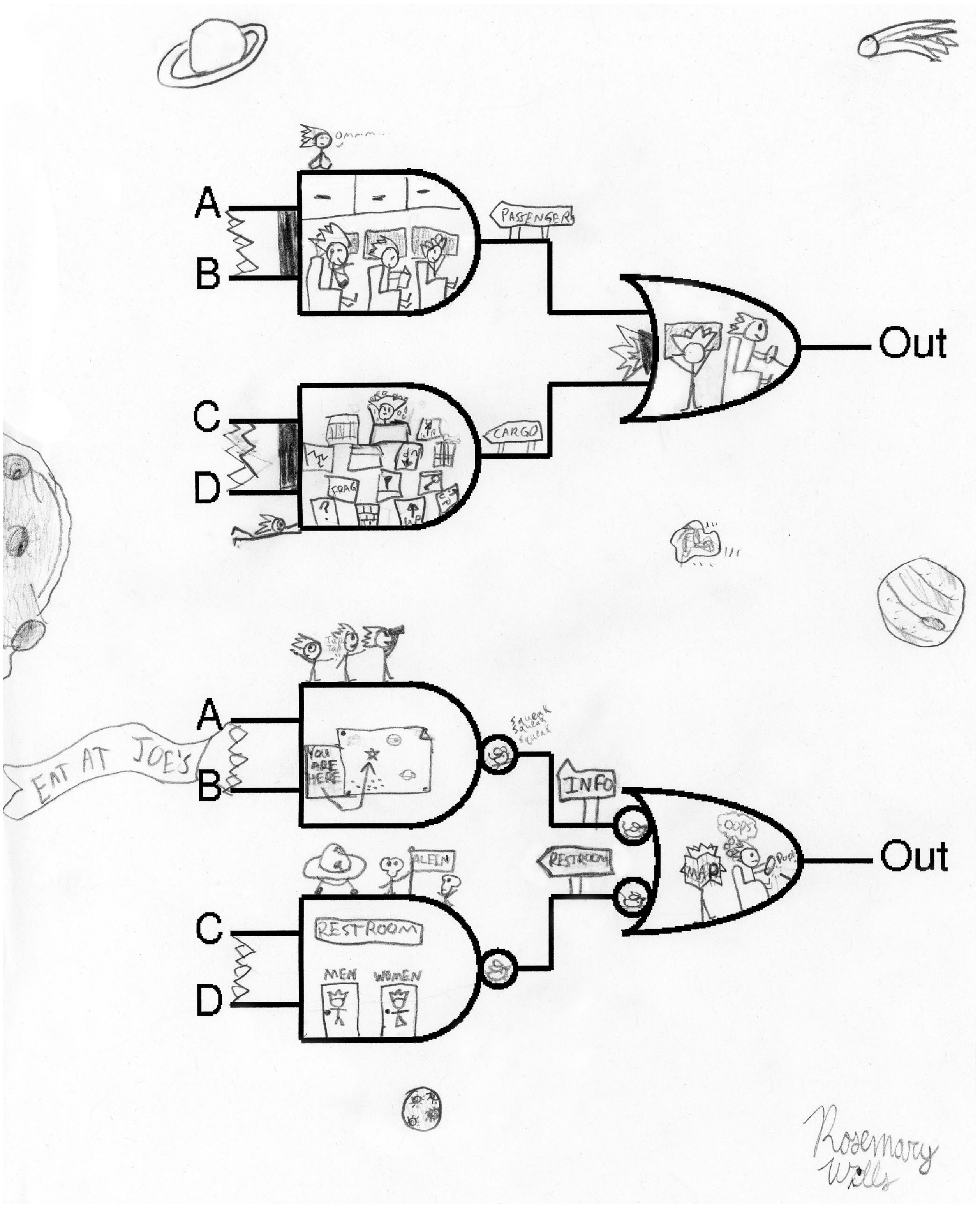
Designing Computer Systems

Gate Design



Designing Computer Systems

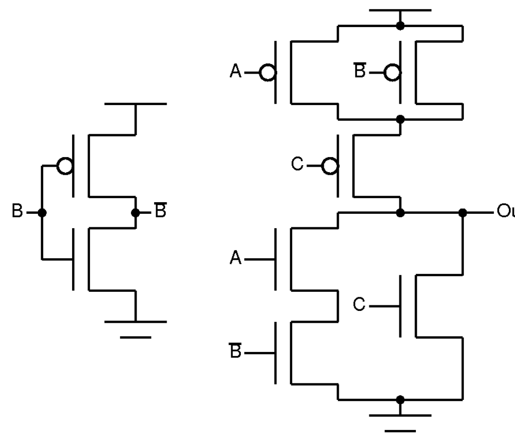
Gate Design

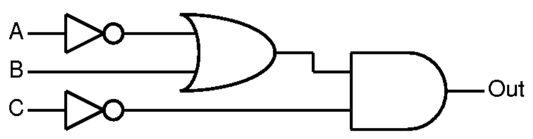
Logical functions that are specified in Boolean algebra, can be implemented with switches and wire. The resulting designs are often the fastest and most efficient implementations possible. But the time and effort required for design is often greater. And switch design requires the manipulating the desired expression so that only input variables are complemented (no big bars). Often after the design process, the desired expression is lost. Is there a way to implement a Boolean expression quickly, without distorting the expression?

Yes!

We can simplify the design process by using more powerful components. We'll work with *gates*, building blocks that match the logical operations in our expression. Wires still connect outputs to inputs. Data still is digital. In fact, we use switches to implement these new gate abstractions.

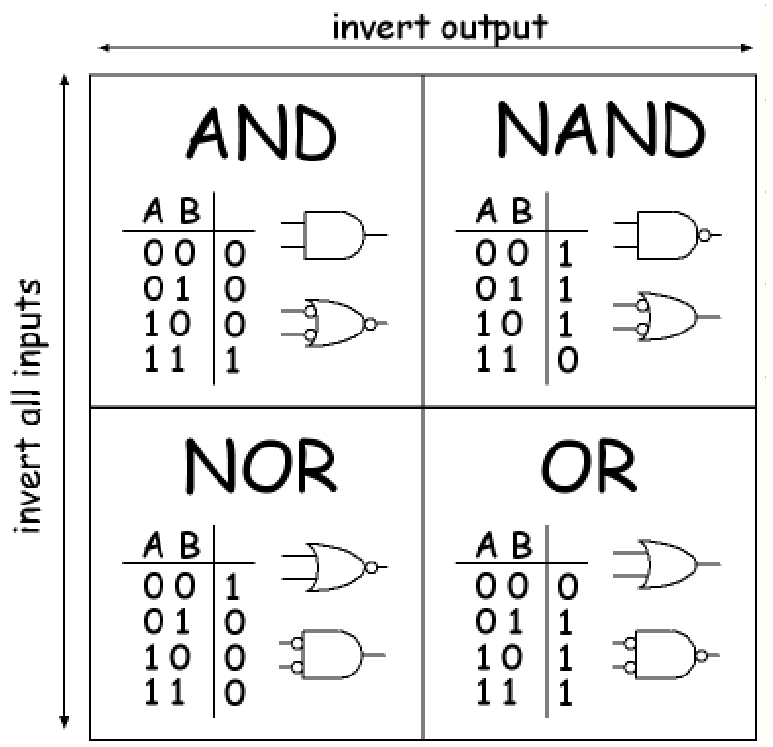
Suppose we want to implement the expression Out = $(\overline{A} + B) \cdot \overline{C}$. Using switches, details of the implementation technology (e.g., P-type switches are active low and pull high) are visible and affect the design. Using gates, technology details are hidden and the desired expression is easily discerned. Unfortunately this gate design is twice as slow and uses twice as many switches. Convenience has a cost!





Of course, gate design can be improved if the choice of implementation components is not tied to the desired expression. For CMOS technology, NAND and NOR gates require fewer switches than AND and OR. So in this example, the OR and AND gates can be replaced by NOR gates. Unfortunately, this requires DeMorgan transformations of the desired expression. This distorts the expression, increases design time, and increases the possibility for errors. Why can't we leave the expression alone?

We can. DeMorgan's square suggests that all gate types have two equivalent representations. One is built on an AND body. The other employs an OR body.



AND		NAND	
A B		A B	
0 0	0	0 0	1
0 1	0	0 1	1
1 0	0	1 0	1
1 1	1	1 1	0

NOR		OR	
A B		A B	
0 0	1	0 0	0
0 1	0	0 1	1
1 0	0	1 0	1
1 1	0	1 1	1

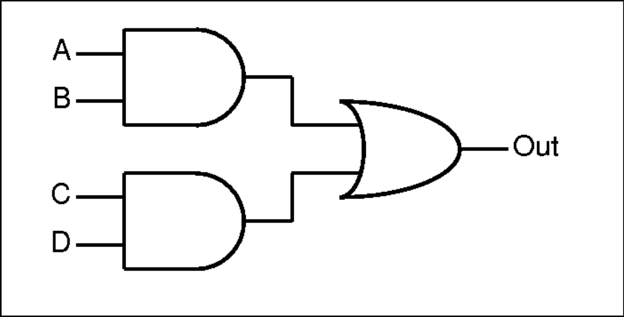
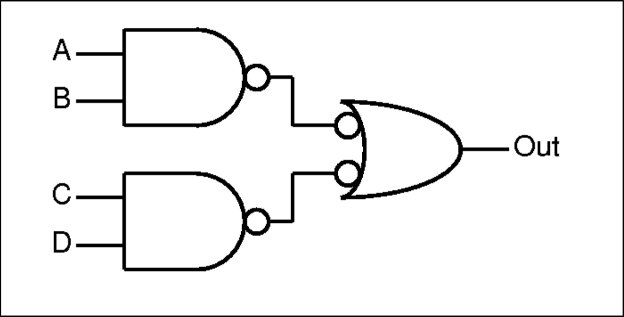
At first this duplicity may seem a complication. But it can be productively used to separate *specification* from *implementation*. Here's how.

When a desired expression is derived, AND and OR functions provide the relationship between binary variables. The choice of gate types can improve the implementation efficiency and performance. But it should not distort the meaning of the desired expression.

Since each gate function can be drawn with either an AND or OR body, a desired logical function can be realized using any gate type by simply adding a bubble to the inputs and/or output. Unfortunately, a bubble also changes the behavior by inverting the signal. But bubble pairs (bubbles at both ends of a wire) cancel out and the behavior is unchanged.

So we can draw a gate design using the logical functions in the desired expression. Then we can then add bubble pairs to define the implementation gate type without changing the gate body (i.e., distort the expression being captured).

Here's an example: Out = $A \cdot B + C \cdot D$

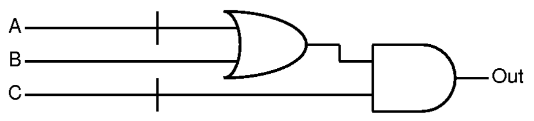
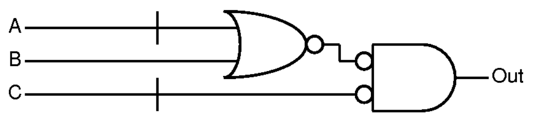
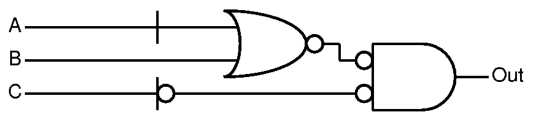
	
using AND and OR	using NAND

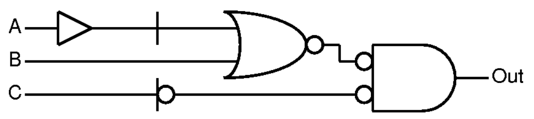
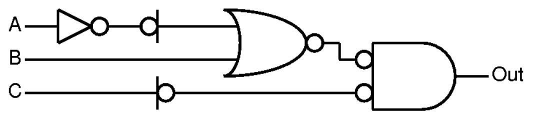
If we draw the circuit directly using AND and OR gates, the expression is clear. But the implementation cost is high (18 switches). If we preserve the gate bodies, but add bubble pairs, the behavior is unchanged. But the implementation cost is lowered (12 switches).

A bar over an input or subexpression indicates that an inversion is required. This bar is part of the desired expression and should be preserved along with gate bodies. But the implementation must include, in some way, the required inversion of the signal. Again, bubble pairs can help.

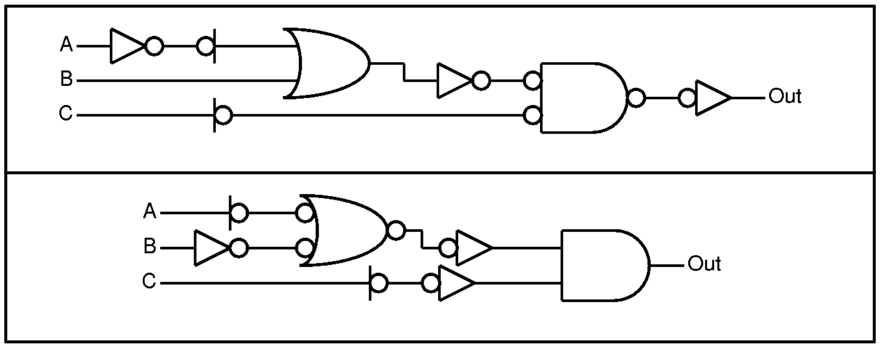
Let's add bars to our gate design, not as active devices, but as a notational reminder that a real signal inversion is needed. Then during implementation, we'll place exactly one bubble on the bar. Bubble pairs are added to change the implementation without changing the behavior. If one bubble in a bubble pair does not actually cause a real inversion (because it is a notation), the signal will be inverted (by the other bubble).

Consider the expression $Out = (\overline{A} + B) \cdot \overline{C}$.

	First let's draw the expression as a circuit. Bars are added where they appear in the expression. This is *not* an implementation.
	Now suppose we want to use NOR gates for the implementation. This is partially accomplished by adding a bubble pair between the gates. But the bubble on the lower input of the AND gate is unmatched.
	We can put the matching bubble on the bar, indicating that we really do want the inversion. The bubbled bar does not actually do anything; it's just notation. The bubble on the AND input does the required inversion.

	We also need a bubble on the A input. But we can't add the matching bubble to the OR gate body without changing its implementation. Instead let's add a buffer on the A input.
	Now we can add a bubble pair between the buffer and bar. The implementation gate type is NOR, all bubbles are matched, and all bars have exactly one bubble. This implementation is complete.

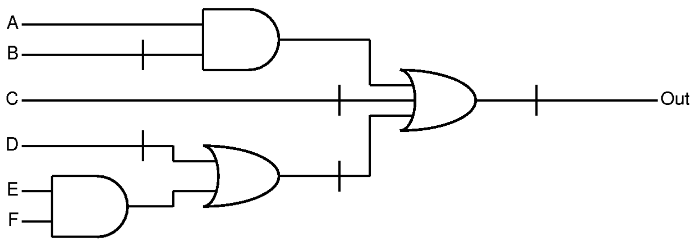
The gate implementation of this example requires ten switches. That's two more than the switch design. But it is six less that the original gate implementation. Note that by ignoring bubble pairs and buffers, we still see the desired expression, graphically displayed. Specification and implementation are now decoupled.



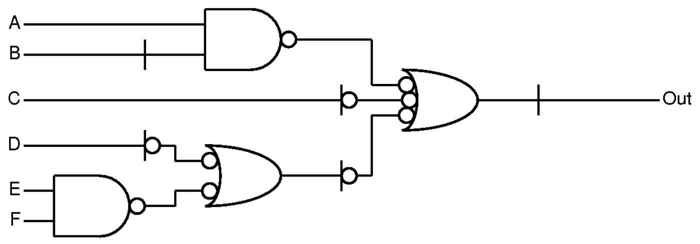
We can also implement the design using OR or AND gates. DeMorgan's equivalence allows any gate body to be implemented in any multi-input gate type. In CMOS technology, OR and AND implementations requires more switches (18 for this design).

Here's another example: $Out=\overline{A\cdot\overline{B}+\overline{C}+\overline{\overline{D}+E\cdot F}}$

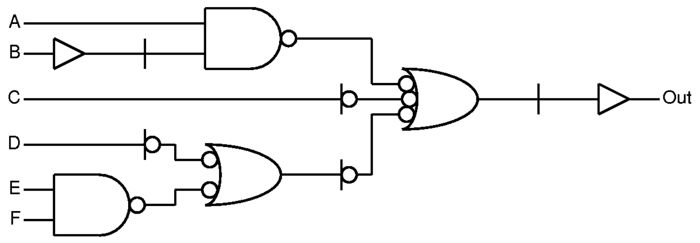
We start with the expression as a graph using gates and bars. It captures the function. But its not an implementation.



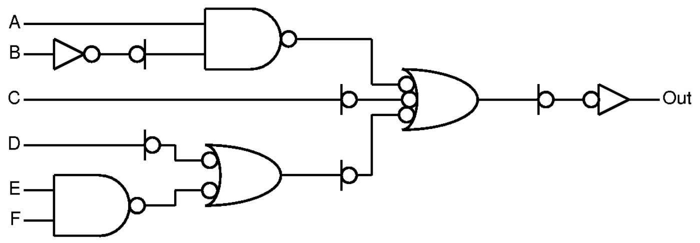
Now we select a good implementation gate. One doesn't always need to use one gate type for a design. The technology may favor an implementation approach. In CMOS, inverting gates (NAND and NOR) use fewer switches than non-inverting gates (AND and OR). In this case, we use NAND gates. Bubble pairs are added to gate bodies to transform the implementation.



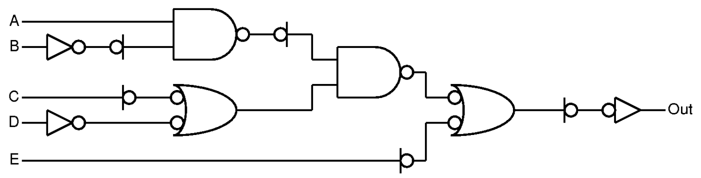
Buffers are added where bubbles pairs are still needed for bars.



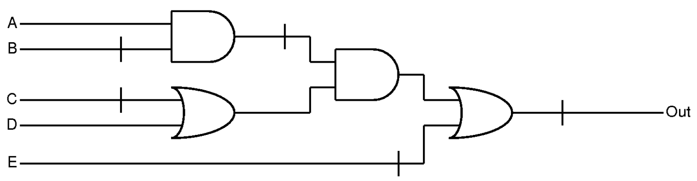
Finally bubbles pairs are added to complete the implementation.



Desired Expression: This gate design technique is called *mixed logic*. Its name is derived from the fact the implementation combines positive (active true) and negative (active false) logic. A key advantage is the ability to preserve the desired expression (i.e., the expression the designer specified) in an implementation. For example, the circuit below is built with NAND gates.

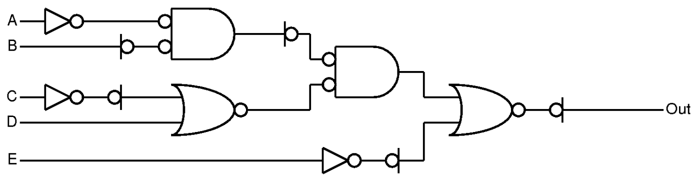


To see the desired expression, ignore the bubbles and buffers and read the expression from the gate bodies and bars.



This expression is $Out=\overline{\overline{A\cdot\overline{B}}\cdot(\overline{C}+D)+\overline{E}}$

If we wish to reimplement it, say using NOR gates, we just move around bubble pairs, adding and removing buffer bodies as needed.

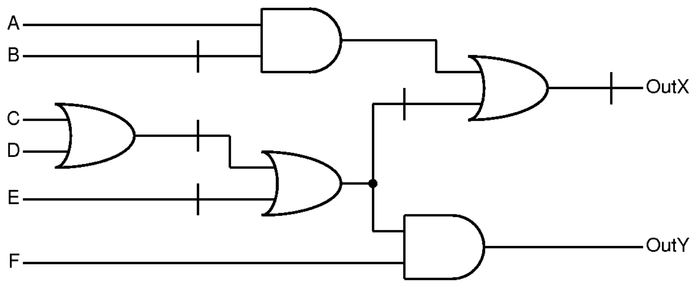


Note that the desired expression has not changed.

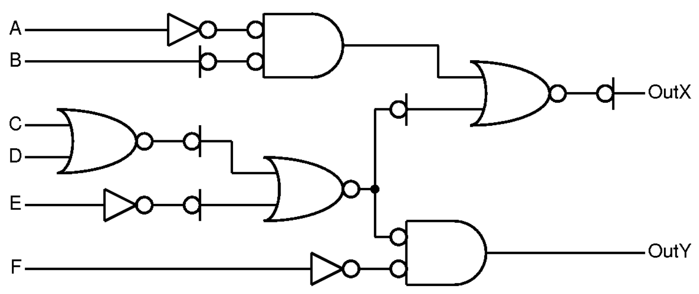
Common Subexpressions: Often in design, a logical expression is required for multiple outputs. It would be wasteful to build multiple copies. We can just use a computed value in multiple places. This is called *fanout* since a single gate output fans out to multiple gate inputs. Consider these two equations.

$$OutX=\overline{A\cdot\overline{B}+\overline{\overline{\overline{C+D}+\overline{E}}}} \qquad OutY=(\overline{C+D}+\overline{E})\cdot F$$

Both expressions require the subexpression $\overline{C+D}+\overline{E}$ so it can used in creating both outputs.



During implementation, here using NOR gates, special attention is needed for fan out connections. In order to ignore a bubble on an output, there must be a bubble on each input that uses it. The bubble pair on the output of the subexpression becomes a *bubble trio*.



Propagation Delay

When considering the speed of circuits, one must look at underlying technology – here, switches and wire. The two parameters that dominate delay are *resistance* and *capacitance*.

Resistance is an abundance of charge carriers. It is proportional to the availability of charge carriers brought in by the electronic field on the gate. It is proportional to the charge carrier mobility. Metals are a charge carrier gas. They have clouds of electrons that are easy to acquire. A semiconductor has more bound charge carriers that are harder to acquire and more difficult to move around with a field, leading to higher resistance to pull a node to a high voltage or to a low voltage.

The charge it takes to reach a high or low voltage is proportional to a node's *capacitance* C. Capacitance forms automatically when two insulated conductors are near one another separated by a dielectric material. The higher the dielectric constant, the higher capacitance. Dislike charges attract to form an electric field when the insulated conductive dislike charges appear on the conducting surfaces (for example, the polysilicon gate oxide on the switch).

The bigger the switch, the more charge carriers are needed to charge the switch voltage to the On level, which is a product of the switch resistance R and the gate capacitance C. RC is proportional to the propagation delay through the switch.

CompuCanvas models delay as unit delay, which assumes a fixed constant delay through each gate.

Energy

Energy is proportional to the product of induced voltage on a node and channel conductance, which is the inverse of the resistance through a conducting channel of a turned on switch. This resistance is proportional to the major charge carrier mobility. Conductors have an electron cloud of free electrons that can be easily moved by a field. Doped silicon has limited charge carrier mobility that limits conductance and energy.

Summary: Gate design of Boolean expression is a fast and clean alternative to switch design.

- Gate design is easier to understand than switches and is independent of implementation technology.
- Gate implementations often require more switches than direct switch implementations, but designs can still be optimized.
- DeMorgan's gate equivalence allows specification and implementation to be separated using mixed logic design.
- Mixed logic design also preserves the designer's desired expression.