Chapter 4

CMOS Logic Circuits

4.1 Switch Logic

In digital systems we use binary variables to represent information and switches controlled by these variables to process information. Figure 4.1 shows a simple switch circuit. When binary variable a is false (0), (Figure 4.1(a)), the switch is open and the light is off. When a is true (1), the switch is closed, current flows in the circuit, and the light is on.

We can do simple logic with networks of switches as illustrated in Figure 4.2. Here we omit the voltage source and light bulb for clarity, but we still think of the switching network as being *true* when its two terminals are connected - i.e., so the light bulb, if connected, would be on.

Suppose we want to build a switch network that will launch a missile only if two switches (activated by responsible individuals) are closed. We can do this as illustrated in Figure 4.2(a) by placing two switches in series controlled by logic variables a and b respectively. For clarity we usually omit the switch symbols and denote a switch as a break in the wire labeled by the variable controlling

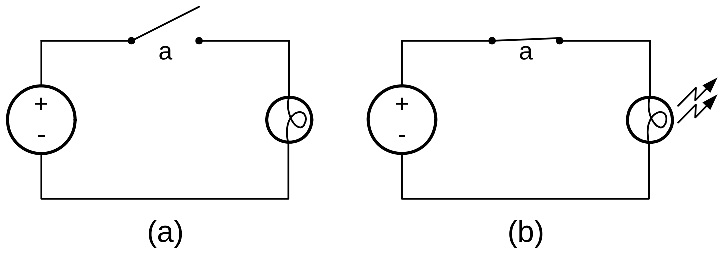


Figure 4.1: A logic variable a controls a switch that connects a voltage source to a light bulb. (a) When $a = 0$ the switch is open and the bulb is off. (b) When $a = 1$ the switch is closed and the bulb is on.

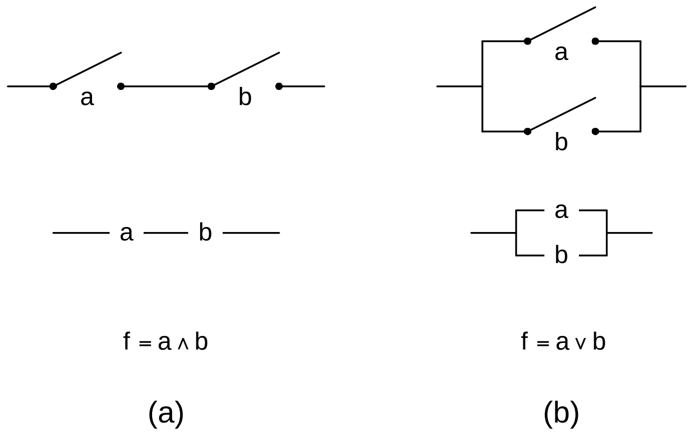


Figure 4.2: AND and OR switch circuits. (a) Putting two switches in series, the circuit is closed only if both logic variable a and logic variable b are true ($a \wedge b$). (b) Putting two switches in parallel, the circuit is closed if either logic variable is true ($a \vee b$). (bottom) For clarity we often omit the switch symbols and just show the logic variables.

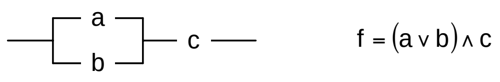


Figure 4.3: An OR-AND switch network that realizes the function $(a \vee b) \wedge c$.

the switch as shown at the bottom of the figure. Only when both a and b are true are the two terminals connected. Thus, we are assured that the missle will only be launched if both a and b agree that it should be launched. Either a or b can stop the launch by not closing its switch. The logic function realized by this switch network is $f = a \wedge b$.[1]

When launching missiles we want to make sure that everyone agrees to launch before going forward. Hence we use an AND function. When stopping a train, on the other hand, we would like to apply the brakes if anyone sees a problem. In that case, we use an OR function as shown in Figure 4.2(b) placing two switches in parallel controlled by binary variables a and b respectively. In this case, the two terminals of the switch network are connected if either a, or b, or both a and b are true. The function realized by the network is $f = a \vee b$.

We can combine series and parallel networks to realize arbitrary logic functions. For example, the network of Figure 4.3 realizes the function $f = (a \vee b) \wedge c$.

[1] Recall from Chapter 3that $\wedge$ denotes the logical AND of two variables and $\vee$ denotes the logical OR of two variables.

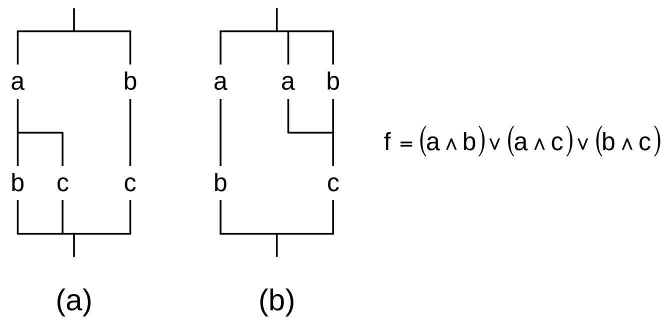


Figure 4.4: Two realizations of a 3-input *majority* function (or 2 out of 3 function) which is true when at least 2 of its 3 inputs is true.

To connect the two terminals of the network c must be true, and either a or b must be true. For example, you might use a circuit like this to engage the starter on a car if the key is turned c and either the clutch is depressed a or the transmission is in neutral b.

More than one switch network can realize the same logical function. For example, Figure 4.4 shows two different networks that both realize the three-input *majority* function. A majority function returns true if the majority of its inputs are true; in the case of a three-input function, if at least two inputs are true. The logic function realized by both of these networks is $f = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$.

There are several ways to analyze a switch network to determine the function it implements. One can enumerate all 2^n combinations of the n inputs to determine the combinations for which the network is connected. Alternatively, one can trace all paths between the two terminals to determine the sets of variables, that if true, make the function true. For a series-parallel network, one can also reduce the network one-step at a time by replacing a series or parallel combination of switches with a single switch controlled by an AND or OR of the previous switches expressions.

Figure 4.5 shows how the network of Figure 4.4(a) is analyzed by replacement. The original network is shown in Figure 4.5(a). We first combine the parallel branches labeled b and c into a single switch labeled $b \vee c$ (Figure 4.5(b)). The series combination of b and c is then replaced by $b \wedge c$ (Figure 4.5(c)). In Figure 4.5(d) the switches labeled a and $b \vee c$ are replaced by $a \wedge (b \vee c)$. The two parallel branches are then combined into $[a \wedge (b \vee c)] \vee (b \wedge c)$ (Figure 4.5(e)). If we distribute the AND of a over $(b \vee c)$ we get the final expression in Figure 4.5(f).

So far we have used only positive switches in our network - that is switches that are closed when their associated logic variable or expression is true (1). The set of logic functions we can implement with only positive switches is very limited

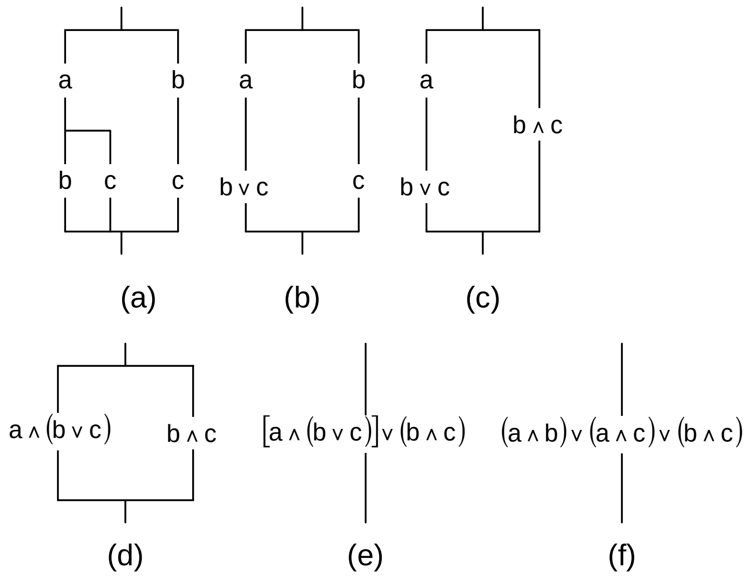


Figure 4.5: We can analyze any series parallel switch network by repeatedly replacing a series or parallel subnetwork by a single switch controlled by the equivalent logic equation.

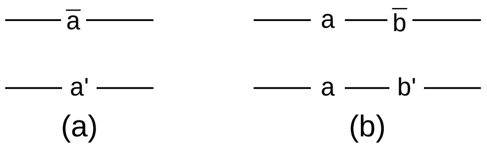


Figure 4.6: A negated logic variable is denoted by a prime a' or an overbar $\bar{a}$. (a) This switch network is closed (true) when variable $a = 0$. (b) A switch network that realized the function $a \wedge \bar{b}$.

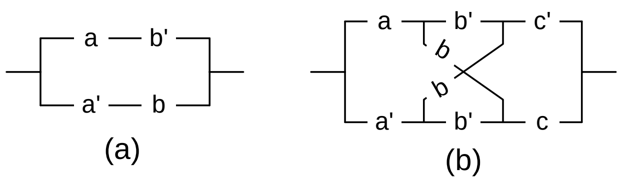


Figure 4.7: Exclusive-or (XOR) switch networks are true (closed) when an odd number of their inputs are true. (a) A two input XOR network. (b) A three-input XOR network.

(monotonically increasing functions). To allow us to implement all possible functions we need to introduce negative switches - switches that are closed when their controlling logic variable is false (0). As shown in Figure 4.6(a)we denote a negative switch by labeling its controlling variable with either a prime a' or an overbar $\bar{a}$. Both of these indicate that the switch is closed when a is false (0). We can build logic networks that combine positive and negative switches. For example, Figure 4.6(b) shows a network that realizes the function $f = a \wedge \bar{b}$.

Often we will control both positive and negative switches with the same logic variable. For example, Figure 4.7(a) shows a switch network that realizes the two-input exclusive-or (XOR) function. The upper branch of the circuit is connected if a is true and b is false while the lower branch is connected if a is false and b is true. Thus this network is connected (true) if exactly one of a or b is true. It is open (false) if both a and b are true.

This circuit should be familar to anyone who has ever used a light in a hallway or stairway controlled by two switches: one at either end of the hall or stairs Changing the state of either switch changes the state of the light. Each switch is actually two switches – one positive and one negative – controlled by the same variable: the position of the switch control. [2]. They are wired exactly as shown in the figure – with switches $a, \bar{a}$ at one end of the hall, and $b, \bar{b}$ at the other end.

In a long hallway, we sometimes would like to be able to control the light from the middle of the hall as well as from the ends. This can be accomplished with the three-input XOR network shown in Figure 4.7(b). An n-input XOR function is true is an odd number of the inputs are true. This three-input XOR network is connected if exactly one of the inputs a, b or c is true or if all three of them are true. To see this is so, you can enumerate all eight combinations of a, b, and c or you can trace paths. You cannot, however, analyze this network by replacement as with Figure 4.5 because it is not a series-parallel network. If you want to have more fun analyzing non-series-parallel networks, see Exercises 4–3 and 4–4.

In the hallway application, the switches associated with a and c are placed at either end of the hallway and the switches associated with b are placed in

[2]Electricians call these three-terminal, two switch units *three-way switches*.

the center of the hall. As you have probably observed, if we want to add more switches controlling the same light, we can repeat the four-switch pattern of the *b* switches as many times as necessary, each time controlled by a different variable[3].

4.2 A Switch Model of MOS Transistors

Most modern digital systems are built using CMOS (Complementary Metal Oxide Semiconductor) field-effect transistors as switches. Figure 4.8 shows the physical structure and schematic symbol for a MOS transistor. A MOS transistor is formed on a semiconductor substrate and has three terminals[4]: the gate, source, and drain. The source and drain are identical terminals formed by diffusing an impurity into the substrate. The gate terminal is formed from polycrystalline silicon (called *polysilicon* or just *poly* for short) and is insulated from the substrate by a thin layer of oxide. The name MOS, a holdover from the days when the gate terminals were metal (aluminum), refers to the layering of the gate (metal), gate oxide (oxide) and substrate (semiconductor).

Figure 4.8(d), a top view of the MOSFET, shows the two dimensions that can be varied by the circuit or logic designer to determine transistor performance[5]: the device *width* W and the device *length* L. The gate length L is the distance that charge carriers (electrons or holes) must travel to get from the source to the drain and thus is directly related to the speed of the device. Gate length is so important that we typically refer to a semiconductor process by its gate length. For example, most new designs today (2003) are implemented in 0.13μm CMOS processes - i.e., CMOS processes with a minimum gate length of $0.13mu$m. Almost all logic circuits use the minimum gate length supported by the process. This gives the fastest devices with the least power dissipation.

The channel width W controls the strength of the device. The wider the device the more charge carriers that can traverse the device in parallel. Thus the larger W the lower the on resistance of the transistor and the higher the current the device can carry. A large W makes the device faster by allowing it to discharge a load capacitance more quickly. Alas, reduced resistance comes at a cost - the gate capacitance of the device also increases with W. Thus as W increases it takes longer to charge or discharge the gate of a device.

Figure 4.8(c) shows the schematic symbols for an n-channel MOSFET (NFET) and a p-channel MOSFET (PFET). In an NFET the source and drain are n-type semiconductor in a p-type substrate and the charge carriers are electrons.

[3]Electricians call this four-terminal, four-switch unit where the connections are straight through when the variable is false (switch handle down) and crossed when the variable is true (switch handle up) a *four-way switch*. To control one light with $n \geq 2$ switches requires two three-way switches and $n-2$ four-way switches. Of course, one can always use a four-way switch as a three-way switch by leaving one terminal unconnected.

[4]The substrate is a fourth terminal that we will ignore at present.

[5]The gate oxide thickness is also a critical dimension, but it is set by the process and cannot be varied by the designer. In contrast, W and L are determined by the mask set and hence can be adjusted by the designer.

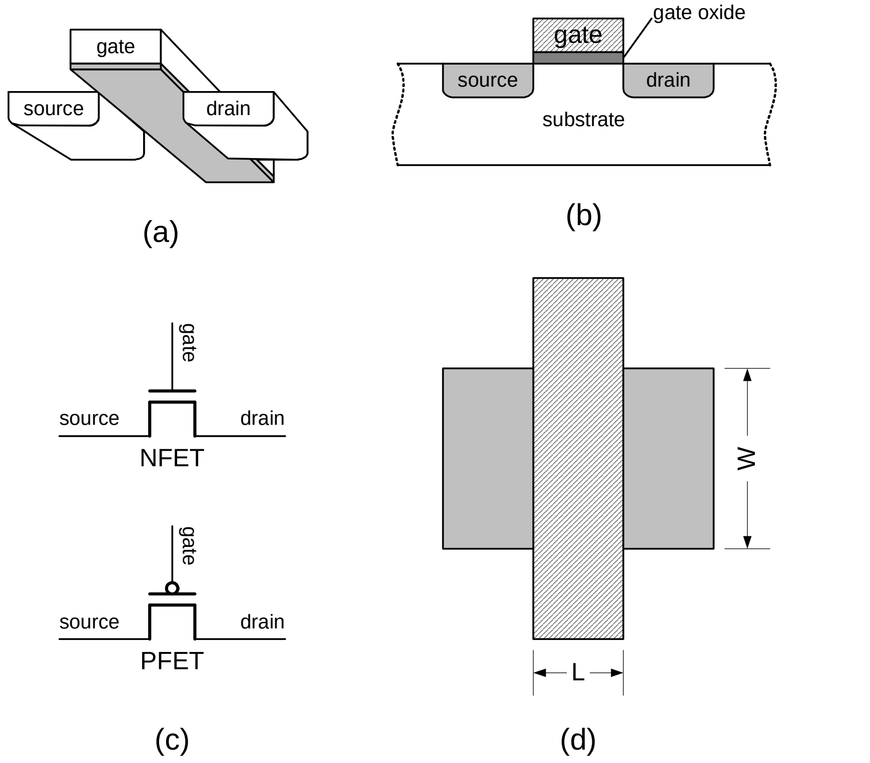


Figure 4.8: A MOS field-effect transistor (FET) has three terminals. Current passes between the source and drain (identical terminals) when the device is on. The voltage on the gate controls whether the device is on or off. (a) The structure of a MOSFET with the substrate removed. (b) A side view of a MOS FET. (c) Schematic symbols for a n-channel FET (NFET) and a p-channel FET (PFET). (d) Top view of a MOSFET showin its width W and length L.

In a PFET the types are reversed - the source and drain are p-type in a n-type substrate (usually a n-well diffused in a p-type substrate) and the carriers are holes.

If you haven't got a clue what n-type and p-type semiconductors, holes, and electrons are, don't worry we will abstract them away shortly. Bear with us for the moment.

Figure 4.9 illustrates a simple digital model of operation for an n-channel FET[6]. As shown in Figure 4.9(a), when the gate of the NFET is a logic 0, the source and drain are isolated from one another by a pair of p-n junctions (back-to-back diodes) and hence no current flows from drain to source, $I_{DS} = 0$. This is reflected in the schematic symbol in the middle panel. We model the NFET in this state with a switch as shown in the bottom panel.

When the gate terminal is a logic 1 *and* the source terminal is a logic zero, as shown in Figure 4.9(b), the NFET is turned on. The positive voltage between the gate and source induces a negative charge in the *channel* beneath the gate. The presence of these negative charge carriers (electrons) makes the channel effectively n-type and forms a conductive region between the source and drain. The voltage between the drain and the source accelerates the carriers in the channel, resulting in a current flow from drain to source, I_{DS}. The middle panel shows the schematic view of the on NFET. The bottom panel shows a switch model of the on NFET. When the gate is 1 and the source is 0, the switch is closed.

It is important to note that if the source[7] is 1, the switch will *not* be closed even if the gate is 1 because there is no net voltage between the gate and source to induce the channel charge. The switch is not open in this state either - because it will turn on if either terminal drops a threshold voltage below the 1 voltage. With source = 1 and gate = 1, the NFET is in an undefined state (from a digital perspective). The net result is that an NFET can reliably pass only a logic 0 signal. To pass a logic 1 requires a PFET

Operation of a PFET, illustrated in Figure 4.10 is identical to the NFET with the 1s and 0s reversed. When the gate is 0 and the source is 1 the device is on. When the gate is 1 the device is off. When the gate is 0 and the source is 0 the device is in an undefined state. Because the source must be 1 for the device to be reliably on, the PFET can reliably pass only a logic 1. This nicely complements the NFET which can only pass a 0.

The NFET and PFET models of Figures 4.9 and 4.10 accurately model the function of most digital logic circuits. However to model the delay and power of logic circuits we must complicate our model slightly by adding a resistance in series with the source and drain and a capacitance from the gate to ground as shown in Figure 4.11[8]. The capacitance on the gate node is proportional to

[6]A detailed discussion of MOSFET operation is far beyond the scope of these notes. Consult a textbook on semiconductor devices for more details.

[7]Physically the source and drain are identical and the distinction is a matter of voltage. The source of an NFET (PFET) is the most negative (positive) of the two non-gate terminals.

[8]In reality there is capacitance on the source and drain nodes as well - usually each has a capacitance equal to about half of the gate capacitance (depending on device size and

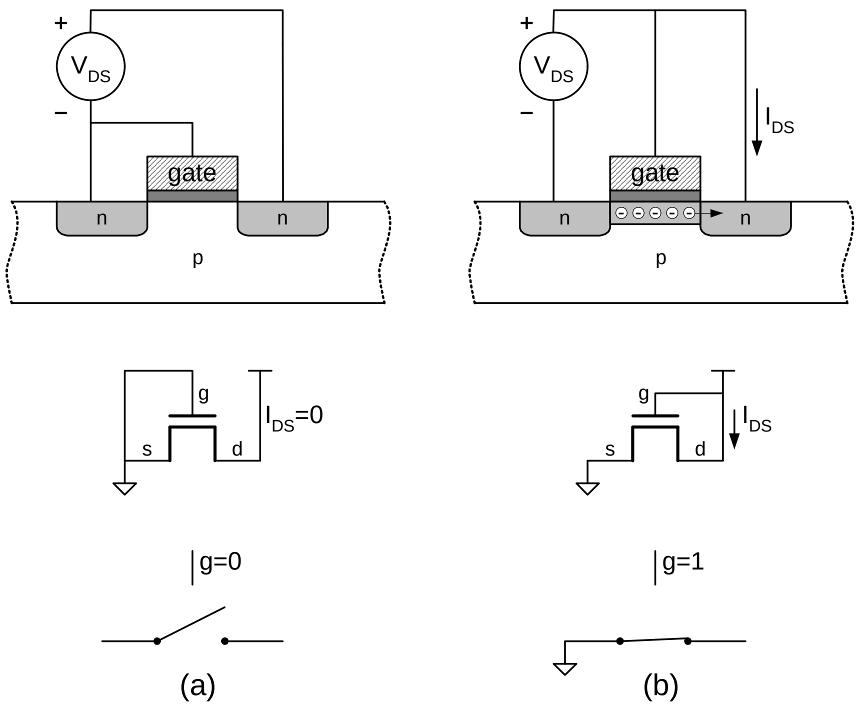


Figure 4.9: Simplified operation of a n-channel MOSFET. (a) When the gate is at the same voltage as the source, no current flows in the device because the drain is isolated by a reverse-biased p-n junction (a diode). (b) When a positive voltage is applied to the gate, it induces negative carriers in the *channel* beneath the gate, effectively inverting the p-type silicon to become n-type silicon. This connects the source and drain allowing a current I_{DS} to flow. The top panel shows what happens physically in the device. The middle panel shows the schematic view. The bottom panel shows a switch model of the device.

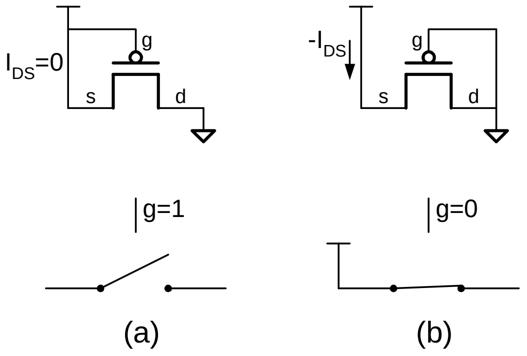


Figure 4.10: A p-channel MOSFET operates identically to an NFET with all 0s and 1s switched. (a) When the gate is high the PFET is off regardless of source and drain voltages. (b) When the gate is low and the source is high the PFET is on and current flows from source to drain.

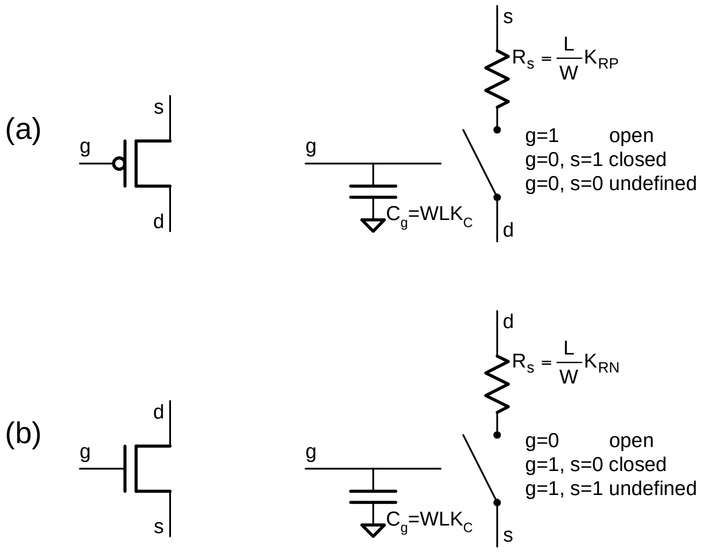


Figure 4.11: A p-channel MOSFET operates identically to an NFET with all 0s and 1s switched. (a) When the gate is high the PFET is off regardless of source and drain voltages. (b) When the gate is low and the source is high the PFET is on and current flows from source to drain.

Parameter	Value	Units
K_C	2×10^{-16}	Farads/L_{min}^2
K_{RN}	2×10^4	Ohms/square
K_P	2.5	
$K_{RP} = K_P K_{RN}$	5×10^4	Ohms/square
$\tau_N = K_C K_{RN}$	4×10^{-12}	seconds
$\tau_P = K_C K_{RP}$	1×10^{-11}	seconds

Table 4.1: Device parameters for a typical 0.13μm CMOS process.

the area of the device, WL. The resistance, on the other hand is proportional to the aspect ratio of the device L/W.

For convenience, and to make our discussion independent of a particular process generation, we will express W and L in units of L_{min} the minimum gate length of a technology. For example, in an 0.13μm technology, we will refer to a device with $L = 0.13\mu$m and $W = 1.04\mu$m device as a $L = 1$, $W = 8$ device, or just as a $W = 8$ device since $L = 1$ is the default. In some cases we will scale W by $W_{\text{min}} = 8L_{\text{min}}$ that is we will refer to a minimum sized $W/L = 8$ device as a unit-sized device and size other device in relation to this device.

Table 4.2 gives typical values of K_C, K_{RN}, and K_{RP} for an 0.13μm technology. The key parameters here are τ_N and τ_P, the basic time constants of the technology. As technology scales, K_C (expressed as Farads/L_{min}^2) remains roughly proportional to gate length and can be approximated as

$$K_C \approx 1.5 \times 10^{-9} L_{\text{min}}. \tag{4.1}$$

Where L_{min} is expressed in m. The resistances remain roughly constant as technology scales causing both time constants to also scale linearly with L_{min}.

$$\tau_N \approx 3 \times 10^{-5} L_{\text{min}}. \tag{4.2}$$

$$\tau_P = K_P \tau_N \approx 7.5 \times 10^{-5} L_{\text{min}}. \tag{4.3}$$

4.3 CMOS Gate Circuits

In Section 4.1 we learned how to do logic with switches and in Section 4.2 we saw that MOS transistors can, for most digital purposes, be modeled as switches. Putting this information together we can see how to make logic circuits with transistors.

geometry). For the purposes of these notes, however, we'll lump all of the capacitance on the gate node.

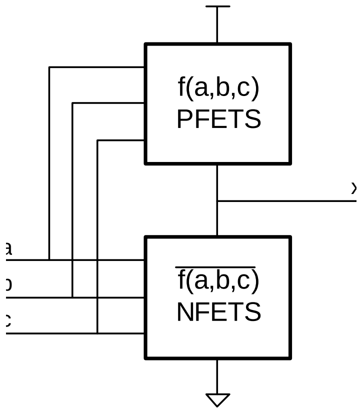


Figure 4.12: A CMOS gate circuit consists of a PFET switch network that pulls the output high when function f is true and an NFET switch network that pulls the output low when f is false.

A well-formed logic circuit should support the digital abstraction by generating an output that can be applied to the input of another, similar logic circuit. Thus, we need a circuit that generates a voltage on its output — not just connects two terminals together. The circuit must also must be restoring, so that degraded input levels will result in restored output levels. To achieve this, the voltage on the output must be derived from a supply voltage, not from one of the inputs.

A *static CMOS gate* circuit realizes a logic function f while generating a restoring output that is compatible with its input as shown in Figure 4.12. When function f is true, a PFET switch network connects output terminal x to the positive supply (V_{DD}). When function f is false, output x is connected to the negative supply by an NFET switch network. This obeys our constraints of passing only logic 1 (high) signals through PFET switch networks and logic 0 (low) signals through NFET networks. It is important that the functions realized by the PFET network and the NFET network be complements. If the functions should overlap (both be true at the same time), a short circuit across the power supply would result drawing a large amount of current and possibly causing permanent damage to the circuit. If the two functions don't cover all input states (there are some input states where neither is true), then the output is undefined in these states.[9]

Because NFETs turn on with a high input and generate a low output and PFETs are the opposite, we can only generate *inverting* (sometimes called monotonically decreasing) logic functions with static CMOS gates. A positive (neg-

[9]We will see in Chapter 23 how CMOS circuits with unconnected outputs can be used for storage.

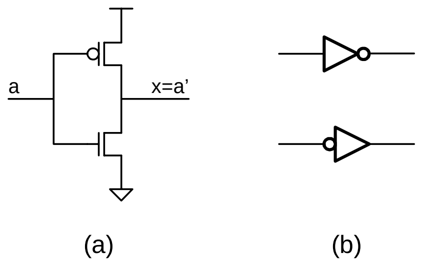


Figure 4.13: A CMOS inverter circuit. (a) A PFET connects x to 1 when $a = 0$ and an NFET connects x to 0 when $a = 1$. (b) Logic symbols for an inverter. The bubble on the input or output denotes the NOT operation.

ative) transition on the input of a single CMOS gate circuit can either cause a negative (positive) transition on the output or no change at all. Such a logic function where transitions in one direction on the inputs cause transitions in just a single direction on the output is called a *monotonic* logic function. If the transitions on the outputs are in the opposite direction to the transitions on the inputs its a monotonic decreasing or inverting logic function. If the transitions are in the same direction, its a monotonic increasing function. To realize a non-inverting or non-monotonic logic function requires multiple stages of CMOS gates.

We can use the principle of duality, Equation (3.9), to simplify the design of gate circuits. If we have an NFET pulldown network that realizes a function $f_n(x_1, \ldots, x_n)$, we know that our gate will realize function $f_p = \overline{f_n(x_1, \ldots, x_n)}$. By duality we know that $f_p = \overline{f_n(x_1, \ldots, x_n)} = f_n^D(\overline{x_1}, \ldots, \overline{x_n})$. So for the PFET pullup network, we want the dual function with inverted inputs. The PFETs give us the inverted inputs, since they are "on" when the input is low. To get the dual function, we take the pulldown network and replace ANDs with ORs and vice-versa. In a switch network, this means that a series connection in the pulldown network becomes a parallel connection in the pullup network and vice-versa.

The simplest CMOS gate circuit is the inverter, shown in Figure 4.13(a). Here the PFET network is a single transistor that connects output x to the positive supply whenever input a is low — $x = \overline{a}$. Similarly the NFET network is a single transistor that pulls output x low whenever the input is high.

Figure 4.13(b) shows the schematic symbols for an inverter. The symbol is a rightward facing triangle with a *bubble* on its input or output. The triangle represents an amplifier — indicating that the signal is restored. The bubble (sometimes called an *inversion bubble*) implies negation. The bubble on the input is considered to apply a NOT operation to the signal before it is input to the amplifer. Similarly a bubble on the output is considered to apply a NOT operation to the output signal after it is amplified. Logically, the two symbols

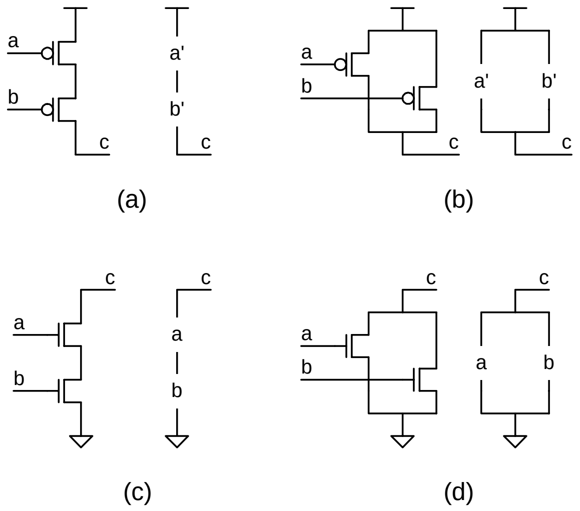


Figure 4.14: Switch networks used to realize NAND and NOR gates. (a) Series PFETs connect the output c high when all inputs are low, $f = \overline{a} \wedge \overline{b} = \overline{a \vee b}$. (b) Parallel PFETs connect the output if either input is low, $f = \overline{a} \vee \overline{b} = \overline{a \wedge b}$. (c) Series NFETs pull the output low when both inputs are high, $f = \overline{a \vee b}$. (d) Parallel NFETs pull the output low when either input is true, $f = \overline{a \wedge b}$.

are equivalent. It doesn't matter if we consider the signal to be inverted before or after amplification. We choose one of the two symbols to obey the *bubble rule* which states that:

> **Bubble Rule:** Where possible signals that are output from a gate with an inversion bubble on its output shall be input to a gate with an inversion bubble on its input.

Schematics drawn using the bubble rule are easier to read than schematics where the polarity of logic signals changes from one end of the wire to the other. We shall see many examples of this in Chapter 6.

Figure 4.14 shows some example NFET and PFET switch networks that can be used to build NAND and NOR gate circuits. A parallel combination of PFETs (Figure 4.14(b)) connects the output high if either input is low, so $f = \overline{a} \vee \overline{b} = \overline{a \wedge b}$. Applying our principle of duality, this switch network is used in combination with a series NFET network (Figure 4.14(c)) to realize a NAND gate. The complete NAND gate circuit is shown in Figure 4.15(a) and two schematic symbols for the NAND are shown in Figure 4.15(b). The upper symbol is an AND symbol (square left side, half-circle right side) with an inversion bubble on the output — indicating that we AND inputs a and b and then invert the output, $f = \overline{a \wedge b}$. The lower symbol is an OR symbol (curved

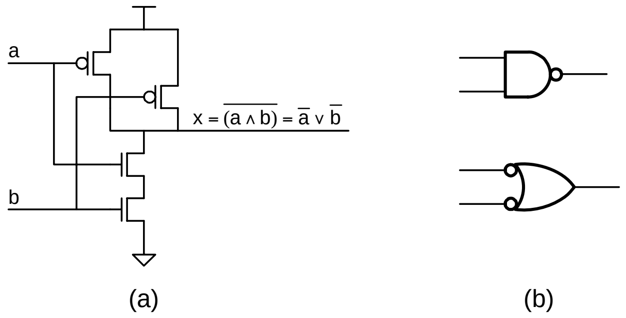


Figure 4.15: A CMOS NAND gate. (a) Circuit diagram — the NAND has a parallel PFET pull-up network and a series NFET pull-down network. (b) Schematic symbols — the NAND function can be though of as an AND with an inverted output (top) or an OR with inverted inputs (bottom).

left side, pointy right side) with inversion bubbles on all inputs — the inputs are inverted and then the inverted inputs are ORed, $f = \overline{a} \vee \overline{b}$. By DeMorgan's law (and duality), these two functions are equivalent. As with the inverter, we select between these two symbols to observe the bubble rule.

A NOR gate is constructed with a series network of PFETs and a parallel network of NFETs as shown in Figure 4.16(a). A series combination of PFETs (Figure 4.14(a)) connects the output to 1 when a and b are both low, $f = \overline{a} \wedge \overline{b} = \overline{a \vee b}$. Applying duality, this circuit is used in combination with a parallel NFET pulldown network (Figure 4.14(d)). The schematic symbols for the NOR gate are shown in Figure 4.16(b). As with the inverter and the NAND, we can choose between inverted inputs and inverted outputs depending on the bubble rule.

We are not restricted to building gates from just series and parallel networks. We can use arbitrary series-parallel networks, or even networks that are not series-parallel. For example, Figure 4.17(a) shows the transistor-level design for an AND-OR-Invert (AOI) gate. This circuit compute the function $f = \overline{(a \wedge b) \vee c}$. The pull-down network has a series connection of a and b in parallel with c. The pull-up network is the dual of this network with a parallel connection of a and b in series with c.

Figure 4.18 shows a majority-invert gate. We cannot build a single-stage majority gate since it is a monotonic increasing function and gates can only realize inverting functions. However we can build the complement of the majority function as shown. The majority is an interesting function in that it is its own dual. That is, $\text{maj}(\overline{a}, \overline{b}, \overline{c}) = \overline{\text{maj}(a, b, c)}$. Because of this we can implement the majority gate with a pull-up network that is identical to the pull-down network as shown in Figure 4.18(a). The majority function is also a *symmetric* logic function in that the inputs are all equivalent. Thus we can permute the inputs to the PFET and NFET networks without changing the function.

A more conventional implementation of the majority-invert gate is shown

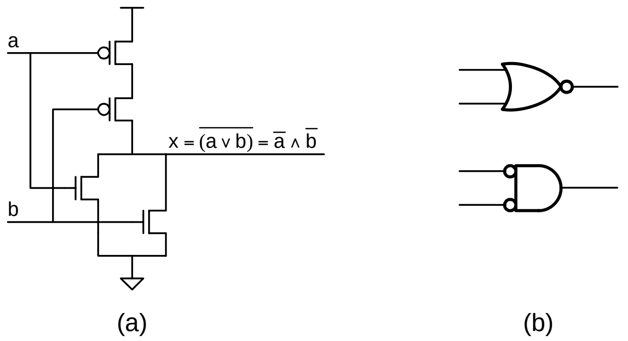


Figure 4.16: A CMOS NOR gate. (a) Circuit diagram — the NOR has a series PFET pull-up network and a parallel NFET pull-down network. (b) Schematic symbols — the NOR can be thought of as an OR with an inverted output or an AND with inverted inputs.

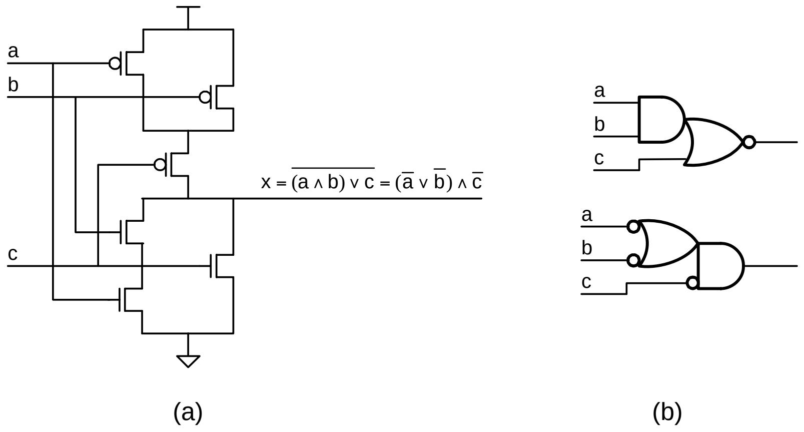


Figure 4.17: An AND-OR-Invert (AOI) gate. (a) Transistor-level implementation uses a parallel-series NFET pull-down network and its dual series-parallel PFET pull-up network. (b) Two schematic symbols for the AOI gate.

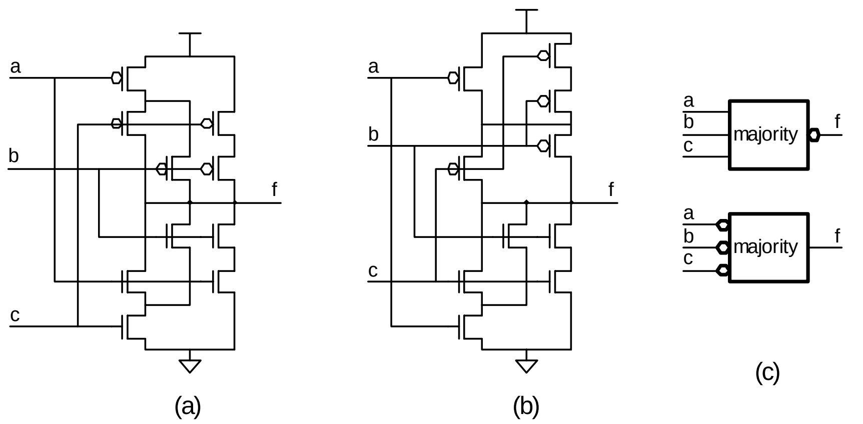


Figure 4.18: A majority-invert gate. The output is false if 2 or 3 of the inputs are true. (a) Implementation with symmetric pull-up and pull-down networks. (b) Implementation with pull-up network that is a dual of the pull-down network. (c) Schematic symbols — the function is still majority whether the inversion is on the input or the output.

in Figure 4.18(b). The NFET pull-down network here is the same as for Figure 4.18(a) but the PFET pull-up network has been replaced by a dual network — one that replaces each series element with a parallel element and vice-versa. The parallel combination of b and c in series with a in the pulldown network, for example, translates to a series combination of b and c in parallel with a in the pull-up network. A PFET pull-up network that is the dual of the NFET pull-down network will always give a switching function that is the complement of the pull-down network because of Equation (3.9).

Figure 4.18(c) shows two possible schematic symbols for the majority-invert gate. Because the majority function is self-dual, it doesn't matter whether we put the inversion bubbles on the inputs or the output. The function is a majority either way. If at least 2 out of the 3 inputs are high the output will be low — a majority with a low-true output. It is also the case that if at least 2 of the 3 inputs are low the output will be high — a majority with low-true inputs.

Strictly speaking, we cannot make a single-stage CMOS exclusive-or (XOR) gate because XOR is a non-monotonic function. A positive transition on an input may cause either a positive or negative transition on an output depending on the state of the other inputs. However, if we have inverted versions of the inputs, we can realize a two-input XOR function as shown in Figure 4.19(a), taking advantage of the switch network of Figure 4.7. A three-input XOR function can be realized as shown in Figure 4.19(b). The switch networks here are not series-parallel networks. If inverted inputs are not available, , it is

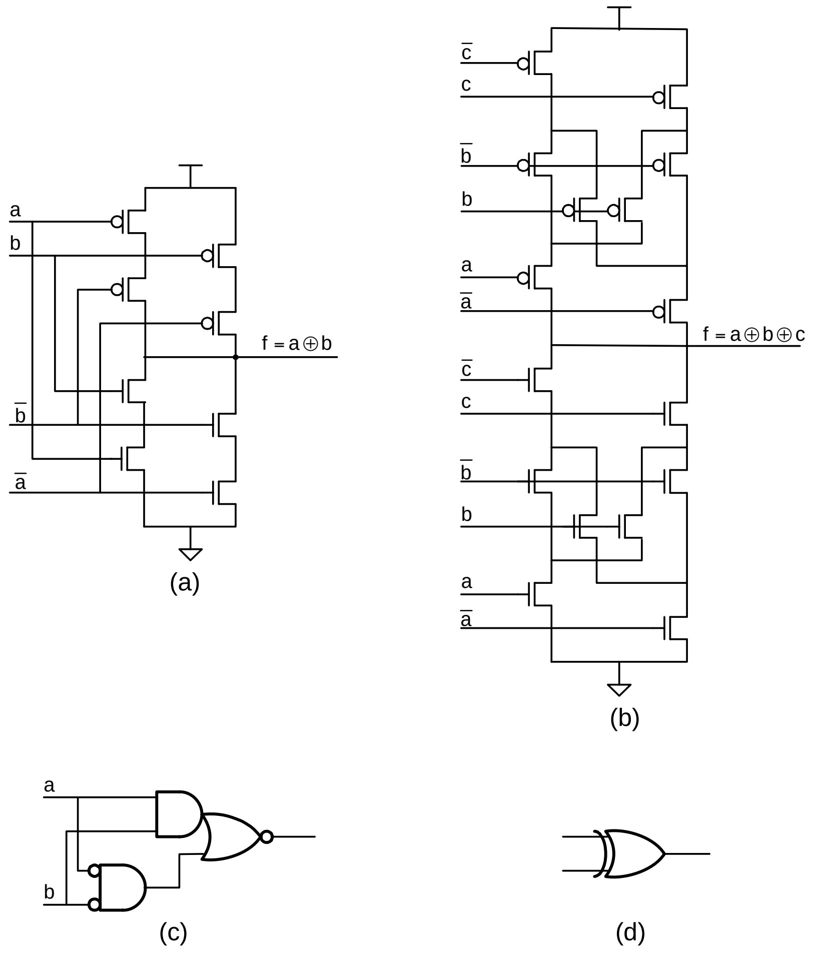


Figure 4.19: Exclusive-or (XOR) gates.

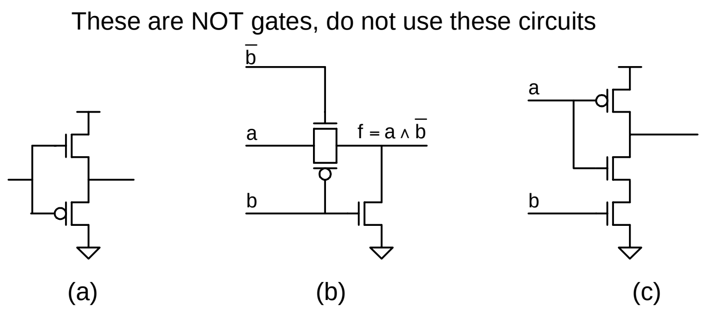


Figure 4.20: Three circuits that are *not* gates and shoudl not be used. (a) Attempts to pass a 1 through an NFET and a 0 through a PFET. (b) Does not restore high output values. (c) Does not drive the output when $a = 1$ and $b = 0$.

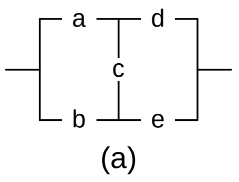
more efficient to realize a 2-input XOR gate using two CMOS gates in series as shown in Figure 4.19(c). We leave the transistor-level design of this circuit as an exercise. An XOR symbol is shown in Figure 4.19(d).

Before closing this chapter its worth examining a few circuits that aren't gates and don't work but represent common errors in CMOS circuit design. Figure 4.20 shows three representative mistakes. The would-be buffer in Figure 4.20(a) doesn't work because it attempts to pass a 1 through a PFET and a 0 through an NFET. The transistors cannot reliably pass those values and so the output signal is undefined — attenuated from the input swing at best. The AND-NOT circuit of Figure 4.20(b) does in fact implement the logical function $f = a \wedge \overline{b}$. However, it violates the digital abstraction in that it does not *restore* its output. If $b = 0$ an noise on input a is passed directly to the output.[10] Finally, the circuit of Figure 4.20(c) leaves its output disconnected when $a = 1$ and $b = 0$. Due to parasitic capacitance, the previous output value will be *stored* for a short period of time on the output node. However, after a period, the stored charge will leak off and the output node becomes an undefined value.

4.4 Bibliographic Notes

Kohavi gives a detailed treatment of switch networks. The switch model of the MOS transistor was first proposed by Bryant. A digital circuit design text like Rabaye is a good source of more detailed information on digital logic circuits.

[10]Such circuits can be used with care in isolated areas, but must be followed by a restoring stage before a long wire or another non-restoring gate. In most cases its better to steer clear of such *short-cut* gates.



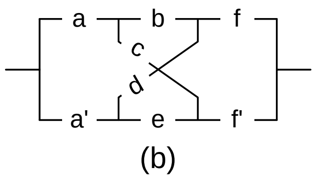


Figure 4.21: Switch network for Exercises 4–3 and 5.7.

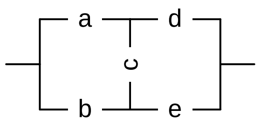


Figure 4.22: Switch network for Exercise 4–5.

4.5 Exercises

4–1 Analyze a simple switch circuit.

4–2 Synthesize a simple switch circuit.

4–3 Write down the logic function that describes the conditions under which the switch network of Figure 4.21(a) connects its two terminals. Note that this is not a series-parallel network.

4–4 Write down the logic function for the network of Figure 4.21(b).

4–5 Write down the logic function for the network of Figure 5.7.

4–6 Draw a schematic using NFETs and PFETs for a restoring logic gate that implements the function $f = a \wedge (b \vee c)$.

4–7 Write down the logic function implemented by the CMOS circuit of Figure ??.

4–8 Draw a transistor-level schematic for the XOR gate of Figure 4.19(c).

Chapter 5

Delay and Power of CMOS Circuits

The specification for a digital system typically includes not only its function, but also the delay and power (or energy) of the system. For example, a specification for an adder describes the function, that the output is to be the sum of the two inputs, as well as the delay, that the output must be valid within 2ns after the inputs are stable, and its energy, that each add consume no more than 5pJ. In this chapter we shall derive simple metohds to estimate the delay and power of CMOS logic circuits.

5.1 Delay of Static CMOS Gates

As illustrated in Figure 5.1 the delay of a logic gate, t_p, is the time from when the input of the gate crosses the 50% point between V_0 and V_1. Specifying delay in this manner allows us to compute the delay of a chain of logic gates by simply summing the delays of the individual gates. For example, in Figure 5.1 the delay from a to c is the sum of the delay of the two gates. The 50% point on the output of the first inverter is also the 50% point on the input of the second inverter.

Because the resistance of the PFET pull-up network may be different than that of the NFET pull-down network, a CMOS gate may have a rising delay that is different from its falling delay. When the two delays differ we denote the rising delay, the delay from a falling input to a rising output, as t_{pr} and the falling delay as t_{pf} as shown in Figure 5.1.

We can use the simple switch model derived in Section 4.2 to estimate t_{pr} and t_{pf} by calculating the RC time constant of the circuit formed by the output resistance of the driving gate and the input capacitance of its load(s).[1] Because

[1] In reality the driving gate has output capacitance roughly equal to its input capacitance. We ignore that capacitance here to simplify the model.

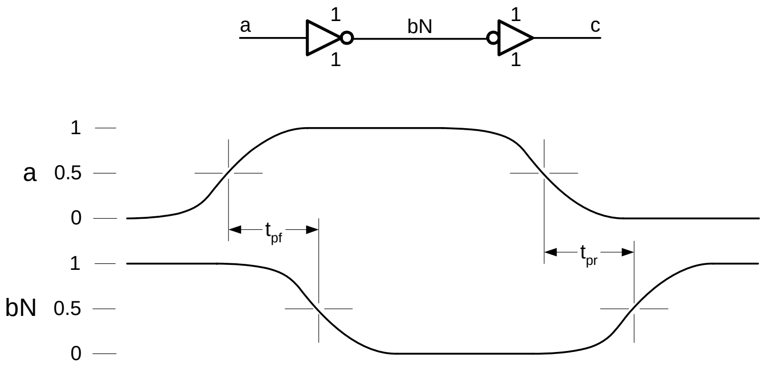


Figure 5.1: We measure delay from the 50% point of an input transition to the 50% point of an output transition. This figure shows the waveforms on input a and output bN with the falling and rising propagation delays, t_{pf} and t_{pr}, labeled

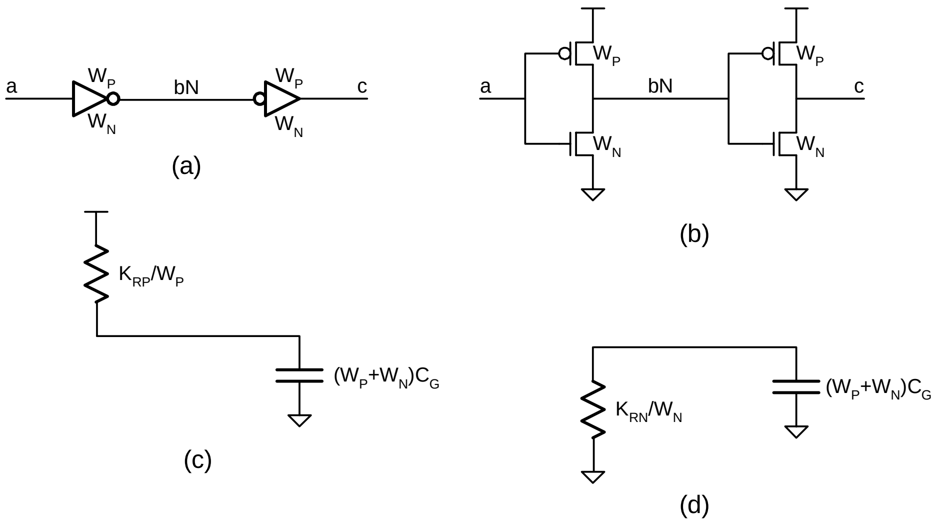


Figure 5.2: Delay of an inverter driving an identical inverter. (a) Logic diagram (all numbers are device widths), (b) Transistor-level circuit. (c) Switch-level model to compute rising delay, (d) Switch-level model for falling delay.

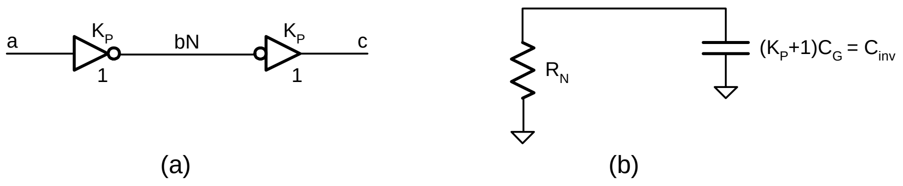


Figure 5.3: An inverter pair with equal rise-fall delays. (a) Logic diagram (sizings reflect parameters of Table 4.2). (b) Switch-level model of falling delay (rising delay is identical).

this time constant depends in equal parts on the driving and receiving gates, we cannot specify the delay of a gate by itself, but only as a function of output load.

Consider, for example, a CMOS inverter with a pullup of width W_P and a pulldown of width W_N driving an identical inverter, as shown in Figures 5.2(a) and (b).[2] For both rising and falling edges, the input capacitance of the second inverter is the sum of the capacitance of the PFET and NFET: $C_{\text{inv}} = (W_P + W_N)C_G$. When the output of the first inverter rises, the output resistance is that of the PFET with width W_P as shown in Figure 5.2(c): $R_P = K_{RP}/W_P = K_P K_{RN}/W_P$. Thus for a rising edge we have:

$$t_{pr} = R_P C_{\text{inv}} = \frac{K_P K_{RN}(W_P + W_N)C_G}{W_P}. \tag{5.1}$$

Similarly, for a falling edge, the output resistance is the resistance of the NFET pulldown as shown in Figure 5.2(d): $R_N = K_{RN}/W_N$. This gives a falling delay of:

$$t_{pf} = R_N C_{\text{inv}} = \frac{K_{RN}(W_P + W_N)C_G}{W_N}. \tag{5.2}$$

Most of the time we wish to size CMOS gates so that the rise and fall delays are equal; that is so $t_{pr} = t_{pf}$. For an inverter, this implies that $W_P = K_P W_N$, as show in Figure 5.3. We make the PFET K_P times wider than the NFET to account for the fact that its resistivity (per square) is K_P times larger. The PFET pull-up resistance becomes $R_P = K_{RP}/W_P = (K_P K_{RN})/(K_P W_N) = K_R N/W_N = R_N$. This gives equal resistance and hence equal delay. Equivalently, substituting for W_P in the formulae above gives.

$$t_{\text{inv}} = \frac{K_{RN}}{W_N}(K_P + 1)W_N C_G = (K_P + 1)K_{RN}C_G = (K_P + 1)\tau_N. \tag{5.3}$$

[2]W_P and W_N are in units of $W_{\text{min}} = 8L_{\text{min}}$. C_G here is the capacitance of a gate with width $8L_{\text{min}}$, so $C_G = 1.6$fF.

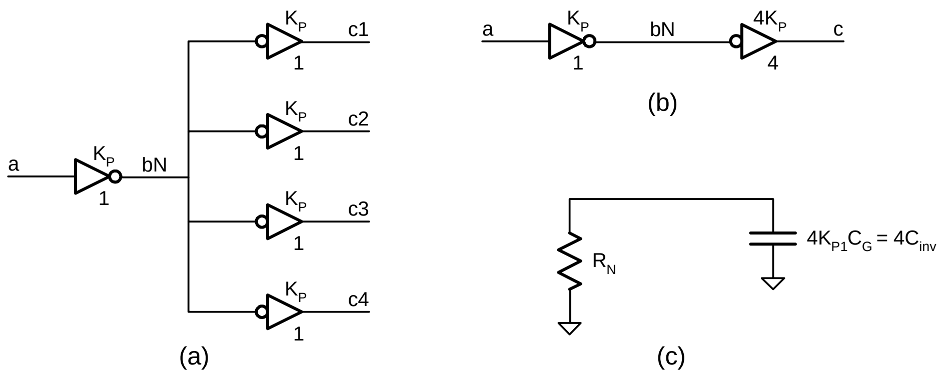


Figure 5.4: An inverter driving four-times its own load (a) Driving four other inverters. (b) Driving one large (4×) inverter. (c) Switch-level model of falling delay.

Note that the W_N term cancels out. The delay of an inverter driving an identical inverter, t_{inv} is independent of device width. As the devices are made wider R decreases and C increases leaving the total delay RC unchanged. For our model 0.13μm process with $K_P = 2.5$ this delay is $3.5\tau_N = 14$ps.[3]

Because the quantity $K_P + 1$ will appear frequently in our delay formulae, we will abbreviate this as $K_{P1} = K_P + 1$.

5.2 Fanout and Driving Large Loads

Consider the case where a single inverter of size 1 $W_N = W_{\min}$ sized for equal rise/fall delay ($W_P = K_P W_N$) drives four identical inverters as shown in Figure 5.4(a). The equivalent circuit for calculating the RC time constant is shown in Figure 5.4(c). Compared to the situation with identical inverters (fanout of one), this fanout-of-four situation has the same driving resistance, R_N, but four times the load capacitance, $4C_{\text{inv}}$. The result is that the delay for a fanout of four is four times the delay of the fanout of one circuit. In general, the delay for a fanout of F is F times the delay of a fanout of one circuit:

$$t_F = F t_{\text{inv}}. \tag{5.4}$$

The same situation occurs if the unit-sized inverter drives a single inverter that is sized four-times larger, as shown in Figure 5.4(b). The load capacitance on the first inverter is four times its input capacitance in both cases.

[3]For a minimum-sized $W_N = 8L_{\min}$ inverter, with equal rise/fall delay, $C_{\text{inv}} = 5.6fF$ in our model process.

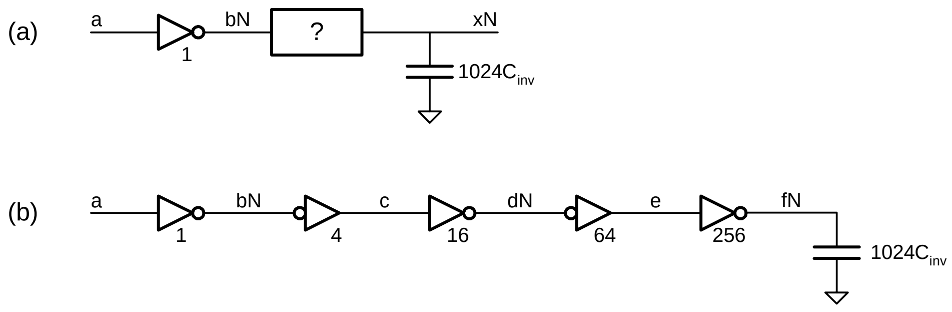


Figure 5.5: Driving a large capacitive load. (a) The output of a unit sized inverter needs to drive a fanout of 1024. We need a circuit to buffer up the signal bN to drive this large capacitance. (b) Minimum delay is achieved by using a chain of inverters that increases the drive by the same factor (in this case 4) at each stage.

When we have a very large fanout, it is advantageous to increase the drive of a signal in stages rather than all at once. This gives a delay that is logarithmic, rather than linear in the size of the fanout. Consider the situation shown in Figure 5.5(a). Signal bN, generated by a unit-sized inverter[4], must drive a load that is 1024 times larger than a unit-sized inverter (a fanout of $F = 1024$). If we simply connect bN to xN with a wire, the delay will be $1024t_{\text{inv}}$. If we increase the drive in stages, as shown in Figure 5.5(b), however, we have a circuit with five stages each with a fanout of four for a much smaller total delay of $20t_{\text{inv}}$.

In general, if we divide a fanout of F into n fanout of $\alpha = F^{1/n}$ stages, our delay will be

$$t_{Fn} = nF^{1/n}t_{\text{inv}} = \log_\alpha F\alpha t_{\text{inv}}. \tag{5.5}$$

We can solve for the minimum delay by taking taking the derivative of Equation (5.5) with respect to n (or α) and setting this derivative to zero. Solving shows that the minimum delay occurs for a fanout per stage of $\alpha = e$. In practice fanouts between 3 and 6 give good results. Fanouts much smaller than 3 result in too many stages while fanouts larger than 6 give too much delay per stage. A fanout of 4 is often used in practice. Overall, driving a large fanout, F, using multiple stages with a fanout of α reduces the delay from one that increases linearly with F to one that increases logarithmically with F — as $\log_\alpha F$.

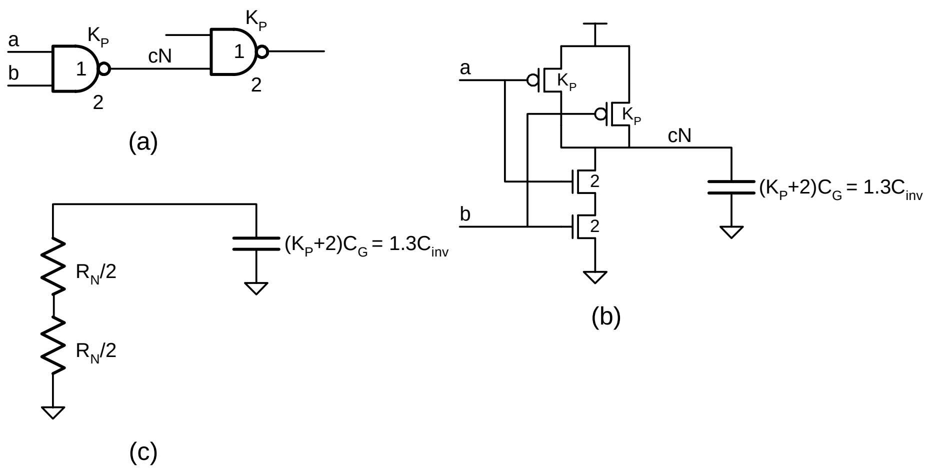


Figure 5.6: (a) A NAND gate driving an identical NAND gate. Both are sized for equal rise and fall delays. (b) Transistor-level schematic. (c) Switch-level model.

5.3 Fan-in and Logical Effort

Just as fan-out increases delay by increasing load capacitance, fan-in increases the delay of a gate by increasing output resistance — or equivalently input capacitance. To keep output drive constant, we size the transistors of a multi-input gate so that both the pull-up and pull-down the series resistance is equal to the resistance of an equal rise/fall inverter with the same relative size.

For example, consider a two-input NAND gate driving an identical NAND gate as shown in Figure 5.6(a). We size the devices of each NAND gate so each has the same worst-case up and down output resistance as a unit-drive equal rise/fall inverter as shown in Figure 5.6(b). Since in the worst-case only a single of the pullup PFETs is on, we size these PFETS $W_P = K_P$, just as in the inverter. We get no credit for the parallel combination of PFETs since both are on in only one of the three input states where the output is high (both inputs zero). To give a pull-down resistance equal to R_N each NFET in the series chain is sized at twice the minimum width. As shown in Figure 5.6(c) putting these two $R_N/2$ devices in series gives a total pull-down resistance of R_N. The capacitance of each input of this unit-drive NAND gate is the sum of the PFET and NFET capacitance: $(2 + K_P)C_G = \frac{2+K_P}{1+K_P}C_{\text{inv}}$.

We refer to this increase in input capacitance for the same output drive as the *logical effort* of the two input NAND gate. It represents the effort (in additional charge that must be moved compared to an inverter) to perform the

[4]From now on we may drop W_P from our diagrams whenever gates are sized for equal rise and fall.

2-input NAND logic function. The delay of a gate driving an identical gate (as in Figure 5.6(a)) is the product of its logical effort and t_{inv}.

In general, for a NAND gate with fan-in F, we size the PFETs K_P and the NFETs F giving an input capacitance of:

$$C_{\text{NAND}} = (F + K_P)C_G = \frac{F + K_P}{1 + K_P}C_{\text{inv}}, \tag{5.6}$$

and hence a logical effort of:

$$LE_{\text{NAND}} = \frac{F + K_P}{1 + K_P}, \tag{5.7}$$

and a delay of

$$t_{\text{NAND}} = LE_{\text{NAND}}t_{\text{inv}} = \frac{F + K_P}{1 + K_P}t_{\text{inv}}, \tag{5.8}$$

With a NOR gate the NFETs are in parallel, so a unit-drive NOR gate has NFETs pulldowns of size 1. In the NOR, the PFETs are in series, so a unit-drive NOR with a fan-in of F has PFET pullups of size FW_P. This gives a total input capacitance of:

$$C_{\text{NOR}} = (1 + FK_P)C_G = \frac{1 + FK_P}{1 + K_P}C_{\text{inv}}, \tag{5.9}$$

and hence a logical effort of:

$$LE_{\text{NOR}} = \frac{1 + FK_P}{1 + K_P}. \tag{5.10}$$

For reference, Table 5.3 gives the logical effort as a function of fan-in, F, for NAND and NOR gates with 1 to 5 inputs both as functions of K_P and numerically for $K_P = 2.5$ (the value for our model process).

5.4 Delay Calculation

The delay of each stage i of a logic circuit is the product of its fanout or *electrical effort* from stage i to stage $i+1$ and the logical effort of stage $i+1$. The fanout is the ratio of the drive of stage i to stage $i+1$. The logical effort is the capacitance multiplier applied to the input of stage $i+1$ to implement the logical function of that stage.

For example, consider the logic circuit shown in Figure 5.9. We calculate the delay from a to e one stage at a time as shown in Table 5.2. The first