The previous sections showed us how to build a circuit that computes a given sum-of-products expression.

An interesting question to ask is if we can implement the same functionality using fewer gates or smaller gates?

In other words is there an equivalent Boolean expression that involves fewer operations?

Boolean algebra has many identities that can be used to transform an expression into an equivalent, and hopefully smaller, expression.

The reduction identity in particular offers a transformation that simplifies an expression involving two variables and four operations into a single variable and no operations.

Let’s see how we might use that identity to simplify a sum-of-products expression.

Here’s the equation from the start of this chapter, involving 4 product terms.

We’ll use a variant of the reduction identity involving a Boolean expression alpha and a single variable A.

Looking at the product terms, the middle two offer an opportunity to apply the reduction identity if we let alpha be the expression (C AND B).

So we simplify the middle two product terms to just alpha, i.e., (C AND B), eliminating the variable A from this part of the expression.

Considering the now three product terms, we see that the first and last terms can also be reduced, this time letting alpha be the expression (NOT C and A).

Wow, this equivalent equation is much smaller!

Counting inversions and pair-wise operations, the original equation has 14 operations, while the simplified equation has 4 operations.

The simplified circuit would be much cheaper to build and have a smaller tPD in the bargain!

Doing this sort of Boolean simplification by hand is tedious and error-prone.

Just the sort of task a computer program could help with.

Such programs are in common use, but the computation needed to discover the smallest possible form for an expression grows faster than exponentially as the number of inputs increases.

So for larger equations, the programs use various heuristics to choose which simplifications to apply.

The results are quite good, but not necessarily optimal.

But it sure beats doing the simplification by hand!

Another way to think about simplification is by searching the truth table for “don’t-care” situations.

For example, look at the first and third rows of the original truth table on the left.

In both cases A is 0, C is 0, and the output Y is 0.

The only difference is the value of B, which we can then tell is irrelevant when both A and C are 0.

This gives us the first row of the truth table on the right, where we use X to indicate that the value of B doesn’t matter when A and C are both 0.

By comparing rows with the same value for Y, we can find other don’t-care situations.

The truth table with don’t-cares has only three rows where the output is 1.

And, in fact, the last row is redundant in the sense that the input combinations it matches (011 and 111) are covered by the second and fourth rows.

The product terms derived from rows two and four are exactly the product terms we found by applying the reduction identity.

Do we always want to use the simplest possible equation as the template for our circuits?

Seems like that would minimize the circuit cost and maximize performance, a good thing.

The simplified circuit is shown here.

Let’s look at how it performs when A is 1, B is 1, and C makes a transition from 1 to 0.

Before the transition, C is 1 and we can see from the annotated node values that it’s the bottom AND gate that’s causing the Y output to be 1.

When C transitions to 0, the bottom AND gate turns off and the top AND gate turns on, and, eventually the Y output becomes 1 again.

But the turning on of the top AND is delayed by the tPD of the inverter, so there’s a brief period of time where neither AND gate is on, and the output momentarily becomes 0.

This short blip in Y's value is called a glitch and it may result in short-lived changes on many node values as it propagates through other parts of the circuit.

All those changes consume power, so it would be good to avoid these sorts of glitches if we can.

If we include the third product term BA in our implementation, the circuit still computes the same long-term answer as before.

But now when A and B are both high, the output Y will be 1 independently of the value of the C input.

So the 1-to-0 transition on the C input doesn't cause a glitch on the Y output.

If you recall the last section of the previous chapter, the phrase we used to describe such circuits is "lenient".