Part IA Engineering

Digital Circuits &

Information Processing

Handout 1

Combinational Logic

Richard Prager
Tim Flack
January 2009

Aims

The aims of the course are to:

- Familiarize students with combinational and sequential digital logic circuits, the analogue-digital interface, and the hardware and basic operation of microprocessors, memory and the associated electronic circuits which are required to build microprocessor based systems.

- Teach the engineering relevance and application of digital and microprocessor-based systems, give students the ability to design simple systems of this kind, and understand microprocessor operation at the assembly-code level.

From Semiconductors to Computers

- build transistors from semiconductors
- build gates from transistors
- build logic functions from gates
- build flip-flop bistables from logic
- build counters and sequencers from flip-flops
- build microprocessors from sequencers
- build computers from microprocessors

Contents of Handout 1

Handout 1 Section A

Introduction to Logic Gates

In this section we introduce Boolean algebra and logic gates.

Logic gates are the building blocks of digital circuits.

Logic Variables

- Logic variables
- Binary variables
- Boolean variables

All names for the same thing.

A variable that can take only two values:

- TRUE or FALSE
- 1 or 0

In electronic circuits the two states of a logic variable are represented by two voltage levels. For example, a high voltage for 1 and a low voltage for 0.

Uses of Simple Logic

Heating Boiler

If chimney not blocked and house is cold and pilot light lit then open main fuel valve to start up boiler.

B = chimney blocked
C = house is cold
P = pilot alight
V = open fuel valve

V = (NOT B) AND C AND P

Uses of Simple Logic

Washing Machine

If drum not turning and no water in drum and program finished then permit door to be opened.

T = drum turning
W = water in drum
P = program active
L = door unlocked

L = (NOT T) AND (NOT W) AND (NOT P)

We can write this using bars above the symbols to denote NOT.

L = $\overline{T}$ AND $\overline{W}$ AND $\overline{P}$

Logic Gates

Electronic circuits that have logic signals as their inputs and outputs are known as LOGIC CIRCUITS or DIGITAL CIRCUITS.

Basic logic circuits with one or more inputs, and one output, are also known as GATES.

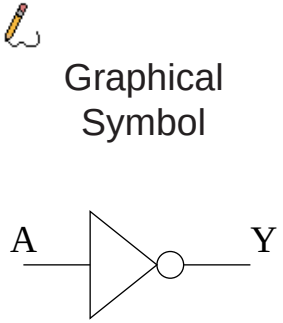
GATES are used as building blocks in the design of more complex logic circuits.

There are several ways of representing logic functions:

- Graphical symbols used to represent the gates.
- Input-output maps.
- Boolean algebra.

NOT Gate

Graphical Symbol | Input-output Map | Boolean representation



A	0	1
	1	0

$Y = \overline{A}$

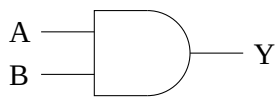
A NOT gate is called an 'inverter'.

Y is TRUE if and only if A is FALSE.

A circle on the output of a gate always implies that it is an inverting output.

AND Gate

Graphical Symbol



Input-output Map

A \ B	0	1
0	0	0
1	0	1

Boolean representation

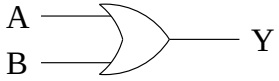
$$Y = A.B$$

Y is TRUE if and only if A is TRUE and B is TRUE.

In Boolean algebra AND is represented by a dot $.$

OR Gate

Graphical Symbol



Input-output Map

A \ B	0	1
0	0	1
1	1	1

Boolean representation

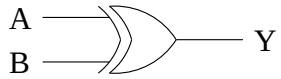
$$Y = A + B$$

Y is TRUE if A is TRUE or B is TRUE (or both).

In Boolean algebra OR is represented by a plus sign $+$

EXCLUSIVE OR Gate

Graphical Symbol



Input-output Map

B \ A	0	1
0	0	1
1	1	0

Boolean representation

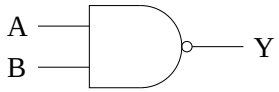
$Y = A \oplus B$

Y is TRUE if A is TRUE or B is TRUE but not both.

In Boolean algebra XOR is represented by an $\oplus$ sign.

NAND Gate

Graphical Symbol



Input-output Map

B \ A	0	1
0	1	1
1	1	0

Boolean representation

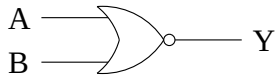
$Y = \overline{A.B}$

Y is TRUE if A is FALSE or B is FALSE (or both).

Y is FALSE if and only if A is TRUE and B is TRUE.

NOR Gate

Graphical Symbol



Input-output Map

B \ A	0	1
0	1	0
1	0	0

Boolean representation

$$Y = \overline{A + B}$$

Y is TRUE if and only if A is FALSE and B is FALSE.

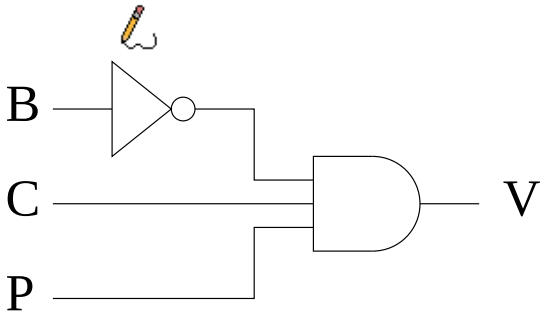
Y is FALSE if A is TRUE or B is TRUE (or both).

Boiler Example

If chimney not blocked and house is cold and pilot light lit then open main fuel valve to start up boiler.

B = chimney blocked
C = house is cold
P = pilot alight
V = open fuel valve

$$V = \overline{B}.C.P$$

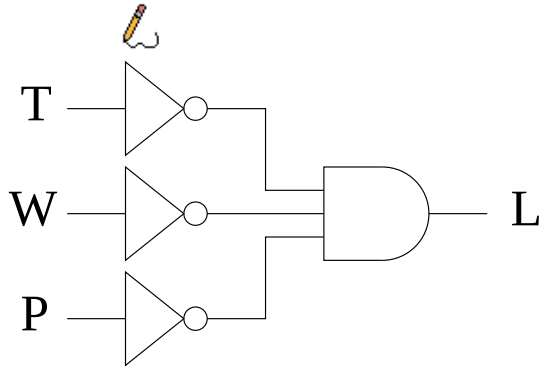


Washing Machine Example

If drum not turning and no water in drum and program finished then permit door to be opened.

$$\begin{aligned} T &= \text{drum turning} \\ W &= \text{water in drum} \\ P &= \text{program active} \\ L &= \text{door unlocked} \end{aligned}$$

$$L = \overline{T}.\overline{W}.\overline{P}$$



Handout 1 Section B

Building Gates from Transistors

Logic circuits are non-linear so we first have to learn a graphical technique for analyzing non-linear circuits.

The construction of an NMOS inverter from an N-channel field effect transistor is described.

We then outline the structure of NMOS AND gates and OR gates, and estimate the speed of NMOS circuits.

A discussion of the power consumption of NMOS gates leads to the introduction of the CMOS inverter circuit.

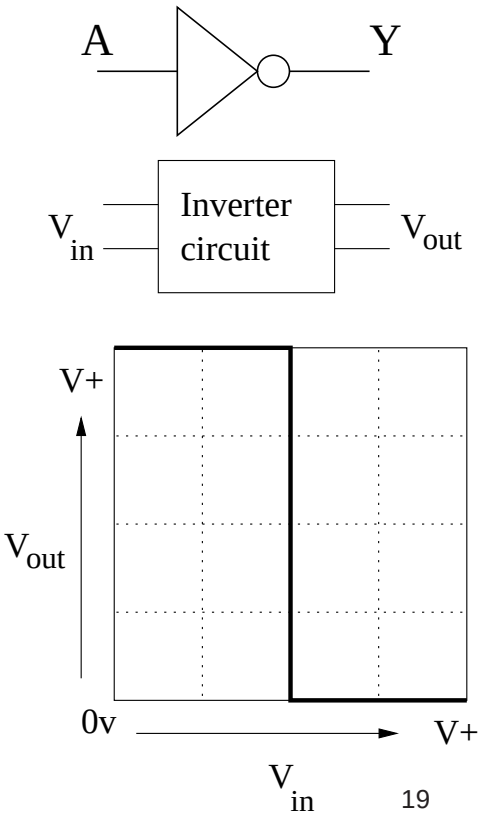
Implementing Logic Gates

A high voltage represents logic 1.

A zero voltage represents logic 0.

The graph shows an ideal characteristic for an inverter circuit.

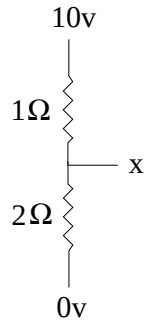
Note that it is non-linear.

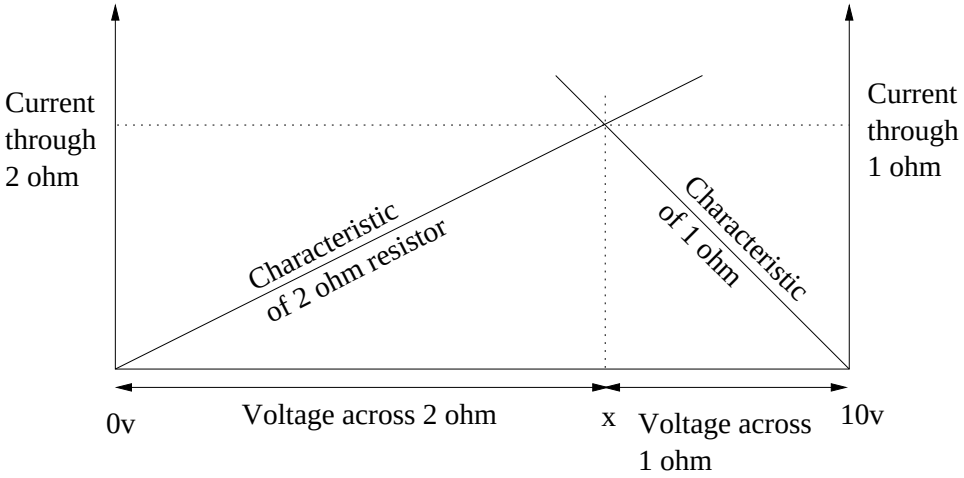


Solving Non-linear Circuits

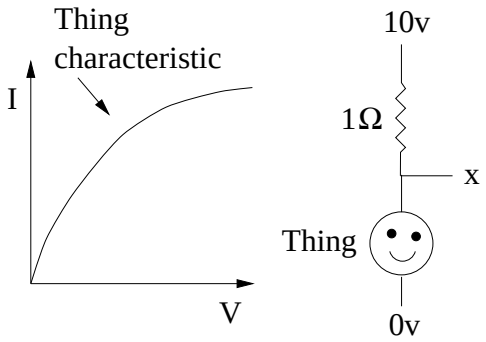
Solve a simple circuit graphically.

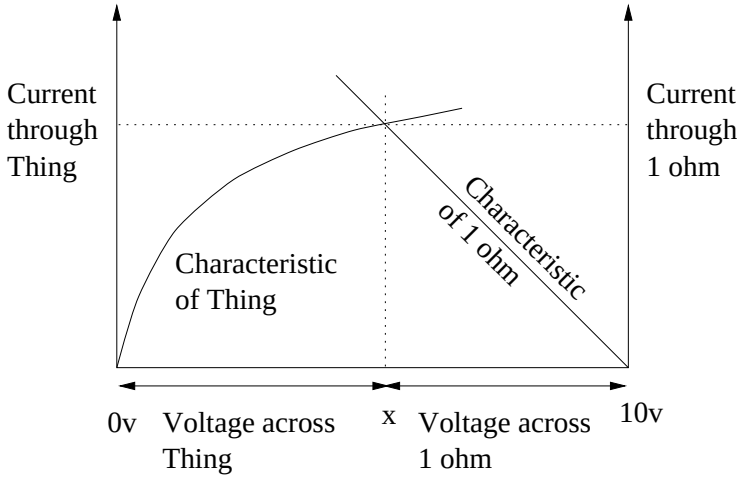
The same technique will still work when we introduce non-linear components.



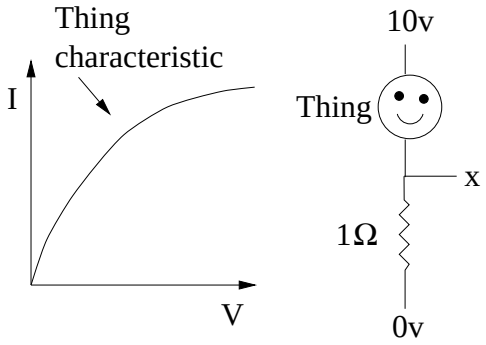


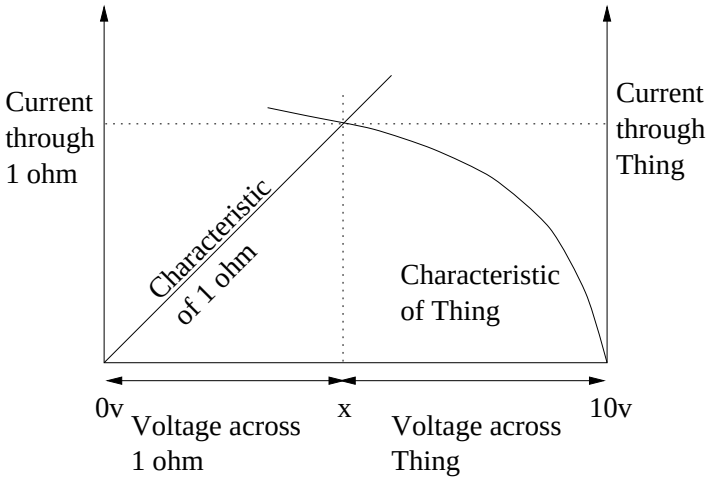
Suppose we now replace the lower resistor with a non-linear component (that we will call a ‘Thing’).





The characteristic of the upper component is drawn backwards along the V axis, starting at the supply voltage.





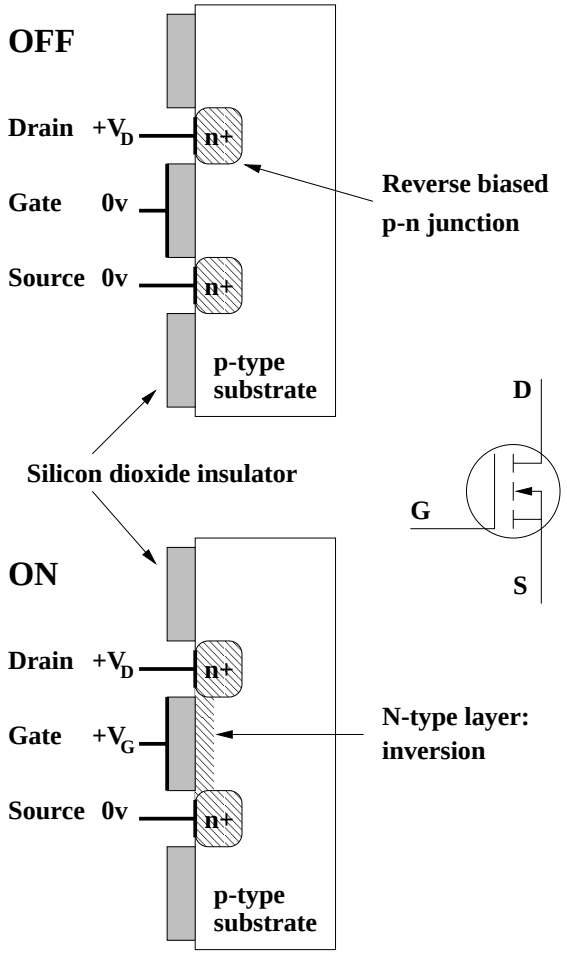
Building Gates from Transistors

We start by building logic gates out of N-channel MOSFETs.

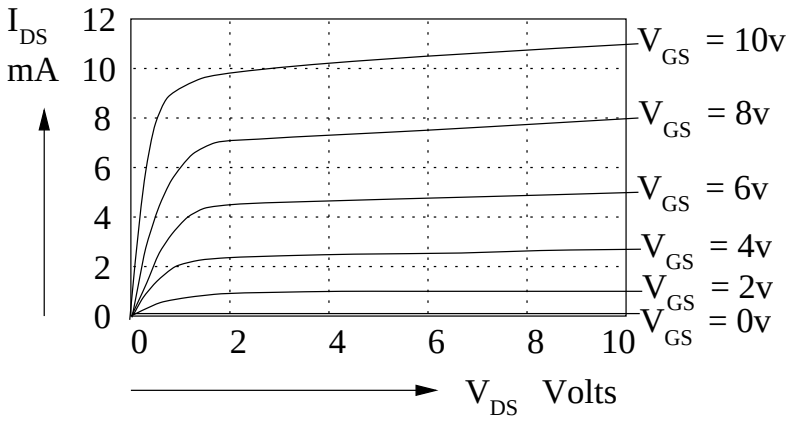
Metal Oxide Semiconductor Field Effect Transistor

Enhancement mode: means off when $V_{GS} = 0$. The alternative is a depletion mode transistor which is on (i.e. conducts from drain to source) when $V_{GS} = 0$.

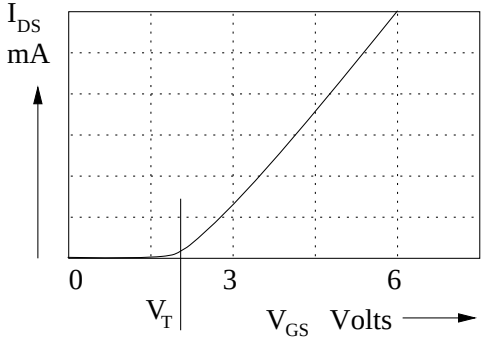
N-channel MOSFET



NMOSFET Characteristics



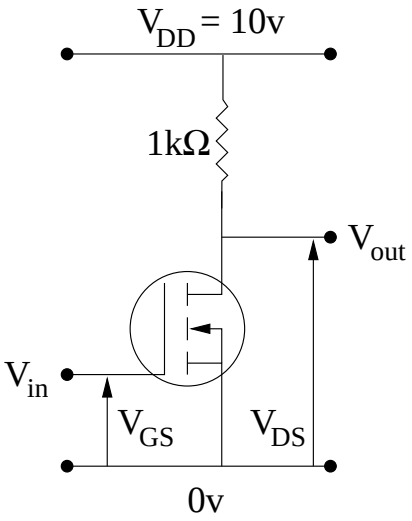
I_{DS} as a function of V_{GS} at constant V_{DS}

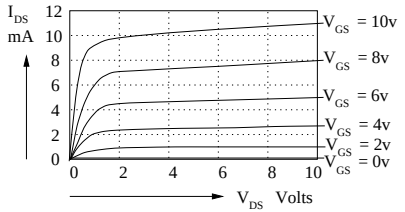


The transistor conducts when V_{GS} reaches the Threshold voltage: V_T

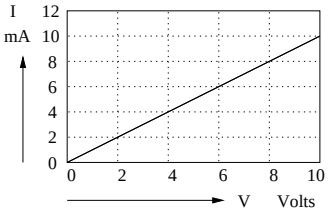
NMOS Inverter

Here is the circuit to implement an inverter using an NMOS FET and a 1kΩ resistor.

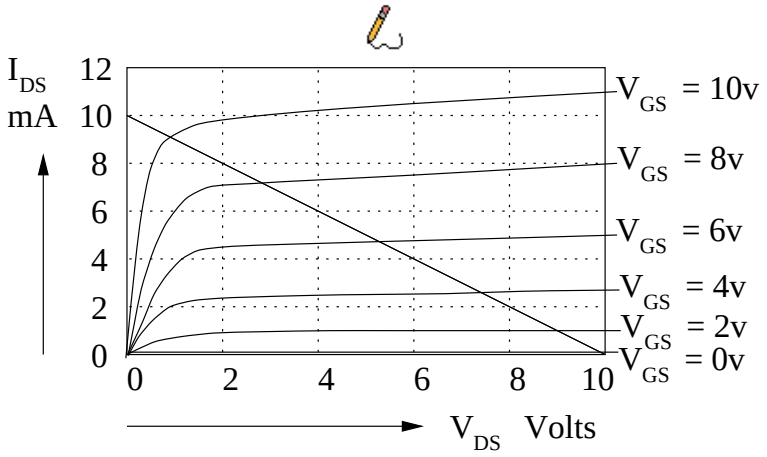


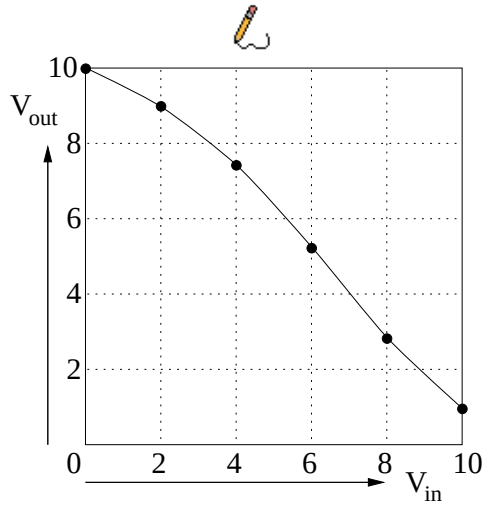


NMOS FET Characteristic



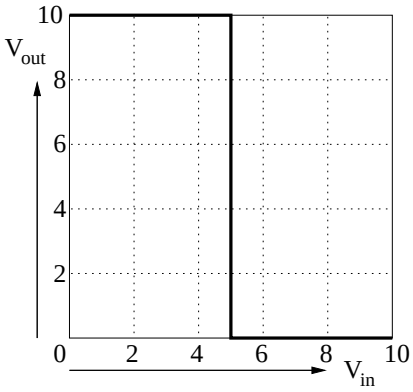
Resistor Characteristic





Voltage Levels

The input-output characteristic of the gate is far from the ideal we originally wanted.

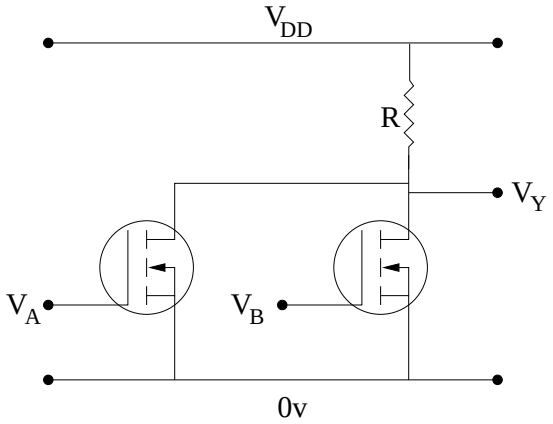


However if we say:

voltage $>$ 9v is logic 1
voltage $<$ 2v is logic 0

the gate will work:

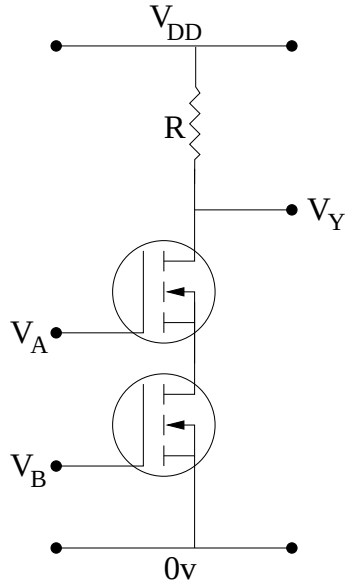
$$V_{in} > 9\text{v} \Rightarrow V_{out} < 2\text{v}$$
$$V_{in} < 2\text{v} \Rightarrow V_{out} > 9\text{v}$$



What sort of gate is this?
Think of the transistors as switches. When V_{GS} is high they conduct. Otherwise they don't conduct.

A	B	Y
low	low	high
high	low	low
low	high	low
high	high	low

This corresponds to a NOR gate: $Y = \overline{A + B}$



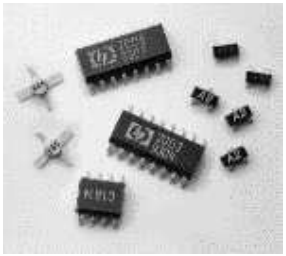
A	B	Y
low	low	high
high	low	high
low	high	high
high	high	low

This corresponds to a NAND gate: $Y = \overline{A.B}$

Benefit of Using Gates

When we have a complicated logic function to implement, (eg. the boiler example previously discussed $V = \overline{B}.C.P$), you don't have to design a special transistor circuit to provide the functionality.

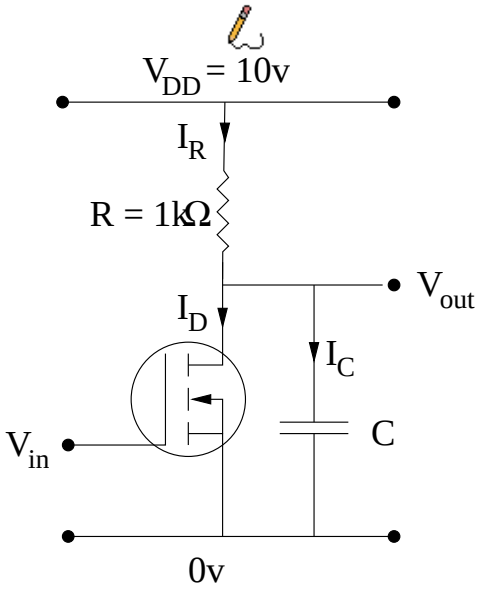
Instead you just buy an integrated circuit (or chip) that provides the appropriate gates.



For the boiler example we would need a three-input AND gate and an inverter.

Speed of NMOS Logic

One of the main speed limitations in real NMOS logic is due to stray capacitance. The output of each gate is connected by a length of metal track to the input of the next. This has capacitance to ground. We therefore modify the circuit model.



Assume that initially V_{in} is high, hence $V_{out} = 1$ V.

At time $t = 0$, V_{in} falls to 0 V, so $I_D = 0$.

$$\begin{aligned} I_R &= I_C \\ \frac{V_{DD} - V_{out}}{R} &= C\frac{dV_{out}}{dt} \end{aligned}$$

So

$$\frac{dV_{out}}{dt} + \frac{V_{out}}{RC} = \frac{V_{DD}}{RC}$$

$$\Rightarrow V_{out} = V_{DD} + (1 - V_{DD}) \exp\left(\frac{-t}{RC}\right)$$

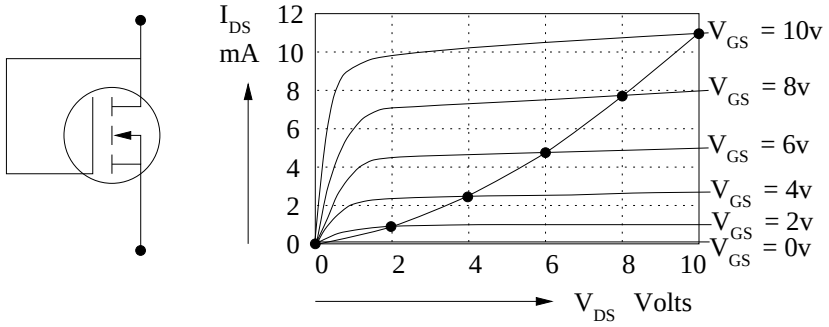
Which means that 95% of the voltage change will take a time of 3RC.

Using a FET to replace the Resistor

A big advantage of chips is that they are small.

Implementing a resistor on a chip takes up a lot of space on the silicon.

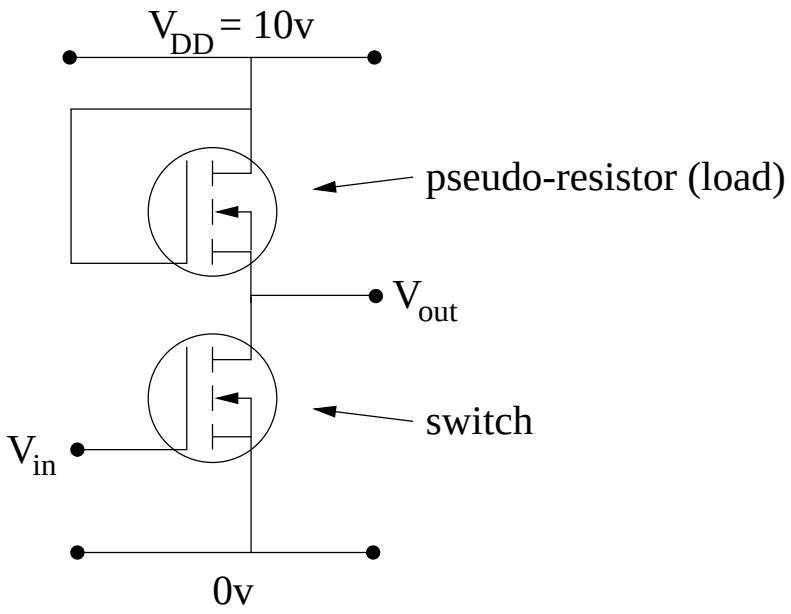
Fortunately it is possible to make a MOSFET behave (almost) like a resistor by connecting the Gate to the Drain. Thus $V_{GS} = V_{DS}$.



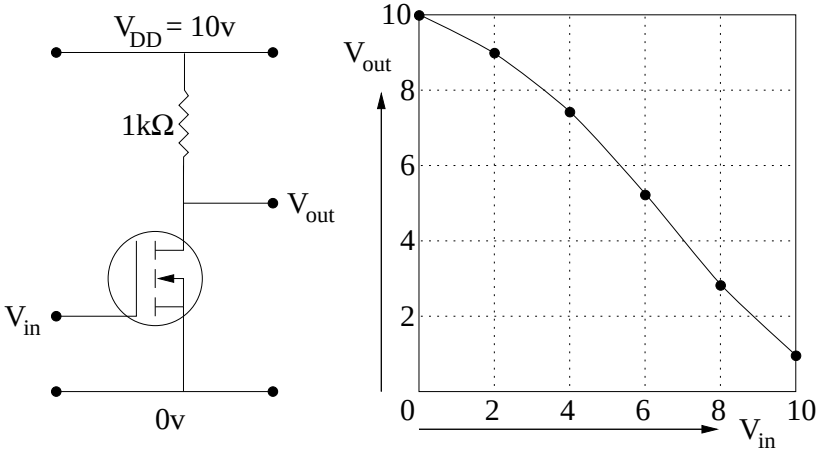
It's not quite a straight line, so behaviour will not be the exactly same as a resistor. However, it's roughly like 900Ω.

Smaller NMOS Inverter

Using a FET to approximate a resistor we can produce a revised design for an inverter circuit that is easier to miniaturize.



Power Consumption



When $V_{out} = 10\text{v}$ there is no current ($I_D = 0$. There is therefore no power dissipated.

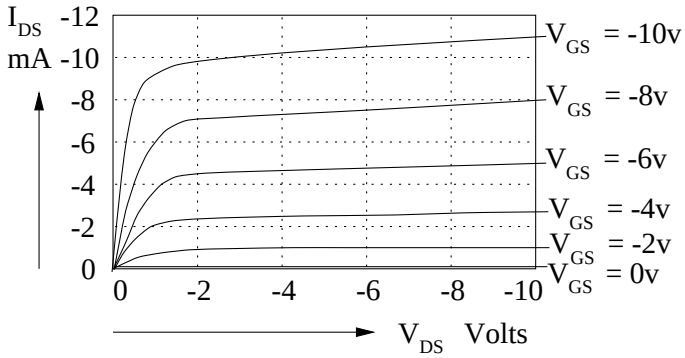
However, when $V_{out} \neq 10\text{v}$ current will flow down through the resistor and heat will be generated.

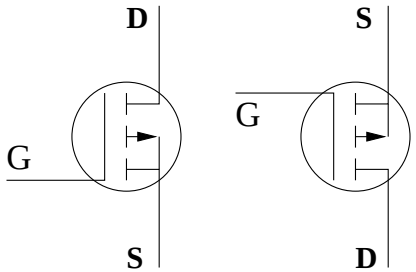
For example, when $V_{in} = 10\text{v} \Rightarrow V_{out} = 1\text{v}$ and $I_D = 9\text{mA}$ so 81mW will be dissipated in the resistor and 9mW will be dissipated in the FET.

Complementary MOS

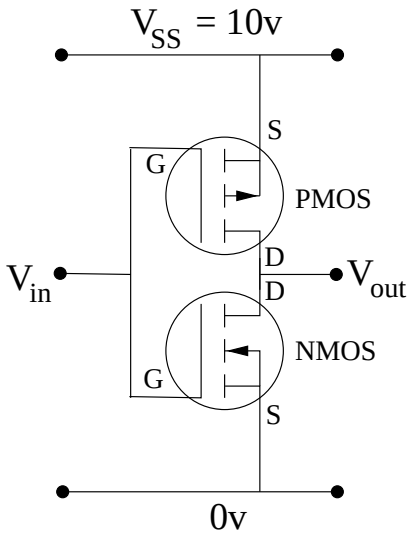
The problem with NMOS logic is power dissipation in the resistors. To solve this, CMOS logic was invented. This uses both NMOSFETS and PMOSFETS.

PMOSFETS are essentially NMOSFETS with all the polarities reversed:

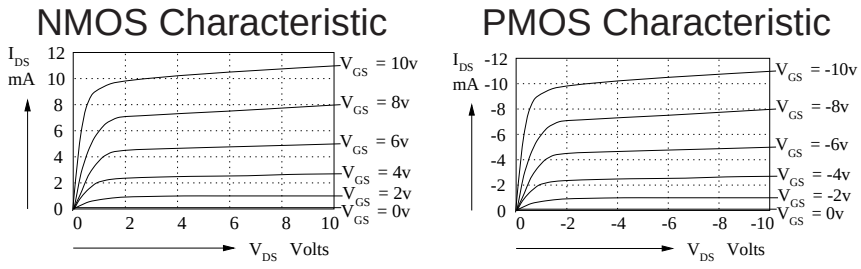


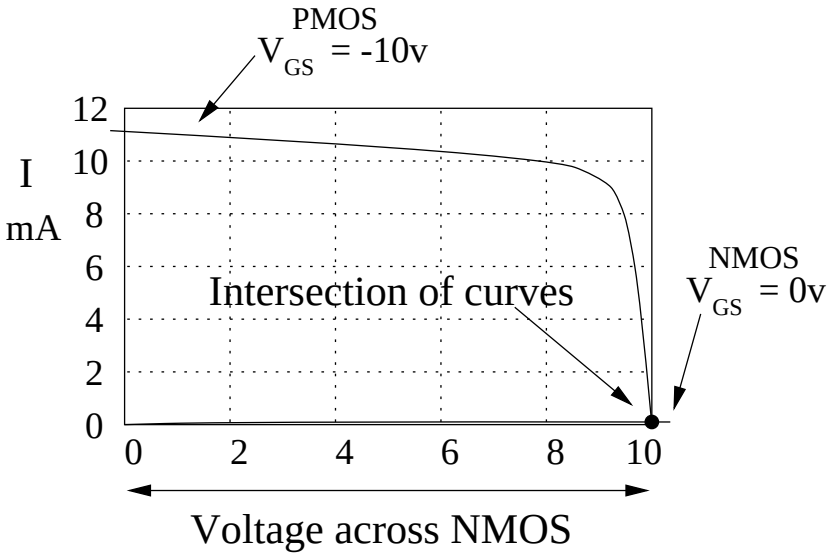


CMOS Inverter



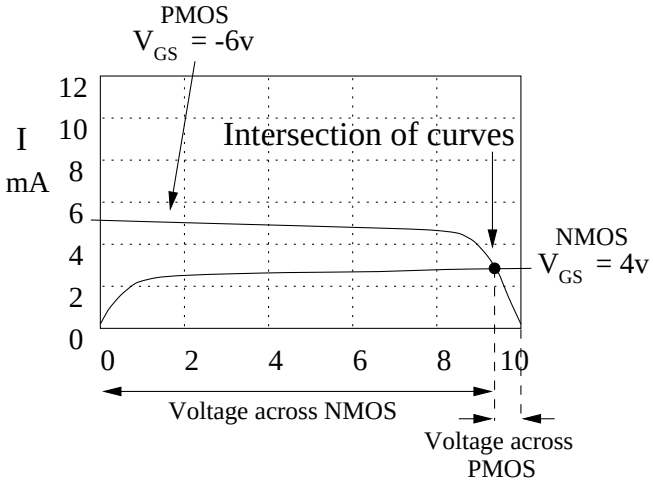
V_{in}	NMOS	PMOS	V_{out}
low	off	on	high
high	on	off	low



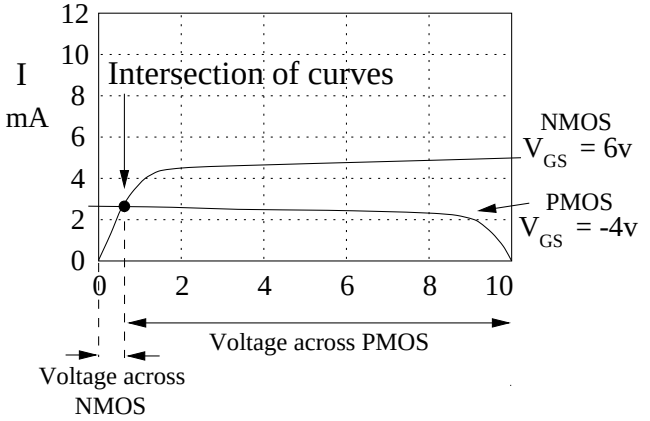


(a)

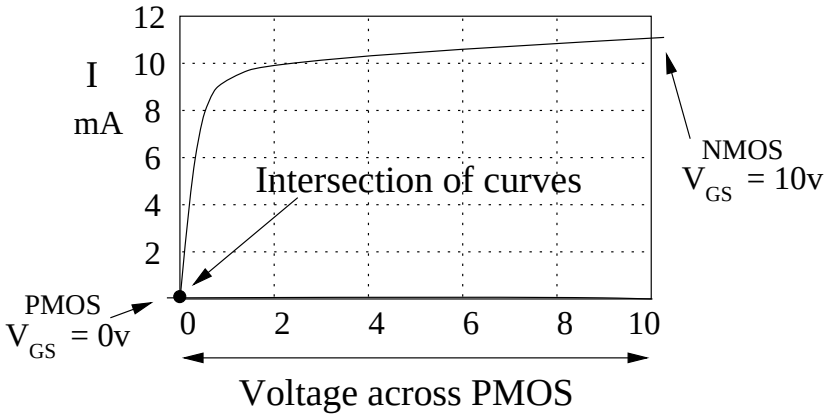
$V_{in} = 0\text{v} \Rightarrow V_{out} = 10\text{v}$. No current flows.



(b) $V_{in} = 4\text{v} \Rightarrow V_{out} = 9.5\text{v}$, $I = 2.7\text{mA}$.



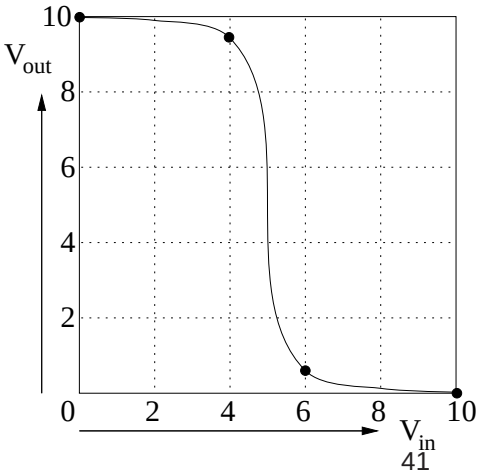
(c) $V_{in} = 6\text{v} \Rightarrow V_{out} = 0.5\text{v}$, $I = 2.7\text{mA}$.



(d)

$V_{in} = 10\text{v} \Rightarrow V_{out} = 0\text{v}$. No current flows.

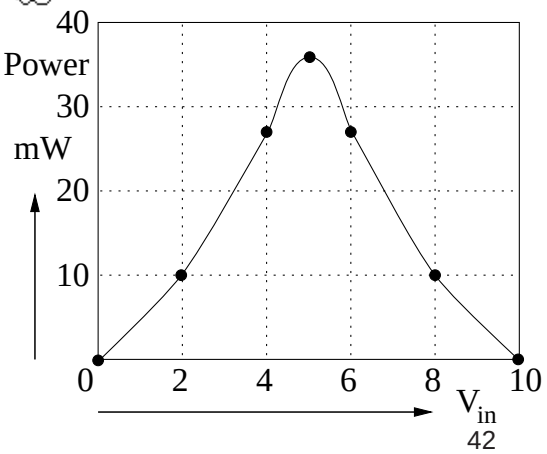
Using these values we can construct the input-output characteristic of the inverter circuit. Note that the CMOS inverter is much closer to our ideal than the NMOS inverter was.



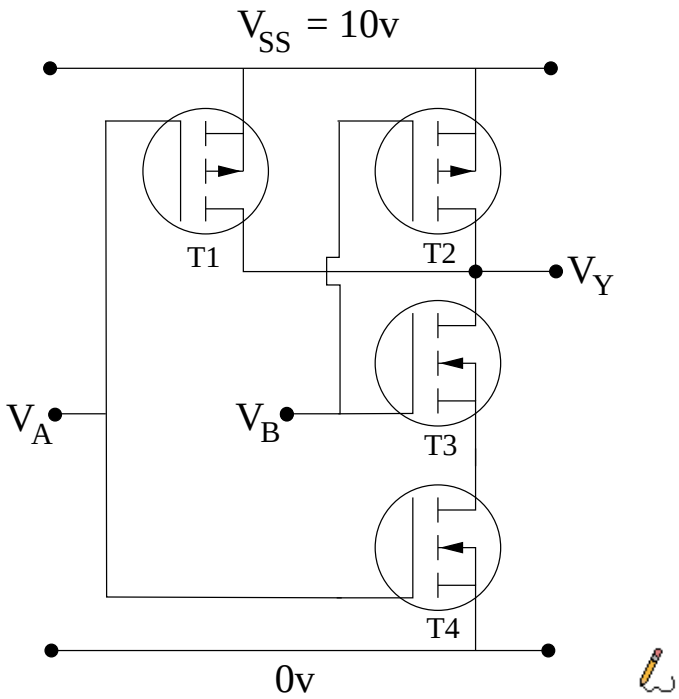
Power in CMOS Inverter

V_{in}	I /mA	Power /mW
0	0.0	0
2	1.0	10
4	2.7	27
5	3.6	36
6	2.7	27
8	1.0	10
10	0.0	0

Note that current only flows when the gate is changing state. This means that power is only dissipated as the gate switches on or off.



CMOS NAND Gate



V_A	V_B	T1	T2	T3	T4	V_Y
low	low	on	on	off	off	high
low	high	on	off	on	off	high
high	low	off	on	off	on	high
high	high	off	off	on	on	low

Logic Families

NMOS Compact, slowish, cheap.

CMOS Propagation delay 8–50 nS, max clock frequency 12–40 MHz for gates in individual packages. Power consumption $< 10^{-6}$ W/gate when not changing, and about 10^{-4} W/gate when changing at 100 kHz.

TTL Constructed from bipolar transistors. Propagation delay 1.5–10 nS, max clock frequency 35–200 MHz. Power consumption is about 10^{-2} W/gate.

ECL Constructed from bipolar transistors. High speed. High power consumption; current flows all the time.

Handout 1 Section C

Boolean Algebra for Logic Design

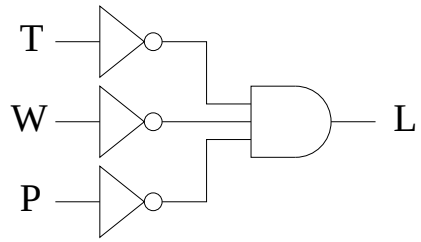
In this section we introduce the laws of Boolean algebra and show how it can be used to design combinational logic circuits. We then introduce the hardware description language VHDL. This language can be used to enable computer-aided design of logic circuits.

Combinational logic circuits do not have an internal stored state; the output is a function of the current inputs. (Later in the course we will study circuits with a stored internal state. These are called sequential logic circuits.)

Combinational Logic Design

In our washing machine example we needed to implement the logic function

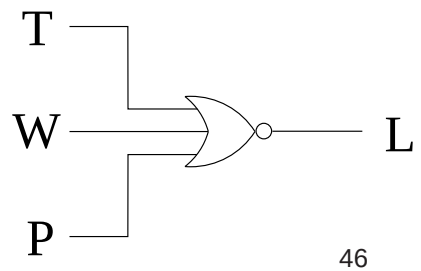
$$L = \overline{T}.\overline{W}.\overline{P}$$



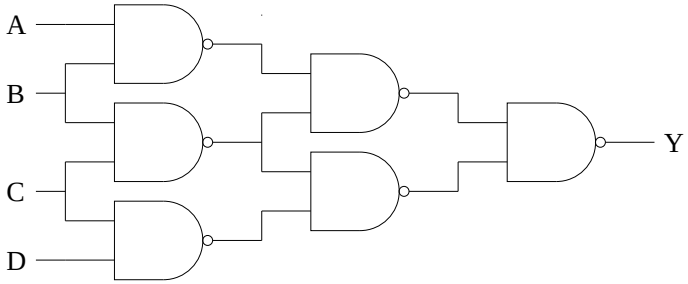
This could have been achieved more simply using the fact that

$$L = \overline{T}.\overline{W}.\overline{P} = \overline{(T + W + P)}$$

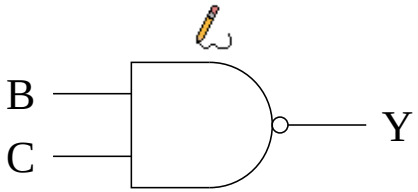
so a single NOR gate will do the whole job.



In an extreme example, it would be very useful to be able to work out that



can be replaced by



This sort of problem can be solved using Boolean algebra and Karnaugh maps.

We need:

1. Techniques for simplifying logic expressions. Simpler expressions mean fewer gates which lead to lower cost.

2. Techniques for manipulating expressions so that the required function can be computed using gates of only certain types. We thus only need to stock a limited range of gates which can be readily available and cheap.

We are going to study two ways of solving these problems: Boolean algebra and Karnaugh maps.

Boolean algebra rigorous, computable.

Karnaugh maps visual, exhaustive, copes with up to 5 variables.

Boolean Algebra

Commutation

$A + B = B + A$	normal
$A.B = B.A$	normal

Association

$(A + B) + C = A + (B + C)$	normal
$(A.B).C = A.(B.C)$	normal

Distribution

$A.(B + C + \ldots) = (A.B) + (A.C) + \ldots$	normal
$A + (B.C.\ldots) = (A + B).(A + C).\ldots$	NEW

Absorption

$A + (A.C) = A$	NEW
$A.(A + C) = A$	NEW

ORs	ANDs
$A + 0 = A$	$A.0 = 0$
$A + A = A$	$A.A = A$
$A + 1 = 1$	$A.1 = A$
$A + \overline{A} = 1$	$A.\overline{A} = 0$

AND takes precedence over OR.

For example $A.B + C.D = (A.B) + (C.D)$.

Every Boolean law has a dual: any valid statement is also valid with 

.	replaced by	+
+	replaced by	.
0	replaced by	1
1	replaced by	0

Show that $A.(\overline{A} + B) = A.B$

$$\begin{aligned} A.(\overline{A} + B) &= A.\overline{A} + A.B \\ &= 0 + A.B \\ &= A.B \end{aligned}$$

Show that $A + \overline{A}.B = A + B$

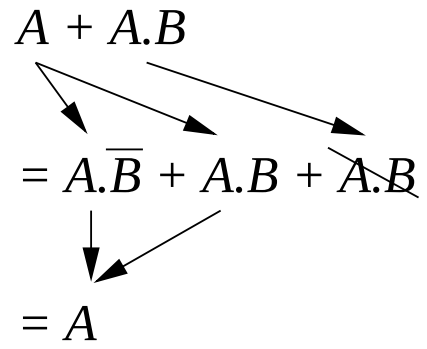
$$\begin{aligned} A + (\overline{A}.B) &= (A + \overline{A}).(A + B) \\ &= 1.(A + B) \\ &= A + B \end{aligned}$$

Every Variable in Every Term

A useful technique is to expand each term until it includes one instance of each variable (or its compliment). It may be possible to simplify the expression by canceling terms in this expanded form.

If there are two variable (A and B) then terms like $A.B$ or $A.\overline{B}$ are fine as they stand because they already include all the variables. A term A would need to be expanded to $A.B + A.\overline{B}$.

Here, as an illustration, the technique is used to prove the absorption rule:



Simplify $X.Y + \overline{Y}.Z + X.Z + X.Y.Z$

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

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

$= X.Y.Z + X.Y.\overline{Z} + X.\overline{Y}.Z + \overline{X}.\overline{Y}.Z + \cancel{X.Y.Z} + \cancel{X.\overline{Y}.Z}$

$= X.Y + \overline{Y}.Z$

De Morgan's Theorem

$$\begin{aligned} \overline{A + B + C + \ \ldots} &= \overline{A}.\overline{B}.\overline{C}.\ \ldots \\ \overline{A.B.C.\ \ldots} &= \overline{A} + \overline{B} + \overline{C} + \ \ldots \end{aligned}$$

$$\begin{aligned} A + B + C + \ \ldots &= \overline{\overline{A}.\overline{B}.\overline{C}.\ \ldots} \\ A.B.C.\ \ldots &= \overline{\overline{A} + \overline{B} + \overline{C} + \ \ldots} \end{aligned}$$

In a simple expression like $A + B + C$ (or $A.B.C$) you can change all the operators from OR to AND (or vice versa) provided you put a bar over each term individually and a further bar over the whole expression.

Proof of De Morgan's Theorem

For two variables we can prove $\overline{A+B} = \overline{A}.\overline{B}$ and $\overline{A.B} = \overline{A} + \overline{B}$ using a truth table.

A	B	$\overline{A+B}$	$\overline{A.B}$	$\overline{A}$	$\overline{B}$	$\overline{A}.\overline{B}$	$\overline{A}+\overline{B}$
0	0	1	1	1	1	1	1
0	1	0	1	1	0	0	1
1	0	0	1	0	1	0	1
1	1	0	0	0	0	0	0

Extending to more variables is by induction:

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

etc.

Washing Machine Example

In our washing machine example we needed to implement the logic function $L = \overline{T}.\overline{W}.\overline{P}$. This can be simplified with a single application of De Morgan's theorem:

$$L = \overline{T}.\overline{W}.\overline{P}$$

$$= \overline{T+W+P}$$

Show that $A.B+A.(B+C)+B.(B+C) = B+A.C$

$A.B + A.(B + C) + B.(B + C)$

$= A.B + A.B + A.C + B.B + B.C$ (distribute)

$= A.B + A.B + A.C + B + B.C$ $(B.B = B)$

$= A.B + A.C + B + B.C$ (repeated $A.B$)

$= A.B + A.C + B$ $(B + B.C = B)$

$= A.C + B$ $(A.B + B = B)$

Simplify $A.\overline{B} + A.(\overline{B + C}) + B.(\overline{B + C})$

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

$= A.\overline{B} + A.\overline{B}.\overline{C} + B.\overline{B}.\overline{C}$ (De Morgan)

$= A.\overline{B} + A.\overline{B}.\overline{C}$ $(B.\overline{B} = 0)$

$= A.\overline{B}$ (absorption))

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

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

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

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

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

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

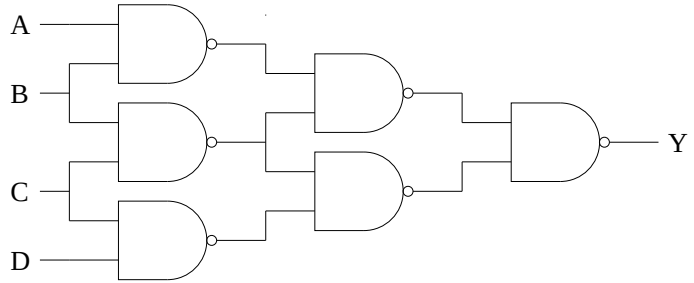
$$= A.B.C.D + \overline{A}.C.D + \overline{B}.C.D$$
(cancel $A.B.\overline{D}.C.D$)

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

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

$$= C.D$$
$(A.B + \overline{A.B} = 1)$

Simplify:



$$Y = \overline{\overline{\overline{A.B}.\overline{B.C}}.\overline{\overline{B.C}.\overline{C.D}}}$$

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

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

$$= \overline{A.B.C + A.B.C.D + B.C + B.C.D}$$
(remove repeated variables)

$$= \overline{B.C}$$
(absorption)

Algebraic Logic Design

A power plant is cooled by 3 ventilation fans, numbered 1 to 3, with flow rates F, $2F$ and $3F$ respectively. An alarm is to be sounded if the plant is running and the air flow rate is less than $3.5F$. Design a logic circuit to do this.

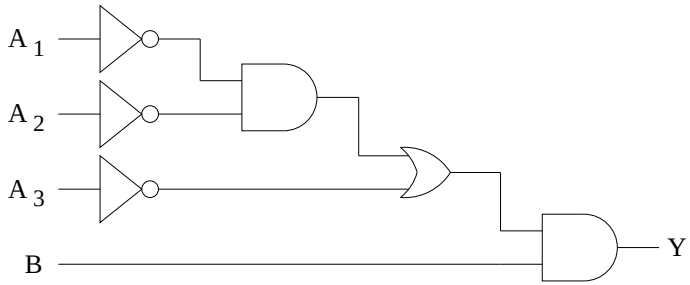
Stage 1: assign logic variables.

- A_1 implies that fan 1 is running.
- A_2 implies that fan 2 is running.
- A_3 implies that fan 3 is running.
- B implies that the plant is running.
- Y sounds the alarm.

Stage 2: convert the problem to algebraic form.
If $A_3 = 0$ or if both A_1 and A_2 equal 0 then, provided the plant is running, we must sound the alarm.

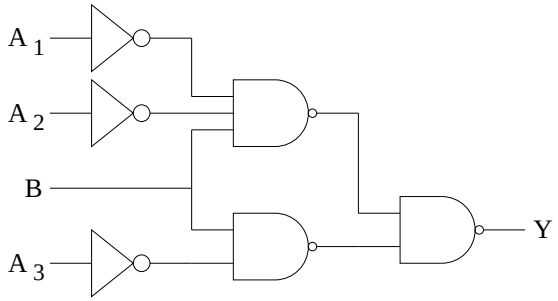
$$\begin{aligned} Y &= (\overline{A_3} + \overline{A_1}.\overline{A_2} + \overline{A_1}.\overline{A_2}.\overline{A_3}).B \\ &= (\overline{A_3} + \overline{A_1}.\overline{A_2}).B \end{aligned}$$

Solution using any gates.



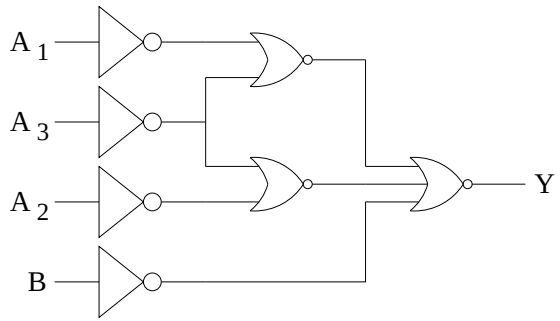
Solution using only NAND gates and inverters.

$$\begin{aligned} Y &= (\overline{A_3} + \overline{A_1}.\overline{A_2}).B \\ &= \overline{A_3}.B + \overline{A_1}.\overline{A_2}.B \\ &= \overline{(\overline{\overline{A_3}.B}).(\overline{\overline{A_1}.\overline{A_2}.B})} \end{aligned}$$



To produce a solution using only NOR gates it is often best to go back to the original problem and write down an expression for when the alarm should be off i.e. $\overline{Y}$

$$\begin{aligned}
\overline{Y} &= A_1.A_3 + A_2.A_3 + \overline{B} \\
&= (\overline{\overline{A_1} + \overline{A_3}}) + (\overline{\overline{A_2} + \overline{A_3}}) + \overline{B} \\
\Rightarrow Y &= \overline{(\overline{\overline{A_1} + \overline{A_3}}) + (\overline{\overline{A_2} + \overline{A_3}}) + \overline{B}}
\end{aligned}$$



Standard Boolean forms

Sum of Products (SOP): usually best to write down an expression for Y directly.

$$Y = \overline{A_3}.B + \overline{A_1}.\overline{A_2}.B$$

Product of Sums (POS): usually best to write down an expression for $\overline{Y}$ and use De Morgan's theorem.

$$\begin{aligned}
\overline{Y} &= A_1.A_3 + A_2.A_3 + \overline{B} \\
&= (\overline{\overline{A_1} + \overline{A_3}}) + (\overline{\overline{A_2} + \overline{A_3}}) + \overline{B} \\
\Rightarrow Y &= (\overline{A_1} + \overline{A_3}).(\overline{A_2} + \overline{A_3}).B
\end{aligned}$$

It is not easy to convert between POS and SOP by algebraic manipulation. Best to consider all the binary permutations not included in one expression and then write down the other to include them. A Karnaugh map makes this easier.

VHDL

Computer-aided design tools are required for the development of complex digital systems. These tools enable the simulation, modelling and testing of designs before they are built.

A hardware description language is required to describe the systems. It must be clear and readable for the designer, yet sufficiently precise to enable rigorous testing of the design.

Very high speed integrated circuit hardware description language,

abbreviated VHDL, has been developed over many years and is the IEEE standard 1076-1993.

Structure of VHDL

VHDL describes circuits in terms of design entities. Each entity consists of two parts. The interface and the architecture specification.

```
entity compont_name is
   list of input and output ports
end compont_name;

architecture arch_name of compont_name is
   declarations of internal signals;
begin
   description of what the the entity does
   and how it is implemented
end arch_name;
```

Definition of Logic Gates

```
entity INV is
  port(A :  in BIT;
       Y :  out BIT);
end INV;

architecture IMOD of INV is
begin
  Y <= not A;
end IMOD;
```

```
entity NAND2 is
  port(A, B :  in BIT;
       Y    :  out BIT);
end NAND2;

architecture NA2M of NAND2 is
begin
  Y <= not (A and B);
end NA2M;
```


Definition of More Logic Gates

```
entity NOR2 is
  port(A, B :  in BIT;
       Y    :  out BIT);
end NOR2;

architecture NO2M of NOR2 is
begin
  Y <= not (A or B);
end NO2M;
```

```
entity NOR3 is
  port(A, B, C :  in BIT;
       Y       :  out BIT);
end NOR3;

architecture NO3M of NOR3 is
begin
  Y <= not (A or B or C);
end NO3M;
```

Using a Gate in a Larger Design

To use a gate directly in a larger design you can use the following syntax.

unique label to refer to this version of the gate
:
entity
name of gate entity
(
name of gate architecture
)
port map
(
'port list' of signals connected to gate
);

---

eg.

```
V1:entity INV(IMOD) port map(A, ABAR);
```


Plant Alarm Using NAND Gates

```
entity PLANT_ALARM is
  port(A1, A2, A3, B :  in BIT;
       Y             :  out BIT);
end PLANT_ALARM;

architecture NAND_GATES of PLANT_ALARM is
  signal A1X,A2X,A3X,R12,R3 :  BIT;
begin
  I1:entity INV(IMOD) port map(A1,A1X);
  I2:entity INV(IMOD) port map(A2,A2X);
  I3:entity INV(IMOD) port map(A3,A3X);
  N1:entity NAND3(NA3M)
             port map(A1X,A2X,B,R12);
  N2:entity NAND2(NA2M)
             port map(A3X,B,R3);
  N3:entity NAND2(NA2M)
             port map(R12,R3,Y);
end NAND_GATES;
```

Plant Alarm Using NOR Gates

```
entity PLANT_ALARM is
  port(A1, A2, A3, B :  in BIT;
       Y             :  out BIT);
end PLANT_ALARM;

architecture NOR_GATES of PLANT_ALARM is
  signal A1X,A2X,A3X,BX,R1,R2 :  BIT;
begin
  I1:entity INV(IMOD) port map(A1,A1X);
  I2:entity INV(IMOD) port map(A2,A2X);
  I3:entity INV(IMOD) port map(A3,A3X);
  I4:entity INV(IMOD) port map(B,BX);
  N1:entity NOR2(NO2M)
             port map(A1X,A3X,R1);
  N2:entity NOR2(NO2M)
             port map(A2X,A3X,R2);
  N3:entity NOR3(NO3M)
             port map(R1,R2,BX,Y);
end NOR_GATES;
```


Handout 1 Section D

Karnaugh Maps for Logic Design

In this section we introduce Karnaugh maps and show how they can be used to design combinatorial logic circuits.

Static and dynamic hazards are introduced and techniques for removing them using Karnaugh maps are described.

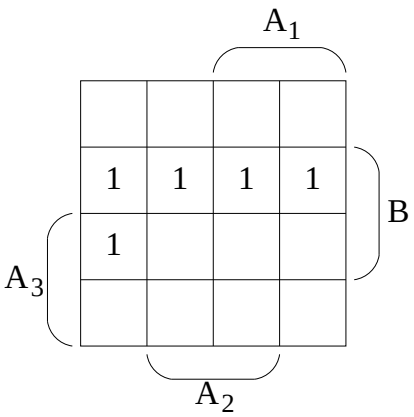
Finally, there is a discussion of the relationship between Karnaugh maps and unit-distance codes.

Karnaugh Map

Karnaugh maps are a powerful visual tool for carrying out simplification and manipulation of logic expressions with up to 5 input variables.

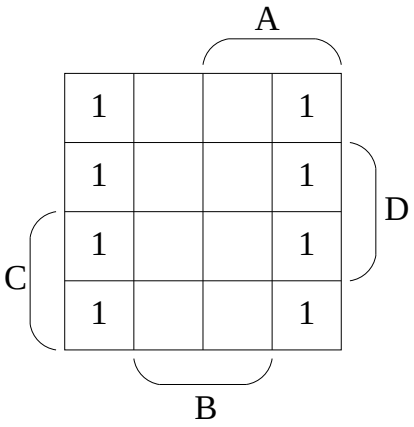
The Karnaugh map is a rectangular array of cells. Each possible state of the input variables corresponds uniquely to one of the cells, and in the cell we write the corresponding output state.

$$Y = \overline{A_3}.B + \overline{A_1}.\overline{A_2}.B$$



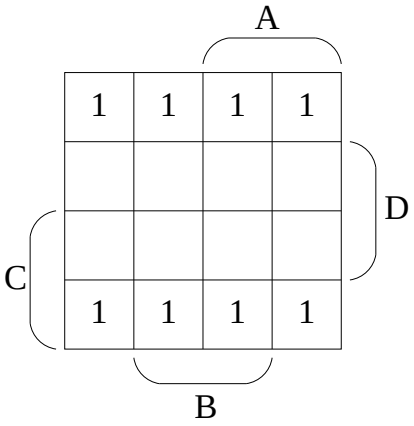
$A_3 B$ \ $A_1 A_2$	00	01	11	10
00				
01	1	1	1	1
11	1			
10				

Map $\overline{B}$ on the Karnaugh map.



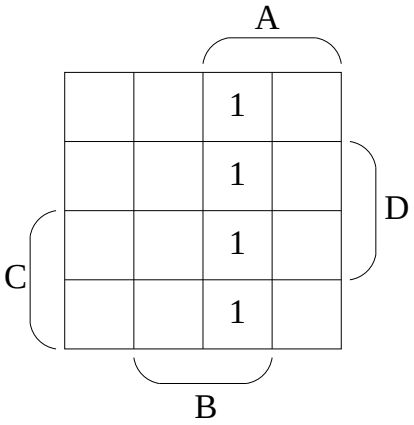
CD \ AB	00	01	11	10
00	1			1
01	1			1
11	1			1
10	1			1

Map $\overline{D}$ on the Karnaugh map.



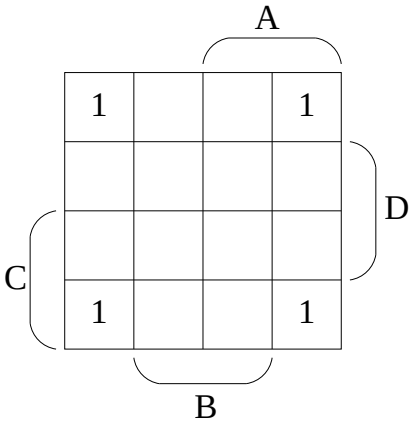
CD \ AB	00	01	11	10
00	1	1	1	1
01				
11				
10	1	1	1	1

Map $A.B$ on the Karnaugh map.



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

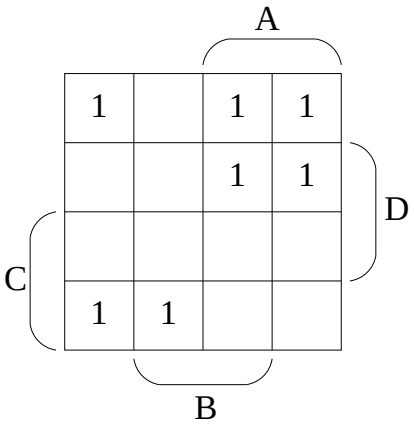
Map $\overline{B}.\overline{D}$ on the Karnaugh map.



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

Map the following expression on a Karnaugh map.

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



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

For a 4 variable Karnaugh map here are the shapes that different Boolean terms produce:

Four variable expression
eg. A.B.C.D

1			

Three variable expression
eg. A.B.C

1	1		

Two variable expression
eg. A.B

1	1		
1	1		

or

1	1	1	1

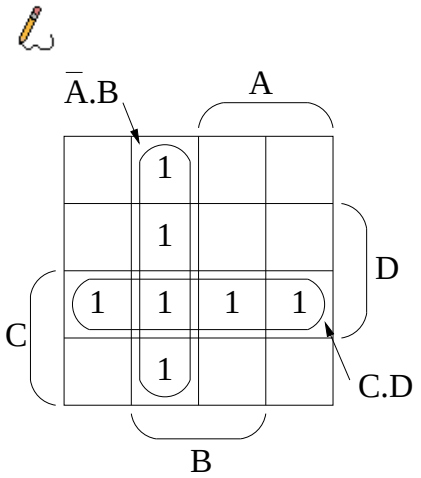
One variable expression
eg. A

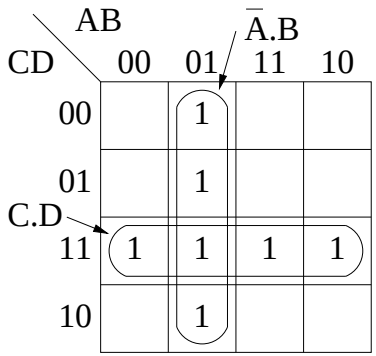
1	1	1	1
1	1	1	1

Plus rotations of these shapes by 90 degrees.

Remember that the Karnaugh map wraps round top-bottom and side-to-side.

Simplify $\overline{A}.B.\overline{D} + B.C.D + \overline{A}.B.\overline{C}.D + C.D$ using a Karnaugh map.





Thus the simplified expression is:

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

Karnaugh Map Circuit Design

For a circuit using NAND gates:

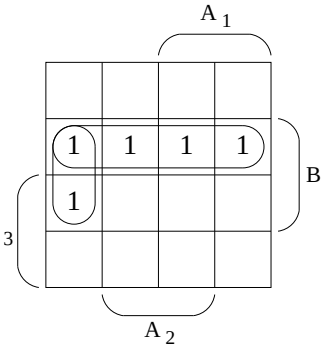
- Write down simplest sum-of-products expression for the output from the Karnaugh map.
- Use De Morgan to convert this to an expression using NAND gates.

For a circuit using NOR gates:

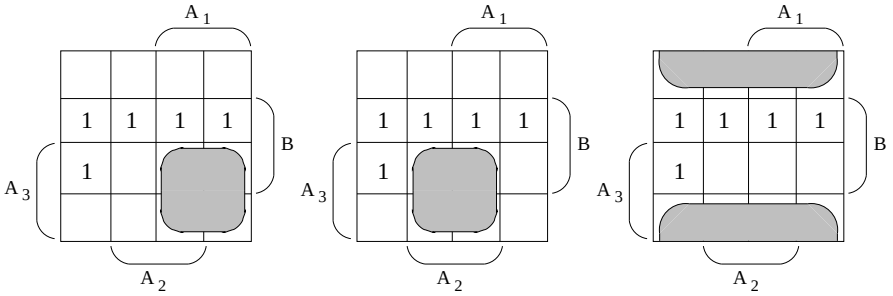
- Use the Karnaugh map to write down simplest sum-of-products expression for **the inverse of the output**. This involves finding terms to cover the blanks between the boxes filled with 1s.
- Use De Morgan to convert this to an expression using NOR gates.

NAND gates for the power-station fans example:

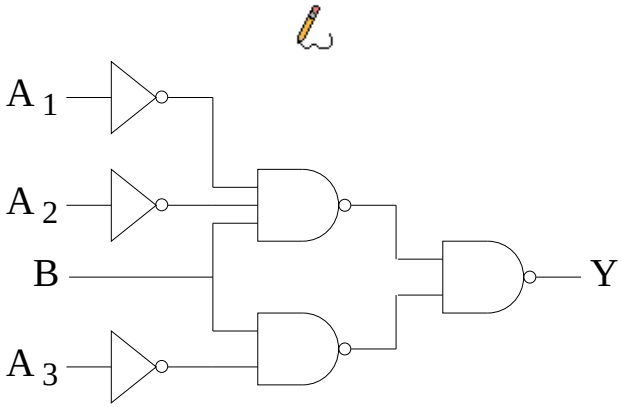
$$\begin{aligned} Y &= \overline{A_3}.B + \overline{A_1}.\overline{A_2}.B \\ &= \overline{(\overline{\overline{A_3}.B}).(\overline{\overline{A_1}.\overline{A_2}.B})} \end{aligned}$$

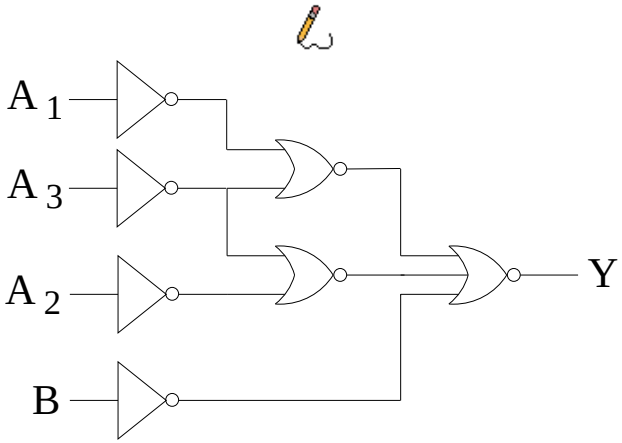


NOR gates for the power-station fans example:



$$\begin{aligned} \overline{Y} &= A_1.A_3 + A_2.A_3 + \overline{B} \\ &= (\overline{\overline{A_1} + \overline{A_3}}) + (\overline{\overline{A_2} + \overline{A_3}}) + \overline{B} \\ \Rightarrow Y &= \overline{(\overline{\overline{A_1} + \overline{A_3}}) + (\overline{\overline{A_2} + \overline{A_3}}) + \overline{B}} \end{aligned}$$





Mapping the Inverse

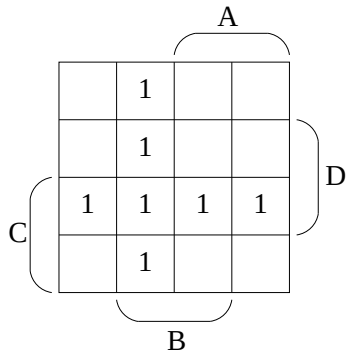
So far the rule has been ‘map the 1s to produce a simple NAND circuit and map the 0s (or gaps) to produce a simple NOR circuit.’

Sometimes you can produce a simpler circuit by mapping the opposite way (i.e. 0s for NAND and 1s for NOR) and then use a final inverter gate to change the output back again.

For example, if we require a NOR solution for this map then it is probably easiest to use

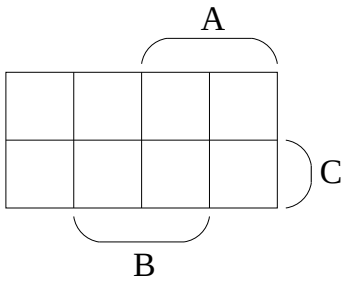
$$Y = \overline{A}.B + C.D$$

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



Other Sizes of Karnaugh Map

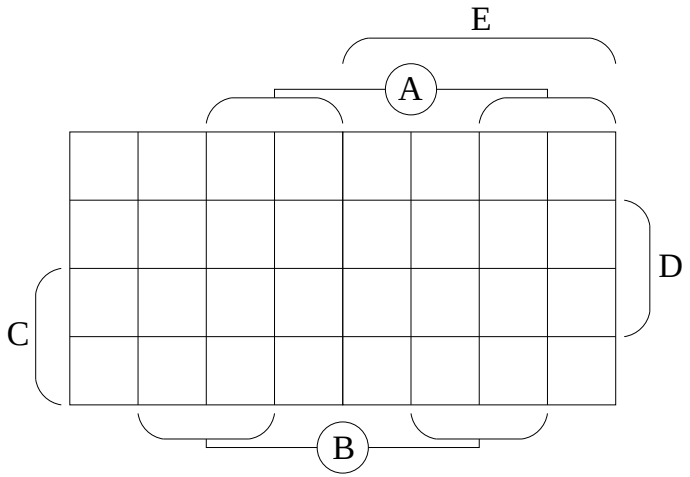
A three variable Karnaugh map.



C \ AB	00	01	11	10
0				
1				

Other Sizes of Karnaugh Map

A five variable Karnaugh map.



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

Don't Care States

In some applications the output state for certain combinations of input variables may not matter. Such states are known as **don't care states** and are marked with an **X** on the Karnaugh map.

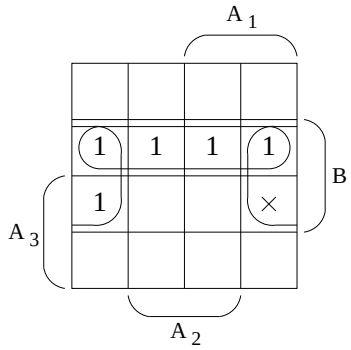
They can be chosen to be 0 or 1, whichever helps to produce the simplest logic (i.e. fewest, simplest terms).

For example suppose, in the power-station fans example, it happens to be impossible for fan number 2 to fail on it's own. We therefore **don't care** whether or not the alarm goes off when only fan number 2 fails (because we know that, in practice, this situation will never occur). This would make $A_1.\overline{A_2}.A_3.B$ a don't care state.

NAND gates for the power-station fans example with don't care state $A_1.\overline{A_2}.A_3.B$.

$$Y = \overline{A_3}.B + \overline{A_2}.B$$

$$= \overline{(\overline{\overline{A_3}.B}).(\overline{\overline{A_2}.B})}$$

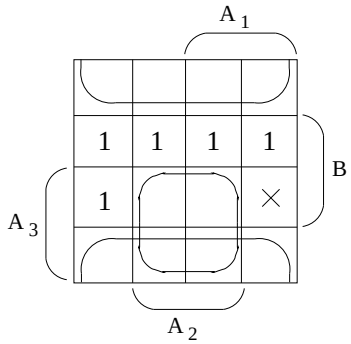


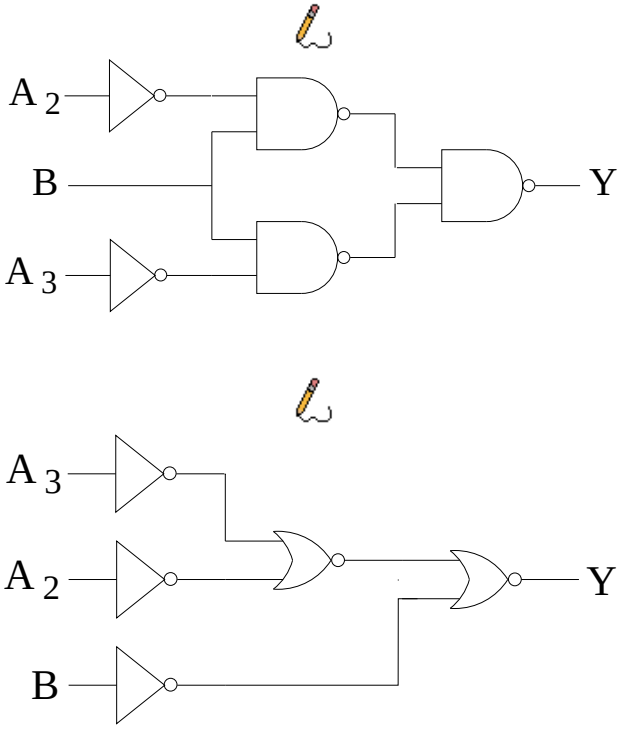
NOR gates for the power-station fans example with don't care state $A_1.\overline{A_2}.A_3.B$.

$$\overline{Y} = A_2.A_3 + \overline{B}$$

$$= (\overline{\overline{A_2} + \overline{A_3}}) + \overline{B}$$

$$\Rightarrow Y = \overline{(\overline{\overline{A_2} + \overline{A_3}}) + \overline{B}}$$

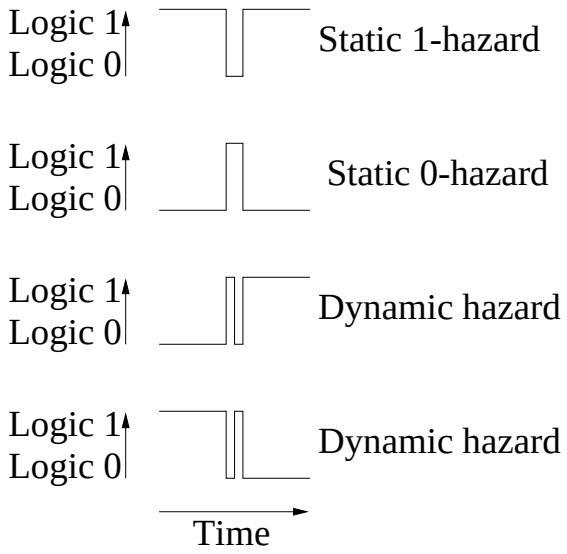




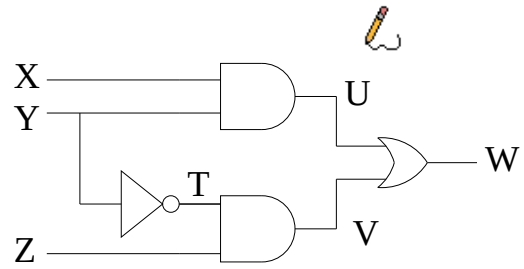
Hazards

A static hazard is when a signal undergoes a momentary transition when it is supposed to remain unchanged.

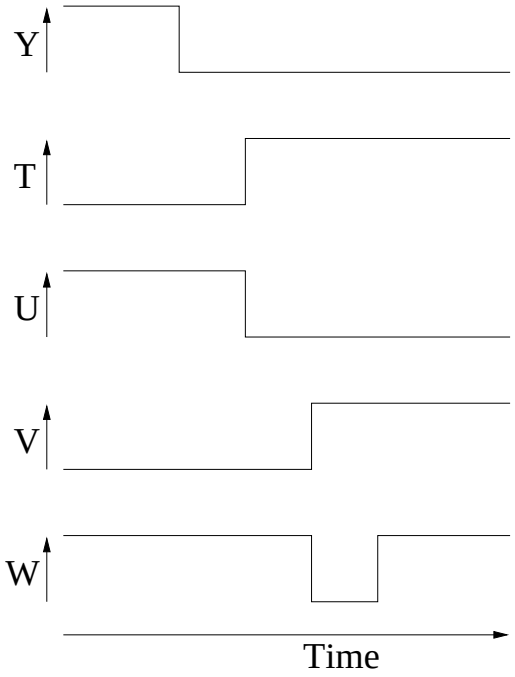
A dynamic hazard is when a signal changes more than once when it is supposed to change just once.



Static 1-Hazard



This circuit implements $W=X.Y+Z.\overline{Y}$



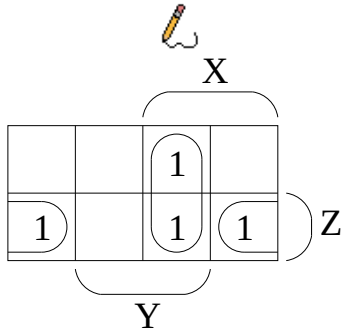
Consider what happens when Z=1 and X=1 and Y changes from 1 to 0.

Removing Hazards

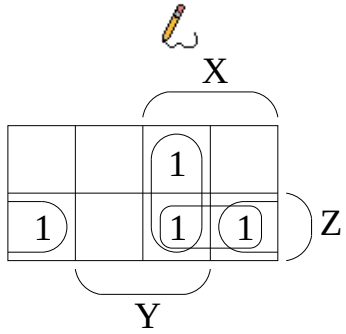
- Fix a static 1-hazard by drawing the Karnaugh map of the output concerned. Make sure all the sum-of-products terms overlap.
- Fix a static 0-hazard by drawing the Karnaugh map of the inverse of the output concerned. Make sure all the sum-of-products terms representing the inverse overlap.
- Fix dynamic hazards by redesigning the circuit to simplify the logic. (This is a 3rd year topic; don't worry about dynamic hazards for now.)

Removing the Hazard

$W = X.Y + Z.\overline{Y}$

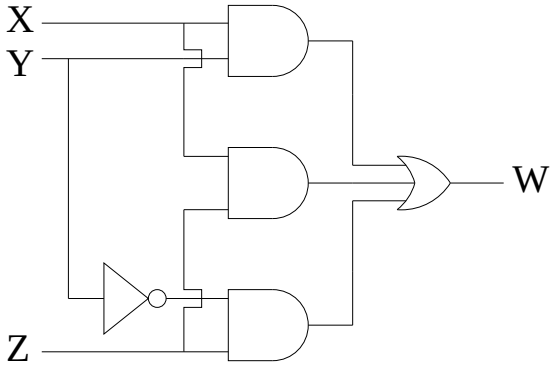


$W = X.Y + Z.\overline{Y} + X.Z$



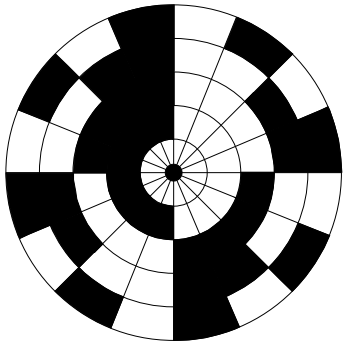
Hazard-Free Circuit

The hazard is removed from the circuit by adding an extra gate to compute the term added to the Karnaugh map: $W = X.Y + Z.\overline{Y} + X.Z$



Unit Distance Codes: Motivation

Applications exist where a code is required in which not more than one bit changes between consecutive numbers.

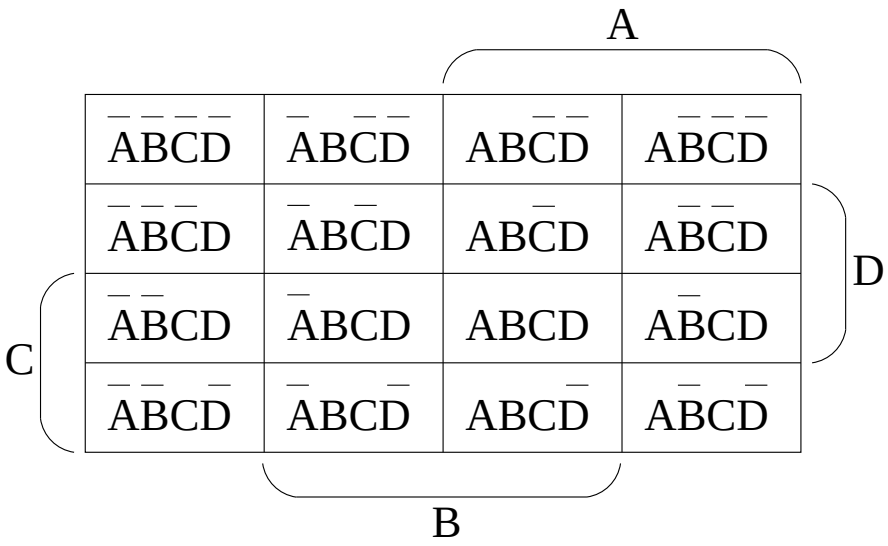


For example, a shaft encoder.

Note the potential for ambiguity during the transition from 0111 to 1000.

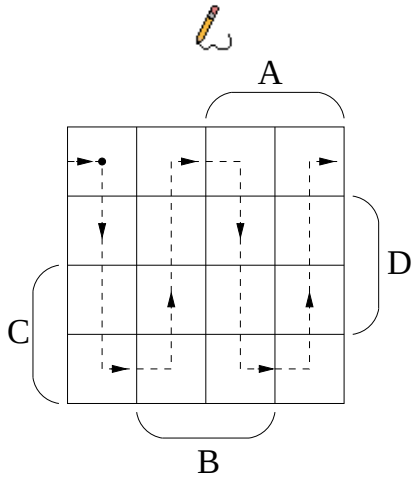
Unit Distance Codes

As you move from a cell to its neighbour in a Karnaugh map only one variable changes. Because the map wraps round this is also true when moving from top to bottom and from far left to far right.



This property can be used to produce a code in which only one bit changes at a time.

An example of generating a unit distance code with a Karnaugh map.



A	B	C	D	A	B	C	D
0	0	0	0	1	1	0	0
0	0	0	1	1	1	0	1
0	0	1	1	1	1	1	1
0	0	1	0	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
0	1	0	1	1	0	0	1
0	1	0	0	1	0	0	0

- This particular unit distance code is called the Grey code.

- Other unit distance codes are possible using a different path around the Karnaugh map.

A unit distance code can be used to produce an improved shaft encoder.