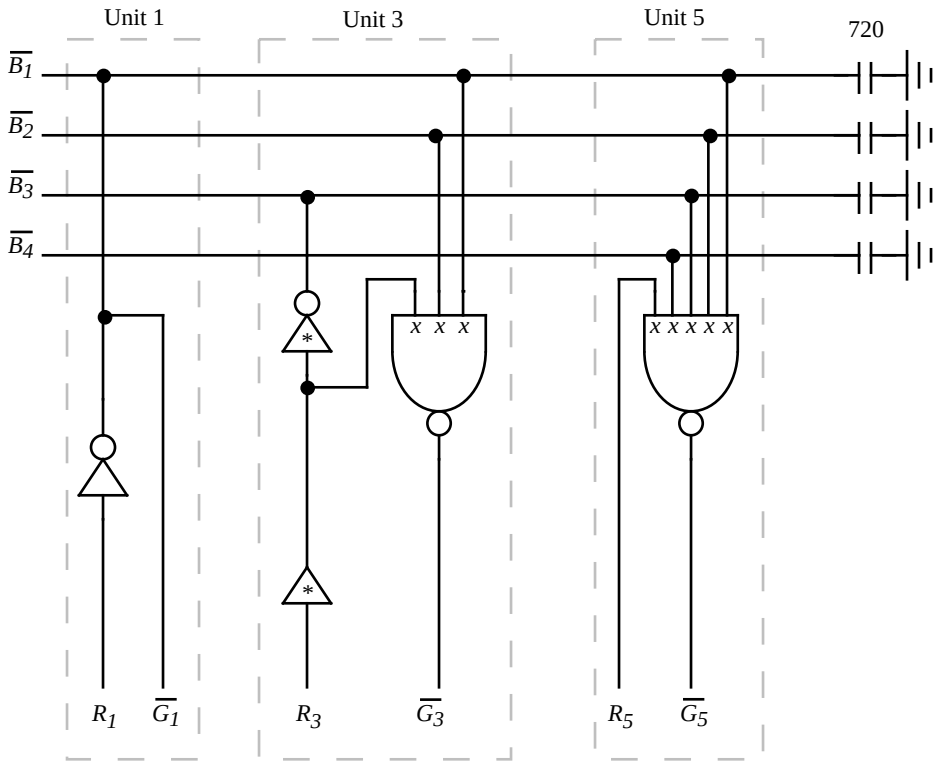


Figure 2.7: Arbitration circuit using broadcast requests.

2.3.3 Restructuring the problem

The arbitration circuit can be made even faster by changing its structure. The weakness of the current design is the daisy chain, which bears a large stray capacitive load. The four segments of the chain are in series with logic, so that the electrical and logical efforts compound to produce a very large path effort, which leads to large delays.

An alternative structure is to transmit the four request signals $R_i, i = 1, 2, 3, 4$ to all units and place logic in each unit to compute the grant signals. Figure 2.7 shows this structure; note that the capacitive load on the four broadcast signals is four times the load on each of the daisy-chain signals because the broadcast signals are four times as long.

Let us consider the effort along the path from R_1 to $\overline{G_5}$. The electrical effort is 1, because the load on R_1 is 10 units and the load on $\overline{G_5}$ is also 10 units. The logical effort is the logical effort of the 5-input NAND gate, which is $7/3$. The branching effort is $(720 + 4x)/x$, where x is the input capacitance of the NAND gate. The path effort is thus $F = GBH = (7/3) \times (720 + 4x)/x \times 1$. To obtain least delay, we should minimize this effort by choosing x as large as possible, but excessive values will lead to layout problems. Also as x becomes comparable to the load capacitance it drives, there is little benefit to increasing x. Although large x theoretically would reduce the branching effort, the delay driving the load is already small. We will choose a modest value, $x = 10$, in part because R_5 can then drive the NAND gate directly, and in part because this is a convenient number. Thus $F = 170.3$, which from Table 1.3 suggests a 4-stage design. Since the NAND gate represents one stage, we shall use three inverters to amplify R_1 for driving the broadcast wires.

Even though we have yet to compute transistor sizes, we can estimate the delay of this design. The effort delay in each stage will be $\hat{f} = F^{1/N} = 170.3^{1/4} = 3.6$, for a total delay of $4 \times 3.6 = 14.4$. The parasitic delay will be 3×0.6 for the three inverters and 5×0.6 for the NAND gate, for a total of 4.8. The overall delay is thus $\hat{D} = 14.4 + 4.8 = 19.2$. This represents a further improvement over the previous designs, at the expense of additional long wires.

2.4 Summary

The design examples in this chapter illustrate a number of points about designing for high speed.

- Tree structures are an attractive way to combine a great many inputs, especially when the electrical effort is large. These structures show up in adders, decoders, comparators, etc. Chapter 11 shows further design examples of tree structures.

- Forks are used to produce true and complementary versions of a signal. The input capacitance is divided among the legs so that the effort delay is equalized.

- Minimizing the number of gates is not always a good idea. The design of Figure 2.6 uses twice as many gates in the critical path as the design of Figure 2.5, but is substantially faster. The best number of stages depends on the overall path effort.

- Because delay grows only as the logarithm of the capacitive load, it is almost always wise to consolidate load in one part of the circuit rather than to distribute it around. Thus the broadcast scheme in Figure 2.7 is better than the daisy-chain method. Section 7.4 considers this problem further.

- When a path has a large fixed load, such as wire capacitance, the path can be made faster by using a large receiving gate on the node because the larger gates will provide much more current, yet only slightly increase the total node capacitance. In other words, the larger receiver reduces the branching effort of the path.

- While the parasitic delay is important to estimate the actual delay of a design, it rarely enters directly into our calculations. Rather, it enters indirectly into the choice of the best number of stages and, equivalently, the best effort borne by each stage.

2.5 Exercises

2-1 [20] Compare the delays of the three cases in Figure 2.1 by plotting three curves on one graph, one curve for each of the delays predicted by Equations 2.1 to 2.3. The graph should show total delay as a function of electrical effort, H, up to $H = 200$. Consider also a case similar to case c, but with two more inverters connected to the output. Write the delay equation for this case and add its plot to the graph. What does the graph show?

2-2 [20] Find the network that computes the OR function of six inputs in least time, assuming an electrical effort of 140. The network may use NAND and NOR gates with up to four inputs, as well as inverters.

2-3 [20] Since we did not include logical effort in the estimate of the number of decoder stages, we may have underestimated the best number of stages. Suppose the decoder design with true and complementary inputs from Figure 2.2 were modified to use 4 stages instead of 3 by adding another input inverter. Find the best size for each stage and the delay of the decoder. Is it better or worse than the 3 stage design? Is the difference significant?

2-4 [15] The critical path for the middle units of the arbitration circuit in Figure 2.6 is from C_{i-1} to C_i. This suggests that the sizes of the gates associated with R_i and G_i can be made as small as we wish, e.g., $w = z = 1$. Is this a good idea? Why or why not?

2-5 [10] The design in Figure 2.6 uses a NAND gate in each stage. Why not use a NOR gate?

2-6 [25] The design in Figure 2.6 uses some rather large transistors. Suppose that the largest logic gate you may use has an input capacitance of 30 units. How fast a design can you obtain?

2-7 [25] Using the reasoning outlined in Section 2.3.3, compute transistor sizes for the design in Figure 2.7, without assuming that $x = 10$. Why is $x \neq 10$ for the fastest design?

2-8 [30] Suppose you are told to design an arbitration circuit like the ones described in Section 2.3, with the requirement that its overall delay be no more than 60 units. Which structure would you choose? Show a detailed design.

Chapter 3

Deriving the Method of Logical Effort

The method of logical effort is a direct result of a simple model of logic gates in which delays result from charging and discharging capacitors through resistors. The capacitors model transistor gates and stray capacitances; the resistors model networks of transistors connected between the power supply voltages and the output of a logic gate. The derivations presented in this chapter provide a physical basis for the following notions:

- The logical effort, electrical effort, and parasitic delay are parameters of a linear equation that gives the delay in a logic gate.
- The least delay along a path of logic gates is obtained when each logic gate bears the same effort.
- The number of stages to use in a path for least delay can be computed knowing only the effort along the path and, remarkably, the parasitic delay of an inverter.
- The extra delay incurred by using the wrong number of stages is small unless the error in the number of stages is large.

These results validate the method of logical effort.

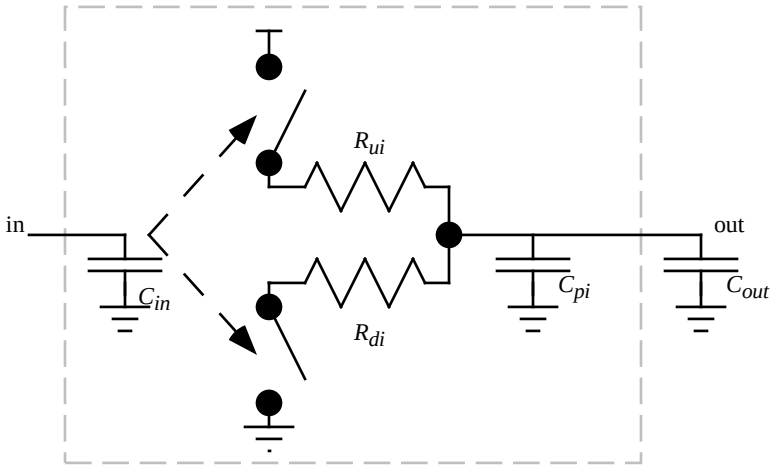


Figure 3.1: Conceptual model of a logic gate, showing only one input. The output is driven HIGH or LOW through a resistor.

3.1 Model of a logic gate

An electrical model that approximates the behavior of a single logic gate designed as a static circuit is shown in Figure 3.1. The figure shows an input signal loaded by a capacitance C_{in}, the capacitance of the transistor gates connected to the input terminal. The voltages on the input terminals, of which only one is shown in the figure, determine which transistors will be switched on and which off. If the upper switch conducts, it connects the output of the logic gate to the positive power supply, through a *pullup* resistance R_{ui} that models the resistance of the pullup network of transistors that conduct current from the positive supply to the output terminal. Alternatively, the bottom switch may conduct, connecting the output of the logic gate to ground through a *pulldown* resistance R_{di}. The output of the logic gate is loaded by two capacitances: C_{pi}, a parasitic capacitance associated with components of the logic gate itself, and a load capacitance C_{out}, which represents the load presented to the logic gate by the input capacitance of logic gates it drives and by the stray capacitance of the wiring connected to the gate's output terminal.

The logic gate is modeled by the four quantities C_{in}, R_{ui}, R_{di}, and C_{pi}, which are related in various ways depending on the particular logic function, the performance of the transistors in the CMOS process used, and so on. Because we are interested in choosing transistor sizes to obtain minimum delay, we shall view a

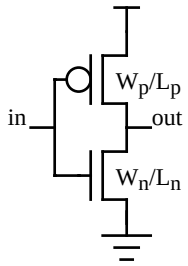


Figure 3.2: A design for an inverter. The transistors are labeled with the ratio of the width to length of the transistor.

logic gate as a scaled version of a *template circuit*. To obtain a particular logic gate, we scale the widths of all transistors in the template by a factor α. The template will have input capacitance C_t, equal pullup and pulldown resistances R_t, and parasitic capacitance C_{pt}. Thus the four quantities in the model are related to corresponding template properties and the scale factor α:

$$C_{in} = \alpha C_t \tag{3.1}$$

$$R_i = R_{ui} = R_{di} = R_t/\alpha \tag{3.2}$$

$$C_{pi} = \alpha C_{pt} \tag{3.3}$$

The scaling of the template increases the widths of all transistors by the factor α, leaving the transistor lengths unchanged. As a transistor's width is scaled, its gate capacitance increases by the scale factor, while its resistance decreases by the scale factor. The relationships shown in these equations also reflect an assumption that the pullup and pulldown resistances are equal, so as to obtain equal rise and fall times when the output of the logic gate changes. This restriction makes circuits slightly slower overall; it will be relaxed in Chapter 9.

The model shown in Figure 3.1 relates easily to the design of an inverter, such as the template shown in Figure 3.2. The n-type pulldown transistor, with width W_n and length L_n, is modeled in Figure 3.1 by the switch and resistor R_{di} that form a path from the output to ground. The p-type pullup transistor, with width W_p and length L_p, is modeled by the switch and resistor R_{ui} forming a path to the positive power supply. The input signal is loaded by the capacitance formed by the gates of both transistors, which is proportional to the area of the transistor gates:

$$C_t = \kappa_1 W_n L_n + \kappa_1 W_p L_p$$

where κ_1 is a constant that depends on the fabrication process. The resistances are

determined by:

$$1/R_t = \kappa_2 \mu_n W_n/L_n = \kappa_2 \mu_p W_p/L_p$$

where κ_2 is a constant that depends on the fabrication process, and the μ's characterize the relative mobilities of carriers in n and p-type transistors. Note that this equation implies a constraint on the design of the inverter template to insure that pullup and pulldown resistances are equal, namely $\mu_n W_n/L_n = \mu_p W_p/L_p$.

The model of Figure 3.1 also relates easily to logic gates other than inverters. Each input is loaded by the capacitance of the transistor gates it drives. The circuit of the logic gate is a network of source-to-drain connections of transistors such that the output of the logic gate can be connected either to the power supply or to ground, depending on the voltages present on the input signals that control the transistors in the network. The pullup and pulldown resistances shown in the model are the effective resistances of the network when the pullup or pulldown path is active. We shall defer until Chapter 4 a detailed analysis of popular logic gates and their correspondence to the model.

3.2 Delay in a logic gate

The delay in a logic gate modeled by Figure 3.1 is just the RC delay associated with charging and discharging the capacitance attached to the output node:

$$\begin{aligned} d_{abs} &= \kappa R_i (C_{out} + C_{pi}) && (3.4) \\ &= \kappa (R_t/\alpha) C_{in} (C_{out}/C_{in}) + \kappa (R_t/\alpha)(\alpha C_{pt}) \\ &= (\kappa R_t C_t)(C_{out}/C_{in}) + \kappa R_t C_{pt} && (3.5) \end{aligned}$$

where κ is a constant characteristic of the fabrication process that relates RC time constants to delay. The third equation is obtained from the first by rearranging terms and substituting values for R_i, C_{in}, and C_{pi} obtained from Equations 3.1 to 3.3. It is a characteristic of our formulation that the scale factor α is absent in the final form; it is hidden in C_{in}.

We can rewrite Equation 3.5 to obtain the key equations of logical effort:

$$\begin{aligned} d_{abs} &= \tau(gh + p) && (3.6) \\ \tau &= \kappa R_{inv} C_{inv} && (3.7) \\ g &= \frac{R_t C_t}{R_{inv} C_{inv}} && (3.8) \end{aligned}$$

$$h = \frac{C_{out}}{C_{in}} \tag{3.9}$$

$$p = \frac{R_t C_{pt}}{R_{inv} C_{inv}} \tag{3.10}$$

where C_{inv} is the input capacitance of the inverter template, and R_{inv} is the resistance of the pullup or pulldown transistor in the inverter template.

Equation 3.6 gives the delay of a logic gate in terms of logical effort g, electrical effort h, and parasitic delay p. This equation expresses absolute delay, unlike its counterpart, Equation 1.5, where delay is measured in *delay units*. Absolute delay and delay units are related by the time, τ, that is characteristic of the fabrication process. It is the delay of an ideal inverter with electrical effort of 1 and no parasitic delay. With more accurate transistor models and a reformulation of Equation 3.4, we could develop an analytic value for τ, expressed in terms of transistor length and width, gate oxide thickness, mobility, and other process parameters. We shall use an alternative approach, extracting the value of τ from suitable test circuits (see Section 5.1).

The logical effort, given by Equation 3.8, is determined by the circuit topology of the template for the logic gate, and is independent of the scale factor α. In effect, the logical effort compares the characteristic RC time constant of a logic gate with that of an inverter. Note that the logical effort of an inverter is chosen to be 1.

The electrical effort, defined by Equation 3.9, is just the ratio of the load capacitance of the logic gate to the capacitance of a particular input. This is the same as the definition in Equation 1.4. Observe that the size of the transistors used in the logic gate influences the electrical effort, because it determines the gate's input capacitance. This is the only remnant of the scale factor α.

Finally, Equation 3.10 defines the parasitic delay of the logic gate. Because this equation is independent of the logic gate's scale, α, it represents a fixed delay associated with the operation of the gate, irrespective of its size or load. Observe that for an inverter, the parasitic delay, p is the ratio of the parasitic capacitance to the input capacitance.

The linear relationship between delay and load expressed in Equation 3.6 is a more general result than the formulation of our model might suggest. Although our derivation has assumed that transistors behave like resistors, we would obtain the same linear relationship if we had assumed that transistors are current sources. In fact, our result is correct for any model of transistor behavior that combines a current source and a resistor, and thus handles both the linear and saturated regions

of transistor behavior. If we use a resistor model of transistors, their output current follows an exponential waveform that distorts only by stretching linearly in time for different values of capacitance and transistor width. If we use a current source model of the transistors, their output current follows a sawtooth waveform form that also distorts only by stretching in time for different values of capacitance and width.

Actually, Equation 3.6 requires only that delay grow linearly with load and diminish linearly as the widths of transistors are scaled. The exponential behavior of the output voltage in the simple model is described by a differential equation relating the rate of change of output voltage to the value of the output voltage. As the output voltage approaches its final value, its rate of change decreases because of the smaller current provided by the resistors. If any of the parameters we have assumed to be constant vary instead with output voltage, the differential equation becomes more complex, but its solution retains the same character. For example, if the capacitance of the transistor gates that form the driven load depends on their voltage, as it really does, the behavior of the output voltage will be distorted from exponential, but it will not change its general character. Similarly, if the current through the transistors depends on their drain to source voltage, as it really does, the behavior of the output voltage will be distorted from exponential, but again will not change its general character.

Some effects that the model ignores have little effect on its application to the method of logical effort. One of the most important is the variation in output current because of different input gate voltages, which leads to variations in the delay of a logic gate due to different risetimes of input signals. Long input risetimes increase the delay of the logic gate because the pullup and pulldown networks are not switched fully on or off while the input voltage is near the switching threshold. If all risetimes are equal, our simple model again holds because all logic gates will exhibit identical charging current waveforms and thus the same output voltage waveforms. Because the method of logical effort leads to nearly equal risetimes by equalizing effort borne by all logic gates, we are justified in omitting risetime effects from Equation 3.6.

Further evidence to support the model is obtained from detailed circuit simulations, described in Section 5.1. Although the delay model is very simple, it is quite accurate when it is suitably calibrated. It is, indeed, the basis of models used by most static timing analyzers.

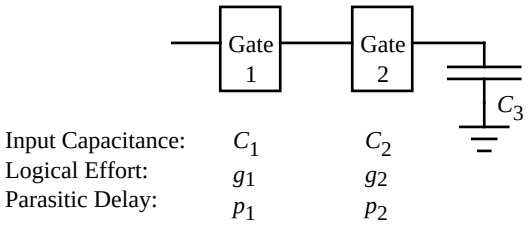


Figure 3.3: Generic two stage path.

3.3 Minimizing delay along a path

The delay model for a single logic gate leads to a method for minimizing the delay in a sequence of logic gates connected in series. The key result is that path delay is minimized when the effort borne by each logic gate along the path is the same.

Consider the two stage path in Figure 3.3. The path's input capacitance is C_1, the input capacitance of the first stage. Capacitance C_3 loads the second stage. According to Equation 3.6, the total delay, measured in units of τ, is:

$$D = (g_1 h_1 + p_1) + (g_2 h_2 + p_2) \tag{3.11}$$

While the logical efforts, g_1 and g_2, and parasitic delays, p_1 and p_2, in this equation are fixed, the electrical efforts in each stage can be adjusted to minimize the delay. The electrical efforts are constrained, however, by the input capacitance C_1 and the load capacitance C_3, which are fixed:

$$\begin{aligned} h_1 &= C_2/C_1 \\ h_2 &= C_3/C_2 \end{aligned}$$

and since the branching effort is 1

$$h_1 h_2 = C_3/C_1 = H$$

The path electrical effort, H, is a given constant that we cannot adjust. Substituting $h_2 = H/h_1$ into Equation 3.11, we obtain

$$D = (g_1 h_1 + p_1) + (g_2 H/h_1 + p_2) \tag{3.12}$$

To minimize D, we take the partial derivative with respect to h_1, set the result equal to zero, and solve for h_1:

$$\frac{\partial D}{\partial h_1} = g_1 - g_2 H/h_1^2 = 0 \tag{3.13}$$

$$g_1 h_1 = g_2 h_2 \tag{3.14}$$

Thus, delay is minimized when each stage bears the same effort, which is the product of the logical effort and the electrical effort. This result is independent of the scale of the circuits and of the parasitic delays. It does not say that the delays in the two stages will be equal—the delays will differ if the parasitic delays differ.

This result generalizes to paths with any number of stages (Exercise 3-3) and to paths that include branching effort. The fastest design always equalizes effort in each stage.

Let us now see how to compute the effort in each stage. We have for a path of length N:

$$h_1 h_2 \cdots h_N = BH \tag{3.15}$$

where the path electrical effort H is the ratio of the load on the last stage to the input capacitance of the first stage and the branching effort B is the product of the branching efforts at each stage. Define the path logical effort to be:

$$g_1 g_2 \cdots g_N = G \tag{3.16}$$

Multiplying these two equations together, we obtain the path effort, F:

$$(g_1 h_1)(g_2 h_2) \cdots (g_N h_N) = GBH = F \tag{3.17}$$

To obtain minimum delay, the N factors on the left must be equal, so that each stage bears the same effort $\hat{f} = gh$. Thus the equation can be rewritten as:

$$\hat{f}^N = F \tag{3.18}$$

or

$$\hat{f} = F^{1/N} \tag{3.19}$$

Given G, B, H, and N for the path, we can compute F and therefore the stage effort, $\hat{f}$, that achieves least delay. (Recall that our notation places a hat over a quantity chosen to achieve least path delay.) Now we can solve for the electrical effort h_i of each stage: $h_i = \hat{f}/g_i$. To calculate transistor sizes, we work backward or forward along the path, choosing transistor sizes to obtain the required electrical effort in each stage. This is the procedure outlined in Section 1.2.

The path delay obtained by this optimization procedure is

$$\hat{D} = \sum (g_i h_i + p_i) = N F^{1/N} + P \tag{3.20}$$

Although the parasitic delays do not affect the procedure for designing the path to obtain least delay, they do affect the actual delay obtained. We will see in the next section that parasitic delay also influences the best number of stages in a path.

3.4 Choosing the length of a path

Although equalizing the effort borne by each stage in a path minimizes delay for a given path, the delay can sometimes be reduced further by adjusting the number of stages in the path. This optimization is also a straightforward result of our delay model.

Consider a path of logic gates containing n_1 stages, to which we append n_2 additional inverters to obtain a path with a total of $N = n_1 + n_2$ stages. We will assume that the original n_1 stages cannot be altered except by scaling because they perform necessary logic functions, while the number of inverters can be altered if necessary to reduce delay. Although preserving the correct logic function requires that an even number of inverters be used, we will assume that an odd number of inverters can be accommodated by changing the logic function as necessary. We will assume that the path effort $F = GBH$ is known: the logical and branching efforts are properties of the n_1 logic stages that will not be altered by adding inverters, and the electrical effort is determined by the input and load capacitances required.

The minimum delay of the N stages is the sum of the delay in the logic stages and in the inverter stages:

$$\hat{D} = NF^{1/N} + \left(\sum_{i=1}^{n_1} p_i\right) + (N - n_1)p_{inv} \tag{3.21}$$

The first term is the delay obtained by distributing effort equally among the N stages, as shown in the preceding section. The second term is the parasitic delay of the logic stages, and the third term is the parasitic delay of the inverters. Differentiating this expression with respect to N and setting the result to zero, we obtain:

$$\frac{\partial \hat{D}}{\partial N} = -F^{1/N}\ln(F^{1/N}) + F^{1/N} + p_{inv} = 0 \tag{3.22}$$

Let us define the solution to this equation to be $\hat{N}$, the number of stages to use to obtain least delay. If we define $\rho = F^{1/\hat{N}}$ to be the effort borne by each stage when the number of stages is chosen to minimize delay, the solution of the equation can be expressed as:

$$p_{inv} + \rho(1 - \ln \rho) = 0 \tag{3.23}$$

In other words, the fastest design is one in which each stage along a path bears an effort equal to ρ, where ρ is a solution of Equation 3.23. Thus we call ρ the *best stage effort.*

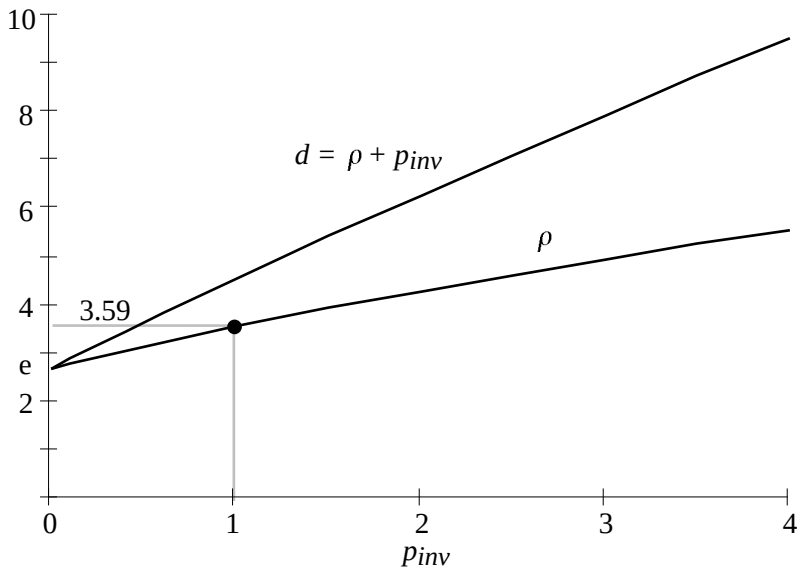


Figure 3.4: Best effort per stage, ρ, and corresponding best stage delay $\rho + p_{inv}$, as a function of p_{inv}. Calculated from Equation 3.23.

It is important to understand the relationship between ρ and $\hat{f}$, both of which appear to specify the stage effort required to achieve least delay. The expressions for $\hat{f}$, such as Equation 3.19, determine the best stage effort when the number of stages, N, is known. By contrast, the value ρ, which is a constant independent of the properties of a path, represents the stage effort that will result when a path uses the number of stages required to achieve least delay.

Equation 3.23 shows that the best effort, ρ, is a function of the parasitic delay of an inverter. This result has an intuitive explanation. The stray capacitance of the logic gates in the network is fixed—you can't do much about it, and it simply adds a fixed delay to the path. Adjusting the sizes of the logic gates will change their effort delay, but not the delay contribution due to their parasitic delay. When you add an inverter as a gain element in the hope of speeding up the circuit, you need to know its actual delay, including parasitic contributions, to compare the delay of the extra inverter to the improvement in delay of the rest of the circuit. As p_{inv} grows, adding inverters becomes less advantageous because their extra stray load blunts the improvement they might otherwise offer. Therefore, the best number of stages diminishes.

Although Equation 3.23 has no closed-form solution, it is not hard to solve

for values of ρ given values of p_{inv}. Figure 3.4 shows the solution as a function of an inverter's parasitic delay. Note that if we assume that the parasitic delay of an inverter is zero, then $\rho = e = 2.718$; this is the familiar result when parasitic delay is ignored [6]. Although Equation 3.23 is nonlinear, the equation:

$$\rho = 0.71 p_{inv} + 2.82 \tag{3.24}$$

fits it well over the range of reasonable inverter parasitics. For most of our examples, we shall assume that $p_{inv} = 1.0$ and thus that $\rho = 3.59$.

The quantity ρ is sometimes called the *best step-up ratio*, because it is the ratio of the sizes of successive inverters in a string of inverters designed to drive a large capacitive load. Figure 3.4 shows the stage delay obtained when the best step-up ratio is used. From Equation 3.6, the stage delay is the sum of the effort and the parasitic delay.

Actual designs will require us to choose a step-up ratio that differs somewhat from ρ because the design must use an integral number of stages. Given the path effort F, we must find the number of stages $\hat{N}$ that gives the least delay; this result will have a stage delay close to ρ. Table 3.1 shows how to select $\hat{N}$, given the effort F and several values of the parasitic delay of an inverter. The values of F in the table satisfy $\hat{N}(F^{1/\hat{N}} + p_{inv}) = (\hat{N}+1)(F^{1/(\hat{N}+1)} + p_{inv})$. These are the values of path effort for which the best $\hat{N}$-stage design exhibits just as much delay as the best $(\hat{N}+1)$-stage design.

Some designs will not speed up when inverters are added. For example, if the path effort is 10 and there are three stages of logic, the logic network already has more stages than the optimum, which is two stages. In this case, we might try to consolidate the three stages of logic into two; this may result in a speedup.

Equations 3.18 and 3.19 allow us to derive equations that approximate the number of stages and delays when F is large. Using the fact that $F = \rho^{\hat{N}}$, we find:

$$\hat{N} \approx \frac{\ln F}{\ln \rho} = \log_\rho F \tag{3.25}$$

$$D \approx \hat{N}\rho + \sum p_i \tag{3.26}$$

As the effort gets large, we see that the stage delay approaches $\rho + p$. For an inverter chain, these two equations can be combined to read:

$$D \approx \frac{\ln F}{\ln \rho}(\rho + p_{inv}) \tag{3.27}$$

$\hat{N}$	$p_{inv} = 0.0$	$p_{inv} = 0.6$	$p_{inv} = 0.8$	$p_{inv} = 1.0$
1	0	0	0	0
2	4.0	5.13	5.48	5.83
3	11.4	17.7	20.0	22.3
4	31.6	59.4	70.4	82.2
5	86.7	196	245	300
6	237	647	848	1090
7	648	2130	2930	3920
8	1770	6980	10100	14200
9	4820	22900	34700	51000
10	13100	74900	120000	184000
11	35700	245000	411000	661000
12	97300	802000	1410000	2380000
13	265000	2620000	4860000	8560000
14	720000	8580000	16700000	30800000
15	1960000	28000000	57400000	111000000
	5330000	91700000	197000000	398000000

Table 3.1: Table of ranges of path effort, F, and the best number of stages, $\hat{N}$.

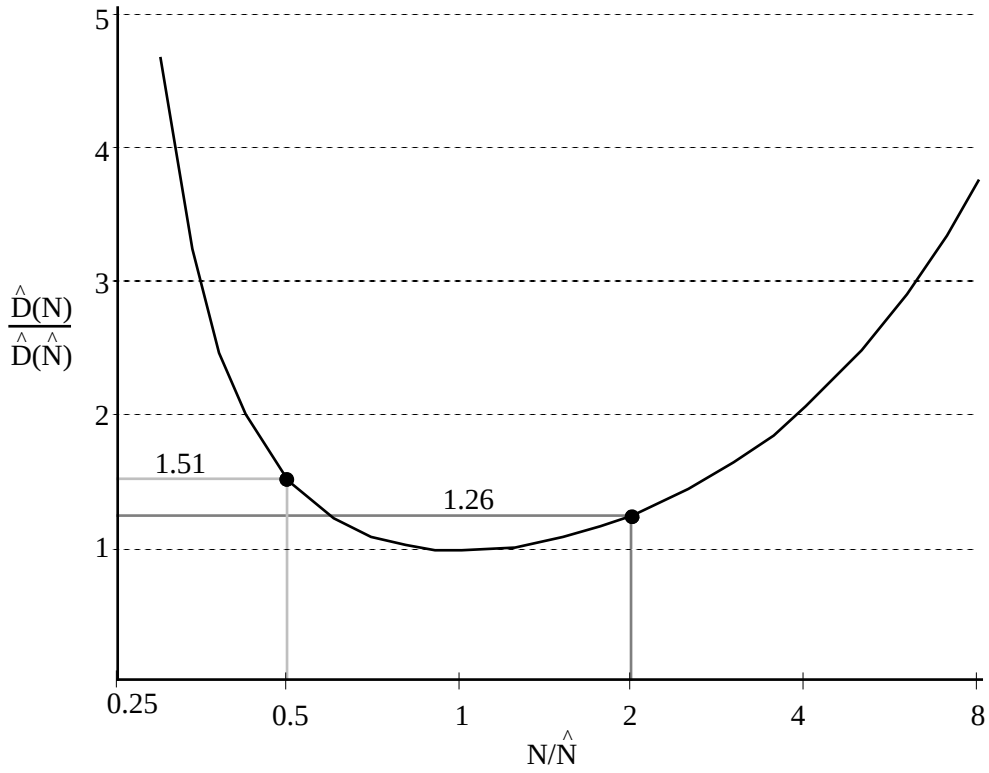


Figure 3.5: The relative delay compared to the best possible, as a function of the relative error in the number of stages used, $N/\hat{N}$. Assumes $p_{inv} = 1$.

When a stage effort of 4 is used, this reduces to $D = \log_4 F$ fanout-of-4 (FO4) inverter delays, where an FO4 delay is 5τ. We will see in the next section that delay is is almost independent of stage effort for stage efforts near optimal, so this delay formula is a good estimate of the delay of an inverter chain using any reasonable stage effort. Moreover, it is a handy estimate of the delay of any circuit with path effort F. Paths with more complex gates will have higher parasitics, but to first order gate delay dominates and the estimate is useful for quickly comparing different circuit topologies by computing only the path effort. Finally, FO4 delays are a useful way to express delay in a process-independent way because most designers know the delay of a fanout-of-4 inverter in their process and can therefore estimate how your circuit will scale to their process.

3.5 Using the wrong number of stages

It is interesting to ask how much the delay for a properly optimized circuit is changed by using the wrong number of stages. The answer, as shown in Figure 3.5, is that delay is quite insensitive to the number of stages, provided the deviation from optimum is not too large.

To develop the curve in the figure, we start by assuming that the number of stages is wrong by a factor s, i.e., the number of stages is $s\hat{N}$, where $\hat{N}$ is the best number to use. The delay can be expressed as a function of N:

$$D(N) = N(F^{1/N} + p) \tag{3.28}$$

where we assume the parasitic delay of each stage is the same. Let r be the ratio of the delay when using $s\hat{N}$ stages to the delay when using the best number of stages, $\hat{N}$:

$$r = D(s\hat{N})/D(\hat{N}) \tag{3.29}$$

Since $\hat{N}$ is best, we know that $F = \rho^{\hat{N}}$. Solving for r, we obtain:

$$r = \frac{s(\rho^{1/s} + p)}{\rho + p} \tag{3.30}$$

This is the relationship plotted in Figure 3.5 for $p = 1$ and thus $\rho = 3.59$.

As the graph shows, doubling the number of stages from optimum increases the delay only 26%. Using half as many stages as the optimum increases the delay 51%. Thus one should not slavishly stick to exactly the correct number of stages, and it is slightly better to err in the direction of using more stages than the optimum. A stage or two more or less in a design with many stages will make little difference, provided proper transistor sizes are used. Only when very few stages are required does a change of one or two stages make a large difference.

A designer often faces the problem of deciding whether it would be beneficial to change the number of stages in an existing circuit. This can easily be done by calculating the stage effort. If the effort is between 2 and 8, the design is within 35% of best delay. If the effort is between 2.4 and 6, the design is within 15% of best delay. Therefore, there is little benefit in modifying a circuit unless the stage effort is grossly high or low.

Targeting a stage effort of 4 is convenient because 4 is a round number and it is easy to compute the desired number of stages mentally. For values of p_{inv} between 0.7 and 2.5, a stage effort of 4 also produces delays within 2% of minimum.

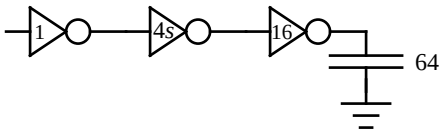


Figure 3.6: A string of inverters with a missized middle stage.

3.6 Using the wrong gate size

It is also interesting to ask how much the delay for a properly optimized circuit is changed if some of the gates are missized. For example, a standard cell library has only a discrete set of gate sizes so it is not always possible to use exactly the desired size.

Consider the effect of missizing a stage in a string of inverters. The string in Figure 3.6 has a best stage effort of 4, but the middle inverter is is missized so that it has an actual stage effort of $4/s$ while the predecessor has an actual stage effort of $4s$.

Figure 3.7 plots the delay of the string relative to the best possible delay, as a function of s. The figure shows that for values of s from 0.5 to 2, the actual delay is within 15% of minimum and for values of s from $2/3$ to 1.5 the actual delay is within 5% of minimum. Therefore, the designer has a great deal of freedom to select gate sizes different from those specified by the logical effort computation. This is the reason that standard cell libraries with a limited repetoir of gate sizes can achieve acceptable performance.

Since minor errors in gate sizes have almost no effect on overall delay, a designer can save time by making "back of the envelope" calculations of sizes to one or two significant figures [7]. With practice, most logical effort calculations can be done mentally.

3.7 Summary

This chapter has presented all of the major results of the method of logical effort. These are summarized as follows:

- The absolute delay in a single logic gate is modeled as

$$d = \tau(gh + p) \tag{3.31}$$

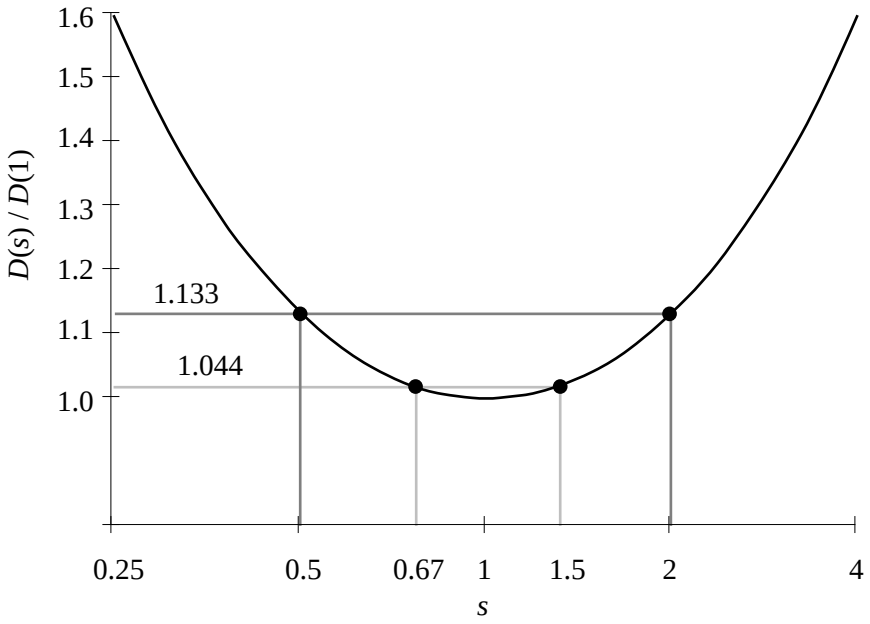


Figure 3.7: The relative delay compared to the best possible, as a function of s, the size error of a stage. Assumes $p_{inv} = 1$.

The next chapter shows how to estimate or measure the logical effort and parasitic delay of logic gates for a particular fabrication process, and how to measure τ.

- The least delay along a path is obtained when each logic gate bears the same effort. This result leads to the equation for delay along a path:

$$D = NF^{1/N} + \sum p_i \tag{3.32}$$

where F is the path effort.

- Delay along a path is least when each stage bears effort ρ, a quantity calculated from the parasitic delay of an inverter (Equation 3.23 and Figure 3.4). This in turn determines the best number of stages to use, for any path effort (Table 3.1). In practice, the stage effort deviates slightly from ρ because the number of stages, N, must be an integer.

- Select ρ about 4. Any value from 2 to 8 gives reasonable results and any value from 2.4 to 6 gives nearly optimal results, so you can be sloppy and still have a good design.

- Estimate the delay of a path from the path effort as $\log_4 F$ fanout-of-4 inverter delays.

3.8 Exercises

3-1 [25] Show that modeling transistors as current sources leads to the same basic results (Equations 3.6 to 3.10).

3-2 [30] Using process parameters from your favorite CMOS process, estimate values for κ and τ.

3-3 [30] Generalize the result of Section 3.3 to show that the least delay in a path of N stages results when all stages bear the same effort.

3-4 [25] Redo the analysis in Section 3.4 to choose the best number of stages N assuming we add 2-input NAND gates rather than inverters.

3-5 [30] One impediment to scaling each stage precisely is the resolution of widths supported by the lithographic equipment used in fabrication. Suppose the process could support only three distinct widths of each transistor type (n and p), but that you could choose these widths. What would you choose? How might you get the effect of widths greater than those chosen?

3-6 [15] If a logic string must be increased in length, the inverters can be added either before or after the logic gates, or between them. What practical considerations would cause one to choose one location over the other?

Chapter 4

Calculating the Logical Effort of Gates

The simplicity of the theory of logical effort follows from assigning to each kind of logic gate a number—its logical effort—that describes its drive capability relative to that of a reference inverter. The logical effort is independent of the actual size of the logic gate, allowing one to postpone detailed calculations of transistor sizes until after the logical effort analysis is complete.

Each logic gate is characterized by two quantities: its logical effort and its parasitic delay. These parameters may be determined in three ways:

- Using a few process parameters, one can estimate logical effort and parasitic delay as described in this chapter. The results are sufficiently accurate for most design work.
- Using test circuit simulations, the logical effort and parasitic delay can be simulated for various logic gates. This technique is explained in Chapter 5.
- Using fabricated test structures, logical effort and parasitic delay can be physically measured.

Before turning to methods of calculating logical effort, we present a discussion of different definitions and interpretations of logical effort. While these are all equivalent, in some sense, each offers a different perspective to the design task and each leads to different intuitions.

4.1 Definitions of logical effort

Logical effort captures enough information about a logic gate's topology—the network of transistors that connect the gate's output to the power supply and to ground—to determine the delay of the logic gate. In this section, we give three equivalent concrete definitions of logical effort.

Definition 4.1 *The logical effort of a logic gate is defined as the number of times worse it is at delivering output current than would be an inverter with identical input capacitance.*

Any topology required to perform logic makes a logic gate less able to deliver output current than an inverter with identical input capacitance. For one thing, a logic gate must have more transistors than an inverter, and so to maintain equal input capacitance, its transistors must be narrower on average and thus less able to conduct current than those of an inverter with identical input capacitance. If its topology requires transistors in parallel, a conservative estimate of its performance will assume that not all of them conduct at once, and therefore that they will not deliver as much current as could an inverter with identical input capacitance. If its topology requires transistors in series, it cannot possibly deliver as much current as could an inverter with identical input capacitance. Whatever the topology of a simple logic gate, its ability to deliver output current must be worse than an inverter with identical input capacitance. Logical effort is a measure of how much worse.

Definition 4.2 *The logical effort of a logic gate is defined as the ratio of its input capacitance to that of an inverter that delivers equal output current.*

This alternative definition is useful for computing the logical effort of a particular topology. To compute the logical effort of a logic gate, pick transistor sizes for it that make it as good at delivering output current as a standard inverter, and then tally up the input capacitance of each input signal. The ratio of this input capacitance to that of the standard inverter is the logical effort of that input to the logic gate. The logical effort of a logic gate will depend slightly on the mobilitiy ratio in the fabrication process used to build it. These calculations are shown in detail later in this chapter.

Definition 4.3 *The logical effort of a logic gate is defined as the slope of the gate's delay vs. fanout curve divided by the slope of an inverter's delay vs. fanout curve.*

This alternative definition suggests an easy way to measure the logical effort of any particular logic gate by experiements with real or simulated circuits of various fanouts.

4.2 Grouping input signals

Because logical effort relates the input capacitance to the output drive current available, a natural question arises: for a logic gate with multiple inputs, how many of the input signals should we consider when computing logical effort? It is useful to define several kinds of logical effort, depending on how input signals are grouped. In each case, we define an *input group* to contain the input signals that are relevant to the computation of logical effort:

- Logical effort *per input*, in which logical effort measures the effectiveness of a single input in controlling output current. The input group is the single input in question. All of the discussion in preceding chapters uses logical effort per input.
- Logical effort of a *bundle*, a group of related inputs. For example, a multiplexer requires true and complement select signals; this pair might be grouped into a bundle. Because bundles of complementary pairs of signals occur frequently in CMOS circuits, we adopt a special notation: $s*$ stands for a bundle containing the true signal s and the complement signal $\overline{s}$. The input group of a bundle contains all the signals in the bundle.
- *Total* logical effort, the logical effort of all inputs taken together. The input group contains all the input signals of the logic gate.

Terminology and context determine which kind of logical effort applies. The adjective "total" is always used when total logical effort is meant, while the other two cases are distinguished by the signals associated with them in context. "The total logical effort of a 2-input NAND gate" is the logical effort of both inputs taken together, while "the logical effort of a 2-input NAND gate" is the logical effort per input of one of its two inputs.

The logical effort of an input group is defined analogously to the logical effort per input, shown in the previous section. The analog of Definition 4.2 is: the logical effort g_b of an input group b is just

$$g_b = \frac{C_b}{C_{inv}} = \frac{\sum_b C_i}{C_{inv}} \tag{4.1}$$

where C_b is the combined input capacitance of every signal in the input group b, and C_{inv} is the input capacitance of an inverter designed to have the same drive capabilities as the logic gate whose logical effort we are calculating.

A consequence of Equation 4.1 is that the logical efforts associated with input groups sum in a straightforward way. The total logical effort is the sum of the logical effort per input of every input to the logic gate. The logical effort of a bundle is the sum of the logical effort per input of every signal in the bundle. Thus a logic gate can be viewed as having a certain total logical effort that can be allocated to its inputs according to their contribution to the gate's input capacitance.

4.3 Calculating logical effort

Definition 4.2 provides a convenient method for calculating the logical effort of a logic gate. We have but to design a gate that has the same current drive characteristics as a reference inverter, calculate the input capacitances of each signal, and apply Equation 4.1 to obtain the logical effort.

Because we compute the logical effort as a ratio of capacitances, the units we use to measure capacitance may be arbitrary. This observation simplifies the calculations enormously. First, assume that all transistors are of minimum length, so that a transistor's size is completely captured by its width, w. The capacitance of the transistor's gate is proportional to w and its ability to produce output current, or conductance, is also proportional to w. In most CMOS processes, pullup transistors must be wider than pulldown transistors to have the same conductance. $\mu = \mu_n/\mu_p$ is the ratio of PMOS to NMOS width in an inverter for equal conductance. γ is the actual ratio of PMOS to NMOS width in an inverter. For simplicity, we will often assume that $\gamma = \mu = 2$. Under this assumption, an inverter will have a pulldown transistor of width w and a pullup transistor of width $2w$, as shown in Figure 4.1a, so the total input capacitance can be said to be $3w$. In this chapter, we will also find general expressions for logical effort as a function of γ. In Chapter 9, we will consider the benefits of choosing $\gamma \neq \mu$.

Let us now design a 2-input NAND gate so that it has the same drive characteristics as an inverter with a pulldown of width 1 and a pullup of width 2. Figure 4.1b shows such a NAND gate. Because the two pulldown transistors of the NAND gate are in series, each must have twice the conductance of the inverter pulldown transistor so that the series connection has a conductance equal to that of the inverter pulldown transistor. Therefore, these transistors are twice as wide as the inverter pulldown transistor. This reasoning assumes that transistors in series

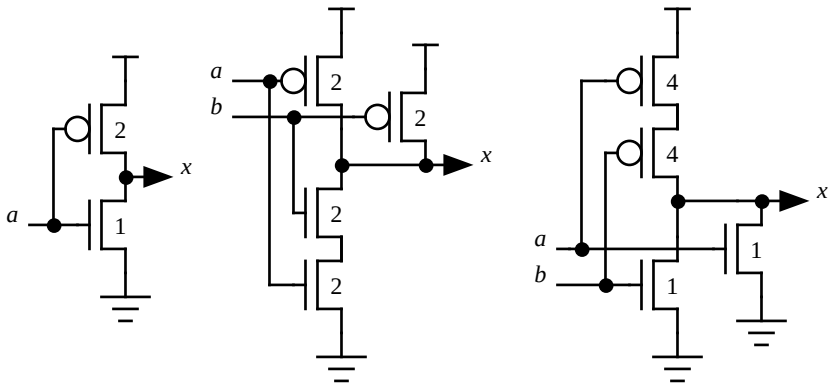


Figure 4.1: Simple gates. (a) The reference inverter. (b) A two-input NAND gate. (c) A two-input NOR gate.

obey Ohm's law for resistors in series. By contrast, each of the two pullup transistors in parallel need be only as large as the inverter pullup transistor to achieve the same drive as the reference inverter. Here we assume that if either input to the NAND gate is LOW, the output must be pulled HIGH, and so the output drive of the NAND gate must match that of the inverter even if only one of the two pullups is conducting.

We find the logical effort of the NAND gate in Figure 4.1b by extracting capacitances from the circuit schematic. The input capacitance of one input signal is the sum of the width of the pulldown transistor and the pullup transistor, or $2 + 2 = 4$. The input capacitance of the inverter with identical output drive is $C_{inv} = 1 + 2 = 3$. According to Equation 4.1, the logical effort per input of the 2-input NAND gate is therefore $g = 4/3$. Observe that both inputs of the NAND gate have identical logical efforts. Chapter 8 considers asymmetric gate designs favoring the logical effort of one input at the expense of another.

We designed the NOR gate in Figure 4.1c in a similar way. To obtain the same pulldown drive as the inverter, pulldown transistors one unit wide suffice. To obtain the same pullup drive, transistors four units wide are required, since two of them in series must be equivalent to one transistor two units wide in the inverter. Summing the input capacitance on one input, we find that the NOR gate has logical effort, $g = 5/3$. This is larger than the logical effort of the NAND gate because pullup transistors are less effective at generating output current than pulldown transistors. Were the two types of transistors similar, i.e., $\gamma = 1$, both

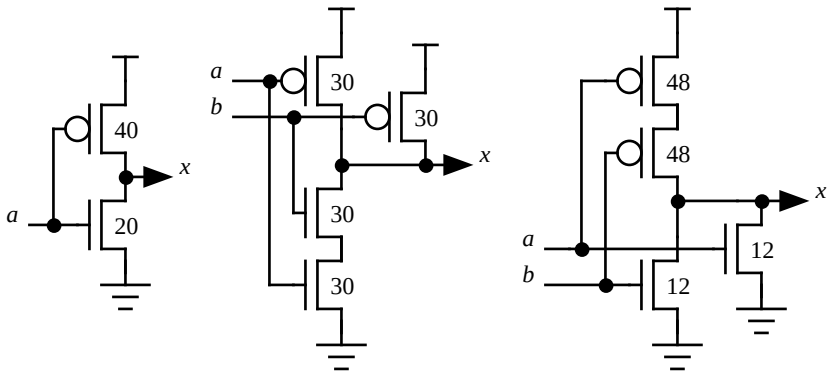


Figure 4.2: Simple gates with 60 input capacitance of 60 unit-sized transistors.

NAND and NOR gates would both have a logical effort of 1.5.

All of the sizing calculations in this monograph compute the input capacitance of gates. This capacitance is distributed among the transistors in the gate in the same proportions as are used when computing logical effort. For example, Figure 4.2 shows an inverter, NAND, and NOR gate, each with input capacitance equal to 60 unit-sized transistors.

When designing logic gates to produce the same output drive as the reference inverter, we are modeling CMOS transistors as pure resistors. If the transistor is off, the resistor has no conductance; if the transistor is on, it has a conductance proportional to its width. To determine the conductance of a transistor network, the conductances of the transistors are combined using the standard rules for calculating the conductance of a resistor network containing series and parallel resistor connections. While this model is only approximate, it characterizes logic gate performance well enough to design fast structures. More accurate values for logical effort can be obtained by simulating or measuring test circuits, as discussed in Chapter 5.

An important limitation of the model is that it does not account for velocity saturation. The velocity of carriers, and hence the current of a transistor, normally scales linearly with the electric field across the channel. When the field reaches a critical value, carrier velocity begins to saturate and no longer increases with field strength. The field across a single transistor is proportional to V_{DD}/L. In sub-micron processes, V_{DD} is usually scaled with L so that an NMOS transistor in an inverter is on the borderline of velocity saturation. PMOS transistors have lower mobility and thus are less prone to velocity saturation. Also, series NMOS transistors have a lower field across each transistor and therefore are less velocity

saturated. The effect of velocity saturation to remember is that series stacks of NMOS transistors in sub-micron processes tend to have less resistance than suggested by the model. Thus, structures with series NMOS transistors have slightly lower logical effort than our model predicts.

4.4 Asymmetric logic gates

Unlike the NAND and NOR gates, not all logic gates induce the same logical effort per input for all inputs. Equal logical effort per input is a consequence of the symmetries of the logic gates we have studied thus far. In this section, we will analyze an example in which the logical effort differs for different inputs.

Figure 4.3 shows one form of and-or-invert gate with an asymmetric configuration. The transistor widths in this gate have been chosen so that the output drive matches the reference inverter in Figure 4.1a: the pulldown structure is equivalent to a single pulldown transistor of width 1 and the pullup structure is equivalent to a single pullup transistor of width 2. The total logical effort of the gate, computed using Equation 4.1, is $17/3$.

The logical effort of the distinct inputs of the and-or-invert gate can be calculated individually. The logical effort per input for inputs a and b is $6/3 = 2$. The logical effort of the asymmetric input, c, is $5/3$. The c input has a slightly lower logical effort than the other inputs, reflecting the fact that the c input presents less capacitive load than the other inputs. Input c is "easier to drive" than the other two inputs.

Asymmetries in the logical effort of inputs arise in several different ways. The and-or-invert gate is topologically asymmetric, giving rise to unequal logical efforts of its inputs. Topologically symmetric gates, such as NAND and NOR, can be built with unequal transistor sizes to make them asymmetric so as to reduce the logical effort on some inputs, and thus reduce the logical effort along critical paths in a network. Other gates, such as XOR, have both asymmetric and symmetric forms, as discussed in Section 4.5.4. These techniques are explored further in Chapter 8.

4.5 Catalog of logic gates

The techniques for calculating logical effort are used in this section to develop Table 4.1. The expressions are slightly more general than those exhibited in earlier

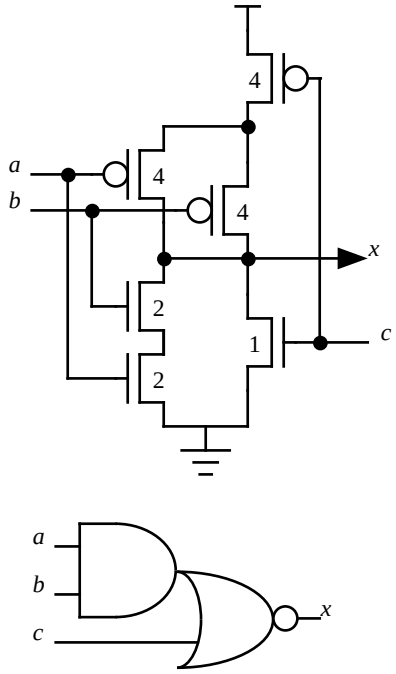


Figure 4.3: An asymmetric and-or-invert gate.

Gate type	Logical effort	Formula	$n=2$ $\gamma=2$	$n=3$ $\gamma=2$	$n=4$ $\gamma=2$
NAND	total	$\frac{n(n+\gamma)}{1+\gamma}$	8/3	5	8
	per input	$\frac{(n+\gamma)}{1+\gamma}$	4/3	5/3	2
NOR	total	$\frac{n(1+n\gamma)}{1+\gamma}$	10/3	7	12
	per input	$\frac{1+n\gamma}{1+\gamma}$	5/3	7/3	3
multiplexer	total	$4n$	8	12	16
	d, $s*$	2,2	2,2	2,2	2,2
XOR, XNOR, parity (symmetric)	total	$n^2 2^{n-1}$	8	36	128
	per bundle	$n2^{n-1}$	4	12	32
XOR, XNOR, parity (asymmetric)	total		8	24	48
	per bundle		4,4	6,12,6	8,16,16,8
majority (symmetric)	total			12	
	per input			4	
majority (asymmetric)	total			10	
	per input			4,4,2	
C-element	total	n^2	4	9	16
	per input	n	2	3	4
latch (dynamic)	total	4			
	d, $\phi*$	2,2			
upper bounds	total	$\frac{\gamma n^2 2^n}{1+\gamma}$	32/3	48	512/3
	per bundle	$\frac{\gamma n 2^n}{1+\gamma}$	16/3	16	128/3

Table 4.1: Summary of calculations of the logical effort of logic gates.

sections in two ways. First, the expressions apply to logic gates with an arbitrary number of inputs, n. Second, they use a parameter for the ratio of p-type to n-type transistor widths, so as to permit calculation of logical effort for gates fabricated with various CMOS processes. Whereas the reference inverter in Figure 4.1a has a pullup-to-pulldown width ratio of $2:1$, a ratio of $\gamma:1$ is used throughout this section. Each logic gate will be designed to have a pulldown drive equivalent to an n-type transistor of width 1 and a pullup drive equivalent to a p-type transistor of width γ.

4.5.1 NAND gate

A NAND gate with n inputs, designed to have the same output drive as the reference inverter, will have a series connection of pulldown transistors, each of width n, and a parallel connection of pullup transistors, each of width γ. Using Equation 4.1, the total logical effort is:

$$g_{tot} = \frac{n(n+\gamma)}{1+\gamma} \tag{4.2}$$

The logical effort per input is just $1/n$ of this value, because the input capacitance is equally distributed among the n inputs.

Table 4.1 includes the expressions for logical effort and calculations for several common cases: $\gamma = 2$, $n = 2, 3, 4$. Note from the equation that the logical effort changes only slightly for a wide range of γ: when γ ranges from 1 to 3, the total logical effort for $n = 2$ ranges from 3 to 2.5.

4.5.2 NOR gate

The n-input NOR gate consists of a parallel connection of pulldown transistors, each of width 1, and a series connection of pullup transistors, each of width $n\gamma$. The total logical effort is therefore:

$$g_{tot} = \frac{n(1+n\gamma)}{1+\gamma} \tag{4.3}$$

Again, the logical effort per input is just $1/n$ times this value. Table 4.1 includes examples of the logical effort of a NOR gate. For CMOS processes in which $\gamma > 1$, the logical effort of a NOR gate is greater than that of a NAND gate. If the CMOS fabrication process were perfectly symmetric, so that we could choose $\gamma = 1$, then the logical effort of NAND and NOR gates would be equal.

4.5.3 Multiplexers, tri-state inverters

An n-way inverting multiplexer is shown schematically in Figure 4.4. There are n data inputs, $d_1 \ldots d_n$, and n bundles of complementary select signals, $s*_1 \ldots s*_n$. Each data input is wired to a four-transistor *select arm,* which is in turn connected to the output c. To select input i, only bundle $s*_i$ is driven TRUE, which enables current to flow through the pullup or pulldown structures in the select arm associated with d_i.

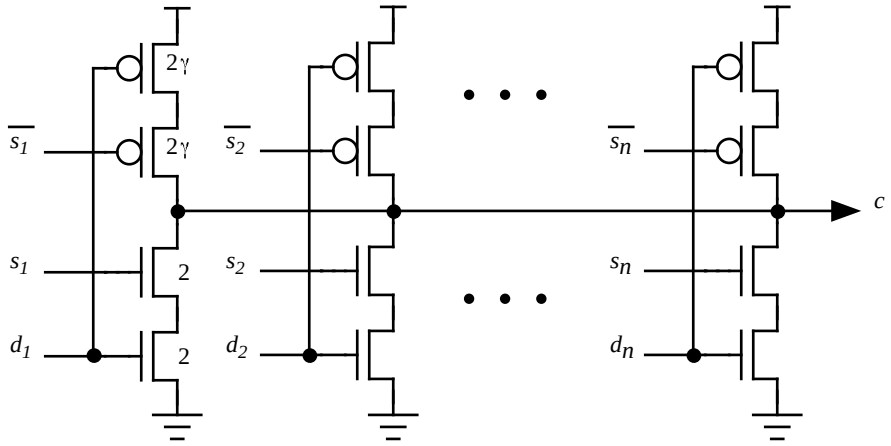


Figure 4.4: An n-way multiplexer. Each arm of the multiplexer has a data input d_i and a select bundle $s*_i$.

The total logical effort of a multiplexer is $n(4 + 4\gamma)/(1 + \gamma) = 4n$. The logical effort per data input is just $(2 + 2\gamma)/(1 + \gamma) = 2$, and the logical effort per select bundle is also 2. Note that the logical effort per input of a multiplexer does not depend on the number of inputs. Although this property suggests that large, fast, multiplexers could be built, stray capacitance in large multiplexers limits their growth. This problem is analyzed fully in Chapter 11. Also, increasing the number of multiplexer inputs tends to increase the logical effort of the select generation logic.

A single multiplexer arm is sometimes called a *tri-state inverter*. When a multiplexer is distributed across a bus, the individual arms are often drawn separately as tri-state inverters. Note that the logical efforts of the s and $\overline{s}$ inputs may differ.

4.5.4 XOR, XNOR, and parity gates

Figure 4.5 shows an XOR gate with two inputs, $a*$ and $b*$, and output c. The gate has two bundled inputs; the $a*$ bundle contains a complementary pair a and $\overline{a}$, and the $b*$ bundle contains b and $\overline{b}$. The total logical effort of the gate is $(8 + 8\gamma)/(1 + \gamma) = 8$. The logical effort per input is just $1/4$ this amount, or 2. The logical effort per input bundle is just the sum of the logical effort per input of the two inputs in the bundle, or 4.

The structure shown in Figure 4.5 can be generalized to compute the parity of n inputs. As an example, Figure 4.6a shows a 3-input XOR gate. The n-input gate

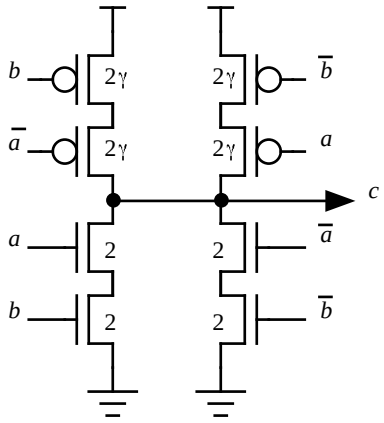


Figure 4.5: A two-input XOR gate, with input bundles $a*$ and $b*$, and output c.

will have 2^{n-1} pulldown chains, each with n transistors in series, each of width n. There will be 2^{n-1} pullup chains, each with n transistors in series, each of width $n\gamma$. Thus the total logical effort will be $2^{n-1}n(n+n\gamma)/(1+\gamma) = n^2 2^{n-1}$. The logical effort per input will be $1/(2n)$ times this figure, or $n2^{n-2}$, and the logical effort per input bundle will be $1/n$ times the total logical effort, or $n2^{n-1}$.

For $n = 3$ and above, symmetric structures such as the one shown in Figure 4.6a fail to yield least logical effort. Figure 4.6b shows a way to share some of the transistors in separate pullup and pulldown chains to reduce the logical effort. Repeating the calculation, we see that the total logical effort is 24, which is a substantial reduction from 36, the total logical effort of the symmetric structure in Figure 4.6a. In the asymmetric version, bundles $a*$ and $c*$ have a logical effort per bundle of 6. Bundle $b*$ has a logical effort of 12, which is the same as in the symmetric version because no transistors connected to b or $\overline{b}$ are shared in the asymmetric gate.

The XOR and parity gates can be altered slightly to produce an inverted output: simply interchange the a and $\overline{a}$ connections. Note that this transformation does not change any of the logical effort calculations.

4.5.5 Majority gate

Figure 4.7 shows two designs for an inverting 3-input majority gate. Its output is LOW when two or more of its inputs are HIGH. The symmetric design is shown in Figure 4.7a. The total logical effort is $(12+12\gamma)/(1+\gamma) = 12$, distributed evenly among the inputs. The logical effort per input is therefore 4. Figure 4.7b

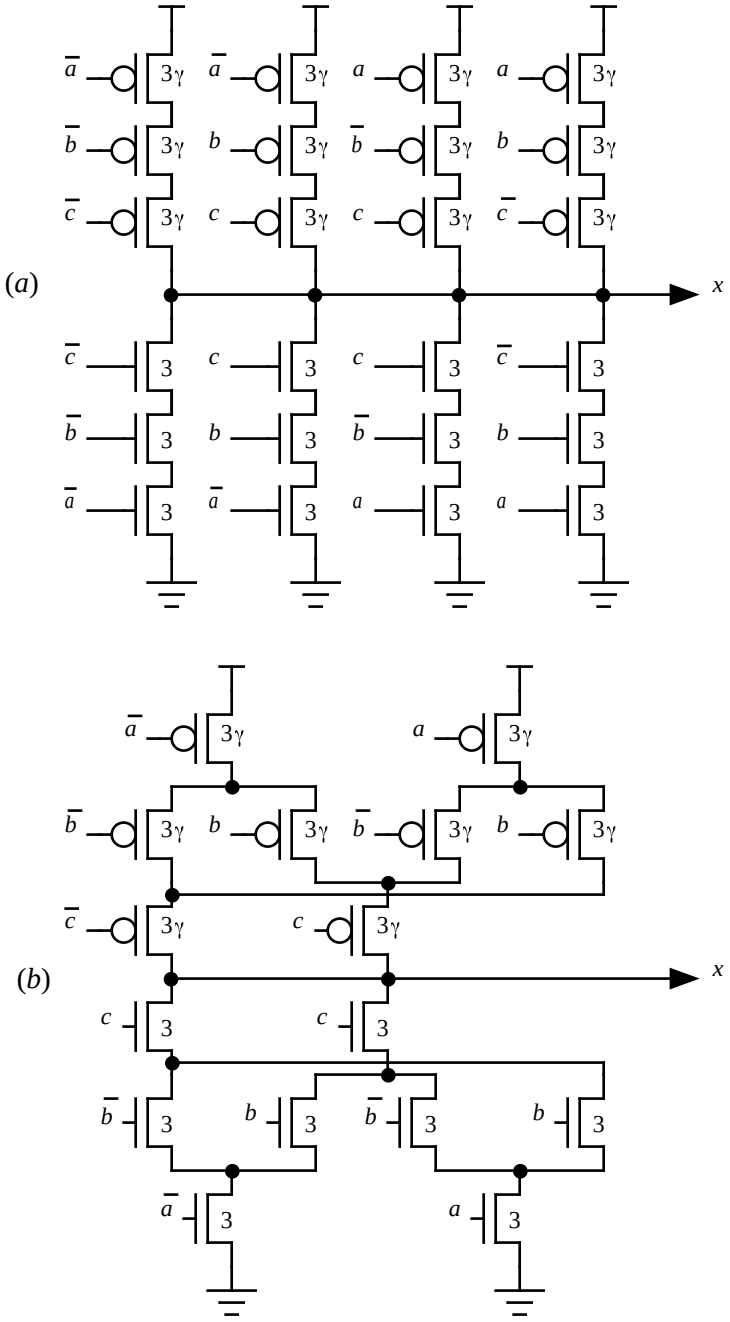


Figure 4.6: Two designs for three-input parity gates. (a) A symmetric design. (b) An asymmetric design with reduced logical effort.

shows an asymmetric design, which shares transistors as does the XOR design in Figure 4.6b. The total logical effort of this design is 10, and it is unevenly distributed among the inputs. The a input has a logical effort of 2, while the b and c inputs have logical efforts of 4 each.

4.5.6 Adder carry chain

Figure 4.8 shows one stage of a ripple-carry chain in an adder. The stage accepts carry C_{in} and delivers a carry out in inverted form on $\overline{C}_{out}$. The inputs g and $\overline{k}$ come from the two bits to be summed at this stage. The signal g is HIGH if this stage generates a new carry, forcing $\overline{C}_{out} = 0$. Similarly, $\overline{k}$ is LOW if this stage kills incomming carries, forcing $\overline{C}_{out} = 1$.

The total logical effort of this gate is $(5 + 5\gamma)/(1 + \gamma) = 5$. The logical effort per input for C_{in} is 2; for the g input it is $(1 + 2\gamma)/(1 + \gamma)$; and for the $\overline{k}$ input it is $(2 + \gamma)/(1 + \gamma)$.

4.5.7 Dynamic latch

Figure 4.9 shows a dynamic latch: when the clock signal ϕ is HIGH, and its complement $\overline{\phi}$ is LOW, the gate output q is set to the complement of the input d. The total logical effort of this gate is 4; the logical effort per input for d is 2, and the logical effort of the $\phi*$ bundle is also 2. Altering the latch to make it statically stable increases its logical effort slightly (see Exercise 4-2).

4.5.8 Dynamic Muller C-element

Figure 4.10 shows an inverting dynamic Muller C-element with two inputs. Although this gate is rarely seen in designs for synchronous systems, it is a staple of asynchronous system design. The behavior of the gate is as follows: When both inputs are HIGH, the output goes LOW; when both inputs go LOW, the output goes HIGH. In other conditions, the output retains its previous value—the C-element thus retains state. The total logical effort of this gate is 4, equally divided between the two inputs.

An n-input C-element can be formed in the obvious way, by making series pullup and pulldown chains of n transistors each. The width of a pulldown transistor is n, and of a pullup transistor is $n\gamma$. The total logical effort is thus $n(n + n\gamma)/(1 + \gamma) = n^2$, and the logical effort per input is just n.

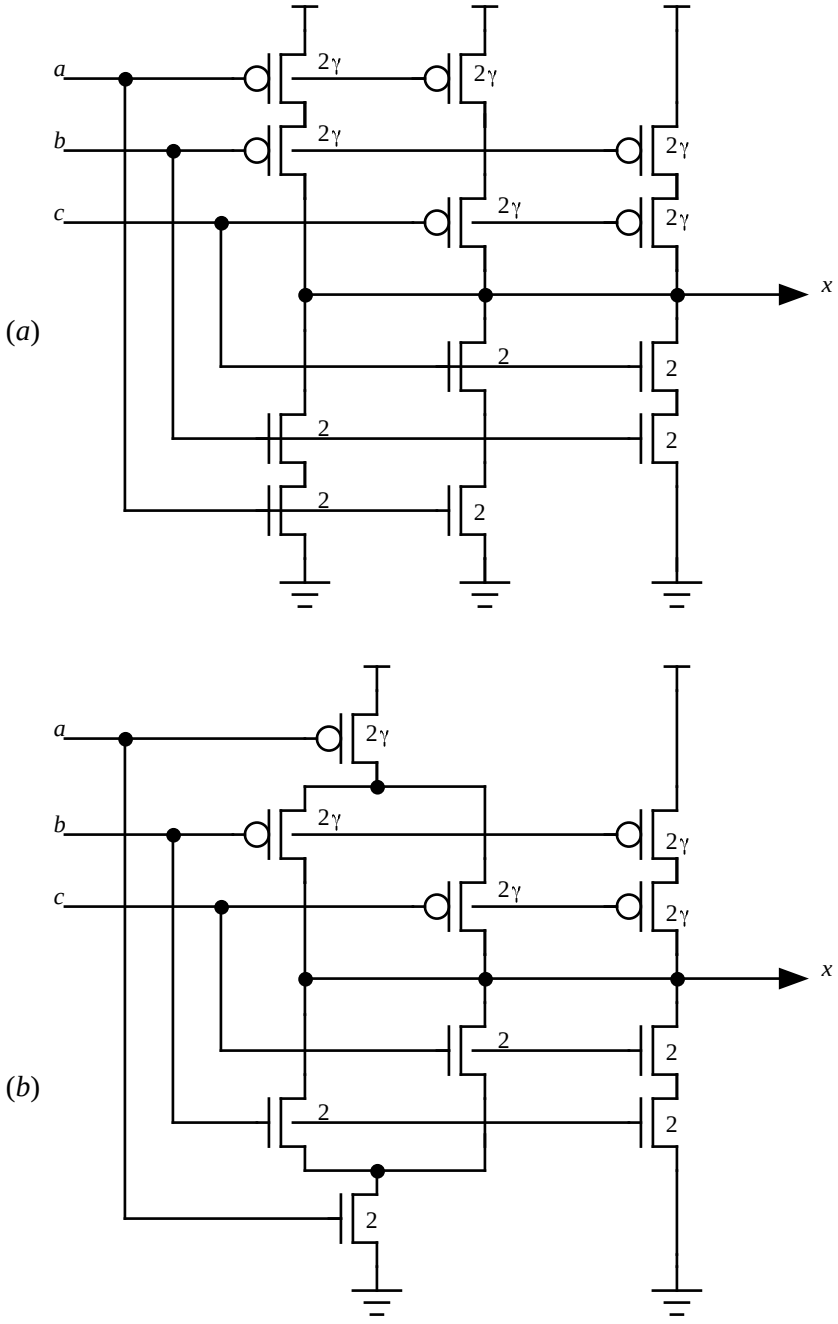


Figure 4.7: Two designs for three-input majority gates. (a) A symmetric design. (b) An asymmetric design with reduced logical effort.

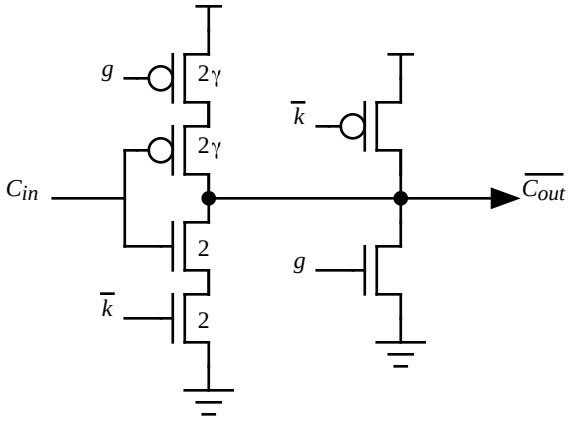


Figure 4.8: A carry-propagation gate. The carry arrives on C_{in} and leaves on $\overline{C}_{out}$. The g input is HIGH if a carry is generated at this stage, and the $\overline{k}$ input is LOW if a carry is killed at this stage.

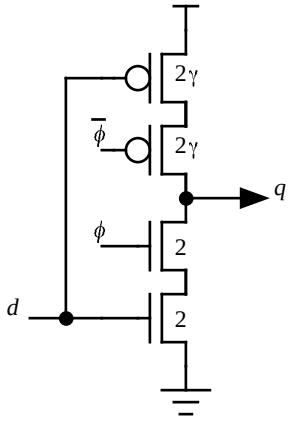


Figure 4.9: A dynamic latch with input d and output q. The clock bundle is $\phi*$.

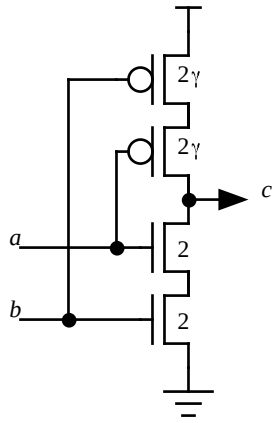


Figure 4.10: A two-input inverting dynamic Muller C-element. The inputs are a and b, and the output is c.

4.5.9 Upper bounds on logical effort

It is easy to establish an upper bound for the logical effort of a gate with n inputs. For any truth table, construct a gate with 2^n arms, each consisting of a series connection of n transistors, each of which receives the true or complement form of a different input. For entries in the truth table that require a LOW output, the series transistors in the corresponding arm are all n-type pulldown transistors and the series string bridges ground and the logic gate output. The transistor gates in the string receive inputs in such a way that the series connection conducts current when the input conditions for the truth table entry are met. For entries in the truth table that require a HIGH output, the series transistors are all p-type pullups and the series string spans the positive power supply and the logic gate output. The transistor gates receive the complement of the appropriate input. To design such a gate to have the same output drive as the reference inverter, each n-type transistor must have width n, and each p-type transistor must have width γn. To compute the worst-case logical effort, assume that $\gamma \geq 1$ and inputs are connected only to p-type transistors, which are larger than n-type transistors and so offer more load. Thus the worst-case input capacitance is $\gamma n^2 2^n$, and the worst-case logical effort is therefore $\gamma n^2 2^n/(1+\gamma)$.

This result shows that in the worst case, the logical effort of a logic gate grows exponentially with the number of inputs. These bounds are not particularly tight, and may perhaps be improved. Any improvement will hinge on reducing the number of transistors in a gate by sharing.

4.6 Estimating parasitic delay

Calculating the parasitic delay of logic gates is not as easy as calculating logical effort. The principal contribution to the parasitic capacitance is the capacitance of the diffused regions of transistors connected to the output signal. The capacitance of these regions will depend on their layout geometry and on process parameters. However, a crude approximation can be obtained by imagining that a transistor of width w has a diffused region of capacitance equal to wC_d associated with its source and an identical region associated with its drain. The constant C_d is a property of the fabrication process and the inverter layout.

This model allows us to compute the parasitic delay of an inverter. The output signal is connected to two diffused regions: the one associated with the pulldown of width 1 will have capacitance C_d, and the one associated with the pullup of width γ will have capacitance γC_d. The input capacitance of the inverter is likewise proportional to the transistor widths, but with a different constant of proportionality characteristic of transistor gate capacitance. Thus the input capacitance is $(1+\gamma)C_g$. The parasitic delay is the ratio of the parasitic capacitance to the input capacitance of the inverter, which is just $p_{inv} = C_d/C_g$. The two constants of proportionality can be determined from layout geometry and process parameters (see Exercise 4-10). We shall adopt a nominal value of $p_{inv} = 1.0$, which is representative of inverter designs. This quantity can be measured from test circuits, as shown in Section 5.1.

We can estimate the parasitic delay of logic gates from the inverter parameters. The delay will be greater than that of an inverter by the ratio of the total width of diffused regions connected to the output signal to the corresponding width of an inverter, provided the logic gate is designed to have the same output drive as the inverter. Thus we have

$$p = \left(\frac{\sum w_d}{1+\gamma}\right) p_{inv} \tag{4.4}$$

where w_d is the width of transistors connected to the logic gate's output. For this estimate to apply, we assume that transistor layouts in the logic gates are similar to those in the inverter. Note that this estimate ignores other stray capacitances in a logic gate, such as contributions from wiring and from diffused regions that lie between transistors that are connected in series.

This approximation can be applied to an n-input NAND gate, which has one pulldown transistor of width n and n pullup transistors of width γ connected to the output signal, so $p = np_{inv}$. An n-input NOR gate likewise has $p = np_{inv}$.

Gate type	Formula	Parasitic delay when $p_{inv} = 1.0$			
		$n = 1$	$n = 2$	$n = 3$	$n = 4$
inverter	p_{inv}	1			
NAND	np_{inv}		2	3	4
NOR	np_{inv}		2	3	4
multiplexer	$2np_{inv}$		4	6	8
XOR, XNOR, parity	$n2^{n-1}p_{inv}$		4	12	
majority	$6p_{inv}$			6	
C-element	np_{inv}		2	3	4
latch	$2p_{inv}$	2			

Table 4.2: Estimates of the parasitic delay of logic gates.

An n-way multiplexer has n pulldowns of width 2 and n pullups of width 2γ, so $p = 2np_{inv}$. Table 4.2 summarizes some of these results.

The parasitic estimation has a serious limitation in that it predicts linear scaling of delay with number of inputs. In actuality, the parasitic delay of a series stack of transistors increases quadratically with stack height because of internal diffusion and gate-source capacitances. The Elmore delay model [9] handles distributed RC networks and shows that stacks of more than about four series transistors are best broken up into multiple stages of shorter stacks. Since parasitics are so geometry-dependent, the best way to find parasitic delay is to simulate circuits with extracted layout data.

4.7 Properties of logical effort

The calculation of logical effort for a logic gate is a straightforward process:

- Design the logic gate, picking transistor sizes that make it as good a driver of output current as the reference inverter.
- The logical effort per input for a particular input is the ratio of the capacitance of that input to the total input capacitance of the reference inverter.
- The total logical effort of the gate is the sum of the logical efforts of all of its inputs.

Table 4.1 reveals a number of interesting properties. The effect of circuit topology on logical effort is generally more pronounced than the effect of fabrication

technology. For CMOS with $\gamma = 2$, the total logical effort for 2-input NAND and NOR gates is nearly, but not quite, three. If CMOS were exactly symmetric ($\gamma = 1$), the total logical effort for both NAND and NOR would be exactly three; the asymmetry of practical CMOS processes favors NAND gates over NOR gates.

In contrast to the weak dependence on γ, the logical effort of a gate depends strongly on the number of inputs. For example, the logical effort per input of an n-input NAND gate is $(n + \gamma)/(1 + \gamma)$, which clearly increases with n. When an additional input is added to a NAND gate, the logical effort of each of the existing inputs increases through no fault of its own. Thus the total logical effort of a logic gate includes a term that increases as the square of the number of inputs; and in the worst case, logical effort may increase exponentially with the number of inputs. When many inputs must be combined, this non-linear behavior forces the designer to choose carefully between single-stage logic gates with many inputs and multiple-stage trees of logic gates with fewer inputs per gate. Surprisingly, one logic gate escapes super-linear growth in logical effort—the multiplexer. This property makes it attractive for high fan-in selectors, which are analyzed in greater detail in Chapter 11.

The logical effort of gates covers a wide range. A two-input XOR gate has a total logical effort of 8, which is very large compared to the effort of NAND and NOR of about 3. The XOR circuit is also messy to lay out because the gates of its transistors interconnect with a criss-cross pattern. Are the large logical effort and the difficulty of layout related in some fundamental way? Whereas the output of most other logic functions changes only for certain transitions of the inputs, the XOR output changes for every input change. Is its large logical effort related in some way to this property?

The designs for logic gates we have shown in this chapter do not exhaust the possibilities. In Chapter 8, logic gates are designed with reduced logical effort for certain inputs that can lower the overall delay of a particular path through a network. In Chapter 9, we consider designs in which the rising and falling delays of logic gates differ, which saves space in CMOS and permits analysis of ratioed NMOS designs with the method of logical effort.

4.8 Exercises

4-1 [20] Show that Equation 4.1 corresponds to the definition of logical effort given in Equation 3.8.

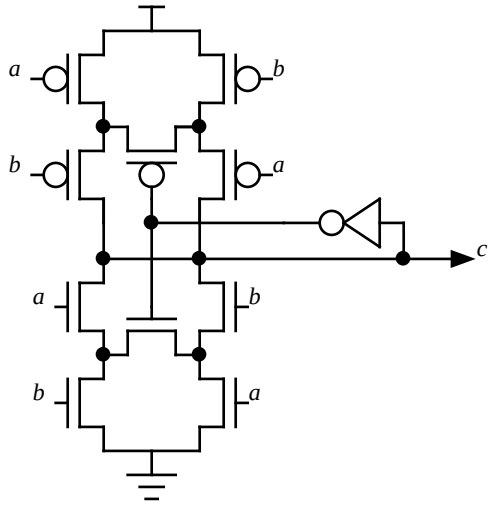


Figure 4.11: A static Muller C-element.

4-2 [20] Modify the latch shown in Figure 4.9 so that its output is statically stable, even when the clock is LOW. How big should the transistors be? What is the logical effort of the new circuit?

4-3 [20] In a fashion similar to Exercise 4-2, modify the dynamic C-element so that its output is static. How big should the transistors be? What is the logical effort of the new circuit?

4-4 [20] Another way to construct a static C-element is shown in Figure 4.11. What relative transistor sizes should be used? What is the logical effort of the gate?

4-5 [20] Figure 4.8 shows an adder element that inverts the polarity of the carry signal. A different design will be required for stages that accept a complemented carry input and generate a true carry output. Design such a circuit and calculate the logical effort of each input.

4-6 [10] In many CMOS processes the ratio of pullup to pulldown conductance, γ, is greater than 2. How high does γ have to be before the logical effort of NOR is twice that of NAND?

4-7 [20] The choice of transistor sizes for the inverter of Figure 4.1 was influ-

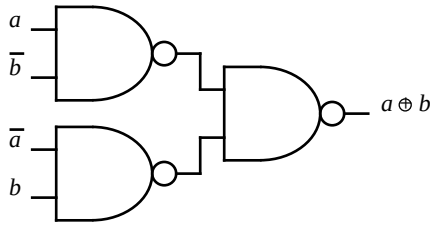


Figure 4.12: A two-stage XOR circuit.

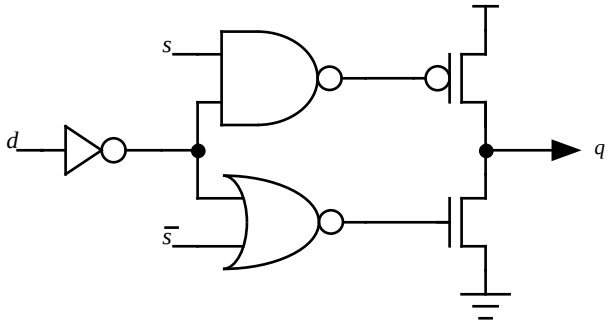


Figure 4.13: An inverting bus driver circuit, equivalent to the function of a tri-state inverter.

enced by the value of γ. Express the best pullup and pulldown transistor sizes in an inverter as a function of γ to obtain minimum delay in a two-inverter pair. Consider rising and falling delays separately.

4-8 [20] Compare the logical effort of a two-stage XOR circuit such as shown in Figure 4.12 with that of the single stage XOR of Figure 4.4. Under what circumstances is each perferable?

4-9 [20] Figure 4.13 shows a design for an inverting bus driver that achieves the same effect as a tri-state inverter. Compare the logical effort of the two circuits. Under whiat circumstances is each perferable?

4-10 [25] Measure the gate and diffusion capacitances of your process. From these values, estimate p_{inv}.

Chapter 5

Calibrating the Model

One can calculate the logical effort and parasitic delay of a logic gate from simple transistor models, as in the preceding chapter, or can obtain more accurate values by measuring the behavior of suitable test circuits. This chapter shows how to design and measure such circuits to obtain the two parameter values. The reader who wishes to skip this chapter may wish to glance at Table 5.1, which summarizes the characterization of one set of test circuits.

5.1 Calibration technique

We calibrate by measuring the delay of a logic gate as a function of its load—its electrical effort—and fiting a straight line to the results. Figure 5.1 shows simulated data for an inverter design. Since the logical effort of an inverter is 1, we expect from Equation 3.6 that the delay will be $d = \tau(h + p_{inv})$. The straight line that connects the points will have slope τ and will intercept the $h = 0$ axis at $d = \tau p_{inv}$. Thus the measurements yield values for τ and p_{inv}.

The measurements are an independent verification of the linear delay model on which the method of logical effort is based. While only two measurements are required to obtain values of τ and p_{inv}, more measurements will increase the precision of the result and increase our confidence in the linear model. The illustration shows four data points at different values of electrical effort, fitting a straight line very well.

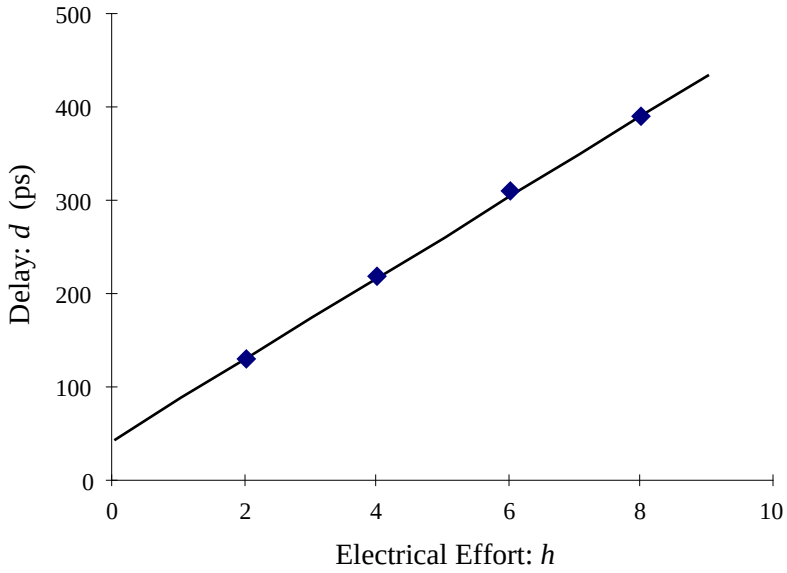


Figure 5.1: Simulated delay of inverters driving various loads. Results from 0.6μ, 3.3v process.

Figure 5.2 shows a similar set of data for a two-input NAND gate. The straight line in this case is fitted with the equation $d = \tau(g_{2nand}h + p_{2nand})$, where the value of τ was determined from the inverter characterization. This figure presents delay along the vertical axis in units of τ, using the value of τ computed from Figure 5.1. As a result, the slope of the fitted line will be the logical effort of the NAND gate and the intercept will be its parasitic delay. Similar simulations will calibrate an entire family of logic gates; some results are shown in Table 5.1.

Notice that the logical effort of NOR gates agrees fairly well with our model, but that the logical effort of NAND gates is lower than predicted. This can be attributed to velocity saturation, as discussed in Section 4.3.

The parasitic delay depends on layout and on the order of input switching. These effects are discussed later in this chapter.

The values in these figures and table were obtained through simulation. The remainder of this chapter discusses methods and pitfalls of logical effort characterization.

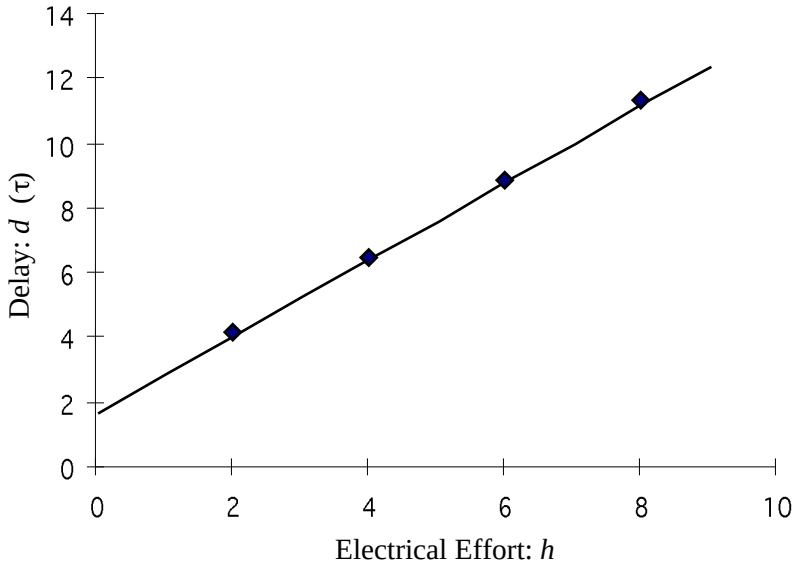


Figure 5.2: Simulated delay of two-input NAND gate driving various loads. Results from 0.6μ, 3.3v process. The vertical axis is marked in units of τ.

Gate type	Number of inputs	Logical effort		Parasitic delay	
		from simulation	model (Table 4.1)	from simulation	model (Table 4.2)
Inverter	1	1.00	1.00	1.08	1.00
NAND	2	1.18	1.33	1.36	2.00
	3	1.40	1.67	2.12	3.00
	4	1.66	2.00	2.39	4.00
NOR	2	1.58	1.66	1.98	2.00
	3	2.18	2.33	3.02	3.00
	4	2.81	3.00	3.95	4.00

Table 5.1: Values for logical effort and parasitic delay for several kinds of logic gates for a 0.6μ, 3.3v process with $\gamma = 2$. From simulation, $\tau = 43$ ps.

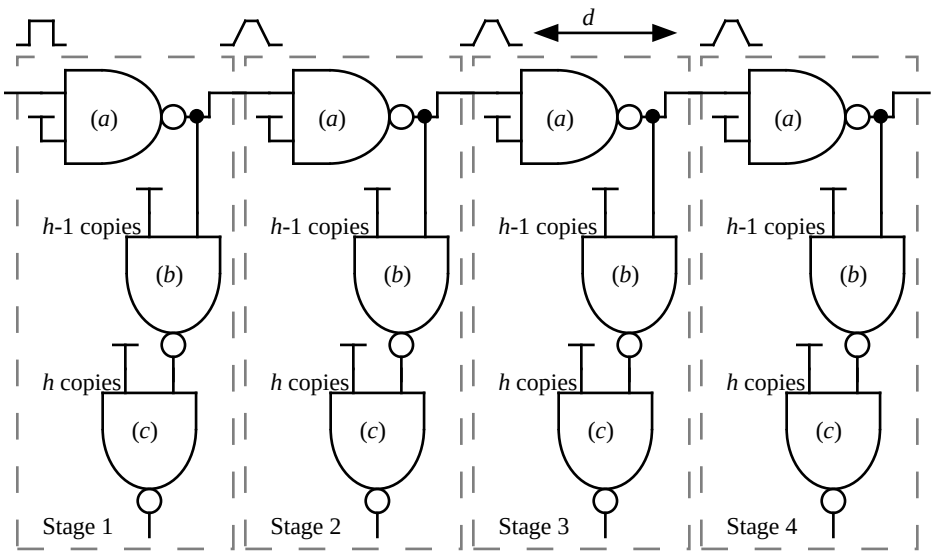


Figure 5.3: Test circuit for 2-input NAND calibration.

5.2 Designing test circuits

Designing a good test circuit is more subtle than one might initially imagine. A reasonable first attempt would be to load a gate with a capacitor, apply a step input, and measure the delay to the output crossing 50%. Such a circuit has two major problems. It does not account for the input slope dependence of delay, and it neglects the nonlinearity of MOS capacitors.

A better test circuit is shown in Figure 5.3 for a 2-input NAND. The circuit is divided into four stages. The first two stages are responsible for shaping the input slope. The third stage contains the gate being characterized. The final stage serves as a load on the gate. Each stage contains a primary gate (a), a load gate (b), and a load on the load (c)!

Gate (c) is necessary because of gate-drain overlap capacitance. If gate (c) were removed, the output of gate (b) would switch very rapidly. Because of the Miller effect, this would increase the effective input capacitance to gate (b). Simulation shows that this leads to an 8% overestimation of the delay of a fanout-of-4 inverter.

5.2.1 Rising, falling, and average delays

The rising and falling delays of gates usually are not equal. Which should we use? In Section 9.1, we show that considering only average gate delays is sufficient to minimize the average delay of a path. Therefore, we normally define the logical effort and parasitic delay of a gate to be the average of the values from the rising and falling transitions.

Occasionally it is useful to do a case analysis, considering rising and falling delays separately. The logical effort and parasitic delay for rising and falling transitions can be found from a curve fit in just the same way as for average delay. The results should still be normalized to the average delay of an inverter.

5.2.2 Choice of input

Designing test circuits with logic gates other than inverters requires deciding which input signals to use to propagate signals along the circuit. Our estimates of logical effort from Chapter 4 assumed that when transistors are in parallel exactly one turns on, while when transistors are in series, all series transistors turn on simultaneously to give a resistance of R through each. In contrast, real circuits tend to be modeled with a single latest input which arrives after all other inputs have settled. This leads to lower effective resistance through series transistors because the transistors with early inputs are fully turned on when the late input arrives and thus provide more current. Which of the two or more inputs to a logic gate should we choose as the late input? It turns out that the inputs have distinct properties, so we could find the logical effort and parasitic delay parameters for each input separately. The unused inputs must be wired so that the gate's output will be controlled by the single input: unused NAND inputs are wired HIGH, and unused NOR inputs are wired LOW.

The variation of logical effort and parasitic delay with different choices of input signal is tabulated in Table 5.2. An input signal is identified by a number that records the largest number of transistors between the transistors controlled by the signal and the output node of the logic gate, as shown in Figure 5.4. Signal 0 connects to two transistors that have drains connected directly to the output node. Signal 1 connects to two transistors, one of which is connected to the output node, but the other is one transistor away from the output node. This numbering scheme works for NOR and NAND gates equally. Signal 0 is called the innermost signal, while signal $n - 1$ is the outermost.

The parasitic delay of a gate changes greatly with choice of input, as shown in

Number of inputs	Input number	Logical effort	Parasitic delay
2	0	1.18	1.36
	1	1.11	1.89
3	0	1.40	2.12
	1	1.32	3.06
	2	1.28	3.64
4	0	1.66	2.39
	1	1.58	3.89
	2	1.49	5.04
	3	1.48	5.59

Table 5.2: Variation of the logical effort of NAND gate inputs for different choices of input signal. Data obtained from simulation of a 0.6μ 3.3 volt process.

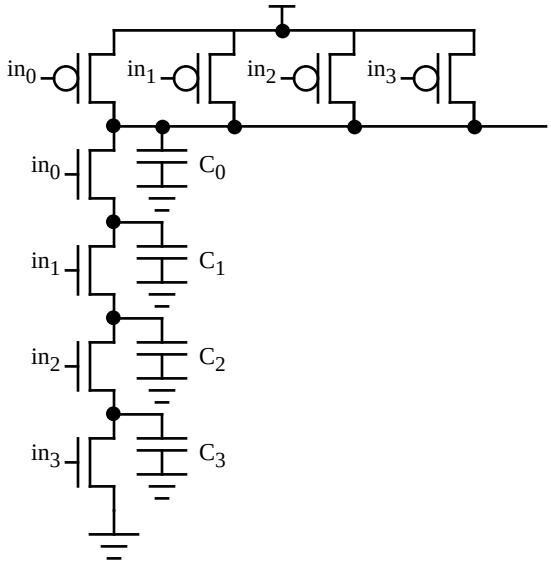


Figure 5.4: Input numbering and parasitic capacitances of a 4-input NAND.

Figure 5.4. When input 0 of a NAND gate rises, all the other NMOS transistors were already on and had discharged diffusion capacitances C_1–C_3 to ground. The gate has little parasitic delay because only the diffusion C_0 on the output node must switch. On the other hand, when the outer input (e.g. input 3 of a 4-input NAND) rises last, all the diffusion capacitances C_0–C_3 are initially charged up to near V_{DD}. Discharging this capacitance diverts some output current and increases the parasitic delay. Indeed, parasitic delay from the outer input scales quadratically with the number of inputs as discussed in Section 4.6. On account of parasitic delay, it is usually best to place the latest arriving signal on input 0.

Multiple-input gates also have somewhat lower logical effort than computed in Chapter 4 because only one input is switching, as we have seen in Table 5.1. The other transistors have already turned on and have a lower effective resistance than the switching transistor. If inputs to series transistors arrive simultaneously, the logical effort will be greater than these simulations have indicated.

5.2.3 Parasitic capacitance

It is essential to specify accurate diffusion capacitances to simulate realistic parasitic delays. Most SPICE models fail to account for diffusion capacitance unless explicitly requested. The diffusion capacitance is specified with the AS, AD, PS, and PD parameters corresponding to the area and perimeter of the diffusion. These dimensions are shown in Figure 5.5. The diffusion perimeter measures only the length of the junction between the diffusion region and the substrate; the boundary between diffusion and channel is not counted, because the diffusion wall capacitance at the edge of the transistor gate is deliberately reduced by the fabrication process.

Diffusion area and perimeter depend on layout. Diffusion nodes which contact to metal wires are larger than diffusion areas between series transistors without contacts. Good layouts share diffusion nodes wherever possible. Large cells can further reduce diffusion by folding transistors. Figure 5.6 shows the diffusion capacitances of a simple inverter, an inverter with folded transistors, and a NAND gate.

Ideally, wire capacitance should also be included in the parasitic estimate. This can be done by extracting parasitic values from actual cell layouts. The parasitic delays reported in this chapter include realistic diffusion capacitances, but omit wire capacitance.

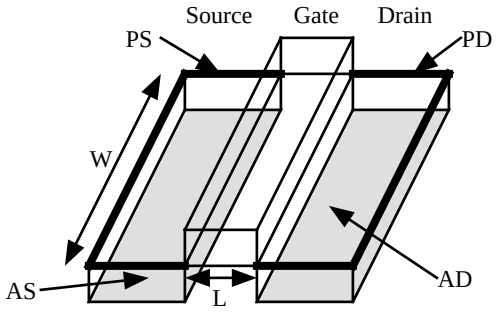


Figure 5.5: Simplified transistor structure illustrating diffusion area and perimeter for capacitance computation.

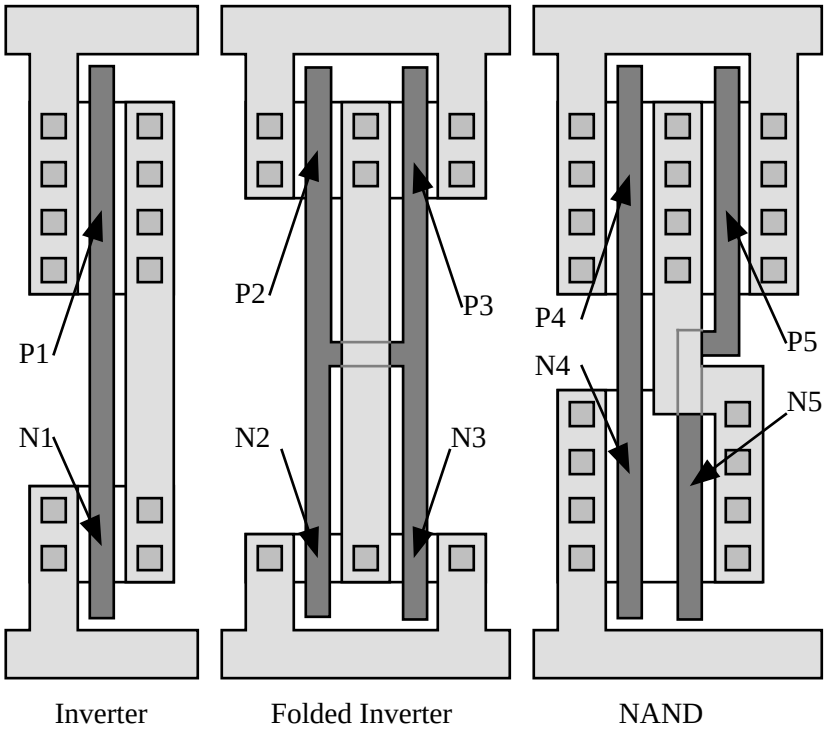


Figure 5.6: Layout of CMOS gates for measuring diffusion capacitance.

Transistor	W	AS	AD	PS	PD
N1	8	40	40	18	18
P1	16	80	80	26	26
N2	4	20	12	14	6
N3	4	20	12	14	6
P2	8	40	24	18	6
P3	8	40	24	18	6
N4	16	80	24	26	3
N5	16	24	80	3	26
P4	16	80	48	26	6
P5	16	80	48	26	6

Table 5.3: Diffusion area and perimeter capacitances of transistors in Figure 5.6. Reported in units of λ^2 and λ, respectively, where λ is half of the minimum drawn channel length. Layout according to MOSIS submicron design rules.

5.2.4 Process sensitivity

The value of τ depends on process, voltage, and temperature. Figure 5.7 shows how these parameters differ for a wide variety of processes and voltages at a nominal temperature of $70°C$.

Ideally, logical effort of a gate would be independent of process parameters, as was found in Chapter 4. In reality, effects like velocity saturation cause logical effort to differ slightly with process and operating conditions. Table 5.4 shows this variation.

Similarly, parasitic capacitance differs with process and environment. Table 5.5 shows this variation.

5.3 Other characterization methods

Simulation is usually sufficient to characterize a cell library. Sometimes, accurate SPICE models for the library are unavailable. In such a case, logical effort can still be estimated from vendor datasheets or measured from test chips.

Some cell libraries come with delay vs. fanout information in the data sheet, but lack simulation models. Logical effort can easily be extracted from the delay vs. fanout curves. Care must be taken to convert fanout to electrical effort if the fanout is expressed in terms of unit inverters rather than C_{out}/C_{in}.

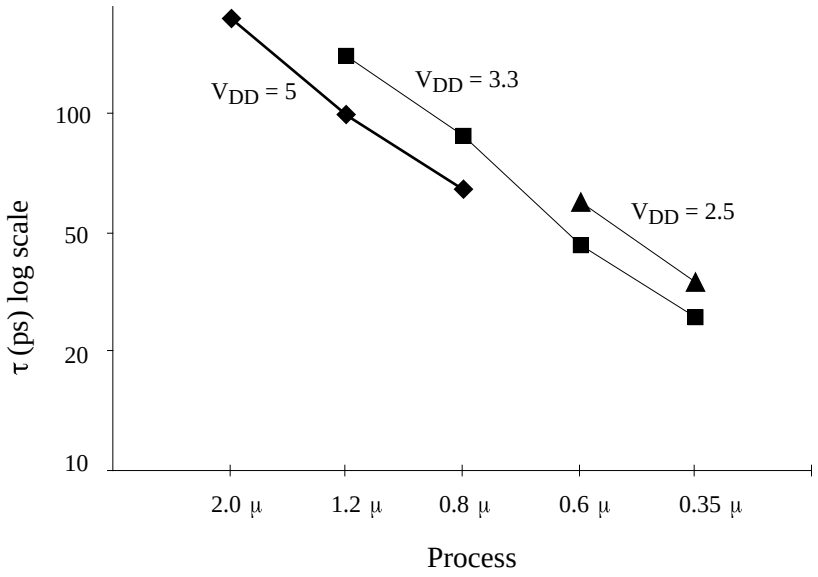


Figure 5.7: τ for various processes and voltages. Simulated using MOSIS SPICE parameters

Process	V_{DD}	NAND-2	NAND-3	NAND-4	NOR-2	NOR-3	NOR-4
2.0μ	5.0	1.16	1.35	1.55	1.58	2.13	2.69
1.2μ	5.0	1.20	1.41	1.65	1.50	2.00	2.47
1.2μ	3.3	1.24	1.48	1.74	1.51	2.04	2.59
0.8μ	5.0	1.15	1.32	1.53	1.51	2.00	2.55
0.8μ	3.3	1.17	1.36	1.58	1.50	2.07	2.60
0.6μ	3.3	1.18	1.40	1.66	1.58	2.18	2.81
0.6μ	2.5	1.17	1.42	1.68	1.55	2.13	2.78
0.35μ	3.3	1.17	1.37	1.57	1.54	2.03	2.61
0.35μ	2.5	1.20	1.42	1.65	1.57	2.15	2.61
Average		1.18	1.39	1.62	1.54	2.08	2.63

Table 5.4: Logical effort of gates for different processes and voltages.

Process	V_{DD}	inv	NAND-2	NAND-3	NAND-4	NOR-2	NOR-3	NOR-4
2.0μ	5.0	0.94	1.24	1.88	2.29	1.78	2.89	3.79
1.2μ	5.0	0.91	1.16	1.80	2.11	1.63	2.52	3.42
1.2μ	3.3	0.95	1.21	1.84	2.18	1.67	2.58	3.18
0.8μ	5.0	0.98	1.27	1.86	2.16	1.77	2.89	3.59
0.8μ	3.3	0.95	1.30	1.98	2.30	1.82	2.69	3.45
0.6μ	3.3	1.08	1.36	2.12	2.39	1.98	3.02	3.95
0.6μ	2.5	1.07	1.53	2.29	2.69	2.07	3.19	3.86
0.35μ	3.3	1.06	1.42	2.07	2.52	1.84	2.76	3.18
0.35μ	2.5	1.16	1.54	2.21	2.64	1.87	2.49	3.34
Average		1.01	1.34	2.01	2.36	1.83	2.78	3.53

Table 5.5: Parasitic delay of gates across process and voltage.

Logical effort can also be measured from fabricated chips by plotting the frequency of ring oscillators. The oscillator should contain an odd number of inverting stages. The frequency of the ring oscillator is related to the delay of the gate, as was discussed in Example 1.1. Oscillators with different fanouts provide data for the delay vs. electrical effort curves and thus values for logical effort and parasitic delay.

Care should be taken to load the load gates suitably to avoid excessive Miller multiplication of the load capacitance. Also, fabricated chips will include wire capacitance, which may have been neglected in simulation. Finally, the output should be tapped off one of the load gates to avoid extra branching effort on the ring oscillator. Unfortunately, this is not possible in rings built from fanout-of-1 gates.

5.4 Calibrating special circuit families

The calibration techniques so far work well for gates with roughly equal rise and fall times because the output of one gate is a realistic input to the next gate. Unfortunately, the techniques break down for other circuit families. For instance, a dynamic gate cannot directly drive another dynamic gate because the monotonicity rule would be violated. If gates are skewed to favor a critical transition, the edge rate of the other transition is unrealistically slow and should not be used to set the input slope.

Dynamic gates have a very low switching threshold. Since one dynamic gate

cannot directly drive another, there must be an inverting static gate between dynamic stages. Attempting to measure delay from input crossing 50% to output crossing 50% often leads to misleading results. If the slope of the input is slow, dynamic gates may even have a negative delay. A better approach is to characterize the delay of the dynamic gate and subsequent inverter as a pair. Remember to use an electrical effort for the pair equal to the produce of the electrical efforts of each stage. Initial estimates of logical effort can be used to determine the size of the static gate such that the stage effort of the dynamic and static gate are approximately equal.

Static gates are sometimes skewed to favor a particular transition, as will be discussed in Section 9.2.1. For example, a high-skew gate with a larger PMOS transistor may be used after a dynamic gate. Characterizing a chain of identical skewed gates also leads to misleading results. We would like to characterize the logical effort of the rising output of a high-skew gate because that is the delay which would appear in a critical path. If a chain of such skewed gates is used, the input slope will come from a falling transition and will be unreasonably slow. This retards the rising output as well. To avoid this problem, characterize skewed gates as part of a unit, just as recommended for dynamic gates.

5.5 Summary

This chapter has explored the accuracy of the method of logical effort through circuit simulation. The results suggest that the calculation methods described in Chapter 4 are good, but that somewhat greater accuracy and confidence can be obtained from more detailed calibrations.

Since the logical effort and parasitic delay of gates change only slightly with process, τ is a powerful way to characterize the speed of a process with a single number. Parasitic delay varies more than logical effort, but since effort delay usually exceeds parasitic delay, the variation is a smaller portion of the overall delay. By expressing the delay of circuits in terms of τ, or in the more widely recognized unit of fanout-of-4 inverter (FO4) delay (1 FO4$= 5\tau$), the designer can communicate with others in a process-independent way and can easily predict how gate performance will improve in more advanced processes.

5.6 Exercises

5-1 [25] Determine the logical effort and parasitic delay of an inverter, 2-input NAND, and 2-input NOR gate in your process. How well do the numbers agree with the estimates from Chapter 4 and the measurements in this chapter?

5-2 [20] Make plots of delay vs. electrical effort for each of the three inputs of a three-input NAND gate, using values from Tables 5.1 and 5.2. What general advice can you extract from your plot?

5-3 [15] The two inputs of an ordinary 2-input NAND gate differ because one of them connects to a transistor close to the output and the other to a transistor close to a power rail. Show how to build a 2-input NAND with identical logical efforts per input using two ordinary 2-input NAND gates with a shorted output.

Chapter 6

Forks of Amplifiers

The most difficult problems in applying the method of logical effort stem from branching. When a logic signal divides within a network and flows along multiple paths, we must decide how to allocate the input current. How much should each path load the common input? In general, paths that have higher logical or electrical effort should receive a greater share of the input signal's drive. When a logic signal has significant parasitic capacitance, for example when it drives a long wire, it branches: as some of the signal is diverted to charge the parasitic load, less drive is available to the logic path.

Optimizing networks that branch usually involves adjusting branching effort along paths to equalize the delays in several paths through the network. Determining the branching factors adds a new element of difficulty to our design method that can be quite tricky to handle. One of the complications is that different paths through a network often have different numbers of stages. Branching can sometimes be straightforward, depending on the design problem. For example, branching is simple in the synchronous arbitration problem of Section 2.3.

This chapter covers a simple but common case of branching: generating the true and complement forms of a logic signal. We call such circuits "forks," after the general appearance of their circuit diagrams. Forks are interesting not only for their own utility, but also as a further exercise in applying the method of logical effort. Many CMOS circuits require forks to produce such true and complement signals. For example, an arm of the multiplexer circuit of Figure 4.4 is switched on when one of its control lines, s_i, is made HIGH, and the other, $\overline{s_i}$, is made LOW. The XOR circuit shown in Figure 4.5 also requires true and complement forms of

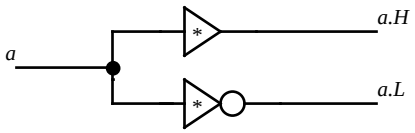


Figure 6.1: General form of an amplifier fork. One leg inverts the input signal and one does not.

its two input signals. We often use the notation $x.H$ and $x.L$ to indicate true and complement forms of a signal x, respectively. The signal $x.H$ is HIGH and $x.L$ is LOW when x is true; $x.H$ is LOW and $x.L$ is HIGH when x is false.

6.1 The fork circuit form

An amplifier fork, or simply a "fork," consists of two strings of inverters that share a common input, as shown in Figure 6.1. One of the strings contains an odd number of inverters and the other contains an even number. True and complement signals of this kind, particularly for driving multiplexers, are often required at relatively high power levels. For example, if an entire word consisting of 64 bits is to be multiplexed onto a bus, the true and complement signals for the entire word must drive 64 multiplexers. As we have already learned, least delay in driving such large loads will be obtained by including the proper number of amplifying inverters in the drive path. The figure illustrates a notation we use when the exact number of inverters in each path is not known: an inverter symbol with a star inside it represents a string of an odd number of inverters, while a triangle symbol with an embedded star but without the small circle at its output stands for an even number of inverters. We have chosen to name forks by the number of amplifiers in each string. A 3-2 fork, for example, has three amplifiers in one string and two in the other. Figure 6.2 shows a 2-1 fork and a 3-2 fork.

It is useful to think of pairs of amplifier strings as a fork only when the true and complement output signals must emerge at the same time. The driver for the address lines in Figure 2.3 has this property; delays in either leg of the fork can result in increased overall delay. We therefore wish to design forks so that the delay in each leg is the same.

In this chapter we shall assume that the load driven by each leg of the fork is the same. A fork that drives an XOR gate such as that of Figure 4.5 has this property; both a and $\overline{a}$, for example, drive the same load. A multiplexer like the

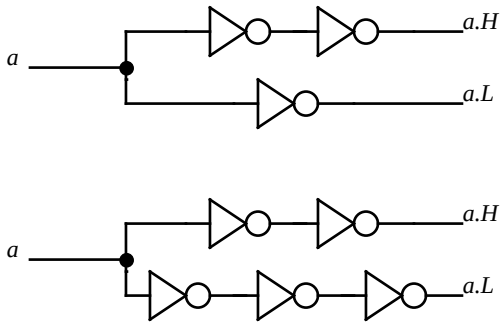


Figure 6.2: A 2-1 fork and a 3-2 fork, both of which produce the same logic signals.

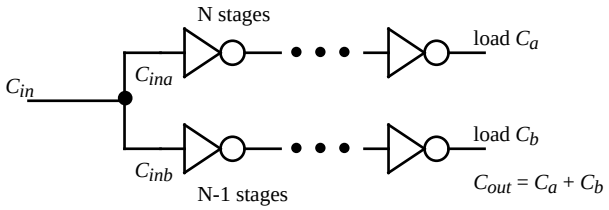


Figure 6.3: A general fork, showing notation for load capacitances.

one in Figure 4.4, however, presents unequal loads, because the pullup transistor driven by $\overline{s}$ is wider than the pulldown transistor driven by s. We shall defer to Chapter 7 consideration of circuits with multiple outputs driving different load capacitance (but see Exercise 6-1).

The design of a fork starts out with a known load on the output legs and known total input capacitance. As shown in Figure 6.3, we shall call the two output capacitances C_a and C_b. The combined total load driven we will call $C_{out} = C_a + C_b$. The total input capacitance for the fork we shall call $C_{in} = C_{in_a} + C_{in_b}$, and can thereby describe the electrical effort for the fork as a whole to be $H = C_{out}/C_{in}$. This electrical effort of the fork may differ from the electrical efforts of the individual legs, C_a/C_{in_a} and C_b/C_{in_b}.

The input current to an optimized fork may divide unequally to drive its two legs. Even if the load capacitances on the two legs of the fork are equal, it is not in general true that the input capacitances to the two legs of the fork are equal. Because the legs have a different number of amplifiers but must operate with the same delay, their electrical efforts may differ. The leg that can support the larger electrical effort, usually the leg with more amplifiers, will require less input cur-

rent than the other leg, and can therefore have a smaller input capacitance. If we call the electrical efforts of the two legs H_a and H_b, using the notation of Figure 6.3, then $H_a = C_a/C_{in_a}$ and $H_b = C_b/C_{in_b}$. Even if $C_a = C_b$, H_a may not equal H_b and C_{in_a} and C_{in_b} may also differ.

The design of a fork is a balancing act. Either leg of the fork can be made faster by reducing its electrical effort, which is done by giving it wider transistors for its initial amplifier. Doing so, however, takes input current away from the other leg of the fork and will inevitably make it slower. A fixed value of C_{in} provides, in effect, only a certain total width of transistor material to distribute between the first stages of the two legs; putting wider transistors in one leg requires putting narrower transistors in the other leg. The task of designing a minimum delay fork is really the task of allocating the available transistor width set by C_{in} to the input stages of the two legs.

Example 6.1 *Design a 2-1 fork with input capacitance $C_{in} = 10$ and total output capacitance $C_{out} = 200$. What is the total delay of the fork?*

Using the notation of Figure 6.3, we have $C_{in} = 10$ and $C_a = C_b = 100$. Let's use the symbol β to denote the fraction of input capacitance allocated to the 2-inverter leg, i.e., $C_{in_a} = \beta C_{in}$. The remainder is allocated to the 1-inverter leg. We want to find the value of β that will equalize delays in the two legs. Applying Equation 1.17 to both legs, we have:

$$2\left(\frac{100}{10\beta}\right)^{1/2} + 2p_{inv} = \left(\frac{100}{10(1-\beta)}\right) + p_{inv} \qquad (6.1)$$

We solve this equation numerically to find that $\beta = 0.258$, so $C_{in_a} = 10\beta = 2.6$ and $C_{in_b} = 10(1-\beta) = 7.4$. The second inverter in the 2-inverter leg has input capacitance $C_{a2} = 2.6 \times \sqrt{100/2.6} = 16.1$. The delay in the one-inverter leg is $C_b/C_{in_b} + p_{inv} = 100/7.4 + 1 = 14.5$. The delay in the two-inverter leg is $C_{a2}/Cin_a + C_a/C_{a2} + 2p_{inv} = 16.1/2.6 + 100/16.1 + 2 = 14.5$. Thus, the two legs have equal delay as expected.

Example 6.2 *Design a 3-2 fork with the same input and output capacitances as in the previous example.*

Again using β to determine the fraction of input capacitance allocated to the longer leg, we have:

$$3\left(\frac{100}{10\beta}\right)^{1/3} + 3p_{inv} = 2\left(\frac{100}{10(1-\beta)}\right)^{1/2} + 2p_{inv} \qquad (6.2)$$

This equation solves to $\beta = 0.513$, with a delay of 11.1. Note that this delay is less than the delay we computed for the 2-1 fork with the same input and output capacitances.

These two examples show that obtaining the least delay requires choosing the right number of stages in the fork. This result is not surprising. In fact, we should have anticipated that the first design could be improved because the effort of the one-inverter leg is 13.5, very far from the best ρ. This result suggests that we should develop a method to determine the best number of stages to use in a fork. The next section turns to this problem.

6.2 How many stages should a fork use?

An optimized fork must have legs that differ in length by at most one stage. We can see that this is true by examining in detail the relationship between total delay and electrical effort that was discussed in Chapter 1. Figure 6.4 shows schematically a plot of delay versus electrical effort for amplifiers with $N-1$, N and $N+1$ stages. The thick curve represents the fastest possible amplifier for any given electrical effort, and so no amplifier design may lie below it. For different electrical efforts, different numbers of stages are required to obtain this optimum design, as the figure shows.

The task of designing a fork is specified by giving an electrical effort that the combined activities of its two legs are to support. In Figure 6.4 such an electrical effort is represented by the vertical line. One possible design for the fork requires each leg to support exactly this electrical effort. Since the two legs of the fork must produce true and complement signals, their lengths must differ by an odd number of inverters. Thus if one leg has N stages, its delay can be reduced to the point labeled z in the figure, but the other leg must have either $N-1$ or $N+1$ stages, and its delay can be reduced only to the points labeled x and y respectively. For the particular electrical effort we have chosen, point y is faster than point x. Thus we have a fork with two legs that support equal electrical effort but have unequal delay.

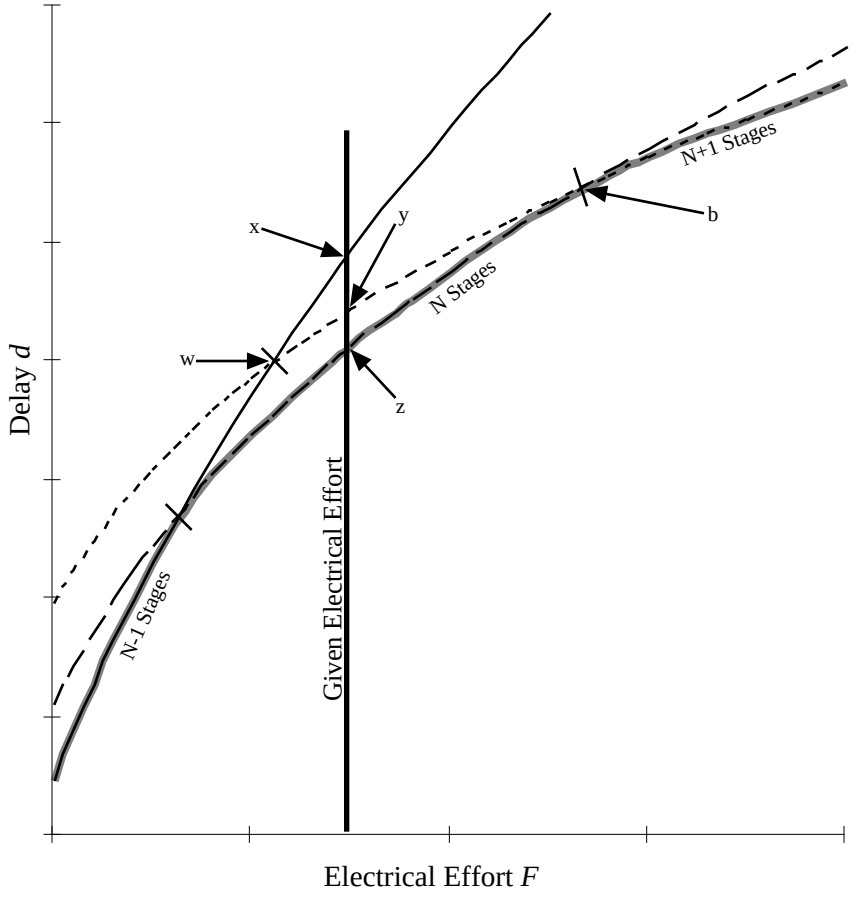


Figure 6.4: A plot of delay vs. electrical effort for reasoning about forks.

We can improve such a fork by raising the electrical effort of one leg and reducing the electrical effort of the other in such a way as to continue supporting the required total electrical effort. In effect we slide to the right from point z, thus increasing its delay, and the left from point y, thus decreasing its delay. We do this by reallocating transistor width from the transistors of the first amplifier in one leg to the transistors of the first amplifier in the other leg. Our intent, of course, is to decrease the delay of the slower leg as much as we can, which will be until the two legs are equal in delay.

One might think that there are two possible discontinuities in the process of reallocating the input transistor width. From point z moving to the right we may reach point b before the equal delay condition is met, or from point y moving to the left we may reach point w before the equal delay condition is met. Z could not possibly reach point b, however, because the delay at point y is already less than that at point b. If point y reaches point w before the equal delay condition is met, we should change it from $N + 1$ to $N - 1$ stages and continue along the $N - 1$ stage curve until we reach the equal delay condition. It is not hard to see that for any placement of the given electrical effort line, this optimization procedure will result in a fork with legs that differ in length by a single amplifier.

One leg of a fork will always have exactly the same number of stages as would an optimum amplifier supporting an equal electrical effort. This is easy to see from Figure 6.4. If the given electrical effort line crosses through the dark optimum curve in the segment where N stages are best, one arm will have N stages. The other arm will have either $N + 1$ or $N - 1$ stages. Thus one simple way to design nearly optimal forks is to choose the number of stages for one leg from Table 1.3, and then use one more or one fewer stage for the other leg. The electrical effort for the fork as a whole, $H = C_{out}/C_{in}$, can be used as a guide, since the electrical efforts of each leg will turn out to be nearly that value. Applying this technique to Example 6.1, $H = 20$ would have correctly suggested a 3-2 fork as the best design.

A more precise guide for choosing the number of stages in a fork appears in Table 6.1. For any given electrical effort, the table shows what kind of fork to use. Remember that the electrical effort of the fork is the total load of all the legs divided by the total input capacitance. The break points in Table 6.1 lie between the break points in Table 1.3. It is easy to see that this must be so. There are certain electrical efforts, namely 5.83, 22.3, 82.2, and so on, for which strings of amplifiers N and $N + 1$ long give identical delays. Obviously, for an electrical effort of 22.3, for example, a 3-2 fork is best, because both strings of 3 amplifiers and strings of 2 amplifiers have the same delay for this electrical effort. For an

Electrical effort		Fork
from	to	structure
	9.68	2-1
9.68	38.7	3-2
38.7	146	4-3
146	538	5-4
538	1970	6-5
1970	7150	7-6

Table 6.1: Break points for forks assuming $p_{inv} = 1.0$.

electrical effort of 82.2, a 4-3 fork is best for a similar reason. In these special cases, moreover, the input capacitance to the legs will also be identical. For some electrical effort between 22.3 and 82.2, a 4-3 fork and a 3-2 fork will give identical results. This is the break point recorded in Table 6.1.

It is easy to see how Table 6.1 was computed. Consider the break point where a 3-2 fork and a 4-3 fork provide identical results. At this break point the two forks exhibit identical delays. The three-amplifier legs in each fork must be identical. Moreover, the amount of input current left over from the three-amplifier leg in each fork must also be the same, and thus the input currents of the two-amplifier leg in one fork and the four-amplifier leg in the other fork must also be the same. Thus at the break point between 3-2 and 4-3 forks, the electrical effort of the two-amplifier leg and of the four-amplifier leg must be the same, and they are operating with identical delays. In terms of Figure 6.4, they must both be operating at point w. This reasoning leads directly to equations that can be solved to find the electrical effort of optimal forks at these break points (see Exercise 6-3).

Example 6.3 *Design a path to drive the enables on a bank of 64 tri-state bus drivers. The first stage of the path can present an input capacitance of twelve unit-sized transistors and the tri-state drivers are each 6 times unit size. A unit sized tri-state is shown in Figure 6.5.*

The load on each of the true and complementary enable signals is $64 \times 6 \times 2 = 768$ unit-sized transistors. Therefore the electrical effort of the entire bundle is $(768 + 768)/12 = 128$. From Table 6.1, we find that a 4-3 fork is best. Now we must divide the input capacitance between the two legs. If the 4-inverter leg gets a fraction β, then we have:

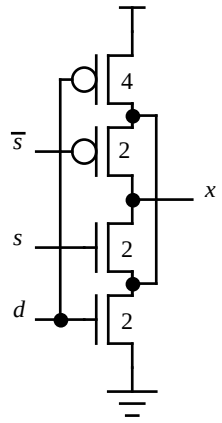


Figure 6.5: A unit-sized tri-state inverter.

$$4\left(\frac{768}{12\beta}\right)^{1/4} + 4p_{inv} = 3\left(\frac{768}{12(1-\beta)}\right)^{1/3} + 3p_{inv} \qquad (6.3)$$

This equation solves to $\beta = 0.46$. Therefore, the input capacitance on the 4-fork is $12 \times 0.46 = 5.5$ and the input capacitance on the 3-fork must be $12 - 4.8 = 6.5$. Hence, the electrical effort of the 4-fork is $768/5.5 = 140$ and the electrical effort of the 3-fork is $768/6.5 = 118$. Notice how the electrical effort of the entire bundle, 128, is unevenly distributed among the legs to improve the delay of the slower leg at the expense of the faster until the two are equalized at 17.7. The stage effort of the 4-fork is $140^{1/4} = 3.44$ and the stage effort of the 3-fork is $118^{1/3} = 4.90$. Therefore, the capacitances of each gate can be computed, as shown in Figure 6.6.

If very large amplification is required, forks with large numbers of amplifiers in each leg may be used. One may well ask whether to prefer an 11-10 fork to a string of 8 amplifiers followed by a 3-2 fork. In fact, there is little to be gained by using forks with more than 3-2 amplifiers; this is quantified in Exercise 6-4. On the other hand, long strings of amplifiers will no doubt contain very large transistors that will be laid out in sections anyway. Thus the layout penalty of a long fork may be small.

Ironically, the most difficult case occurs when very *small* amplification is required. It is almost never advisable to use a 1-0 fork such as the one shown in

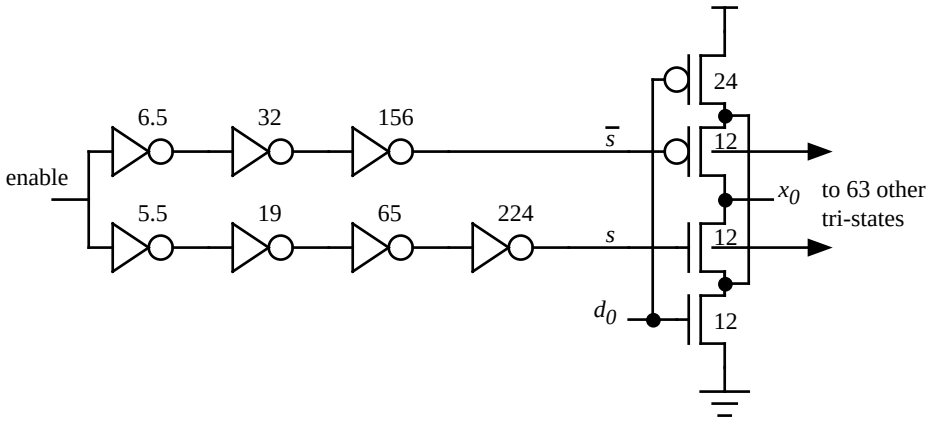


Figure 6.6: The tri-state enable path, properly sized.

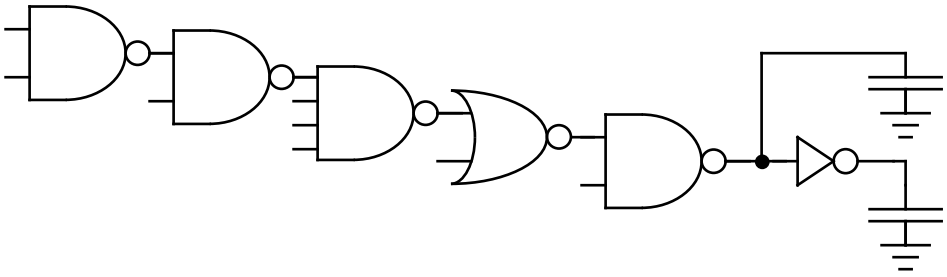


Figure 6.7: A path ending with a 1-0 fork.

Figure 6.7, because the delays in the two paths cannot be equalized: the delay of the zero-stage path is guaranteed to be less than that of the one-stage path. Exercise 6-5 examines the performance penalty of 1-0 forks. Rather than use a 1-0 fork, it is better to use a 2-1 fork and, if necessary, remove a stage somewhere else in the network.[1]

6.3 Summary

The forks we have designed in this chapter are a special case of more complex circuits with branches that operate in parallel. While the general cases are more

[1]Rather than remove a stage of the network, the stage immediately preceding the fork can be duplicated in each of the fork's paths, in effect moving the branch point from after this gate to before this gate.

difficult to solve, two of the techniques shown in this chapter apply to more complex branching problems covered in the next chapter:

- The path effort of a network, measured as the total load capacitance divided by the input capacitance, can be used to estimate the correct number of stages to use.
- Once a network topology is selected, it's a simple matter to write equations that compute the delay in each path, and solve for the branching factors β that equalize delays in multiple paths.

In the case of amplifier forks driving equal loads, we have shown that the number of stages in the two paths is nearly the same. However, as we shall see in more general cases that have different efforts along different paths, the lengths of different paths in a network may differ substantially.

6.4 Exercises

6-1 [25] A common use of forks in CMOS designs is to drive the enable signals of multiplexers, which present different loads to the true and complement signals. Many multiplexers may be driven by the same logic signal, resulting in a very large load. Modify the analysis of this chapter to apply to these forks, assuming the load on the leg with an odd number of stages is twice the load on the leg with an even number of stages; note that this assumption is equivalent to saying that the higher load must be driven by a signal that is the logical complement of the input to the fork. Build a table analogous to Table 6.1 that tells a designer what kind of fork to use.

6-2 [20] Generalize Exercise 6-1 to guide a designer when either the true or complement form of the input signal is available.

6-3 [25] Set up equations for computing the entries in Table 6.1. Solve them and verify that your answers match those of the table. (You'll want to write a computer program or spreadsheet to find the solutions.)

6-4 [30] Suppose that for $H > 38.7$, rather than building a pure fork, we use a string of inverters driving a 3-2 fork. How far does this strategy depart from the optimum fork designs?

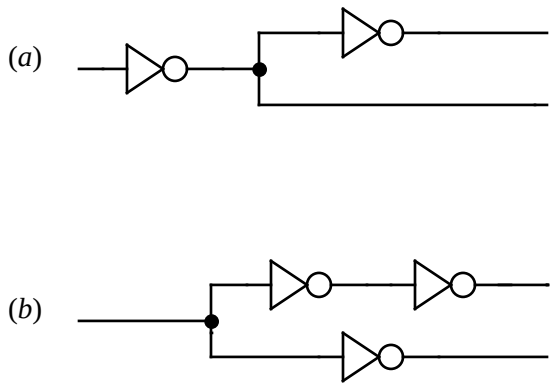


Figure 6.8: Comparing these two designs illustrates the problems with 1-0 forks.

6-5 [20] Propose an improvement to the design of Figure 6.7 that uses a 2-1 fork. If each of the load capaitances is 400 times as large as the input capacitance, what are the delays of the original and improved designs?

6-6 [20] Consider the two designs in Figure 6.8. The first uses a 1-0 fork, while the other avoids this structure. Compare the delays of the two designs over a range of plausible electrical efforts. Is the first design ever preferred?

Chapter 7

Branches and Interconnect

Logical effort is easy to use on circuits with easily computed branching efforts. For example, circuits with a single output or a regular structure are easy to design. Real circuits often involve more complex branching and fixed wire loads. There is often no closed-form expression for the best design of such circuits, but this chapter develops approximations and iterative methods that lead to good designs in most cases.

Designing networks that branch requires not only finding the best topology for the network, but also deciding how to divide the drive at a branch so that delays in all paths are equalized. The previous chapter covered the special case of "forks of amplifiers" that generate true and complemented forms of an input signal. In this chapter, we build on the previous results to handle more general cases. We shall consider circuits with two or more legs, where each leg may contain a different number of stages, each leg may perform a different logic function, and each leg may drive a different load. As one might guess, we can combine the logical and electrical efforts associated with each leg to obtain a composite effort on which to base our computations.

The simplicity of the forks considered in the previous chapter makes it possible to choose their topology from a table on the basis of the overall electrical effort imposed on the fork. In this chapter we will consider a more complex and varied set of circuits. We will use the theory of logical effort to write equations that relate the size of individual logic elements to their delay. By balancing the delay in the various legs of the circuit, we will be able to reduce the delay in the worst path.

The first section analyzes a series of examples in order to develop some intuitive arguments about branching networks. These examples are generalizations of forks of amplifiers. Next, we turn to an exclusive-or network that involves not only branching but also recombination of signals within the network. Interconnect presents new problems because the capacitance of the wire does not scale with gate size. However, circuits with interconnect can be analyzed on a case by case basis. We close the chapter with an outline of a general design procedure for dealing with branching networks.

Although it is possible to formulate network design problems as a set of delay equations and solve for a minimum, the method of logical effort often provides simple insights that yield good designs without a lot of numerical work. If necessary, these initial designs can be adjusted based on more detailed timing analysis.

7.1 Circuits that branch at a single input

This section analyzes a series of increasingly complex circuits that branch. We will start with simple circuits that branch immediately at a single input, and postpone to later circuits with logic functions preceding a branch point. Such circuits are very similar to the forks of Chapter 6, except that we shall now consider also unequal loading on the outputs and unequal logical effort in the legs.

7.1.1 Branch paths with equal lengths

The first example, a 2-path fork shown in Figure 7.1, will serve as a vehicle to remind us that logical effort can make certain branching decisions very easy. The variables C_1 and C_2 are the input capacitance for the first inverters in each path, as shown. The electrical efforts borne by the two paths are H_1 and H_2, so the total electrical effort of the circuit is $H_1 + H_2$. We can assume, without loss of generality, that the total input capacitance is 1, and simply scale all capacitances appropriately. Writing the delay equations for each path and setting them equal quickly tells us that

$$\frac{H_1}{C_1} = \frac{H_2}{C_2} \tag{7.1}$$

This equation holds for paths of any length, provided there are the same number of stages in each path and that the parasitic delay of each path is equal. It shows that the input drive should be allocated in proportion to the electrical effort borne

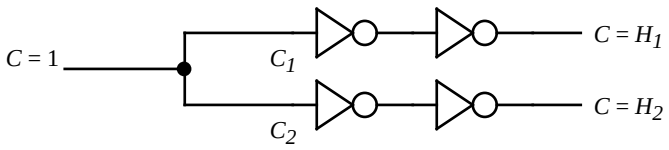


Figure 7.1: A 2-2 fork with unequal effort.

by the path. Once we have determined how to allocate the input capacitance, we can calculate transistor sizes for each path independently.

What happens if the paths include logic, rather than simply inverters? The method of logical effort teaches us that logical effort and electrical effort are interchangeable. So if path 1 had a total logical effort of G_1 and path 2 a total logical effort of G_2, then Equation 7.1 becomes simply

$$\frac{F_1}{C_1} = \frac{F_2}{C_2} \tag{7.2}$$

and $F_i = G_i H_i$ for each path. In other words, the input capacitance should be allocated in proportion to the total effort borne by each path.

Even more importantly, the entire configuration of Figure 7.1 is equivalent to a single string of two inverters with output capacitance $F_1 + F_2$. The important point is that *the equivalent configuration has no branch;* the effect of the branch has been captured by summing the efforts of the two paths. This property allows us to analyze branching networks by working backward from the outputs, replacing branching paths with their single-path equivalents. The branching effort of the leg is:

$$B = \frac{F_{leg} + F_{others}}{F_{leg}} \tag{7.3}$$

The path effort $F = G_{leg} B H_{leg}$ and stage effort $f = F^{1/N}$ can then be calculated. This stage effort is the same for all legs of the branch. This is a powerful technique for designing networks with branches, as illustrated in the following example.

Example 7.1 *Size the circuit in Figure 7.2 for minimum delay.*

The electrical effort of the top leg is $H_1 = 144/12 = 12$ and the electrical effort of the bottom leg is $H_2 = 192/12 = 16$. The logical

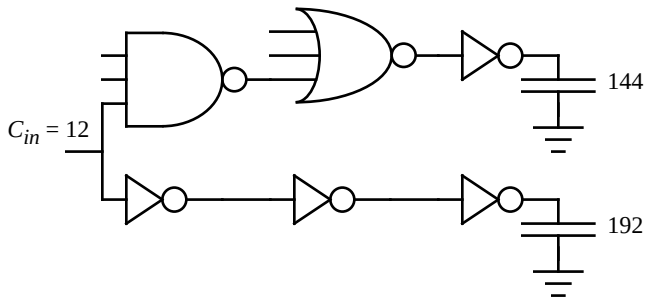


Figure 7.2: A path with different logical and electrical efforts on each leg.

effort of the top leg is $G_1 = 5/3 \times 7/3 \times 1 = 3.89$ and the logical effort of the bottom leg is $G_2 = 1 \times 1 \times 1 = 1$. Thus, the path effort of the top leg is $F_1 = G_1 H_1 = 46.7$ and the path effort of the bottom leg is $F_2 = G_2 H_2 = 16$. The overall path effort is $F_1 + F_2 = 62.7$.

First consider sizing the top leg. From the perspective of the top leg, the circuit has a branching effort of $B = (F_1 + F_2)/F_1 = 1.34$. Thus, the circuit can be designed as if it had only one leg with $F = G_1 B H_1 = 62.7$. The stage effort is $f = 62.7^{1/3} = 3.97$. Starting at the output and working backward, we find gate input capacitances of 36, 31, and 9 for the inverter, NOR, and NAND, respectively.

The bottom leg is now easy to size because we know the stage effort must be the same, 3.97. Working backward, we find input capacitances of 48, 12, and 3, respectively. The total input capacitance of the path is $3 + 9 = 12$, meeting the original specification. Also, the input capacitance is divided between the legs in a 3:1 ratio, just as path effort is in a $46.7 : 16 \approx 3 : 1$ ratio.

The delay of the top leg, including parasitics, is $3(3.97) + 3 + 3 + 1 = 18.91$. The delay of the bottom leg is $3(3.97) + 1 + 1 + 1 = 14.91$. Although we had attempted to size each leg for equal delay, the different parasitics result in different delays. To equalize delay, a larger portion of the input capacitance must be dedicated to the top leg with more parasitics. Even when this is done, however, the delay of each path is 18.28, representing only a 3% speedup.

This example illustrates a general problem: unequal parasitics damage Equation 7.1. Sometimes the difference in parasitic delays is small, and our previous analysis is very nearly correct. Even if the difference is large, as it was in the

example, the overall improvement from devoting more input capacitance to the slower leg is often small.

If differences in parasitic delay are too great, we can use our analysis to find an initial design, but to get the best design, we will need to adjust the branching allocation. Usually, this is simply a matter of making accurate delay calculations for each path, and modifying the branching allocation slightly. In effect, we are finding a value of C_1 for which

$$D = N\left(\frac{F_1}{C_1}\right)^{1/N} + P_1 = N\left(\frac{F_2}{1-C_1}\right)^{1/N} + P_2 \tag{7.4}$$

Because parasitic delay adds considerable complexity to the algebra for analyzing branching circuits, we will omit the effects of parasitic delay in the other examples in this chapter. In all cases, assuming zero parasitic delay leads to a pretty good design that can then be refined by more accurate delay calculations and adjustments. A spreadsheet program is a handy tool for making such calculations.

7.1.2 Branch paths with unequal lengths

The amplifier forks we studied in Chapter 6 are simple networks that have branch paths of unequal lengths. We will revisit their analysis here to introduce the problem of designing arbitrary branching networks. As a first example, consider the simple case of a 2-1 fork with unequal loading, as shown in Figure 7.3. We use similar conventions as in Figure 7.1 for denoting inverter input capacitances, but to remind us that our analysis accommodates different logical effort in each path, we have labeled each path's load with the effort, F_i. If we want the delay in each path to be D, the inverter in the first path will have delay D and each inverter in the second path will have delay $D/2$. Recalling Equation 1.17, when $P = 0$, we have $F = (D/N)^N$ for each path:

$$\begin{aligned} \frac{F_1}{C_1} &= D \\ \frac{F_2}{C_2} &= \left(\frac{D}{2}\right)^2 \end{aligned} \tag{7.5}$$

Setting $C_1 + C_2 = 1$ yields a quadratic equation:

$$D^2 - F_1 D - 4F_2 = 0 \tag{7.6}$$

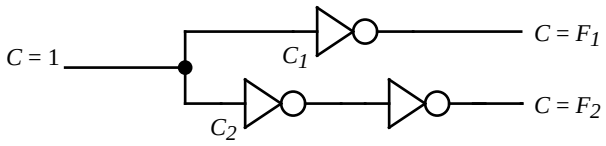


Figure 7.3: A 2-1 fork with unequal effort.

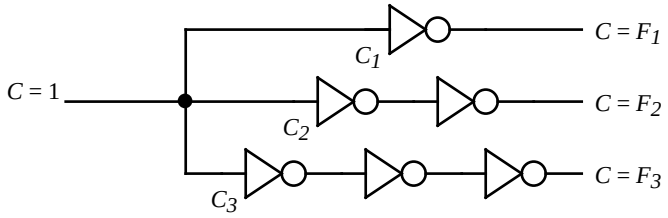


Figure 7.4: A 3-2-1 fork with unequal effort.

that can be solved for D to obtain:

$$D = \frac{F_1 + \sqrt{F_1^2 + 16F^2}}{2} \tag{7.7}$$

We can verify some known configurations, e.g., $F_1 = 0$ implies $C_1 = 0$, and $F_2 = 0$ implies $C_2 = 0$. In the special case when $F_1 = F_2 = 2$, we find that the input current divides equally between the two legs, and $D = 4$.

The analysis generalizes easily to a fork with three paths as shown in Figure 7.4. For each path i, we have, by analogy to Equation 7.5,

$$\frac{F_i}{C_i} = \left(\frac{D}{i}\right)^i \tag{7.8}$$

Not surprisingly, solving yields a cubic equation in D:

$$D^3 - F_1 D^2 - 4F_2 D - 27F_3 = 0 \tag{7.9}$$

As expected, when $F_3 = 0$, this reduces to Equation 7.6.

Now it is time to step back and consider what we have done. First of all, assuming that all three legs of the fork of Figure 7.4 must operate in the same delay, is there any point to having one leg with a single stage and another with three? There is not, for the same reason that it is pointless to make the two legs of a simple fork differ in length by more than one. For any value of F_1, F_2, and F_3

we could improve the performance of the circuit of Figure 7.4 by removing one leg. If the delay is short, because F_1, F_2 and F_3 are small compared to $C = 1$, then a simple 2-1 fork will have less maximum delay, and we should eliminate two amplifiers from the three amplifier leg, in effect collapsing it into the first leg. If the delay is long, because F_1, F_2 and F_3 are large compared to $C = 1$, then a 3-2 fork will have less maximum delay, and we should add two amplifiers to the one amplifier leg, thus combining it with the three amplifier leg.

Of course, our example involves only inverters, and we want to consider cases where each leg contains a logic function as well. When there are logic functions involved, one might argue that the particular logic functions require the given number of stages. That may be a valid argument for preserving the three-stage leg, because there may be logic functions that can be done with less logical effort in 3 stages than in a single stage. Thus if the delay is short, because F_1, F_2, and F_3 are small compared to 1 and the logical efforts of the legs are also small, we may nevertheless require the three-stage leg. On the other hand, if the delay is long, because F_1, F_2, and F_3 are large compared to 1 or large logical efforts are involved, we could improve the design by augmenting the single stage leg with a pair of inverters. We will think it unusual, therefore, to find a least delay circuit whose legs differ in length by more than one. An important exception arises when the problem is constrained by minimum drawn device widths.

This reasoning suggests that for purposes of branch analysis, we can always collapse N-way branches ($N > 2$) into 2-way branches by combining paths. This simplifies the equations for allocating input capacitance: we will have at most two equations like Equation 7.8. Moreover, because logical and electrical effort are interchangeable, all these branching problems are equivalent to designing amplifier forks, covered in Chapter 6. Note, however, that when we model parasitic delay properly, collapsing paths of equal length but different parasitic delay may introduce errors.

In summary, circuits with a single input and multiple outputs can be analyzed as forks, except that the effective load capacitance on each output must be increased by the logical effort of the leg driving it. Because the minimum delay circuits will generally have paths of nearly the same length, a good approximation to their performance can be obtained by assuming that all paths are the same length, summing the path efforts, and analyzing the entire network as a single path. We learned in Section 3.5 that the performance of strings of amplifiers or strings of logic is relatively insensitive to small changes in their length. A good first approximation, therefore, lumps all the effort of the network for choosing a suitable path length.

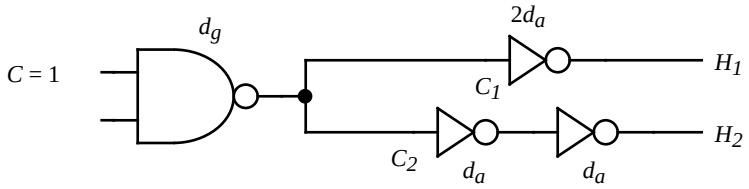


Figure 7.5: Branching after logic.

7.2 Branches after logic

We are now ready to consider circuits with logic functions preceding a branch point. One common form of circuit contains a logic element followed by a 2-1 fork, as shown in Figure 7.5. Such circuits have a "fan in" part followed by a "fan out" part joined by a single connection that is the obvious place to break the circuit for analysis. The delay in each of the inverters in the lower leg of the fork is d_a. The delay in the logic element with logical effort g is d_g. We have used separate variables for these two delays because there is no reason to believe that they will be equal. In fact, we expect that in designs for least overall delay, d_g will have a value somewhere between d_a and $2d_a$; the longer leg of the fork will use inverters operating faster than the logic element, and the shorter leg of the fork will use inverters operating slower than the logic element.

Analyzing this case as we did the forks, we write equations for effort:

$$\begin{aligned} H_1/C_1 &= 2d_a \\ H_2/C_2 &= d_a^2 \\ g(C_1 + C_2) &= d_g \end{aligned}$$

Using the first two equations to eliminate C_1 and C_2 from the third, and writing an equation for D, we have:

$$D = d_g + 2d_a = g\left(\frac{H_1}{2d_a} + \frac{H_2}{d_a^2}\right) + 2d_a \tag{7.10}$$

Taking the partial derivative of D with respect to d_a and setting the result to zero, to find a minimum, yields:

$$d_a^3 - \frac{gH_1}{4}d_a - gH_2 = 0 \tag{7.11}$$

Now let us choose sample values to match closely the special case of 2-1 fork with stage delay of 2. The logical effort of the NAND gate is $4/3$, and $H_1 = H_2 =$

3, so the total effort is 8, considering the outputs as a bundled pair, which would give a stage delay of 2 in a pure three-stage design. If both sides of the fork had two stages, a per stage delay of 2 would be best, and the overall delay would be 6. One side of the fork, however, has only one stage instead of two. One might think that because of the similarity of a single inverter with stage delay of 4 to a pair with stage delays of 2 each substituting the single inverter for two to make the fork would leave the stage delay unchanged. This is simply not so, as the numbers show. Solving Equation 7.11 for the sample values, we find $d_a = 1.796$ and $d_g = 2.35$. In other words, a slightly faster circuit can be obtained by using more time in the logic element and less in the fork, with an overall delay of 5.94 instead of 6. This is not entirely unreasonable because the single amplifier leg of the fork is not as good at driving heavy loads as is the two amplifier side. Nevertheless, in keeping with the relative insensitivity of delay to the number of stages, the improvement of 1% may not be worth the effort of calculating how to get it.

Consider an example with an overall effort of 20, namely $g = 5/3$, and $H_1 = H_2 = 6$. Here three stages are more obviously required. If both sides of the fork had two stages, we would have a pure three stage design with a stage delay of 2.71, and an overall delay of 8.14. Solving Equation 7.11, we find $d_a = 2.54$ and $d_g = 3.52$, we find the best we can do is $D = 8.60$, reflecting the fact that there is only one amplifier in one side of the fork. As we expected, $d_g = 3.52$ lies between $d_a = 2.54$ and $2d_a = 5.08$.

In some circuits several stages of logic or amplification precede a single branch point that leads to a fork. For such circuits, least delay will be obtained with a stage delay in the early circuitry that lies between the stage delay of the longer leg of the fork and the stage delay of the shorter leg of the fork. As the number of stages grows larger, the influence of one stage more or less on the overall delay becomes less, as we learned in Section 3.5, and thus nearly minimal delays can be obtained by treating the outputs as a bundle. If more accurate results are desired, it is easy to write equations similar to those in the figures for any particular case. The solution of such equations leads to the fastest design.

7.3 Circuits that branch and recombine

Some circuits do not fit either of the forms we have so far considered; they both branch and recombine. Such circuits can be analyzed in the same way as we have done with the previous examples. The simple delay model provides expressions

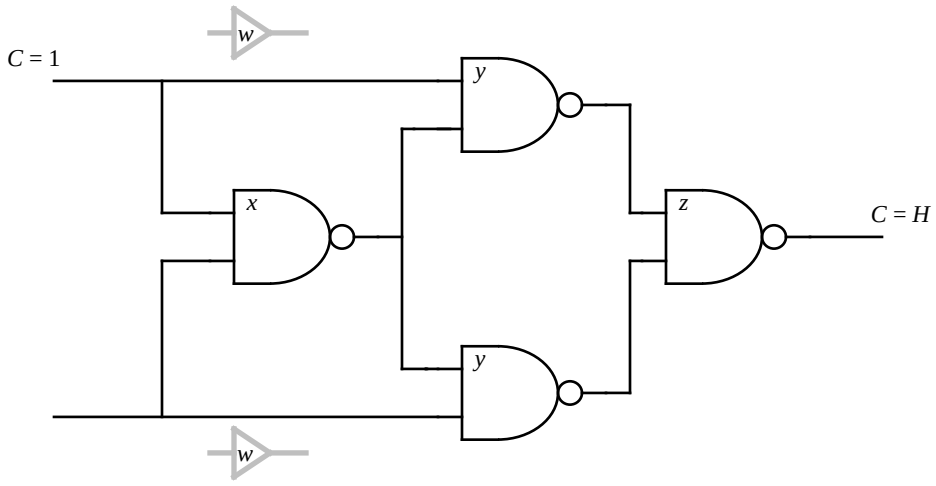


Figure 7.6: The 4-NAND XOR illustrating branching and recombining.

for the input capacitance of each logic gate. Where optimization is required, these expressions can be differentiated.

An interesting example of such a circuit is the form of XOR shown in Figure 7.6. While this circuit has only one output, its early stages branch and recombine in a way that requires an analysis similar to others in this chapter. The topology of this circuit involves both some paths with three stages and some paths with two stages. The output of the first stage recombines with direct inputs at the second stage. Our interest in this example lies in learning how to analyze it and in understanding the resulting delays.

We will solve this example three times. First, we shall assume that all delays are equal and obtain the circuit with least delay that meets that condition. Second, we shall permit the delay in the first stage, c, to differ from that of the other two stages, d, and again obtain the circuit with least delay. We are interested in whether minimum delay in this second situation requires c to be longer or shorter than d, and whether the circuit where c and d differ will be faster or slower than when they are required to be the same. Third, as a thought experiment, we shall add mythical non-inverting amplifiers to the circuit, as indicated by the dotted symbols in the figure. Although such amplifiers are not realizable, it will be instructive to study the changes that they would cause to circuit performance by providing three stages in each path.

As shown in the figure, the circuit consists of four NAND gates, which we shall treat as having a logical effort of $4/3$ per input; in order not to write $4/3$ over

and over, we shall use the symbol g for it. In order to have a particular electrical effort to work with, we shall assume that the output loading, H, is the same as the combined input capacitance, 2. We will also take advantage of symmetries in the circuit to deal only with capacitances labeled x, y and z. Now we can write three delay equations for each stage of the circuit and one equation constraining the branch

$$\begin{aligned} c &= g(2y/x) \\ d &= g(z/y) \\ d &= g(H/z) \\ x+y &= 1 \end{aligned}$$

Eliminating x, y, z, we find an equation for c in terms of d. In the special case where we insist $c = d$, we can solve to obtain $d = 2.67$, so $D = c + 2d = 8$. To obtain the lowest delay possible, we write an equation for $D = c + 2d$, substitute the expression for c, take the partial derivative of D with respect to d, and set it to zero. We obtain a fourth-order equation in d, which we solve to find $d = 2.98$, implying $c = 1.78$, for an overall delay of $D = 7.74$. This is an improvement over the value $D = 8$ obtained with $c = d$. Notice that the delay is distributed unequally to the three stages; the first stage operates with less delay, and the remaining two with more delay than when all three delays were forced to be equal. This is because NAND x must operate faster than usual to try to equal the zero delay through the zero-stage path from the input to gates y.

Notice that the first stage operates in parallel with the direct connection from the inputs to the second stage. Since the direct connection is not an amplifier, it pays to provide more amplification in later stages by making them operate relatively more slowly.

We have assumed in these calculations that the delay, d, in the second and third stages should be equal. Why is this a reasonable assumption? Proving that it is so may be an interesting exercise for the reader.

As a final exercise, let us put in the mythical non-inverting amplifiers in the positions shown as dotted symbols in Figure 7.6. This case is easy to solve, because we want the delays in all three stages to be equal. We can easily write expressions for the input capacitances of the two paths:

$$\begin{aligned} w &= g^2 H/d^3 \\ x &= 2g^3 H/d^3 \end{aligned}$$

Note how the path effort, namely the product of the branching, logical, and electrical effort, enters into these expressions. Setting $w + x = 1$ and solving, we find $d = 2.35$ and $D = 7.06$, an improvement over both of the other cases we have considered. Clearly, failure to amplify the direct input signals during the time the first stage of logic operates costs delay. The lesson to be learned from this example is to seize every opportunity to buffer less-critical signals because such amplification in one path can make available more source current for other paths and thus improve overall performance. Carried to an extreme, faster paths are buffered until all paths complete simultaneously.

7.4 Interconnect

Interconnect introduces particular problems for designing with logical effort because it has fixed capacitance. The branching effort of a gate driving a wire to another gate load is $(C_{gate} + C_{wire})/C_{gate}$. This branching effort changes whenever transistor sizes in the network are adjusted because the wiring capacitance does not change in proportion to the transistor size changes. Therefore, our handy rule that delay is minimized when the effort of each stage is equal breaks down; the gate driving the wire may use higher effort, while the gate at the end of the wire will use lower effort. This problem leads us to approximate the branching effort or to solve complex equations for the exact optimum. This section addresses approximations for circuits with interconnect. The necessity to make such approximations is one of the most unsatisfying limitations of logical effort.

When doing designs that account for wiring capacitance, it is helpful to relate the total capacitance of a wire to the input capacitance of logic gates. The gate capacitance of a minimum-length transistor is approximately 2 fF/μm and has remained such over many process generations because both length and dielectric thickness scale by about the same amount. The wire capacitance of minimum-pitch interconnect is approximately 0.2 fF/μm and also remains constant when wire thickness, wire pitch, and dielectric thickness scale uniformly. Therefore, a handy rule of thumb is that wire capacitance per unit distance is 1/10 that of gate capacitance. Of course this does not apply for wider wires and depends somewhat on the details of your process. It is very useful to know the ratio for your process to one significant figure so that you can quickly convert wire capacitances into equivalent amounts of gate width.

7.4.1 Short wires

Within a functional block, most wires are short and gate delay is dominated by gate capacitance. Moreover, actual wire lengths are very difficult to estimate until layout is complete. What effect do such short wires have on gate sizing?

Short wires can be treated as additional parasitic capacitance. Given the average length of a wire and the average size of a gate, one can compute the average ratio of parasitic diffusion capacitance to parasitic wire capacitance. The total parasitic capacitance is the sum of these two components. A best stage effort ρ can be computed from this total parasitic capacitance; this effort usually is slightly over 4 for paths with reasonably short wires.

This result makes intuitive sense. As wire parasitics increase relative to gate capacitance, it is wise to use fewer stages with greater stage effort to minimize the number of wires to drive. Fortunately, we found in Section 3.5 that the path delay is good for stage efforts anywhere near ρ. Designing with a target stage effort of 4 is sufficient for most problems.

7.4.2 Long wires

Wires between functional blocks can be hundreds or thousands of times larger than most transistors within the functional blocks. We consider a wire to be long when its capacitance is large compared to the gate capacitance it drives.

A path containing a long wire can therefore be split into two parts. The first part drives the wire. The second part receives its input from the wire. The size of the receiving gate makes little difference on the delay of the first path because the gate capacitance is small compared to the wire capacitance. In other words, the branching effort is a weak function of the receiving gate size. However, the size directly affects the electrical effort and thus the delay of the second path, so it should be made as large as possible, limited by area and power considerations.

The first path typically ends with an inverter chain to drive the large wire capacitance. The final inverter can be very large, consuming significant area and power. Therefore, choosing a stage effort of closer to 8 for this stage reduces the cost and only slightly increases delay. Because it is expensive in time and area to build the inverter chains that drive long wires, it is best to clump logic as much as possible so a path passes through few long wires. For example, the arbitration circuit of Section 2.3 was greatly improved by computing the function at a central location rather than daisy-chaining the logic across multiple wires.

When wires are very long, the resistance of the wire becomes important. Since

both wire resistance and capacitance are proportional to wire length, wire delay scales quadratically with wire length. Therefore, it is beneficial to break long wires into sections, each driven by an inverter or buffer called a repeater. Wires with proper repeaters have delay that is only a linear function of length [1]. When using repeaters, the designer must plan where the repeaters will be located on the chip floorplan.

7.4.3 Medium wires

The most difficult sizing problems occur when interconnect capacitance is comparable to the gate load capacitance. Such medium-length wires introduce branching efforts that are a strong function of the sizes of the gates they drive.

A brute force solution to sizing paths with medium wires is to write the delay equation as a function of the sizes of all the gates along the path and of the wire capacitance. This leads to a polynomial function which can be differentiated with respect to the gate sizes and solved numerically for the best sizes.

Such a solution is usually more work than is worthwhile. For most circuits, reasonable results can be obtained by maintaining a stage effort of about 4 through the path. Choosing the best number of stages is complicated by the fact that the branching effort caused by the wires is initially unknown. This leads to a few simple iterations, described in the next section.

7.5 A design approach

The examples of this chapter all illustrate common themes in the design of branching networks. The analytical treatment is intended to give insight into design tradeoffs. In practice, however, designers are unlikely to take partial derivatives and solve Nth order polynomial equations just to achieve a few percent improvement in delay. Instead, a practical design approach uses the insight about branching networks to select a reasonable topology and make an initial guess of stage efforts. Then the designer iterates on paths which are unacceptably slow until the path either meets specification or it becomes clear that a better topology is required.

Moreover, real circuits frequently contain fixed capacitances. For example, interconnect capacitances are independent of the gate sizes, as discussed in Section 7.4. There is a minimum size each transistor may be drawn, which limits how small a load can be presented by a non-critical leg of a circuit. Finally, all outputs

may not be required at the same time, and so devoting a larger portion of the input capacitance to critical outputs can hasten them at the expense of non-critical outputs.

When fixed capacitances are small compared to other capacitances on the node, ignore them. When fixed capacitances are large compared to other capacitances, the fixed capacitance dominates delay. If other gates loading the node are not fast enough, increase them in size to reduce their own delay while only slightly increasing the total capacitance on the node. The most difficult case is when fixed capacitances are comparable to gate capacitances. In such a case, the designer may have iterate to achieve acceptable results.

Here we will try to summarize some of these techniques by suggesting a design procedure.

1. Draw a network.

2. Buffer non-critical paths with minimum-sized gates to minimize their load on the important paths. Try to make all critical paths have similar lengths.

3. Estimate the total effort along each path, e.g., by working backwards from the outputs, combining efforts at each branch point.

4. Verify that the number of stages for the network is appropriate for the total effort that the network bears.

5. Assign a branching ratio to each branch; work backwards from the outputs, considering each branch you reach. Estimate a branching ratio based on the ratio of the effort required by each path leaving the branch. You may choose not to optimize certain paths that bear very little effort or whose speeds are not critical for your purpose.

6. Compute accurate delays for your design, including the effects of parasitic delay. Adjust the branching ratios to minimize these delays. You can write algebraic equations, but it is usually easier to use repeated evaluations of the delay equations for the competing paths, observing the effects of small adjustments. If a path is too slow, allocate more drive to that path by using a greater fraction of the input capacitance.

Although the general problem of optimizing a network requires a complex optimization algorithm, this procedure works well for most cases.

7.6 Exercises

7-1 [25] Modify Equation 7.5 to account for parasitic delay in the inverters, and solve for D. If Equation 7.7 is used as an approximation to determine x, how much does the resulting design differ from the optimum?

7-2 [20] In Section 7.3, we assume that the delay, d, of the second and third stages are equal. Why is this a good assumption?

7-3 [50] Develop better heuristics for selecting topologies and chosing gate sizes in the presence of capacitive interconnect.

Chapter 8

Asymmetric Logic Gates

Logic gates sometimes have different logical effort for different inputs. Such gates are called *asymmetric*. For example, the and-or-invert gate from Section 4.4 is inherently asymmetric. The 3-input XOR and majority gates from Sections 4.5.4 and 4.5.5 can be built in either symmetric or asymmetric forms, but the asymmetric forms have lower logical effort. Finally, normally symmetric gates such as NAND or NOR can be made asymmetric by sizing transistors to reduce the logical effort of one or more inputs at the expense of increasing the logical effort of the other inputs. These asymmetric gates can be used to speed up critical paths in a network by reducing the logical effort along the critical paths. This attractive property has a price, however: the total logical effort of the logic gate increases. This chapter discusses design issues arising from biasing a gate to favor particular inputs.

8.1 Designing asymmetric logic gates

Figure 8.1 shows a NAND gate designed so that the widths of the two pulldown transistors are allowed to differ: input a has width $1/(1-s)$, while input b has width $1/s$. The parameter s, $0 < s < 1$, called the *symmetry factor*, determines the amount by which the logic gate is asymmetric. If $s = 1/2$, the gate is symmetric, the pulldown transistors have equal sizes, and the logical effort is the same as we computed in Section 4.3. Values of s between 0 and $1/2$ favor the a input by making its pulldown transistor will be smaller than the pulldown transistor for b. Values of s between $1/2$ and 1 favor the b input.

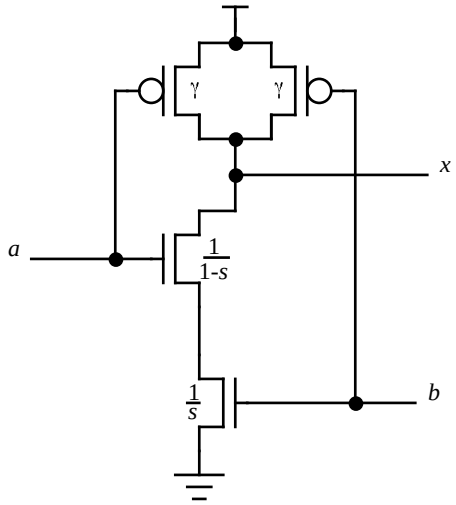


Figure 8.1: An asymmetric NAND gate.

Despite the flexibility to favor one of the two inputs, the gate still has the same output drive as the reference inverter with a pulldown transistor of width 1 and a pullup transistor of width γ. We can verify that the conductance of the pulldown connection is 1:

$$\frac{1}{\frac{1}{1/(1-s)} + \frac{1}{1/s}} = 1 \tag{8.1}$$

Using Equation 4.1, we can compute g_a and g_b, the logical effort per input for inputs a and b, and the total logical effort g_{tot}:

$$g_a = \frac{1/(1-s)+\gamma}{1+\gamma} \tag{8.2}$$

$$g_b = \frac{1/s+\gamma}{1+\gamma} \tag{8.3}$$

$$g_{tot} = \frac{\frac{1}{s(1-s)}+2\gamma}{1+\gamma} \tag{8.4}$$

The logical effort of input a is minimized by choosing s as small as possible, say 0.01. This design results in a pulldown transistor of width 1.01 for input a and a transistor of width 100 for input b. The logical effort of input a is then $(1.01+\gamma)/(1+\gamma)$, or almost exactly 1. The logical effort of input b becomes $(100+\gamma)/(1+\gamma)$, or about 34 if $\gamma = 2$. The total logical effort is about 35, again assuming $\gamma = 2$.

Extremely asymmetric designs, such as when $s = 0.01$, are able to achieve a logical effort for one input that almost matches that of an inverter, namely 1. The price of this achievement is an enormous total logical effort, 35, as opposed to $8/3$ for a symmetric design. Moreover, the huge size of the pulldown transistor will certainly cause layout problems, and the benefit of the reduced logical effort on input a may not be worth the enormous area of this transistor.

Less extreme asymmetry is more practical. If $s = 1/4$, the pulldown transistors will have widths $4/3$ and 4, and the logical effort of input a will be $(4/3+\gamma)/(1+\gamma)$, which is 1.1 if $\gamma = 2$. The logical effort of input b is 2, and the total logical effort is 3.1, which is very little more than $8/3$, the total logical effort of a symmetric design. This design achieves a logical effort for the favored input, a, that is only 10% greater than that of an inverter, without a huge increase in total logical effort.

Asymmetric gate designs require attention to stray capacitances. It is unwise, for example, to use values of $s > 1/2$ in the NAND gate design because the smaller pulldown transistor attached to input b not only must discharge the load capacitance but also must discharge the stray capacitance of the large pulldown transistor attached to input a. It is best to order transistors in series strings so that smaller transistors are near the output node. In the design shown in Figure 8.1, this means that we should use only values in the range $s \leq 1/2$, which favor the a input.

One can also approach the design of asymmetric logic gates by specifying the desired logical effort g_f of the favored input and deriving the necessary transistor sizes. This approach allows us to calculate the logical effort of the unfavored input, g_u in terms of the logical effort of the favored input. The following equation is derived from Equations 8.2 and 8.3 (see Exercise 8-1):

$$(g_f - 1)(g_u - 1) = \frac{1}{(1+\gamma)^2} \tag{8.5}$$

Equation 8.5 shows the symmetric relationship between the favored and unfavored logical efforts: the logical effort of the unfavored input will increase as the logical effort of the favored input decreases.

Figure 8.2 presents some results that summarize the effects of varying the symmetry factor of a two-input NAND gate. Recall that, in a single-stage design, an effort f will result in a delay of f delay units, plus parasitic delay. To achieve a 0.13 unit reduction in the delay of the favored input (1.33 to 1.2 units), we incur a 0.23 unit increase in the delay of the unfavored input (1.33 to 1.56 units).

The same design techniques we have illustrated for a two-input NAND gate apply to other logic gates as well. Rather than catalog all these designs, we shall

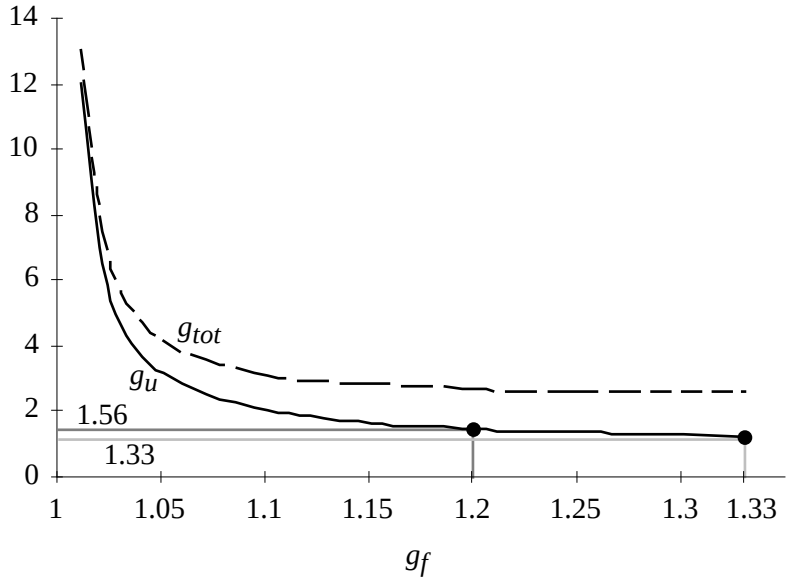


Figure 8.2: Relationship between favored and unfavored logical efforts for a two-input NAND gate with $\gamma = 2$.

develop and analyze asymmetric designs as the need arises. You might wish to repeat the analysis shown here for a two-input NOR gate or for a three-input NAND gate.

8.2 Applications of asymmetric logic gates

The principal applications of asymmetric logic gates occur where one path may be very fast or very slow. For example, in a ripple-carry adder or counter, the carry path must be fast. The best design uses an asymmetric circuit that speeds the carry even though the sum output is slowed somewhat.

Paradoxically, the other principal use of asymmetric logic gates occurs when a signal path may be unusually slow, as in a reset signal. Figure 8.3 shows a design for a buffer amplifier whose output is forced LOW when the reset signal, $\overline{reset}$, is LOW. The buffer consists of two stages: a NAND gate and an inverter. During normal operation, when $\overline{reset}$ is HIGH, the first stage has an output drive equivalent to that of an inverter with pulldown width 6 and pullup width 12, but the capacitive load on the in signal is 7+12 units of width. Thus the logical effort

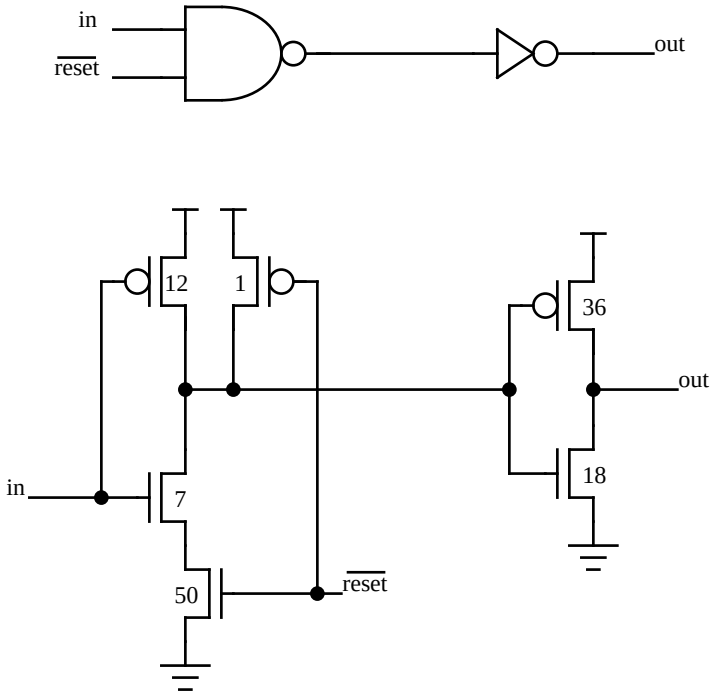


Figure 8.3: A buffer amplifier with reset input. When $\overline{reset}$ is LOW, the output will always be LOW.

of the in input is slightly larger than that of the corresponding inverter:

$$g = \frac{7 + 12}{6 + 12} = 1.05 \tag{8.6}$$

This circuit takes advantage of the slow response allowed to changes on $\overline{reset}$ by using the smallest pullup transistor possible. This choice reduces the area required to lay out the gate and partially compensates for the large pulldown transistor. The design violates the practice of designing gates that have the same drive characteristics as the reference inverter, because the pullup and pulldown drives controlled by $\overline{reset}$ differ from those of the standard inverter. In this case, the exception seems justified because we are interested only in performance when reset is not active, when the gate's output drive is nearly identical to that of the reference inverter.

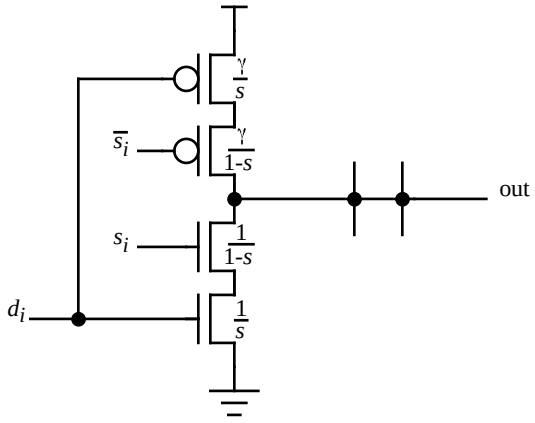


Figure 8.4: One arm of a multiplexer. The data input is d_i and the select bundle is s_i and $\overline{s}_i$.

8.2.1 Multiplexers

Just as the CMOS multiplexer is unique in that its logical effort per input does not grow as the number of inputs increases, asymmetric multiplexer designs have some peculiar properties. An n-way multiplexer can be viewed as containing n "arms," each of which contains the transistors connected to one data input and one select bundle (Figure 8.4). The unique properties of the multiplexer arise because the individual arms do not interact, so that each arm may be designed independently and is insensitive to the presence of other arms.[1]

An arm of a multiplexer may be asymmetric so as to favor the speed of the data or select signals. Favoring the select path may be appropriate when the control signals arrive late, such as in a carry select adder where the result is speculatively computed for both $carry = 0$ and $carry = 1$, then selected when the carry arrives. Values of $s < 1/2$ in Figure 8.4 will produce the required asymmetry.

Favoring the data input is more problematic. Choosing $1/2 < s < 1$ leads to suitable transistor sizes, but the design shown in Figure 8.4 will not tolerate much asymmetry before the stray capacitance of the larger transistors connected to select inputs slows the multiplexer and defeats the effort to reduce the load on the data input. In some cases, the data and select transistors can be interchanged in the pullup and pulldown chains to avoid loading the small transistors with large strays. This may lead to severe charge-sharing problems, depending on the design

[1]This is a first-order model. When stray capacitance is included in delay calculations, each arm of a multiplexer contributes a stray capacitance that indeed affects the other arms. See Chapter 11.

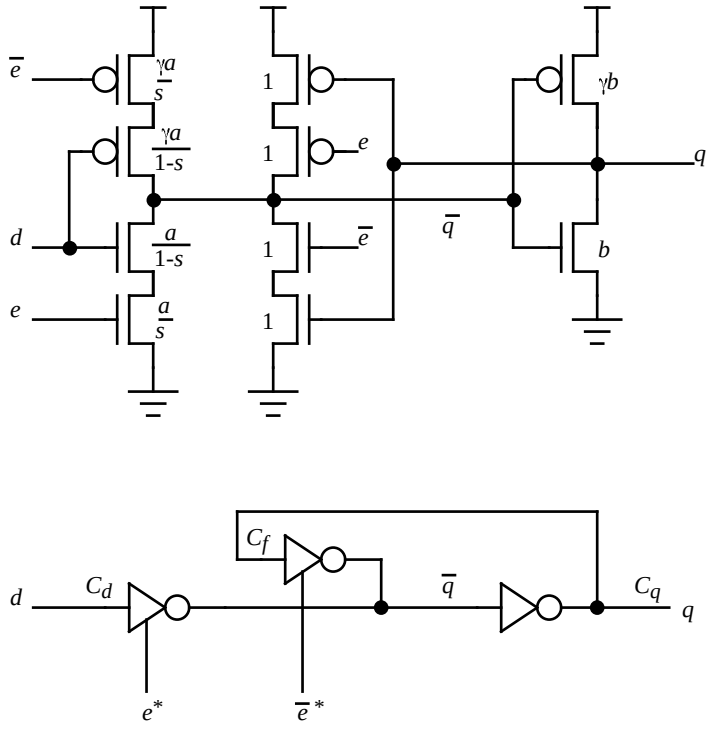


Figure 8.5: A static latch, consisting of a two-way multiplexer and an inverter. The data input is d and the latch output is q.

of the other arms of the multiplexer.

Multiplexers can be asymmetric in another way as well, by varying the conductance of different arms. Non-critical paths may use arms with lower conductance, and thus less input load. A good example of this kind of asymmetry is the static latch. Figure 8.5a shows a circuit diagram of a static latch, and Figure 8.5b shows a schematic representation in which each arm of the multiplexer is shown as a separate tri-state. The objective in the design will be to minimize the propagation delay through the latch when it is transparent, i.e., when e is HIGH and $\overline{e}$ is LOW.

The left arm of the multiplexer is configured to favor the data input, which will experience a logical effort of $1/(1-s)$. The transistor sizes on this arm are marked in terms of three parameters: s, the symmetry factor of the gate; a, an overall scale factor; and γ, the ratio of p- to n-type transistor widths. The right arm of the multiplexer can use minimum-size transistors because it is never

charges or discharges its output load, but rather supplies a trickle of current to counteract leakage.

Along the critical path from d to q, the logical effort of the d input to the multiplexer is $1/(1-s)$ and the logical effort of the inverter is 1, so that the logical effort of the path is $1/(1-s)$. The electrical effort is the ratio of the load on the inverter to the capacitance of the d input. If we define C_q to be the load capacitance, C_f to be the input capacitance of the multiplexer on the feedback path, and C_d to be the capacitance of the d input, the electrical effort is just $H = (C_q + C_f)/C_d$. Thus we have:

$$F = \left(\frac{1}{1-s}\right)\left(\frac{C_q + C_f}{C_d}\right) = \frac{C_q}{(1-s)rC_d} \tag{8.7}$$

where $r = C_q/(C_q + C_f)$ is the fraction of the inverter output drive available as useful output. It is clear from this equation that the effort is minimized for given input and output loads by maximizing r and $1-s$. Not surprisingly, this means minimizing the feedback capacitance, C_f, and biasing the multiplexer in favor of the d input as much as practical.

Modifying the multiplexer to favor the data input has the side effect of increasing the logical effort of the select bundle, $e*$. This need not impact speed because favoring the data input implies that the select was non-critical. Moreover, if the multiplexer is serves as a latch, the select is a clock signal whose delay can be absorbed into the clock distribution network. Nevertheless, biasing the select increases power consumption of the select driver, and so asymmetry should not be unreasonably large.

8.3 Summary

The theory of logical effort shows how to design logic gates with transistor sizes chosen so as to bias the logical effort in favor of one input at the expense of the remaining inputs. This will have the effect of reducing the delay on the path through the favored input, while increasing the delay on paths through the other inputs. Although biasing a gate in this way raises the total logical effort of the logic gate, the technique can be used to reduce the delay along critical paths.

The benefits of asymmetric designs are most evident when many asymmetric logic gates are connected serially along a path so as to reduce the delay along the path. Carry chains are an important application of such techniques.

8.4 Exercises

8-1 [15] Derive Equation 8.5 from Equations 8.2 and 8.3.

8-2 [25] Show how to design an asymmetric a 3-input NAND gate using two parameters $0 < s, t < 1$ to specify the logical effort on two of the three inputs. Derive an expression for the total logical effort in terms of s and t.

8-3 [20] Complete the design of the static latch shown in Figure 8.5 when $C_d = 9$, $C_q = 6C_d$, assuming $\gamma = 2$. What is the delay from d to q when the latch is transparent? The logical effort?

8-4 [20] Repeat the preceding exercise, but minimize the delay from d to $\overline{q}$, assuming the q output is not used at all.

8-5 [20] The left-most multiplexer arm in Figure 8.5 is by itself an inverting dynamic latch. The remaining circuits are required in order to make the latch static. What is the "cost" in terms of logical effort for making the latch static rather than dynamic?

8-6 [20] Suppose that the static latch of Figure 8.5 must drive a very large load, e.g., $C_q = 100$, while $C_d = 3$. How would you change the design?

8-7 [25] Figure 8.6a shows a conventional set-reset latch. (a) How should you choose symmetry factors s_1 and s_2 to obtain short and equal and set and reset delays, i.e., equal delays from the falling of $\overline{S}$ or $\overline{R}$ to the change on Q? You may need to make appropriate assumptions about the load on Q. (b) Devise modifications to the circuit in Figure 8.6b so that it is statically stable and so that the impact on logical effort is small. (c) Compare the operation and performance of the two circuits.

8-8 [25] Which of the NAND gates in Figure 8.7 might be made asymmetric in order to yield the fastest design? Assume that both inputs should be equally favored. Does your answer depend on the electrical effort of the "gate"?

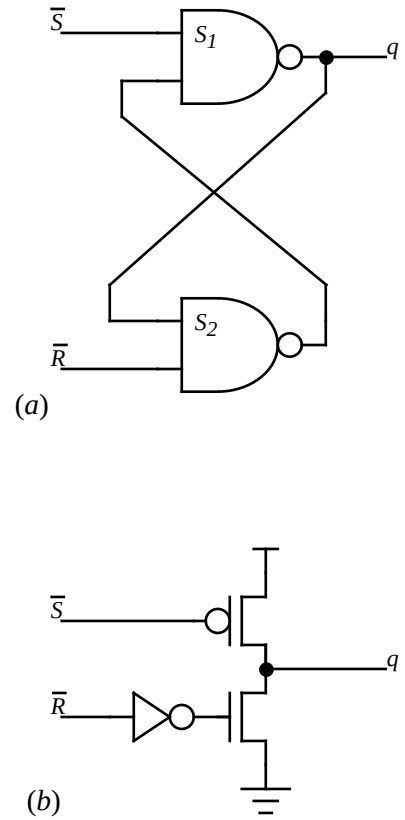


Figure 8.6: A set-reset flip-flop in conventional form, (a), and dynamic CMOS form, (b).

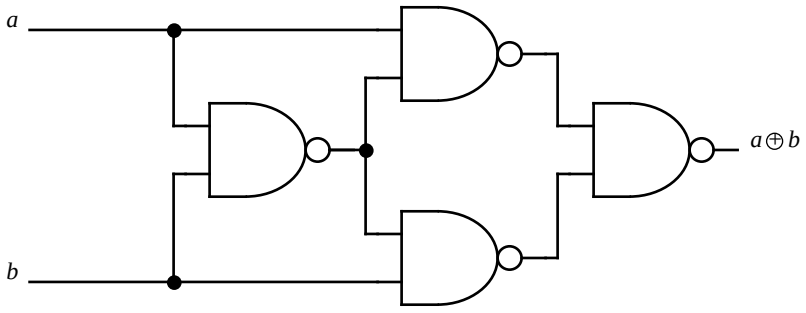


Figure 8.7: A network that computes the XOR function of two inputs.

Chapter 9

Unequal Rising and Falling Delays

All of our analysis of delays in CMOS logic gates has assumed that the delay of a logic gate is the same for rising and falling output transitions. It is easy to relax this condition and to consider the rising and falling delays separately. Allowing rise and fall times to differ permits us to analyze a greater range of designs, including pseudo-NMOS circuits, skewed static gates, CMOS domino logic, and precharged circuits of all kinds. It also allows us to design static CMOS gates with various choices for γ, the ratio of widths of PMOS to NMOS transistors.

The principal result is that for all but the most demanding cases, the techniques of logical effort can be used without modification to determine the best transistor sizes even when rising and falling delays differ. When calculating the total delay along a path, however, τ, the delay of a reference inverter, must be replaced by the average of the unequal rising and falling delays through the reference inverter.

Most often, the analysis need concern only the average of the rising and falling delays, because a signal flowing through a network of gates will alternately rise and fall as it propagates through each stage. Thus the number of rising and falling transitions differs by at most one, and the average stage delay is usually an adequate measure of the network's performance. If the speed of propagation of a particular transition is more important than that of other combinations, skewed static gates can reduce the logical effort of that transition at the expense of larger effort on other transitions.

One of the interesting applications of this analysis is to find the best value of γ, the ratio of widths of pullup to pulldown transistors in static CMOS designs. If γ is too small, then the rising transition will be too slow, because the conductance

of the pullup transistor will be diminished. On the other hand, if γ is too large, the rising transition will be suitably fast, but the transistor gate capacitance of the pullup transistor will be so large that the circuit driving it will slow down. The best value of γ will find a compromise between these extremes. It turns out that the value of γ for least total delay leads to rising and falling delays that differ.

9.1 Analyzing delays

The analysis of delays when rising and falling delays differ is a variation of the analysis we used in Chapter 3 when the delays are the same. In this section, we carry out the analysis and show that the average delay is minimized using exactly the same techniques of logical effort described earlier. In all cases, rising or falling refers to the output transition, not the input transition.

The delay of an individual stage of logic can be modeled by one of the following two expressions, derived using techniques similar to those in Section 3.2:

$$d_u = (g_u h + p_u) \tag{9.1}$$

$$d_d = (g_d h + p_d) \tag{9.2}$$

where the delays are measured in terms of τ. Notice that the logical efforts, parasitic delays, and stage delays differ for rising transitions (u) and falling transitions (d). The efforts and parasitic delays of each transition can be extracted from a plot of delay versus electrical effort, as discussed in Chapter 5. The electrical effort is independent of the transition direction.

In a path containing N logic gates, we use one of two equations for the path delay, depending whether the final output of the path rises or falls. In the equations, i is the distance from the last stage, ranging from 0 for the final gate to $N-1$ for the first gate.

$$D_u = \sum_{i\,\text{odd}} (g_{di} h_i + p_{di}) + \sum_{i\,\text{even}} (g_{ui} h_i + p_{ui}) \tag{9.3}$$

$$D_d = \sum_{i\,\text{odd}} (g_{ui} h_i + p_{ui}) + \sum_{i\,\text{even}} (g_{di} h_i + p_{di}) \tag{9.4}$$

The first equation models the delay incurred when a network produces a rising transition. In this equation, the first sum tallies the delay of falling transitions at the output of stages whose distance from the last stage is odd, and the second the delay of rising transitions in stages an even distance from the last stage, including

the last stage itself. Note that every path through a network of logic gates will experience alternating rising and falling transitions, as this sum indicates. Equation 9.4 is similar to its companion, but models the network producing a falling transition: the falling edges occur in even stages, and rising ones occur in odd stages. These two equations model the two separate cases we must consider.

In most cases, we want the delays experienced by launching a rising or falling transition into a network to be similar. The delays cannot, in general, be identical, because the two cases will experience different numbers of rising and falling delays. A reasonable goal is to minimize the average delay:

$$\begin{aligned} \overline{D} &= \frac{1}{2}(D_u + D_d) = \sum\left(\left(\frac{g_{ui}+g_{di}}{2}\right)h_i + \frac{p_{ui}+p_{di}}{2}\right) \\ &= \sum(g_i h_i + p_i) = \sum(f_i + p_i) \end{aligned} \tag{9.5}$$

subject to the usual constraint on the total effort, $F = \prod f_i$. Notice that the logical effort g_i and parasitic delay p_i of a stage are the average of the rising and falling quantities. Once again, the observation of Section 3.3 applies, and we see that the average delay is minimized by making the total effort borne by each stage the same, so that $f_i = f = F^{1/N}$. Then we have for the average delay:

$$\overline{D} = (Nf + P) = (NF^{1/N} + P) \tag{9.6}$$

This is the same result as we obtained for equal delays, Equation 3.20. Therefore, we are justified in using the average value of rising and falling logical effort and parasitic delay to minimize the average path delay, regardless of differences between rising and falling delays. All values are normalized so that the average logical effort of an inverter is 1.

The maximum delay through a path, however, may be different than the delay predicted by the original theory. To find maximum delay, one must select the worst of the delays for a rising output and for a falling output.

Example 9.1 *Size the path shown in Figure 9.1 for minimum average delay, using the logical effort and parasitic delay data from Table 9.1. What is the average and worst case delay?*

Notice how the average logical effort and parasitic delays in the table are the same as we are accustomed to, but that the rising values are larger than the falling values.

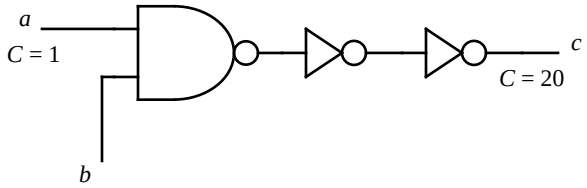


Figure 9.1: A NAND gate driving a heavy load.

Gate	Logical Effort g			Parasitic Delay p		
	Rising	Falling	Average	Rising	Falling	Average
Inverter	6/5	4/5	1	6/5	4/5	1
2-NAND	24/15	16/15	4/3	12/5	8/5	2
2-NOR	6/3	4/3	5/3	12/5	8/5	2

Table 9.1: Estimated logical effort and parasitic delay of various gates designed with $\gamma = 2$ in a process with $\mu = 3$.

We size the gates along the path for minimum average delay using average effort values. We find $G = 4/3$ and $H = 20$, so $F = 80/3$. Table 1.3 recommends $N = 3$, verifying our choice of two inverters in Figure 9.1. The effort of each stage is thus $\rho = (80/3)^{1/3} = 2.99$. Working from the output, we obtain the transistor sizes shown in Figure 9.2.

Now we can compute the delays from Equations 9.4 and 9.3:

$$\begin{aligned} D_u &= (g_{u1}h_1 + p_{u1}) + (g_{d2}h_2 + p_{d2}) + (g_{u3}h_2 + p_{u3}) \\ &= (24/15) \times (2.25/1) + 12/5 + (4/5) \times (6.72/2.25) + \\ &\quad 4/5 + (6/5) \times (20/6.72) + 6/5 = 13.96 \qquad (9.7) \\ D_d &= (g_{d1}h_1 + p_{d1}) + (g_{u2}h_2 + p_{u2}) + (g_{d3}h_2 + p_{d3}) \\ &= (16/15) \times (2.25/1) + 8/5 + (6/5) \times (6.72/2.25) + \\ &\quad 6/5 + (4/5) \times (20/6.72) + 4/5 = 11.96 \qquad (9.8) \end{aligned}$$

The average is 12.96, which agrees with the direct calculation using Equation 9.6.

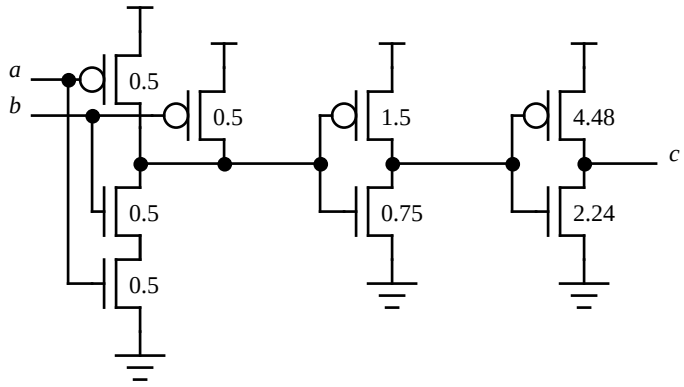


Figure 9.2: The network of Figure 9.1, optimized for the average of rising and falling delays, assuming $\gamma = 2$.

9.2 Case analysis

An alternative to the previous section is to consider minimizing the delay experienced by either a rising or a falling transition propagating along a path rather than minimizing the average delay. This problem frequently arises in precharged circuits where precharging sets a signal HIGH and we are interested in minimizing the propagation time of a falling transition as it travels through a network. In other words, we may want to minimize Equation 9.4 or Equation 9.3 alone, rather than the average of the two.

In such a case, we use the appropriate logical effort g_{xi} and parasitic delay p_{xi}, where x represents u or d for stages making rising or falling transitions, respectively. All of our theory of logical effort still applies. This method yields a short delay for the prescribed transition, but increases the complementary delay, as is shown in the following example.

Example 9.2 *Repeat Example 9.1, but with the objective of minimizing the propagation time of a rising transition presented at the input.*

Because the input rises, the NAND gate output will fall. The next inverter output will rise and the final inverter output will fall. Therefore, the logical efforts of interest are $(16/15)$, $6/5$, and $4/5$, respectively. The logical effort of the path is $(16/15)\times(6/5)\times(4/5) = 1.02$. The electrical effort is still $H = 20/1 = 20$, so the path effort is 20.5.

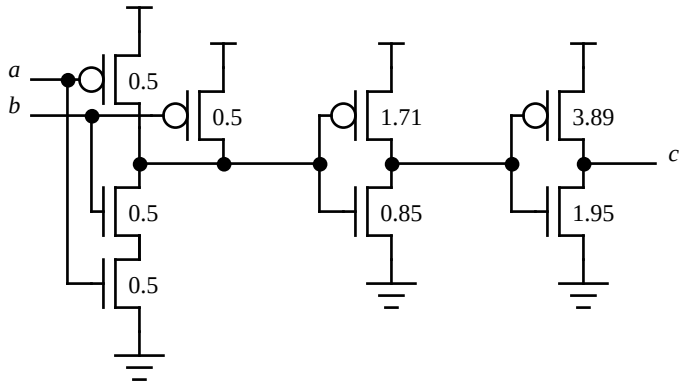


Figure 9.3: The network of Figure 9.1, optimized for the particular case of a rising transition entering on a. $\gamma = 2$.

The stage effort is therefore $\rho = 20.5^{1/3} = 2.74$. Working from the output, the sizes are 5.84, 2.56, and 1, as shown in Figure 9.3.

Let us now analyze the delays, again using Equations 9.3 and 9.4:

$$\begin{aligned} D_u &= (g_{u1}h_1 + p_{u1}) + (g_{d2}h_2 + p_{d2}) + (g_{u3}h_2 + p_{u3}) \\ &= (24/15) \times (2.56/1) + 12/5 + (4/5) \times (5.84/2.56) + \\ &\quad 4/5 + (6/5) \times (20/5.84) + 6/5 = 14.43 \qquad (9.9) \\ D_d &= (g_{d1}h_1 + p_{d1}) + (g_{u2}h_2 + p_{u2}) + (g_{d3}h_2 + p_{d3}) \\ &= (16/15) \times (2.56/1) + 8/5 + (6/5) \times (5.84/2.56) + \\ &\quad 6/5 + (4/5) \times (20/5.84) + 4/5 = 11.81 \qquad (9.10) \end{aligned}$$

Notice that the case we have optimized, $D_d = 11.81$, is indeed slightly better than the value of 11.96 given in Equation 9.8, obtained by optimizing the average delay. However, the complementary delay, 14.43, is substantially worse the the corresponding delay obtained for the previous design, 13.96. The effort delays f are all 2.74 for the critical transition, as we should expect, but are larger and unequal for the other transition.

This example shows clearly that optimizing one of the two complementary delays yields slightly faster circuits at the expense of a large increase in the delay of the other transition.

Gate	Logical Effort g		
	Rising	Falling	Average
Normal-skew Inverter	1	1	**1**
Normal-skew 2-NAND	4/3	4/3	**4/3**
Normal-skew 2-NOR	5/3	5/3	**5/3**
High-skew Inverter	**5/6**	5/3	5/4
High-skew 2-NAND	**1**	2	3/2
High-skew 2-NOR	**3/2**	3	9/4
Low-skew Inverter	4/3	**2/3**	1
Low-skew 2-NAND	2	**1**	3/2
Low-skew 2-NOR	2	**1**	3/2

Table 9.2: Estimated logical effort of skewed gates with $\gamma = 2$, $\mu = \gamma$. The important effort of each gate is bold.

9.2.1 Skewed gates

When one transition is more critical than another, it is possible to design the gate to favor that important transition. Such gates are called *skewed* gates and use a greater fraction of their input capacitance for the critical transistors. *High-skew* gates favor rising output transitions, while *low-skew* gates favor falling output transitions. Such gates are shown in Figure 9.4. Compare with *normal-skew* gates from Figure 4.1. Do not confuse skewed and asymmetric gates; skewed gates favor a particular transition, while asymmetric gates favor a particular input.

Skewed gates produce output current for the critical transition equal to that of a reference inverter, while producing less output current for the non-critical transition. Therefore, the input capacitance is less than for a gate which produces current equal to that of the reference inverter for both transitions. Hence, the logical effort is lower for the critical transition. The cost is greater logical effort for the non-critical transition. The logical efforts of various skewed gates are shown in Table 9.2, assuming $\gamma = \mu = 2$. Usually we are concerned only about the logical effort of the critical transition in skewed gates, not of the non-critical or average transition.

How far should a gate be skewed? In other words, how much smaller should the non-critical transistors be made? Extreme skews improve logical effort only slightly on the critical transition, but severely slow the non-critical transition. Such gates also can be difficult to lay out and may suffer hot electron reliability problems if the non-critical edge rates become too slow. A reasonable choice is

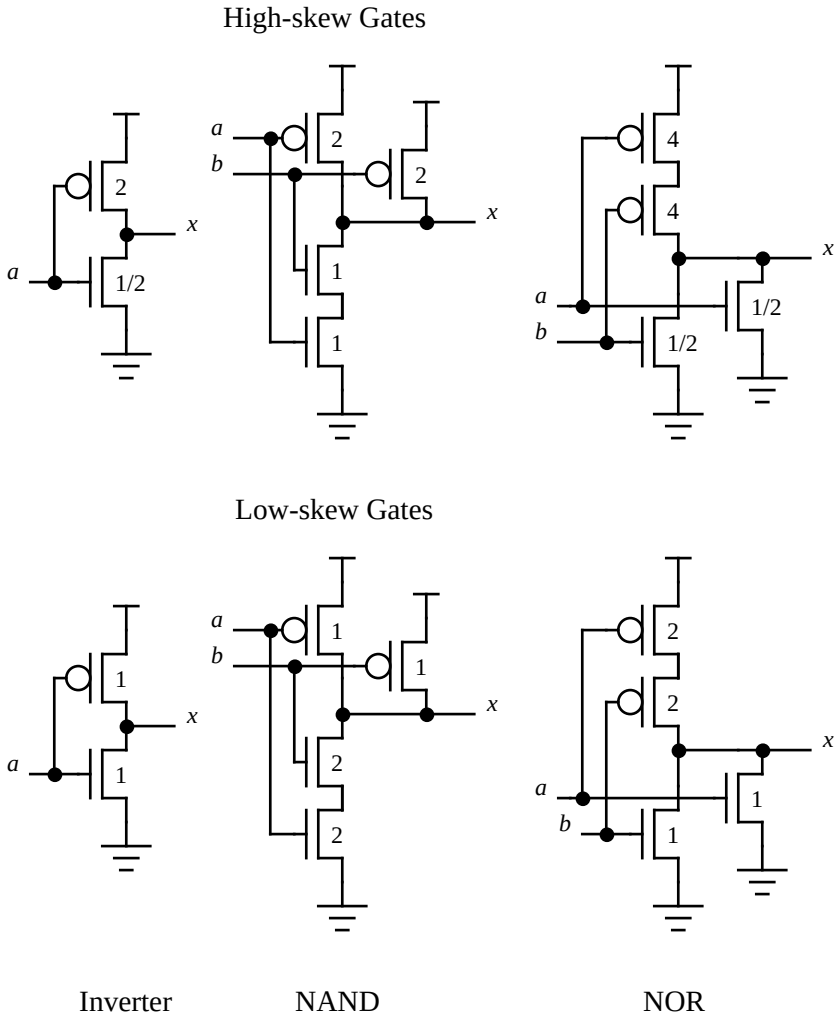


Figure 9.4: High-skewed and low-skewed inverters, NAND gates, and NOR gates, assuming $\gamma = 2$.

to make the non-critical transistors half the size they would have been in a normal gate, as was done in Figure 9.4.

9.2.2 Impact of γ and μ on logical effort

Before continuing, we will explore the effect of the shape factor γ and mobility ratio μ on the logical effort of gates. When $\gamma = \mu$, normal gates have equal rise and fall times. In practice, μ is between 2 and 3 and γ is usually less than μ. This does not affect the average logical effort of a normal gate, but it does lead to larger logical efforts for rising than for falling transitions, as was seen in Table 9.1. These logical efforts can be calculated by comparing the output current to the average of that of an inverter with equal input capacitance. The analysis shows that when $\gamma > \mu$, rising efforts of gates greater by a factor of $2\mu/(\mu + \gamma)$ and falling efforts are less by a factor of $2\gamma/(\mu + \gamma)$ than they would be for $\gamma = \mu$.

When $\gamma < \mu$, skewed gates also have larger rising efforts and smaller falling efforts. This can lead to results which at first seem counter-intuitive. For example, a high-skew NOR gate with $\gamma = 2$ is shown in Figure 9.4. If the mobility of the process is $\mu = 3$, the rising effort will be 9/5, which is actually larger than the average effort of a normal-skew NOR, 5/3! Does this mean that a high-skew NOR is worse than a regular NOR gate? The key to this puzzle is that the high-skew gate is only used for critical rising transitions and should be compared to the rising logical effort of the normal-skew gate, which is 2. Therefore, the high-skew gate is better than the normal gate for rising transitions, as expected.

Similarly, low-skew gates may appear to have unexpectedly low falling efforts. Dynamic gate logical effort is also measured for the falling transition and is lower than would be predicted assuming $\gamma = \mu$.

9.3 Optimizing CMOS P/N ratios

Now that we no longer insist that rising and falling delays be equal, we may ask what is the best ratio of pullup to pulldown transistor sizes in CMOS. The designs from Chapter 4 are simple to analyze because they have equal rising and falling delays. However, the wide PMOS transistors contribute a large amount of input capacitance and area. Could gates be smaller and on average faster if they use smaller PMOS transistors, thereby reducing the input capacitance and improving the falling delay at the expense of the rising delay? In this section, we will show that the P/N ratio giving best average delay is the square root of the P/N ratio

giving equal rising and falling delays. We will also see that the minimum is very flat, so the best size is a weak function of process parameters.

Consider a gate for which the P/N ratio, i.e., the ratio of the size of PMOS transistors to the size of NMOS transistors, is $k\mu$ for equal rising and falling delays. As we can recall from Chapter 4, for an inverter, $k = 1$. For a 2-input NAND gate, $k = 0.5$. For a 2-input NOR gate, $k = 2$. If the actual P/N ratio is r, the falling, rising, and average gate delay are proportional to:

$$\begin{aligned} d_d &\propto (1+r) \\ d_u &\propto \frac{k\mu}{r}(1+r) \\ d &\propto \frac{(1+\frac{k\mu}{r})(1+r)}{2} \end{aligned} \tag{9.11}$$

where the first term reflects the rising and falling currents and the second term reflects the input capacitance. Taking the partial derivative with respect to r and setting it to 0 shows that minimum average delay is achieved for:

$$r = \sqrt{k\mu} \tag{9.12}$$

For typical CMOS processes, $\mu = \mu_n/\mu_p$ is between 2 and 3, which implies that the best P/N ratio of an inverter is between 1.4 and 1.7.

How sensitive is delay to the P/N ratio? Figure 9.5 plots delay of a fanout-of-4 inverter as a function of γ, the inverter's P/N ratio, for three values of μ. It assumes $p_{inv} = 1$. The vertical axis has units of τ, the delay of a fanout-of-1 inverter with $\gamma = \mu = 1$.

Figure 9.5 shows that the delay curves are very flat near the best value of γ. Indeed, γ values from 1.4 to 1.7 give fanout-of-4 inverter delays within 1% of minimum for any value of μ from 2 to 3. Moreover, the minimum delay at $\gamma = \sqrt{\mu}$ is only 2-6% better than the delay at $\gamma = \mu$. The minimum is so flat that simulation-based optimization programs often do not converge to a minimum at $\gamma = \sqrt{\mu}$. However, the flat minimum is convenient because it means the γ value can be selected with little regard to actual process parameters. $\gamma = 1.5$ is a convenient choice because it offers good performance and relatively easy layout.

The most important benefit of optimizing the P/N ratio is not average speed, but rather reduction in area and power consumption. Remember, however, that rising and falling delays may differ substantially. For short paths, this may cause the worst-case delay to be significantly longer than the average delay. Also, certain

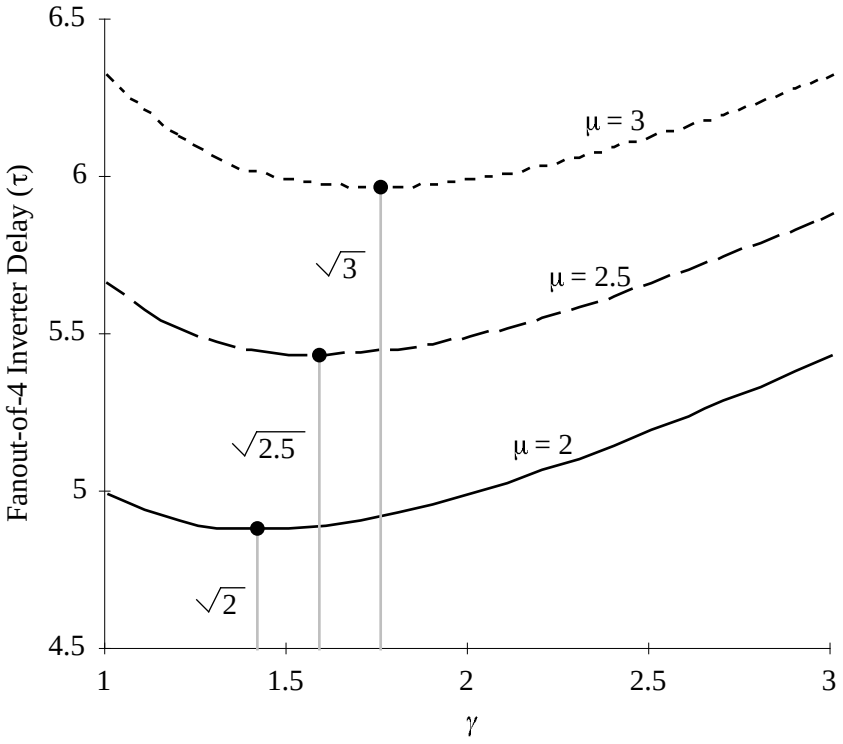


Figure 9.5: Delay vs. γ of fanout-of-4 inverters for $\mu = 2, 2.5, 3$.

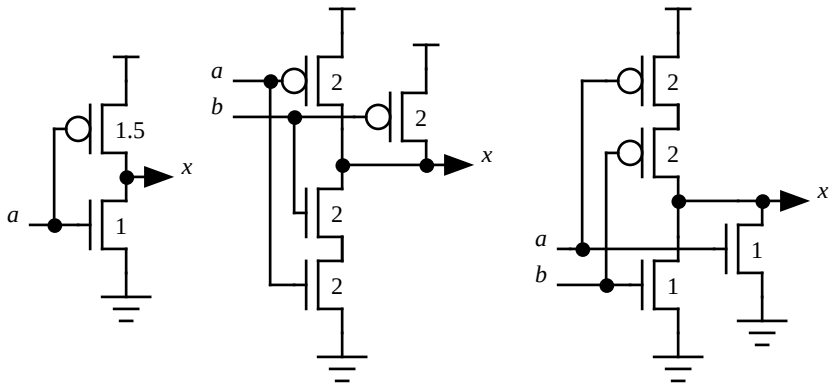


Figure 9.6: Inverter, NAND and NOR gates sized for improved area and average delay.

circuits such as clock drivers need equal rise and fall times and should be designed with $\gamma = \mu$.

Other gates should use a P/N ratio of about $\sqrt{k\mu}$. Typical values are 1 for a 2-input NAND, 1.5 for an inverter or multiplexer, and 2 for a 2-input NOR, as shown in Figure 9.6. The area and power savings are especially large for NOR gates.

9.4 Summary

The analysis presented in this chapter shows how to apply the theory of logical effort to designs using logic gates that introduce different delays for rising and falling outputs. We assign different rising and falling logical efforts, normalized such that the average logical effort of an inverter is 1. There are two ways to design paths with unequal rise/fall delays:

- Assume that each logic stage has the average of the rising and falling delays (Section 9.1). This method applies the techniques of logical effort without alteration. The maximum delay through a network may be slightly greater than the average delay.

- Use case analysis to minimize the delay of the particular transition whose propagation through the network must be fast (Section 9.2). The propaga-

tion delay of this transition can be reduced only at the expense of lengthening the delay of the complementary transition. Skewed logic gates can be used to favor a critical transition even more.

The analysis of delays also leads to a calculation for the best value of γ, the ratio of pullup transistor width to pulldown transistor width. While $\gamma = \mu$ yields equal rising and falling delays for an inverter, $\gamma = \sqrt{\mu}$ yields designs whose average delay is slightly less. Using $\gamma = 1.5$ yields designs within 1% of least delay over a wide range of processes and saves area and power relative to a circuit with equal rising and falling delays. The different rising and falling delays lead to slightly different logical efforts for gates. For simplicity, it is good enough to use the values of logical effort calculated in Chapter 4. However, if more accurate effort values are found from simulation or direct measurement, they may be used instead.

The analysis presented in this chapter must be used cautiously. The accuracy of our simple delay model of MOS logic gates is poor for modeling delay when arriving input signals have different rising and falling transitions. The accuracy would be improved by using a more accurate delay model, such as the one proposed by Horowitz [3], which considers the risetime of input transitions explicitly to predict the delay of a logic gate.

9.5 Exercises

9-1 [15] Sketch high-skew and low-skew 3-input NAND and NOR gates. What are logical efforts of each gate on its critical transition?

9-2 [20] Derive the rising, falling, and average logical efforts of the gates with unequal γ and μ in Table 9.1.

9-3 [20] Derive the rising, falling, and average logical efforts of skewed gates presented in Table 9.2.

9-4 [20] Derive the delay vs. γ information shown in Figure 9.5.

9-5 [15] Prove Equation 9.12.

Chapter 10

Circuit Families

So far, we have applied logical effort primarily to analyze static CMOS circuits. High-performance integrated circuits often use other circuit families to achieve better speed at the expense of power consumption, noise margins, or design effort. This chapter computes the logical effort of gates in different circuit families and shows how to optimize such circuits. We begin by examining pseudo-NMOS logic and the closely related symmetric NOR gates. Then we delve into the design of domino circuits. Finally, we analyze transmission gate circuits by combining the transmission gates and driver into a single complex gate.

The method of logical effort does not apply to arbitrary transistor networks, but only to *logic gates.* A logic gate has one or more inputs and one output, subject to the following restrictions:

- The gate of each transistor is connected to an input, a power supply, or the output; and
- Inputs are connected only to transistor gates.

The first condition rules out multiple logic gates masquerading as one, and the second keeps inputs from being connected to transistor sources or drains, as in transmission gates without explicit drivers.

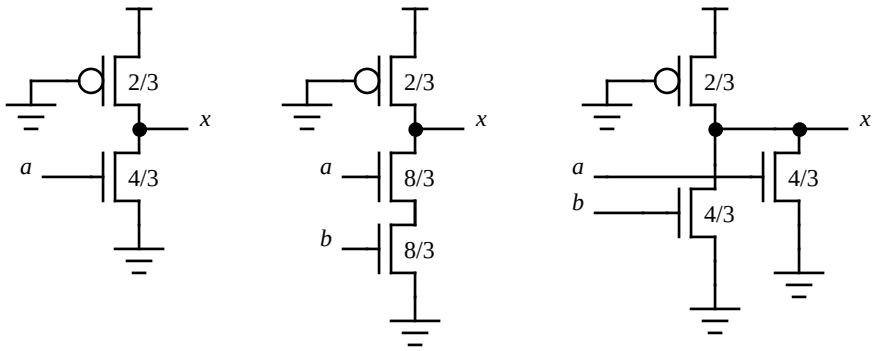


Figure 10.1: Pseudo-NMOS inverter, NAND and NOR gates, assuming $\mu = 2$.

10.1 Pseudo-NMOS circuits

Static CMOS gates are slowed because an input must drive both NMOS and PMOS transistors. In any transition, either the pullup or pulldown network is activated, meaning the input capacitance of the inactive network loads the input. Moreover, PMOS transistors have poor mobility and must be sized larger to achieve comparable rising and falling delays, further increasing input capacitance. Pseudo-NMOS and dynamic gates offer improved speed by removing the PMOS transistors from loading the input. This section analyzes pseudo-NMOS gates, while section 10.2 explores dynamic logic.

Pseudo-NMOS gates resemble static gates, but replace the slow PMOS pullup stack with a single grounded PMOS transistor which acts as a pullup resistor. The effective pullup resistance should be large enough that the NMOS transistors can pull the output to near ground, yet low enough to rapidly pull the output high. Figure 10.1 shows several pseudo-NMOS gates ratioed such that the pulldown transistors are about four times as strong as the pullup.

The analysis presented in Section 9.1 applies to pseudo-NMOS designs. The logical effort follows from considering the output current and input capacitance compared to the reference inverter from Figure 4.1. Sized as shown, the PMOS transistors produce 1/3 of the current of the reference inverter and the NMOS transistor stacks produce 4/3 of the current of the reference inverter. For falling transitions, the output current is the pulldown current minus the pullup current which is fighting the pulldown, $4/3 - 1/3 = 1$. For rising transitions, the output current

Gate type	Logical Effort g		
	Rising	Falling	Average
2-NAND	8/3	8/9	16/9
3-NAND	4	4/3	8/3
4-NAND	16/3	16/9	32/9
n-NOR	4/3	4/9	8/9
n-mux	8/3	8/9	16/9

Table 10.1: Logical efforts of pseudo-NMOS gates.

is just the pullup current, 1/3.

The inverter and NOR gate have an input capacitance of 4/3. The falling logical effort is the input capacitance divided by that of an inverter with the same output current, or $g_d = (4/3)/3 = 4/9$. The rising logical effort is three times greater, $g_u = 4/3$, because the current produced on a rising transition is only one third that of a falling transition. The average logical effort is $g = (4/9 + 4/3)/2 = 8/9$. This is independent of the number of inputs, explaining why pseudo-NMOS is a way to build fast wide NOR gates. Table 10.1 shows the rising, falling, and average logical efforts of other pseudo-NMOS gates, assuming $\mu = 2$ and a 4:1 pulldown to pullup strength ratio. Comparing this with Table 4.1 shows that pseudo-NMOS multiplexers are slightly better than CMOS multiplexers and that pseudo-NMOS NAND gates are worse than CMOS NAND gates. Since pseudo-NMOS logic consumes power even when not switching, it is best used for critical NOR functions where it shows greatest advantage.

Similar analysis can be used to compute the logical effort of other logic technologies, such as classic NMOS and bipolar and GaAs. The logical efforts should be normalized so that an inverter in the particular technology has an average logical effort of 1.

10.1.1 Symmetric NOR gates

Johnson [4] proposed a novel structure for a 2-input NOR, shown in Figure 10.2. The gate consists of two inverters with shorted outputs, ratioed such that an inverter pulling down can overpower an inverter pulling up. This ratio is exactly the same as is used for pseudo-NMOS gates. The difference is that when the output should rise, both inverters pull up in parallel, providing more current than is available from a regular pseudo-NMOS pullup.

The input capacitance of each input is 2. The worst-case pulldown current is

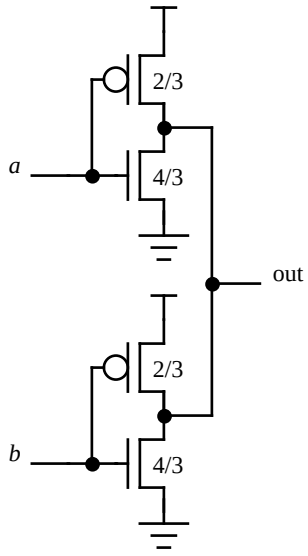


Figure 10.2: Johnson's symmetric 2-input NOR.

equal to that of a unit inverter, as we had found in the analysis of pseudo-NMOS NOR gates. The pullup current comes from two PMOS transistors in parallel and is thus $2/3$ that of a unit inverter. Therefore, the logical effort is $2/3$ for a falling output and 1 for a rising output. The average effort is $g = 5/6$, which is better than that of a pseudo-NMOS NOR and far superior to that of a static CMOS NOR!

Johnson also shows that symmetric structures can be used for wider NOR functions and even for NAND gates. Exercises 10-3 and 10-4 examine the design and logical effort of such structures.

10.2 Domino circuits

Pseudo-NMOS gates eliminate the bulky PMOS transistors loading the inputs, but pay the price of quiescent power dissipation and contention between the pullup and pulldown transistors. Dynamic gates offer even better logical effort and lower power consumption by using a clocked *precharge* transistor instead of a pullup that is always conducting. The dynamic gate is *precharged* HIGH then may *evaluate* LOW through an NMOS stack.

Unfortunately, if one dynamic inverter directly drives another, a race can corrupt the result. When the clock rises, both outputs have been precharged HIGH.

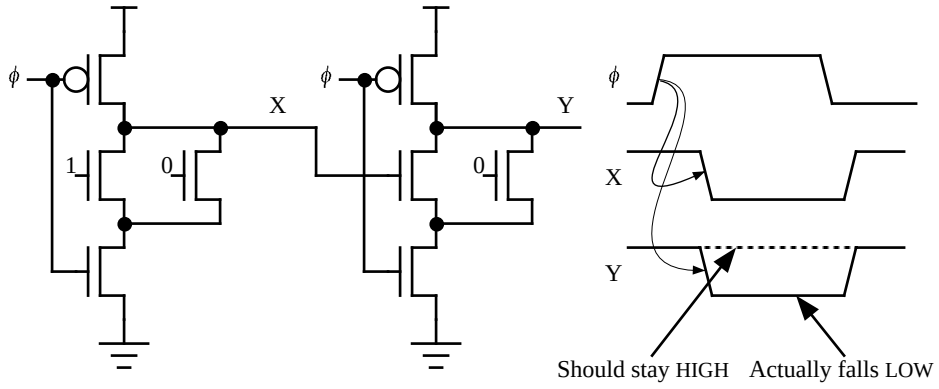


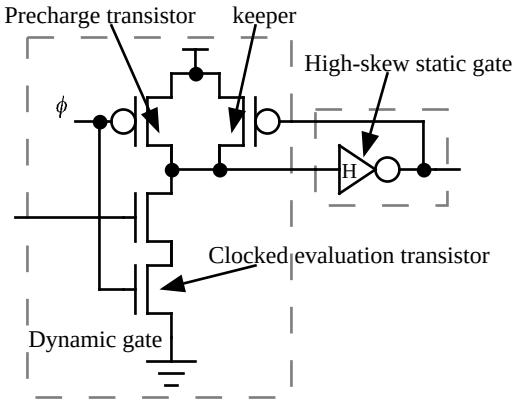
Figure 10.3: Dynamic gates cannot be cascaded directly.

The HIGH input to the first gate causes its output to fall, but the second gate's output also falls in response to its initial HIGH input. The circuit therefore produces an incorrect result because the second output will never rise during evaluation, as shown in Figure 10.3. *Domino* circuits solve this problem by using inverting static gates between dynamic gates so that the input to each dynamic gate is initially LOW. The falling dynamic output and rising static output ripple through a chain of gates like a chain of toppling dominos. In summary, domino logic runs 1.5 to 2 times faster than static CMOS logic [2] because dynamic gates present a much lower input capacitance for the same output current and have a lower switching threshold, and because the inverting static gate can be skewed to favor the critical monotonically rising evaluation edges.

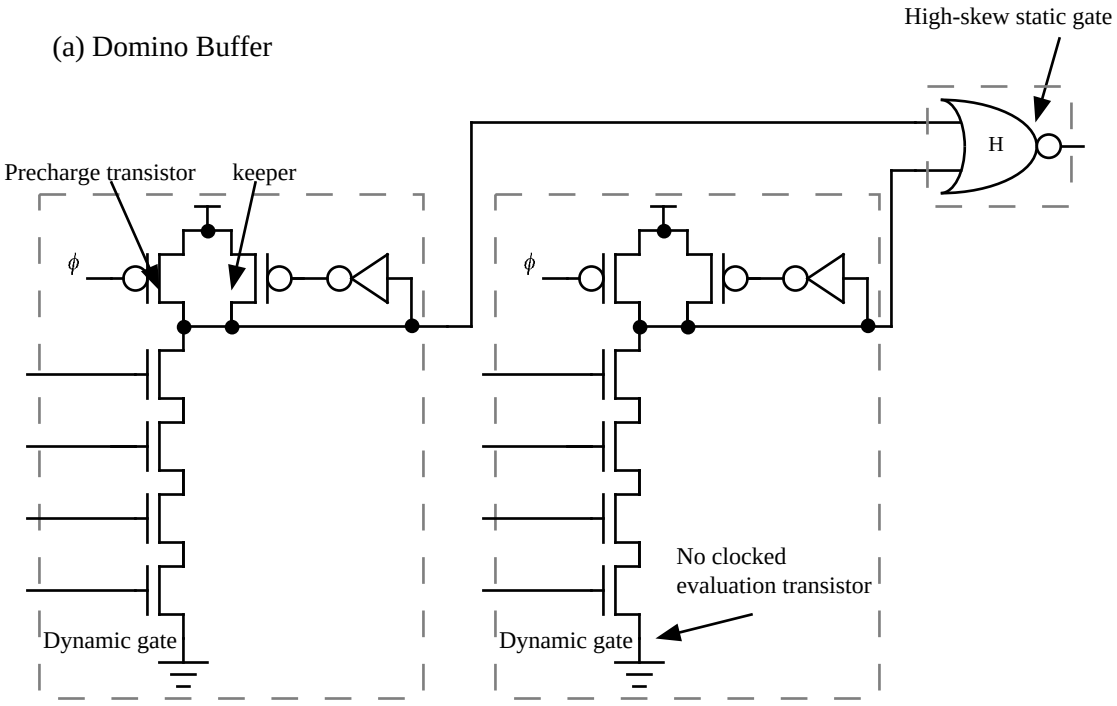
Figure 10.4 shows some domino gates. Each *domino* gate consists of a dynamic gate followed by an inverting static gate[1]. The static gate is often but not always an inverter. Since the dynamic gate's output falls monotonically during evaluation, the static gate should be skewed high to favor its monotonically rising output, as discussed in Section 9.2.1. We have calculated the logical effort of high-skew gates in Table 9.2 and will compute the logical effort of dynamic gates in the next section. The logical effort of a domino gate is then the product of the logical effort of the dynamic gate and of the high-skew gate. Remember that a domino gate counts as two stages when choosing the best number of stages.

A dynamic gate may be designed with or without a clocked evaluation transistor; the extra transistor slows the gate but eliminates any path between power

[1]Note that a domino "gate" actually refers to two stages, rather than a single gate. This is unfortunate, but accepted in the literature.



(a) Domino Buffer



(b) Domino 8-input AND

Figure 10.4: Domino buffer and 8-input AND gate.

and ground during precharge when the inputs are still high. Some dynamic gates include weak PMOS transistors called keepers so that the dynamic output will remain driven if the clock stops high.

Domino designers face a number of questions when selecting a circuit topology. How many stages should be used? Should the static gates be inverters, or should they perform logic? How should precharge transistors and keepers be sized? What is the benefit of removing the clocked evaluation transistors? We will show that domino logic should be designed with a stage effort of 2–2.75, rather than 4 that we found for static logic. Therefore, paths tend to use more stages and it is rarely beneficial to perform logic with the inverting static gates.

10.2.1 Logical effort of dynamic gates

The logical effort for dynamic gates can be computed just as for static gates. Figure 10.5 shows several dynamic gates with NMOS stacks sized for current equal to that of a unit inverter. Precharge is normally not a critical operation, so only the pulldown current affects logical effort. The logical efforts are shown in Table 10.2.

Logical effort partially explains why dynamic gates are faster than static gates. In static gates, much of the input capacitance is wasted on slow PMOS transistors that are not even used during a falling transition. Therefore, a dynamic inverter enjoys a logical effort only 1/3 that of a static inverter because all of the input capacitance is dedicated to the critical falling transition.

Our simple model for estimating logical effort fails to capture two other reasons that dynamic gates are fast. One is the lower switching threshold of the gate: the dynamic gate output will begin switching as soon as inputs rise to V_t, rather than all the way to $V_{DD}/2$. Another is the fact that velocity saturation makes the resistance of long NMOS stacks lower than our resistive model predicts. Therefore, simulations show that dynamic gates have even lower logical effort than Table 10.2 predicts.

Notice that dynamic NOR gates have less logical effort than NAND s and indeed have effort independent of the number of inputs. This is reversed from static CMOS gates and motivates designers to use wide NOR gates where possible.

10.2.2 Stage effort of domino circuits

In Section 3.4, we found that the best stage effort ρ is about 4 for static CMOS paths. This result depends on the fact that extra amplification could be provided

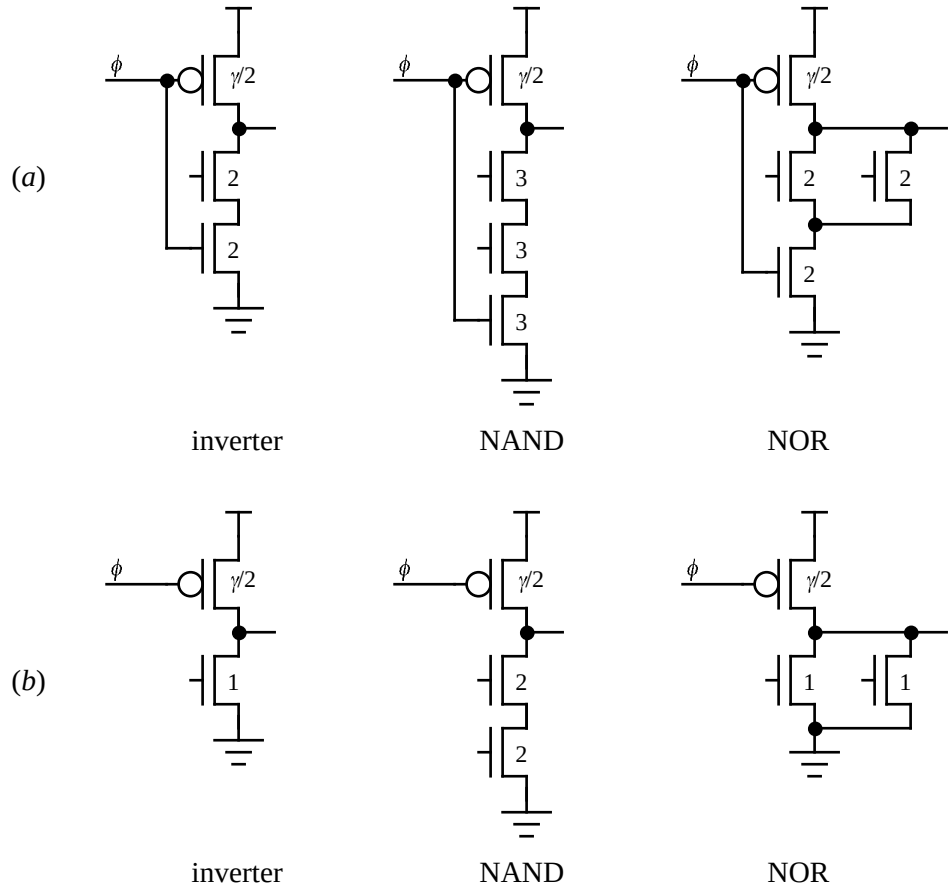


Figure 10.5: Dynamic gates (a) with and (b) without clocked evaluation transistors.

Gate type	Clocked evaluation transistor?	Formula	$n=2$	$n=3$	$n=4$
inverter	yes	$2/3$			
	no	$1/3$			
NAND	yes	$(n+1)/3$	1	4/3	5/3
	no	$n/3$	2/3	1	4/3
NOR	yes	$2/3$	2/3	2/3	2/3
	no	$1/3$	1/3	1/3	1/3
multiplexer	yes	1	1	1	1
	no	$2/3$	2/3	2/3	2/3

Table 10.2: Logical effort per input of dynamic gates.

by a string of inverters with logical effort 1. Domino paths are slightly different because extra amplification can be provided by domino buffers with logical effort less than 1. Adding more buffers actually reduces F, the path effort! Therefore, we would expect that domino paths would benefit from using more stages, or equivalently, that the best stage effort ρ is lower for domino paths. In this section, we will compute this best stage effort.

Our arguments parallel those in Section 3.4. We begin with a path that has n_1 stages and path effort F. We contemplate adding n_2 additional stages to obtain a path with a total of $N = n_1 + n_2$ stages. This time, however, the gates we add are not static inverters. Instead, they may be dynamic buffers. In general, the additional stages have logical effort g and parasitic delay p. The extra gates therefore change the path effort. The minimum delay of the path is:

$$\hat{D} = N\left(Fg^{N-n_1}\right)^{1/N} + \left(\sum_{i=1}^{n_1} p_i\right) + (N-n_1)p \tag{10.1}$$

We can differentiate and solve for $\hat{N}$ which gives minimum delay. When parasitics are non-zero, it proves to be more convenient to compute the best stage effort $\rho(g,p)$, which depends on the logical effort and parasitic delay of the extra stages. The mathematics is hideous, but the conclusion is remarkably elegant:

$$\rho(g,p) = g\rho(1,p/g) \tag{10.2}$$

where $\rho(1,p_{inv})$ is the best stage effort plotted in Figure 3.4 and fit by Equation 3.24. This result depends only on the characteristics of the stages being added, not on any properties of the original path.

Let us apply this result to domino circuits, where the extra stages are domino buffers. The logical effort of a domino buffer, like the one in Figure 10.4, is $10/18$ with series evaluation transistors and $5/18$ without. Estimate parasitic delays to be $(5/6)p_{inv}$ for a high-skew static inverter, p_{inv} for a dynamic inverter with series evaluation transistor. Since the buffer with series evaluation transistor consists of two stages, each stage has $g = \sqrt{10/18} = 0.75$ and $p = (11/6)p_{inv}/2 = (11/12)p_{inv}$. If $p_{inv} = 1$, the best stage effort is $\rho(0.75, 0.92) = 0.75\rho(1, 0.92/0.75) = 2.76$. The same reasoning applies to dynamic inverters with no clocked evaluation transistors having $g = 0.52$ and $p = 2/3$, yielding a best stage effort of 2.0.

In summary, domino paths with clocked evaluation transistors should target a stage effort around 2.75, rather than 4 used for static paths. If the stage effort is higher, the path may be improved by using more stages. If the stage effort is lower, the path may be improved by combining logic into more complex gates. Similarly, domino paths with no clocked evaluation transistors should target a stage effort of 2.0. Since it is impractical to leave out all of the clocked evaluation transistors, many domino paths mix clocked and unclocked dynamic gates and should target an effort between 2 and 2.75. As with static logic, the delay is a weak function of path effort around the best effort, so the designer has freedom to stray from the best effort without severe performance penalty.

10.2.3 Building logic in static gates

When, if ever, is it beneficial to build logic into the high-skew static gates? For example, consider the two ways of building an 8-input domino AND gate shown in Figure 10.6. One design consists of (dynamic 4-NAND, inverter, dynamic 2-NAND, inverter). Another design is (dynamic 4-NAND, high-skew 2-NOR). Which is better? The logical effort of the path is always larger when high-skew gates are used. However, the high-skew gate could reduce the parasitic delay if it reduces the total number of stages. If the stage effort of the first design is very small, the path may become faster by using fewer stages. But if the stage effort is large, the first design is best. In this section, we will quantify "small" and "large" to develop guidelines on the use of logic in static gates.

We can use our results from the previous section to address this question. The topology should be chosen to obtain a stage effort $\rho = \rho(g, p)$, where g and p are the mean logical effort and parasitic delay of the difference between the longer and shorter paths. For stage effort below ρ, the shorter path is better, meaning it is

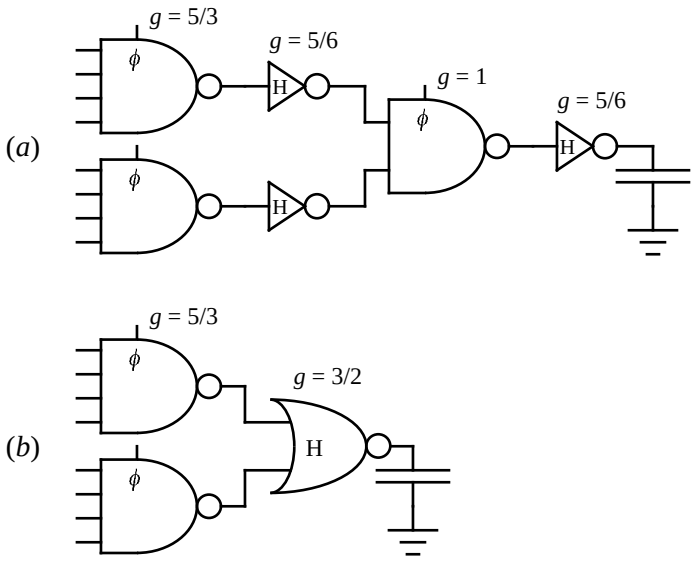


Figure 10.6: Two designs for 8-input domino AND.

advantageous to use logic in the static stage. For stage efforts above ρ, the longer path is better. An example will clarify this calculation:

Example 10.1 *Which of the designs in Figure 10.6 is best if the path has electrical effort of 1? If the path has electrical effort of 5?*

We could solve this problem by computing the delays of each design and directly comparing speed. Instead, we will use the stage effort criteria derived in this section. The change in logical effort from design (b) to design (a) is $\frac{(5/6)\times(1)\times(5/6)}{3/2} = 0.46$. This occurs over two extra stages, so the logical effort per stage is $g = \sqrt{0.42} = 0.68$. We could work out the parasitic delay exactly, but we recall that over a range of parasitics, $\rho(1, p)$ is about 4. Therefore, $\rho(0.68, p/0.68) \approx 0.68 \times 4 = 2.72$. If the stage effort is below 2.7, design (b) is best. If the stage effort is above 2.7, design (a) will be better.

Design (b) has a logical effort of $(5/3) \times (3/2) = 2.5$ and thus a path effort of $2.5H$ and stage effort of $\sqrt{2.5H}$. If $H = 1$, the stage effort is 1.6 and design (b) is best. If $H = 5$, the stage effort is 3.5 and design (a) would be better.

This seems like too much work for a simple example in which comparing delays is easy. The advantage of the method is the insight it gives: one should build logic into static gates only when the stage effort is below about 2.7. Moreover, it is best to first reduce the number of stages by using more complex dynamic gates. Simple calculations show that a dynamic gate with up to 4 series transistors followed by a high-skew inverter generally has lower logical effort than a smaller dynamic gate followed by a static gate. An exception is very wide dynamic NOR gates and multiplexers, which may be faster when divided into narrower chunks feeding a high-skew NAND gate to reduce parasitic delay.

In summary it is rarely beneficial to build logic into the static stages of domino gates. If a domino path has stage effort below about 2.7, the path can be improved by reducing the number of stages. The designer should first use more complex dynamic gates with up to 4 series transistors. If the stage effort is still below 2.7, only then should the designer consider replacing some of the static inverters with actual static gates.

10.2.4 Designing dynamic gates

In addition to the logic transistors, dynamic gates have transistors to control precharge and evaluation and to prevent the output from floating. How should each transistor be sized?

The size of the precharge transistor influences precharge time. A reasonable choice is to size it as if the dynamic gate were a low skew gate; hence the PMOS transistor can source half the current of the pulldown stack. Figure 10.5 uses such sizes.

The designer has several choices regarding the evaluation transistor. If the circuit inputs can be designed in such a way that there is no path to ground during precharge, the clocked evaluation transistor can be safely omitted. Even if there is a path to ground during part of precharge, the transistor can be removed if some extra power consumption is tolerable and the precharge transistor is strong enough to pull the output acceptably high in the time available. Frequently a clocked evaluation transistor is necessary. How big should it be?

One reasonable choice is to make the clocked evaluation transistor equal in size to the logic transistors in the dynamic gate. For higher speed, the clocked device can be made larger, just as an unbalanced gate can favor the critical inputs at the expense of the non-critical ones. For example, Figure 10.7 shows a dynamic inverter with a clocked pulldown twice as large as in Figure 10.5. The input transistor is selected so that the total pulldown resistance matches that of a normal

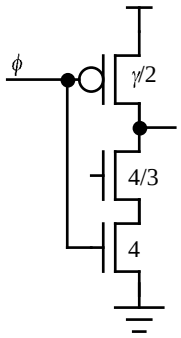


Figure 10.7: Dynamic inverter with double-sized clocked evaluation transistor.

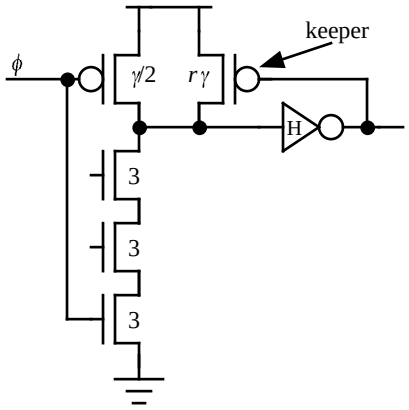


Figure 10.8: Keeper on a dynamic gate.

inverter. The logical effort is thus only 4/9, much better than the effort of 2/3 with a normal pulldown size and nearly as good as 1/3 for a dynamic inverter with no clocked pulldown. The main cost of large clocked transistors is the extra clock power. Therefore, a small amount of unbalancing such as 1.5 or $2\times$ is best.

Some dynamic gates use keepers to prevent the output from floating high during evaluation, as shown in Figure 10.8. The keepers also slightly improve the noise margin on the input of the dynamic gate. They have little effect on the noise margin at the output because they are usually too small to respond rapidly. The drawback of keepers is that they initially fight a falling output and slow the dynamic gate. How should keepers be sized?

The keeper current is subtracted from the pulldown stack current during evaluation. If the ratio of keeper current to pulldown stack current is r, the logical effort of the dynamic gate increases by $1/(1-r)$. Therefore, a reasonable rule of

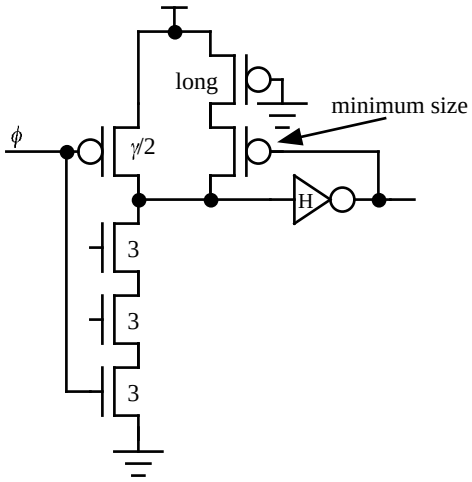


Figure 10.9: Weak keeper split into two parts.

thumb is to size keepers at $r = 1/4$ to $1/10$ of the strength of the pulldown stack. For small dynamic gates, this implies that keepers must be weaker than minimum sized devices. Increasing the channel length of the keeprs will weaken them, but also add to the capacitive loading on the inverter. A better approach is to split the keeper into two series transistors, as shown in Figure 10.9. Such an approach minimizes the load on the inverter while reducing keeper current.

10.3 Transmission gates

Many transmission gate circuits can be analyzed with the method of logical effort by incorporating the transmission gate into the logic gate that drives it. Figure 10.10 shows an inverter driving a transmission gate, and then shows the same circuit redrawn. The second circuit is essentially a leg of a multiplexer (Figure 4.4).

The PMOS and NMOS transistors in the transmission gate should be equal in width because both transistors operate in parallel while driving the output. The strong NMOS transistor helps the weaker PMOS transistor during a rising output transition. We can model the two transistors in parallel as an ideal switch with resistance equal to that of an NMOS transistor for both rising and falling transitions. A larger PMOS transistor would slightly improve current drive on rising outputs, but would add significant diffusion capacitance which slows both transitions.

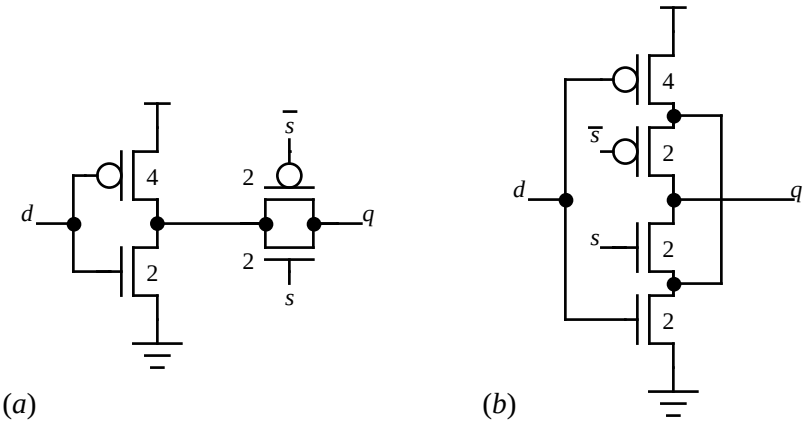


Figure 10.10: An inverter driving a transmission gate, and the same circuit redrawn so that it can be considered to be a single logic gate for the purposes of logical effort analysis.

Given this model, the circuit has drive equal to that of an inverter for both rising and falling transitions. The logical effort is 2 for input a and only $4/3$ for $s*$. This improvement in logical effort on $s*$ relative to a normal tri-state inverter comes at the expense of increased diffusion capacitance, leaving no great advantage for transmission gate tri-states over normal tri-states.

In general, transmission gate circuits are sized with equal PMOS and NMOS transistors and compared to an inverter with equal output current. As long as a delay equation such as Equation 3.6 describes the delay of a circuit, the method of logical effort applies. However, the parasitic capacitance increases rapidly with series transmission gates, so practical circuits are normally limited to about two series transmission gates.

A common fallacy when characterizing circuits with transmission gates is to measure the delay from the input to the output of the transmission gate. This makes transmission gate logic seem very fast, especially if the input were driven with a voltage source. As logical effort shows, the only meaningful way to characterize a transmission gate circuit is in conjunction with the logic gate that drives it.

10.4 Summary

This chapter used ideas of best stage effort, unbalanced gates, and unequal rise/fall delays to analyze circuit families other than static CMOS. Quantifying the logical effort of these circuit families enables us to understand better their advantages over static CMOS and to choose the most effective topologies.

We first examined ratioed circuits, such as pseudo-NMOS gates, by computing separate rising and falling logical efforts. The analysis shows that Johnson's symmetric NOR is a remarkably efficient way to implement the NOR function.

We then turned to domino circuits and found a remarkable result for the best stage effort of a path when considering adding extra stages, given in Equation 10.2. The equation tells us that the best stage effort of dynamic circuits is in the range of 2–2.75, depending on the use of clocked evaluation transistors. The equation also tells us when it is beneficial to break a path into more stages of simpler gates. We conclude that a path should incorporate logic into static gates only when the dynamic gates are already complex and the stage effort is still less than 2.7.

Finally, we explored transmission gate circuits. The logical effort of transmission gate circuits can be found by redrawing the driver and transmission gates as a single complex gate. Neglecting the driver is a common pitfall which makes transmission gates appear faster than they actually perform.

10.5 Exercises

10-1 [20] Derive the logical efforts of pseudo-NMOS gates shown in Table 10.1.

10-2 [20] Design an 8-input AND gate with an electrical effort of 12 using pseudo-NMOS logic. If the parasitic delay of an n-input pseudo-NMOS NOR gate is $(4n + 2)/9$, what is the path delay? How does it compare to the results from Section 2.1?

10-3 [25] Design a 3-input symmetric NOR gate. Size the inverters so that the worst-case pulldown is four times as strong as the pullups. What is the average logical effort? How does it compare to a pseudo-NMOS NOR? To static CMOS?

10-4 [20] Design a 2-input symmetric NAND gate. Size the inverters so that the worst-case pulldown is four times as strong as the pullups. What is the average logical effort? How does it compare to static CMOS?

10-5 [30] Prove Equation 10.2.

10-6 [25] Design a 4-16 decoder like the one in Section 2.2, using domino logic. You may assume you have true and complementary address inputs available.

10-7 [25] A 4:1 multiplexer can be constructed from two levels of transmission gates. Design such a structure and compute its logical effort.

Chapter 11

Wide Structures

One of the applications of logical effort is the analysis of wide structures, such as decoders or high fan-in gates and multiplexers, to find the topological structure that offers the best performance. This chapter presents four examples. The first is the design of an n-input AND structure. Then we design an n-input Muller C-element, in which the n-input AND structure can be used. Third, we present alternative designs for decoders that form 2^n selection outputs from an n-bit address. Finally, we analyze high fan-in multiplexers and show that it is best to partition wide multiplexers into trees of 4-input multiplexers.

11.1 An n-input AND structure

It is sometimes necessary to combine a large number of inputs in an AND function, for example, to detect that the output of an ALU is zero, or that a large number of conditions are all true. Let us find a circuit structure that minimizes the logical effort of the function.

11.1.1 Minimum logical effort

The simplest way to build an n-input AND function is to use an n-input NAND gate followed by an inverter. In Section 4.5.1, we found that the logical effort per

n	$G_{and}(n)$	Tree structure	Number of stages
1	1.0	1,1	2
2	1.333	2,1	2
3	1.667	3,1	2
4	1.778	2,1,2,1	4
5–6	2.222	3,1,2,1	4
7–8	2.370	2,1,2,1,2,1	6
9	2.778	3,1,3,1	4
10–12	2.963	3,1,2,1,2,1	6
13–16	3.160	2,1,2,1,2,1,2,1	8
17–18	3.704	3,1,3,1,2,1	6
19–24	3.951	3,1,2,1,2,1,2,1	8
25–32	4.214	2,1,2,1,2,1,2,1,2,1	10
33–36	4.938	3,1,3,1,2,1,2,1	8
37–48	5.267	3,1,2,1,2,1,2,1,2,1	10
49–64	5.619	2,1,2,1,2,1,2,1,2,1,2,1	12

Table 11.1: AND tree designs that minimize logical effort, for $\gamma = 2$. The tree structure gives the number of inputs of the gates at each level of the tree, starting with the NAND gate at the leaves, and ending with the NOR gate at the root.

input of this structure is:

$$g = \frac{n+\gamma}{1+\gamma} \tag{11.1}$$

Although this is a simple solution, its logical effort grows rapidly as the number of inputs increases. An n-input NOR gate could also be used, with an inverter on each input, to compute the AND function. But since the logical effort of an n-input NOR gate is always greater than that of an n-input NAND gate, this structure is not an improvement.

To avoid the linear growth of logical effort, we can build a tree of NAND and NOR gates to compute the AND function. Figure 11.1 shows such a tree: it has a NOR gate at the root, alternating levels of NAND and NOR gates, and an even number of levels. Observe that the number of inputs to gates at different levels in the tree may differ. In the figure, most levels have 2-input gates, but the gates at the leaves of the tree use 3-input gates. In some cases, the gates at certain levels in the tree may have only one input, i.e., they will be inverters. Figure 11.2 shows an example, in which the root NOR gate is an inverter.

The tree of Figure 11.1 has a logical effort per input of 6.17 (for $\gamma = 2$),

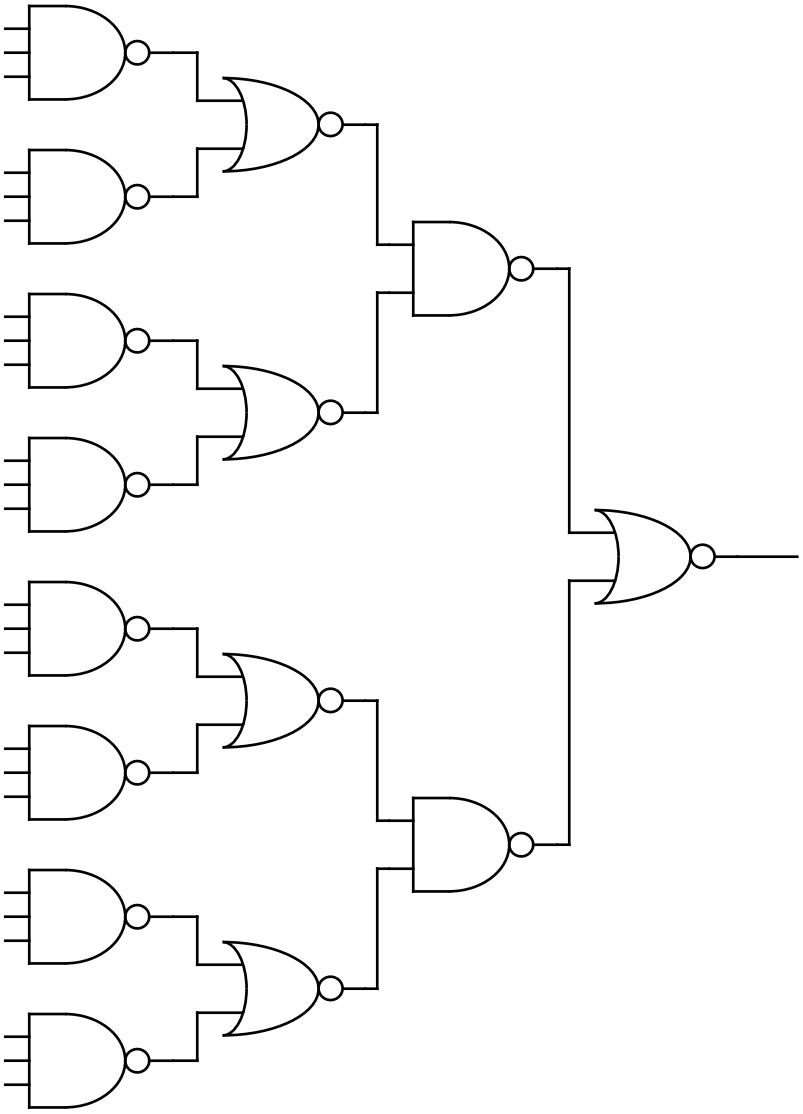


Figure 11.1: A 3,2,2,2 AND tree composed of alternating levels of NAND and NOR gates.

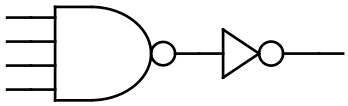


Figure 11.2: A degenerate case of the tree, in which the NOR gate has only one input, i.e., it is an inverter.

while an equivalent 24-input NAND gate and inverter would have a logical effort per input of 8.67. Of course, a 24-input NAND gate is also impractical because parasitic delay grows quadratically with stack height. The tree in Figure 11.1 does not yield the lowest logical effort for 24 inputs: as we shall shortly see, a tree with 8 levels and a logical effort per input of 3.95 is best.

A simple procedure can find the tree structure with the least logical effort. The design process searches recursively through all plausible tree structures with the right number of inputs. When designing a tree for n inputs, we first calculate the logical effort of using a single n-input gate at the current level, perhaps using inverters on its inputs to make the number of levels in the tree even. Then we consider trees with a b-input gate at the root, and subtrees that have $\lceil n/b \rceil$ inputs, where b ranges from 1 to n. The determination of the best subtree design is achieved by a recursive call on the same tree-design procedure. Care is required in the control of recursion to be sure that we don't explore endlessly deep trees that use 1-input gates at every level.

Logical effort offers several hints about the nature of the solution. Because the logical effort of NOR gates exceeds that of NAND gates, we expect the NOR gates in the tree to have fewer inputs than the NAND gates. In fact, to obtain minimum logical effort, all the NOR gates in the tree will have only one input—they will be inverters! Rather than inserting a NOR gate, the design procedure will find it advantageous to use a NAND gate at the next lower level in the tree.

Table 11.1 shows designs for trees with up to 64 inputs. Notice that the trees are very skinny, using only 2- and 3-input gates. Observe too that *no* NOR gates with multiple inputs are used. This table shows the minimum effort design for the 24-input problem. Note that it is a tree eight levels deep.

The results of these designs can be used to formulate a lower bound on the logical effort of an n-input AND tree. The tree will contain only 2-input NAND gates, alternating with inverters, with as many levels as necessary to accommodate n inputs. Thus if l is the number of levels of NAND gates, $2^l = n$, or $l = \log_2 n$.

n	$H = 1$		$H = 5$		$H = 200$	
	delay	tree	delay	tree	delay	tree
2	5.3	2,1	8.2	2,1	21.2	2,1,1,1
3	6.6	3,1	9.8	3,1	23.1	3,1,1,1
4	7.0	2,2	10.7	2,2	23.4	2,1,2,1
5-6	8.3	3,2	12.5	3,2	25.4	3,1,2,1
7-8	9.7	4,2	14.2	4,2	25.8	2,1,2,1,2,1
16	12.9	4,4	16.9	2,2,2,2	28.2	2,2,2,1,2,1
32	16.6	4,2,2,2	19.9	4,2,2,2	30.9	2,2,2,2,2,1
64	19.3	4,2,4,2	22.9	4,2,4,2	33.6	2,2,2,2,2,2
128	22.5	4,4,4,2	26.5	4,2,2,2,2,2	36.7	2,2,2,2,2,2,2,1

Table 11.2: AND tree designs that minimize delay, for $\gamma = 2$, when the total electrical effort is specified. Note that these trees are different than those in Table 11.1 because the electrical effort influences the number of stages to use.

The logical effort per input is:

$$G_{and}(n) = \left(\frac{2+\gamma}{1+\gamma}\right)^{\log_2 n} = n^{\log_2\left(\frac{2+\gamma}{1+\gamma}\right)} \tag{11.2}$$

When $\gamma = 2$, this simplifies to $G_{and} = n^{0.415}$. Note that the logical effort of AND trees grows much more slowly than the linear growth of a single NAND gate (Equation 11.1).

11.1.2 Minimum delay

While these skinny trees offer the least logical effort, they are not always the best choice in a given situation. It may happen that the path effort is so small that the best design requires fewer stages than are called for in the tree. For example, if $n = 6$ and the electrical effort $H = 1$, the design with the least delay is a 3-input NAND gate followed by a 2-input NOR. Only when the electrical effort is large will the skinny trees be fastest.

We can modify the design procedure to determine the fastest structure for a given electrical effort. Again, we use a procedure that evaluates all branching factors and recursively evaluates the required subtrees. When the procedure encounters a leaf node, it knows the logical effort of the proposed structure, so the path effort $F = GH$ can be calculated, and the delay can then be determined.

We find the best tree by minimizing this delay rather than minimizing the logical effort.

Table 11.2 shows some results for the electrical efforts of $H = 1$, $H = 5$, and $H = 200$. The trees with low effort are bushier than those which minimize logical effort because the limited total effort will make designs with too many stages slow. The trees with high effort, on the other hand, use one of the skinny designs from Table 11.1, possibly followed by additional inverters to yield the best number of stages. For example, when $n = 2$, the least logical effort tree has logical effort of 4/3. Thus, the path with electrical effort of 200 has total effort 800/3. Table 3.1 shows a 5-stage design would be fastest, but the number of stages must be even. Four stages turns out to be better than 6, so 2 additional inverters are used after the 2-stage minimum logical effort tree.

11.1.3 Other wide functions

DeMorgan's Law helps us to transform AND trees to compute OR, NAND, or NOR functions. In all cases, the trees have alternating layers of NAND and NOR gates. To perform the OR function, the order of NAND and NOR gates in the AND tree is reversed. Similar trees having an odd number of stages, obtained by appending an inverter to the tree, implement the complementary functions NAND and NOR. Hence, the minimum logical effort of all four functions is the same. The minimum delay trees can be found in much the same way that we computed Table 11.2.

11.2 An n-input Muller C-element

Muller C-elements are used in asynchronous circuit designs to detect when a group of processes have completed. The C-element's output becomes HIGH only after all of its inputs become HIGH, and its output becomes LOW only after all of its inputs become LOW. For other combinations of inputs values, the output of the C-element retains its previous state.

11.2.1 Minimum logical effort

Figure 11.3 shows the simplest way to build an n-input C-element, which we shall call a "simple-C." It consists of a dynamic "C-arm" with series pullup and pulldown transistors, followed by an inverter. The logical effort per input of this

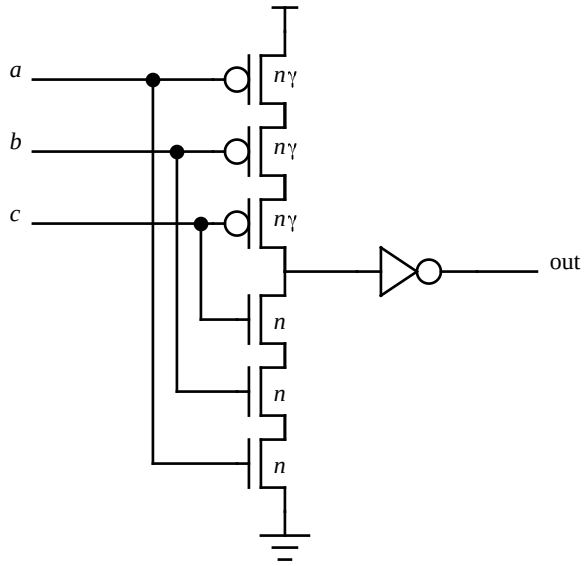


Figure 11.3: The "simple-C" design for a Muller C-element, with $n = 3$ inputs.

gate is just n (see Section 4.5.8). Variations of this dynamic circuit make it static by adding some form of feedback. Although the feedback will increase the logical effort slightly, we will ignore this effect.

Figure 11.4 shows another way to build a C-element using AND trees to detect when the inputs are all HIGH and when they are all LOW. We shall call this design an "AND-C." If we seek a design with the least logical effort, then the design of the two AND trees will be identical, and each will have the same logical effort. It might seem that the calculation of the logical effort for the entire C-element would require deciding on the fraction of input current that is directed into each AND tree. However, we can appeal to the results on bundles and observe that both top and bottom paths experience the same logical effort in the AND trees, and so signals x and y can be treated as a bundle, as shown in Figure 11.5. This bundle drives a circuit that is identical to an inverter, which has a logical effort of 1. So we see that the minimum logical effort of an n-input C-element is equal to the logical effort of an n-input AND tree. The design of these trees was addressed in the previous section.

If we study Table 11.1, we can see that the AND-C design has lower logical effort than the simple-C design for any number of inputs. The column labeled G_{and} in this table gives the logical effort of the n-input AND tree, which is the logical effort of the n-input AND-C design. In comparison, the logical effort of an

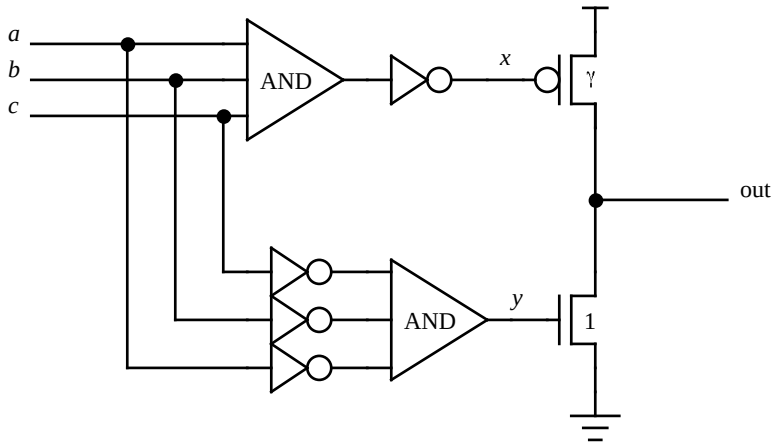


Figure 11.4: The "AND-C" design for a Muller C-element, using AND trees to determine when all inputs are HIGH or LOW.

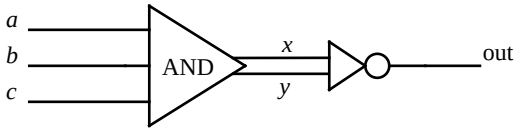


Figure 11.5: A different drawing of Figure 11.4, showing the bundle of two signals computed by the AND trees.

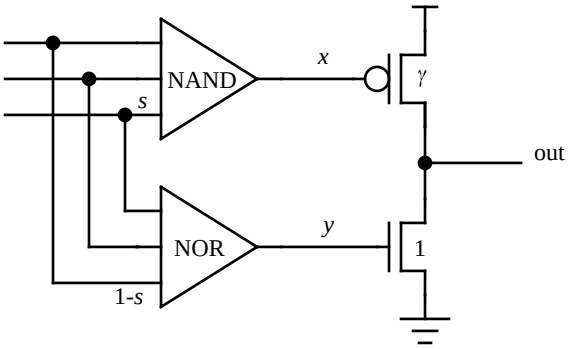


Figure 11.6: More precise version of Figure 11.4, showing NAND and NOR functions driving the output stage.

n-input simple-C design is n.

11.2.2 Minimum delay

As with AND trees, the structure with the lowest logical effort does not always offer the lowest delay, because it may have an improper number of stages. Obtaining the design with the least delay requires knowing the overall electrical effort that the C-element must bear.

The analysis of the simple-C circuit follows the familiar form used in all our logical effort calculations. If the electrical effort of the n-input gate is H, the total effort is nH, which is used to determine the best number of stages, N. Then the delay is $D = N(nH)^{1/N}$. The circuit can be modified to have the right number of stages by adding inverters or by building a tree of C-arms.

The analysis of the AND-C circuit depends on a slightly better design, shown in Figure 11.6. NAND and NOR functions, rather than the AND function and inverters shown in Figure 11.4, compute the x and y signals. These designs are logically equivalent, and have the same minimum logical effort. However, when we consider limited electrical effort, the improved design allows 2-stage solutions, while the design in Figure 11.4 must have at least four stages, because the AND tree has at least two.

As an initial design, we can assume that the NAND and NOR trees have equal logical efforts. Therefore, we should choose $s = \gamma/(1+\gamma)$ to divide the input capacitance between the legs of the fork in proportion to the load each leg drives. Of course, the logical effort and parasitic delay of the NOR tree is somewhat larger.

Therefore, speed could be improved somewhat by iteratively adjusting s until delays of the two legs are equal. The improvement is small and generally not worth the bother.

Table 11.3 compares designs for the simple-C and AND-C designs, when the electrical effort that must be borne by the circuit is $H = 1$ and $H = 5$. Notice that the AND-C design is almost always faster than the simple-C design, even for small numbers of inputs. The exceptions occur for designs with low electrical efforts where the AND-C requires too many stages.

Nevertheless, a closer look at Table 11.3 shows that the relative advantage of the AND-C design is small. This may seem surprising because the minimal logical effort of the AND-C design scales as $n^{0.415}$ while the minimal logical effort of the simple-C design scales as n. To understand why, consider the various components of delay. Equation 3.26 can be rewritten with $\rho = 4$ as:

$$D \approx 4\left(\log_4 G + \log_4 H\right) + P \tag{11.3}$$

This shows that logical effort is only one of three components of delay. The AND-C design can reduce the logical effort delay component by a factor of

$$\frac{\log n}{\log n^{0.415}} \approx 2 \tag{11.4}$$

by using a minimal effort tree. However, the parasitic delays are comparable to the logical effort delay, and so the reduced logical effort delay is a smaller fraction of total delay. Moreover, when electrical effort is small, the trees are bushy and do not achieve minimal logical effort; they may also have stage efforts well below optimal. When electrical effort is large, the electrical effort delay term is large, also making the savings in logical effort delay a smaller fraction of the total delay. The conclusion is that for both large and small electrical efforts, the overall speedup of the AND-C design is much less than one would expect by considering logical effort alone.

11.3 Decoders

Efficient decoders are important for addressing memories and microprocessor register files, where speed is critical. Decoding structures tend to have large total effort because the fanout of address bits to all decoders and the fanout of the decoder output to the transistors in the memory word are both large. In this section, we analyze three decoder designs from the perspective of logical effort.

n	H	simple-C		AND-C		
		delay	stages	delay	NAND tree	NOR tree
2	1	5.8	2	5.6	2	2
	5	9.3	2	8.8	2	1,2,1
3	1	7.5	2	7.1	3	3
	5	11.7	2	10.8	3,1,1	1,3,1
4	1	8.0	2	8.5	4	4
	5	12.9	2	12.7	2,1,2	2,2,1
6	1	9.9	2	12.2	3,1,2	2,3,1
	5	16.4	4	14.7	3,1,2	2,3,1
8	1	11.7	2	12.5	2,2,2	2,2,2
	5	17.1	4	15.3	2,2,2	2,2,2
16	1	16.0	2	15.1	4,2,2	2,4,2
	5	20.0	4	18.2	4,2,2	2,4,2
32	1	19.5	4	18.1	4,2,4	4,4,2
	5	25.2	6	21.6	4,2,4	2,2,2,2,2
64	1	23.3	4	21.2	4,4,4	4,4,4
	5	28.9	6	24.9	4,2,2,2,2	2,4,2,2,2

Table 11.3: Comparison of minimum-delay designs for Muller C-elements, for $\gamma = 2$, when the total electrical effort is specified.

The considerations that affect decoder design are many, and minimizing logical effort may not be paramount. Layout considerations are important, because often the decoder must fit on the same layout pitch as the memory cells it addresses. Overall decoder size and power are important; a design that minimizes logical effort may require too much power or too many transistors to be practical. Finally, many decoder structures use precharging to reduce logical effort; we will not analyze such designs here.

11.3.1 Simple decoder

The simplest form of decoder appears in Figure 11.7: each output is computed by an AND tree, wired to the true or complement form of n address bits. Each address bit is wired to 2^n decoders, half in true form and half in complement form. The path effort from an address bit to the output is therefore:

$$F = 2^n G_{and}(n) H \tag{11.5}$$

By way of example, consider a 64-entry file of 32-bit registers, so $n = 6$. If the load on each address bit is 8 times the capacitance of a register cell, $H = 32/8 = 4$. A lower bound on G_{and} from Equation 11.2 is 2.10, yielding a total effort of 538. This is a large total effort, which calls for a 5 stage AND tree design with minimum logical effort.

11.3.2 Predecoding

Figure =refwsf-8 illustrates the idea of predecoding. The n address bits form p groups of q each. Each group is decoded to yield 2^q predecode values. Then a second layer of p-input AND trees combines the predecoded signals to generate 2^n final signals. Let us compute the total effort on the longest path through the decoder. Each address bit fans out to 2^q decoders in the first layer, so there will be a branching effort of 2^q. The decoder will introduce a logical effort of $G_{and}(q)$, the logical effort of an AND tree with q inputs. Then there will be a fanout to 2^{n-q} AND gates in the second layer, each with logical effort $G_{and}(p)$. The path effort from an address bit to the output is:

$$F = 2^q G_{and}(q) 2^{n-q} G_{and}(p) H \tag{11.6}$$

We can compare this result with Equation 11.5, given $n = pq$. If we try a few values, we find that predecoding has about the same logical effort as the simple

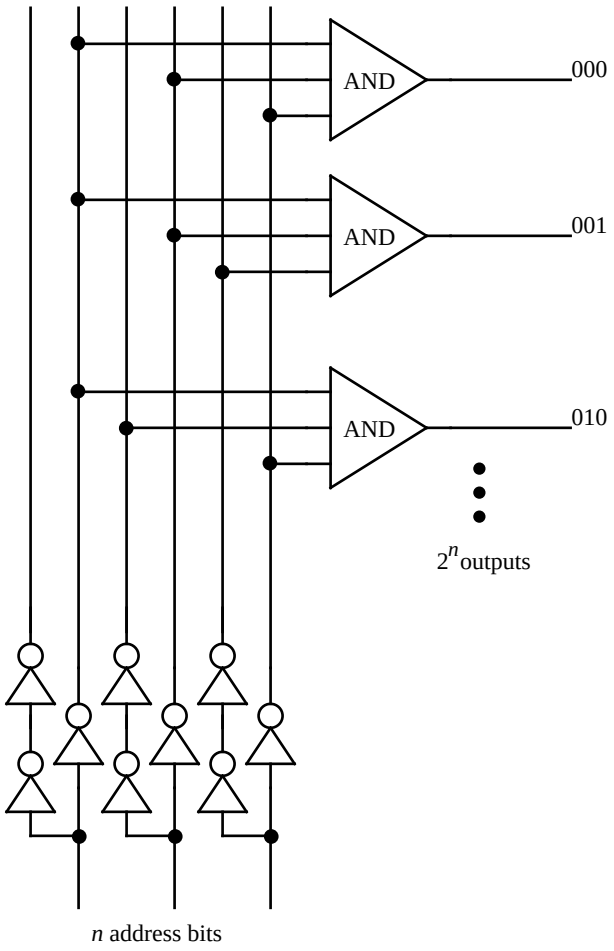


Figure 11.7: A decoder generating 2^n outputs from an n-bit address.

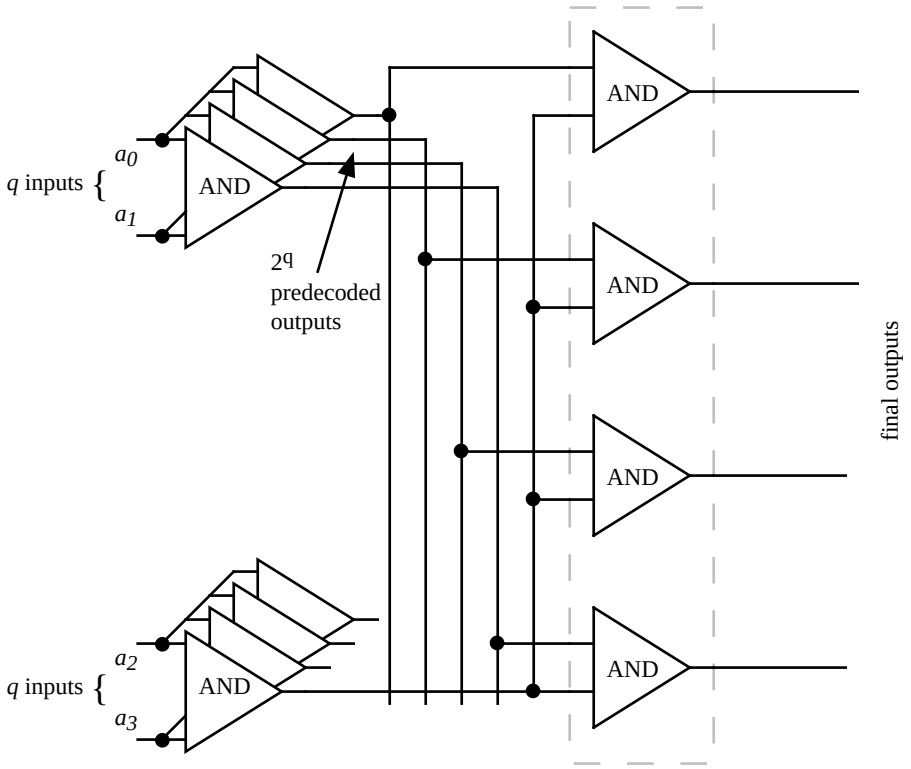


Figure 11.8: A decoder with q-bit predecoding.

structure. This result has an intuitive explanation: the predecoder structure is really a reorganization of an AND tree that moves the fanout from the address inputs to internal points in the tree.

Do not conclude from this analysis that predecoding offers no benefits. It requires fewer transistors than other designs, and leads to more compact structures than the skinny trees of the previous section. Predecoding represents an intermediate point between using a single n-input gate as a decoder and using a minimum-effort AND tree.

11.3.3 A better decoder

Lyon and Schediwy [5] have invented a decoder that reduces the logical effort by taking advantage of the fact that most outputs will be LOW. A NOR gate is a good way to pull an output LOW, but usually has poor logical effort because large series-connected PMOS transistors must pull the output HIGH. Since only one output of

a decoder will be HIGH, it is possible to share the PMOS transistors across all of the decoder gates! This is shown in Figure 11.9 for a 3:8 decoder. The decoder can be viewed as eight 3-input NOR gates that share PMOS pullups.

Rather than making the pullups all the same size, we shall make the transistors higher in the tree wider. The lowest-level pullups will have width w, the next pullup will have width $2w$, then $4w$, and so on. This scheme has the effect of loading the input lines equally, so they will all have equal logical effort. Other sizing schemes might reduce the logical effort of certain inputs and increase the logical effort of others. If we compute the conductance of the n series pullup transistors sized in this way, and equate it to the conductance of a PMOS transistor of width γ from the reference inverter, we find:

$$w = \gamma \left(\frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}} \right) \approx 2\gamma \tag{11.7}$$

Now that we have designed the decoder to have the same output drive as the reference inverter, the logical effort per input is just the input capacitance of each input, divided by $1 + \gamma$, the input capacitance of the reference inverter. Observe that each input is connected to 2^{n-1} pulldown transistors, each of width 1, and to a total pullup width of $2^{n-1}w$. So the logical effort per input is:

$$G(n) = 2^{n-1} \left(\frac{1 + \gamma \left(\frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}} \right)}{1 + \gamma} \right) \tag{11.8}$$

and the total effort is:

$$F = G(n)H \tag{11.9}$$

Compare this equation to Equation 11.5: the fanout of address bits to all parts of the decoder is incorporated into Equation 11.8. For $\gamma = 2$, the Lyon-Schediwy decoder has path effort:

$$F = G(2, n)H = 2^n \left(\frac{5 - 2^{2-n}}{6} \right) H \approx 2^n (5/6) H \tag{11.10}$$

The corresponding expression for the AND tree, using Equation 11.5 and the lower bound from Equation 11.2:

$$F = 2^n n^{0.415} H \tag{11.11}$$

The second factors in the two equations are the only differences, so we see that the Lyon-Schediwy decoder always has lower effort than the AND tree.

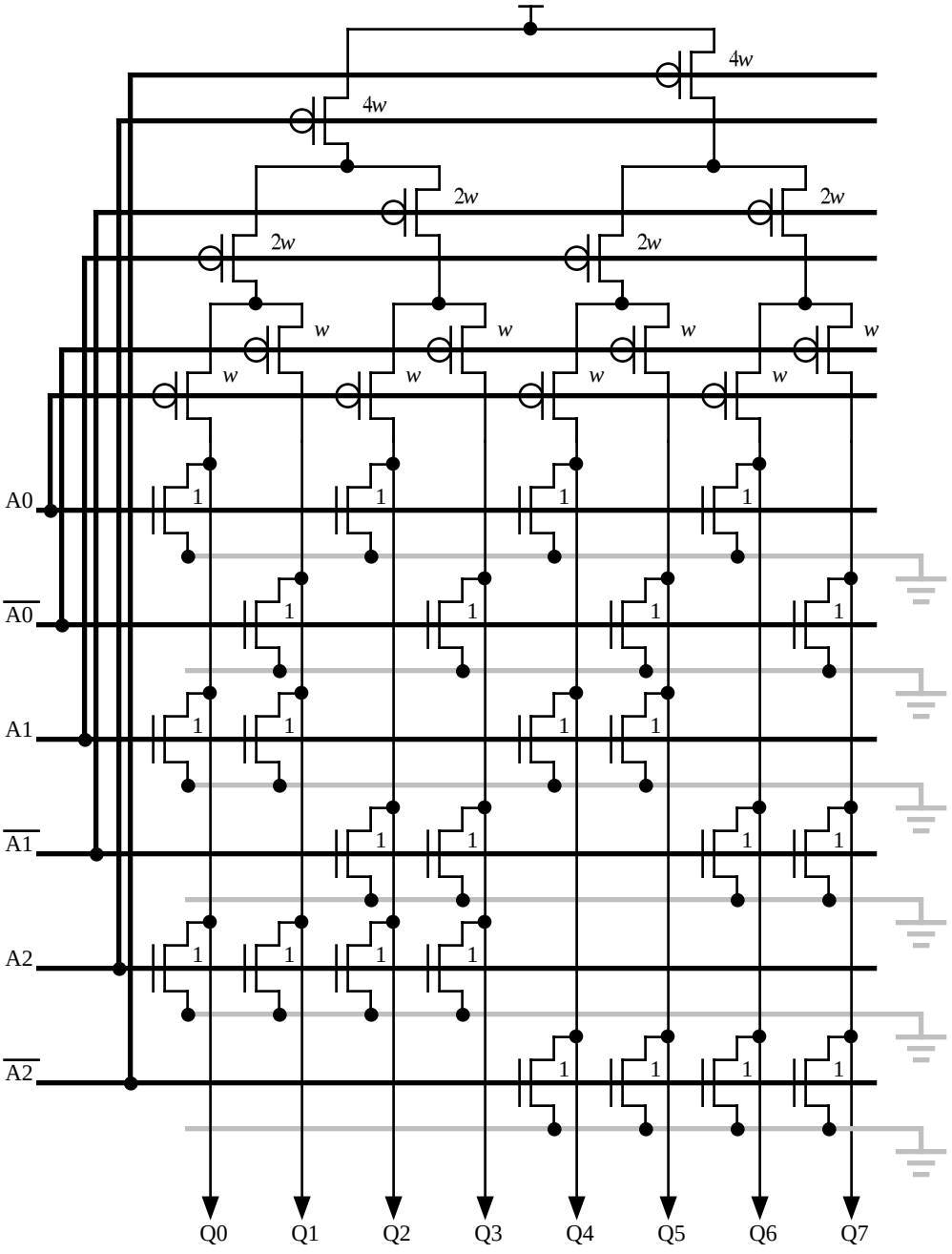


Figure 11.9: The Lyon-Schediwy decoder, for 3 inputs.

11.4 Multiplexers

CMOS multiplexers are interesting structures because the logical effort of a multiplexer is independent of the number of inputs. This suggests that multiplexers could have a large number of inputs without speed penalty. Common sense tells us otherwise. One problem is that decoding select signals for wide multiplexers requires large effort, though this does not impact delay from data inputs. When stray capacitance is considered, we discover that multiplexers should not be very broad at all. In fact, over a broad range of assumptions, the best multiplexer has four inputs. To select one of a large number of signals, we will see that it is best to build a tree of 4-input multiplexers. Nevertheless, it is sometimes beneficial to use multiplexers with up to 8 inputs.

11.4.1 How wide should a multiplexer be?

A multiplexer that selects one of r inputs has r independent *arms*. Figure 11.10 shows how each arm is designed and defines several important capacitances. C_{out} is the load capacitance driven by the multiplexer. C_{in} is the capacitance of the data input transistor gates. C_s is the stray capacitance, principly from drain diffusions, contributed by each arm of the multiplexer. Notice that the circuit has the selection signals s and $\overline{s}$ near the output so that unselected multiplexers present the least stray capacitance to their common output. Our model of stray capacitance (see Table 1.2) estimates a parasitic delay of $2rp_{inv}$ for an r-way multiplexer.

Figure 11.11 shows a branching structure of multiplexers that together select one of n inputs. The branching structure consists of n_m layers of multiplexing, each with an r-way branch. The branching structure is followed by n_a stages of amplifiers. Let C_{out} be the load capacitance driven by the amplifier string and C_{in} be the input capacitance of one of the n inputs. Thus the electrical effort per input is $H = C_{out}/C_{in}$.

We can now begin to develop some expressions that describe properties of the multiplexer tree. We have $n = r^{n_m}$, or alternatively $r = n^{1/n_m}$ or $n_m = (\ln n)/(\ln r)$. We will define the electrical effort of a multiplexer stage as h_m and the electrical effort of an amplifier stage as h_a. Using these values, we can compute:

$$d_m = 2h_m + 2rp_{inv} \quad (11.12)$$

$$d_a = h_a + p_{inv} \quad (11.13)$$

$$D = 2n_m(h_m + rp_{inv}) + n_a(h_a + p_{inv}) \quad (11.14)$$

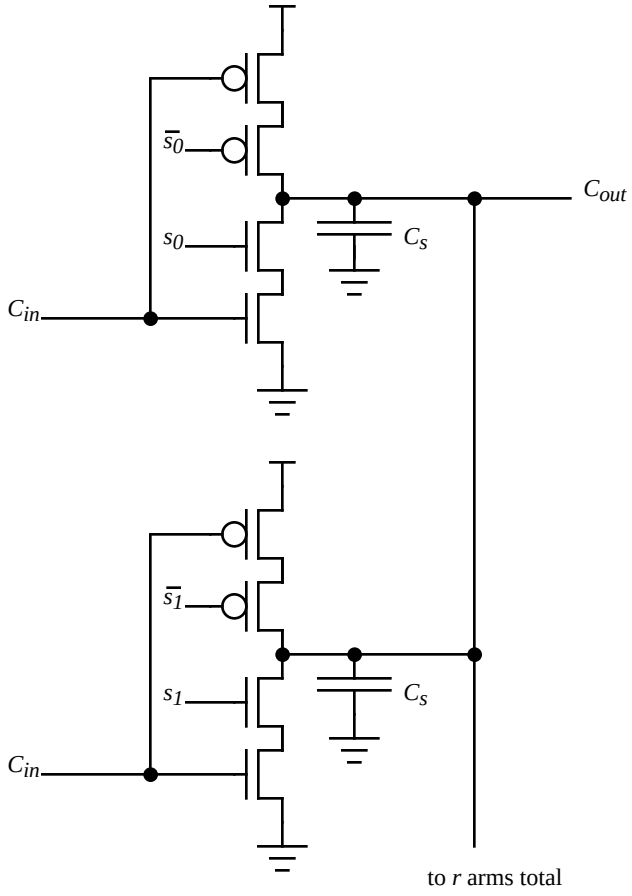


Figure 11.10: The transistor structure of an r-input multiplexer.

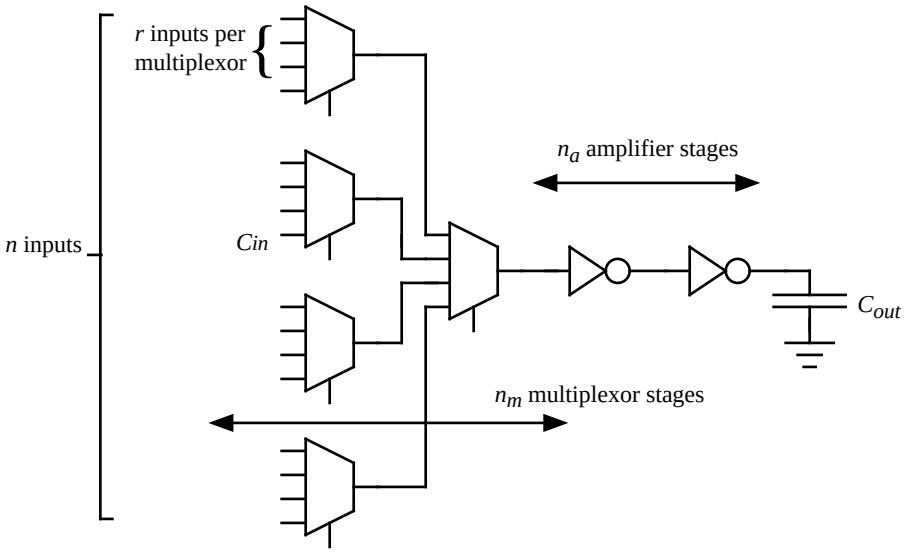


Figure 11.11: A tree of r-way multiplexers to select one of N inputs.

where d_m is the delay of a multiplexer stage; d_a, the delay of an amplifier stage; and D, the total delay.

Note how the logical effort per input of a multiplexer, 2, enters into the first equation. We want to minimize the delay D subject to a constraint on total electrical effort:

$$H = h_m^{n_m} h_a^{n_a} \tag{11.15}$$

The theory of logical effort teaches us that for the best speed, the effort borne by all stages should be equal, which suggests that $2h_m = h_a$. Although all stages have equal effort, they will not introduce equal delay, because the delay through a stage is the sum of effort delay and parasitic delay. But once again, the principal lesson of the theory of logical effort—to equalize the effort borne by each stage—results in the lowest overall delay, regardless of parasitic delay. However, the parasitic delays will influence the best number of stages.

Let us now turn to the selection of the best number of logic stages. The structure of the multiplexer tree requires that there be n_m multiplexer stages, but we can vary n_a, the number of amplifier stages, to achieve the minimum delay. Taking notice that $2h_m = h_a$ and $r = n^{1/n_m}$, we obtain from Equation 11.14:

$$D = (n_m + n_a)\left(2^{n_m} H\right)^{\frac{1}{n_m + n_a}} + 2n_m n^{1/n_m} p_{inv} + n_a p_{inv} \tag{11.16}$$

The first term is the familiar $NF^{1/N}$ effort delay in a network; the second term is

the parasitic delay of the multiplexer stages; and the third is the parasitic delay of the amplifier stages. We can find the fastest network by computing the values of n_m and n_a that minimize D. From the best value of n_m we can obtain the best multiplexer width $r = n^{1/n_m}$.

Before starting to minimize the delay, let us try to anticipate the result. Suppose that the overall electrical effort is H. We observe that the logical effort of the n_m multiplexer stages in cascade will be 2^{n_m}, so the total effort will be $F = 2^{n_m}H$. If the best effort borne by each stage should is ρ, the best number of stages is $n_m + n_a = (\ln F)/(\ln \rho)$. Solving for n_a, we obtain:

$$n_a = n_m \left(\frac{\ln 2}{\ln \rho} - 1 \right) + \frac{\ln H}{\ln \rho} \tag{11.17}$$

This equation shows that as the electrical effort grows, the number of stages increases. But it also shows that there will be cases where no amplifiers are required, i.e., $n_a = 0$. For example, if $H = 1$, then $n_a = 0$ because it is always true that $\rho \geq 2$ (see Equation 3.23). For values of H not much greater than one, the number of amplifiers will still be zero.

This result has an intuitive explanation. The logical effort of a multiplexer stage, 2, is less than the best step-up ratio, ρ, which is always $e = 2.718\ldots$ or greater. Thus a multiplexer stage has some "gain." If the electrical effort per stage is less than $\rho/2$, no additional amplifier stages are necessary. For sufficiently large electrical effort, of course, extra amplifiers are required.

Let us now find the best value of n_m, and therefore the best width for a multiplexer, $r = n^{1/n_m}$. Minimizing Equation 11.16 is quite complex, because there are two independent variables, n_m and n_a, and because the equation is itself complex. In the simple case that $H = 1$, we have observed that $n_a = 0$. In this case, we obtain:

$$D_{H=1} = 2n_m(1 + p_{inv}n^{1/n_m}) = 2\left(\frac{\ln n}{\ln r}\right)(1 + rp_{inv}) \tag{11.18}$$

Taking the partial derivative with respect to r and setting it to zero, we find:

$$1 + \frac{1}{rp_{inv}} - \ln r = 0 \tag{11.19}$$

This striking equation lets us calculate r, the width of a multiplexer, given only some information about the stray capacitance of the multiplexer design. The best width is independent of the total number of inputs, n.

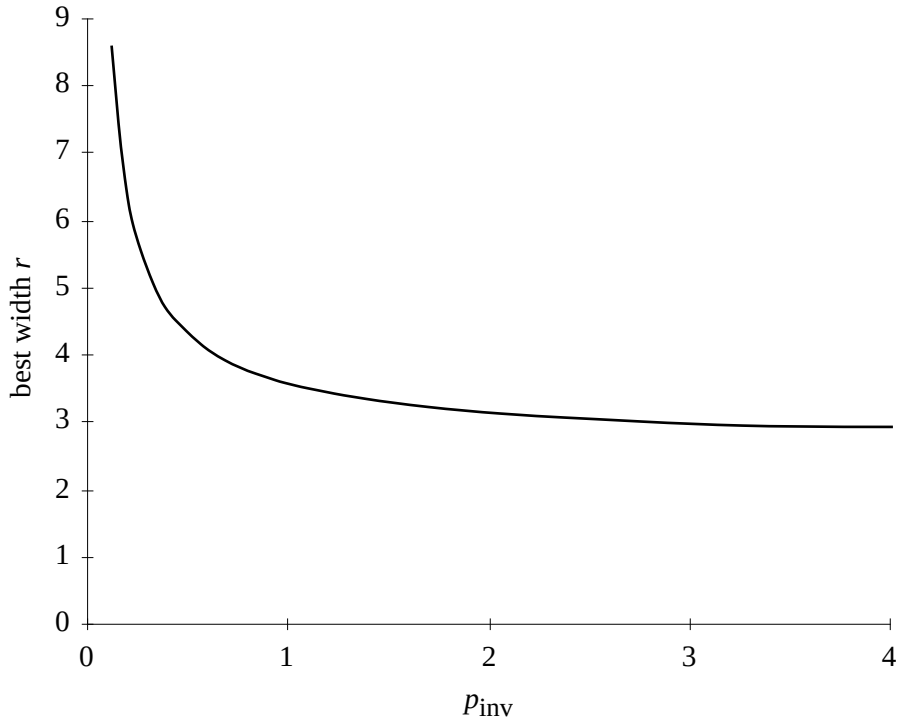


Figure 11.12: Best multiplexer width, as a function of stray capacitance p_{inv}.

Figure 11.12 plots r for different values of p_{inv}, computed using Equation 11.19. In practice, to make decoding manageable, we will require r to be a power of two. With this constraint, it is clear from the table that for reasonable contributions of stray capacitance, multiplexers should have four inputs.

To be sure of this result, we should analyze Equation 11.14 for electrical efforts other than one. This analysis leads to slightly different values for r than those predicted by Equation 11.19, but the best width for a practical multiplexer is still four!

11.4.2 Medium-width multiplexers

The analysis in the last section offers advice on designing very wide multiplexers as trees of 4-input multiplexers. But what if a multiplexer has 6 inputs? 10 inputs? Is it better to build a partial tree of a 4-input multiplexer followed by a 2 or 3-input multiplexer, or to use a single 6 or 10-input multiplexer? Such medium-width

multiplexers are common in superscalar execution unit bypass paths.

The answer depends on the electrical effort borne by the path as well as the number of inputs. At higher electrical efforts, a 2-stage design is helpful to drive the load as well as to reduce parasitic capacitance. Therefore, we consider three topologies for medium-width multiplexers:

- an n-input multiplexer
- an n-input multiplexer followed by an inverter
- a 4-input multiplexer followed by an $\lceil n/4 \rceil$ multiplexer

The best design can be determined by comparing the delay equations of the three choices. Figure 11.13 shows the ranges of n and H for which each design is best. The choice between the first and second designs of adding an inverter depends on the electrical effort: larger electrical efforts are best driven by more stages. The third design is better than than the first driving large electrical efforts, but it is not as good as the second. Therefore, the number of inputs at which the multiplexer is best divided into a tree varies with the electrical effort. At electrical efforts above 12, it is worth considering three stage designs as well.

The plot shows that it is useful to have multiplexers with up to 6 or 7 inputs in a library. This cutoff depends on the parasitic capacitance; if the capacitance were cut by two, multiplexers with 8-10 inputs become useful.

11.5 Summary

This chapter surveyed a number of tree-structured designs. The logical effort of a tree structure grows more slowly with the number of inputs than does the logical effort of a single gate that computes the same function. We made two observations from the design of the tree structures in this chapter:

- Trees that minimize logical effort are deep and have low branching factor, i.e., the number of inputs to gates is 2 or 3. Moreover, NOR gates never appear in these trees, because a NAND gate and inverter computes the same function with less logical effort.
- It is not always advisable to use the tree with the lowest logical effort, because the tree may have too many stages for best speed. Bushier trees with larger logical effort and fewer stages may result in less delay.

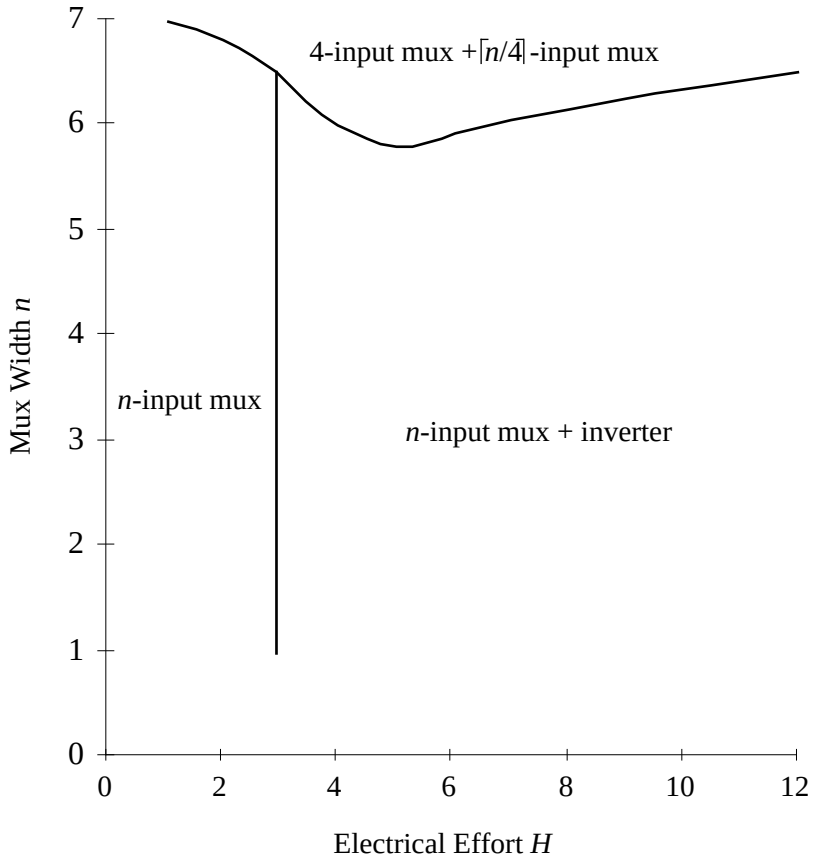


Figure 11.13: Best multiplexer design given n and H, assuming $p_{inv} = 1$.

We have applied the tree structures to the design of C-elements and decoders. We have seen, however, that logical effort is but one component of the delay and that often electrical effort and parasitics dominate total delay. Thus, the delay advantage of tree-based C-elements is lower than a simple logical effort analysis would predict. Similarly, wide multiplexers are best divided into trees of 4-input multiplexers to reduce the parasitic delay, despite the increase in logical effort.

11.6 Exercises

11-1 [20] For what values of p and q does Equation 11.6 show substantial differences between AND trees and predecoding? (Use Table 11.1.) Why is there a difference?

11-2 [25] The Lyon-Schediwy decoder is presented in its NOR form. Show the NAND form, and compute its logical effort. Which form has less logical effort?

11-3 [30] Determine the best multiplexer width for values of $H > 1$, as suggested at the end of Section 11.4. Your results will depend on n.

11-4 [25] Make a plot similar to Figure 11.13 if $p_{inv} = 0.5$.

11-5 [25] Wide domino NOR gates are similar to multiplexers in that the logical effort is independent of the number of inputs. Suppose the parasitic delay of an N-input domino NOR is Np_{diff}. Make a plot similar to Figure 11.12 of the best NOR gate width as a function of p_{diff}. Assume the NOR gates have clocked evaluation transistors and a logical effort of 2/3.

Chapter 12

Conclusions

12.1 The theory of logical effort

The theory of logical effort seeks to answer ubiquitous questions of circuit designers. What is the fastest way to compute my logic function? How many stages of logic should I use? How should I size each gate? What circuit family and topology should I select?

Many designers know Mead and Conway's [6] result that strings of inverters with no parasitics have minimum delay with a uniform step-up ratio of e. How does this apply to more complex logic functions? What happens when realistic parasitics are considered?

The theory of logical effort stems from a simple model that the delay of a gate has two parts: an intrinsic delay driving internal parasitics, and an effort delay driving a capacitive load. The effort depends on the ratio of the load size to gate size and also on the complexity of the gate. We call the first term *electrical effort*, defined as:

$$h = C_{out}/C_{in} \tag{12.1}$$

We characterize the complexity of the gate by a number called *logical effort*. Logical effort, g, is the ratio of the input capacitance of a gate to the input capacitance of an inverter that can produce equal current; in other words, it describes how much bigger than an inverter a gate must be to drive loads as well as the inverter can. By this definition, an inverter has logical effort of 1.

Gate type	Static CMOS	High Skew	Low Skew	Dynamic	Pseudo-NMOS
inverter	1	5/6	2/3	2/3	8/9
2-NAND	4/3	1	1	1	16/9
3-NAND	5/3	7/6	4/3	4/3	8/3
4-NAND	2	4/3	5/3	5/3	32/9
2-NOR	5/3	3/2	1	2/3	8/9
3-NOR	7/3	13/6	4/3	2/3	8/9
4-NOR	3	17/6	5/3	2/3	8/9
n-mux	2	5/3	4/3	1	16/9
2-XOR	4	10/3	8/3	2	32/9

Table 12.1: Typical logical effort per input for gates built with various circuit families, with $\gamma = 2$.

The delay through a single logic gate can now be written as

$$d = gh + p \tag{12.2}$$

The results are in units of τ, the delay of an inverter driving another identical inverter with no parasitics. The first term gh is called f, the stage effort or effort delay.

One can estimate logical effort by sketching gates sized for output drive equal to that of an inverter, or may extracted logical effort from simulated delay vs. fanout curves. The logical effort depends on γ, the ratio of PMOS to NMOS transistor sizes in an inverter. Using $\gamma = 2$ is representative of CMOS processes and is convenient for calculation. Table 12.1 lists the logical efforts of various gates in different circuit styles.

A similar calculation finds the delay through a path. The path's logical effort G, is the product of the logical efforts of gates along the path. The path's electrical effort, H, is the path's load capacitance divided by its input capacitance. The path's branching effort, B, accounts for internal fanout. The product of these three terms is the path effort, F, which must be the product of the stage efforts of each stage. Finally, the path's intrinsic delay, P, is the sum of intrinsic delays of gates along the path. We found that delay of a particular path is minimized when the stage efforts are equal:

$$\hat{f} = g_i h_i = F^{1/N} \tag{12.3}$$

We now know how to compute the sizes of gates along a given path to minimize delay, taking into account the varying complexity of gates. But how did we

know the path itself was a good design? A good path uses the right number of stages and selects gates for each stage with low logical effort and parasitic delay. The path effort and the best stage effort set the best number of stages. For gates with no parasitics, the best stage effort is $e = 2.718\ldots$. For gates with realistic parasitics, the best stage effort is larger because it is better to use fewer stages and reduce parasitic delay of paths. Stage effort of 4 is an excellent choice over a range of assumptions. The designer has significant freedom to deviate from this best stage effort, however. Stage efforts from 2.4 to 6 give delays within 15% of minimum. The best number of stages is thus about:

$$N \approx \log_4 F \tag{12.4}$$

The designer should not only select a reasonable number of stages, but should also employ gates with low logical efforts. For example, NAND gates are better than NOR gates in static CMOS . Multiple stages of low fan-in gates have lower logical effort than a single gate with many inputs. Indeed, considering parasitics and logical effort, fast gates generally have no more than 4 series transistors. Path design may involve iteration because the path's logical effort is not known until the topology is chosen, but the right number of stages cannot be known accurately without knowing the logical effort.

Logical effort also explains and quantifies the benefits of various circuit families. For example, domino circuits are faster than static because they have lower logical effort. Pseudo-NMOS wide NOR structures are also fast because of low logical effort. When static CMOS is insufficient to meet delay requirements, consider other circuit families.

12.2 Insights from logical effort

The theory of logical effort is most valuable for the insights it lends into several aspects of circuit design. While the same results might emerge from long design experience or from many circuit simulations, they emerge quite readily from logical effort. We list the following among the interesting results:

1. The idea of a numeric "logical effort" that characterizes the delay characteristics of a logic gate or a path through a network is very powerful. It allows us to compare alternative circuit topologies and to show that some topologies are uniformly better than others.

2. Circuits are fastest when the effort delay of each stage is the same. Moreover, one should select the number of stages to make this effort about 4. CAD tools can automatically check a design and flag nodes with poorly chosen efforts.

3. Fortunately, path delay is very insensitive to modest deviations from the optimum. Therefore, the designer has freedom to adjust the number of stages. Sizing calculations can be done "on the back of an envelope" or in the designer's head to one or two significant figures. The final results will be very close to minimum delay for the topology, so there is little benefit to tweaking transistor sizes in a circuit simulator.

4. The delay of a well-designed path is about $4(\log_4 G + \log_4 H) + P$. Each quadrupling of the load driven by the path adds about the delay of a fanout-of-4 (FO4) inverter. Control signals that must drive a 64-bit datapath therefore incur an amplification delay of about 3 FO4 inverters.

5. The logical effort of each input of a gate increases through no fault of its own as the number of inputs grows. This vividly illustrates the cost of gates with large fan-in. Logical effort can be used to compare designs that are bushy and shallow with those that are narrow and deep.

6. Considering both logical effort and parasitic capacitance, we find a practical limit of about four series transistors in logic gates and four inputs to multiplexers. Beyond this width, it is best to split gates into multiple stages of skinnier gates.

7. Circuits that branch should generally have path lengths differing by no more than 1 gate between the branches. Input capacitance is divided among the legs in proportion to the effort of each leg. It is much better to use 1-2 forks or 2-3 forks than 0-1 forks because the capacitance can be balanced between the legs.

8. The average delay of a gate is minimized by choosing a P/N ratio equal to the square root of the ratio which gives equal rising and falling delays. This also improves the area and power consumption of the gate. Other ratios in the vicinity of this value give excellent results, so a P/N ratio of 1.5 works well for virtually all processes.

9. Logical effort quantifies the benefits of different circuit families. It shows that pseudo-NMOS is good for wide NOR structures. Johnson's symmetric NOR gate is even better. Domino logic is faster than static logic because it uses dynamic gates with low logical effort and also uses high-skew static gates that favor the critical transition. The best stage effort for domino logic is about 2-3 because domino buffers provide more amplification than static inverters.

It may be that logical effort is a useful measure of computational complexity. What is the minimum logical effort required to add two N-bit numbers? To multiply them? A model of the cost of computation based on logical effort far more accurately portrays the time and space required to complete a calculation than does a simple count of logic gates, perhaps with restricted fan-in. Extending complexity results to logical effort might lend new insights into the limits of computation. The point is not that one should become preoccupied with reducing logical effort, but rather that logical effort is a uniform basis on which to assess the performance impact of different circuit choices.

12.3 A design procedure

We can apply the method of logical effort with a simple design procedure shown in Figure 12.1. When there is little branching, the path effort is easy to compute and no iteration is necessary. When there is complex branching and wire capacitance, the procedure helps refine an initial guess at path effort into a good design with a few iterations.

The design procedure must begin with a *block specification* describing the function of the path, the input and output capacitances, and the maximum tolerable delay. A common mistake of beginning circuit designers is to specify only the function and the output capacitance. Unless a specification includes an input capacitance limit, the block can be made arbitrarily fast by increasing the sizes of gates. This inevitably causes a previous block to slow down. Similarly, if no delay specification is given, the designer has no way of knowing when the design is "good enough." While feasibility studies may explore the fastest possible implementations, real designs waste area, power, and design time if they are made faster than necessary.

Given the block spec, the designer can select a topology. Critical paths may use domino circuits or other special families for extra speed, but when in doubt

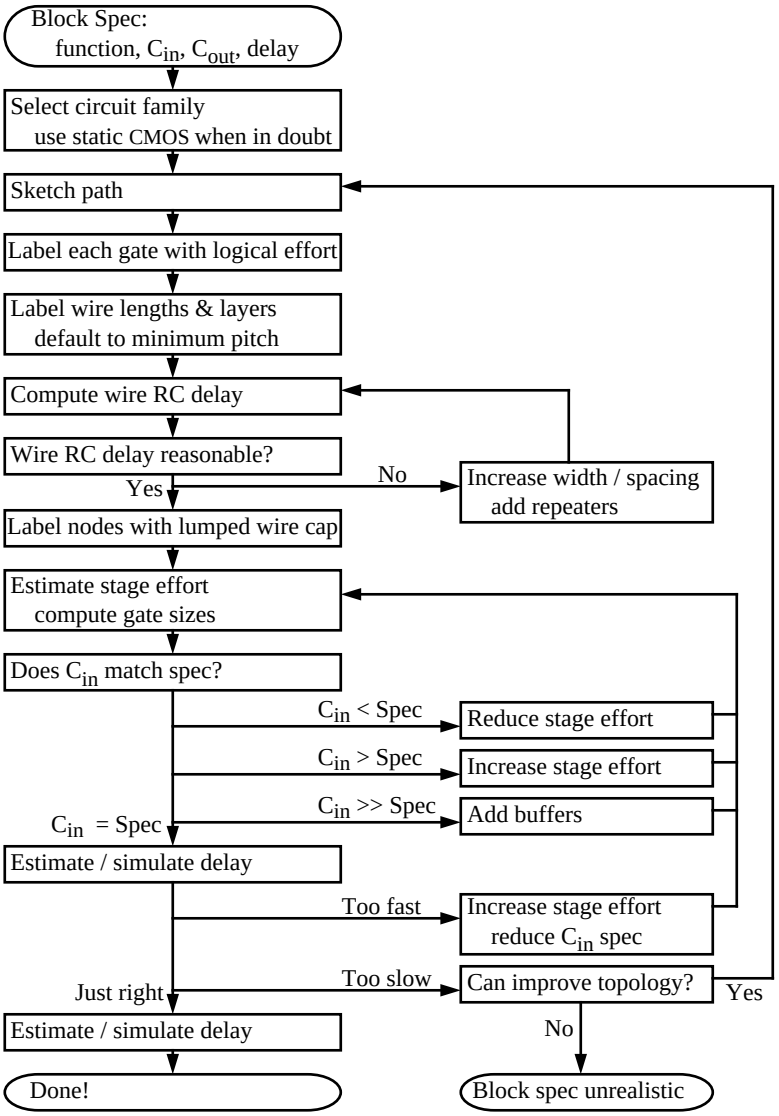


Figure 12.1: A flowchart for path design.

it is best to start with static CMOS. The best number of stages emerges from preliminary logical effort calculations, but may be revised later. Label each gate with its logical effort.

Next, the designer should consider interconnect because the flight time across long wires is independent of the driver size. Each wire should be labeled with its length, metal layer (e.g. metal4), and width and spacing. Wires can default to minimum width and spacing unless they prove to be critical. From these parameters, the designer can compute the wire resistance R, capacitance C, and distributed delay, $RC/2$. If this delay is small enough, perhaps less than a gate delay, the wire is acceptable. If the delay is too large, the designer may increase the width or spacing or insert repeaters. Once the wire design is complete, the wire can be treated as a lumped capacitance for logical effort purposes.

It is now time to pick sizes for the gates. The designer should estimate the path effort and thus compute the stage effort. The stage effort should be about 4 for static logic and about 2.75 for domino logic; if it is far off, stages should be added or combined. If the path is simple branching, the path effort is easy to determine. If the path has complex branches and medium-length wires, the estimate may be inaccurate, but will be corrected later. The designer starts at the end and works backward, applying a capacitance transformation to compute the size of each gate. Practical constraints sometimes restrict the choice of sizes. For instance, transistors have a minimum allowable size or a library may limit choice of gate sizes. Sometimes a large driver should be undersized to save area and power.

After assigning gate sizes, compare the actual input capacitance to the specification. If the input capacitance is smaller than specification, the stage effort was larger than necessary and can be reduced. If the input capacitance exceeds the specification, the stage effort was smaller than necessary and must be increased. If the input capacitance greatly exceeds the specification, the design probably has too few stages and buffers should be added to the end of the path.

Once the input capacitance meets specification, compute the delay of the circuit by simulation, static timing analysis, or hand estimation. If you are fortunate, the design is faster than necessary. Increase the stage effort and reduce the input capacitance to reduce the area of the design and present a smaller load to the previous path.

Murphy's Law dictates that the design is usually too slow. A common mistake among beginners is to tweak the sizes of transistors in the path in the hope of improving speed. This is doomed to failure if logical effort was correctly applied because the gate sizes are already right for theoretically minimum delay, as accu-

rately as the model allows. Often the problem is "solved" by upsizing all of the devices in the path, but such a solution violates the input capacitance specification and pushes the problem to the previous path. Moreover, it leads to designs with overly large gates. A better approach is to rethink the overall topology. Perhaps it is possible to use faster circuit families or rearrange gates to favor late inputs. If a better topology is found, repeat the sizing process. If the topology and sizes are thoroughly optimized, the block spec is infeasible and the specification must be modified. Sometimes logical effort allows one to reject a specification as unrealistic with very little design work.

With practice, this design procedure is easy to use and works for a wide range of circuit problems. It is intended only as a general guide; the designer should also trust his or her own intuition and special knowledge of the problem.

12.4 Other approaches to path design

12.4.1 Simulate and tweak

Junior circuit designers with no instruction on sizing techniques tend to use the *simulate and tweak* method. This method begins with a randomly selected topology that implements the logic function. The engineer simulates the circuit and finds that it is too slow. Therefore, he or she tries increasing the size of gates. This only pushes the problem from one gate to another. After extensive simulation, the engineer concludes the topology is too slow and tries compressing the function into fewer stages with the hope of reducing the number of gate delays by reducing the number of gates. If the stage effort was too large in the first topology, this only makes delay worse. After sufficient experience, the engineer begins to develop heuristics for path selection and design. Nevertheless, the design method involves tedious and time-consuming simulation and spoils the joy of circuit design.

Fortunate engineers either get instruction from veteran designers or realize themselves that simulate and tweak is a bad method and therefore derive better techniques.

12.4.2 Equal fanout

One better technique is to use equal fanout per stage and to target a fanout of about 4. This method is an intuitive extension of the result that inverter chains with fanouts of 4 are fastest. The term *fanout* is used in place of *electrical effort*.

Equal fanout design is sufficient for circuits like decoders in which the logical effort tends to be low, but is suboptimal for paths with large logical effort because it results in stage efforts well above 4.

12.4.3 Equal delay

Another improved design technique is *delay allocation* or equal delay per stage. This is also provably non-optimal because it equalizes the sum of the effort and parasitic delays rather than just the effort delays. Nevertheless, because path delay is a weak function of exact sizes, equal delay sizing tends to give good results unless parasitic delays are very large. An optimal delay per stage can be found for a particular process. Given a delay specification, this dictates the number of stages that should be used. Equal delay sizing has many practical advantages. Designers usually are given specifications in picoseconds, which directly relate to the number of stages they should employ. CAD tools also report delays in picoseconds, so circuits can be optimized by adjusting sizes until the delays are equal. This is easier than adjusting sizes until efforts are equal because efforts cannot be determined directly from simulation or static timing analysis.

Besides non-optimal results, equal delay sizing has other theoretical drawbacks. It gives less insight about circuits and about the cost of fanout. It also produces process-dependent delay results, expressed in picoseconds. Therefore, intuition developed in one process is more difficult to scale to the next generation of process.

Both equal delay sizing and logical effort have a place in the designer's toolbox. Logical effort is most useful for reasoning about circuits and doing simple calculations to determine topology. Equal delay sizing is most convenient when finding a low cost circuit to meet a delay specification and when tuning a circuit based on simulation or static timing analysis.

12.4.4 Numerical optimization

There are many tools available that harness the speed of computers to optimize circuit sizes numerically. Visweswariah [8] surveys the principles and challenges of numerical circuit optimization. Because these tools can obtain optimal results and can use more accurate models than the simple RC delay model, why are manual techniques relevant?

The greatest value of logical effort is the insight it provides. While a numerical optimizer can tweak a given path for maximum speed, it does not explain why

the path is fast nor whether the topology was a good one in the first place. Moreover, numerical methods are prone to get stuck in local optima and are unlikely to produce meaningful results unless the user knows approximately what results should be expected. Synthesis tools make some effort to explore topologies, but still cannot match experienced designers on critical paths. Moreover, designers have in their heads many constraints on the design, such as performance, floorplan, wiring, and interfaces with other circuits. Merely specifying all of these constraints to an optimization tool may take longer than selecting reasonable sizes by hand. Finally, accurate circuit optimization is fundamentally a nonlinear problem which tends to have runtime and convergence problems when applied to real designs.

12.5 Shortcomings of logical effort

Logical effort is based on a very simple premise: equalize the effort delay of each stage. The simplicity is the method's greatest strength, but it also results in a number of limitations:

- The RC delay model is overly simplistic. In particular, it fails to capture the effects of velocity saturation and of variable rise times [3]. Fortunately, rise times tend to be about equal in well-designed circuits with equal effort delays and velocity saturation can be handled by characterizing logical effort of gates with simulation.

- Logical effort explains how to design a path for maximum speed, but does not easily show how to design a path for minimum area or power under a fixed delay constraint.

- Logical effort does not provide simple closed-form solutions to paths that branch and have a different number of stages or different parasitic delays on each branch. Usually iteration is required to tune such circuits. Iteration is also required when fixed wire capacitances are comparable to gate capacitance.

- Many real circuits are too complex to optimize by hand. For example, problems in Chapter 11 were solved with spreadsheets or with simple scripts. Given that numerical optimization is sometimes necessary, perhaps the optimizer should use a more accurate delay equation.

12.6 Parting words

One of the joys of circuit design is the challenge of designing ever more powerful chips. High speed processors push circuit design to the limit. Low power circuits are also created from high speed designs operating at the lowest feasible voltage. Every circuit designer constantly confronts the question of how to choose fast circuit topologies and how to size the paths for greatest speed. As a result, every good designer has developed a set of heuristics which lead to fast circuits. Logical effort should not displace this insight but should rather supplement it by providing a simple and powerful framework to reason about delay in circuits and a common language for designers to communicate their ideas.

Even more importantly, we hope this monograph will help new circuit designers quickly develop their own intuition. We have experienced the frustration of endless simulation and tweaking of gate sizes before we developed logical effort. Therefore, we hope we have provided a useful teaching tool.

The only way to become skilled with logical effort is to use it. At first, you will find it slow and cumbersome. With practice, however, you will develop proficiency and soon discover logical effort is a very productive way to design fast circuits.

Bibliography

[1] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[2] D. Harris and M. Horowitz. "Skew-tolerant Domino Circuits." *IEEE J. Solid-State Circuits,* 31(11):1687-1696, Nov. 1996.

[3] M. Horowitz. *Timing Models for* MOS *Circuits.* PhD thesis, Stanford University, December 1983. TR SEL83-003.

[4] M. Johnson. "A Symmetric CMOS NOR Gate for High-Speed Applications." *IEEE J. Solid-State Circuits,* 23(5):1233-1236, Oct. 1988.

[5] R.F.Lyon and R.R.Schediwy. "CMOS Static Memory with a New Four-Transistor Memory Cell." *Proc. Stanford Conf. on Advanced Research in VLSI*, March 1987.

[6] C. A. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980, p. 12.

[7] I.E. Sutherland, and R.F. Sproull. "Logical Effort: Designing for Speed on the Back of an Envelope." *Proc. Conf. on Advanced Research in VLSI*, March 1991.

[8] Chandu Visweswariah. "Optimization techniques for high-performance digital circuits" *Proc. IEEE Intl. Conf. Computer Aided Design.*, Nov. 1997.

[9] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*, 2nd edition. Addison-Wesley, 1993, p. 219.

Appendix A

Cast of Characters

The notation used in this monograph obeys certain conventions whenever possible:

- Parameters of the fabrication process and design parameters that are likely to be the same for all logic gates are given by Greek letters.
- Logic gate inputs and outputs are single lower-case letters, in the set a, b, c whenever possible. Subscripts are often used to indicate different stages of logic along a path in a network.
- Quantities used in equations for modeling transistor properties are chosen to match existing conventions.

The principal notational symbols are:

d The delay in a single stage of a logic network, or "stage delay." Often subscripted to identify a single stage of a network.

D The total delay along a path through a logic network.

$\hat{D}$ The total delay along a path through a logic network when the design of the network is optimized to obtain least delay.

g The logical effort per input or bundle of a logic gate. Often subscripted to denote the particular input or bundle and/or to identify a single stage of a

network. (The letter g represents logical effort because it is the first letter in the word "logical" that is not easily confused with other symbols—l with one and o with zero.)

g_{tot} The total logical effort of a single logic gate. Often subscripted to identify a single stage of a network.

G The path logical effort borne by one or more paths through a logic network. When subscripted with characters, denotes the logical effort between two points in a network: G_{ab} is the logical effort along the path from a to b.

h The electrical effort borne by a single stage: $h = C_{out}/C_{in}$. This is the ratio of a logic gate's load capacitance to the input capacitance of a single input. Often subscripted to identify a single stage of a network. (The letter h is chosen so that the formula $f = gh$ reads in alphabetical order.)

H The path electrical effort borne by one or more paths through a logic network. When subscripted with characters, denotes the electrical effort between two points in a network: H_{ab} is the electrical effort along the path from a to b.

b The branching effort borne at the output of a single logic gate. Often subscripted to identify a stage of a network.

B The path branching effort borne by one or more paths through a network. Note that branching effort at the last stage in a network is not counted, since the electrical effort reflects the effort of branching in the last stage.

f The effort, electrical and logical, borne by a single stage: $f = gh$. Often subscripted to identify a single stage of a network. Sometimes called the *effort delay*, because it is the contribution to delay in a single logic gate that is induced by the effort the gate bears. (The letter f was chosen to represent the word "effort;" the letter e being too easily confused with the constant 2.718.)

$\hat{f}$ The optimum value of f to minimize delay along a path with a given number of stages.

ρ The optimum value of $\hat{f}$ when the number of stages in a path is chosen to minimize delay.

F The path effort, electrical, branching, and logical, borne by one or more paths through a logic network: $F = GBH$. When subscripted with characters, denotes the path effort between two points in a network: F_{ab} is the path effort along the path from a to b.

p The parasitic delay of a logic gate.

p_{inv} The parasitic delay of an inverter.

P The parasitic delay along a path of a network.

N The number of stages along a logic path.

R Resistance. R_w is the resistance per unit length of a metal wire.

C Capacitance. C_{in} is the input capacitance of a logic gate or of a path through a logic network. C_{out} is the load capacitance of a logic gate or of a path through a logic network. C_w is the capacitance per unit length of a metal wire.

L The length of a transistor. In an inverter, L_n is the length of the n-type transistor and L_p is the length of the p-type transistor.

W The width of a transistor. In an inverter, W_n is the width of the n-type transistor and W_p is the width of the p-type transistor.

τ The delay of an ideal inverter with no stray capacitance driving an identical inverter. If rising and falling delays differ, then τ_u is the rising delay, and τ_d is the falling delay.

γ The ratio of the shape factor of p-type pullup transistors to that of n-type pulldown transistors in an inverter: $\gamma = (W_p/L_p)/(W_n/L_n)$. Usually $\gamma > 1$.

P/N **ratio** The ratio of the shape factor of p-type pullup transistors to that of n-type pulldown transistors in an arbitrary logic gate. For inverters, the P/N ratio equals γ. For a 2-input NOR gate, the P/N ratio must be 2γ to have rising and falling delays proportional to those of an inverter.

μ_n The mobility in n-channel devices.

μ_p The mobility in p-channel devices.

μ The ratio of n-channel mobility to p-channel mobility: $\mu = \mu_n/\mu_p$. Usually $\mu > 1$. *Warning:* Whereas μ is the ratio of n to p mobilities, γ is the ratio of p to n shape factors.

The adjectives "stage" and "path" are applied to logical effort, electrical effort, effort, effort delay, and parasitic delay. The adjective "total" is applied to "logical effort" only, and means the sum of the logical effort per input of all inputs of a logic gate.

Appendix B

Logical Effort Tools

The Logical Effort web page offers several tools to assist with logical effort. The page can be found at:

`http://velox.stanford.edu/TBD`

Documentation for the tools is also online.

B.1 Library characterization

The Perl script used in Chapter 5 to characterize the logical effort of gates is online. The script takes a SPICE netlist of the gates, a process file, and a list of input stimulus for each gate. It measures the logical effort and parasitic delay of each gate using the test setup described in Chapter 5.

B.2 Wide gate design

A Perl script is available to design wide NAND, NOR, AND, and OR gates. It takes the number of inputs and the electrical effort of the path and computes the minimum-delay tree, as discussed in Section 11.1. The tool can be used from a form-based interface on the web, or downloaded for use on your computer.

Index