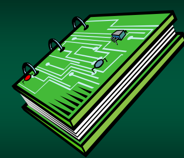
Arbitrary Circuits

Part 12

1



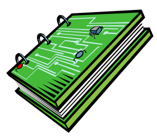
Creating an Arbitrary Circuit

From Truth Table to Wires

2

Creating an Arbitrary Circuit

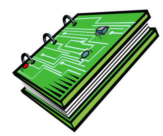
- We converted between Boolean expressions and circuits
- It maintained a one-to-one correspondence between gates in the circuit and operators in the equation



3

Creating an Arbitrary Circuit

- Given an arbitrary logic table, how do we realize a circuit for it?
- Simple, we look at the inputs that make it true, and write them out in an expression using or's.



4

Example: 1 Bit Add Mod 2

5

Example: 1 Bit Add Mod 2

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

6

Example: 1 Bit Add Mod 2

```
We want a circuit that is true
when:

(a = F and b = T) or
(a = T and b = F)

out = a' * b + a * b'
```


Example 2: One Bit Adder

Example 2: One Bit Adder

a	b	Out $_1$	Out $_0$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Example: One Bit Adder (Logic)

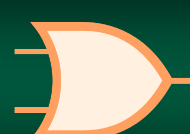
```
out1 = (a = T and b = T)

out0 = (a = F and b = T) or
       (a = T and b = F)
```


Example: One Bit Adder (algebra)

```
out1 = a * b

out0 = a' * b + a * b'
```

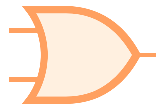



Disjunctive Normal Form

Express Logic With Ease

Disjunctive Normal Form

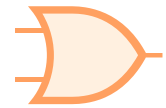
- Best approach to converting tables into circuits is use *Disjunctive Normal Form*
- In this form, the expressions consists of OR's (disjuncts) connecting AND sub-expressions



13

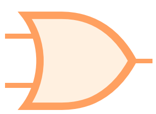
Definitions

- A *literal* is a Boolean variable *v* or its complement (e.g. v *or* v')
- A *minterm* of Boolean product $v_1 * v_2 * \ldots v_n$



14

Definitions



- Hence, a minterm is a "product" of *n* literals, with one literal for each variable
- An equation written only as the "OR" of minterms is in *disjunctive normal form* (also called *sum-of-products* form)

15

Algorithm

1. Find the rows that indicates a 1 for output
   - ignore the ones with 0 as output
   - we are making an equation based on true
2. Write a minterm for each of them
3. "OR" all the minterms

16

Example

a	b	y (out)
0	0	1
0	1	1
1	0	0
1	1	0

17

Example

```
DNF of the table is:

y = (a' * b') + (a' * b)

For brevity, for this point on, let's
write as:

y = a'b' + a'b
```

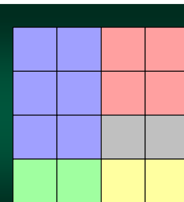

18

Example

```
We can simply using Boolean algebra:

y = a'b' + a'b
  = a' (b' + b)        Distributive
  = a' (1)             Complement
  = a'                 Identity
```


19



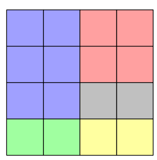
Karnaugh Maps

The Right-Brain Gets to Help

20

Karnaugh Maps

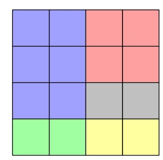
- A *Karnaugh Map* (pronounced "car-no") is a visual tool to help see relations between minterms.
- A K-Map for *n* variables is a grid of 2^n squares



21

Karnaugh Maps

- Every possible minterm of *n* variables is represented
- *Every square is a minterm*
- It is arranged is such a way that we can simplify our table



22

Gray Code

- Literals are ordered using *gray code*
  - values in the table are not ordered in normal ascending order
  - each square differs in exactly one literal
  - why? we will cover this later
- NOTE: squares wrap-around to the sides

23

Two Variable Example

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

24

Two Variable K-Map

		A: 0	A: 1
B	0	1	1
	1	1	0

Each combination of A

Squares have the output of the circuit

25

Three Variable Example

a	b	c	out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

26

Three Variable K-Map

		AB: 00	01	11	10
C	0	1	0	1	1
	1	0	0	1	1

27

Four Variable K-Map

		AB: 00	01	11	10
CD	00	0	0	1	1
	01	1	1	0	0
	11	1	1	1	0
	10	0	1	1	0

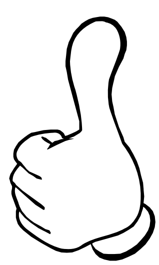
28

How to Use a K-Map

1. Mark the squares of a K-map corresponding to the function
2. Select a minimal set of rectangles where
   - each rectangle has a power-of-two area and is as large as possible
   - cover every marked square
3. Translate each rectangle into a single midterm and sum (or) all these

29

Converting a Rectangle to Minterm



- If any literal contains both 1 and 0, in the rectangle, it is eliminated
- The goal is to draw the **biggest** rectangles possible

30

Example Square: 1×1

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

A' B C' D

31

Example Square: 2×1

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

B C' D

32

Example Square: 1×4

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

C' D

33

Example Square: 2×2

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

A' D

34

Example Square: 2×2: Wrapped

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

B' D

35

Example Square: 2×2: Wrapped

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	0	1
10	1	0	0	1

B' D'

36

Example Square: 4×2

AB

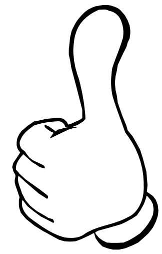
CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

D

37

Tips

- There is no magic way to do Step 2. Look and play around until you find the answer
- You can overlap squares – just as long as you "cover" all the 1's



38

Four-Variable K-Map

AB

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

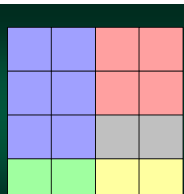
A'D + BC + AC'D'

39

Efficiency of K-Maps

- A K-Map does not necessarily make the *best* expression/circuit
- All expressions made this way are sums-of-products and some can be made simpler
- For example: a(b+c) is the same as ab+ac, but uses fewer gate inputs

40



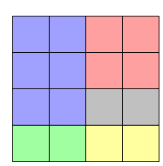
How K-Maps Work

You are doing more than you think

41

How K-Maps Work

- The order of gray code, and the 2^n squares allow us to factor out literals
- Every time you eliminate a literal, you are performing **three** Boolean algebra laws
- This is done visually, so it is invisible!



42

How K-Maps Work

1. First you use the *Distribution Law* on the minterms leaving **(v + v')** - which is the terminal that *changed*
2. You then use the *Complement Law* on **(v + v')** leaving 1
3. Finally, you remove the 1 using the *Identity Law*

Let's Look at This Again...

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

ABC'D' + AB'C'D'

B changes

Let's Look at This Again...

```
ABC'D' + AB'C'D'

AC'D'(B + B')       Distributive

AC'D'(1)            Complement

AC'D                Identity
```


How About Another Rectangle?

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

Both A and D change

A'BCD + ABCD +
A'BCD' + ABCD'

How About Another Rectangle?

```
A'BCD  + ABCD + A'BCD' + ABCD'

BCD(A' + A) + BCD'(A' + A)

BCD(1) + BCD'(1)

BCD + BCD'
```

A eliminated

… and it keeps going…

```
BCD + BCD'

BC(D + D')

BC(1)

BC
```

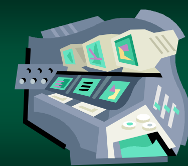
D eliminated

Please Wait

CSC 28
will begin shortly

Please open the chat window.

49



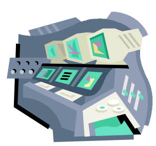
K-Maps and Programming

Using it to simplify code

50

K-Maps and Programming

- The Boolean expressions, that you use in your Java programs, are the same as the expressions we cover
- So, you can apply K-Maps to your Java code to simplify expressions



51

K-Maps Can Simplify Expressions

- The following is a complex expression that, on the surface, looks difficult to simplify
- K-Maps can help

```
if (a && !b && c  || a && b && !c ||
    a && !b && !c || a && b && c)
```


52

K-Maps Can Simplify Expressions

- First, let's put the expression in the Computer Engineer notation
- Ah, we can see the structure now!

```
ab'c' + abc' + ab'c + abc
```


53

K-Maps Can Simplify Expressions

c \ ab	00	01	11	10
0				
1				

ab'c' + abc' + ab'c + abc

54

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0				1
1				

ab'c' + abc' + ab'c + abc

55

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0			1	1
1				

ab'c' + abc' + ab'c + abc

56

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0			1	1
1				1

ab'c' + abc' + ab'c + abc

57

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0			1	1
1			1	1

ab'c' + abc' + ab'c + abc

58

K-Maps Can Simplify Expressions

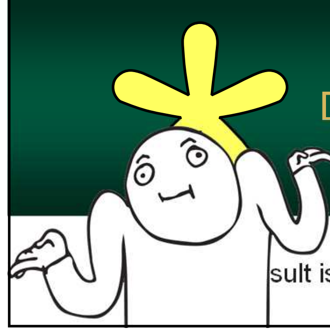
ab

c \ ab	00	01	11	10
0			1	1
1			1	1

ab'c' + abc' + ab'c + abc = a

59

Don't Care

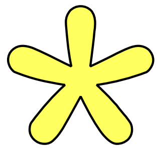


sult is Meaningless

60

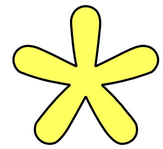
Don't Care

- Sometimes *we don't really care* what output the circuit generates for some combinations of inputs
- So, for those inputs, the results are simply not significant



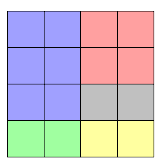
Don't Care

- In truth tables, the value "Don't Care" is represented with an asterisk
- It can be considered True or False – whichever is more *convenient* for the circuit



Karnaugh Maps and Don't Care

- We can construct a Karnaugh Map like before
- Except the squares corresponding to don't care outputs are marked (with an asterisk)



Karnaugh Maps and Don't Care

- Then, when outlining blocks, we can (at our convenience) consider the "don't care" squares as either 0 or 1
- Since we want to make the largest outlines possible, we will sometimes consider a don't care to be true, and sometimes false

Example

- We want to guarantee that the output of a circuit is 1 if both inputs are 1
- And 0 when both inputs are 0
- But otherwise we do not care

Example

a	b	out
0	0	0
0	1	*
1	0	*
1	1	1

K-Map For The Example

B \ A	0	1
0	0	*
1	*	1

We don't need B!

out = A

67

… or we can do this

B \ A	0	1
0	0	*
1	*	1

Just B!

out = B

68

Four-Variable (with Don't Care)

CD \ AB	00	01	11	10
00	0	1	1	*
01	1	1	*	0
11	1	1	1	0
10	*	1	1	0

out = B + A'D

69

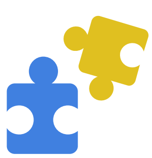
Functional Completeness

Just How Much Do We Need?

70

Functional Completeness

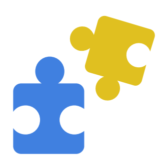
- We can construct a circuit for any Boolean expression using and / or / not
- This means the set of gates {and, or, not} is *functionally complete*



71

Function Completeness

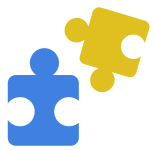
- However, we don't need all three gates
- DeMorgan's laws shows us that we can construct:
  - an OR using an AND
  - and AND using an OR



72

We Don't Need Or!

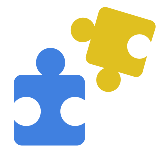
- So {and, not} are also complete because by DeMorgan's Law:
  x + y = (x'y')'
- So, any expression that can be written using {and, or, not} can be written using just {and, not}



73

or… We Don't Need And!

- Also {or, not} is functionally complete since xy = (x'+y') '
- So, any expression that can be written using {and, or, not} can be written using just {or, not}



74

Functional Completeness

- So, are any of the singular sets {and}, {or}, {not} functionally complete?
- In other words, can and/or/not all be converted into a single type of gate?
- **No.** Neither {and} or {or} can be converted to a {not}

75

NAND

- So, is there a gate that can, alone, be functional complete?
- What about NAND (negated And)?
  - x nand y = (xy)'
  - Note: the NAND gate is not implemented with an AND gate and a NOT gate. It just has the same truth table as (xy)'

76

NAND

- To show that {nand} is functionally complete, we need to show that we can implement {and, or, not} using it
- The result would be greatly beneficial!
  - we would have to just construct 1 gate to create any circuit
  - this would greatly aid construction

77

Not → Nand

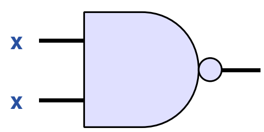
```
Converting not to nand:

x'  =  x'
    =  (xx)'         Idempotent
    =  x nand x      nand format
```

We can implement NOT by using a NAND. Both input will be x

78

Not → Nand



79

Or → Nand

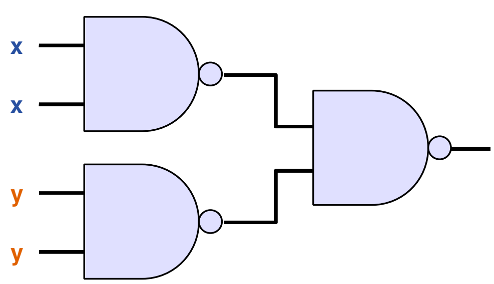
```
Note: x' = x nand x

x + y =  x + y
      =  (x'y')'               DeMorgan
      =  x' nand y'            nand format
      =  (x nand x) nand (y nand y)
```

Last proof let us convert NOT into NAND

80

Or → Nand



81

And → Nand

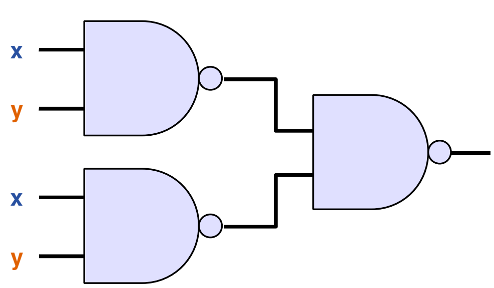
```
Note: x' = x nand x

xy    =  xy
      =  ( (xy)')'             Involution
      =  (x nand y)'           Negate nand
      =  (x nand y) nand (x nand y)
```

Last proof let us convert NOT into NAND

82

And → Nand



83

Summary

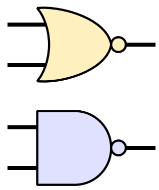
- The expressions below show that nand can be used to implement NOT, OR, AND
- So, we can just use NAND since it is *functionally complete*

```
x'    =  x nand x
xy    =  (x nand y) nand (x nand y)
x + y =  (x nand x) nand (y nand y)
```


84

How Hardware Works

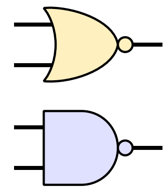
- Also NOR is functionally complete
- P NOR Q = (P + Q)'
- Hardware can alternatively use this gate rather than NAND



85

How Hardware Works

- If our hardware can just implement NAND or NOR, then we can create a circuit with just one gate
- In fact, many fabrication processes use only NAND or NOR gates



86