Episode 5.01 – The Sum-of-Products Expression

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series, we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. Due to the computational nature of this episode, you might want to visit the transcript page found at intermation.com to download the episode worksheet.

In the design of digital systems, there are some standards that are regularly applied to combinational logic. Over time, design tools and programmable hardware have been developed to take advantage of these standards allowing for quick implementation of digital logic. In this episode, we're going to take take a look at the first of two representations of combinational logic: the sum-of-products expression. The sum-of-products, or SOP, expression is a form of digital logic where, aside from a possible inverter, all of the signals pass through exactly two layers of logic gates. This makes the circuit optimally fast. In addition, we're going to show how this circuit can be implemented entirely with NAND gates, and if you listened to Episode 4.04 – NAND, NOR, and Exclusive-NOR Logic, you heard that using current technology, the NAND gate is the quickest, cheapest, and smallest logic device we have.

Remember that AND and OR are just conjuctions. Looking at it this way, we can see that we've been using sum-of-product expressions our whole lives. For example, imagine Cullen won't eat pizza unless it's topped with mushroom and onion or melon and prosciutto. Putting aside any judgement of Cullen's unique palate, if you made a truth table with a nearly infinite number of rows describing every possible combination of pizza toppings, Cullen's table would only have ones in two rows: the row that has a one in the mushroom column and a one in the onion column while zeros are in all the other toppings columns and the row that has a one in the melon column and a one in the prosciutto column with zeros in all the other columns. Turns out that this is a sum-of-products expression.

Let's say that we misunderstood Cullen, and he actually said that he would eat any pizza as long as it had mushrooms, but no onions. This makes for a very different truth table. In this case, there would be a one in the output column for every row that had a one in the mushroom column and a zero in the onion column, regardless of the ones and zeros in the other topping columns. Believe it or not, this is still a sum-of-products expression.

Okay, that's enough about pizza. Let's talk about Boolean expressions. Basically, a sum-of-products expression identifies each set of conditions that when true make the expression true, and merges them using the OR operation. The term sum-of-products comes from the expression's form: a sum (OR) of one or more products (AND). In circuit form, an SOP expression takes the output of one or more AND gates and OR's them together with a single OR gate which generates the output. The inputs to the AND gates are input signals that are either inverted or not inverted. This means that the maximum succession of gates that any input signal passes through before the effects of its value reach the output are an inverter, an AND gate, and an OR gate. Since each gate causes a delay in the transition from input to output, and since the SOP format forces all signals to go through at most two gates (not counting the inverters), an SOP expression gives us predictable performance regardless of which input to the circuit changes.

Let's look at an SOP expression that uses four inputs: A·B·C·D + A·B·D + A·D (read A-bar AND B AND C AND D-bar OR-ed with A AND B-bar AND D OR-ed with A AND D-bar).

So what does this expression look like in a truth table? Remember that if any input to an OR gate is a one, the output is a one. Therefore, if we can identify the rows where each AND generates a one, we have identified all the ones in the final truth table. Let's start with the first product, A·B·C·D (read A-bar AND B AND C AND D-bar). The only time that this product outputs a one is if A is not ONE and B is one and C is one and D is not one. There is exactly one row in the truth table where this situation occurs, and that is in the seventh row of the four-input truth table where A is zero, B is one, C is one, and D is zero. Next, let's figure out where the second product generates a one in the truth table, in other words, when is A·B·D (read A AND B-bar AND D) equal to one? This product equals one when A is one and when B is zero and when D is one. But what about C? It isn't part of this expression. Since C is not part of this product, it can be whatever it wants to be. Therefore, this product adds a one to the truth table in two places: first, where A is a one, B is a zero, C is a zero, and D is a one, and second, where A is a one, B is a zero, C is a one, and D is a one. This happens in the tenth and the twelfth rows.

Now what about this last product, A·D (read A AND D-bar)? This product outputs a one if A is a one and D is not a one. There are two input terms missing from this product, B and C. That means that they can take on any value they want to as long as A equals a one and D equals a zero. There are four possible patterns of ones and zeros that B and C can take on: 0-0, 0-1, 1-0, and 1-1. Enumerating these four possible patterns along with A equals one and D equals zero gives us the four rows where this product places a one in the output column: 1-0-0-0, 1-0-1-0, 1-1-0-0, and 1-1-1-0.

Notice that when a product uses all of the circuit's inputs, it generates a single one in the output column of the SOP truth table. If one input is removed from the product, two rows have ones in the SOP truth table. Two inputs missing generates four ones in the truth table. Three inputs missing from a product generates 2^3 or 8 ones in the truth table, and so on.

Notice that there are no parentheses in an SOP expression since parentheses would necessitate additional levels of logic. This also means that an SOP expression cannot have more than one variable combined under a single inversion bar since an inversion bar covering multiple inputs is treated like parentheses in that everything under the bar must be processed before being inverted.
For example, A·B·C + A·C (read the inverse of the product A AND B then AND-ed with C OR-ed with A AND C-bar) may look like an SOP expression, but with the first term having both A and B under a single inverse, A and B must pass through a NAND gate before being AND-ed with C. This creates a third level of logic.

To fix this problem, we need to break up the NAND using DeMorgan's Theorem. This changes the inverse of the product A AND B to A-bar OR B-bar. We can then distribute the C across this OR to get A-bar AND C OR-ed with B-bar AND C. This expression, when OR-ed with the product A AND C-bar, is now considered to be in SOP format.

And now for the third part. There isn't a truth table around that cannot be represented with an SOP expression. To show this, we're going to make up a truth table and derive its corresponding SOP expression.

Before we do, let's talk about the AND operation. Remember that the truth table for an AND operation, regardless of the number of inputs, has exactly one row with a one for its output. All of the other rows have a zero output. It turns out that we can move this single one to any of the other rows by inverting one or more of the inputs.

For example, the truth table for A AND B AND C has a single one in the bottom or eighth row where A is one, B is one, and C is one. If we invert A in our expression to get A-bar AND B AND C, then our single one moves to the row where A is not one, in other words, it's zero, B is one, and C is one too. Now we have a truth table with a one in the fourth row where the inputs are 0-1-1. If, on the other hand, we want to move that single one to the fifth row where A is one, B is zero, and C is zero, we would need to invert B and invert C to get the expression A AND B-bar AND C-bar. For each of these products, we have a single one in the output column.

Now think about the OR gate. The output of an OR gate is a one if any of the inputs are one. Therefore, when we OR two or more products together, a one appears in the output column for each of the products. For example, if we OR-ed together the three products described earlier to get A AND B AND C OR A-bar AND B AND C OR A AND B-bar AND C-bar, the resulting truth table would have a one in the eighth row, a one in the fourth row, and a one in the fifth row with zeros everywhere else.

It turns out that we can reverse this process to turn a truth table into an SOP expression. All we need to do is identify the rows where there are ones in the truth table, create the unique product for each of those rows using inverses over the inputs to move the active row from the bottom row to the desired row, and then OR the products together. Note that each product in this resulting SOP expression will use all of the variables for inputs, in other words, it won't be simplified. No worries though, we can simplify the expression using the tools described in earlier episodes while still keeping the expression in a valid sum-of-products form. Later in this series, we will present a tool that allows us to go from truth table directly to the most simplified SOP expression.

Now for an example. Let's say we want to derive the SOP expression for a three-input truth table with ones in the second, fourth, fifth, and seventh rows. Since there are four ones in our truth table, there will be four products, all of which will include the inputs A, B, and C, either inverted or non-inverted. First, let's create a product that generates a one in the second row where A is zero, B is zero, and C is one. This product needs to go active when A is not one and B is not one and C is one. That means A and B must be inverted while C is left non-inverted. This gives us the product A-bar AND B-bar AND C. By itself, this product puts a one in the second row.

To place a one in the fourth row, where A is zero, B is one, and C is one, we need to have a product where A is inverted while B and C are not inverted. This gives us A-bar AND B AND C and places a one in the fourth row. To place a one in the fifth row, where A is one, B is zero, and C is zero, our product inverts B and inverts C while leaving A non-inverted. The resulting product, A AND B-bar AND C-bar, places a one in the fifth row, and only the fifth row. Lastly, to place a one in the seventh row, where A and B are one and C is a zero, we need a product with only C inverted. This gives us a product A AND B AND C-bar, which generates our one in the seventh row.

Now that we have a product to generate a one in the second row, a product to generate a one in the fourth row, a product to generate a one in the fifth row, and a product to generate a one in the seventh row, all we need to do is OR them together to create a Boolean expression for our truth table. This gives

us A·B·C + A·B·C + A·B·C + A·B·C (read A-bar AND B-bar AND C OR-ed with A-bar AND B AND C OR-ed with A AND B-bar AND C-bar OR-ed with A AND B AND C-bar).

This expression can be simplified a bit. Simplification of an SOP expression usually works like this. Identify two products that differ only by an inverse on one of the inputs. Using the distributive law, pull all the inputs out as a common product from each of these products so that we have a product AND-ed across the sum of an input OR-ed with its inverse. Anything OR-ed with its inverse equals one, and that one drops out of the product due to the identity law for the AND operation. For example, the first and second products both contain A-bar AND C. Pull this out of the first two products using the distributive law, and they become A·C·(B· + B) (read A-bar AND C AND-ed with the sum of B-bar OR B). B-bar OR B simplifies to one leaving A·C·1 (read A-bar AND C AND one). Anything AND-ed with one is itself, which leaves us with A·C (read A-bar AND C).

A similar sequence of simplifications can be performed on the last two products which share the term A AND C-bar. Pull this out of the last two products using the distributive law, and they become A·C·(B· + B) (read A AND C-bar AND-ed with the sum of B-bar OR B). B-bar OR B again simplifies to one using the inverse law for the OR operation leaving A·C·1 (read A AND C-bar AND one). Anything AND-ed with one is itself, which leaves us with A·C (read A AND C-bar).

This means we have taken the truth table described above and created the simplified SOP expression A·C + A·C (read A-bar AND C OR A AND C-bar).

And by the way, I can hear some of you asking why we're bothering to come up with a process to go from the truth table to the Boolean expression. Well, it's quite simple. When designing digital logic, typically we have a requirement that starts with the truth table. For example, assume we have four bits, each of which turns on a different bank of lights, and we need to turn on a cooling fan whenever three or more banks of lights are turned on. That would give us a truth table with ones in the eighth row (0-1-1-1), the twelfth row (1-0-1-1), the fourteenth row (1-1-0-1), the fifteenth row (1-1-1-0), and the sixteenth or bottom row (1-1-1-1).

This would give us the SOP expression A·B·C·D + A·B·C·D + A·B·C·D + A·B·C·D + A·B·C·D (read A-bar AND B AND C AND D OR-ed with A AND B-bar AND C AND D OR-ed with A AND B AND C-bar AND D OR-ed with A AND B AND C AND D-bar OR-ed with A AND B AND C AND D). I'll leave the simplification up to you.
In our next episode, we're going to show how any sum-of-product expression can be implemented entirely with NAND gates. For episode transcripts, worksheets, links, or other podcast notes, please visit us at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages. Until the next episode remember that while the scope of what makes a computer is immense, it's all just ones and zeros.