Introduction to Digital Circuits

University of Reading

Module: SE1EB5 Computer and Internet Technologies

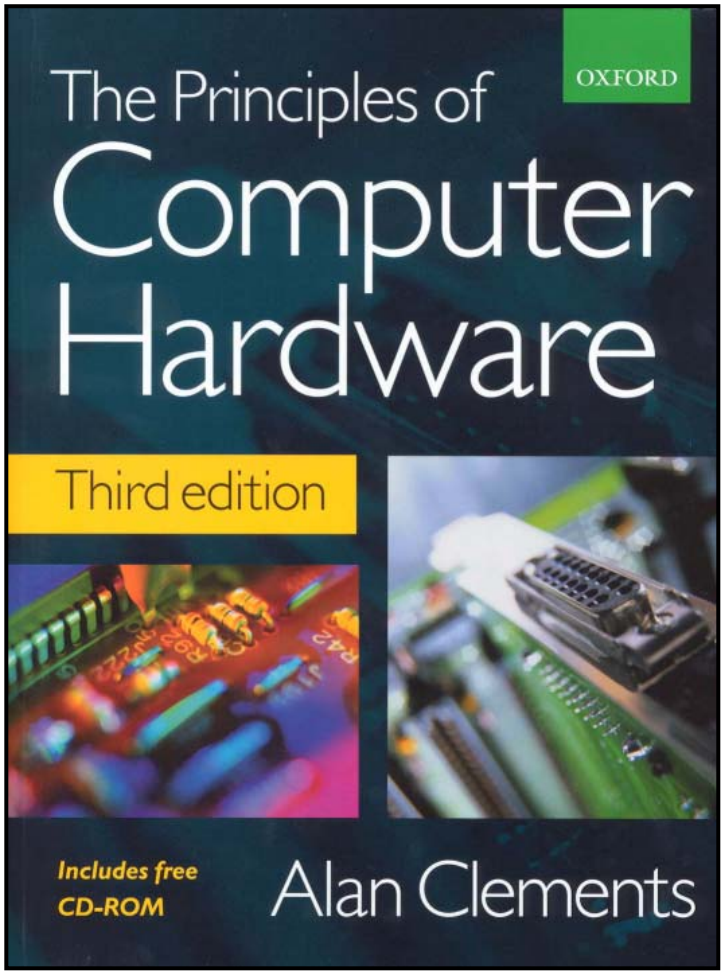
Lecturer: James Grimbleby
URL: http://www.elec.rdg.ac.uk/jbg.html
email: j.b.grimbleby@reading.ac.uk

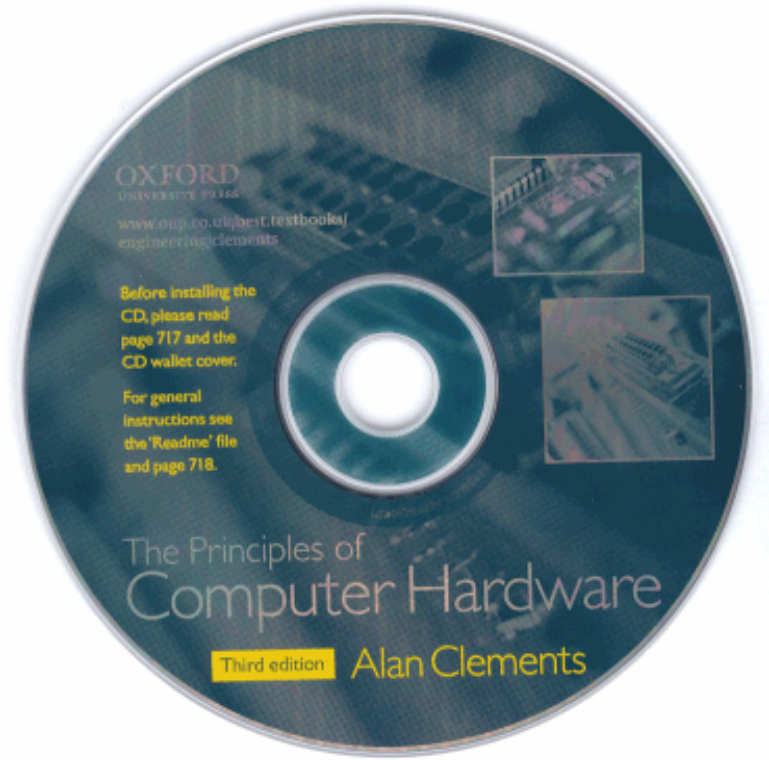
Number of Lectures: 10

Recommended text book:
Alan Clements:
The Principles of Computer Hardware (3rd edition)
Oxford University Press 1999
ISBN 019856453-8

Introduction to Digital Circuits

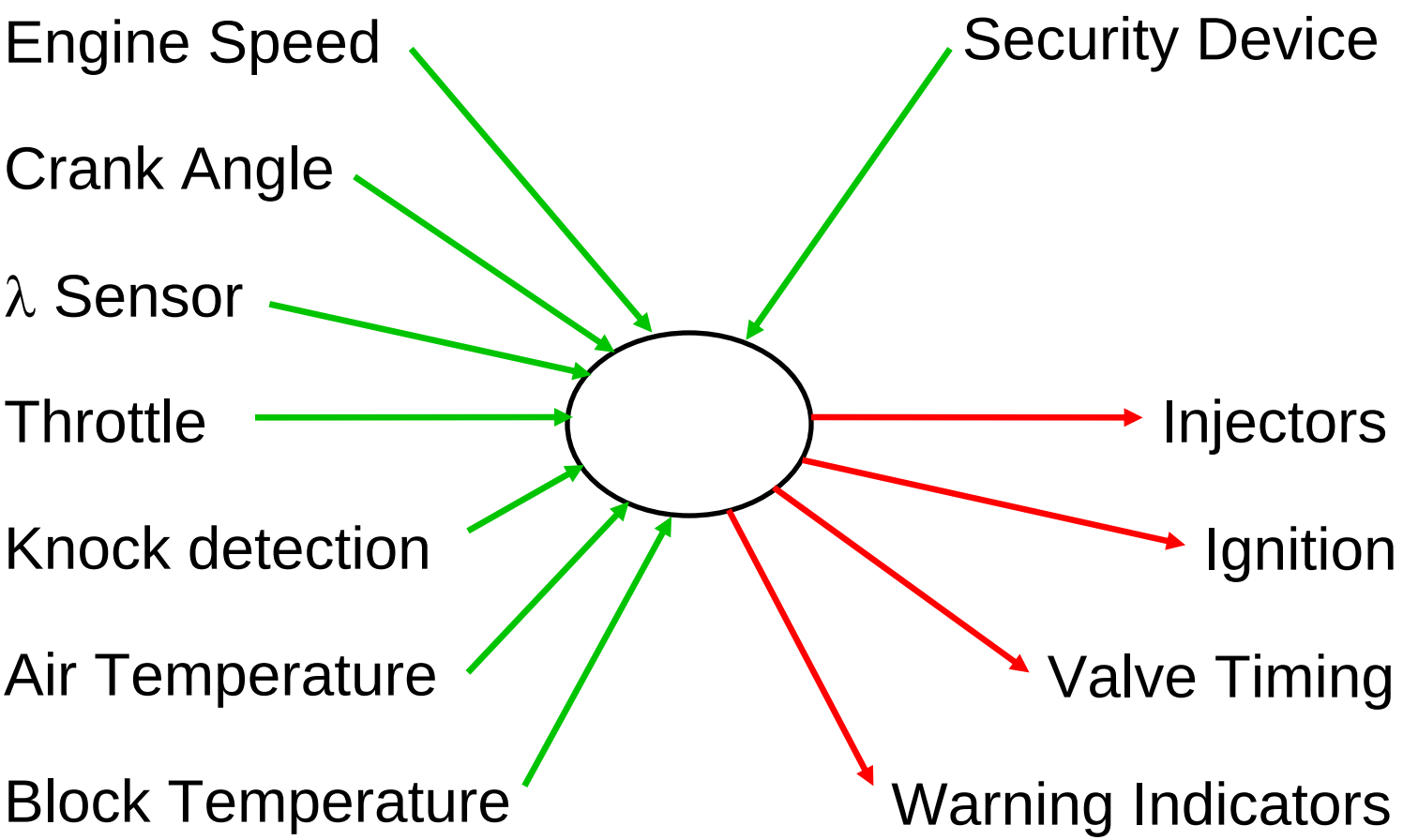
University of Reading





Price: £25 (approx)

Engine Management System



Syllabus

University of Reading

Analogue vs digital; binary codes; error-detecting and error-correcting codes; Hamming distance

Boolean algebra; operations NOT, AND, OR; truth tables; Boolean identities and theorems; Boolean functions; Logic gates; combinational logic systems; analysis of combinational systems; duality

Karnaugh maps; Boolean simplification using K-maps; sum-of-products and product-of-sums forms; incompletely-specified Boolean functions; implementation using logic gates

Syllabus

University of Reading

Universal logic gates; simple implementation method; formal method for NAND implementation; formal method for NOR implementation

Hazards in combinational logic systems; static and dynamic hazards; origin of static hazards in 2-level NAND systems; origin of dynamic hazards; design of hazard-free NAND and NOR systems

Other types of logic gate: EXOR gates; MSI combinational functions: 7-segment, decoders and encoders, multiplexers; programmable logic

Syllabus

University of Reading

Sequential logic systems; RS Flip-flops; D-latch; edge-triggered flip-flops; edge-triggered D-type flip-flop

Asynchronous counters; synchronous counters; shift registers; design of synchronous counters using edge-triggered D-type flip-flops

Edge-triggered JK flip-flops; asynchronous counters, shift registers

Types of logic gate: TTL + variants, CMOS, ECL; comparisons of gate types; decoupling.

Analogue vs Digital

University of Reading

Analogue signals represent physical quantities by a simple proportional relationship

The precision of analogue signals is limited by noise/drift relative to the maximum signal and by non-linearity

Digital signals use numbers to represent physical quantities

There is in principle no limit to the precision of digital signals

Digital systems use binary numbers, rather than decimal, to define digital values

Binary Codes

University of Reading

Digital electronics uses binary values: 0 or 1

These are represented in digital circuits by voltages or currents, for example:

Logic 0:	0 V	→ 0.8 V
Logic 1:	2.0 V	→ 5.0 V

Binary digits are called *bits*

Bits are normally processed in groups: a group of binary digits is called a *binary word*, or more simply, a *word*

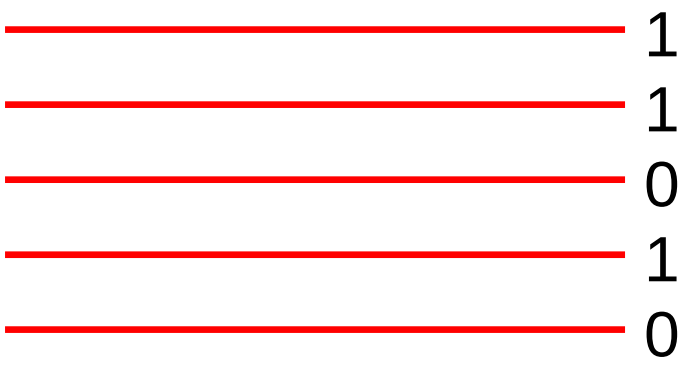
Binary Codes

University of Reading

Digital words can be transmitted / processed in serial or parallel form:

5-bit word - serial data 11010:

5-bit word - parallel data 11010:



Binary Codes

University of Reading

Words represent information by the use of binary codes

Examples of binary codes:

- Natural binary code
- Signed binary code
- Floating-point codes
- Gray code
- Alpha-numeric codes

Extra bits can be added to any of these codes to make them error-detecting or error-correcting

Natural Binary Code

University of Reading

An *n*-bit word:

$$d_{n-1}, \ldots \; d_2, d_1, d_0$$

where d_r are the individual bits (value 0 or 1), represents the positive integer value:

$$2^{n-1}d_{n-1} + \ldots + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$

So the 6-bit natural binary word 010110 represents the integer value:

$$2^5.0 + 2^4.1 + 2^3.0 + 2^2.1 + 2^1.1 + 2^0.0$$
$$= 16 + 4 + 2 = 22$$

Natural Binary Code

University of Reading

Natural binary is the most efficient code in the sense that is uses the minimum number of bits

It is easy to perform arithmetic operations on natural binary code

8-bit natural binary code represents: 0 → 255

16-bit natural binary code represents: 0 → 65535

32-bit natural binary code represents: 0 → 4295M

In general an *n*-bit natural binary code represents: 0 → (2^n-1)

4-Bit Natural Binary Code

University of Reading

Decimal	Natural binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	Natural binary
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Signed Binary Code

University of Reading

In many applications it is necessary to represent negative as well as positive numbers

For example: audio signals, bank account

The most common (and most satisfactory) representation for signed integers is 2s-complement binary

The operations required to add or subtract 2s-complement numbers are identical to those for natural binary

2s-Complement Binary Code

University of Reading

To obtain the 2s-complement representation of a negative number:

1. Obtain natural binary representation of its magnitude
2. Complement (0 → 1, 1 → 0)
3. Add 1

Example: -5

Natural binary	000101	5
Complement	111010	
Add 1	111011	-5

2s-Complement Binary Code

University of Reading

The sign of a 2s-complement number is determined by the most-significant (left-most) bit

8-bit signed binary code represents: -128 → +127

16-bit signed binary code represents: -32768 → +32767

32-bit signed binary code represents: -2147M → +2147M

n-bit signed binary code represents: -2^{n-1} → $+(2^{n-1}-1)$

Hamming Distance

The Hamming distance between two code words is the number of bits that must change to convert one code word into the other

1 0 1 1 0 0 1 1

1 0 0 1 1 0 1 1

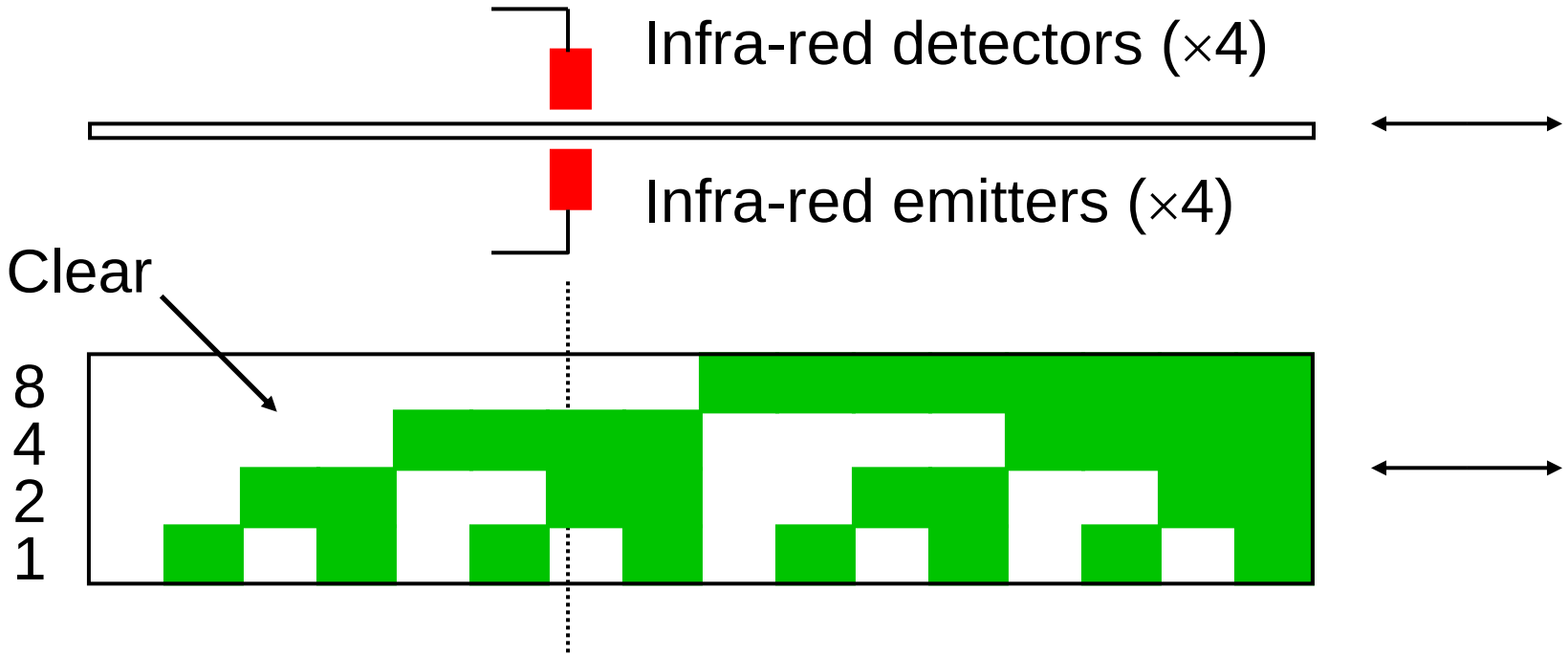
Hamming distance = 2

The minimum distance of a code is the minimum Hamming distance between any pair of words belonging to the code

An *n*-distance code is a code sequence where the Hamming distance between consecutive code words is *n*

Gray Code

Digital position transducer coded using natural binary:



Position = 1001 (binary) = 8 + 1 (decimal) = 9

Gray Code

University of Reading

This should work in principle, but fails in practice because the bits do not change simultaneously:

8	0	0	**1**	1	1
4	1	**0**	0	0	0
2	1	1	1	1	**0**
1	1	1	1	**0**	0
Indicated position:	7	3	11	10	8

Thus spurious positions of 3, 11, and 10 are generated between 7 and 8

This problem is overcome by using Gray code

4-Bit Gray Code

University of Reading

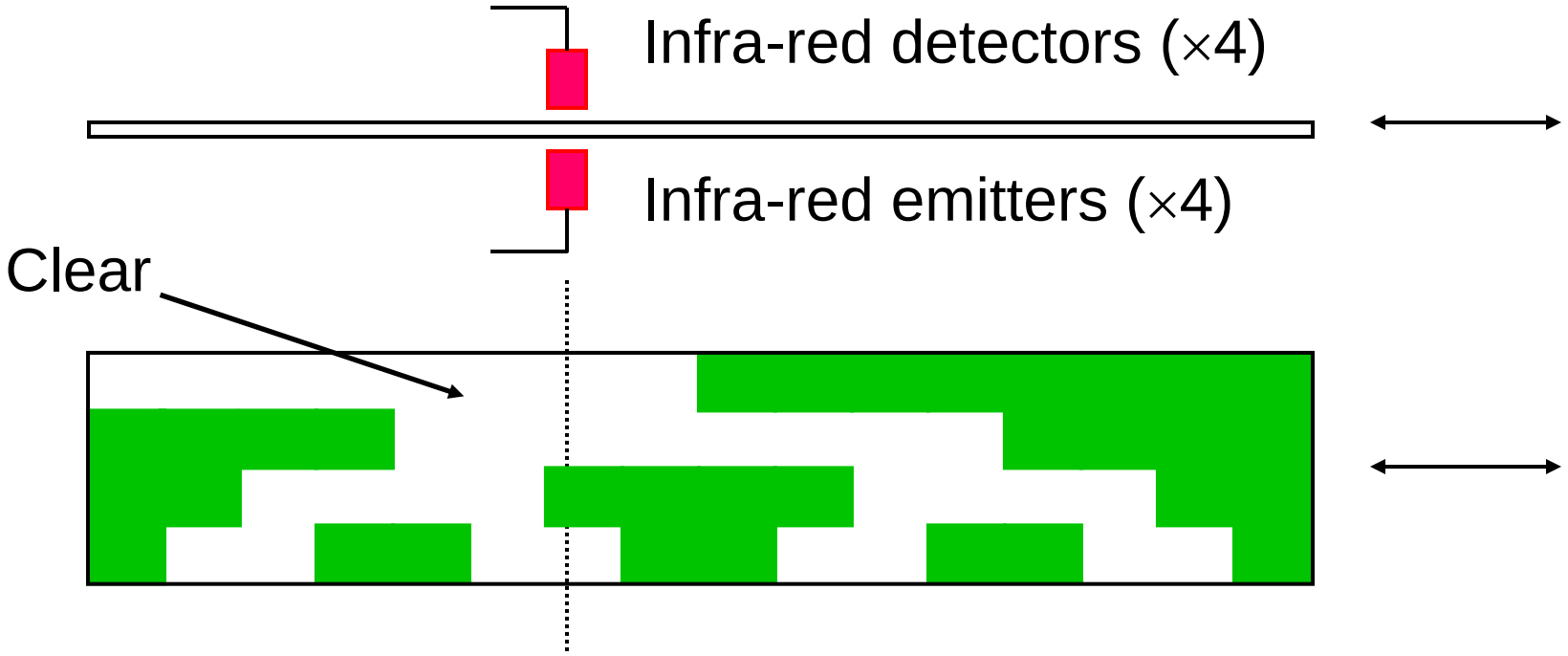
Decimal	Gray code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100

Decimal	Gray code
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Gray Code

University of Reading

Digital position transducer coded using Gray code (unit-distant):



Position = 1101 = 9

Alpha-Numeric Codes

University of Reading

ASCII (American Standard Code for Information Interchange) is a 7-bit code representing alpha-numeric characters

The code includes control characters, punctuation, symbols, digits, letters:

0000000: nul	0001101: cr
0011011: esc	0101011: +
0110000: 0	0111001: 9
1000001: A	1011010: Z
1100001: a	1111010: z
1111111: del	

Error-Detecting Codes

University of Reading

An extra bit added to each binary word allows single errors to be detected (but not corrected)

This bit is called the parity bit and is chosen to make the total number of 1s even (even parity) or odd (odd parity)

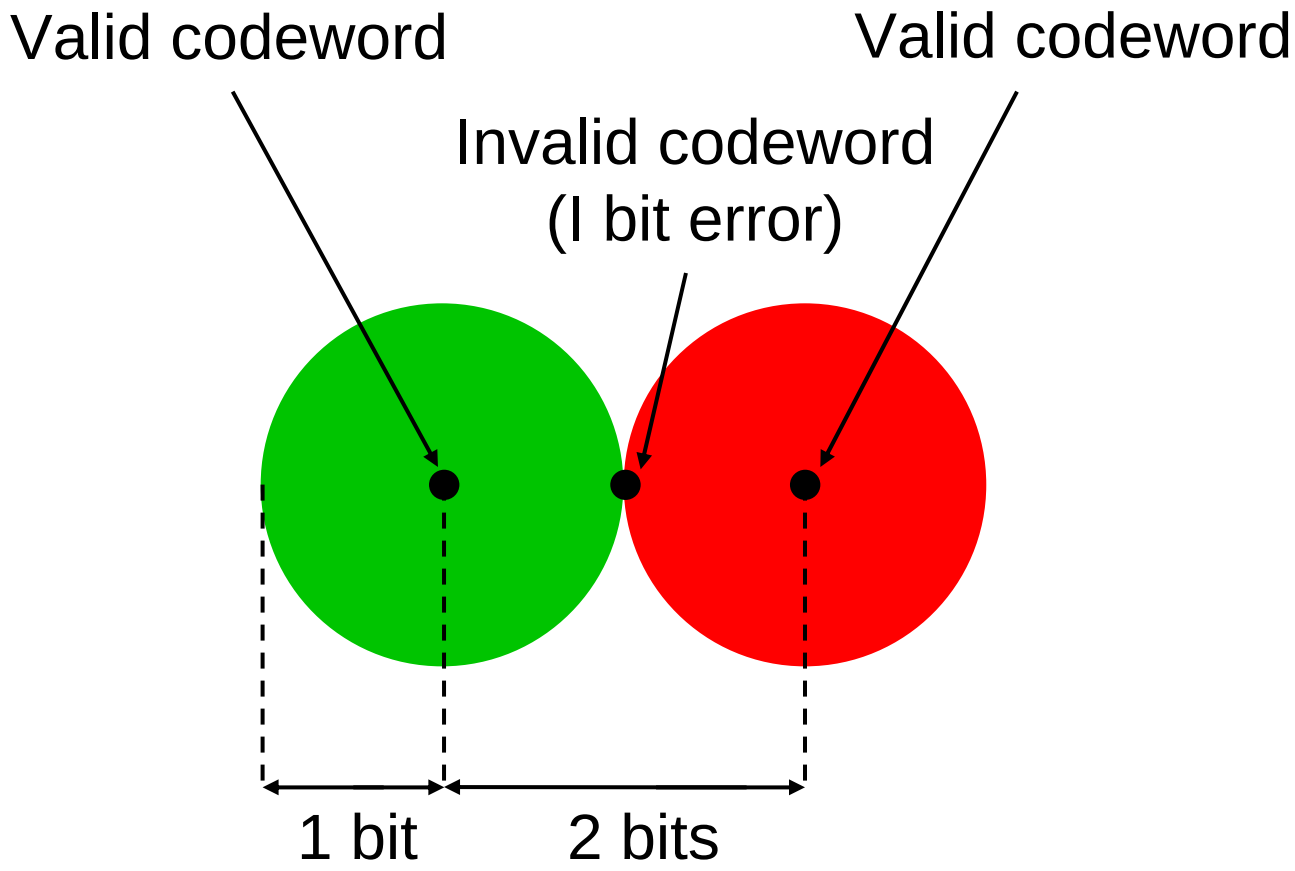
A single error will lead to the number of 1s changing from odd to even, or from even to odd; the change in parity then indicates an error

Parity codes are redundant, and have a minimum Hamming distance of 2

Error-Detecting Codes

University of Reading

Error-detecting codes have a minimum Hamming distance of 2:



Natural Binary Code / Even Parity

University of Reading

Decimal	Even Parity
0	00000
1	10001
2	10010
3	00011
4	10100
5	00101
6	00110
7	10111

Decimal	Even Parity
8	11000
9	01001
10	01010
11	11011
12	01100
13	11101
14	11110
15	01111

Natural Binary Code / Odd Parity

University of Reading

Decimal	Odd Parity
0	10000
1	00001
2	00010
3	10011
4	00100
5	10101
6	10110
7	00111

Decimal	Odd Parity
8	01000
9	11001
10	11010
11	01011
12	11100
13	01101
14	01110
15	11111

Error-Correcting Codes

University of Reading

Error-detecting codes are of little value in 1-way communications systems (cannot be corrected)

If a single bit error occurs in a code with a minimum distance of 3 or greater then the error can be both detected and corrected

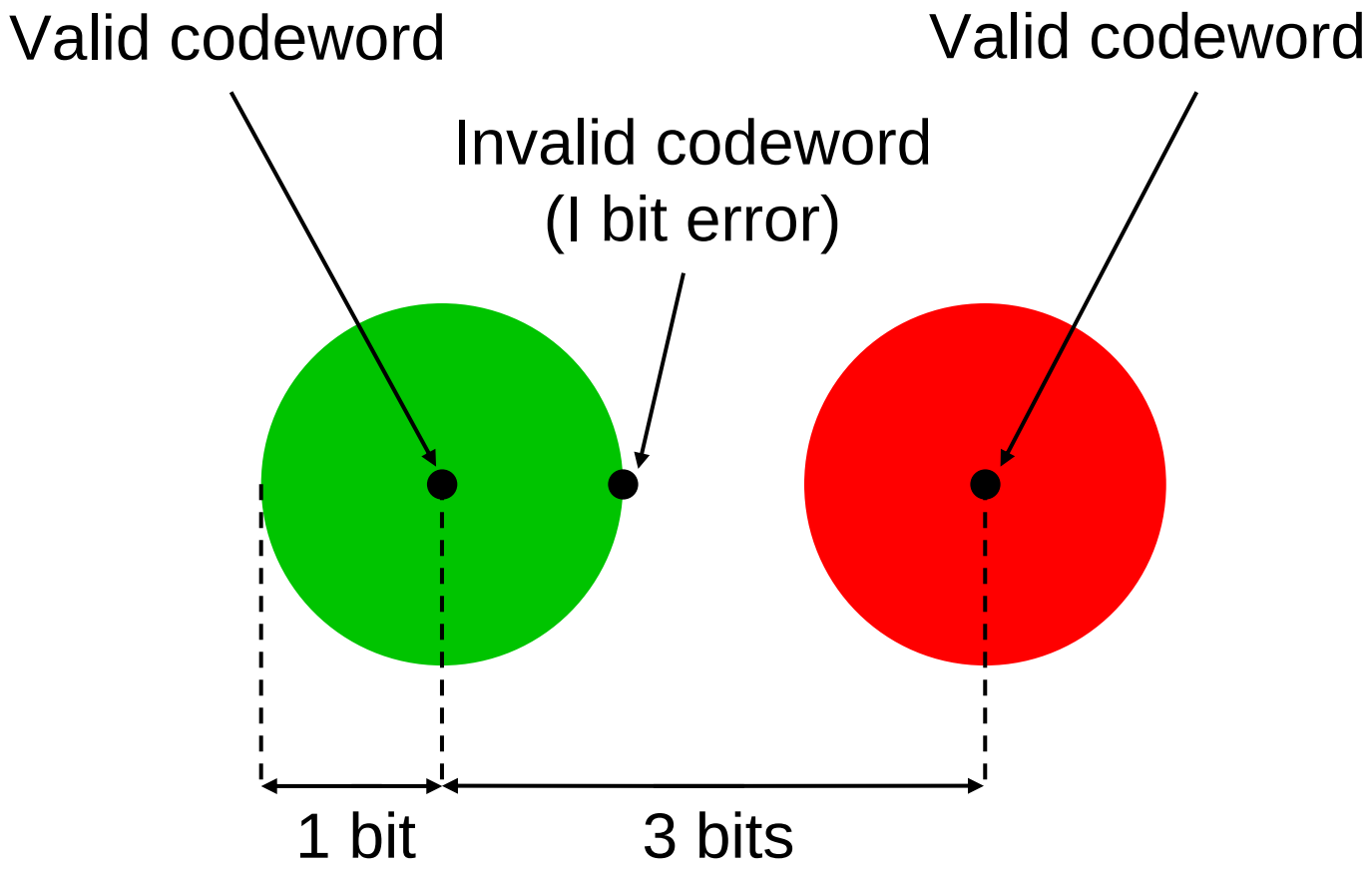
This is because the single error generates a new word which is distance 1 from the original word; it is still distance 2 from any other code word

Sometimes called “forward error correction”

Error-Correcting Codes

University of Reading

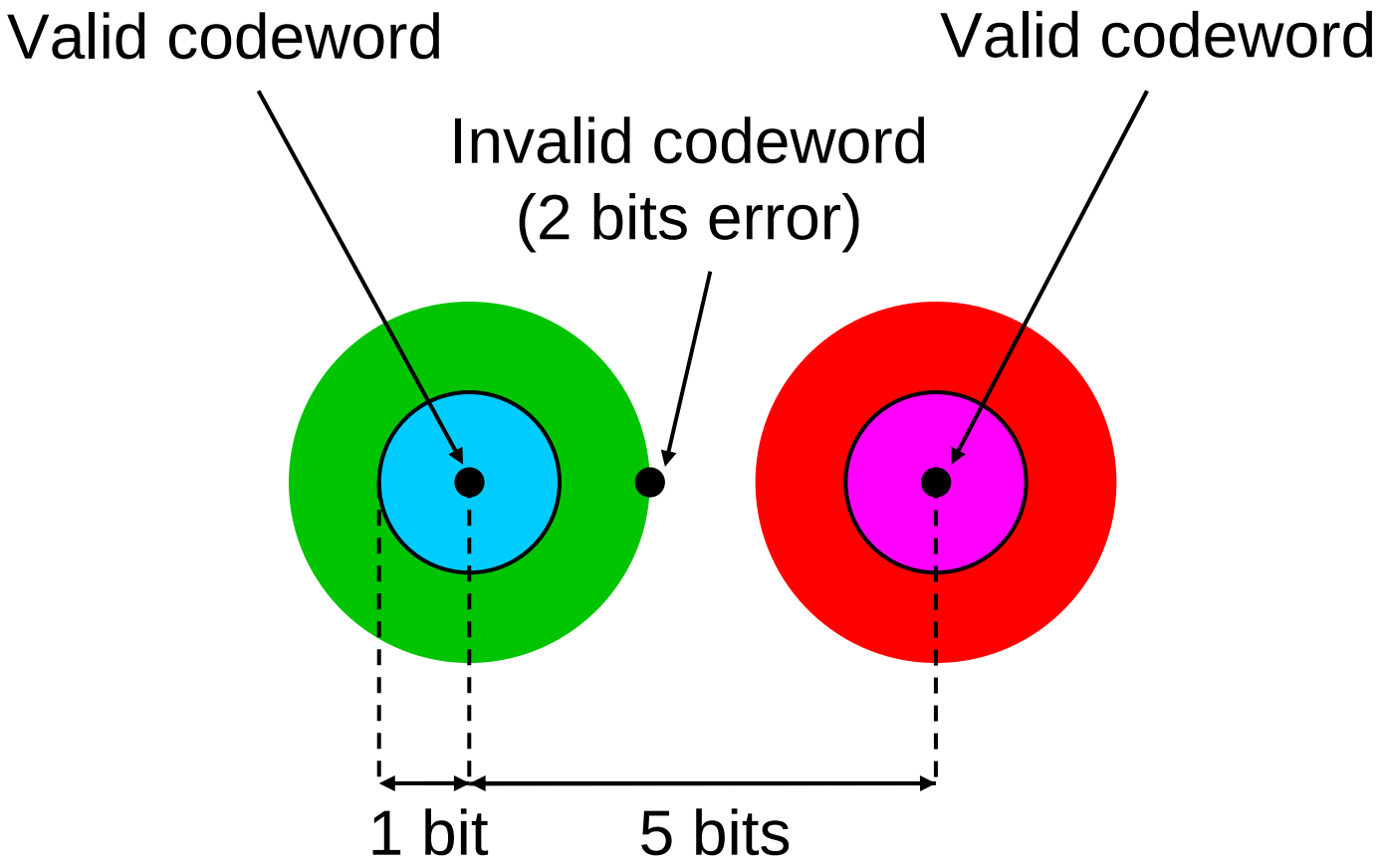
Error-correcting code with minimum Hamming distance of 3:



Error-Correcting Codes

University of Reading

Error-correcting code with minimum Hamming distance of 5:



Error-Correcting Codes

University of Reading

Decimal	E-C code
0	000000
1	011001
2	110010
3	101011

Decimal	E-C code
4	101100
5	110101
6	011110
7	000111

Right-most 3 bits are natural binary, left-most 3 bits are error-correction

Example: 110110 is not a valid codeword

Nearest valid codeword is 110010 (decimal 2)

Error-Correcting Codes

University of Reading

This deliberately-damaged CD can be read without errors

Boolean Algebra

University of Reading

Boolean algebra is an algebra of 2-state variables

Boolean variables can have the values 0 or 1

Boolean Algebra is used to describe digital systems where the binary signals have 2 values

There are three fundamental operations in Boolean algebra: NOT, AND, OR

These are defined by the use of truth tables

Boolean Algebra

NOT A or $\overline{A}$ or A'

A	$\overline{A}$
0	1
1	0

A AND B or $A.B$

A	B	$A.B$
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Algebra

University of Reading

A OR B or $A + B$

A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	1

There are other Boolean operators such as $\oplus$ (exclusive OR) and $\Rightarrow$ (implies), but these can be written in terms of the operators NOT, AND, OR:

$$A \oplus B \equiv A.\overline{B} + \overline{A}.B$$

Boolean Algebra

The operations of AND and OR obey the usual laws of algebra:

Associative laws:

$$A.(B.C) \equiv (A.B).C$$

$$A + (B + C) \equiv (A + B) + C$$

Commutative laws:

$$A.B \equiv B.A$$

$$A + B \equiv B + A$$

Distributive laws:

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

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

Proof: A.(B+C) ≡ A.B + A.C

University of Reading

A	*B*	*C*	*B*+*C*	*A*.(*B*+*C*)	*A*.*B*	*A*.*C*	*A*.*B*+*A*.*C*
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

$A.(B+C) = A.B + A.C$ for all combinations of the variables:
thus $A.(B+C) \equiv A.B + A.C$

Boolean Identities

A number of other identities follow from the definitions of NOT, AND, OR:

$$A + 0 \equiv A \qquad A.0 \equiv 0$$

$$A + 1 \equiv 1 \qquad A.1 \equiv A$$

$$A + A \equiv A \qquad A.A \equiv A$$

$$A + \overline{A} \equiv 1 \qquad A.\overline{A} \equiv 0$$

Theorems of DeMorgan:

$$\overline{A.B} \equiv \overline{A} + \overline{B}$$

$$\overline{A + B} \equiv \overline{A}.\overline{B}$$

Proof of DeMorgan's Theorems

University of Reading

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

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

Boolean Functions

University of Reading

There is no unique form for a Boolean algebraic expression. Consider the function:

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

By "multiplying out" the brackets and using DeMorgan's Theorems:

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

This is the *sum-of-products* standard form. An alternative standard form is the *product-of-sums*:

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

Algebraic Manipulation of Boolean Functions

University of Reading

Some simplification can be achieved by multiplying out brackets and using Boolean identities (including DeMorgan's theorems):

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

This is the simplest *sum-of-products* form for *F*

Algebraic Manipulation of Boolean Functions

University of Reading

This process does not guarantee to produce the simplest *sum-of-products* form

Consider the function:

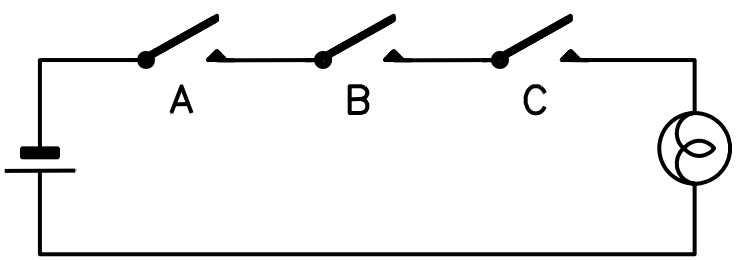
$$F = X.Y + \overline{X}.Z + Y.Z$$

There is no obvious way of simplifying this function algebraically. Nevertheless, the simplest form of the function is:

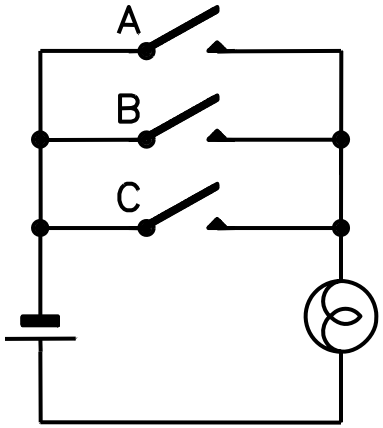
$$F = X.Y + \overline{X}.Z$$

Graphical methods for obtaining the simplest form will be discussed later

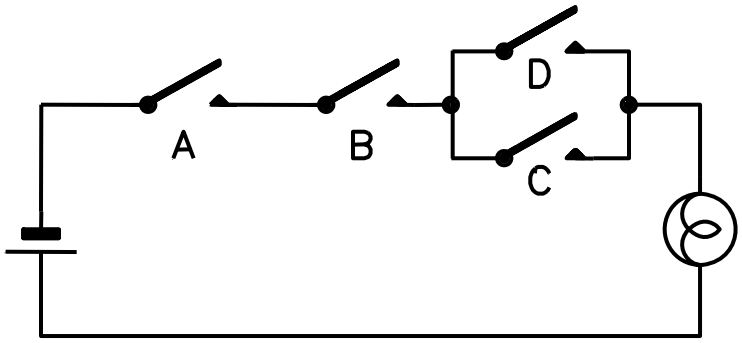
Switch Logic



$$F = A.B.C$$



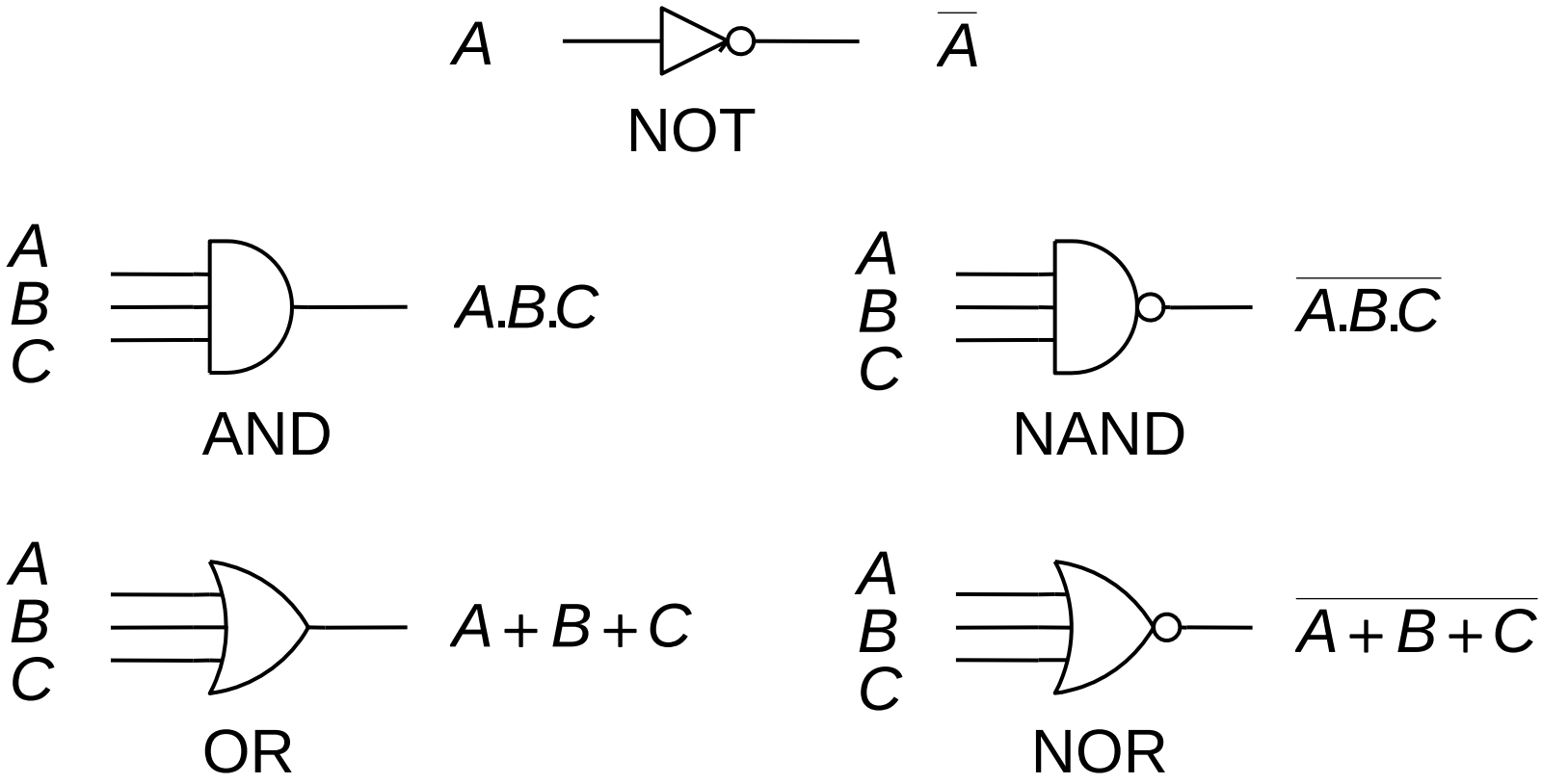
$$F = A + B + C$$



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

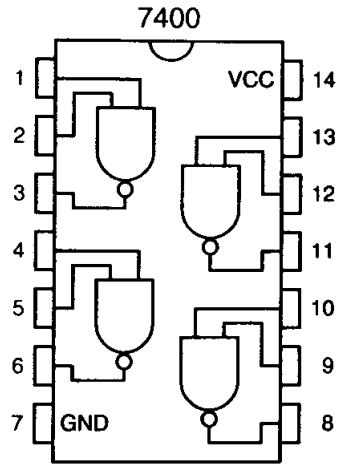
Logic Gates

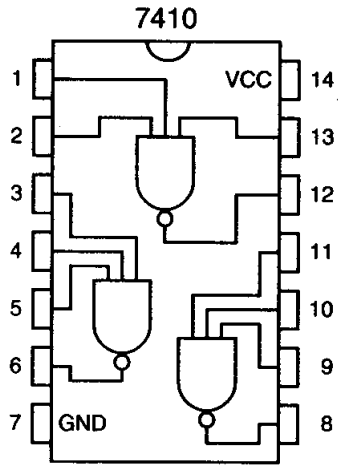
Binary signals are processed in electronic digital systems by logic gates:

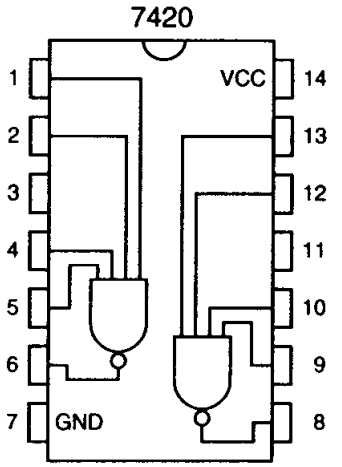


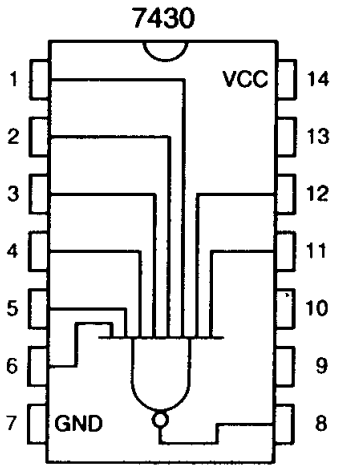
Logic Gates

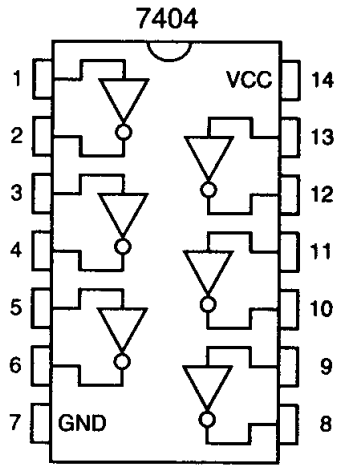
University of Reading

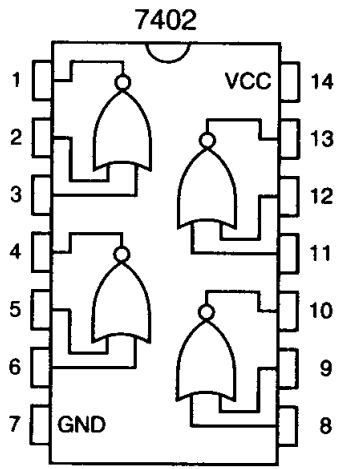






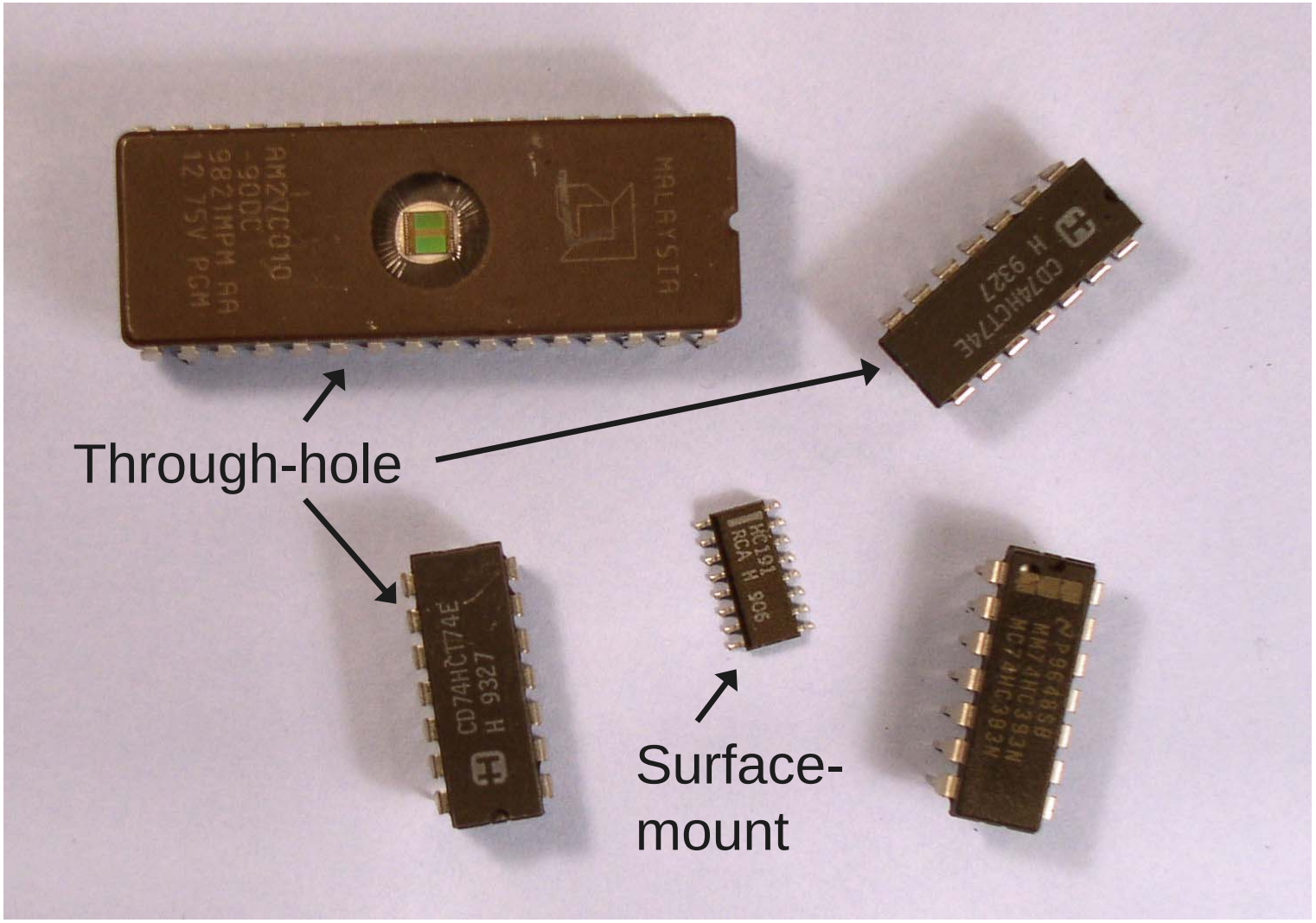






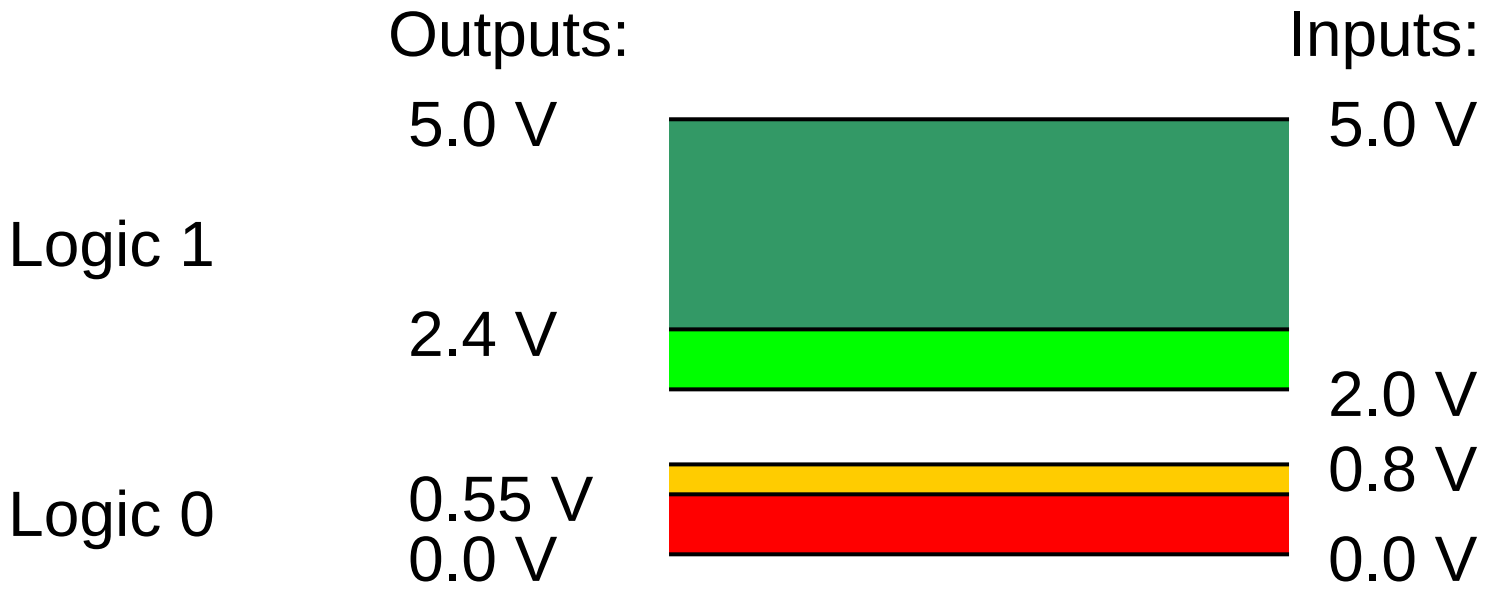
Logic Gates

University of Reading



Logic Signals

Binary values are normally represented by voltages. For example in HCT logic:



Noise immunity: Logic 0 = 0.25 V
Logic 1 = 0.4 V

Combinational Logic Systems

University of Reading

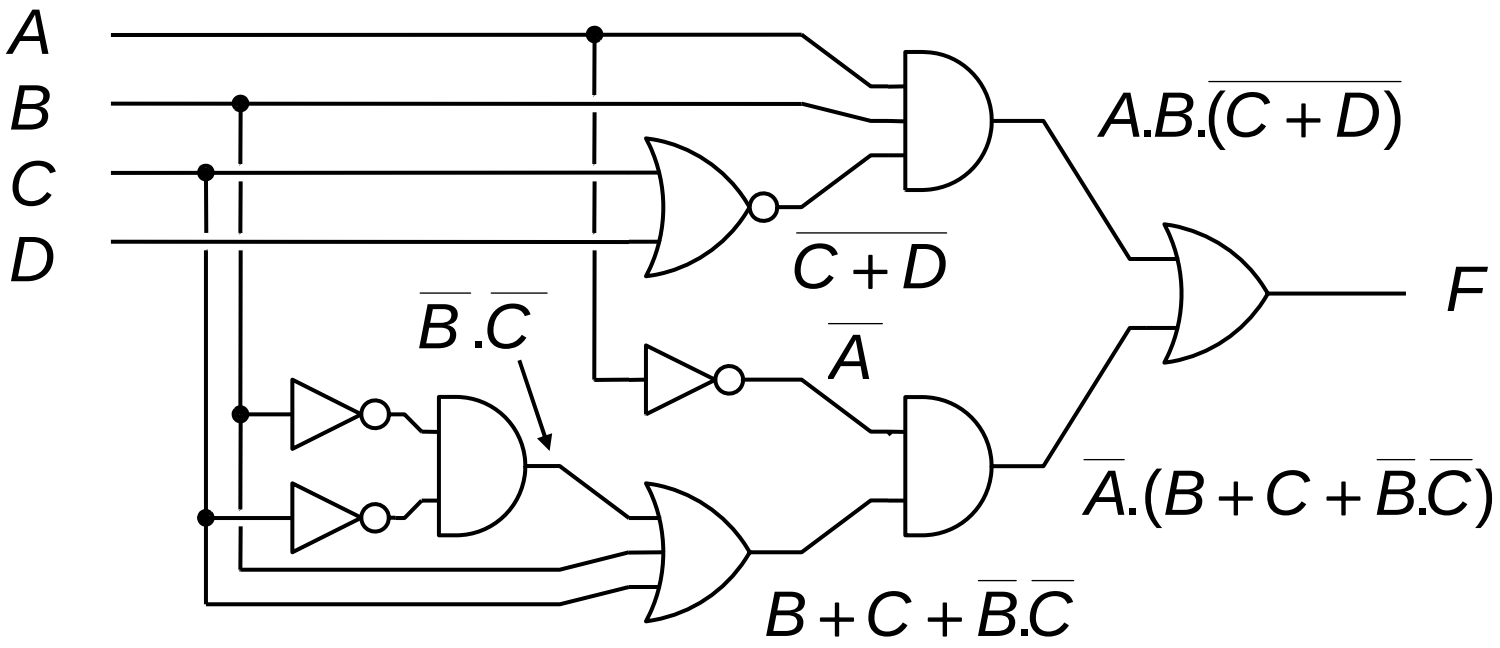
Systems of logic gates where there is no feedback are called *combinational* logic systems

Combinational logic systems have the property that the output or outputs depend only on the present state of the inputs

Combinational systems can therefore be specified by a Boolean expression.

Logic systems with feedback have the property of memory and cannot be specified by a single Boolean expression.

Combinational Logic Systems



This combinational system implements the Boolean function:

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

Duality

A Boolean equation remains valid when converted to its *dual* by changing all variables and constants to their inverses, and replacing AND by OR, and OR by AND

For example if: $F = \overline{A}.(B + C + \overline{B}.\overline{C}) + A.B.\overline{(C + D)}$

or: $F = \{\overline{A}.(B + C + \{\overline{B}.\overline{C}\})\} + \{A.B.\overline{(C + D)}\}$

then: $\overline{F} = \{A + (\overline{B}.\overline{C}.\{B + C\})\}.\{\overline{A} + \overline{B} + \overline{(\overline{C}.\overline{D})}\}$

or: $\overline{F} = \{A + \overline{B}.\overline{C}.\{B + C\}\}.\{\overline{A} + \overline{B} + \overline{\overline{C}.\overline{D}}\}$

Karnaugh Maps

University of Reading

Karnaugh maps (or K-maps) are graphical representations of Boolean functions and are used to simplify Boolean expressions

A K-map has one square for each combination of values of the variables

1-variable K-Map:

$$F = \overline{A}$$

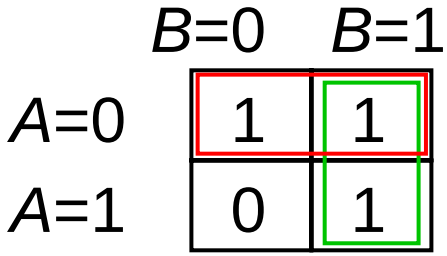
A=0	1
A=1	0

Karnaugh Maps

University of Reading

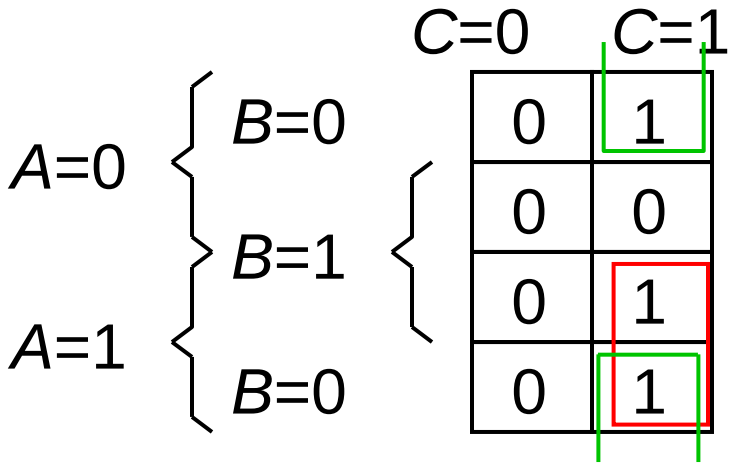
2-variable K-map:

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



3-variable K-map:

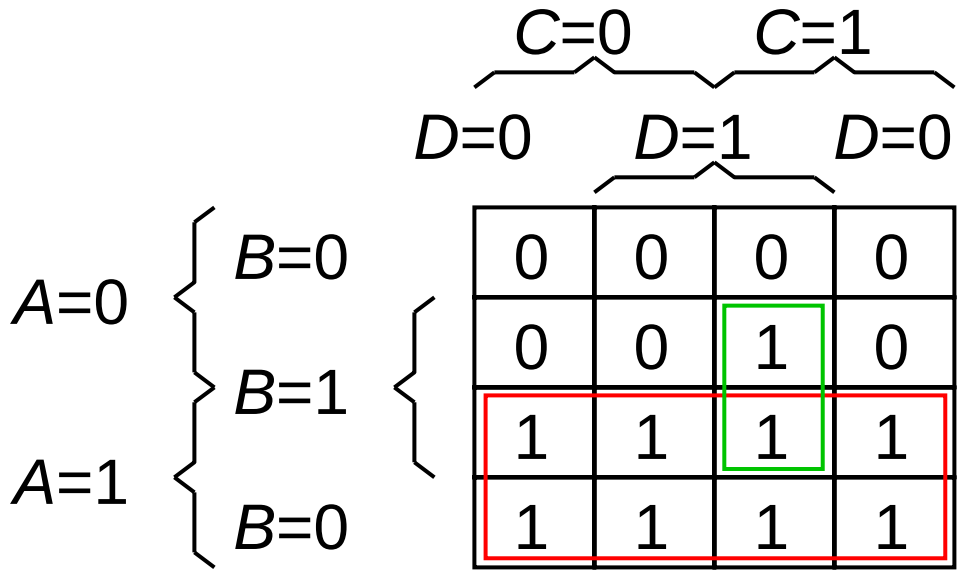
$$F = C.(A + \overline{B})$$



Karnaugh Maps

4-variable K-map:

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



Note that adjacent columns and adjacent rows are unit-distant

Karnaugh Maps

A complication of 3- and 4-variable maps is that they must be regarded as being wrapped around on themselves:

1	1	0	0
0	0	0	0
0	0	0	0
1	1	0	0

$$F = \overline{B}.\overline{C}$$

0	0	0	0
1	0	0	1
0	0	0	0
0	0	0	0

$$F = \overline{A}.B.\overline{D}$$

1	0	0	1
0	0	0	0
0	0	0	0
1	0	0	1

$$F = \overline{B}.\overline{D}$$

Boolean Simplification

University of Reading

The preferred method for simplifying Boolean expressions in 4 variables or less uses K-maps

The first stage is to obtain the *sum-of-products* standard form using algebraic manipulation

The function is then represented in K-map form

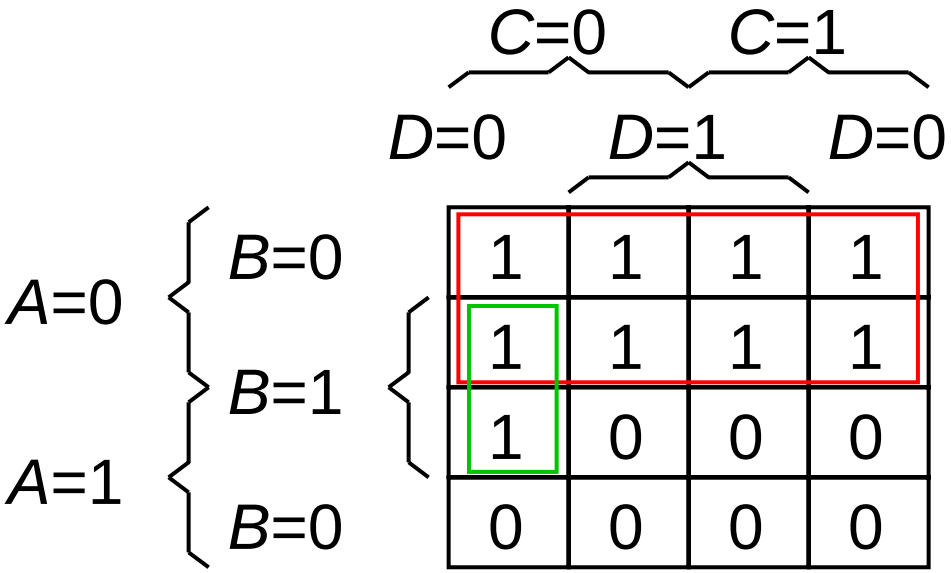
Finally the simplest form for the expression is extracted from the K-map

To obtain the simplest *sum-of-products* form the K-map is inspected for the largest groups first; then for progressively smaller groups

Boolean Simplification

University of Reading

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



Groups of 16: none
Groups of 8: $\overline{A}$
Groups of 4: none
Groups of 2: $B.\overline{C}.\overline{D}$
Groups of 1: none

Simplest *sum-of-products* form for *F* is:

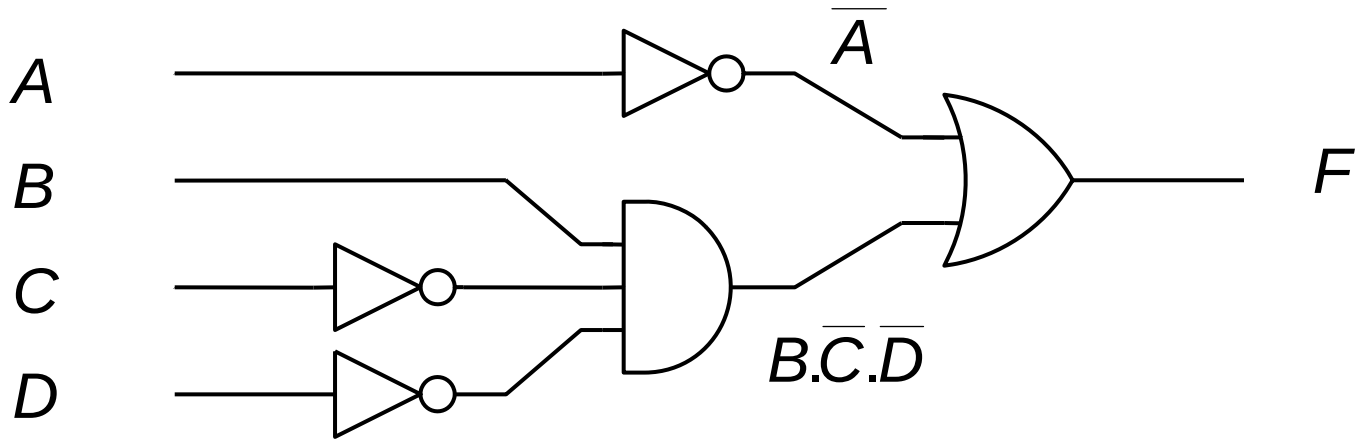
$$F = \overline{A} + B.\overline{C}.\overline{D}$$

Boolean Implementation

The simplest *sum-of-products* form for *F*:

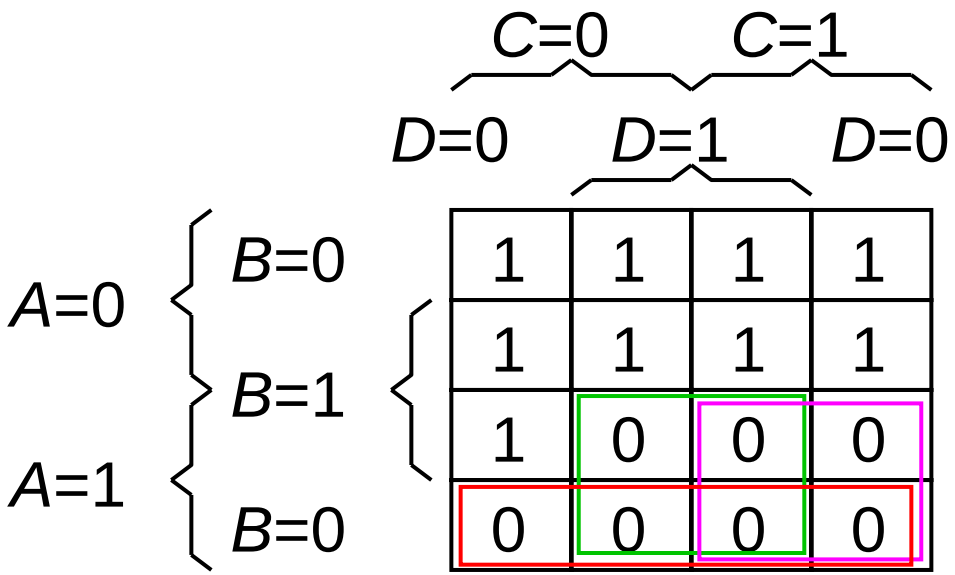
$$F = \overline{A} + B.\overline{C}.\overline{D}$$

can now be implemented using a combination of AND, OR and NOT gates:



Boolean Simplification

To obtain a function F in simplest *product-of-sums* form, the first step is to obtain $\overline{F}$ in simplest *sum-of-products* form:



Simplest *sum-of-products* form for $\overline{F}$ is:

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

Boolean Simplification

University of Reading

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

Now use duality to obtain *F* in simplest *product-of-sums* form:

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

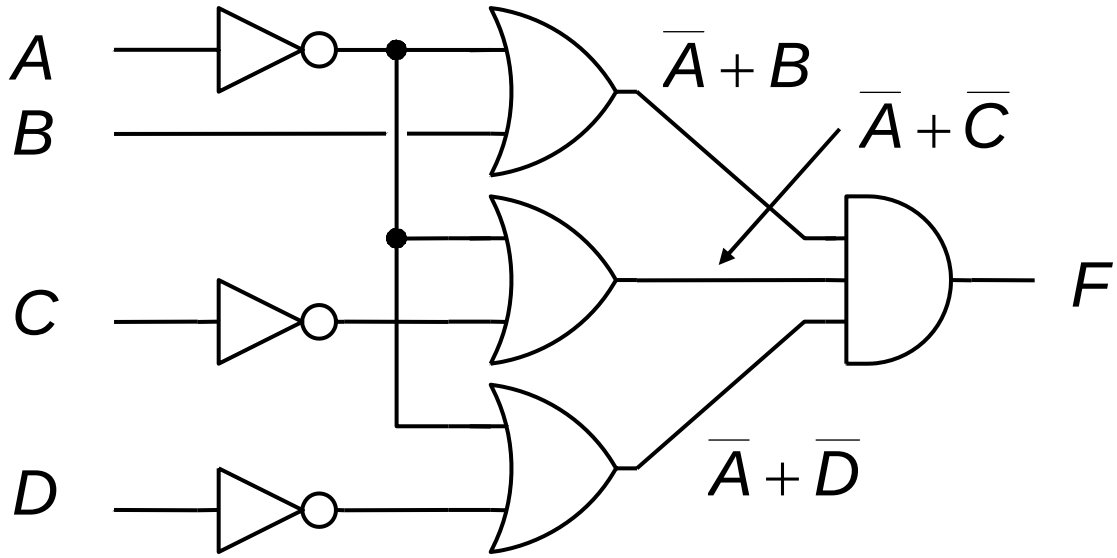
OR invert and use DeMorgan's Theorems twice to obtain *F* in simplest *product-of-sums* form:

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

Boolean Implementation

University of Reading

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



Incompletely-Specified Functions

University of Reading

It often happens that the value of a function is not specified for some combinations of the input variables

In other words the value of the function is "don't-care" and this is represented by X

K-maps can be used to simplify incompletely-specified functions, with the don't-care conditions being assigned either 0 or 1

Incompletely-specified functions cannot be defined by a Boolean expression; instead a truth table is used

Incompletely-Specified Functions

University of Reading

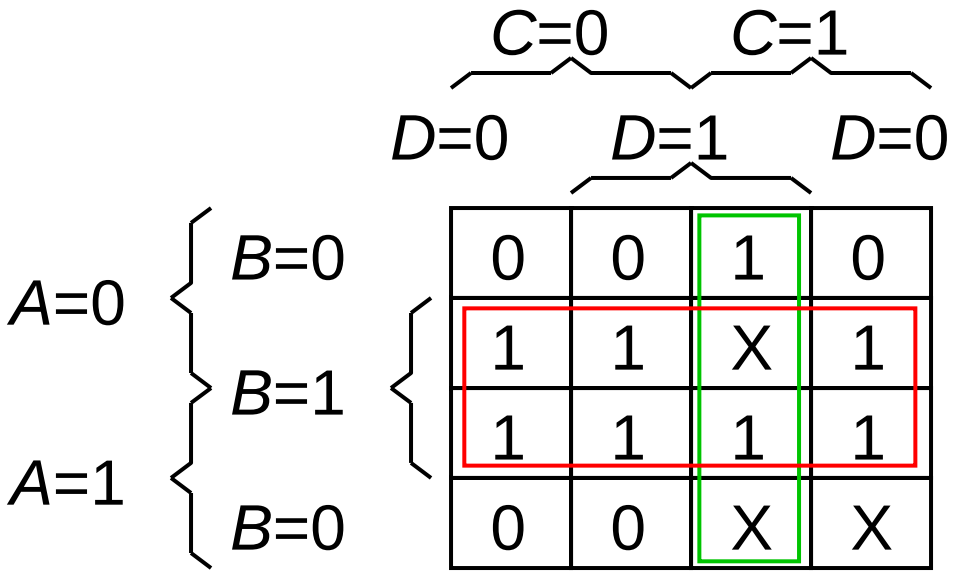
Function *F* defined by a truth table:

A	*B*	*C*	*D*	*F*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	X

A	*B*	*C*	*D*	*F*
1	0	0	0	0
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Incompletely-Specified Functions

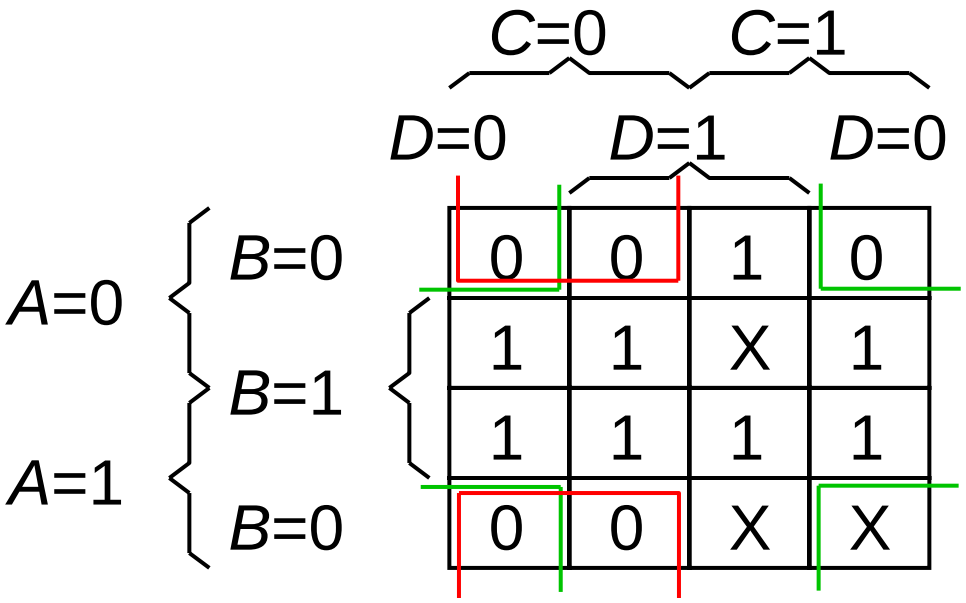
Each row of the truth table corresponds to one square of the K-map:



Simplest *sum-of-products* form for *F* is:

$$F = B + C.D$$

Incompletely-Specified Functions



Simplest *sum-of-products* form for $\overline{F}$ is:

$$\overline{F} = \overline{B}.\overline{C} + \overline{B}.\overline{D}$$

Simplest *product -of-sums* form for F is:

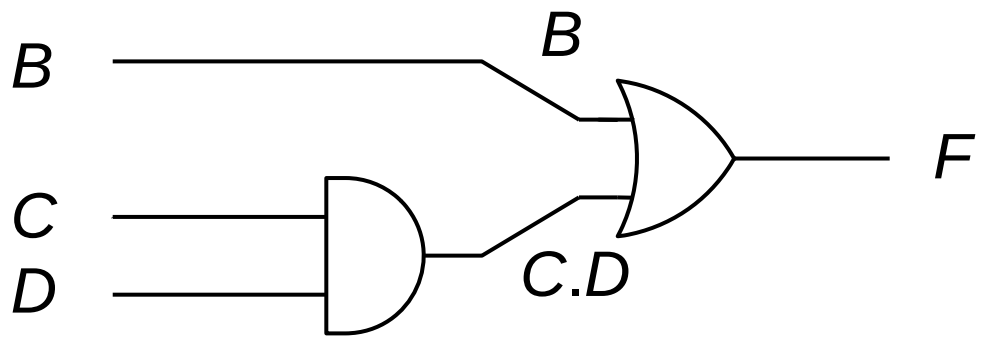
$$F = (B + C).(B + D)$$

Boolean Implementation

A Boolean function can be implemented by first obtaining the simplest *sum-of-products* form:

$$F = B + C.D$$

The overall OR function is then implemented using an OR gate, each of the terms is implemented using AND gates, and NOT gates are used where required to invert the inputs:

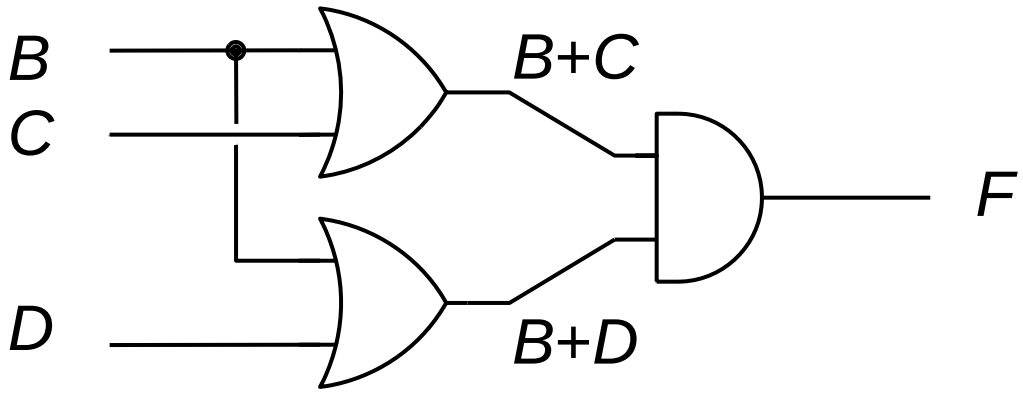


Boolean Implementation

Alternatively implementation can start from the simplest *product-of-sums* form:

$$F = (B + C).(B + D)$$

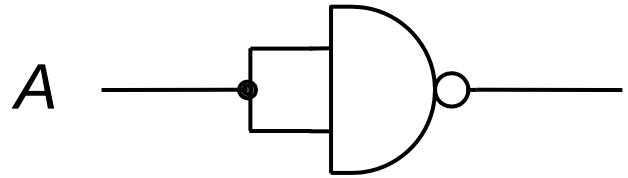
The overall AND function is then implemented using an AND gate, each of the factors is implemented using OR gates, and NOT gates are used where required to invert the inputs:



Universal Logic Gates

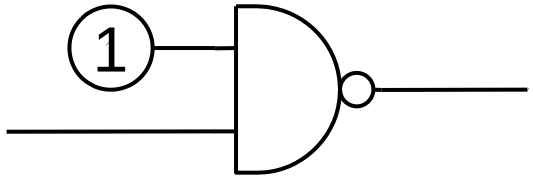
NAND gates:

NOT



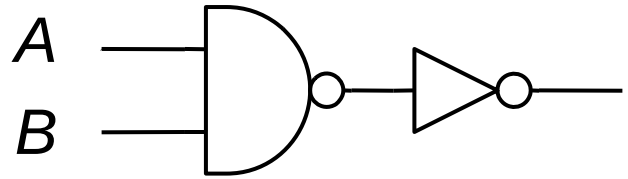
$$\overline{(A.A)} = \overline{A}$$

NOT



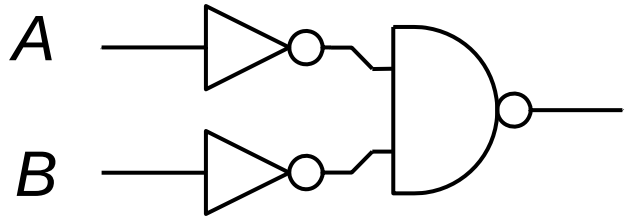
$$\overline{(A.1)} = \overline{A}$$

AND



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

OR

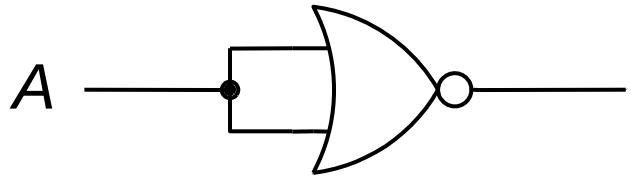


$$\overline{(\overline{A}.\overline{B})} = A + B$$

Universal Logic Gates

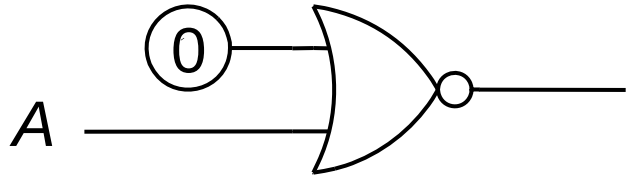
NOR gates:

NOT



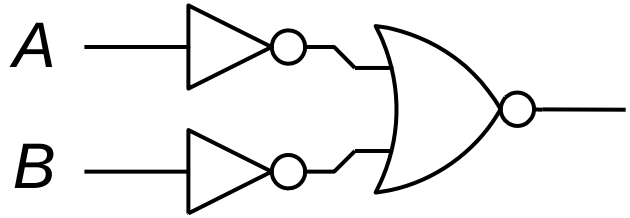
$$\overline{(A+A)} = \overline{A}$$

NOT



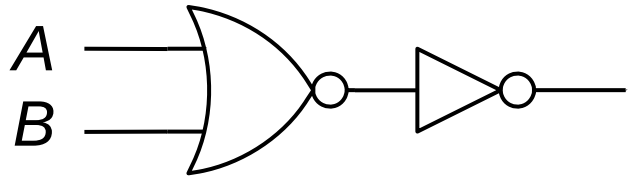
$$\overline{(A+0)} = \overline{A}$$

AND



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

OR



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

Implementation in NAND

University of Reading

1. Obtain the function in simplest *sum-of-products* form:

$$F = A.B + C.D.E + ...$$

2. Invert the function twice (thus leaving it unchanged):

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

3. Use DeMorgan's theorem on lower inversion:

$$F = \overline{\overline{(A.B)}.\overline{(C.D.E)}. ...}$$

4. The function can now be implemented using only NAND gates

Implementation in NAND

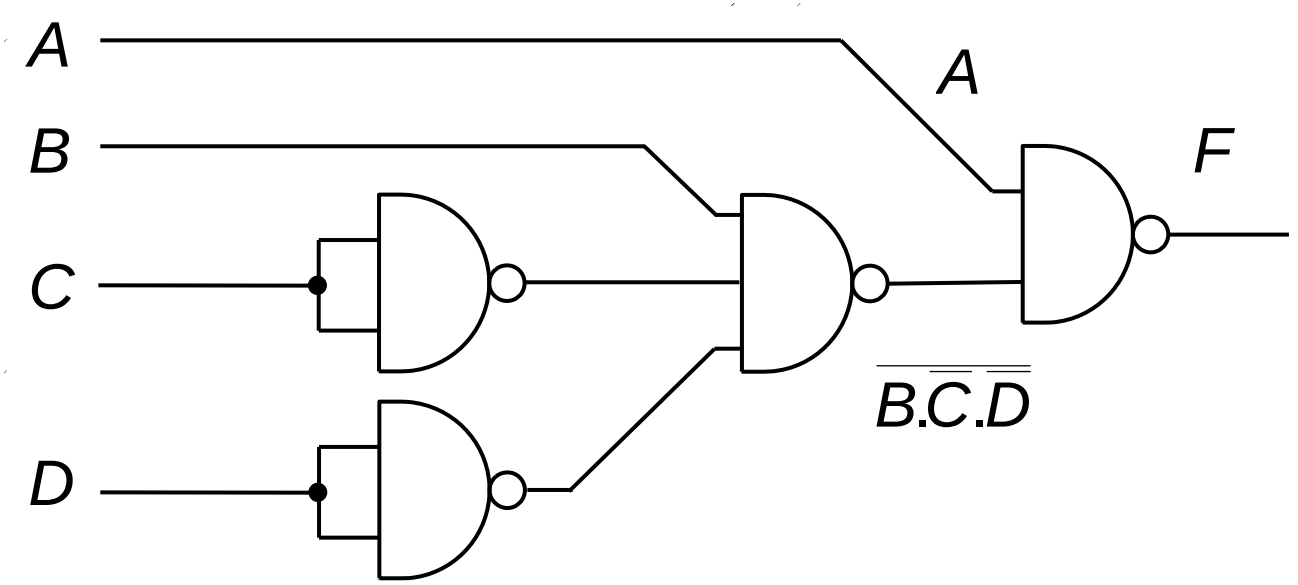
Example:

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

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

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

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



Design Example

University of Reading

The days of the week are represented by a 3-bit binary code:

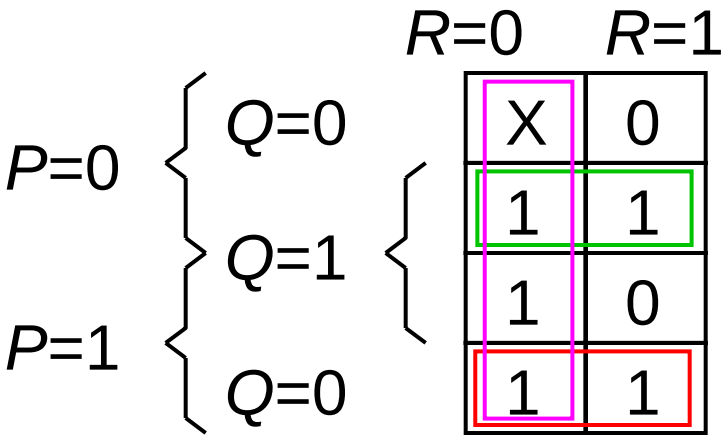
	P	*Q*	*R*
Sunday	0	0	1
Monday	0	1	0
Tuesday	0	1	1
Wednesday	1	0	0
Thursday	1	0	1
Friday	1	1	0
Saturday	1	1	1

The code 000 will never occur

Design a NAND system to give 1 for the work-days, and 0 for the rest-days

Design Example

The first stage is to construct a K-map:



Unused	Sunday
Monday	Tuesday
Friday	Saturday
Wednesday	Thursday

Note the use of X for the unused code: 000

The simplest sum-of-products form is:

$$F = \overline{R} + \overline{P}.Q + P.\overline{Q}$$

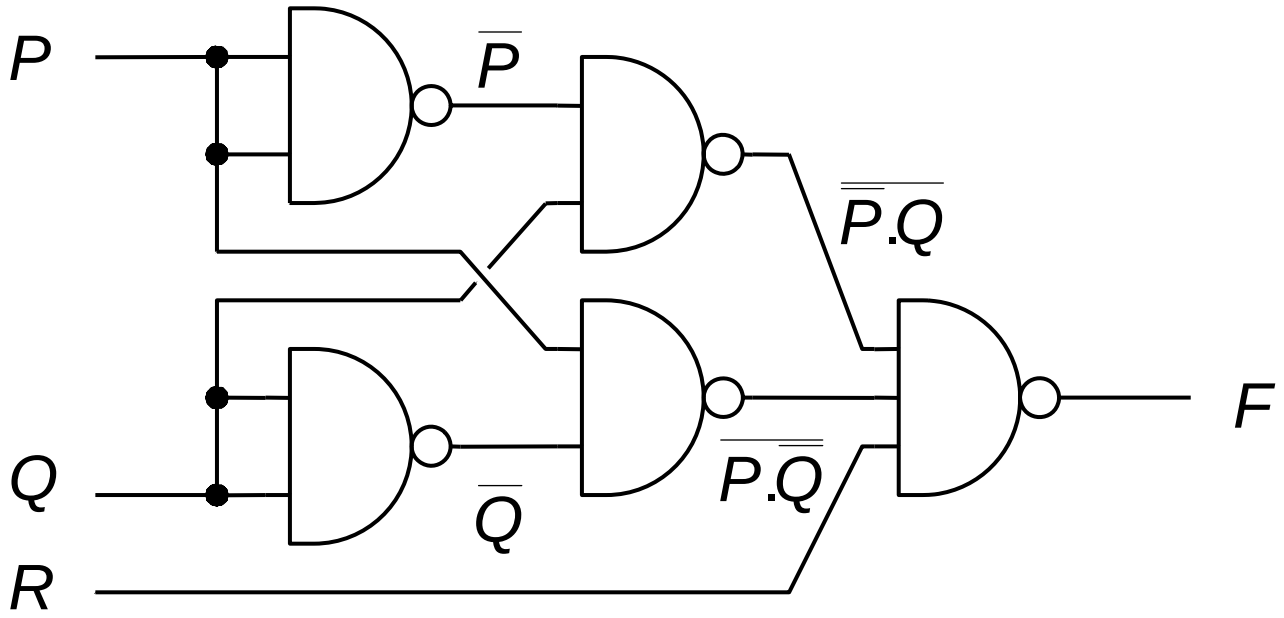
Design Example

University of Reading

$$F = \overline{\overline{\overline{R} + \overline{P}.Q + P.\overline{Q}}}$$

$$= \overline{\overline{\overline{R}} . \overline{(\overline{P}.Q)} . \overline{(P.\overline{Q})}}$$

$$= \overline{R . \overline{(\overline{P}.Q)} . \overline{(P.\overline{Q})}}$$



Design Example

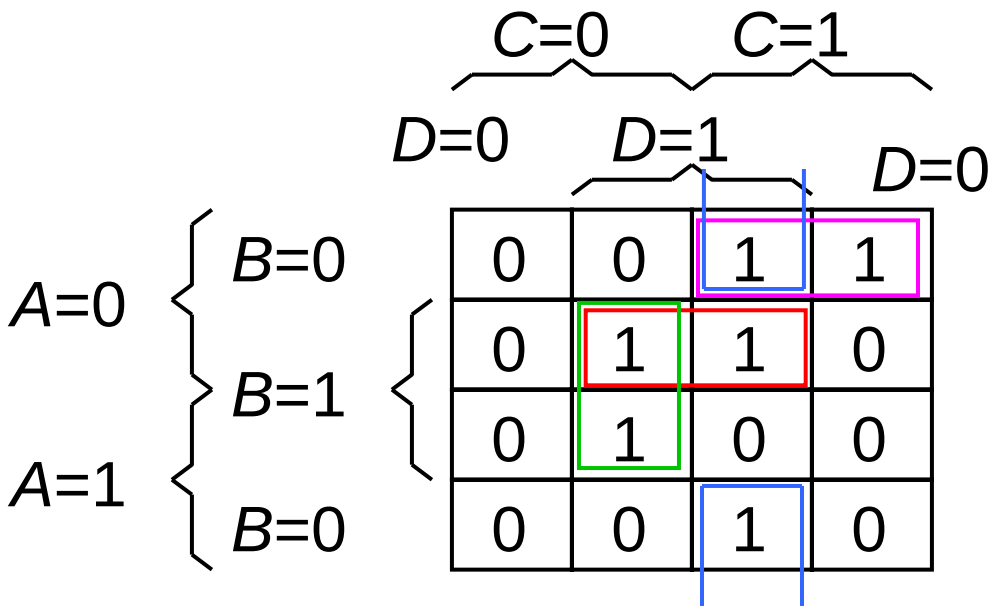
The numbers 0 to 15 are represented by the 4-bit natural binary code *ABCD* (*A* is the most-significant bit, *D* the least-significant

A function *F* is required to have a value of 1 for prime numbers, and to have a value of 0 for composite (non-prime) numbers.

For the purpose of this design example the numbers 0 and 1 will be taken to be composite.

Design a combinational logic system consisting of NAND gates to implement the function *F* of the variables *A*, *B, C* and *D*.

Design Example



0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

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

The 1s on the K-map can be grouped in a different way:

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

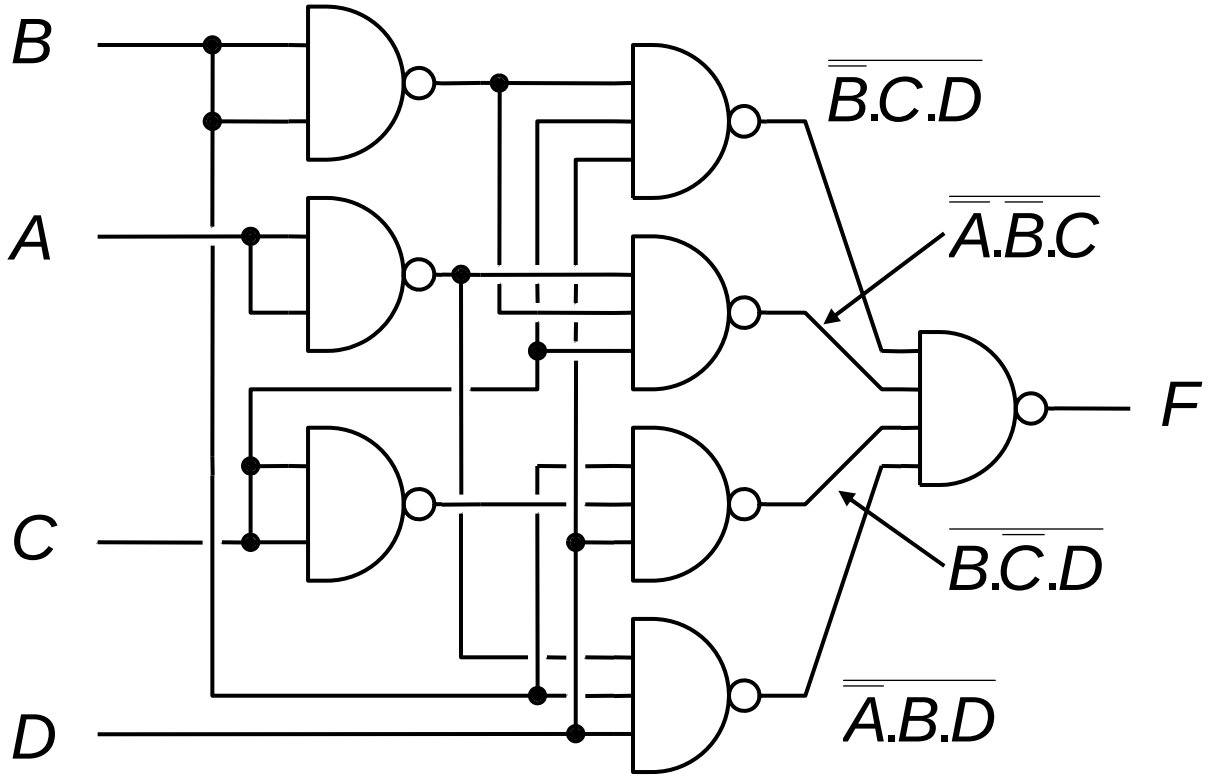
Design Example

University of Reading

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

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

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



Implementation in NOR

University of Reading

1. Obtain the function in simplest *product-of-sums* form:

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

2. Invert the function twice (thus leaving it unchanged):

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

3. Use DeMorgan's theorem on lower inversion:

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

4. The function can now be implemented using only NOR gates

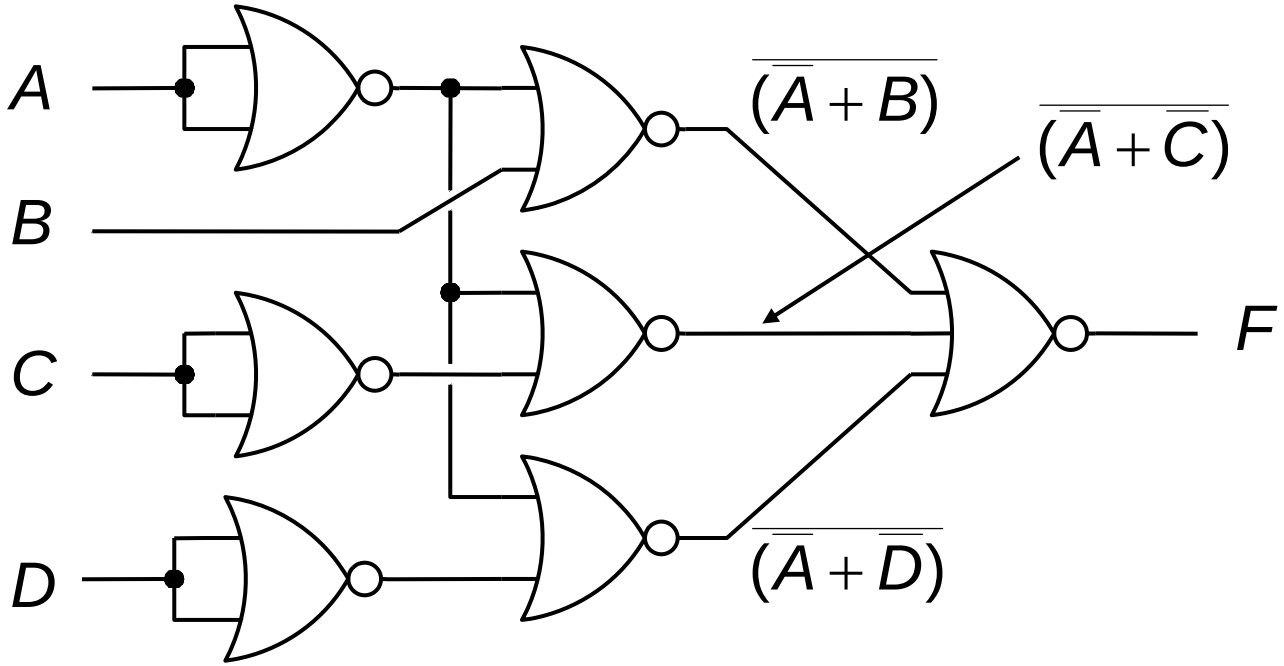
Implementation in NOR

Example:

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

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

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



Design Example

University of Reading

The days of the week are represented by a 3-bit binary code:

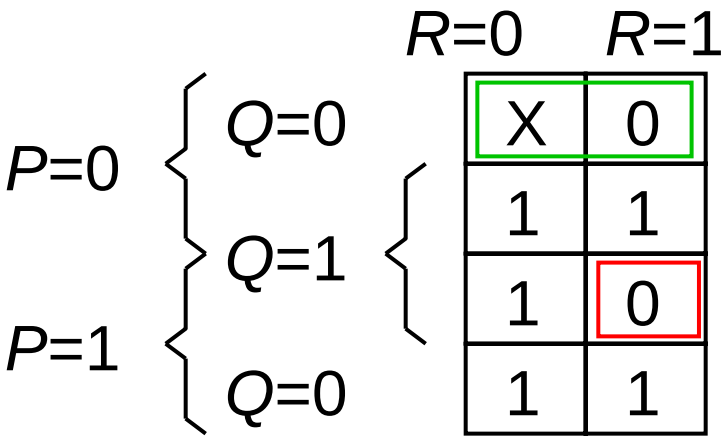
	P	*Q*	*R*
Sunday	0	0	1
Monday	0	1	0
Tuesday	0	1	1
Wednesday	1	0	0
Thursday	1	0	1
Friday	1	1	0
Saturday	1	1	1

The code 000 will never occur

Design a NOR system to give 1 for the work-days, and 0 for the rest-days

Design Example

The first stage is to construct a K-map:



Unused	Sunday
Monday	Tuesday
Friday	Saturday
Wednesday	Thursday

The simplest *sum-of-products* form for $\overline{F}$ is:

$$\overline{F} = \overline{P}.\overline{Q} + P.Q.R$$

The simplest *product-of-sums* form for F is:

$$F = (P + Q).(\overline{P} + \overline{Q} + \overline{R})$$

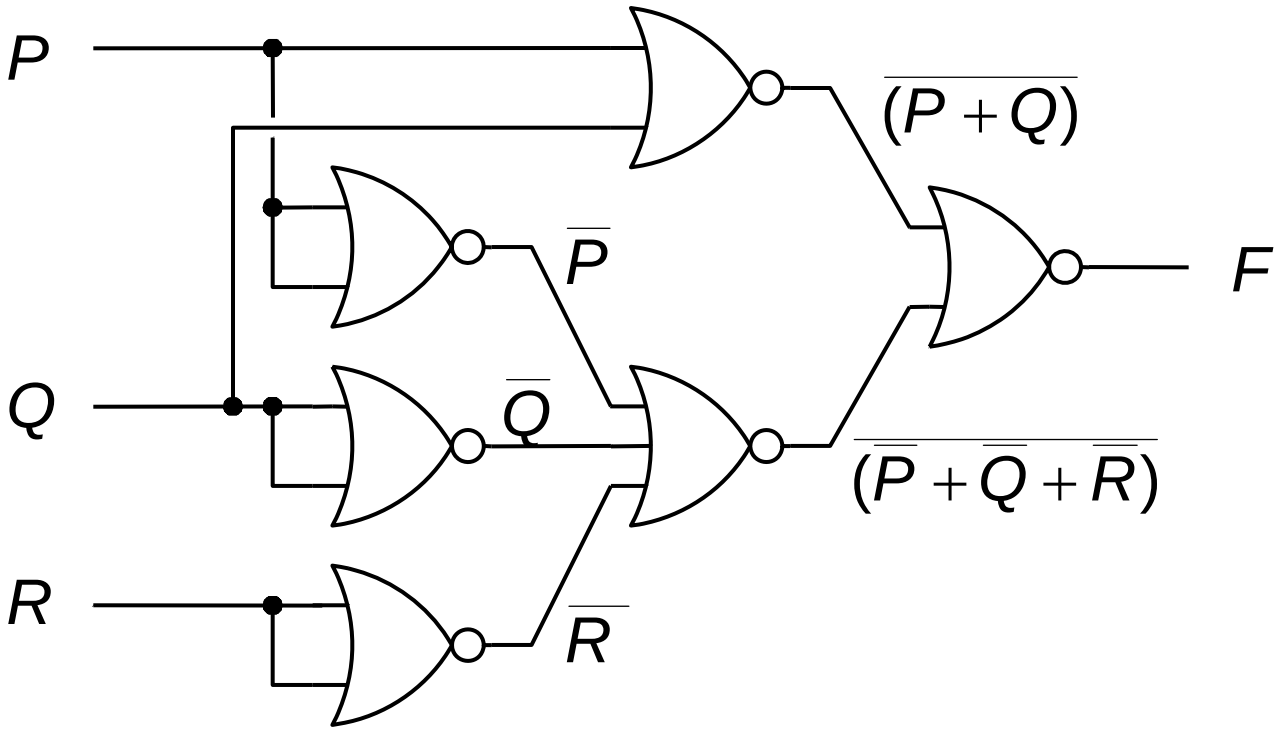
Design Example

University of Reading

$$F = (P + Q).(\overline{P} + \overline{Q} + \overline{R})$$

$$= \overline{\overline{(P + Q).(\overline{P} + \overline{Q} + \overline{R})}}$$

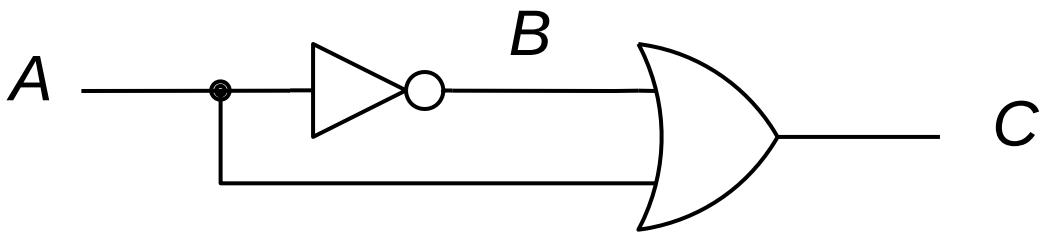
$$= \overline{\overline{(P + Q)} + \overline{(\overline{P} + \overline{Q} + \overline{R})}}$$



Hazards in Combinational Logic Systems

A hazard is a transient change that occurs in a logic system following a change in an input

Hazards are the result of gate delays



The output *C* is given by:

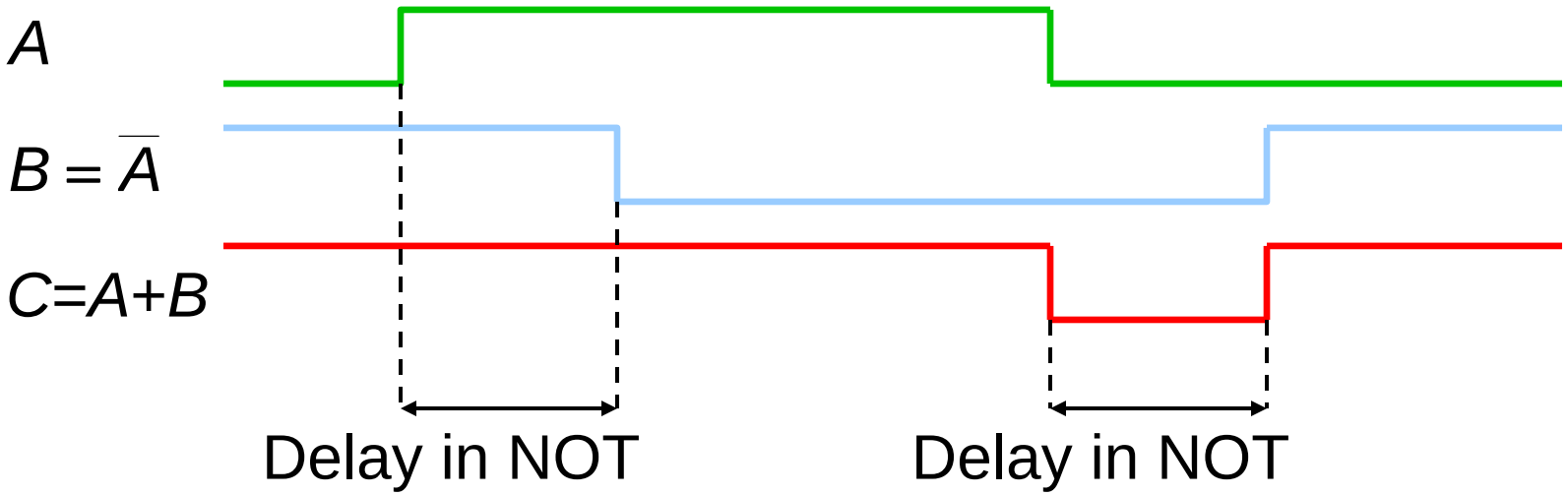
$$C = A + B = A + \overline{A} = 1$$

Thus the output should stay at logic 1, and be independent of the input.

Hazards in Combinational Logic Systems

University of Reading

In fact an output pulse occurs following the 1→0 input transition:



Hazard exists for a time equal to the gate delay (typically a few ns)

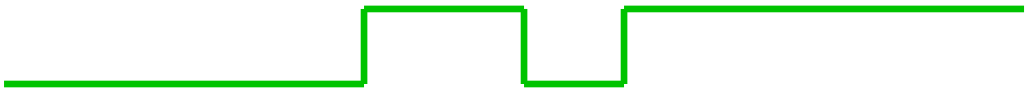
Hazards in Combinational Logic Systems

University of Reading

There are two types of hazard: static hazards and dynamic hazards

Static hazard:

Dynamic hazard:



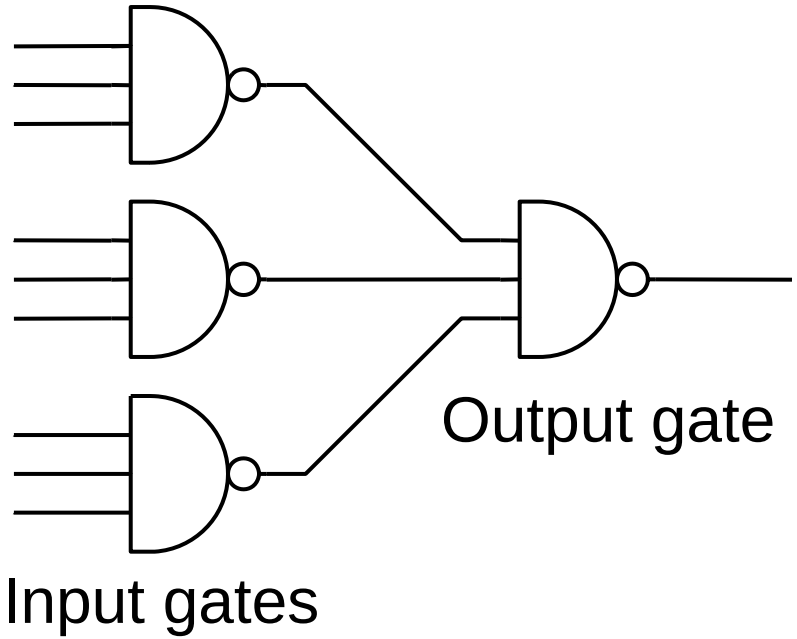
Whether hazards are significant depends on the application

Hazards are always a problem if they occur in logic providing the input to a system with memory

Hazards in NAND Systems

University of Reading

2-level NAND system:



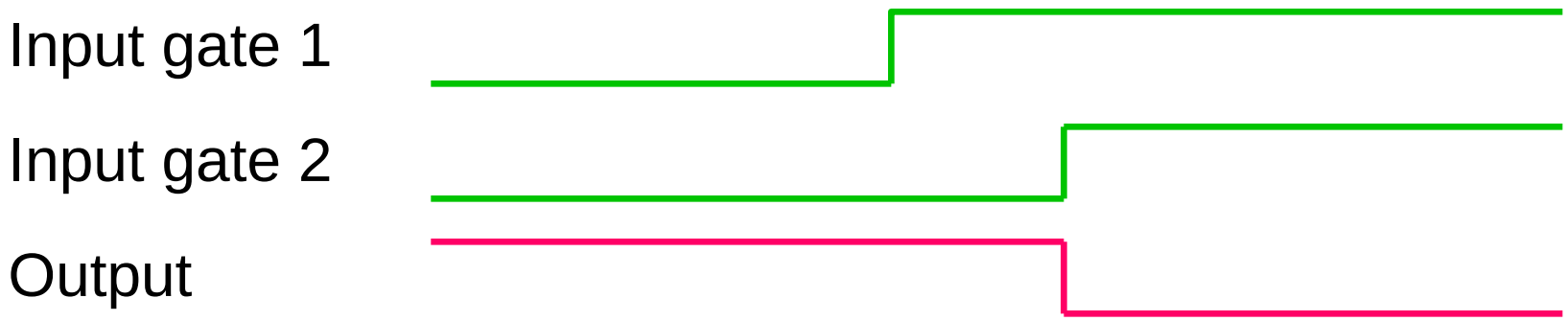
Suppose that an input Q changes: no input gate can have both Q and $\overline{Q}$ for inputs because:

$$\overline{Q.\overline{Q}.\ldots} = \overline{0} = 1$$

Hazards in NAND Systems

University of Reading

Case 1: (potential dynamic hazard)

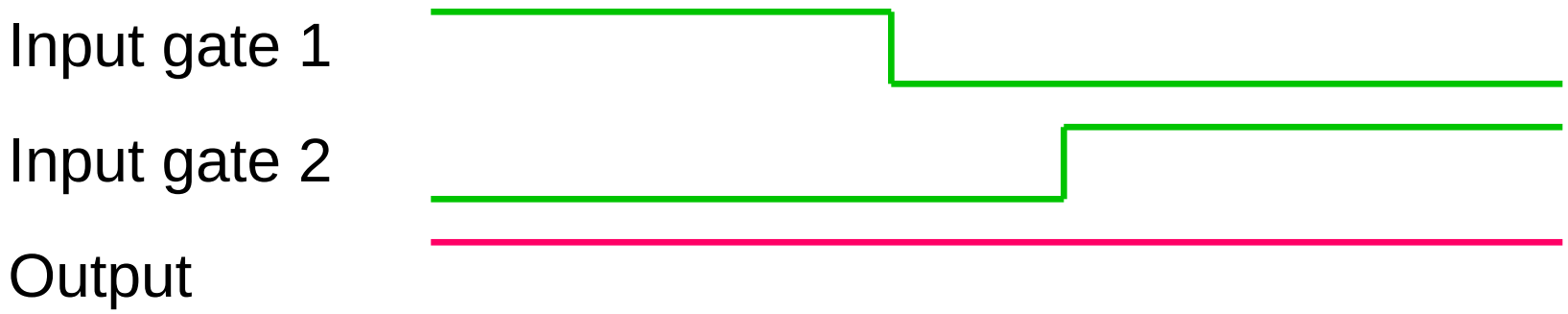


No Hazard

Hazards in NAND Systems

University of Reading

Case 2: (potential static hazard)

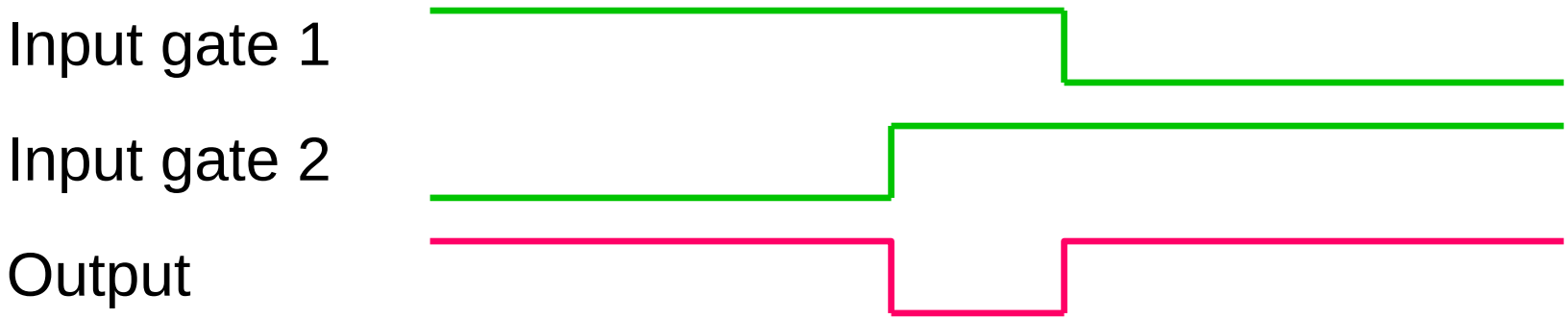


No Hazard

Hazards in NAND Systems

University of Reading

Case 3: (potential static hazard)



Dynamic hazards do not occur in 2-level NAND systems

Static hazards can occur if one input gate changes from 0→1 whilst another input gate changes from 1→0

Properties of K-maps

University of Reading

Moving one square horizontally or vertically corresponds to changing a single input variable

Each input gate corresponds to a group of 1s on the K-map: moving in or out of a group causes the corresponding input gate to change state

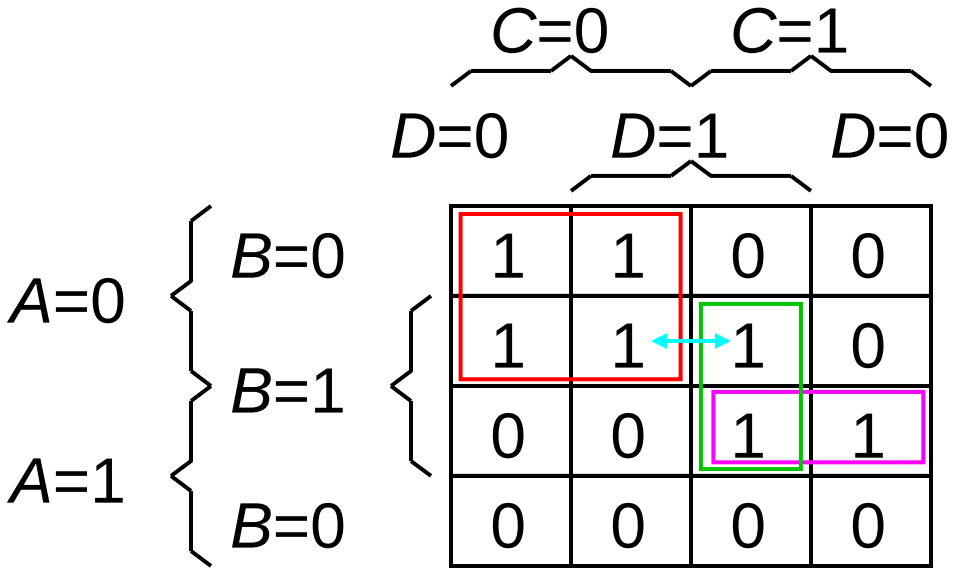
If 2 groups of 1s on the K-map are adjacent and non-overlapping (horizontally or vertically) then changing a single input variable can move out of one group and into the other

This is the condition for a static hazard

Example of a Static Hazard

University of Reading

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



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

A static hazard may occur if the inputs change from A=0, B=1, C=0, D=1 to A=0, B=1, C=1, D=1

Example of a Static Hazard

University of Reading

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

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

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

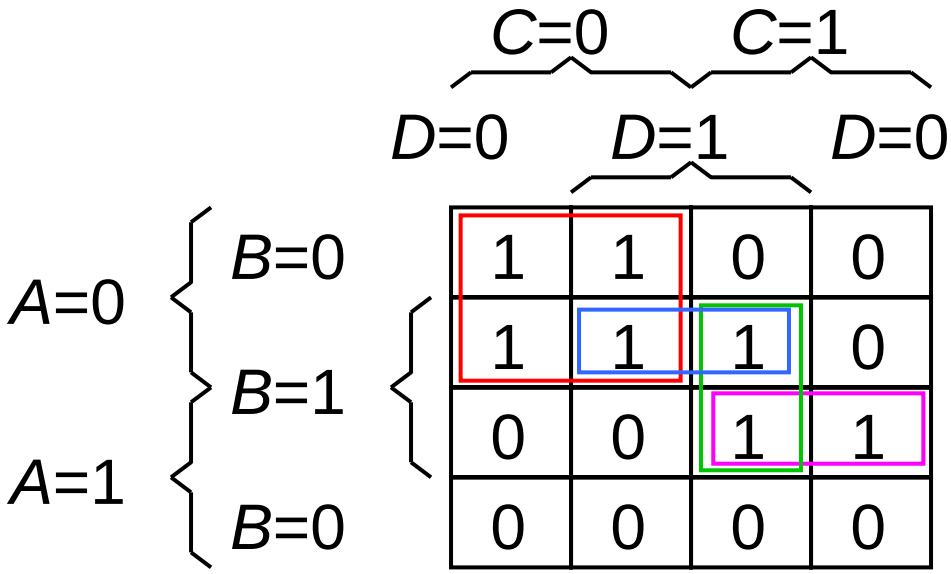
$$= \overline{Q1.Q2.Q3}$$

	$ABCD$	$ABCD$
	0101	0111
$Q1 = \overline{(\overline{A}.\overline{C})}$	0	1
$Q2 = \overline{(B.C.D)}$	1	0
$Q3 = \overline{(A.B.C)}$	1	1

This is condition for a static hazard

Eliminating Static Hazards

Static hazards are eliminated by including extra groups which overlap the offending transitions:



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

The input gate corresponding to the new group remains at 0 throughout the transition

Eliminating Static Hazards

University of Reading

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

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

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

$$= \overline{Q1.Q2.Q3.Q4}$$

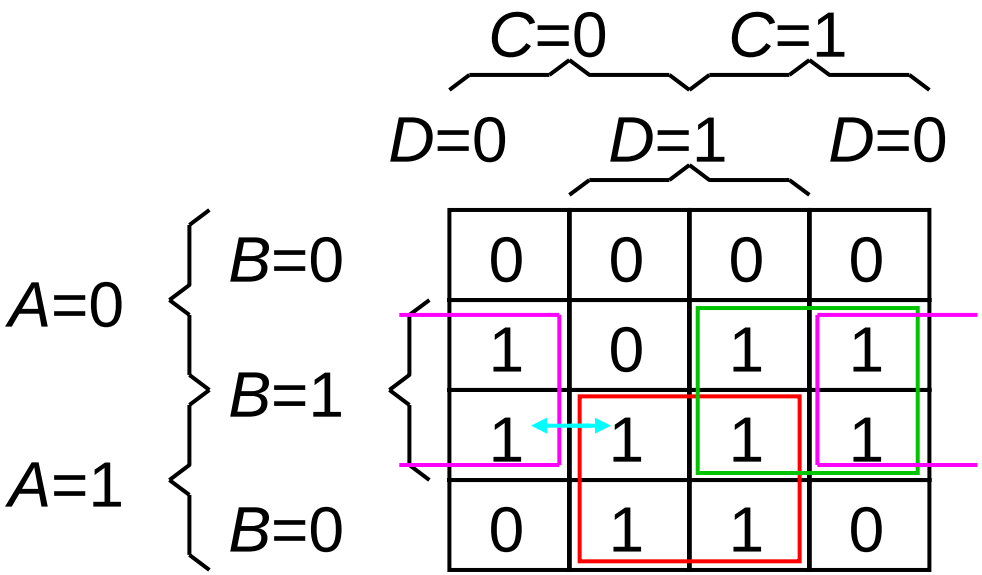
	$ABCD$	$ABCD$
	0101	0111
$Q1 = \overline{(\overline{A}.\overline{C})}$	0	1
$Q2 = \overline{(B.C.D)}$	1	0
$Q3 = \overline{(A.B.C)}$	1	1
$Q4 = \overline{(\overline{A}.B.D)}$	0	0

Static hazard has been eliminated

Design Example

University of Reading

Implement the function: $F = A.D + B.C + B.\overline{D}$
in hazard-free form using NAND gates

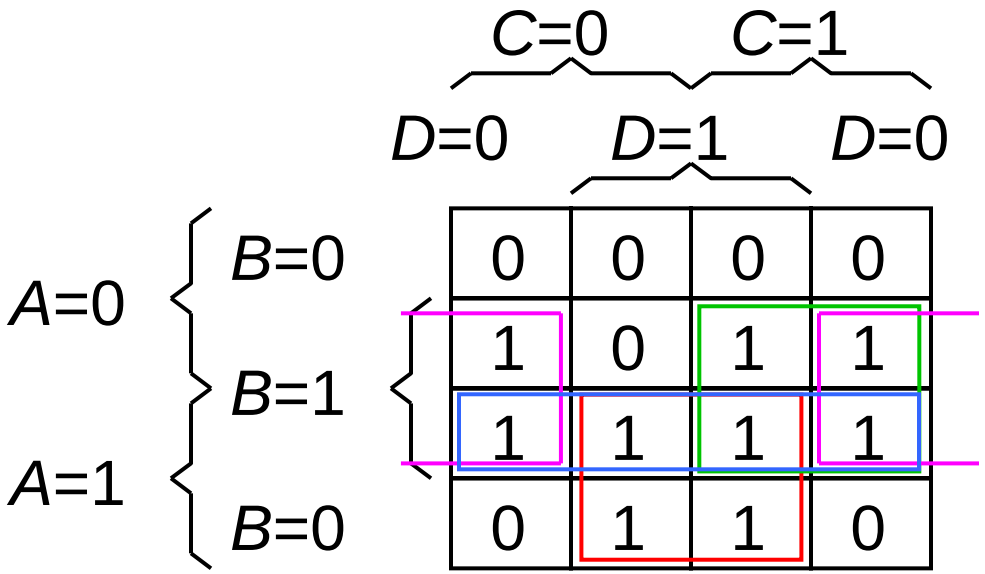


Simplest sum-of-products form:

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

Design Example

Include an extra group to overlap the border between the adjacent groups:



Hazard-free sum-of-products form:

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

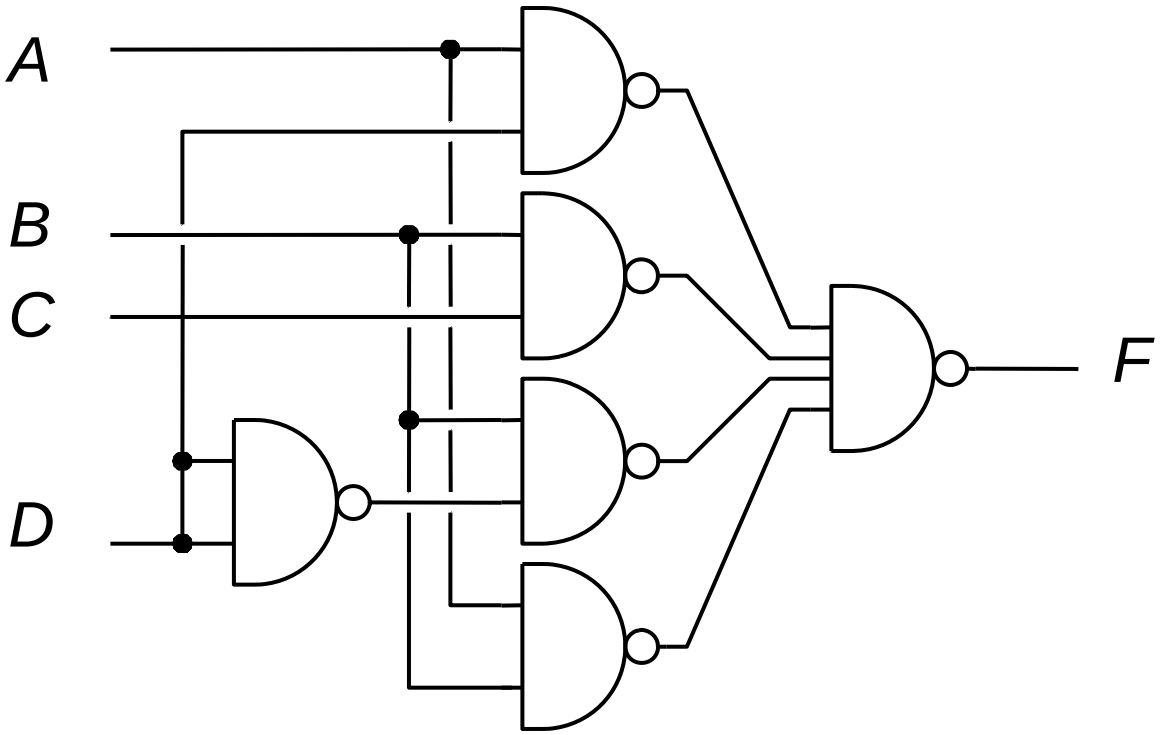
Design Example

University of Reading

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

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

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



Hazards in NOR Systems

University of Reading

Dynamic hazards do not occur in 2-level NOR systems

Static hazards can occur if one input gate changes from 0→1 whilst another input gate changes from 1→0

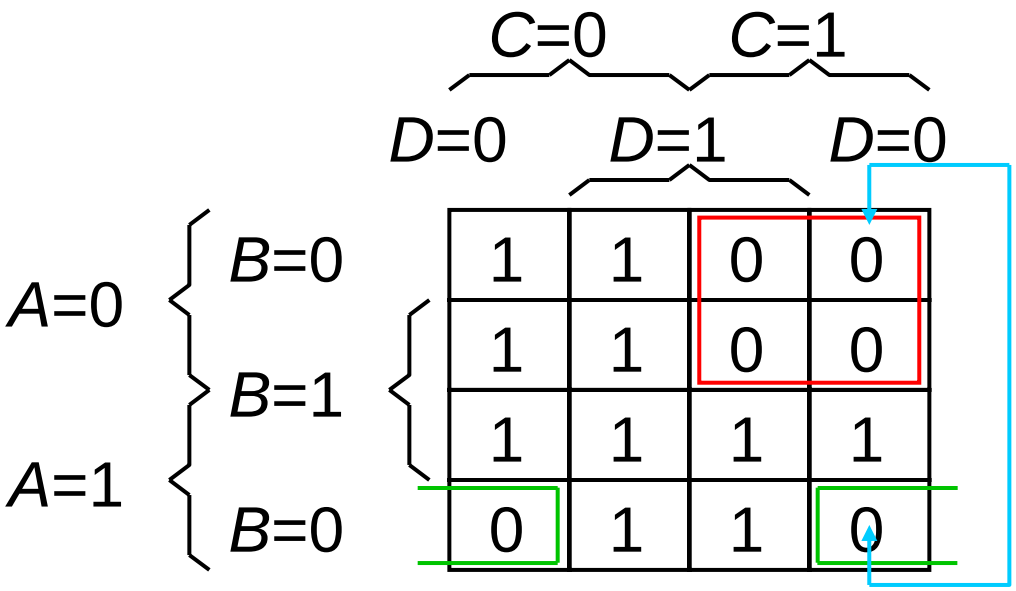
If 2 groups of 0s on the K-map are adjacent and non-overlapping (horizontally or vertically) then changing a single input variable can move out of one group and into the other

This is the condition for a static hazard

Example of a Static Hazard

University of Reading

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



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

A static hazard may occur if the inputs change from A=1, B=0, C=1, D=0 to A=0, B=0, C=1, D=0

Example of a Static Hazard

University of Reading

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

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

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

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

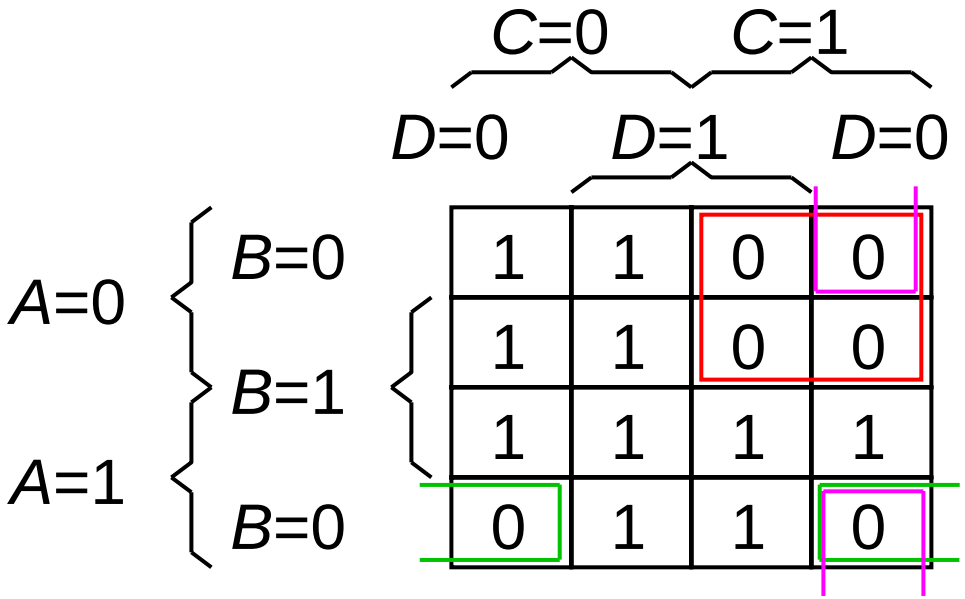
$$= \overline{Q1 + Q2}$$

	$ABCD$	$ABCD$
	1010	0010
$Q1 = \overline{(A + \overline{C})}$	0	1
$Q2 = \overline{(\overline{A} + B + D)}$	1	0

This is condition for a static hazard

Eliminating Static Hazards

Static hazards are eliminated by including extra groups which overlap the offending transitions:



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

The input gate corresponding to the new group remains at 1 throughout the transition

Eliminating Static Hazards

University of Reading

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

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

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

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

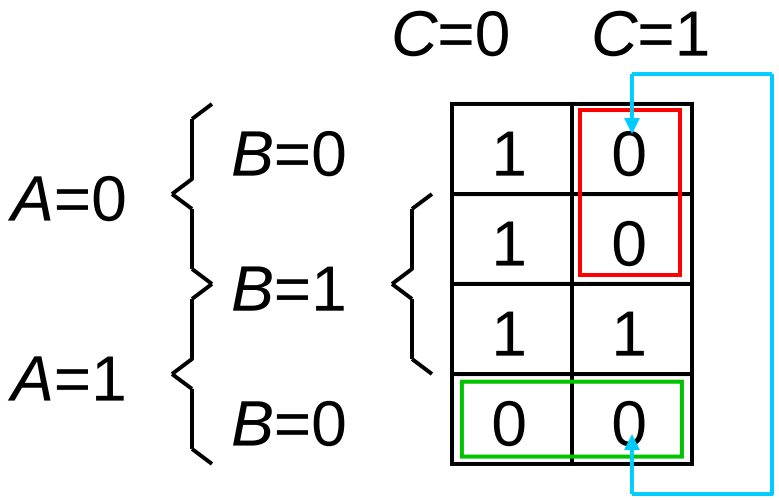
$$= \overline{Q1 + Q2 + Q3}$$

	$ABCD$	$ABCD$
	1010	0010
$Q1 = \overline{(A + \overline{C})}$	0	1
$Q2 = \overline{(\overline{A} + B + D)}$	1	0
$Q3 = \overline{(B + \overline{C} + D)}$	1	1

Static hazard has been eliminated

Design Example

Implement the function: $F = \overline{A}.\overline{C} + B.\overline{C} + A.B.C$
in hazard-free form using NOR gates



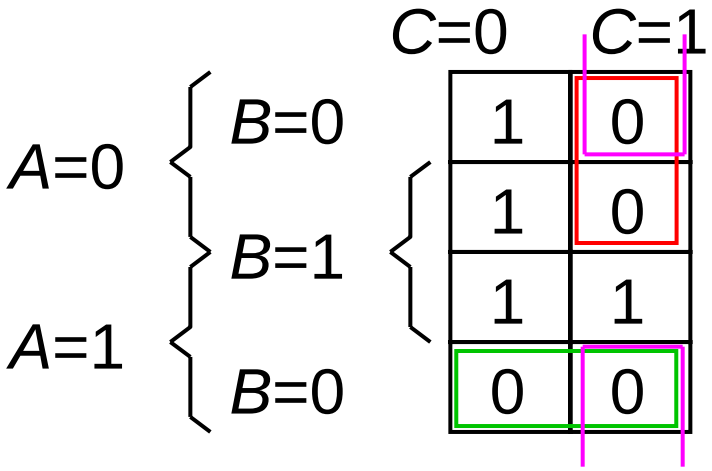
Simplest sum-of-products form for $\overline{F}$:

$$\overline{F} = \overline{A}.C + A.\overline{B}$$

Design Example

University of Reading

Include an extra group to overlap the border between the adjacent groups:



Hazard-free sum-of-products form:

$$\overline{F} = \overline{A}.C + A.\overline{B} + \overline{B}.C$$

Design Example

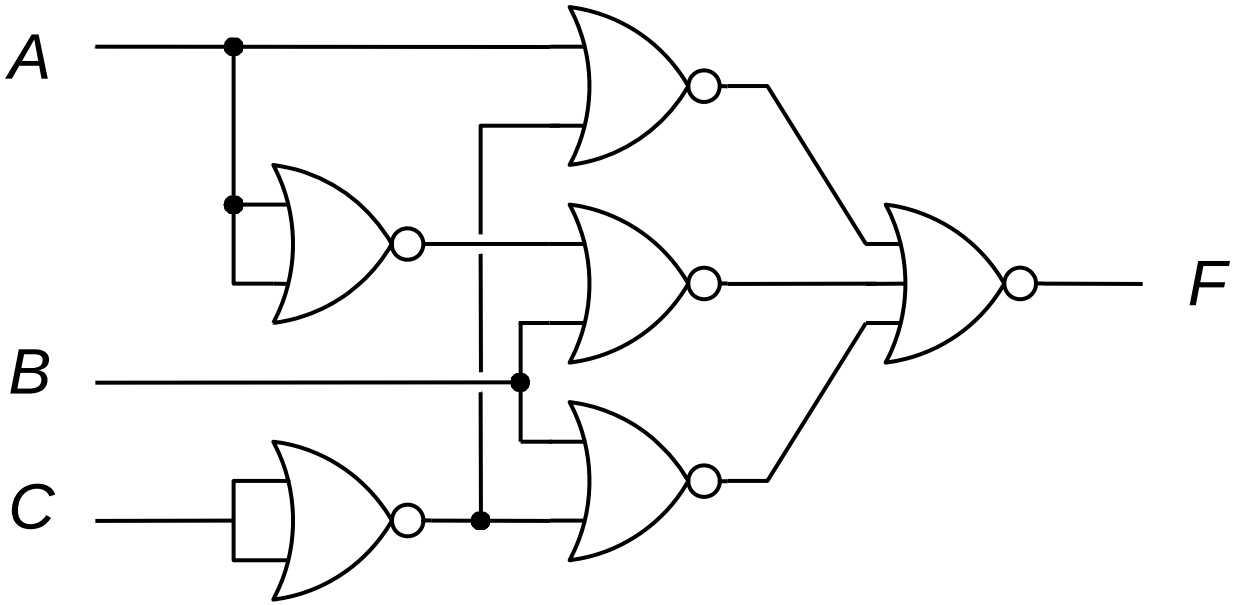
University of Reading

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

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

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

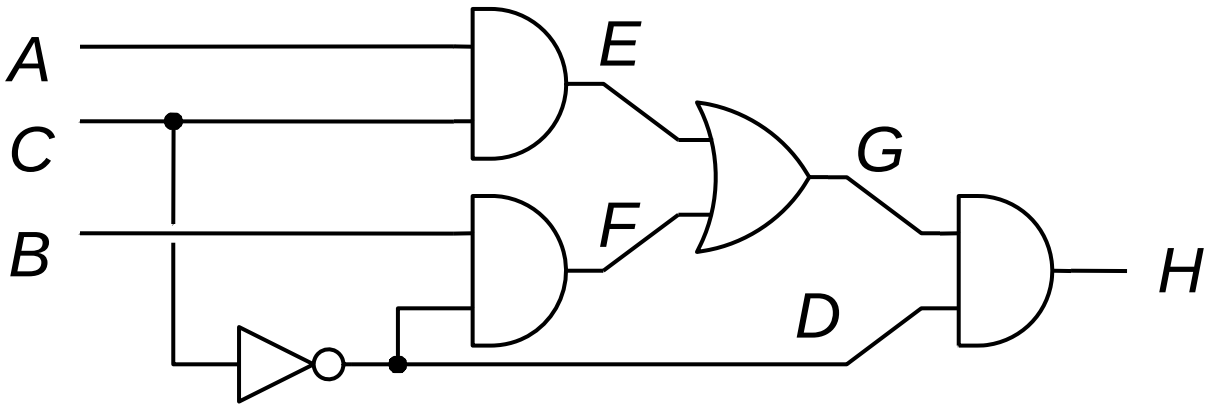
$$= \overline{\overline{(A + \overline{C})} + \overline{(\overline{A} + B)} + \overline{(B + \overline{C})}}$$



Dynamic Hazards

University of Reading

There must be at least three paths between the input and output to give the minimum of three transitions that constitute a dynamic hazard:



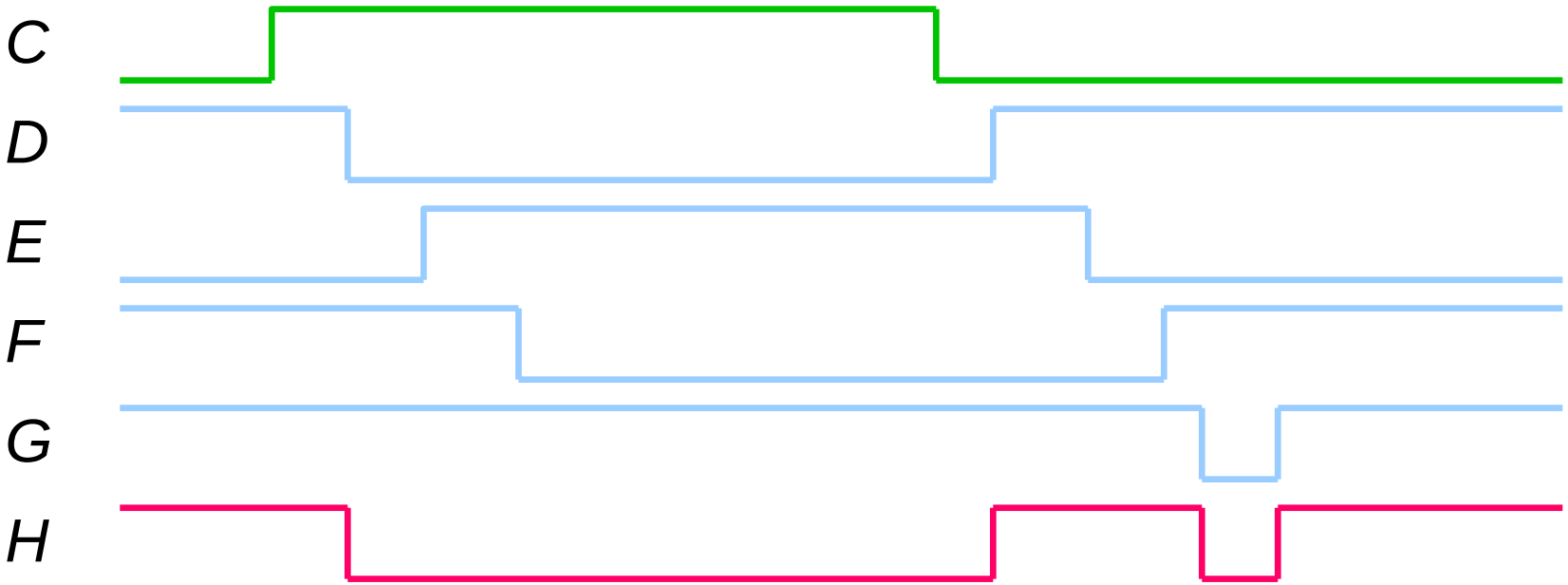
$$D = \overline{C}$$
$$E = A.C$$
$$F = B.D = B.\overline{C}$$
$$G = E + F = A.C + B.\overline{C}$$
$$H = D.G = \overline{C}.(A.C + B.\overline{C}) = B.\overline{C}$$

Dynamic Hazards

University of Reading

Let A=1
B=1

$$D = \overline{C}$$
$$E = A.C$$
$$F = B.D = B.\overline{C}$$
$$G = E + F = A.C + B.\overline{C}$$
$$H = D.G = \overline{C}.(A.C + B.\overline{C}) = B.\overline{C}$$



Exclusive-OR Gates

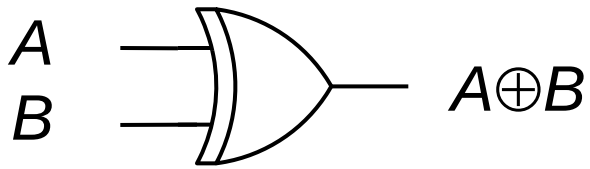
University of Reading

A XOR B or $A \oplus B$

A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	0

This function is similar to the normal (inclusive) OR function except for the case A=1, B=1

Electrical symbol:



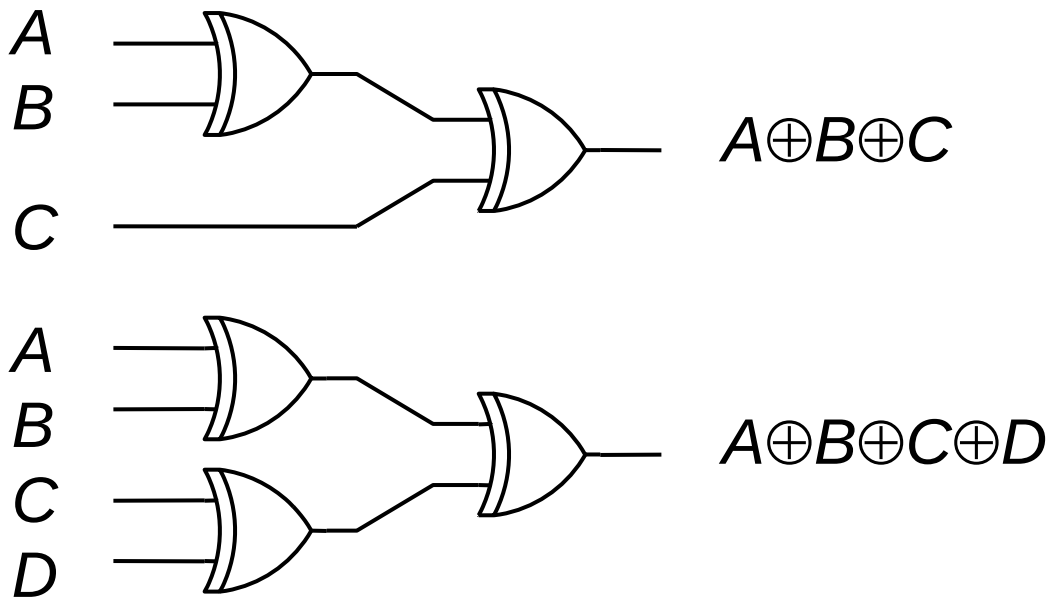
Identities:

$$A \oplus B \equiv B \oplus A$$

$$(A \oplus B) \oplus C \equiv A \oplus (B \oplus C)$$

Exclusive-OR Gates

Exclusive-OR functions of more than 2 variables can be generated by using several 2-input gates:



Operating between several variables the exclusive-OR function is 1 if an odd number of the input variables are 1

Exclusive-OR Gates

University of Reading

	B=0	B=1
A=0	0	1
A=1	1	0

$A \oplus B$

		C=0	C=1
A=0	B=0	0	1
	B=1	1	0
A=1	B=1	0	1
	B=0	1	0

$A \oplus B \oplus C$

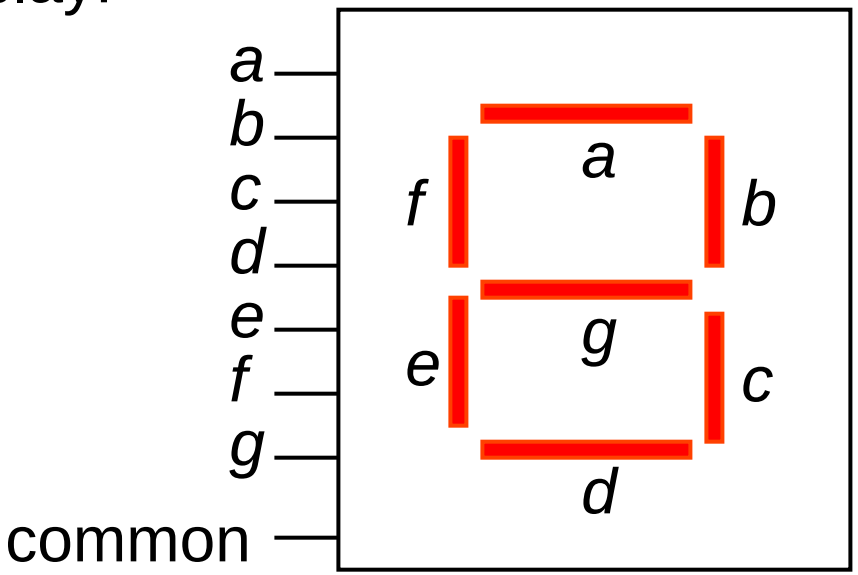
			C=0		C=1	
			D=0	D=1	D=0	
A=0	B=0		0	1	0	1
	B=1		1	0	1	0
A=1	B=1		0	1	0	1
	B=0		1	0	1	0

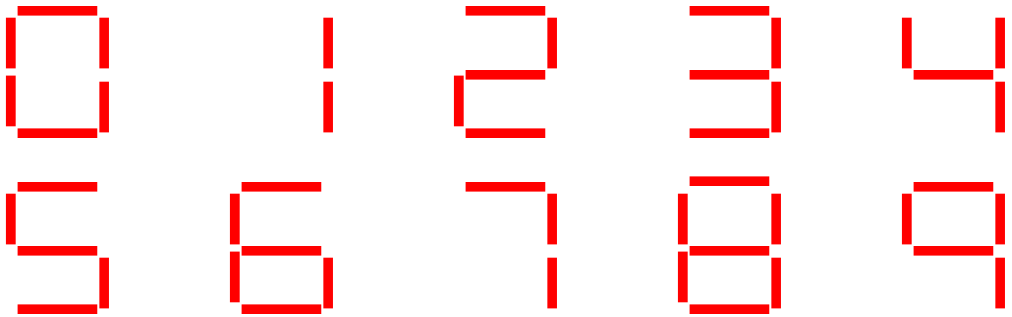
$A \oplus B \oplus C \oplus D$

7-Segment Decoder

University of Reading

7-segment display:



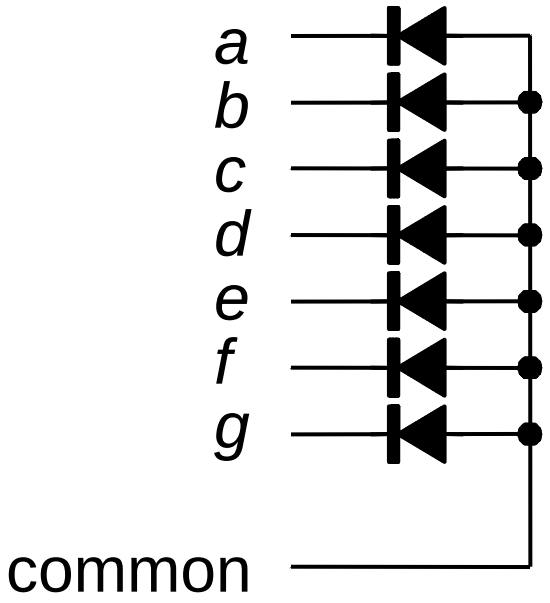


7-Segment Decoder

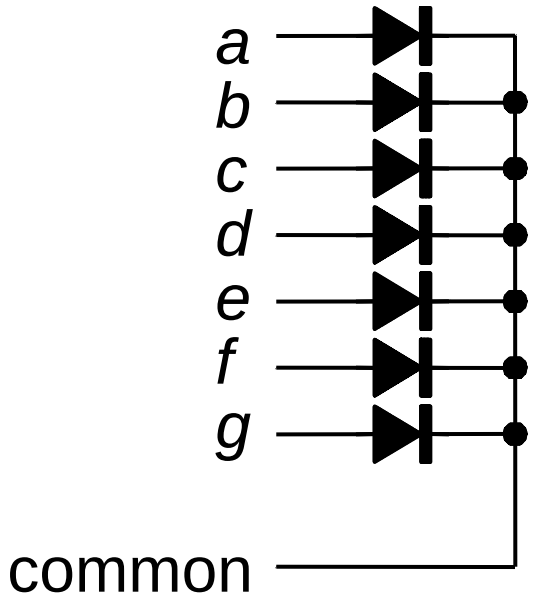
University of Reading

To reduce the number of connections the cathodes or anodes of the LEDs are connected in common:

Common anode



Common cathode



7-Segment Decoder

University of Reading

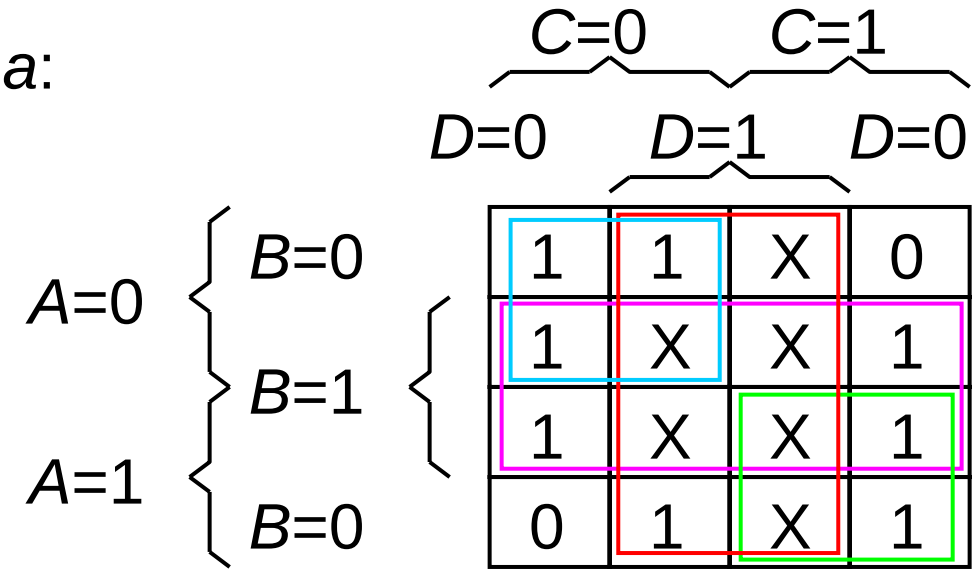
Value Binary 7-Segment Display

	D	*C*	*B*	*A*	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

7-Segment Decoder

Functions *a*, *b*, *c*, *d*, *e*, *f* of the binary code *A*, *B*, *C*, *D* can now be obtained using a K-map:

Function *a*:

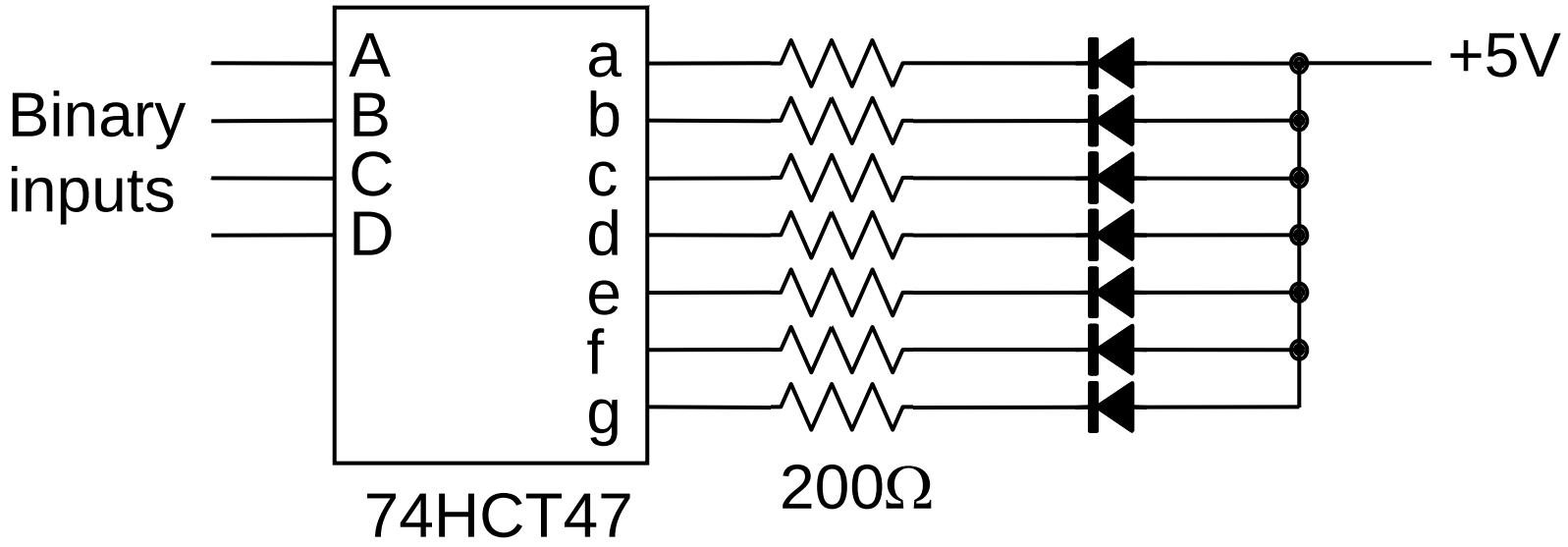


The simplest *sum-of-products* form for *a* is:

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

7-Segment Decoder

Medium-Scale Integration (MSI) devices are available which implement these Boolean functions and also provide current drive:



Devices are also available to drive Liquid-Crystal Displays (LCD) and multi-digit displays

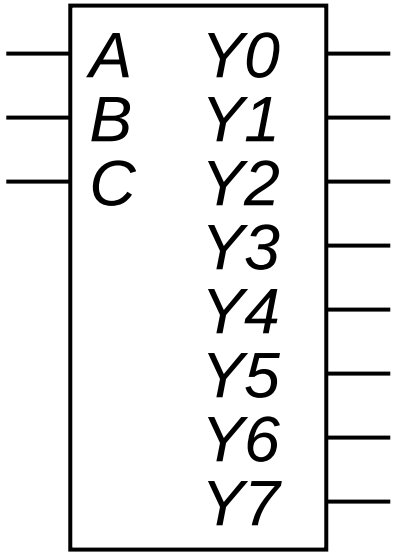
Decoders

A decoder converts the information on *n* input lines to 2^n output lines

For any input combination one, and only one, output line is set to logical 1

For example, a
3-to-8 line decoder

74HCT138



Decoders

University of Reading

Truth table for a 3-to-8 line decoder:

C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

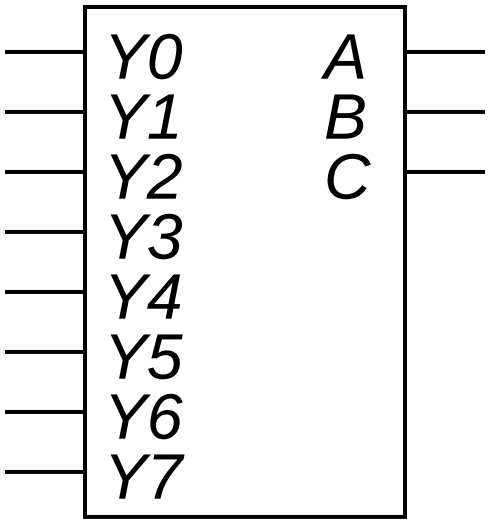
Encoders

Encoders have 2^n input lines, n output lines, and perform the opposite operation to decoders

If one of the input lines is at logical 1 then the output lines indicate the code for this input

For example, an
8-to-3 line encoder

74HCT148



Priority Encoders

University of Reading

Truth table for an 8-to-3 line priority encoder:

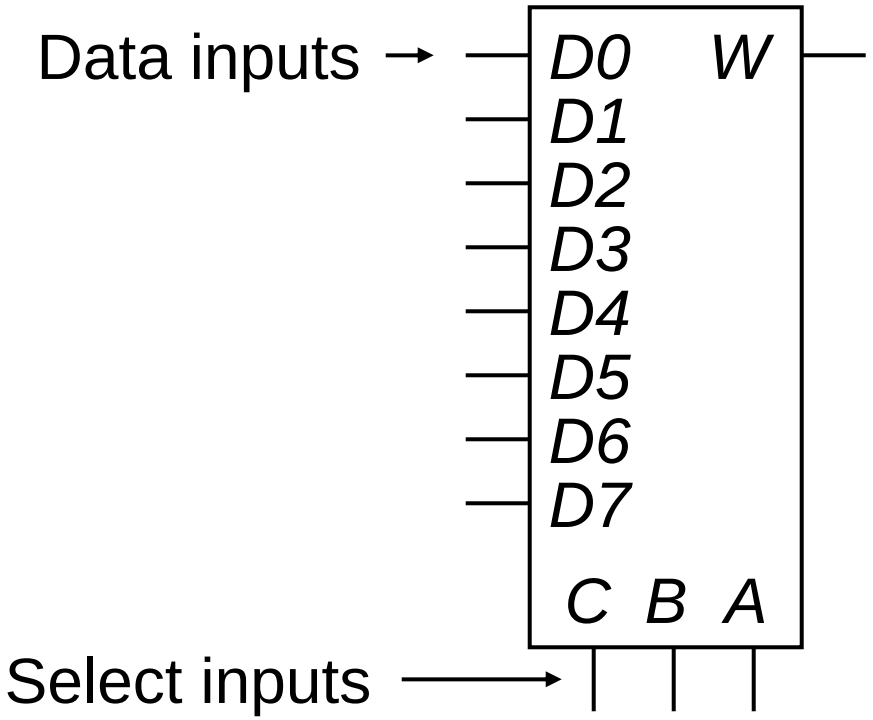
Y0	*Y1*	*Y2*	*Y3*	*Y4*	*Y5*	*Y6*	*Y7*	*C*	*B*	*A*
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
X	1	0	0	0	0	0	0	0	0	1
X	X	1	0	0	0	0	0	0	1	0
X	X	X	1	0	0	0	0	0	1	1
X	X	X	X	1	0	0	0	1	0	0
X	X	X	X	X	1	0	0	1	0	1
X	X	X	X	X	X	1	0	1	1	0
X	X	X	X	X	X	X	1	1	1	1

Multiplexers

A multiplexer (MUX) is a combinational logic device that selects binary information from one of several input lines

For example, an 8-to-1 line MUX

74HCT152



Multiplexers

University of Reading

Truth table for a 1-to-8 line multiplexer:

C	*B*	*A*	*W*
0	0	0	*D0*
0	0	1	*D1*
0	1	0	*D2*
0	1	1	*D3*
1	0	0	*D4*
1	0	1	*D5*
1	1	0	*D6*
1	1	1	*D7*

Programmable Logic

University of Reading

Programmable logic is increasingly being used instead of standard gates

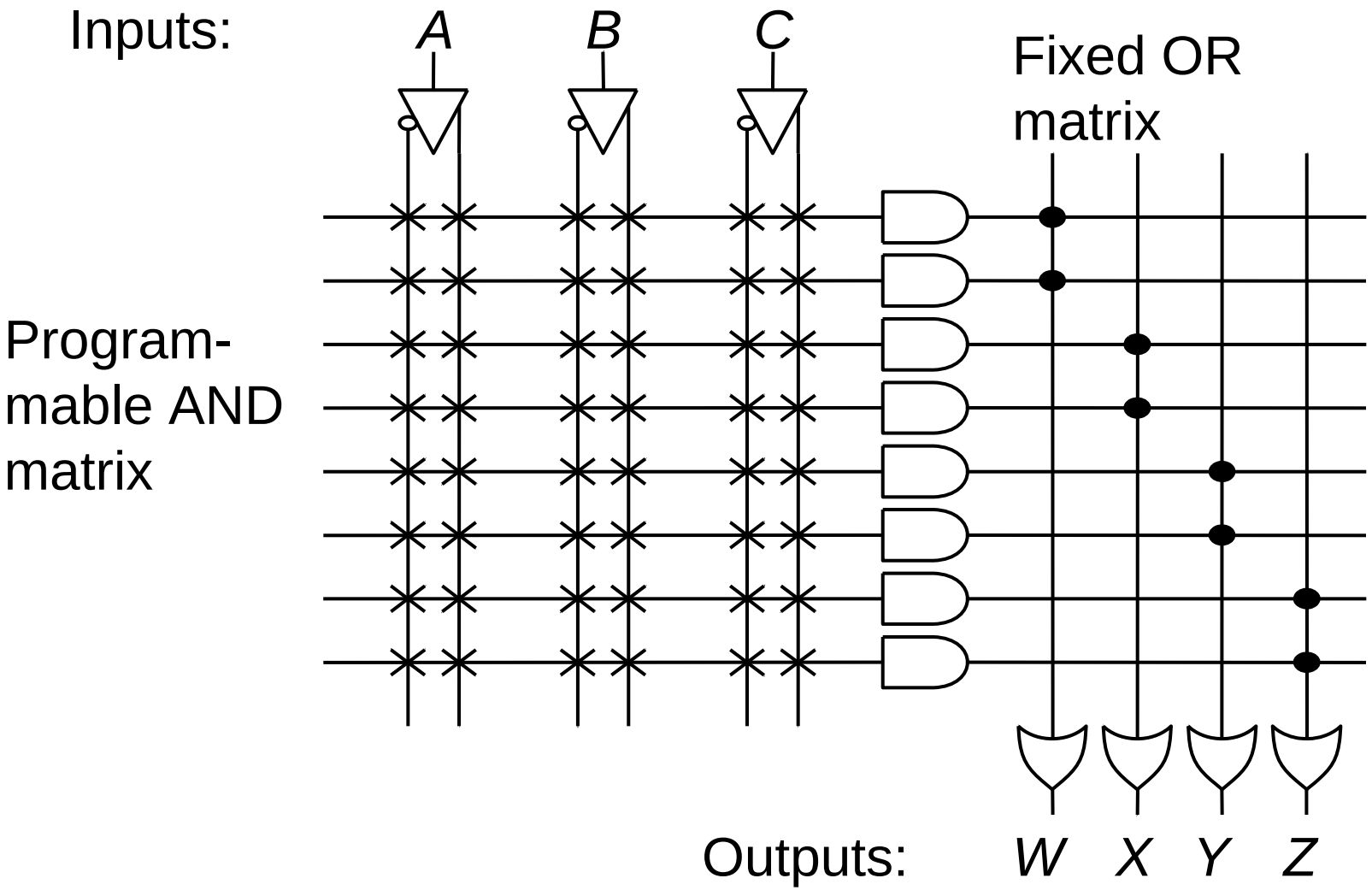
Programmable logic often leads to lower cost (both design costs and hardware costs)

Types of programmable logic include: PROMs, PALs, GALs and FPGAs

The function of programmable logic is determined by the state of “fuses” on the integrated circuit

These fuses are programmed in a special-purpose programmer connected to a PC

Programmable Array Logic



HDL Definition File

University of Reading

```
Name      EX1;
Partno    EX0000;
Date      6/10/98;
Rev       01;
Designer  J.B.Grimbleby;
Company   University of Reading;
Assembly  None;
Location  None;
Device    p16l8;

/* input pins */
PIN 1  = A;
PIN 2  = B;
PIN 3  = C;
PIN 4  = D;

/* output pins */
PIN 19 = F;

/* equations */
F = !A & (B # C # !B & !C) # A & B & !(C # D);
```

Boolean Operator Symbols:

! → NOT

& → AND

→ OR

HDL Documentation File

University of Reading

```
Expanded Product Terms:

!F =>
    A & !B
  # A & C
  # A & D


Symbol Table:

Pin Variable                    Pterms  Max    Min
Pol   Name    Ext Pin Type      Used   Pterms Level
--- -------- --- --- ---- ------ ------ -----

      A            1    V      -      -      -
      B            2    V      -      -      -
      C            3    V      -      -      -
      D            4    V      -      -      -
      F           19    V      3      7      1
```

JEDEC Output File

University of Reading

```
CUPL(WM)          4.9a  Serial# MW-20007187
Device            p16l8  Library DLIB-h-39-8
Created           Mon Oct 05 19:23:13 1998
Name              EX1
Partno            EX0000
Revision          01
Date              6/10/98
Designer          J.B.Grimbleby
Company           University of Reading
Assembly          None
Location          None
*QP20
*QF2048
*G0
*F0
*L00000 11111111111111111111111111111111
*L00032 10011111111111111111111111111111
*L00064 11010111111111111111111111111111
*L00096 11011111011111111111111111111111
*C0FD1
* 76D1
```

Sequential Logic Systems

University of Reading

Any logic system that has feedback can have memory

The output is not simply a function of the present value of the inputs, but also of previous states of the system

Digital circuits with memory are termed *sequential*

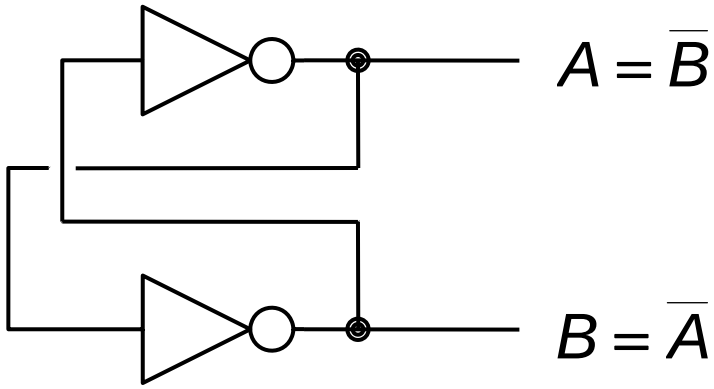
General sequential logic systems are formalised as Finite-State Machines (FSMs)

This course will only cover counters which are a subset of FSMs

RS Flip-Flops

Logic circuits having two stable states are called flip-flops

The simplest flip-flop consists of two cross-coupled NOT gates:



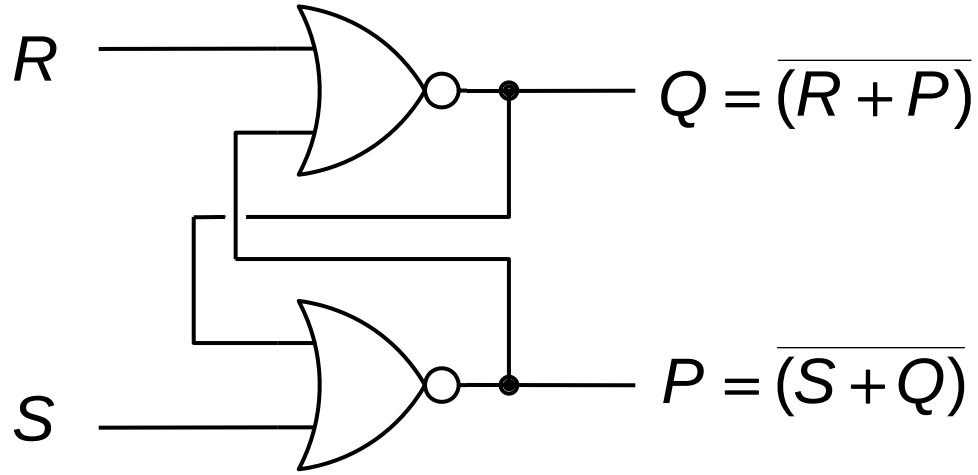
There are two stable states:

A=0, B=1
A=1, B=0

RS Flip-Flops

University of Reading

Replacing the inverters by NOR gates:



$$R=0, S=0: \quad Q=\overline{(0+P)}=\overline{P} \qquad P=\overline{(0+Q)}=\overline{Q}$$

$$R=1, S=0: \quad Q=\overline{(1+P)}=\overline{1}=0 \qquad P=\overline{(0+Q)}=\overline{0}=1$$

$$R=0, S=1: \quad P=\overline{(1+Q)}=\overline{1}=0 \qquad Q=\overline{(0+P)}=\overline{0}=1$$

RS Flip-Flops

The behaviour of the RS flip-flop can be summarised in the table:

S	*R*	*Q*
0	0	*Q*-
1	0	1
0	1	0
1	1	X

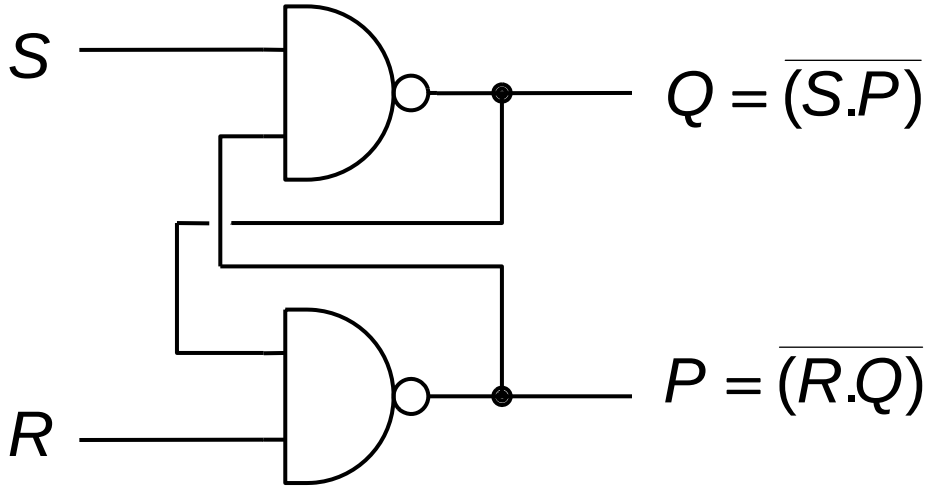
where *Q*- represents the previous value of *Q*, and X means that the result is undetermined

RS flip-flops can be used to store 1 bit of data

RS Flip-Flops

University of Reading

An alternative RS flip-flop using NAND gates:



$R = 1, S = 1$: $Q = \overline{(1.P)} = \overline{P}$ $P = \overline{(1.Q)} = \overline{Q}$

$R = 1, S = 0$: $Q = \overline{(0.P)} = \overline{0} = 1$ $P = \overline{(1.Q)} = \overline{1} = 0$

$R = 0, S = 1$: $P = \overline{(0.Q)} = \overline{0} = 1$ $Q = \overline{(1.P)} = \overline{1} = 0$

RS Flip-Flops

The behaviour of the NAND RS flip-flop can be summarised in the table:

S	*R*	*Q*
1	1	*Q*-
0	1	1
1	0	0
0	0	X

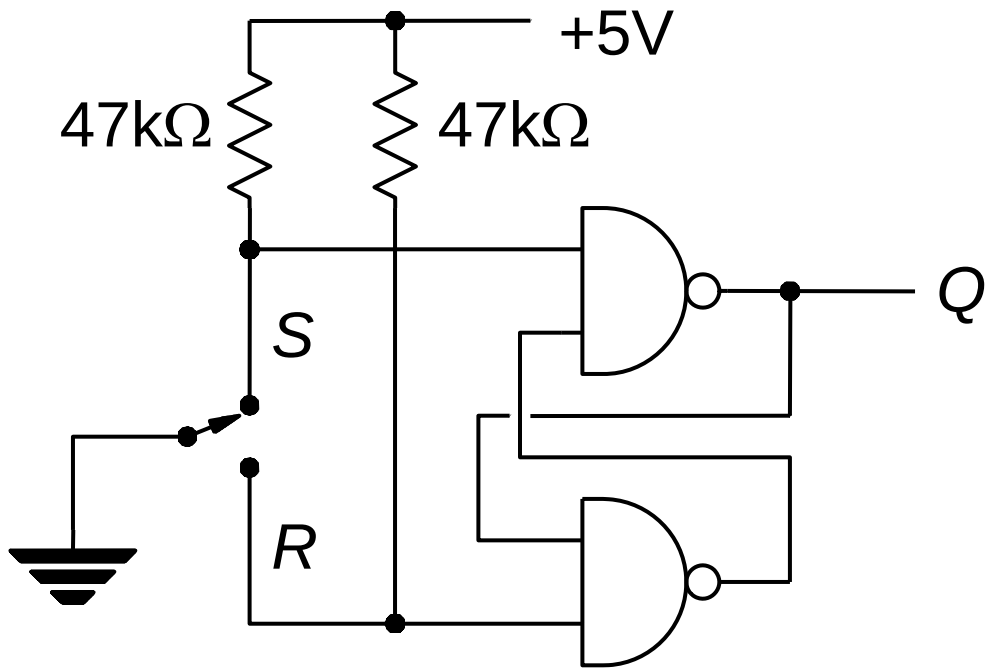
Like NOR-based RS flip-flops, NAND RS flip-flops are not normally used for data storage

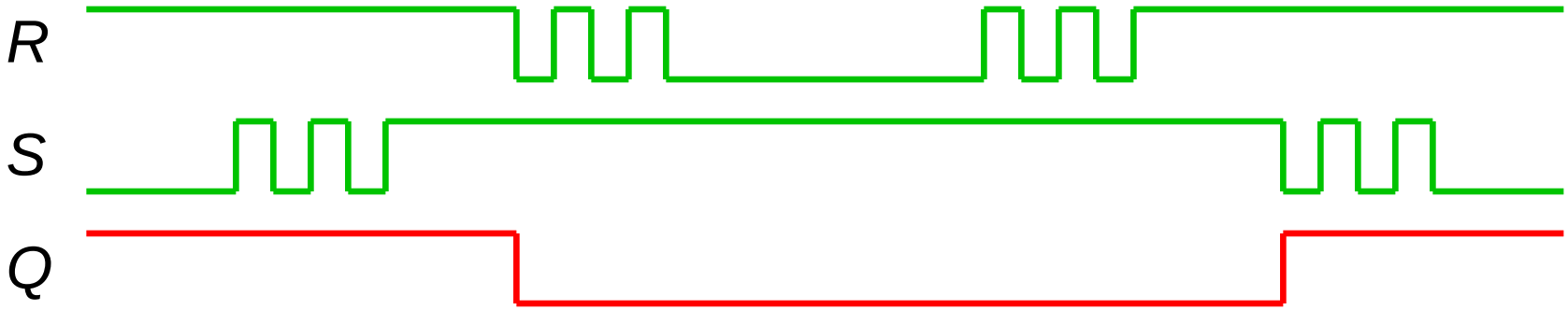
A common application is switch de-bouncing

Switch De-Bouncer

University of Reading

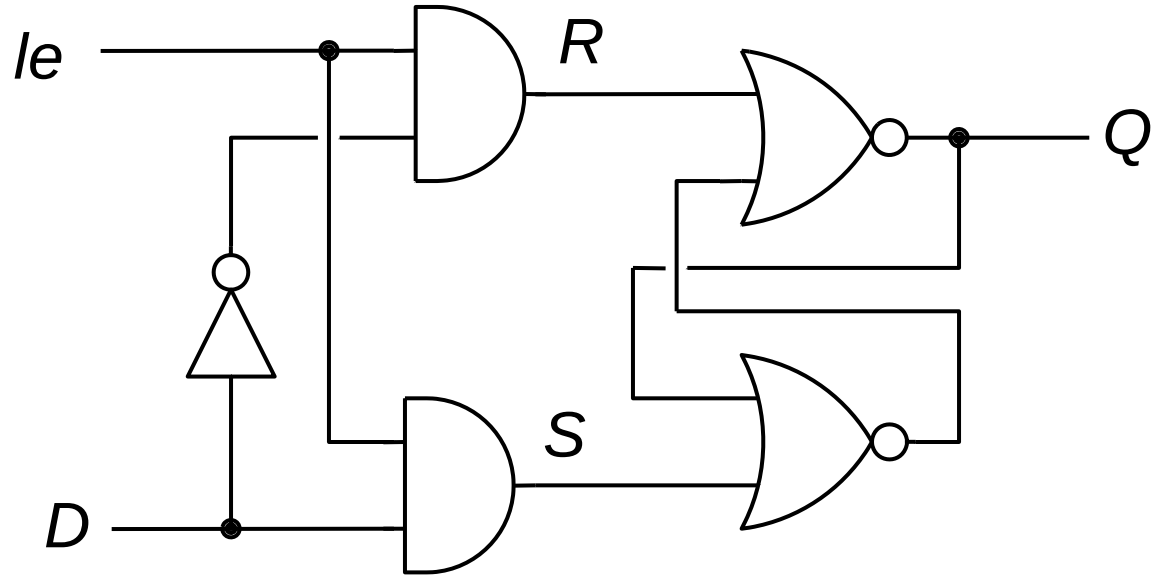
Assume break before make





The D-Latch

With the addition of two AND gates and an inverter the RS flip-flop becomes a D-latch:



$le = 0 \rightarrow S = 0, R = 0:$ $Q = Q-$ (latch holds data)

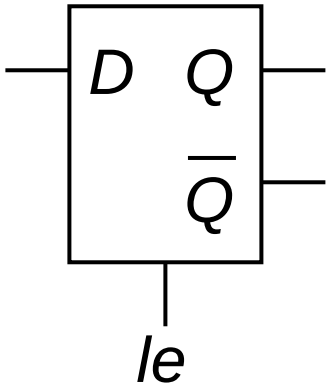
$le = 1 \rightarrow S = D, R = \overline{D}:$ $Q = D$ (latch accepts data)

NB: $S = 1, R = 1$ cannot occur

The D-Latch

The operation of the D-latch is to accept data when *le*=1, and to store the data when *le*=0

Most D-latches also have an inverted output $\overline{Q}$



le	*Q*
0	*Q*-
1	*D*

Note that the control terminal is level-sensitive: as long as *le*=1 the latch remains transparent (the output *Q* follows the input *D*)

Edge-Triggered Flip-Flops

University of Reading

Level-sensitive flip-flops are not suitable for use in clocked sequential systems such as counters

For such applications it is necessary to use a different type of flip-flop which is edge-sensitive

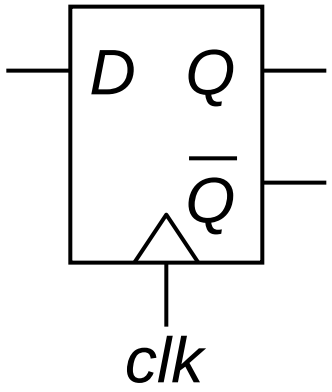
Edge-sensitive flip-flops respond only to a change in the value of the control input (clock)

In an edge-sensitive flip-flop the output can only change at the instant of the clock transition

The output value depends on the data inputs immediately before the clock transition

Edge-Triggered Flip-Flops

The edge-triggered D-type flip-flop has a single data input *D*, and a clock input *clk*



The output *Q* changes only on a 0-to-1 transition of *clk*:

$$Q+ = D-$$

where *Q*+ is the output value after the transition and *D*- is the data input value immediately before the transition

Synchronous and Asynchronous Systems

University of Reading

Sequential systems using edge-triggered flip-flops are classified according to how the clock inputs of the flip-flops are connected

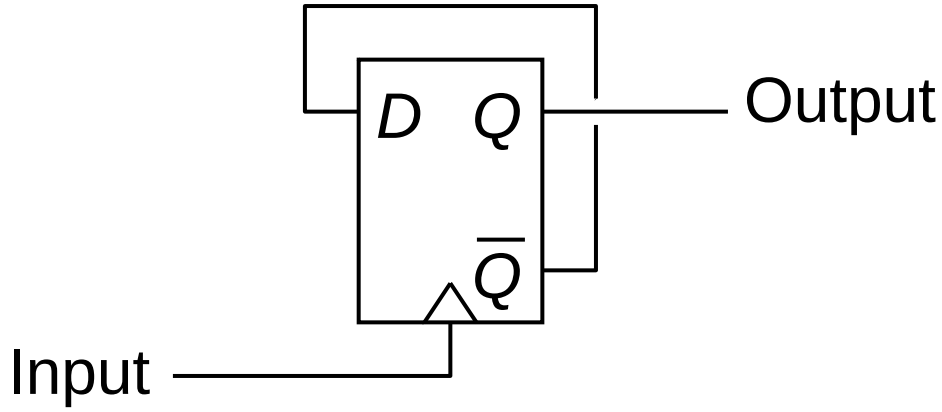
Synchronous systems:
The clocks of all flip-flops are connected together to a common source

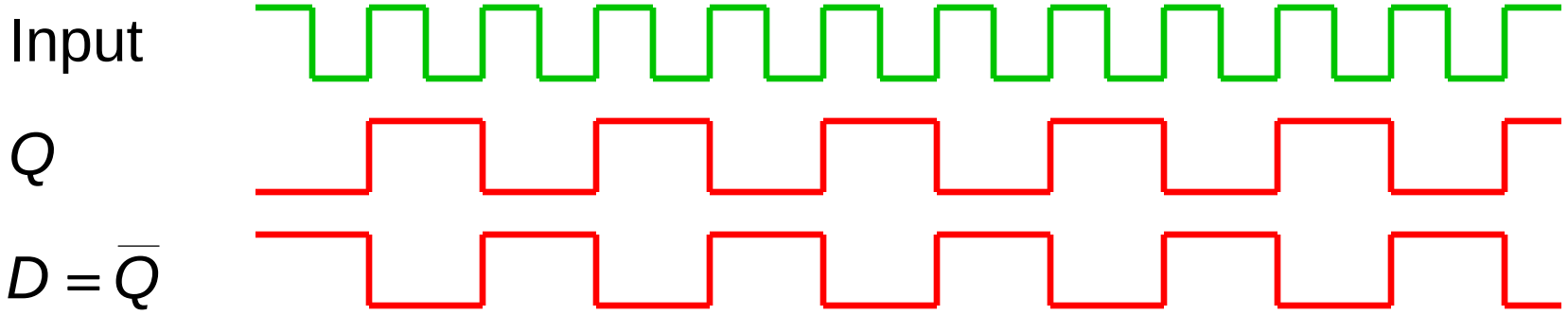
Asynchronous systems:
The clocks of the flip-flops are derived from different sources (usually the outputs of other flip-flops)

Asynchronous Counters

University of Reading

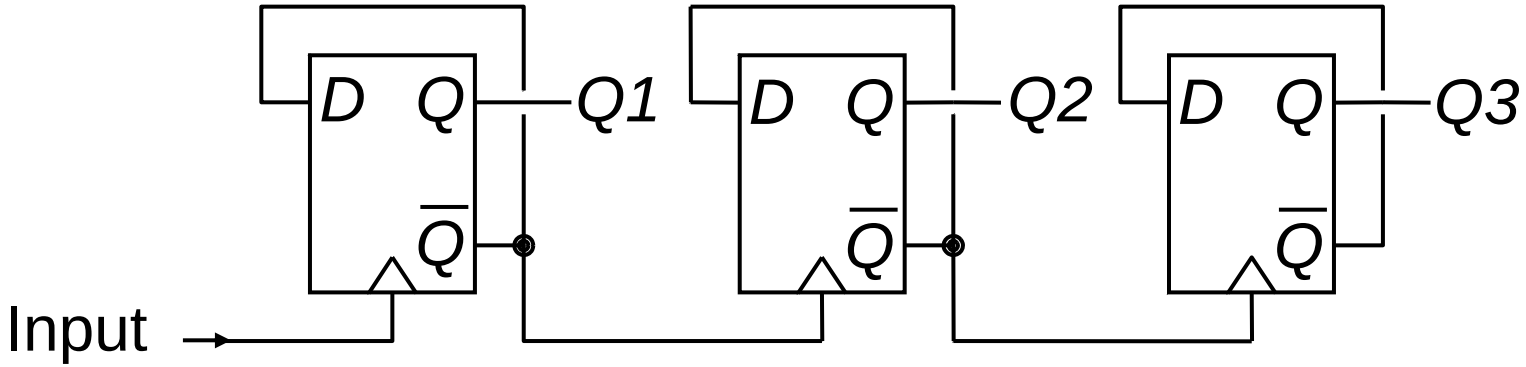
A divide-by-2 counter can be constructed by connecting the *D* input to the inverted output

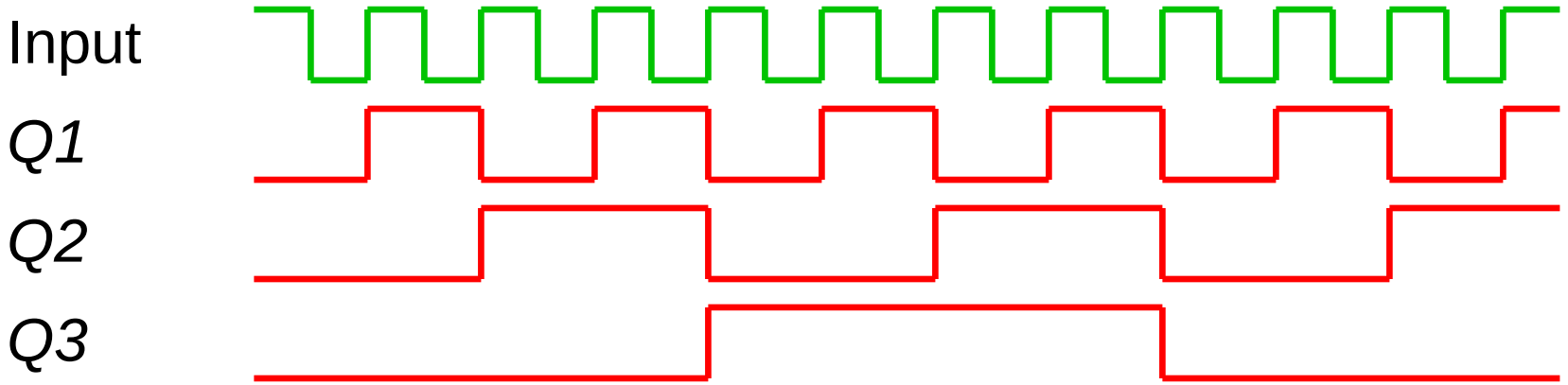




Asynchronous Counters

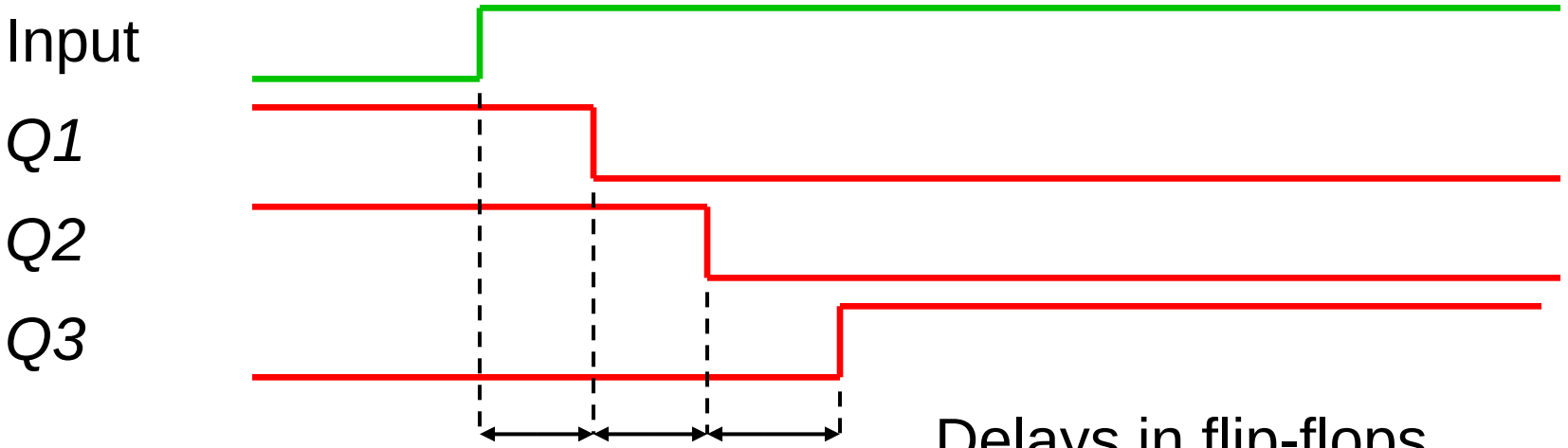
Divide-by-2 stages can be cascaded to construct a natural binary counter:





Asynchronous Counters

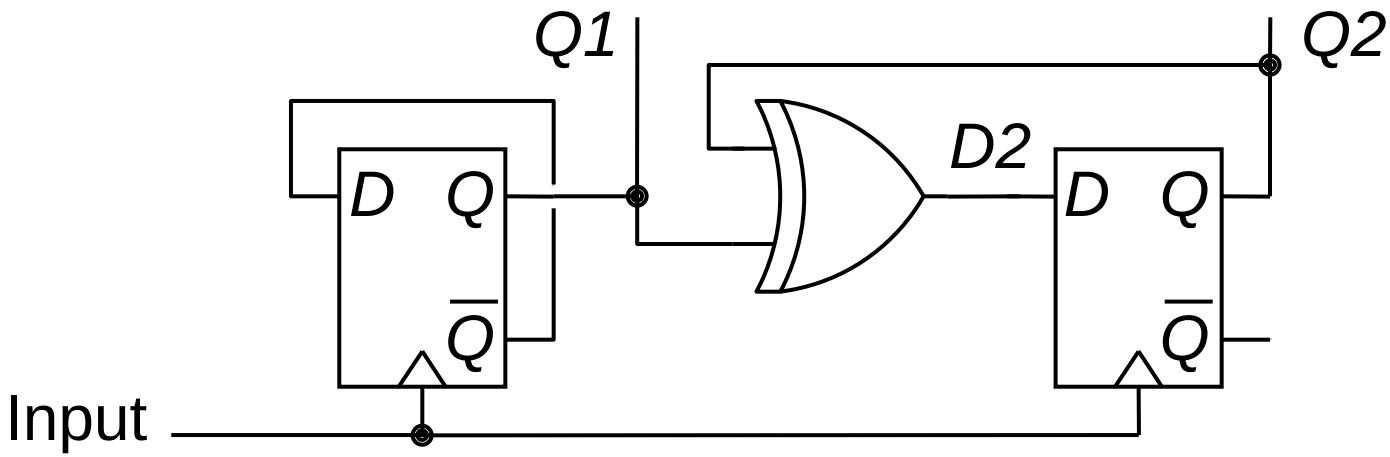
Delays in the flip-flops cause spurious outputs while the flip-flops are in the process of changing state following an input transition:

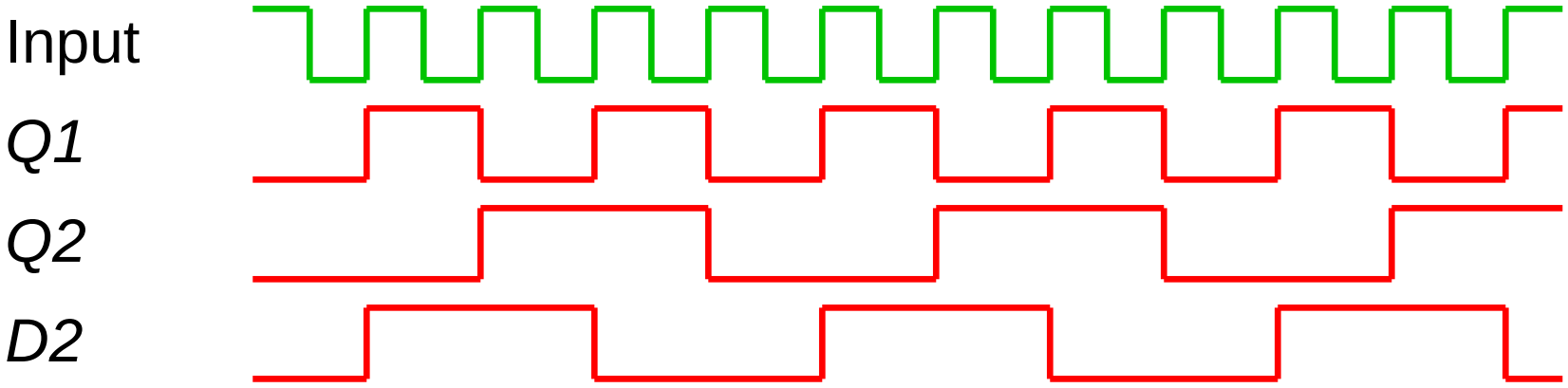


Q1:	1	0	0	0
Q2:	1	1	0	0
Q3:	0	0	0	1

Synchronous Counters

In synchronous counters the clock terminals of all the flip-flops are connected to the input

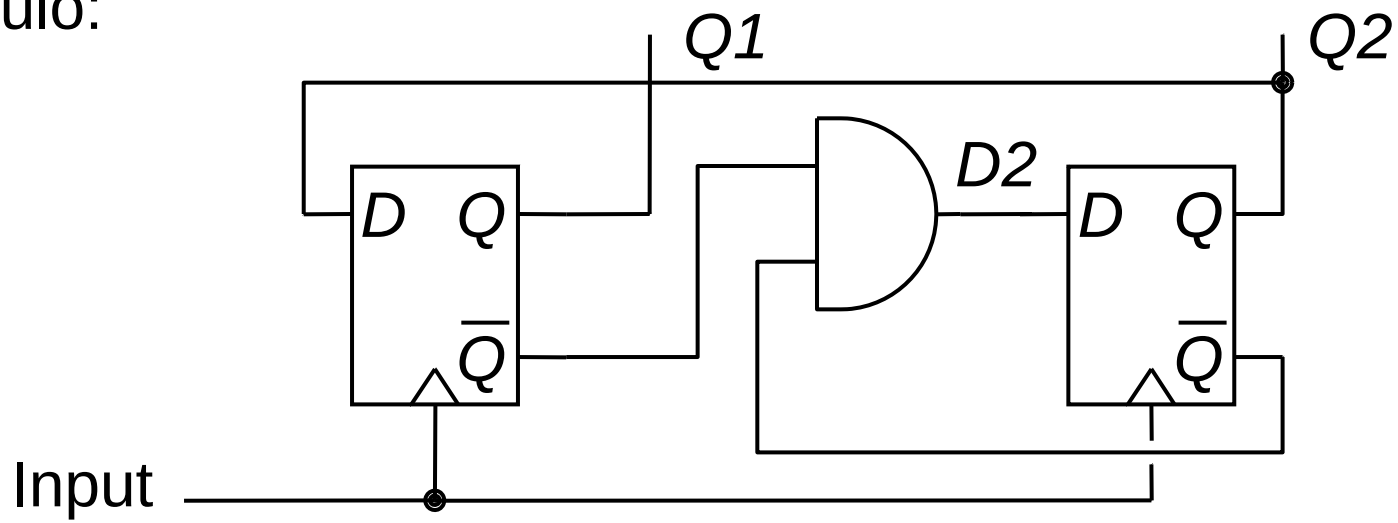


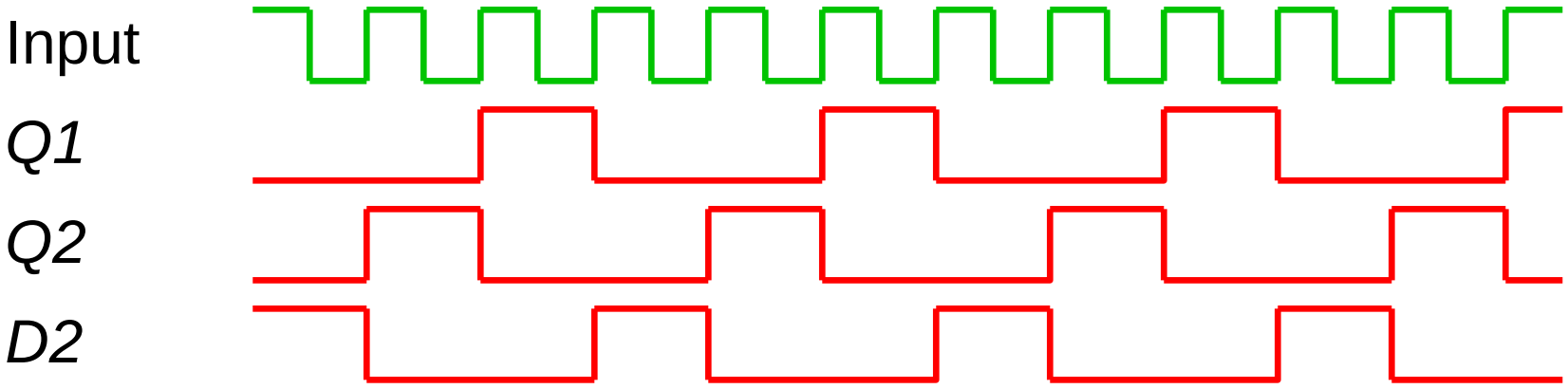


Synchronous Counters

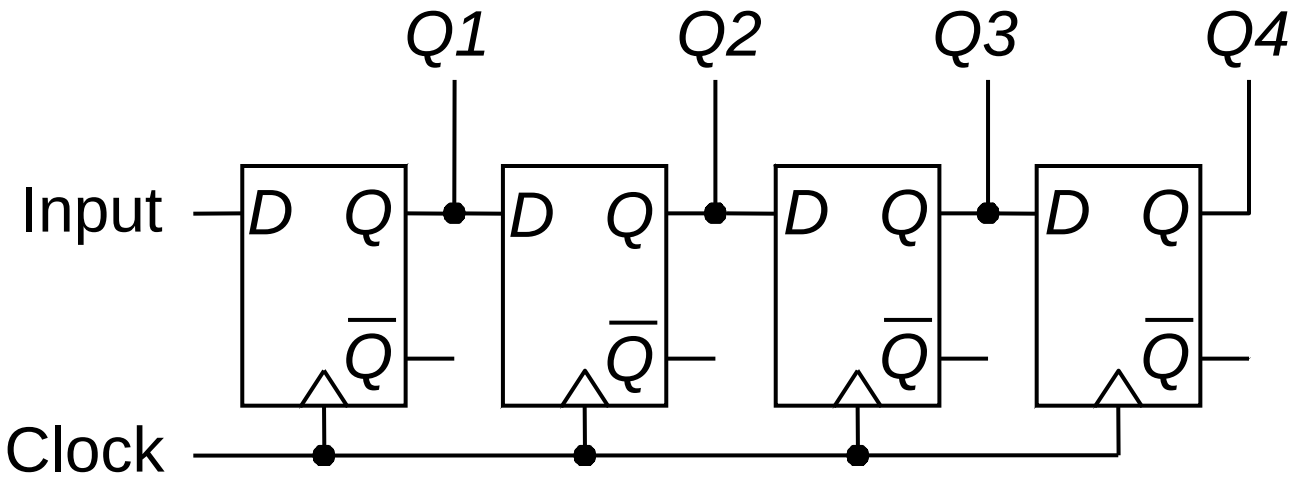
University of Reading

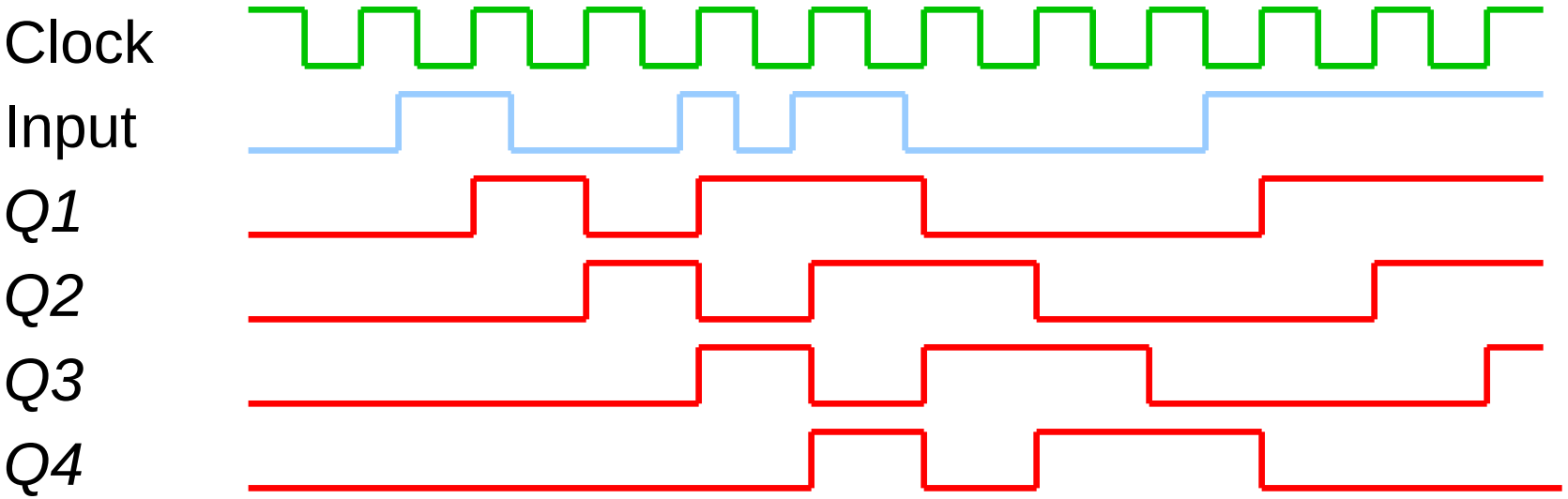
Synchronous counters can easily be made to count in any modulo:





Shift Registers





Serial-to-Parallel Conversion

University of Reading

Serial data can be clocked into the shift register until the complete data word is stored

The data is then available in parallel form at the outputs of the shift register stages

Parallel-to-serial conversion is more difficult using the standard D-type flip-flop

Some flip-flops have asynchronous load which allows data to be forced onto the output

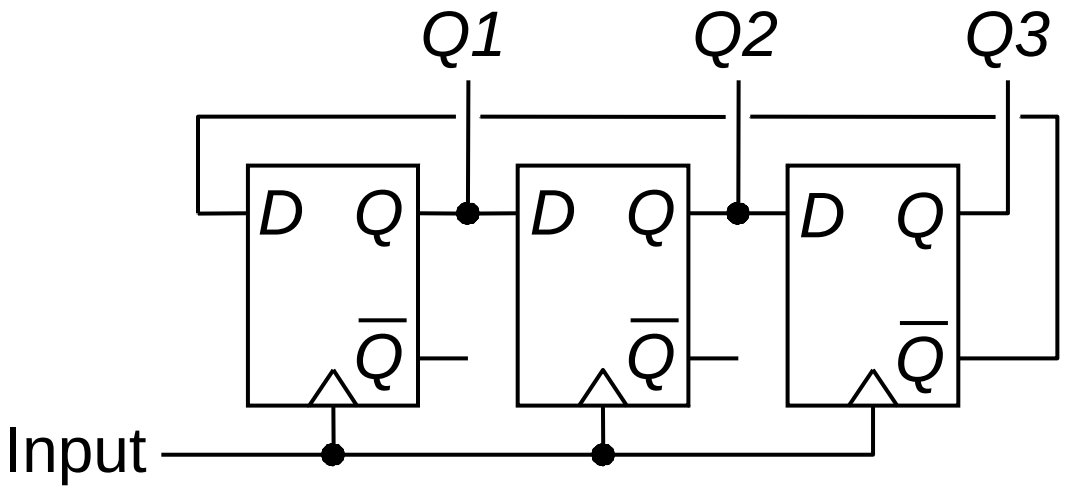
Data loaded in parallel into the flip-flops can be clocked out in serial form

Johnson-Code Counters

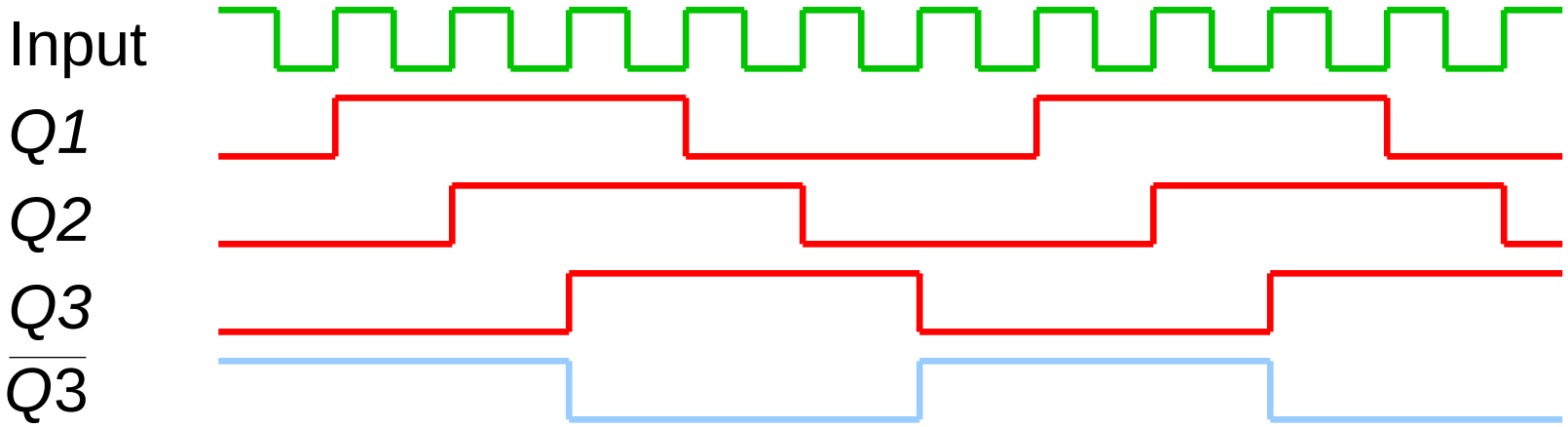
Johnson-code counters are shift registers with feedback from the inverting output.

They are synchronous and count modulo 2*n* where *n* is the number of flip-flops

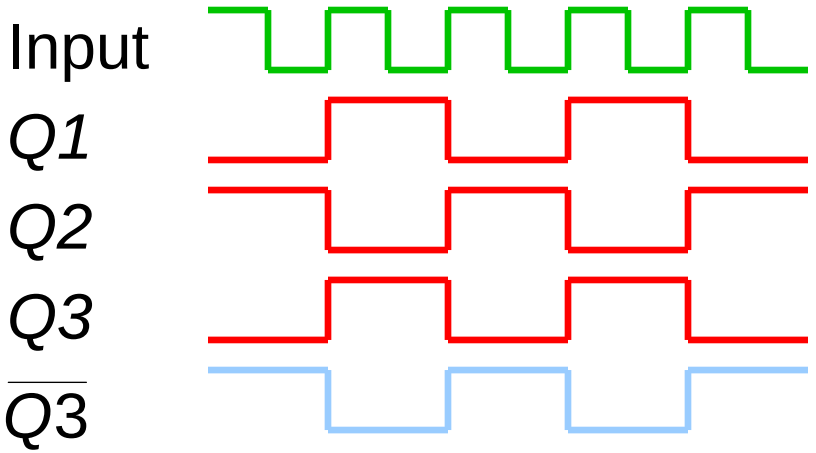
3-stage Johnson-code counter:



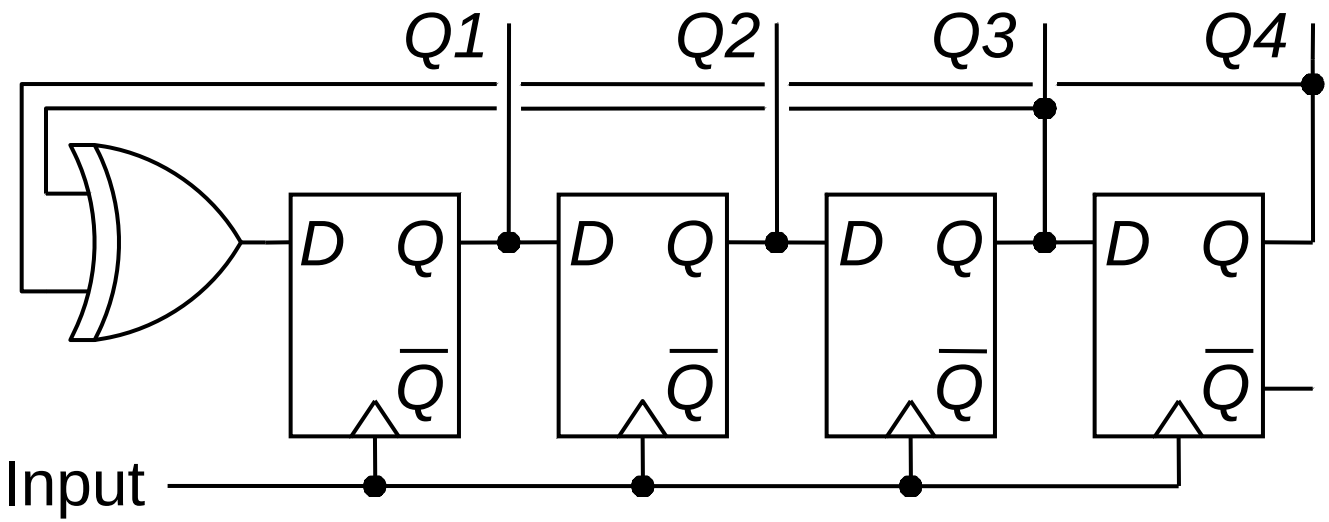
Johnson-Code Counters

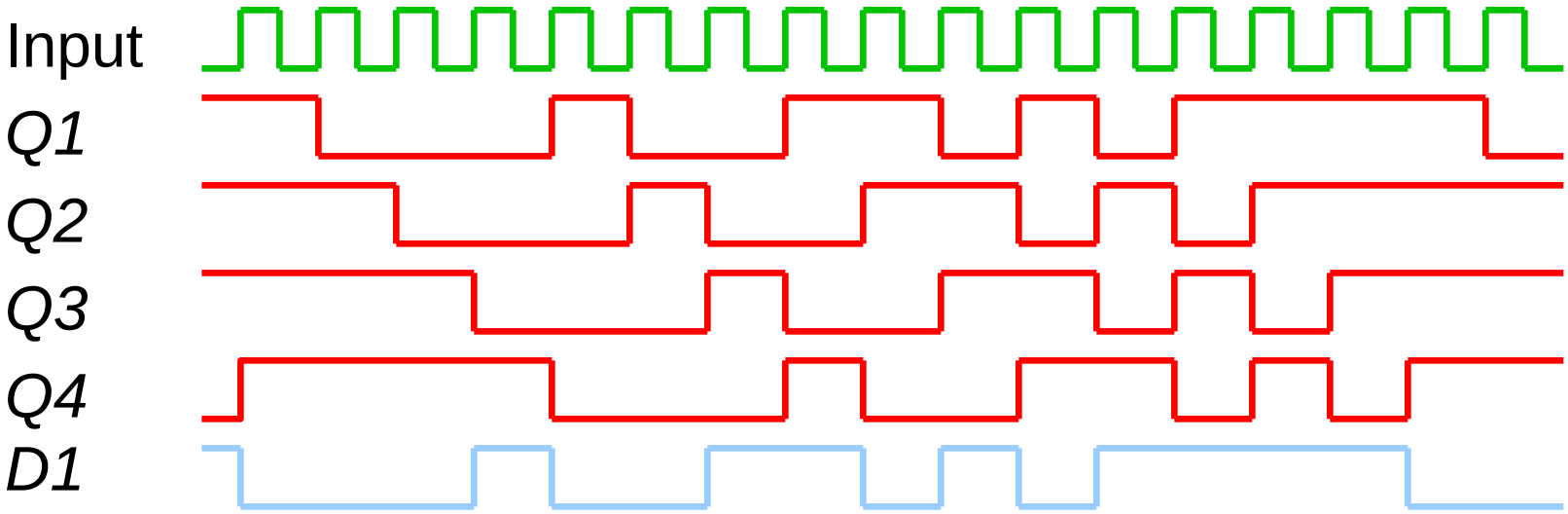


Precautions must be taken to prevent the counter getting into one of the unused states:



Chain-Code Generators

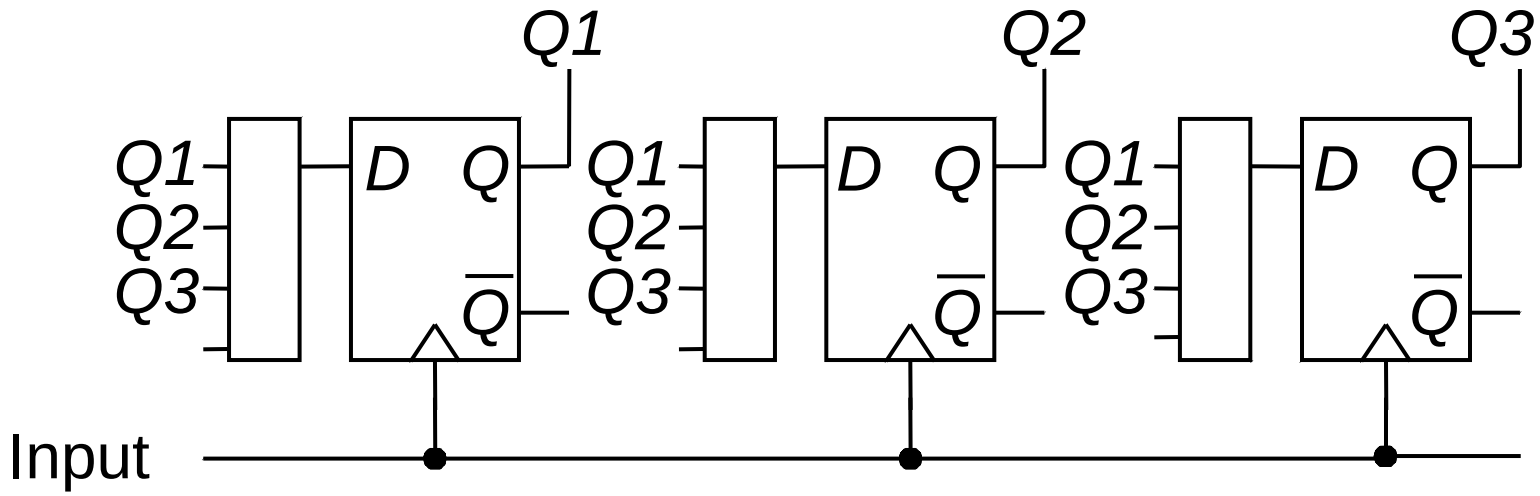




Design of Synchronous Counters

Counter is required to count modulo-*m*: then *n* flip-flops will be required where $2^n \geq m$

The counter will consist of the *n* flip-flops connected to a common clock, each with a combinational logic network generating its *D* input from all the outputs:

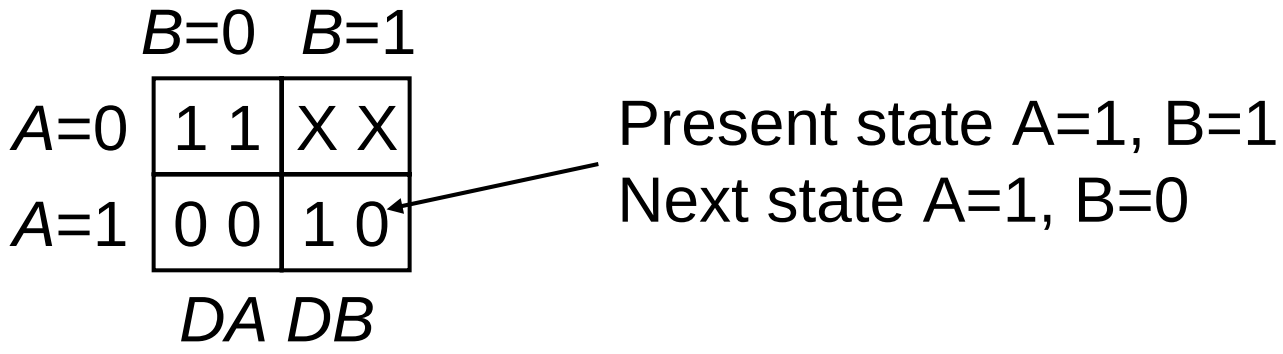


Design of Synchronous Counters

Modulo-*3* counter with *2* outputs *A* and *B*:

State	*A*	*B*	
1	0	0	
2	1	1	
3	1	0	
1	0	0	etc.

Y-map shows next state for each present state:

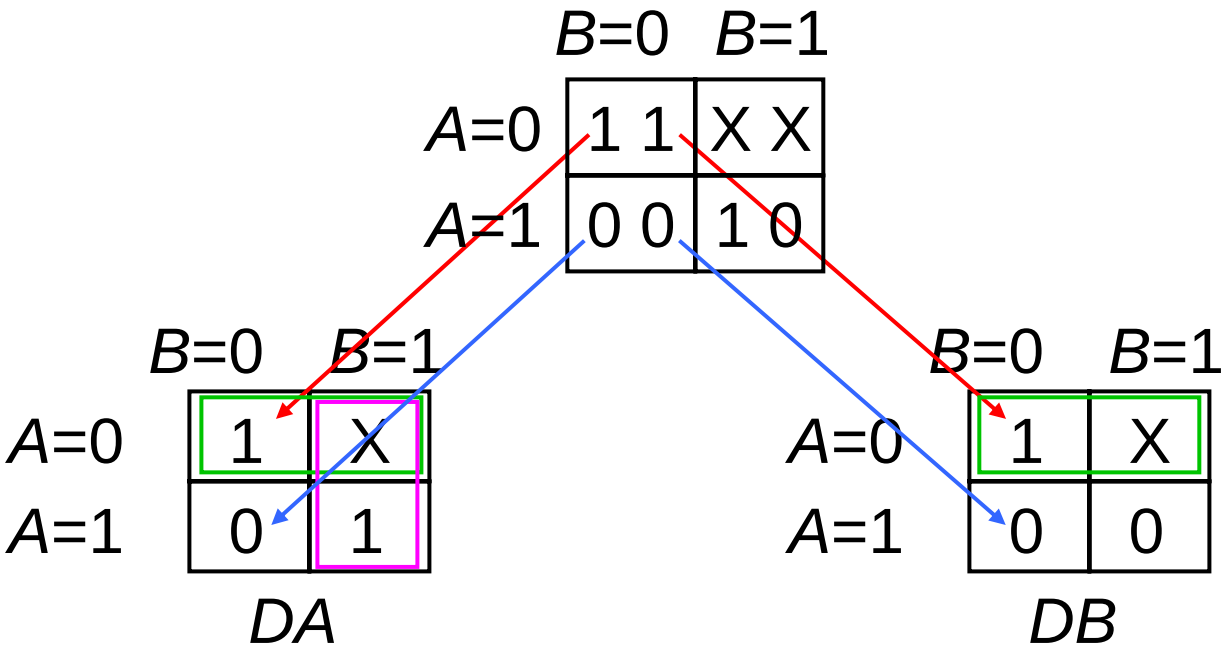


State *A*=0, *B*=1 does not occur in the required sequence, and is termed an *unused state*

Design of Synchronous Counters

University of Reading

Now split Y-map into separate K-maps:



Thus: $DA = \overline{A} + B$ $DB = \overline{A}$

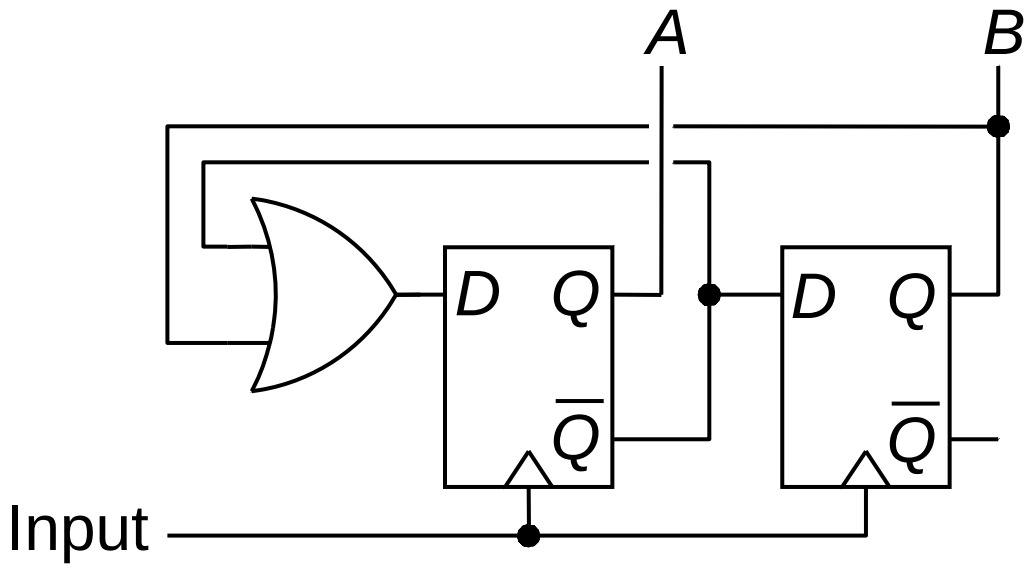
Design of Synchronous Counters

University of Reading

Check unused states:

$$DA = \overline{A} + B \qquad DB = \overline{A}$$

A=0, B=1 : DA=1+1=1, DB=1 → A=1, B=1 ✓



Design of Synchronous Counters

If the counter gets into any unused state it should regain the correct counting sequence after a limited number of input clock cycles

If necessary modify K-maps to force the unwanted state (01) back to a correct sequence state (00):

	$B=0$	$B=1$
$A=0$	1	0
$A=1$	0	1

DA

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

	$B=0$	$B=1$
$A=0$	1	0
$A=1$	0	0

DB

$$DB = \overline{A}.\overline{B}$$

These functions are more complex than originals

Design Example

University of Reading

Design a modulo-5 synchronous counter, using D-type flip-flops and NAND gates, that has 3 outputs *A*, *B*, *C* and counts in the sequence:

State	*A*	*B*	*C*	
1	0	1	0	
2	1	0	0	
3	0	0	1	
4	0	1	1	
5	1	1	0	
1	0	1	0	etc.

The number of flip-flops *n* required is given by:

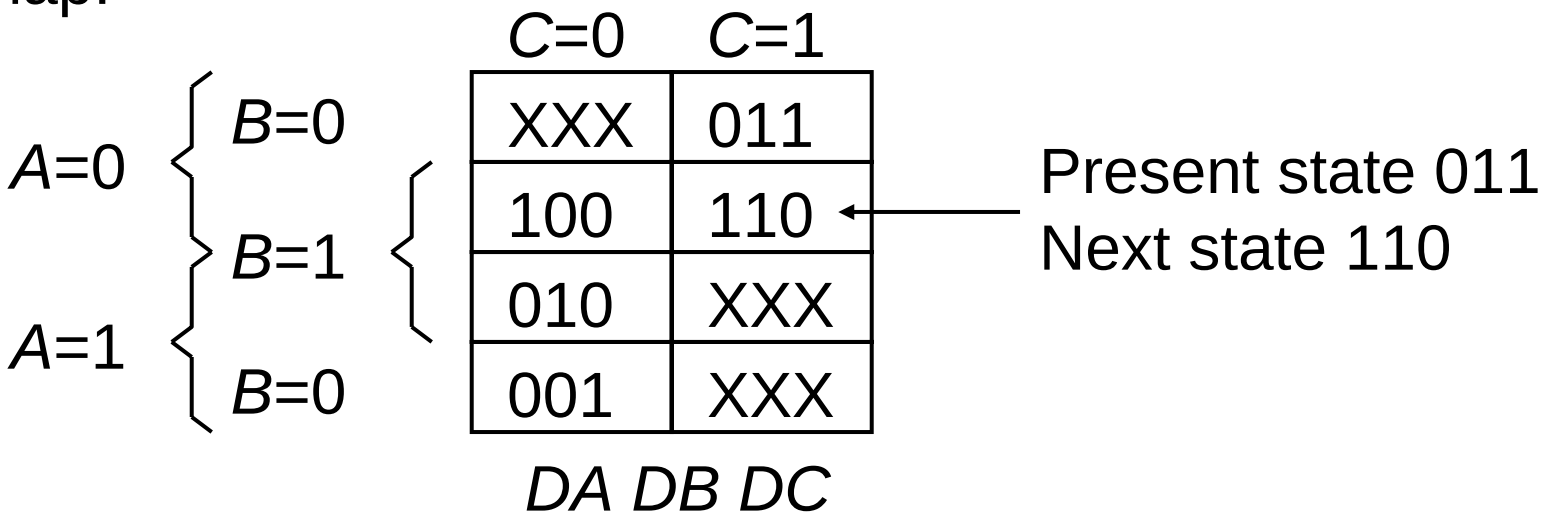
$$2^n \geq 5 \rightarrow n = 3$$

Design Example

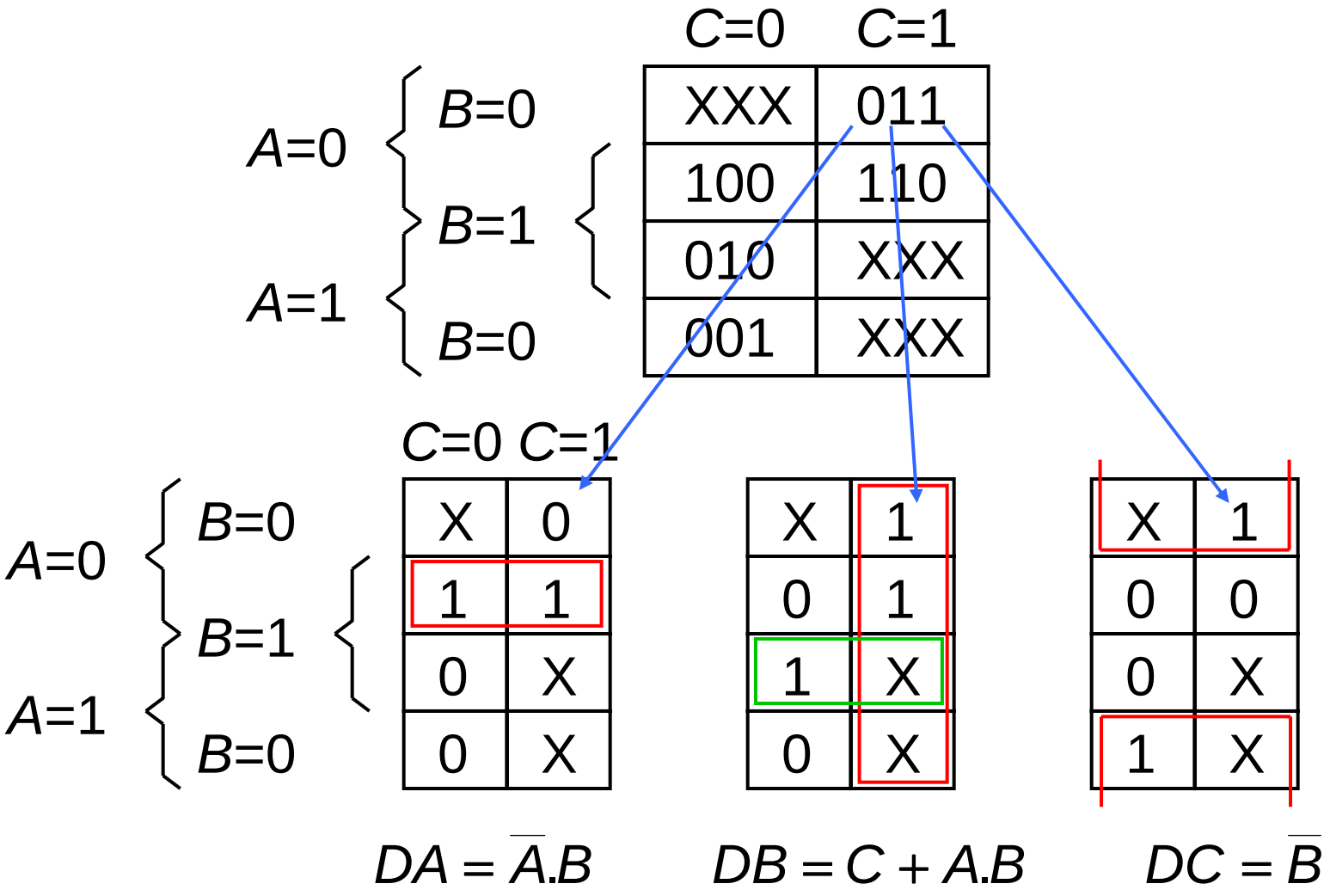
University of Reading

State	*A*	*B*	*C*	
1	0	1	0	
2	1	0	0	
3	0	0	1	
4	0	1	1	
5	1	1	0	
1	0	1	0	etc.

Y-map:



Design Example



Design Example

University of Reading

$$DA = \overline{A}.B \qquad DB = C + A.B \qquad DC = \overline{B}$$

Unwanted states:

A=0, B=0, C=0 : DA=0, DB=0, DC=1 → A=0, B=0, C=1 ✓
A=1, B=0, C=1 : DA=0, DB=1, DC=1 → A=0, B=1, C=1 ✓
A=1, B=1, C=1 : DA=0, DB=1, DC=0 → A=0, B=1, C=0 ✓

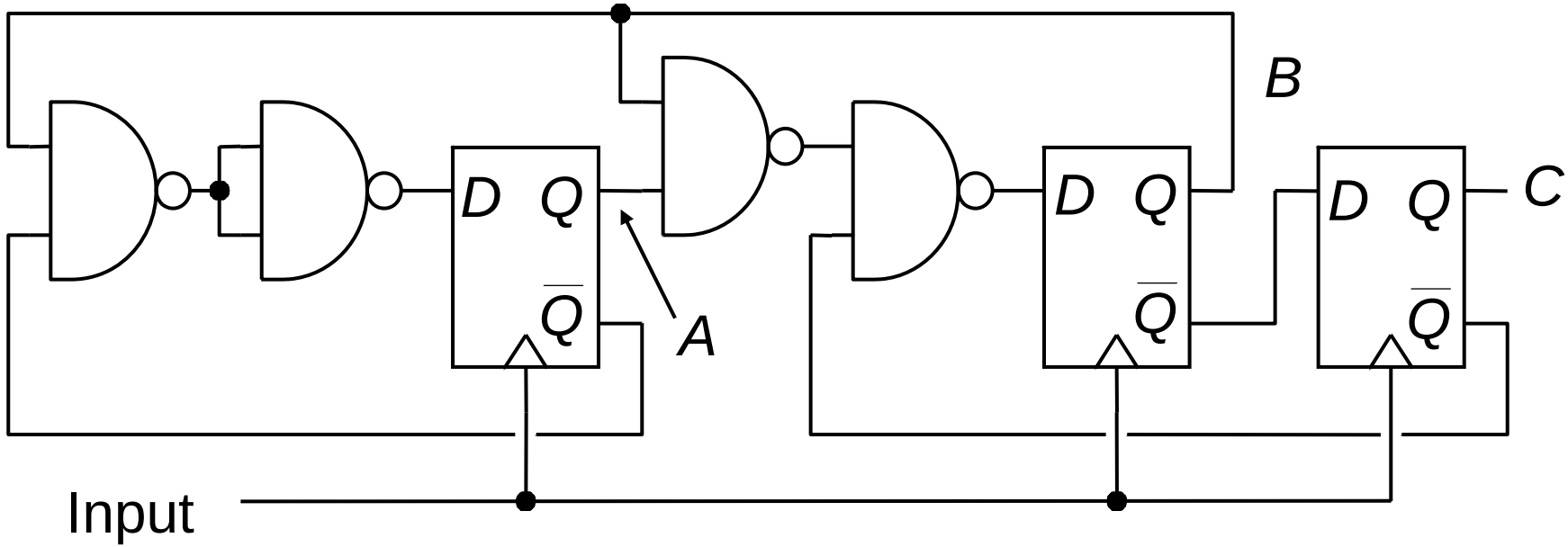
Convert to NAND form:

$$DA = \overline{\overline{A}.B} \qquad DB = \overline{\overline{C} . \overline{(A.B)}} \qquad DC = \overline{B}$$

Design Example

University of Reading

$$DA = \overline{\overline{A}.B} \qquad DB = \overline{C} \,.\, \overline{(A.B)} \qquad DC = \overline{B}$$



Design Example

University of Reading

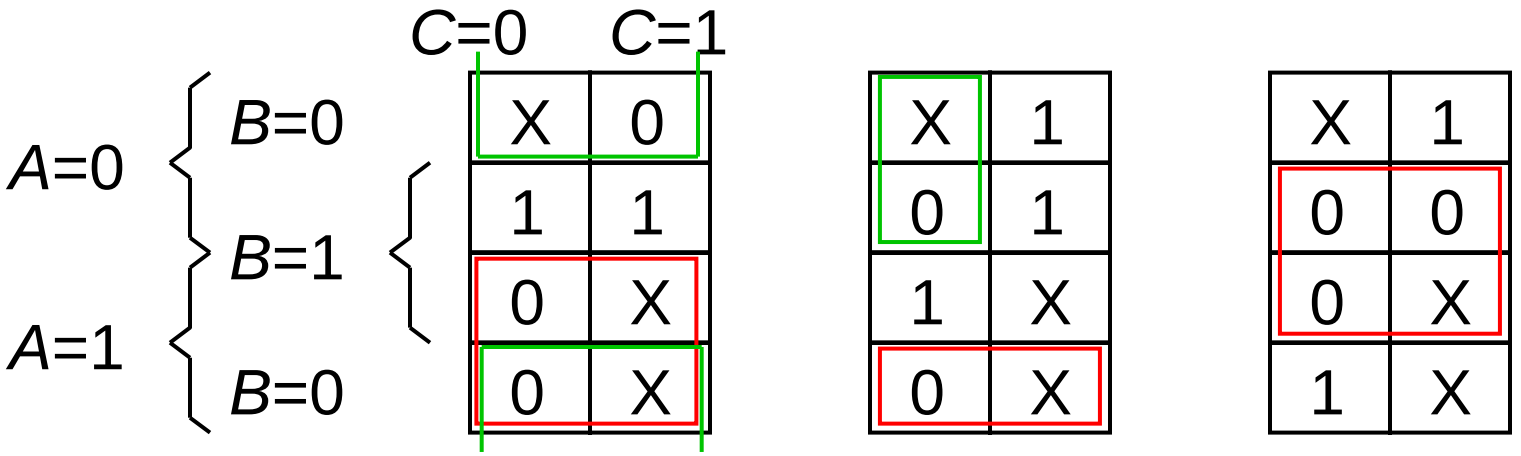
Design a modulo-5 synchronous counter, using D-type flip-flops and NOR gates, that has 3 outputs *A*, *B*, *C* and counts in the sequence:

State	*A*	*B*	*C*	
1	0	1	0	
2	1	0	0	
3	0	0	1	
4	0	1	1	
5	1	1	0	
1	0	1	0	etc.

The number of flip-flops *n* required is given by:

$$2^n \geq 5 \rightarrow n = 3$$

Design Example



$$\overline{DA} = A + \overline{B} \quad \overline{DB} = A.\overline{B} + \overline{A}.\overline{C} \quad \overline{DC} = B$$

Duality:

$$DA = \overline{A}.B \quad DB = (\overline{A} + B).(A + C) \quad DC = \overline{B}$$

Design Example

University of Reading

$$DA = \overline{A}.B \qquad DB = (\overline{A} + B).(A + C) \qquad DC = \overline{B}$$

Unwanted states:

A=0, B=0, C=0 : DA=0, DB=0, DC=1 → A=0, B=0, C=1 ✓
A=1, B=0, C=1 : DA=0, DB=0, DC=1 → A=0, B=0, C=1 ✓
A=1, B=1, C=1 : DA=0, DB=1, DC=0 → A=0, B=1, C=0 ✓

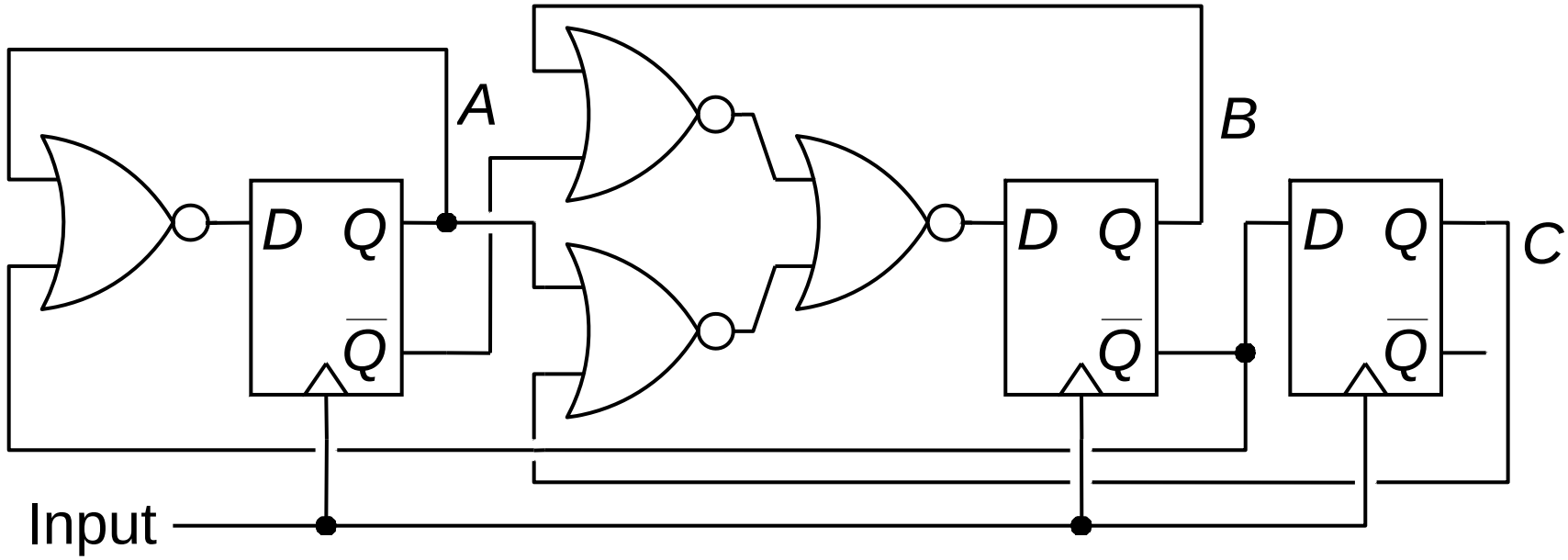
Convert to NOR form:

$$DA = \overline{\overline{A} + \overline{B}} \qquad DB = \overline{\overline{(\overline{A} + B)} + \overline{(A + C)}} \qquad DC = \overline{\overline{B}}$$

Design Example

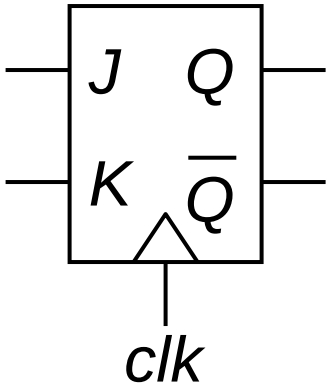
University of Reading

$$DA = \overline{A + \overline{B}} \qquad DB = \overline{(\overline{A} + B) + (A + C)} \qquad DC = \overline{B}$$



Edge-Triggered JK Flip-Flops

The edge-triggered JK flip-flop has 2 data inputs *J*, *K*, and a clock input *clk*



The output *Q* changes only on a 0-to-1 transition of *clk*

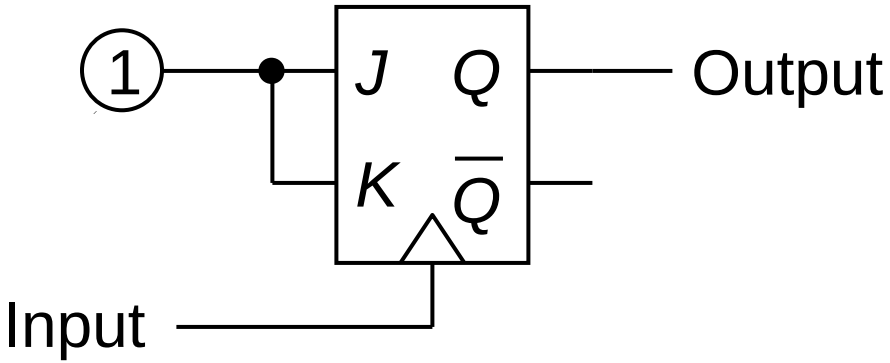
The output *Q*+ immediately after the clock transition depends on the output *Q*- and the inputs *J*-, *K*- immediately before the transition

Edge-Triggered JK Flip-Flops

University of Reading

J-	K-	Q+
0	0	Q-
1	0	1
0	1	0
1	1	$\overline{Q}$-

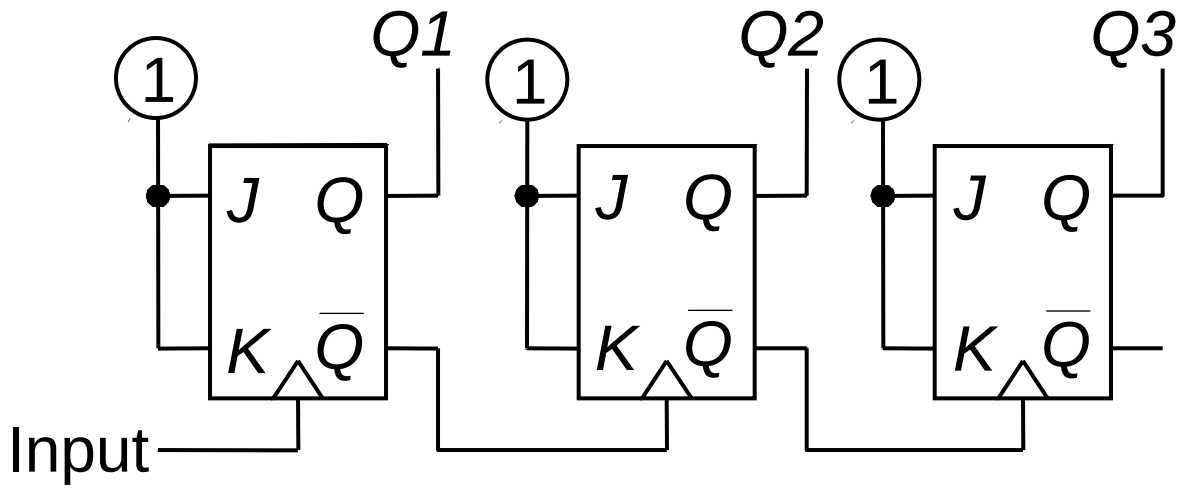
A divide-by-2 counter can be constructed by connecting the J and K inputs to logic 1:

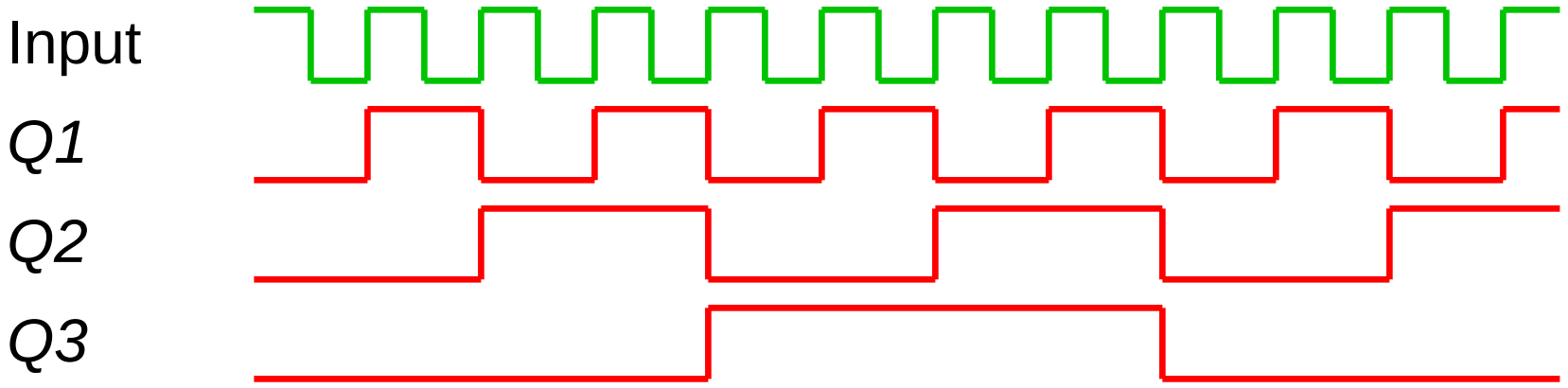


Edge-Triggered JK Flip-Flops

University of Reading

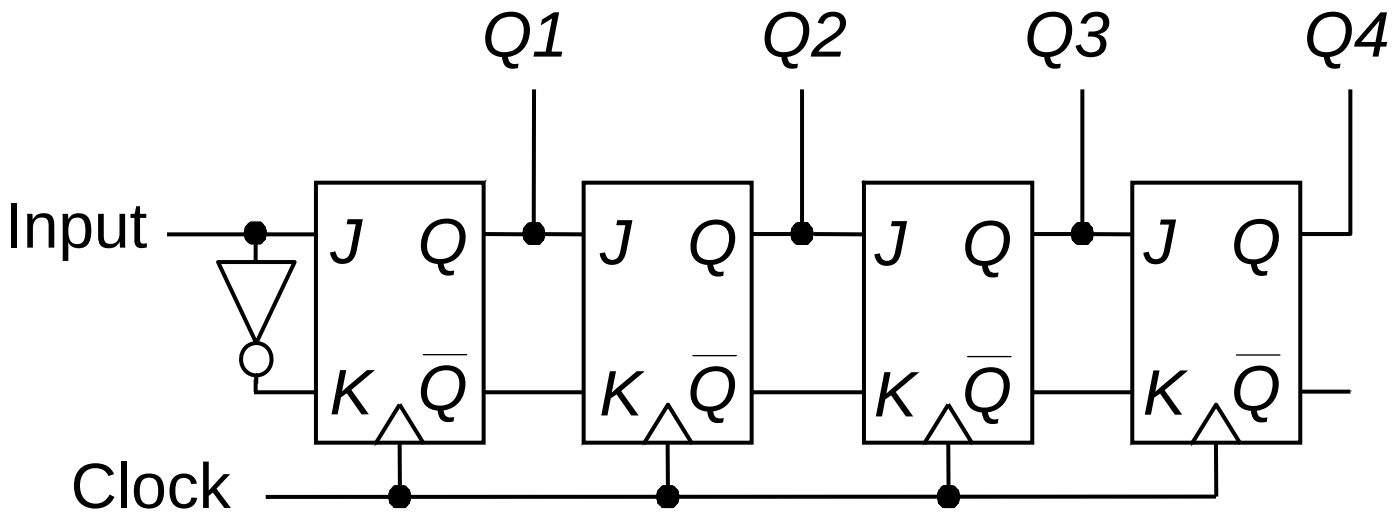
Asynchronous natural binary counter:





Edge-Triggered JK Flip-Flops

Shift register:



JK flip-flops can be used to design synchronous counters. The design process is more complicated than design with D-type flip-flops, but the result is often more economical

Types of Logic Gate

University of Reading

Electromechanical relays: slow (~ 10 ms), poor reliability, physically large (~ 5 cm^3), high power consumption

Vacuum tubes: faster (~ 10 μs), poor reliability, physically large (~ 2 cm^3), high power consumption

Semiconductor devices: fast, (~ 10 ns), excellent reliability, very small (~ 10^{-9} cm^3), low power consumption

Three main families of semiconductor logic gates: TTL, CMOS and ECL

Types of Logic Gate

University of Reading

The important parameters of a logic family are:

Power dissipation: this is usually dependent on the transition frequency at the gate output.

Propagation delay: determines the maximum operating speed

Noise margin: high value reduces the susceptibility to interference

Fan-out: the maximum number of gates inputs that can be connected to a single gate output

Transistor-Transistor Logic (TTL)

TTL was the first integrated logic family and is still the most commonly used

Logic levels:	logic 0	0.0 → 0.8 V
	logic 1	2.0 → 5.0 V

Basic TTL logic gate is NAND

Available in a number of variants including 74xxx, 74Sxxx, 74Lxxx, 74LSxxx, 74ASxxx, 74ALSxxx, 74Fxxx

Only the 74ALSxxx and 74Fxxx should be used in new designs

Properties of TTL Logic

University of Reading

(Values apply to the 74ALSxxx logic family)

Supply voltage: 5 V $\pm$ 0.25 V

Propagation delay: 4 ns

Speed-power product (10 MHz): 5 pJ

Quiescent power consumption: 1 mW

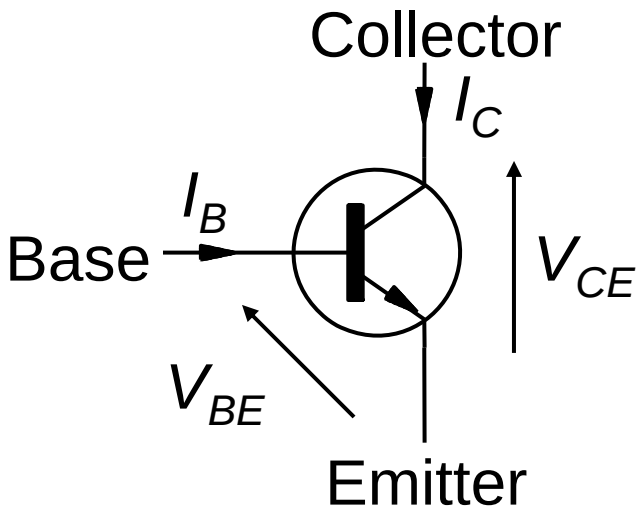
Noise margin: 0.3 V

Fan-out: 20

Cost: Medium

Transistor-Transistor Logic (TTL)

The bipolar transistor:



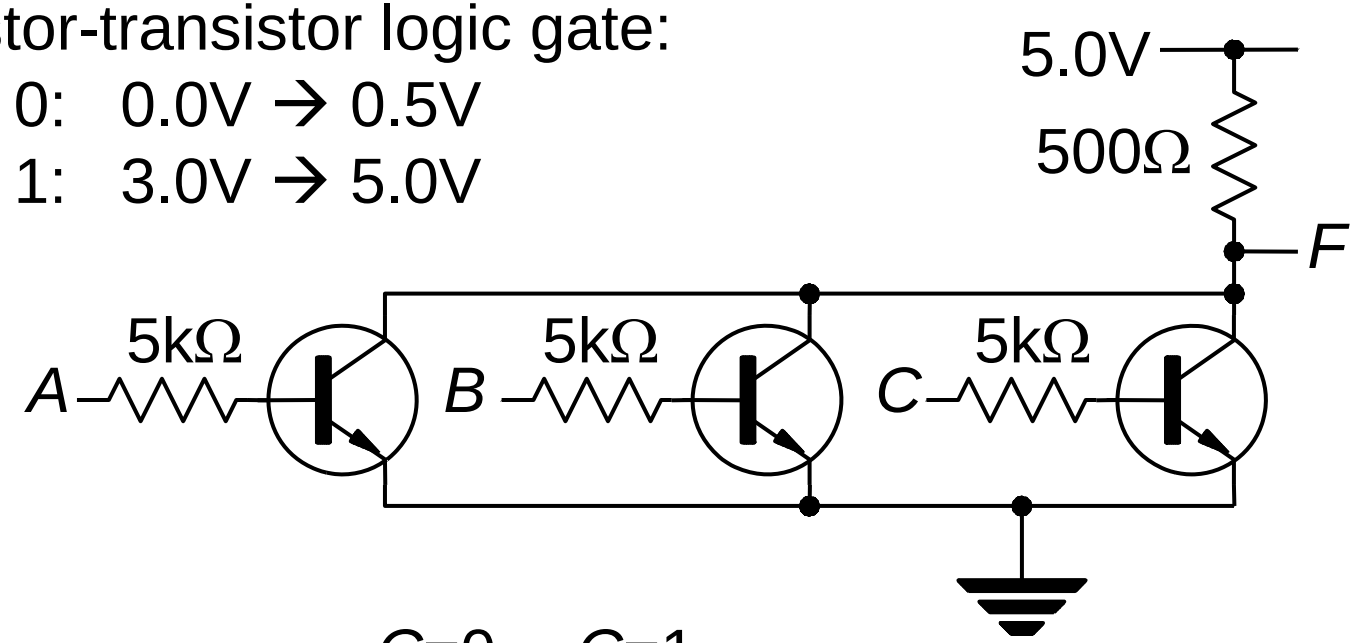
When used as a switch the transistor has two states:

OFF: $I_B = 0$ $I_C = 0$

ON: $I_B \gg I_C/\beta$ $V_{BE} = 0.6V$ $V_{CE} < 0.2V$

Transistor-Transistor Logic (TTL)

Resistor-transistor logic gate:
Logic 0: 0.0V → 0.5V
Logic 1: 3.0V → 5.0V



		C=0	C=1
A=0	B=0	1	0
	B=1	0	0
A=1	B=1	0	0
	B=0	0	0

$$F = \overline{A}.\overline{B}.\overline{C}$$

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

NOR gate

Complementary Metal-Oxide Semiconductor (CMOS) Logic

University of Reading

CMOS was introduced after TTL and is the technology used in nearly all LSI devices

It is now replacing TTL in most SSI and MSI applications because of its speed and lower power consumption

Logic levels:	logic 0	0.0 → 1.5 V
(Typical)	logic 1	3.5 → 5.0 V

There are also TTL-compatible CMOS logic families: 74HCTxxx and 74VHCT

Properties of CMOS Logic

University of Reading

(Values apply to the 74HCxxx logic family)

Supply voltage: 2V → 6V

Propagation delay: 10 ns

Speed-power product (10 MHz): 50 pJ

Quiescent power consumption: 10 μW

Noise margin: 1.25 V

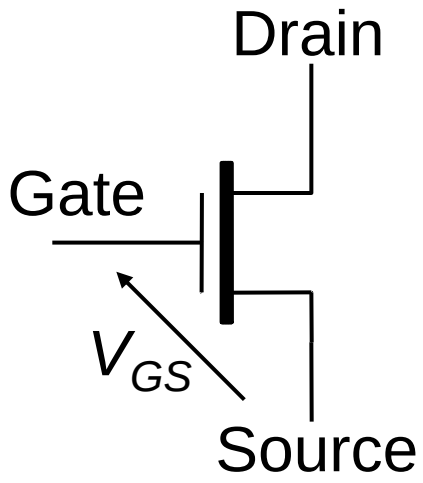
Fan-out: 20

Cost: low

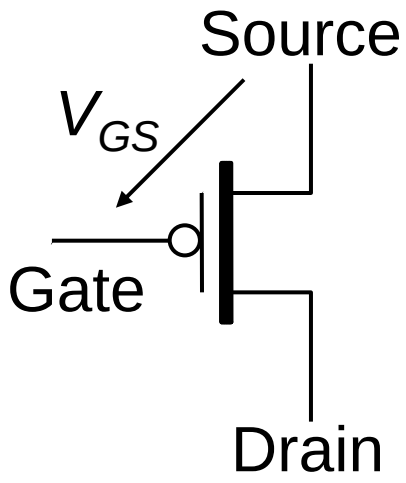
CMOS Logic

Metal-oxide semiconductor (MOS) transistors:

n-channel MOSFET



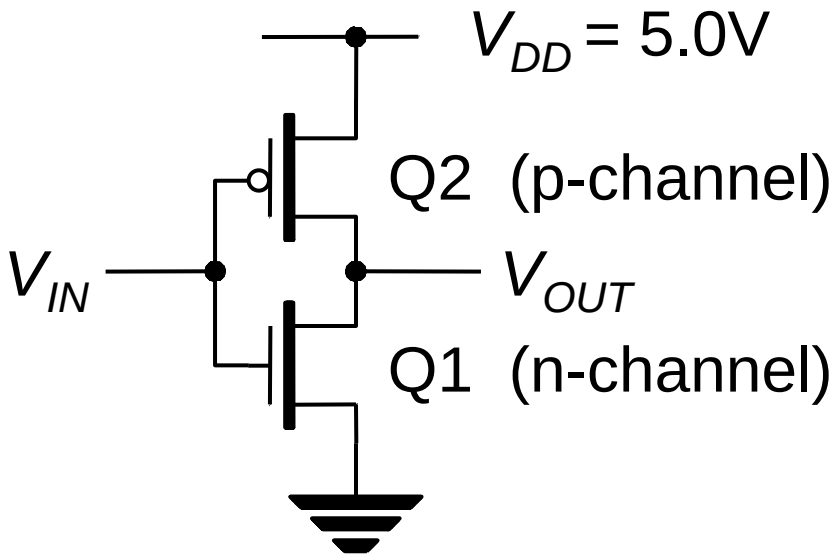
p-channel MOSFET



OFF: V_{GS} = 0.0→1.5V
R_{DS} > 1MΩ
ON: V_{GS} = 3.5→5.0V
R_{DS} < 10Ω

OFF: V_{GS} = -1.5→0.0V
R_{DS} > 1MΩ
ON: V_{GS} =-5.0→-3.5V
R_{DS} < 10Ω

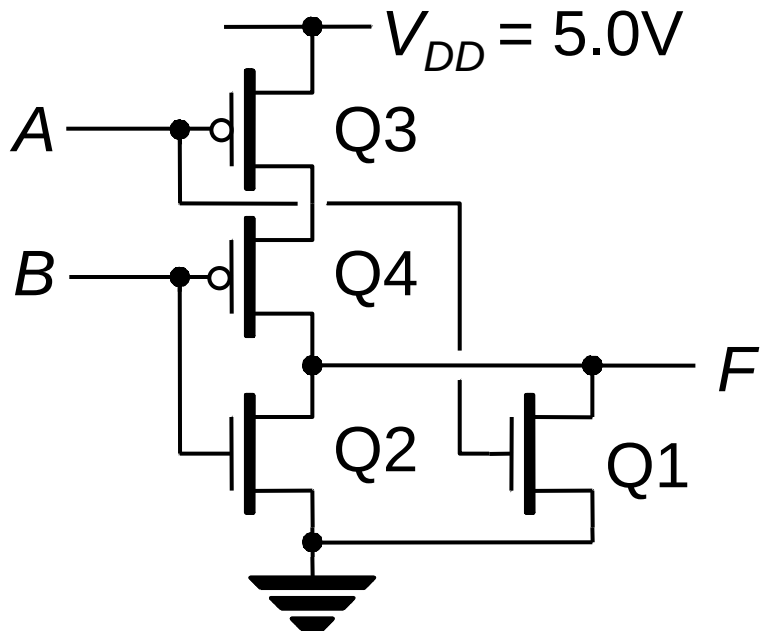
CMOS NOT Gate



V_{IN}	Q1	Q2	V_{OUT}
0.0 → 1.5V	OFF	ON	5.0V
3.5 → 5.0V	ON	OFF	0.0V

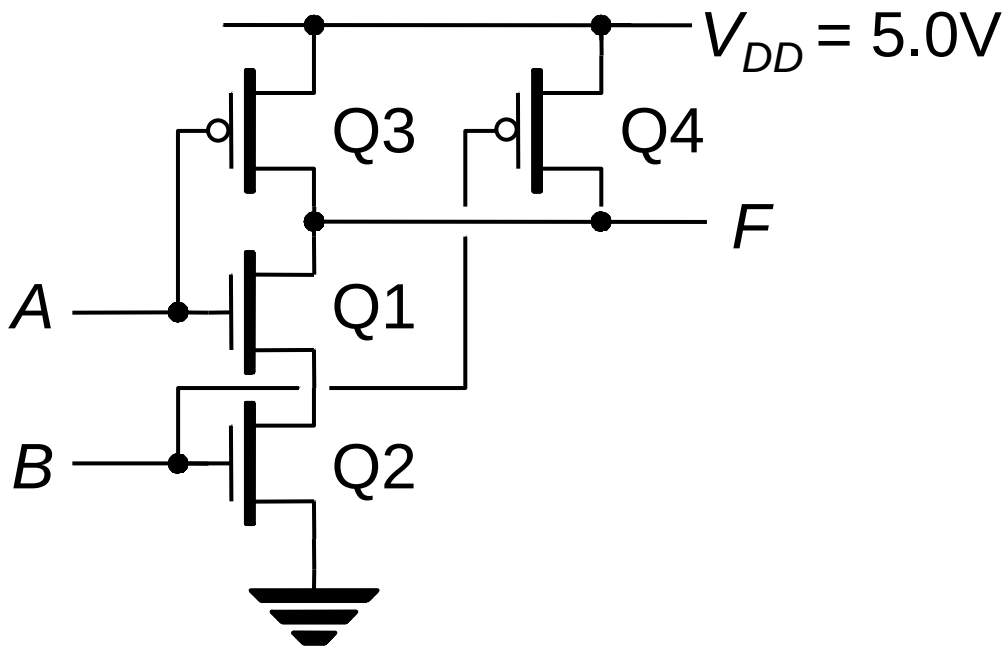
Thus when input is low, output is high, and when input is high, output is low

CMOS NOR Gate



A	*B*	Q1	Q2	Q3	Q4	*F*
lo	lo	OFF	OFF	ON	ON	hi
lo	hi	OFF	ON	ON	OFF	lo
hi	lo	ON	OFF	OFF	ON	lo
hi	hi	ON	ON	OFF	OFF	lo

CMOS NAND Gate



A	*B*	Q1	Q2	Q3	Q4	*F*
lo	lo	OFF	OFF	ON	ON	hi
lo	hi	OFF	ON	ON	OFF	hi
hi	lo	ON	OFF	OFF	ON	hi
hi	hi	ON	ON	OFF	OFF	lo

Emitter-Coupled Logic (ECL)

University of Reading

ECL is the fastest, and most difficult to use, logic family

Logic levels: logic 0 -1.850 → -1.475 V
logic 1 -1.105 → -0.810 V

Basic ECL logic gate is OR / NOR

Difficult to interface to other logic families

Extreme switching speed means that length of interconnections becomes significant

Used only when the fastest switching speed is essential

Properties of ECL Logic

University of Reading

(Values apply to the 100K logic family)

Supply voltage: -4.5 V

Propagation delay: 0.75 ns

Speed-power product: 80 pJ

Quiescent power consumption: 40 mW

Noise margin: 0.125 V

Fan-out: 20

Cost: high

Decoupling

University of Reading

All logic devices take current pulses from the power supply when switching output state

This causes negative-going pulse on the power supply (because of supply impedance)

Thus switching in one device could affect another device

This problem is overcome by decoupling: capacitors are placed across the supply close to the logic devices

Typically 47 nF ceramic capacitors are used

For high-speed logic a decoupling capacitor is required close to each device

Introduction to Digital Circuits

University of Reading