CS 221 Computer Architecture

Week 11: Combinational Networks

Fall 2001

Course Schedule

W1	Sep 11- Sep 14	Introduction	
W2	Sep 18- Sep 21	Information Representation (1)	(Chapter 3)
W3	Sep 25- Sep 28	Information Representation (2)	(Chapter 3)
W4	Oct 2- Oct 5	Computer Architecture Pep/6	(Chapter 4)
W5	Oct 9- Oct 12	Assembly Language	(Chapter 5)
W6*	**Oct 16- Oct 19**	**High-Level Languages (1)**	**(Chapter 6)**
W7*	**Oct 23- Oct 26**	**High-Level Languages (2)**	**(Chapter 6)**
	Midterm*		
W8	Oct 30- Nov 2	Instruction Sets	(Tanenbaum)
W9	Nov 6 - Nov 9	Storage Management	(Chapter 9)
W10*	**Nov 13- Nov 16**	**Devices and Interrupts**	**(Chapter 8)**
W11	Nov 20- Nov 23	Combinational Logic	(Chapter 10)
W12	Nov 27- Nov 30	Sequential Logic & Review	(Chapter 11)

**Marked classes will be taught by Dave Dove, same place same slot*

Assignment 4

- **Assignment 4 will be posted today.**
- **The task is to translate a higher-level language (*java*) program to PEP/6 assembly**

Due: **Tuesday December 4th, 12:00**
Where: **Dropbox CISC 221 Goodwin Hall**

Plan for week 11

- **Logic gate level — Combinational networks**
  - Truth tables, Boolean algebra and logic gates
  - Combinational Analysis
  - Karnaugh maps
  - Combinational Devices
    - Multiplexer, Demultiplexer, Binary Decoders, ALU
- **Readings**
  - Chapter 10
  - Next week chapter 11

I.
Truth Tables, Boolean Algebra and Logic Gates

Level 1: Logic gate

- **We have finally arrived at the end of our journey**
  - the implementation of operations using logic gates.
  - Level 2, microprogramming, is beyond our scope, as are lower levels such as physics
- **So what is a logic gate?**
  - Logic Gates are used to control the flow of electricity around an electrical circuit.
  - Opening or closing gates allows electricity to flow or impedes its flow.
  - Logic gates are more than simple switches because they usually have two inputs and the output depends upon the combination of inputs.
  - There are different types of Logic Gates with different matches of input to output.

Logic Gates (2)

- **Modern logic gates are built of transistors and have many thousands on a simple chip**
  - but the logic is easier to see in an old fashioned circuit that uses electomagnetic switches
  - i.e. the input streams control the switches in the circuit that allows the electricity to flow or not.
- **There are several types of logic gates. The most common are**

AND gates
Perform and operations
OR gates
Perform OR operations
NOT gates
Perform NOT operations

- **We can use these to implement a CPU and Memory**

Digital Logic: Definitions

- **Logic gates can be combined to form a circuit implementing some higher-level behavior**
  - A circuit is a collection of (electronic) devices connected by wires to form a network or net.
- **A network views a circuit as a black box which produces a set of output signals, given a set of input signals.**
  - In the physical implementation of a network:
  - the signals are wires leading into and out of the circuit
  - the value of each signal is represented by voltage on the wire.
- **Can be implemented using positive or negative logic**
- **Positive logic**
  - high voltages represent 1 and low voltages represent 0.
- **Negative logic**
  - low voltages represent 1and high voltages represent 0.

An example logic network

Since we won't get into the electronics of implementing digital circuits, we treat a network as a black box.

It's possible for both positive and negative logic to be used in a single computer system, just because some circuits are easier to implement with positive logic, and some are easier to implement with negative logic. To avoid the complications of knowing which form of logic is being used, we'll talk about 1s and 0s or about asserted signals (1) and unasserted signals (0), instead of high and low voltages.

Two Types of Networks at Level 1

- **Combinational networks :**
  - the value of the output signals at any instant is determined entirely by the value of the input signals.
  - Remember: each output signal is a mathematical function of the combination of the input signals.
  - Used to implement arithmetic and logic operations.
- **Sequential networks :**
  - the value of the output signals depends on the past history of the network as well as the current value of the input signals.
  - Remember: each output signal depends on the sequence of past input and output signals.
  - used to implement the notion of state, i.e., memory.

Representations of combinational nets

- **Truth table**
  - A table showing the value of everyoutput for each possible combination of input values.
- **Boolean algebra**
  - Describes a combinational circuit using expressions containing:

+ : a +b means a OR b.

• : a • b means a AND b.

' : a' means NOT a.

- **Logic diagrams**
  - show the logic gates connected by wires that implement the combinational net in hardware.

These describe behavior of box

a	b	c	x	y
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

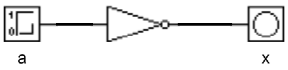
(x is the sum of a binary adder, y the carry-out)

Some Basic Example Nets: NOT

- **Boolean algebra** $x = a'$
- **Truth Table**

a	x
0	1
1	0

- **Logic Gate**

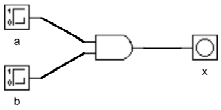


AND

- **Boolean algebra** $x = a \bullet b$
- **Truth Table**

a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

- **Logic Gate**

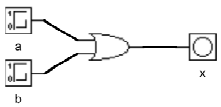


OR

- **Boolean algebra** $x = a + b$
- **Truth Table**

a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

- **Logic Gate**

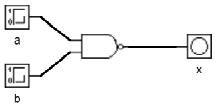


NAND

- **Boolean algebra** $x = (a \bullet b)'$
- **Truth Table**

a	b	x
0	0	1
0	1	1
1	0	1
1	1	0

- **Logic Gate**

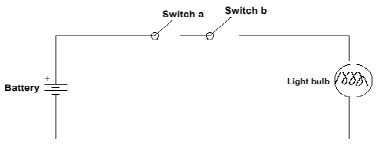


NAND (2)

- **NAND gates are the equivalent of an AND gate followed by an inverter (NOT).**
  - They are used frequently because in many of the technologies for implementing logic gates, it is easier to build NAND gates than AND gates.
- **This provides us with a convenient point for a short aside in how logic gates themselves are implemented.**
  - Using the sort of simple circuit that you might have put together with flashlight batteries in elementary school, you can create things that act as AND and OR gates.

Aside: implementing logic gates with switches

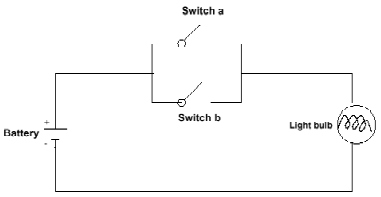
Elementary school AND



- Each switch represents an input of 1 if it is closed, 0 if it is open.
- Both switches must be closed for the light to come on.

Aside: implementing logic gates with switches

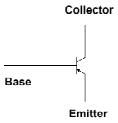
Elementary school OR



- If either or both of the switches is closed, the light will come on.

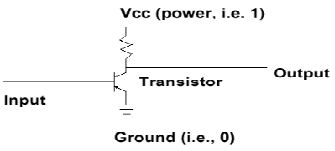
Aside: Transistors

- **A transistor can be thought of as a switch, where the finger turning the switch on and off has been replaced with another electrical input, called the base, which turns the switch on or off.**



- **If the base voltage is low enough (0, with positive logic), then the transistor acts as an infinite resistor, opening the switch between the collector and the emitter, so that no current flows between them.**
- **If the base voltage is high (1, with positive logic), then the transistor connects the collector and the emitter so that they have the same voltage, closing the switch between A and B.**

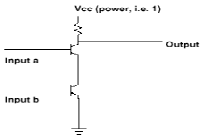
NOT gate implemented with a transistor



- if input voltage is high (1, with positive logic), the switch is closed, so the output voltage gets pulled down to ground (0).
- if input voltage is low (0), the switch is open, so the output voltage is equal to that of V_{cc} (1).

So the output is the opposite of the input. This is an invertor

NAND gate implemented with two transistors



- if both input voltages a and b are high (1, with positive logic), both switches are closed, so the output voltage gets pulled down to ground (0).
- if either a or b (or both) are low (0), the output voltage is not connected to ground, and shows the same value as V_{cc} (1).

So the output is "not a and b", this is a NAND gate.

NAND (3)

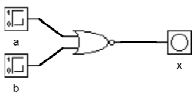
- **NAND gates can be used to build NOT, AND and OR gates (think about how you would do so).**
- **So it is possible to build any combinational network entirely with NAND gates.**
  - This is important because with many chip-producing technologies, NAND are easy to make
  - They require only two transistors
- **So they are common in the implementations of computer circuits.**

NOR

- **Boolean algebra** $x = (a + b)'$
- **Truth Table**

a	b	x
0	0	1
0	1	0
1	0	0
1	1	0

- **Logic Gate**

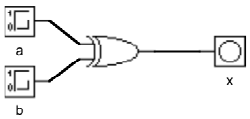


XOR (Exclusive OR)

- **Boolean algebra** $x = a \oplus b$
- **Truth Table**

a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

- **Logic Gate**



Properties of Boolean Algebra

Commutative	$x + y = y + x$
	$x \bullet y = y \bullet x$
Associative	$(x + y) + z = x + (y + z)$
	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$
Distributive	$x + (y \bullet z) = (x + y) \bullet (x + z)$
	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$
Identity	$x + 0 = x$
	$x \bullet 1 = x$
Complement	$x + x' = 1$
	$x \bullet x' = 0$

Properties of Boolean Algebra (3)

- **Try to think of the intuition behind each of these properties.**
  - For example, the complement property makes sense if you just say to yourself that "either x or not x must be true", and "x and not x cannot both be true, so x and not x is equivalent to false".
- **The distributive property might be harder to see intuitively**
  - in these cases, you can construct a truth table.

The distributive property

x	y	z	$x + y$	$x + z$	$(x + y) \bullet (x + z)$	$(y \bullet z)$	$x + (y \bullet z)$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

- **That `(x +y) • (x +z)` is equivalent to `x + (y • z)` is demonstrated by the fact that their columns in the truth table are identical.**
- **If you put it into words: if either x is true or (y and z) is true, then both (x or y) is true and (x or z) is true.**

Duality Property

- **The pairs of properties are examples of the duality of Boolean algebra.**
  - For any true expression in Boolean algebra (any property or theorem), there is a dual expression which is also true
  - Replace + with •,• with +, 0 with 1, and 1 with 0 throughout expression (flipping it)
- **Look at the example of**
  - "X or false is equivalent to X", versus
  - "X and true is equivalent to X".

The Indempotent Property

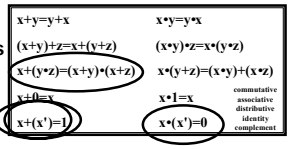
- **There are two more theorems that are useful in the analysis and design of combinational nets**
- **The idempotent property and its dual**

  x + x = x x • x = x

- **Proving this theorem requires a sequence of substitution steps using the 10 basic properties (5* their duals)**

Proving Boolean Algebra Theorems

We need to use the rules to go from
x + x to = x



x + 0 = (x + x)• 1 identity
= (x + x)•(x + x') complement
= x +(x • x') distributive
= x + 0 complement
= x

- You can do the same thing with the dual *x•x=x*

Absorption Properties

- Another useful set of theorems:

  x + 1 = 1 *x • 0 = 0*

- The next theorem is called the absorption properties because y is absorbed in x

  x + x • y = x x • (x + y) = x

  Please verify proof for these at home!

De Morgan's Law

- **The complement of x = x'**
  - We also know that x + (x') = 1 and x • (x') = 0
  - This is the complement rule
- **To prove that an expression is a complement of another expression, they have to follow that rule.**
- **An Example: De Morgan's Law states that**
  - The complement of a • b is in fact the complement of a + the complement of b
  - in boolean algebra: (a•b)' = a' + b'

II. Combinational Analysis

Combinational Analysis

- **So every boolean expression has a corresponding logic diagram and vice versa.**
  - A one-to-one correspondence between these
  - However, this is not true for truth tables.
  - They can be implemented in several ways.
- **So how do we turn one into the other?**
  - This is the subject of combinational analysis.

Turning Boolean Expressions into Diagrams

- **A boolean expression consists of one or more variables combined with AND, OR and NOT**
  - The number of inputs in the net is equal to vars.
  - We will start by looking at nets with one output.
- **To draw the logic diagram for a given expression:**
  - Draw an AND gate for each AND operation
  - An OR gate for each OR operation
  - And an inverter for each NOT operation
  - Connect the output of one gate to the input of another according to the expression.
  - The output of the combinational net is the output of the last gate

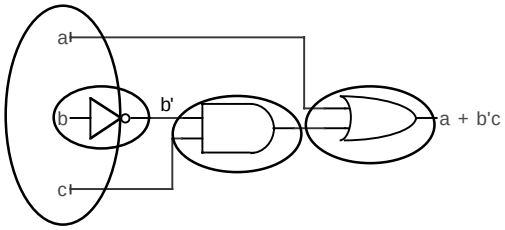
Precedence

- **To do this correctly, we have to know which operations take precedence over other**
  - The order of calculation of the expression
  - Note that these follow regular algebraic rules.
    - Note that AND is equivalent to a product
    - OR is equivalent to a sum
    - We know NOT is called a complement or invertor
- **We use the following precedence rules:**
  1. complements (NOT)
  2. products (AND)
  3. sums (OR)

Example 1: Using Precedence

a + (b') c

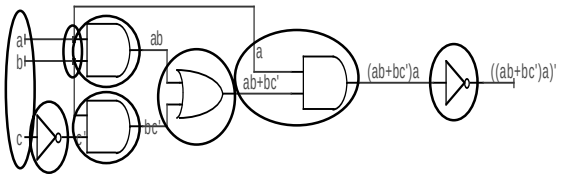
we can omit the and (as in multiplication) : a + b'c



Example 2: Using Parentheses

- **When things get complicated, or to force order, one can use parentheses:** ***((ab + bc')a)'***

Notice the junction?

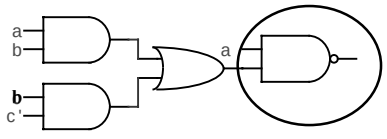


Junctions in Circuits

- **In circuits, one should think of variables as electrical signals.**
  - To duplicate a variable, you simply connect another line to its input (for example, input a).
- **When signal from input a reaches the junction, it does not act like water in a river with a fork.**
  - In a river, some water goes left, the other goes right
  - Thus, the signal would be half its strength
- **Because voltages is a measure of electrical potential, and the wires have low resistance**
  - The voltages stay the same after a junction!
- **We hide variable-duplication as well as inverters**
  - We assume every var and its complement is available as input to any gate

Abbreviated Version of Example 2

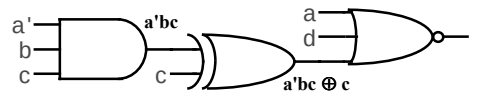
- **The below figure shows the abbreviated version of the previous example ((ab+bc')a)'**
  - It recognizes that an AND gate followed by an inverter is equivalent to a NAND gate



Converting back to Boolean

- **To convert a diagram back, label the outputs:**

NOR gate: OR and then invert

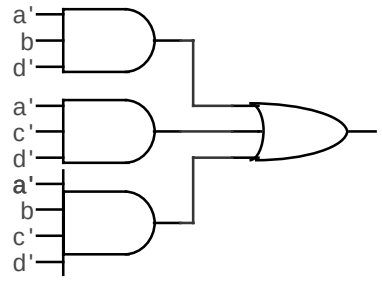


(a'bc ⊕ c + a + d)'

Two-Level Networks

- **Every boolean expression can be transformed into an OR of AND terms**
  - For example, a⊕b can be written as a'b + ab'
  - Using simple inverters! We'll get to why this is in a second.
- **Since gates operate in the real world, every step introduces a little delay in computation**
  - The trick is to reduce the number of steps.
- **We can reduce steps using inverters and duplication**
  - work parallel using ANDs and ORs with larger inputs
  - Number of inputs of the AND gates depends on the number of variables in the AND term
  - Number of inputs of the OR gate and the number of AND gates required depends on number of AND terms

A Two-level version of previous example



From Truth Tables to Boolean Expressions

- **To construct a boolean expression from truth table**
  - in terms of OR of ANDs for a two-level net:

truth table

a ⊕ b			***x***	***We observe that x is only 1***
0	*0*	=	*0*	*when a•b = 0•1 + 1•0*
1	*0*	=	*1*	*when a is 0 b should be 1 and v.v.*
0	*1*	=	*1*	*so a is the complement of b*
1	*1*	=	*0*	*so x= a'•b + a•b'*

Truth Tables to Expressions (2)

- **Another example:**

a	**b**	**c**	**x**	
0	**0**	**0**	**0**	
0	**0**	**1**	**1**	***x = 1 if abc = 001 + 011***
0	**1**	**0**	**0**	***x = a'b'c + a'bc***
0	**1**	**1**	**1**	***substitute each 0 by complement***
1	**0**	**0**	**0**	***where x = 1 then or the results***
1	**0**	**1**	**0**	
1	**1**	**0**	**0**	
1	**1**	**1**	**0**	

From Expression to Table

- **This may require the evaluation of your expression for every table entry**
- **For example x(a,b) = (a ⊕ b)' + a'**

	a	**b**	**x**
x(0,0) = (0 ⊕ 0)' + 0' = 1	**0**	**0**	**1**
x(0,1) = (0 ⊕ 1)' + 0' = 1	**0**	**1**	**1**
x(1,0) = (1 ⊕ 0)' + 1' = 0	**1**	**0**	**0**
x(1,1) = (1 ⊕ 1)' + 1' = 1	**1**	**1**	**1**

- **Or you can convert the expression to an OR of AND terms first using boolean algebra theorems.**

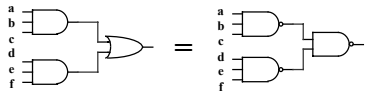
From Expression to Table

- **Once you have converted your expression to an OR of AND terms using Boolean algebra:**
  - Your truth table output (x) will contain a 1 for each AND term.
  - It will be located where the AND of values will yield that term to be 1: *when all values are 1.*
    x = (a'bc ⊕ c + a + d)'
    = a'bd' + a'c'd' + a'bc'd' if a=0 b=1 c=0 d=0
    010 + 000 + 0100 (0 when complement else 1)
    = 111 + 111 + 1111
  - So in this example, the truth table will contain 1 as output when abd = 010 acd=000 or abcd = 0100
  - All other output entries are 0

The Ubiquitous NAND

- **Not only do we want our networks to be 2-level**
  - We want them to consist of NAND gates only
  - This is because these are easy to manufacture
- **We can turn any OR of AND terms into an NAND of NAND terms using De Morgan' Law (a+b)'=a'•b':**
  - and a complement operation

  [(abc) + (def)]' = [(abc)'•(def)'] so
  [(abc) + (def)] = [(abc)'•(def)']'



III. Combinational Design

Combinational Design

- **In combinational design, we try not just to optimize the number of levels of a network**
- **But also the number of components or gates**
  - Using the absorption property x+xy=x

    x= a'bd' + a'c'd' + ~~a'bc'd'~~

    = a'bd' + (a'c'd') + (a'c'd')b

    = a'bd' + a'c'd'
- **We now have a network with only 2 AND gates!**

Canonical Expressions

- **To optimize the two-level network, we also try to have each AND term**
  - contain all input variables only once.
  - Such an AND term is called a ***minterm***

    x = abc + a'bc + ab

    = abc + a'bc + ab(c+c') (complement)

    = abc + a'bc + abc' (idempotent)
- **The last expression is a *canonical expression***
  - An OR of minterms in which no two identical minterms appear
  - Canonical expressions are convenient as they match one-to-one to a truth table
  - Each minterm represents a 1 in the truth table!!!

Canonical Expressions

- **Since the canonicial expression for x in our previous example had three minterms**
  - its truth table contains three 1's as outputs
  - it has four columns: 1 output and 3 vars: abc
  - it has 8 rows: 2^3 as there are 3 vars

    x= abc + a'bc +abc'

    111 011 110
- **It also allows the convient *Sigma notation***
  - a Σ followed by the row numbers in which x=1 in the truth table
- **x(a,b,c) = Σ (3,6,7)**

	a b c	x
0	0 0 0	0
1	0 0 1	0
2	0 1 0	0
3	0 1 1	1
4	1 0 0	0
5	1 0 1	0
6	1 1 0	1
7	1 1 1	1

Sigma and Pi Notation

- **Since there is always a dual expression that is also true, there is also a dual of Sigma notation.**
  - This is called pi notation Π
  - The dual canonical expression for the previous e.g. is

    x(a,b,c) = (a+b+c)(a + b+ c')(a + b' + c)(a' + b + c)(a' + b + c')
- **This can be written using pi notation as**

  x(a,b,c) = Π (0,1,2,4,5)
- **Since it is the inverse of the sigma notation**
  - Pi notation refers to the row numbers in the truth table that contain zeros!
- **Sigma notation is a shorthand for canonical boolean expressions and their corresponding truth tables.**

Distance between Minterms

- **A canonical expression is convenient, but simply enumerates all cases where the combination of variables yields 1**
- **We now want to further reduce the number of AND terms and the number of variables to yield a more optimal circuit design.**
- **We can do this by minimizing the differences or distance between each AND term**
  - For example: *x = a'bc + abc + abc'*
  - *a'bc* and *abc* differ by 1 variable: a vs. a'
  - *a'bc* and *abc'* differ by 2 variables: a vs a' and c vs c'
- **So the distance between the latter minterms is 2**
  - We can reduce differences between minterms algebraically: x= a'bc + abc + abc'

    = a'bc + abc + abc + abc'

    = (a' + a)bc + ab(c + c') (idempotent)

    = bc + ab
  - In the latter expression distance is 1

Using Karnaugh Maps

- **A more convenient way to optimize distances is the Karnaugh Map**
- **It is a truth table arranged such that adjacent entries represent minterms that differ by 1**
- **To obtain a minimized expression, you rearrange the truth table such that distances match the cells**

	a b c	x	
	0 0 0	0	
	0 0 1	0	
	0 1 0	0	
	0 1 1	1	
differ 1	1 0 0	0	differ 2
differ 1	1 0 1	0	differ 2
	1 1 0	1	differ 1
	1 1 1	1	differ 1

For this, we need a gray coding

Gray Coding

- **In gray coding, the entries in the table are organized such that only one variable changes between rows.**

0 0		0 0	
0 1	as opposed to	0 1	
1 1		1 0	2 vars different
1 0		1 1	

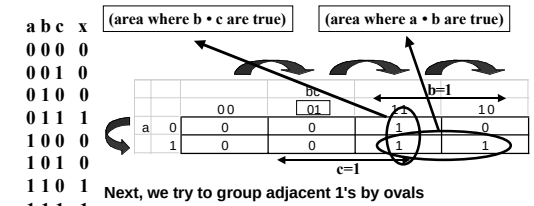
- **We can now construct a truth table in which only one variable changes between rows**
- **This is essentially a Karnaugh Map**

Constructing Karnaugh Map

x = a'bc + abc + abc'

so the minimized expression is *x= bc + ab*

a b c	x
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1

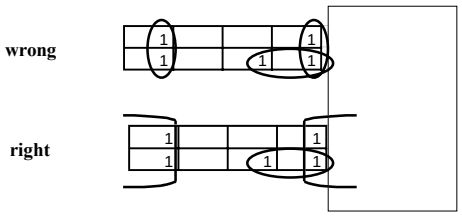


Next, we try to group adjacent 1's by ovals

- Grouping with ovals represent combining minterms that differ by one variable
- We can find the optimal expression by finding the min. set of ovals that cover all 1's
- Each oval covering as many 1's possible,
- Each oval represents an AND term!

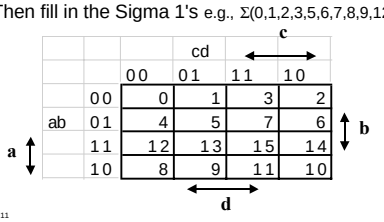
Constructing Karnaugh Maps (2)

- **Note that corners are allowed to be group with the opposite cells**



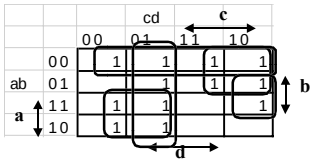
Four-variable Karnaugh Maps

- **Minimization of a four-variable network follows the same principle.**
  - Except the Karnaugh map has twice as many entries.
  - It's handy to label each cell in the table using the row numbers from the truth table.
  - Then fill in the Sigma 1's e.g., Σ(0,1,2,3,5,6,7,8,9,12,13,14)

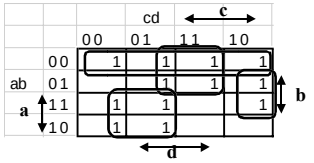


Four-var Karnaugh Maps (2)

- **wrong**



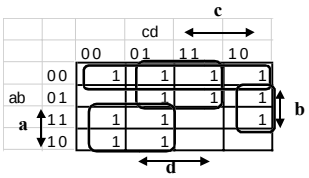
- **right**



may be multiple optimal solutions!

Solution

- **For every oval, find which variables keep same value within the area of that oval and group them.**
  - Within, one term, this can only be the variable OR its complement. So a'b is good, aa'b is wrong.



$$x = a'b' + ac' + a'd + bcd'$$

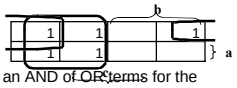
Dual Karnaugh Maps

- **Karnaugh Maps also have a dual expression.**
  - The aim is to get an expression that contains least number of gates
- **It is the same map, but now fill in 1's where the original truth table has zeroes**
  - Basically, turn the pi function of the original equation into a sigma function for its complement!

  $x(a,b,c) = ba + bc = \Sigma(3,6,7) = \Pi(0,1,2,4,5)$
  - Now minimize and find the OR of AND terms for the complement function

  $x'(a,b,c) = \Sigma(0,1,2,4,5)$

  $= b' + a'c'$

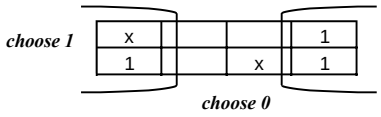


  - Invert these back again to get an AND of OR terms for the original equation.

    $= (b' + a'c')' = (b+c)(a+b) = b(a+c)$ only 2 gates!

Don't Care Conditions

- **In cases where it is irrelevant whether the output function for that combination of input values is zero or one**
  - You can put an x in the Karnaugh map.
- **Then, you can choose the x to be one or zero, whichever produces the most optimal oval map.**



Questions?