Digital Electronics

BY

Dr.FANUEL KEHEZE

Introduction

Aims

- To familiarise students with
  - Combinational logic circuits
  - Sequential logic circuits
  - How digital logic gates are built using transistors
  - Design and build of digital logic systems

Course Structure

- 11 Lectures
- Hardware Labs
  - 6 Workshops
  - 7 sessions, each one 3h, alternate weeks
  - Thu. 10.00 or 2.00 start, beginning week 3
  - In Cockroft 4 (New Museum Site)
  - In groups of 2

Objectives

- At the end of the course you should
  - Be able to design and construct simple digital electronic systems
  - Be able to understand and apply Boolean logic and algebra – a core competence in Computer Science
  - Be able to understand and build state machines

Books

- Lots of books on digital electronics, e.g.,
  - D. M. Harris and S. L. Harris, ‘Digital Design and Computer Architecture,’ Morgan Kaufmann, 2007.
  - R. H. Katz, ‘Contemporary Logic Design,’ Benjamin/Cummings, 1994.
  - J. P. Hayes, ‘Introduction to Digital Logic Design,’ Addison-Wesley, 1993.
- Electronics in general (inc. digital)
  - P. Horowitz and W. Hill, ‘The Art of Electronics,’ CUP, 1989.

Other Points

- This course is a prerequisite for
  - ECAD (Part IB)
  - VLSI Design (Part II)
- Keep up with lab work and get it ticked.
- Have a go at supervision questions plus any others your supervisor sets.
- Remember to try questions from past papers

Semiconductors to Computers

- Increasing levels of complexity
  - Transistors built from semiconductors
  - Logic gates built from transistors
  - Logic functions built from gates
  - Flip-flops built from logic
  - Counters and sequencers from flip-flops
  - Microprocessors from sequencers
  - Computers from microprocessors

Semiconductors to Computers

- Increasing levels of abstraction:
  - Physics
  - ***Transistors***
  - ***Gates***
  - ***Logic***
  - Microprogramming (Computer Design Course)
  - Assembler (Computer Design Course)
  - Programming Languages (Compilers Course)
  - Applications

Combinational Logic

Introduction to Logic Gates

- We will introduce Boolean algebra and logic gates
- Logic gates are the building blocks of digital circuits

Logic Variables

- Different names for the same thing
  - Logic variables
  - Binary variables
  - Boolean variables
- Can only take on 2 values, e.g.,
  - TRUE or False
  - ON or OFF
  - 1 or 0

Logic Variables

- In electronic circuits the two values can be represented by e.g.,
  - High voltage for a 1
  - Low voltage for a 0
- Note that since only 2 voltage levels are used, the circuits have greater immunity to electrical noise

Uses of Simple Logic

- Example – Heating Boiler
  - If chimney is not blocked and the house is cold and the pilot light is lit, then open the main fuel valve to start boiler.

    b = chimney blocked

    c = house is cold

    p = pilot light lit

    v = open fuel valve
  - So in terms of a logical (Boolean) expression

    v = (NOT b) AND c AND p

Logic Gates

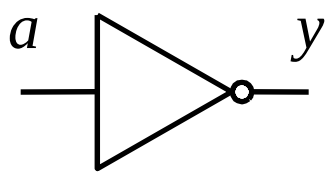
- Basic logic circuits with one or more inputs and one output are known as *gates*
- *Gates* are used as the building blocks in the design of more complex digital logic circuits

Representing Logic Functions

- There are several ways of representing logic functions:
  - Symbols to represent the gates
  - Truth tables
  - Boolean algebra
- We will now describe commonly used gates

NOT Gate

Symbol



Truth-table

a	y
0	1
1	0

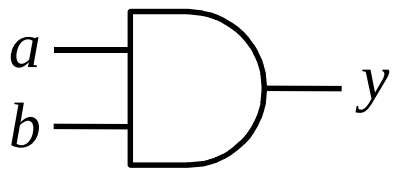
Boolean

$$y = \overline{a}$$

- A NOT gate is also called an ‘inverter’
- y is only TRUE if a is FALSE
- Circle (or ‘bubble’) on the output of a gate implies that it as an inverting (or complemented) output

AND Gate

Symbol



Truth-table

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

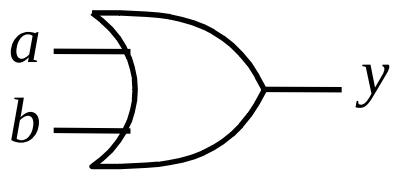
Boolean

$$y = a.b$$

- y is only TRUE only if a is TRUE and b is TRUE
- In Boolean algebra AND is represented by a dot .

OR Gate

Symbol



Truth-table

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

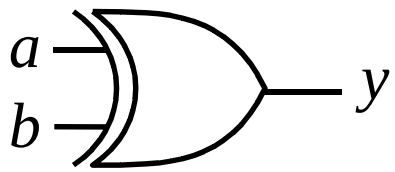
Boolean

$$y = a + b$$

- 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 (XOR) Gate

Symbol



Truth-table

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

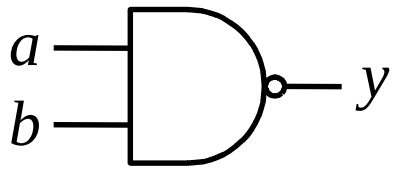
Boolean

$$y = a \oplus b$$

- y is TRUE if a is TRUE or b is TRUE (but not both)
- In Boolean algebra XOR is represented by an $\oplus$ sign

NOT AND (NAND) Gate

Symbol



Truth-table

a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

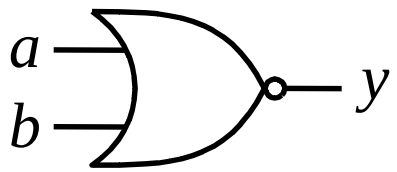
Boolean

$$y = \overline{a.b}$$

- y is TRUE if a is FALSE or b is FALSE (or both)
- y is FALSE only if a is TRUE and b is TRUE

NOT OR (NOR) Gate

Symbol



Truth-table

a	b	y
0	0	1
0	1	0
1	0	0
1	1	0

Boolean

$$y = \overline{a + b}$$

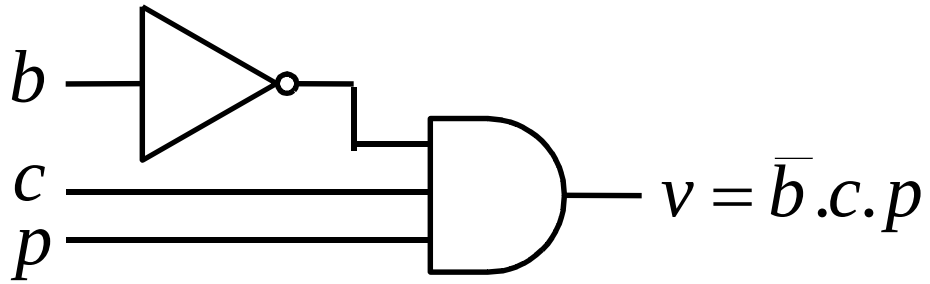
- y is TRUE only if a is FALSE and b is FALSE
- y is FALSE if a is TRUE or b is TRUE (or both)

Boiler Example

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

b = chimney blocked c = house is cold

p = pilot light lit v = open fuel valve



Boolean Algebra

- In this section we will introduce the laws of Boolean Algebra
- We will then see how it can be used to design *combinational logic* circuits
- Combinational logic circuits do not have an internal stored state, i.e., they have no memory. Consequently the output is solely a function of the current inputs.
- Later, we will study circuits having a stored internal state, i.e., sequential logic circuits.

Boolean Algebra

OR	AND
$a+0=a$	$a.0=0$
$a+a=a$	$a.a=a$
$a+1=1$	$a.1=a$
$a+\bar{a}=1$	$a.\bar{a}=0$

- AND takes precedence over OR, e.g.,

  $a.b+c.d=(a.b)+(c.d)$

Boolean Algebra

- Commutation

  $a + b = b + a$

  $a.b = b.a$

- Association

  $(a + b) + c = a + (b + c)$

  $(a.b).c = a.(b.c)$

- Distribution

  $a.(b + c + \ldots) = (a.b) + (a.c) + \ldots$

  $a + (b.c.\ldots) = (a + b).(a + c).\ldots$ NEW

- Absorption

  $a + (a.c) = a$ NEW

  $a.(a + c) = a$ NEW

Boolean Algebra - Examples

Show

$$a.(\overline{a}+b)=a.b$$

$$a.(\overline{a}+b)=a.\overline{a}+a.b=0+a.b=a.b$$

Show

$$a+(\overline{a}.b)=a+b$$

$$a+(\overline{a}.b)=(a+\overline{a}).(a+b)=1.(a+b)=a+b$$

Boolean Algebra

- 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 cancelling terms in this expanded form
  e.g., to prove the absorption rule:

$$a + a.b = a$$

$$a.b + a.\bar{b} + \cancel{a.b} = a.b + a.\bar{b} = a.(b + \bar{b}) = a.1 = a$$

Boolean Algebra - Example

Simplify

$$x.y + \bar{y}.z + x.z + x.y.z$$

$$x.y.z + x.y.\bar{z} + x.\bar{y}.z + \bar{x}.\bar{y}.z + x.y.z + x.\bar{y}.z + x.y.z$$

$$x.y.z + x.y.\bar{z} + x.\bar{y}.z + \bar{x}.\bar{y}.z$$

$$x.y.(z + \bar{z}) + \bar{y}.z.(x + \bar{x})$$

$$x.y.1 + \bar{y}.z.1$$

$$x.y + \bar{y}.z$$

DeMorgan's Theorem

$$\overline{a+b+c+\ \ldots} = \bar{a}.\bar{b}.\bar{c}.\ \ldots$$

$$\overline{a.b.c.\ \ldots} = \bar{a}+\bar{b}+\bar{c}+\ \ldots$$

- In a simple expression like $a+b+c$ (or $a.b.c$) simply change all operators from OR to AND (or vice versa), complement each term (put a bar over it) and then complement the whole expression, i.e.,

$$a+b+c+\ \ldots = \overline{\bar{a}.\bar{b}.\bar{c}.\ \ldots}$$

$$a.b.c.\ \ldots = \overline{\bar{a}+\bar{b}+\bar{c}+\ \ldots}$$

DeMorgan's Theorem

- For 2 variables we can show $\overline{a+b} = \overline{a}.\overline{b}$ 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 by induction

$$\overline{a+b+c} = \overline{(a+b)}.\overline{c} = (\overline{a}.\overline{b}).\overline{c} = \overline{a}.\overline{b}.\overline{c}$$

DeMorgan's Examples

- Simplify $a.\overline{b} + a.(\overline{b+c}) + b.(\overline{b+c})$

$= a.\overline{b} + a.\overline{b}.\overline{c} + b.\overline{b}.\overline{c}$ (DeMorgan)

$= a.\overline{b} + a.\overline{b}.\overline{c}$ $(\mathrm{b}.\overline{\mathrm{b}} = 0)$

$= a.\overline{b}$ (absorbtion)

DeMorgan's Examples

- Simplify $(a.b.(c+\overline{b.d})+\overline{a.b}).c.d$

$= (a.b.(c+\overline{b}+\overline{d})+\overline{a}+\overline{b}).c.d$ (De Morgan)

$= (a.b.c+a.b.\overline{b}+a.b.\overline{d}+\overline{a}+\overline{b}).c.d$ (distribute)

$= (a.b.c+a.b.\overline{d}+\overline{a}+\overline{b}).c.d$ $(a.b.\overline{b}=0)$

$= a.b.c.d+a.b.\overline{d}.c.d+\overline{a}.c.d+\overline{b}.c.d$ (distribute)

$= a.b.c.d+\overline{a}.c.d+\overline{b}.c.d$ $(a.b.\overline{d}.c.d=0)$

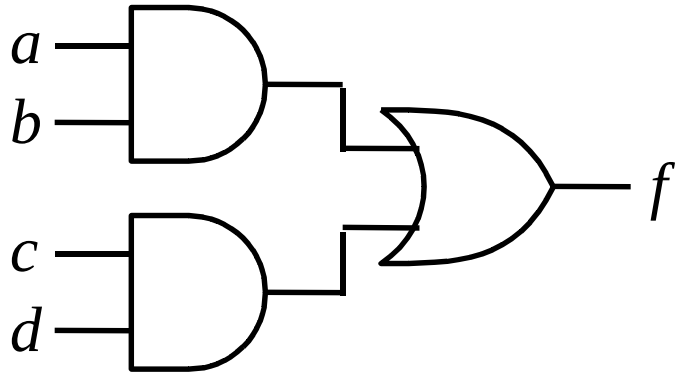
$= (a.b+\overline{a}+\overline{b}).c.d$ (distribute)

$= (a.b+\overline{a.b}).c.d$ (DeMorgan)

$= c.d$ $(a.b+\overline{a.b}=1)$

DeMorgan's in Gates

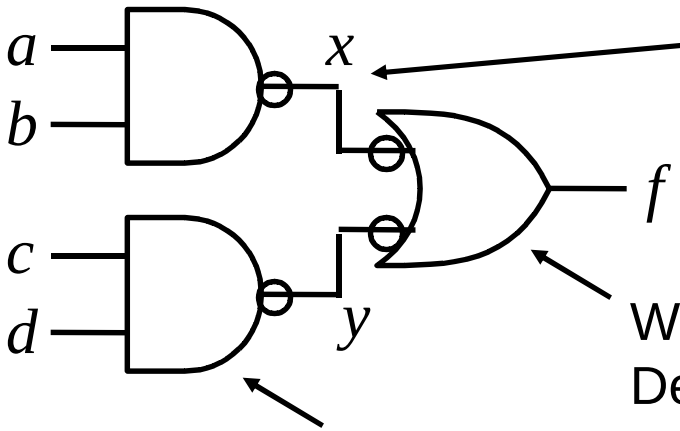
- To implement the function $f = a.b + c.d$ we can use AND and OR gates



- However, sometimes we only wish to use NAND or NOR gates, since they are usually simpler and faster

DeMorgan's in Gates

- To do this we can use 'bubble' logic



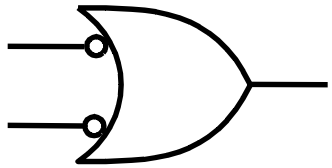
Two consecutive 'bubble' (or complement) operations cancel, i.e., no effect on logic function

What about this gate?
DeMorgan says $\overline{x} + \overline{y} = \overline{x.y}$

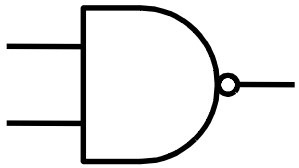
See AND gates are now NAND gates

Which is a NOT AND (NAND) gate

So

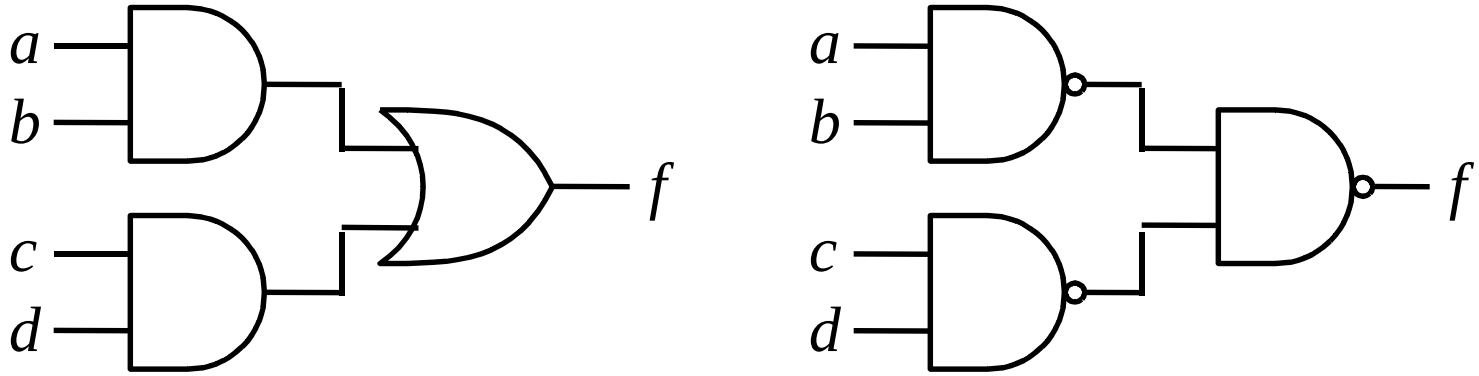


is equivalent to



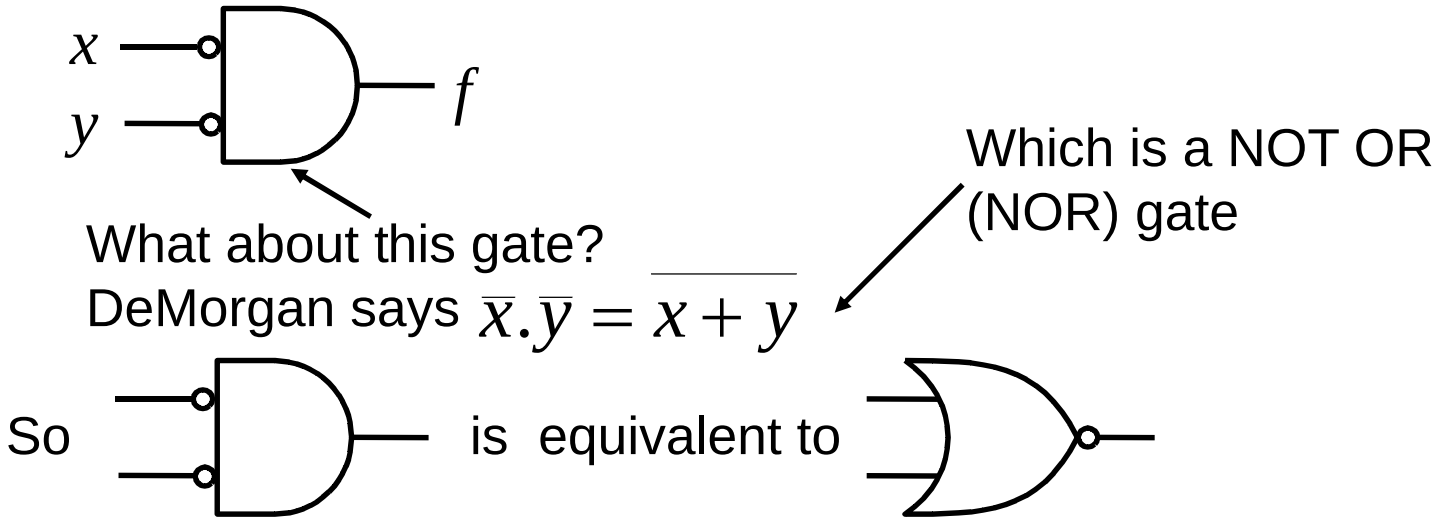
DeMorgan's in Gates

- So the previous function can be built using 3 NAND gates



DeMorgan's in Gates

- Similarly, applying 'bubbles' to the input of an AND gate yields



- Useful if trying to build using NOR gates

Logic Minimisation

- Any Boolean function can be implemented directly using combinational logic (gates)
- However, simplifying the Boolean function will enable the number of gates required to be reduced. Techniques available include:
  - Algebraic manipulation (as seen in examples)
  - Karnaugh (K) mapping (a visual approach)
  - Tabular approaches (usually implemented by computer, e.g., Quine-McCluskey)
- K mapping is the preferred technique for up to about 5 variables

Truth Tables

- f is defined by the following truth table

x	y	z	f	minterms
0	0	0	1	$\bar{x}.\bar{y}.\bar{z}$
0	0	1	1	$\bar{x}.\bar{y}.z$
0	1	0	1	$\bar{x}.y.\bar{z}$
0	1	1	1	$\bar{x}.y.z$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$x.y.z$

- A *minterm* must contain all variables (in either complement or uncomplemented form)
  - Note variables in a minterm are ANDed together (conjunction)
  - One minterm for each term of f that is TRUE

- So $\bar{x}.y.z$ is a minterm but $y.z$ is not

Disjunctive Normal Form

- A Boolean function expressed as the disjunction (ORing) of its minterms is said to be in the Disjunctive Normal Form (DNF)

$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.\bar{y}.z + \bar{x}.y.\bar{z} + \bar{x}.y.z + x.y.z$$

- A Boolean function expressed as the ORing of ANDed variables (not necessarily minterms) is often said to be in Sum of Products (SOP) form, e.g.,

$$f = \bar{x} + y.z$$ Note functions have the same truth table

Maxterms

- A maxterm of n Boolean variables is the disjunction (ORing) of all the variables either in complemented or uncomplemented form.
  - Referring back to the truth table for f, we can write,

$$\bar{f} = x.\bar{y}.\bar{z} + x.\bar{y}.z + x.y.\bar{z}$$

Applying De Morgan (and complementing) gives

$$f = (\bar{x} + y + z).(\bar{x} + y + \bar{z}).(\bar{x} + \bar{y} + z)$$

So it can be seen that the maxterms of f are effectively the minterms of $\bar{f}$ with each variable complemented

Conjunctive Normal Form

- A Boolean function expressed as the conjunction (ANDing) of its maxterms is said to be in the Conjunctive Normal Form (CNF)

$$f = (\bar{x} + y + z).(\bar{x} + y + \bar{z}).(\bar{x} + \bar{y} + z)$$

- A Boolean function expressed as the ANDing of ORed variables (not necessarily maxterms) is often said to be in Product of Sums (POS) form, e.g.,

$$f = (\bar{x} + y).(\bar{x} + z)$$

Logic Simplification

- As we have seen previously, Boolean algebra can be used to simplify logical expressions. This results in easier implementation

  Note: The DNF and CNF forms are not simplified.

- However, it is often easier to use a technique known as Karnaugh mapping

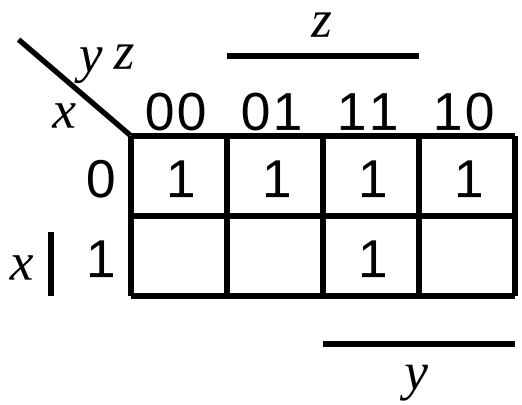
Karnaugh Maps

- Karnaugh Maps (or K-maps) are a powerful visual tool for carrying out simplification and manipulation of logical expressions having up to 5 variables
- The K-map is a rectangular array of cells
  - Each possible state of the input variables corresponds uniquely to one of the cells
  - The corresponding output state is written in each cell

K-maps example

- From truth table to K-map

x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

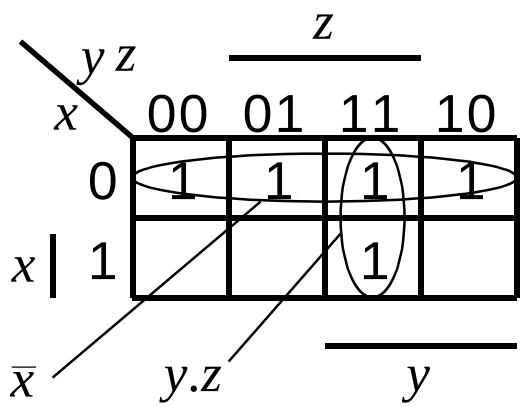


Note that the logical state of the variables follows a Gray code, i.e., only one of them changes at a time

The exact assignment of variables in terms of their position on the map is not important

K-maps example

- Having plotted the minterms, how do we use the map to give a simplified expression?



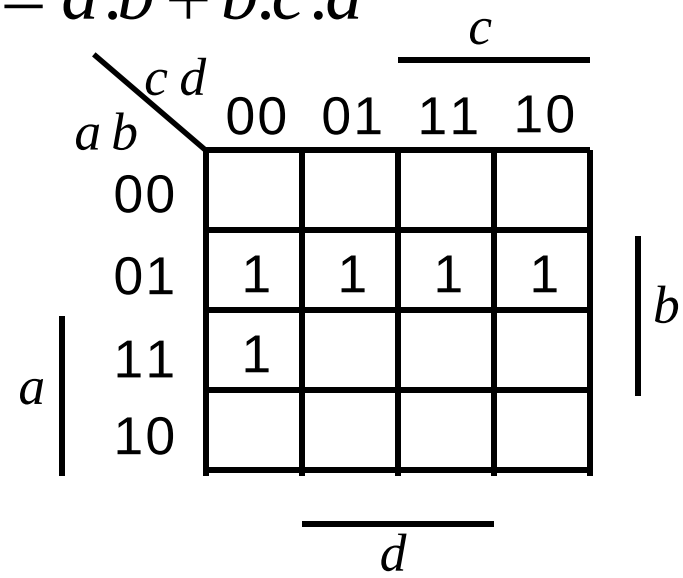
- Group terms
  - Having size equal to a power of 2, e.g., 2, 4, 8, etc.
  - Large groups best since they contain fewer variables
  - Groups can wrap around edges and corners

So, the simplified func. is,

$f = \bar{x} + y.z$ as before

K-maps – 4 variables

- K maps from Boolean expressions
  - Plot $f = \bar{a}.b + b.\bar{c}.\bar{d}$

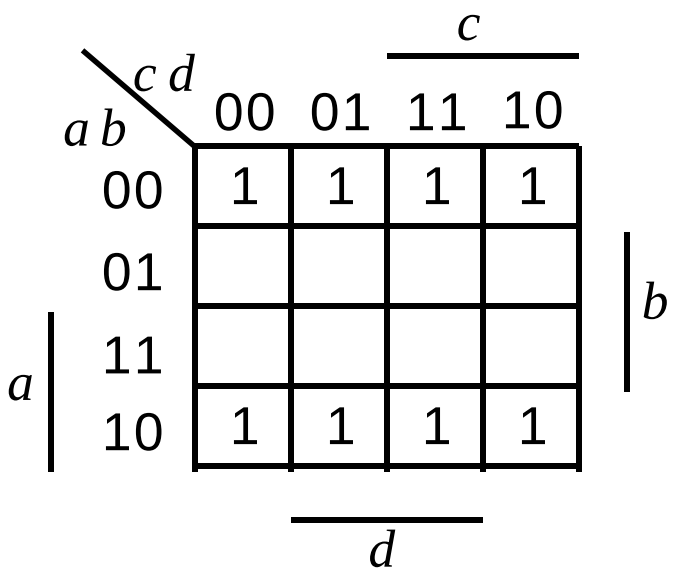


- See in a 4 variable map:
  - 1 variable term occupies 8 cells
  - 2 variable terms occupy 4 cells
  - 3 variable terms occupy 2 cells, etc.

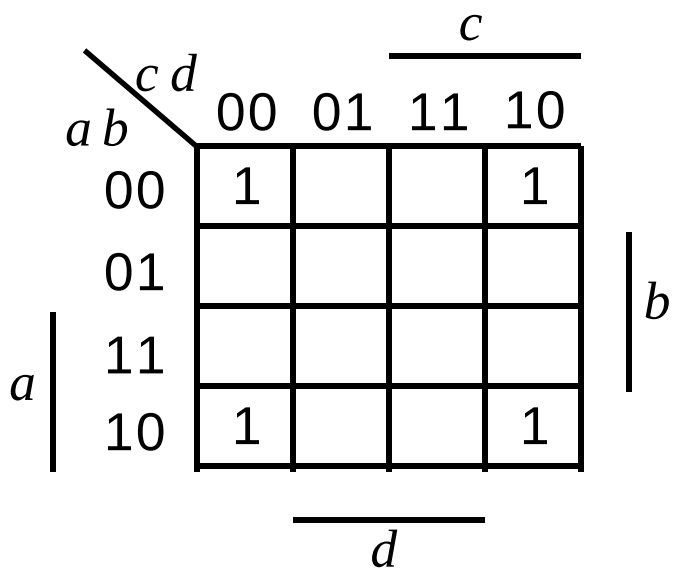
K-maps – 4 variables

- For example, plot

$f = \overline{b}$

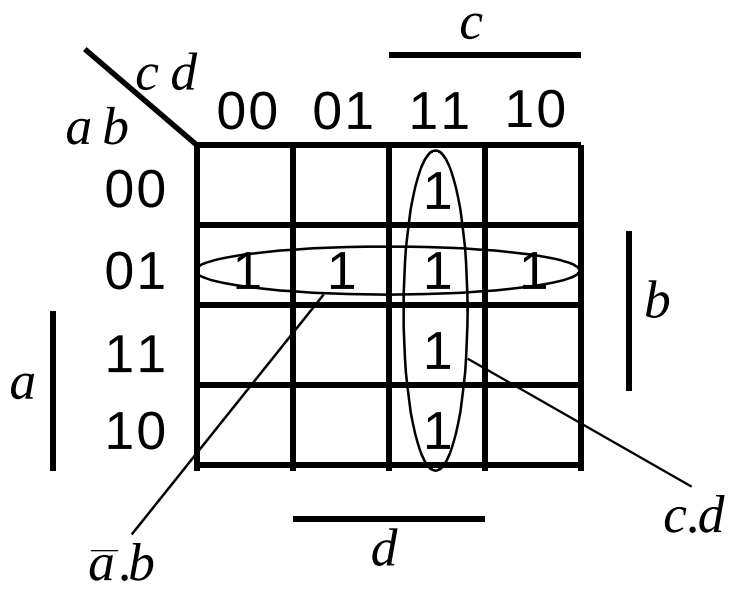


$f = \overline{b}.\overline{d}$



K-maps – 4 variables

- Simplify, $f = \bar{a}.b.\bar{d} + b.c.d + \bar{a}.b.\bar{c}.d + c.d$



So, the simplified func. is,

$$f = \bar{a}.b + c.d$$

POS Simplification

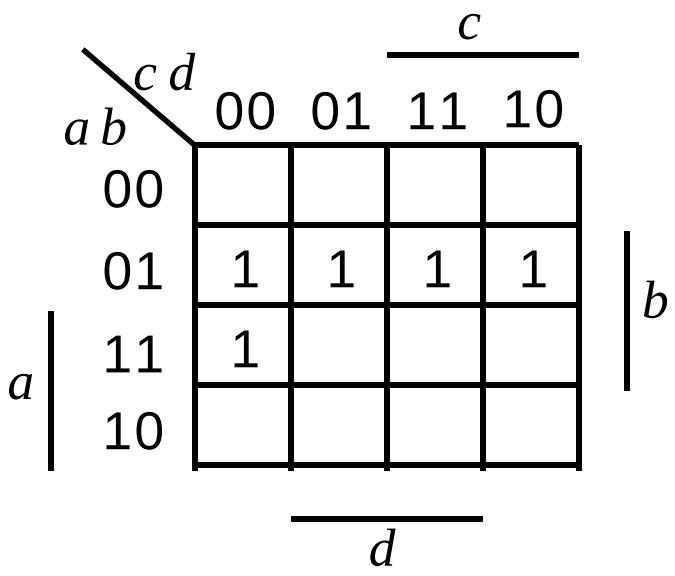
- Note that the previous examples have yielded simplified expressions in the SOP form
  - Suitable for implementations using AND followed by OR gates, or only NAND gates (using DeMorgans to transform the result – see previous Bubble logic slides)
- However, sometimes we may wish to get a simplified expression in POS form
  - Suitable for implementations using OR followed by AND gates, or only NOR gates

POS Simplification

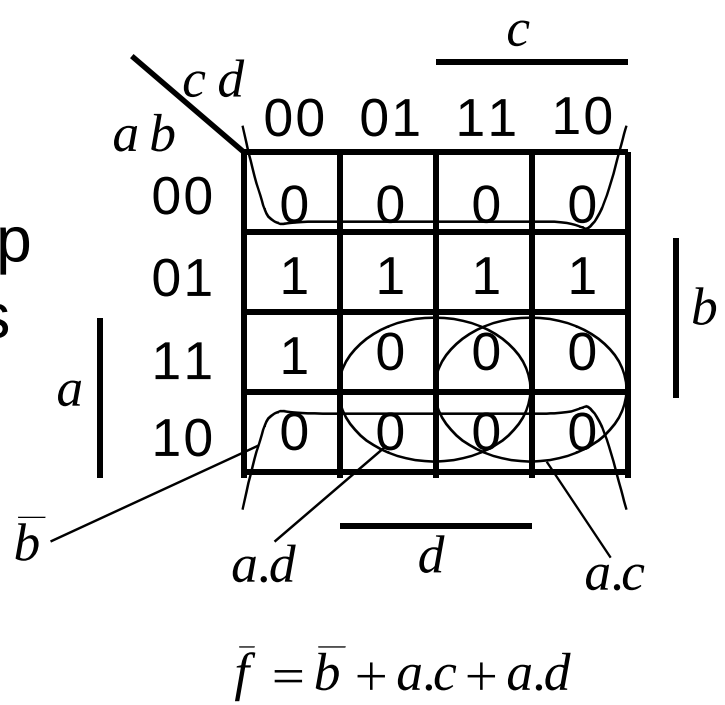
- To do this we group the zeros in the map
  - i.e., we simplify the complement of the function
- Then we apply DeMorgans and complement
- Use ‘bubble’ logic if NOR only implementation is required

POS Example

- Simplify $f = \bar{a}.b + b.\bar{c}.\bar{d}$ into POS form.



Group zeros



$$\bar{f} = \bar{b} + a.c + a.d$$

POS Example

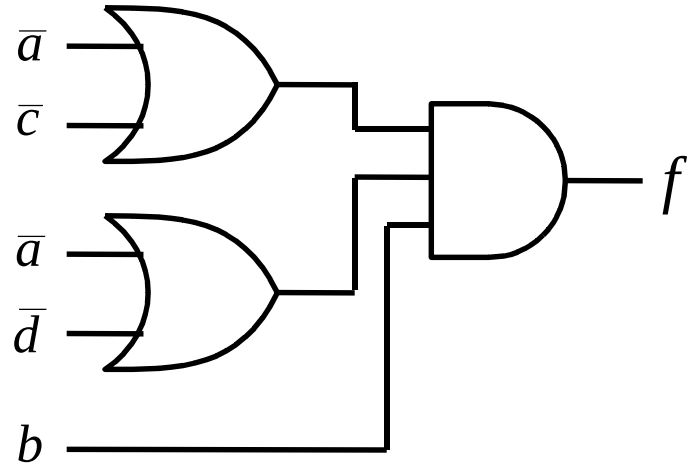
- Applying DeMorgans to

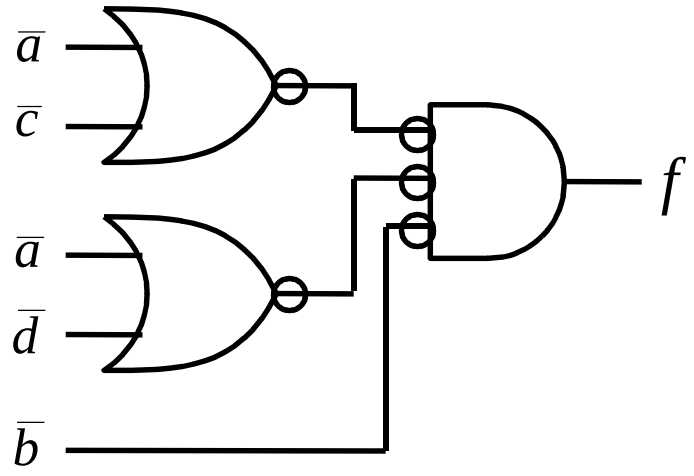
$$\bar{f} = \bar{b} + a.c + a.d$$

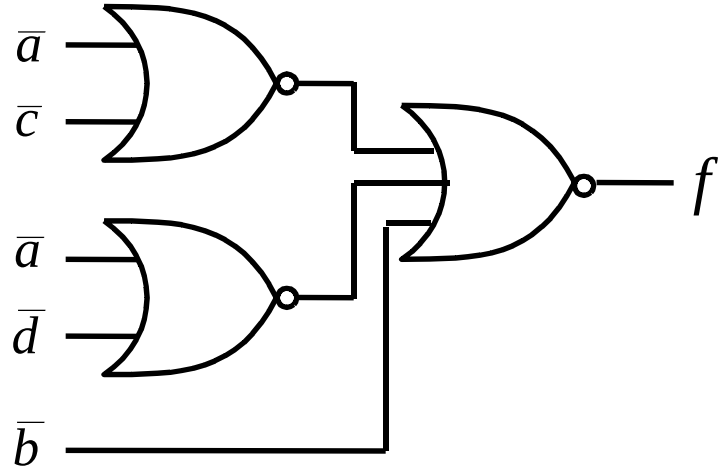
gives,

$$\bar{f} = \overline{b.(\bar{a} + \bar{c}).(\bar{a} + \bar{d})}$$

$$f = b.(\bar{a} + \bar{c}).(\bar{a} + \bar{d})$$







Expression in POS form

- Apply DeMorgans and take complement, i.e., $\bar{f}$ is now in SOP form
- Fill in zeros in table, i.e., plot $\bar{f}$
- Fill remaining cells with ones, i.e., plot f
- Simplify in usual way by grouping ones to simplify f

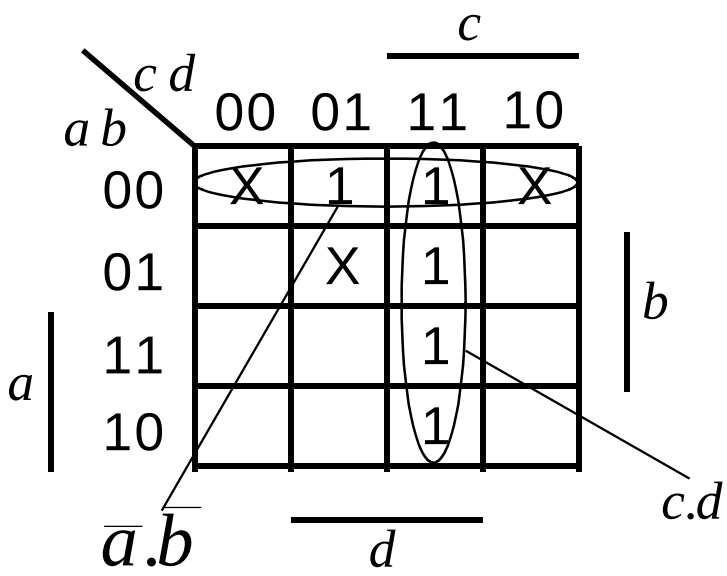
Don’t Care Conditions

- Sometimes we do not care about the output value of a combinational logic circuit, i.e., if certain input combinations can never occur, then these are known as *don’t care conditions*.
- In any simplification they may be treated as 0 or 1, depending upon which gives the simplest result.
  - For example, in a K-map they are entered as Xs

Don’t Care Conditions - Example

- Simplify the function $f = \bar{a}.\bar{b}.d + \bar{a}.c.d + a.c.d$

With don’t care conditions, $\bar{a}.\bar{b}.\bar{c}.\bar{d}$, $\bar{a}.\bar{b}.c.\bar{d}$, $\bar{a}.b.\bar{c}.d$



See only need to include Xs if they assist in making a bigger group, otherwise can ignore.

$f = \bar{a}.\bar{b} + c.d$ or, $f = \bar{a}.d + c.d$

Some Definitions

- Cover – A term is said to cover a minterm if that minterm is part of that term
- Prime Implicant – a term that cannot be further combined
- Essential Term – a prime implicant that covers a minterm that no other prime implicant covers
- Covering Set – a minimum set of prime implicants which includes all essential terms plus any other prime implicants required to cover all minterms

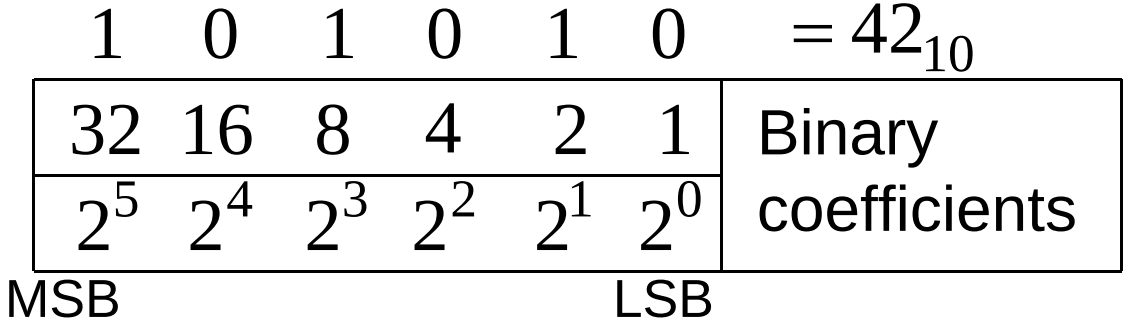
Number Representation, Addition and Subtraction

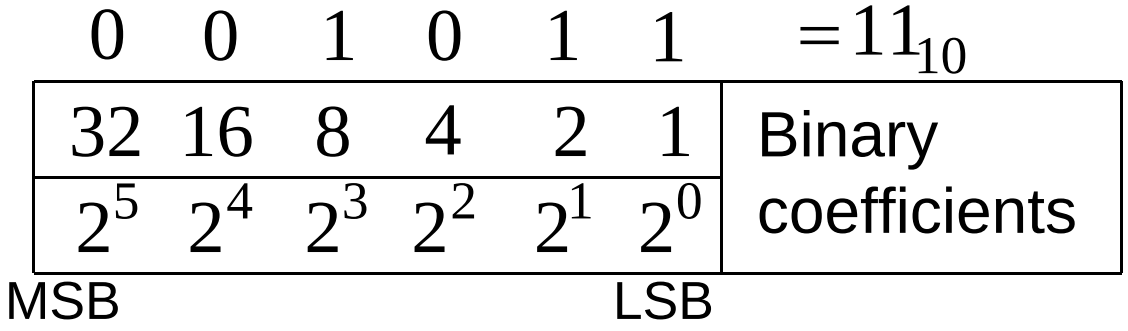
Binary Numbers

- It is important to be able to represent numbers in digital logic circuits
  - for example, the output of a analogue to digital converter (ADC) is an n-bit number, where n is typically in the range from 8 to 16
- Various representations are used, e.g.,
  - unsigned integers
  - 2's complement to represent negative numbers

Binary Numbers

- Binary is base 2. Each digit (known as a bit) is either 0 or 1.
- Consider these 6-bit unsigned numbers





MSB – most significant bit

LSB – least significant bit

Unsigned Binary Numbers

- In general, an n-bit binary number, $b_{n-1}b_{n-2}\ldots b_1 b_0$ has the decimal value,

$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

- So we can represent positive integers from 0 to $2^n - 1$
- In computers, binary numbers are often 8 bits long – known as a *byte*
- A byte can represent unsigned values from 0 to 255

Unsigned Binary Numbers

- Decimal to binary conversion. Perform successive division by 2.
  - Convert 42_{10} into binary

$$42/2 = 21 \quad \text{remainder} = 0$$

$$21/2 = 10 \quad \text{remainder} = 1$$

$$10/2 = 5 \quad \text{remainder} = 0$$

$$5/2 = 2 \quad \text{remainder} = 1$$

$$2/2 = 1 \quad \text{remainder} = 0$$

$$1/2 = 0 \quad \text{remainder} = 1$$

- So the answer is 101010_2 (reading upwards)

Octal: Base 8

- We have seen base 2 uses 2 digits (0 & 1), not surprisingly base 8 uses 8 digits : 0, 1, 2, 3, 4, 5, 6, 7.

0	5	2	$= 42_{10}$
64	8	1	Octal coefficients
8^2	8^1	8^0	
MSB		LSB	

- To convert from decimal to base 8 either use successive division, i.e.,

$$42/8 = 5 \quad \text{remainder} = 2$$

$$5/8 = 0 \quad \text{remainder} = 5$$

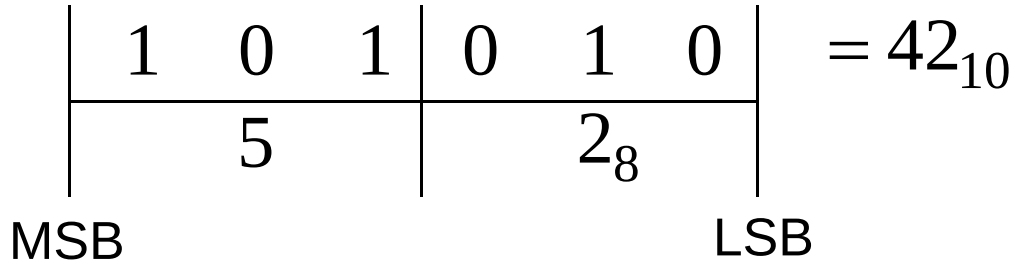
- So the answer is 52_8 (reading upwards)

Octal: Base 8

- Or alternatively, convert to binary, divide the binary number into 3-bit groups and work out the octal digit to represent each group. We have shown that

$$42_{10} = 101010_2$$

- So,



$= 42_{10}$

Hexadecimal: Base 16

- For base 16 we need 16 different digits. Consequently we need new symbols for the digits to represent 10-15

$1010_2 = 10_{10} = A_{16}$ $\quad$ $1101_2 = 13_{10} = D_{16}$

$1011_2 = 11_{10} = B_{16}$ $\quad$ $1110_2 = 14_{10} = E_{16}$

$1100_2 = 12_{10} = C_{16}$ $\quad$ $1111_2 = 15_{10} = F_{16}$

$0 \quad 2 \quad A_{16} = 42_{10}$

256	16	1	Hex coefficients
16^2	16^1	16^0	

MSB LSB

Hex: Base 16

- To convert from decimal to base 16 use either use successive division by 16, i.e.,

$$42/16 = 2 \quad \text{remainder} = \text{A}$$

$$2/16 = 0 \quad \text{remainder} = 2$$

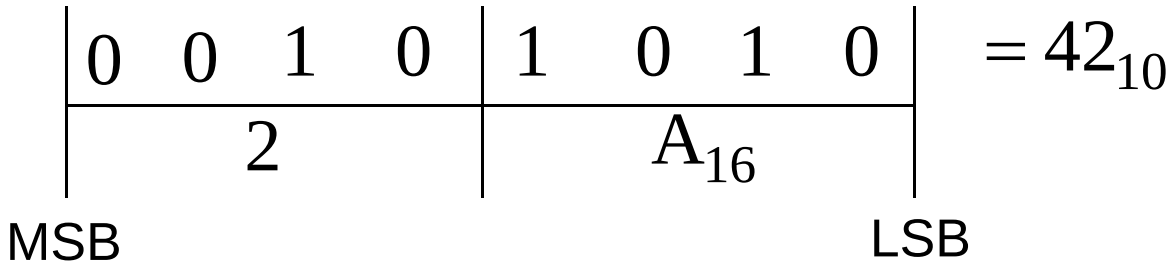
- So the answer is 2A_8 (reading upwards)

Hex: Base 16

- Or alternatively, convert to binary, divide the binary number into 4-bit groups and work out the hex digit to represent each group. We have shown that

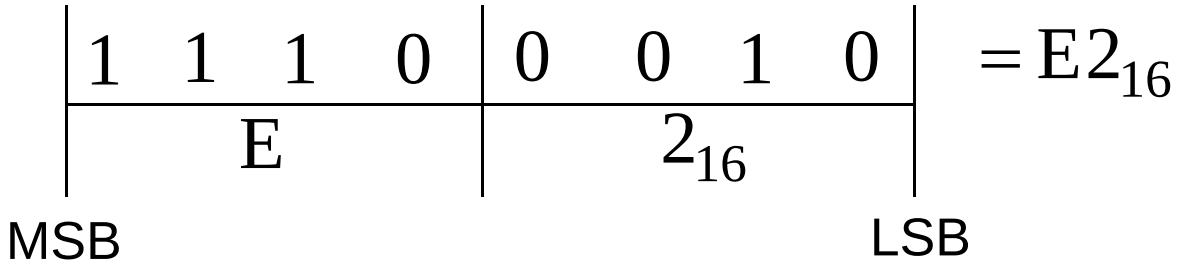
$$42_{10} = 101010_2$$

- So,



Hex: Base 16

- Hex is also used as a convenient way of representing the contents of a byte (an 8 bit number), so for example 11100010_2

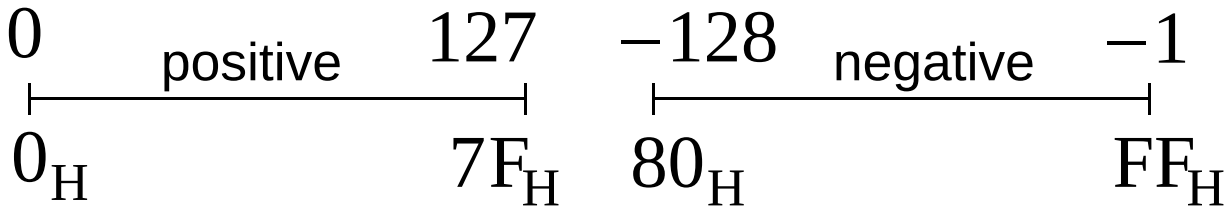


Negative numbers

- So far we have only been able to represent positive numbers. For example, we have seen an 8-bit byte can represent from 0 to 255, i.e., $2^8 = 256$ different combinations of bits in a byte
- If we want to represent negative numbers, we have to give up some of the range of positive numbers we had before
  - A popular approach to do this is called *2's complement*

2’s Complement

- For 8-bit numbers:



- Note all negative numbers have the MSB set
- The rule for changing a positive 2’s complement number into a negative 2’s complement number (or vice versa) is:

Complement all the bits and add 1.

2's Complement

- What happens when we do this to an 8 bit binary number x ?
  - Invert all bits: $x \rightarrow (255 - x)$
  - Add 1: $x \rightarrow (256 - x)$
- Note: 256 (= 100_H) will not fit into an 8 bit byte. However if we ignore the 'overflow' bit, then $256 - x$ behaves just like $0 - x$
- That is, we can use normal binary arithmetic to manipulate the 2's complement of x and it will behave just like -x

2's Complement Addition

	0	0	0	0	0	1	1	1	7
	0	0	0	0	0	1	0	0	+4
(0)	0	0	0	0	1	0	1	1	11

- To subtract, negate the second number, then add:

	0	0	0	0	0	1	1	1	7
	1	1	1	1	1	0	0	1	+–7
(1)	0	0	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	1	9
	1	1	1	1	1	0	0	1	+–7
(1)	0	0	0	0	0	0	1	0	2

2’s Complement Addition

	0	0	0	0	0	1	0	0	4
	1	1	1	1	1	0	0	1	+−7
(0)	1	1	1	1	1	1	0	1	−3

	1	1	1	1	1	0	0	1	−7
	1	1	1	1	1	0	0	1	+−7
(1)	1	1	1	1	0	0	1	0	−14

2's Complement

- Note that for an n-bit number $b_{n-1}b_{n-2}\ldots b_1b_0$, the decimal equivalent of a 2's complement number is,

$$= -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$$

- For example, $1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0$

$$= -b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$$

$$= -1\times 2^7 + 1\times 2^6 + 1\times 2^5 + 1\times 2^4 + 1\times 2^1$$

$$= -128 + 64 + 32 + 16 + 2 = -14$$

2’s Complement Overflow

- For example, when working with 8-bit unsigned numbers, we can use the ‘carry’ from the 8th bit (MSB) to indicate that the number has got too big.
- With *signed* numbers we deliberately ignore any carry from the MSB, consequently we need a new rule to detect when a result is out of range.

2’s Complement Overflow

- The rule for detecting 2’s complement overflow is:
  - The carry into the MSB **does not equal** the carry out from the MSB.
- We will now give some examples.

2’s Complement Overflow

	0	0	0	0	1	1	1	1	15
	0	0	0	0	1	1	1	1	+15
(0)	0	0	0	1	1	1	1	0	30

OK

	0	1	1	1	1	1	1	1	127
	0	0	0	0	0	0	0	1	+1
(0)	1	0	0	0	0	0	0	0	−128

overflow

2’s Complement Overflow

	1	1	1	1	0	0	0	1	-15	
	1	1	1	1	0	0	0	1	$+-15$	
(1)	1	1	1	0	0	0	1	0	-30	OK

	1	0	0	0	0	0	0	1	-127	
	1	1	1	1	1	1	1	0	$+-2$	
(1)	0	1	1	1	1	1	1	1	127	overflow

Binary Coded Decimal (BCD)

- Each decimal digit of a number is coded as a 4 bit binary quantity
- It is sometimes used since it is easy to code and decode, however it is not an efficient way to store numbers.

$$1248_{10} = 0001\ 0010\ 0100\ 1000_{\text{BCD}}$$
$$1234_{10} = 0001\ 0010\ 0011\ 0100_{\text{BCD}}$$

Alphanumeric Character Codes

- ASCII: American Standard Code for Information Exchange:
  - Standard version is a 7 bit code with the remaining bit usually set to zero
  - The first 32 are ‘control codes’ originally used for controlling modems
  - The rest are upper and lower case letters, numbers and punctuation.
  - An extended version uses all 8 bits to provide additional graphics characters

Alphanumeric Character Codes

- EBCDIC – a legacy IBM scheme, now little used
- Unicode – a 16 bit scheme, includes Chinese characters etc.

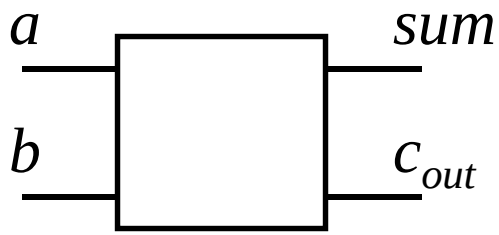
Binary Adding Circuits

- We will now look at how binary addition may be implemented using combinational logic circuits. We will consider:
  - Half adder
  - Full adder
  - Ripple carry adder

Half Adder

- Adds together two, single bit binary numbers a and b (note: no carry input)
- Has the following truth table:

a	b	c_{out}	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



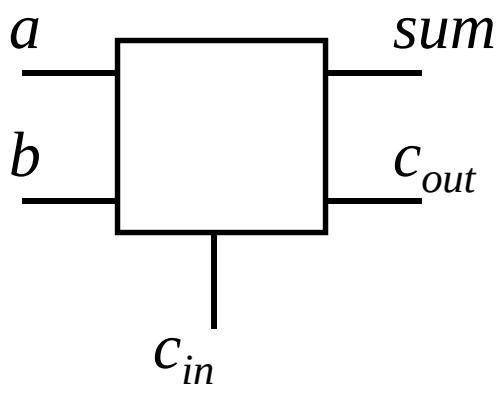
- By inspection:

$$sum = \overline{a}.b + a.\overline{b} = a \oplus b$$

$$c_{out} = a.b$$

Full Adder

- Adds together two, single bit binary numbers a and b (note: with a carry input)



- Has the following truth table:

Full Adder

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$sum = \overline{c}_{in}.\overline{a}.b + \overline{c}_{in}.a.\overline{b} + c_{in}.\overline{a}.\overline{b} + c_{in}.a.b$$

$$sum = \overline{c}_{in}.(\overline{a}.b + a.\overline{b}) + c_{in}.(\overline{a}.\overline{b} + a.b)$$

From DeMorgan

$$\overline{a}.\overline{b} + a.b = \overline{(a+b).(\overline{a}+\overline{b})}$$

$$= \overline{(a.\overline{a} + a.\overline{b} + b.\overline{a} + b.\overline{b})}$$

$$= \overline{(a.\overline{b} + b.\overline{a})}$$

So,

$$sum = \overline{c}_{in}.(\overline{a}.b + a.\overline{b}) + c_{in}.\overline{(\overline{a}.b + a.\overline{b})}$$

$$sum = \overline{c}_{in}.x + c_{in}.\overline{x} = c_{in} \oplus x = c_{in} \oplus a \oplus b$$

Full Adder

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{out} = \bar{c}_{in}.a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b} + c_{in}.a.b$$

$$c_{out} = a.b.(\bar{c}_{in} + c_{in}) + c_{in}.\bar{a}.b + c_{in}.a.\bar{b}$$

$$c_{out} = a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b}$$

$$c_{out} = a.(b + c_{in}.\bar{b}) + c_{in}.\bar{a}.b$$

$$c_{out} = a.(b + c_{in}).(b + \bar{b}) + c_{in}.\bar{a}.b$$

$$c_{out} = b.(a + c_{in}.\bar{a}) + a.c_{in} = b.(a + c_{in}).(a + \bar{a}) + a.c_{in}$$

$$c_{out} = b.a + b.c_{in} + a.c_{in}$$

$$c_{out} = b.a + c_{in}.(b + a)$$

Full Adder

- Alternatively,

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{out} = \overline{c}_{in}.a.b + c_{in}.\overline{a}.b + c_{in}.a.\overline{b} + c_{in}.a.b$$

$$c_{out} = c_{in}.(\overline{a}.b + a.\overline{b}) + a.b.(c_{in} + \overline{c}_{in})$$

$$c_{out} = c_{in}.(a \oplus b) + a.b$$

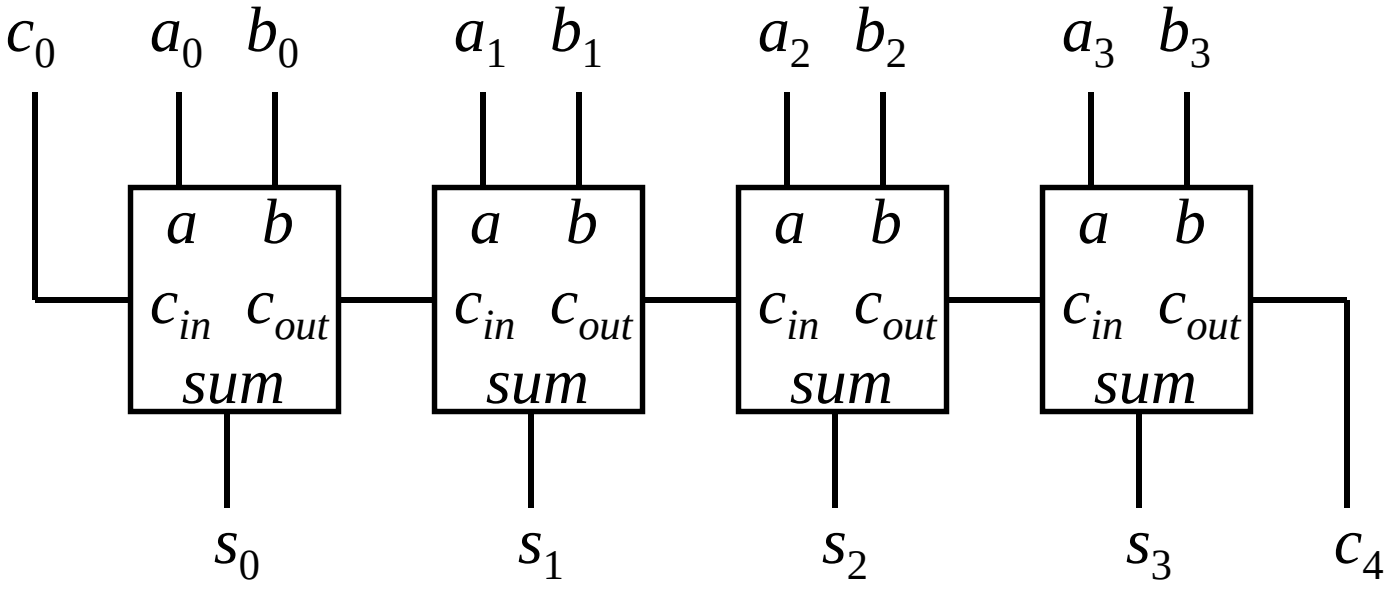
- Which is similar to previous expression except with the OR replaced by XOR

Ripple Carry Adder

- We have seen how we can implement a logic to add two, one bit binary numbers (inc. carry-in).
- However, in general we need to add together two, n bit binary numbers.
- One possible solution is known as the Ripple Carry Adder
  - This is simply n, full adders cascaded together

Ripple Carry Adder

- Example, 4 bit adder



- Note: If we complement a and set c_o to one we have implemented $s = b - a$

Combinational Logic Design

Further Considerations

Multilevel Logic

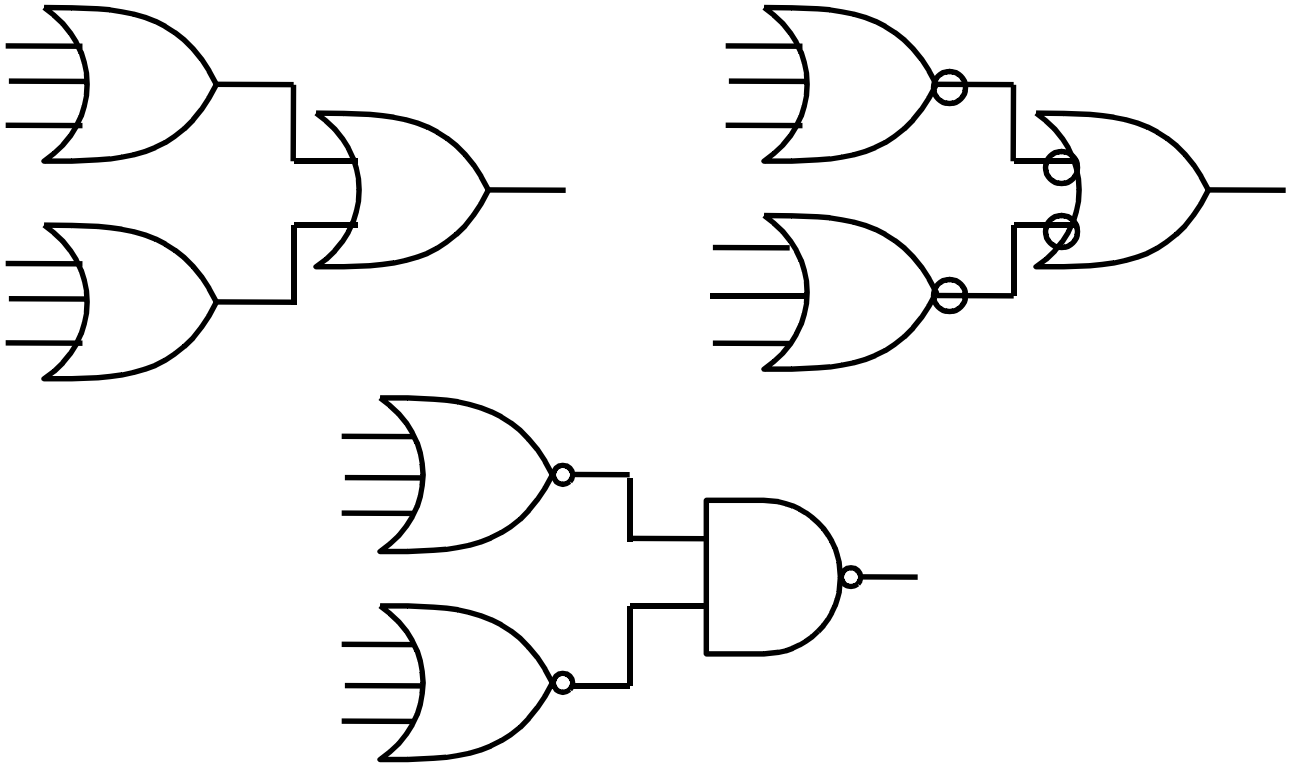
- We have seen previously how we can minimise Boolean expressions to yield so called ‘2-level’ logic implementations, i.e., SOP (ANDed terms ORed together) or POS (ORed terms ANDed together)
- Note also we have also seen an example of ‘multilevel’ logic, i.e., full adders cascaded to form a ripple carry adder – see we have more than 2 gates in cascade in the carry chain

Multilevel Logic

- Why use multilevel logic?
  - Commercially available logic gates usually only available with a restricted number of inputs, typically, 2 or 3.
  - System composition from sub-systems reduces design complexity, e.g., a ripple adder made from full adders
  - Allows Boolean optimisation across multiple outputs, e.g., common sub-expression elimination

Building Larger Gates

- Building a 6-input OR gate



Common Expression Elimination

- Consider the following minimised SOP expression:

$$z = a.d.f + a.e.f + b.d.f + b.e.f + c.d.f + c.e.f + g$$

- Requires:
  - Six, 3 input AND gates, one 7-input OR gate – total 7 gates, 2-levels
  - 19 literals (the total number of times all variables appear)

Common Expression Elimination

- We can recursively factor out common literals

$$z = a.d.f + a.e.f + b.d.f + b.e.f + c.d.f + c.e.f + g$$

$$z = (a.d + a.e + b.d + b.e + c.d + c.e).f + g$$

$$z = ((a + b + c).d + (a + b + c).e).f + g$$

$$z = (a + b + c).(d + e).f + g$$

- Now express z as a number of equations in 2-level form:

$$x = a + b + c \qquad x = d + e \qquad z = x.y.f + g$$

- 4 gates, 9 literals, 3-levels

Gate Propagation Delay

- So, multilevel logic can produce reductions in implementation complexity. What is the downside?
- We need to remember that the logic gates are implemented using electronic components (essentially transistors) which have a finite switching speed.
- Consequently, there will be a finite delay before the output of a gate responds to a change in its inputs – *propagation delay*

Gate Propagation Delay

- The cumulative delay owing to a number of gates in cascade can increase the time before the output of a combinational logic circuit becomes valid
- For example, in the Ripple Carry Adder, the sum at its output will not be valid until any carry has ‘rippled’ through possibly every full adder in the chain – clearly the MSB will experience the greatest potential delay

Gate Propagation Delay

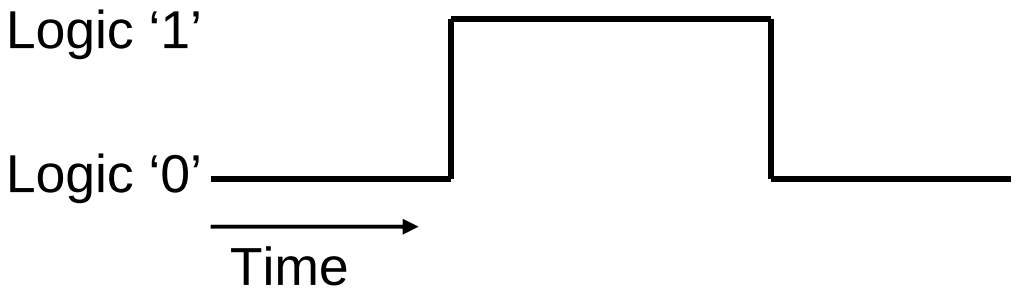
- As well as slowing down the operation of combinational logic circuits, gate delay can also give rise to so called '*Hazards*' at the output
- These *Hazards* manifest themselves as unwanted brief logic level changes (or *glitches*) at the output in response to changing inputs
- We will now describe how we can address these problems

Hazards

- Hazards are classified into two types, namely, static and dynamic
- Static Hazard – The output undergoes a momentary transition when it is supposed to remain unchanged
- Dynamic Hazard – The output changes more than once when it is supposed to change just once

Timing Diagrams

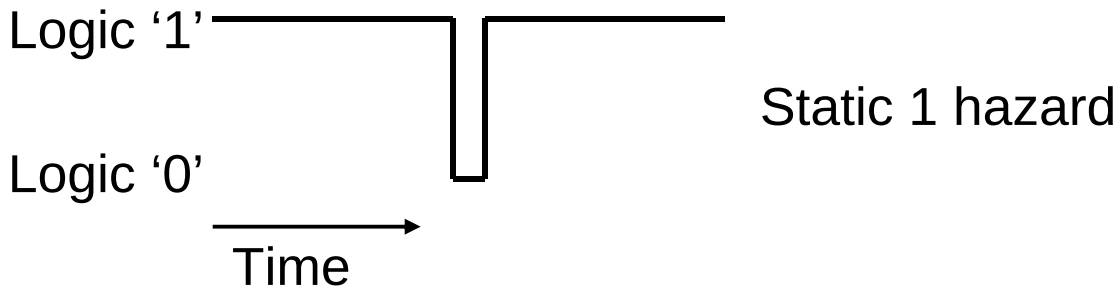
- To visually represent Hazards we will use the so called *‘timing diagram’*
- This shows the logical value of a signal as a function of time, for example the following timing diagram shows a transition from 0 to 1 and then back again



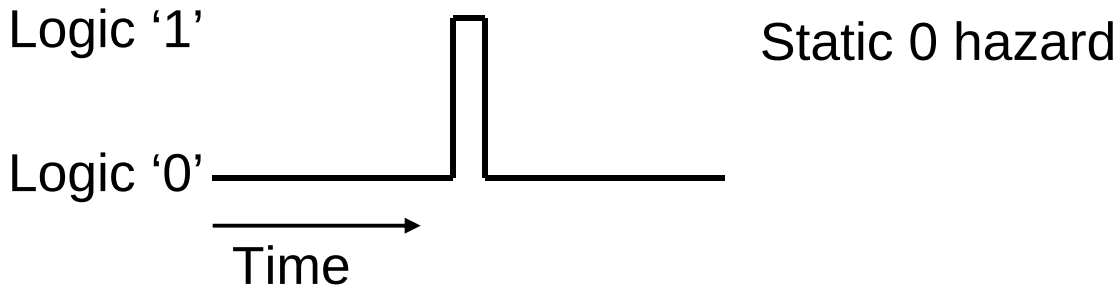
Timing Diagrams

- Note that the timing diagram makes a number simplifying assumptions (to aid clarity) compared with a diagram which accurately shows the actual voltage against time
  - The signal only has 2 levels. In reality the signal may well look more ‘wobbly’ owing to electrical noise pick-up etc.
  - The transitions between logic levels takes place instantaneously, in reality this will take a finite time.

Static Hazard

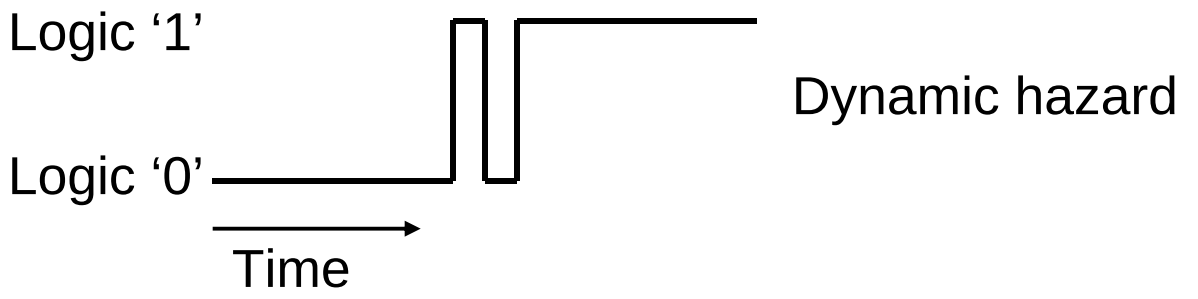


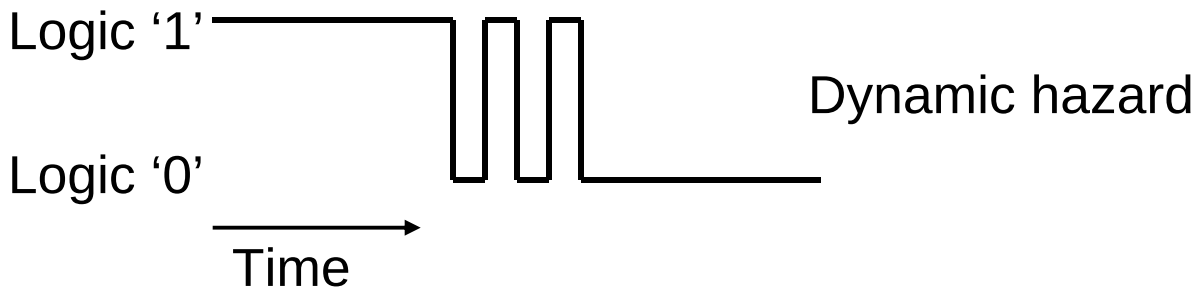
Static 1 hazard



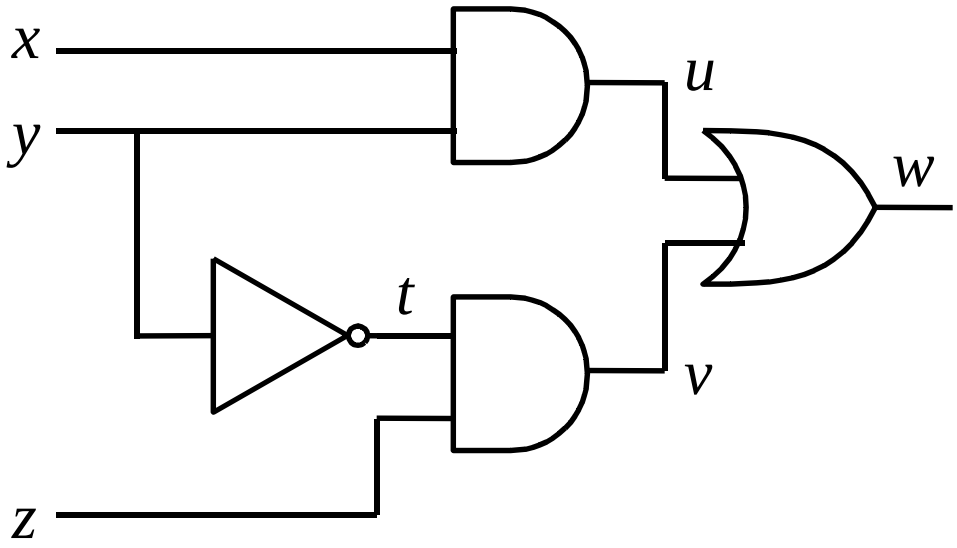
Static 0 hazard

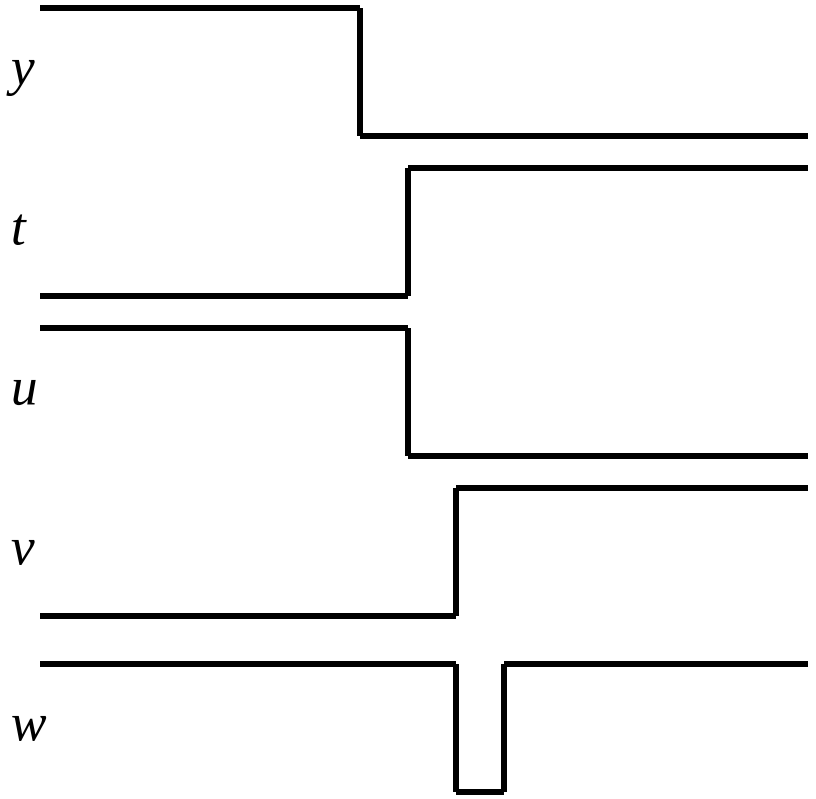
Dynamic Hazard





Static 1 Hazard





This circuit implements,

$$w = x.y + z.\bar{y}$$

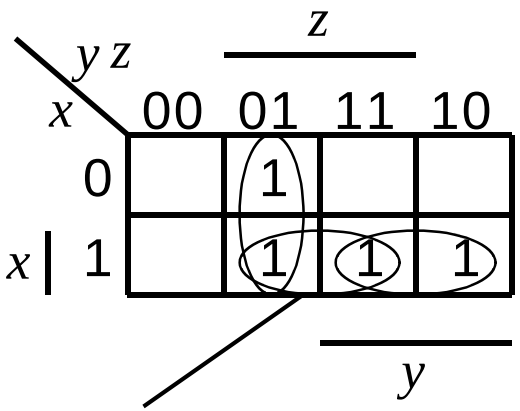
Consider the output when $z = x = 1$ and y changes from 1 to 0

Hazard Removal

- To remove a 1 hazard, draw the K-map of the output concerned. Add another term which overlaps the essential terms
- To remove a 0 hazard, draw the K-map of the complement of the output concerned. Add another term which overlaps the essential terms (representing the complement)
- To remove dynamic hazards – not covered in this course!

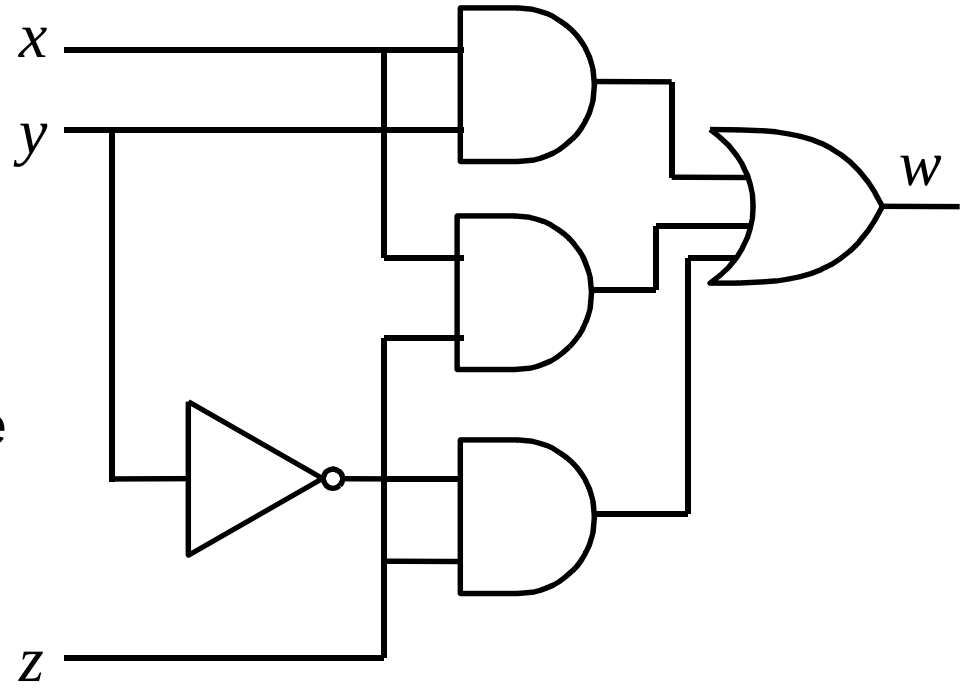
Removing the static 1 hazard

$$w = x.y + z.\bar{y}$$



Extra term added to remove hazard, consequently,

$$w = x.y + z.\bar{y} + x.z$$



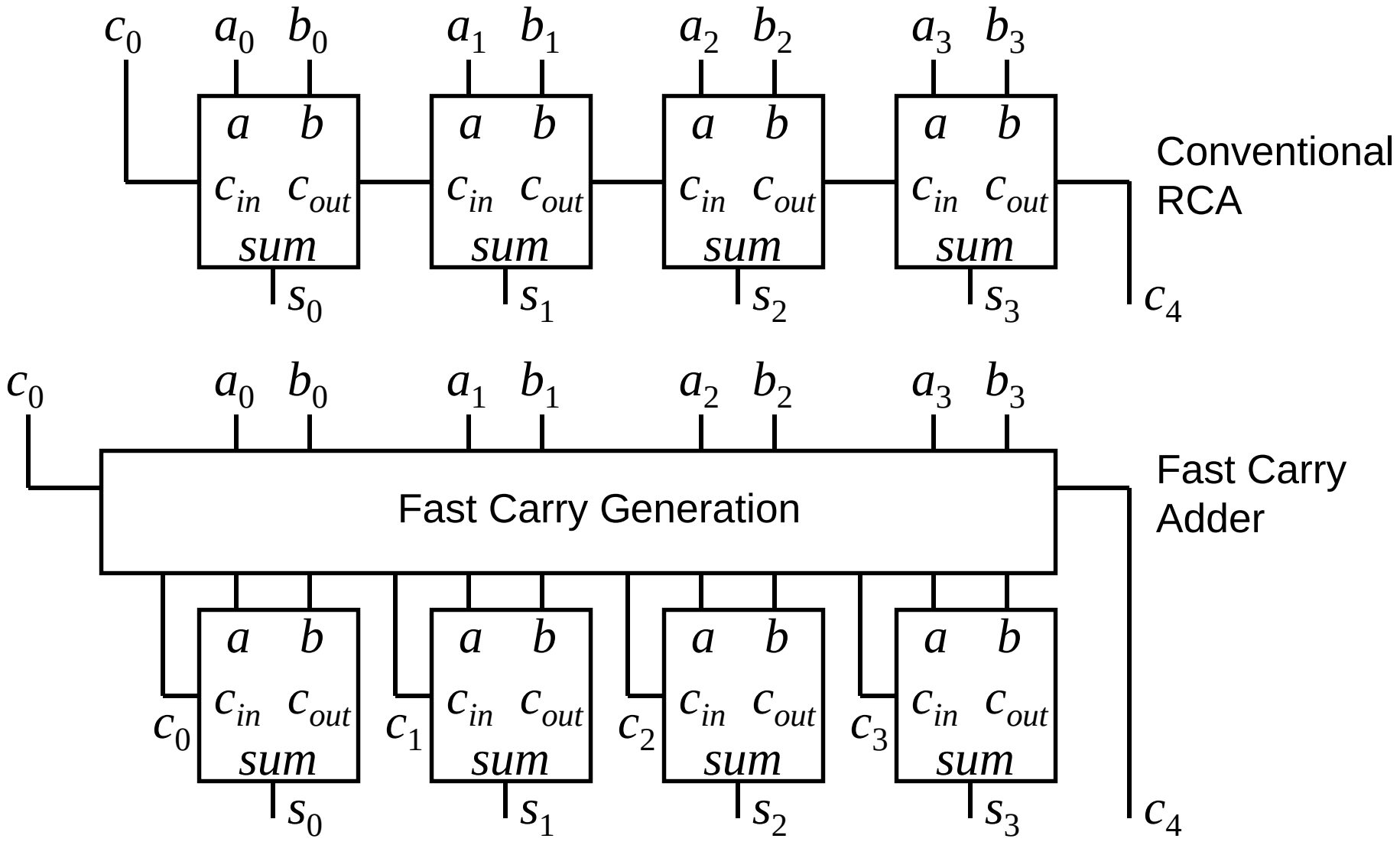
To Speed up Ripple Carry Adder

- Abandon compositional approach to the adder design, i.e., do not build the design up from full-adders, but instead design the adder as a block of 2-level combinational logic with $2n$ inputs (+1 for carry in) and n outputs (+1 for carry out).
- Features
  - Low delay (2 gate delays)
  - Need some gates with large numbers of inputs (which are not available)
  - Very complex to design and implement (imagine the truth table!

To Speed up Ripple Carry Adder

- Clearly the 2-level approach is not feasible
- One possible approach is to make use of the full-adder blocks, but to generate the carry signals independently, using fast carry generation logic
- Now we do not have to wait for the carry signals to ripple from full-adder to full-adder before output becomes valid

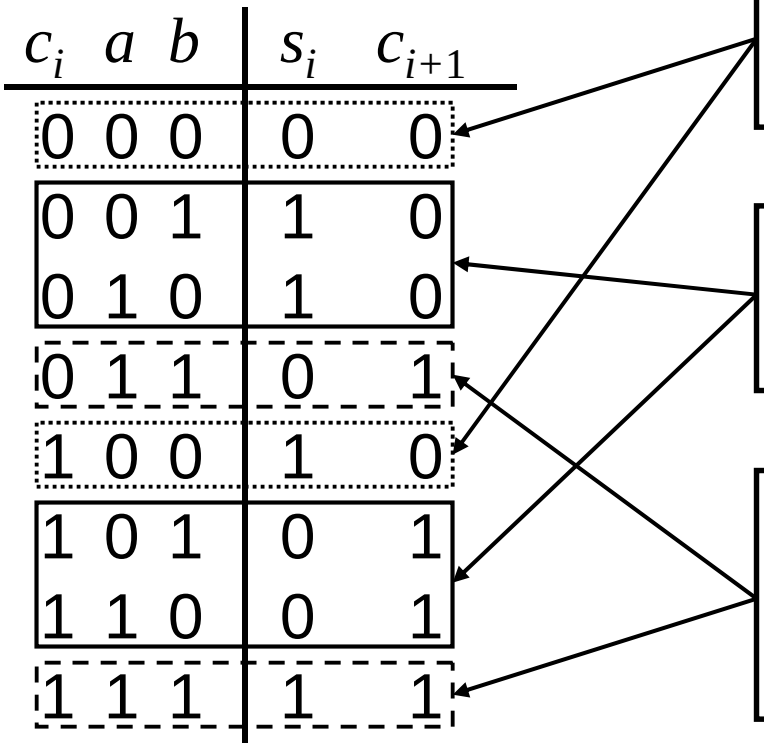
Fast Carry Generation



Fast Carry Generation

- We will now determine the Boolean equations required to generate the fast carry signals
- To do this we will consider the carry out signal, c_{out}, generated by a full-adder stage (say i), which conventionally gives rise to the carry in (c_{in}) to the next stage, i.e., c_{i+1}.

Fast Carry Generation



c_i	a	b	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Carry out always zero.

Call this *carry kill*

$$k_i = \overline{a}_i . \overline{b}_i$$

Carry out same as carry in.

Call this *carry propagate*

$$p_i = a_i \oplus b_i$$

Carry out generated independently of carry in.

Call this *carry generate*

$$g_i = a_i . b_i$$

Also (from before), $s_i = a_i \oplus b_i \oplus c_i$

Fast Carry Generation

- Also from before we have,

$c_{i+1} = a_i.b_i + c_i.(a_i + b_i)$ or alternatively,

$c_{i+1} = a_i.b_i + c_i.(a_i \oplus b_i)$

Using previous expressions gives,

$c_{i+1} = g_i + c_i.p_i$

So,

$c_{i+2} = g_{i+1} + c_{i+1}.p_{i+1}$

$c_{i+2} = g_{i+1} + p_{i+1}.(g_i + c_i.p_i)$

$c_{i+2} = g_{i+1} + p_{i+1}.g_i + p_{i+1}.p_i.c_i$

Fast Carry Generation

Similarly,

$$c_{i+3} = g_{i+2} + c_{i+2}.p_{i+2}$$

$$c_{i+3} = g_{i+2} + p_{i+2}.(g_{i+1} + p_{i+1}.(g_i + c_i.p_i))$$

$$c_{i+3} = g_{i+2} + p_{i+2}.(g_{i+1} + p_{i+1}.g_i) + p_{i+2}.p_{i+1}.p_i.c_i$$

and

$$c_{i+4} = g_{i+3} + c_{i+3}.p_{i+3}$$

$$c_{i+4} = g_{i+3} + p_{i+3}.(g_{i+2} + p_{i+2}.(g_{i+1} + p_{i+1}.g_i) + p_{i+2}.p_{i+1}.p_i.c_i)$$

$$c_{i+4} = g_{i+3} + p_{i+3}.(g_{i+2} + p_{i+2}.(g_{i+1} + p_{i+1}.g_i)) + p_{i+3}.p_{i+2}.p_{i+1}.p_i.c_i$$

Fast Carry Generation

- So for example to generate c_4, i.e., i = 0,

  $$c_4 = g_3 + p_3.(g_2 + p_2.(g_1 + p_1.g_0)) + p_3.p_2.p_1.p_0.c_0$$

  $$c_4 = G + Pc_0$$

  where,

  $$G = g_3 + p_3.(g_2 + p_2.(g_1 + p_1.g_0))$$

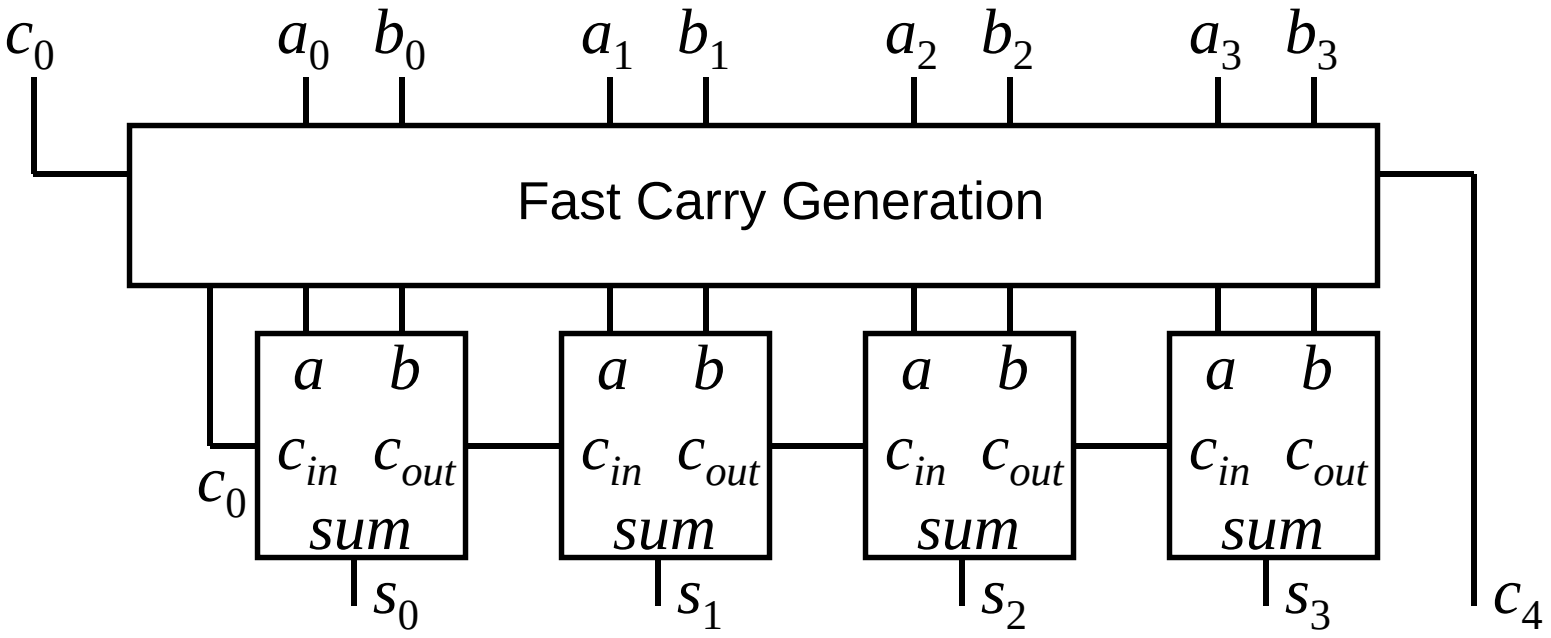
  $$P = p_3.p_2.p_1.p_0$$

- See it is quick to evaluate this function

Fast Carry Generation

- We could generate all the carrys within an adder block using the previous equations
- However, in order to reduce complexity, a suitable approach is to implement say 4-bit adder blocks with only c_4 generated using fast generation.
  - This is used as the carry-in to the next 4-bit adder block
  - Within each 4-bit adder block, conventional RCA is used

Fast Carry Generation



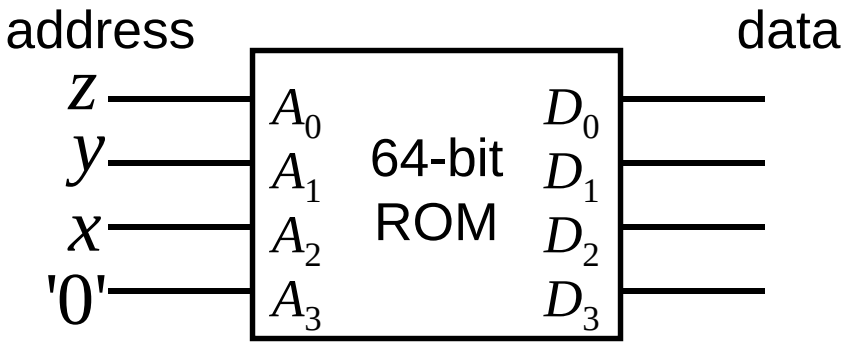
Other Ways to Implement Combinational Logic

- We have seen how combinational logic can be implemented using logic gates, e.g., AND, OR etc.
- However, it is also possible to generate combinational logic functions using memory devices, e.g., Read Only Memories (ROMs)

ROM Overview

- A ROM is a data storage device:
  - Usually written into once (either at manufacture or using a programmer)
  - Read at will
  - Essentially is a look-up table, where a group of input lines (say n) is used to specify the *address* of locations holding m-bit *data* words
  - For example, if n = 4, then the ROM has $2^4 = 16$ possible locations. If m = 4, then each location can store a 4-bit word
  - So, the total number of bits stored is $m \times 2^n$, i.e., 64 in the example (very small!) ROM

ROM Example



address (decimal)	x	y	z	f	data D_3	D_2	D_1	D_0
0	0	0	0	1	X	X	X	1
1	0	0	1	1	X	X	X	1
2	0	1	0	1	X	X	X	1
3	0	1	1	1	X	X	X	1
4	1	0	0	0	X	X	X	0
5	1	0	1	0	X	X	X	0
6	1	1	0	0	X	X	X	0
7	1	1	1	1	X	X	X	1

Design amounts to putting minterms in the appropriate address location

No logic simplification required

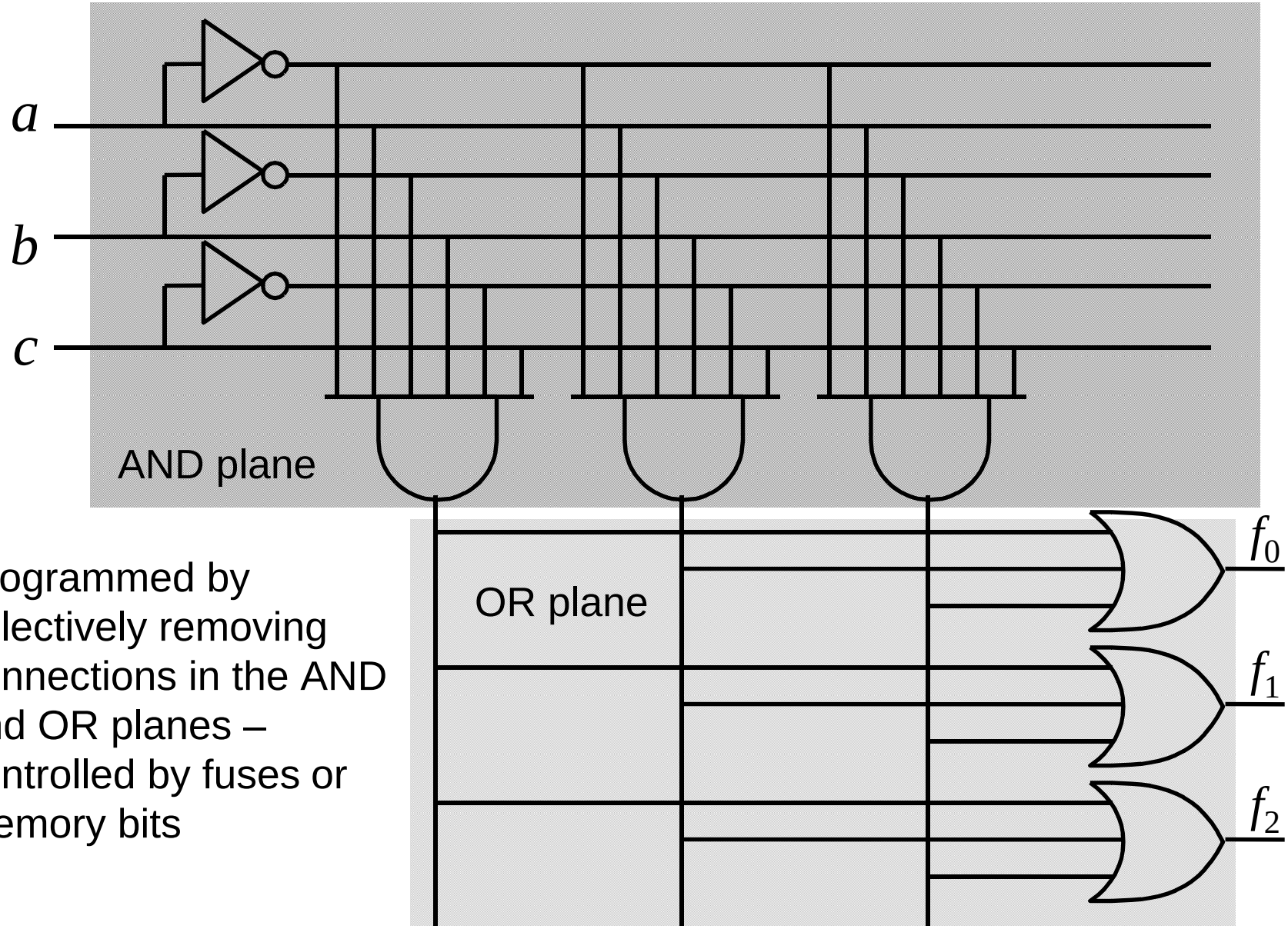
Useful if multiple Boolean functions are to be implemented, e.g., in this case we can easily do up to 4, i.e., 1 for each output line

Reasonably efficient if lots of minterms need to be generated

ROM Implementation

- Can be quite inefficient, i.e., become large in size with only a few non-zero entries, if the number of minterms in the function to be implemented is quite small
- Devices which can overcome these problems are known as programmable array logic (PAL)
- In PALs, only the required minterms are generated using a separate AND plane. The outputs from this plane are ORed together in a separate OR plane to produce the final output

Basic PAL Structure



Programmed by selectively removing connections in the AND and OR planes – controlled by fuses or memory bits

Other Memory Devices

- Non-volatile storage is offered by ROMs (and some other memory technologies, e.g., FLASH), i.e., the data remains intact, even when the power supply is removed
- Volatile storage is offered by Static Random Access Memory (SRAM) technology
  - Data can be written into and read out of the SRAM, but is lost once power is removed

Memory Application

- Memory devices are often used in computer systems
- The central processing unit (CPU) often makes use of busses (a bunch of wires in parallel) to access external memory devices
- The *address bus* is used to specify the memory location that is being read or written and the data bus conveys the data too and from that location
- So, more than one memory device will often be connected to the same data bus

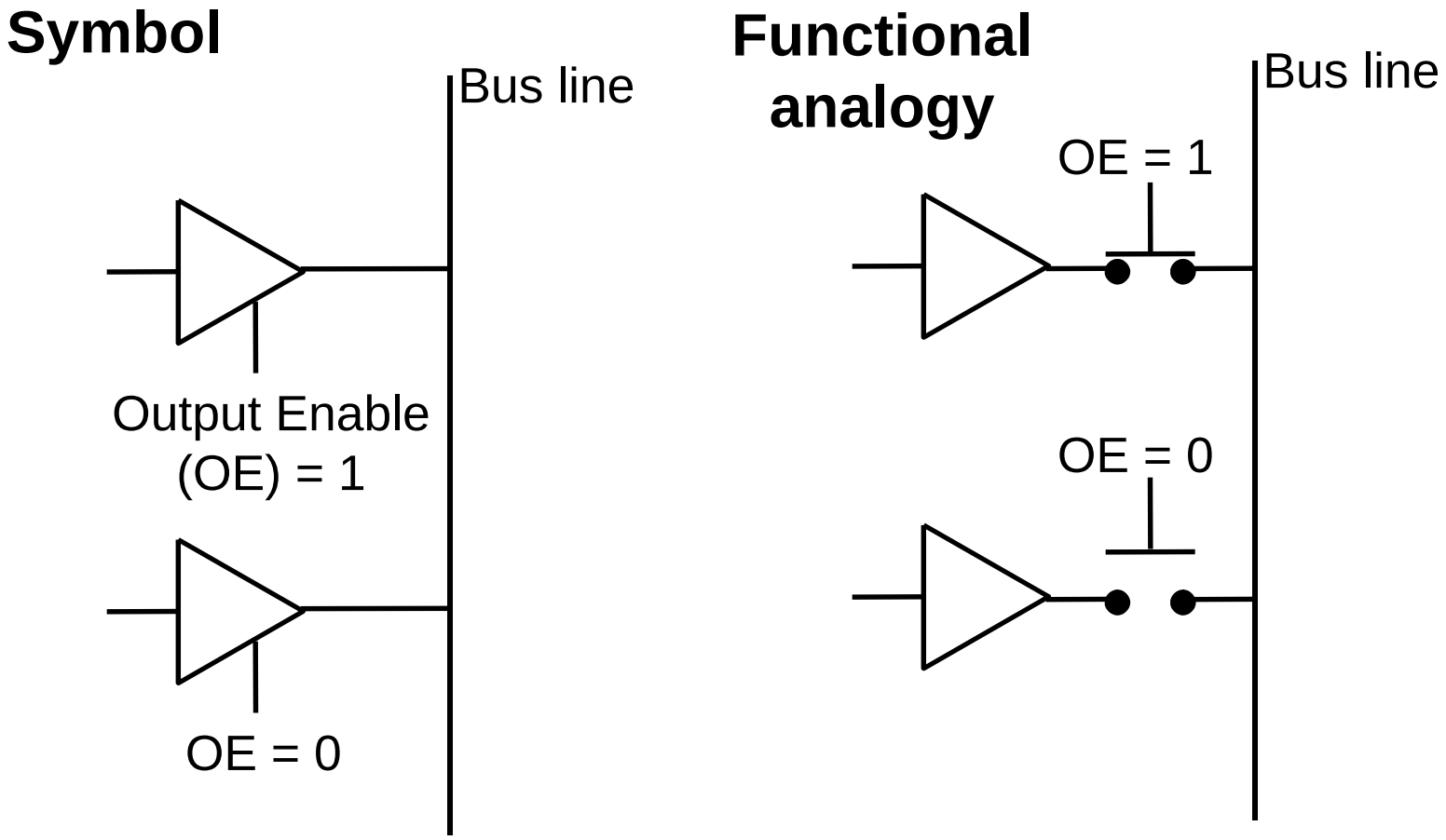
Bus Contention

- In this case, if the output from the data pin of one memory was a 0 and the output from the corresponding data pin of another memory was a 1, the data on that line of the data bus would be invalid
- So, how do we arrange for the data from multiple memories to be connected to the some bus wires?

Bus Contention

- The answer is:
  - *Tristate* buffers (or drivers)
  - Control signals
- A tristate buffer is used on the data output of the memory devices
  - In contrast to a normal buffer which is either 1 or 0 at its output, a tristate buffer can be electrically disconnected from the bus wire, i.e., it will have no effect on any other data currently on the bus – known as the *'high impedance'* condition

Tristate Buffer



Control Signals

- We have already seen that the memory devices have an additional control input (OE) that determines whether the output buffers are enabled.
- Other control inputs are also provided:
  - Write enable (WE). Determines whether data is written or read (clearly not needed on a ROM)
  - Chip select (CS) – determines if the chip is activated
- Note that these signals can be active low, depending upon the particular device

Sequential Logic

Flip-flops and Latches

Sequential Logic

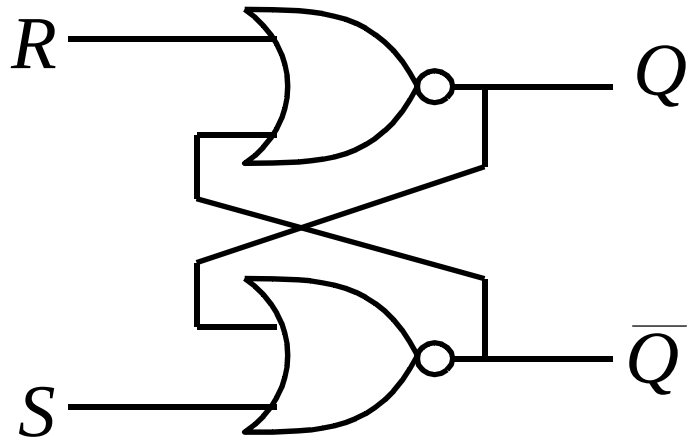
- The logic circuits discussed previously are known as *combinational*, in that the output depends only on the condition of the latest inputs
- However, we will now introduce a type of logic where the output depends not only on the latest inputs, but also on the condition of earlier inputs. These circuits are known as *sequential*, and implicitly they contain *memory* elements

Memory Elements

- A memory stores data – usually one bit per element
- A snapshot of the memory is called the *state*
- A one bit memory is often called a *bistable*, i.e., it has 2 stable internal states
- *Flip-flops* and *latches* are particular implementations of bistables

RS Latch

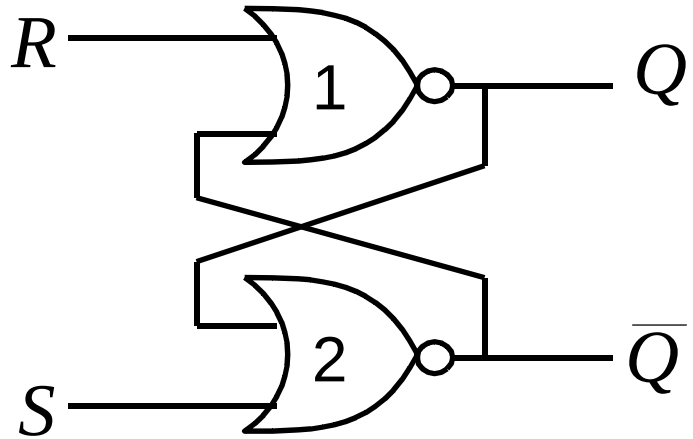
- An RS latch is a memory element with 2 inputs: Reset (R) and Set (S) and 2 outputs: Q and $\overline{Q}$.



S R	Q' $\overline{Q}'$	comment
0 0	Q $\overline{Q}$	hold
0 1	0 1	reset
1 0	1 0	set
1 1	0 0	illegal

Where Q' is the next state and Q is the current state

RS Latch - Operation

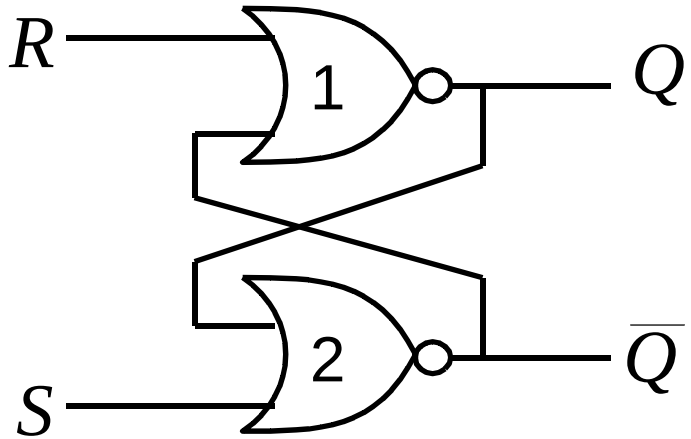


NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- R = 1 and S = 0
  - Gate 1 output in ‘always 0’ condition, $Q = 0$
  - Gate 2 in ‘complement’ condition, so $\overline{Q} = 1$
- This is the (R)eset condition

RS Latch - Operation

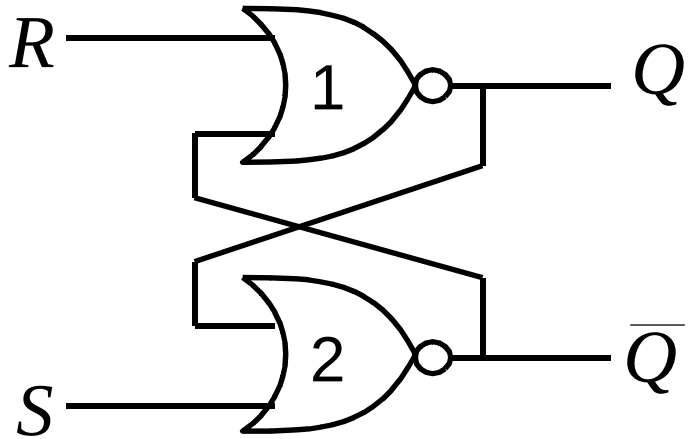


NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- S = 0 and R to 0
  - Gate 2 remains in ‘complement’ condition, $\overline{Q} = 1$
  - Gate 1 into ‘complement’ condition, $Q = 0$
- This is the hold condition

RS Latch - Operation

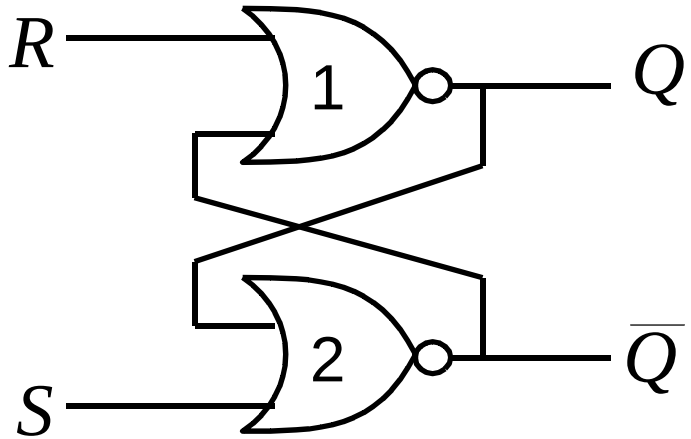


NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- S = 1 and R = 0
  - Gate 1 into ‘complement’ condition, $Q = 1$
  - Gate 2 in ‘always 0’ condition, $\overline{Q} = 0$
- This is the (S)et condition

RS Latch - Operation



NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- S = 1 and R = 1
  - Gate 1 in ‘always 0’ condition, $Q = 0$
  - Gate 2 in ‘always 0’ condition, $\overline{Q} = 0$
- This is the illegal condition

RS Latch – State Transition Table

- A *state transition table* is an alternative way of viewing its operation

Q	S	R	Q'	comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

- A state transition table can also be expressed in the form of a *state diagram*

RS Latch – State Diagram

- A *state diagram* in this case has 2 states, i.e., Q=0 and Q=1
- The state diagram shows the input conditions required to transition between states. In this case we see that there are 4 possible transitions
- We will consider them in turn

RS Latch – State Diagram

Q	S	R	Q'	comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

$Q = 0 \quad Q' = 0$

From the table we can see:

$$\bar{S}.\bar{R} + \bar{S}.R + S.R =$$

$$\bar{S}.(\bar{R} + R) + S.R = \bar{S} + S.R =$$

$$(\bar{S} + S).(\bar{S} + R) = \bar{S} + R$$

$Q = 1 \quad Q' = 1$

From the table we can see:

$$\bar{S}.\bar{R} + S.\bar{R} = \bar{R}.(\bar{S} + S) =$$

$$\bar{R}$$

RS Latch – State Diagram

Q	S	R	Q'	comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

$Q = 1 \quad Q' = 0$

From the table we can see:

$\overline{S}.R + S.R =$

$R.(\overline{S} + S) = R$

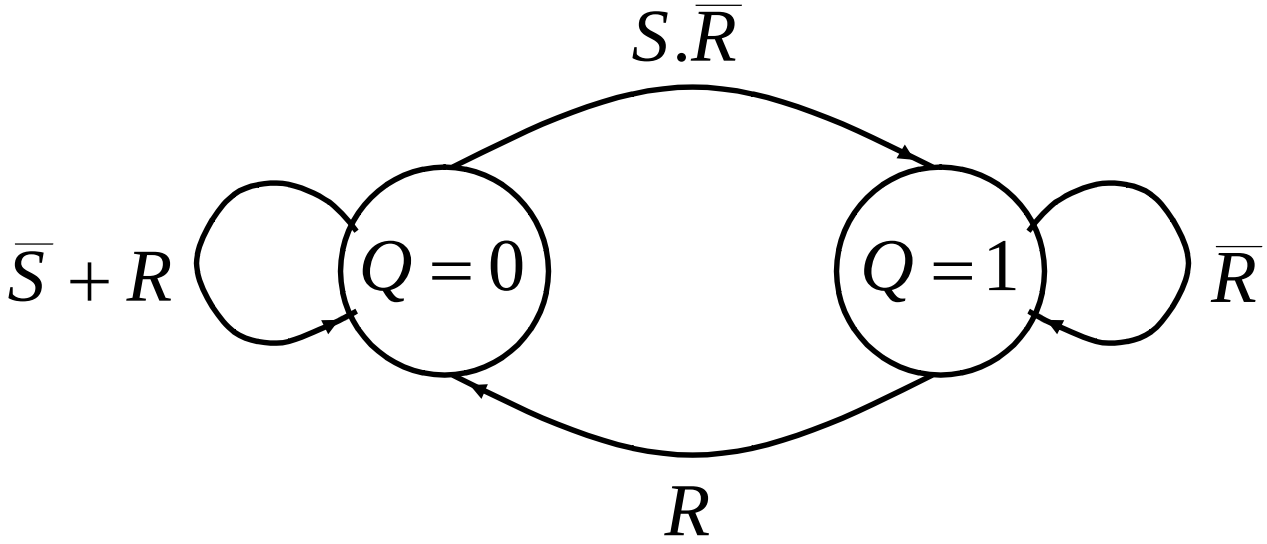
$Q = 0 \quad Q' = 1$

From the table we can see:

$S.\overline{R}$

RS Latch – State Diagram

- Which gives the following state diagram:



- A similar diagram can be constructed for the $\overline{Q}$ output
- We will see later that state diagrams are a useful tool for designing sequential systems

Clocks and Synchronous Circuits

- For the RS latch we have just described, we can see that the output state changes occur directly in response to changes in the inputs. This is called *asynchronous* operation
- However, virtually all sequential circuits currently employ the notion of *synchronous* operation, that is, the output of a sequential circuit is constrained to change only at a time specified by a global *enabling* signal. This signal is generally known as the system *clock*

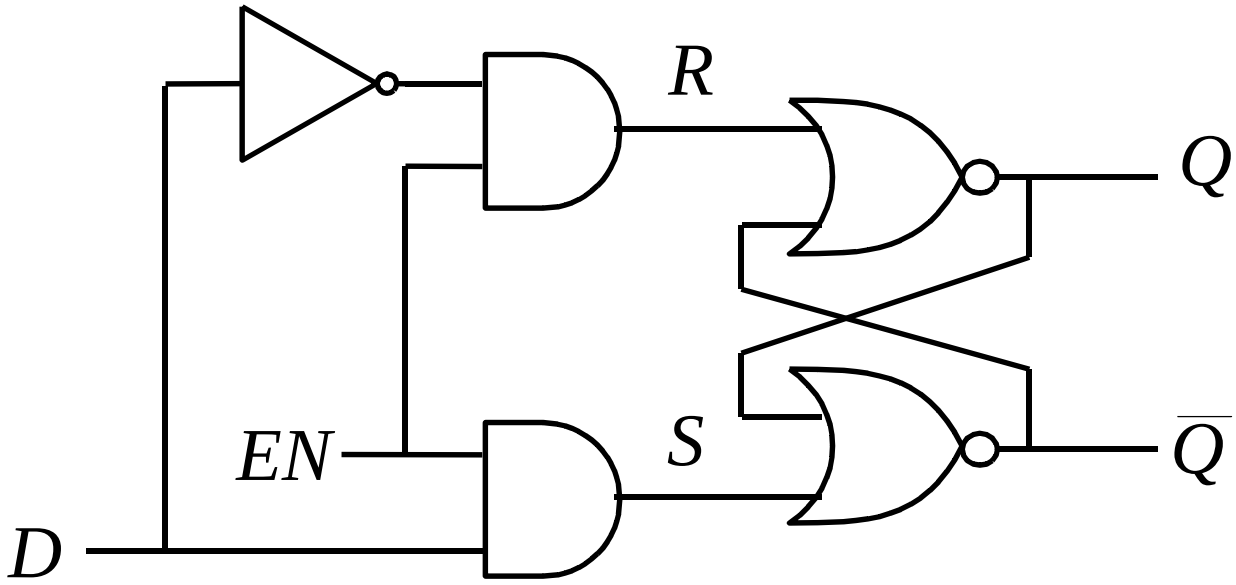
Clocks and Synchronous Circuits

- The Clock: What is it and what is it for?
  - Typically it is a square wave signal at a particular frequency
  - It imposes order on the state changes
  - Allows lots of states to appear to update simultaneously
- How can we modify an asynchronous circuit to act synchronously, i.e., in synchronism with a clock signal?

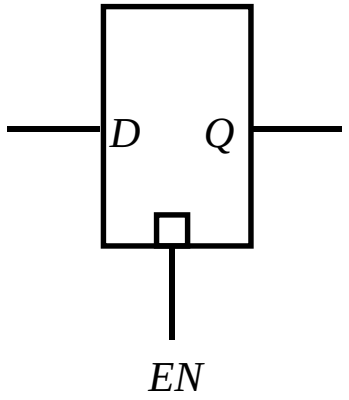
Transparent D Latch

- We now modify the RS Latch such that its output state is only permitted to change when a valid enable signal (which could be the system clock) is present
- This is achieved by introducing a couple of AND gates in cascade with the R and S inputs that are controlled by an additional input known as the *enable* (EN) input.

Transparent D Latch



Symbol

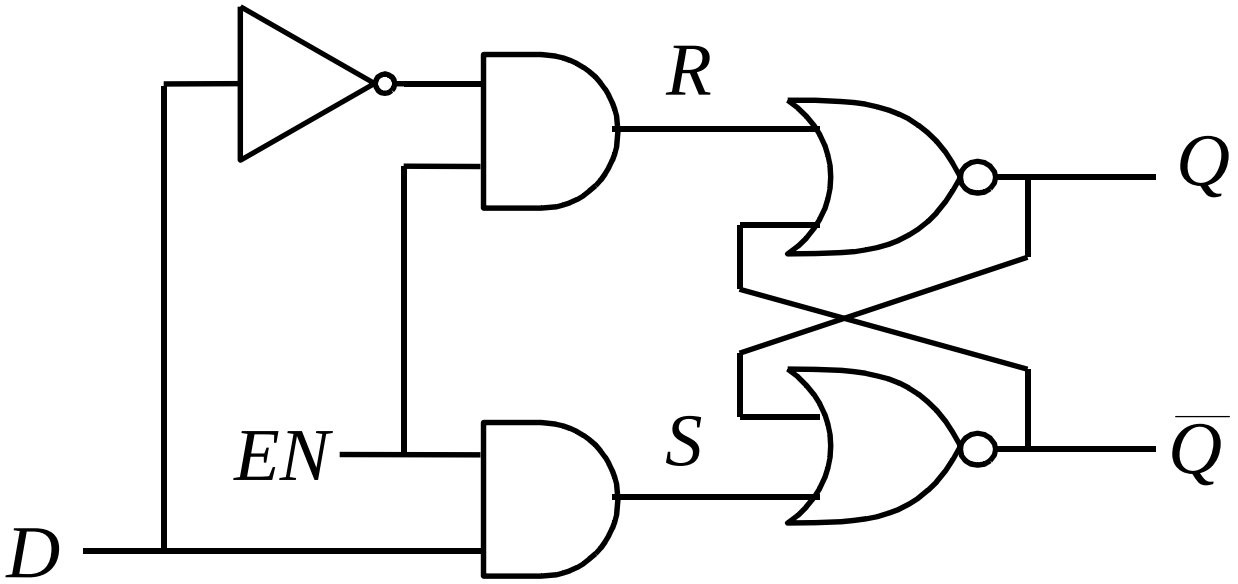


- See from the AND truth table:
  - if one of the inputs, say a is 0, the output is always 0
  - Output follows b input if a is 1
- The complement function ensures that R and S can never be 1 at the same time, i.e., illegal avoided

AND truth table

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Transparent D Latch



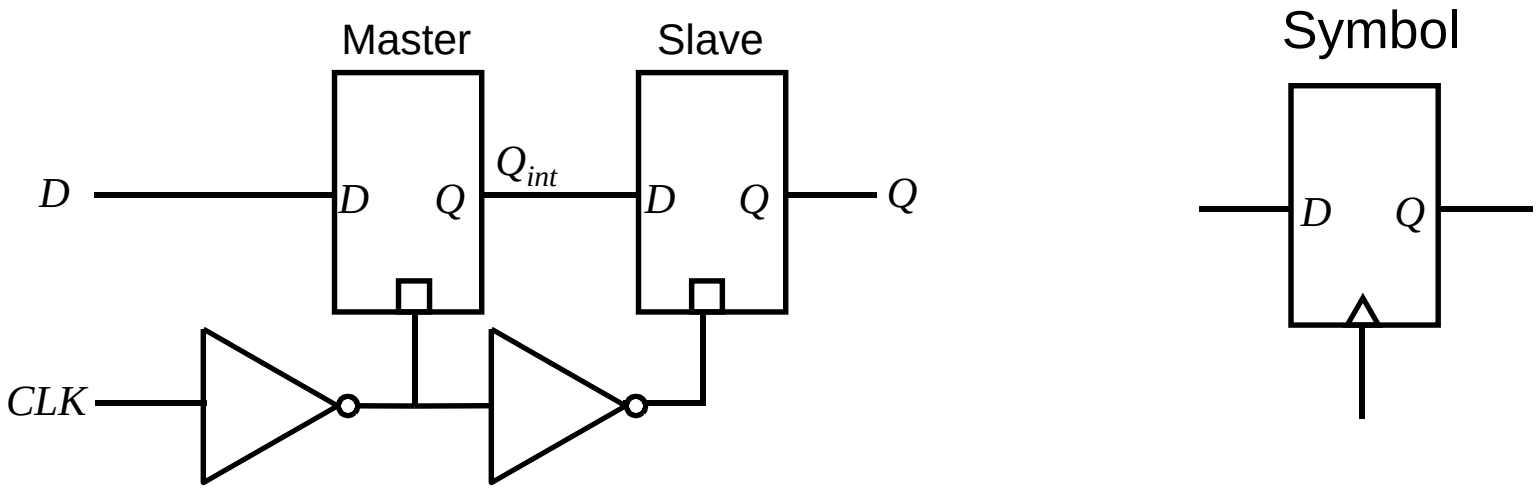
D	EN	Q'	$\overline{Q}'$	comment
X	0	Q	$\overline{Q}$	RS hold
0	1	0	1	RS reset
1	1	1	0	RS set

- See Q follows D input provided EN=1.
  If EN=0, Q maintains previous state

Master-Slave Flip-Flops

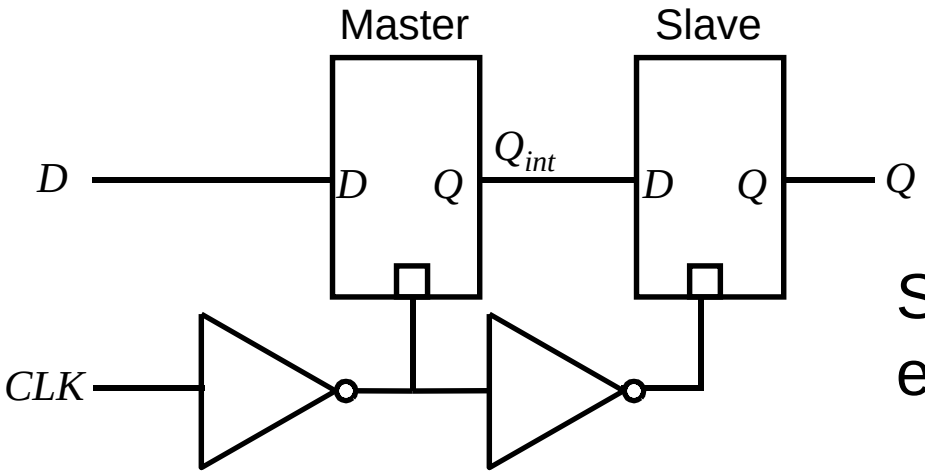
- The transparent D latch is so called '*level*' triggered. We can see it exhibits transparent behaviour if *EN*=1. It is often more simple to design sequential circuits if the outputs change only on the either rising (positive going) or falling (negative going) '*edges*' of the clock (i.e., enable) signal
- We can achieve this kind of operation by combining 2 transparent D latches in a so called *Master-Slave* configuration

Master-Slave D Flip-Flop

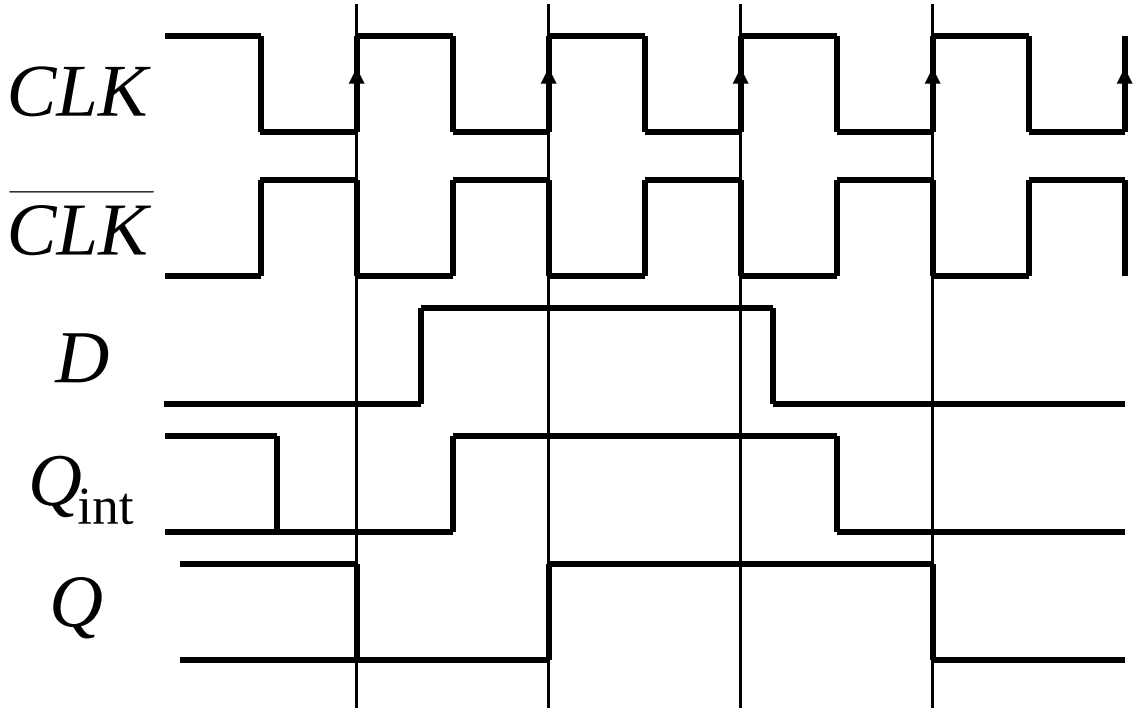


- To see how this works, we will use a timing diagram
- Note that both latch inputs are effectively connected to the clock signal (admittedly one is a complement of the other)

Master-Slave D Flip-Flop



See Q changes on rising edge of CLK



Note propagation delays have been neglected in the timing diagram

D Flip-Flops

- The Master-Slave configuration has now been superseded by new F-F circuits which are easier to implement and have better performance
- When designing synchronous circuits it is best to use truly edge triggered F-F devices
- We will not consider the design of such F-Fs on this course

Other Types of Flip-Flops

- Historically, other types of Flip-Flops have been important, e.g., J-K Flip-Flops and T-Flip-Flops
- However, J-K FFs are a lot more complex to build than D-types and so have fallen out of favour in modern designs, e.g., for field programmable gate arrays (FPGAs) and VLSI chips

Other Types of Flip-Flops

- Consequently we will only consider synchronous circuit design using D-type FFs
- However for completeness we will briefly look at the truth table for J-K and T type FFs

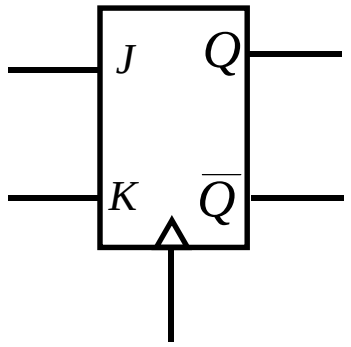
J-K Flip-Flop

- The J-K FF is similar in function to a clocked RS FF, but with the illegal state replaced with a new ‘toggle’ state

J K	Q' $\overline{Q}'$	comment
0 0	Q $\overline{Q}$	hold
0 1	0 1	reset
1 0	1 0	set
1 1	$\overline{Q}$ Q	toggle

Where Q' is the next state and Q is the current state

Symbol



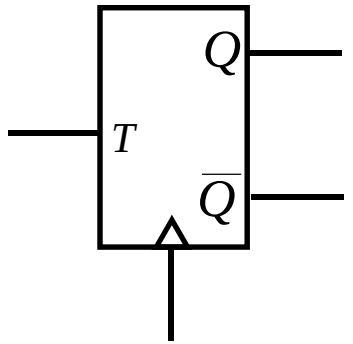
T Flip-Flop

- This is essentially a J-K FF with its J and K inputs connected together and renamed as the T input

T	Q' $\overline{Q}'$	comment
0	Q $\overline{Q}$	hold
1	$\overline{Q}$ Q	toggle

Where Q' is the next state
and Q is the current state

Symbol



Asynchronous Inputs

- It is common for the FF types we have mentioned to also have additional so called ‘asynchronous’ inputs
- They are called asynchronous since they take effect independently of any clock or enable inputs
- Reset/Clear – force Q to 0
- Preset/Set – force Q to 1
- Often used to force a synchronous circuit into a known state, say at start-up.

Timing

- Various timings must be satisfied if a FF is to operate properly:
  - *Setup time*: Is the minimum duration that the data must be stable at the input before the clock edge
  - *Hold time*: Is the minimum duration that the data must remain stable on the FF input after the clock edge

Applications of Flip-Flops

- Counters
  - A clocked sequential circuit that goes through a predetermined sequence of states
  - A commonly used counter is an n-bit binary counter. This has n FFs and 2^n states which are passed through in the order 0, 1, 2, ….2^n-1, 0, 1, .
  - Uses include:
    - Counting
    - Producing delays of a particular duration
    - Sequencers for control logic in a processor
    - Divide by m counter (a divider), as used in a digital watch

Applications of Flip-Flops

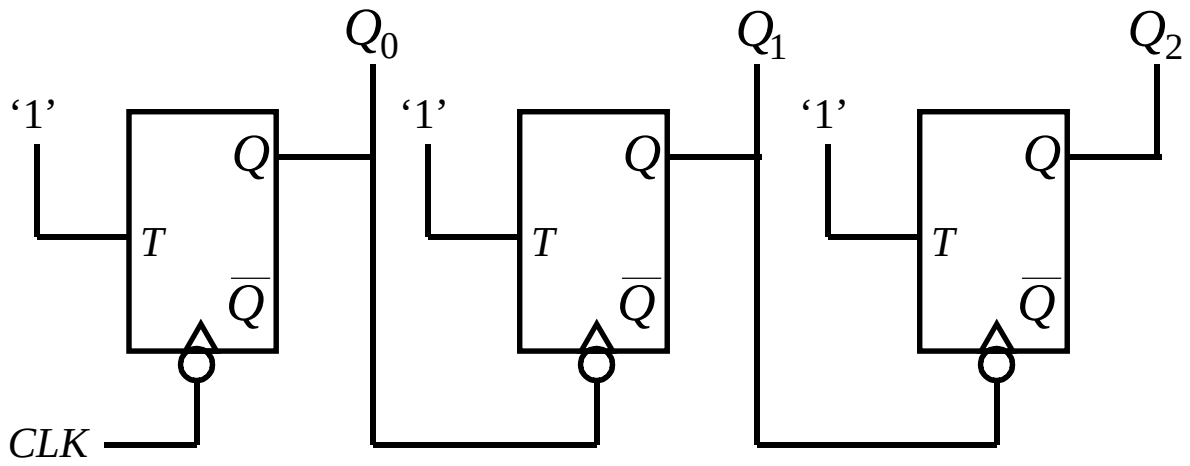
- Memories, e.g.,
  - Shift register
    - Parallel loading shift register : can be used for parallel to serial conversion in serial data communication
    - Serial in, parallel out shift register: can be used for serial to parallel conversion in a serial data communication system.

Counters

- In most books you will see 2 basic types of counters, namely *ripple* counters and *synchronous* counters
- In this course we are concerned with synchronous design principles. Ripple counters do not follow these principles and should generally be avoided if at all possible. We will now look at the problems with ripple counters

Ripple Counters

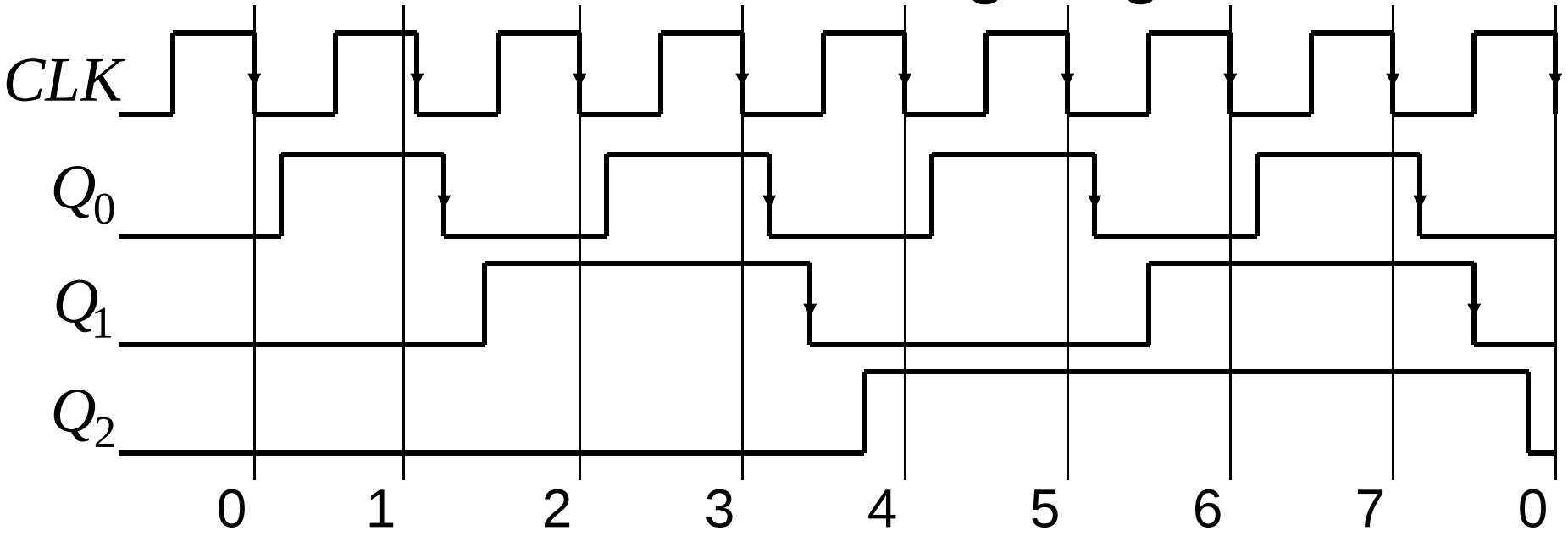
- A ripple counter can be made be cascading together negative edge triggered T-type FFs operating in ‘toggle’ mode, i.e., T =1



- See that the FFs are not clocked using the same clock, i.e., this is **not** a synchronous design. This gives some problems….

Ripple Counters

- We will now draw a timing diagram



- Problems:

See outputs do not change at the same time, i.e., synchronously. So hard to know when count output is actually valid.

Propagation delay builds up from stage to stage, limiting maximum clock speed before miscounting occurs.

Ripple Counters

- If you observe the frequency of the counter output signals you will note that each has half the frequency, i.e., double the repetition period of the previous one. This is why counters are often known as dividers
- Often we wish to have a count which is not a power of 2, e.g., for a BCD counter (0 to 9).To do this:
  - use FFs having a Reset/Clear input
  - Use an AND gate to detect the count of 10 and use its output to Reset the FFs

Synchronous Counters

- Owing to the problems identified with ripple counters, they should not usually be used to implement counter functions
- It is recommended that *synchronous* counter designs be used
- In a synchronous design
  - all the FF clock inputs are directly connected to the clock signal and so all FF outputs change at the same time, i.e., *synchronously*
  - more complex combinational logic is now needed to generate the appropriate FF input signals (which will be different depending upon the type of FF chosen)

Synchronous Counters

- We will now investigate the design of synchronous counters
- We will consider the use of D-type FFs only, although the technique can be extended to cover other FF types.
- As an example, we will consider a 0 to 7 up-counter

Synchronous Counters

- To assist in the design of the counter we will make use of a modified *state transition table.* This table has additional columns that define the required FF inputs (or e*xcitation* as it is known)
  - Note we have used a state transition table previously when determining the state diagram for an RS latch
- We will also make use of the so called '*excitation table*' for a D-type FF
- First however, we will investigate the so called *characteristic table* and *characteristic equation* for a D-type FF

Characteristic Table

- In general, a characteristic table for a FF gives the next state of the output, i.e., Q' in terms of its current state Q and current inputs

Q	D	Q'
0	0	0
0	1	1
1	0	0
1	1	1

Which gives the characteristic equation,

$$Q' = D$$

i.e., the next output state is equal to the current input value

Since Q' is independent of Q the characteristic table can be rewritten as

D	Q'
0	0
1	1

Excitation Table

- The characteristic table can be modified to give the excitation table. This table tells us the required FF input value required to achieve a particular next state from a given current state

Q	Q'	D
0	0	0
0	1	1
1	0	0
1	1	1

As with the characteristic table it can be seen that Q', does not depend upon, Q , however this is not generally true for other FF types, in which case, the excitation table is more useful. Clearly for a D-FF, $D = Q'$

Characteristic and Excitation Tables

- Characteristic and excitation tables can be determined for other FF types.
- These should be used in the design process if D-type FFs are not used
- We will now determine the modified state transition table for the example 0 to 7 up-counter

Modified State Transition Table

- In addition to columns representing the current and desired next states (as in a conventional state transition table), the modified table has additional columns representing the required FF inputs to achieve the next desired FF states

Modified State Transition Table

- For a 0 to 7 counter, 3 D-type FFs are needed

Current state	Next state	FF inputs
$Q_2 Q_1 Q_0$	$Q_2' Q_1' Q_0'$	$D_2 D_1 D_0$
0 0 0	0 0 1	0 0 1
0 0 1	0 1 0	0 1 0
0 1 0	0 1 1	0 1 1
0 1 1	1 0 0	1 0 0
1 0 0	1 0 1	1 0 1
1 0 1	1 1 0	1 1 0
1 1 0	1 1 1	1 1 1
1 1 1	0 0 0	0 0 0

The procedure is to:

- Write down the desired count sequence in the current state columns
- Write down the required next states in the next state columns
- Fill in the FF inputs required to give the defined next state

Note: Since $Q' = D$ (or $D = Q'$) for a D-FF, the required FF inputs are identical to the Next state

Synchronous Counter Example

- Also note that if we are using D-type FFs, it is not necessary to explicitly write out the FF input columns, since we know they are identical to those for the next state
- To complete the design we now have to determine appropriate combinational logic circuits which will generate the required FF inputs from the current states
- We can do this from inspection, using Boolean algebra or using K-maps.

Synchronous Counter Example

Current state			Next state			FF inputs		
Q_2	Q_1	Q_0	Q_2'	Q_1'	Q_0'	D_2	D_1	D_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

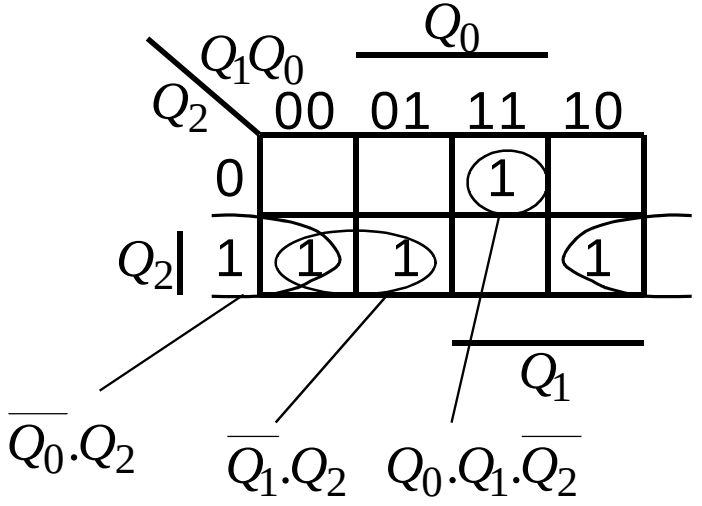
By inspection,

$$D_0 = \overline{Q_0}$$

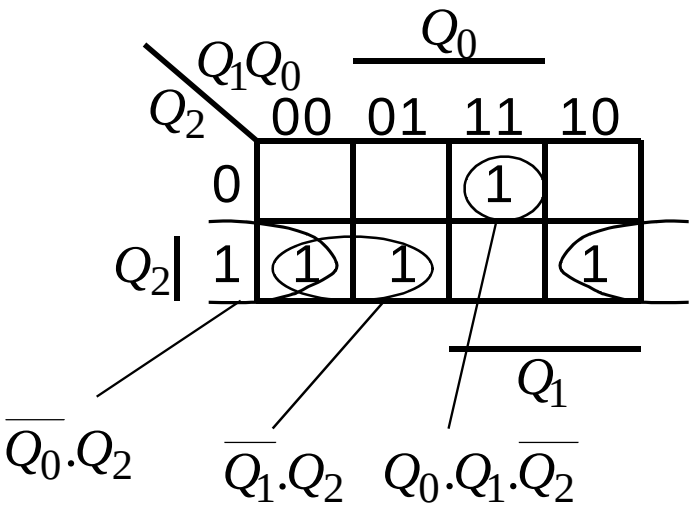
Note: FF_0 is toggling

Also, $D_1 = Q_0 \oplus Q_1$

Use a K-map for D_2 ,



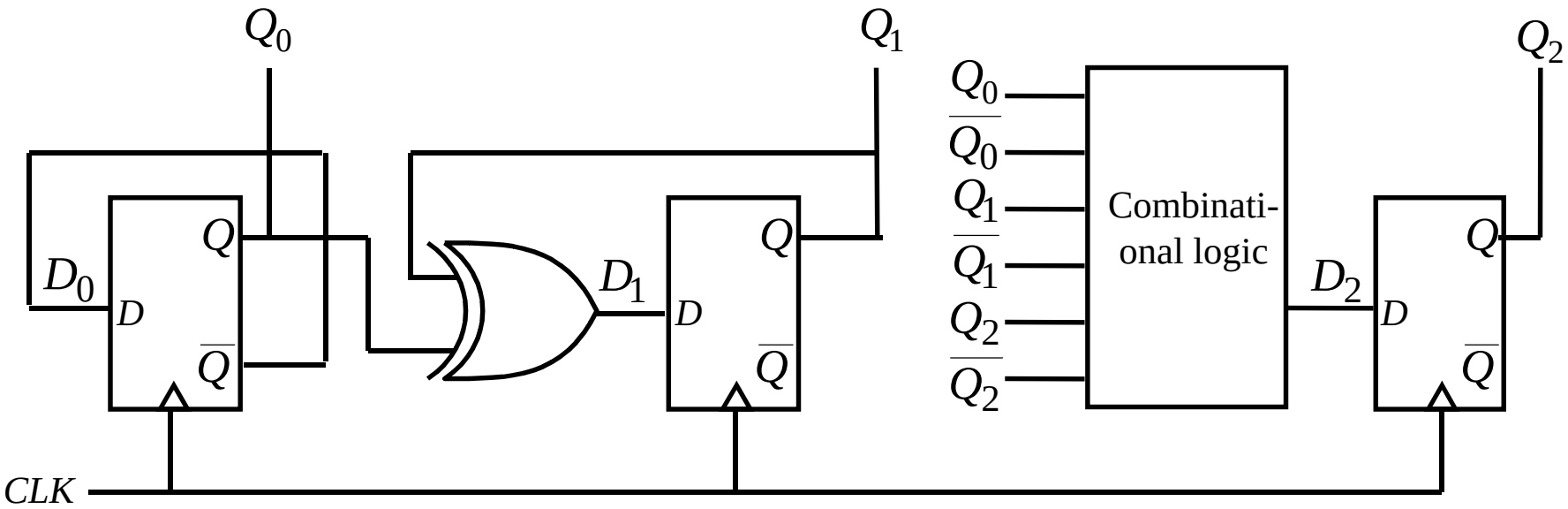
Synchronous Counter Example



So,

$$D_2 = \overline{Q_0}.Q_2 + \overline{Q_1}.Q_2 + Q_0.Q_1.\overline{Q_2}$$

$$D_2 = Q_2.(\overline{Q_0}. + \overline{Q_1}) + Q_0.Q_1.\overline{Q_2}$$

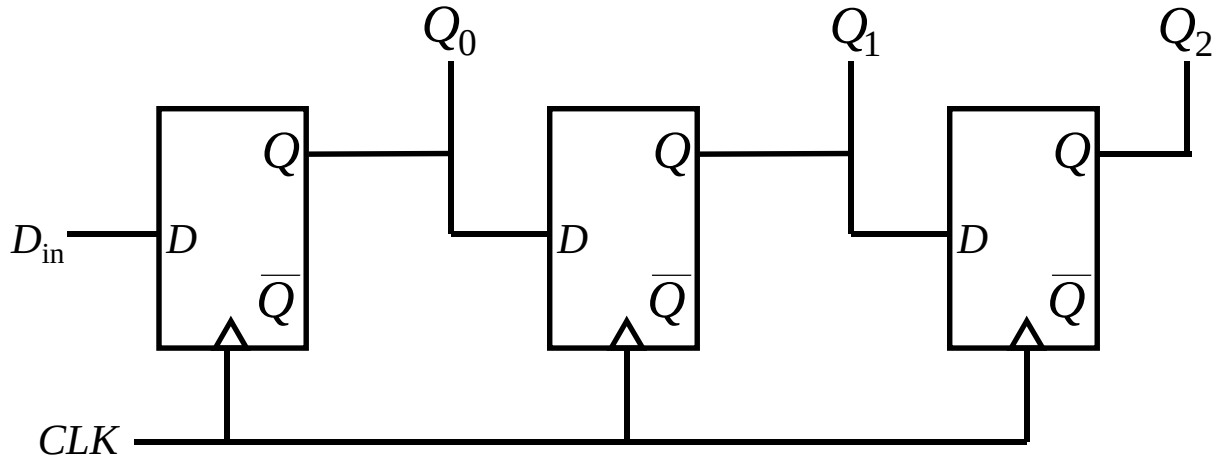


Synchronous Counter

- A similar procedure can be used to design counters having an arbitrary count sequence
  - Write down the state transition table
  - Determine the FF excitation (easy for D-types)
  - Determine the combinational logic necessary to generate the required FF excitation from the current states – **Note:** remember to take into account any unused counts since these can be used as don’t care states when determining the combinational logic circuits

Shift Register

- A shift register can be implemented using a chain of D-type FFs

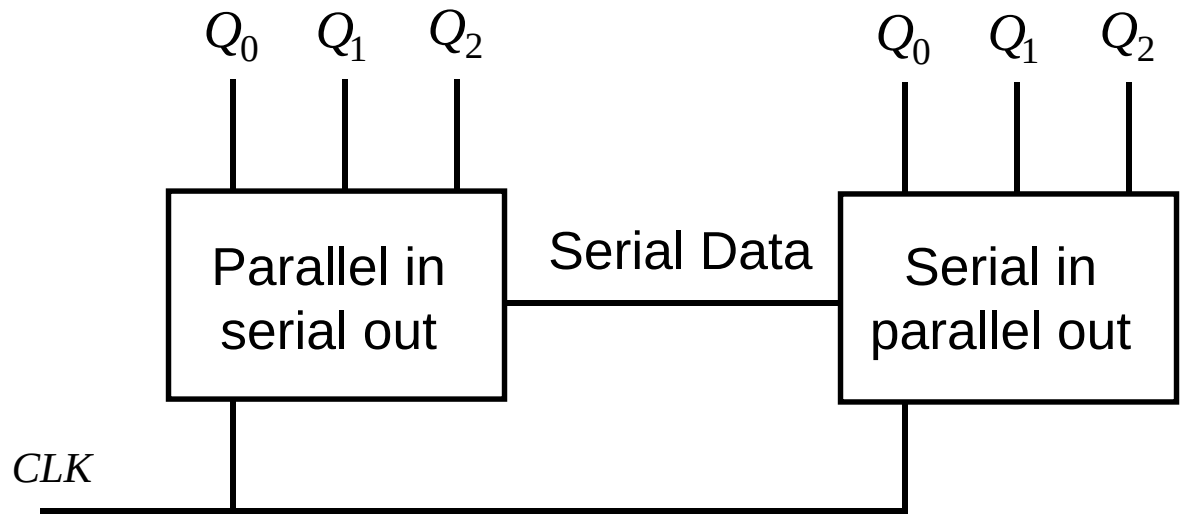


- Has a serial input, D_{in} and parallel output Q_0, Q_1 and Q_2.
- See data moves one position to the right on application of clock edge

Shift Register

- Preset and Clear inputs on the FFs can be utilised to provide a parallel data input feature
- Data can then be clocked out through Q_2 in a serial fashion, i.e., we now have a parallel in, serial out arrangement
- This along with the previous serial in, parallel out shift register arrangement can be used as the basis for a serial data link

Serial Data Link



- One data bit at a time is sent across the serial data link
- See less wires are required than for a parallel data link

Synchronous State Machines

Synchronous State Machines

- We have seen how we can use FFs (D-types in particular) to design synchronous counters
- We will now investigate how these principles can be extended to the design of synchronous state machines (of which counters are a subset)
- We will begin with some definitions and then introduce two popular types of machines

Definitions

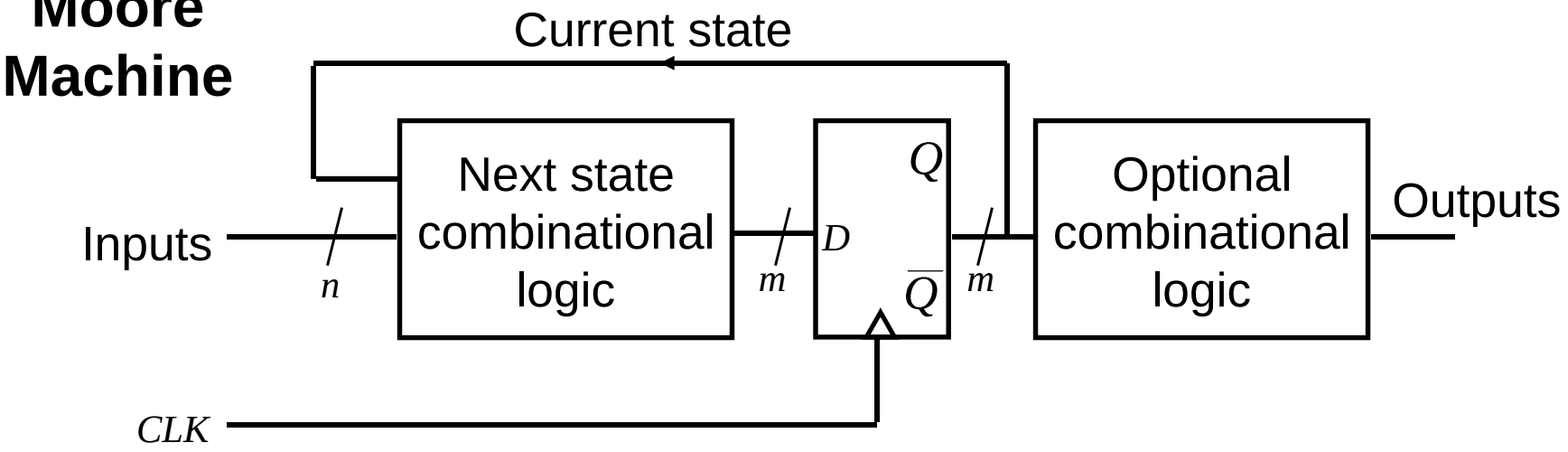
- **Finite State Machine (FSM)** – a deterministic machine (circuit) that produces outputs which depend on its internal state and external inputs
- **States** – the set of internal memorised values, shown as circles on the state diagram
- **Inputs –** External stimuli, labelled as arcs on the state diagram
- **Outputs –** Results from the FSM

Types of State Machines

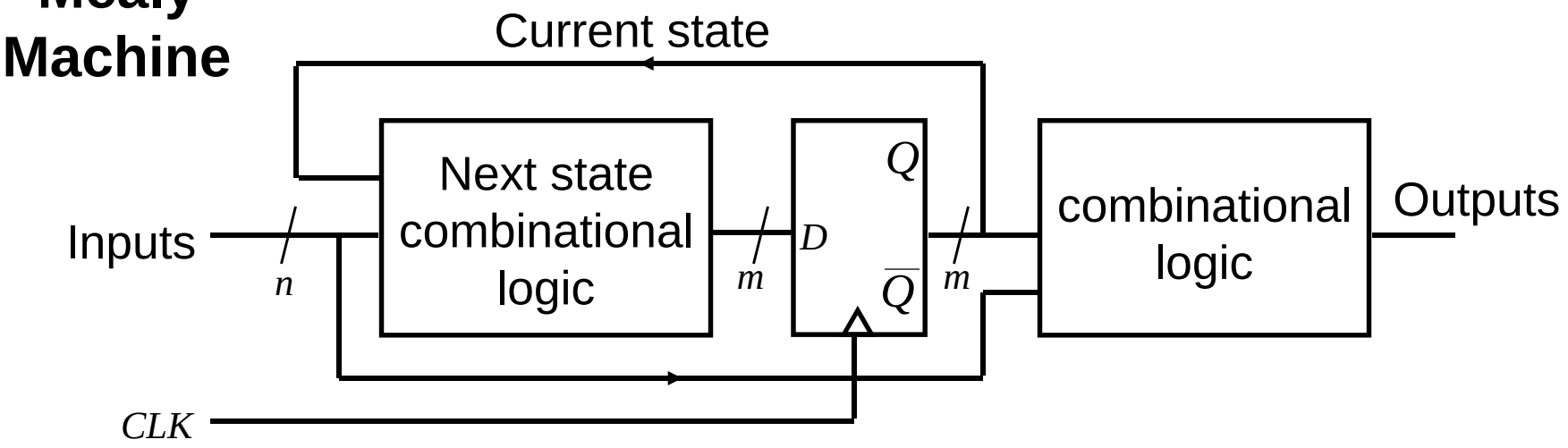
- Two types of state machines are in general use, namely *Moore* machines and *Mealy* machines
- In this course we will only look in detail at FSM design using Moore machines, although for completeness we will briefly describe the structure of Mealy machines

Machine Schematics

Moore Machine



Mealy Machine



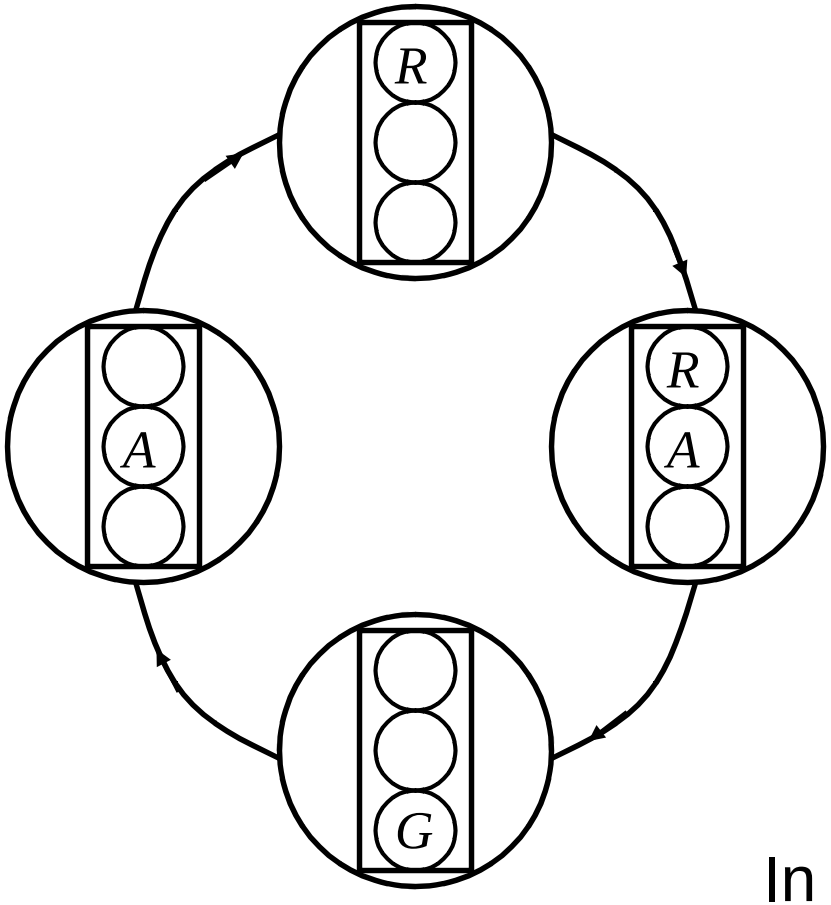
Moore vs. Mealy Machines

- Outputs from Mealy Machines depend upon the timing of the inputs
- Outputs from Moore machines come directly from clocked FFs so:
  - They have guaranteed timing characteristics
  - They are glitch free
- Any Mealy machine can be converted to a Moore machine and vice versa, though their timing properties will be different

Moore Machine - Example

- We will design a Moore Machine to implement a traffic light controller
- In order to visualise the problem it is often helpful to draw the state transition diagram
- This is used to generate the state transition table
- The state transition table is used to generate
  - The next state combinational logic
  - The output combinational logic (if required)

Example – Traffic Light Controller



See we have 4 states

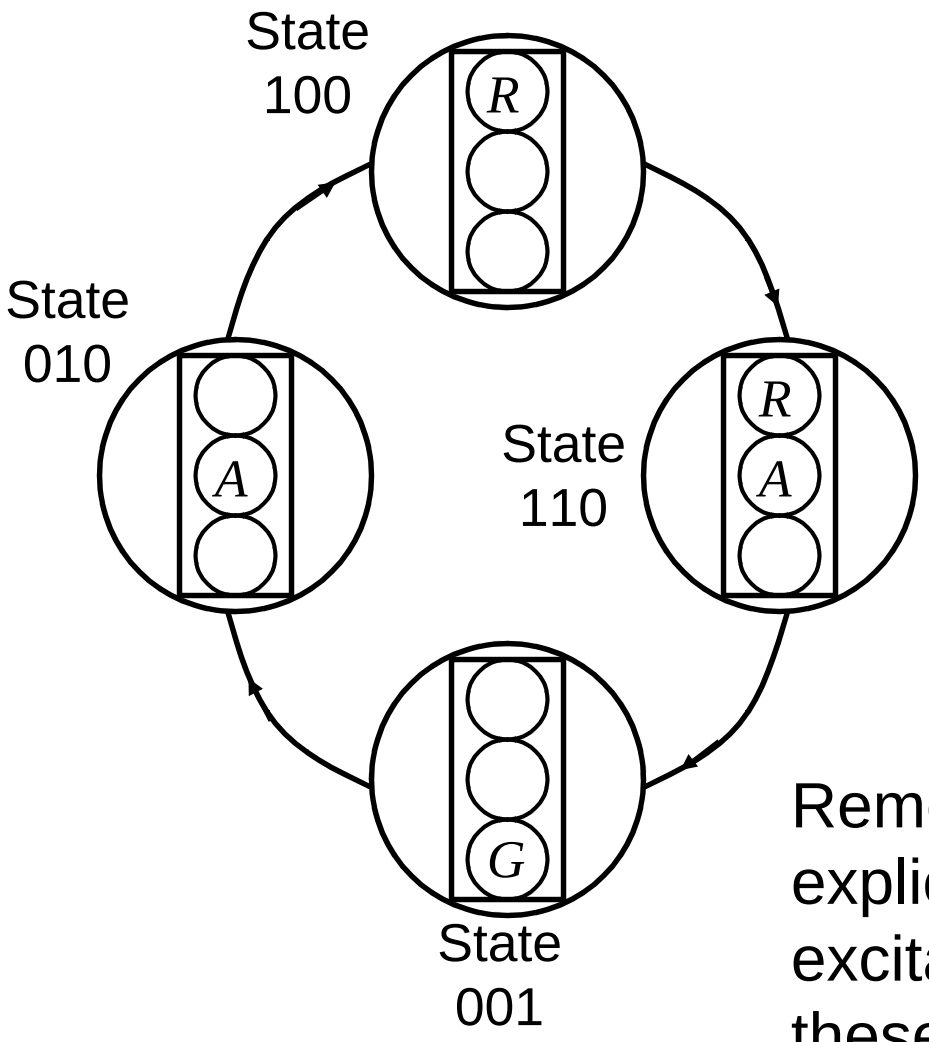
So in theory we could use a minimum of 2 FFs

However, by using 3 FFs we will see that we do not need to use any output combinational logic

So, we will only use 4 of the 8 possible states

In general, state assignment is a difficult problem and the optimum choice is not always obvious

Example – Traffic Light Controller



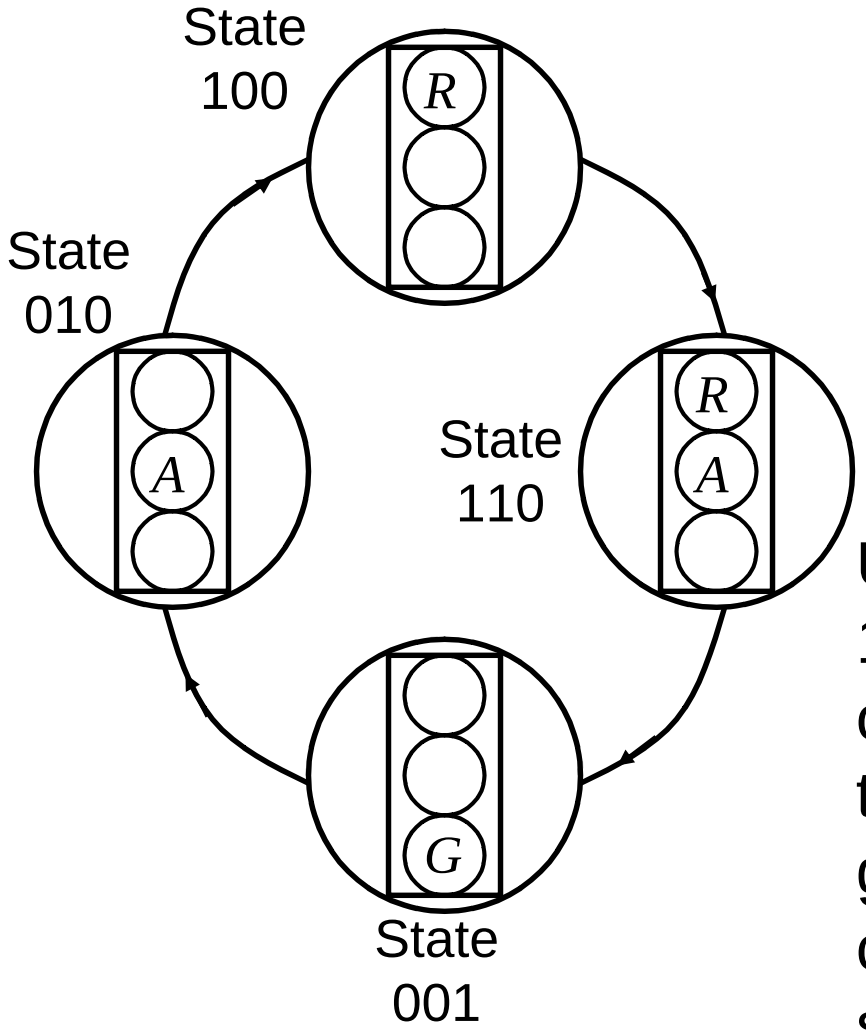
By using 3 FFs (we will use D-types), we can assign one to each of the required outputs (R, A, G), eliminating the need for output logic

We now need to write down the state transition table

We will label the FF outputs R, A and G

Remember we do not need to explicitly include columns for FF excitation since if we use D-types these are identical to the next state

Example – Traffic Light Controller



Current state			Next state		
R	A	G	R'	A'	G'
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0

Unused states, 000, 011, 101 and 111. Since these states will never occur, we don't care what output the next state combinational logic gives for these inputs. These don't care conditions can be used to simplify the required next state combinational logic

Example – Traffic Light Controller

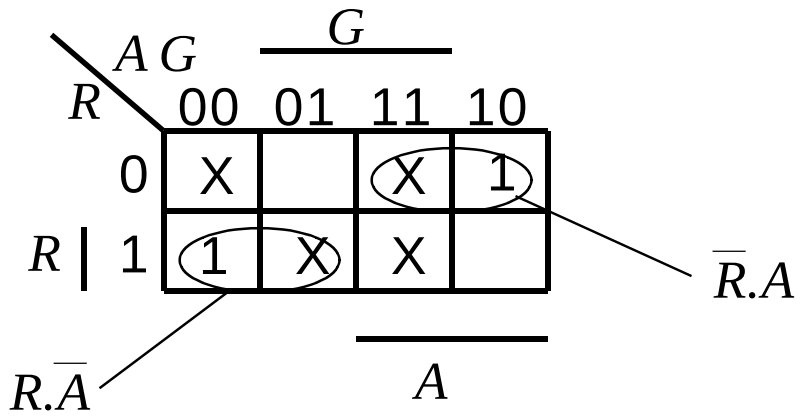
Current state			Next state		
R	A	G	R'	A'	G'
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0

Unused states, 000, 011, 101 and 111.

We now need to determine the next state combinational logic

For the R FF, we need to determine D_R

To do this we will use a K-map



$$D_R = R.\overline{A} + \overline{R}.A = R \oplus A$$

Example – Traffic Light Controller

Current state			Next state		
R	A	G	R'	A'	G'
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0

Unused states, 000, 011, 101 and 111.

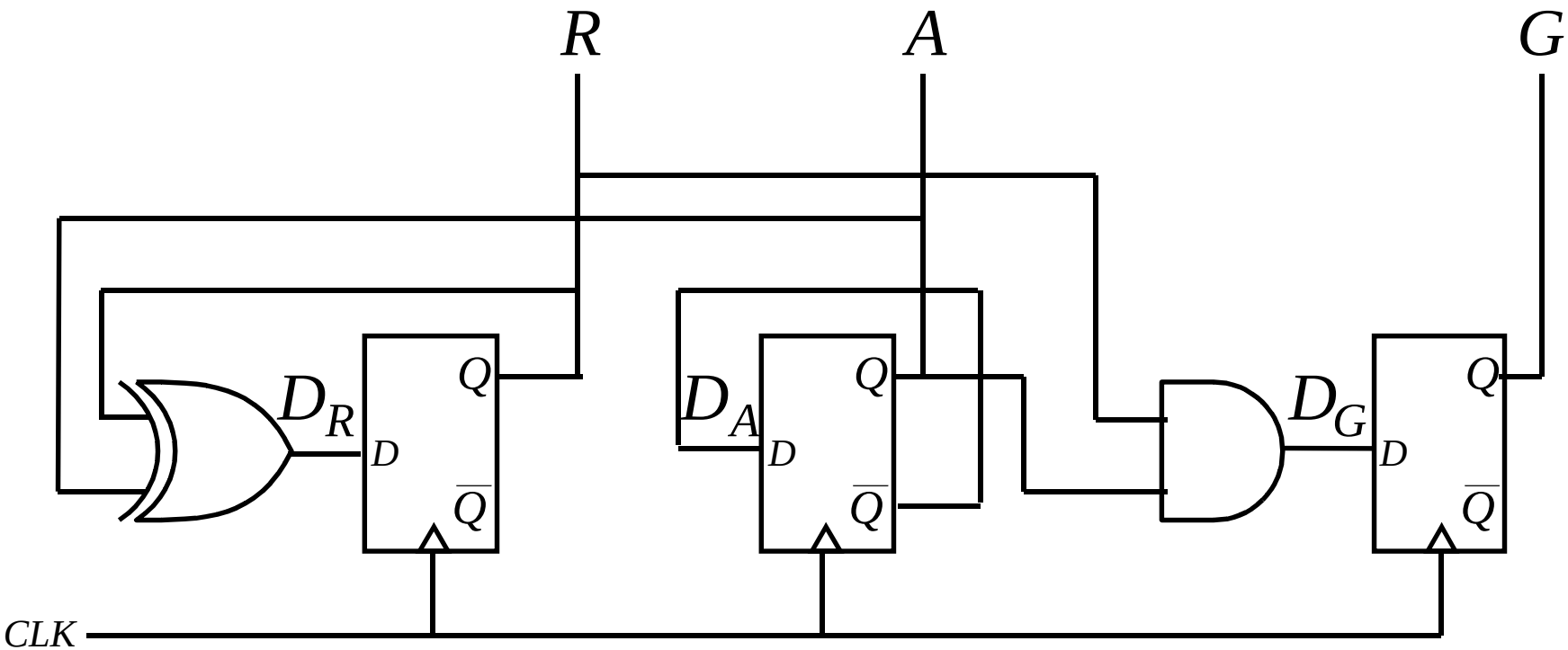
By inspection we can also see:

$$D_A = \overline{A}$$

and,

$$D_G = R.A$$

Example – Traffic Light Controller



FSM Problems

- Consider what could happen on power-up
- The state of the FFs could by chance be in one of the unused states
  - This could potentially cause the machine to become stuck in some unanticipated sequence of states which never goes back to a used state

FSM Problems

- What can be done?
  - Check to see if the FSM can eventually enter a known state from any of the unused states
  - If not, add additional logic to do this, i.e., include unused states in the state transition table along with a valid next state
  - Alternatively use asynchronous Clear and Preset FF inputs to set a known (used) state at power up

Example – Traffic Light Controller

- Does the example FSM self-start?
- Check what the next state logic outputs if we begin in any of the unused states
- Turns out:

Start state	Next state logic output
000	010
011	100
101	110
111	001

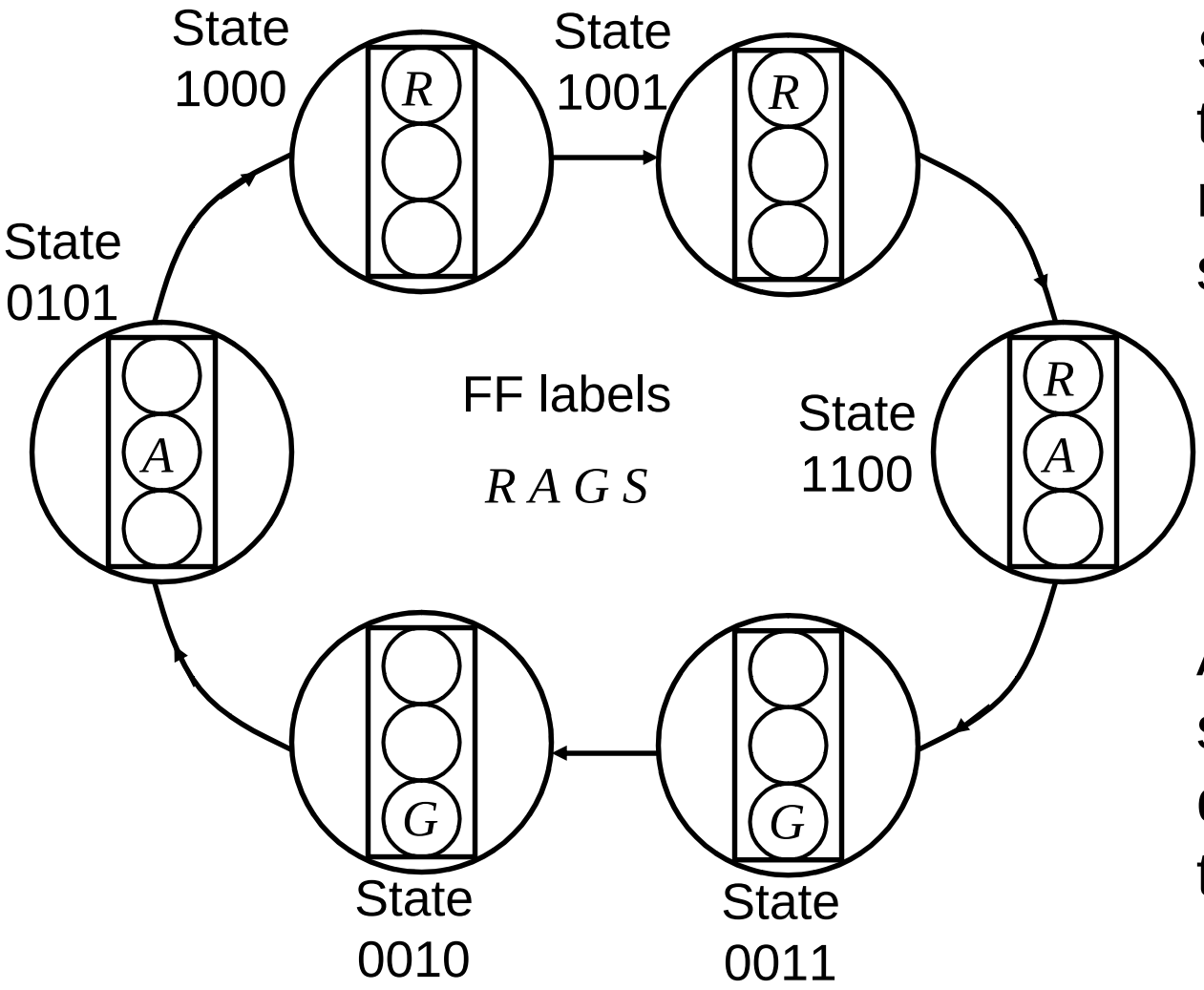
Which are all valid states

So it does self start

Example 2

- We extend Example 1 so that the traffic signals spend extra time for the R and G lights
- Essentially, we need 2 additional states, i.e., 6 in total.
- In theory, the 3 FF machine gives us the potential for sufficient states
- However, to make the machine combinational logic easier, it is more convenient to add another FF (labelled S), making 4 in total

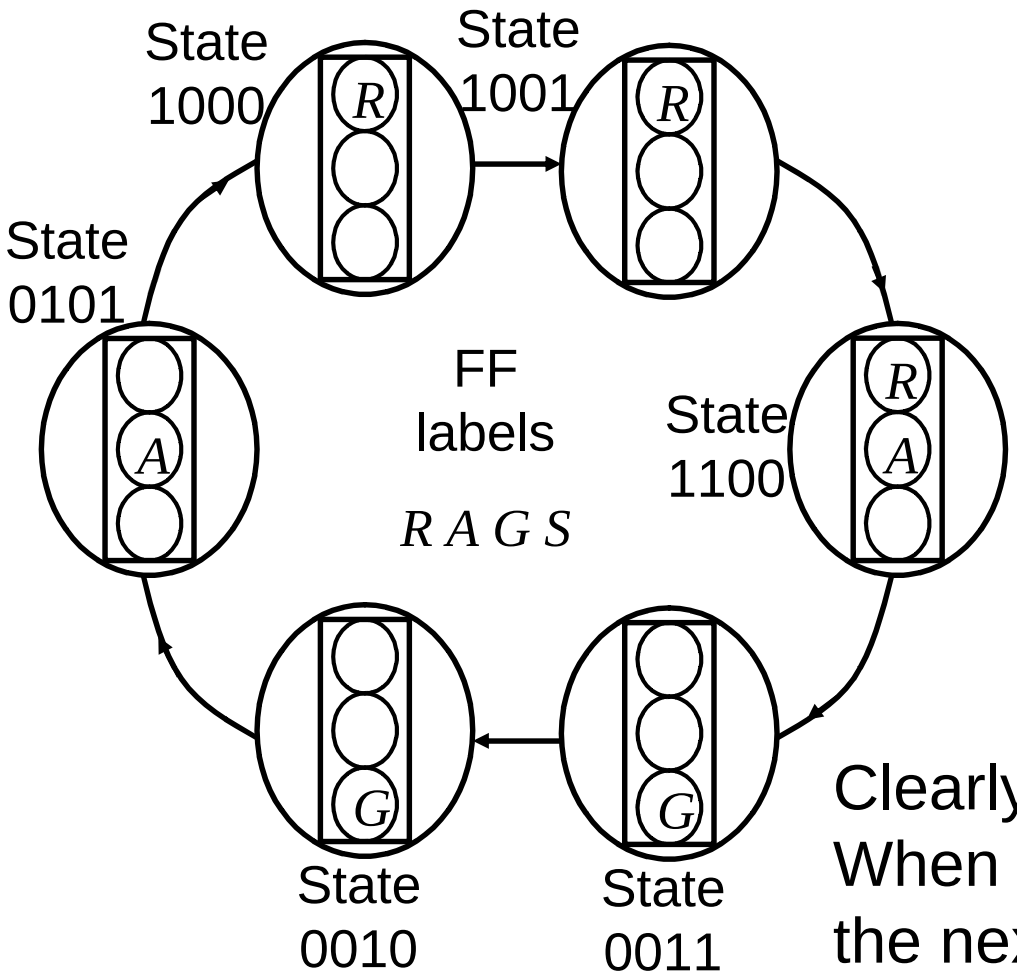
Example 2



See that new FF toggles which makes the next state logic easier

As before, the first step is to write down the state transition table

Example 2



Current state				Next state			
R	A	G	S	R'	A'	G'	S'
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	1
0	1	0	1	1	0	0	0

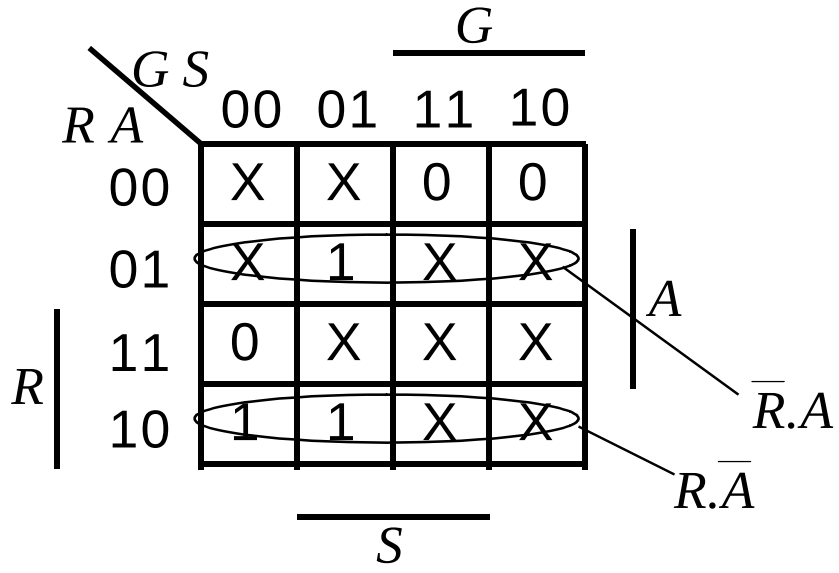
Clearly a lot of unused states. When plotting k-maps to determine the next state logic it is probably easier to plot 0s and 1s in the map and then mark the unused states

Example 2

Current state				Next state			
R	A	G	S	R'	A'	G'	S'
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	1
0	1	0	1	1	0	0	0

We will now use k-maps to determine the next state combinational logic

For the R FF, we need to determine D_R



$$D_R = R.\overline{A} + \overline{R}.A = R \oplus A$$

Example 2

Current state				Next state			
R	A	G	S	R'	A'	G'	S'
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	1
0	1	0	1	1	0	0	0

We can plot k-maps for D_A and D_G to give:

$$D_A = R.S + G.\overline{S} \quad \text{or}$$

$$D_A = R.S + \overline{R}.\overline{S} = \overline{R \oplus S}$$

$$D_G = R.A + G.S \quad \text{or}$$

$$D_G = G.S + A.\overline{S}$$

By inspection we can also see:

$$D_S = \overline{S}$$

State Assignment

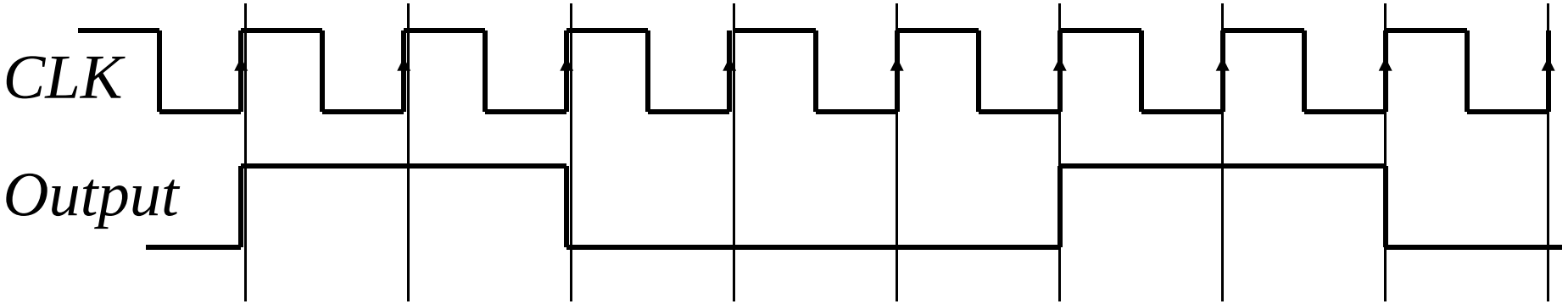
- As we have mentioned previously, state assignment is not necessarily obvious or straightforward
  - Depends what we are trying to optimise, e.g.,
    - Complexity (which also depends on the implementation technology, e.g., FPGA, 74 series logic chips).
      - FF implementation may take less chip area than you may think given their gate level representation
      - Wiring complexity can be as big an issue as gate complexity
    - Speed
  - Algorithms do exist for selecting the ‘optimising’ state assignment, but are not suitable for manual execution

State Assignment

- If we have m states, we need at least $\log_2 m$ FFs (or more informally, bits) to encode the states, e.g., for 8 states we need a min of 3 FFs
- We will now present an example giving various potential state assignments, some using more FFs than the minimum

Example Problem

- We wish to investigate some state assignment options to implement a divide by 5 counter which gives a 1 output for 2 clock edges and is low for 3 clock edges



Sequential State Assignment

- Here we simply assign the states in an increasing natural binary count
- As usual we need to write down the state transition table. In this case we need 5 states, i.e., a minimum of 3 FFs (or state bits). We will designate the 3 FF outputs as c, b, and a
- We can then determine the necessary next state logic and any output logic.

Sequential State Assignment

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0

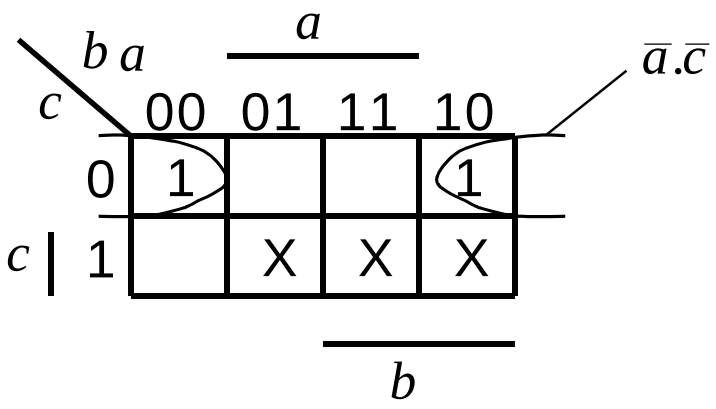
Unused states, 101, 110 and 111.

By inspection we can see:

The required output is from FF b

Plot k-maps to determine the next state logic:

For FF a:



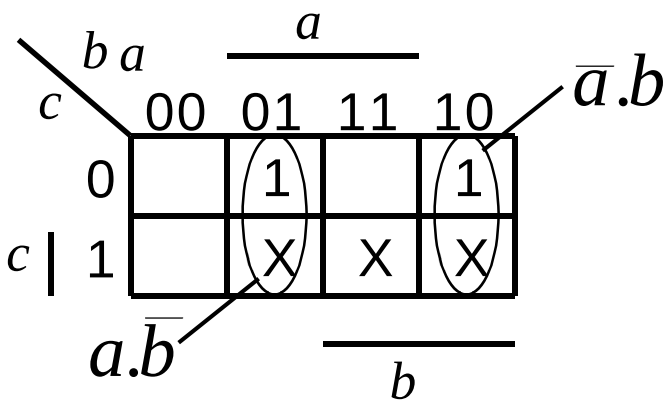
$$D_a = \bar{a}.\bar{c}$$

Sequential State Assignment

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0

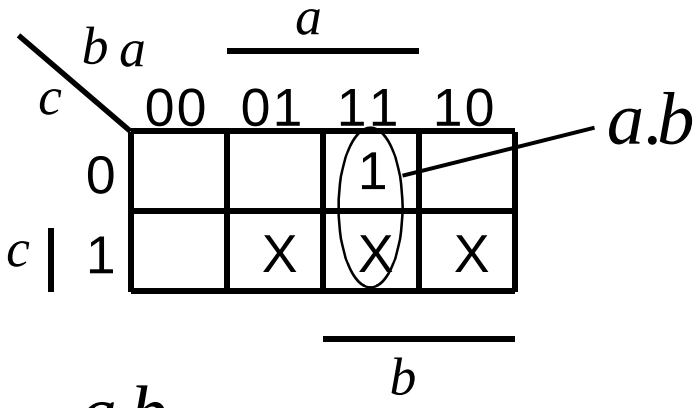
Unused states, 101, 110 and 111.

For FF b:



$$D_b = \bar{a}.b + a.\bar{b} = a \oplus b$$

For FF c:



$$D_c = a.b$$

Sliding State Assignment

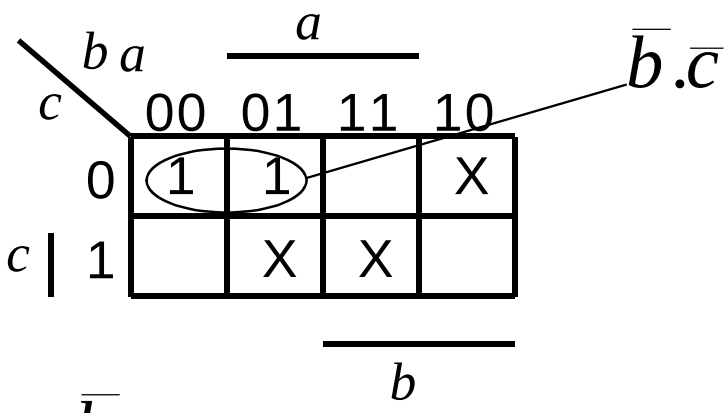
Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	1	1	0
1	1	0	1	0	0
1	0	0	0	0	0

Unused states, 010, 101, and 111.

By inspection we can see that we can use any of the FF outputs as the wanted output

Plot k-maps to determine the next state logic:

For FF a:



$$D_a = \bar{b}.\bar{c}$$

Sliding State Assignment

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	1	1	0
1	1	0	1	0	0
1	0	0	0	0	0

Unused states, 010, 101, and 111.

By inspection we can see that:

For FF b:

$$D_b = a$$

For FF c:

$$D_c = b$$

Shift Register Assignment

- As the name implies, the FFs are connected together to form a shift register. In addition, the output from the final shift register in the chain is connected to the input of the first FF:
  - Consequently the data continuously cycles through the register

Shift Register Assignment

Current state					Next state				
e	d	c	b	a	e'	d'	c'	b'	a'
0	0	0	1	1	0	0	1	1	0
0	0	1	1	0	0	1	1	0	0
0	1	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0	1	1

Unused states. Lots!

Because of the shift register configuration and also from the state table we can see that:

$$D_a = e$$
$$D_b = a$$
$$D_c = b$$
$$D_d = c$$
$$D_e = d$$

By inspection we can see that we can use any of the FF outputs as the wanted output

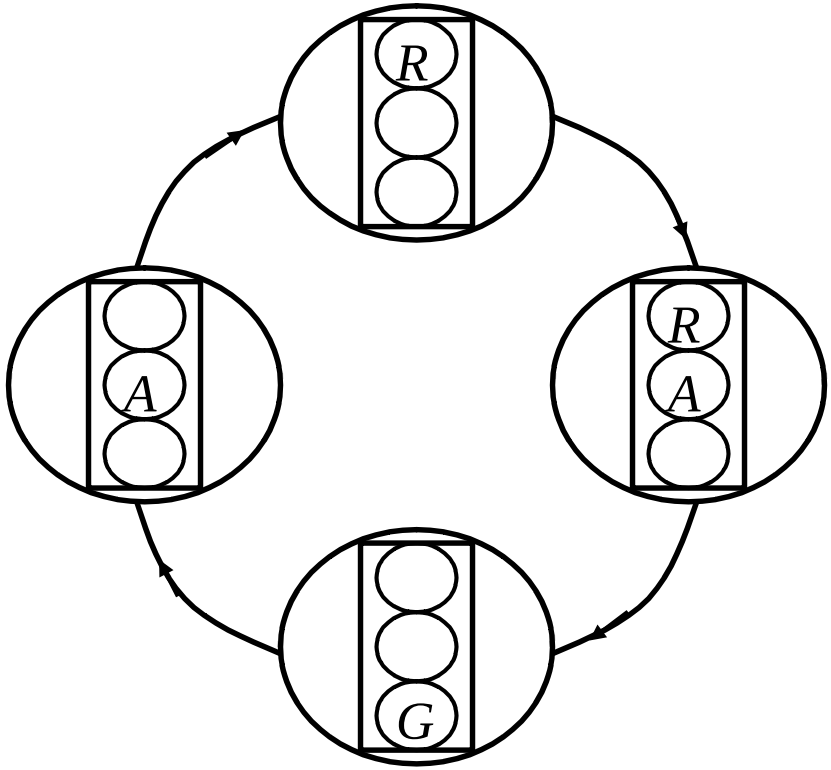
See needs 2 more FFs, but no logic and simple wiring

One Hot State Encoding

- This is a shift register design style where only FF at a time holds a 1
- Consequently we have 1 FF per state, compared with $\log_2 m$ for sequential assignment
- However, can result in simple fast state machines
- Outputs are generated by ORing together appropriate FF outputs

One Hot - Example

- We will return to the traffic signal example, which recall has 4 states

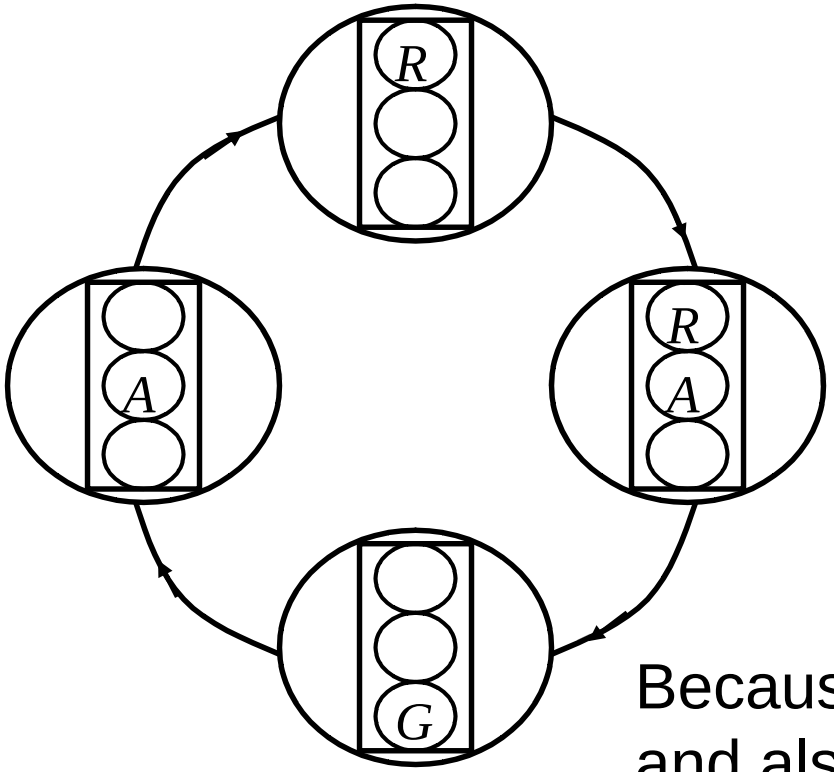


For 1 hot, we need 1 FF for each state, i.e., 4 in this case

The FFs are connected to form a shift register as in the previous shift register example, however in 1 hot, only 1 FF holds a 1 at any time

We can write down the state transition table as follows

One Hot - Example



Current state				Next state			
r	ra	g	a	r'	ra'	g'	a'
1	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0

Unused states. Lots!

Because of the shift register configuration and also from the state table we can see that: $D_a = g \quad D_g = ra \quad D_{ra} = r \quad D_r = a$

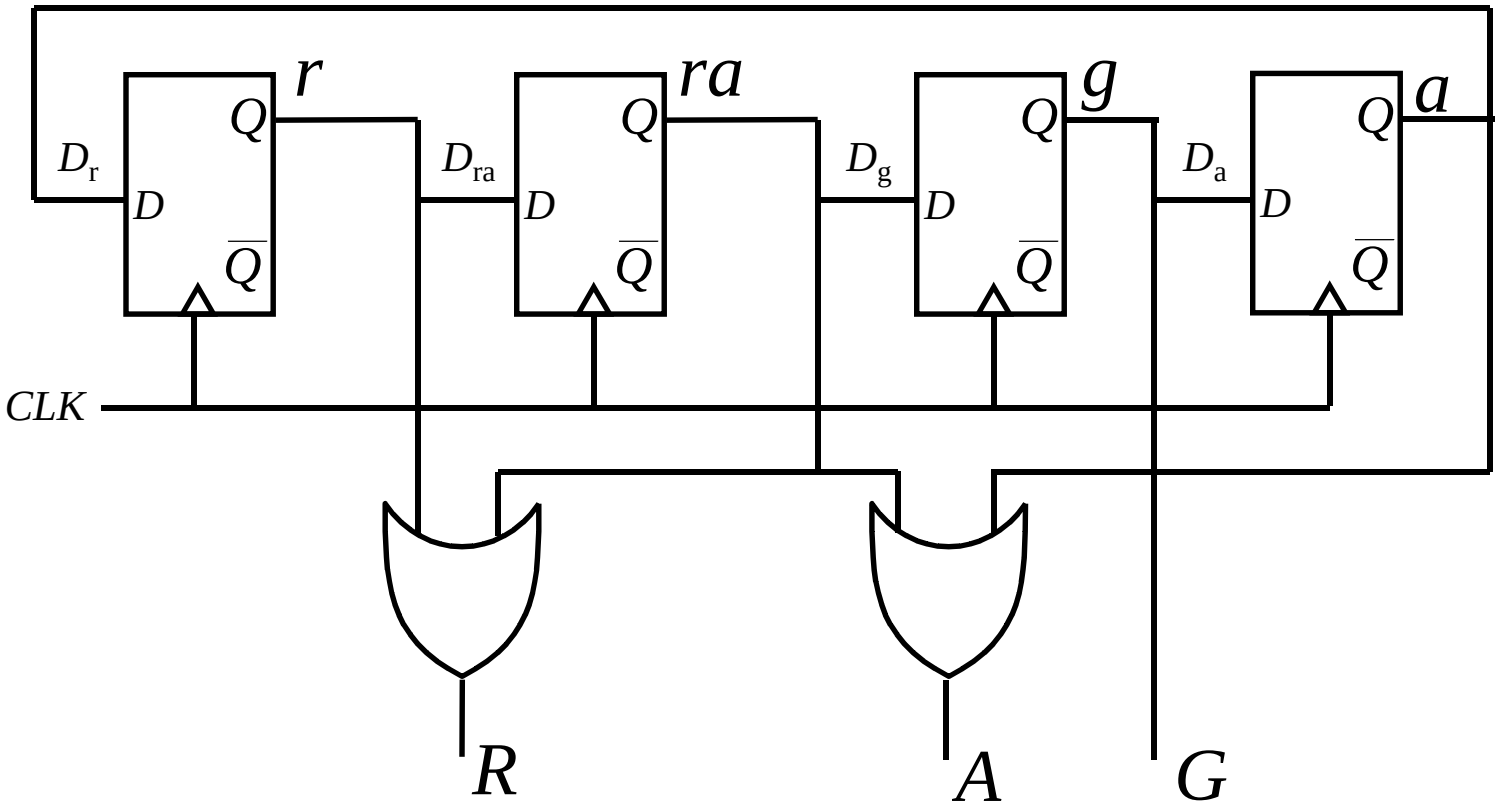
To generate the R, A and G outputs we do the following ORing:

$$R = r + ra \qquad A = ra + a \qquad G = g$$

One Hot - Example

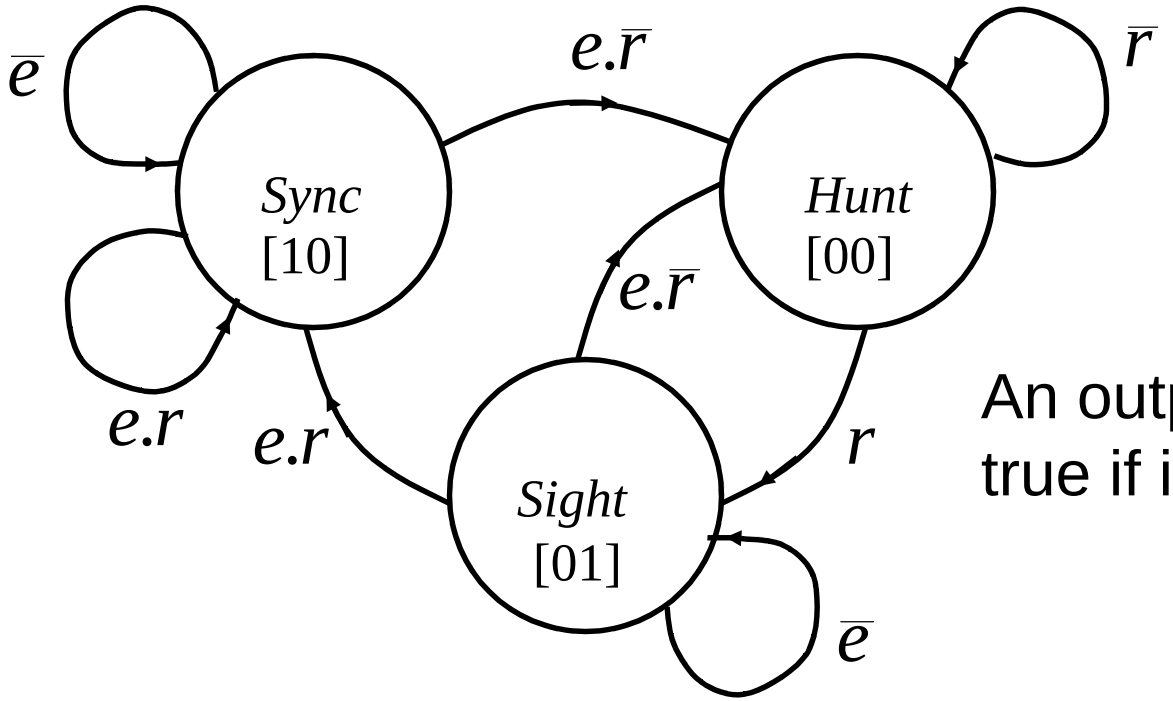
$$D_a = g \quad D_g = ra \quad D_{ra} = r \quad D_r = a$$

$$R = r + ra \qquad A = ra + a \qquad G = g$$



Tripos Example

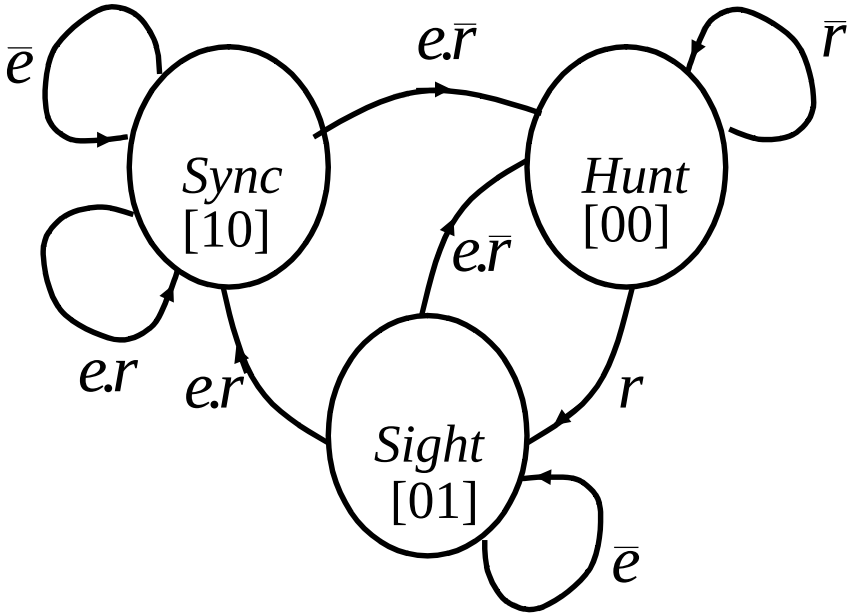
- The state diagram for a synchroniser is shown. It has 3 states and 2 inputs, namely e and r. The states are mapped using sequential assignment as shown.



FF labels
$[s_1\ s_0]$

An output, s should be true if in *Sync* state

Tripos Example



Unused state 11

From inspection, $s = s_1$

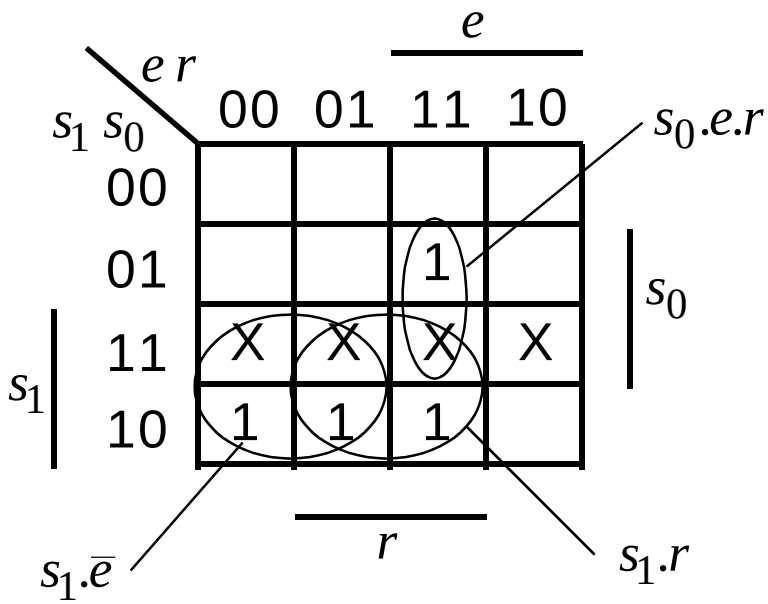
Current state		Input		Next state	
s_1	s_0	e	r	s_1'	s_0'
0	0	X	0	0	0
0	0	X	1	0	1
0	1	0	X	0	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	X	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	X	X	X	X

Tripos Example

Current state		Input		Next state	
s_1	s_0	e	r	s_1'	s_0'
0	0	X	0	0	0
0	0	X	1	0	1
0	1	0	X	0	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	X	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	X	X	X	X

Plot k-maps to determine the next state logic

For FF 1:



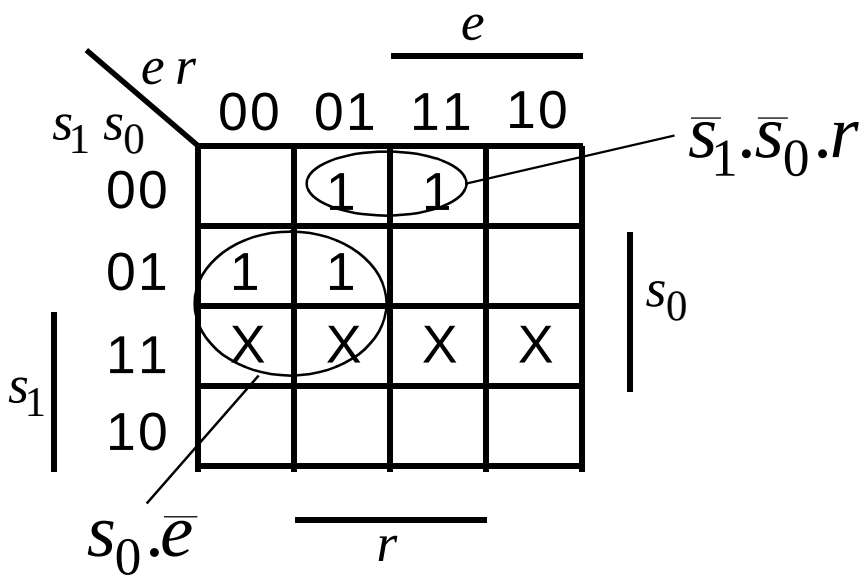
$$D_1 = s_1.\bar{e} + s_1.r + s_0.e.r$$

Tripos Example

Current state		Input		Next state	
s_1	s_0	e	r	s_1'	s_0'
0	0	X	0	0	0
0	0	X	1	0	1
0	1	0	X	0	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	X	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	X	X	X	X

Plot k-maps to determine the next state logic

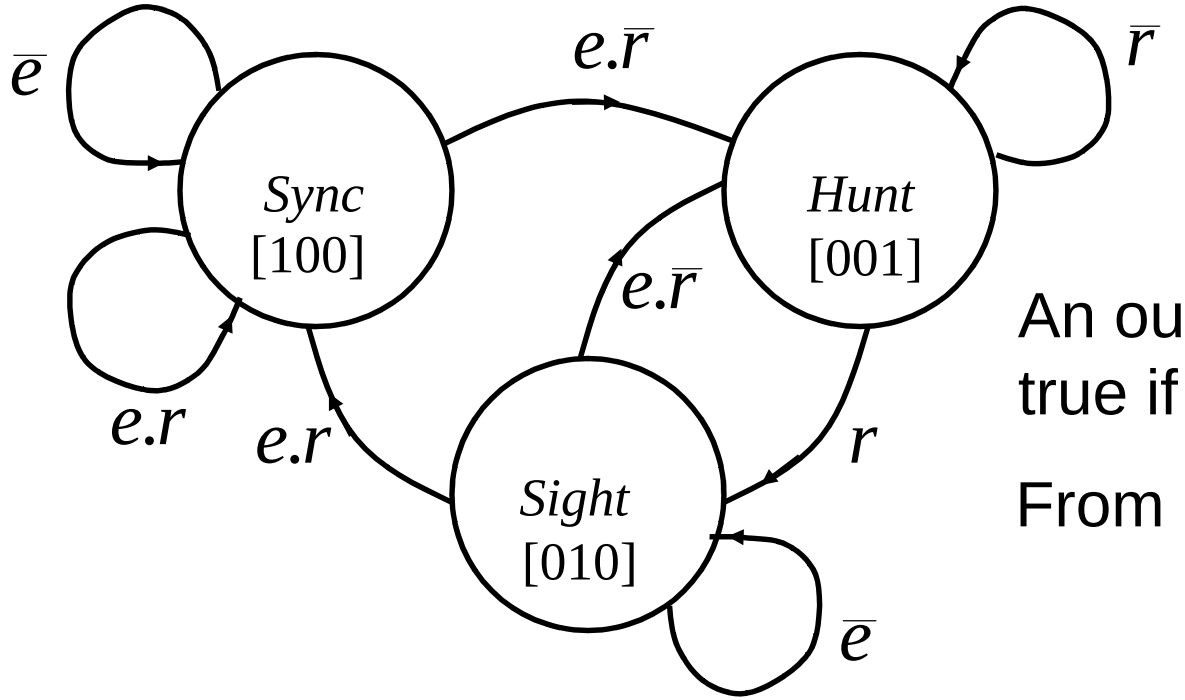
For FF 0:



$$D_0 = s_0.\bar{e} + \bar{s}_1.\bar{s}_0.r$$

Tripos Example

- We will now re-implement the synchroniser using a 1 hot approach
- In this case we will need 3 FFs

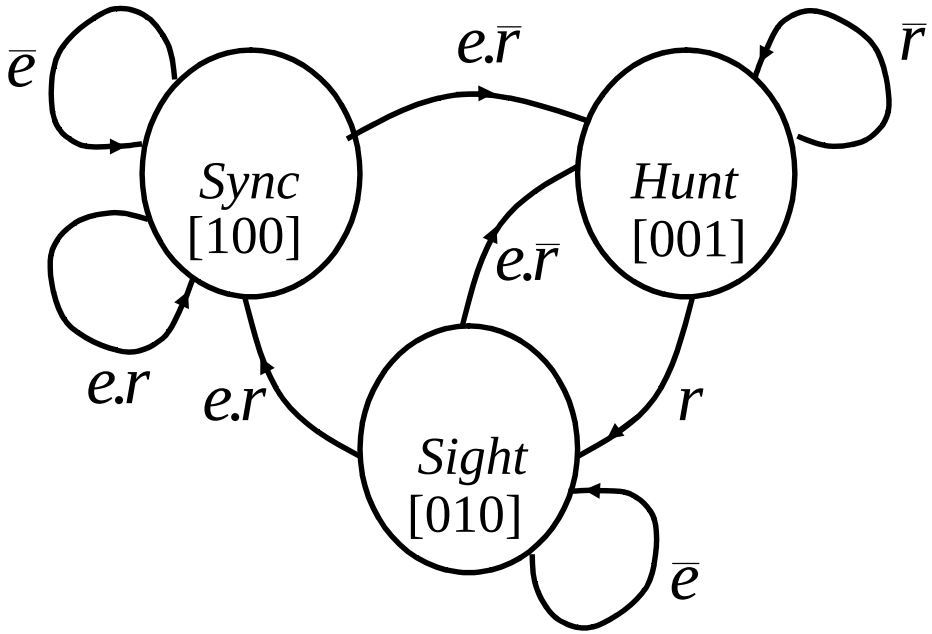


FF labels
$[s_2\ s_1\ s_0]$

An output, s should be true if in *Sync* state

From inspection, $s = s_2$

Tripos Example



Current state			Input		Next state		
s_2	s_1	s_0	e	r	s_2'	s_1'	s_0'
0	0	1	X	0	0	0	1
0	0	1	X	1	0	1	0
0	1	0	0	X	0	1	0
0	1	0	1	0	0	0	1
0	1	0	1	1	1	0	0
1	0	0	0	X	1	0	0
1	0	0	1	0	0	0	1
1	0	0	1	1	1	0	0

Remember when interpreting this table, because of the 1-hot shift structure, only 1 FF is 1 at a time, consequently it is straightforward to write down the next state equations

Tripos Example

Current state			Input		Next state		
s_2	s_1	s_0	e	r	s_2'	s_1'	s_0'
0	0	1	X	0	0	0	1
0	0	1	X	1	0	1	0
0	1	0	0	X	0	1	0
0	1	0	1	0	0	0	1
0	1	0	1	1	1	0	0
1	0	0	0	X	1	0	0
1	0	0	1	0	0	0	1
1	0	0	1	1	1	0	0

For FF 2:

$$D_2 = s_1.e.r + s_2.\bar{e} + s_2.e.r$$

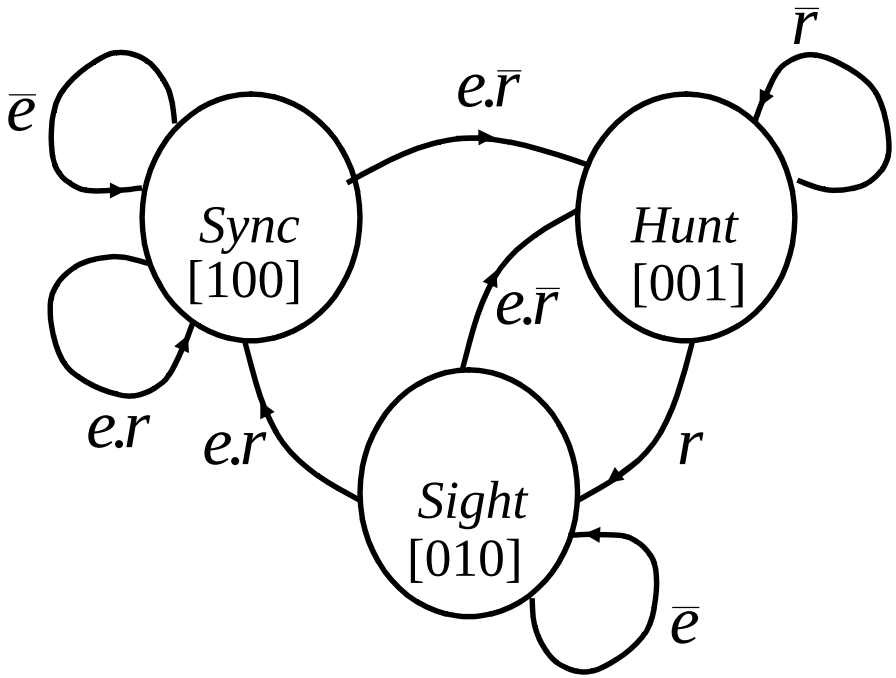
For FF 1:

$$D_1 = s_0.r + s_1.\bar{e}$$

For FF 0:

$$D_0 = s_0.\bar{r} + s_1.e.\bar{r} + s_2.e.\bar{r}$$

Tripos Example



Note that it is not strictly necessary to write down the state table, since the next state equations can be obtained from the state diagram

It can be seen that for each state variable, the required equation is given by terms representing the incoming arcs on the graph

For example, for FF 2: $D_2 = s_1.e.r + s_2.\bar{e} + s_2.e.r$

Also note some simplification is possible by noting that:

$s_2 + s_1 + s_0 = 1$ (which is equivalent to e.g., $s_2 = \overline{s_1 + s_0}$)

Tripos Example

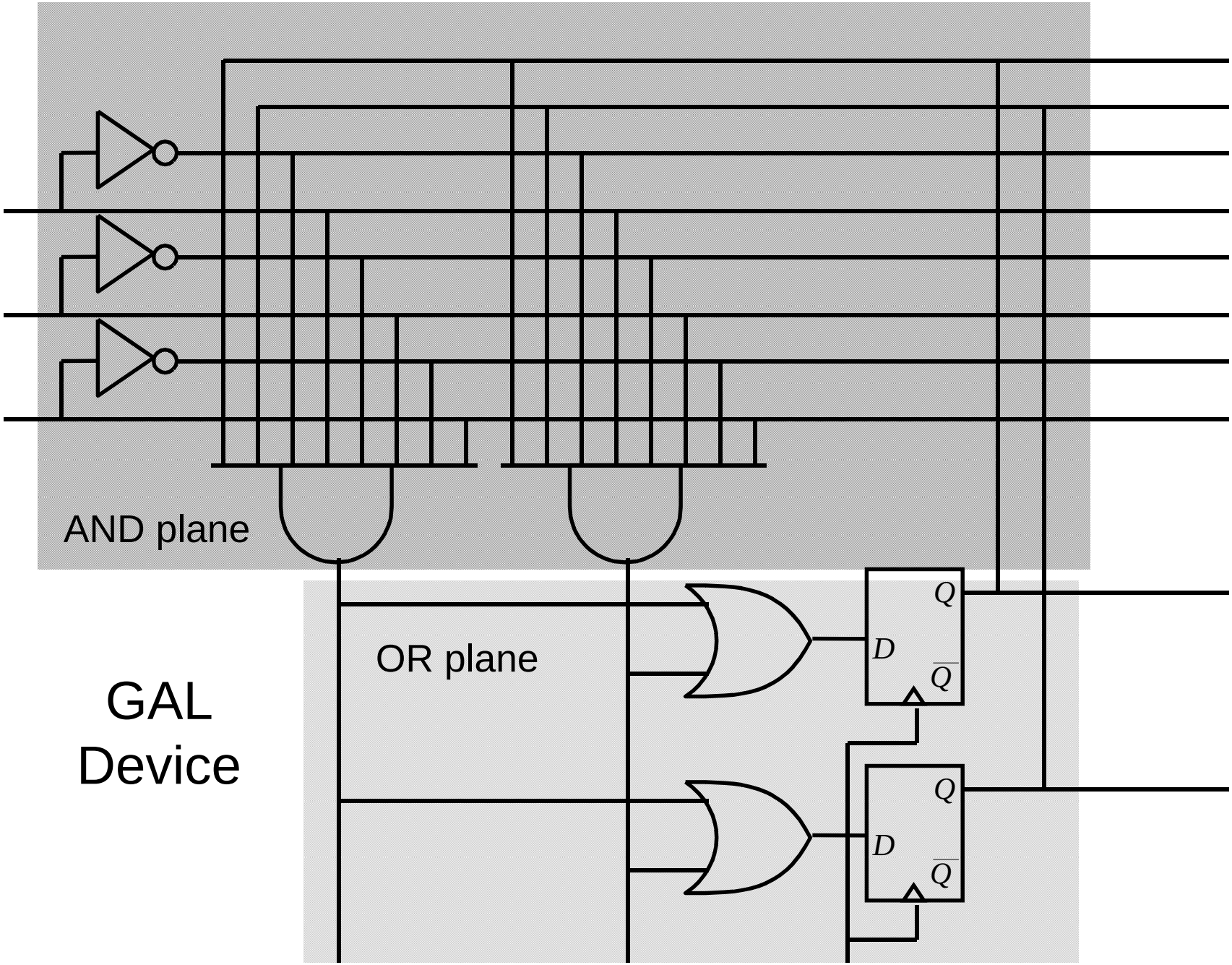
- So in this example, the 1 hot is easier to design, but it results in more hardware compared with the sequential state assignment design

Implementation of FSMs

- We saw previously that programmable logic can be used to implement combinational logic circuits, i.e., using PAL devices
- PAL style devices have been modified to include D-type FFs to permit FSMs to be implemented using programmable logic
- One particular style is known as Generic Array Logic (GAL)

GAL Devices

- They are similar in concept to PALs, but have the option to make use of a D-type flip-flops in the OR plane (one following each OR gate). In addition, the outputs from the D-types are also made available to the AND plane (in addition to the usual inputs)
  - Consequently it becomes possible to build programmable sequential logic circuits


AND plane
OR plane
GAL
Device
Q
D
Q
Q
D
Q

FPGA

- Field Programmable Gate Array (FPGA) devices are the latest type of programmable logic
- Are a sea of programmable wiring and function blocks controlled by bits downloaded from memory
- Function units contain a 4-input 1 output look-up table with an optional D-FF on the output