# Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 3 - Digital Logic

# Session Outline

Basic logic functions, expressions & truth tables

Logic gates, circuits, & finding the truth table

Simplifying logic expressions using Boolean algebra and Karnaugh maps

Common combinatorial, i.e., no memory, logic circuits

Sequential, i.e., with memory, logic circuits

Three state logic

Physical implementations

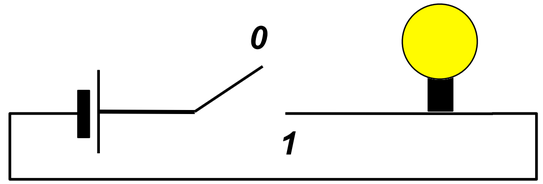
# Logic Gates, Circuits and Truth Tables

# Basic Logic Functions

Consider a simple switch: it has two positions:

OFF = 0,  ON = 1

We can use the switch to allow current to flow to light a bulb

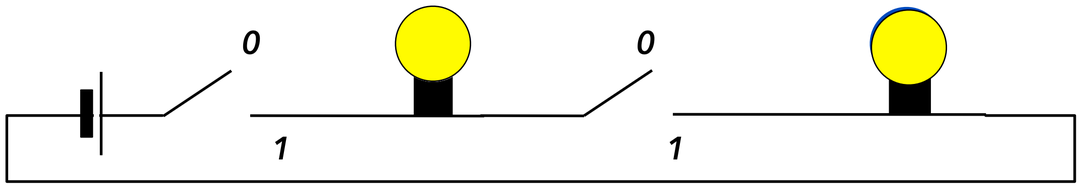



Consider f to indicate if the bulb is on, by denoting the switch to be A we can write:

f = A

# Switches in Series (AND)

Two switches can be used in series




The bulb is on if switch A AND switch B are both on

$f = A \cdot B$

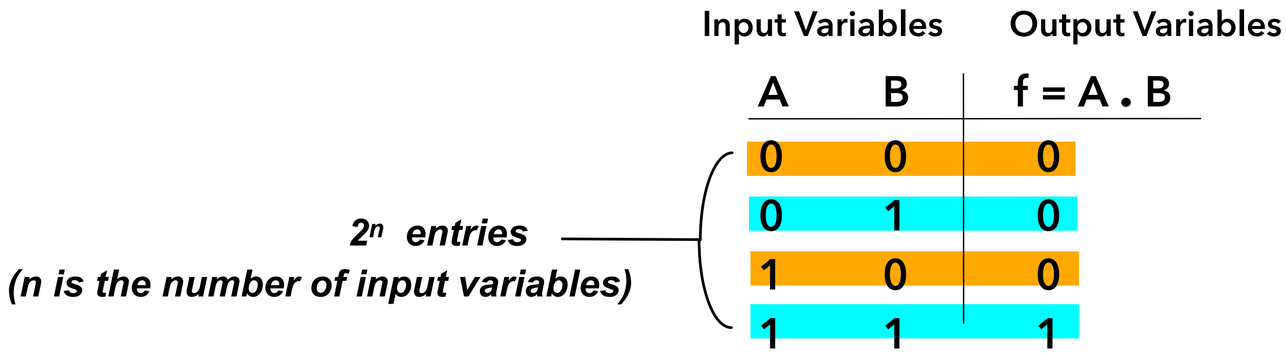
. is used to denote AND in Boolean algebra (also $\wedge$)

# Truth Tables - AND

A truth table defines the output values of a function in relation to ALL possible input variables.

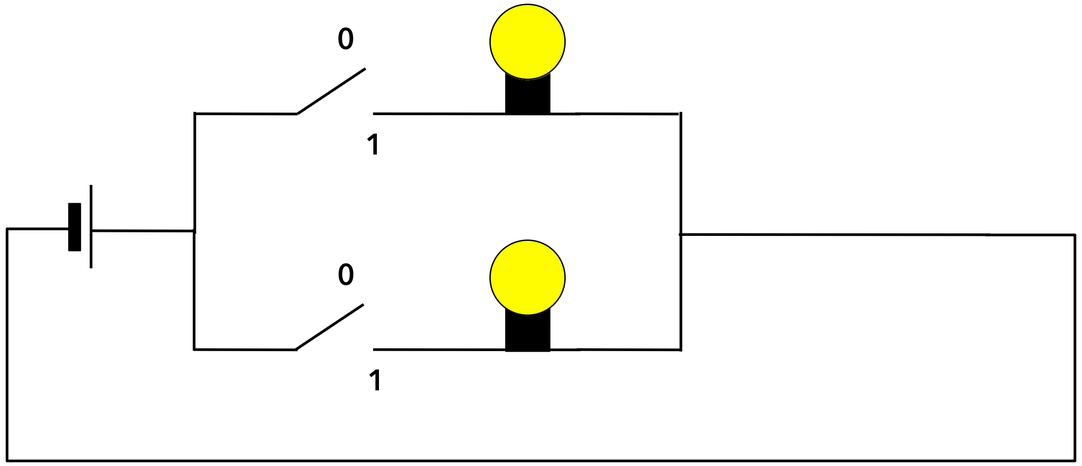
Each line in the truth table represents a UNIQUE value of the input variables.

The truth table for two switches in series f = A . B is shown below



# Switches in Parallel (OR)

We can also consider two switches in parallel




The bulb is on if switch A OR switch B are on

f = A + B

+ is used to denote OR in Boolean algebra (also ∨)

# Truth Tables - OR

We've seen the truth table for AND

Important that you appreciate the difference between the function of AND and OR

Inspecting their truth tables the most sensible way to do this

| Input Variables | | Output Variables |
|---|---|---|
| A | B | f = A . B |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| Input Variables | | Output Variables |
|---|---|---|
| A | B | f = A + B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Dots and Plus? Why?

AND written as . (or Λ)

OR written as + (or V)

Potential for confusion, since when we communicate otherwise “+” can mean “and” but in Boolean algebra “+” is OR

Notation is this way around because of mathematical properties

| ***Algebra of real numbers*** | | ***Boolean algebra*** |
|---|---|---|
| ***Multiply x by 1 (x.1), get x*** | ⟷ | **AND x with 1 (x.1), get x** |
| **Add x to 0 (x+0), get x** | ⟷ | **OR x with 0 (x+0), get x** |

# An Alternative View of OR

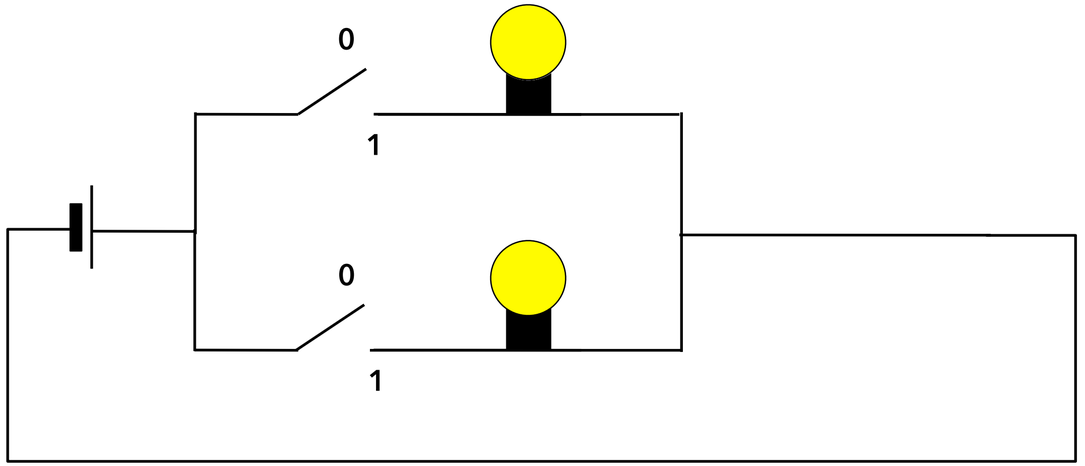
If (switch A is NOT on) AND (switch B is NOT on) then the bulb is NOT on

This is $\overline{f} = \overline{A} \cdot \overline{B}$ where the bar over each letter is negation in Boolean algebra

We know that $\overline{f} = \overline{A + B}$

This means that $f = A + B$ (negate both sides)

Therefore $\overline{f} = \overline{A} \cdot \overline{B}$ (this is de Morgan's law)

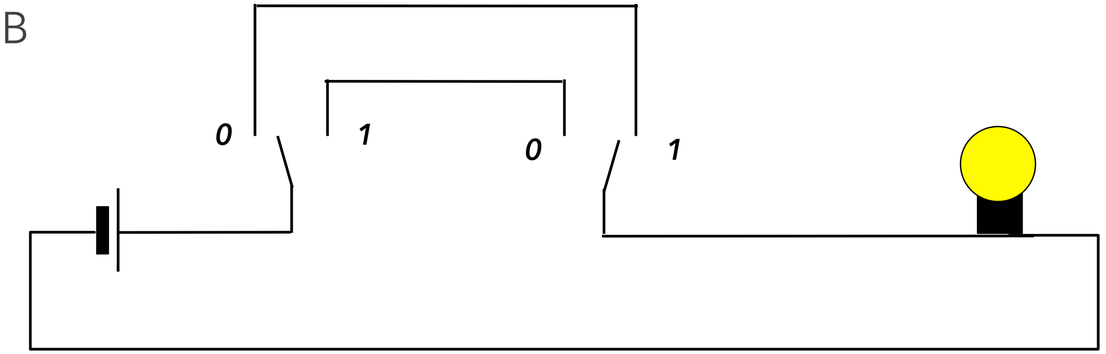


# Exclusive OR (EX-OR)

One more important logical function

The bulb is on when only one switch is on

$f = A \oplus B$




| Input Variables | | Output Variables |
|---|---|---|
| A | B | f = A ⊕ B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

⊕ is used to denote EX-OR in Boolean Algebra

EX-OR is often referred to as an Equivalence relation (i.e. the output is zero if both inputs are equivalent)

# Functions of Two Binary Variables

16 ( = $2^4$) functions can be found between two binary variables:

We can interpret each of these functions algebraically

$f_0 = 0$

$f_1 = A \cdot B$

$f_2 = A \cdot \overline{B}$

$f_3 = A$

$f_{13} = \overline{A \cdot \overline{B}}$

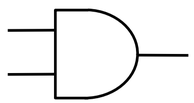
| A | B | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | $f_{12}$ | $f_{13}$ | $f_{14}$ | $f_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

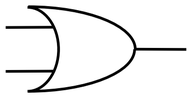
# Electronic Logic Gates

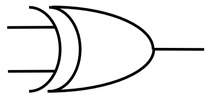
There are seven electronic logic gates whose function you should understand

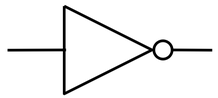
You will definitely be familiar with AND, OR, EX-OR and NOT

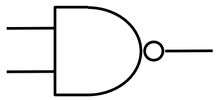
You should try to construct the truth tables for NAND, NOR and EX-NOR

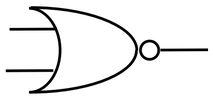

AND

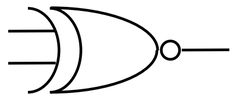

OR


EX-OR


NOT


NAND


NOR

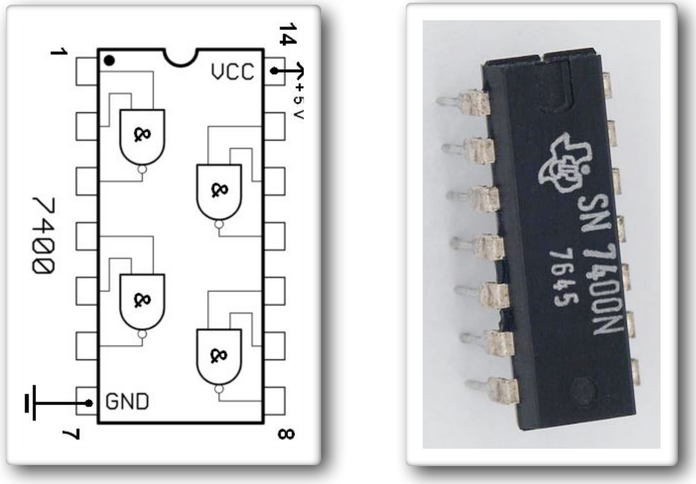

EX-NOR

# Generations of Computer Systems

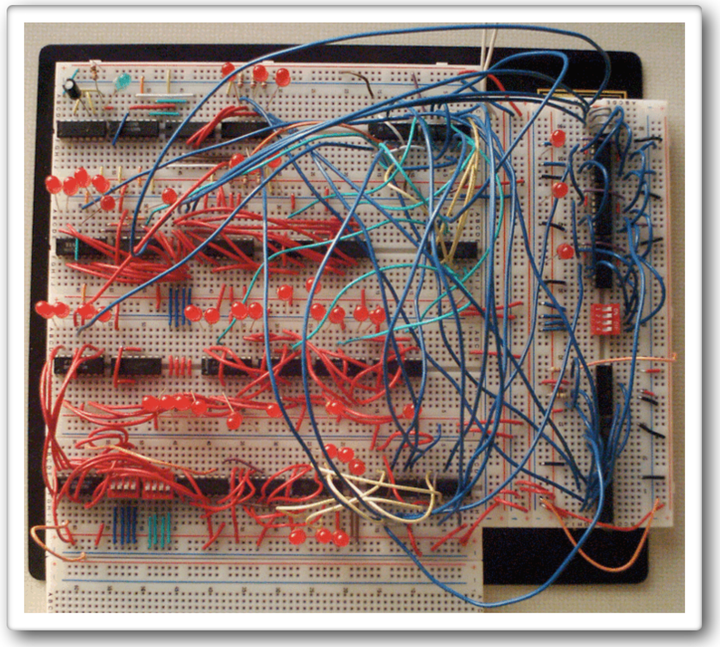
| Generation | Approximate Dates | Technology | Typical Speed (operations per second) |
|---|---|---|---|
| 1 | 1946–1957 | Vacuum tube | 40,000 |
| 2 | 1958–1964 | Transistor | 200,000 |
| 3 | 1965–1971 | Small and medium scale integration | 1,000,000 |
| 4 | 1972–1977 | Large scale integration | 10,000,000 |
| 5 | 1978–1991 | Very large scale integration | 100,000,000 |
| 6 | 1991- | Ultra large scale integration | 1,000,000,000 |

Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

# Physical Packaging of Logic Gates

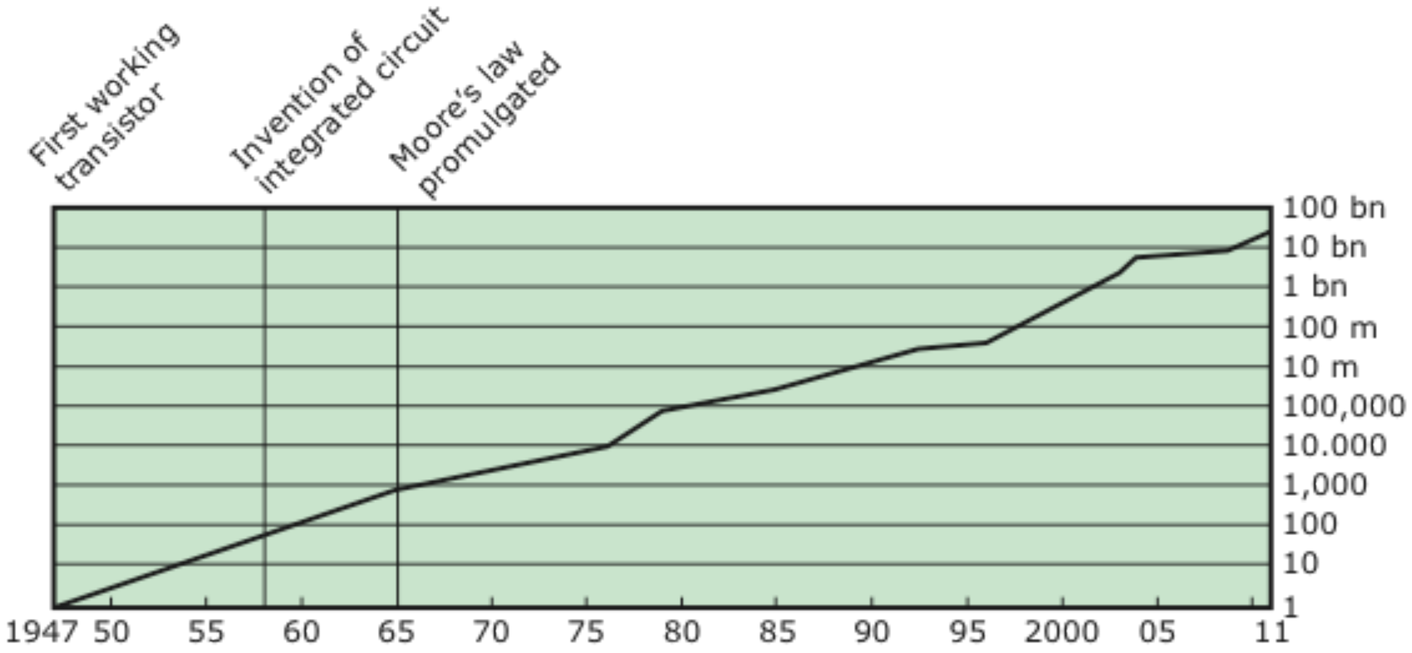



*A 7400 chip: 4 NAND gates*



*A 4-bit two register computer built entirely from 74-series chips*

# Moore’s Law




Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

# Moore’s Law

In 1965 Gordon Moore, co-founder of Intel, predicted the number of transistors on a chip would double every 12-18 months

The consequences of Moore’s law are profound:

- The cost of computer logic and memory circuitry has fallen at a dramatic rate
- As logic and memory is placed closer together on more densely packed chips, the electrical path is shortened, increasing operating speed
- Computers becoming smaller, making them more convenient in a variety of environments, especially embedded systems
- Reduction in power and cooling requirements
- Interconnections on ICs are more reliable than solder connections so having fewer off-chip connections increases reliability

# Combinatorial Logic Circuits

A logic circuit whose output is a logical function of its input

Consider the EX-OR function

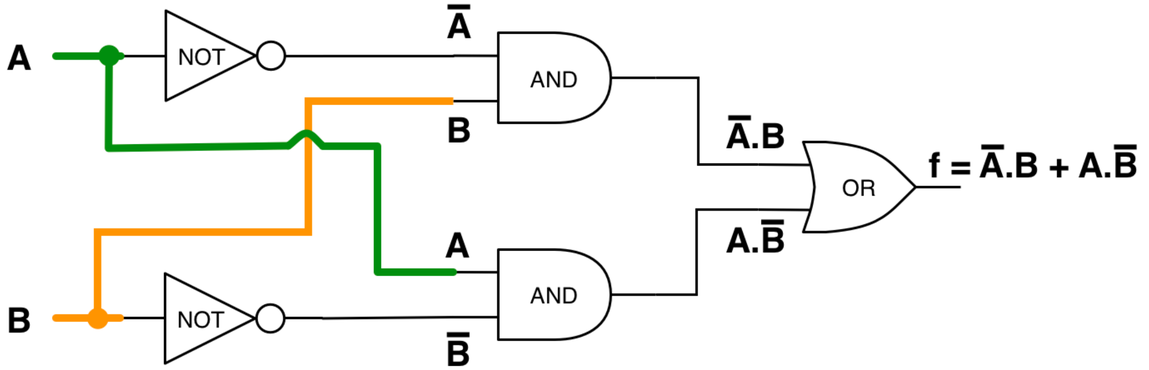
$$f = A \oplus B$$

From the truth table we can write:

$$f = \overline{A} . B + A . \overline{B}$$

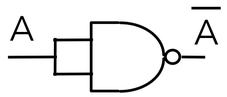
The logic circuit to the right implements the EX-OR function

| Input Variables | | Output Variables |
|---|---|---|
| A | B | $f = A \oplus B$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

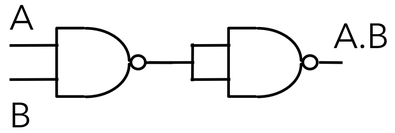


# NAND Gates

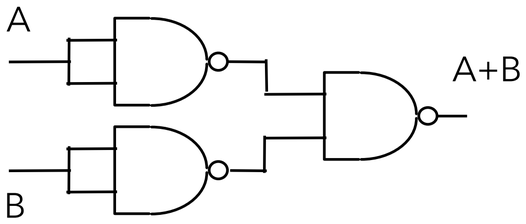
NOT $f = \overline{A} = \overline{A.A}$




AND $f = A.B = \overline{\overline{A.B}}$



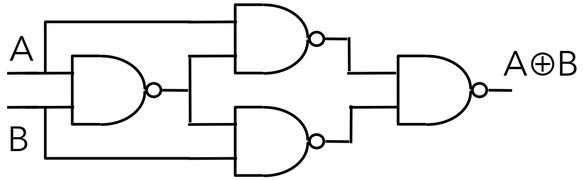

OR $f = A+B = \overline{\overline{A}.\overline{B}}$




EX-OR $f = \overline{A}.B + A.\overline{B}$

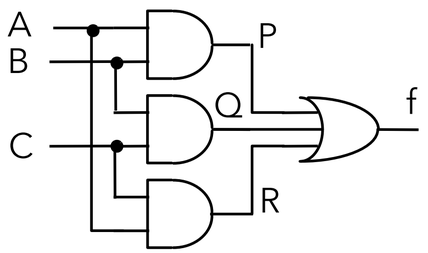
Can you work out why this is correct?

Diagram ? Algebra? Maps?



# Logic Circuit to Truth Table

We can construct a truth table for a given logic circuit




***Consider intermediate signals P, Q, R***

***P = A.B, Q = B.C, R = A.C***

***The output is f = A.B + B.C + A.C***

| A | B | C | P | Q | R | f |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

This is an important skill and will help you understand logic circuits, truth tables and how we can manipulate logical expressions

You should try creating logic circuits and deriving their truth tables

# Simplifying Logical Expressions

Logic expressions can be simplified to reduce complexity

- Reduces number of gates required for their implementation

Two mains ways to simplify logical expressions

- Boolean algebra (the manipulation of an algebraic representation)
- Karnaugh Maps (simplifying the “sum of products” form graphically)

Using either method the truth table for the logical expression before and after minimisation must be identical

# Simplifying Expressions Using Boolean Algebra

# Circuit Equivalence

We have many concerns when implementing logic circuits

- Circuits with more gates are more expensive
- The gates available depends upon the physical implementation, e.g., we may only want use NAND gates or we may have some 4-input gates

When given a function we need to be able to create equivalent circuits to meet several design criteria

- Perform the designated function
- Use the types of gate available
- Minimise the number of gates used and hence cost

A Boolean function can have many different but equivalent Boolean expressions

# Truth Table to Boolean Equation

The sum of products form can be obtained directly from a truth table

| A | B | C | f | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | $\bar{A}.\bar{B}.\bar{C}$ |
| 0 | 0 | 1 | 1 | $\bar{A}.\bar{B}.C$ |
| 0 | 1 | 0 | 0 | |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | $A.\bar{B}.\bar{C}$ |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | $A.B.\bar{C}$ |
| 1 | 1 | 1 | 1 | $A.B.C$ |

***So function f is TRUE when any of these combinations are TRUE***

$$f = \bar{A}.\bar{B}.\bar{C} + A.\bar{B}.\bar{C} + A.\bar{B}.\bar{C} + A.B.\bar{C} + A.B.C$$

We consider this to be a sum of products because an OR '+' is considered as a sum and an AND '.' as a product

# Laws of Boolean Algebra

Boolean algebra obeys a convenient set of laws that permit simplification

| Name | AND form | OR form |
|---|---|---|
| Identity law | $1A = A$ | $0 + A = A$ |
| Null law | $0A = 0$ | $1 + A = 1$ |
| Idempotent law | $AA = A$ | $A + A = A$ |
| Inverse law | $A\overline{A} = 0$ | $A + \overline{A} = 1$ |
| Commutative law | $AB = BA$ | $A + B = B + A$ |
| Associative law | $(AB)C = A(BC) = ABC$ | $(A + B) + C = A + (B + C) = A + B + C$ |
| Distributive law | $A + BC = (A + B)(A + C)$ | $A(B + C) = AB + AC$ |
| Absorption law | $A(A + B) = A$ | $A + AB = A$ |
| De Morgan's law | $\overline{AB} = \overline{A} + \overline{B}$ | $\overline{A + B} = \overline{A}\,\overline{B}$ |

# Remembering the Laws of Boolean Algebra

Can generate OR form from AND form, or vice-versa:

Swap AND ⇔ OR (‘.’ ⇔ ‘+’)

Swap 0 ⇔ 1

Most laws are intuitive but several are less intuitive

Distributive (AND form only)

Absorption

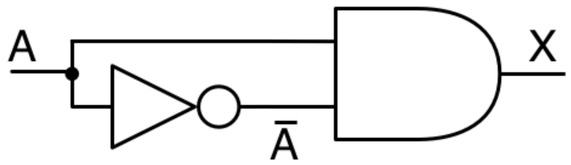
de-Morgan’s

# Checking the Inverse Law

Not sure if you've remembered one correctly? Check it with a truth table!

For example, we can construct a truth table for the inverse law

$$A.\overline{A} = 0$$




| A | $\overline{A}$ | X |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |

*… this is what we expected*

# De Morgan's Law - Algebraically

To change the form of expression:

1. Break the negation bar
2. Change AND to OR and vice versa

$$\overline{A \cdot B} = \overline{A} + \overline{B} \qquad \overline{A + B} = \overline{A} \cdot \overline{B}$$

***EXAMPLES:***

$$(\overline{A \cdot (\overline{C + D})}) = \overline{A} + (\overline{\overline{C+D}}) = \overline{A} + C + D$$

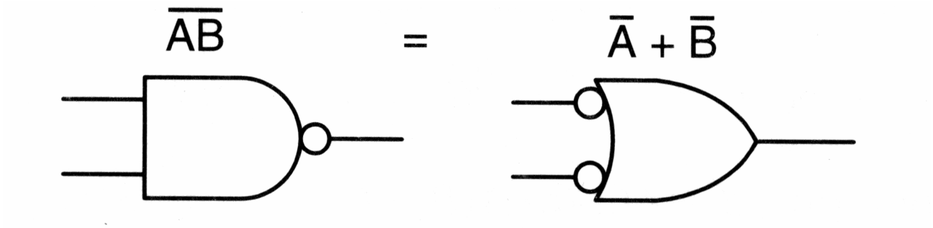
$$(\overline{A \cdot C}) + D = (\overline{A} + \overline{C}) + D$$

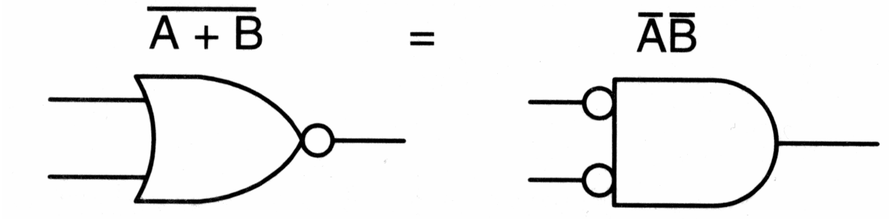
$$(\overline{A \cdot B}) + (\overline{C \cdot D}) = \overline{A} + \overline{B} + \overline{C} + \overline{D}$$

# De Morgan's Law - Graphically

For many people it makes sense to think about De Morgan's law graphically

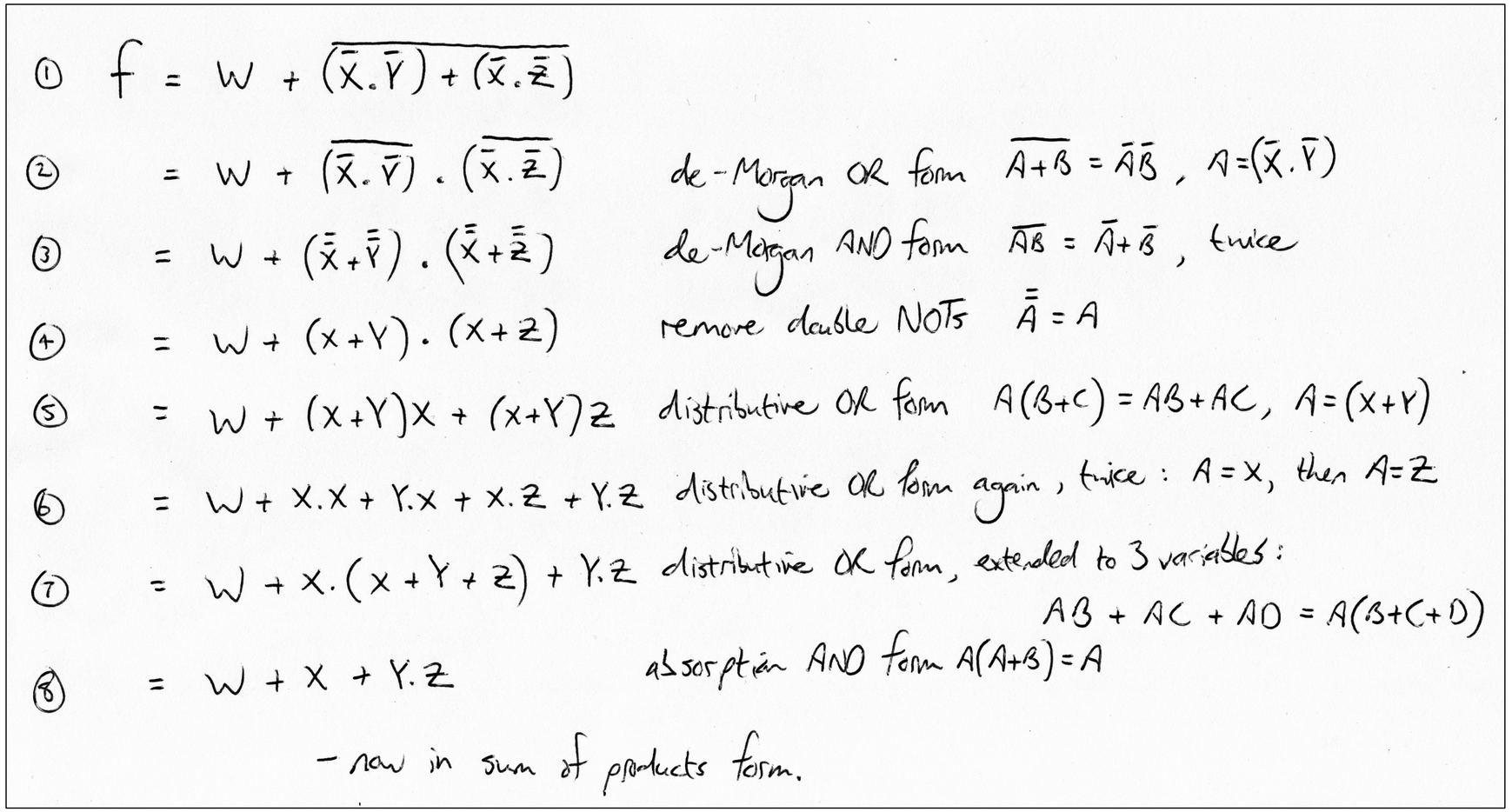
Consider the equivalence of the logic circuits involved



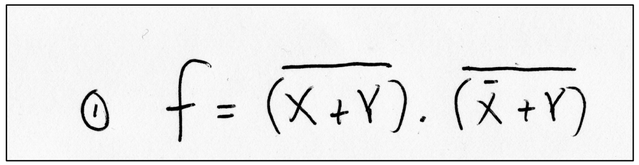
# Simplification Using Boolean Algebra - Q1

First I will simplify an expression for you...



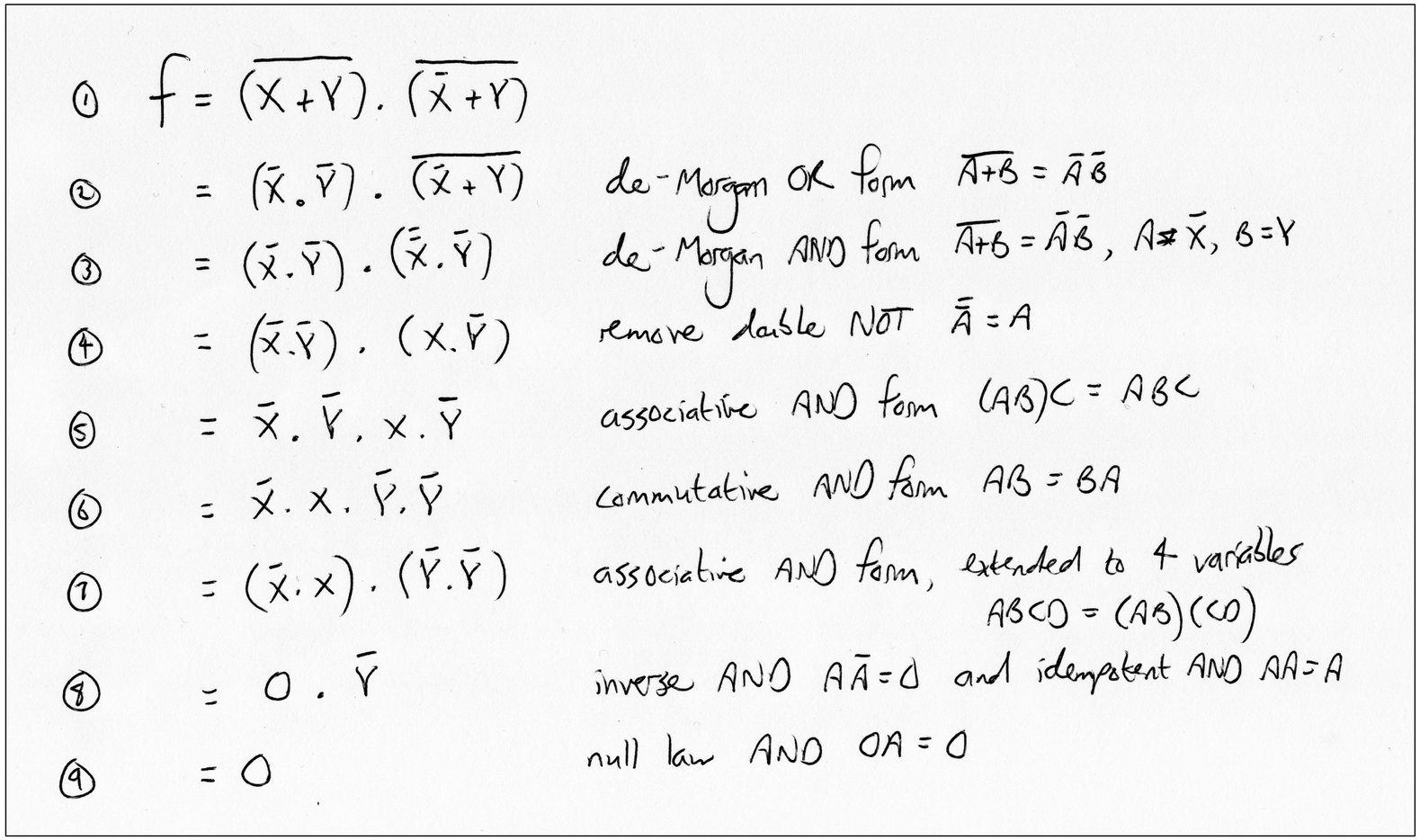
# Simplification Using Boolean Algebra - Q2

Try simplifying this…



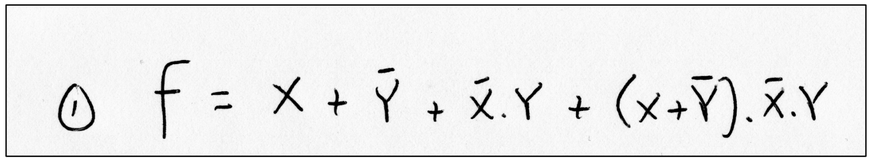
# Simplification Using Boolean Algebra - Q2

Try simplifying this…



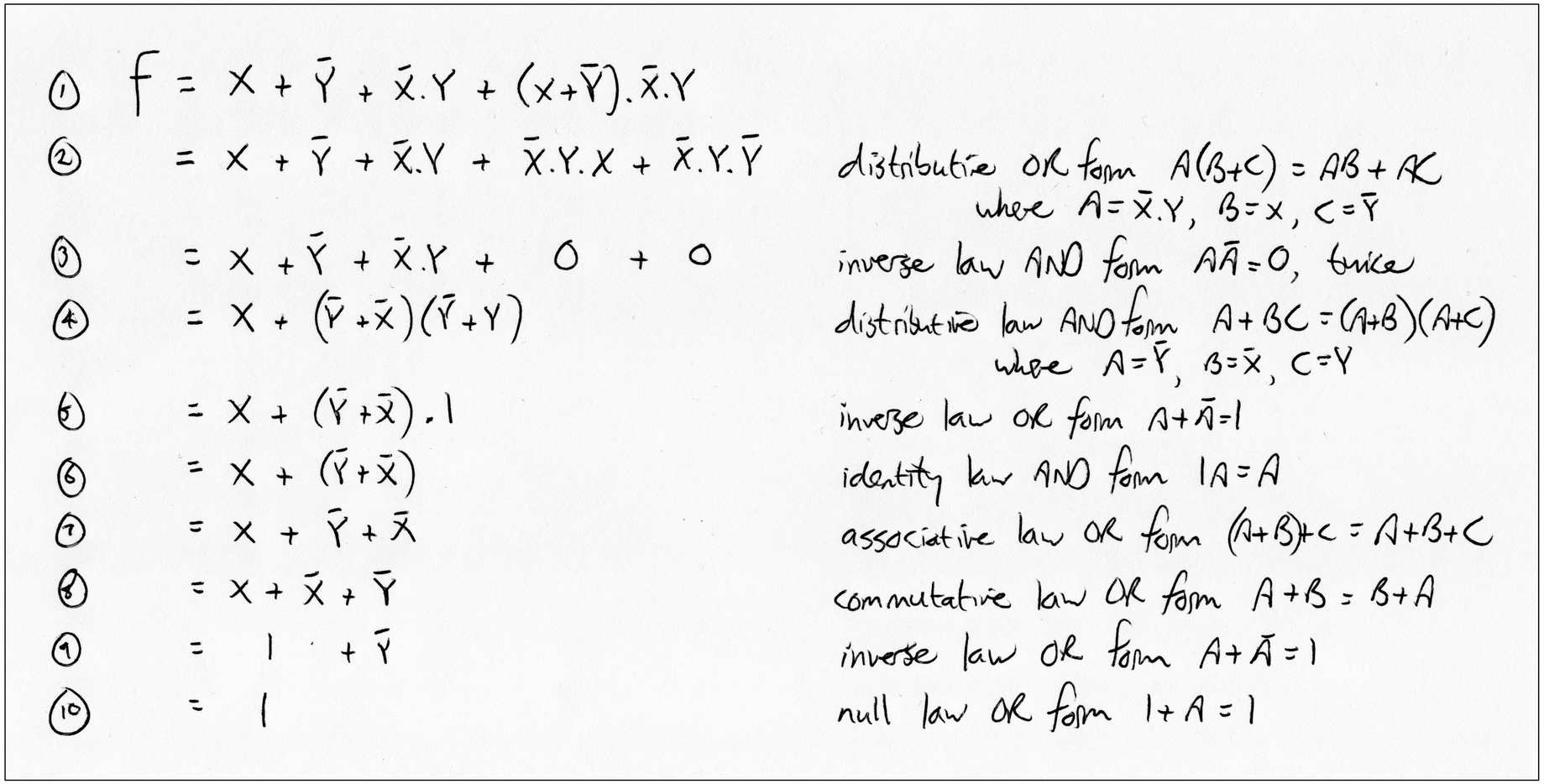
# Simplification Using Boolean Algebra - Q3

Now try…



# Simplification Using Boolean Algebra - Q3

Now try…



# Simplification Using Boolean Algebra - Try These

$$\overline{\overline{X.Y} \; \overline{X.Z}}$$

$$(X + \overline{Y}).(\overline{X} + Z).(Y + \overline{Z})$$

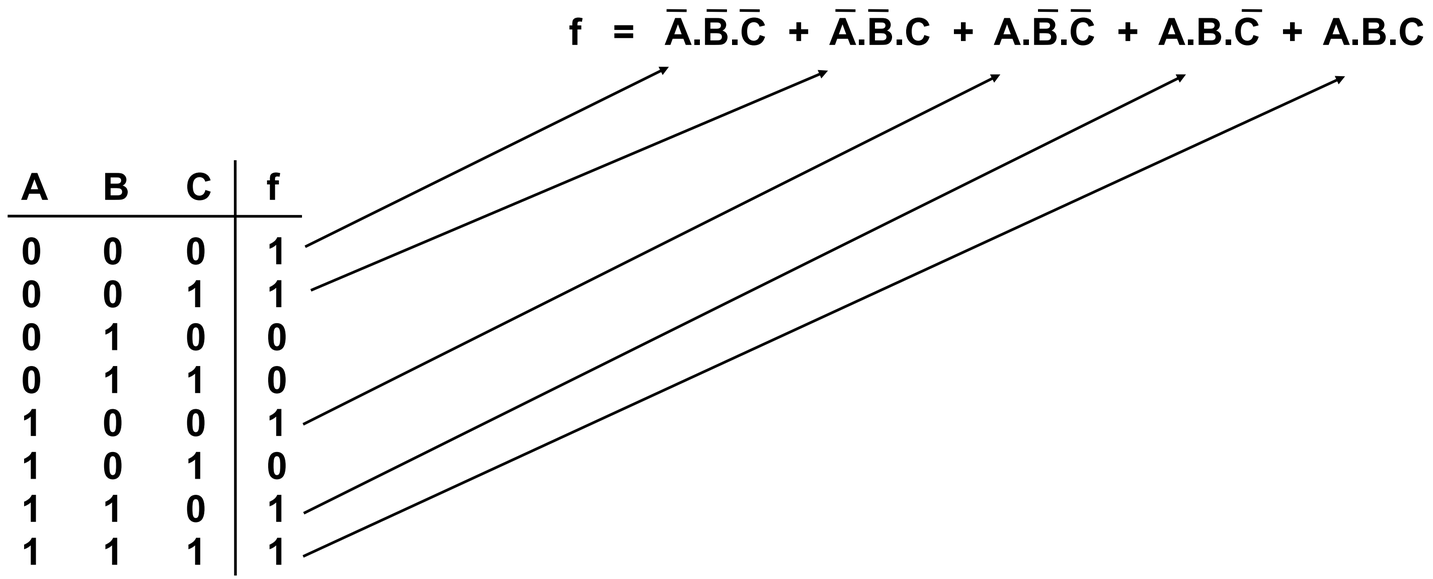
$$(X + Y + Z).(\overline{X} + Y + Z).(\overline{X} + Y + \overline{Z})$$

# Simplifying Expressions Using Karnaugh Maps

# Sum of Products

Recall that the sum of products form can be obtained from a truth table

The sum of products is often used in the simplification of logical expressions



# Simplifying a Sum of Products

Sum of products form can be simplified by looking for terms that differ by only one variable and its complement

$$\overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C = \overline{A}.\overline{B}.(\overline{C} + C) \quad \text{Distributive law}$$

$$= \overline{A}.\overline{B} \quad \text{Inverse law}$$

When using algebraic techniques can be difficult to tell whether an equation is in simplest form or to see the next step

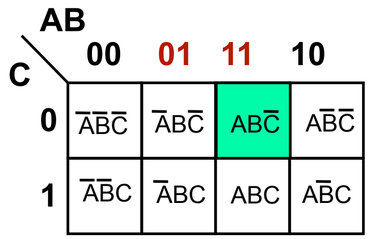
Karnaugh maps show unambiguously when a Boolean expression is in its simplest form

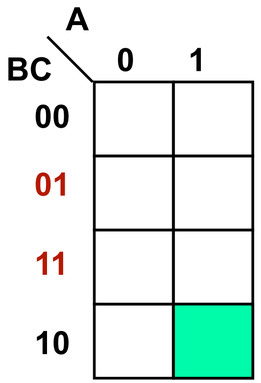
# Karnaugh Maps (K-maps)

2D truth tables

See the simplest Boolean expression of a few variables from the location of 1s on the map

A three variable function can be represented by a 4 x 2 rectangle or a 2 x 4 rectangle

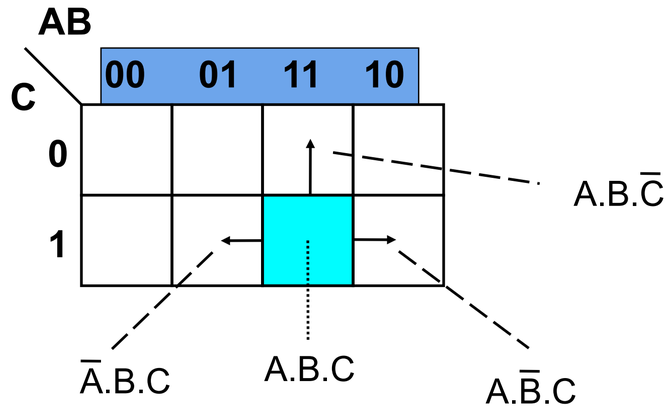



in both maps stands for $A.B.\overline{C}$

# Adjacency in a Karnaugh Map

Adjacent squares differ by exactly one variable




***NOTE : Numbering on the edge of the Karnaugh Map - "Gray coded" for inventor Frank Gray***

# Example Karnaugh Map

Simplification the graphical way

Important to try a lot of examples to develop your simplification skills

$$f = A.B.\overline{C}.D + A.\overline{B}.\overline{C}.D + \overline{A}.\overline{B}.C.D + \overline{A}.B.C.D$$

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

# Karnaugh Map Grouping

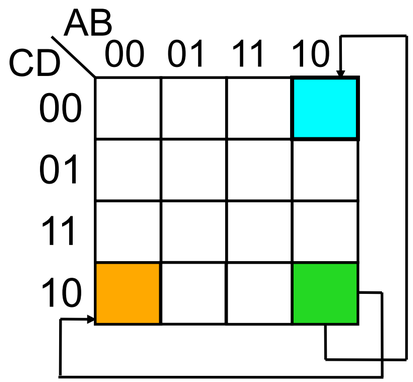
Wrap-around

Left most-column is adjacent to right most-column, and top row is adjacent to bottom row

Groupings may overlap

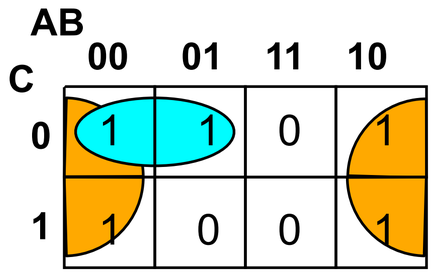
Minimum logic expression is obtained on minimum number of groupings

Number of elements in the group must be a power of two, i.e., 1,2,4,8 etc...

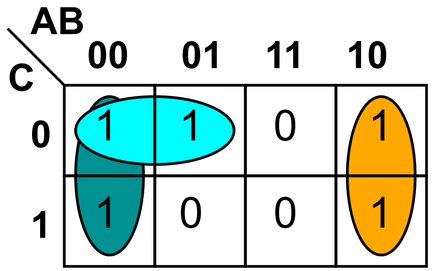


# Karnaugh Maps - Example 1

Essentially we're looking for the smallest number of groupings that capture a logical expression




$\Rightarrow \quad f = \overline{B} + \overline{A}.\overline{C}$



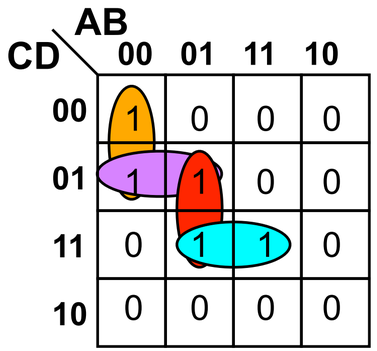

$\Rightarrow \quad f = \overline{A}.\overline{B} + A.\overline{B} + \overline{A}.\overline{C}$

Correct, but not minimal. The top result was simpler - don't forget to wrap around!

# Karnaugh Maps - Example 2

It’s possible to have more than one equivalent expressions

Overlap in groups permits equivalence




$\bar{A}.\bar{B}.\bar{C}$

$\bar{A}.\bar{C}.D$

$\bar{A}.B.D$

$B.C.D$

**Two possibilities :** $f = \bar{A}.\bar{B}.\bar{C} + \bar{A}.B.D + B.C.D$

**or**

$f = \bar{A}.\bar{B}.\bar{C} + \bar{A}.\bar{C}.D + B.C.D$

**Can result in more than one equivalent logic expression**

# Karnaugh Maps - Example 3

Sometimes it's not possible to minimise a logical expressions

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

**Cannot get any groupings ⇒ no expression minimisation possible**

**This is actually an EX-OR function between the 4 input variables:**

$$f = (A \oplus B) \oplus (C \oplus D)$$

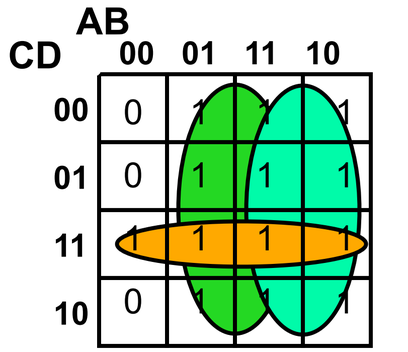
# Minimising Complex Logical Expressions

Seemingly complex logical expressions often simplify to expressions that you could implement using very few integrated circuits

$$f = A + \overline{(\overline{B} . \overline{C})} + \overline{(\overline{B} . \overline{D})}$$

**Construct Truth Table:**

| A | B | C | D | $\overline{B}.\overline{C}$ | $\overline{B}.\overline{D}$ | $\overline{(\overline{B}.\overline{C})+(\overline{B}.\overline{D})}$ | f |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |



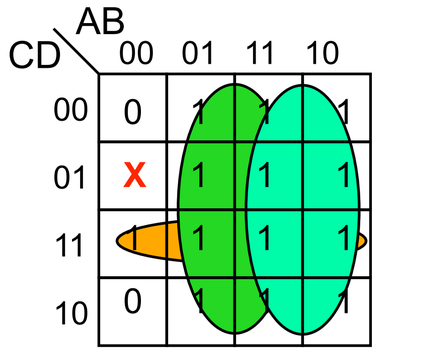

$$f = B + A + C.D$$

# Don’t Care Conditions

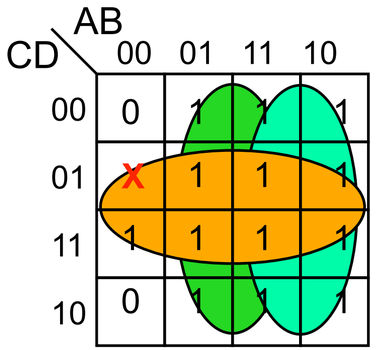
Sometimes a certain combination of inputs can’t happen, or if it does, we don’t care what the output is

An “**X**” is used to denote these “don’t care” conditions, which may be assumed to be either 1 or 0

Don’t care condition can allow us to create a simpler logic expression




f = A + B + C.D




f = A + B + D

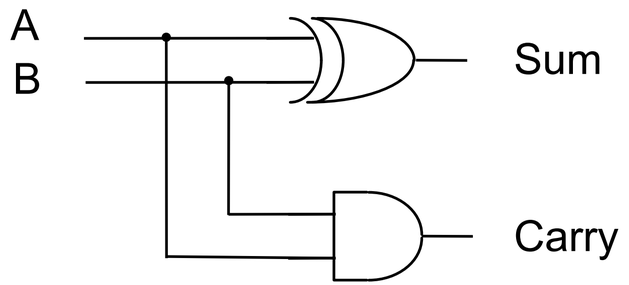
# Combinatorial Logic Circuits

# 1-bit Half-Adder

The 1-bit half-adder performs the addition of two bits

Can be extended to form a 1-bit full-adder

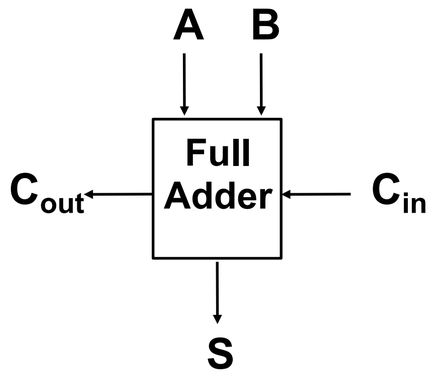
| A | B | Sum | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



# 1-bit Full-Adder

Performs the addition of three bits

Two significant bits and a previous carry




| $C_{in}$ | A | B | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

*Inputs - A,B, $C_{in}$ (each one bit)*

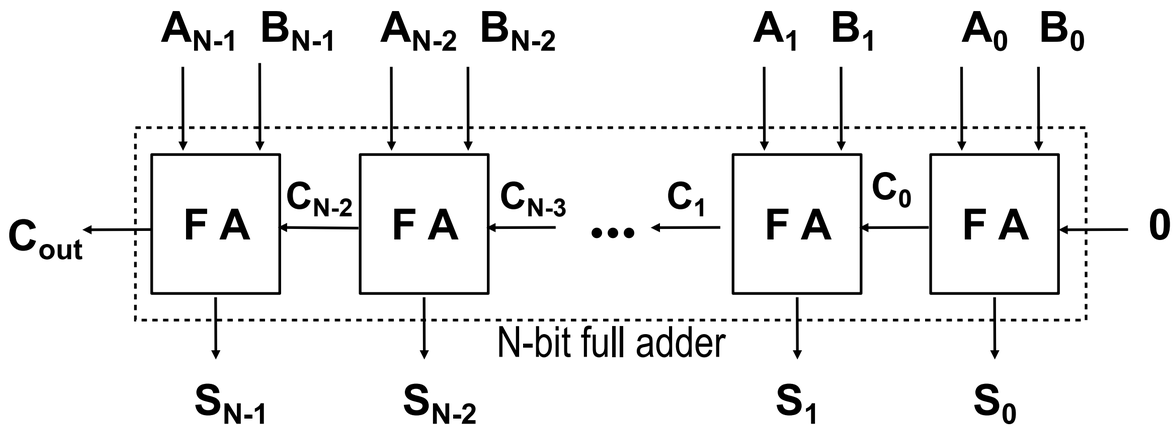
*Outputs - S, $C_{out}$ (each one bit)*

*Homework: Design a circuit of a one-bit full adder*

# N-bit Full-Adder

A combination of what we've already seen

The carry output from one 1-bit full-adder is the carry input to the next 1-bit full-adder in all but the first case



# Adder to Adder/Subtractor

To convert an adder into an adder/subtractor we simply add a control input Z such that:

$Z = 0 \Rightarrow S = A + B$

$Z = 1 \Rightarrow S = A - B$

| Z | B | Z ⊕ B | |
|---|---|---|---|
| 0 | 0 | 0 | } =B |
| 0 | 1 | 1 | |
| 1 | 0 | 1 | } $=\overline{B}$ |
| 1 | 1 | 0 | |

Remember that A - B = A + (-B)

We can calculate (-B) using two's complement

1. Invert the N-bit binary number B by finding Z ⊕ B
2. Add 1 (Carry In)

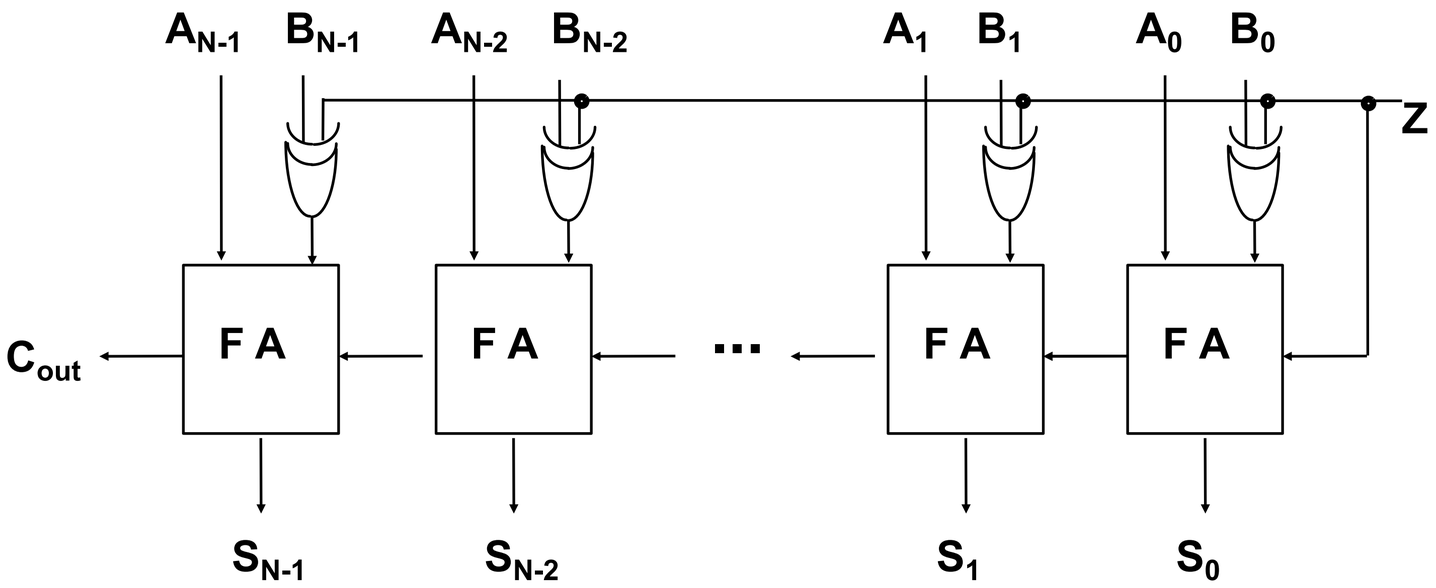
# N-bit Adder/Subtractor

A reasonably complex combinatorial circuit

One of the more interesting circuits we've seen with respect to its function

Try to see how this works for yourself!

What does the final carry indicate?




Z = 0 ⇒ Adder

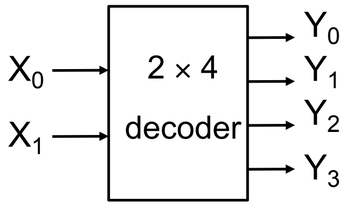
Z = 1 ⇒ Subtractor

# Decoders

Decoder has, e.g., binary input pins, and one output pin per possible input state

Activate one output pin corresponding to current input state

Consider decoding 2 inputs to 4 unique outputs




| $X_0$ | $X_1$ | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

***Note that the two inputs select which output is active***
***The system is active low, i.e., active = 0, in this case***

Decoders are often used to address unique memory locations in a microprocessor system

# Active High or Active Low?

In some circuit applications, outputs and inputs have “active” and “inactive” states.

Important that inputs and outputs conform to the same standard

Active high

0 = inactive, 1 = active

Active low

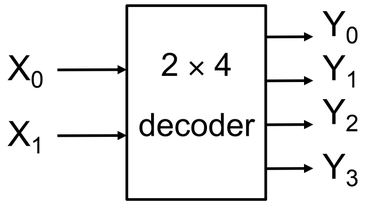
0 = active, 1 = inactive

Pin sometimes labelled with a bar, e.g., $\overline{\text{Enable}}$

# Active High Decoder

Consider a 2 to 4 line decoder with active high output

Observe the difference between in the 2 to 4 line decoder with active low output we saw previously




| $X_0$ | $X_1$ | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

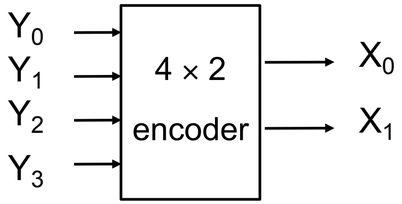
# Encoders

Encode a set of inputs into a defined representation on the outputs (note that this is the reverse of a decoder)

- Multiple input pins, only one of which should be active at a time
- Smaller number of output pins
- Circuit puts code, e.g., binary, of currently active input onto output pins

Consider encoding 4 inputs to 2 outputs

- Does the encoder shown below have active high or active low inputs?



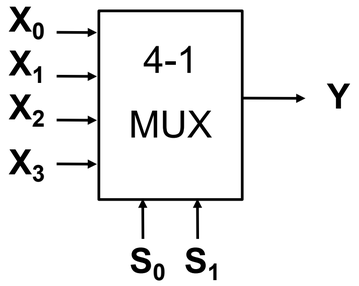

| $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ | $X_1$ | $X_0$ |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

***Encoders are often used as simple input circuits - though this is limited because you couldn't have 2+ keys held down***

# Multiplexers (MUX)

Multiplexer - Output is a selected input.

Consider a 4-1 multiplexer (four inputs to one output)

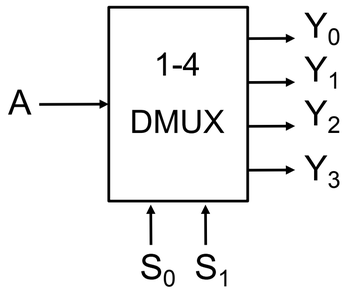



| $S_0$ | $S_1$ | Y |
|---|---|---|
| 0 | 0 | $X_0$ |
| 0 | 1 | $X_1$ |
| 1 | 0 | $X_2$ |
| 1 | 1 | $X_3$ |

$$Y = X_0.\bar{S}_0.\bar{S}_1 + X_1\,\bar{S}_0.S_1 + X_2\,S_0.\bar{S}_1 + X_3.S_0.S_1$$

# De-Multiplexers (DE-MUX)

Allow an input to appear on any one of the outputs (note that this is the reverse of a multiplexer)




| $S_0$ | $S_1$ | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|
| 0 | 0 | A | 1 | 1 | 1 |
| 0 | 1 | 1 | A | 1 | 1 |
| 1 | 0 | 1 | 1 | A | 1 |
| 1 | 1 | 1 | 1 | 1 | A |

***A 1 represents inactive: therefore active low***

$$Y_0 = A + \overline{S}_0.S_1 + S_0.\overline{S}_1 + S_0.S_1$$

$$= A + \overline{S}_0.S_1 + S_0.(\overline{S}_1+S_1)$$

$$= A + \overline{S}_0.S_1 + S_0$$

$$= A + S_1 + S_0$$

***Homework exercise: what are expressions for $Y_1$, $Y_2$, $Y_3$?***

# Common Applications of Multiplexers

Source selection control

- Home stereo, e.g., send iPod, CD or radio to speakers – note that this is analogue not digital

Share one communication line between multiple senders

- Requires both MUX and DE-MUX

Parallel to serial conversion

- Parallel input on X, clock signal on S, serial output on Y

Circuit that can be configured to produce any truth table relationship between S inputs and Y output

- Set up the truth table required on X inputs

# Common Applications of De-Multiplexers

Share one communication line between multiple senders

- Requires both MUX and DE-MUX

Serial to parallel conversion

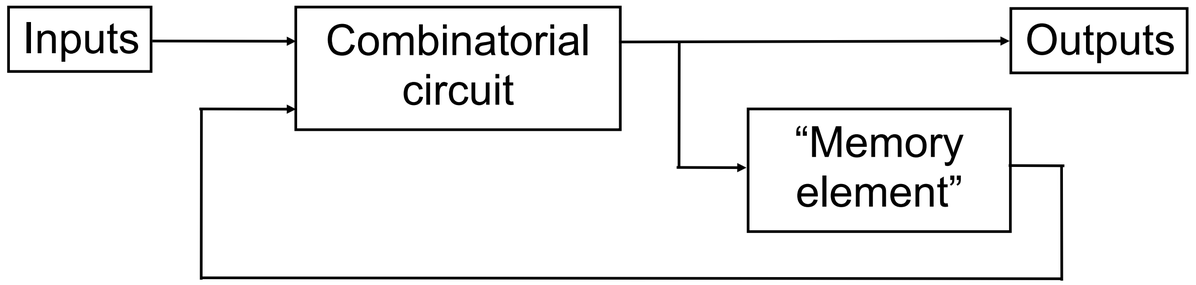
A control for multiple lights

- In a gambling machine you might connect a processor to A and S and connect Y outputs to lights, such that the processor runs in a rapid loop addressing each light sequentially

# Sequential Logic Circuits

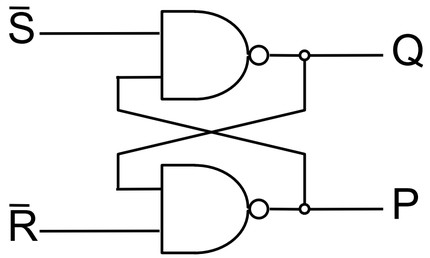
# Sequential Logic

A logic circuit whose **outputs** are logical functions of its **inputs** and its **current state**



# Flip-Flops

A circuit whose outputs are fed back to inputs




| A | B | $\overline{A.B}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$\overline{S}$ and $\overline{R}$ are active low **SET** and **RESET** inputs

Consider: $\overline{S} = 1, \overline{R} = 0 \Rightarrow P = 1, Q = 0$

then if $\overline{S} = 1, \overline{R} = 1 \Rightarrow P = 1, Q = 0$

and then $\overline{S} = 0, \overline{R} = 1 \Rightarrow P = 0, Q = 1$

then $\overline{S} = 1, \overline{R} = 1 \Rightarrow P = 0, Q = 1$

# Flip-Flop Operation

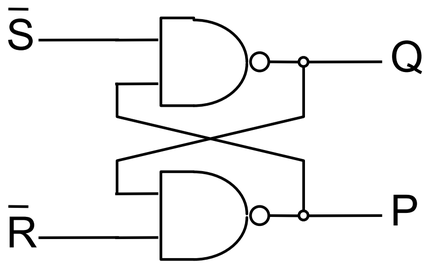
Q is Set (to one) when $\overline{S} = 0$ (and $\overline{R} = 1$)

Q is Reset (to zero) when $\overline{R} = 0$ (and $\overline{S} = 1$)

If $\overline{S} = \overline{R} = 1$ then Q does not change

We can capture this behaviour in a truth table

| $\overline{S}$ | $\overline{R}$ | Q | P | |
|---|---|---|---|---|
| 0 | 0 | X | X | ← **hazard condition** |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | |
| 1 | 1 | **no change** | | |



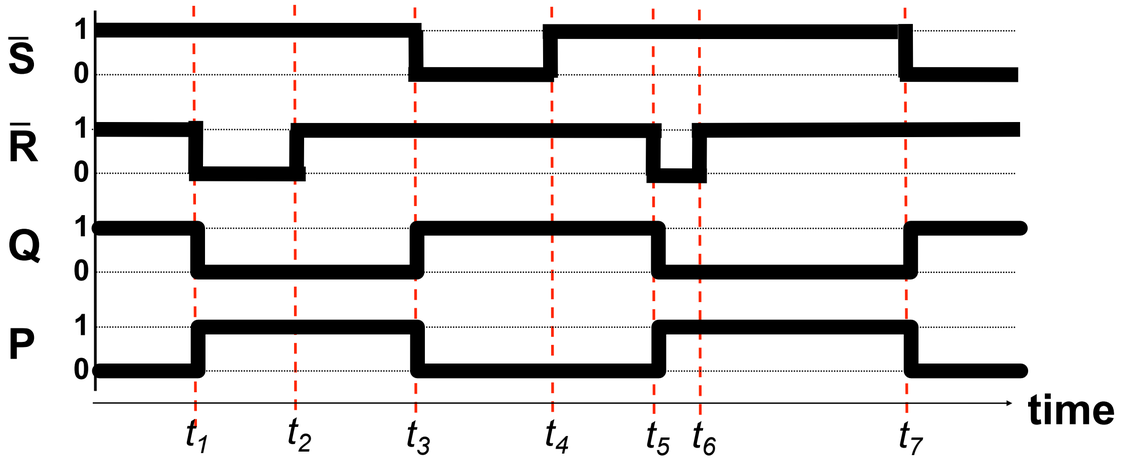

The flip-flop is able to **STORE** a value when both $\overline{S} = \overline{R} = 1$

# Flip-Flop Timing Diagram

We can also visualise the operation of a flip-flop in a timing diagram

Flip-flop output is not just dependent on the input, as in combinatorial logic circuits, but also on its previous state

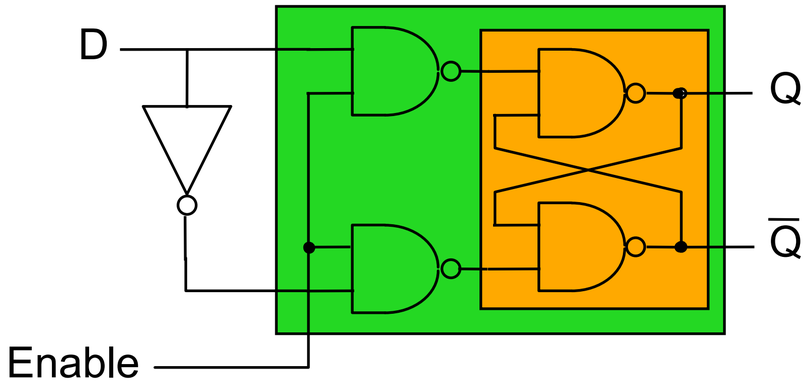
You should step through the timing diagram to gain an understand of how such a circuit provide useful function



# D-Type Latch

Using a modified form of the flip-flop circuit we've seen it is possible to design the D-type latch

Essentially a 1-bit memory circuit




| Enable | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Q | $\overline{Q}$ |
| 0 | 1 | Q | $\overline{Q}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Output can change only when the Enable line is high, i.e., enable triggers circuit operation.

D-type latch is a 1-bit memory with the Enable terminal as the 'WRITE' command.

# D-Type Latch Truth Table

The truth table for the d-type latch can be expressed in several different, yet equivalent, ways

Observe that “don’t care” conditions can be use to reduce complexity but further abstraction makes things even simpler

| Enable | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | x | Q | $\overline{Q}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

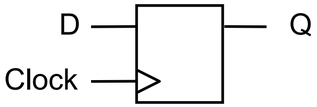
| Enable | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Q | $\overline{Q}$ |
| 0 | 1 | Q | $\overline{Q}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| Enable | Q | $\overline{Q}$ |
|---|---|---|
| 0 | Q | $\overline{Q}$ |
| 1 | D | $\overline{D}$ |

# Clocked Flip-Flops

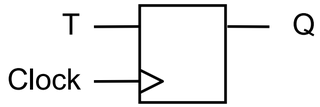
Flip-flops whose output changes only on the rising edge of the clock input

D-type ("delay")



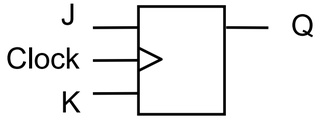

| $D_n$ | Clk | Q |
|---|---|---|
| 0 | ↑ | 0 |
| 1 | ↑ | 1 |

T-type ("toggle")

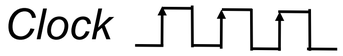



| T | Clk | Q |
|---|---|---|
| 0 | ↑ | Q |
| 1 | ↑ | $\overline{Q}$ |

JK-type




| J | K | Clk | Q |
|---|---|---|---|
| 0 | 0 | ↑ | Q |
| 0 | 1 | ↑ | 0 |
| 1 | 0 | ↑ | 1 |
| 1 | 1 | ↑ | $\overline{Q}$ |




Clocked flip-flops often have additional inputs to allow the SET and RESET of Q

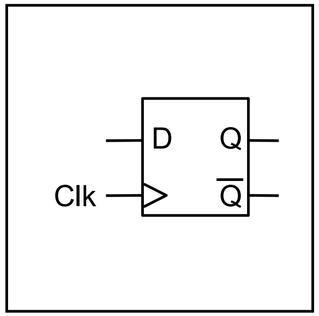
An exceptionally useful building blocks in other sequential logic circuits

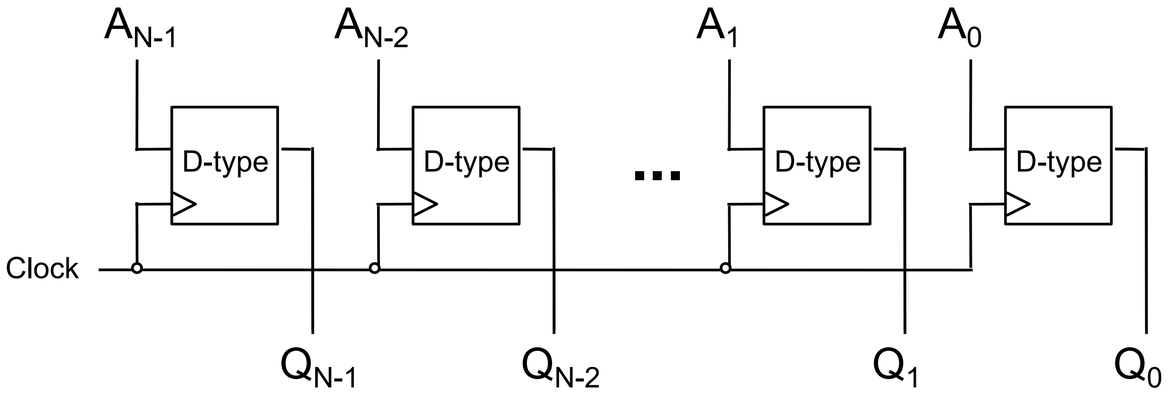
# N-bit Register

A multi-bit memory

N-bit number $A_{N-1}$ $A_{N-2}$...$A_1A_0$ is stored in the register on a low to high transition of the clock

Stored number appears on the outputs $Q_{N-1}$...$Q_0$

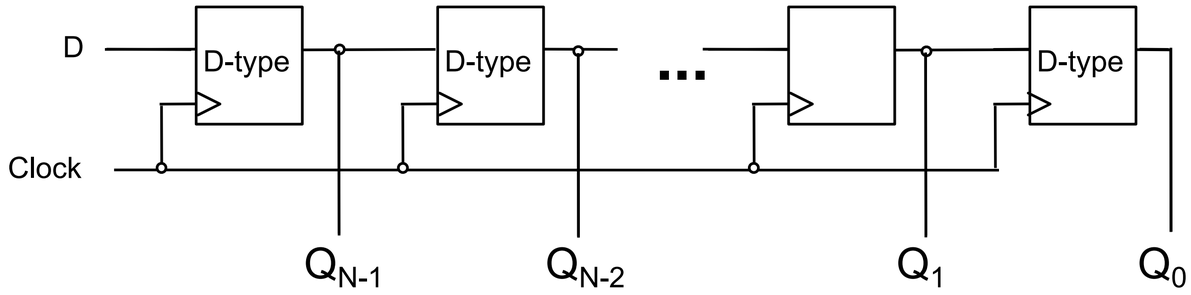


# N-bit Shift Register

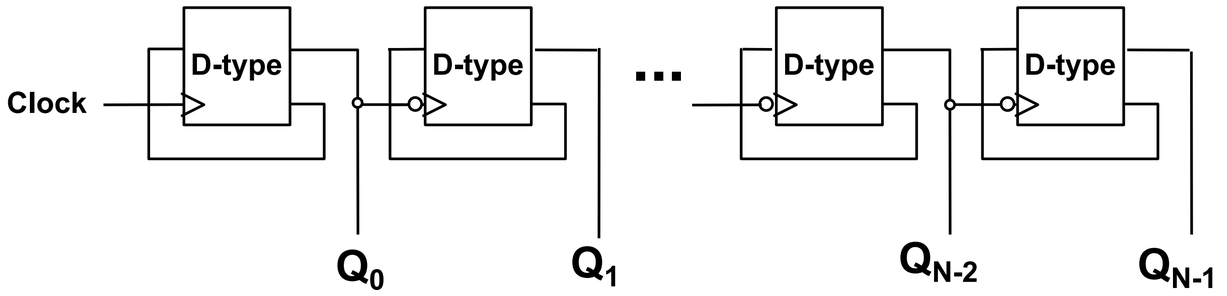
A register that stores and shifts a number one bit at a time

For serial to parallel conversion we make D=serial bit 0, trigger clock, then D=bit 1, clock… etc… after N bits, read out all bits in parallel from Q




When a clock transition occurs, each bit in the register will be shifted one place to the right

# N-bit Counter




**Output:** 0000
1000
0100
1100
0010
etc...

Note the circles (on the inputs to the D-types, except the left-most one) to indicate an inverter.

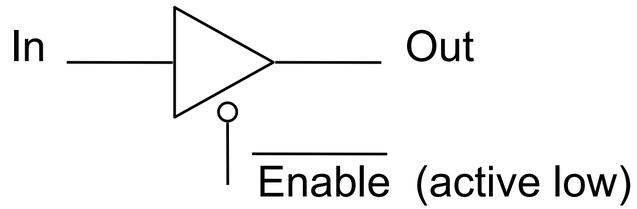
Each flip-flop (except the left-most one) is clocked only on a transition of high ⇒ low (falling clock edge) of the previous output

# Three State Logic

# Three-state Logic

The logic components considered so far have two possible logic states, i.e., 1 or 0

There is a further gate, termed a three-state buffer, whose output can be placed in a third state, known as UNCONNECTED



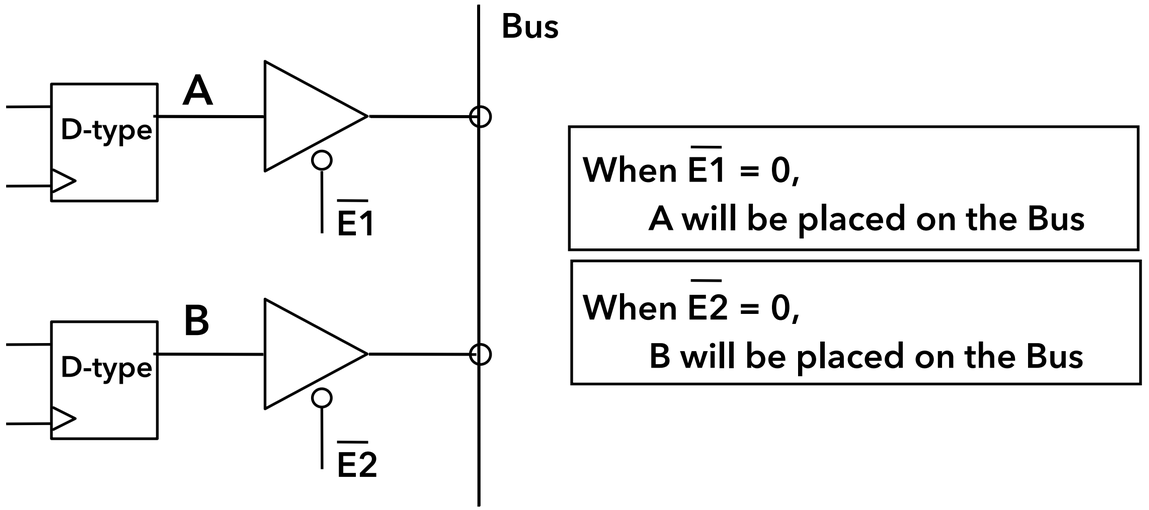

When $\overline{\text{Enable}}$ is high, the input is disconnected from the output

When $\overline{\text{Enable}}$ is low, the input is connected to the output

# Three-state Buses

Can use three state buffers to allow different sources of data onto a common BUS

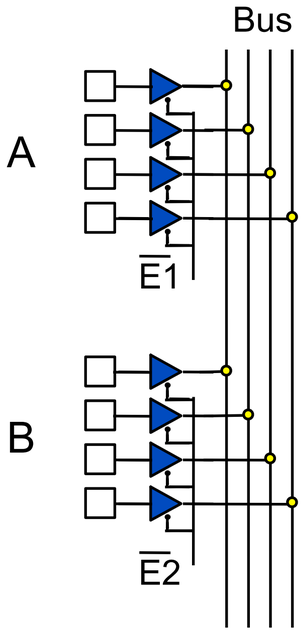
For example, consider a 1-bit wide bus




**N.B. Should not have $\overline{E1}$=0 and $\overline{E2}$ = 0 at the same time!**

# Multi-bit Bus

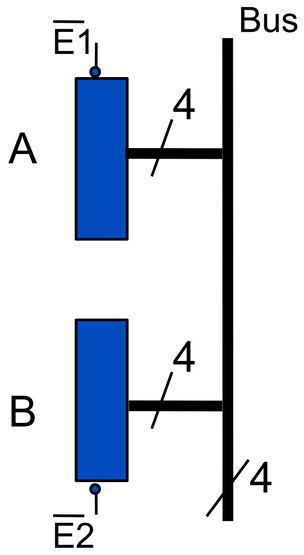
We can consider a 4-bit bus




When $\overline{E1}$ = 0,
A will be placed on the Bus

When $\overline{E2}$ = 0,
B will be placed on the Bus

Or, can represent this more concisely:

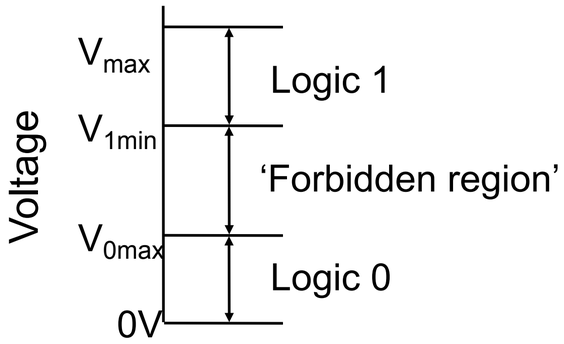


# Physical Implementations

# Properties of Logic Gates

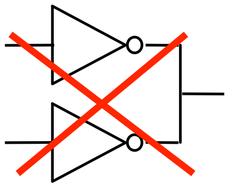
Several basic points to remember when it come to the construction of logic circuits, particularly in lab sessions

Logic values by Voltage level:




Typically TTL(Transistor-Transistor Logic),
$V_{max} = 5$, $V_{1min} = 2.8$, $V_{0max} = 0.8$

Do NOT connect outputs together except three-state buffers



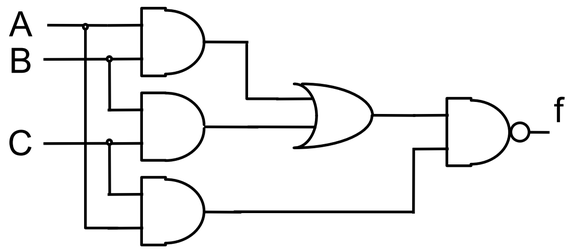
…because you could be connecting 1 to 0, causing a short circuit

# Propagation Delay

Each logic gate has a propagation delay, typically nano-seconds, i.e., $1x10^{-9}$s, or less

Limits the speed at which logic circuits work

Propagation delay can be reduced by putting logic gates close together, e.g., on the same integrated circuit




| Gate | Prop. Delay(ns) |
|---|---|
| NOT | 1.0 |
| NAND | 1.2 |
| AND | 1.7 |
| OR | 1.2 |
| NOR | 1.5 |

(values for illustration only)

What is the maximum propagation delay in this circuit?

Answer = 1.7 (AND) + 1.2 (OR) + 1.2 (NAND) = 4.1ns

# Logic Integrated Circuits (ICs)

Elementary logic gates and functions can be obtained in small ICs, e.g., the 7400 series, though programmable devices allow much larger circuits to be created inside a single chip

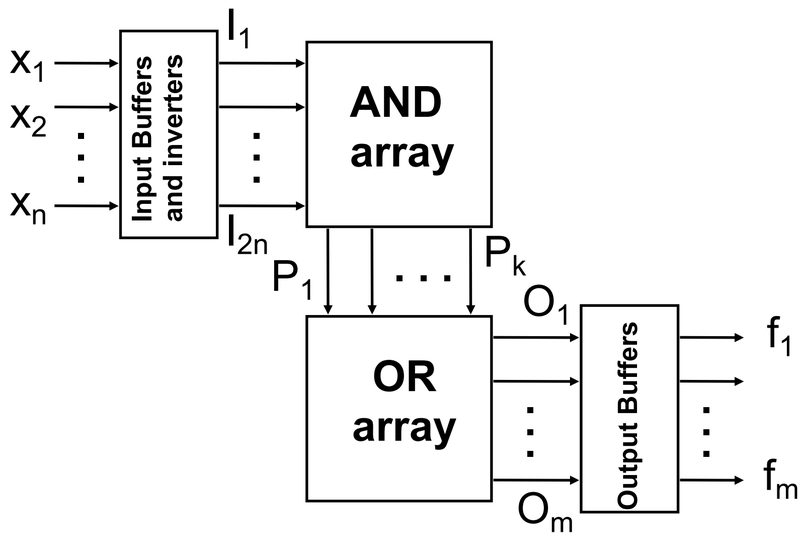
There are many types of programmable logic device

- Programmable Array Logic (PAL) - The first popular programmable device was one-time programmable
- Programmable Logic Array (PLA) - Device contains an AND array, which feeds an OR array, providing a sum of products in hardware
- Field Programmable Gate Array (FPGA) - One of several modern possibilities, which can contain millions of gates, i.e., enough for an entire processor

# PLA Example

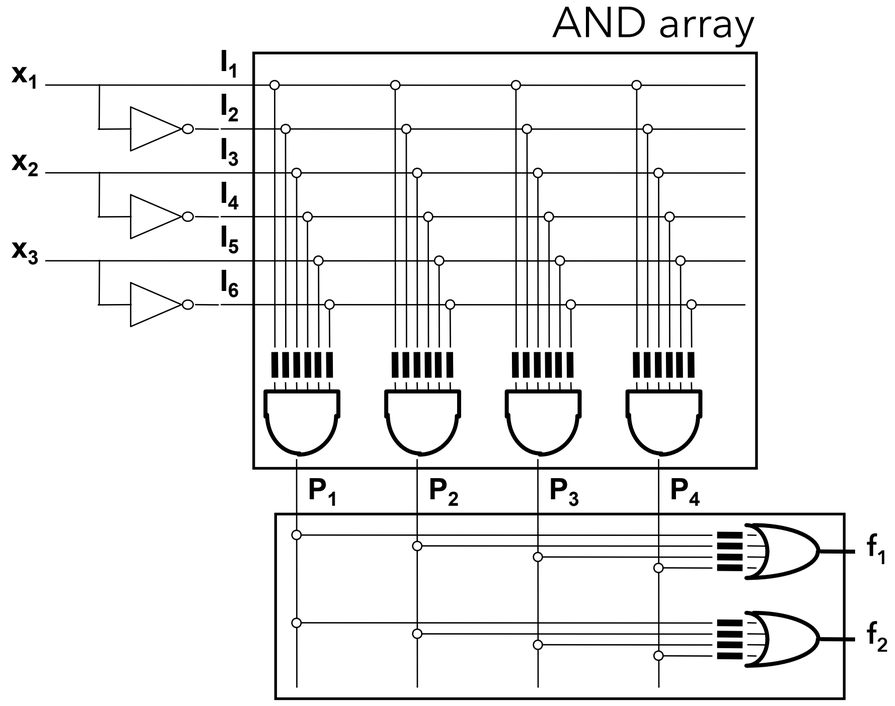
We can perform an AND function on a combination of any inputs, and their inverses, followed by an OR

Essentially any sum-of-products combinatorial function



# PLA Organisation

A generic structure and destructible links provide ease of use

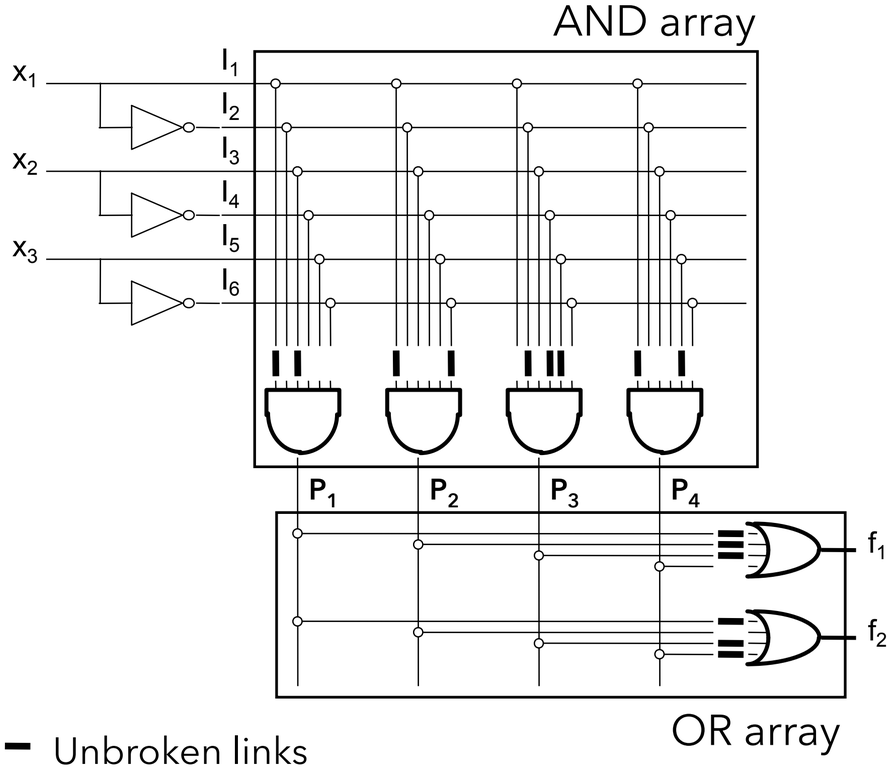



- Links (fuses) that can be broken

# PLA Function Implementation

An example logic function implemented in a PLA is shown below

Observe how the unbroken links correspond to the implemented function




$P_1 = X_1 \cdot X_2$

$P_2 = X_1 \cdot X_3$

$P_3 = X_1 \cdot X_2 \cdot X_3$

$P_4 = X_1 \cdot X_3$

$f_1 = P_1 + P_2 + P_3$

$f_2 = P_1 + P_3 + P_4$