Chapter 2

CMOS logic

2.1 Basics

We are going to stop worrying about discretizing time for a while, and examine how boolean-valued functions are implemented in CMOS. There are two nullary (no inputs) boolean functions: *true* and *false*. Their implementation is immediate from the choices we have made in the previous chapter; we implement them as voltage sources *Vdd* and *GND* respectively. Let us next look at the unary functions (one input). There are four of these:

```
Input | Output      I | O      I | O      I | O
--------------      -----      -----      -----
  0   |   0         0 | 1      0 | 0      0 | 1
  1   |   0         1 | 1      1 | 1      1 | 0

  FALSE             TRUE       IDENTITY   NOT
```


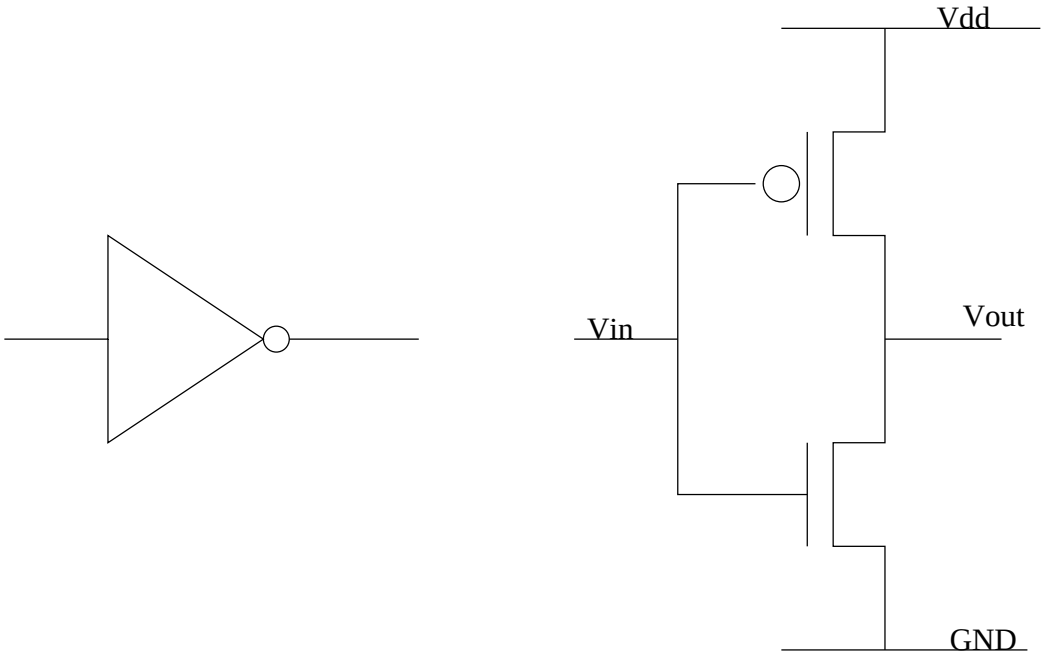

Figure 2.1: Two levels of abstraction of the CMOS inverter

The first two, `TRUE` and `FALSE`, can be implemented by ignoring the input and choosing the appropriate nullary function for the output. The third function, `IDENTITY`, can be implemented with a wire, provided that we can ignore the voltage drop, which we can do if there is no (sustained) current and if time is unlimited. For now we assume these conditions are met. The fourth function `NOT` can be implemented with a inverter. In Figure 2.1 we have revealed one more level of abstraction of this circuit.

We see that the inverter is built from two transistors, which we shall refer to as p-channel, or just p-transistor, and n-channel, or just n-transistor. The p-transistor is the one with the circle on its gate.

To understand how we may build other functions using these components, we examine their input-output characteristics in Figures 2.6-2.9. In Figures 2.2 and 2.6 we have a p-transistor hooked up to *Vdd* with the input voltage on the control terminal (the gate), and with one side of the channel (the source) hooked up to *Vdd* and the other (the drain) to the output and a resistor to ground. The pictures of the transistors suggest that the gate is isolated from the other two terminals. As we shall see later, this is indeed the case. There is no difference between the terminals that are called source and drain; for CMOS transistors this is strictly a naming convention. We see that the p-transistor connected this way behaves much like an inverter, and almost exactly like an inverter when the input is either V_h or V_l. The same is true for an n-transistor with its source terminal connected to *GND* (Figures 2.5 and 2.9).

The other figures show n and p-transistors connected to *Vdd* and *GND* respectively. We see that the output does not fall within the V_h or V_l ranges, even when the input does fall within those ranges. This is a reason to not use the transistors in those configurations, because we would loose a part of our voltage range every time such a circuit is used.

In summary we may say that n-transistors conduct "lows" well and "highs" poorly. The converse holds for "p-transistors": they conduct "highs" well and "lows" poorly.

A few points need clarification. If we can make a good inverter with an n-transistor and a resistor, or with a p-transistor and a resistor, why do we bother to use two transistors for each inverter? The answer is twofold. First, resistors are hard to make (reliably) in integrated circuits. We can fake one

by using a transistor with its gate connected to one of the voltage rails, so that it always conducts, but that doesn't help us in reducing the number of transistors! Another good reason for not using resistors, or transistors as resistors, is that the inverter with two transistors has an input that connects only to gates. Electrically we may then consider the input as a capacitor, and this fits nicely with our earlier requirement that wires should not draw a sustained current. Furthermore no having sustained currents saves power, unless the inverter switches it will draw virtually no power!

The second point we need to clarify is that you can see from Figures 2.6 and 2.9 that in the middle region both the p and the n-transistors are conducting and are connecting the output to opposite values! This, however is not really a problem in CMOS, because the channel resistance, even when the transistors are conducting, is relatively high (about 10kOhm for a typical two-micron process). However, the "off" resistance is about a factor of 10^5 higher, so there is plenty difference between "off" and "on."

Let us agree on a little bit of terminology. We shall say that the n-transistor is "cut" (off, non-conducting, open) when the gate voltage is V_l (with respect to the source or drain), and "tied" (on, conducting, closed) when the gate voltage is V_h. A p-transistor is cut when the gate is V_h and tied when the gate is V_l. (Verify this is the characteristics.) The little bubble on the gate of the p-transistor indicates that it operates (in terms of "cut" and "tied") as an n-transistor with an inverter on its gate.

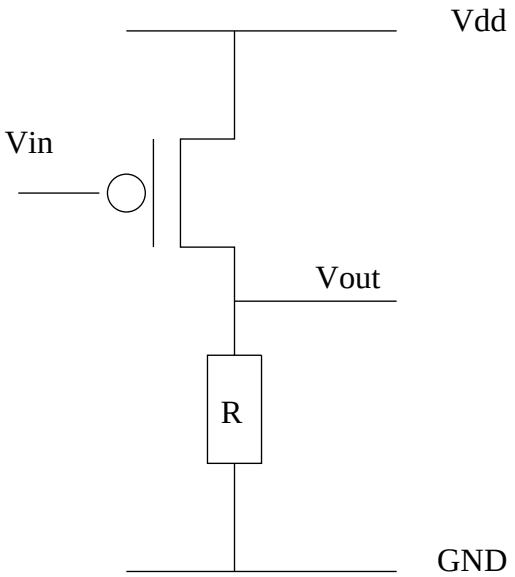

Figure 2.2: p transistor Vdd source

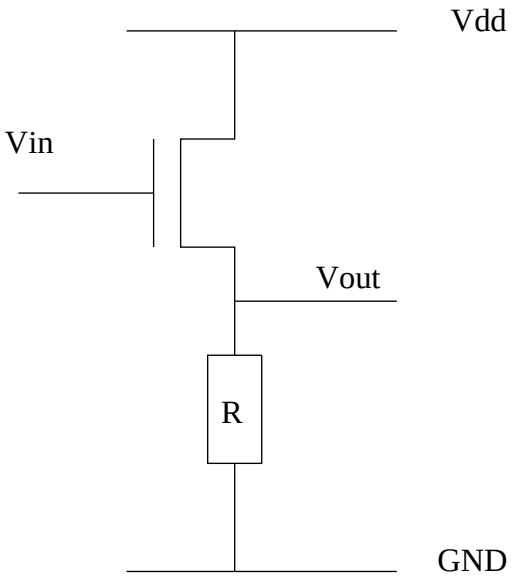

Figure 2.4: n transistor Vdd source

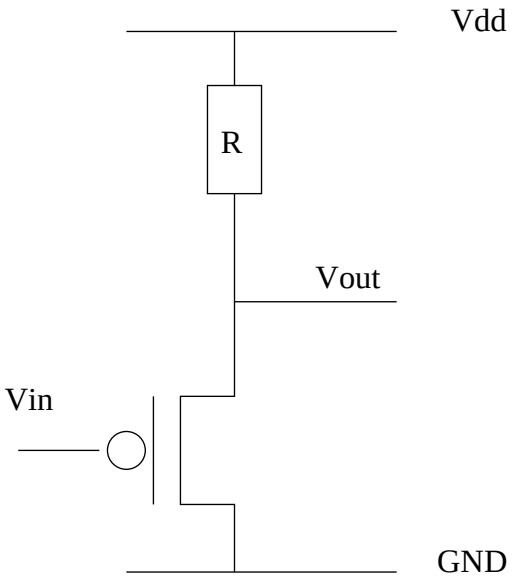

Figure 2.3: p transistor GND source

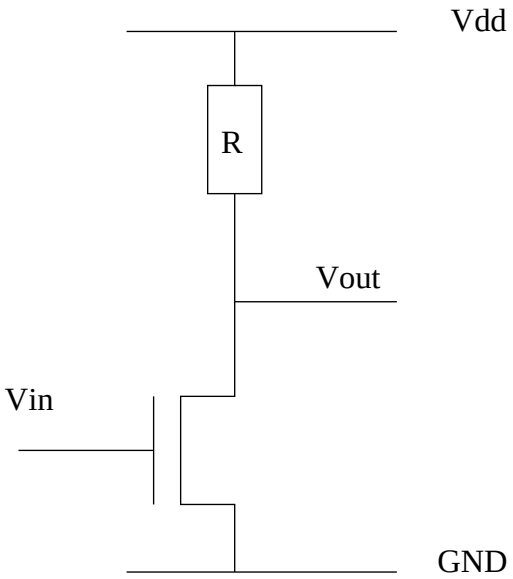

Figure 2.5: n transistor GND source

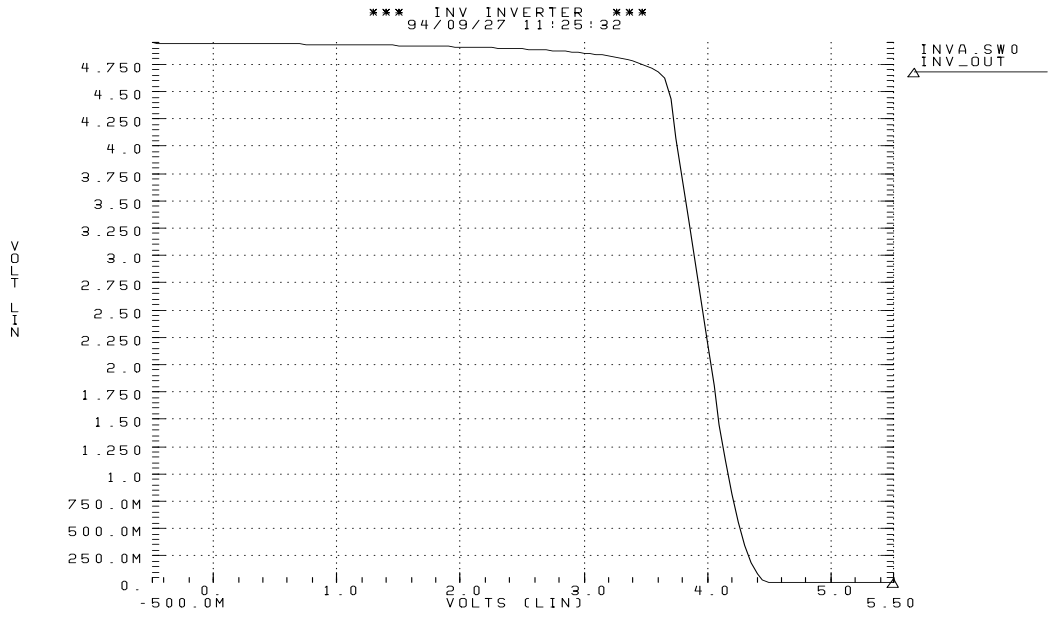

Figure 2.6: p transistor Vdd source

2.2 Scalar functions

In this section we indicate how you can implement any boolean-valued scalar (one output) function. The solution will not always be the best one possible, we will worry about finding efficient solutions later. Our starting point are the observations made in the previous section: p-transistors conduct highs well, are tied when the gate is low and cut when the gate is high, n-transistors conduct lows well, are cut when the gate is low and tied when the gate is high.

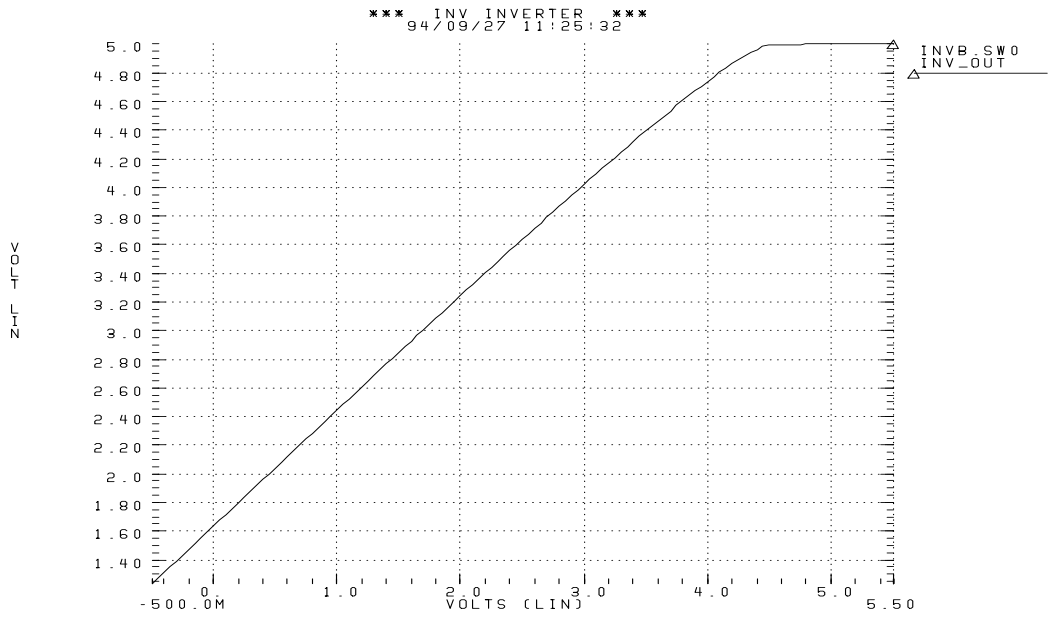

Figure 2.7: p transistor GND source

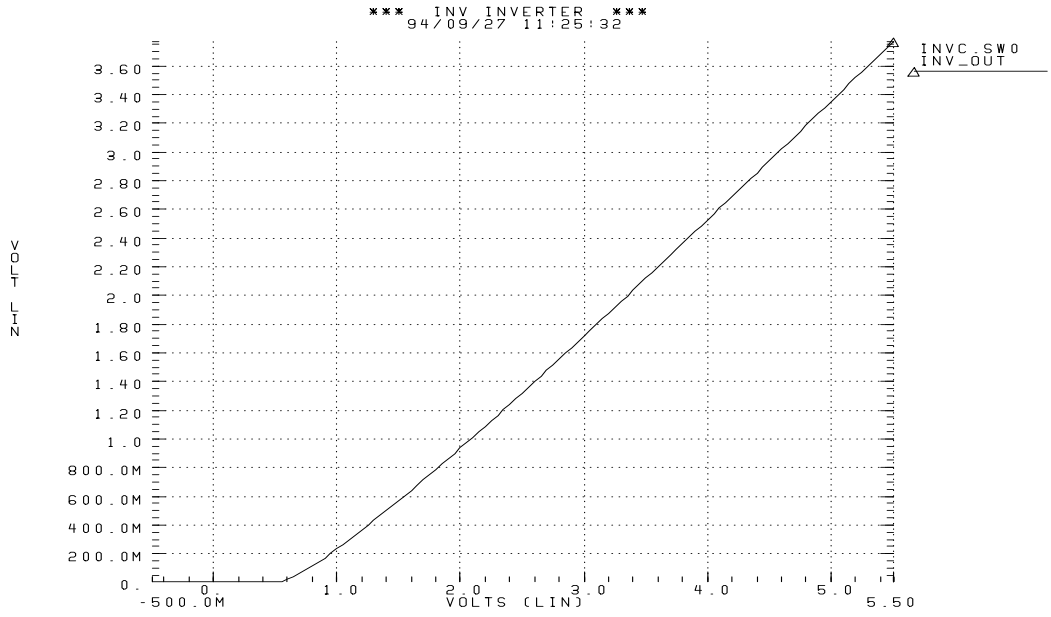

Figure 2.8: n transistor Vdd source

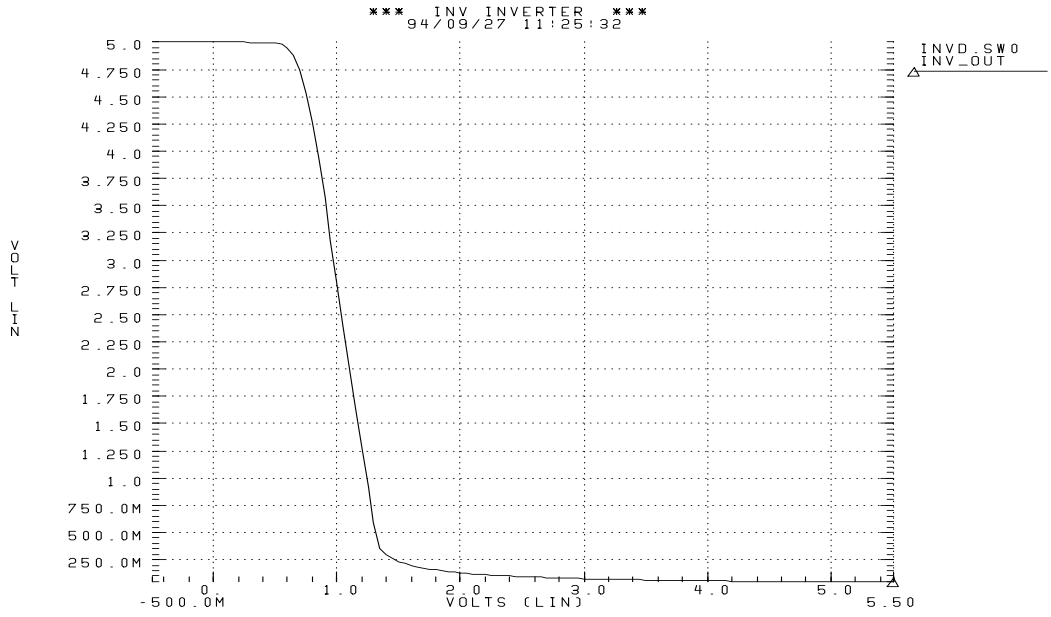

Figure 2.9: n transistor GND source

We require that (static) CMOS logic satisfies the following properties:

- The output is always driven if all inputs are valid(, and it is driven to the right value).
- There are no fights (i.e., transistors driving the output to opposite values) when all inputs are valid.
- If the output is driven high, the path connecting the output to Vdd is through p-transistors only.
- If the output is driven low, the path connecting the output to GND is through n-transistors only.

It is important to realize that for the purpose of checking for fights (only) we do assume that both kinds of transistors conduct both highs and lows.

In Figure 2.10 we have given the CMOS implementations of two circuits; the nand (and gate with inverted output) and nor (or gate with inverted output) gates. The reader is kindly invited to check that all conditions are met for either circuit.

These gates inspire us to examine the following general solution, represented in Figure 2.11.

The pullup network should be constructed from p-transistors only, and the pulldown network should be constructed from n-transistors only. The question is, can we realize all boolean scalar functions this way? The answer is no; we can only generate the anti-monotonic ones. Anti-monotonic means that if in a state where all inputs are valid we change an input from V_l to

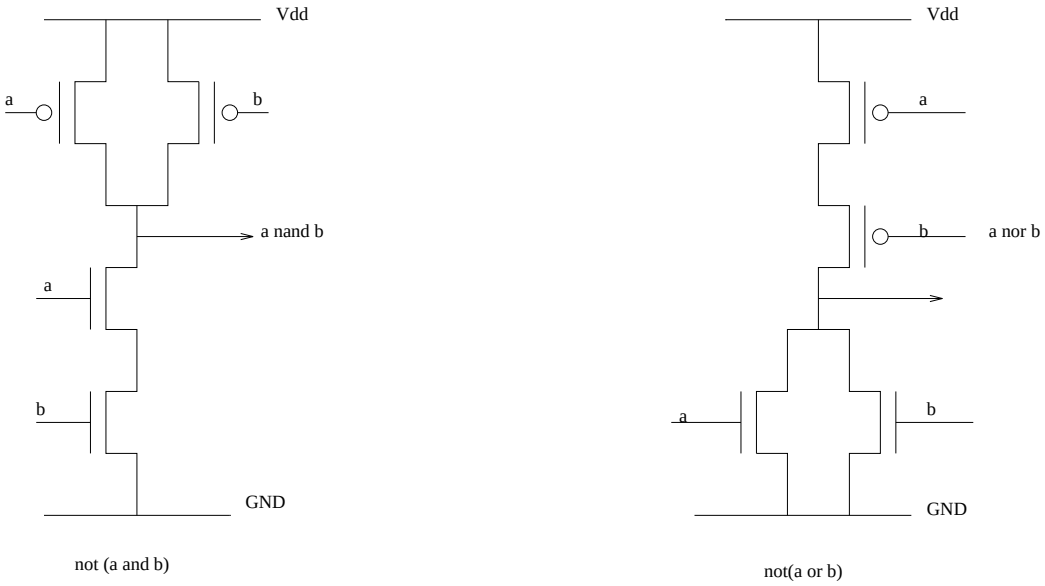

Figure 2.10: CMOS nand and nor gates

V_h, the output will go down (V_h to V_l) or stay the same (V_h to V_h or V_l to V_l), but it will not go up (V_l to V_h). This fact is sometimes expressed as: "CMOS logic is *inverting*." If we are willing to add the inverses of the inputs as possible inputs, we can build *all* boolean scalar functions.

To apply our rules for constructing circuits from the specification of the function, given as a boolean expression, we first rewrite the expression so that it has a negation on the outside, and no other negations except in front of a literal (the variables are called literals). For example:

$$a \wedge \neg(b \wedge \neg c)$$
$$= \{ \text{ Add a double negation } \}$$
$$\neg\neg(a \wedge \neg(b \wedge \neg c))$$
$$= \{ \text{ De Morgan's law } \}$$

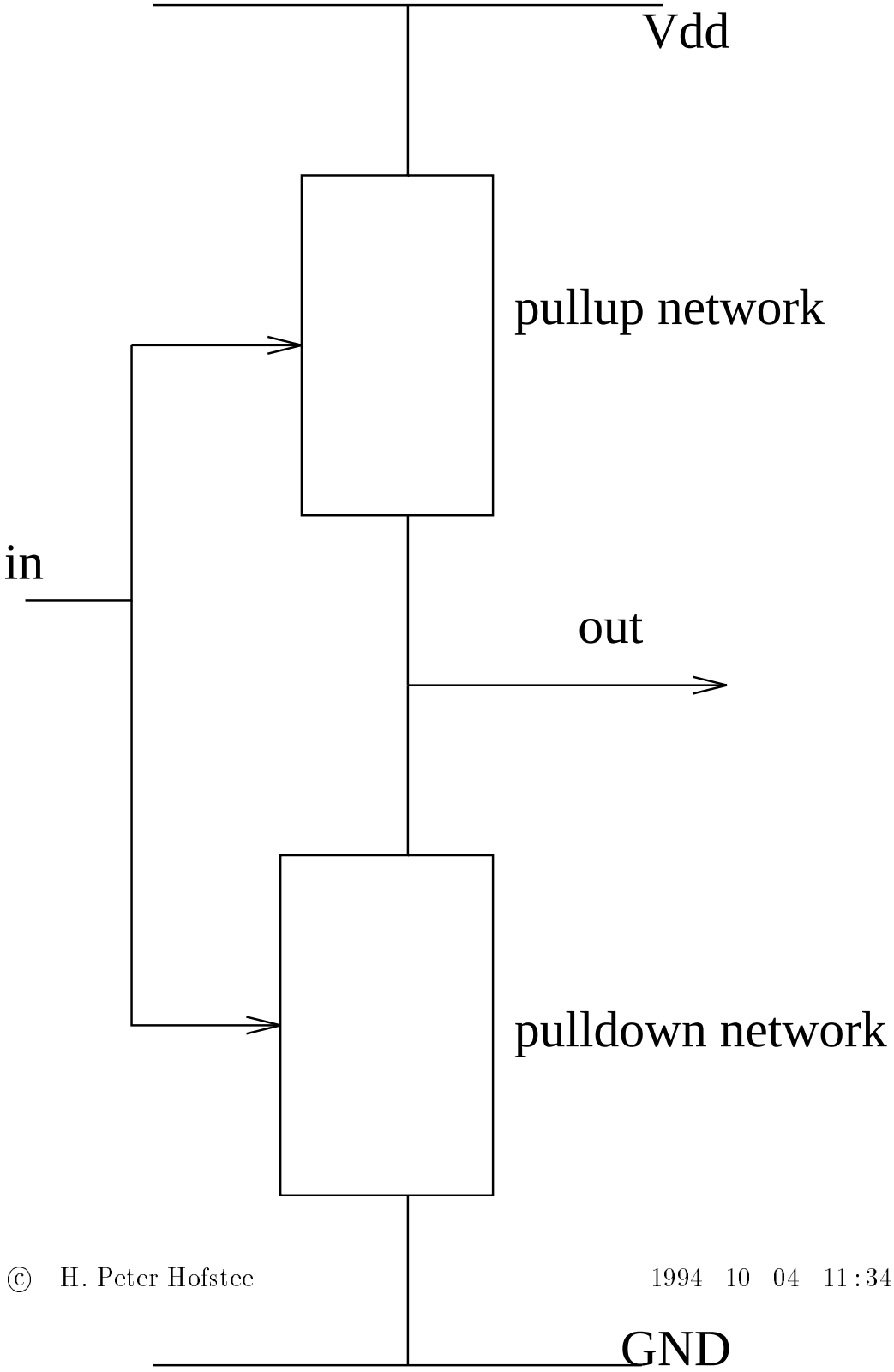

Figure 2.11: pullup-pulldown networks

$$\neg(\neg a \ \vee \ \neg\neg(b \ \wedge \ \neg c))$$
$$= \{ \text{ Double negation } \}$$
$$\neg(\neg a \ \vee \ (b \ \wedge \ \neg c))$$

which is an antimonotonic function of $\neg a, b$, and $\neg c$. A summary of the laws of boolean algebra, used in such calculations, is given in the next section.

Given an antimonotonic function $\neg f$ of the above form, we apply the following rules recursively.

- If f is a, the pullup circuit is a p-transistor, and the pulldown circuit an n-transistor.
- If f is $\neg a$, the pullup circuit is an inverter on the input a, with its output connected to the gate of a p-transistor and the pulldown circuit is an inverter on the input a with its ouput connected to the gate of an n-transistor.
- If f is $f0 \ \wedge \ f1$, the pullup circuit is the pullup circuit for $f0$ in parallel with the pullup circuit for $f1$ and the pulldown circuit is the pulldown circuit for $f0$ in series with the pulldown circuit for $f1$.
- If f is $f0 \ \vee \ f1$, the pullup circuit is the pullup circuit for $f0$ in series with the pullup circuit for $f1$ and the pulldown circuit is the pulldown circuit for $f0$ in parallel with the pulldown circuit for $f1$.

The rules are represented pictorially in Figure 2.12. Application of the rules to our example $\neg(\neg a \vee (b \ \wedge \ \neg c))$ results in the circuit in Figure 2.13.

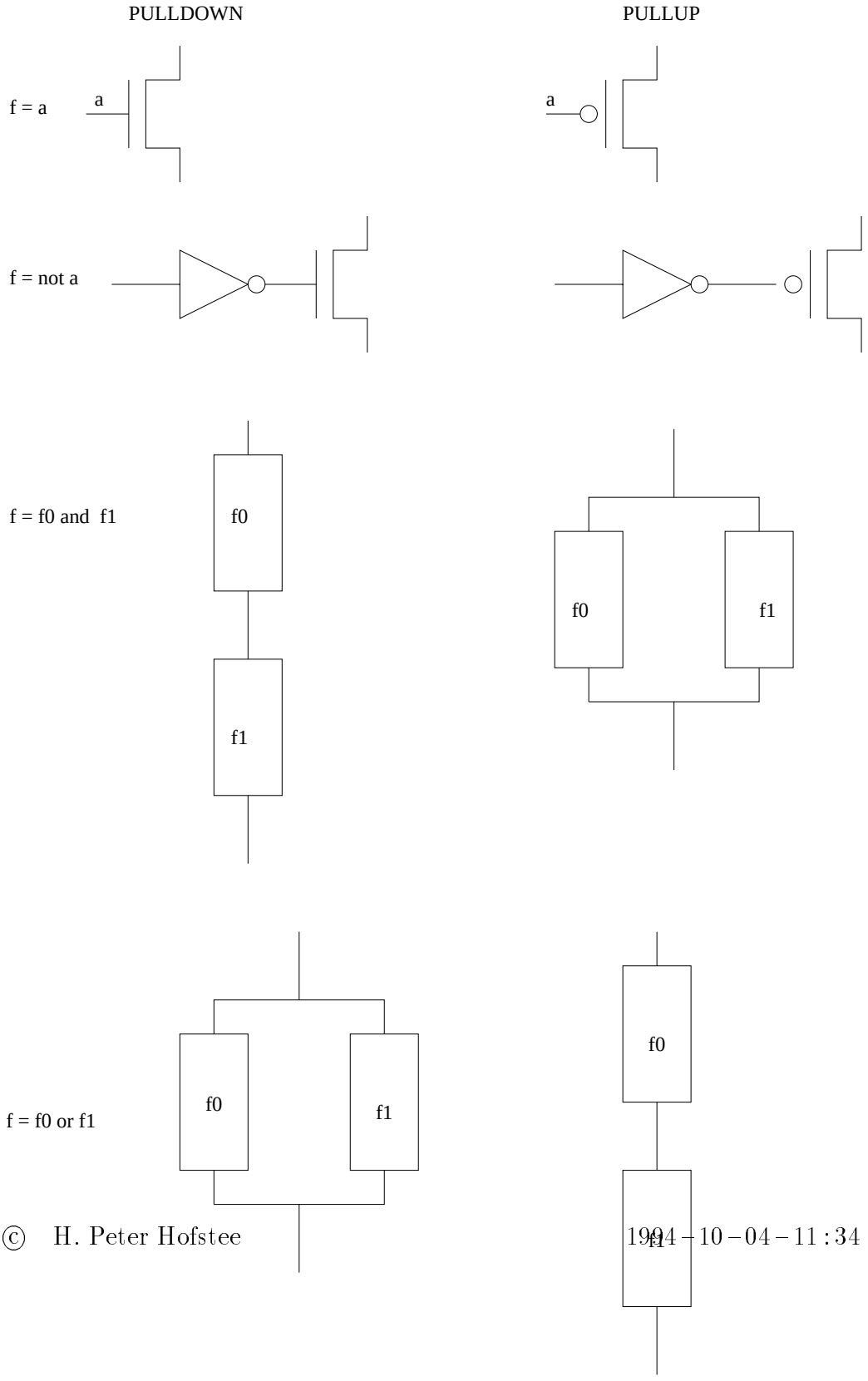

Figure 2.12: Pullup-pulldown network construction

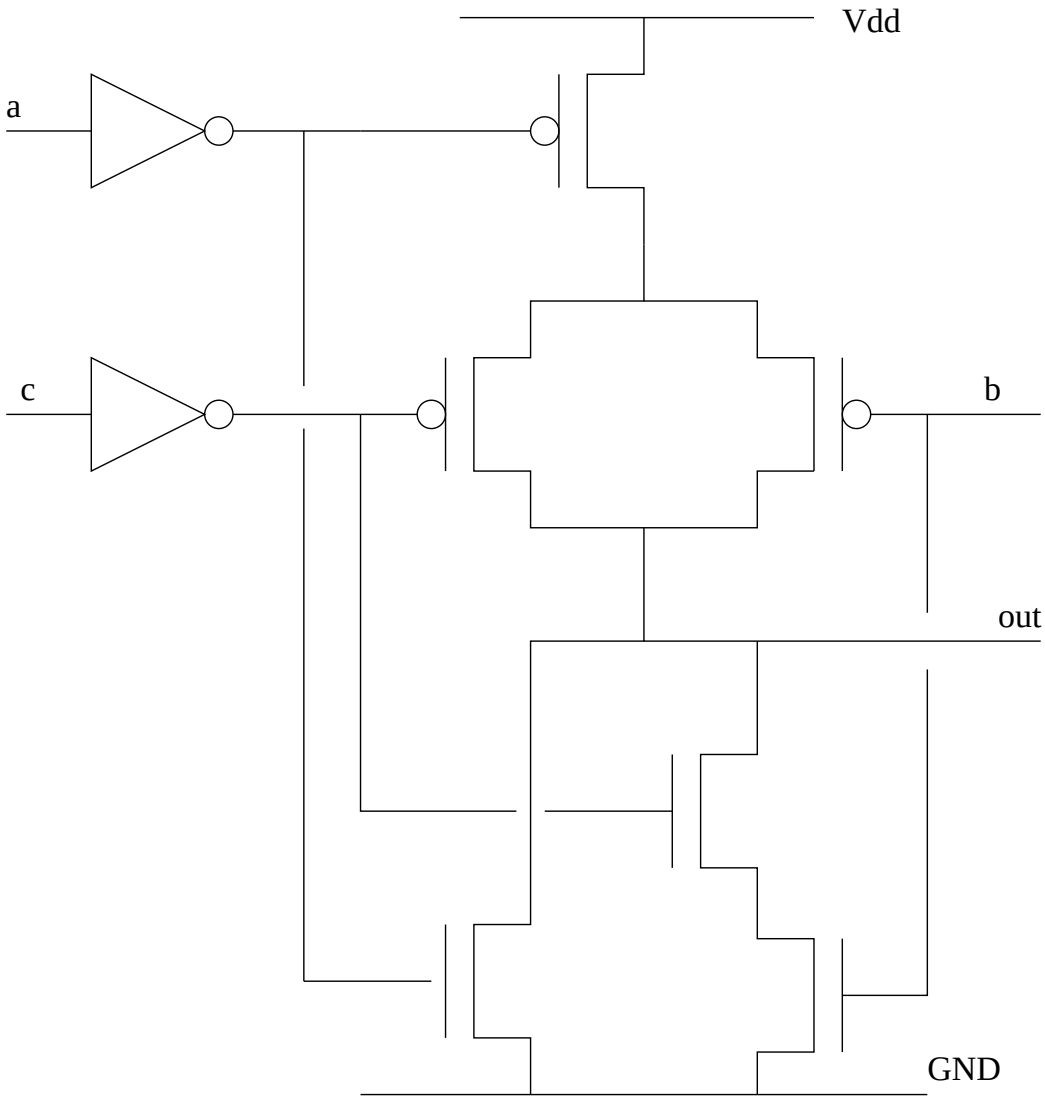

Figure 2.13: Circuit for $\neg(\neg a \vee (b \wedge \neg c))$

2.3 Boolean algebra

In this section we summarize the laws of boolean algebra. Some of the laws are redundant, i.e., they can be proven from the other ones, but this is a good collection of laws to remember and work with.

- Commutativity $E_0 \wedge E_1 = E_1 \wedge E_0$ $E_0 \vee E_1 = E_1 \vee E_0$
- Associativity $E_0 \wedge (E_1 \wedge E_2) = (E_0 \wedge E_1) \wedge E_2$
  $E_0 \vee (E_1 \vee E_2) = (E_0 \vee E_1) \vee E_2$
- Distributivity $E_0 \wedge (E_1 \vee E_2) = (E_0 \wedge E_1) \vee (E_0 \wedge E_2)$
  $E_0 \vee (E_1 \wedge E_2) = (E_0 \vee E_1) \wedge (E_0 \vee E_2)$
- De Morgan $\neg(E_0 \wedge E_1) = (\neg E_0 \vee \neg E_1)$
  $\neg(E_0 \vee E_1) = (\neg E_0 \wedge \neg E_1)$
- Negation $\neg\neg E_0 = E_0$
- Excluded middle $E_0 \vee \neg E_0$
- Contradiction $\neg(E_0 \wedge \neg E_0)$
- Or-simplification $E_0 \vee E_0 = E_0$ $E_0 \vee \mathit{true} = \mathit{true}$
  $E_0 \vee \mathit{false} = E_0$ $E_0 \vee (E_0 \wedge E_1) = E_0$
- And-simplification $E_0 \wedge E_0 = E_0$ $E_0 \wedge \mathit{true} = E_0$
  $E_0 \wedge \mathit{false} = \mathit{false}$ $E_0 \wedge (E_0 \vee E_1) = E_0$

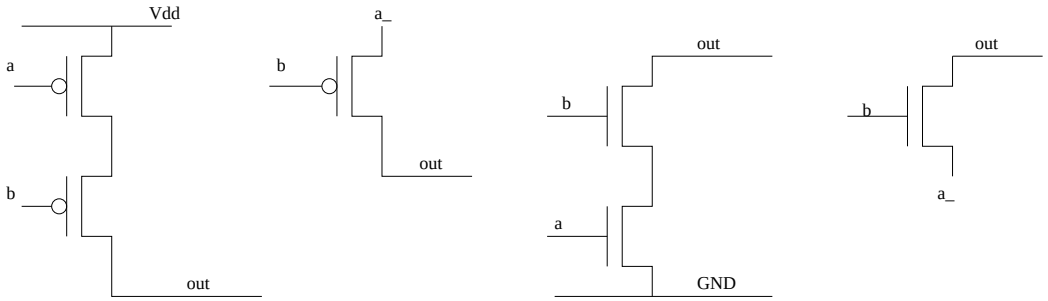

Figure 2.14: Non-restoring (pass gate) logic.

2.4 Non-restoring logic

In some situations we can save a few transistors by using a signal, rather than one of the power rails, as a voltage source. Figure 2.14 explains how this is done. The reason the first example works is that the circuit is only supposed to drive the output hih if both a and b are low. In that case $\neg a$ is high and can be used as a voltage source. This way of using transistors is referred to as *pass-gate*, or *non-restoring* logic ("gate" in pass-gate means transistor).

Using such an arrangement often saves a considerable amount of space. However, it must be used with great care, and the following points must be kept in mind.

- The circuit must remain logically correct (see the rules for restoring logic).
- Do not put more than 5 transistors in series. Transistors get weaker with increasing channel length (see next chapter). You should keep this rule in mind especially when you are designing a cell that is to be

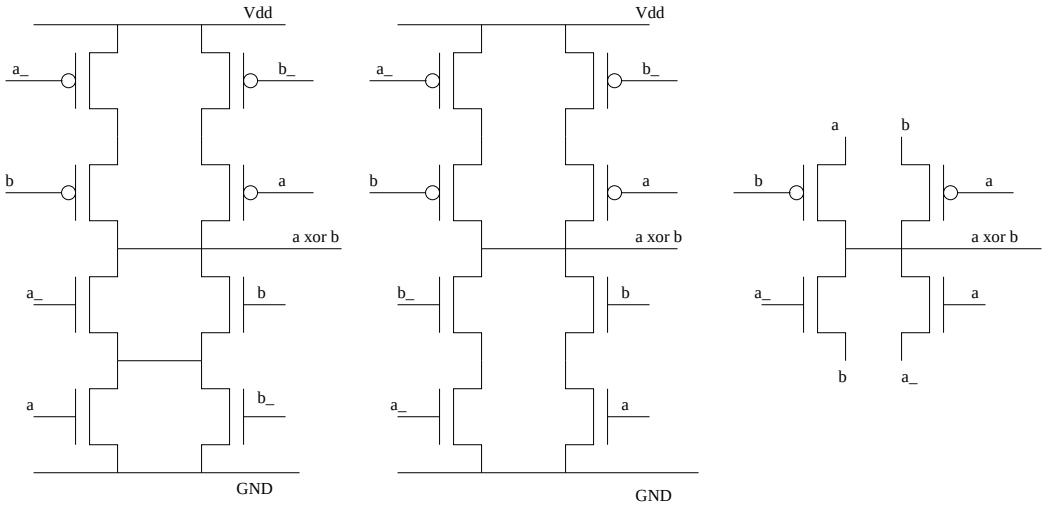

Figure 2.15: "Tricky" xor

composed.

- Keep in mind that transistors are symmetric! Make sure that the direction of the currents is guaranteed by the circuit, and is not just what you have in mind.

In Figure 2.15 we show two transformations of an xor circuit, the last one introduces pass gates.