Today’s lecture:

- Review and explanation of last time.
- A more systematic approach—applying boolean algebra to CMOS design (the transistor as switch).
- Production rules and CAST.
Composable Design

In order to be able to compose circuits arbitrarily in a system, we have to be concerned about signal “quality.”

- Noise suppression around Vdd and GND due to thresholds
- strong zero and strong one
- gain (amplification): \( \frac{\Delta V_{out}}{\Delta V_{in}} \gg 1 \) around \( V_{out} = V_{in} \)

Restoring logic: a circuit must be able to supply more energy to its output than was supplied to its inputs. Which implies it must draw power from the power supply rather than its inputs.
The transistors

Review the properties and naming conventions of the transistors.

- pFET passes highs well
- nFET passes lows well
- Almost zero current to gate in steady-state.
- Very good switches. $R_{\text{off}} / R_{\text{on}} \approx 10^5$
- Gain
Restoring combinational logic

Restoring combinational (static) logic is the simplest and most basic way to design CMOS circuitry.

Four rules:
- Highs passed by pFETs.
- Lows passed by nFETs.
- Output always driven to VDD or GND.
- No short circuits between the power supplies.

Restoring, static logic is the default design style.
The CMOS Pass Gate

(A different way of thinking about CMOS circuits.)

When we use this in restoring logic designs, we can *always* omit one of the transistors!

This is usually not true in non-restoring designs since we do not know what value the input has, so the circuit may have to pass either a high or a low.

- Pass-gate logic can be bad since it allows the signal to degrade. We do not allow more than a few pass gates in series along a signal path.
- Sometimes pass-gate design allows us to save transistors or stages of logic (e.g., when implementing non-inverting logic—see Lab 2).
An example

2-input multiplexer.

Two data inputs $a$ and $b$, one control input $s$. Output is $x$.

\[ s = 0 \Rightarrow x := a \]
\[ s = 1 \Rightarrow x := b \]

We already know how to do this ('153 MSI package):

A horrible way of doing it! (20 transistors as drawn and slow as well—five stages of logic!)
Restoring multiplexer

The specification requires us to generate the outputs in the same sense as the inputs. Restoring CMOS logic can’t do this in one step. (12 transistors, three stages, including inverter for $s$.)
Pass-gate multiplexer

6 transistors, 2 stages. Very simple (and fast). But it does not restore the quality of the logic signals.
Restoring multiplexer with pass gates

If we want to make it restoring, add inverters on the input (12 transistors, 3 stages, including inverter for $s$):

Do we need all the connections? What’s left?

We went from a pass-gate implementation to a restoring implementation. In Lab 2, you will find out how to go the other way.
Circuit Synthesis

- We have described many of the features that we can demand from our circuits, such as the four rules of restoring combinational logic design.
- You should be able to check that a given CMOS circuit obeys the four rules, and you should also be able to check what function it generates.

How do we synthesize a circuit so that it is guaranteed to satisfy these rules and at the same time implement the function we want?
Boolean Algebra

Applications for us:

- Checking (satisfying) our rules for logic design.
- Logic minimization.

We actually use boolean algebra less than you might think. Many of the operations can be automated. Also, modern high-performance designs often emphasize very simple logic where these methods are overkill.

Notation note:

\[ AB + CD = (A \land B) \lor (C \land D). \]
Axioms of boolean (switching) algebra:

1. Associativity:

\[ a \lor (b \lor c) = (a \lor b) \lor c \text{ and } a \land (b \land c) = (a \land b) \land c \]

2. Commutativity:

\[ a \lor b = b \lor a \text{ and } a \land b = b \land a \]

3. Distributivity:

\[ a \lor (b \land c) = (a \lor b) \land (a \lor c) \]

\[ \text{and} \]

\[ a \land (b \lor c) = (a \land b) \lor (a \land c) \]

4.

\[ a \lor 0 = a \text{ and } a \land 1 = a \]

5.

\[ a \lor \neg a = 1 \text{ and } a \land \neg a = 0 \]

6.

\[ a \lor (\neg a \land b) = a \lor b \]
Example of Switching Algebra

De Morgan’s Laws.

To prove: \( \neg(a \land b) = \neg a \lor \neg b \):

let \( s = \neg a \lor \neg b \), \( u = \neg(a \land b) \). calculate \( s \lor \neg u \) and \( s \land \neg u \).

\[
\begin{align*}
    s \land \neg u &= (\neg a \lor \neg b) \land \neg \neg(a \land b) \\
    &= ((\neg a \lor \neg b) \land a) \land b \\
    &= ((\neg a \land a) \lor (\neg b \land a)) \land b = (\neg b \land a) \land b = 0.
\end{align*}
\]
Example Cont’d

Now check $s \lor \neg u$.

\[
    s \lor \neg u = (\neg a \lor \neg b) \lor (a \land b)
\]

\{ associativity \}

\[
    = \neg a \lor (\neg b \lor (a \land b))
\]

\{ distributivity \}

\[
    = \neg a \lor ((\neg b \lor a) \land (\neg b \lor b)) = \neg a \lor (\neg b \lor a) = 1.
\]
The Duality Slide

Notice that all the rules are entirely symmetric. Using slightly strange symbols, we have:

\[
\begin{align*}
X \land X &= X \\
X \land T &= X \\
X \land \bot &= \bot \\
X \land H &= H \land X \\
X \land (H \land Z) &= (X \land H) \land Z \\
X \land (H \lor Z) &= (X \land H) \lor (X \land Z) \\
X \land \chi &= \bot \\
(X \neq \bot) &= (X = T)
\end{align*}
\]

And much more... (Seitz 1985.)
First application of the Theory

De Morgan’s Theorem(s):

\[-(a \land b) = \neg a \lor \neg b\]

\[-(a \lor b) = \neg a \land \neg b\]

Application: We want to design a CMOS circuit to implement the logical function \( f \). Suppose we are given (or know) the (series-parallel) pulldown network in nFETs... How do we generate the pullup network?
Example, continued

Answer. Given the pulldown network, we know under what conditions the output \( f \) should be zero.

But then we also know when it should be one, since we always have to drive the output. We need to generate a network of pFETs to drive the output high whenever we are not driving the output low with the nFETs. The nFETs pull the output low if the inputs satisfy some logical equation. E.g.,

\[
a \land b \land c \rightarrow \text{nFETs conduct.}
\]

Then we should have

\[
\neg(a \land b \land c) \rightarrow \text{pFETs conduct.}
\]

but we can write this with the help of De Morgan’s theorem:

\[
\neg a \lor \neg b \lor \neg c \rightarrow \text{pFETs conduct.}
\]

This is all we need to do since pFETs conduct when their inputs are low!
Example—Conclusion

To easily generate static series-parallel logic gate, make sure that the pullup network is the \textit{logical dual} of the pulldown network. (E.g., nFETs in series in the pulldown network correspond to pFETs in parallel in the pullup network.)

Note: The topological dual is not always the best implementation (see multiplexer).
Implementing Boolean Expressions

Boolean minimization procedures generally deal with expressions in sum-of-products form:

\[ ab + cd + ae \leftrightarrow (a \land b) \lor (c \land d) \lor (a \land e) \]

We can implement a fairly complex expression in a single stage of logic. . .

How much we can do in a single stage of logic is limited by performance considerations.
The old-fashioned way

Sometimes, we have to use trees for our logic. This could happen, for instance, when we would otherwise have too many transistors in series. This is more of a “traditional” approach...

(These are all NAND-type gates! This kind of structure is probably best to implement with a PLA, where we use timing tricks to eliminate most of the pullup network.) We try to avoid tree structures such as this one because they are a layout nightmare, and it is often possible to implement the function in a single stage of logic.
Implementing Boolean Functions in CMOS

We know enough to implement any boolean function in static CMOS.

- Find under what conditions the output should be zero. Express this as a sum-of-products expression.
- Use nFETs to construct a pulldown network that accomplishes this.
- You may have to use inverters to complement some inputs.
- Construct the pullup network as a dual network (to the nFET network) of pFETs. You will not need any extra inverters here!
Why the pulldowns?

It is often possible to do (much) better than this.

The main reason for emphasizing the pulldown network is that pFETs, being weaker than nFETs, are undesirable, and many circuit design techniques concentrate on eliminating as many pFETs as possible. The pulldown networks used in those designs are usually similar or identical to the simple combinational ones we are talking about today.

More about this next week.
Next Time

Next time, we will continue with applying boolean algebra to CMOS design.

- Homework 2 questions.
- Boolean minimization—Karnaugh maps, Quine-McCluskey method.
- Minimal-gate networks.
- Layout techniques for combinational logic.