Now would be a good time to take a moment to look at the documentation for the library of logic gates we'll use for our designs.

Look for "The Standard Cell Library" handout in the Updates & Handouts tab, which is next to the Courseware tab.

The information on this slide is taken from there.

The library has both inverting gates (such as inverters, NANDs and NORs) and non-inverting gates (such as buffers, ANDs and ORs).

Why bother to include both types of gates?

Didn't we just learn we can build any circuit using only NAND or NOR?

Good questions!

We get some insight into the answers if we look at these three implementations for a 4-input AND function.

The upper circuit is a direct implementation using the 4-input AND gate available in the library.

The tPD of the gate is 160 picoseconds and its size is 20 square microns.

Don't worry too much about the actual numbers, what matters on this slide is how the numbers compare between designs.

The middle circuit implements the same function, this time using a 4-INPUT NAND gate hooked to an inverter to produce the AND functionality we want.

The tPD of this circuit is 90 picoseconds, considerably faster than the single gate above.

The tradeoff is that the size is somewhat larger.

How can this be?

Especially since we know the AND gate implementation is the NAND/INVERTER pair shown in the middle circuit.

The answer is that the creators of the library decided to make the non-inverting gates small but slow by using MOSFETs with much smaller widths than used in the inverting logic gates, which were designed to be fast.

Why would we ever want to use a slow gate?

Remember that the propagation delay of a circuit is set by the longest path in terms of delay from inputs to outputs.

In a complex circuit, there are many input/output paths, but it's only the components on the longest path that need to be fast in order to achieve the best possible overall tPD.

The components on the other, shorter paths, can potentially be a bit slower.

And the components on short input/output paths can be very slow indeed.

So for the portions of the circuit that aren't speed sensitive, it's a good tradeoff to use slower but smaller gates.

The overall performance isn't affected, but the total size is improved.

So for faster performance we'll design with inverting gates, and for smallest size we'll design with non-inverting gates.

The creators of the gate library designed the available gates with this tradeoff in mind.

The 4-input inverting gates are also designed with this tradeoff in mind.

For the ultimate in performance, we want to use a tree circuit of 2-input gates, as shown in the lower circuit.

This implementation shaves 10 picoseconds off the tPD, while costing us a bit more in size.

Take a closer look at the lower circuit.

This tree circuit uses two NAND gates whose outputs are combined with a NOR gate.

Does this really compute the AND of A, B, C, and D?

Yup, as you can verify by building the truth table for this combinational system using the truth tables for NAND and NOR.

This circuit is a good example of the application of a particular Boolean identity known as Demorgan's Law.

There are two forms of Demorgan's law, both of which are shown here.

The top form is the one we're interested in for analyzing the lower circuit.

It tells us that the NOR of A with B is equivalent to the AND of (NOT A) with (NOT B).

So the 2-input NOR gate can be thought of as a 2-input AND gate with inverting inputs.

How does this help?

We can now see that the lower circuit is actually a tree of AND gates, where the inverting outputs of the first layer match up with the inverting inputs of the second layer.

It's a little confusing the first time you see it, but with practice you'll be comfortable using Demorgan's law when building trees or chains of inverting logic.

Using Demorgan's Law we can answer the question of how to build NANDs and NORs with large numbers of inputs.

Our gate library includes inverting gates with up to 4 inputs.

Why stop there?

Well, the pulldown chain of a 4-input NAND gate has 4 NFETs in series and the resistance of the conducting channels is starting to add up.

We could make the NFETs wider to compensate, but then the gate gets much larger and the wider NFETs impose a higher capacitive load on the input signals.

The number of possible tradeoffs between size and speed grows rapidly with the number of inputs, so it's usually just best for the library designer to stop at 4-input gates and let the circuit designer take it from there.

Happily, Demorgan's law shows us how build trees of alternating NANDs and NORs to build inverting logic with a large number of inputs.

Here we see schematics for an 8-input NAND and an 8-input NOR gate.

Think of the middle layer of NOR gates in the left circuit as AND gates with inverting inputs and then it's easy to see that the circuit is a tree of ANDs with an inverting output.

Similarly, think of the middle layer of NAND gates in the right circuit as OR gates with inverting inputs and see that we really have a tree of OR gates with an inverting output.

Now let's see how to build sum-of-products circuits using inverting logic.

The two circuits shown here implement the same sum-of-products logic function.

The one on the top uses two layers of NAND gates, the one on the bottom, two layers of NOR gates.

Let's visualize Demorgan's Law in action on the top circuit.

The NAND gate with Y on its output can be transformed by Demorgan's Law into an OR gate with inverting inputs.

So we can redraw the circuit on the top left as the circuit shown on the top right.

Now, notice that the inverting outputs of the first layer are cancelled by the inverting inputs of the second layer, a step we can show visually by removing matching inversions.

And, voila, we see the NAND/NAND circuit in sum-of-products form: a layer of inverters, a layer of AND gates, and an OR gate to combine the product terms.

We can use a similar visualization to transform the output gate of the bottom circuit, giving us the circuit on the bottom right.

Match up the bubbles and we see that we have the same logic function as above.

Looking at the NOR/NOR circuit on the bottom left, we see it has 4 inverters, whereas the NAND/NAND circuit only has one.

Why would we ever use the NOR/NOR implementation?

It has to do with the loading on the inputs.

In the top circuit, the input A connects to a total of four MOSFET switches.

In the bottom circuit, it connects to only the two MOSFET switches in the inverter.

So, the bottom circuit imposes half the capacitive load on the A signal.

This might be significant if the signal A connected to many such circuits.

The bottom line: when you find yourself needing a fast implementation for the AND/OR circuitry for a sum-of-products expression, try using the NAND/NAND implementation.

It'll be noticeably faster than using AND/OR.