Digital Electronics

Dr. I. J. Wassell

Digital Electronics

Introduction

Aims

- To familiarise students with
  - Combinational logic circuits
  - Sequential logic circuits
  - How digital logic gates are built using transistors
  - Simple processor architectures
  - The design, build and testing of digital logic systems

Course Structure

- 12 Lectures
- Hardware Labs
  - 4 Workshops
  - Each Workshop lasts 2.5/3 h
  - In Intel Lab. (SW11), William Gates Building (WGB)
  - Done individually
  - Lab. Sessions begin in week 3 of M Term and run throughout the L Term

Objectives

- At the end of the course you should
  - Be able to design, build and test 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 design finite 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 (1st Ed.), 2012 (2nd Ed.).
  - 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.

Simulation Software

- There are a number of packages available that enable simulation of digital electronic circuits using a graphical interface e.g.,
  - National Instruments (NI) Multisim
  - Yenka Electronics (Technology Package)
- The former is much more powerful (and expensive), but the latter is cheap and relatively straightforward to use
- If you download Yenka, you can use the Dept. activation key to unlock it

Other Points

- This course is a prerequisite for
  - Introduction to Computer Architecture, ECAD and Architecture Practical Classes (Part IB)
  - Advanced Computer Architecture (Part II)
  - Advanced Topics in Computer Architecture (MPhil/Part III)
- Keep up with lab work and get it ticked.
- Have a go at supervision questions plus any others your supervisor sets.
- Remember to attempt questions from past papers
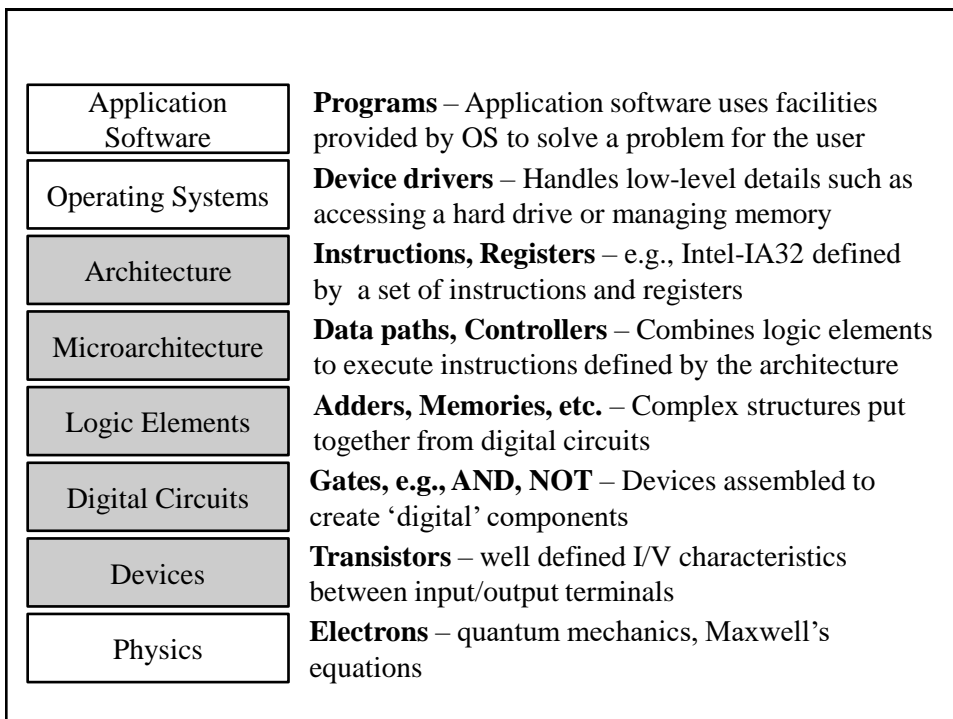
The Bigger Picture

- As you may be aware, probably the most significant application of digital logic is to implement *microprocessors* and microprocessor based computer systems.

- However, digital logic is also employed to build a wide variety of *other* electronic systems that are not microprocessor based.

Managing Complexity

- Modern digital systems e.g., microprocessors, are typically built from millions of transistors.

- It would be impossible for a human to design such a system by for example, writing equations describing the movement of electrons in each transistor and then attempting to solve these equations simultaneously.

- Consequently, we have to manage complexity so that we are not swamped in a mass of detail.

- To do this we employ *abstraction*.

Abstraction

- Abstraction, i.e., hiding details when they are not important.
- Indeed a system can be viewed from many different levels of abstraction.
- For example, for an electronic computing system, we can consider levels of abstraction from pure physics (electrons) at the bottom level through to application software (programs) at the top level.
- In this course we will address topics in the shaded boxes in the following figure.

Application Software	**Programs** – Application software uses facilities provided by OS to solve a problem for the user
Operating Systems	**Device drivers** – Handles low-level details such as accessing a hard drive or managing memory
Architecture	**Instructions, Registers** – e.g., Intel-IA32 defined by a set of instructions and registers
Microarchitecture	**Data paths, Controllers** – Combines logic elements to execute instructions defined by the architecture
Logic Elements	**Adders, Memories, etc.** – Complex structures put together from digital circuits
Digital Circuits	**Gates, e.g., AND, NOT** – Devices assembled to create 'digital' components
Devices	**Transistors** – well defined I/V characteristics between input/output terminals
Physics	**Electrons** – quantum mechanics, Maxwell's equations

Abstraction

- So, you can browse the web without any regard quantum theory or the organisation of memory in the computer.
- That said, when working at a particular level of abstraction, it is good to know something about the levels of abstraction immediately above and below where you are working, e.g.,
  - A device designer needs to understand the circuits in which it will be used,
  - Code cannot be optimised without understanding the architecture for which it is being written.

Course Structure

- We will begin by looking at so called *combinational logic* and some basic logic functions, e.g., NOT, OR, AND.
- We note that for a combinational logic function, its current output depends only on its current inputs.
- Then we consider Boolean algebra and apply it and other approaches to simplify logic functions, with the aim of reducing implementation complexity.

Course Structure

- We will then see how we can design logic functions to perform multi-bit arithmetic. The trade-off between complexity and speed of operation will be explored.
- Other effects of finite gate propagation delay include the potential for undesirable changes in the output of combinational logic circuits (hazards). An approach to eliminate them with the use of additional logic will be described.

Course Structure

- The final purely combinational logic topics that we will address include more complex functions such as multiplexers, demultiplexers and programmable devices, e.g., PALs.
- We will then move on to consider *sequential logic*, where the current output not only depends on the current inputs, but also on previous inputs, i.e., we have introduced the notion of memory.

Course Structure

- Initially, we will consider *asynchronous* devices such as latches, before moving on the consider *synchronous* devices, e.g., flip-flops, that will only change output state under the control of a global enabling (clock) signal.
- We will then look in detail at the design of synchronous finite state machines (FSMs) using flip-flops.
- We will then see how logic gates are implemented using CMOS transistors.
- Finally, we will briefly introduce microprocessor architecture and design.

Digital Electronics: Combinational Logic

Logic Gates and Boolean Algebra

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	Boolean

a	y
0	1
1	0
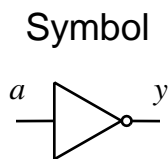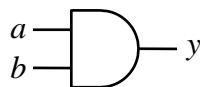
$$y = \bar{a}$$
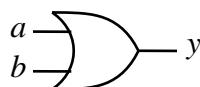
- A NOT gate is also called an 'inverter'
- y is only TRUE if a is FALSE
- Circle (or 'bubble') on the output of a gate implies that it as an inverting (or complemented) output

AND Gate

| Symbol | Truth-table | Boolean |

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

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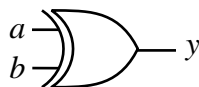
- y is only TRUE only if a is TRUE and b is TRUE
- In Boolean algebra AND is represented by a dot **.**

OR Gate

| Symbol | Truth-table | Boolean |

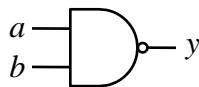| a b	y
0 0	0
0 1	1
1 0	1
1 1	1

$$y = a + b$$

- 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   Boolean

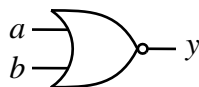| a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a \oplus b$$

- 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   Boolean

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

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

- 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   Boolean

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

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

- 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

$$v = \overline{b}.c.p$$
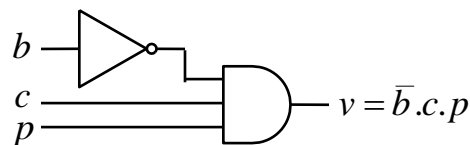
Boolean Algebra

- 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

<div align="center">

OR   AND

</div>

$$a + 0 = a \qquad a.0 = 0$$
$$a + a = a \qquad a.a = a$$
$$a + 1 = 1 \qquad a.1 = a$$
$$a + \bar{a} = 1 \qquad a.\bar{a} = 0$$

- 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
  $$a.(b + c + \ldots) = (a.b) + (a.c) + \ldots$$
  $$a + (b.c\ldots.) = (a + b).(a + c).\ldots \quad \text{NEW}$$
- Absorption
  $$a + (a.c) = a \quad \text{NEW}$$
  $$a.(a + c) = a \quad \text{NEW}$$

Boolean Algebra

- Consensus theorem

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

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

  Note that this theorem can be used to add or eliminate terms when simplifying a Boolean expression

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:

$$a + a.b = a$$

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

Boolean Algebra - Example

Simplify

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

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

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

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

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

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

Boolean Algebra - Example

Prove consensus theorem

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

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

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

$a.b + \bar{a}.c$

Boolean Algebra - Example

Using consensus theorem

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

Eliminating consensus terms gives

$$\bar{a}.\bar{b} + a.c + b.\bar{c}$$

DeMorgan's Theorem

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

- 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.,

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

DeMorgan's Theorem

- For 2 variables we can show $\overline{a+b} = \bar{a}.\bar{b}$ and $\overline{a.b} = \bar{a}+\bar{b}$ using a truth table.

a b	$\overline{a+b}$	$\overline{a.b}$	$\bar{a}$ $\bar{b}$	$\bar{a}.\bar{b}$	$\bar{a}+\bar{b}$
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

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

DeMorgan's Examples

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

$$= a.\bar{b} + a.\bar{b}.\bar{c} + b.\bar{b}.\bar{c} \quad \text{(DeMorgan)}$$
$$= a.\bar{b} + a.\bar{b}.\bar{c} \quad (\text{b.}\bar{b} = 0)$$
$$= a.\bar{b} \quad \text{(absorbtion)}$$

DeMorgan's Examples

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

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

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

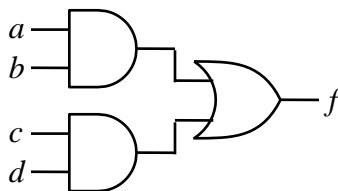$= (a.b + \overline{a} + \overline{b}).c.d$ (distribute)

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

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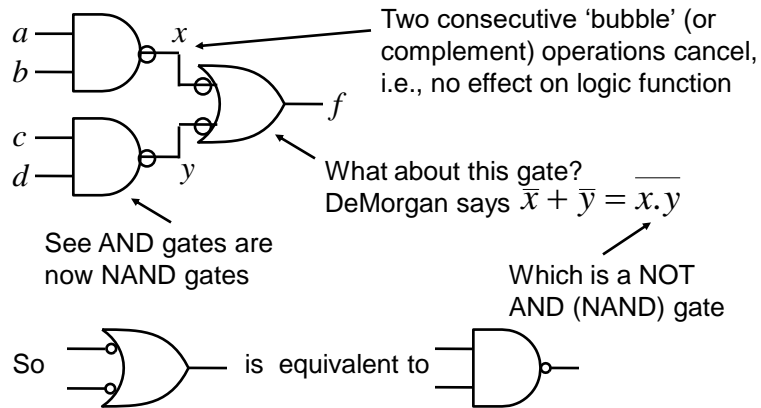

DeMorgan's in Gates

- 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
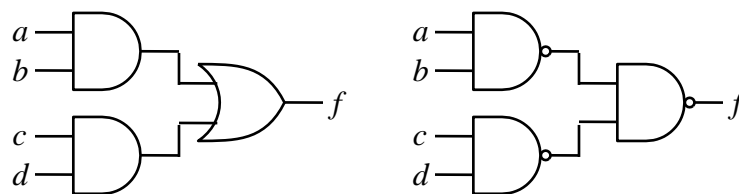
DeMorgan's in Gates

- To do this we can use 'bubble' logic

a, b, x

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

c, d, y, f

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
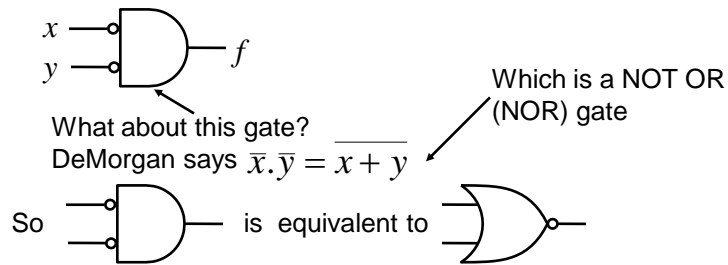
a, b, f

c, d

a, b, f

c, d

$$f = a.b + c.d$$

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

DeMorgan's in Gates

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

x ⎯⊸⎤
y ⎯⊸⎦D⎯ f

Which is a NOT OR
(NOR) gate

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

So ⊸⎤D⎯⊸ is equivalent to ⟩D⊸⎯

- Useful if trying to build using NOR gates

Digital Electronics: Combinational Logic

Logic Minimisation

Introduction

- 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.,

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

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

$$\begin{array}{ccc|c} x & y & z & f \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}$$

$x \diagdown yz$	00	01	11	10
0	1	1	1	1
x| 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?

$x \diagdown yz$	00	01	11	10
0	1	1	1	1
x| 1			1	

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

- 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,

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

K-maps – 4 variables

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

$c\,d$ / $a\,b$	00	01	11	10
00				
01	1	1	1	1
11	1			
10				

- 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.
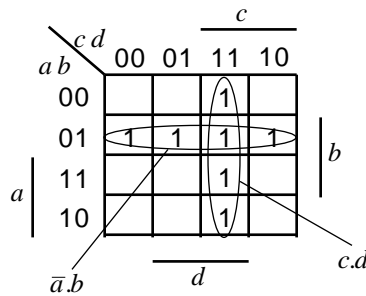
K-maps – 4 variables

- For example, plot

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

$c\,d$ / $a\,b$	00	01	11	10
00	1	1	1	1
01				
11				
10	1	1	1	1

$c\,d$ / $a\,b$	00	01	11	10
00	1			1
01				
11				
10	1			1

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

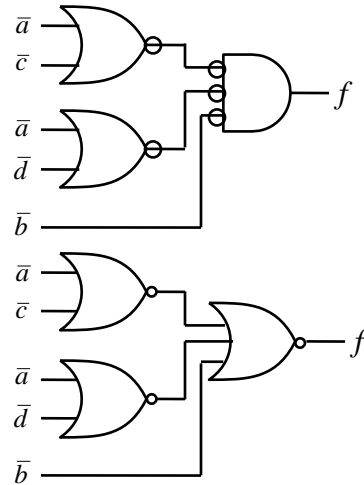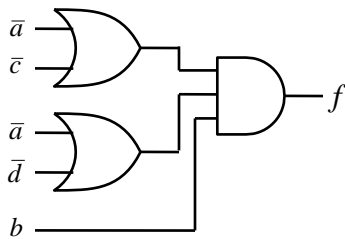- Simplify $f = \bar{a}.b + b.\bar{c}.\bar{d}$ into POS form.

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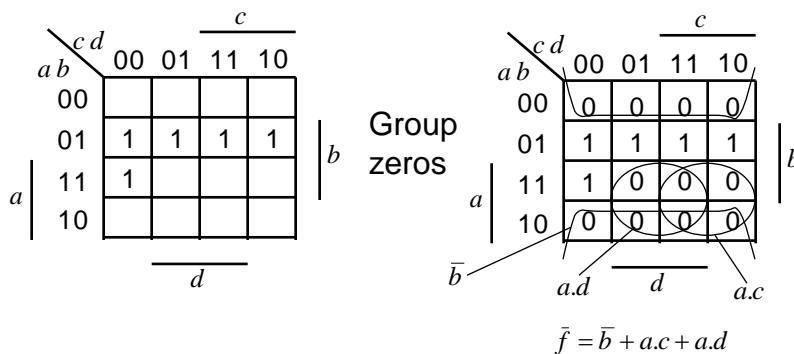
$$\bar{f} = \bar{b} + a.c + a.d$$

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, $\bar{a}.\bar{b}.\bar{c}.\bar{d}, \bar{a}.\bar{b}.c.\bar{d}, \bar{a}.b.\bar{c}.d$
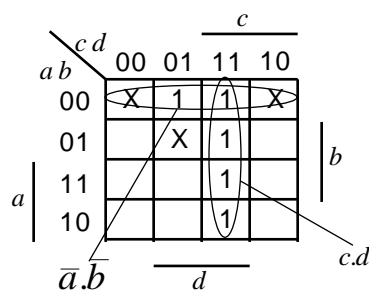
See only need to include Xs if they assist in making a bigger group, otherwise can ignore.

$f = \bar{a}.\bar{b} + c.d$ **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 Prime Implicant – 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

Some Definitions - Example
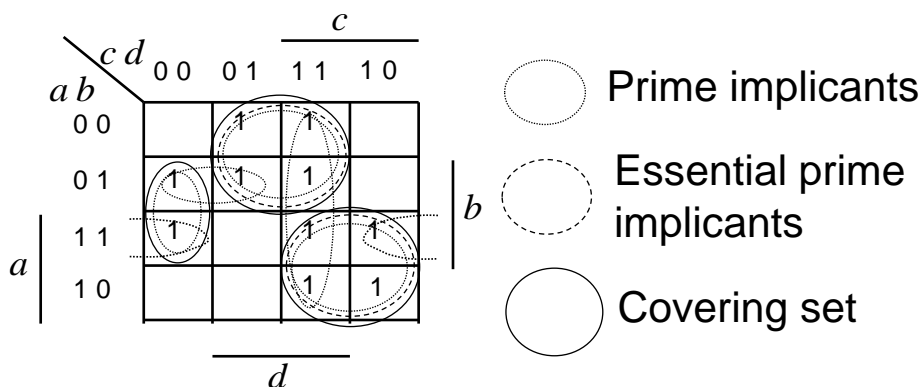
Tabular Simplification

- Except in special cases or for sparse truth tables, the K-map method is not practical beyond 6 variables
- A systematic approach known as the *Quine-McCluskey (Q-M) Method* finds the minimised representation of any Boolean expression
- It is a tabular method that ensures all the prime implicants are found and can be automated for use on a computer

Q-M Method

- The Q-M Method has 2 steps:
  - First a table, known as the *QM implication table*, is used to find all the prime implicants;
  - Next the minimum cover set is found using the *prime implicant* chart.
- We will use a 4 variable example to show the method in operation:
  - Minterms are: 4,5,6,8,9,10,13
  - Don't cares are: 0,7,15.

Q-M Method

- The first step is to list all the minterms and don't cares in terms of their minterm indices represented as a binary number
  - Note the entries are grouped according to the number of 1s in the binary representation
  - The 1st column contains the minterms
  - After applying the method, the 2nd column will contain 3 variable terms. Similarly for subsequent columns.

Q-M Method

- The method begins by listing groups of minterms and don't cares in groups containing ascending numbers of 1s with a blank line between the groups
  - Thus the first group has zero ones, the second group has a single 1 and the third has two 1s and so on
- We next apply the so called *uniting theorem* iteratively as follows

Q-M Method – Uniting Theorem

- Compare elements in the 1st group (no 1s) with all elements in the 2nd group. If they differ by a single bit, it means the terms are adjacent (think K-map)
- Adjacent terms are placed in the 2nd column with the single bit that differs replaced by a dash (-). Terms in the 1st column that contribute to a term in the second are *ticked*, i.e., they are *not* prime implicants.
- Now compare elements in the 2nd group with those in the 3rd group. Once again, terms that differ by a single bit are moved to the 2nd column and have that bit replaced by a dash (-).

Q-M Method – Uniting Theorem

- Continue the previous procedure until element comparisons between all adjacent groups in the 1st column have been done.
- Mark any unticked elements in the 1st column with an asterix (*). This means that they are prime implicants.
- Now repeat previous procedure for the groups in the 2nd column
- As before groups must differ only by a single bit but they must also have a – in the same position
- Groups in 2nd column that do not contribute to the 3rd column are marked with an asterix (*), i.e., they are prime implicants
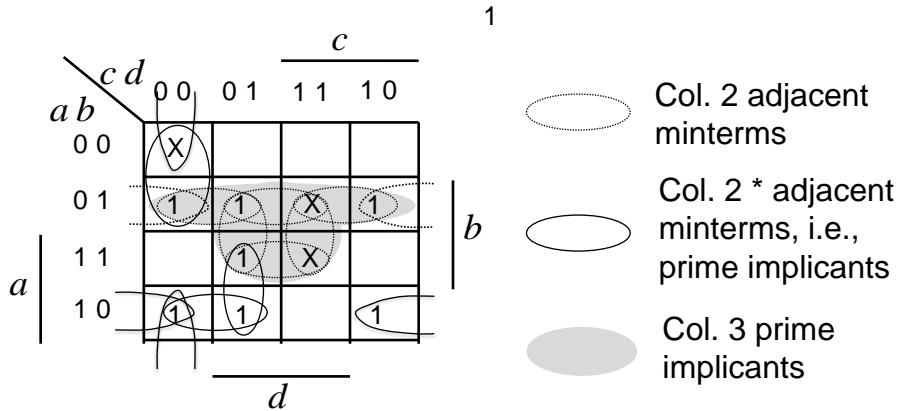
Q-M Method – Uniting Theorem

- Now repeat previous procedure for the groups in the 3nd column
- Note we need to find elements that differ in 1 bit position, but also have 2 dashes in the same positions.
- In this case, we have no terms that satisfy this condition, so we terminate the procedure and mark any unticked terms with asterisks to indicate that they are prime implicants.

Q-M – Implication Table

- Minterms are: 4,5,6,8,9,10,13
- Don't cares are: 0,7,15.

Column 1	Column 2	Column 3
0 0 0 0 ✓	0 - 0 0 *	0 1 - - *
	- 0 0 0 *	
0 1 0 0 ✓		- 1 - 1 *
1 0 0 0 ✓	0 1 0 - ✓	
0 1 0 1 ✓	0 1 - 0 ✓	
0 1 1 0 ✓	1 0 0 - *	
1 0 0 1 ✓	1 0 - 0 *	
1 0 1 0 ✓	0 1 - 1 ✓	
	- 1 0 1 ✓	
0 1 1 1 ✓	0 1 1 - ✓	
1 1 0 1 ✓	1 - 0 1 *	
1 1 1 1 ✓	- 1 1 1 ✓	
	1 1 - 1 ✓	

K-map view of Q-M example

Col. 2 adjacent minterms

Col. 2 * adjacent minterms, i.e., prime implicants

Col. 3 prime implicants

Q-M – Finding Min Cover

– The second step is to find the lowest number of prime implicants that cover the function – this is achieved using the *prime implicant chart*

– This chart is organised as follows:

  • Label columns with the minterm indices (don't include don't cares)

  • Label rows with minterms covered by a given prime implicant. To do this dashes (-) in a prime implicant are replaced by all combinations of 0s and 1s

  • Place an X in the (row, column) location if the minterm represented by the column index is covered by the prime implicant associated with the row

  • The next slide shows the initial prime implicant chart

Q-M – Prime Implicant Chart

```
                              4 5 6 8 9 10 13
              0,4 (0-0 0)  X
              0,8 (-0 0 0)          X                Minterms (exc.
* Terms in    8,9 (1 0 0 -)         X X              don't cares)
Implication
Table         8,10 (1 0-0)          X    X
              9,13 (1-0 1)             X     X
          4,5,6,7 (0 1- -) X X X
       5,7,13,15 (- 1-1)     X             X
```

- Now we look for the *essential prime implicants* – These are indicated when there is only a single X in any column, i.e., This means there is a minterm covered by only one prime implicant

---

Q-M – Prime Implicant Chart

- The essential terms must be included in the final cover
  - Draw lines in the column and row that have a X associated with an essential prime implicant and draw a box around the prime
  - Minterms intersected by lines are also covered by the essential primes

```
                          4 5 6 8 9 10 13
          0,4 (0-0 0)  X
          0,8 (-0 0 0)         X
          8,9 (1 0 0 -)        X X
        | 8,10 (1 0-0) | ······X··(X)······
          9,13 (1-0 1)            X    X
      | 4,5,6,7 (0 1- -) | X X (X)·····
     5,7,13,15 (- 1-1)     X          X
```

Q-M – Prime Implicant Chart

- The essential prime implicants usually cover additional minterms.
  - We must also cross out any columns that have an X in a row associated with an essential prime since these minterms are already covered by the essential primes
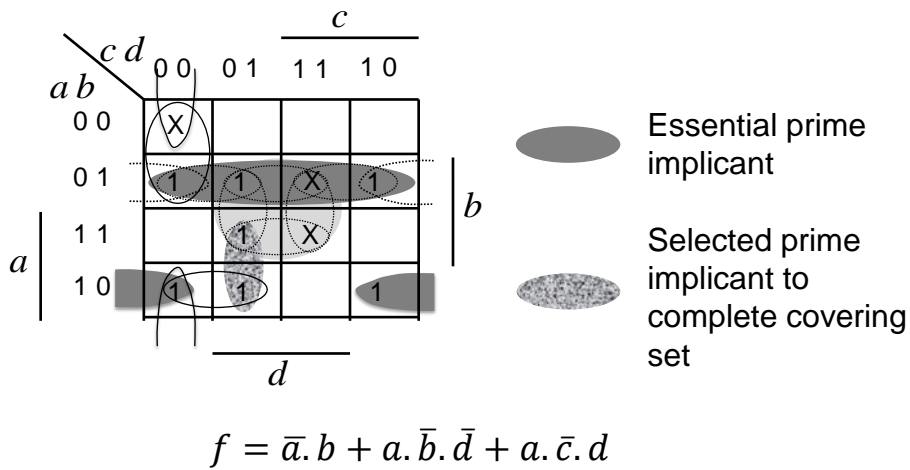

Q-M – Prime Implicant Chart

- We see 2 minterms are still uncovered (cols. 9 and 13)
  - The final step is to find as few primes as possible to cover the remaining minterms
  - We see the single prime implicant 1-01 covers both of them
  - The boxed terms show the final covering set

Final K-Map view of Q-M Example

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

Digital Electronics: Combinational Logic

Binary Adders

Introduction

- 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

a ——[ ]—— sum
b ——[ ]—— c_{out}
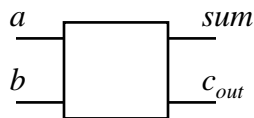
- By inspection:

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

Full Adder

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

a ——[ ]—— sum
b ——[ ]—— c_{out}
c_{in}
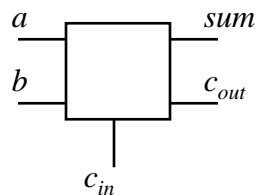
- Has the following truth table:

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

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

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

From DeMorgan

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

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

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

So,

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

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

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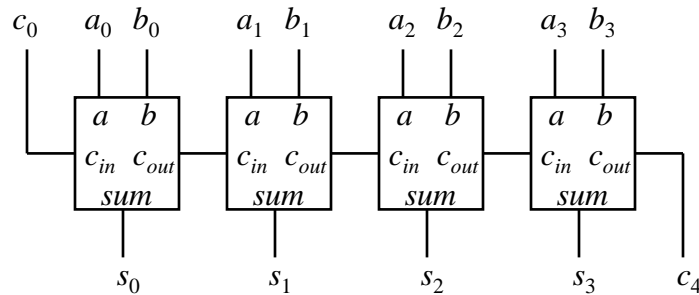
- Note: If we complement a and set c_o to one we have implemented $s = b - a$

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
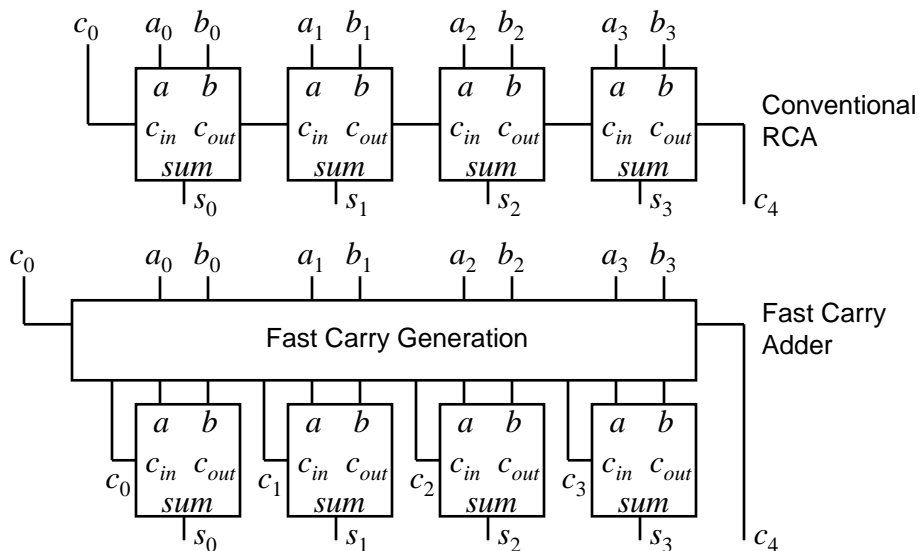
Fast Carry Generation

- 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*

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

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), $s_i = a_i \oplus b_i \oplus c_i$
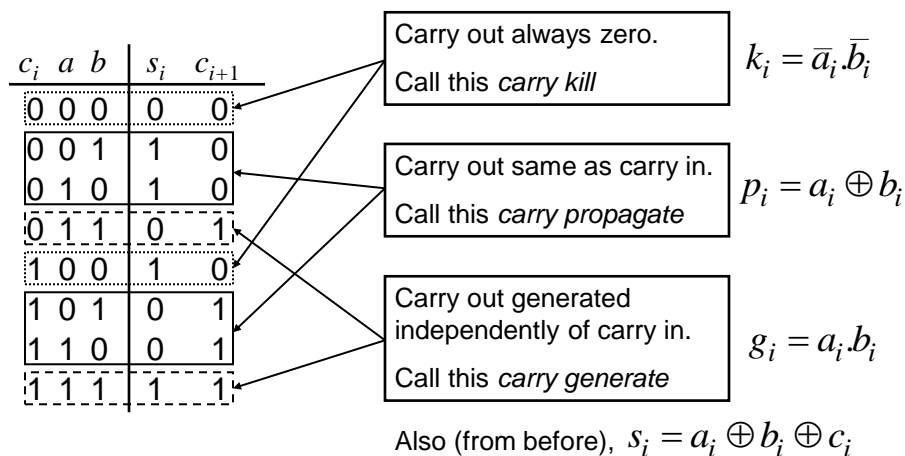
Fast Carry Generation

- 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., $i = 0$,

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

$$c_4 = G + Pc_0$$

where,

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

$$P = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

- 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
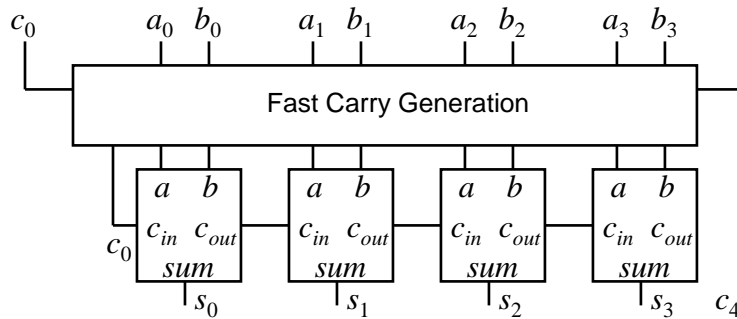
Fast Carry Generation
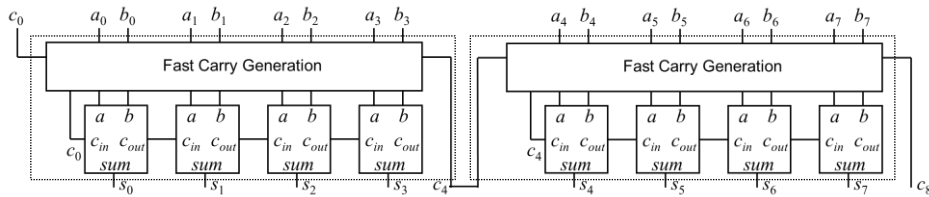
- Conventional ripple carry within 4-bit blocks
- Fast carry generation between 4-bit blocks
- Trade-off between complexity and speed

Digital Electronics: Combinational Logic

Multilevel Logic and Hazards

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
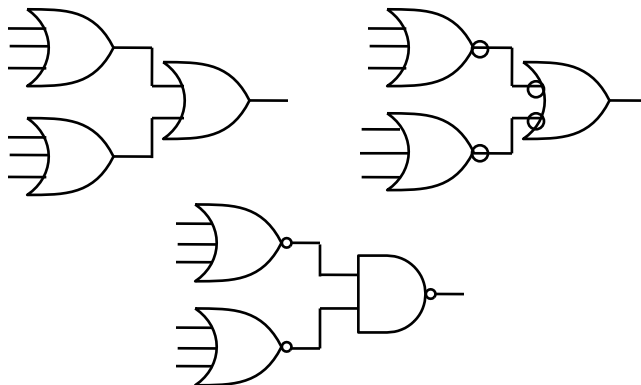
Common Expression Elimination

- 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:

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

- 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 *one input changes* 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
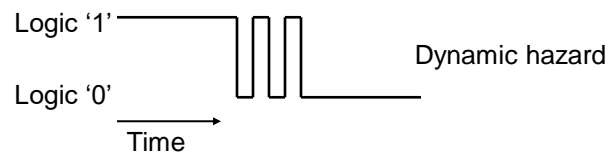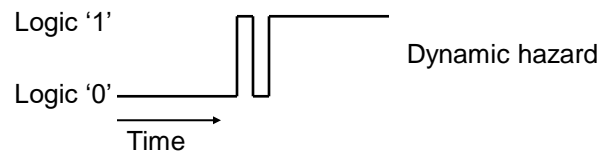
Logic '1'

Logic '0'
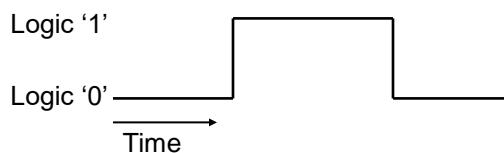
Time

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.
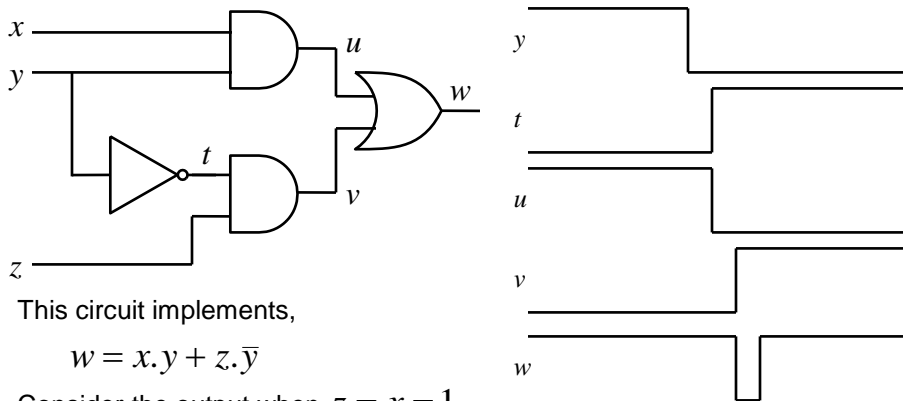
Static Hazard

Logic '1'
Logic '0'
Time

Static 1 hazard

Logic '1'
Logic '0'
Time

Static 0 hazard

Dynamic Hazard

Logic '1'
Logic '0'
Time

Dynamic hazard

Logic '1'
Logic '0'
Time
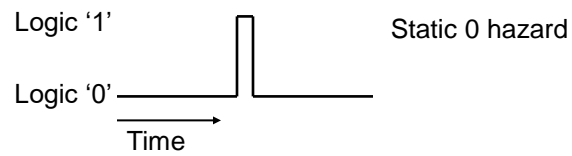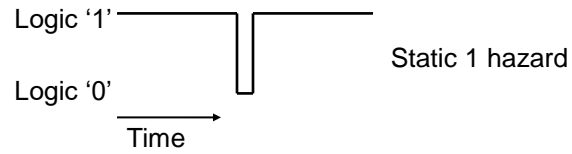
Dynamic hazard

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)
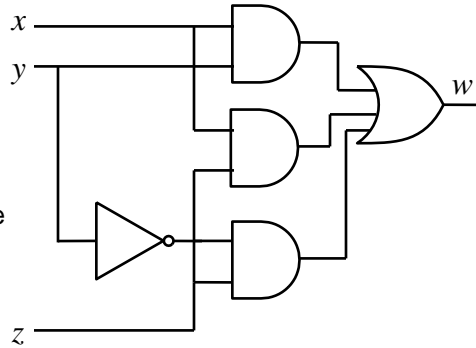- To remove dynamic hazards – not covered in this course!

Removing the static 1 hazard

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

Extra term added to remove hazard, consequently,

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

# Digital Electronics: Combinational Logic

Beyond Simple Logic Gates

Multiplexer

- **Multiplexer (Mux)/selector** – chooses 1 of many inputs to steer to its single output under the direction of control inputs, e.g., if the input to a circuit can come from several places a Mux is one way to funnel the multiple sources selectively to the single output.

Multiplexer

- The hazard example is actually a 2-to-1 (2:1) Mux, i.e., it can select either input x or z to appear at output w under control of y
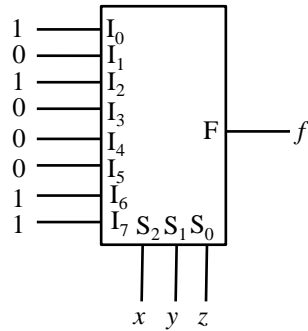

x	z	y	w
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1
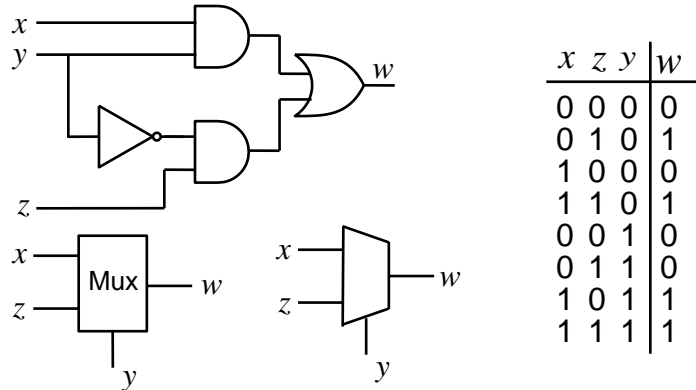
Multiplexer

- Clearly an n-to-1 (n:1) Mux is also possible. For example, an 8-to-1 (8:1) Mux will need 3 control inputs.
- A Mux can also be used to implement combinational logic functions. For example, an 8 input Mux can be used to implement functions having 3 variables expressed as a sum of minterms, i.e., DNF.

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

Multiplexer

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

```
1 ──── I₀
0 ──── I₁
1 ──── I₂
0 ──── I₃
0 ──── I₄     F ──── f
0 ──── I₅
1 ──── I₆
1 ──── I₇  S₂ S₁ S₀
              │  │  │
              x  y  z
```

- The control inputs are used to select the minterms required at the output. The Mux is sometimes called a hardware *look-up* table.
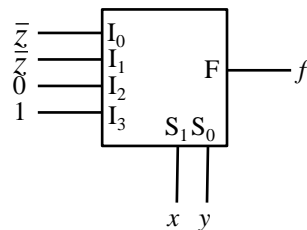

Multiplexer

- In this example, if we use one of the logic variables as input to the multiplexer, then we can get away with a 4-to-1 (4:1) Mux

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

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

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

```
z̄ ──── I₀
z̄ ──── I₁     F ──── f
0 ──── I₂
1 ──── I₃   S₁ S₀
              │  │
              x  y
```

Multiplexer

- We see it can also be designed via a truth table based approach, e.g.,

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

$I_0 = \bar{z}$

$I_1 = \bar{z}$

$I_2 = 0$

$I_3 = 1$
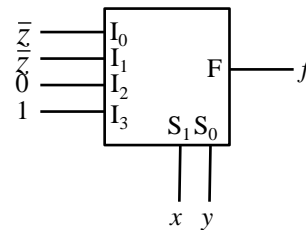
Demultiplexer

- A demultiplexer is the opposite of a Mux, i.e., a single input is directed to exactly one of its outputs
- The truth table for a 1-to-2 (1:2) Demux, i.e., 1 control input and 2 outputs is:
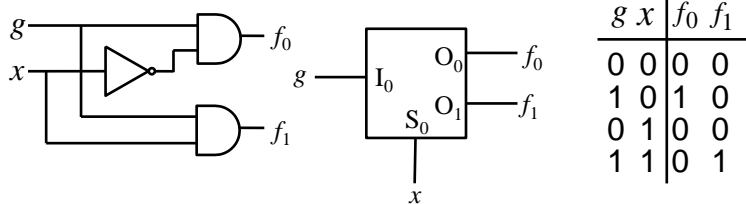
g	x	f_0	f_1
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

Demultiplexer

- Clearly a larger Demux are also possible. For example, a 3-to-8 (3:8) Demux has 3 control inputs and 8 outputs.
- A related function is a *Decoder*. In this case the input g is permanently connected to a logic 1. This yields a 1-of-2 decoder (also known as a 1:2 decoder)

g	x	f_0	f_1
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

$g = 1$

x	f_0	f_1
0	1	0
1	0	1

- See only one output is logic 1 at a time

Decoder

- Clearly an 1-of-n Decoder is possible. For example, a 1-of-8 Decoder (i.e., a 3:8 demux) has 3 control inputs and 8 outputs.
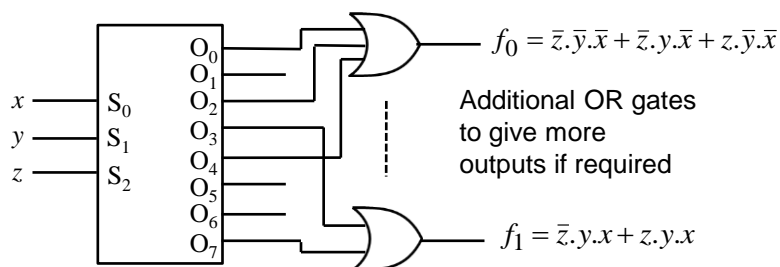- A typical application would be to 'Enable (EN)' 1 out-of-n logic sub-systems.
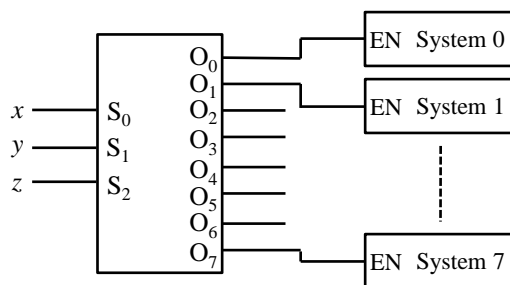
- So, letting $x=1$, $y=z=0$ will enable System 1

Decoder

- We can see that a 1-of-n Decoder will generate all the possible minterms having n variables.

- Consequently, a logical expression having DNF form can be implemented by ORing together the required minterms at the decoder output.

- Multiple output logic blocks can be created by using multiple OR gates at the decoder output, i.e., one for each output.

Decoder

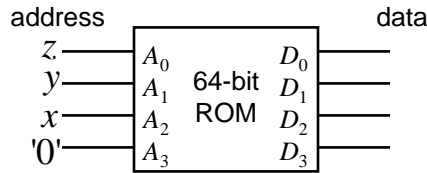- Decoder implementation of a 3 variable, 2 output combinational logic block.

$$f_0 = \bar{z}.\bar{y}.\bar{x} + \bar{z}.y.\bar{x} + z.\bar{y}.\bar{x}$$

Additional OR gates to give more outputs if required

$$f_1 = \bar{z}.y.x + z.y.x$$

Even More Ways to Implement Combinational Logic

- We have seen how combinational logic can be implemented using logic gates (e.g., AND, OR), Mux and Demux.

- 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   data

	A_0		D_0	
z	A_0	64-bit	D_0	
y	A_1	ROM	D_1	
x	A_2		D_2	
'0'	A_3		D_3	

address (decimal)	x	y	z	f	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 logic array (PLA)
- In PLAs, 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 PLA Structure

a

b

c

AND plane

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

OR plane

f_0

f_1

f_2
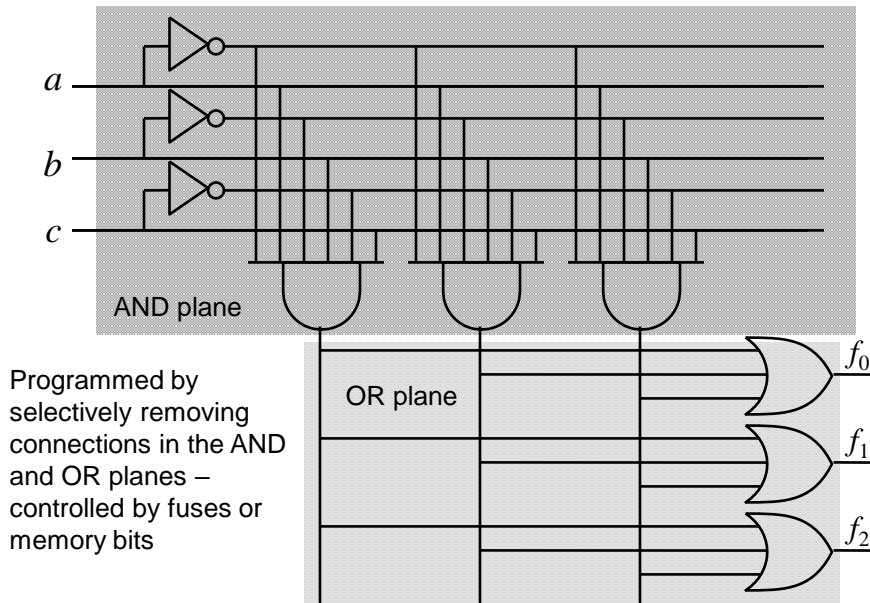
Other PLA Style Structures

- In PLAs, only the required minterms are generated using a separate AND plane. Output from this plane are available to all OR gates to give the final output
- A modified structure known as Programmable Array Logic (PAL) does not have a programmable OR array and so outputs from the AND array can not be shared among the OR gates to give the final outputs.
- This simplifies the structure, but at the cost of lower efficiency

Basic PAL Structure

OR plane

AND plane

a

b

c

f_0

f_n
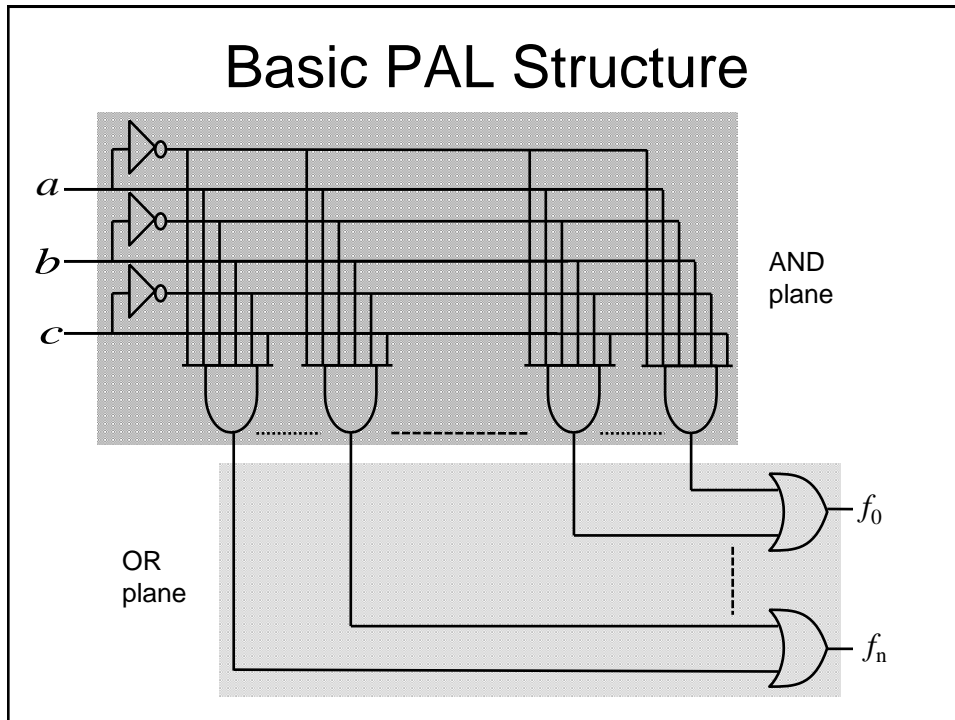
Other Memory Devices

- 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 same 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

Symbol Bus line

Output Enable (OE) = 1

OE = 0
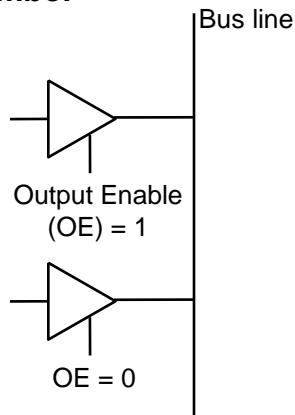
Functional analogy Bus line

OE = 1

OE = 0
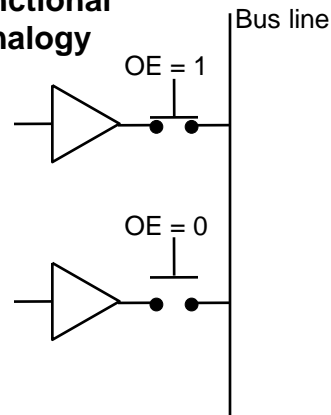
Control Signals

- 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

Digital Electronics: Sequential Logic

Introduction, Latches and Flip-Flops

Introduction

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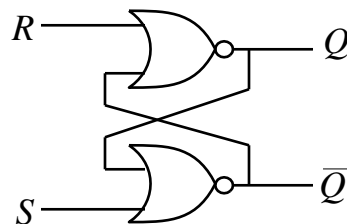
S R	Q'	$\overline{Q}'$	comment
0 0	Q	$\overline{Q}$	hold
0 1	0	1	reset
1 0	1	0	set
1 1	0	0	illegal

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

RS Latch - Operation
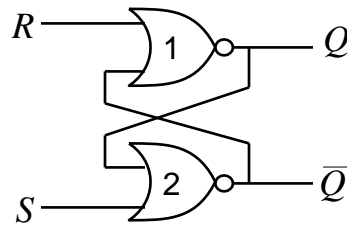
NOR truth table

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 $\overline{Q} = 1$
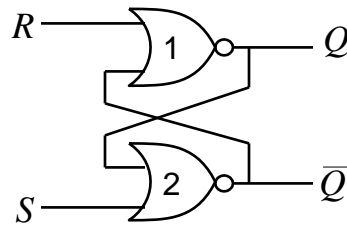- This is the (R)eset condition

RS Latch - Operation
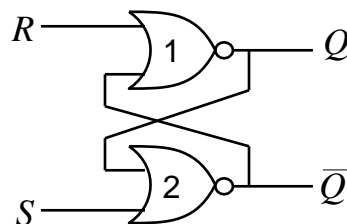
NOR truth table

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
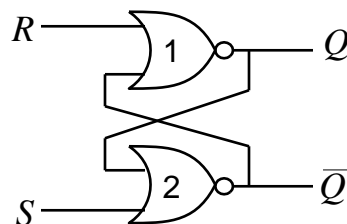
NOR truth table

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:

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

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

$Q = 0 \quad Q' = 1$

From the table we can see:

$S.\bar{R}$

RS Latch – State Diagram

- Which gives the following state diagram:

$$S.\bar{R}$$

$$\bar{S} + R \quad \boxed{Q = 0} \qquad \boxed{Q = 1} \quad \bar{R}$$

$$R$$
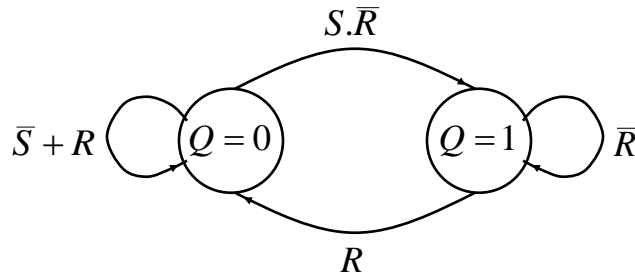
- A similar diagram can be constructed for the $\bar{Q}$ output

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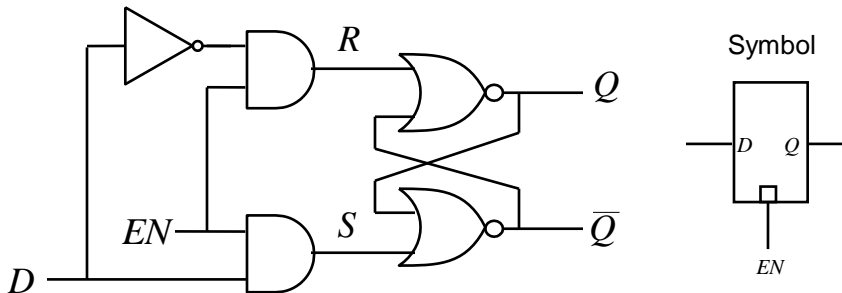
- See from the AND truth table:
  - if one of the inputs, say a is 0, the output is always 0
  - Output follows b input if a is 1
- The complement function ensures that R and S can never be 1 at the same time, i.e., illegal avoided

AND truth table

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

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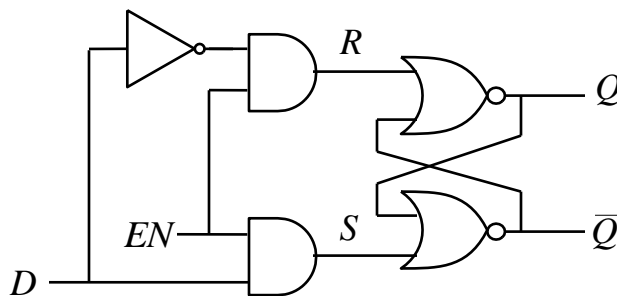
D	EN	Q'	$\overline{Q}'$	comment
X	0	Q	$\overline{Q}$	RS hold
0	1	0	1	RS reset
1	1	1	0	RS set

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

85

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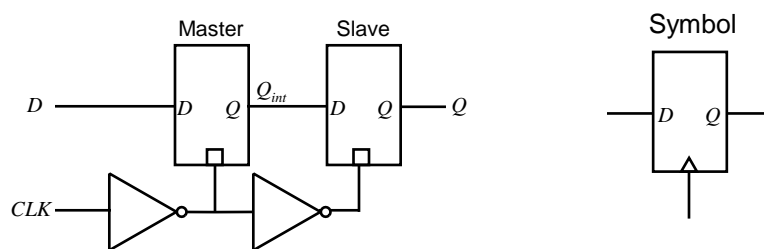
- To see how this works, we will use a timing diagram
- Note that both latch inputs are effectively connected to the clock signal (admittedly one is a complement of the other)

Master-Slave D Flip-Flop

Master   Slave

D —— D Q $\xrightarrow{Q_{int}}$ D Q —— Q

See Q changes on rising edge of CLK

CLK ——▷○——▷○

CLK

$\overline{CLK}$

D

Note propagation delays have been neglected in the timing diagram

Q_{int}

Q
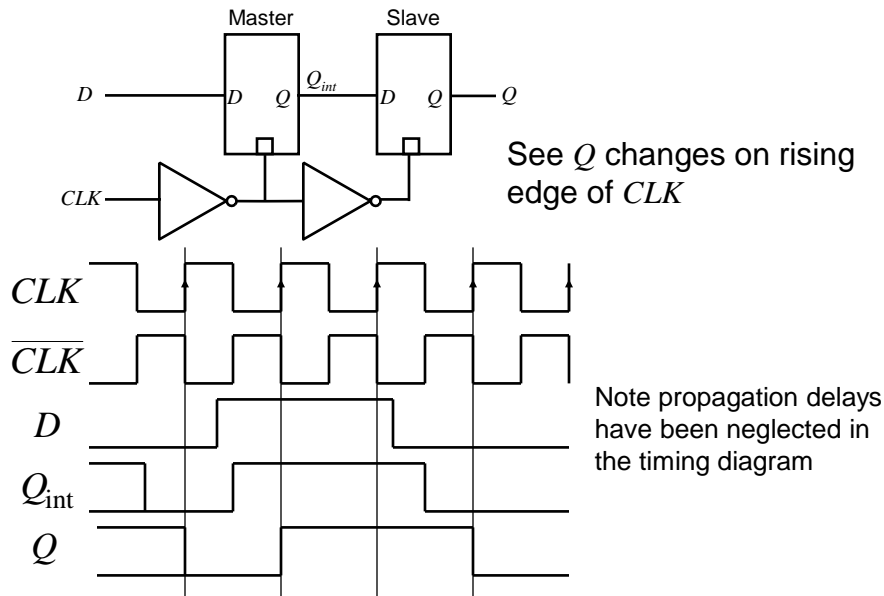
D Flip-Flops

- 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
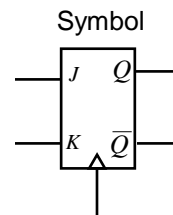
T Flip-Flop

- 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
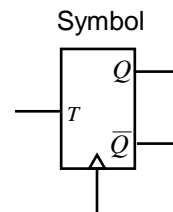
Asynchronous Inputs

- 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

Timing

CLK

D

t_{su} t_h

Q

t_p
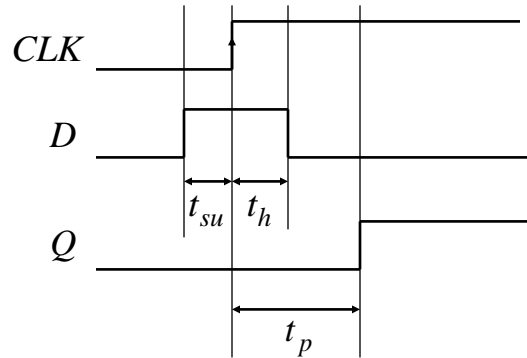
t_{su} Set-up time

t_h Hold time

t_p Propagation delay

Digital Electronics: Sequential Logic

Flip-Flop Applications and Timing Considerations

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

Memories

- For example,
  - 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., $T = 1$
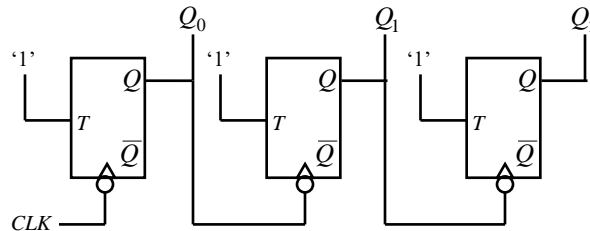
- See that the FFs are not clocked using the same clock, i.e., this is **not** a synchronous design. This gives some problems….

Ripple Counters

- We will now draw a timing diagram
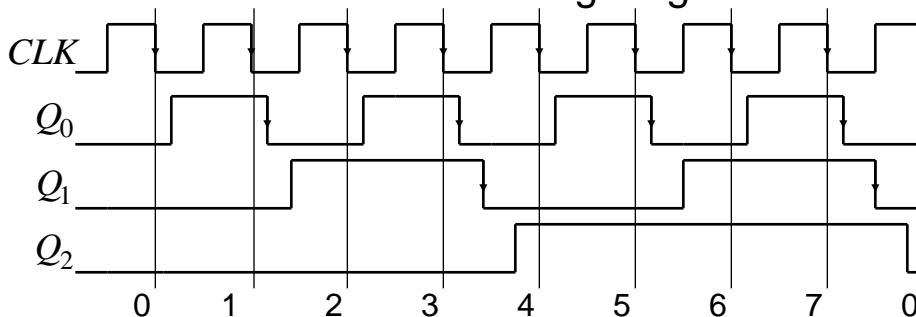
- Problems:

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
- For example, for a J-K FF the following tables are appropriate:

Characteristic and Excitation Tables

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

Truth table

Q	Q'	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table

- 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 $Q_2Q_1Q_0$			Next state $Q_2'Q_1'Q_0'$			FF inputs $D_2D_1D_0$		
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

- If using J-K FFs for example, we need J and K input columns for each FF
- 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 $Q_2 Q_1 Q_0$			Next state $Q_2' Q_1' Q_0'$			FF inputs $D_2 D_1 D_0$		
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,
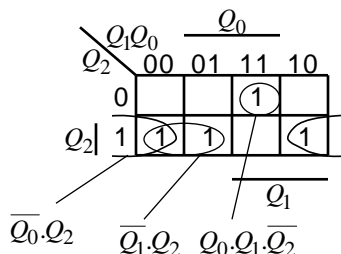
$\overline{Q_0}.Q_2$ $\overline{Q_1}.Q_2$ $Q_0.Q_1.\overline{Q_2}$

Synchronous Counter Example

So,

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

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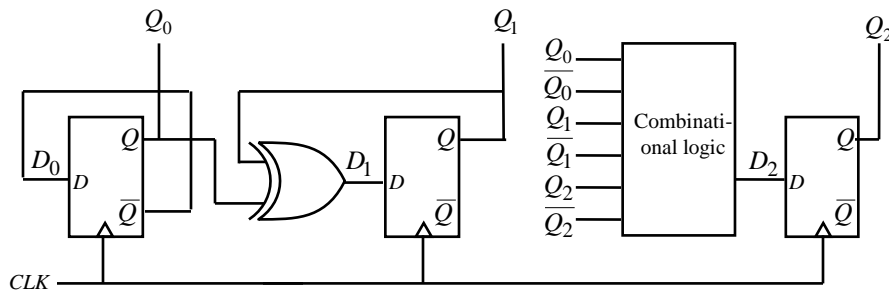
Synchronous Counter

- 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
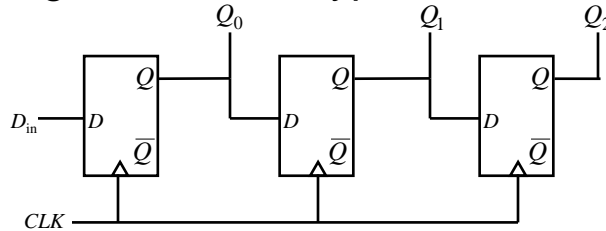
- Has a serial input, D_{in} and parallel output Q_0, Q_1 and Q_2.

Shift Register
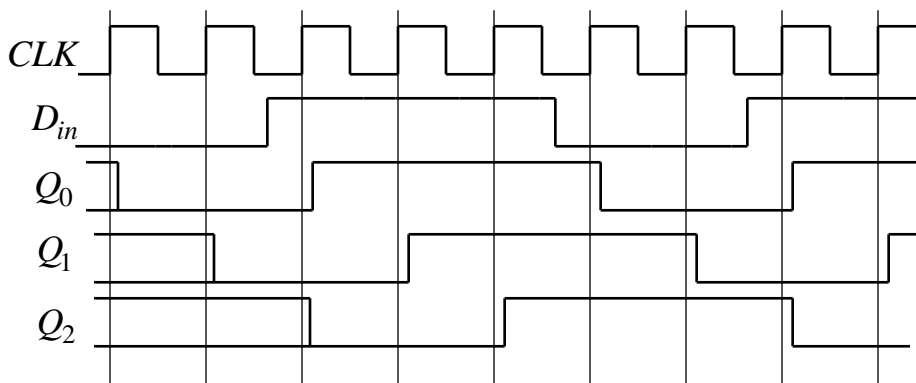
- See data moves one position to the right on application of each clock edge

Shift Register

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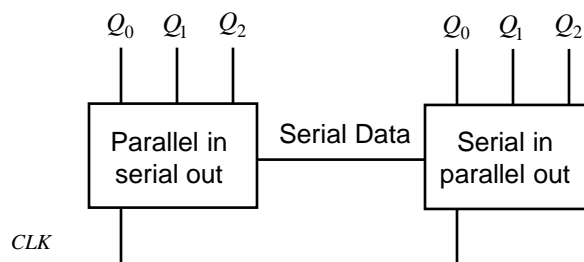
- One data bit at a time is sent across the serial data link
- See fewer wires are required than for a parallel data link

System Timing

- The clock period, T_c, is the time between the rising edges of a repetitive clock signal
- The clock frequency, f_c, is the reciprocal of the clock period, i.e., $f_c = 1/T_c$
- Note the unit of frequency is Hz, though typical modern processors can operate up to several GHz
- All things being equal, increasing the clock frequency increases the 'work' that a digital system can accomplish per unit time

Set-up Time Constraint

- Previously, we saw the timing constraints that apply for correct operation of an edge triggered D-FF
- We will now see how these constraints affect system clock speed.
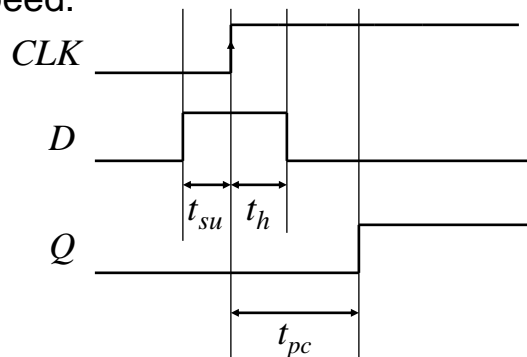
t_{su} Set-up time t_h Hold time t_{pc} CLK-to-Q Propagation delay

Set-up Time Constraint
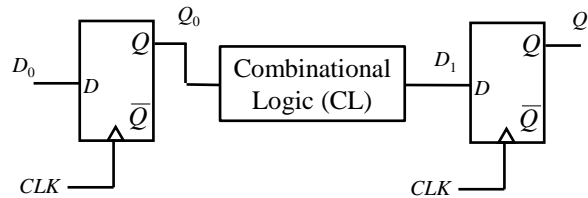
- The above diagram shows a generic path in a synchronous sequential circuit
- On the rising edge of CLK, FF0 gives output Q_0 (after delay t_{pc}).
- This signal enters a block of combinational logic (CL) producing D_1 (after a delay of t_{pd} from Q_0 changing), which is the input to FF1
- To satisfy the setup time for FF1, D_1 must settle no later than the setup time before the next CLK edge

Set-up Time Constraint
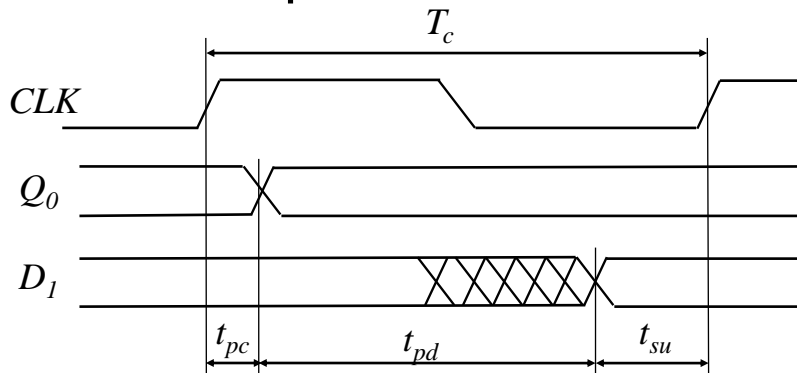
- The diagram shows the maximum propagation delay t_{pd} that will enable the worst case setup time to be satisfied (assuming worst case t_{pc}), i.e., the minimum clock period is given by,

$$T_C \geq t_{pc} + t_{pd} + t_{su}$$

Set-up Time Constraint

- Note that the clock period of a system (i.e., the clock speed) is often set by the marketing dept!
- Since the worst case (i.e., maximum) values of t_{pc} and t_{su} are specified by the chip manufacturer, we can rearrange the previous equation to solve for the maximum propagation delay through the combinational logic, which is usually the only variable under the control of the system designer,

$$t_{pd} \leq T_C - (t_{pc} + t_{su})$$

- If this cannot be achieved by redesigning the combinational logic, the clock period has to be increased to ensure correct operation

Hold Time Constraint
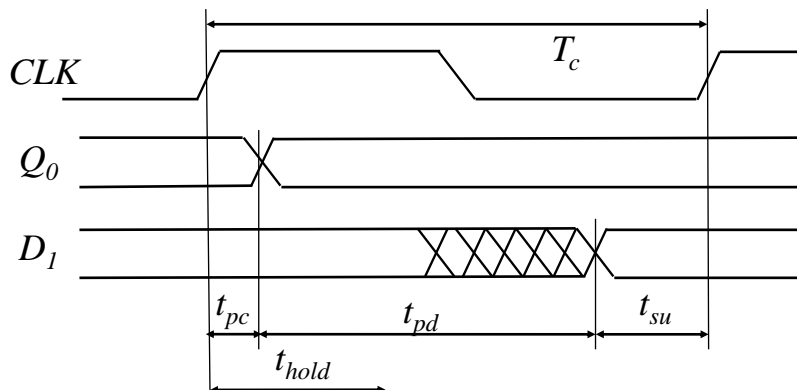
- The diagram shows that D_1 must not change in a time shorter than t_{hold} (the min FF hold time). Thus the min value of $t_{pc}+t_{pd}$ must be greater than t_{hold}, i.e.,

$$(t_{pc} + t_{pd})min \geq t_{hold}$$

Hold Time Constraint

- We would expect 2 FFs to be able to be directly cascaded i.e., with no combinational logic between them, without any timing issues. In this case, $t_{pd} = 0$, so

$$(t_{pc})min \geq t_{hold}$$

- So, a reliable FF must have a minimum hold time less than the min propagation delay time

- Often FFs are designed with $t_{hold} = 0$, hence the above condition is always satisfied

- Note that hold time violations cannot be overcome by adjusting the clock period. Consequently, they can be hard to fix and have to be taken seriously

Propagation Delay - Note

- In some books, e.g., Harris and Harris, the minimum propagation delay of a FF or of some combinational logic is called the *contamination delay*, i.e., $(t_{pc})min$ and $(t_{pd})min$ are the clock-to-Q and combinational *contamination delay* respectively

- As we have seen, the maximum propagation delay is usually our main concern, since this limits the maximum clock rate. However, we have also seen that contamination delay must be considered regarding the hold time constraint

Clock Skew

- In the previous slides, we have assumed that the system clock reaches all the FFs at the same time
- Owing to the physical layout of the clock wiring giving rise to different wire lengths and hence different propagation delays, in reality, the clock edges will not arrive at the FFs at the same time. This variation is known as *clock skew*.
- In the following case, the clock to FF1 (CLK_1) is in advance (by t_{skew} seconds) of the clock to FF0 (CLK_0)

Clock Skew – Set-up Time
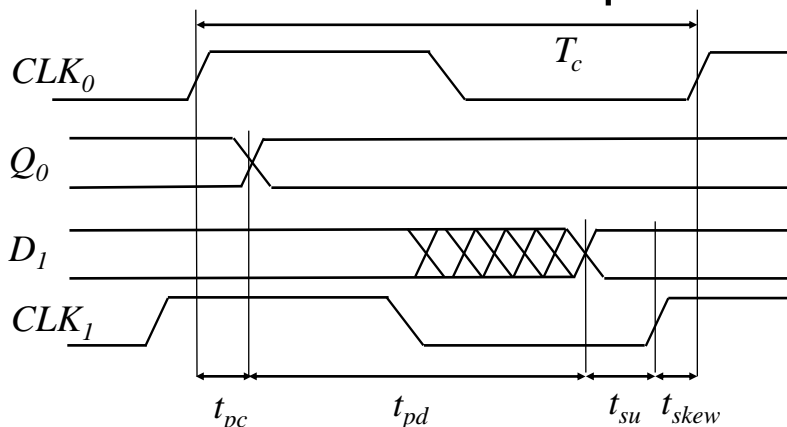
- The diagram shows the max propagation delay t_{pd} that will enable the worst case setup time to be satisfied (with worst case t_{pc}), i.e., the min clock period is given by, $T_C \geq t_{pc} + t_{pd} + t_{su} + t_{skew}$

Clock Skew – Set-up Time

- So the max propagation delay through the combinational logic is

$$t_{pd} \leq T_C - (t_{pc} + t_{su} + t_{skew})$$

- Thus in this case the clock skew has the effect of reducing the allowable propagation delay through the combinational logic
- Equivalently, for a fixed value of combinational logic propagation delay, the clock period T_C must be increased, i.e., the clock frequency decreased

Clock Skew – Hold Time

Clock Skew – Hold Time

- The diagram shows that D_1 must not change in a time shorter than t_{hold} (the min FF hold time). Thus t_{skew} plus the min value of $t_{pc}+t_{pd}$ must be greater than t_{hold}, i.e.,

$$t_{skew} + (t_{pc} + t_{pd})min \geq t_{hold}$$

- We see that in this case, the presence of clock skew makes it easier to satisfy the hold time constraint, i.e., in effect, the availability of hold time is increased.

Metastability

- It is not always possible to control when a FF input changes in relation to the clock edge
- For example, this can occur when the input signal comes from an external user input, e.g., a button
- Consider the following example when the D input change violates the dynamic requirements
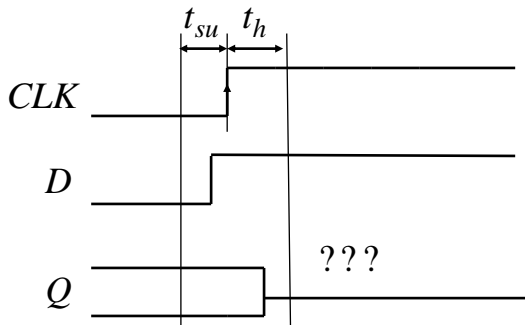
- This causes the output Q to be undefined
- Momentarily it can take on a voltage between 0 and V_{DD}, i.e., in the invalid range

Metastability

- This is known as *metastability* and one way to visualise it is to consider a ball on the summit of a hill between 2 valleys
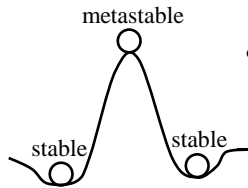
- The 2 valleys are stable states, i.e., 0 or 1, and the ball will remain there as long as it is not disturbed

- The hill top is called metastable, because the ball would remain there if it were perfectly balanced
- Since nothing is perfect, the ball will eventually roll one side or the other to reach a stable state
- How long this takes depends upon how well balanced the ball was in the first place

Metastability

- Eventually, the FF output will *resolve* to a stable valid 0 or 1 voltage level
- In theory, the resolution time is unbounded, however, we can model the probability of the resolution time, t_{res} exceeding some arbitrary time t

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}}$$

where T_c is the clock period, and T_0 and τ are characteristics of the FF.

We can view T_0 / T_c as the probability that the input changes at a 'bad' time since we see it decreases with increasing T_c, and τ is a time constant indicating how fast the FF will exit the metastable state
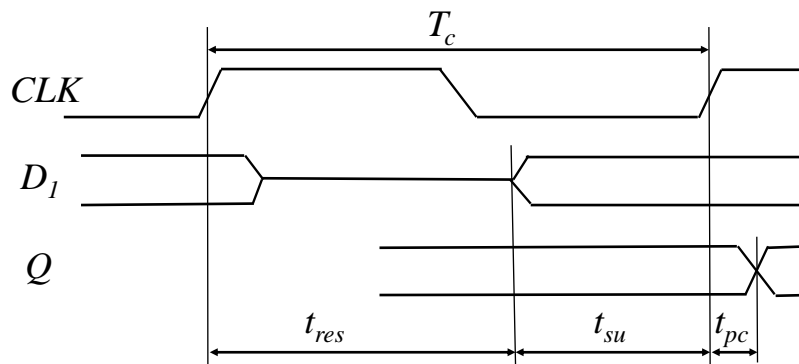
Metastability

- We will not go in to the derivation of this model, but the key point is that this probability decreases exponentially as t increases, i.e., the longer we wait, the lower is the probability of the output being in a metastable state
- Metastability gives rise to severe system problems and we must minimise the probability of it occurring
- One way to do this is to use a 'synchroniser' consisting of cascaded FFs, often one more in addition to the original input FF.

Metastability

- To minimise the probability of *metastablity* we use a *synchroniser.* In its simplest form it uses 1 more FF.
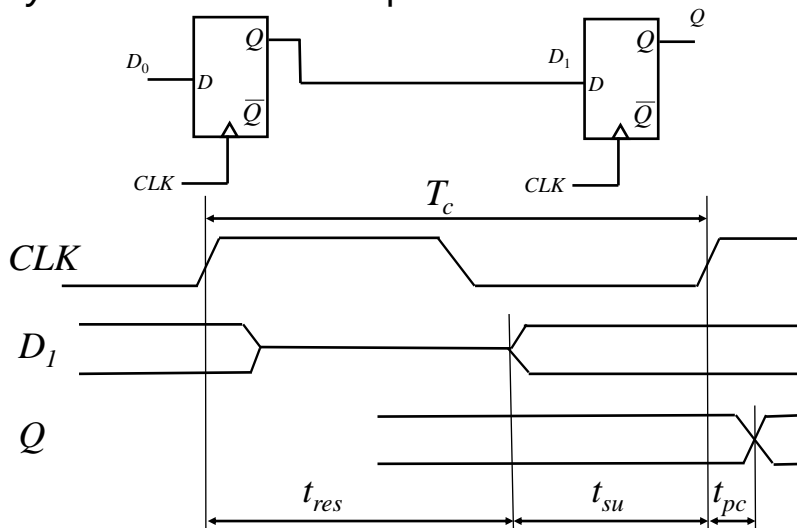
Metastability

- The output from FF0, D_1, will resolve to a valid level with high probability if T_c is long enough
- FF1 now has valid input that satisfies both its setup and hold times and yields a valid output Q

Metastability

- To reduce the probability of an invalid output from the synchroniser, we need to wait a longer time for the metastable condition at D_1 to resolve, i.e., we need to increase time t_{res}
- So to satisfy the setup time t_{su} for FF1, we need to increase the clock period T_c, i.e., slow the clock rate
- We say that the synchroniser *fails* if output Q becomes metastable
- This may happen if D_1 has not resolved to a valid level before it needs to satisfy the setup condition on FF1, that is, if $t_{res} > T_c - t_{su}$

Metastability

- So the probability of failure for a single input change is

$$P_{fail} = \frac{T_0}{T_c} e^{-\frac{T_c - t_{su}}{\tau}}$$

- If input D_0 changes once per second, the probability of failure per second is just P_{fail}
- However, if D_0 changes N times per second, the probability of failure per second is N times greater

$$P_{fail}/s = NP_{fail}$$

Metastability

- System reliability is usually measured as the *mean time between failures (MTBF)*
- This is just the reciprocal of the probability that a system will fail in any given second

$$MTBF = \frac{1}{P_{fail}/s} = \frac{T_c e^{\frac{T_c - t_{su}}{\tau}}}{NT_0}$$

Metastability - Example

- The FFs in the example synchroniser have the following characteristics: τ = 200 ps, T_0 = 150 ps and t_{su} = 500 ps. The input data changes 0.2 times per second on average
- How long must be the synchroniser clock period be for the MTBF to exceed 1 year?
- Solution –
  – So, 1 year $\approx 31.5 \times 10^6$ s

  and using the previous equation,

$$31.5 \times 10^6 = \frac{T_c e^{\frac{T_c - 500 \times 10^{-12}}{200 \times 10^{-12}}}}{(0.2)150 \times 10^{-12}}$$

  This equation has no closed form solution, but by trial and error we can get T_C = 3.04 ns

Metastability

- If T_c becomes excessive to achieve a specified MTBF, it is possible to cascade additional FFs. So for a synchroniser with a total of K additional FFs,

$$P_{fail\_K} = (P_{fail})^K$$

Hence, $P_{fail\_K}/s = N\,P_{fail\_K} = N(P_{fail})^K$

Yielding,

$$MTBF = \frac{1}{N(P_{fail})^K}$$

  – For this to work well for a reasonable number FFs, the probability of metastability at the output of each FF has to be much lower than 1

Digital Electronics:
Sequential Logic

Synchronous State Machines 1

Introduction

- 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
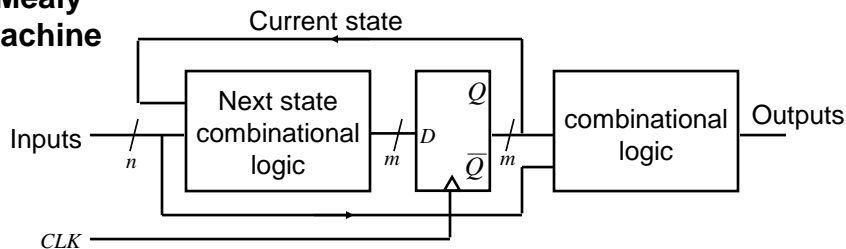- **Outputs –** Results from the FSM

Types of State Machines

- Two types of state machines are in general use, namely *Moore* machines and *Mealy* machines
- We will see that the state diagrams (and associated state tables) corresponding with the 2 types of machine are slightly different

Machine Schematics

Moore Machine
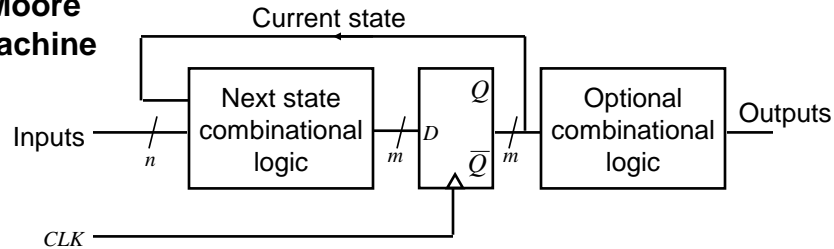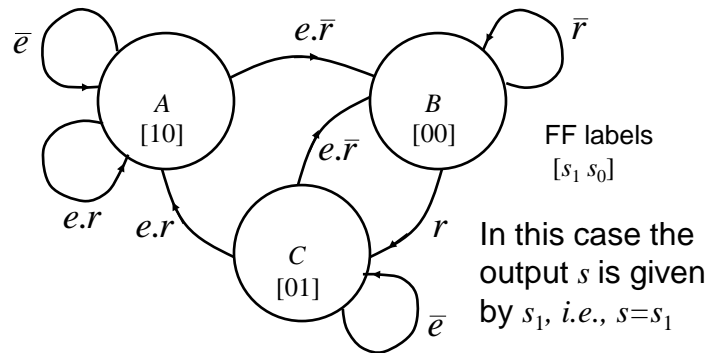

Mealy Machine

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 State Diagram

- Example FSM has 3 states (A, B and C), inputs e and r, and output s

FF labels
$[s_1 \, s_0]$

In this case the output s is given by s_1, *i.e.*, $s = s_1$

- See **inputs only** appear on transitions between states, i.e., next state is given by current state and current inputs
- Outputs determined from current state via combinational logic (if required)

Mealy Machine State Diagram

- Example FSM has 3 states (A, B and C), inputs x and y, and output s

FF labels:
$[s_1 \, s_0]$

Transition labels:
Inputs/Output
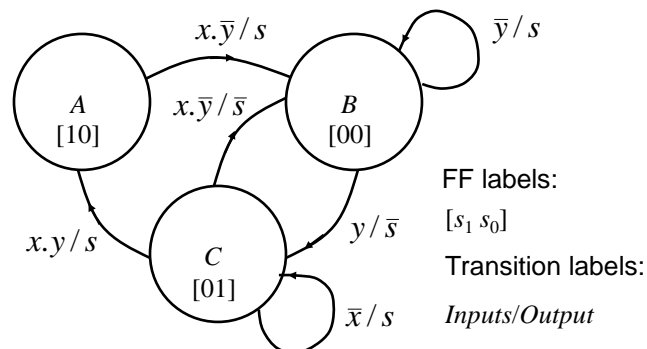
- Inputs **and outputs** appear on transitions between states, i.e., next state is given by current state and current inputs
- Output determined from current state and inputs via combinational logic

119

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
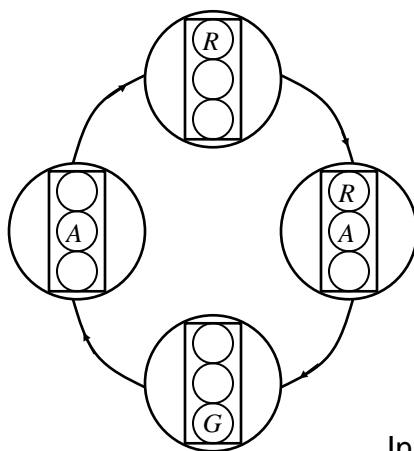
See we have 4 states

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

State
100

State
010
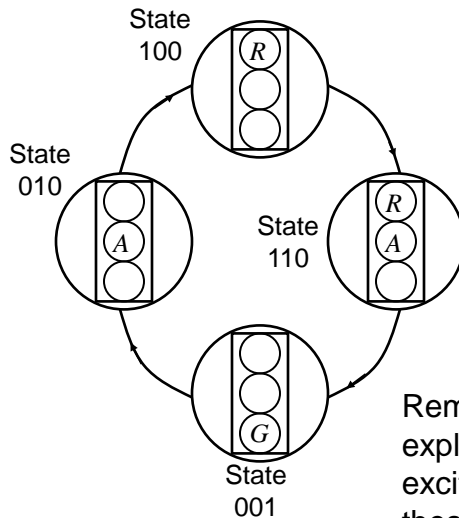
State
110

State
001

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
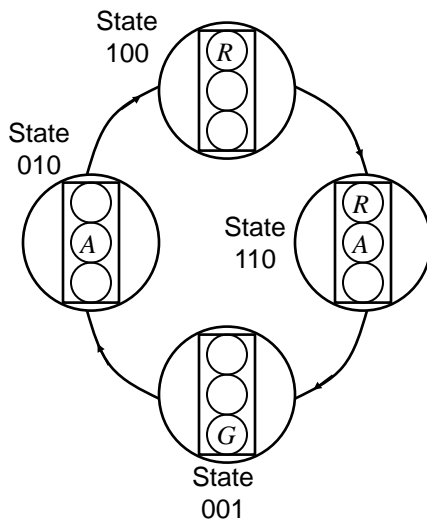
State
100

State
010

State
110

State
001

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

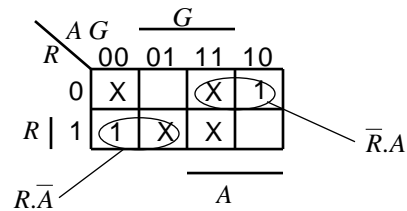# Example – Traffic Light Controller

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

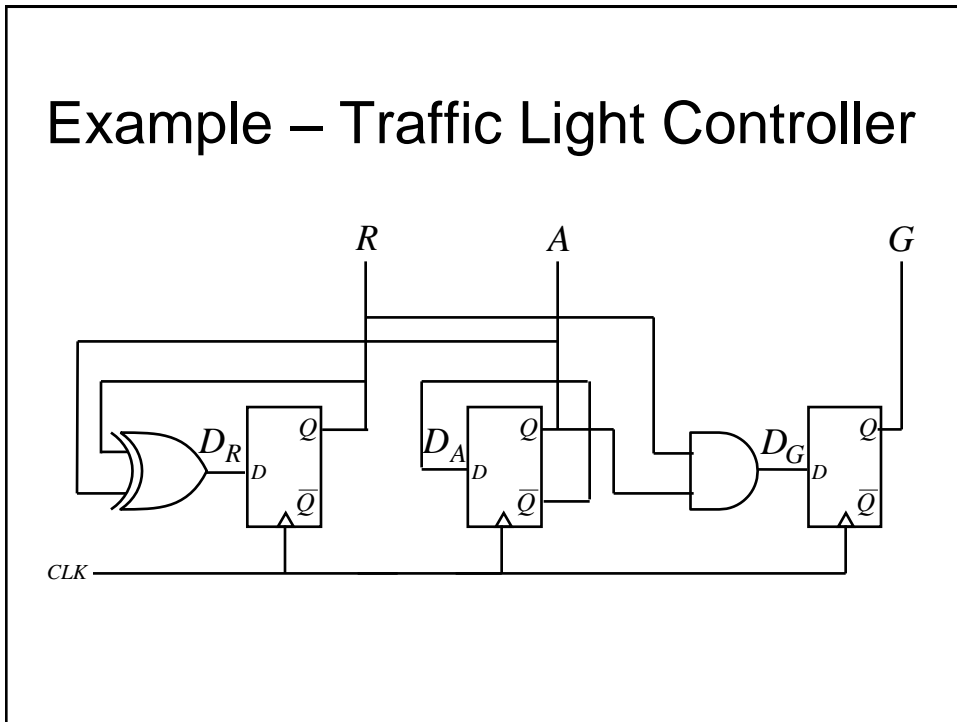Unused states, 000, 011, 101 and 111.

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
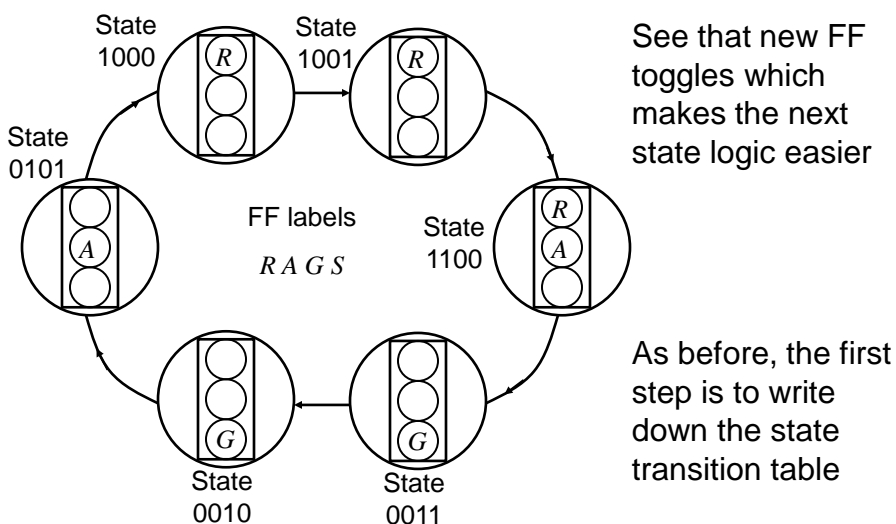
Example – Traffic Light Controller

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

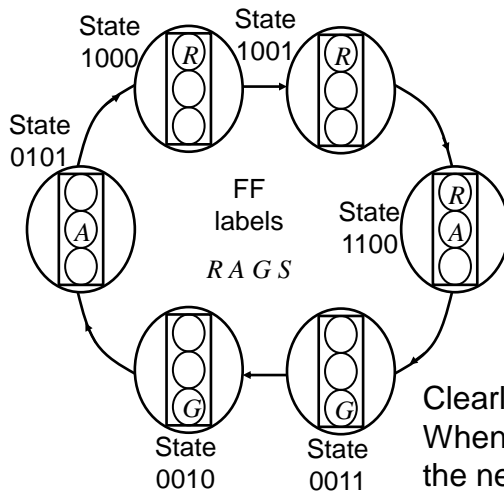| Start state	Next state logic output		
000	010		
011	100	Which are all	So it does
101	110	valid states	self start
111	001		

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

State 1000
State 1001
State 0101
FF labels
$R\ A\ G\ S$
State 1100
State 0010
State 0011

Example 2

State
1000

State
1001

State
0101

FF
labels

$R\ A\ G\ S$

State
1100

State
0010

State
0011

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

$R\,A$ \ $G\,S$	00	01	11	10
00	X	X	0	0
01	X	1	X	X
11	0	X	X	X
10	1	1	X	X

G

A

$\overline{R}.A$

$R.\overline{A}$

S

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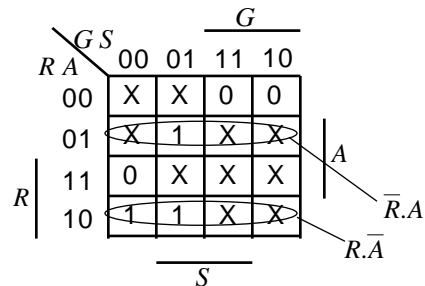
$$D_R = R.\overline{A} + \overline{R}.A = R \oplus A$$

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:

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

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

By inspection we can also see:

$$D_S = \bar{S}$$

Digital Electronics: Sequential Logic

Synchronous State Machines 2

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 0 for 3 clock edges
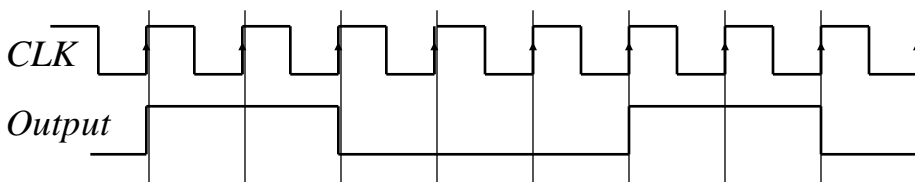
Sequential State Assignment

- 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

For FF b:

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.

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

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	By inspection we can see that:

$$c \quad b \quad a \mid c' \ b' \ a'$$

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

By inspection we can see that:

For FF b:

$$D_b = a$$

For FF c:

$$D_c = b$$

Unused states, 010, 101, and 111.

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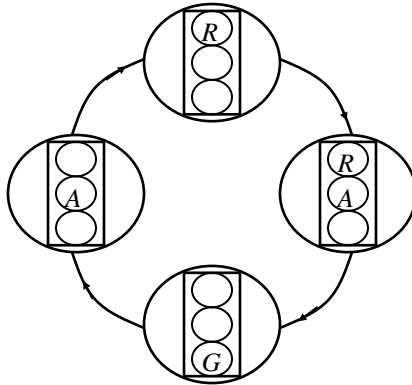
For 1 hot, we need 1 FF for each state, i.e., 4 in this case

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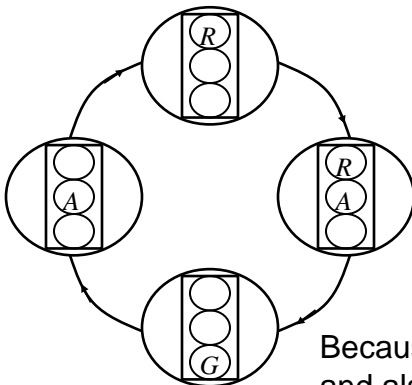
Current state				Next state			
r	ra	g	a	r'	ra'	g'	a'
1	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0

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:

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

One Hot - Example

$$D_a = g \quad D_g = ra \quad D_{ra} = r \quad D_r = a$$
$$R = r + ra \quad A = ra + a \quad G = g$$
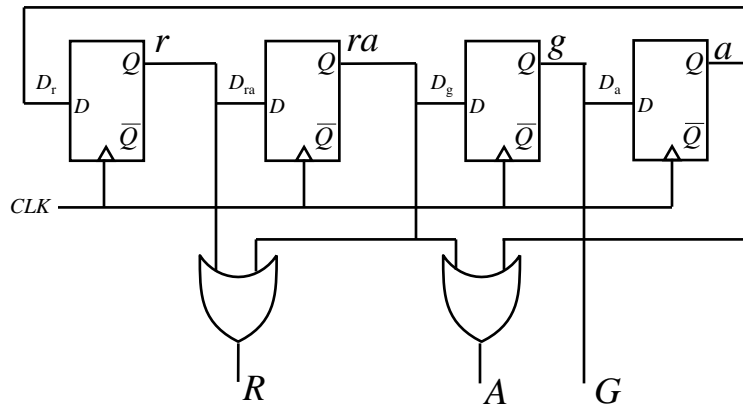
Tripos Example

- 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

135
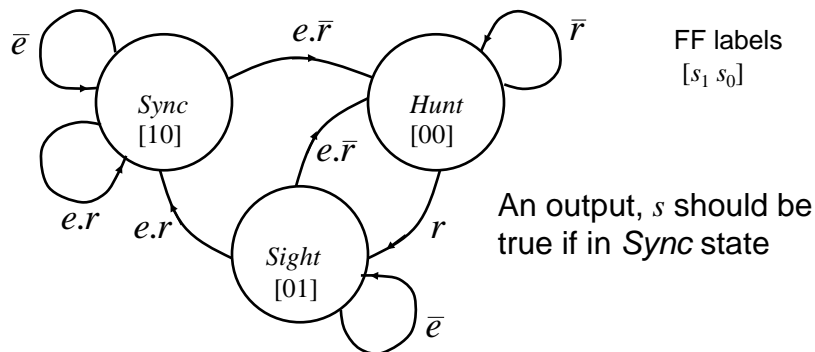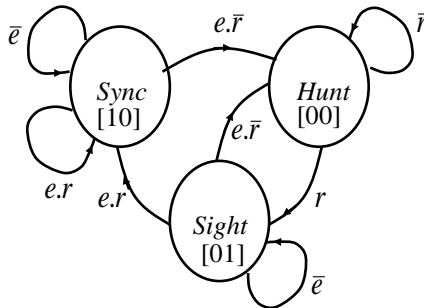
Tripos Example

Unused state 11

From inspection, $s = s_1$

Current state s_1 s_0		Input e r		Next state 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 s_1 s_0		Input e r		Next state 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

Plot k-maps to determine the next state logic

For FF 0:

Current state s_1 s_0	Input e r	Next state 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
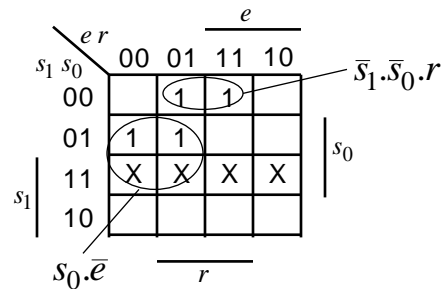
$$D_0 = s_0.\bar{e} + \bar{s}_1.\bar{s}_0.r$$

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:

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

Simplification is possible since:

$$s_2 + s_1 + s_0 = 1$$

so, $\bar{s_0} = s_1 + s_2$, hence,

$$D_2 = \bar{s_0}.e.r + s_2.\bar{e}$$

For FF 1: $D_1 = s_0.r + s_1.\bar{e}$

For FF 0:

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

Tripos Example

Sync [100], *Hunt* [001], *Sight* [010] with transitions labelled $\bar{e}$, $e.\bar{r}$, $\bar{r}$, $e.\bar{r}$, $e.r$, $e.r$, r, $\bar{e}$
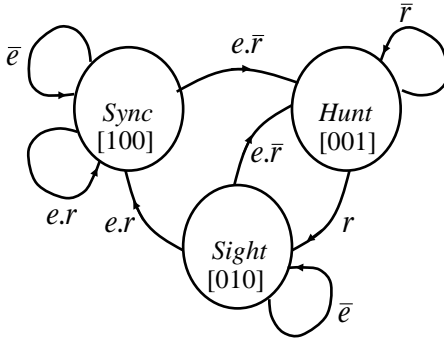
Note that it is not strictly necessary to write down the state table, since the next state equations can be obtained from the state diagram

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

Tripos Example

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

Digital Electronics: Sequential Logic

Further Considerations

Elimination of Redundant States

- Sometimes, when designing state machines it is possible that unnecessary states may be introduced

- In general, reducing the number of states may reduce the number of FFs required and may also reduce the complexity of the next state logic owing to the presence of more unused states (don't cares)

Elimination of Redundant States - Example

- Consider the following State Table that corresponds with a Mealy Machine implementation

- This is so, since the inputs and outputs from the machine are on the transitions (arcs) between states

- The following state table is drawn in a compact form by incorporating the 2 possible input values as parallel columns within both the next state and output columns of the table

Example

Current State	Next State X=0	X=1	Output (Y) X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	I	0	0
E	J	K	0	0
F	L	M	0	0
G	N	P	0	0
H	A	A	0	0
I	A	A	0	0
J	A	A	0	1
K	A	A	0	0
L	A	A	0	1
M	A	A	0	0
N	A	A	0	0
P	A	A	0	0

- From the table, we see that there is no way of telling states H and I apart, so we can replace I with H when it appears in the Next State portion of the table

Example

Current State	Next State X=0	X=1	Output (Y) X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	K	0	0
F	L	M	0	0
G	N	P	0	0
H	A	A	0	0
J	A	A	0	1
K	A	A	0	0
L	A	A	0	1
M	A	A	0	0
N	A	A	0	0
P	A	A	0	0

- We also see that there is now no way to get to state I so we can remove row I from the table
- Similarly, rows K, M, N and P have the same next state and output as H and can be replaced by H

Example

Current State	Next State X=0	X=1	Output (Y) X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	H	0	0
F	L	H	0	0
G	H	H	0	0
H	A	A	0	0
J	A	A	0	1
L	A	A	0	1

- Similarly, there is now no way to get to states K, M, N and P and so we can remove these rows from the table
- Also, the next state and outputs are identical for rows J and L, thus L can be replaced by J and row L eliminated from the table

Example

Current State	Next State		Output (Y)	
	X=0	X=1	X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	H	0	0
F	J	H	0	0
G	H	H	0	0
H	A	A	0	0
J	A	A	0	1

- Now rows D and G are identical, as are rows E and F.
- Consequently, G can be replaced by D, and row G eliminated. Also, F can be replaced by E and row F eliminated from the table

Example

Current State	Next State		Output (Y)	
	X=0	X=1	X=0	X=1
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	H	H	0	0
E	J	H	0	0
H	A	A	0	0
J	A	A	0	1

- The procedure employed to find equivalent states in this example is known as *row matching*.
- However, we note row matching is not sufficient to find all the equivalent states except for certain special cases

Elimination of Redundant States – State Equivalence

- The previous row matching approach only works in certain cases.

- We will now consider a more general approach that identifies state equivalence to help eliminate states

- Two states, say p and q, can be considered equivalent, i.e., $p \equiv q$, if from each of these states, a finite state machine generates the same output sequence in response to any input bit sequence.

State Equivalence

- In practice, 2 states p and q can be considered to be equivalent if for any input bit sequence, the corresponding outputs y_p and y_q, are identical, i.e., $y_p = y_q$, and the next states $S_p{}'$ and $S_q{}'$, are equivalent, i.e., $S_p{}' \equiv S_q{}'$

Present State	Next State X=0	Next State X=1	Output (Y)
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1
E	A	D	0

- Regardless if starting from state C or E, the machine goes through identical next states and yields the same output.

- States C and E are equivalent and so state E can be eliminated from the table

State Equivalence

- Thus the reduced state table is,

Present State	Next State X=0	X=1	Output (Y)
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

- Actually, in this example we have the special case where the next states are identical and not just equivalent, i.e., this is row matching that we saw previously

- What we need is a method to identify equivalent and not just identical next states

State Equivalence

- We can rewrite the previous theorem as,

$$\lambda(p, x) = \lambda(q, x) \text{ and } \delta(p, x) \equiv \delta(q, x)$$

  where, $\lambda(p, x)$ is the output given the present state p and input x and,

  $\delta(p, x)$ is the next state given the present state p and input x

- We will now use this theorem to find all the equivalent states in a state table

Determination of State Equivalence using an Implication Table

- The procedure will be described via an example
- We need to perform a pairwise comparison between all possible pairs of states to see if we can discover equivalent state pairs.
- The Implication Table facilitates this procedure
  - It has a cell for every possible pair of states
  - Note cells above the diagonal are omitted (since they already exist below the diagonal)
  - Diagonal cells are also omitted since they correspond to same state pairs

Example – Implication Table

Present State	Next State X=0	Next State X=1	Output (Y)
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

Implication Table

- So we perform a pairwise comparison between all possible pairs of states to see if we can discover equivalent state pairs.

- So in each pairwise comparison, i.e., a cell in the table, we will indicate any 'implied' equivalent state pairs (if any).

- A cell will not contain any 'implied' state pairs if the outputs are different, since this does not satisfy our earlier equivalence theorem

- The ultimate objective is to identify which of the implied state pairs are actually equivalent state pairs

Implication Table

- To fill in 1st column
  - Compare row A with each of the other rows
  - We see that the output for row A is different to the output for row C, so we place an X in this cell to indicate that A ≠ C
  - Similarly we place an X in cells A-E, A-F and A-H to indicate that A ≠ E, A ≠ F and A ≠ H because of the output differences
  - State A and B have the same outputs, hence from the theorem, A ≡ B if D ≡ F and C ≡ H.
  - To indicate this we write the 'implied pairs' D-F and C-H in the A-B cell

Implication Table

– Similarly, since State A and D have the same outputs, we write the 'implied pairs' A-D and C-E in the A-D cell to indicate that, A ≡ D if A ≡ D and C ≡ E

– The entries B-D and C-H in the A-G cell indicate that A ≡ G if B ≡ D and C ≡ H

– Next row B of the state table is compared pairwise with the remaining rows in the table and so column B is filled-in

– Similarly the remaining columns in the implication table are filled-in

– Note that 'self implied' pairs are removed from the table, e.g., in the A-D cell we have A ≡ D if A ≡ D

Example

Present State	Next State X=0	X=1	Output (Y)
A	D	C	0
B	F	H	0
C	E	D	1
D	A	E	0
E	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

Implication table:

	A	B	C	D	E	F	G
B	D-F / C-H						
C	✕	✕					
D	A-D / C-E	A-F / E-H	✕				
E	✕	✕	C-E / A-D	✕			
F	✕	✕	E-F / B-D	✕	C-F / A-B		
G	B-D / C-H	B-F	✕	A-B / E-H	✕	✕	
H	✕	✕	C-E / D-G	✕	A-G	C-F / B-G	✕

Implication Table

– At this stage the cells in the implication table are filled-in either with implied pairs or an X

– We now check each implied pair

– If one of the pairs in say cell i-j is not equivalent, then $i \neq j$

– So, looking at cell A-B, we see it has 2 implied pairs D-F and C-H. Since $D \neq F$ (see the D-F cell has an X in it), $A \neq B$ and we place an X in the A-B cell as shown in the following updated table

– Continuing with the 1st column we see cell A-D contains implied pair C-E. Since cell C-E does not contain an X, we cannot determine at this stage whether $A \equiv D$ or not

– Similarly with cell A-G

Example

– We can place an X in cells B-D and B-G since $A \neq F$ and $B \neq F$

– Similarly we can check the remaining columns and place an X in cells C-F, D-G, E-F and F-H

– In going from the original to the updated table, note that we found several additional non-equivalent state pairs

– So we must go through again to see if the added Xs make any other pairs non-equivalent

Example

	D-F C-H						
B							
C	X	X					
D	A-D C-E	A-F E-H	X				
E	X	X	C-E A-D	X			
F	X	X	E-F B-D	X	C-F A-B		
G	B-D C-H	B-F	X	A-B E-H	X	X	
H	X	X	C-E D-G	X	A-G	C-F B-G	X
	A	B	C	D	E	F	G

– Rechecking col. A we can place an X in cell A-G since cell B-D has an X

– Similarly with cells C-H and E-H

– Since we added some more Xs we must go through again to see if the added Xs make any other pairs non-equivalent

– No more Xs are added, so all cells with non equivalent states have now been Xed out

Example

- The 'coordinates' of the remaining cells correspond to the equivalent state pairs, i.e., cell A-D and cell C-E so,
  – A ≡ D and C ≡ E
- So in the state table we can replace D with A and E with C and then eliminate rows D and E

Present State	Next State X=0	X=1	Output (Y)
A	A	C	0
B	F	H	0
C	C	A	1
F	F	B	1
G	B	H	0
H	C	G	1

150

Implementation of FSMs

- We saw previously that programmable logic can be used to implement combinational logic circuits, i.e., using PLA 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 Logic Array (GLA)

GLA Devices

- They are similar in concept to PLAs, 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

GLA
Device
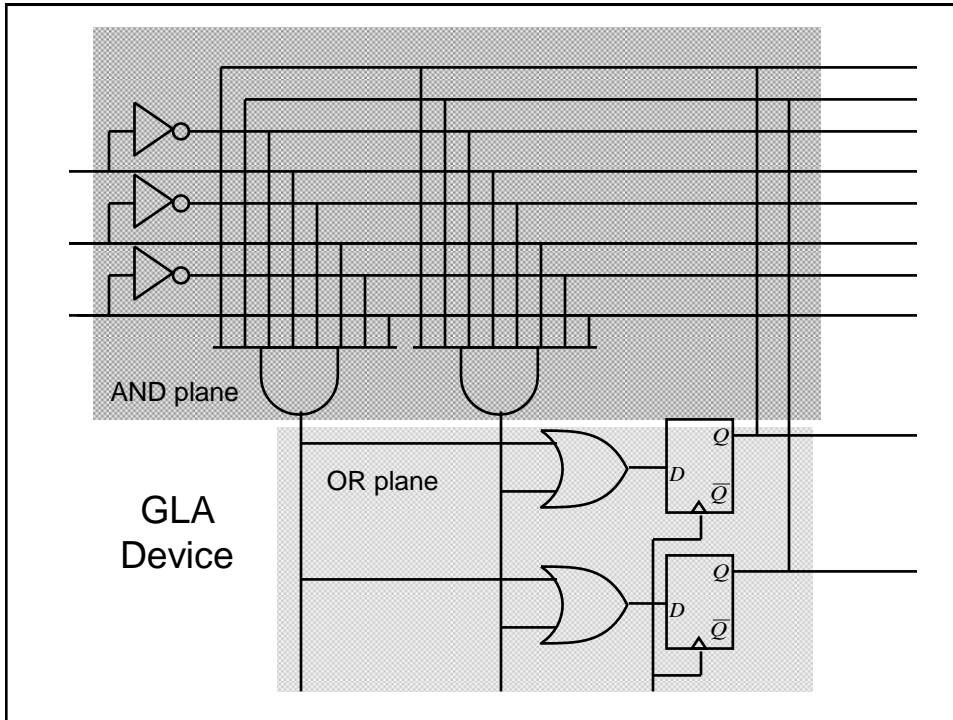
GLA Devices

• A modified form of a GLA known as a Generic Array Logic (GAL) is used in the Hardware Laboratory classes to implement various FSMs.

GAL Devices

a

b

AND plane

OR plane

D

Q

$\bar{Q}$

f_0

D

Q

$\bar{Q}$

f_n

CLK
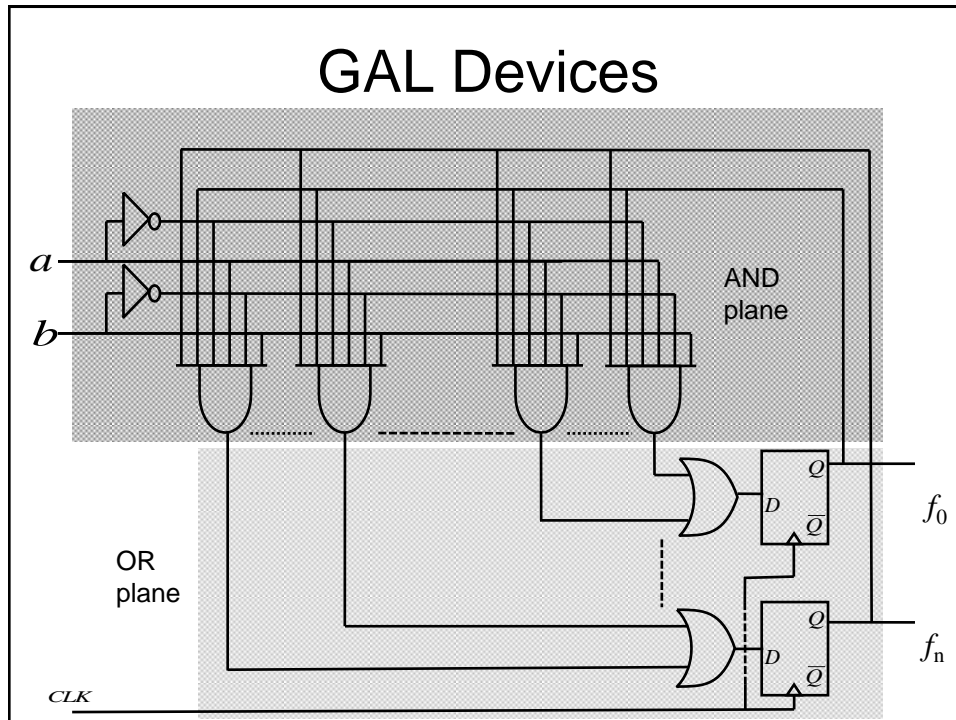
FPGA

- Field Programmable Gate Arrays (FPGAs) are the latest type of programmable logic
- Are an array of configurable logic blocks (CLBs) surrounded by Input Output Blocks (IOBs):
  - programmable routing channels permit CLBs to be connected to other CLBs and to IOBs
  - CLBs contain look up tables (LUTs), multiplexers (MUXs) and D-type FFs
  - The FPGA is configured by specifying the contents of the LUTs and select signals for the MUXs

FPGA – Xilinx Spartan
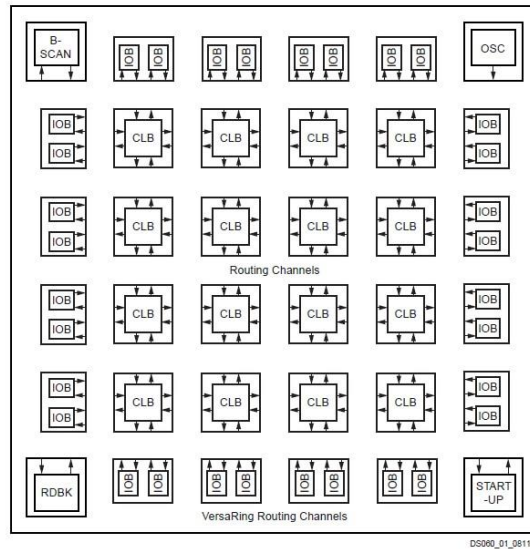
Figure 1: **Basic FPGA Block Diagram**

FPGA – Xilinx Spartan

- Simplified schematic showing CLBs and programmable routing channels, i.e., wires plus programmable switch matrices (SMs)
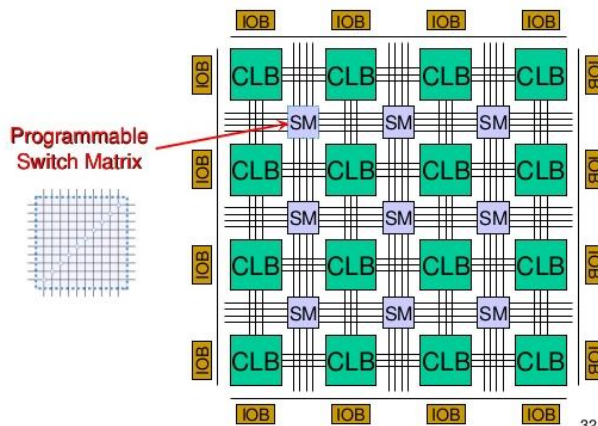
FPGA - Spartan CLB
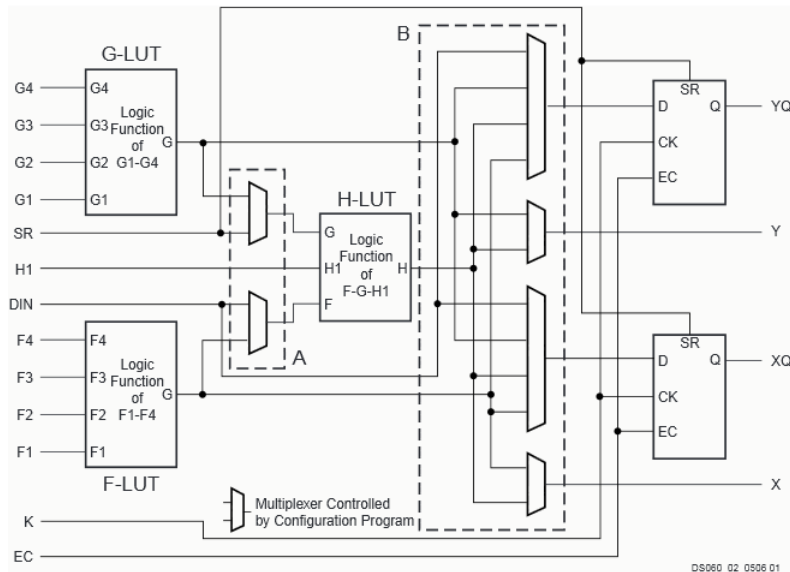
FPGA - Spartan CLB

- Has 2, 4-input LUTs (F and G) and 1, 3 input LUT (H)
- Has to 'combinational' outputs (Y and X) and 2 'registered' outputs (i.e., from D-FFs) YQ and XQ
- Depending on MUX configuration Y is given by output of either G or H LUTs and X from either F or H LUTs.
- D-FF inputs come from DIN, or from F, G, or H LUTs

FPGA - Spartan CLB

- Thus each CLB can perform up to 2 combinational and/or 2 registered functions
- All functions can involve at least 4 input variables (e.g., G1 to G4, and F1 to F4), but can be up to 9 (owing to the possibility of implementing 2-level combinational logic functions), i.e., G1 to G4, F1 to F4, H1.
- Created using either a schematic (block) diagram or more likely a Hardware Description Language (HDL) of the design

FPGA - Spartan CLB

- The synthesis tool determines how the LUTs, MUXs and routing channels are configured
- This configuration information is then downloaded to the FPGA
- Xilinx devices store their configuration information in static RAM (SRAM) so can be easily reprogrammed
- The SRAM contents can be downloaded either from a computer or from an EEPROM device when the system is powered-up

FPGA

- Other FPGA manufacturers are available, e.g., Altera.

- Particular manufacturers have many different product lines

- Main differences will be the no. of CLBs, the structure of the CLBs, internal or external ROM, additional features such as specialised arithmetic blocks, user RAM etc.

Digital Electronics:
Electronics, Devices and
Circuits

Dr. I. J. Wassell

Digital Electronics:
Electronics, Devices and
Circuits

Underlying Concepts

Introduction

- In the coming lectures, ultimately we will consider how logic gates can be built using electronic circuits
- In the first part, basic concepts concerning electrical concepts, electrical circuits, materials and circuit theory (both linear and non-linear) will be presented
- In the second part, we will consider transistor operation and characteristics followed by gate circuit design and characteristics

Basic Electricity

- An electric current is produced when charged particles (e.g., electrons in metals, or electrons and ions in a gas or liquid) move in a definite direction
- In metals, the outer electrons are held loosely by their atoms and are free to move around the fixed positive metal ions
- This free electron motion is random, and so there is no net flow of charge in any direction, i.e., no current flow

Basic Electricity

- If a metal wire is connected across the terminals of a battery, the battery acts as an 'electron pump' and forces the free electrons to drift toward the +ve terminal and in effect flow through the battery
- The drift speed of the free electrons is low, e.g., < 1 mm per second owing to frequent collisions with the metal ions.
- However, they all start drifting together as soon as the battery is applied

Basic Electricity

- The flow of electrons in one direction is known as an electric current and reveals itself by making the metal warmer and by deflecting a nearby magnetic compass

Direction of current flow

Flow of electrons in metal wire connected across a battery
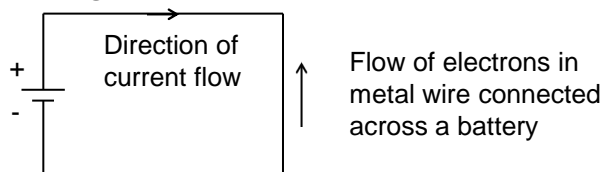
- Before electrons were discovered it was imagined that the flow of current was due to positively charged particles flowing out of +ve toward –ve battery terminal

Basic Electricity

- Note that 'conventional' current flow is still defined as flowing from the +ve toward the –ve battery terminal (i.e., the opposite way to the flow of the electrons in the metal)!
- A huge number of charged particles (electrons in the case of metals) drift past each point in a circuit per second.
- The unit of charge is the *Coulomb* (C) and one electron has a charge of $1.6*10^{-19}$ C

Basic Electricity

- Thus one C of charge is equivalent to $6.25*10^{18}$ electrons
- When one C of charge passes a point in a circuit per second, this is defined as a current (*I*) of 1 *Ampere* (A), i.e., $I = Q/t$, where *Q* is the charge (C) and *t* is time in seconds (s), i.e., current is the rate of flow of charge.

Basic Electricity

- In the circuit shown below, it is the battery that supplies the electrical force and energy that drives the electrons around the circuit.

Flow of current

V_B + - $\otimes$ V_L Lamp
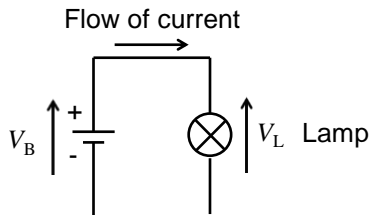
- The electromotive force (emf) V_B of a battery is defined to be 1 *Volt* (V) if it gives 1 *Joule* (J) of electrical energy to each C of charge passing through it.

Basic Electricity

- The lamp in the previous circuit changes most of the electrical energy carried by the free electrons into heat and light

- The potential difference (pd) V_L across the lamp is defined to be 1 *Volt* (V) if it changes 1 *Joule* (J) of electrical energy into other forms of energy (e.g., heat and light) when 1 C of charge passes through it, i.e., $V_L=E/Q$, where E is the energy dissipated (J) and Q is the charge (C)

Basic Electricity

- Note that pd and emf are usually called *voltages* since both are measured in V
- What is the power dissipated (P_L) in the lamp in the previous circuit?
- $P_L = E/t$ (J/s). Previously we have, $E = QV_L$, and so, $P_L = QV_L/t$ (W) .
- Now substitute $Q = It$ from before to give, $P_L = It\,V_L/t = IV_L$ (W) , an expression that hopefully is familiar

Basic Electricity

- So far, we have only considered metallic conductors where the charge is carried by 'free' electrons in the metal lattice.
- We will now consider the electrical properties of some other materials, specifically, *insulators* and *semiconductors*

Basic Materials

- The electrical properties of materials are central to understanding the operation of electronic devices
- Their functionality depends upon our ability to control properties such as their current-voltage characteristics
- Whether a material is a conductor or insulator depends upon how strongly bound the outer valence electrons are to their atomic cores

Insulators

- Consider a crystalline insulator, e.g., diamond
- Electrons are strongly bound and unable to move
- When a voltage difference is applied, the crystal will distort a bit, but no charge (i.e., electrons) will flow until breakdown occurs
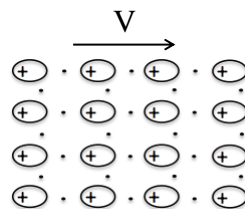
Conductors

- Consider a metal conductor, e.g., copper
- Electrons are weakly bound and free to move
- When a voltage difference is applied, the crystal will distort a bit, but charge (i.e., electrons) will flow
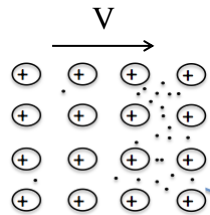
Semiconductors

- Since there are many free electrons in a metal, it is difficult to control its electrical properties
- Consequently, what we need is a material with a low free electron density, i.e., a semiconductor, e.g., Silicon
- By carefully controlling the free electron density we can create a whole range of electronic devices

Semiconductors

- Silicon (Si, Group IV) is a poor conductor of electricity, i.e., a *semiconductor*

Si is *tetravalent*, i.e., it has 4 electrons in its *valance* band

Si crystals held together by *'covalent'* bonding

8 valence electrons yield a stable state – each Si atom now appears to have 8 electrons, though in fact each atom only has a half share in them. Note this is a much more stable state than is the exclusive possession of 4 valence electrons
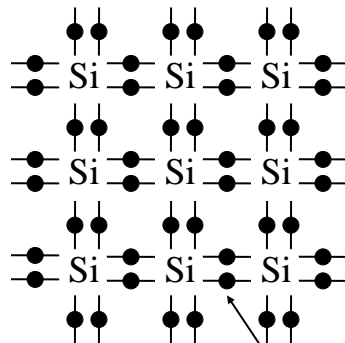
Si crystalline lattice – poor conductor at low temperatures

Shared valence electrons

Semiconductors

- As temperature rises conductivity rises
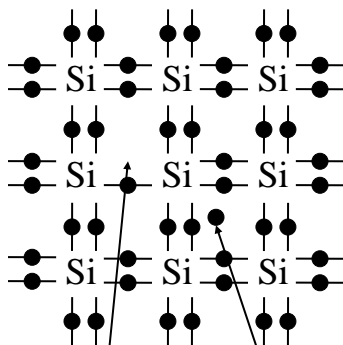
As temperature rises, thermal vibration of the atoms causes bonds to break: electrons are free to wander around the crystal.

When an electron breaks free (i.e., moves into the *'conduction band'* it leaves behind a *'hole'* or absence of negative charge in the lattice

The hole can appear to move if it is filled by an electron from an adjacent atom

The availability of free electrons makes Si a conductor (a poor one at room temperature)

Hole

Free electron

n-type Si

- n-type silicon (Group IV) is doped with arsenic (Group V) that has an additional electron that is not involved in the bonds to the neighbouring Si atoms
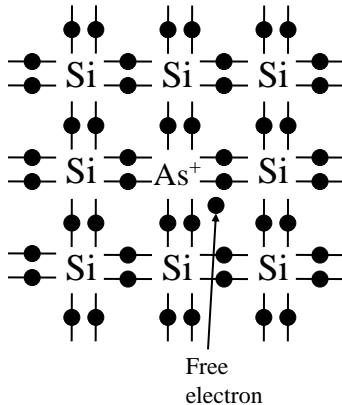
Free
electron

The additional electron needs only a little energy to move into the conduction band.

This electron is free to move around the lattice

Owing to its negative charge carriers (free electrons), the resulting semiconductor is known as *n-type*

Arsenic is known as a *donor* since it donates an electron

---

p-type Si

- p-type silicon (Group IV) is doped with boron (B, Group III)
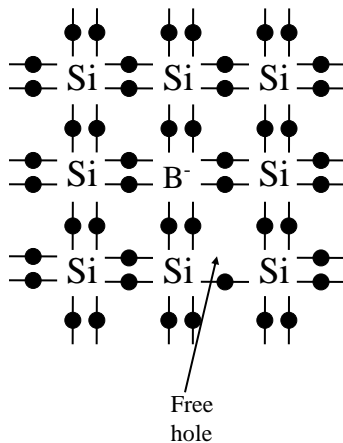
Free
hole

The B atom has only 3 valence electrons, it accepts an extra electron from one of the adjacent Si atoms to complete its covalent bonds

This leaves a *hole* (i.e., absence of a valence electron) in the lattice

This hole is free to move in the lattice – actually it is the electrons that do the shifting, but the result is that the hole is shuffled from atom to atom

Owing to its positive charge carriers (free holes), the resulting semiconductor is known as *p-type*

B is known as an *acceptor*

Semiconductors

- The Metal Oxide Semiconductor Field Effect Transistor (MOSFET) devices that are used to implement virtually all digital logic circuits are fabricated from *n* and *p* type silicon
- Later on, we will see how MOSFETs can be used to implement digital logic circuits
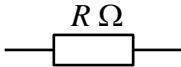
Circuit Theory

- Electrical engineers have an alternative (but essentially equivalent) view concerning pd.
- That is, conductors, to a greater or lesser extent, oppose the flow of current. This 'opposition' is quantified in terms of *resistance* (R). Thus the greater is the resistance, the larger is the potential difference measured across the conductor (for a given current).

Circuit Theory

- The *resistance* (R) of a conductor is defined as $R=V/I$, where V is the pd across the conductor and I is the current through the conductor.

- This is know as *Ohms Law* and is usually expressed as $V=IR$, where resistance is defined to be in Ohms (Ω).

- So for an *ohmic* (i.e., linear) conductor, plotting I against V yields a straight line through the origin

Circuit Theory

- Conductors made to have a specific value of resistance are known as *resistors*.

- They have the following symbol in an electrical circuit: $R\,\Omega$

- Analogy:
  - The flow of electric charges can be compared with the flow of water in a pipe.
  - A pressure (voltage) difference is needed to make water (charges) flow in a pipe (conductor).

Circuit Theory

- Kirchhoff's Current Law – The sum of currents entering a junction (or node) is zero, e.g.,

$I_1 + I_2 - I_3 = 0$ or $I_1 + I_2 = I_3$
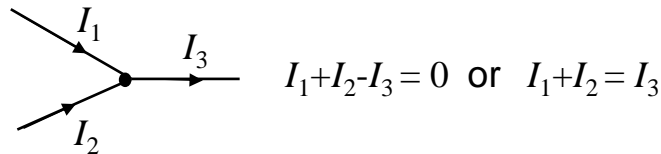
- That is, what goes into the junction is equal to what comes out of the junction – Think water pipe analogy again!

Circuit Theory

- Kirchhoff's Voltage Law – In any closed loop of an electric circuit the sum of all the voltages in that loop is zero, e.g.,

$V_1 - V_2 - V_3 - V_4 - V_5 + V_6 = 0$
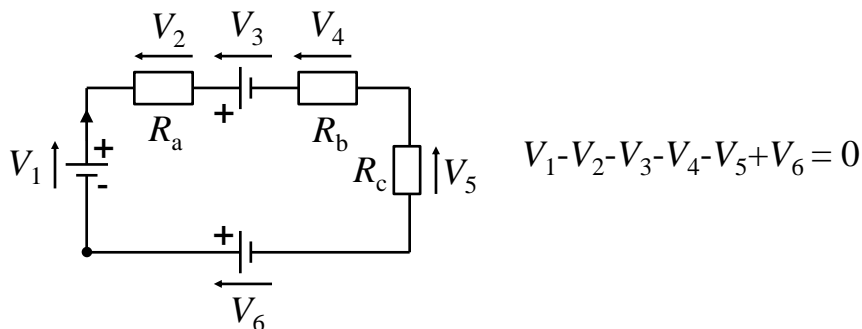
- We will now analyse a simple 2 resistor circuit known as a *potential divider*

Potential Divider

- What is the voltage at point x relative to the 0V point?
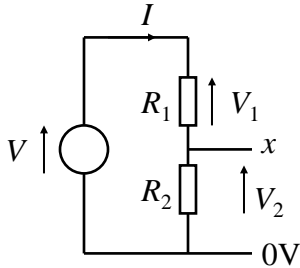
$$V = V_1 + V_2$$

$$V_1 = IR_1 \qquad V_2 = IR_2$$

$$V = IR_1 + IR_2 = I(R_1 + R_2)$$

$$I = \frac{V}{(R_1 + R_2)}$$

$$V_x = V_2 = \frac{V}{(R_1 + R_2)} R_2 = V\left(\frac{R_2}{R_1 + R_2}\right)$$

Note: circle represents an ideal voltage source, i.e., a perfect battery

Solving Non-linear circuits

- Not all electronic devices have linear I-V characteristics, importantly in our case this includes the FETs used to build logic circuits
- Linear means that superposition applies:
  - If an input $x_1(t)$ gives an output $y_1(t)$, and input $x_2(t)$ gives an output $y_2(t)$, then input $[x_1(t)+x_2(t)]$ gives an output $[y_1(t)+y_2(t)]$
- For a circuit that includes a non-linear component, we cannot use the algebraic approach. As an introduction, we will now use a graphical approach to solve the previous (linear) potential divider circuit.

Potential Divider

- From the previous potential divider equation, we can get the analytical solution as follows
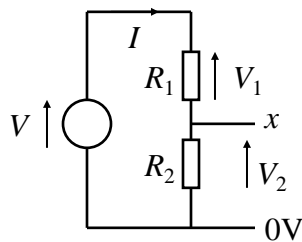
So if $V = 10V$, $R_1 = 1\Omega$ and $R_2 = 2\Omega$

$$V_x = V\left(\frac{R_2}{R_1 + R_2}\right) = 10\left(\frac{2}{1+2}\right) = 6.7\text{V}$$

- How can we do this graphically?

Potential Divider
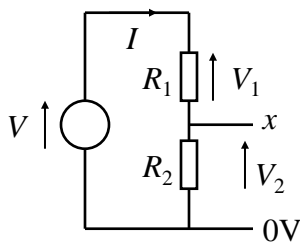
- First we can plot the current/voltage curve for R_2 i.e., straight line – Ohms law

Current through R_2 (2Ω)

(Amps)

0   Voltage across R_2 i.e., V_2 (Volts)
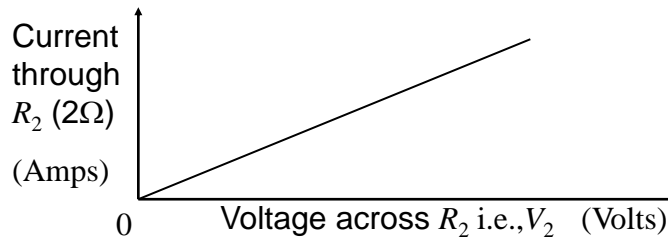
Potential Divider

- Then we can plot the current/voltage curve for R_1 i.e., straight line – Ohms law
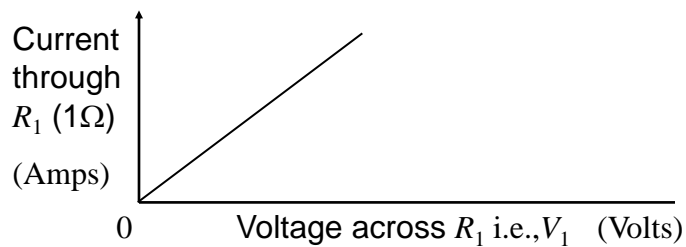
Potential Divider

- So, how is the voltage across R_1 , i.e., V_1 , reflected in voltage V_2 ?
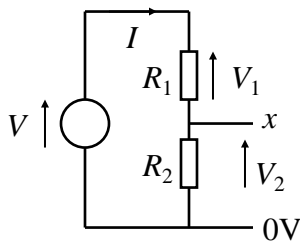
- We see that $V_2 = V - V_1$, where V = 10 Volts, so, $V_2 = 10 - V_1$
- i.e., as V_1 increases, V_2 decreases
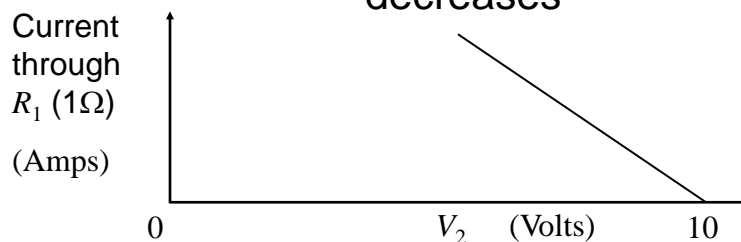
Potential Divider

- Now plotting both curves simultaneously
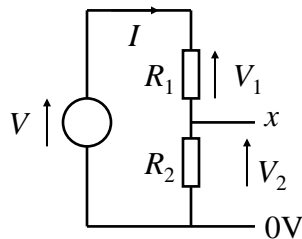
- The solution for V_2 is when the current through both resistors is equal, i.e., where the 2 curves intersect
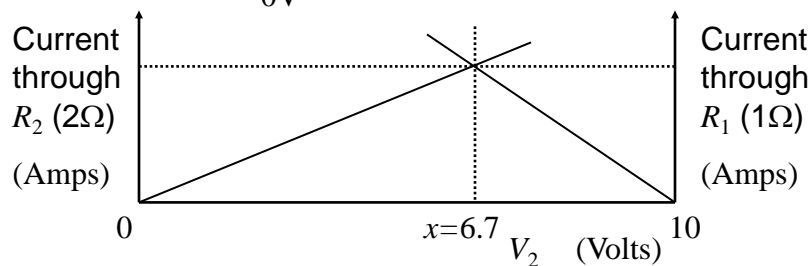
Graphical Approach

- Clearly this approach works for a linear circuit.
- How could we apply this if we have a non-linear device, e.g., a transistor in place of R_2?
- What we do is substitute the $V\text{-}I$ characteristic of the non-linear device in place of the linear characteristic (a straight line due to Ohm's Law) used previously for R_2

Graphical Approach
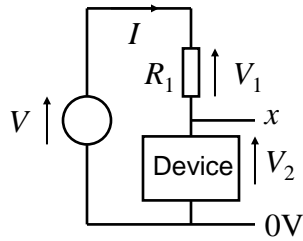
So if $V = 10V$ and $R_1 = 1\Omega$

The voltage at x is a Volts as shown in the graph

Current through Device (Amps)

Current through R_1 (1Ω) (Amps)

Device characteristic

0 $x=a$ 10

V_2 (Volts)
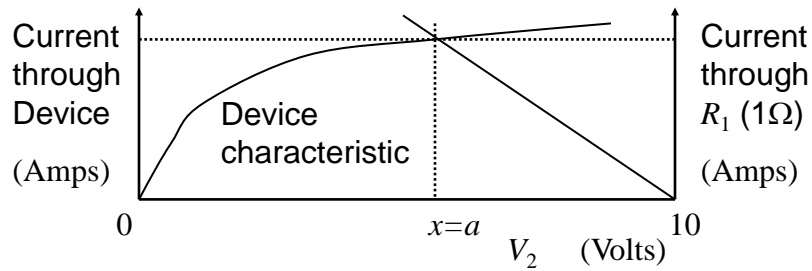
Digital Electronics: Electronics, Devices and Circuits

Transistors and Gates

Introduction

- Basic introduction to the p-n junction
- Operation an characteristics of Metal Oxide Semiconductor Field Effect Transistors (MOSFETs)
- n-MOS inverter, characteristics and problems
- Complimentary MOS (CMOS) inverter and other logic gates
- Other logic families
- Noise margin

p-n Junction

- The key to building useful devices is combining p and n type semiconductors to form a p-n junction
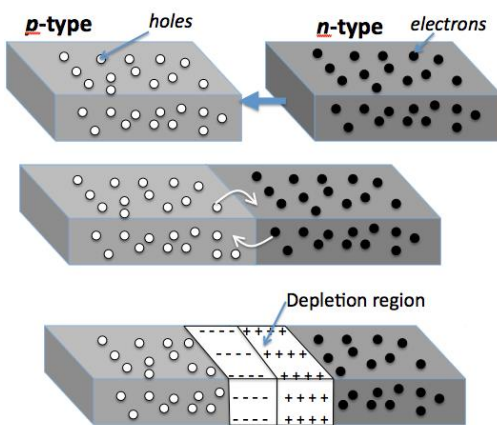
- Electrons and holes diffuse across junction due to large concentration gradient
- On n-side, diffusion out of electrons leaves +ve charged atoms
- On p-side, diffusion out of holes leaves -ve charged atoms
- Leaves a space-charge (depletion) region with no free charges
- Space charge gives rise to electric field that opposes diffusion

- Equilibrium is reached where no more charges move across junction

Biased p-n Junction
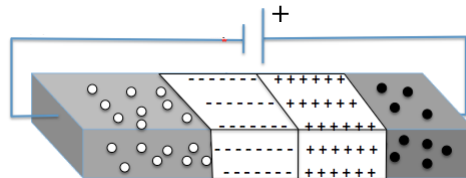
- **_Reverse bias_:** By making n-type +ve, electrons are removed from it increasing size of space charge region. Similarly holes are removed from p-type region. Thus space charge region and its associated field are increased.

- The current flow, known as the reverse saturation current is of the order of nA, i.e., essentially zero.

- So when a p-n junction is 'reverse biased' no current flows.

Biased p-n Junction
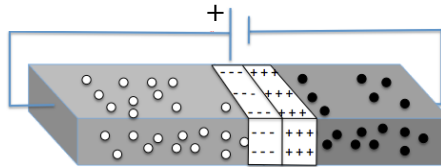
- With **_forward bias_**, on the p-side holes are pushed toward junction where they neutralise some of the –ve space charge.
- Similarly on the n-side, electrons are pushed toward the junction and neutralise some of the +ve space charge.
- So depletion region and associated field are reduced.
- This allows diffusion current to increase significantly
- Thus the p-n junction allows significant current flow in only one direction
- So a significant current flows only when 'forward' biased
- A device with a single p-n junction is known as a **_diode_**

---

n-Channel MOSFET

- We will now briefly introduce the n-channel MOSFET

- The charge carriers in this device are electrons

Drain
(D)

Gate
(G)

Source
(S)
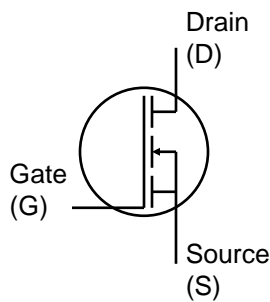
The current flow from D to S (I_{DS}) is controlled by the voltage applied between G and S (V_{GS}), i.e., G has to be +ve wrt S for current I_{DS} to flow (transistor **On**)

We will consider enhancement mode devices in which no current flows (I_{DS}=0, i.e., the transistor is **Off**) when V_{GS}=0V

n-Channel MOSFET

OFF

$+V_D$

D

G

S

Silicon dioxide insulator

n+

n+

p-type

0V

Reverse biased p-n junctions
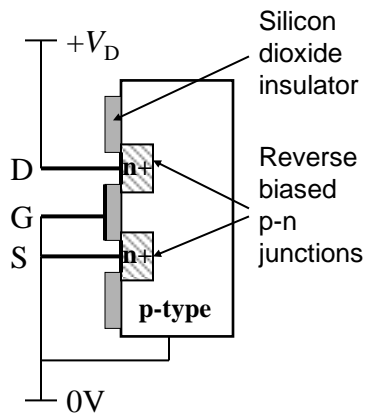
Drain (and Source) diode reverse biased, so no path for current to flow from S to D, i.e., the transistor is **off**

---

n-Channel MOSFET

ON

$+V_D$

$+V_G$

D

G

S

Silicon dioxide insulator

n+

n+

p-type

0V

n-type layer: 'inversion'
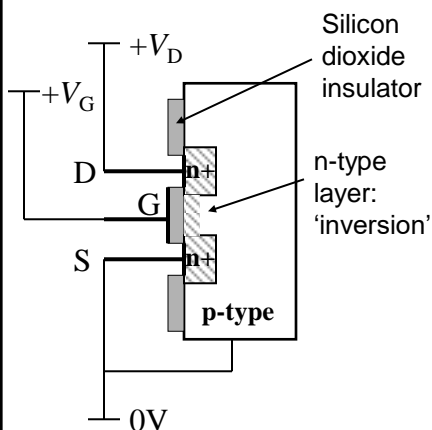
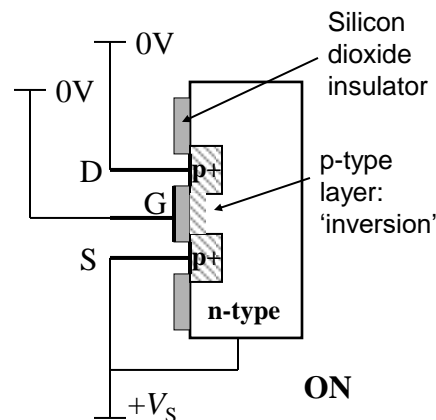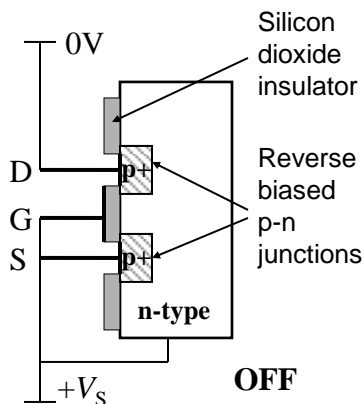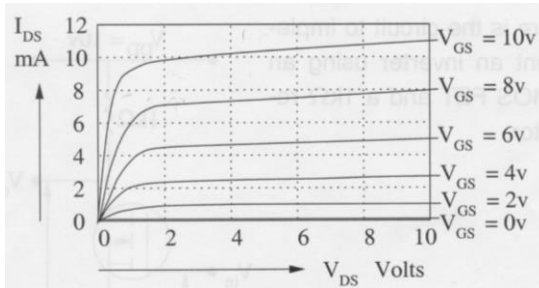Consider the situation when the Gate (G) voltage (V_G) is raised to a positive voltage, say V_D

Electrons attracted to underside of the G, so this region is 'inverted' and becomes n-type. This region is known as the *channel*

There is now a continuous path from n-type S to n-type D, so electrons can flow from S to D, i.e., the transistor is **on**

The G voltage (V_G) needed for this to occur is known as the *threshold voltage* (V_t). Typically 0.3 to 0.7 V.

p-Channel MOSFET

- Similarly we have p-channel MOSFETs where the charge carriers are holes

Drain
(D)

Gate
(G)

Source
(S)
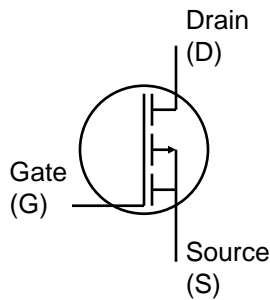
The current flow from S to D (I_{DS}) is controlled by the voltage applied between G and S (V_{GS}), i.e., G has to be -ve wrt S for current I_{DS} to flow (transistor **On**)

We will be consider enhancement mode devices in which no current flows (I_{DS}=0, i.e., the transistor is **Off**) when V_{GS}=0V

p-Channel MOSFET

- p-channel MOSFETs have the opposite construction to n-channel MOSFETs, i.e., n-type substrate and p-type S and D regions

0V

Silicon dioxide insulator

D

G

S

p+

p+

Reverse biased p-n junctions

n-type

$+V_S$

OFF

0V

0V

D

G

S

p+

p+

Silicon dioxide insulator

p-type layer: 'inversion'

n-type

$+V_S$

ON

n-MOSFET Characteristics

Plots V-I characteristics of the device for various Gate voltages (V_{GS})
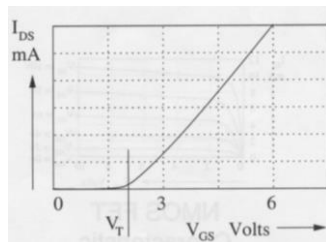
At a constant value of V_{DS}, we can also see that I_{DS} is a function of the Gate voltage, V_{GS}

The transistor begins to conduct when the Gate voltage, V_{GS}, reaches the Threshold voltage: V_T

n-MOS Inverter

$R_1 = 1\text{k}\Omega$, V_1, I

$V_{DD} = 10\text{V}$

V_{out}, V_{DS}, V_{in}, V_{GS}, 0V
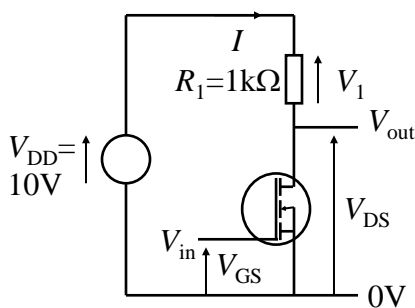
We can use the graphical approach to determine the relationship between V_{in} and V_{out}
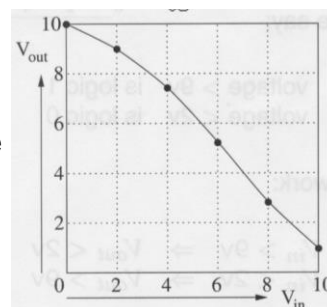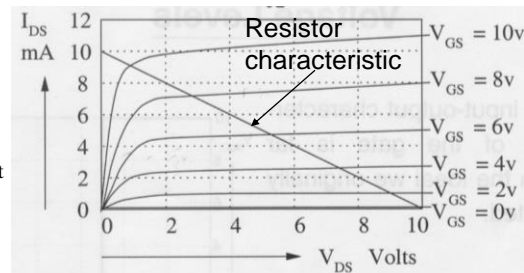
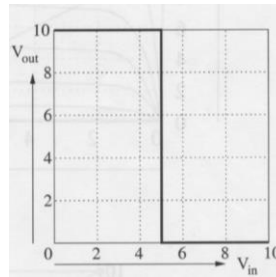Note $V_{in}=V_{GS}$ and $V_{out}=V_{DS}$

n-MOS Inverter

• Note it does not have the 'ideal' characteristic that we would like from an 'inverter' function
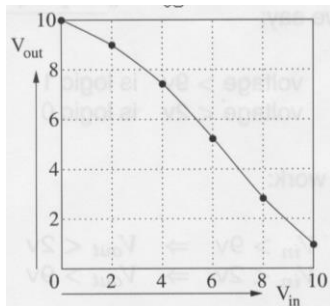
Actual
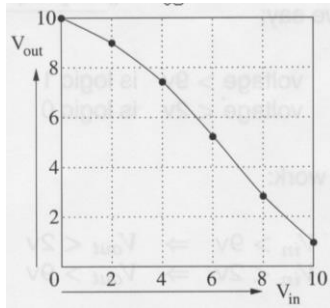
Ideal

However if we specify suitable voltage thresholds, we can achieve a 'binary' action.

---

n-MOS Inverter

Actual

So if we say:

voltage > 9V is logic 1

voltage < 2V is logic 0

The gate will work as follows:

$V_{in} > 9V$ then $V_{out} < 2V$ and if

$V_{in} < 2V$ then $V_{out} > 9V$

n-MOS Logic

- It is possible (and was done in the early days) to build other logic functions, e.g., NOR and NAND using n-MOS transistors
- However, n-MOS logic has fundamental problems:
  - Power consumption
  - Slow output transition times from low to high voltage levels when connected to capacitive loads
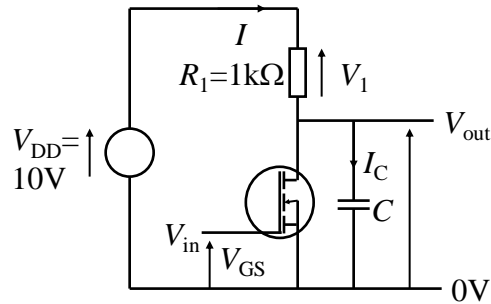
n-MOS Logic

- For example the metal track used on circuit boards to connect gate inputs and outputs has a finite capacitance to ground, i.e., to the 0V connection.
  - We modify the circuit model to include this *stray capacitance C*
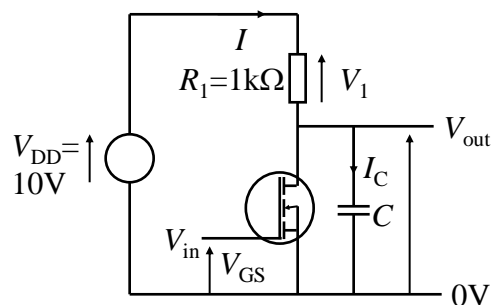
- This significantly increases the rise time of the output signal, V_{out}

n-MOS Logic

- When the transistor turns-off (open circuit), capacitor C modelling the stray capacitance, charges through R_1. So the rise-time of V_{out} is controlled by R1. When the transistor turns-on (short circuit), C discharges through the transistor with on resistance R_{ON}. So the fall-time of V_{out} is controlled by R_{ON}.

- Since $R_1 > R_{ON}$, rise time > fall time for V_{out}

n-MOS Logic

- Power consumption is also a problem
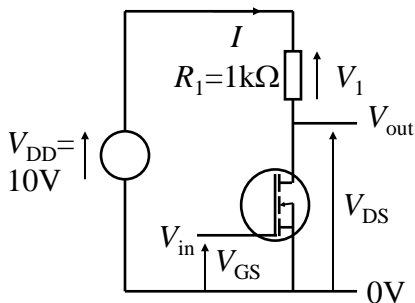
Transistor OFF

No problem since no current is flowing through R_1, i.e., V_{out} = 10V
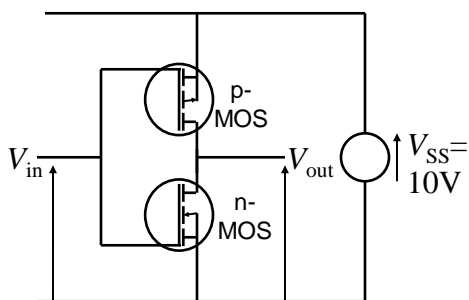
Transistor ON

This is a problem since current is flowing through R_1 . For example, if V_{out} = 1V (corresponds with V_{in} = 10V and $I_D = I =$ 9mA), the power dissipated in the resistor is the product of voltage across it and the current through it, i.e.,

$$P_{disp} = I \times V_1 = 9 \times 10^{-3} \times 9 = 81\,\text{mW}$$

CMOS Logic

- To overcome these problems, complementary MOS (CMOS) logic was developed
- As the name implies it uses p-channel as well as n-channel MOS transistors
- Essentially, p-MOS transistors are n-MOS transistors but with all the polarities reversed!

CMOS Inverter

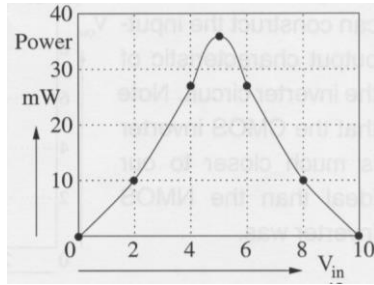Using the graphical approach we can show that the switching characteristics are now much better than for the n-MOS inverter

V_{in}	N-MOS	P-MOS	V_{out}
low	off	on	high
high	on	off	low

185
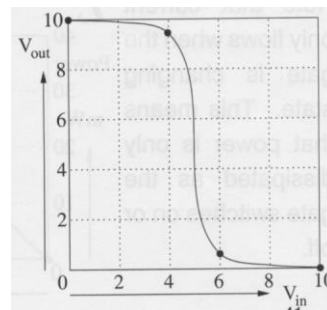
CMOS Inverter

- It can be shown that the transistors only dissipate power while they are switching.

This is when both transistors are on. When one or the other is off, the power dissipation is zero

CMOS is also better at driving capacitive loads since it has a p-MOS transistor (instead of a resistor) controlling the rising edge of the output signal

CMOS Gates

- CMOS can also be used to build NAND and NOR gates
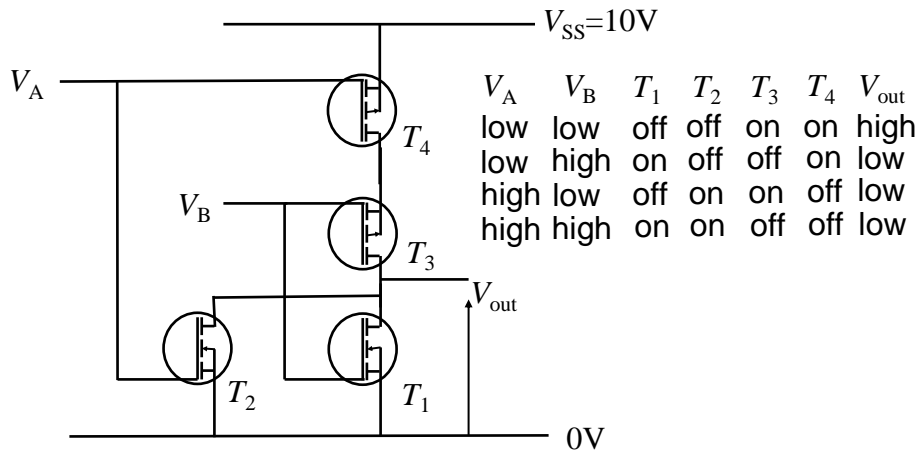- They have similar electrical properties to the CMOS inverter

CMOS Gates

- To ease analysis of the following circuits it is worth recapping the function of the transistors.
- For both n and p-type MOS transistors
  - If there is no potential difference (pd) between Gate (G) and Source (S), the transistor is Off, i.e., an open circuit between Source (S) and Drain (D)
  - If there is a sufficiently large pd between Gate and Source, the transistor is On, i.e., a short circuit between Source (S) and Drain (D) – Note for n-MOS G is more +ve than S and for p-MOS G is more -ve than S

CMOS NAND Gate
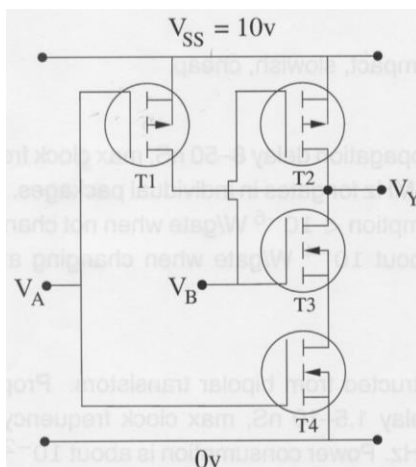
V_A	V_B	T1	T2	T3	T4	V_Y
low	low	on	on	off	off	high
low	high	on	off	on	off	high
high	low	off	on	off	on	high
high	high	off	off	on	on	low

CMOS NOR Gate

$V_{SS}=10V$

V_A	V_B	T_1	T_2	T_3	T_4	V_{out}
low	low	off	off	on	on	high
low	high	on	off	off	on	low
high	low	off	on	on	off	low
high	high	on	on	off	off	low

Logic Families

- **NMOS** – compact, slow, cheap, obsolete
- **CMOS** – Older families slow (4000 series about 60ns), but new ones (74AC) much faster (3ns). 74HC series popular
- **TTL** – Uses bipolar transistors. Known as 74 series. Note that most 74 series devices are now available in CMOS. Older versions slow (LS about 16ns), newer ones faster (AS about 2ns)
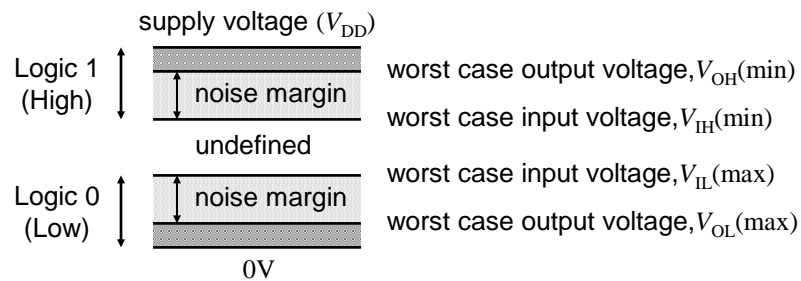- **ECL** – High speed, but high power consumption

Logic Families

- Best to stick with the particular family which has the best performance, power consumption cost trade-off for the required purpose
- It is possible to mix logic families and sub-families, but care is required regarding the acceptable logic voltage levels and gate current handling capabilities

Meaning of Voltage Levels

- As we have seen, the relationship between the input voltage to a gate and the output voltage depends upon the particular implementation technology
- Essentially, the signals between outputs and inputs are 'analogue' and so are susceptible to corruption by additive noise, e.g., due to cross talk from signals in adjacent wires
- What we need is a method for quantifying the tolerance of a particular logic to noise

Noise Margin

- Tolerance to noise is quantified in terms of the noise margin

supply voltage (V_{DD})

Logic 1 (High) — noise margin — worst case output voltage, $V_{OH}(min)$
worst case input voltage, $V_{IH}(min)$

undefined

worst case input voltage, $V_{IL}(max)$
Logic 0 (Low) — noise margin — worst case output voltage, $V_{OL}(max)$

0V

Logic 0 noise margin = $V_{IL}(max)$ - $V_{OL}(max)$
Logic 1 noise margin = $V_{OH}(min)$ - $V_{IH}(min)$

Noise Margin

- For the 74 series High Speed CMOS (HCMOS) used in the hardware labs (using the values from the data sheet):

Logic 0 noise margin = $V_{IL}(max)$ - $V_{OL}(max)$
Logic 0 noise margin = $1.35 – 0.1 = 1.25$ V

Logic 1 noise margin = $V_{OH}(min)$ - $V_{IH}(min)$
Logic 1 noise margin = $4.4 – 3.15 = 1.25$ V

See the worst case noise margin = 1.25V, which is much greater than the 0.4 V typical of TTL series devices. Consequently HCMOS devices can tolerate more noise pick-up before performance becomes compromised