

Forum: [Hack Forums](#)

Thread: [Boolean Algebra assistance needed](#)

Kalvaxas - June 8, 2023, 10:32 pm

I understand the NOT, AND, and OR operators, and I'm starting to understand truth tables in terms of each variable in the equation will have several different combinations in order to go through all possible combinations. I understand that the TRUE/FALSE variables in the proceeding columns are based on the equation, and the final column is the solution... What I'm not understanding is how to get to that solution? Like, how do you pick apart the equation? Is there an order of operations to follow so to speak? Like how in regular math you have order of operations?

In terms of assembling a Karnaugh map - I'm fully lost with my only understanding being that the result of the Karnaugh map is the same result as it would be if you did a truth table. So therefore, what is the reasoning behind a Karnaugh map, and...why does it even exist?

Faith Connors - June 9, 2023, 12:00 am

pardon the image size, but i just made couple short notes on the subject.

we're starting off with one of the first digital logic circuits you will probably have to make in your digital systems class: the full adder

what makes it a bit wonky is that the full-adder has two outputs: S and Cout
the inputs are A and B, two digits (bits) and a third bit, Cin

C stands for carry.

the digits A and B can be 0 or one.

if they are both 0, then the output of the circuit is zero.

if one of them is 1, then the result is still 1 ($1 + 0 = 1$)

but if they are both one, then they would add up to two. ($1 + 1 = 2$)

but of course, we can't express 2 as a binary number without another bit, so we use the carry bit to help us in outputting 10 (2 in binary)

the full logic equation for the circuit is:

Code $S = (A \oplus B) \oplus C$

$Cout = (AB) + ((A \oplus B) \cdot C)$ i'm sure you can work out the order of operations here.

usually when you get a logic equation, you will see the order by the nature of bracketing everything.

but assuming there's no bracketing for some reason, there are [rules to follow](#)

enough with the explanation of the circuit.

i only wrote it because i'm assuming you probably havent fully built one yet in your logic/digital systems class.

so what about kmaps?

well, the kmap is just a 2d truth table.

where a truth table is "2d" because it has expressions in one dimension and resulting output values in the other, a kmap has inputs in both dimensions, and between them lies the overarching result.

now, why do you care?

well, say on an exam you get a question kinda like this:

"here's a truth table of 4 inputs and the overall output of those 4 inputs. there's 16 different possible outcomes, but we're not going to tell you their intermediate steps. you need to figure out what the logic equation behind it, and then you need to draw it."

yeah, your prof is a dick. it can take a while to actually draw everything out, even if you have mastered kmaps and can come up with a somewhat expanded logic equation.

oh, the logic equation?

well, consider the truth table of the full-adder i drew.

it doesn't include any of those intermediate steps in the full equation.

partially because there are too many inputs and outputs for me to bother, partially because my page won't have enough space.

but mostly because i wanted to demonstrate the kmap method of generating logic.

in digital logic, we can represent any logic equation with a generalized, expanded form called the "minterms".

these are the minimum conditions that need to be met in order to generate an output of TRUE/1.

on the right side of the page, the first arrow is the truth table in/out to come up with the minterms (every output for Cout that is equal to 1, or TRUE.)

Code m =

NOT-A,B,Cin +

A,NOT-B,Cin +

A,B,NOT-Cin +

A,B,Cin

not bad, as you can see, for every 1 in the output of Cout in the table, i put all the input values (A, B, and Cin) together to form one AND expression.

if any one of those conditions is met, the overall OR expression will result in TRUE.

make sense?

okay, that's fine, but that's a lot of writing, and that's still really far from the equation i put above.

we can do better!

here's how the kmap works:

on the horizontal axis (specifically on the top), i wrote out AB and have 00, 01, 11, 10 for each column to indicate if A or B are 0 or 1.

on the vertical axis, i have Cin, which is either 0 or 1.

in each box, i put the corresponding output.

(e.g., for all three being off, the top left corner, i put a 0 because NOT-A, NOT-B, NOT-Cin returns false.

eg., for all three being on, the bottom right corner, i put a 1 because A, B, Cin returns true.)

this is where it gets interesting.

notice how all the 1's are close together.

we can actually group them.

you see those little brackets i put at the bottom of the kmap? they are a simplified equation for when the kmap is 1. the bottom middle two 1's are equal to B and Cin, as you can see A changes from 0 to 1 and has no influence on the output.

there's also AB for the column of two where the Cin value doesn't make a difference, and A and Cin where B doesn't make a difference.

so, we can put all three of these equations together, and we have a new minterms equation:

Code m =

A,B +

A,Cin +

B,Cin

if you were to draw out that circuit and draw the truth table for it, you would find that you get the same result as the equation above.

but it's much simpler than the equation we got from the truth table minterms.

from there (or from the truth table minterms,) you can use all your fancy logic rules like demorgan's to get a fully simplified equation.

but, once you have that equation, you can draw out that diagram and impress your professor on the exam.

so basically, that's why we use kmaps.

they're way faster to draw out than a truth table, and they facilitate generating a minterms equation

order of operations has rules, but most of the time you'll see brackets in the equation.

the order of operations is also very apparent in logic diagrams.

trashy - June 9, 2023, 2:05 am

Great explanation by part of [Faith Connors](#)

Imma add something more in theory than in circuits like he did.

(Jun 8, 2023 10:32 PM)?Sasuke Wrote: [\[CITE\]](#) I understand that the TRUE/FALSE variables in the proceeding columns are based on the equation, and the final column is the solution... What I'm not understanding is how to get to that solution? Like, how do you pick apart the equation? Is there an order of operations to follow so to speak? Like how in regular math you have order of operations?

So, you will be normally given a Function. Lets call it F. If you recieve an scheme or XORs, NANDS, and stuff, follow the first post and get that function. No matter what's your function, It could be reduced, Always. I recommend you to study before boolean and morgan's properties:

Now, the easy way is expressing everything as the sum of multiplications. You could obtain the called minterms, but in this case, we'll work with the function:

$F = ABC + B(!C) + AB$, see that we have a function expressed as sums of multiplications, well those multiplications are values that makes the function have "1" as solution. So you go part by part writing those at the Truth Table. If, for example, you have AB and missing C, you can express it as $AB = AB(!C) + ABC$. Because of the complement and distributive rule. Those are pure maths, so we get the values that makes $F = 1$:

And now you just make the truth table with all the possible values of A, B and C:

Hope it helps you aswell.?

That's like a more basic part than Faith Connors did, but it may help you to get concepts. It depends on how is your knowledge about Boolean, I guess this post it's like the "Start of" Boolean.

Faith Connors - June 9, 2023, 11:48 am

yeah thats a better theory explanation above i admit

i usually default to digital systems bc thats where i got the most "experience" with formal logic in my studies, but if you're learning it in philosophy, discrete math, or even in later-year linguistics, then the same rules still apply, but the terminology and symbolry might be slightly different.

it all boils down to:

1. generate a generic equation from the truth table or kmap (the kmap will often be "more" simplified)
2. use the rules of logic (e.g., distributive, commutative, demorgans, etc) to simplify the above equation
3. once it's fully simplified, then you can draw it all out and test it.

one more interesting property of sum-of-products/minterms notation is that you can use it to represent what a circuit would look like if you only used NAND-gates.

<https://www.asic-world.com/digital/gates3.html>

my professor absolutely loved asking questions like this, so it might be a helpful resource if you are

taking a digital systems class and they constantly ask you about SOP/NAND on your labs, reports, exams, etc.

Hackers - June 10, 2023, 7:32 am

If you are looking for a simpler way to think about it, it's like "Does this statement / equation = true or false?"

"My name is nodes" - true

"my name is bobbles" - false

true: $2 + 2 = 4$

true: $4 - 2 = 2$

false: $4 + 2 = 2$

So what you could do is something like this:

let foo = $(2 + 2) == 4$;

foo is a boolean equal to true, because it is the result of the evaluation. Because $2+2$ does = 4, foo = true.

let blah = $(2 + 3) == 4$;

blah is a boolean equal to false, because it is the result of the evaluation. Because $2 + 3$ does not = 4, blah = false.

When doing boolean algebra, you might see something like this:

let bank = { value: 105 }, stock = { value: 99 };

let stockCheap = stock.value < 100;

let haveFunds = bank.value >= stock.value;

let buy = stockCheap && haveFunds;

let sell = stockCheap == false || !haveFunds;

buy is a boolean equal to true because stockCheap is true, and haveFunds is true.

These are all very simple examples, but I wanted to give an example of equations that result in booleans and then how you can combine those results to create new equations. Not sure if this is what you are looking for, but hope it helps.

as far as and / or / not, I think mostly of SQL but obviously most languages have and/&& or/|| and NOT/NOT IN/etc commands in some way. I'm not 100% sure of the algebra syntax you're looking for and if you're mostly looking for general math syntax or for a specific language, but I imagine it's pretty similar.

As for order of operations, usually it's going to be in PEMDAS format, but it depends on the specific tech. Things can get wonky.

Kalvaxas - June 12, 2023, 7:08 pm

(Jun 9, 2023 12:00 AM)Faith Connors Wrote: [\[CITE\]](#) pardon the image size, but i just made couple short notes on the subject.

we're starting off with one of the first digital logic circuits you will probably have to make in your

digital systems class: the full adder

what makes it a bit wonky is that the full-adder has two outputs: S and Cout
the inputs are A and B, two digits (bits) and a third bit, Cin

C stands for carry.

the digits A and B can be 0 or one.

if they are both 0, then the output of the circuit is zero.

if one of them is 1, then the result is still 1 ($1 + 0 = 1$)

but if they are both one, then they would add up to two. ($1 + 1 = 2$)

but of course, we can't express 2 as a binary number without another bit, so we use the carry bit to help us in outputting 10 (2 in binary)

the full logic equation for the circuit is:

Code $S = (A \oplus B) \oplus C$

$Cout = (AB) + ((A \oplus B) \cdot C)$ i'm sure you can work out the order of operations here.

usually when you get a logic equation, you will see the order by the nature of bracketing everything.

but assuming there's no bracketing for some reason, there are [rules to follow](#)

enough with the explanation of the circuit.

i only wrote it because i'm assuming you probably haven't fully built one yet in your logic/digital systems class.

so what about kmaps?

well, the kmap is just a 2d truth table.

where a truth table is "2d" because it has expressions in one dimension and resulting output values in the other, a kmap has inputs in both dimensions, and between them lies the overarching result.

now, why do you care?

well, say on an exam you get a question kinda like this:

"here's a truth table of 4 inputs and the overall output of those 4 inputs. there's 16 different possible outcomes, but we're not going to tell you their intermediate steps. you need to figure out what the logic equation behind it, and then you need to draw it."

yeah, your prof is a dick. it can take a while to actually draw everything out, even if you have mastered kmaps and can come up with a somewhat expanded logic equation.

oh, the logic equation?

well, consider the truth table of the full-adder i drew.

it doesn't include any of those intermediate steps in the full equation.

partially because there are too many inputs and outputs for me to bother, partially because my page won't have enough space.

but mostly because i wanted to demonstrate the kmap method of generating logic.

in digital logic, we can represent any logic equation with a generalized, expanded form called the "minterms".

these are the minimum conditions that need to be met in order to generate an output of TRUE/1.

on the right side of the page, the first arrow is the truth table in/out to come up with the minterms (every output for Cout that is equal to 1, or TRUE.)

Code m =

NOT-A,B,Cin +

A,NOT-B,Cin +

A,B,NOT-Cin +

A,B,Cin

not bad, as you can see, for every 1 in the output of Cout in the table, i put all the input values (A, B, and Cin) together to form one AND expression.

if any one of those conditions is met, the overall OR expression will result in TRUE.

make sense?

okay, that's fine, but that's a lot of writing, and that's still really far from the equation i put above.

we can do better!

here's how the kmap works:

on the horizontal axis (specifically on the top), i wrote out AB and have 00, 01, 11, 10 for each column to indicate if A or B are 0 or 1.

on the vertical axis, i have Cin, which is either 0 or 1.

in each box, i put the corresponding output.

(e.g., for all three being off, the top left corner, i put a 0 because NOT-A, NOT-B, NOT-Cin returns false.

eg., for all three being on, the bottom right corner, i put a 1 because A, B, Cin returns true.)

this is where it gets interesting.

notice how all the 1's are close together.

we can actually group them.

you see those little brackets i put at the bottom of the kmap? they are a simplified equation for when the kmap is 1. the bottom middle two 1's are equal to B and Cin, as you can see A changes from 0 to 1 and has no influence on the output.

there's also AB for the column of two where the Cin value doesn't make a difference, and A and Cin where B doesn't make a difference.

so, we can put all three of these equations together, and we have a new minterms equation:

Code m =

A,B +

A,Cin +

B,Cin

if you were to draw out that circuit and draw the truth table for it, you would find that you get the same result as the equation above.

but it's much simpler than the equation we got from the truth table minterms.

from there (or from the truth table minterms,) you can use all your fancy logic rules like demorgan's to get a fully simplified equation.

but, once you have that equation, you can draw out that diagram and impress your professor on the exam.

so basically, that's why we use kmaps.

they're way faster to draw out than a truth table, and they facilitate generating a minterms equation

order of operations has rules, but most of the time you'll see brackets in the equation.

the order of operations is also very apparent in logic diagrams.

(Jun 9, 2023 02:05 AM)trashy Wrote: [\[CITE\]](#) Great explanation by part of [Faith Connors](#)

Imma add something more in theory than in circuits like he did.

So, you will be normally given a Function. Lets call it F. If you recieve an scheme or XORs, NANDS, and stuff, follow the first post and get that function. No matter what's your function, It could be reduced, Always. I recommend you to study before boolean and morgan's properties:

Now, the easy way is expressing everything as the sum of multiplications. You could obtain the called minterms, but in this case, we'll work with the function:

$F = ABC + B(!C) + AB$, see that we have a function expressed as sums of multiplications, well those multiplications are values that makes the function have "1" as solution. So you go part by part writing those at the Truth Table. If, for example, you have AB and missing C, you can express it as $AB = AB(!C) + ABC$. Because of the complement and distributive rule. Those are pure maths, so we get the values that makes $F = 1$:

And now you just make the truth table with all the possible values of A, B and C:

Hope it helps you aswell.?

That's like a more basic part than Faith Conors did, but it may help you to get concepts. It depends on how is your knowledge about Boolean, I guess this post it's like the "Start of" Boolean.

(Jun 10, 2023 07:32 AM)nodes Wrote: [\[CITE\]](#) If you are looking for a simpler way to think about it, it's like "Does this statement / equation = true or false?"

"My name is nodes" - true

"my name is bobbles" - false

true: $2 + 2 = 4$

true: $4 - 2 = 2$

false: $4 + 2 = 2$

So what you could do is something like this:

let foo = $(2 + 2) == 4$;

foo is a boolean equal to true, because it is the result of the evaluation. Because $2+2$ does = 4, foo = true.

let blah = $(2 + 3) == 4$;

blah is a boolean equal to false, because it is the result of the evaluation. Because $2 + 3$ does not = 4, blah = false.

When doing boolean algebra, you might see something like this:

let bank = { value: 105 }, stock = { value: 99 };

let stockCheap = stock.value < 100;

let haveFunds = bank.value >= stock.value;

let buy = stockCheap && haveFunds;

let sell = stockCheap == false || !haveFunds;

buy is a boolean equal to true because stockCheap is true, and haveFunds is true.

These are all very simple examples, but I wanted to give an example of equations that result in booleans and then how you can combine those results to create new equations. Not sure if this is what you are looking for, but hope it helps.

as far as and / or / not, I think mostly of SQL but obviously most languages have and/&& or/|| and NOT/NOT IN/etc commands in some way. I'm not 100% sure of the algebra syntax you're looking for and if you're mostly looking for general math syntax or for a specific language, but I imagine it's pretty similar.

As for order of operations, usually it's going to be in PEMDAS format, but it depends on the specific tech. Things can get wonky.

Thank you all so much for the help! I have a better understanding to the point where I can actually study now and not feel lost at every corner. I'll be honest, I truthfully didn't know there were different properties for boolean algebra. I feel so stupid not knowing them either because some of them I remember quite well from regular algebra, like distributive. Big thanks to Faith as well. Your explanations are the reason I'm finally able to figure things out now beyond just how the operators work.

