# Digital Logic Analysis - Boolean Algebra, Logic Gates, and Binary Arithmetic

## Intro

### ILOs:

1. Boolean Algebra & Logic Gates
   1. Solve for the truth table of a Boolean algebra expression or logic gate circuit
   2. Convert logic gate circuits into Boolean algebra expressions & vice versa
   3. Binary
      1. Convert between binary and decimal representations
      2. Complete binary addition
      3. Interpret binary arithmetic using both unsigned and signed (2s complement), recognizing carry vs. overflow
2. Digital Logic Design
   1. Understand minimum SOP & POS forms of Boolean expressions
   2. Understand minterms and maxterms
   3. Design efficient gate implementations from truth tables using Karnaugh Mapping
   4. Convert a gate implementation or Boolean expression into an equivalent gate implementation; entirely NOR or entirely NAND

### Topic 6 Videos:

(Note: This was split into two topics prior to 2020, Topic #6 and Topic #7)

1. Binary Arithmetic: <https://www.youtube.com/watch?v=V8z12ie3-eE&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=26>
2. Logic Gate Analysis: <https://www.youtube.com/watch?v=xclVjoDY2Cs&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=27>
3. Boolean Algebra Extra Problem: <https://www.youtube.com/watch?v=JNR0_6AIshA&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=28>
4. Logic Gates Analytical & Multisim: <https://www.youtube.com/watch?v=Zq0eF1ZWFh0&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=29>
5. K-Mapping:
   1. "Sample Lab 7": <https://www.youtube.com/watch?v=hXzW8ZCBL-o&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=30>
   2. Extra Problem: <https://www.youtube.com/watch?v=g0lkaISb8uI&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=32>
6. Lab Skills:
   1. Digital Logic Intro Lab ("Sample Lab 6"): <https://www.youtube.com/watch?v=9ledlSwDZpg&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=25>
   2. Implementing Logic Design ("Sample Lab 7"): <https://www.youtube.com/watch?v=9fEWRogJFeM&list=PLhbHWgMknRJT_eKLFXB843NkaNHfJ37Pw&index=31>
7. Hantek Lab Modification: NA

### Deliverables for this week

Note: As always, your full objective for this topic is to review these notes, the videos, practice problems, live class sessions and forum content, then to write-up creating and tri-solving a variation problem of the week's topic that demonstrates you've mastered the content. With that all still in mind, following are some specific guidelines & tips for this week.

Tasks for your write-up:

1. Choose an example desired logic function that takes 4 binary inputs to implement like in the K-mapping videos (i.e., 4 input bits and 1 output bit)
2. Show the k-mapping process you used to come up with a logic expression you can implement with gates.
   1. **Note:** like described in the K-mapping Sample Lab (SL7, now Topic 6.2), you should do this to get both the SoP and the PoS simplified expressions, then choose either of these to work with and simplify in step 3 (possibly choosing both; i.e., SoP for the AND+OR+NOT version and PoS for the NAND version or vice-versa).
3. Use Boolean algebra to then implement this two different ways:
   1. Using a combination of AND gates, OR gates, and INVERTERs, and
   2. Using *only* NAND gates.
4. For each of the two gate implementations from steps 3a and 3b:
   1. Analytically check that it works by generating a truth table from the gates and comparing with the desired behaviour for your logic function,
   2. Build the circuit in multisim and demonstrate that it produces the desired functionality, and
   3. Physically implement the circuit and show that it produces the desired functionality.

***Note:*** In both multisim and your physical build, you should use LEDs to show the output and take a screenshot & image of all 16 input states that shows both input states and output states and verifies your truth tables.

***Hint:*** to save on the screenshots, you can optionally set up two versions of the circuit simultaneously; one for each gate implementation (i.e., the version with ANDs, ORs, and NOT gates, and the version with just NAND gates)

***Hint 2:*** optionally, you can replace the 16 multisim screenshots AND/OR the 16 physical implementation pictures with short (<2 minute) videos that demonstrate the results like the screenshots would. This will be required for H7 and the design project, so is good practice, and can save time here too. In that case, you should still include one screenshot each of the multisim and physical implementation in the report to explain the circuit setups and refer to the videos in the report. Make sure the image of the physical implementation is marked up to explain which component, input, output, etc. is which.

**→ Note: make sure to post the video via a visible link (i.e., streams video link, youtube link, Teams uploaded video link, etc.; something where we don't need to download the actual video file to play it)**

***Note 2:*** Your desired logic should ideally be some sort of nontrivial word problem that gives at least some motivation for why the output should depend on the input the way it does rather than an arbitrary set of values in a truth table.

***Note 3:*** You are allowed to have don't care conditions but similarly should motivate them, e.g., by setting up a situation where some input values can't happen or make the output unimportant.

***Note 4:*** It's OK if the logic function isn't too tough, as long as it can still show off your skills. You should have to use at least 5 chips *total* (e.g., two 74HC00 chips for the NAND implementation, then one 74HC04, one 74HC08, and one 74HC32 for the AND, OR, and NOT implementation.) i.e., if you need at least 5 NAND gates for the NAND implementation you're probably OK.

***Note 5:*** Your implementations should not depend on the outputs from the other implementation (i.e., if your NAND implementation needs to invert something, it should use a NAND gate rather than an empty NOT gate from the other implementation). This way you can independently see how much real estate is required for each implementation and compare them, and test their functions independently too.

## Digital Logic Circuits

### Analog and Digital Signals

Analog signals may have any continuous value but digital signals are discrete; specifically, a single bit of information can only have one of two binary values, 0 and 1 (AKA false and true).

### The Binary Number System; signed & unsigned numbers

Binary numbers are numbers written in base-two rather than base-ten; i.e., the number eleven could be written



(the subscript tells you the base).

This is the case for an *unsigned* number (i.e., no distinction of positive sign vs. negative sign).

Great video explaining Binary numbers: <https://youtu.be/1GSjbWt0c9M?list=PL8dPuuaLjXtNlUrzyH5r6jN9ulIgZBpdo>

The **ones complement** of *a* is []

e.g., the ones complement of 0110 is (1111) - 0110 = 1001. This inverts (i.e., takes the "complement" of) every bit.

The **twos complement** of a is [].

e.g., the twos complement of 0110 is 10000 - 0110 = 1010. Note: Taking the twos complement is equivalent to **inverting every bit and then adding 1**: 0110 → 1001+1 = 1010.

An **unsigned** binary number has the most significant bit (MSB) indicating a power of 2 just like all the other bits, but in a **signed** binary number the MSB indicates whether the number is negative. This means you need to know what type of number (unsigned vs. signed) something represents in binary before you can tell which integer it corresponds to.

e.g.,

 as an *unsigned* number means 

1001 as a *signed* number means -(twos complement(001)) = -(110+1) = - 111 = -7

(or, )

Twos complement is used for negative numbers because it lets you use the same addition circuits no matter what kind of numbers you're representing; e.g.,

 regardless of whether these are unsigned or signed numbers; if unsigned it says 

but if signed it says 

Table of 4-bit numbers and their meanings:

|  |  |  |
| --- | --- | --- |
| Binary | Unsigned Interpretation | Signed (2s complement) Interpretation |
| 0111 | 7 | 7 |
| 0110 | 6 | 6 |
| 0101 | 5 | 5 |
| 0100 | 4 | 4 |
| 0011 | 3 | 3 |
| 0010 | 2 | 2 |
| 0001 | 1 | 1 |
| 0000 | 0 | 0 |
| 1111 | 15 | -1 |
| 1110 | 14 | -2 |
| 1101 | 13 | -3 |
| 1100 | 12 | -4 |
| 1011 | 11 | -5 |
| 1010 | 10 | -6 |
| 1001 | 9 | -7 |
| 1000 | 8 | -8 |

Using the table, you can quickly check some examples. Notice that the result is wrong at *different* spots for each interpretation!

e.g., 1010 + 0101 = 1111

Unsigned: 10+5 = 15 (OK)

Signed: -6 + 5 = -1 (OK)

1110 + 0010 = 0000

Unsigned: 14 + 2 = 0 (plus a carry!)

Signed: -2 + 2 = 0 (OK)

0100 + 0100 = 1000

Unsigned: 4 + 4 = 8 (OK)

Signed: 4 + 4 = -8 (overflow error!)

Whether the math has gone out of range depends on whether we're considering the numbers as signed or unsigned. Circuits which add binary numbers set flags to indicate carry or overflow so the programmer can check them and determine if something's gone wrong.

### Boolean Algebra and Logic Gates

Boolean Algebra is logic math, using

1. 1 for True and 0 for False,
2. multiplication for AND and
3. addition for OR.

Truth Table for Boolean Logic Operations:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *A* | *B* | A picture containing application  Description automatically generated  *A* AND *B* | Scatter chart  Description automatically generated with medium confidence  *A* OR *B* | Chart  Description automatically generated  Not(A) | A picture containing scatter chart  Description automatically generated  *A* XOR *B* | A picture containing square  Description automatically generated  *A* NAND *B* | Scatter chart  Description automatically generated with medium confidence  *A* NOR *B* |
| 0 (False) | 0 (False) | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 (True) | 0 (False) | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 (False) | 1 (True) | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 (True) | 1 (True) | 1 | 1 | 0 | 0 | 0 | 0 |

Reasoning:

-The statement "A and B" is only true if both A and B are true; i.e., it's Tuesday AND it's raining is only true if it's both Tuesday and raining; otherwise it's false.

-The statement "A or B" is true as long as either A or B are true; i.e., it's Tuesday OR it's raining is true as long as either it's Tuesday or it's raining (or both); otherwise it's false.

-NOT is an inverter; it returns the opposite of the input signal (false if the input is true, true if the input is false). e.g., "Steve is here" = not("Steve is not here")

-XOR is an exclusive version of or that returns true if exactly 1 of the inputs is true. It's the same as OR except that OR(1,1) = 1 while XOR(1,1) = 0.

e.g., suppose you ask Steve "Did you have soup or salad?" and the response is *Yes*…

…then you ask Steve "Did you have soup xor salad?" and Steve answers *No*.

This means Steve had both soup *and* salad.

-"NAND" means "Not And"; it returns the opposite of what an AND would. This is hinted at with the symbols:  
AND: A picture containing application

Description automatically generated

NOT: Chart, scatter chart

Description automatically generated

NAND: A picture containing square

Description automatically generated

the triangle input does nothing, while the circle inverts the signal.

It's also hinted at with the Boolean algebra symbol:  is a combination of "ANDing" *A* and *B* () and then inverting the output ()

-Similarly, NOR means "Not Or", and returns the same as "ORing" the inputs then inverting the result.

Great video explaining Boolean Logic & Logic Gates: <https://youtu.be/gI-qXk7XojA?list=PL8dPuuaLjXtNlUrzyH5r6jN9ulIgZBpdo>

#### Rules of Boolean algebra:

1. Associativity:
   1. 
   2. 
2. Distributivity:
   1. 
3. Commutitivity:
   1. 
   2. 
4. Absorptive cases (one of the inputs absorbs the others)
   1.  (X or true = true always; the X doesn't matter)
   2.  (X or false = X; the false doesn't matter)
   3.  (X and true = X; the true doesn't matter)
   4.  (X and false can never be true)
   5.  (True IFF X is true)
   6.  (True IFF X is true)
   7.  (since  and )
   8.  (since if X is true it's  and if X is false it's , so it's )
   9.  (there's no situation where the  makes a difference; if X is true it's =Y and if X is false it's )
5. De Morgan's Laws:
   1.  (Not (X or Y) = Not(X) and Not(Y); "neither one is true")
   2.  (Not (X and Y) = Not(X) or Not(Y)); "both aren't false")

*Check for yourself that these rules work by thinking through them AND/OR computing left side and right side for the expressions for possible input values!*

*e.g.,* should ???

Solution:  will return 1 if *either* *B* or *C* is true and will only return 0 if both are false. When we OR that with  in , we have a result that only be false if *A* is false or  is false, and therefore the overall result is only false if all 3 are false, and we don't even need the brackets: .

*e.g., should* ???

 says not(X or Y); since X or Y is true when either is true, not(X+Y) is false when either is true and true only when both are false, that is, only true when both X is false AND Y is false, that is, .

→  *feels* like a magic operation for Boolean algebra, but makes total sense when you think about what it's saying!

#### Logic gates & Rules Summary:

Rules for Logical addition (**OR**), Rules for Logical Multiplication (**AND**) , and finally, (**NOT**)

Diagram

Description automatically generated with medium confidenceA picture containing text, clock

Description automatically generatedA picture containing text, clock, gauge

Description automatically generated

0+0=0 0\*0=0

0+1=1 0\*1=0 0=1

1+0=1 1\*0=0 1=0

1+1=1 1\*1=1

Rules for **NAND**, and **NOR** Gates are shown below:

Diagram

Description automatically generatedDiagram

Description automatically generated

These are equivalent to **AND** or **OR** gates followed by inverters, as implied by the circle symbols:

Diagram

Description automatically generatedDiagram

Description automatically generated with low confidence

 = X NOR Y

Diagram

Description automatically generated

 = NAND

Diagram

Description automatically generated

**Truth Tables:**

**NAND:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X | Y | !X | !Y | !(XY) |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

**NOR:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X | Y | !X | !Y | !(X+Y) |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |

Finally, there is the **XOR** Gate and Truth Table:

Diagram

Description automatically generated

|  |  |  |
| --- | --- | --- |
| X | Y | Z |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

#### Logic Implementation using NAND and NOR gates

A **NAND** gate can be an inverter if you tie both inputs together, meaning multiple NAND gates can act like any other gate you need:

1. NOT gate with NAND:,  
   A picture containing diagram

   Description automatically generated
2. AND gate with NANDs:   
   Diagram

   Description automatically generated
3. OR gate with NANDs:   
   Diagram

   Description automatically generated

*Note: you can also produce the NOR gate with NAND by inverting the result of (3)*

***NOR gates have the same flexibility;***

e.g., , and , so use one NOR gate each to invert A & B then a third NOR gate on the results to get .

At first this seems silly; why bother using *more* NAND gates to implement things like AND and OR gates??

***Reason:*** It takes 50% more transistors to create an AND or OR gate than a NAND or NOR one, meaning that even if it requires more gates to implement logic using NAND & NOR rather than AND, OR, and NOT gates, it may require less architecture overall.

#### Example Problem

Create a Truth Table based on the digital circuit in Figure 1 below. Write a Boolean expression for the circuit outputs  and  (doesn’t have to be the minimum sum-of-products).

**Diagram, schematic

Description automatically generated**

**Figure 1**

**Solution:** The input bits are the three letters A, S, and D (physically, these are voltage sources set at either 0 V or 5 V depending on whether they're binary 0 or 1).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A | S | D | U1A | U2A | U5A | LED2 (Y1) | LED1  (Y2) |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |

The Boolean expression used to represent the circuit is:





### Integrated Circuits (Lab Info)

<https://en.wikipedia.org/wiki/List_of_7400_series_integrated_circuits>

74HC00 series is high-speed CMOS maintaining compatibility with the BJT TTL 74 series, and 00 is a quad 2-input NAND gate:

Diagram

Description automatically generatedDiagram, schematic

Description automatically generated

74HC02 is a quad 2-input NOR gate:

A picture containing diagram

Description automatically generatedDiagram

Description automatically generated

(Notice that the order of the pins is not the same on each chip! Be careful and always read the datasheet!)

## Logic gate Design and K-Maps

### Application: Adders

Great video explaining adders: <https://youtu.be/1I5ZMmrOfnA?list=PL8dPuuaLjXtNlUrzyH5r6jN9ulIgZBpdo>

Circuits which add two inputs together are called adders. A half-adder adds two bits, while a full-adder adds three bits together, letting it also account for the carry bit from the previous addition. Either adder has 2 outputs: the sum and the carry.

**Half-Adder Truth Table and Design:**

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | **Sum Out** | **Carry Out** |
| 0 | 0 | **0** | **0** |
| 0 | 1 | **1** | **0** |
| 1 | 0 | **1** | **0** |
| 1 | 1 | **0** | **1** |

(The "carry" shows the output was too big to store in just the sum bit; e.g., 1+1=0 (carry the 1))

Diagram

Description automatically generated

Full-Adder Logic Design and Truth Table:

Diagram

Description automatically generated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | Carry In | **Sum Out** | **Carry Out** |
| 0 | 0 | 0 | **0** | **0** |
| 0 | 0 | 1 | **1** | **0** |
| 0 | 1 | 0 | **1** | **0** |
| 0 | 1 | 1 | **0** | **1** |
| 1 | 0 | 0 | **1** | **0** |
| 1 | 0 | 1 | **0** | **1** |
| 1 | 1 | 0 | **0** | **1** |
| 1 | 1 | 1 | **1** | **1** |

*Question*: Can you express the SUM bit of the half adder's operation using fewer gates?

Answer: Yes; it's the same as  (A XOR B)

#### Sum of Products (SOP) & Product of Sums (POS) Form

But how did we get this output for the adder circuit originally?

Consider the truth table. There are exactly 4 input states for ABC that would cause the output to be a 1:

A picture containing calendar

Description automatically generated

So, we can say that the sum function is 1 only if at least one of those states happens, that is,



This form of the output is called the **sum of products** form (SOP form); if any of the states happens, the output will be true.

It takes the form of a number of products summed together:  
Diagram

Description automatically generated

Alternatively, we can look at all the input states which would cause the output to be a zero:

Calendar

Description automatically generated

Then we can write a function that will produce a zero overall only if any of these states happens. To achieve this, we need to AND several expressions which only produce zero when the states happen. e.g., consider the first zero:

A screenshot of a computer

Description automatically generated with low confidence

The function  will give a 1 *unless* all 3 are 0. That is, it only gives a 0 if all 3 are 0, which is what we needed.

→ Note that this expression is equivalent to ; an expression which only returns 1 if all 3 inputs are false (), but inverted so that it only returns 0 if that happens: .

All together, we can write the sum logic using the zeros as:



The second form of this (which we obtained using DeMorgan's law) is called **product of sums** form (POS form).

Diagram

Description automatically generated

**Notation: Minterms & Maxterms**

The products in a sum of products form are often called **minterms**, and denoted using a numbering system from binary counting:  since  is true only when ; , , and so on. Using this, the SOP  is 

Similarly, the sums in a product of sums form are often called **maxterms**, denoted using the same numbering system for the only state that gives a false for the sum; e.g.,  is only false for , and therefore . Using this, the POS form is 

Notice that the maxterms used for POS form are the opposite of the minterms used for the SOP form. This is a good way to check that you haven't missed anything in your implementation.

The minterms & maxterms are useful for understanding the output, but not as useful if you want to obtain an efficient implementation of it. For this we could directly simplify the boolean expression, watching out for terms we can combine; e.g., two terms differing only by one variable mean we can eliminate that variable:

SOP example: 

POS example: 

What's going on in these examples in terms of minterms & maxterms is:





The terms corresponded to 000 & 001. In terms of the binary numbers, to have two terms we can combine into a simpler one we need two minterms or maxterms which are only different by a single bit.

e.g., 



where X symbolizes "don't care", meaning it could be either 1 or 0 and the expression works. Karnaugh mapping is a technique for carrying out these simplifications using a graphical aid.

### Karnaugh Maps (K-Maps) and Gray Code

#### Gray Code

Because we can simplify terms if they're only different by one bit (e.g., 0101 to 0111 or 0001 or 0100 or 1101) it's useful to count in a way that changes only one bit at a time. e.g.,

00

01

11

10

This kind of counting is called **Gray code** (after Bell Labs physicist Frank Gray), and besides Karnaugh maps, it's also useful for error checking: if you can count on only one bit changing at a time, it must be a mistake if two bits have changed.

#### Karnaugh Mapping

**Karnaugh Mapping** or **k-mapping** is a technique to produce simplified Boolean logic expressions using graphical tools called **Karnaugh map**s. It's particularly good for systems with 4 inputs, but also works well for anywhere up to 6 inputs (beyond that it needs 4-dimensional visualization).

Here's the technique for 4-inputs:

1. Lay out the inputs in pairs as headings for the columns and rows, counting in Gray code.
2. Enter a 1, 0, or X (for "don't care") in each interior cell for the desired output value.
3. For SOP expansion:
   1. Make a size 2n rectangle covering every 1, using as few and as big of rectangles as possible
   2. For every rectangle, write a single product term corresponding to it
   3. Sum up the product terms for each rectangle.
4. For POS expansion:
   1. Make a size 2n rectangle covering every 0, using as few and as big of rectangles as possible
   2. For every rectangle, write a single sum term corresponding to it
   3. Multiply the sum terms for each rectangle.

#### Example #1 (Truth Table)

e.g., simplify the logic  corresponding to the following truth table using a k-map

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | S | D | F | Y |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | X |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | X |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |

First, lay out the AS along the rows and the DF entries along columns of a k-map:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AS\DF | 00 | 01 | 11 | 10 |
| 00 |  |  |  |  |
| 01 |  |  |  |  |
| 11 |  |  |  |  |
| 10 |  |  |  |  |

Note the ordering of the numbers uses Gray code; this way adjacent entries will be different by only one bit, which is important for how we're going to simplify it using the rectangles.

Next, enter the outputs from the truth table into the appropriate cells:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AS\DF | 00 | 01 | 11 | 10 |
| 00 | 1 | 1 | X | 0 |
| 01 | X | 0 | 1 | 1 |
| 11 | 1 | 0 | 1 | X |
| 10 | 0 | 0 | 1 | X |

SOP implementation:

For each 1, draw the largest rectangle of size 2n possible covering it, using the least number of rectangles as you can:

A picture containing text, crossword puzzle, clipart

Description automatically generated

Ended up with one size-2 rectangle and two size-4 rectangles (note that size 3 and size 6 rectangles don't work, since neither is a power of 2, and that the rectangles can wrap around the outside).

The top left rectangle corresponds to ; i.e., we get a 1 for if the product term  is true.

Similarly, the other rectangles give  and , for a total SOP expansion of .

As promised, this does indeed reproduce a 1 and 0 where required in the truth table.

POS implementation:

Draw rectangle with the same algorithm as before, but this time for the 0's:

A picture containing text, crossword puzzle

Description automatically generated

For each rectangle, write the corresponding sum term; e.g., the bottom left rectangle means we need a 0 if , meaning we need to and the inverse of this term, . Similarly with the other rectangles the total logic expression is 

Note that while this also reproduces the logic in the table, in this case it wasn't as simple as the SOP implementation. With "don't-care" conditions, the two implementations aren't complements of eachother, in general.

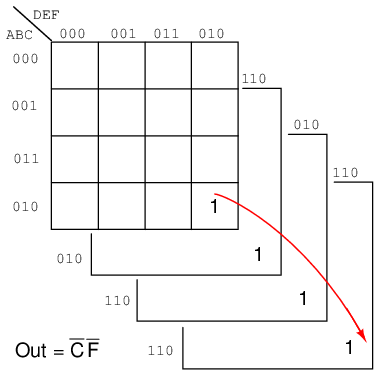
Note also that there are sometimes alternative equally-simplified implementations; e.g., here we could have used:

A picture containing text, crossword puzzle

Description automatically generated

This makes the expression  instead, which is also valid.

For the design project, you'll need to use 5 or even 6 input K-mapping, which works the same but with another dimension. e.g., here's a visualization of a 6-input k-map:



<https://www.allaboutcircuits.com/textbook/digital/chpt-8/larger-5-6-variable-karnaugh-maps/>

In practice, you'd lay out the different grids on paper side by side and just *imagine* them in 3d when drawing the boxes.

#### Example #2 (Boolean Logic)

Simplify  using Karnaugh mapping

This expression already looks pretty simple, but could it be simpler? This time we'll go right to the k-map, calculating the output values from the Boolean expression as-needed and entering them in:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Z\XY | 00 | 01 | 11 | 10 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |

The k-map reveals that we have one too many rectangles implemented:

Chart, box and whisker chart

Description automatically generated

The redundant rectangle is , and without it the expression  is logically equivalent to the starting expression.

Note: while we could've figured this out from the Boolean algebra directly without k-mapping, the k-mapping makes it much easier to detect the optimal logic implementation, especially as the complexity rises.

#### Example #3 (Digital Circuit)

e.g., simplify the logic corresponding to the following truth table using a k-map

**Diagram

Description automatically generated**

Take the inputs as A, B, and C. Then:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | **B** | **C** | U2A | U1A | U5A | U7A | U3A | U4A |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A\BC | 00 | 01 | 11 | 10 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |



Chart

Description automatically generated



Could you have done this without k-mapping? Yes. From the truth table, you could write the maxterms: 

Then simplify this as follows:



(in the last step, used  for every term with  in it,  for every other instance of *AB*, and  for the remaining terms).

A third option is to jump right to the Boolean algebra from the gates directly, without ever going through the truth table:



(Which option is the most efficient will vary from situation to situation.)