About these notes

These notes form part of the teaching materials for the first year course unit COMP12111: Fundamentals of Computer Engineering (formerly COMP10211).

This course introduces digital logic and its application in computer organisation and design. The major emphasis is on practical design work and in the laboratory state-of-the-art computer-aided design tools are used to support the design of digital hardware systems. In the lab you will produce designs, simulate them and then implement them using the electrically reconfigurable gate arrays found on the experimental boards in the laboratory.

The lectures initially support the laboratories but progress to a wider overview of the design and interaction of computer hardware systems. Ultimately a complete - if simple - computer is described as constructed from simple gates.

Organisation of these notes

These handouts contain copies of the lecture slides as well as "extra" information relevant to the material being taught. All material is examinable. Further information can be found on the course eLearning site.

eLearning Resources

Further information, such as copies of these workbooks, copies of the slides, extra notes, corrections to the notes, etc, can be found on blackboard through my.manchester.

The Blackboard site for the course also includes quizzes, worked examples etc. It also includes videos illustrating key concepts covered in this part of the course unit. These videos are in addition to the podcasting of lectures undertaken by the University, which are available through the University Blackboard page for this course unit.

Further help and advice can be found in the School of Computer Engineering Wiki

https://wiki.cs.manchester.ac.uk/engineering/

Assessment

This course unit is assessed by a formal examination and laboratory work (50:50 split). The examination lasts 1hr 30mins and has two parts – section A and section B. Both sections contain two questions, you must answer both questions 1 and 2 in Section A (which is compulsory). You must choose to answer one question from two in section B.

This course has been running in its current form since the 2008-2009 academic year. Hence, when looking at past papers (all of which are available of the University intranet) take care when looking at papers prior to 2008. If the question looks unfamiliar, the chances are the subject matter is no longer covered. No sample answers are given for past papers. However, I will be happy to discuss any questions relating to the material I cover in lectures, providing some attempt to answer them has been made beforehand.

If you are stuck …

I'm happy to answer any questions relating to the module (well, my course material certainly). If you want to speak to me either catch me at the end of lectures, in the lab, or come and see me in my office. If you are planning on visiting me in my office then, if possible, please email me to check my availability, that way I'll make sure I set aside enough time to answer your questions. If you knock on my door out of the blue, then I may not have the time to see you!

Further information, such as copies of these workbooks, copies of the slides, extra notes, corrections to the notes, etc, can be found on the course unit Moodle site.

Finally, if you notice any mistakes, then please let me know … I'm not perfect and my notes definitely aren't … ☺.

References

I acknowledge the following references, which I have used to prepare these notes:

1. COMP10211 Lecture Notes (pre-2010) by Dr Jim Garside, School of Computer Science, The University of Manchester.
2. "Principles of Computer Hardware", A Clements, 4th Edition, Oxford University Press, ISBN 978-0-19-927313-3
3. "Digital Design with RTL Design, VHDL, and Verilog", F Valid, 2nd Edition, Wiley, ISBN 978-0-470-53108-2
4. "The Verilog hardware description language", Thomas and Moorby, 5th Edition, Springer, ISBN 978-0-387-84930-0.
5. "Digital Design", M M Mano, 3rd Edition, Prentice Hall, ISBN 0-13-035525-9
6. "Fundamentals of Logic Design", C H Roth Jr., 4th Edition, PWS Publishing, ISBN 053495472-3.

MANCHESTER 1824

The University of Manchester

COMP12111: Fundamentals of Computer Engineering

Part I
Course Overview &
Introduction to Logic

Christoforos Moutafis

Lecture Aims

The aims of these lectures are:

- to provide an overview of the course unit
- to introduce binary data representation and arithmetic operations
- to introduce the basic Boolean functions and their characteristics
- to discuss the basic rules of Boolean algebra
- to introduce logic gates – the basic building blocks of digital systems
- to investigate the implementation of Boolean expressions using logic gates
- to discuss hierarchy and abstraction
- to introduce sequential circuits and flip-flops

Quiz-time

Q. What has led to the development in consumer products over the past 10 years?

A. Software

A. Hardware

A. Both?

Notes:

Quiz-time

Q. What is the improvement in the performance (approx) of the main processor in an iPhone 6 compared to the Apollo moon mission guidance computer?

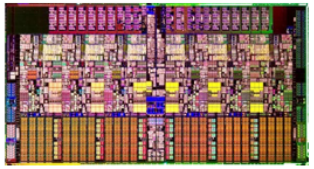
A. About the same
B. Around 1,000x faster
C. Around 1,000,000x faster
D. Around 100,000,000x faster
E. Around 1,000,000,000x faster

Notes:

MANCHESTER 1824

The University of Manchester

Introduction



Intel Core i7 six-core processor

What is this course about?

- Computer hardware design
  - *not* 'electronics' … nothing nasty like transistors!
  - from logic gates to computers

What should you get from it?

- An overview of the hardware design process
  - experience of various 'levels' of design
  - some of the techniques involved
- An appreciation of the complexity involved …
  - … and how it is managed sensibly
- A flavour of the subject area
  - maybe you will want to learn more … maybe not

Notes:

Some 'Gee Whiz' statistics

The numbers involved are very hard – probably impossible – to conceive, but let's try some.

Speed

The speed of light (in vacuum): 3×10^8 m/s or 186,000 miles/s

That's quite fast. Let's put it another way,

"1 foot per nanosecond"

(as you're all metric, a foot is approximately 30 cm, but I guess you knew that).

How long is a nanosecond? Well there are 1,000,000,000 of them in one second … so one nanosecond corresponds to a time that is 1/1,000,000,000 of a second.

Going at this speed it takes the light from the sun approximately 8 minutes to reach the earth. The lecture theatre is about 50' (50 foot) deep. So, for those at the back, a photon from the projector takes about 100ns to make the round trip to the screen and back to your eye.

In that time a workstation processor will execute about 300-500 instructions. That's on each processor core – there may be several. (People at the front of the lecture theatre are over 100 instruction times ahead!)

Or considering arithmetic performance, one silicon chip can compute faster than everyone on Earth, even if you could organise everyone to work together.

Size

You may have heard of 'microchips'; "micro-" is hardly applicable any more. If you look for the wires on a chip (you can try with a microscope) you won't see them. The 'feature size' at the state-of-the-art is now well below 100 nm and the wavelength of visible light lies roughly in the 400-800 nm range, so you can't resolve them. A human hair is about 100,000 nm in diameter, so you could fit nearly 5000 devices across the diameter using today's state-of-the art 22 nm technology.

As the features are so small lots of them can be packed onto a single chip. Chip sizes vary, but think of something less than ~2cm on a side as a guide. The transistor count of a largish chip will be a few billion (U.S. billion = 10^9) – that is a big number (it would take you 40,000 years earning the average British salary to earn £1billion). Transistor count has been observed to obey something called "Moore's Law" which observes that the number of transistors on a chip doubles every (approximately) 18 months. This has (approximately) held true since the first silicon chips – about 40 years or more than 25 generations. Eventually it must stop – but not yet.

For a given design – say a 32-bit ARM microprocessor, which you will encounter in other courses and probably own several of them in various devices such as mobile phones – the silicon area needed is, of course, quite small: well under $1mm^2$.

Cost

'Cheap as chips'. The cost of silicon chips is sufficiently small that they can usually be regarded as disposable items. This is because they are mass produced (and *can be* mass produced) and benefit from economies of scale.

'Masking charges' for a state-of-the-art device (that's making the tools for the production line) will cost ~$1M these days, and is increasing with the difficulty for making smaller and smaller devices.

The factories (or "fabs"[1]) now cost a few billion dollars to build – and go out of date in a few years – so that represents significant investment. Only a few huge companies such as Intel, Samsung, etc. can now afford their own private fabs.

[1] Fabrication plant

[1] **A**merican **S**tandard **C**ode for the **I**nterchange of **I**nformation

Academic Staff

Dr Paul Nutter
Course Unit Leader

Email: p.nutter@manchester.ac.uk
Office: IT119 – Ground floor IT building

Dr Christoforos Moutafis

Email: christoforos.moutafis@manchester.ac.uk
Office: IT113 – Ground floor IT building

Dr Vasilis Pavlidis

Email: vasileios.pavlidis@manchester.ac.uk
Office: IT210 – 2nd floor IT building

Prof Tom Thomson

Email: thomas.thomson@manchester.ac.uk
Office: IT121 – Ground floor IT building

Notes:

Contacting Staff

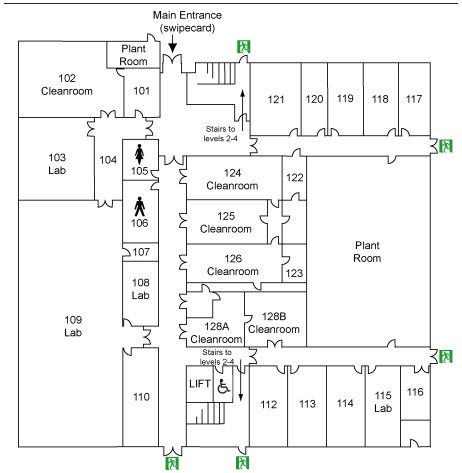
If you need to contact a member of staff regarding material covered in this course unit, or the laboratory exercises, then in the first instance use email to arrange a suitable date/time. Do not rely on us being available in our offices when you need to speak to us.

You can catch us at the end of lectures, or during the laboratory, but this doesn't leave much time to discuss any problems in any great detail.

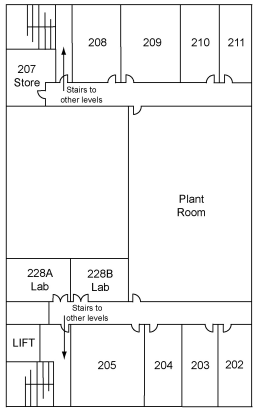
Office Locations

We all live in the IT building. You access the IT building via the bridge on the first floor of Kilburn building (this takes you to level 4 in the IT building). Follow the corridor to the right and near the end you will find the stairs that will take you down to levels 1 (the ground floor) and 2.

Ground floor IT building:



2nd floor IT building:



MANCHESTER 1824

The University of Manchester

What does it involve?

Lectures

- approx. 22 lectures, two per week

Assessment

- Laboratory – put knowledge into practice
- Examination – test lecture and laboratory material
  - 1 ½ hour exam in January
  - tests both **knowledge** and **application** of principles
- Laboratory and examination both count towards 50% of the final mark

Laboratories

- 10, scheduled weekly, each 2 hours (day/time depends on lab group)
- Start next week (Week 2) – you need to collect a copy of the lab manual from the student support office – make sure you prepare beforehand!

Notes:

What will you be doing?

There are plenty of lectures and laboratories to keep you entertained - you will have plenty to do!

Remember, lecture attendance at **ALL** lectures is compulsory for **ALL** 1st year undergraduates; this includes COMP12111. It is important that you attend the lectures, as the lecture material supports the laboratory exercises, and is essential in order for you to pass the exam! In most cases you will be provided with the notes beforehand, so make sure you prepare for the lecture.

Laboratories

You will have lots to do in the laboratories, and not necessarily enough time to complete the exercises. Plan your time effectively: read the manual/exercise before the lab and make a start. That way, if you do have questions, then you have plenty of time to seek advice.

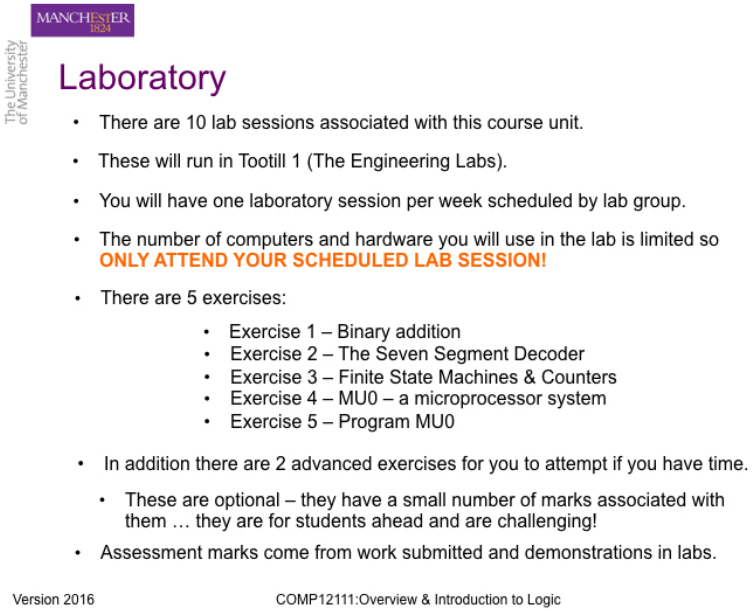
We have demonstrators present in the lab to help you with any problems you are experiencing; these are typically PhD students in the school. Demonstrators are not present to solve problems for you, but to guide you in the right direction and offer advice where necessary.

Lab exercises have deadlines associated with them, by which time you must have shown your work to a demonstrator/member of academic staff and handed in the completed work required.

Assessment

The course unit is assessed by laboratory work and examination (50% each).

The examination consists of two sections: Section A and Section B. The two questions in Section A are compulsory – you must answer 5 out of the 8 sub-questions available for each question. You must also one of the two questions from section B. Generally one question from both sections will cover the work from the first half of the course, i.e. mine, and one question will cover the work from the second half of the course, i.e. Vasilis.



Notes:

Laboratories

There are five exercises:

Exercise 1 – Binary Adders

Prior to this exercise there is a walkthrough that you must follow where we take you through the design of a half adder step-by-step using the Cadence design framework CAD tools. It is important you complete this walkthrough so that you become familiar with the design steps and the tools themselves, and because you will be using your half-adder design to implement a full adder.

Starting from your half-adder design you will continue to design a full-adder, a 4-bit adder and finally a 16-bit adder. The aim of this exercise if to introduce you to the design tools and highlight the use of hierarchy in the design process.

Exercise 2 – Seven Segment Decoder

In this exercise you will design a circuit that takes a 4-bit binary value and converts this to an 8-bit value to drive a standard 7 segment display. This is the first time you will use Verilog, a hardware description language, to implement a design by a software description rather than by drawing a schematic. You will test your design by downloading it down onto an FPGA experimental board in the laboratory where you will see your design working using the 7 segment decoders on the board.

Exercise 3 – Finite State Machines & Counters

In this exercise you will design a circuit for a 0-9 modulo-10 counter in Verilog (a counter that goes from 0 -> 9 and repeats). Following this you will use your counter in the design of a traffic lights to simulate the operation of the traffic lights at a cross roads junction. You will be given part of the finite state machine for this exercise, but it need completing..

Exercise 4 – MU0 – a microprocessor system

You will build your very own RISC processor – the MU0. Starting with a partially completed datapath you will complete the design, test and download it onto the experimental board where you will run some software. There's an opportunity to take this further by writing your own software for the MU0.

Exercise 5 – Program MU0

Freedom to play with MU0 and use the hardware on the experimental board to produce your own programs.

Optional Exercises

There are two optional exercises, which must ONLY be attempted if you are ahead in the labs. They count for 10% of the lab marks and are there for students who want to take the exercises a bit further. The two optional exercises are:

Exercise 3x – Implement a 000-999 counter using instances of your mod-10 counter.

Exercise 5x – Extend the MU0 implementation.

It is important that you acknowledge that you won't get 100% in the lab. It has been designed to give an average mark around 70%.

Lab Sessions

It is **ESSENTIAL** that you **ONLY ATTEND YOUR OWN SCHEDULED LAB SESSION**. We have a limited number of computers/hardware and the labs can be very busy. If you are found using a PC in another lab session, other than your own, you will be asked to leave. You will not be allowed to demonstrate your work outside of your own scheduled lab session. In addition, do not use the scheduled lab to catch up on lab work for other course units, if you are found doing so your attendance mark will not be taken.

Submission of Lab Work

- We use online tools to mark some of your work, with detailed feedback provided.
- You submit your work using the script:

  12111submit

- Exercise 1 will not count towards your final lab mark, but you must complete it (as you will need the designs later) and submit it. You will receive feedback.
- For exercises 2 – 5 you will need to arrange a demonstration of your work in the next lab session after submission.
  - Print out a labprint sheet and complete **BEFORE** your demonstration
  - Write your name and machine number on the whiteboard and wait to be seen by a TA.
- After your demonstration you submit the mark using 12111submit
- You need to request extensions at the **END** of the lab, they are not automatic
  - only to the **START** the next scheduled lab

Notes:

Submission Procedure

The method of submission depends on the exercise. For exercise 1 you should complete the answer sheet provided and have it signed off by a demonstrator. Once you have done this you need to complete a submission card (available at the front of the lab), attach this to your answer sheet and post them in the post box at the front of the lab. For exercises 2-5 (and the optional exercises) you will use submit to submit your work using the script:

12111submit

this will inform Arcade that you have submitted your work for marking, and the date/time will be recorded in order to determine if the deadline has been met or not. If you work is submitted after the deadline has passed it will be recorded as having been submitted LATE.

Once the deadline has passed submitted designs will be marked automatically (after the extended deadline for the lab). The marking system will look at the operation of your design to ensure it works as expected, as well as identifying the level of testing you have undertaken. From this you will receive a mark and feedback via email.

In your next lab you must arrange a follow-up demonstration where you will show your work to a demonstrator who will award additional marks according to the neatness of your design, your Verilog code structure, comments, etc. To arrange a demonstration you must write your name and machine number on the whiteboard at the front of the lab to indicate to us that you would like to demonstrate your work. Once you've done this a demonstrator will come and look at your design, ask you some questions and give you a token and mark for the demonstration, which you should submit using the 12111submit script.

Note: please make sure you follow the instructions carefully when using 12111submit.

Deadlines

The deadlines for each exercise is at the end of the lab associated with the deadline. There are NO AUTOMATIC EXTENSIONS for the exercises. You must request an extension before the end of the lab – whether you are granted an extension depends how far you have got with the exercise! Extensions will run until the START of the next lab (i.e. 3pm for a 3-5pm lab). Work submitted after this point will be considered as LATE.

MANCHESTER 1824

The University of Manchester

eLearning resources

eLearning resources are available in Blackboard via my.manchester

which also contains resources for this course unit: copies of notes, lab information etc

The Blackboard page contains:

- videos of key concepts
- animations to aid understanding
- quizzes and questions

The automatic marking of submitted lab work provides consistent, detailed and timely feedback on your work.

Also see the engineering wiki pages: wiki.cs.manchester.ac.uk/engineering/

Significant work has gone into developing the eLearning resources for this half of the course unit – please make use of them and provide me with feedback!

Notes:

e-learning Resources

A significant amount of time has been invested in developing e-learning resources for this course unit, these include:

1. The use of the submit mechanism to submit work
2. The use of automatic marking of work in order to provide consistent, detailed, and timely feedback.
3. Blackboard resources – course notes, animations, quizzes and questions to aid understanding of the course material.

I'd be interested to hear your opinions on any of these resources, positive or negative. Please note: these are still under development.

The submission and automatic marking mechanism was introduced for the 2012-13 academic year, was significantly improved last year, and has been significantly extended this academic year.

Designing Complex Systems

Any complex engineered system – and computers are examples of complex systems! – is built in hierarchical layers with details abstracted.

Hierarchy

Hierarchy is a group of objects arranged in rank order of magnitude, importance, or complexity. You can consider a country to exhibit a hierarchical structure, as it contains a number of counties, and each county contains a large number of towns and cities.

Hierarchy is useful in digital design as it allows us to manage complex designs by breaking down the system into more manageable, and easier to implement, sections, which can be easily re-used. Thus, hierarchy allows us to produce the building blocks (of much simpler logical functions) that we connect together to form a more complex design.

Abstraction

Abstraction is a means of hiding the detail – it frees the designer from having to remember, or even understand, the low-level details of a component. For example, the designer knows that an adder will add two numbers together. The designer does not need to know how the adder has been implemented internally.

Consider a journey by taxi: you need to know where you want to go and provide some money. You do not care how to drive the vehicle; only the driver needs to know that. He also needs to know where to go for petrol, repairs etc. and (hopefully) where your destination is. Only a mechanic needs to know how to repair the car. It's up to the oil company to provide the fuel.

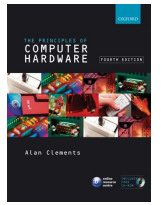
Of course it could be an electric car; then many of the details would be different, but you don't need to care as your interface is still the same.

You can view abstraction being used at the various levels of the design hierarchy. We will look at both hierarchy and abstraction later.

This course is about the engineering that goes to make up a computer. It does not go into details of the underlying electronics – there are very few references to anything electrical here – but works from logic functions implemented with simple logic gates up to complete computers.

This course covers the details of the systems that underlie the instruction set which the compiler (and sometimes the programmer) see. There will be enough information in this course to allow you to build a complete, if simple, computer from 'scratch'.

The laboratories support the earlier parts of the course, exercising the methods introduced in the lectures and leading to operational logic.



Textbooks

We don't recommend you buy any book in particular, as everything you need to know should be in the notes. If, however, you feel that a textbook would be useful then "Principles of Computer Hardware" by Clements (4th Edition) is a good book to purchase, as it contains a lot of useful information outside the limited information provided by this course.

Quiz-time

Q. What does a computer manipulate?

A. Images
B. Text
C. Numbers
D. Little men with pointy hats

Notes:

Quiz-time

Q. What are the basic building blocks of a processor?

A. Lego bricks
B. Transistors
C. Memory
D. Pixie dust

Notes:

Quiz-time

Q. Moore's Law relating to processors states that …?

A. Fewer transistors are required as processors evolve
B. The number of transistors doubles every year
C. The number of transistors doubles every two years
D. The number of transistors doubles every five years

Notes:

Quiz-time

Q. How have we increased the performance of modern processors?

A. Reduced the size of the transistors
B. Used more transistors
C. Made processors bigger
D. All of the above

Notes:

Quiz-time

Q. How many transistors are there in the Intel i7 core chip?

A. Between 1,000,000 and 100,000,000
B. Between 100,000,000 and 500,000,000
C. Between 500,000,000 and 1,000,000,000
D. More than 1,000,000,000

Even more

Notes:

MANCHESTER 1824

The University of Manchester

What is in a computer?

What does a computer actually do?

- It manipulates information - data

How is information represented?

- The data is in a binary form, strings of '1's and '0's

How is the information manipulated?

- The data is manipulated by logic gates that perform the required logic function – AND, OR etc
- Logic gates are implemented as a wired collection of transistors
- Collections of logic gates form more complex functions – ALU etc
- Data can be stored to memory where the value is preserved, and retrieved later

What does the information represent?

- Many things …

Notes:

What is a computer?

A computer is an electronic device that manipulates information ("data") that can be stored to memory and retrieved at a later instance in time for further manipulation. We can think of hardware as being the physical object that is the computer, which contains the motherboard, processor, memory, power supply etc, and software is a series of instructions that runs on the hardware, telling it what to do. The instructions, which are simply represented by strings of binary numbers ('1's and '0's), are decoded within the processor to determine what action to perform. Sequences of instructions are strung together to implement complex programmes that perform a variety of tasks.

Processors have evolved significantly over the last 3 decades with an increase in the number of transistors on the chip – Moore's Law. Transistors, which are semiconductor devices, which in the simplest form can be used to switch electronic signals, form the basic building blocks of the processor. We won't be looking at the operation of the transistor in this course, but if you are interested plenty of information can be found on the internet. Transistors can be connected together to form a basic logic gate that can be used to manipulate binary signals. We will look at logic gates later in the notes, and is the lowest layer of "abstraction" we will be interested in (we don't actually care what is inside it!). An example of a basic logic gate is the AND gate, which in its simplest form has two inputs and one output. When both inputs are logic high, i.e. true, or '1', then the output goes high, otherwise the output stays low, i.e. false, or '0'. We will look at the terms true, false, 0 and 1, later. For now it is important to realise that data can be represented in a binary form, that is a string of '1's and '0's, that are manipulated within the processor using logic gates that are constructed from transistors. Logic gates can be connected together to form more complex logic functions as required for today's complex applications.

Why should I be interested in computer hardware?

Why not?

As a computer scientist it is important that you have a broad understanding of the subject and a knowledge of how computers are built is very important. The hardware forms the basis of a computer, without it how would the software run, what would do the computation. Hence, you could consider the hardware to be the most important part of ANY computer! (Well in my opinion anyway!) In this course you won't be building a processor from scratch, but you will get to know what is required to build a processor and in the labs you will have a go at completing the design of a simple 16-bit processor that you can run on the experimental boards in the lab.

Hopefully, you will get a lot of this course and it will entice you to learn more about hardware and the physical construction of computer systems (hopefully ☺).

MANCHESTER 1824

The University of Manchester

Introduction to Logic

In the first set of lectures we will be looking at binary logic and the electronic components that allow us to manipulate binary data.

- How do we represent signals in a computer?
- Boolean logic functions and the rules of Boolean algebra
- Logic gates – how we implement Boolean functions
- Arithmetic operations – how do computers add and subtract numbers?
- Abstraction and Hierarchy – how do we control complex designs?
- Sequential systems – time dependent behaviour in electronic systems
- … and more

There is lots information, most of which will be new and at first you may not appreciate it's importance. However, don't worry, the notes are self explanatory and most of the information is not covered in great detail (as it would if you were studying electronics for example).

Notes:

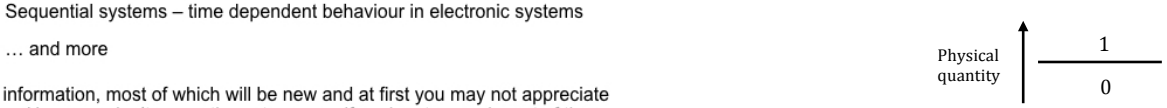
Data Representation

How do we represent information? We could take a set of symbols, such as the alphabet, and create information by constructing sentences from these symbols. We could represent quantity by taking the numerical symbols 0-9 and using them to represent how many objects we have. There are other ways of representing information, such as the use of semaphore, braille, or Morse code, as just a few examples. However, what is common in all these approaches is that information is represented by symbols, whether they are positions of flags or written characters, they are all symbols.

In the case of the digital computer information is represented using the symbols 1 and 0, which is a binary system, also stated as true or false. Such signals are discrete in value and time. Here, a signal representing this information can take up one of two values, which is unlike the continuous world in which we live in whether values can be represented in analogue form, such as temperature, distance etc, where there are no discrete levels in the signal.

Why binary?

The information that a binary signal is representing is often continuous in nature. Consequently, although we talk about digital signals have two discrete states, 0 and 1, they are in fact represented by a signal voltage that is continuous in nature. In such cases, we need to define a threshold signal, above which the signal is determined to be a 1, below which a 0.



A "**threshold**" is defined; values at one side are interpreted as one value, values at the other are a different value.

Having discrete values is useful, as noise, which is inherent in all digital systems, has little effect in a digital system. for example, in a 5V digital system (such as TTL) a signal in the range 0-0.8V is considered to be a 0, and a signal in the 2.4-5V is considered to be a 1. The input circuitry of the system will always be able to correctly distinguish the state of the signal, providing it lies within these ranges. Hence, for a 0 signal to be interpreted as a 1, we would have to add 1.6V of noise to a 0.8V (value of 0) signal, in order to detect it incorrectly as a 1.

We could use other phenomena to represent our digital signals. For example, consider a signal sent as light (e.g. on an optic fibre); it is relatively easy to determine light on/light off. It is harder to determine a signal with three states with a single quantity (off/dim/bright) because one state borders on both the others.

For example a lamp changing from "off" to "bright" must pass through the "dim" stage on its way; this can be open to misinterpretation. Hence, if we define two states, "on" and "off", they we will avoid this misinterpretation – it is a pity that it isn't that simple, but let's not worry about that!

What is voltage?

The concept of voltage can be somewhat difficult to understand in a sentence or two, so let's look at an analogy …

If you have two water reservoirs, one at the top of a hill and one a the bottom, and there is a pipe between them, water will flow from the top reservoir to the bottom reservoir due to the effects of gravity; the water in the top reservoir has a higher potential energy compared to the water in the lower reservoir, consequently water will flow down the pipe from the top reservoir to the bottom one. Voltage is the potential energy that forces an electric current to flow round a circuit from a point of higher potential to a point of lower potential – thus forcing the electrons to flow. The unit of voltage is the *volt* (V).

A '0' or a '1' in digital systems is determined by two voltage levels, typically 0V and a higher voltage that depends upon the electronic system – let's say 5V. So a signal of 0V will be interpreted as a '0', logic 0 or false, and a signal of 5V will be interpreted as '1', logic 1 or true. If the voltage is anywhere in between, then a decision has to be made whether it represents a '0' or a '1' – as above, a threshold needs to be defined. If the voltage is less than 2.5V, then it may be considered to be a '0'; otherwise it is a '1'. The technology used to implement the logic determines this operation, but you need to remember that although we are dealing with binary data – 0's and 1's – these are represented by **analogue** values, that are not always 'ideal' voltage levels!

MANCHESTER 1824

The University of Manchester

Digital Signals

A **signal** is something which carries information.

A **digital signal** is a signal that at any time can have one of a number of finite possible values, or **states** – usually 2 …

… as *opposed to* ***analogue signals*** *that can take an infinite number of possible values.*

The states of digital signals can be interpreted in a variety of ways:

- TRUE/FALSE ('T' / 'F')
- High/Low
- '1' / '0'

'1's and '0's are the more common for representing data, especially when several bits are used (binary data … example: 8-bit, 32-bit).

Notes:

Notes on Logic Values

(taken from pg 27 of Principles of Computer Hardware, Clements, 4th Edition)

1. Every signal must assume one of two discrete states; there are no other states other than 0 and 1 (although there are X and Z, although these are invalid states).
2. A signal can only exist in one state at any one time.
3. A signal that is variable can move between the two states, whereas a constant signal retains the logical value.
4. The signal level that causes an action to occur is arbitrary. If a high voltage causes an action to occur, then this is called ***active-high***. If a low voltage causes an action to occur, then this is called ***active-low***.
5. By convention a system that treats a low level as logic 0, or false, and a high level as logic 1, or true, is called ***positive logic***. This is the standard convention.
6. A signal is often given a name that indicates that action it causes to happen. For example, STOP, COUNT, RESET etc.
7. If we say asserting STOP causes some action to stop, then we are saying that STOP must go **high** to cause the action to stop. IF we say that $\overline{\mathrm{STOP}}$ is asserted, then we are saying that STOP must go **low** to cause the action to stop.

Quiz-time

Q. How many values can a single binary digit represent?

A. 1
B. 2
C. 10
D. A lot

Notes:

Quiz-time

Q. How many values 2 bits represent?

A. 1
B. 2
C. 4
D. 10

Notes:

Quiz-time

Q. How many values n bits represent?

A. 1
B. 2
C. n
D. 2^n

Notes:

Quiz-time

Q. How many bits do I need to represent 16 values?

A. 2
B. 4
C. 8
D. 16

Notes:

Quiz-time

Q. How many bits do I need to represent 17 values?

A. 4
B. 5
C. 6
D. 7

Notes:

MANCHESTER 1824

The University of Manchester

Data Representation

Computers manipulate binary data, this data can represent a number of things ...

- Bits are grouped together and used to represent other variables such as characters, numbers, instructions …

For example:

The bit pattern 01101100011101010110000101010000 (that's 6C756150 in hex)

could represent:

• an ARM instruction	LDCVSL P1, R6, [R5], #-&140
• a decimal number	1819631952
• a floating point number	1.1865×10^{27} (approximately)
• an ascii string	"Paul"

It's a question of how you interpret things! – we will revisit data and what it means later ...

Notes:

Data Representation

A collection of bits can be used to indicate a quantity in a finite set of articles. As we have already the binary representation of data using two states: 0 and 1. Then, more "quantities" can be represented using a larger number of bits to allow more data to be represented in a digital form. With N bits, 2^N values can be represented.

For example, a commonly used character representation is ASCII[1]. ASCII is really a 7-bit code (later extended to 8 bits) and can therefore be used to represent (up to) 128 different characters. This allows:

- The letters A - Z (26)
- The letters a - z (26)
- The numerals 0 - 9 (10)
- The space character (1)
- Some punctuation marks (10-ish)
- A few mathematical symbols (10-ish)
- Some *control characters* (32)

These characters are a finite set, they are not numbers – you can't, meaningfully *add* a "P" to a "="! They can be represented and manipulated in other ways.

Collections of bits are given names for convenience.

- A "byte" (almost) always refers to an 8-bit quantity (we will regard a "byte" as 8 bits).
  "Octet" is occasionally used to be totally unambiguous.
- A "word" refers to the 'natural size' in a given machine; this is a very ambiguous term.
  We will use "word" for 32-bit quantities, as used in most contemporary processors such as the ARM, Pentium, etc.

Some Java "primitive types"

These correspond to collections of bits within the machine.

- byte byte 8 bits
- char 'halfword' 16 bits
- integer word 32 bits
- long 2 words 64 bits
- float word 32 bits
- double 2 words 64 bits

Note: Java allows more different characters than ASCII; it can represent more than Latin text. A word can represent different things, depending on *interpretation*. A Java "integer" is not the same as a mathematical integer. There are 'only' 232 Java "integers"; there are infinite true integers!

The interpretation of these include:

- char an arbitrary set of characters
- integer the numbers -2147483648…-2, -1, 0, 1, 2, 3,…2147483647
- float a 'logarithmic' scale

The representation of "floating point" numbers is rather complex. If you want full details look up "IEEE Standard 754". Here it is sufficient to use an (decimal) analogy. Imagine you have a calculator with a 10 digit display (and no decimal point). You can have only 10^{10} numbers.

These numbers can be:

- integers e.g. **1234567890**
- 'scientific' e.g. **12345678** x 10^{90}

depending on the *interpretation* of the numbers.

In an n-bit number the bits are numbered i, from i=0 to i=n-1, starting at bit 0, i.e.

$$2^{n-1}\ 2^{n-2}\ 2^{n-3} \ldots 2^2\ 2^1\ 2^0$$

where each bit has a significance of 2^i, and a weight of 0 or 1 (in the case of binary).

For a 4-bit number the significance of each bit is $2^3\ 2^2\ 2^1\ 2^0$, which is 8, 4, 2, and 1. The weight determines whether each bit contributes … more later when we look at arithmetic.

[1] **A**merican **S**tandard **C**ode for the **I**nterchange of **I**nformation

Quiz-time

Q. For the AND function the output is true when:

A. All the inputs are false
B. Any one of the inputs is true
C. All of the inputs are true
D. I don't know

Notes:

Quiz-time

Q. The output of the OR function is true, 1, when:

A. All the inputs are 0
B. Any one of the inputs is 1
C. One or both inputs are 1
D. All of the inputs are 1

Notes:

Quiz-time

Q. The output of the inverter is 1 when the input is:

A. 0
B. 1
C. Undefined

Notes:

MANCHESTER 1824

The University of Manchester

Boolean Functions – AND, OR, and NOT

Logic functions have **binary inputs** and **binary outputs**, i.e. the inputs and outputs are either 0 (false) or 1 (true).

The three basic Boolean functions are AND, OR, and NOT.

Name	Input/Output	Operation	Symbol
AND	2 inputs 1 output	Output is 1 (true) if both input variables are 1	.
OR	2 input 1 output	Output is 1 (true) is either input variable is 1	+
NOT	1 input 1 output	Output is 1 (true) if the input is 0 (false) Output is 0 (true) if the input is 1 (false) i.e. the input is **inverted**	a line over the variable, i.e. $\overline{A}$

True = NOT(False), False = NOT(True), or 1 = NOT(0), 1=NOT(0)

Notes:

Boolean Functions

The standard Boolean functions (such as AND) may be expanded to encompass more than two inputs. Thus, 3-, 4-, 5- etc input AND and OR functions are perfectly possible.

Operator Precedence

- the unary NOT is applied first
- AND is higher priority than OR
- Parenthesis (brackets) may be used to specify evaluation order explicitly
- A "NOT" overline may be extended to imply bracketing, e.g.

$$\overline{A.B} + C = (\text{NOT (A AND B)) OR C}$$

Other symbols

Occasionally other symbols are encountered for Boolean operations (which you may see elsewhere), for example:

Name	Schematic	Maths	Java/Verilog
AND	.	$\wedge$	& &&
OR	+	$\vee$	| ||
NOT	¯	$\neg$	~ !
exclusive-OR	$\oplus$	$\neq$	^ !=

(The exclusive-OR function will be described shortly)

Warning! You will meet this again in COMP11120 where some of the names will be different. However, *logic is universal*, so the rules are the same!

Truth Tables

We represent logical functions using logical **truth tables** that indicate all the possible states of a logical function.

AND

Inputs		Output
F	F	F
F	T	F
T	F	F
T	T	T

or (more common)

Inputs		Output
0	0	0
0	1	0
1	0	0
1	1	1

Z=X.Y

OR

Inputs		Output
0	0	0
0	1	1
1	0	1
1	1	1

C=A+B

NOT*

*(or inversion, complement)

Input	Output
0	1
1	0

NOT(Q) = $\overline{Q}$

… truth tables for functions with n inputs have 2^n entries.

It is possible to draw a truth table to define ***any*** function.

Notes:

Truth Table

What is a truth table? It relates the output value to every possible combination of the inputs in tabular form. A circuit with n inputs has 2^n entries in the truth table. The order of the inputs are listed is not important. However, by convention, the order corresponds to the natural binary sequence, or count.

In the slide two truth tables for the AND function are provided, they are equivalent. In one, we have chosen to represent TRUE, 'T', and FALSE, 'F', values, whereas in the other we have used the more common approach where TRUE = '1' and FALSE = '0'. We will be using '1' and '0' throughout this course.

Basic Rules of Boolean Algebra

1. Zero and Unit Rules
$$A \bullet 1 = A \qquad A \bullet 0 = 0$$
$$A + 1 = 1 \qquad A + 0 = A$$
2. Complement Relations
$$A \bullet \overline{A} = 0 \qquad A + \overline{A} = 1$$
$$\overline{(\overline{A})} = A$$
3. Idempotence
$$A \bullet A = A \qquad A + A = A$$
4. Commutative Laws
$$A \bullet B = B \bullet A \qquad A + B = B + A$$
5. Absorption Rules
$$A + A \bullet B = A \text{ (note: } \bullet \text{ has precedence over +)}$$
$$A \bullet (A + B) = A$$
$$A + \overline{A} \bullet B = A + B$$
6. Distributive Laws
$$A \bullet (B + C) = A \bullet B + A \bullet C$$
$$A + B \bullet C = (A + B) \bullet (A + C)$$
7. Associative Laws
$$A + B + C = (A + B) + C = A + (B + C)$$
$$A \bullet B \bullet C = (A \bullet B) \bullet C = A \bullet (B \bullet C)$$
8. De Morgan's Theorem
$$\overline{A + B + C} = \overline{A} \bullet \overline{B} \bullet \overline{C}$$
$$\overline{A \bullet B \bullet C \bullet \ldots} = \overline{A} + \overline{B} + \overline{C} + \ldots$$

Remember you can express the complement of a variable, $\overline{x}$, as x'.

George Boole (1815-1864)

- 1815: born, Lincoln, 2nd November, son of a poor shoemaker
- Studied Latin as a schoolboy
- 1832: Assistant junior schoolteacher
- 1835: opened his own school; began to study mathematics
- Began publishing in the Cambridge Mathematical Journal
- 1849: appointed to the chair of mathematics at Queen's College, Cork
- 1854: published "An Investigation into the Laws of Thought", on which are founded the Mathematical Theories of Logic and Probabilities ('Boolean Algebra')
- 1857: Elected FRS (Fellow of the Royal society)
- 1864: Died 8 December in Ballintemple, County Cork (pneumonia – due to lecturing in wet clothes).

NAND, NOR

Two more Boolean functions are worthy of their own names …

- **NAND** : which is AND followed by NOT (AND with the result inverted)
- **NOR** : which is OR followed by NOT (Or with the result inverted)

NAND

Inputs		AND	**NAND**
0	0	0	**1**
0	1	0	**1**
1	0	0	**1**
1	1	1	**0**

NOR

Inputs		OR	**NOR**
0	0	0	**1**
0	1	1	**0**
1	0	1	**0**
1	1	1	**0**

… NAND and NOR gates are often used in logic circuit design

Notes:

NAND and NOR

NAND is the output from the AND inverted:

NAND function of inputs A and B $Q = \overline{A \bullet B}$

This is interpreted as A AND B followed by inversion of the result (hence the bar above the AND function).

NOR is the output form the OR inverted:

NOR function of inputs A and B $Q = \overline{A + B}$

This is interpreted as A OR B followed by inversion of the result (hence the bar above the OR function).

Jargon

Consider the functions AND and OR again:

AND Q=A.B

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR Q=A+B

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

Note the similarity in both the notation and function to the arithmetic operations of multiplication and addition?

For this reason:

- an **AND** is sometimes known as a Boolean **product**, and
- an **OR** is known as a Boolean **sum**.
- **NOT** may be referred to as a **complement**.[1]

[1] Not the spelling: "complement" means "that which completes", whereas "compliment" is flattery.

Exclusive-OR - XOR

Another useful Boolean logic function is the Exclusive-OR gate (EXOR, XOR)

The two-input OR function is true if one or more of its inputs are true – the XOR is similar, but it excludes the case when both inputs are true.

XOR

Inputs		XOR
0	0	0
0	1	1
1	0	1
1	1	0

- the XOR operation is often indicated by the symbol ⊕
- it may be constructed from other (AND/OR/NOT) functions

$$S=P \oplus Q$$

Notes:

Exclusive-OR (XOR)

Exclusive-OR:

$$Q = A \oplus B$$

and is often referred to as:

- a not-equivalence function – look at the truth table – can you see why?
- N odd parity function.

Parity is a count of the number of input variables that are 'true', i.e. '1'. If zero or two inputs are true, then the input set has even parity (because 0 and 2 are even numbers). Only one input being true gives odd parity. Again, check the truth table.

Strictly speaking 'exclusive-OR' is a function of two inputs only. Larger ('wider') functions are possible but *should* be referred to only as *parity* functions. In practice this is largely ignored!

Exclusive-NOR (XNOR)

There is in fact *another* function that merits mention – the **exclusive-NOR** (XNOR/equivalence/ even parity), which is simply the inverse of the result of the XOR function, i.e.

$$Q = \overline{A \oplus B}$$

(You may find mathematicians using "⇔" for XNOR).

Truth Tables for Arbitrary Logic Functions

We have shown how we can depict the relationship between the input signals and output signal using the truth table. We could do the same for any arbitrary logic function. Consider the function

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

The truth table can be derived as follows:

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

Quiz-time

Q. What function does the truth table represent?

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

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

Notes:

Quiz-time

Q. What Boolean function does the truth table represent?

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

A	B	Z
0	0	0
0	1	1
1	0	0
1	1	0

Notes:

Quiz-time

Q. What Boolean function does the truth table represent?

A	B	Z
0	0	1
0	1	0
1	0	1
1	1	1

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

Notes:

Quiz-time

Q. What Boolean function does the truth table represent?

A	B	Z
0	0	0
0	1	1
1	0	0
1	1	1

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

Notes:

Quiz-time

Q. What Boolean function does the truth table represent?

A. $F=A.B$

B. $F=(A+B)\text{-}\bar{A}.B$

C. $F=A$

Inputs		Output
A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

Notes:

MANCHESTER 1824

The University of Manchester

De Morgan's Theorem

De Morgan's theorem (named after Augustus De Morgan 1806-1871) is a very useful relationship, which allows us to recast Boolean expressions is different forms.

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

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

In words …

"The complement (inversion) of a sum equals the the product of the complements; the complement of a product equals the sum of the complements."

No(fish OR chips) =No(fish) AND No(chips)

No(fish AND chips) =No(fish) OR No(chips)

Notes:

What use is De Morgan's theorem?

De Morgan's theorem is a very powerful tool since the relationships allow an AND function to be implemented using OR gates and inverters, and vice versa.

Example, write an alternative Boolean expression for the following expression:

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

1. Replace + by . and complement the product terms

$$F = \overline{A \bullet B} \bullet \overline{C \bullet D} \bullet \overline{A \bullet D}$$

2. Replace . with + and complement the variables in the product terms

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

it allows you to represent Booleam expressions in different forms, which can be useful when it comes to implementing them in hardware.

What else?

De Morgan's theorem allows circuits to expressed in NAND-only or NOR-only forms.

Why is this an advantage?

Because the electrical implementation (using logic gates – more later) of the NAND function (the NAND gate) is both faster and cheaper that the corresponding AND function (AND gate) .

In practice an AND functions will be implemented as a NAND with it's output inverted, i.e. NOT(NAND) = AND.

How would you implement the function $F = A + B + C$ using just NAND gates?

Easy:

1. Double negation:

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

(this has no effect on the function!)

2. But if you then apply De Morgan' theorem to one of the inversions:

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

and the OR function has been converted into a NAND function, i.e. the function F is the NAND of the inverted inputs A, B, and C.

Quiz-time

Q. In the following expression

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

the output A is …

A. the OR of B and C
B. the NOT of B and C
C. the NOR of B and C
D. the NAND of B and C

Notes:

Quiz-time

Q. According to DeMorgans the following function

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

is equivalent to …

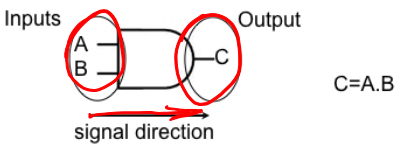
A. $\overline{R}.S$
B. $\overline{R+\overline{S}}$
C. $\overline{\overline{R}}.S$
D. $\overline{R.\overline{S}}$

Notes:

Logic Gates

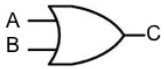
Logic expressions are expressed algebraically. However, when building electronic digital circuits we need to express these functions symbolically. In circuits, the logic functions are represented in **schematics** using **logic gate symbols**.

AND gate



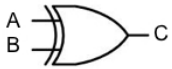
C=A.B

OR gate



C=A+B

XOR gate



C=A⊕B

Notes:

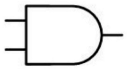
Logic Gates

The digital computer simply consists of an interconnected collection of logic functions – gates – that implement a specific function. In the same way, the brain is a collection of neurons!

The AND gate has a minimum of two inputs and performs the Boolean operation "AND" on these two inputs to generate an output. Similarly, the OR gate performs the Boolean 'OR' operation on two or more inputs. The XOR performs the Boolean XOR operation.

The schematic diagram, or simply 'a schematic', is a diagrammatical representation of a circuit consisting of one or more logical functions as gates. Each of the basic logic functions can be represented by a 'symbol' that is used to illustrate that particular Boolean operation in a schematic.

For example, for an AND operation we would use the symbol:



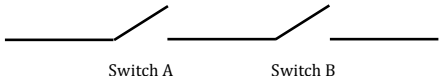
etc.

By connecting multiple gates together in a schematic it is possible to produce a diagrammatical representations of much more complex logical functions.

The word '**gate**' conveys the idea of a two state device – open (to allow things through) or shut (to block). You can visualise gates as being composed of switches (although this is only an analogy, this is more "tri-state" operation).

The AND gate (F=A.B):

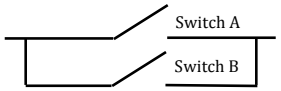
The inputs A & B form the control to two switches.



Only when both switches are closed (i.e. A=1 AND B=1) is the input connected to the output.

The OR gate:

The inputs A & B form the control to two switches.



When either of the switches are closed (i.e. A=1 OR B=1) is the input connected to the output.

This is a very simplistic view of the operation of gate, and by no way represents what really happens when a gate is implemented in hardware. However, it serves to aid in the understanding of how they work.

What is inside a gate?

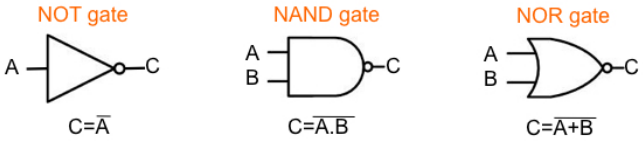
The gate is an abstract device. If we looked inside a gate we would find it is made up of a number of transistors connected together to implement the required logical function.

In these lectures we are not concerned with the operation of transistors and how they can be connected togetherto form a logical function. We are happy to simply take the gates we will look at and implement more complex logical functions – this is abstraction – we will revisit this concept again later.

In the lab you will be downloading your designs (produced from either schematics or generated using hardware description languages) onto field programmable gate arrays (FPGAs) that are programmable electronic 'chips' that can be used to produce customized hardware. More about this later …

Logic Gates

Inversion is represented by a 'bubble':



Gates with more than two inputs are possible, such as 3- and 4-input AND gates.

3-input AND gate

The output is true, '1', when ALL 3 inputs are true, otherwise the output is false, '0'.

Notes:

Logic Gates

The NOR, NAND and NOR are familiar Boolean functions that, again, using symbol representations we can produce schematic representations of complex logical functions that we can easily implement in hardware.

The NAND gate is simply the AND operation with the output inverted. Similarly, the NOR is the OR operation with the output inverted and the XNOR is the XOR operation with the output inverted.

It also possible to invert one or more of the inputs to a conventional logic gate to implement a different function; this is represented by a 'bubble' at the appropriate input..

Schematics Phraseology

Gates are **components** within the schematic, depicted by characteristic **symbols**. They are interconnected by 'wires' or **networks** (sometimes abbreviated to '**nets**'). A net has one gate output connection but may be connected to several gate inputs. (*These are terms you will get used to seeing in the lab*.)

Rules

- One net – one output
- All inputs connected ("driven")

Each net represents a Boolean variable. These may be treated in the same way as software variables. Although it is not essential it is often a good idea to give each network a (meaningful) **name** (or **label**). By convention a name applies across the whole schematic, so two fragments of a net, both with the **same name** are **implicitly connected**, even it the drawing does not show them joined.

Sometimes a net is required that is 'always 1' or 'always 0'. These are provided for by specific symbols. In CMOS technology (currently almost universal) they are called Vdd and Vss (respectively) although the older names are still commonly used.

Voltage	Logic	Symbol	Name	Name
H	1		Vcc	Vdd
L	0		GND	Vss

A gate is an abstraction. It has a number of inputs, a single output, and a function that is easy to describe and understand.

In this course **gates** are the **lowest abstraction level** explored (more later on **abstraction**). We assume certain gates are available and do not worry about how they are implemented.

Quiz-time

Q. The symbol shown represents which Boolean function?

A. OR
B. AND
C. NOR
D. NAND

$\overline{x} + \overline{y} = \overline{xy}$

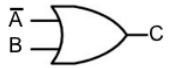
Notes:

MANCHESTER 1824

The University of Manchester

Inverted Inputs

If we have a logic gate with an inverted input, what does this mean in practice?



The inverted input here, $\overline{A}$, means that the input to the gate is true, '1', when the input A is false, '0'. Thus, the function of the logic gate with respect to the inputs A, and B has changed. The modified truth table for this function would be

OR these

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

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

This example shows that a '0' on an input signal can cause an "action" to occur, it isn't just a signal being 1 that cause actions.

Notes:

Inverted Inputs

We have derived inverted inputs on a schematic (having a bar over the variable) as a result of taking a truth table and extracting logical expressions from it, for example, in the case of the NOR function:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

we arrived at a Boolean expression by ORing the conditions where the output is true, in this case:

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

this says, when $A=0$ AND $B=0$ the output is true, since $\overline{A}=1$ when $A=0$ and $\overline{B}=1$ when $B=0$.

Quiz-time

Q. In the last lecture we produced a logical equation for the XOR function. Which of the following is the same function?

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

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

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

Notes:

Quiz-time

Q. For the Boolean expression:

$$Z=\bar{A}.B$$

the output Z is true ('1') when

A. A=0, B=0
B. A=0, B=1
C. A=1, B=0
D. A=1, B=1

Notes:

Quiz-time

Q. For the Boolean expression:

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

the output Z is not true ('0') when

A. A=0, B=0
B. A=0, B=1
C. A=1, B=0
D. A=1, B=1

All b.

Notes:

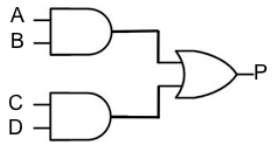
More complex logic functions …

The basic gates (AND, OR, NOT etc.) on their own offer limited functionality. However, by connecting different gates together you can create a circuit to perform a (arbitrary and more complex) logic function.

A simple example of implementing a more complex logic function:

P=A.B+C.D

- AND A and B
- AND C and D
- OR the result



So more complex functions can be made by connecting many logic gates together!

Notes:

Implementation of Boolean Logical Expressions using Gates

A few logic gates can be connected together to create a circuit to implement a simple logic function – the schematic.

Logic expressions are often represented in the form where a number of product terms (ANDs) are summed together (ORed), this is referred to as the sum-of-products form. An alternative representation is the product (ANDed) of sum functions (ORs), which is referred to as the product-of-sums form. The relationship between the two being De Morgan's theorem!

A sum-of-product function can be implemented using AND and OR gates, as shown in the slide!

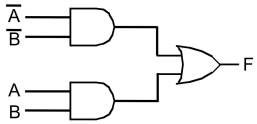
You can implement functions from the truth table by first expressing the function in a sum-of-products form. For example, implement the function:

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

So the logic function can be extracted by observing what input combinations cause the output to be high, '1'. So there are two cases, when A=0 **and** B=0, **or** when A=1 **and** B=1, so the logic expression is:

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

which can be easily implemented:



MANCHESTER 1824

The University of Manchester

Sum of Products form

We have seen a few logical expressions that look like this:

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

This is known as the sum of products form, i.e. it is a sum '+' of product '.' terms.

You will often see logical expressions in this form.

An alternative expression is the product of sums form, where we have a product '.' of sum '+' terms, i.e.

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

Product of sums expressions are often used to represent the inverse of a function, as inverting a function and applying De Morgan's theorem will often result with an expression in a product of sums form.

Notes:

Sum of Products/Product of Sums form

In the example given we have a logical expression that consists of a number of AND terms being ORed together, this common representation is often referred to as the **sum of products** expression. Expressing a logical expression in this form allows us to visual directly how the expression may be implemented directly using AND and OR gates (or NAND gates … the following quiz questions my help understand this relationship).

An alternative expression is the **product of sums** form whereby the logical function is expressed in terms of a number of OR functions that are ANDed together. The inverse of a logical expression can often be represented in a product of sums form. For example, consider the following sum of products expression:

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

The inverse of Z is

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

which using DeMorgan's theorem gives:

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

Quiz-time

Q. Identify the sum of products expression.

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

Notes:

Quiz-time

Q. For the logical expression

$$Q=(A.\bar{B})+(A.B)$$

identify the inverse function:

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

Notes:

$Q = (A \times \bar{B}) + (A \times B)$

$Q' = \overline{(A \times \bar{B}) + (A \times B)}$

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

$= (\bar{A} \times B) + (\bar{A} + \bar{B})$

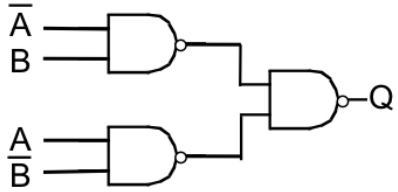
Quiz-time

Q. Does the circuit shown represent the XOR function?

A. No

B. Yes

C. Can't think …



Notes:

Quiz-time

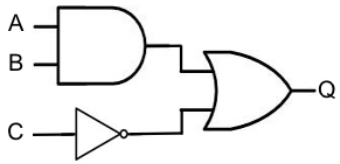
Q. What logical function does the circuit represent?

A. $A+B+C$

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

C. $(A.B)+C$

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

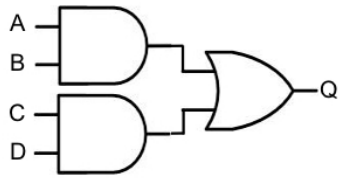


Notes:

Quiz-time

Q. What logical function does the circuit represent?

A. A+B+C+D
B. A.B.C.D
C. (A+B).(C+D)
D. (A.B)+(C.D)

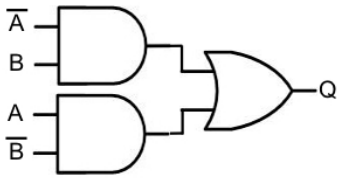


Notes:

Quiz-time

Q. What logical function does the circuit represent?

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



Notes:

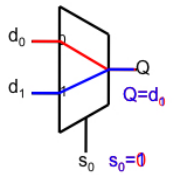
MANCHESTER 1824

The University of Manchester

Multiplexers

Another common device is the multiplexer, or mux. The mux is a switching device, that selects binary information from one of its many input lines and directs it to a single output line.

The 2:1 mux has the circuit symbol :

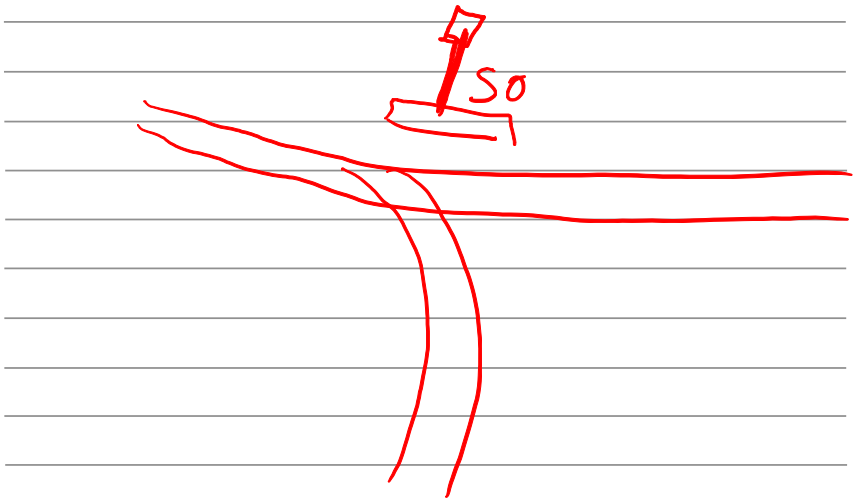


The status of the control input, s_0 determines which of the two inputs is connected to the output.

Thus, the mux acts like an electronic switch that selects which source is connected to the output.

A mux can have any number of inputs, but must have enough control inputs to be able to select each input (a 4:1 mux has 4 inputs and 2 control lines, 32:1 mux has 32 inputs and 5 control lines).

Notes:



Multiplexers

The multiplexer is an electronic switch that has at least one control line that is used to select which one of a number of input lines is connected directly to the output, i.e. it is a data selector! The control input selects which of the data inputs is connected to the output. Data will flow through from the selected input to the output until the control input selects a different input. The multiplexer is a combinatorial device.

For example:

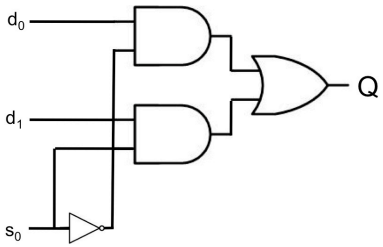
- The 2:1 mux has one control input that select one of the 2 inputs.
- The 4:1 mux has 2 control inputs that select one of the 4 inputs.
- The 8:1 mux has 3 control inputs that selects one of the 8 inputs.
- The 16:1 mux has 4 control inputs to select one of the 16 inputs.
- The 2^n:1 mux has n conrol inputs to select on of the 2^n inputs.

The 2:1 mux

The truth table for the 2:1 mux is

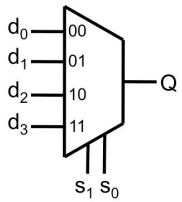
s_0	Q
0	d_0
1	d_1

Inside the 2:1 mux we will find a collection basic gates that are connected in order to perform the required switching operation. A schematic of a 2:1 is:



The 4:1 mux

The 4:1 mux is represented by the symbol:



The truth table for the 4:1 mux is:

s_1	s_0	Q
0	0	d_0
0	1	d_1
1	0	d_2
1	1	d_3

Quiz-time

How many control lines will a 4:1 mux require?

A. 1
B. 2
C. 3
D. 4

Notes:

Quiz-time

How many control lines will a 16:1 mux require?

A. 1
B. 2
C. 3
D. 4

Notes:

Quiz-time

How many control lines will a 2^n:1 mux require?

A. n
B. 2n
C. 2^n

Notes:

MANCHESTER 1824

The University of Manchester

Number Representation

A decimal number, such as **1,529** represents a quantity that is equal to:

1 thousand + **5** hundreds + **2** tens + **9** units

or mathematically

$$\mathbf{1}\times1000 + \mathbf{5}\times100 + \mathbf{2}\times10 + \mathbf{9}\times1 = \mathbf{1}\times10^3 + \mathbf{5}\times10^2 + \mathbf{2}\times10^1 + \mathbf{9}\times10^0$$

so at stage the coefficients take the values 0-9 which are multiplied by the appropriate power of 10.

The decimal number system has a radix, or base, of 10, since it uses 10 digits and the coefficients are multiplied by powers of 10.

The binary number system has a base of 2, it has coefficients of 0 and 1, and these are multiplied by powers of 2.

So

$$01011010$$
$$= \mathbf{0}\times2^7+\mathbf{1}\times2^6+\mathbf{0}\times2^5+\mathbf{1}\times2^4+\mathbf{1}\times2^3+\mathbf{0}\times2^2+\mathbf{1}\times2^1+\mathbf{0}\times2^0$$
$$= \mathbf{0}\times128+\mathbf{1}\times64+\mathbf{0}\times32+\mathbf{1}\times16+\mathbf{1}\times8+\mathbf{0}\times4+\mathbf{1}\times2+\mathbf{0}\times1$$
$$= 0+64+0+16+8+0+2+0$$
$$= \mathbf{90}$$

Simples ..

→ decimal system

→ binary

Version 2016 COMP12111:Overview & Introduction to Logic

Notes:

binary = gud 😊

Binary Number Representation

Digital circuits are used to perform arithmetic operations, they are built using transistors that are characterised by two states, 0 and 1, or "on" and "off", so we can only represent two values 0 and 1.

Why do we naturally work in decimal (or base-10)? Who knows, maybe because we have 10 fingers (or 8 fingers and two thumbs for the puritans).

Base-10 numbers are represented as a list of numbers (of values between 0 and 9), each of a different weight that is a power of 10. For example: the decimal number 725 is

7 – with weight 10^2 – the number of hundreds.
2 – with weight 10^1 – the number of tens.
5 – with weight 10^0 – the number of ones.

So 725 represents $7\times10^2+2\times10^1+5\times10^0=700+20+5=725$.

So in the case of two states (not 10 as in decimal) , which is base-2, if we want to represent a number greater than 1, then we have to introduce another number with a different weight, in this case 2^1 (like if we needed to represent a number greater than 9 in base-10, we introduced 10^1).

So in binary (base-2) the digits can only take value of 0 and 1 and each bit has a weight associated with it that is a power of 2. For example: the binary number 0101 is

0 – with weight 2^3 (from msb to lsb – left to right)
1 – with weight 2^2
0 – with weight 2^1
1 – with weight 2^0

So 0101 represents $0\times2^3+1\times2^2+0\times2^1+1\times2^0=0+4+0+1=5$ (in decimal).

When writing numbers in a particular base, then the base can be indicated with a subscript, i.e. $0101_2=5_{10}$.

Converting from Decimal to Binary

One way to convert from decimal to binary is by division by 2 and remembering the remainder. For example, represent 13 in binary:

2|13		
6	remainder	1 (lsb)
3	remainder	0
1	remainder	1
0	remainder	1 (msb)

So $13_{10}=1101_2$.

Converting from Binary to Decimal

Remember that each digit has a weight associated with it that is a power of 2. For example, represent 0110101_2 in decimal:

$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$0\ \ 1\ \ 1\ \ 0\ 1\ \ 0\ \ 1 = 0\times2^6+1\times2^5+1\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0=0+32+16+0+4+0+1=53_{10}$

Other bases of interest

Two other bases you will come across are hexadecimal (hex, or base-16) - 0 – F, and octal (base-8) – 0-7.

MANCHESTER 1824

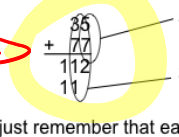
The University of Manchester

Arithmetic Operations

Addition

Using decimal (base-10) notation, we can perform addition very easily …

basic sh*t

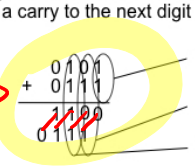


5+7=12, which is 2 carry 1

3+7+1=11, which is 1 carry 1

In binary addition is similar, just remember that each each digit can have a value of 0 and 1, and the sum of 2 generates a carry to the next digit …

not so basic sh*t



1+1=2 – sum of 0 carry of 1

0+1+1=2 – sum of 0 carry of 1

1+1+1=3 – sum of 1 carry of 1 (2+1)

Addition in other bases, such as hexadecimal, follow the same principles.

Notes:

Addition of Binary Numbers

Two *n*-bit binary numbers can be added by treating each pair of individual bits in turn (from the lsbs to the msbs) by summing the two bits and taking into account in the sum, the carry from the preceeding two bits.

Thus, we can deduce a set of sum, S, and carry, c_o, values for all possible combination of input bits, A_i and B_i, and carry in, c_i:

A_i	B_i	c_i	S	c_o
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

It is important that you allow the 'carries' to propagate to the next most significant bit. The carry into the lsb is clearly 0.

Addition is subject to a limit; the result must be 'modulo 2^N' where N is the number of bits available. (The Java "+" operator is subject to this limit.)

This is not normally a problem, as long as N is "big". Pretend it works!

We will see later that addition of two bits can be performed with a few simple logic gates.

53 in binary

64 32 16 8 4 2 1

does 53->64 1

does

MANCHESTER 1824

The University of Manchester

Arithmetic Operations

Subtraction

A – B = A + (–B), i.e. we perform subtraction by "adding" a negative number

… but how do you represent a negative number in binary?

The most common method of representing negative numbers is using **2's complement** representation:

-8 = 1 0 0 0	0 = 0 0 0 0
-7 = 1 0 0 1	1 = 0 0 0 1
-6 = 1 0 1 0	2 = 0 0 1 0
-5 = 1 0 1 1	3 = 0 0 1 1
-4 = 1 1 0 0	4 = 0 1 0 0
-3 = 1 1 0 1	5 = 0 1 0 1
-2 = 1 1 1 0	6 = 0 1 1 0
-1 = 1 1 1 1	7 = 0 1 1 1

Positive numbers remain the same, but the msb is the sign bit, 0 indicates a positive number

Numbers represented in this way can be added (ignore any carry which falls off the end)

```
  0 1 0 1      +5
+ 1 1 1 0    + -2
-------      ----
1 0 0 1 1      +3
```


Notes:

2's complement notation - Subtraction

2's complement representation of binary numbers allows the representation of both positive and negative values. However, note:

- the representation of POSITIVE numbers remains UNCHANGED
- Less positive values are available as half the positive integer values are lost to allow us to represent the negative values.
- The most significant bit (i.e. the leftmost bit) indicates the sign
  - this is referred to as the **sign bit**
  - it is bit 3 in the slide, in the case of a 32-bit value it is bit 31 (as in the ARM processor) – the lsb is always bit 0
  - **'0'** indicates that the number is **positive** number; **'1'** indicates a **negative** number
- Arithmetic is modulo 2^N; carries off the left hand side are ignored

To find the two's complement of a number:

- First: **invert** all the bits (0 ⇔ 1)
- Then: **add 1** to the least significant bit (propagating carry as needed)

You (should) already know how to implement both these operations.

An alternative approach to finding the 2's complement representation of a negative number that doesn't require a sum is to take the positive representation and copy each bit starting from the lsb until the first '1' is encountered, from then on invert all the bits, i.e. 0 ⇒ 1 and 1 ⇒ 0. For example -6 using 4-bits … +6 = 0110, which gives -6 = 1010. To check the result do the same conversion again.

Addition and subtraction of 2's complement numbers

Addition and subtraction of 2's complement representation is exactly the same of for conventional binary representation. Subtraction is often performed by addition of the negative representation of the number.

The maths then simply 'works' – like magic. For example: perform the sum 15-6 in binary.

In fact we perform 15+(-6)
$15_{10} = 01111_2$
$6_{10}=00110_2$. Therefore, -6_{10} is 00110 -> invert bits -> 11001 - > add 1 -> 11010

Perform the sum:

```
  0 1 1 1 1    15
+ 1 1 0 1 0   (-6)
-------------------
(1) 0 1 0 0 1    9
```

ignore the carry out (in brackets), so the result is 01001_2, which is 9_{10}.

Quiz-time

Q. In signed binary representation is the value 0111_2

A. Positive
B. Negative
C. Both

Notes:

Quiz-time

Q. What is the value of 1011_2 in signed representation?

A. 11
B. -4
C. -11
D. -5

Notes:

Quiz-time

Q. What is 6 in 4-bit signed representation?

A. 0110
B. 1001
C. 1010
D. It can't be represented

Notes:

Quiz-time

Q. What is -2 in 4-bit signed representation?

A. 1010
B. 0100
C. 1110
D. It can't be represented

Notes:

Quiz-time

Q. What is -9 in 4-bit signed representation?

A. 1000
B. 0111
C. 1111
D. It can't be represented

Notes:

Quiz-time

Q. Are the numbers 1100_2 and 11111100_2 the same?

A. Yes
B. No
C. My head hurts
D. What are you talking about?

Notes:

MANCHESTER 1824

The University of Manchester

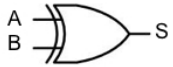
Implementing binary adders

So we know how binary arithmetic works, how do we implement a circuit to perform it?

The basic building block is the half adder, which takes two inputs and generates a sum and a carry, from which we can derive a truth table:

The functions that generate the two outputs can be easily identified:

- The sum, S, is the XOR of the inputs A, B



- The carry out, C_o, is the AND of the inputs A, B

However, we still can not add binary numbers, because we need to take into account a carry in, in the addition.

Notes:

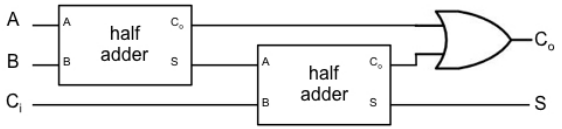
The half adder

The half adder takes two input bits (the augend and addend), and adds them together to generate a sum and a carry. In the case of our numerical examples of binary addition, the half adder can be used to sum the two corresponding bits from the augend and addend, when the carry in is 0. The disadvantage here is that there could be a carry in of 1 from the previous bit calculation, which the half adder will not take into account in its calculation. Consequently, in order to perform binary arithmetic we need to take into account whether there is a carry in to each bit – for this we need the **full adder**.

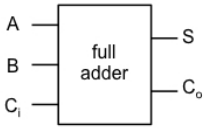
The full adder

The full adder is a circuit that takes two binary values, plus an input carry, and produces a sum and a carry out. Such a circuit is called the full adder.

We could produce a design by analysing the truth table of the full adder. However, using hierarchy we can implement a full adder using 2 half adders:



which we can abstract to its own symbol …



Which we can reuse as much as we want in future designs! (hierarchy … more later)

Notes:

The full adder

In the example full-adder shown we have jumped straight to the half-adder implementation. However, it may be useful to look at the truth table for the full-adder:

Inputs			Outputs	
A	B	C_i	S	C_o
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

The logical expressions for the two functions being implemented, S and C_o, are

$$S=\bar{A}.\bar{B}.C_i+\bar{A}.B.\bar{C}_i+A.\bar{B}.\bar{C}_i+A.B.C_i=(A\oplus B)\oplus C_i$$

and

$$C_o=\bar{A}.B.C_i+A.\bar{B}.C_i+A.B.\bar{C}_i+A.B.C_i=A.B+B.C_i+A.C_i$$

(I'll let you figure out the logic simplification.)

MANCHESTER 1824

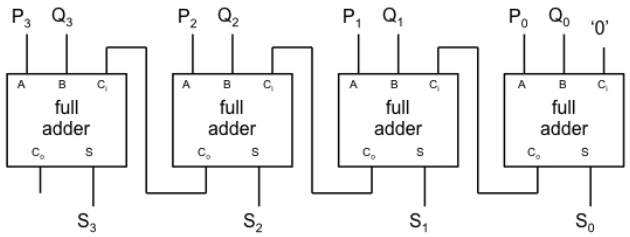
The University of Manchester

Multi-bit adders

The full adder is a 1-bit adder, to do anything meaningful we need to be able to add many bits. This can be performed using a parallel adder, where the carry out from one bit, is fed to the carry in of the next.

Example 4-bit adder

Adding two 4-bit numbers, P and Q, to generate a 4-bit sum, S



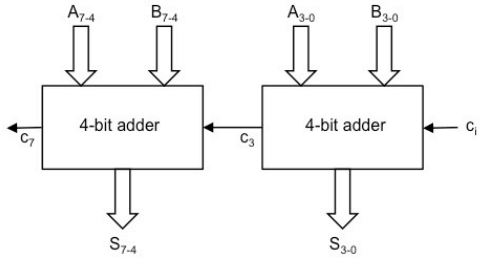
This design can be easily extended to 8-bits, 16-bits, 32-bits, 64-bits …

Notes:

The parallel adder

The parallel adder shown operates by adding the two 4-bit numbers in parallel. However, as each bit relies on the carry from the previous bit (i.e. the calculation of S_1 depends on whether there is a carry from S_0) then each bit in turn must wait until the previous bit addition is complete. So all carries must be generated before a valid sum is seen on the output.

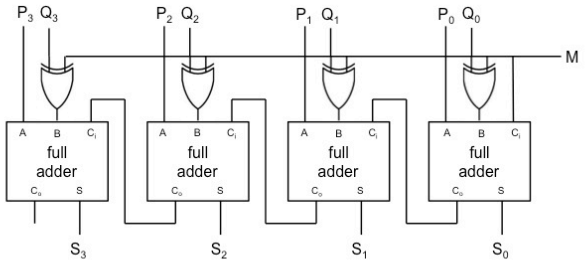
The 4-bit adder is a typical standard library component. Sums involving larger number of bits can be implemented by chaining 4-bit adders together, where the carry out form the carry in for the next 4-bit block, as shown below.



One disadvantage of the parallel adder is the time it takes to complete a sum. The maximum delay will be observed when a carry has to propagate through all the full adders in the design, this is why the design is often referred to as the **ripple carry adder**. The delay for each carry to propagate is because of the number of gates in the full adder. There are alternative designs that suffer less delay, such as the **carry look-ahead adder**, however, this comes at the expense of extra logic gates in the circuit.

The subtractor

how do we perform subtraction using the parallel adder? Remember, subtraction is addition of a binary number, so the actual addition process, and the design of the adder is exactly the same. However, the operand being subtraction needs to converted to a negative, 2's complement form. To do so requires the value inverting and '1' added. Ideally, we would want a design whereby where we can control whether we are adding or subtracting. The circuit shown below is an example of a general adder/subtractor unit.



Here, we have an extra input, M, which defines the type of operation to be performed. When M=0 we have straightforward addition, when M=1 we perform subtraction. How? M serves two purposes, it is connected to the input of a series of XOR gates, along with the input Q, when this input is 0, then the input Q passed unchanged to the adder, when M=1, then each bit of M is inverted. In addition, it is connected to the carry in of the adder, so when M=1 we add '1' to the final sum. Thus, setting M=1 will invert Q and add 1 to the result sum, i.e.

$$S=P+\bar{Q}+1=P+(-Q)\ .$$

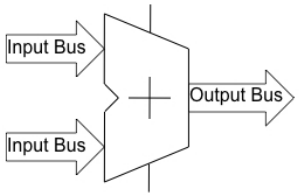
MANCHESTER 1824

The University of Manchester

Buses

You may have heard of "buses" in the context of computers, it is a much used word …

A **bus** is a collection of signals, grouped together for convenience: address bus, data bus



Largely, we will stick to using 'bus' as a designator for a collection of bits that form an individual variable.

Thus, a 32-bit computer (such as an ARM) will have a 32-bit adder but normally this will have two input buses.

In the lab, you will see these named according to two different conventions:

- busname<31:0>
- busname[31:0]

Notes:

Buses

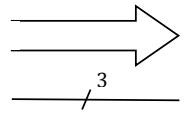
Clearly, if you want to represent a *number* (for example) you don't always want to know about all its constituent bits, at least most of the time. In the same way that a component may comprise multiple instances of the same component, a **bus** is a 'wire' that contains numerous individual wires all heading in the same direction.

As will be seen later, the word "bus" is used in many ways within a computer system. However, it always refers to **a collection of wires** with similar function. Here the simplest meaning will be taken.

In the adder example (above) there are three 4-bit buses, two input buses and an output bus. Each bus is a collection of signals that form a single entity, either an operand, or the result. Treating such a set of bits as a bus is another hierarchical level; it is often easier to think of integers rather than their constituent bits. Thus, it you want to draw the schematic for a 64-bit adder (that's two 64-bit numbers producing a 64-bit result, a carry in and a carry out) you only need to draw five connections instead of 194. The benefits in time, neatness and general wear-and-tear should be obvious. Not that software has very similar levels of hierarchy in its code (procedures, functions etc) and data (arrays, records etc) structures.

Bus Depiction

There are different ways of depicting buses. In these notes we will generally use broad arrows, because it looks artistic. If just sketching it is more normal to draw a line and then slash through it, annotating this with the bus width.



In the Cadence 'Virtuoso' tool we will use in the lab, any wire will do providing it is clear to the tool what it represents. The usual way of doing this is to add a label which specifies the bus name and it width (i.e. number of bits). To make it clearer to read 'at glance', however, a 'wide' wire is usually chosen.

data<31:0> In this format the label consists of a name and the range of the subscripts, so in this figure the bus has 32 signals called :data<31>, data<30>, data<29>, … data<0>".

Bus numbering schemes

Often, for convenience, a bus will have a single name with the constituent bits simply being numbered within it (an *array*, in software terms). Thus, in the slide the output bus could be named "`sum<3:0>`". The *usual* numbering convention is to count, from 0, from the least significant bit (LSB). This means that the bit numbered 'M" has significance 2^N (i.e. $2^0 = 1$, $2^1 = 2$, …).Virtuoso uses '<' and '>' to bracket the bit number or range of a bus signal. This is slightly unusual; '[' and ']' are more common delimiters, as in Java arrays. The 'square brackets' are also used in Verilog, as will be seen later.

> It is normal to begin counting from 0 in computing: think of the four states representable with two bits {00, 01, 10, 11}; these are most usually interpreted as {0, 1, 2, 3}.

Some other uses of 'bus'

Geeky Stuff Warning: this section is a bit geeky! You probably won't recognise all the terminology when you first read it. Hopefully, more of it will make sense later.

Address Bus
A simple bus for carrying the address of the currently desired memory location from a processor to memory.

Data Bus
Complementing the address bus, this carries data to and from the memory, e.g. when the processor stores or loads values. Sometimes this is a bidirectional bus, sometimes separate buses are used or the two directions.

'Frontside bus'
A bus containing address, data and control emerging from a processor for access to the rest of the system: external memory (off processor), peripherals etc.

'Backside bus'
A specialist bus connecting a processor to local cache. Intended to increase performance in this critical area.

PCI bus
Peripheral Component Interconnect, a standardised, slowed down 'frontside' bus for plug-in extension modules. Also implies mechanical configuration.

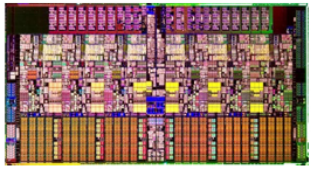
USB
Universal Serial Bus. Not a bus in the sense of a bundle of wires. USB does carry a variety of signals but multiplexes them in time, not space (hence 'serial').

MANCHESTER 1824

The University of Manchester

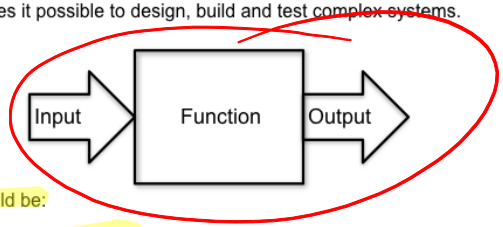
Abstraction

Computers are complex things – far too complex to think about all the elements at once!



Intel Core i7 six-core processor

Abstraction makes it possible to design, build and test complex systems.



Any function should be:

- reasonably sensible
- simple enough to understand
- self-contained
- have defined interfaces

Example:
Add two numbers

We have already come across an example of abstraction, where?

Notes:

Abstraction

It would be impossible to take a big bag of atoms and assemble a computer. The problem would be far too difficult to do without making mistakes and it would take a very long time!

Hierarchy and abstraction make the problem tractable. It depicts the level of 'detail'.

- Materials science allows the building of electronic components with particular properties
  - The computer engineer thinks of these as '**switches**'
- The computer engineer builds circuits from these components
  - These form a **library of gates** with well-defined functions.
- The gates are used to build larger components …
  - … with **functions** like 'add', 'multiply', 'store', etc.
- The computer architect assembles these into a computer
  - The computer performs a set of operations – the **instruction set**.
- The compiler makes sequences of instructions …
  - … from 'high-level' **statements** (e.g. Java).
- The programmer writes 'code' (e.g. objects, libraries etc.) …
  - … which implement (high-level) functions (e.g. draw a window)
- The system designer specifies the programme
  - Built largely from existing libraries.

At each level the problems are tractable.

The programmer does not care *how* the compiler works, as long as it works correctly. Similarly, the engineer is unconcerned with the behaviour of the electrons whizzing around the processor, providing that the working model produces the correct result.

Caveat: a computer scientist is concerned with most of these topics. For example, knowing a little about how the machine works can help choose better ways of structuring code. Similarly, knowing what people want to use the finished machine for will influence the details of its design.

Guidelines on using abstraction

Abstraction should be used throughout a system design. These rules apply for both hardware and software design.

There is no particular rule as to what should be grouped together but:

- The complexity should not be too great
  - whether code (text) or schematics (diagrams) it should fit approximately onto a single page, without crowding
  - it should be comprehensible without strain
  - if is starts to look too complex, introduce another level of hierarchy
- A 'single function' makes sense
  - it should be describable quite easily
- The interfaces should be specified
  - what goes in?
  - what comes out?
- Ideally all interaction should be through the interfaces
  - no "side effects"

In this course …

Gates are the lowest level of abstraction used in this course.

We have already seen their function and some methods of depicting and manipulating them.

The next stage is to build these into more complex functions.

Finally, we make a (simple) computer.

MANCHESTER 1824

The University of Manchester

Hierarchy

Split a complex design into more simpler blocks that are more "manageable" and, more importantly, reusable.

A general purpose binary adder comprises a number of individual operations. Although it would be possible to build this as a single, monolithic entity this would be difficult.

Instead the problem can be broken down hierarchically:

- The sum of n bit numbers requires n adders to perform each sum
- Each sum is the sum of 3 bits (A_i, B_i, and a carry in,C_i)
- We design the logic to add 3 bits (which can be two 2-bit sums) and replicate over all n bits

$A_i+B_i+C_i=(A_i+B_i)+C_i$

It is easiest to:

- Define, design and test each function once
- Encapsulate each function in its own symbol
- Reuse logic as much as is feasible

Keeping individual functions small makes them more tractable

These principles apply to the design of anything … even software!

Version 2016 COMP12111:Overview & Introduction to Logic

Notes:

Hierarchy

Although a computer may be made entirely out of (say) 2-input NAND gates, it would be tedious and confusing to try to do this 'directly'. For practical purposes hierarchy is introduced so that a complex functional block will be made of a number of less complex blocks, each of which comprises a number of simpler blocks. Alternatively, looked at from the 'bottom up', simple gates are used to build a variety of larger components, which can be used in still larger components …

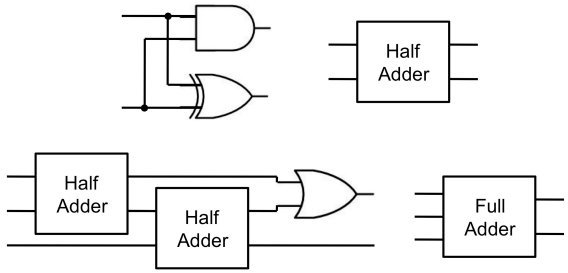
Introducing hierarchy is not just a matter of randomly partitioning a design. It is best to identify parts with similar – or, preferably, identical – functions so that a unit may be produced (and tested) once and then used repeatedly.

Adding multiple bit numbers is somewhat harder. First the bits are paired off according to their significance. They can then be added, but each 1-bit sum produces a 2-bit result. The 'carry' output (be it '0' or '1') needs to be added in to the most significant bit. Therefore, *another* addition must be performed.

Binary addition is therefore the addition of three binary digits, many times over. However, adding three digits can be done by adding pairs of digits, twice, i.e.

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

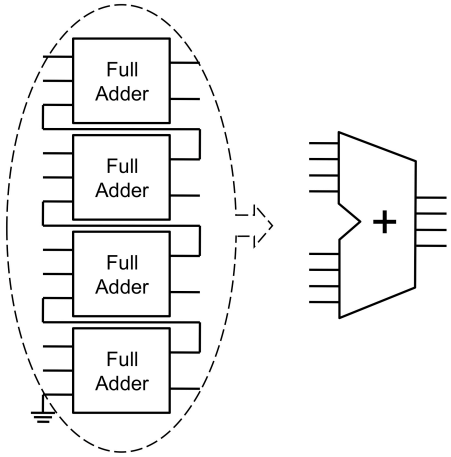
A device which performs the full, three digit addition is known as a **full adder**. A device which adds two digits (as above) is known as a **half adder**. A full adder may be built from two half adders (plus an extra gate).



Another Level of Hierarchy

The full adder is capable of adding two bits plus a carry together, but this is only the same as adding two 1-bit numbers. Usually numbers are larger than this in order to represent real quantities; 32-bit numbers are currently quite common.

It would be normal in the next level of hierarchy in a circuit, such as an adder, to construct a larger, multi-bit adder; here only a 4-bit adder is drawn, but it should be clear how this could be extended to 8-, 16-, 32- bits.



MANCHESTER 1824

The University of Manchester

Sequential Systems

The logic circuits we have looked out so far are called combinatorial circuits, the output signal depends on the values of the input signals to the circuit at any particular instance in time.

However, how would you build more complex circuits that perform very complex logical functions?

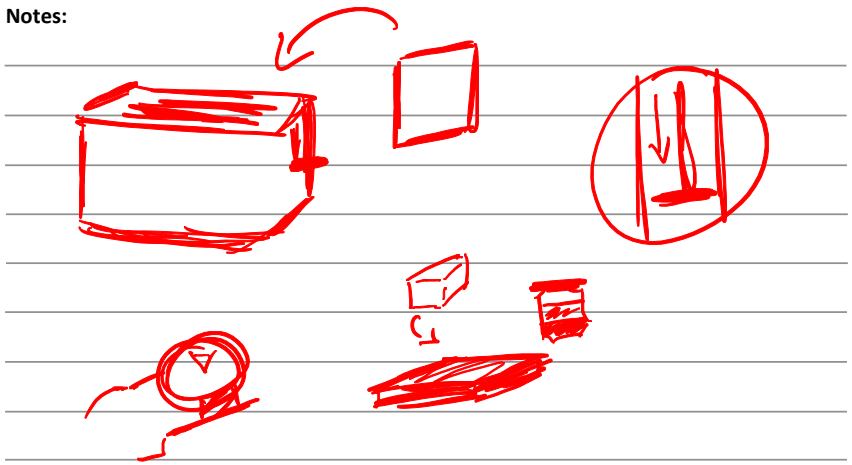
Most digital systems are implementing some form of algorithm as a sequence of steps.

Consider eating jam on toast, this follows a sequence of steps:

1. Put slices of bread in the toaster.
2. Press the leaver on the toaster.
3. Wait for the toast to pop up.
4. Spread the toast with butter.
5. Spread the toast with jam.
6. Eat.

Each step will take a finite time to complete before you move onto the next step.

Notes:



Sequential Systems

It is clear for the example given that you must follow the correct order in the sequence. Of course you can butter your bread directly and have jam on bread, but I don't think you'd want to put it in the toaster at that point!

Follow a sequence of steps in an algorithm correctly is essential if you are to implement the algorithm correctly, sequential circuits help us to achieve the correct "sequence" in our design.

Let's consider another example – a pelican pedestrian crossing[2]. When a pedestrian wishes to cross a busy road using a pelican crossing they press the button and wait patiently for the green man to appear, so they can cross the road. At the same time the traffics lights controlling the traffic must go to red to stop the traffic. The sequence of events is given below:

Traffic lights on green, red man illuminated
⇓
Pedestrian presses button
⇓
Traffic lights go on red, green man lights up
⇓
Wait for a period
⇓
After a period the amber light flashes, along with the green man
⇓
Wait for a period
⇓
Traffic light goes to green, red man is illuminated
⇓
Sequence repeats

If this sequence goes wrong, for whatever reason, then disaster could happen! You will also notice that there are pre-defined delays associated with some of the steps, which differ between steps, so we will reside in some steps longer than others. How do we know how long to stay in each step? We need to time the sequence in some way, to know when each step is complete. We do this by using an external clock as a timing reference, which we can use to determine how long to stay in each step, and when to move on. This is the essence of sequential design, we have a clock synchronising a sequence of events.

[2] http://en.wikipedia.org/wiki/Pelican_crossing

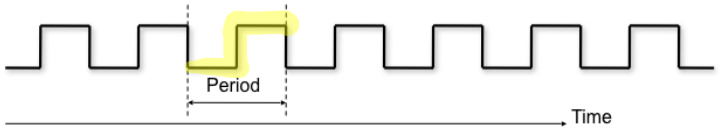
MANCHESTER 1824

The University of Manchester

Sequential Systems – the clock

So in the algorithm the steps are sequenced in time. Hence, we use a **clock** to ensure steps happen at the correct time.

A clock is a signal consisting of an infinite stream of pulses changing between 0 and 1 at a known rate called the **frequency**.



$F = \frac{1}{T}$

In sequential system we ensure actions are performed with respect to the same timing on the clock, for example the rising edge, this way we ensure that steps in the process are sequenced.

The time between identical points in the clock is the **period**, which is the inverse of the clock frequency.

Notes:

Clocks

A typical clock for digital logic will 'tick' many times per second. A slow clock may be in the kilohertz (kHz) range – for example the oscillator in a digital watch is 32.786 kHz, i.e. it oscillates 32,786 times per second – do you know why this particular number? A microprocessor will usually be clocked at frequencies in the gigahertz (GHz) – 10,000,000,000 Hz – range. The maximum speed available increases from year to year! Frequencies in this range are also common on networks et alia.

It is often necessary to convey from clock frequencies to clock periods (i.e. the time between adjacent similar clock edges); this is easy – each is the reciprocal of the other:

$$period = \frac{1}{frequency}$$

Thus, a clock with a frequency of 20MHz will have a period of 1/20 μs, or 50 ns. It is in your interest to become familiar with such conversions so that the units come out correct!

In many applications precise timing is important, especially in applications such as graphics where all the pixels should be the same, constant size. The usual method of guaranteeing adequate precision is to use a quartz **crystal** oscillator. Even a relatively cheap crystal oscillator provides stability of 50ppm (parts per million).

Synchronous Design

In sequential design we have something called a finite state machine, or FSM, to control how we move through the different steps, or more correctly state (more later) of the system. The FSM requires all the state changes to occur at **defined times**, and at the **same time** across the whole system. This means that the clock signal is distributed **globally**.

Each state bit can be defined in terms of some or all of the existing state bits and some or all of the external inputs. The clock slows the circuit down so that the next state of all the flip-flops can be prepared before an of the existing state is lost.

This means that the logic to perform the calculations can be designed without worrying about how long each bit takes to calculate.

Is that really true?

Yes, providing the clock is slow enough that all the logic has evaluated before the next active transition. Of course, in the real world there is a constant desire to go faster and faster, so that influences the logic design.

MANCHESTER 1824

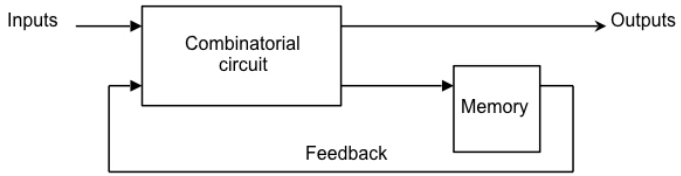
The University of Manchester

Sequential Systems

So how can we implement a series of timed steps in hardware?

… we use a sequential system.

In a sequential system, the output signal depends on the current value of the input signals **AND** the past history of its inputs. It requires some form of memory in order to store information.

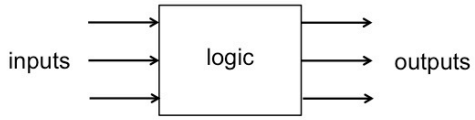


The memory can be implemented using a logic device called a **flip-flop**, which is controlled by a clock to ensure the **state** of the system is updated when required.

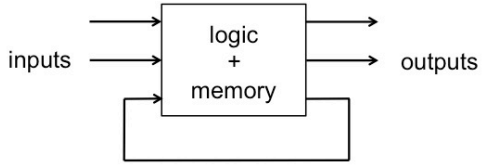
Notes:

Sequential Systems

Combinatorial circuits contain no memory, the output from the logic circuit depends on only the values (or "state") of the input signals at the current time:



Whereas sequential systems have memory, so the output depends on the current state of the input signals AND a previous state.



Consequently, operations can be performed by building on *past* operations in a *sequential* manner.

In sequential systems one of the inputs is usually a clock and the outputs change in relation to changes in the state of the external clock. For example, the outputs may be updated on every rising edge (0->1) of the clock.

The memory is used to hold the current *state* of the system for later use. More about state later.

Combinatorial[1] logic is capable of processing an arbitrary number of inputs to produce an arbitrary number of outputs; it can also evaluate any function. However, this assumes that specialised, complex, reliable, (expensive) logic can be defined for each function required. In practice this is infeasible.

In order to reduce a complex job to a manageable size it can be broken down into smaller jobs (hierarchy). One way of approaching this is to repeat simple hardware, as seen in the adder example. A different approach is to use the same hardware in repeated steps to process different parts of the problem; this is the approach that a computer takes, solving the problem one instruction at a time. This calls for **sequential** logic.

To solve a problem in a series of time steps results from earlier steps much be held and, sometimes, used as inputs to later calculations. As an example consider long multiplication.

You can probably multiply all the single digit numbers in your head. Very few people can multiply four digit numbers mentally – at least without going through some more complex process (long multiplication and remembering the intermediate stages rather than using paper doesn't count!) Instead, the task is broken down into smaller, tractable multiplications and the result of each stored. Finally, the results are recalled and totalled.

These intermediate results are held as part of the **state** of the system (more later). A machine which worked this way would need the requisite number of flip-flops/latches to hold these numbers until they were needed. Furthermore, this is not the whole state of the calculation as it is also necessary to "keep track" of which steps have been done, determine what to do next, and know when the result is ready. There is thus some state stored in the **control** logic, as well as the **data processing** logic.

As another consideration, remember that many jobs are interactive over time; it is therefore necessary to have some means of controlling progress through time. Think of controlling a washing machine, or playing a video game.

[1] Americans call this "combinational" logic

MANCHESTER 1824

The University of Manchester

Flip-flops

A sequential system needs a memory element to store information, this is implemented practically using a device called a flip-flop.

Activation of input(s) causes the flip-flop to assume one of two states '0' and '1', which it "remembers", i.e. it remains in, even after the input(s) are deactivated.

It remains in this state until driven into another state by suitable activation of one or more inputs.

Hence, the flip-flop is a simple one bit memory element, you can store either a '0' or '1' to the device, which it will remember until the value is overwritten.

Flip-flops come in many varieties:

- the RS (or sometimes SR) flip-flop,
- the D-type flip-flop,
- the J-K flip-flop, and others.

However, as computer engineers, the only flip-flop we are interested in is the D-type flip-flop, as this is the most commonly used.

Notes:

Sequential Systems

The previous circuits we have looked at are examples of **combinatorial logic** whose output are *functions of the inputs only*. Another type of circuit is the **sequential circuit** whose output depends upon the value of its inputs AND *past inputs*. A sequential circuit has **memory**; it has the ability to remember previous values of its inputs/output.

A single bit memory element forms the basic building block of a sequential circuit and can be implemented using a flip-flop, or latch. Sequential circuits often have a clock input, which triggers the storage of the current **state** of the input.

Sequential circuits may perform many different functions. General functions include:

Register

A register is made of a number of, *n*, flip-flops in a row and is able to store an *n*-bit word. The memory in a computer is simply a very large array of registers storing (typically) *n*=32 bit values.

Shift Register

A shift register is a special register that allows the data that is stored to be shifted left or right. For example, if an 8-bit shift register holds the value 01101010, then the contents can be shifted to the left 11010100, or to the right 00110101.

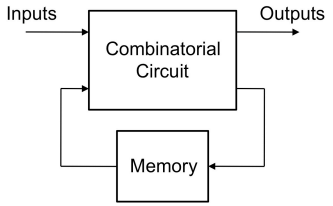
Counter

A counter is another example of a special register that holds an *n*-bit value. However, when the counter is triggered (via some external clock) then the contents of the counter increment by 1. Counters can increment, decrement, or count through any arbitrary sequence.

Finite State Machine

A finite state machine is a digital system that holds the current *state* of a digital system (in the form of a binary code) and moves from one state to another depending upon the current state value and the current status of any inputs. A simple state machine can be used to control the sequence of lights in a traffic light system. Ultimately, a computer is a complex example of a finite state machine that is controlled by a program (inputs) and its data (current state).

A basic sequential circuit has the following block diagram:



The sequential circuit consists of a combinatorial logic block (consisting of conventional NAD, OR, and NOT gates) and a memory element. The binary information stored in the memory defines the *current state* of the sequential circuit at any particular time. The combinatorial logic determines the output signals, as a function of the inputs and the current state stored in the memory, as well as determining the *next state* from the current state and the inputs.

D-type Flip-flop

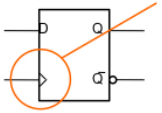
The D-type flip-flop is an edge triggered device, as a minimum it will have two inputs:

- a data input, and
- a clock input.

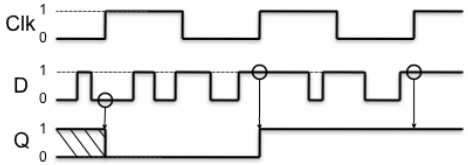
The triangle represents the clock input

As the D-type flip-flop is such an important device, it has its own symbol.

It has 2 complementary outputs, which hold the stored value.



The data on the data input, D, is stored at every rising edge of the clock input



Notes:

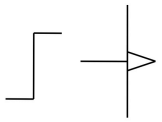
Edge triggered flip-flops

Unlike the level sensitive latches (previous slide), edge triggered devices change state in response to **transitions** between levels. Internally these are more complex than level sensitive devices, but they are often far more convenient to use.

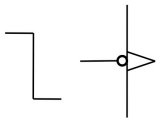
An edge-triggered device can be thought of as changing at a single instant. It never becomes transparent, instead taking a 'snapshot' of its input on demand.

It is normal to trigger off only one edge (the **active** edge), either a **rising edge** (0 ⇒ 1 transition) or a **falling edge** (1 ⇒ 0 transition). These are also called **positive edges** and **negative edges** respectively. The signal that makes the transition is usually referred to as the **clock** (typically abbreviated to 'clk').

The usual symbol used to indicate an edge triggered clock input is a "V" shape:



Positive edge triggered



Negative edge triggered

Other Types of Flip-flop

Occasionally you may encounter other types of flip-flop (though not in this course). These are primarily of historic interest, examples include the JK flip-flop and toggle flip-flop. See what you can find out about the operation of these devices.

In general, a latch is level sensitive whereas a flip-flop is edge-triggered. The most common flip-flop that you will encounter in the lab is the D-type flip-flop.

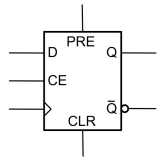
Real D-types

When dealing with edge-triggered flip-flops and circuits which use them it is **vitally important** to distinguish between **synchronous** and **asynchronous** inputs.

- A **synchronous** input causes a change when the device is ***clocked***,
- An **asynchronous** input can cause a change at ***any time***.

On a D-type flip-flop the 'D' input is synchronous (by definition). Other inputs may be synchronous or asynchronous. For example, a 'clear' input could be synchronous (i.e. clears the flip-flop only after the next clock) or asynchronous (i.e. takes effect immediately, irrespective of the state of the clock).

Often flip-flops have additional inputs to provide greater flexibility, such as an asynchronous preset and clear, as well as a flip-flop enable, as shown.



PRE – asynchronous preset that will initial the flip-flop Q output to '0' or '1' depending on the value of PRE.

CLR – asynchronous clear that will clear the flip-flop Q output to 0 when taken high.

The CE, chip enable, input must be high (active) for a clock transition to affect the flip-flop's state. It is used to ensure that the flip-flop state is only changed when it is required to do so.

MANCHESTER 1824

The University of Manchester

State

What does the value stored in the memory in a sequential system represent?

… it defines the state of the sequential circuit at that time.

The next state of a system is determined by its current state and the state of any input signals – more about this later.

As the sequential circuit implements an algorithm as a sequence of steps … the state represents which step of the sequence we are currently in.

Each possible state will be identified by a unique number, determined by the range of values that can be stored to the memory element (we will look at registers later).

- a bit can have one of two states
- a byte (8 bits) has one of 256 states
- a 32-bit word can adopt one of 4,294,967,296 states

We will come back to state again in a later lecture when we start to look at designing sequential systems.

Notes:

Sequential Logic

Combinatorial[1] logic is capable of processing an arbitrary number of inputs to produce an arbitrary number of outputs; it can also evaluate any function. However, this assumes that specialised, complex, reliable, (expensive) logic can be defined for each function required. In practice this is infeasible.

In order to reduce a complex job to a manageable size it can be broken down into smaller jobs (hierarchy). One way of approaching this is to repeat simple hardware, as seen in the adder example. A different approach is to use the same hardware in repeated steps to process different parts of the problem; this is the approach that a computer takes, solving the problem one instruction at a time. This calls for **sequential** logic.

To solve a problem in a series of time steps results from earlier steps much be held and, sometimes, used as inputs to later calculations. As an example consider long multiplication.

You can probably multiply all the single digit numbers in your head. Very few people can multiply four digit numbers mentally – at least without going through some more complex process (long multiplication and remembering the intermediate stages rather than using paper doesn't count!) Instead, the task is broken down into smaller, tractable multiplications and the result of each stored. Finally, the results are recalled and totalled.

These intermediate results are held as part of the **state** of the system. A machine which worked this way would need the requisite number of flip-flops/latches to hold these numbers until they were needed. Furthermore, this is not the whole state of the calculation as it is also necessary to "keep track" of which steps have been done, determine what to do next, and know when the result is ready. There is thus some state stored in the **control** logic, as well as the **data processing** logic.

As another consideration, remember that many jobs are interactive over time; it is therefore necessary to have some means of controlling progress through time. Think of controlling a washing machine, or playing a video game.

Sequential Logic – A Summary

- In general, the outputs from sequential logic circuits depend upon a function of their input values, as well as an arbitrary number of previous input values.
- In other words, the outputs are a function of the history (or sequence) of previous input values, as well as the current input values.
- The usual way of simplifying this view is to regard a sequential logic circuit as having a current state. The sequential circuit stores one or more binary digits (bits) which determine which state it is in.
- Thus, the output of a sequential circuit is a function of the current inputs and the current state.
- The state may (optionally) change under some combinations of current state and inputs. This allows the state to sequence through many different possible values.
- Indeed, useful sequential circuits can be made where there are no external inputs at all. The outputs, and the next state, are solely determined by the current state.

[1] Americans call this "combinational" log

Quiz-time

Q. For the eating jam on toast example, how many states are there?

A. 1
B. 2
C. 6
D. 16

Notes:

Quiz-time

Q. If you had to represent each of the states for the eating jam on toast example as a binary value, how many bits are required?

A. 1
B. 2
C. 3
D. 4

Notes:

MANCHESTER 1824

The University of Manchester

Delays in circuits …

One may assume that the D-type flip-flop stores the value on the D input immediately on seeing a rising edge on the clock input …

… this isn't the case!

Nothing acts immediately in a digital circuit, signals take a finite time to change state and there is always a delay before a input change causes the output of a circuit to change.

Each logic gate will thus have a delay associated with it. The more logic gates a signal must propagate through, the longer the delay. Such delays are inherent to the circuit.

One advantage of using sequential systems is that by taking into account the delay through a circuit, and choosing the clock frequency appropriately, you can ensure that output signals have settled to the desired value before you use them.

Notes:

Quiz-time

Q. What would you choose as the clock frequency?

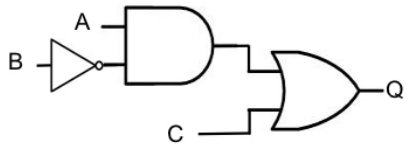
A. Any value, it doesn't matter?
B. A frequency corresponding to the shortest signal delay through the circuit?
C. A frequency corresponding to the longest delay through the circuit?

Notes:

Quiz-time

Q. Which input is likely to contribute to the longest delay through the circuit shown?

A. All of them
B. Signal A
C. Signal B
D. Signal C



Notes:

Quiz-time

Q. If the delay through a full adder design is 30ps, what could be the approximate delay through a 16-bit adder?

A. 30ps
B. 120ps
C. 240ps
D. 480ps

Notes:

MANCHESTER 1824

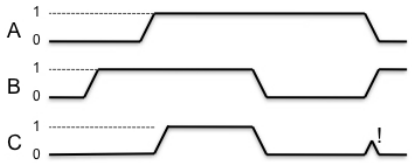
The University of Manchester

Timing Diagrams

Truth tables depict all the possible outcomes of a logic function, but they are **static**, i.e. they do not demonstrate dynamic (time varying) behaviour.

Another method of showing a logic function is to use a **timing diagram**, which will show us how the signals vary in time and, more importantly, can help identify delays in a circuit.

For example, the AND gate:



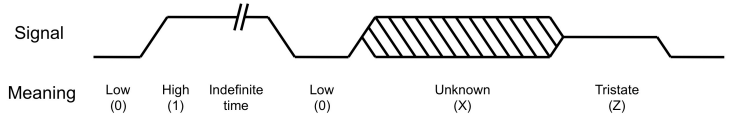
You will observe such "waveforms" in the lab

- a timing diagram may help highlight possible **hazards**, such as **glitches**
- timing diagrams are useful when describing **sequential circuits**

Notes:

Timing Diagrams

In a timing diagram time proceeds from left to right[1] and a line is used to depict the logic level.



A 'high' ('1') is a high line and a 'low' ('0') is lower! Other states are also shown: tri-state is a high impedance state that effectively 'removes' the out from the circuit.

Timing diagrams are used to depict timing relationships. They are **not always drawn to scale**, but later **transitions** should always to be to the right of earlier ones. Sometimes arrows are added between transitions, usually meaning "this precedes that" or, more emphatically, "this" changes causes "that" to change.

In the real world …

A wire's state does not switch instantaneously from 0⇒1, or 1⇒0, instead there is a finite **edge speed**.

The output of a gate does not switch at the instant its inputs change; there is some **propagation delay**.

These inevitable properties of any physical system are sometimes omitted on timing diagrams. In the above diagrams these delays have been included (and exaggerated) for illustrative purposes.

In real circuits edge speeds and propagation delays are of real concern; these are the factors that govern the speed of the circuit (for example the clock rate of a PC). However, for now, note that they exist and we will leave detailed discussion to a later course.

Timing Diagram Applications

Design

Drawing out what you want to happen and when you want it to happen is a useful design aid. You will use timing diagrams extensively when designing sequential circuits and finite state machines, particularly in the lab!

Simulation

Given a design you thin (or hope!) will do your job, simulation software is capable of determining what would really happen. One very useful output form from simulation is a '**waveform trace**', which is a timing diagram. Hardware designers spend a lot of time staring at these!

[1] the traditional direction

MANCHESTER 1824

The University of Manchester

Timing diagrams – signal properties

Timing diagrams allow us to observe characteristics of digital devices, such as the D-type flip-flop.

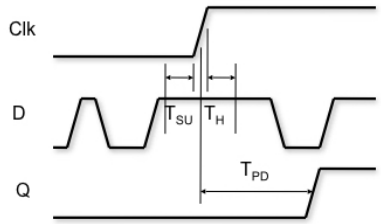
The value stored to the output does not change immediately at the rising edge of the clock, instead the input is sampled at some interval around the clock edge.

The interval is a characteristic of the device and is defined by:

- the **set-up time** (T_{SU})
- the **hold time** (T_H)

The data **must** be stable in this interval if the flip-flop is to function correctly.

There is also an associated time – the **propagation delay** (T_{PD}) – which is the time taken for the output to change.



In a well-engineering system $T_{PD}>T_H$

Notes:

Timing Issues

If the input of a flip-flop changes just as it is being sampled (the edge of the clock in the case of the D-type) then the flip-flop may not sample the signal correctly and enter a random state; this is clearly undesirable.

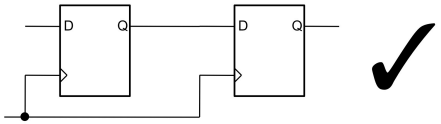
To avoid this, the data input:

- must cease changing at least an interval T_{SU} before the clock edge begins (T_{SU} – set-up time)
- must not change again until at least T_H after the clock edge is complete (T_H – hold time)

This gives a 'window' where the data can be sampled, as shown in the slide. In practice these times are quite small. (There are, or course, no restrictions around the inactive clock edge.)

A third timing characteristic is the **propagation delay** of the flip-flop; this is the time it takes an input change (in this case an active clock transition) to affect the output.

A well-engineered D-type flip-flop will have a hold time smaller than its propagation delay. This allows the output of a flip-flop to be used as the input for another flip-flop with the same clock without violating *its* hold time.



MANCHESTER 1824

The University of Manchester

Summary

We have looked at:

- data representation – how we use binary to represent information in computers
- Boolean logic and the rules for manipulating Boolean expressions
  - Truth tables
  - DeMorgan's theorem
- Gates and schematics for representing digital circuits
- Number representation in binary and simple mathematical operations
  - 2's complement arithmetic
- The issue of timing – signals do not change state instantaneously
- Abstraction & Hierarchy for simplifying design
- Sequential systems. and the D-type flip-flop as a memory storage element

Next, we will look at how we design sequential circuits.

Notes:

Summary

We have covered a lot of 'new' information in these lectures, and in a very short space of time. All this material is examinable and useful when it comes to the laboratory exercises, so make sure you understand the concepts, and have a go at the examples.

Next …

In the next set of notes we will be looking at hardware description languages, in particular Verilog, which allow you to express a hardware design in a software description.