Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 1 - Introduction

Session Outline

Academic Aims

Administration

Assessment

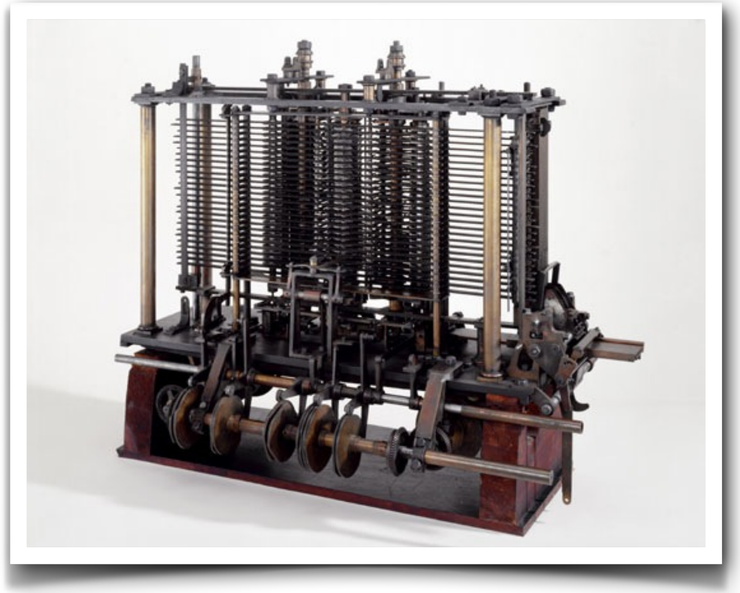
Lecture Topics

Recommended Books

Expectations

What is a Computer?

What is a Computer?



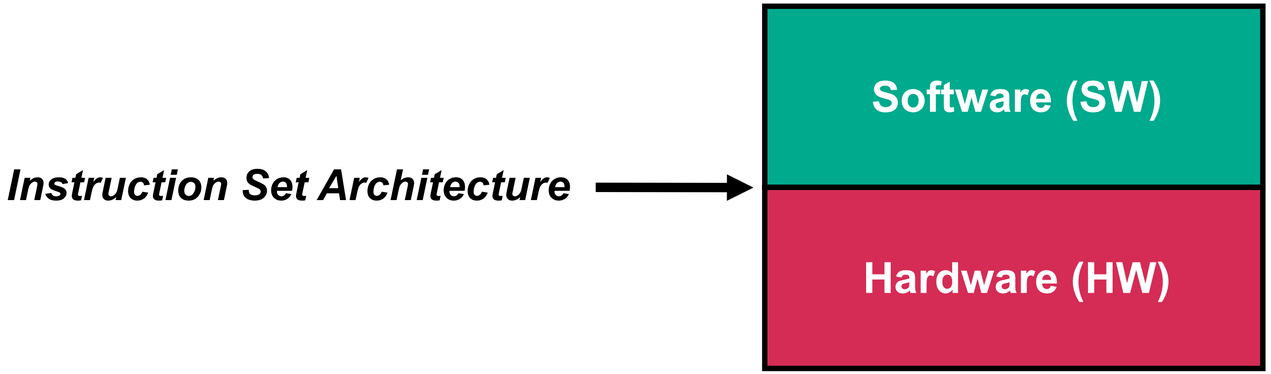
Module Aims

How do we define a computer?

How can we build a computer?

Computer architecture is fundamental

Design the layers of abstraction and the intervening interfaces



Module Administration

Module Organiser:

Matthew Leeke (CS 116, m.leeke@bham.ac.uk)
Niloofer Shanavas (n.shanavas@bham.ac.uk) [Dubai], Qamar Natsheh (q.natsheh@bham.ac.uk) [Dubai]

Module Lectures:

10am-11am on Wednesday (ARTS-Main LT) and 12-1pm on Wednesday (LAW-LT1 303), or 2-4pm on Friday (TLB-LG18). You will be allocated to the two hours on the Wednesday or the two hours on the Friday. You must attend only the two hours to which you are assigned.

Module Page:

https://canvas.bham.ac.uk/courses/77845

Module Assessment

The module is assessed by a examination (80%) and continuous assessment (20%)

Examination:

- Exclusively focused on computer systems topics
- Two-hour paper with three questions
- Takes place in the Main Examination Period

Continuous assessment:

- All details online - note the Thursday of Week 10, Term 1 submission deadline

Lecture Topics

1. Introduction
2. Data Representation
3. Digital Logic
4. Machine Code and Assembler
5. Memory Systems
6. I/O Mechanisms
7. Processor Architecture

Learning Outcomes

You should be able to:

Explain common data representations and their role in computer systems

Discuss the organisation and architecture of computer systems and their components, including logic components, microprocessors and I/O systems

Demonstrate basic systems programming and digital circuit prototyping abilities

Discuss a range of legal, social, ethical and professional issues relating to the design, implementation and deployment of computer systems

Write in an appropriate scientific style on aspects of computer systems and professional practice

Topic 1 - Introduction

Understand what we'll be doing in the module

Recognise that you'll need to exercise your professional practice in pursuit of computer systems

The module coursework requires independent study, beyond what you're likely to have experienced to this point

```
/* ADDR is a pointer to the byte
   at location 0xc0000 */
#define ADDR ( (volatile unsigned char *) 0xc0000 )

unsigned char b;
b = *ADDR;    /* read from location 0xc0000
                 into variable b */
*ADDR = 0x2A; /* write 42 into
                 location 0xc0000 */
```

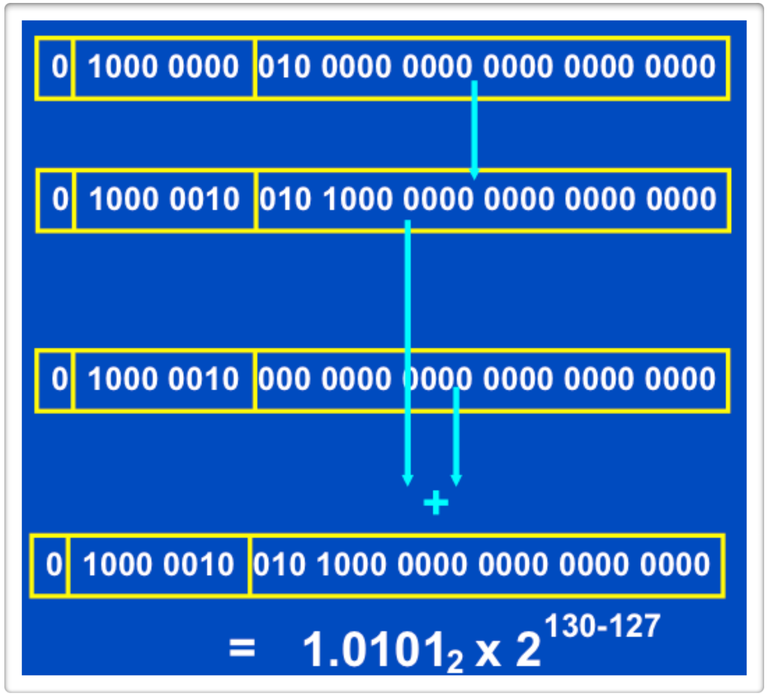
Topic 2 - Data Representation

We focus on representation as it relates to the encoding of numbers

Why numbers systems?

How can representations be manipulated to perform arithmetic operations

Why manipulate representation?

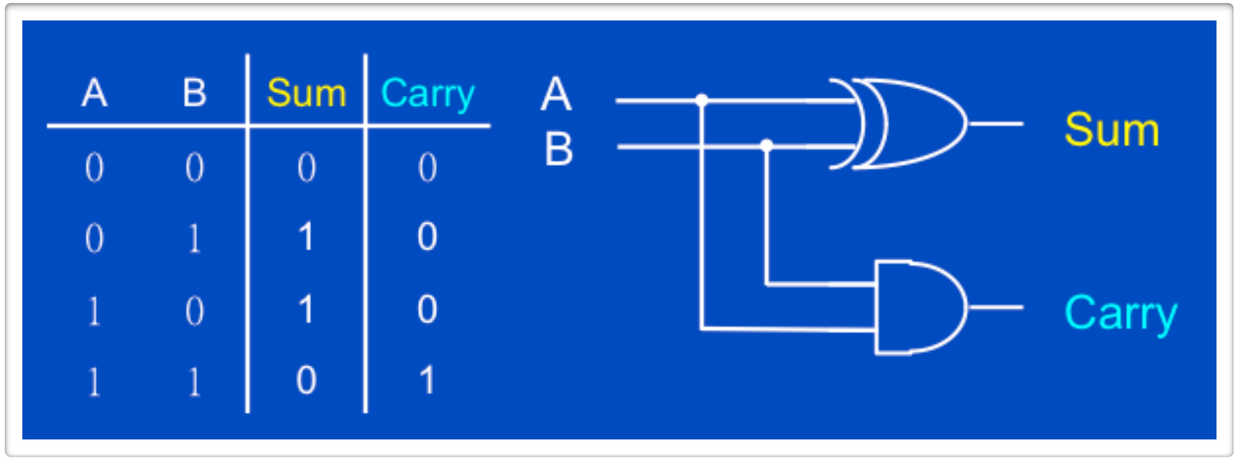


Topic 3 - Digital Logic

Logic gates and how logic circuits can be built from just a few elementary gates

Designing and constructing circuits for calculation / computation

How complicated logic functions can be simplified using Boolean algebra and Karnaugh maps



A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

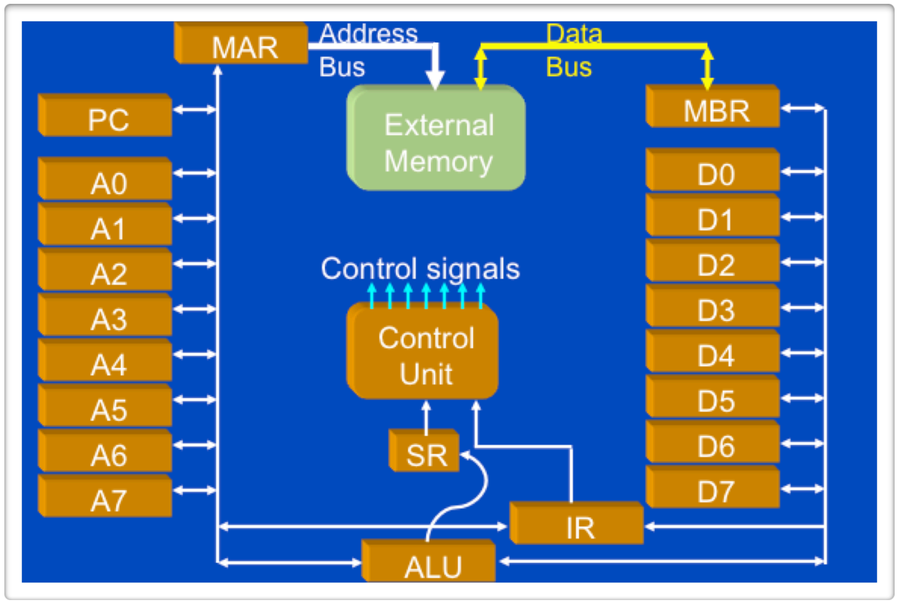
Topic 4 - Machine Code & Assembler

Learn by studying a real computer architecture

Write programs in assembly language

The role of subroutines and stacks

How addressing modes govern the way in which data is located



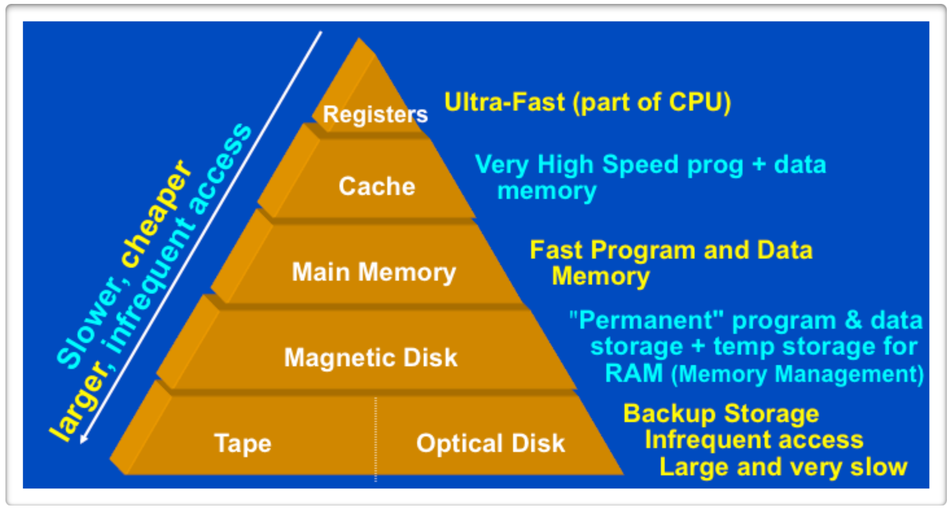
Topic 5 - Memory Systems

Principles and characteristics of storage devices

How the different devices can be organised into a memory hierarchy

Why do we bother with a memory hierarchy?

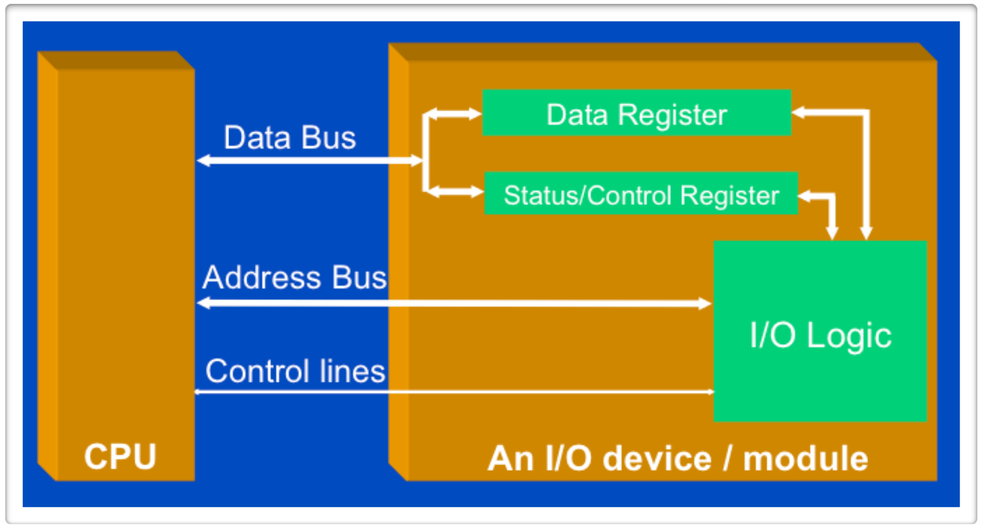
The detection and correction of data errors



Topic 6 - I/O Mechanisms

Input and output techniques that enable a computer system to communicate its environment

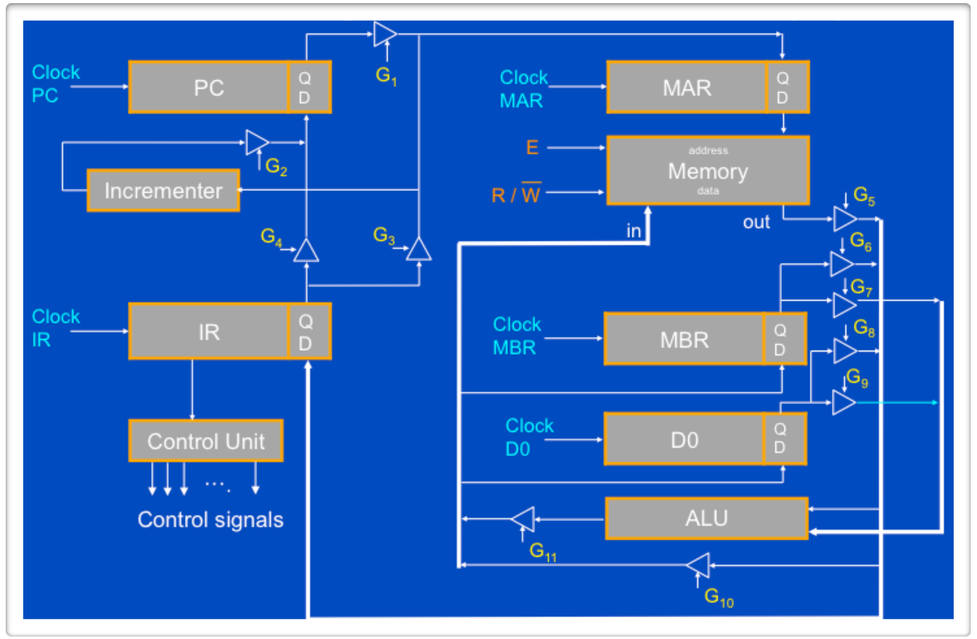
How do devices take input
and provide output?



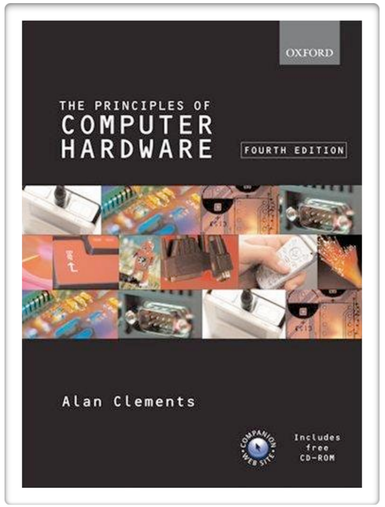
Topic 7 - Processor Architectures

How to combine registers, arithmetic unit, logic unit and control unit to create a processor

Surely the computer systems we use today are more than this?

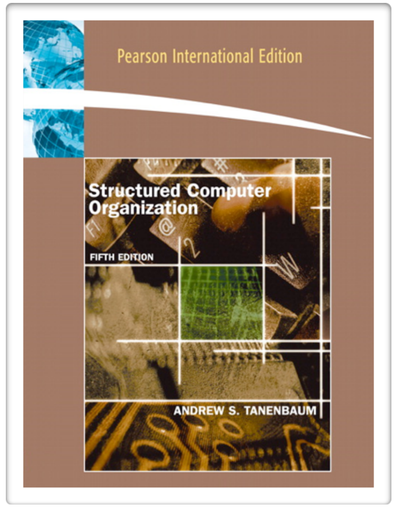


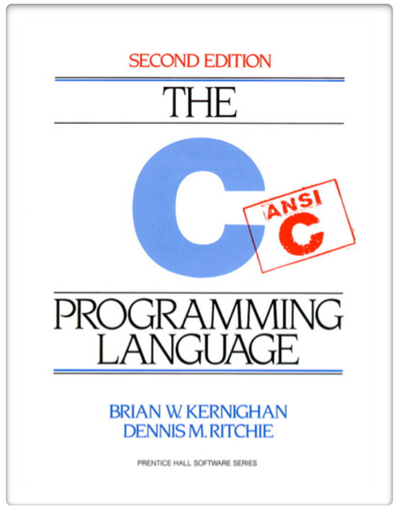
Recommended Books



A. Clements. Principles of Computer Hardware (4th Edition), Oxford University Press, February 2006

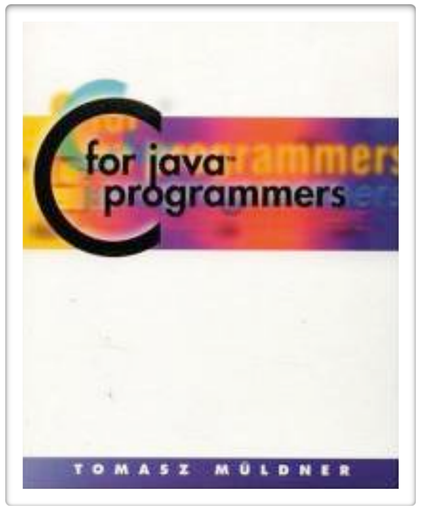
A. S. Tanenbaum. Structured Computer Organization (4th Edition), Pearson Education, February 2009





B. W. Kernighan, D. Ritchie. The C Programming Language (2nd Edition), Prentice Hall, March 1988

T. Muldner. C for Java Programmers (1st Edition), Addison-Wesley, March 2000



Summary

Aim is to provide you with an understanding of how the functional components of a computer system are organised

Emphasis on the hardware aspects of a computer system and how these facilitate the execution of software

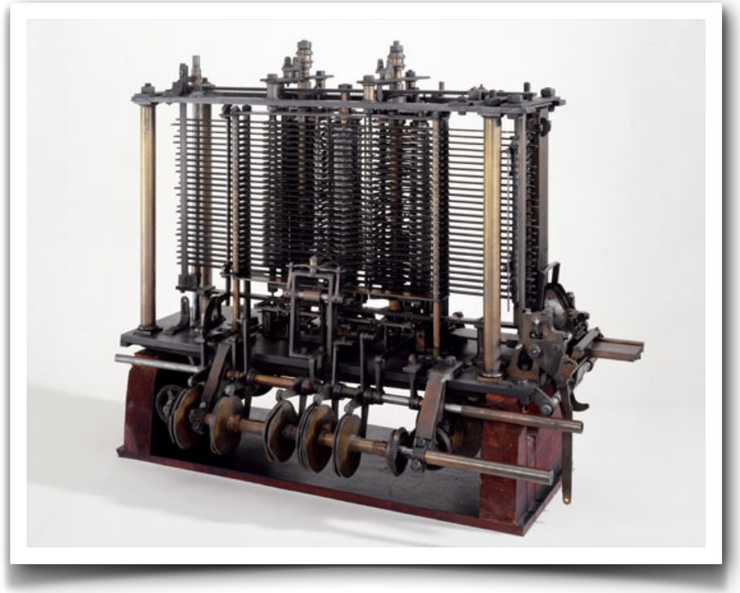
Practical skills in the design, construction and use of computer components, as well as their interfacing to microprocessors

Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 2 - Data Representation

Work It Out Yet?



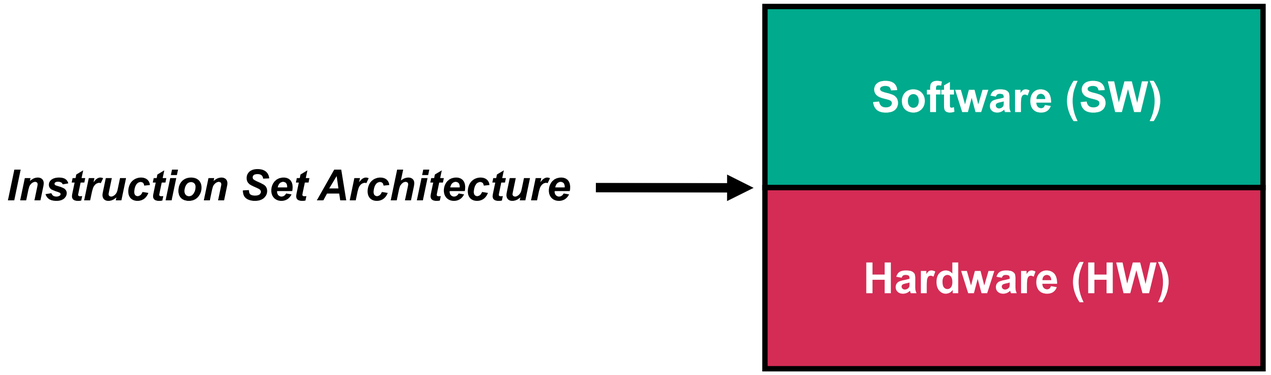
We Should Talk About Computers

How do we define a computer?

How can we build a computer?

Computer architecture is fundamental

Design of layers of abstraction and the intervening interfaces



Session Outline

Three key aspects of data representation in this lecture

Part 01_2: Representation and Number Systems

Part 10_2: Addition and Negative Numbers

Part 11_2: Fractional Numbers

Part 01_2: Representation and Number Systems

Getting Started - Numerical Representations

Tally marks	I, II, III, IIII, ~~IIII~~, ~~IIII~~ I, ~~IIII~~ II
Roman numerals	I, II, III, IV, V, VI, VII, VIII, IX, X, XI
Decimal integers	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
Fractions	1/2, 1/4, 2/16, 100/1100, 50/250
Scientific	2.998×10^8 m/s (≈299792458 m/s)
Decimal fractions	23.5812, 5.07, 4.20, -21.479
Binary	0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011
Octal	0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12 … 16, 17, 20, 21
Hexadecimal	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12 … 18, 19, 1A

One Value, Many Representations

In an abstract sense we all know what we mean when we talk about the number “7”

Not everyone would be able to represent the number “7” in binary, octal or hexadecimal

- Symbols must be known
- Even then there are issues relating to representation

The ideas of value and representation are different

- Value is necessarily independent of representation

Roman	Decimal	Binary	Octal	Hex
?	0	0	0	0
I	1	1	1	1
II	2	10	2	2
III	3	11	3	3
IV	4	100	4	4
V	5	101	5	5
VI	6	110	6	6
VII	7	111	7	7
VIII	8	1000	10	8
IX	9	1001	11	9
X	10	1010	12	A
XI	11	1011	13	B
XII	12	1100	14	C
XIII	13	1101	15	D
XIV	14	1110	16	E
XV	15	1111	17	F
XVI	16	10000	20	10
XX	20	10100	24	14

Bases

What are the available symbols?

Decimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 : Base 10

Octal: 0, 1, 2, 3, 4, 5, 6, 7 : Base 8

Hex: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F : Base 16

Binary: 0, 1 : Base 2

Important to distinguish between representations in use

$16_{10} = 10000_2 = 20_8 = 10_{16}$

$8_{10} = 1000_2 = 10_8 = 8_{16}$

Positional Representation - Decimal

Regardless of how you were taught the decimal number system this calculation is implicit every time you talk about numbers in decimal

£	3	2	3	7
Position:	Thousands	Hundreds	Tens	Units
=	1000	100	10	1
=	10^3	10^2	10^1	10^0
Value =	3×10^3 +	2×10^2 +	3×10^1 +	7×10^0

$$value = \sum_{i=0}^{N-1} symbol_i \times base^i$$

N: The number of columns
i: The index of position

Positional Representation - Binary

Binary number system operates under the same principles as decimal number systems, just with alternative symbols and position values

1011_2	**=**	**1**	**0**	**1**	**1**
Position:		**2^3**	**2^2**	**2^1**	**2^0**
	=	**8**	**4**	**2**	**1**
Value	**=**	**1 x 8 +**	**0 x 4 +**	**1 x 2 +**	**1 x 1**
	=	**11_{10}**			

$$value = \sum_{i=0}^{N-1} symbol_i \times base^i$$

N: The number of columns
i: The index of position

Counting in Binary

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
...
```

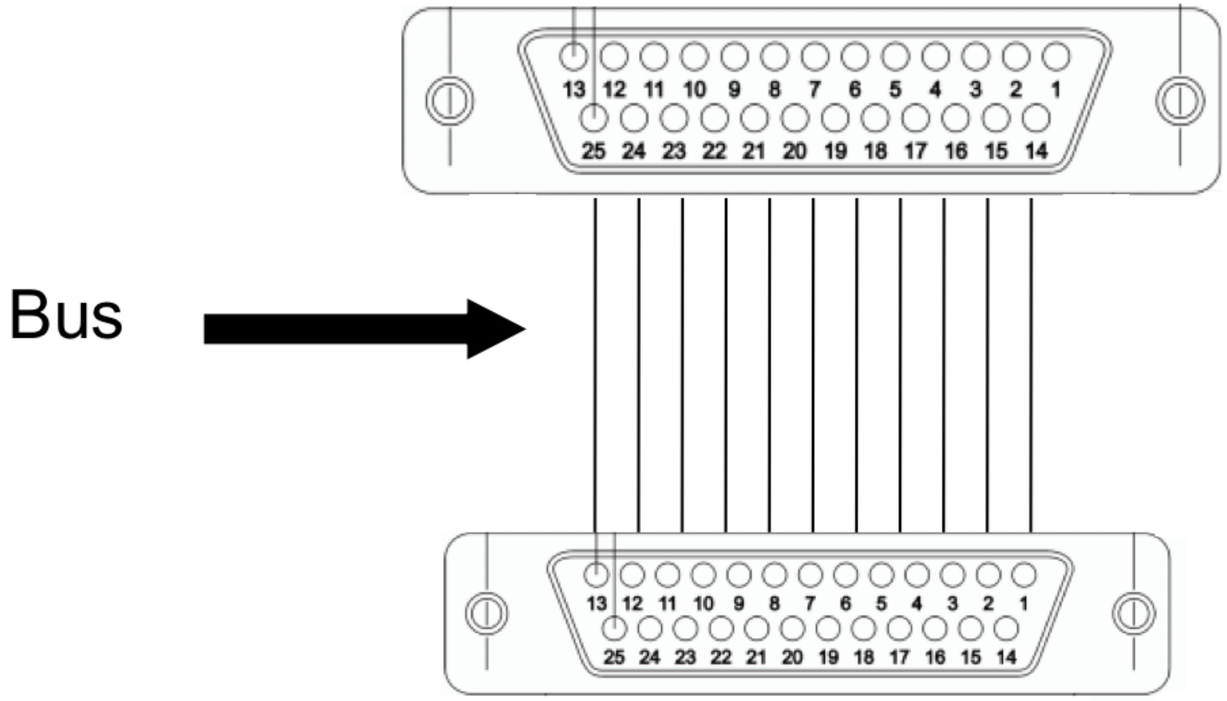
This is how we count in binary – but why is it this particular pattern, and how do we construct it?

Electrical Circuit Representation

TTL (transistor-transistor logic)

0V – 0.8V represents 0

2.4V – 5V represents 1



Parallel bus

Collection of wires communicating a value between sub-circuits

Bits, Bytes, Words and Bus Sizes

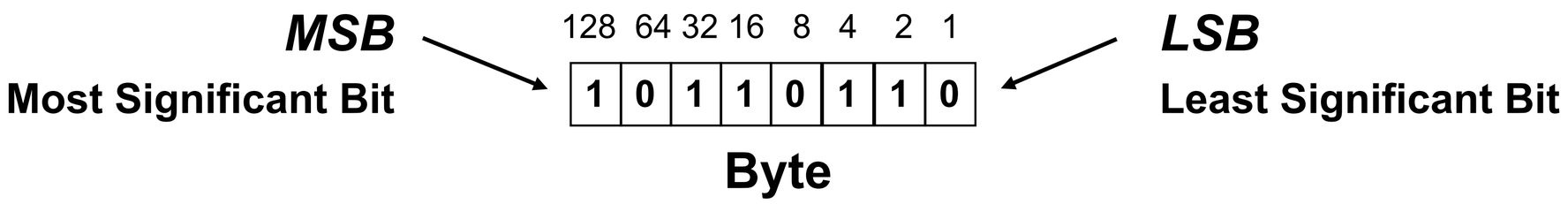
Bit - Binary digit (values 0_2 or 1_2 inclusive)

Byte - 8 bits (values 0_{10} to 255_{10} inclusive)

Word - The number of bits a machine can process simultaneously

4 bits = half a byte = nibble (values 0 - 15_{10})

most significant bit (MSB) and least significant bit (LSB) of a word are the bits with the highest and lowest place values respectively



A Brief History of Processor Representation

Intel 4004 (1971) - The first commercially available microprocessor had a word length of 4-bits

MOS 6502 and Zilog Z80 (~1975) - Popular 8-bit processors in early personal computers, such as the Apple II (6502) and and TRS-80 (Z80).

Motorola 68030, PowerPC G4, and Intel Pentium (upto 2004) - Processor at the era of 32-bit architecture, being widely used in personal computers and workstations.

IBM PowerPC G5 and x86-64 (e.g., Intel Core 2 and AMD64, introduced in 2003–2004) - The transition to 64-bit, offering higher performance, larger addressable memory space, and greater efficiency in modern computing.

Modern Processors (2020s onwards) - Processors like the Intel Core i9, AMD Ryzen, and Apple M-series continue to evolve the 64-bit architecture (with multi-core designs, advanced parallel processing, and coprocessors for AI and machine learning) and be informed by design elements of ARM and RISC-V

Sizes of Symbols

One octal symbol can represent 3 bits

$111_2 = 7_8$

$010\ 100\ 011_2 = 243_8$

One hex symbol can represent 4 bits

$1111_2 = 15_{10} = F_{16}$

$1010\ 0011_2 = A3_{16}$

One decimal symbol requires ≈ 3.3 bits, so hex and octal are more convenient than decimal when describing values on a bus

Converting from Decimal to Binary

Repeatedly divide the number by the base required, e.g., 2 for binary.

Record the remainder, e.g., for binary it is 0 if the number was even and 1 if it was odd

Quotient	**Remainder**
163	-
81	1
40	1
20	0
10	0
5	0
2	1
1	0
0	1

$\longrightarrow$ $163_{10} = 1\ 0\ 1\ 0\ 0\ 0\ 1\ 1_2$

Converting from Decimal to Octal or Hex

The division method can be used to convert to other bases

Divide by the required base, e.g., 8 for octal and 16 for hexadecimal

It may be easier to convert into binary first and then into the required base

$$
\begin{aligned}
23_{10} &= 16 + 4 + 2 + 1 \\
&= 10111_2 \\
&= 27_8 \quad (010\ 111_2) \\
&= 17_{16}\ (0001\ 0111_2)
\end{aligned}
$$

Pop Quiz - Questions

Covert 113_{10} to binary

Covert $011\ 110\ 111_2$ to decimal

Covert $110\ 101\ 011_2$ to octal

Covert $1100\ 0111\ 0110_2$ to hexadecimal

Pop Quiz - Answers

Covert 113_{10} to binary = $\mathbf{01110001_2}$

Covert $011\ 110\ 111_2$ to decimal = $\mathbf{247_{10}}$

Covert $110\ 101\ 011_2$ to octal = $\mathbf{653_8}$

Covert $1100\ 0111\ 0110_2$ to hexadecimal = $\mathbf{C76_{16}}$

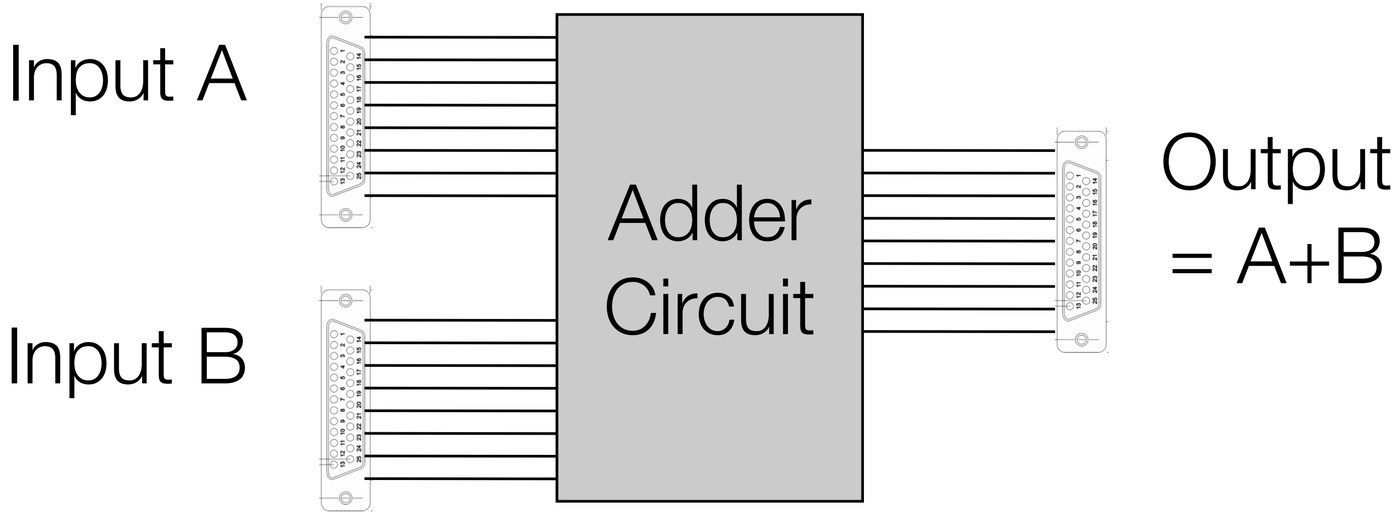
Part 10_2: Addition and Negative Numbers

The Concept of Addition

A simple example - two inputs, add their values and get a value as a result

What is difficult about addition? Don't we do this every day?

We're dealing with values so why does representation make a difference?



Addition in Decimal

You should be able to see what is going the examples above

It's interesting to consider what might happen when you get an overflow

Naively exceeding the capacity of a bus may mean that an incorrect message could be transmitted or that an additional message must be transmitted to account for the overflow

```
   1 2        3 8        6 6
+  3 4      + 1 4      + 6 6
   4 6        5 2      1 3 2
                1        1 1
```

Carry

Overflow: The result generated is too large to fit on the "bus"

Addition in Binary

The overflow problem doesn't go away when you switch base

In fact the example of overflow shown above emphasise the need to account for the problem in the context of computer systems

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

Overflow: The result generated is too large to fit on the "bus"

We will return to the issue of overflow in later lectures

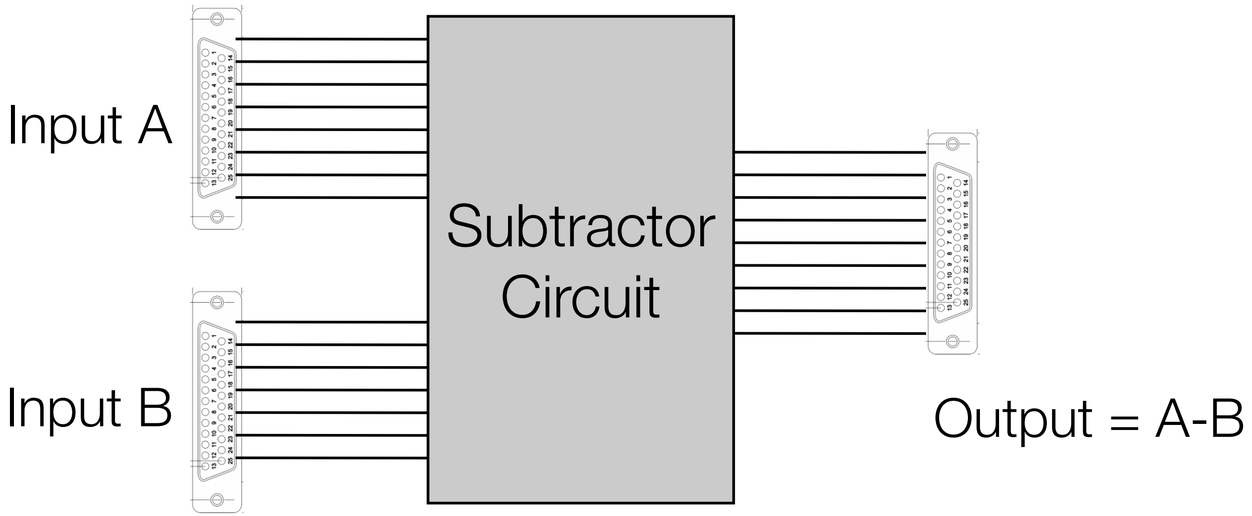
Subtraction as Addition

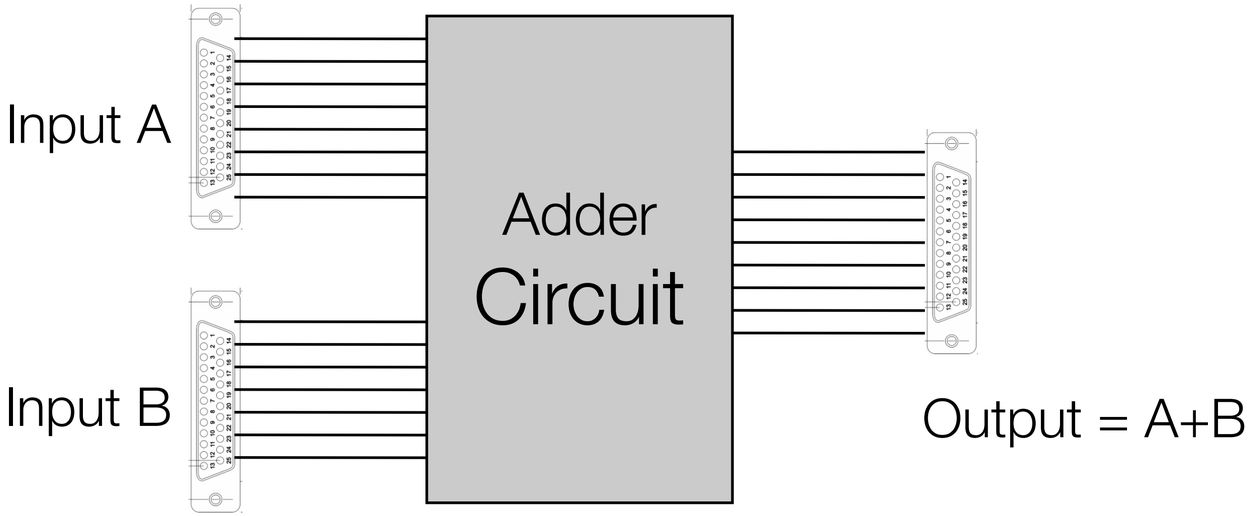
Two viable options for achieving subtraction

- Design a dedicated subtractor circuit
- Create a representation for negative numbers (A and B can be negative) and extend the adder

You will soon design and construct logic circuits for addition and subtraction

- For now it is enough to appreciate that we can use representation to permit subtraction





Signed Magnitude Representation

The MSB is an indicator, i.e., indicates it is negative

The value of the remaining bits are calculated as before

1101_{2SM}:	**1**	**1**	**0**	**1**
Position:	**negative?**	**2^2**	**2^1**	**2^0**
=	**negative?**	**4**	**2**	**1**
Value:	**-1 x**	**(1 x 4 +**	**0 x 2 +**	**1 x 1)**
=	**-5_{10}**			

Two's Complement Representation

The most common signed integer representation in modern computing

The MSB has the same value as in the binary positional representation, i.e.,2^{N-1}, but it is negative, i.e., -2^{N-1}

1101_{2TC}:	**1**	**1**	**0**	**1**
Position:	**-2^3**	**2^2**	**2^1**	**2^0**
=	**-8**	**4**	**2**	**1**
Value:	**(1 x -8 +**	**1 x 4 +**	**0 x 2 +**	**1 x 1)**
=	**-3_{10}**			

Two's Complement Properties

Can represent more negative numbers than positive

The range is asymmetric

$1000_{2TC} = -8$

$0111_{2TC} = +7$

For N bits, the range is -2^{N-1} to $(2^{N-1}-1)$

Unique zero

Signed magnitude has both +0 and –0

Forming Two's Complement

To form the bit pattern for a negative number in two's complement:

Take the representation of the positive number

Invert the bits, ensuring there are enough bits for the MSB to be the sign

Add 1, ignoring any overflow

What is the two's complement bit pattern for –9?

$+9_{10} = 1001_2$
Ensure enough bits: 00001001_2
Invert: 11110110_2
Add 1: 11110111_2
$-9_{10} = 11110111_{2TC}$

Addition in Two's Complement

Addition is almost the same as we've seen previously

Just as before but ignore any overflow

```
 00001001 = +9      00001101 = +13     00000101 = +5
+11110111 = -9     +11110111 = -9     +11110111 = -9
 00000000 =  0      00000100 =  4      11111100 = -4
 11111111           11111111                111
```

Part 11_2: Fractional Numbers

Fixed Point Representation

Add columns to denote the fractional parts of a value

10.11_2	**=**	**1**	**0**	**1**	**1**
Position:		**2^1**	**2^0**	**2^{-1}**	**2^{-2}**
	=	**2**	**1**	**0.5**	**0.25**
Value	**=**	**(1 x 2 +**	**0 x 1 +**	**1 x 0.5 +**	**1 x 0.25)**
	=	**2.75_{10}**			

A Problem with Representation?

How could we represent very large and very small numbers using the representations we've covered so far?

The mass of an electron is 0.000000000000000000000000000009g

The mass of the sun is 2000000000000000000000000000000000g

We would need an incredibly large bus to accurately represent these values in a computer system

Floating Point Representation

Scientific notation allows small and large values to be written succinctly

The speed of light is 3x108 m/s ≈ 299,792,458 m/s

Floating point uses the same principles as scientific notation

Express numbers as (mantissa x $10^{exponent}$)

The mantissa represents the detail of the value to a certain precision

The exponent represents the magnitude of the value

IEEE Floating Point

Floating point representation trades precision for range

- Extending range at the expense of reducing precision

IEEE Standard 754 is widely used and specifies levels of binary precision

- Single precision (using 32 bits)
- Double precision (using 64 bits)
- Quad precision (using 128 bits)

For example, single precision uses 1 bit for the sign (s), 8 bits for the exponent (e) and 23 bits for the mantissa (m)

Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 3 - Digital Logic

Session Outline

Basic logic functions, expressions & truth tables

Logic gates, circuits, & finding the truth table

Simplifying logic expressions using Boolean algebra and Karnaugh maps

Common combinatorial, i.e., no memory, logic circuits

Sequential, i.e., with memory, logic circuits

Three state logic

Physical implementations

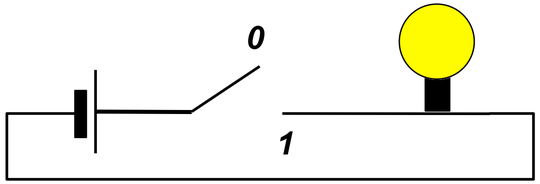
Logic Gates, Circuits and Truth Tables

Basic Logic Functions

Consider a simple switch: it has two positions:

OFF = 0, ON = 1

We can use the switch to allow current to flow to light a bulb

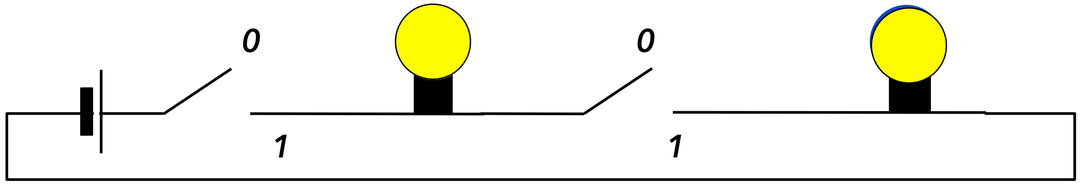


Consider f to indicate if the bulb is on, by denoting the switch to be A we can write:

$$f = A$$

Switches in Series (AND)

Two switches can be used in series



The bulb is on if switch A AND switch B are both on

$f = A \,.\, B$

. is used to denote AND in Boolean algebra (also $\wedge$)

Truth Tables - AND

A truth table defines the output values of a function in relation to ALL possible input variables.

Each line in the truth table represents a UNIQUE value of the input variables.

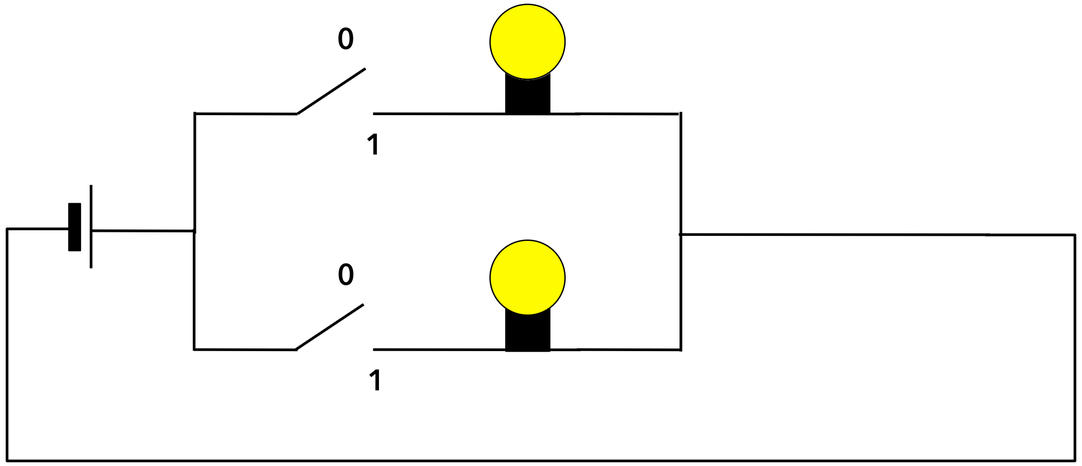
The truth table for two switches in series f = A . B is shown below

Input Variables		Output Variables
A	B	f = A . B
0	0	0
0	1	0
1	0	0
1	1	1

2^n entries
(n is the number of input variables)

Switches in Parallel (OR)

We can also consider two switches in parallel



The bulb is on if switch A OR switch B are on

$f = A + B$

+ is used to denote OR in Boolean algebra (also $\vee$)

Truth Tables - OR

We've seen the truth table for AND

Important that you appreciate the difference between the function of AND and OR

Inspecting their truth tables the most sensible way to do this

Input Variables		Output Variables
A	B	f = A . B
0	0	0
0	1	0
1	0	0
1	1	1

Input Variables		Output Variables
A	B	f = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Dots and Plus? Why?

AND written as . (or ∧)

OR written as + (or V)

Potential for confusion, since when we communicate otherwise "+" can mean "and" but in Boolean algebra "+" is OR

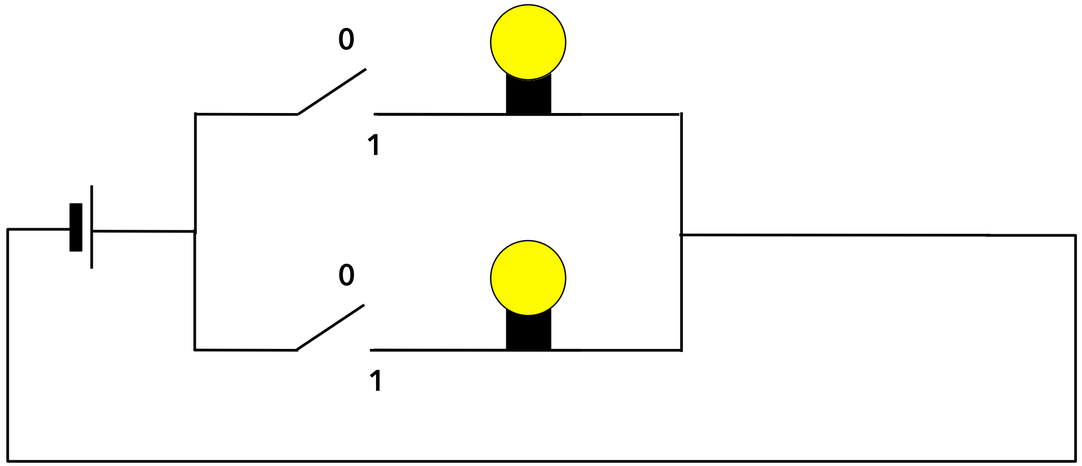
Notation is this way around because of mathematical properties

Algebra of real numbers		***Boolean algebra***
Multiply x by 1 (x.1), get x	⟷	**AND x with 1 (x.1), get x**
Add x to 0 (x+0), get x	⟷	**OR x with 0 (x+0), get x**

An Alternative View of OR

If (switch A is NOT on) AND (switch B is NOT on) then the bulb is NOT on

This is $\overline{f} = \overline{A} . \overline{B}$ where the bar over each letter is negation in Boolean algebra



We know that $\overline{f} = \overline{A + B}$

This means that $f = A + B$ (negate both sides)

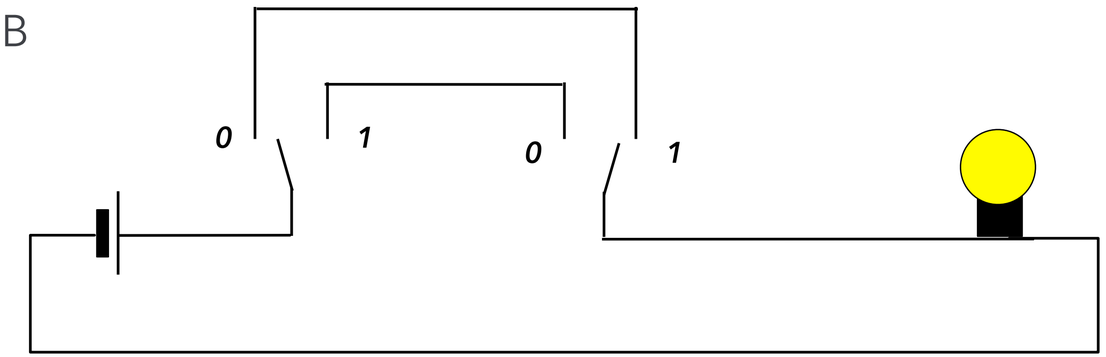
Therefore $\overline{f} = \overline{A} . \overline{B}$ (this is de Morgan's law)

Exclusive OR (EX-OR)

One more important logical function

The bulb is on when only one switch is on

$f = A \oplus B$



Input Variables		Output Variables
A	B	$f = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$\oplus$ is used to denote EX-OR in Boolean Algebra

EX-OR is often referred to as an Equivalence relation (i.e. the output is zero if both inputs are equivalent)

Functions of Two Binary Variables

16 (= 2^4) functions can be found between two binary variables:

We can interpret each of these functions algebraically

$f_0 = 0$

$f_1 = A \,.\, B$

$f_2 = A \,.\, \overline{B}$

$f_3 = A$

$f_{13} = \overline{A \,.\, \overline{B}}$

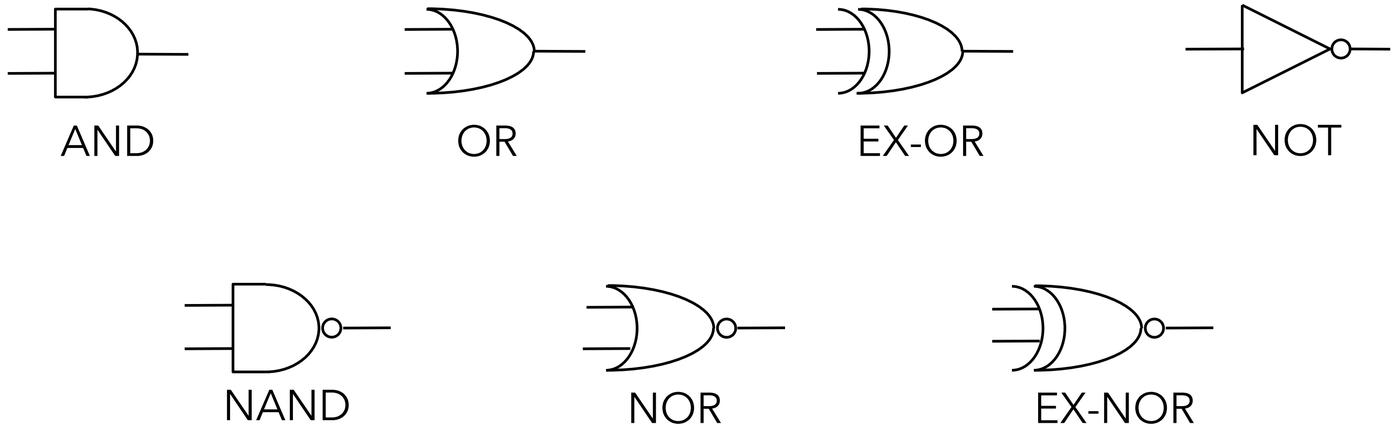
A	B	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Electronic Logic Gates

There are seven electronic logic gates whose function you should understand

You will definitely be familiar with AND, OR, EX-OR and NOT

You should try to construct the truth tables for NAND, NOR and EX-NOR

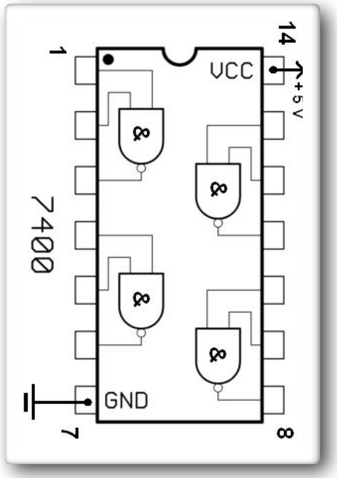


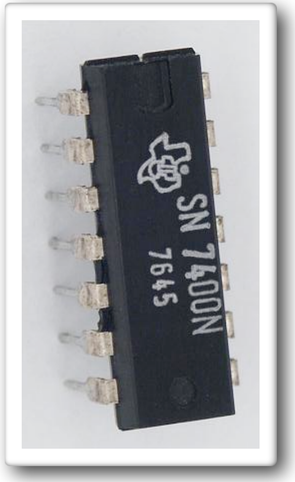
Generations of Computer Systems

Generation	Approximate Dates	Technology	Typical Speed (operations per second)
1	1946–1957	Vacuum tube	40,000
2	1958–1964	Transistor	200,000
3	1965–1971	Small and medium scale integration	1,000,000
4	1972–1977	Large scale integration	10,000,000
5	1978–1991	Very large scale integration	100,000,000
6	1991-	Ultra large scale integration	1,000,000,000

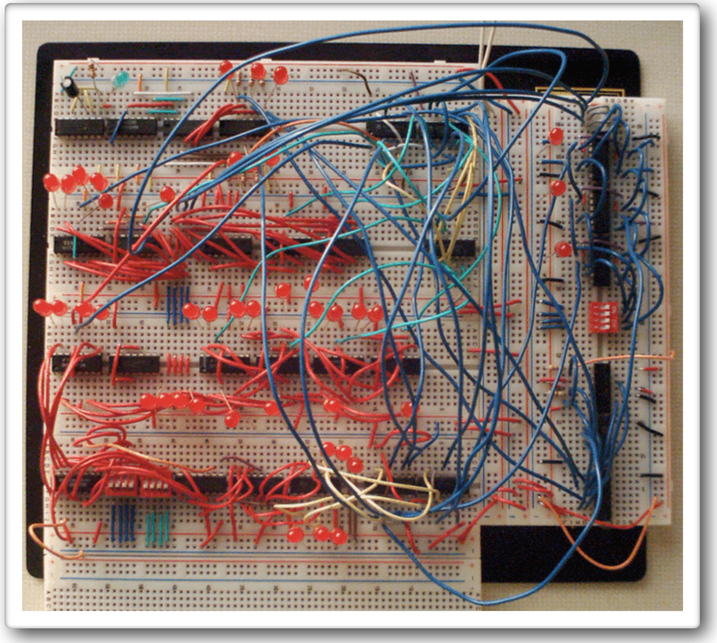
Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

Physical Packaging of Logic Gates



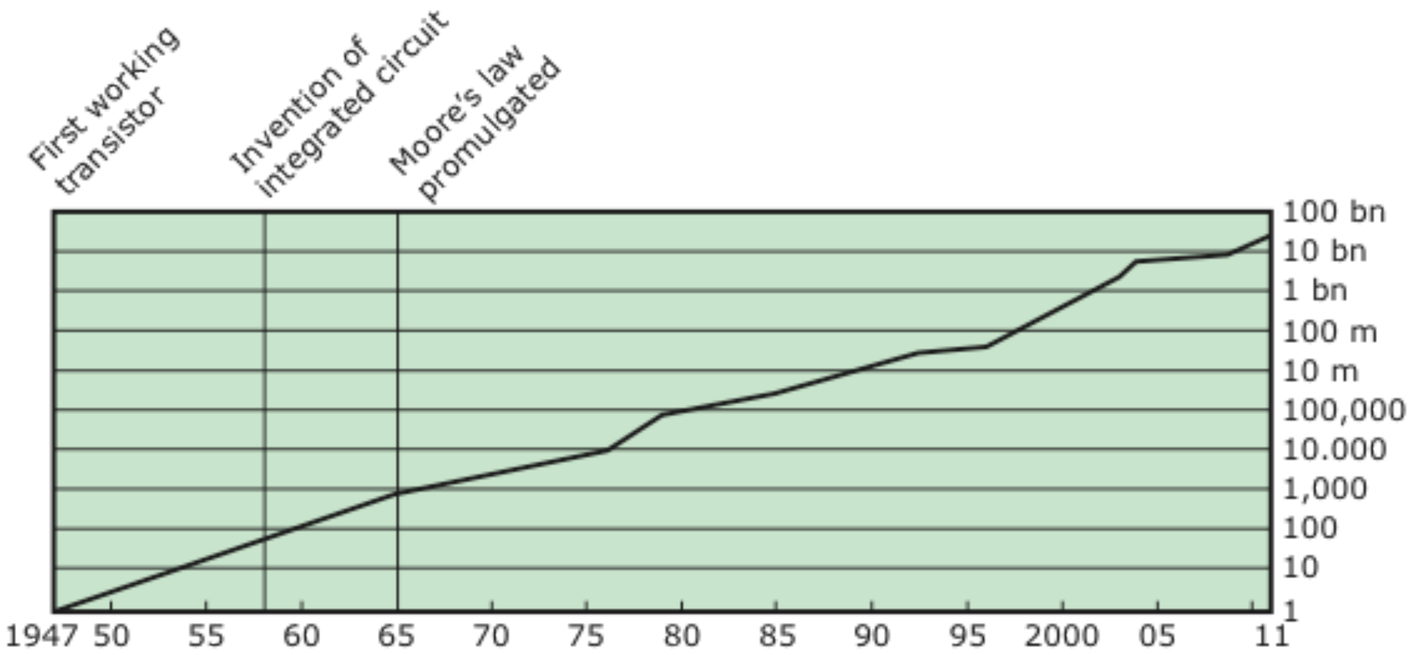


A 7400 chip: 4 NAND gates



A 4-bit two register computer built entirely from 74-series chips

Moore's Law



Reproduced from: W.Stallings, "Computer Organization and Architecture", 9th Edition, Pearson, 2013

Moore's Law

In 1965 Gordon Moore, co-founder of Intel, predicted the number of transistors on a chip would double every 12-18 months

The consequences of Moore's law are profound:

- The cost of computer logic and memory circuitry has fallen at a dramatic rate
- As logic and memory is placed closer together on more densely packed chips, the electrical path is shortened, increasing operating speed
- Computers becoming smaller, making them more convenient in a variety of environments, especially embedded systems
- Reduction in power and cooling requirements
- Interconnections on ICs are more reliable than solder connections so having fewer off-chip connections increases reliability

Combinatorial Logic Circuits

A logic circuit whose output is a logical function of its input

Consider the EX-OR function

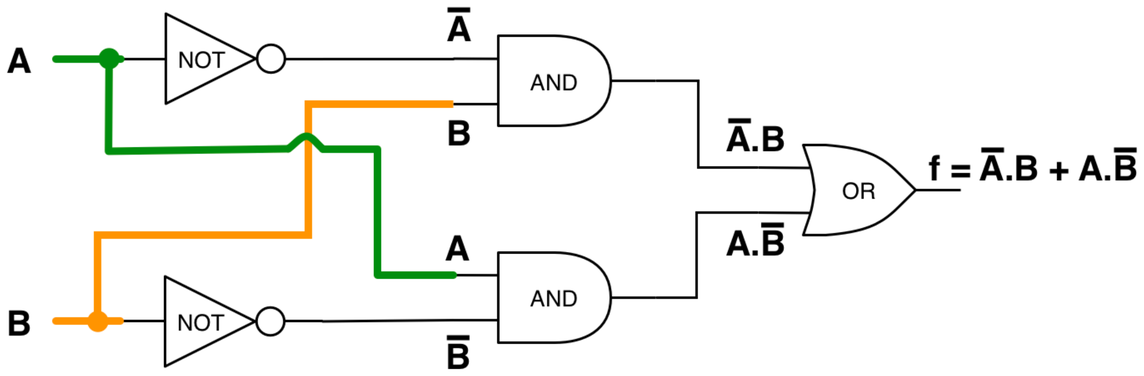
$$f = A \oplus B$$

From the truth table we can write:

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

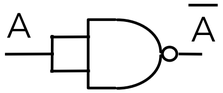
The logic circuit to the right implements the EX-OR function

Input Variables		Output Variables
A	B	$f = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

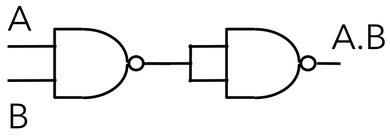


NAND Gates

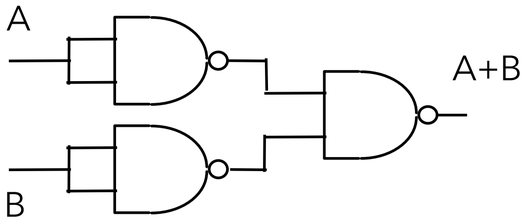
NOT $f = \overline{A} = \overline{A.A}$



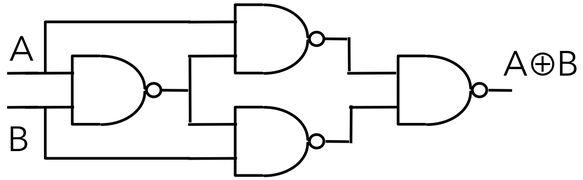
AND $f = A.B = \overline{\overline{A.B}}$



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



EX-OR $f = \overline{A}.B + A.\overline{B}$

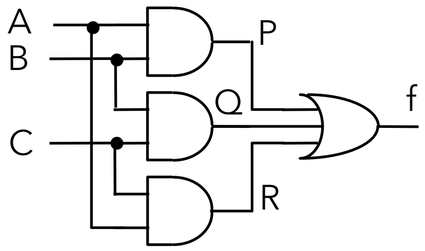


Can you work out why this is correct?

Diagram ? Algebra? Maps?

Logic Circuit to Truth Table

We can construct a truth table for a given logic circuit



Consider intermediate signals P, Q, R

P = A.B, Q = B.C, R = A.C

The output is f = A.B + B.C + A.C

A	B	C	P	Q	R	f
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	1
1	0	0	0	0	0	0
1	0	1	0	0	1	1
1	1	0	1	0	0	1
1	1	1	1	1	1	1

This is an important skill and will help you understand logic circuits, truth tables and how we can manipulate logical expressions

You should try creating logic circuits and deriving their truth tables

Simplifying Logical Expressions

Logic expressions can be simplified to reduce complexity

- Reduces number of gates required for their implementation

Two mains ways to simplify logical expressions

- Boolean algebra (the manipulation of an algebraic representation)
- Karnaugh Maps (simplifying the “sum of products” form graphically)

Using either method the truth table for the logical expression before and after minimisation must be identical

Simplifying Expressions Using Boolean Algebra

Circuit Equivalence

We have many concerns when implementing logic circuits

- Circuits with more gates are more expensive
- The gates available depends upon the physical implementation, e.g., we may only want use NAND gates or we may have some 4-input gates

When given a function we need to be able to create equivalent circuits to meet several design criteria

- Perform the designated function
- Use the types of gate available
- Minimise the number of gates used and hence cost

A Boolean function can have many different but equivalent Boolean expressions

Truth Table to Boolean Equation

The sum of products form can be obtained directly from a truth table

A	B	C	f	
0	0	0	1	$\bar{A}.\bar{B}.\bar{C}$
0	0	1	1	$\bar{A}.\bar{B}.C$
0	1	0	0	
0	1	1	0	
1	0	0	1	$A.\bar{B}.\bar{C}$
1	0	1	0	
1	1	0	1	$A.B.\bar{C}$
1	1	1	1	$A.B.C$

So function f is TRUE when any of these combinations are TRUE

$$f = \bar{A}.\bar{B}.\bar{C} + A.\bar{B}.\bar{C} + A.\bar{B}.\bar{C} + A.B.\bar{C} + A.B.C$$

We consider this to be a sum of products because an OR '+' is considered as a sum and an AND '.' as a product

Laws of Boolean Algebra

Boolean algebra obeys a convenient set of laws that permit simplification

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC) = ABC$	$(A + B) + C = A + (B + C) = A + B + C$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

Remembering the Laws of Boolean Algebra

Can generate OR form from AND form, or vice-versa:

Swap AND ⇔ OR ('.' ⇔ '+')

Swap 0 ⇔ 1

Most laws are intuitive but several are less intuitive

Distributive (AND form only)

Absorption

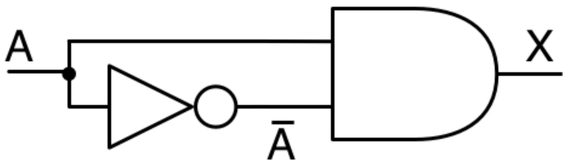
de-Morgan's

Checking the Inverse Law

Not sure if you've remembered one correctly? Check it with a truth table!

For example, we can construct a truth table for the inverse law

$$A.\overline{A} = 0$$



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

… this is what we expected

De Morgan's Law - Algebraically

To change the form of expression:

1. Break the negation bar
2. Change AND to OR and vice versa

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

EXAMPLES:

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

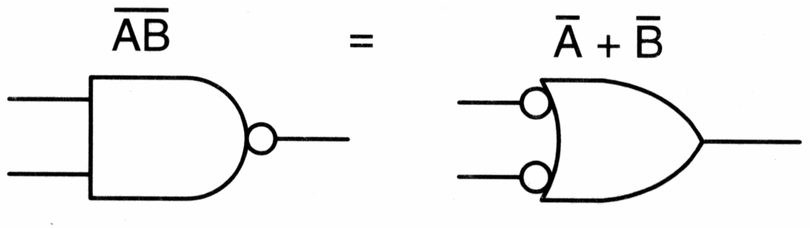
$$(\overline{A \,.\, C}) + D = (\overline{A} + \overline{C}) + D$$

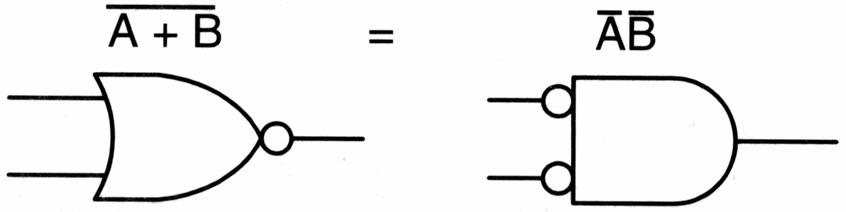
$$(\overline{A \,.\, B}) + (\overline{C \,.\, D}) = \overline{A} + \overline{B} + \overline{C} + \overline{D}$$

De Morgan's Law - Graphically

For many people it makes sense to think about De Morgan's law graphically

Consider the equivalence of the logic circuits involved





Simplification Using Boolean Algebra - Q1

First I will simplify an expression for you...

① $f = W + \overline{(\bar{X}.\bar{Y}) + (\bar{X}.\bar{Z})}$

② $= W + \overline{(\bar{X}.\bar{Y})}.\overline{(\bar{X}.\bar{Z})}$ de-Morgan OR form $\overline{A+B} = \bar{A}\bar{B}$, $A = (\bar{X}.\bar{Y})$

③ $= W + (\bar{\bar{X}}+\bar{\bar{Y}}).(\bar{\bar{X}}+\bar{\bar{Z}})$ de-Morgan AND form $\overline{AB} = \bar{A}+\bar{B}$, twice

④ $= W + (X+Y).(X+Z)$ remove double NOTs $\bar{\bar{A}} = A$

⑤ $= W + (X+Y)X + (X+Y)Z$ distributive OR form $A(B+C) = AB+AC$, $A = (X+Y)$

⑥ $= W + X.X + Y.X + X.Z + Y.Z$ distributive OR form again, twice: $A = X$, then $A = Z$

⑦ $= W + X.(X+Y+Z) + Y.Z$ distributive OR form, extended to 3 variables:

$AB + AC + AD = A(B+C+D)$

⑧ $= W + X + Y.Z$ absorption AND form $A(A+B) = A$

– now in sum of products form.

Simplification Using Boolean Algebra - Q2

Try simplifying this…

① $f = \overline{(X+Y)} \cdot \overline{(\bar{X}+Y)}$

Simplification Using Boolean Algebra - Q2

Try simplifying this…

①	$f = \overline{(X+Y)} \cdot \overline{(\bar{X}+Y)}$	
②	$= (\bar{X} \cdot \bar{Y}) \cdot \overline{(\bar{X}+Y)}$	de-Morgan OR form $\overline{A+B} = \bar{A}\bar{B}$
③	$= (\bar{X} \cdot \bar{Y}) \cdot (\bar{\bar{X}} \cdot \bar{Y})$	de-Morgan AND form $\overline{AB} = \bar{A}+\bar{B}$, $A = \bar{X}$, $B = Y$
④	$= (\bar{X} \cdot \bar{Y}) \cdot (X \cdot \bar{Y})$	remove double NOT $\bar{\bar{A}} = A$
⑤	$= \bar{X} \cdot \bar{Y} \cdot X \cdot \bar{Y}$	associative AND form $(AB)C = ABC$
⑥	$= \bar{X} \cdot X \cdot \bar{Y} \cdot \bar{Y}$	commutative AND form $AB = BA$
⑦	$= (\bar{X} \cdot X) \cdot (\bar{Y} \cdot \bar{Y})$	associative AND form, extended to 4 variables $ABCD = (AB)(CD)$
⑧	$= 0 \cdot \bar{Y}$	inverse AND $A\bar{A} = 0$ and idempotent AND $AA = A$
⑨	$= 0$	null law AND $0A = 0$

Simplification Using Boolean Algebra - Q3

Now try…

① $f = X + \bar{Y} + \bar{X}.Y + (X+\bar{Y}).\bar{X}.Y$

Simplification Using Boolean Algebra - Q3

Now try…

1	$f = X + \bar{Y} + \bar{X}.Y + (X+\bar{Y}).\bar{X}.Y$	
2	$= X + \bar{Y} + \bar{X}.Y + \bar{X}.Y.X + \bar{X}.Y.\bar{Y}$	distributive OR form $A(B+C) = AB + AC$ where $A = \bar{X}.Y$, $B = X$, $C = \bar{Y}$
3	$= X + \bar{Y} + \bar{X}.Y + 0 + 0$	inverse law AND form $A\bar{A} = 0$, twice
4	$= X + (\bar{Y}+\bar{X})(\bar{Y}+Y)$	distributive law AND form $A + BC = (A+B)(A+C)$ where $A = \bar{Y}$, $B = \bar{X}$, $C = Y$
5	$= X + (\bar{Y}+\bar{X}).1$	inverse law OR form $A + \bar{A} = 1$
6	$= X + (\bar{Y}+\bar{X})$	identity law AND form $1A = A$
7	$= X + \bar{Y} + \bar{X}$	associative law OR form $(A+B)+C = A+B+C$
8	$= X + \bar{X} + \bar{Y}$	commutative law OR form $A+B = B+A$
9	$= 1 + \bar{Y}$	inverse law OR form $A + \bar{A} = 1$
10	$= 1$	null law OR form $1 + A = 1$

Simplification Using Boolean Algebra - Try These

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

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

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

Simplifying Expressions Using Karnaugh Maps

Sum of Products

Recall that the sum of products form can be obtained from a truth table

The sum of products is often used in the simplification of logical expressions

$$f = \bar{A}.\bar{B}.\bar{C} + \bar{A}.\bar{B}.C + A.\bar{B}.\bar{C} + A.B.\bar{C} + A.B.C$$

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

Simplifying a Sum of Products

Sum of products form can be simplified by looking for terms that differ by only one variable and its complement

$$\overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C = \overline{A}.\overline{B}.(\overline{C} + C) \quad \text{Distributive law}$$

$$= \overline{A}.\overline{B} \quad \text{Inverse law}$$

When using algebraic techniques can be difficult to tell whether an equation is in simplest form or to see the next step

Karnaugh maps show unambiguously when a Boolean expression is in its simplest form

Karnaugh Maps (K-maps)

2D truth tables

See the simplest Boolean expression of a few variables from the location of 1s on the map

A three variable function can be represented by a 4 x 2 rectangle or a 2 x 4 rectangle

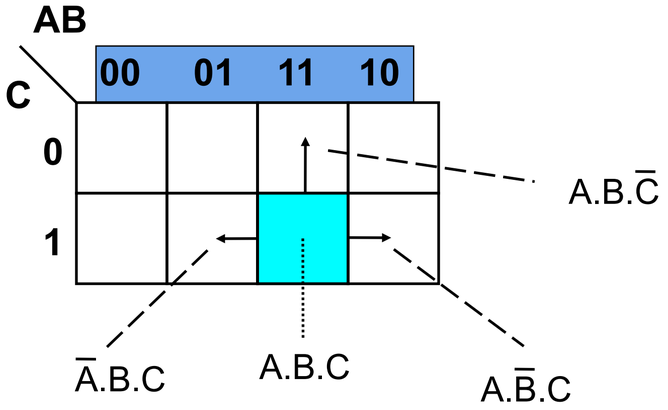
C \ AB	00	01	11	10
0	$\overline{A}\overline{B}\overline{C}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\overline{C}$
1	$\overline{A}\overline{B}C$	$\overline{A}BC$	ABC	$A\overline{B}C$

BC \ A	0	1
00		
01		
11		
10		

in both maps stands for $A.B.\overline{C}$

Adjacency in a Karnaugh Map

Adjacent squares differ by exactly one variable



NOTE : Numbering on the edge of the Karnaugh Map - "Gray coded" for inventor Frank Gray

Example Karnaugh Map

Simplification the graphical way

Important to try a lot of examples to develop your simplification skills

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

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

Karnaugh Map Grouping

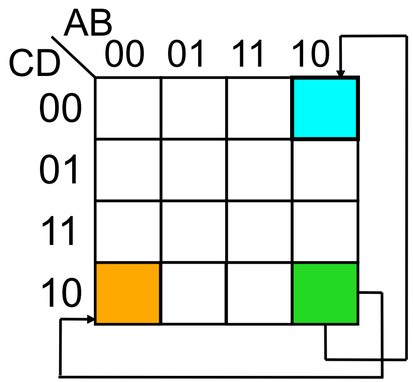
Wrap-around

Left most-column is adjacent to right most-column, and top row is adjacent to bottom row

Groupings may overlap

Minimum logic expression is obtained on minimum number of groupings

Number of elements in the group must be a power of two, i.e., 1,2,4,8 etc...



Karnaugh Maps - Example 1

Essentially we're looking for the smallest number of groupings that capture a logical expression

C \ AB	00	01	11	10
0	1	1	0	1
1	1	0	0	1

$\Rightarrow \quad f = \overline{B} + \overline{A}.\overline{C}$

C \ AB	00	01	11	10
0	1	1	0	1
1	1	0	0	1

$\Rightarrow \quad f = \overline{A}.\overline{B} + A.\overline{B} + \overline{A}.\overline{C}$

Correct, but not minimal. The top result was simpler - don't forget to wrap around!

Karnaugh Maps - Example 2

It's possible to have more than one equivalent expressions

Overlap in groups permits equivalence

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

$\bar{A}.\bar{B}.\bar{C}$

$\bar{A}.\bar{C}.D$

$\bar{A}.B.D$

$B.C.D$

Two possibilities : $f = \bar{A}.\bar{B}.\bar{C} + \bar{A}.B.D + B.C.D$

or

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

Can result in more than one equivalent logic expression

Karnaugh Maps - Example 3

Sometimes it's not possible to minimise a logical expressions

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

Cannot get any groupings ⇒ no expression minimisation possible

This is actually an EX-OR function between the 4 input variables:

$$f = (A \oplus B) \oplus (C \oplus D)$$

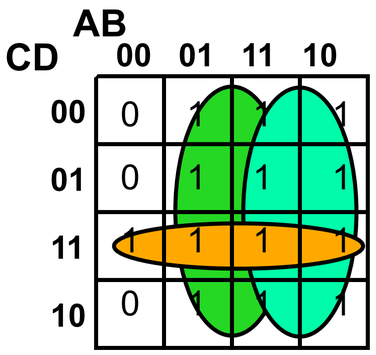
Minimising Complex Logical Expressions

Seemingly complex logical expressions often simplify to expressions that you could implement using very few integrated circuits

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

Construct Truth Table:

A	B	C	D	$\overline{B}.\overline{C}$	$\overline{B}.\overline{D}$	$\overline{(\overline{B}.\overline{C})+(\overline{B}.\overline{D})}$	f
0	0	0	0	1	1	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	1	0	0
0	0	1	1	0	0	1	1
0	1	0	0	0	0	1	1
0	1	0	1	0	0	1	1
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	1
1	0	0	0	1	1	0	1
1	0	0	1	1	0	0	1
1	0	1	0	0	1	0	1
1	0	1	1	0	0	1	1
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	1
1	1	1	1	0	0	1	1



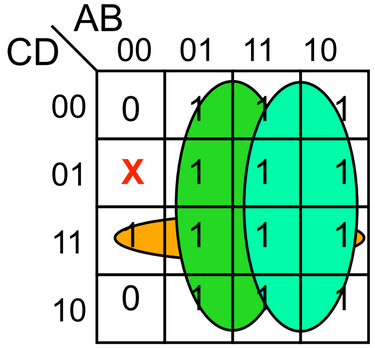
$$f = B + A + C.D$$

Don’t Care Conditions

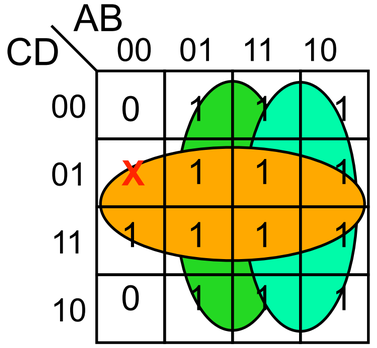
Sometimes a certain combination of inputs can’t happen, or if it does, we don’t care what the output is

An “**X**” is used to denote these “don’t care” conditions, which may be assumed to be either 1 or 0

Don’t care condition can allow us to create a simpler logic expression



$f = A + B + C.D$



$f = A + B + D$

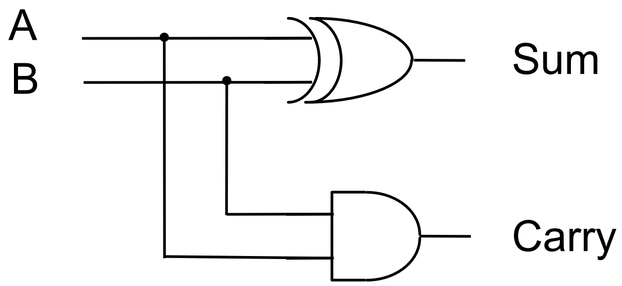
Combinatorial Logic Circuits

1-bit Half-Adder

The 1-bit half-adder performs the addition of two bits

Can be extended to form a 1-bit full-adder

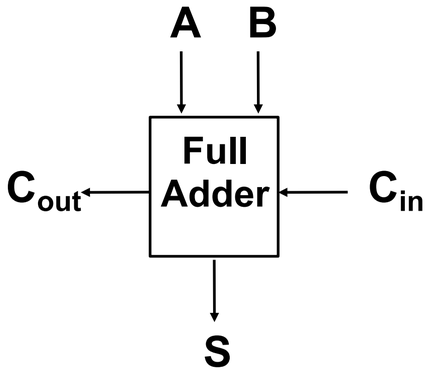
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



1-bit Full-Adder

Performs the addition of three bits

Two significant bits and a previous carry



Inputs - A,B, C_{in} (each one bit)

Outputs - S, C_{out} (each one bit)

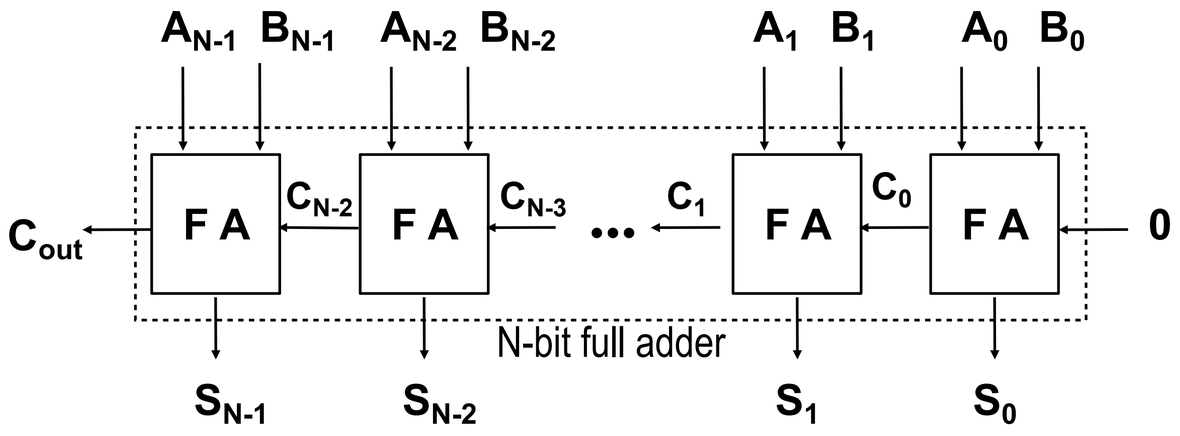
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Homework: Design a circuit of a one-bit full adder

N-bit Full-Adder

A combination of what we've already seen

The carry output from one 1-bit full-adder is the carry input to the next 1-bit full-adder in all but the first case



Adder to Adder/Subtractor

To convert an adder into an adder/subtractor we simply add a control input Z such that:

$Z = 0 \Rightarrow S = A + B$

$Z = 1 \Rightarrow S = A - B$

Z	B	$Z \oplus B$	
0	0	0	$=B$
0	1	1	
1	0	1	$=\overline{B}$
1	1	0	

Remember that $A - B = A + (-B)$

We can calculate (-B) using two's complement

1. Invert the N-bit binary number B by finding $Z \oplus B$
2. Add 1 (Carry In)

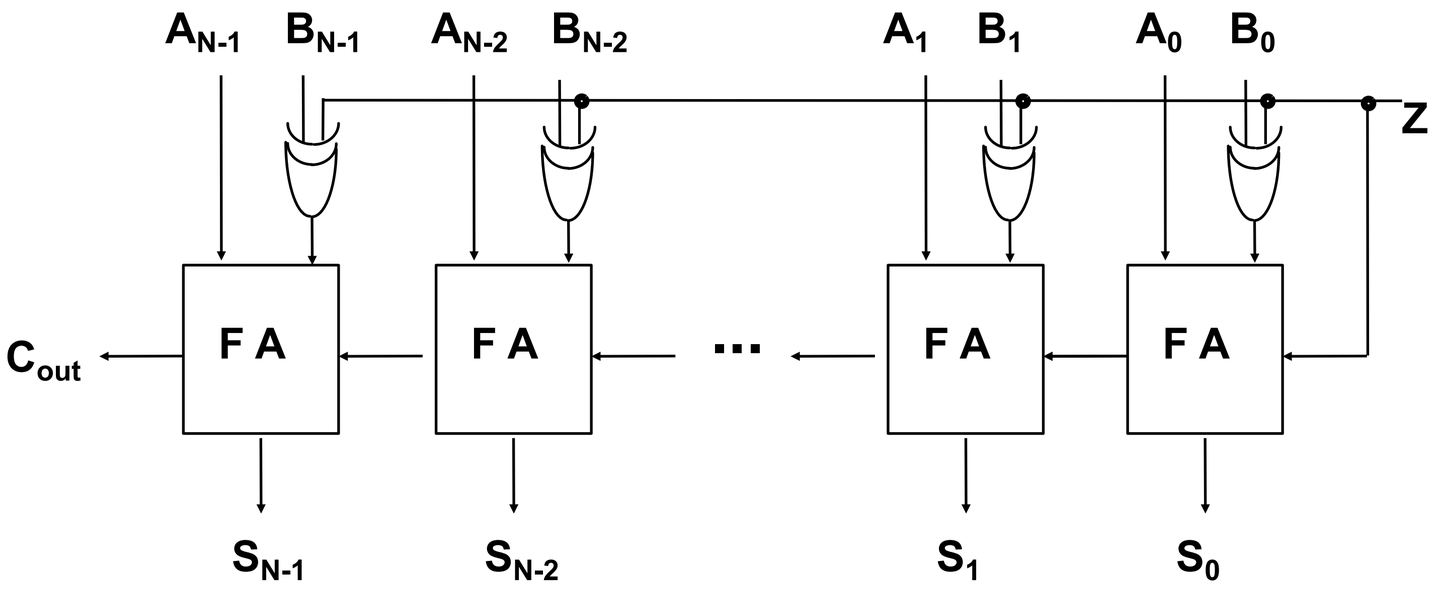
N-bit Adder/Subtractor

A reasonably complex combinatorial circuit

One of the more interesting circuits we've seen with respect to its function

Try to see how this works for yourself!

What does the final carry indicate?



Z = 0 ⇒ Adder

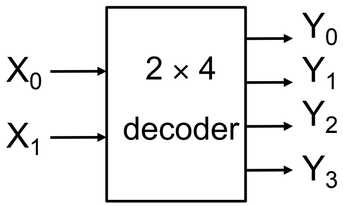
Z = 1 ⇒ Subtractor

Decoders

Decoder has, e.g., binary input pins, and one output pin per possible input state

Activate one output pin corresponding to current input state

Consider decoding 2 inputs to 4 unique outputs



X_0	X_1	Y_0	Y_1	Y_2	Y_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Note that the two inputs select which output is active
The system is active low, i.e., active = 0, in this case

Decoders are often used to address unique memory locations in a microprocessor system

Active High or Active Low?

In some circuit applications, outputs and inputs have “active” and “inactive” states.

Important that inputs and outputs conform to the same standard

Active high

0 = inactive, 1 = active

Active low

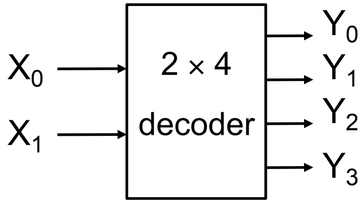
0 = active, 1 = inactive

Pin sometimes labelled with a bar, e.g., $\overline{\text{Enable}}$

Active High Decoder

Consider a 2 to 4 line decoder with active high output

Observe the difference between in the 2 to 4 line decoder with active low output we saw previously



X_0	X_1	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

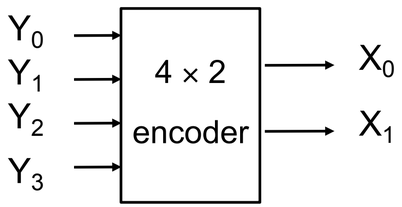
Encoders

Encode a set of inputs into a defined representation on the outputs (note that this is the reverse of a decoder)

- Multiple input pins, only one of which should be active at a time
- Smaller number of output pins
- Circuit puts code, e.g., binary, of currently active input onto output pins

Consider encoding 4 inputs to 2 outputs

- Does the encoder shown below have active high or active low inputs?



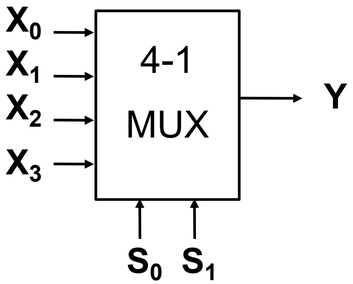
Y_0	Y_1	Y_2	Y_3	X_1	X_0
0	1	1	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1

Encoders are often used as simple input circuits - though this is limited because you couldn't have 2+ keys held down

Multiplexers (MUX)

Multiplexer - Output is a selected input.

Consider a 4-1 multiplexer (four inputs to one output)

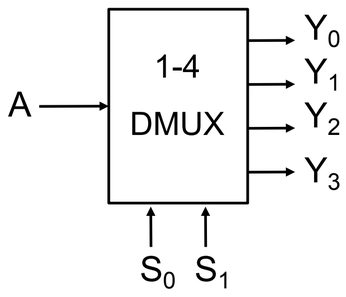


S_0	S_1	Y
0	0	X_0
0	1	X_1
1	0	X_2
1	1	X_3

$$Y = X_0.\bar{S}_0.\bar{S}_1 + X_1\,\bar{S}_0.S_1 + X_2\,S_0.\bar{S}_1 + X_3.S_0.S_1$$

De-Multiplexers (DE-MUX)

Allow an input to appear on any one of the outputs (note that this is the reverse of a multiplexer)



S_0	S_1	Y_0	Y_1	Y_2	Y_3
0	0	A	1	1	1
0	1	1	A	1	1
1	0	1	1	A	1
1	1	1	1	1	A

A 1 represents inactive: therefore active low

$$Y_0 = A + \overline{S}_0.S_1 + S_0.\overline{S}_1 + S_0.S_1$$

$$= A + \overline{S}_0.S_1 + S_0.(\overline{S}_1 + S_1)$$

$$= A + \overline{S}_0.S_1 + S_0$$

$$= A + S_1 + S_0$$

Homework exercise: what are expressions for Y_1, Y_2, Y_3?

Common Applications of Multiplexers

- Source selection control
  - Home stereo, e.g., send iPod, CD or radio to speakers – note that this is analogue not digital
- Share one communication line between multiple senders
  - Requires both MUX and DE-MUX
- Parallel to serial conversion
  - Parallel input on X, clock signal on S, serial output on Y
- Circuit that can be configured to produce any truth table relationship between S inputs and Y output
  - Set up the truth table required on X inputs

Common Applications of De-Multiplexers

Share one communication line between multiple senders

- Requires both MUX and DE-MUX

Serial to parallel conversion

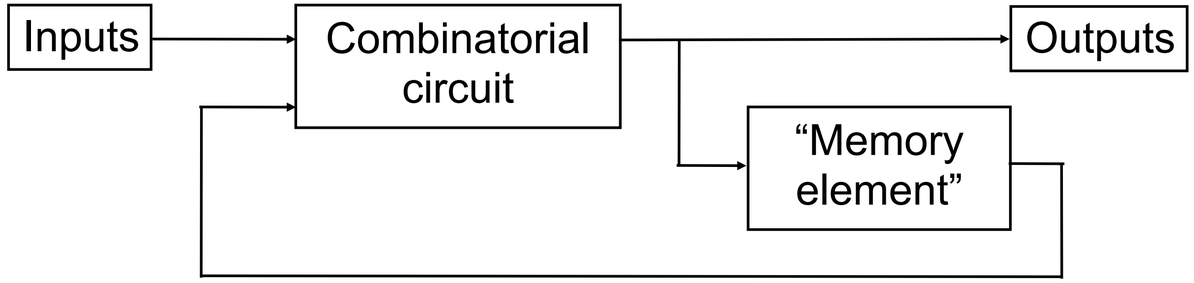
A control for multiple lights

- In a gambling machine you might connect a processor to A and S and connect Y outputs to lights, such that the processor runs in a rapid loop addressing each light sequentially

Sequential Logic Circuits

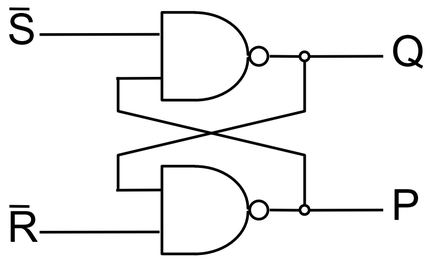
Sequential Logic

A logic circuit whose **outputs** are logical functions of its **inputs** and its **current state**



Flip-Flops

A circuit whose outputs are fed back to inputs



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

$\overline{S}$ and $\overline{R}$ are active low **SET** and **RESET** inputs

Consider:	$\overline{S} = 1, \overline{R} = 0$	$\Rightarrow$	$P = 1, Q = 0$
then if	$\overline{S} = 1, \overline{R} = 1$	$\Rightarrow$	$P = 1, Q = 0$
and then	$\overline{S} = 0, \overline{R} = 1$	$\Rightarrow$	$P = 0, Q = 1$
then	$\overline{S} = 1, \overline{R} = 1$	$\Rightarrow$	$P = 0, Q = 1$

Flip-Flop Operation

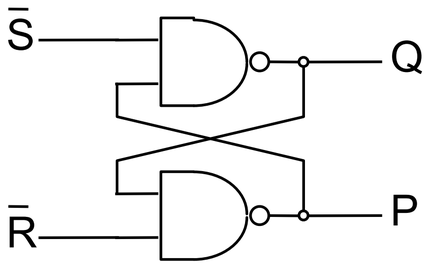
Q is Set (to one) when $\overline{S} = 0$ (and $\overline{R} = 1$)

Q is Reset (to zero) when $\overline{R} = 0$ (and $\overline{S} = 1$)

If $\overline{S} = \overline{R} = 1$ then Q does not change

We can capture this behaviour in a truth table

$\overline{S}$	$\overline{R}$	Q	P	
0	0	X	X	← **hazard condition**
0	1	1	0	
1	0	0	1	
1	1	**no change**		



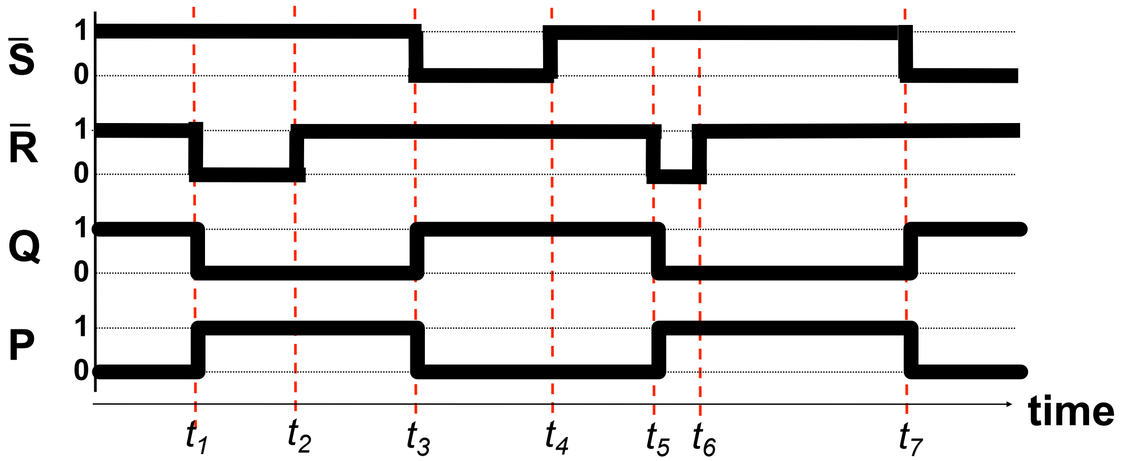
The flip-flop is able to **STORE** a value when both $\overline{S} = \overline{R} = 1$

Flip-Flop Timing Diagram

We can also visualise the operation of a flip-flop in a timing diagram

Flip-flop output is not just dependent on the input, as in combinatorial logic circuits, but also on its previous state

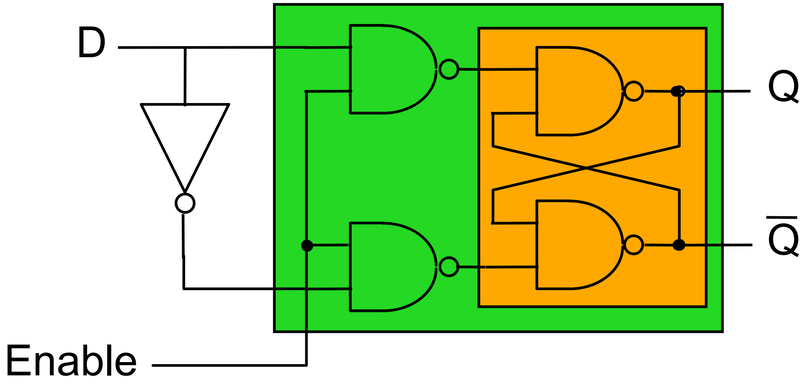
You should step through the timing diagram to gain an understand of how such a circuit provide useful function



D-Type Latch

Using a modified form of the flip-flop circuit we've seen it is possible to design the D-type latch

Essentially a 1-bit memory circuit



Enable	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

Output can change only when the Enable line is high, i.e., enable triggers circuit operation.

D-type latch is a 1-bit memory with the Enable terminal as the 'WRITE' command.

D-Type Latch Truth Table

The truth table for the d-type latch can be expressed in several different, yet equivalent, ways

Observe that "don't care" conditions can be use to reduce complexity but further abstraction makes things even simpler

Enable	D	Q	$\overline{Q}$
0	x	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

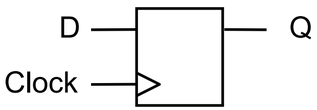
Enable	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

Enable	Q	$\overline{Q}$
0	Q	$\overline{Q}$
1	D	$\overline{D}$

Clocked Flip-Flops

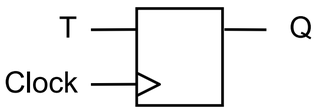
Flip-flops whose output changes only on the rising edge of the clock input

D-type ("delay")



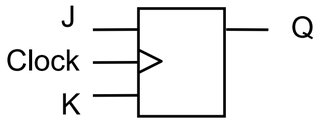
D_n	Clk	Q
0	↑	0
1	↑	1

T-type ("toggle")



T	Clk	Q
0	↑	Q
1	↑	$\overline{Q}$

JK-type



J	K	Clk	Q
0	0	↑	Q
0	1	↑	0
1	0	↑	1
1	1	↑	$\overline{Q}$

Clock

Clocked flip-flops often have additional inputs to allow the SET and RESET of Q

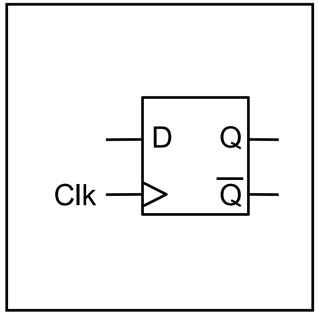
An exceptionally useful building blocks in other sequential logic circuits

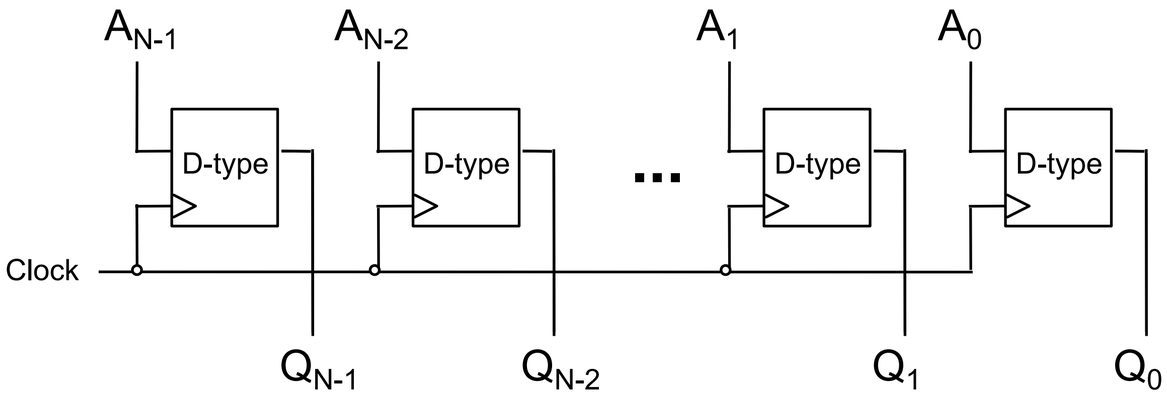
N-bit Register

A multi-bit memory

N-bit number $A_{N-1}\ A_{N-2}...A_1A_0$ is stored in the register on a low to high transition of the clock

Stored number appears on the outputs $Q_{N-1}...Q_0$

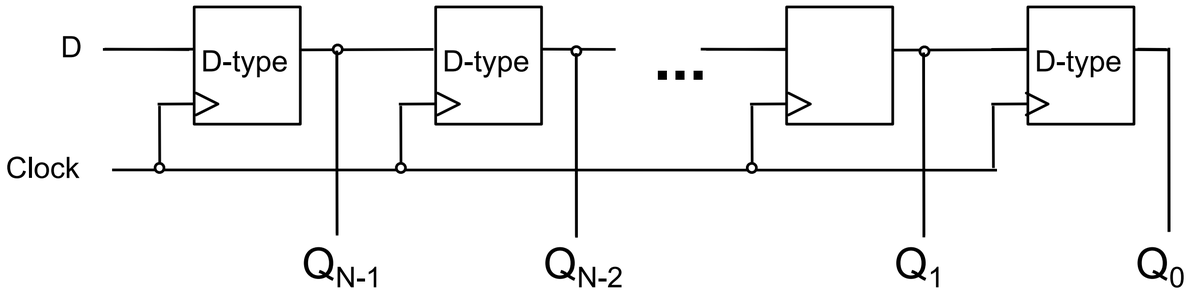




N-bit Shift Register

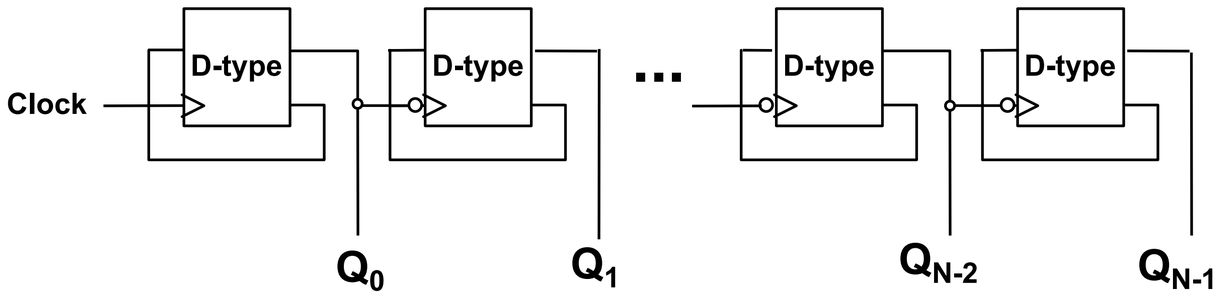
A register that stores and shifts a number one bit at a time

For serial to parallel conversion we make D=serial bit 0, trigger clock, then D=bit 1, clock… etc… after N bits, read out all bits in parallel from Q



When a clock transition occurs, each bit in the register will be shifted one place to the right

N-bit Counter



Output: 0000
1000
0100
1100
0010
etc...

Note the circles (on the inputs to the D-types, except the left-most one) to indicate an inverter.

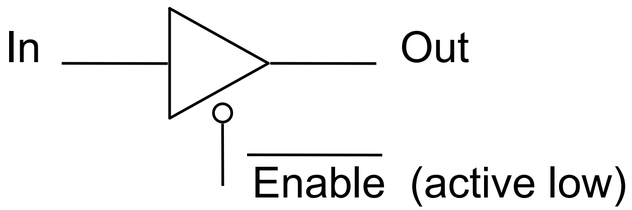
Each flip-flop (except the left-most one) is clocked only on a transition of high ⇒ low (falling clock edge) of the previous output

Three State Logic

Three-state Logic

The logic components considered so far have two possible logic states, i.e., 1 or 0

There is a further gate, termed a three-state buffer, whose output can be placed in a third state, known as UNCONNECTED



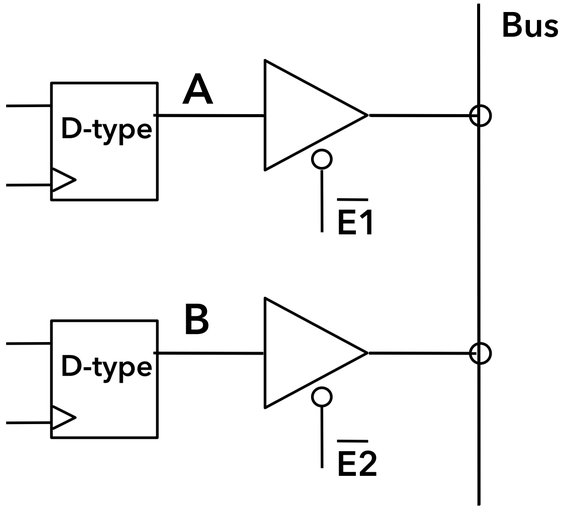
When $\overline{\text{Enable}}$ is high, the input is disconnected from the output

When $\overline{\text{Enable}}$ is low, the input is connected to the output

Three-state Buses

Can use three state buffers to allow different sources of data onto a common BUS

For example, consider a 1-bit wide bus



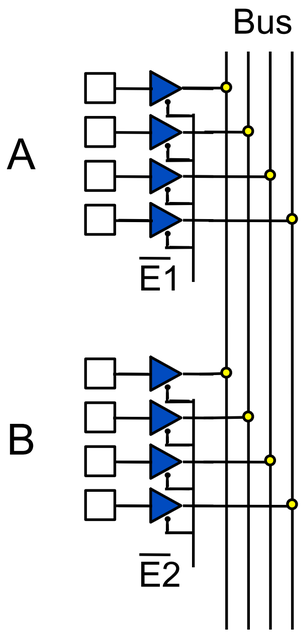
When $\overline{E1}$ = 0,
A will be placed on the Bus

When $\overline{E2}$ = 0,
B will be placed on the Bus

N.B. Should not have $\overline{E1}$=0 and $\overline{E2}$ = 0 at the same time!

Multi-bit Bus

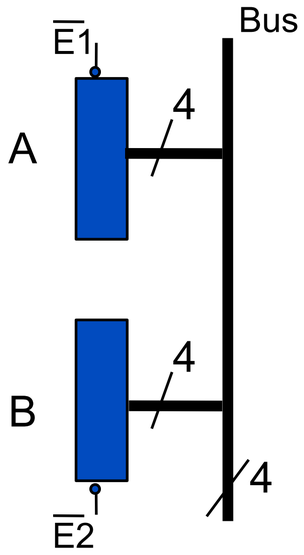
We can consider a 4-bit bus



When $\overline{E1}$ = 0,
A will be placed on the Bus

When $\overline{E2}$ = 0,
B will be placed on the Bus

Or, can represent this more concisely:

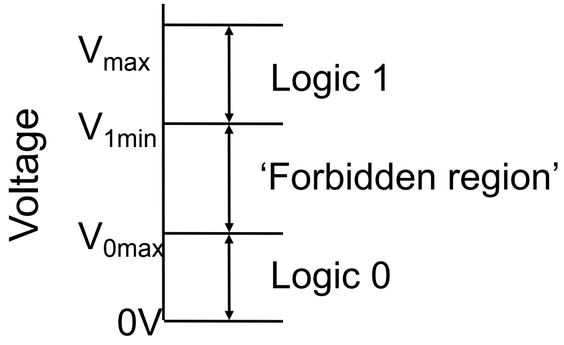


Physical Implementations

Properties of Logic Gates

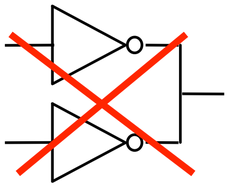
Several basic points to remember when it come to the construction of logic circuits, particularly in lab sessions

Logic values by Voltage level:



Typically TTL(Transistor-Transistor Logic),
$V_{max} = 5$, $V_{1min} = 2.8$, $V_{0max} = 0.8$

Do NOT connect outputs together except three-state buffers



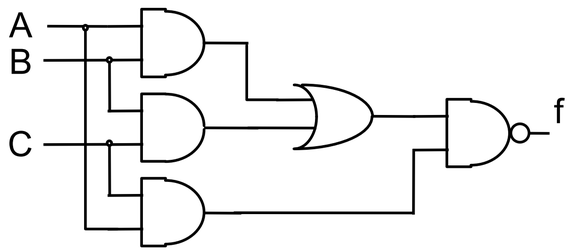
…because you could be connecting 1 to 0, causing a short circuit

Propagation Delay

Each logic gate has a propagation delay, typically nano-seconds, i.e., 1×10^{-9}s, or less

Limits the speed at which logic circuits work

Propagation delay can be reduced by putting logic gates close together, e.g., on the same integrated circuit



Gate	Prop. Delay(ns)
NOT	1.0
NAND	1.2
AND	1.7
OR	1.2
NOR	1.5

(values for illustration only)

What is the maximum propagation delay in this circuit?

Answer = 1.7 (AND) + 1.2 (OR) + 1.2 (NAND) = 4.1ns

Logic Integrated Circuits (ICs)

Elementary logic gates and functions can be obtained in small ICs, e.g., the 7400 series, though programmable devices allow much larger circuits to be created inside a single chip

There are many types of programmable logic device

- Programmable Array Logic (PAL) - The first popular programmable device was one-time programmable

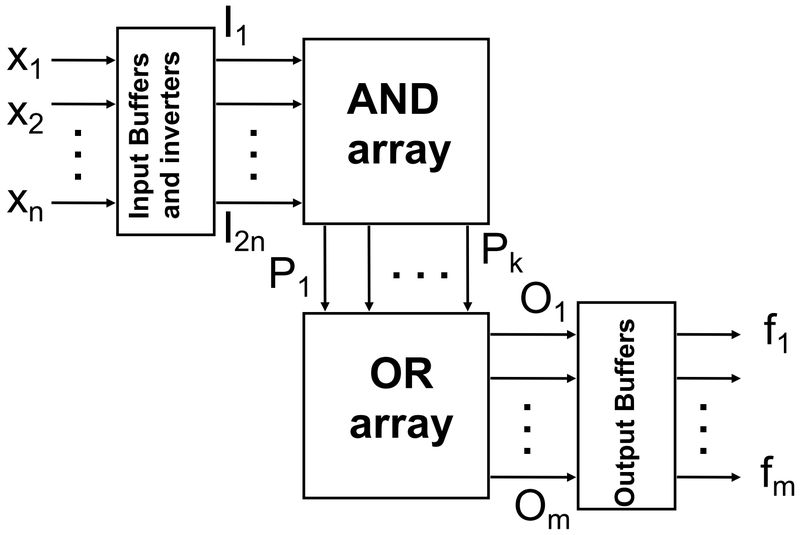
- Programmable Logic Array (PLA) - Device contains an AND array, which feeds an OR array, providing a sum of products in hardware

- Field Programmable Gate Array (FPGA) - One of several modern possibilities, which can contain millions of gates, i.e., enough for an entire processor

PLA Example

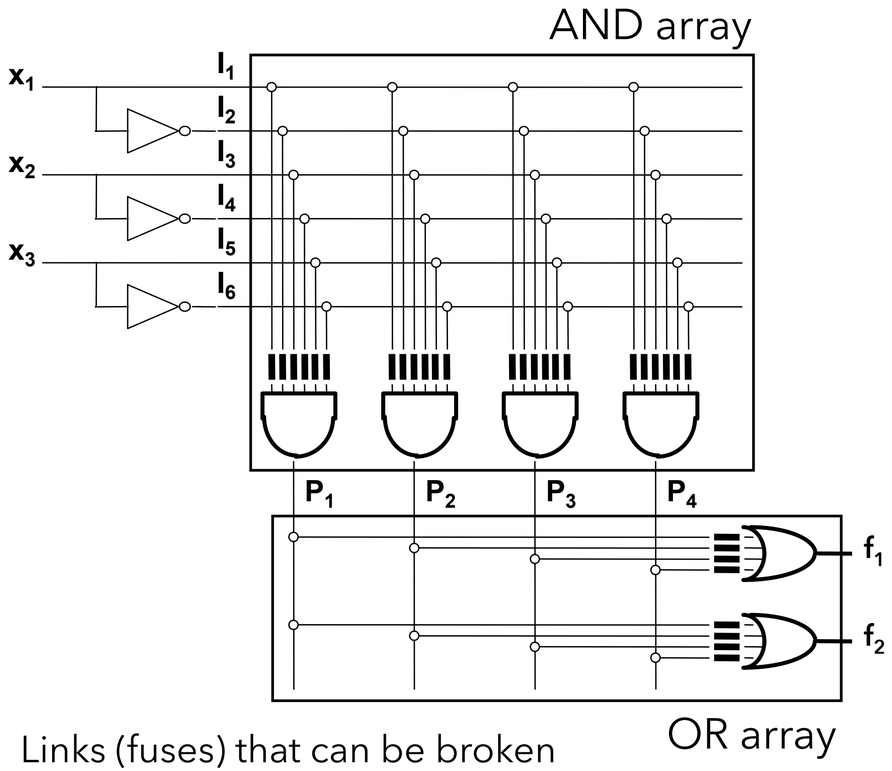
We can perform an AND function on a combination of any inputs, and their inverses, followed by an OR

Essentially any sum-of-products combinatorial function



PLA Organisation

A generic structure and destructible links provide ease of use

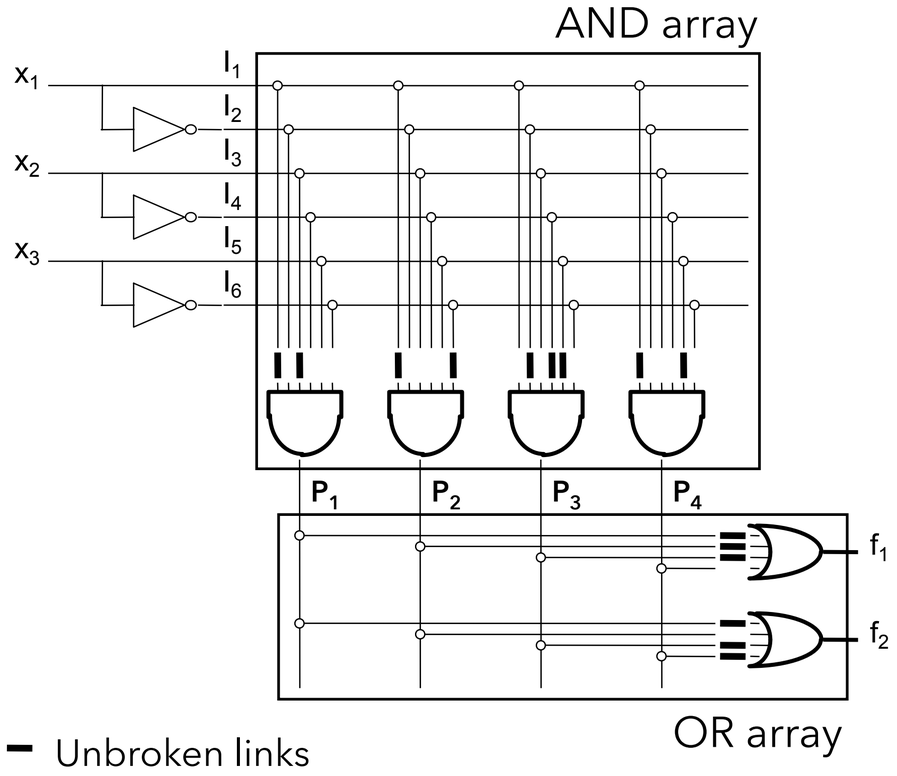


- Links (fuses) that can be broken

PLA Function Implementation

An example logic function implemented in a PLA is shown below

Observe how the unbroken links correspond to the implemented function



$P_1 = X_1 . X_2$

$P_2 = X_1 . X_3$

$P_3 = X_1 . X_2 . X_3$

$P_4 = X_1 . X_3$

$f_1 = P_1 + P_2 + P_3$

$f_2 = P_1 + P_3 + P_4$

Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 4 - Assembler

Session Outline

Microprocessor Fundamentals

Register Transfer Language

Assembly Language

Subroutines and Stacks

Addressing Modes

Microprocessor Fundamentals

Central Processing Unit (CPU)

Controls and performs instructions

Modern CPUs commonly housed on a silicon chip known as a microprocessor

Microprocessor CPU implementations dominate alternatives

CPUs are incredibly complex in their implementation but are usually viewed as having two key components

Arithmetic Logic Unit (ALU) - Performs mathematical and logical operations

Control Unit (CU) - Decodes program instructions and handles logistics for the execution of decoded instructions

Fundamentals of CPU Operation

- CPU continuously performs instruction cycle
  - Computational instructions are retrieved from memory, decoded to form recognisable operations and executed to impact the current state of a CPU
  - Commonly known as the fetch-execute cycle or the fetch-decode-execute cycle
- Instruction cycle takes place over several CPU clock cycles
  - Recall the significance of clocks in sequential logic circuits
- Fetch-decode-execute cycle relies on interaction of several CPU components, including ALU and CU

Fetch-Decode-Execute Components

Arithmetic Logic Unit (ALU) - Performs mathematical and logical operations

Control Unit (CU) - Decodes program instructions and handles logistics for the execution of decoded instructions

Program Counter (PC) - Tracks the memory address of the next instruction to be executed

Instruction Register (IR) - Contains most recent instruction fetched

Memory Address Register (MAR) - Contains address of the region of memory to be read or written, i.e., location of data to be accessed

Memory Data Register (MDR) - Contains data fetched from memory or data ready to be written to memory

Fetch-Decode-Execute Cycle

The specifics of an instruction cycle will vary with microprocessor may vary

Fetch

- Instruction retrieved from memory location held by PC
- Retrieved instruction stored in IR
- PC incremented to point to next instruction in memory

Decode

- Retrieved instruction / operation code / opcode decoded
- Read effective address to establish opcode type

Execute

- CU signals functional CPU components
- May result in changes to data registers, PC, ALU, I/O, etc...

I'm Confused... Where's the Assembler?

Understanding something about microprocessor operation and their instruction cycles provides an ideal context for us to appreciate programming in an assembly language

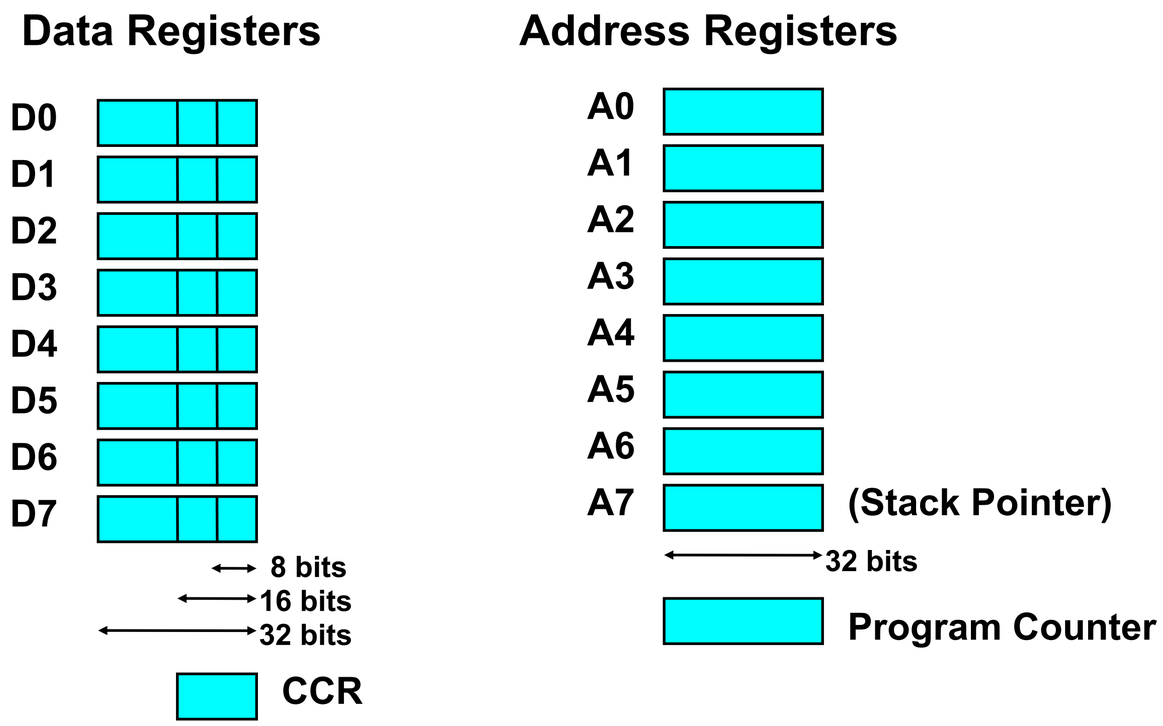
We will revisit many of these issues in subsequent topics

Of course, we can't just start writing assembler without some idea of what we can and can't do

How can we think about programming in assembly?

A Worked Example - The 68008 Architecture

To appreciate the general, we will consider a specific example with common characteristics



Programmer’s Model of 68008 CPU

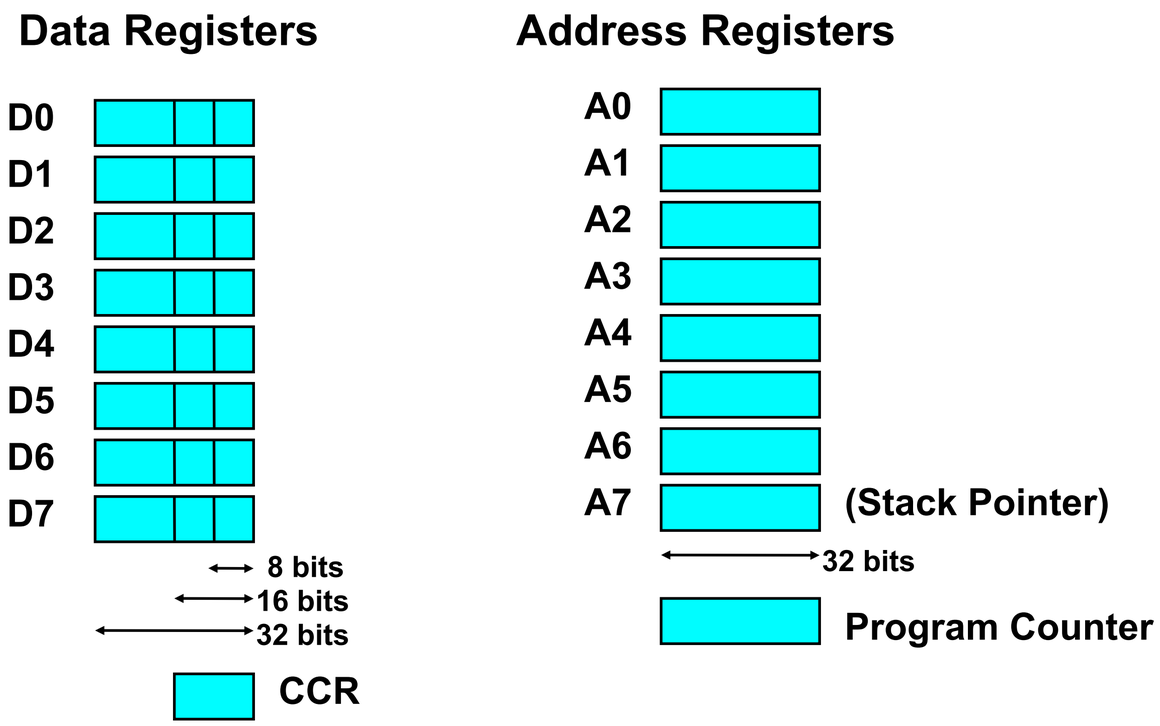
The programmer's model is a commonly used abstraction, invariably used by assembler programmers, of the internal architecture of a processor

68008 processor has identical instruction set to the 68000 processor, but has smaller external buses

- Internal registers are 32-bits wide
- Internal data buses are 16-bits wide
- 68008 has an 8-bit external data bus (16-bit on 68000)
- 68008 has 20-bit external address bus (24-bit for 68000)

A 68008 Programmer's Model

Understanding available registers is the cornerstone of a useful programmer's model



Data Registers

D0 - D7, 32 bit registers, store frequently used values / intermediate results

ON CHIP

Strictly speaking we would only need one register on chip - the advantage of many data registers is that fewer references to external memory are required

Registers can be treated as long, word or byte

Long - 32-bits

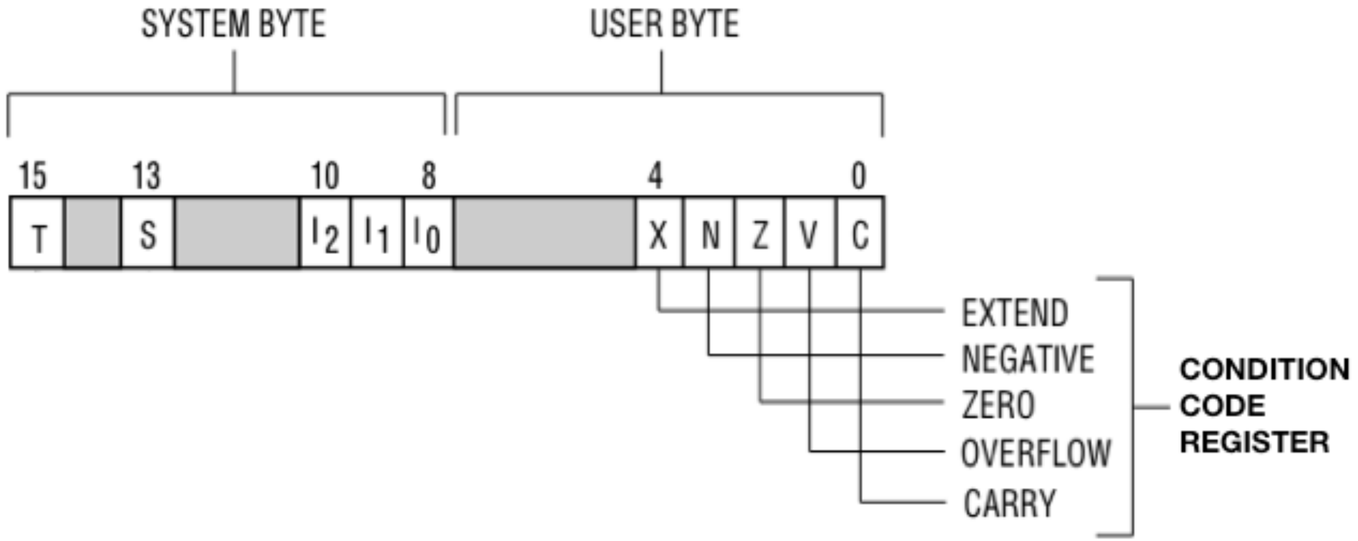
Word - 16-bits (lowest 16 bits)

Byte - 8-bits (lowest 8 bits)

Status Register

16 bit - Consists of two 8-bit registers

Various status bits that are set or reset upon certain conditions arising in the arithmetic and logic unit (ALU)



Address Registers

A0 - A6

Used as POINTER REGISTERS in the calculation of operand addresses

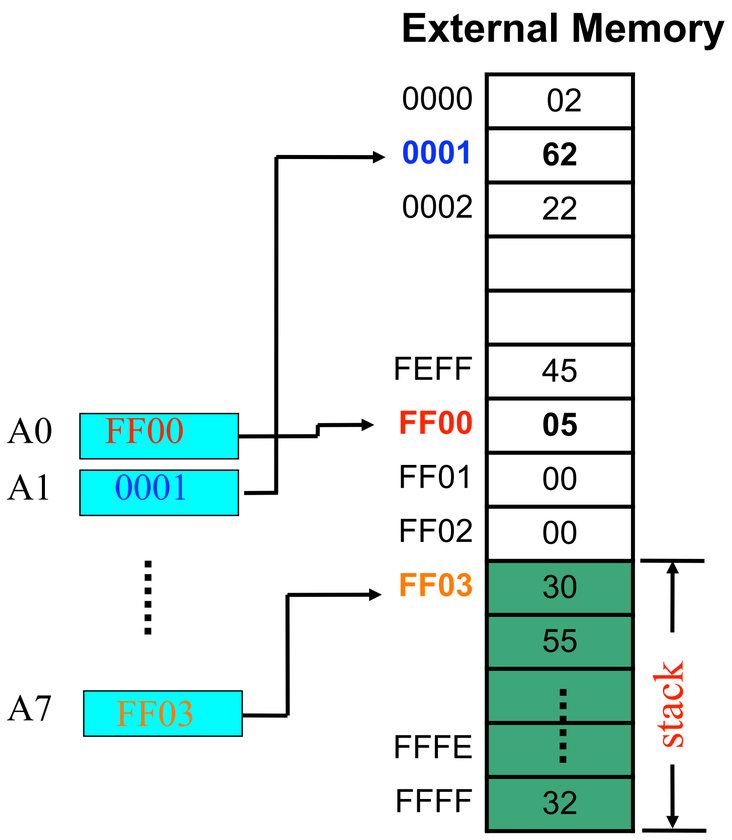
A7 - Additionally used by the processor as a system stack pointer to hold subroutine return addresses etc...

Operations on addresses do not alter status register Condition Code Register (CCR)

ALU has capacity to incur changes in status

A0-A7, Memory and Stack

The diagram below illustrates the relationship between address registers, external memory, i.e., not data registers, and the stack



Stack Pointer

A7 - The stack pointer is used as a pointer into an area of memory called the system stack.

Points to the next free location

The stack is a Last In First Out (LIFO) structure

The stack provides temporary storage of essential processor state, e.g., return addresses and registers, during subroutine calls and interrupts

You will meet this use of the stack again in many different contexts

A0-A6 may also used by programmer as stack pointers for temporary storage of registers, e.g., arithmetic calculations

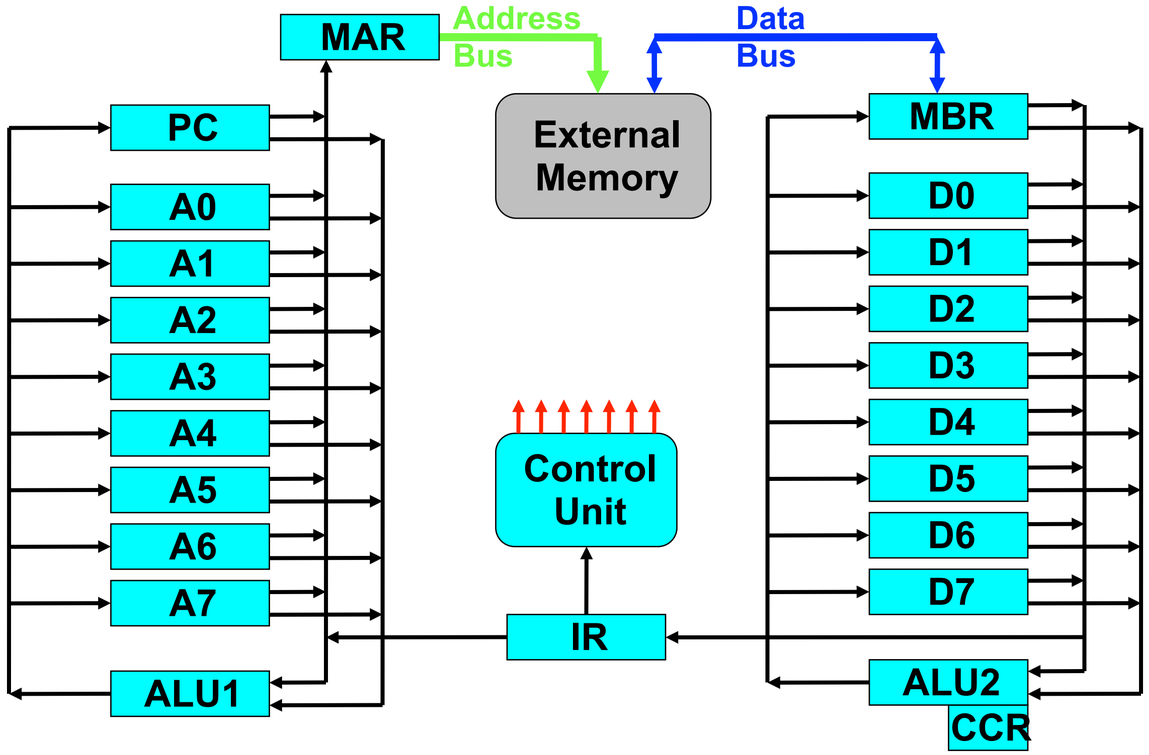
Program Counter (PC)

A 32-bit register that keeps track of the address at which the next instruction will be found

In simple terms, it points to the next instruction in memory

When the current instruction has been read, the PC is incremented to point to the next instruction

68008 Internal Architecture



MAR - Memory Address Register
CCR - Condition Code Register

MBR - Memory Buffer Register
IR - Instruction Register

Register Transfer Language

Register Transfer Language

Used to describe the operations of a microprocessor as it is executing instructions

For example, [MAR] ← [PC] means transfer contents of Program Counter to the Memory Address Register.

Computer's memory is called Main Store (MS)

The contents of memory location 12345 is written [MS(12345)]

Try not to confuse register transfer language with assembler instructions

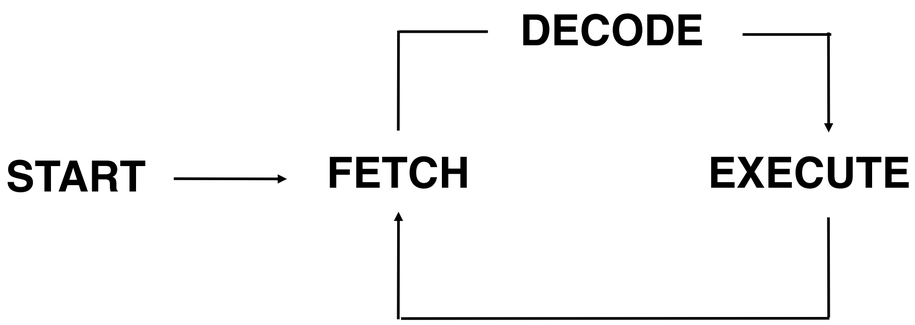
Instruction Cycle

We know quite a bit about the instruction cycle and the components involved in the process

To highlight the power of RTL we can use it represent stages of the instruction cycle

The RTL representation we will see makes no attempt to account for the pipelining of instructions

Pipelining is a common and simple method of speeding up the fetch-execute cycle - you might like to research pipelining in you own time



RTL - Instruction Fetching in Words

1. Contents of Program Counter transferred to MAR address buffers and the Program Counter is incremented

2. MBR loaded from external memory (R/$\overline{W}$ line set to Read)

3. Opcode transferred to Instruction Register from MBR

4. Instruction is decoded

RTL - Instruction Fetching

We can represent the fetch stage of the fetch-execute cycle in RTL

1. [MAR] ← [PC]
2. [PC] ← [PC] +1
3. [MBR] ← [MS([MAR])] ($\overline{R}$/W set to Read)
4. [IR] ← [MBR]
5. CU ← [IR(opcode)]

This part of the cycle is the same for each instruction

RTL - Fetch and Execute

We can represent complex actions, such as adding a constant byte to D0

1. [MAR] ← [PC]
2. [PC] ← [PC] +1
3. [MBR] ← [MS([MAR])] ($\overline{R}$/W set to Read)
4. [IR] ← [MBR]
5. CU ← [IR(opcode)]

…

RTL - Fetch and Execute

…

6. [MAR] ← [PC]

7. [PC] ← [PC] +1

8. [MBR] ← [MS([MAR])]

9. ALU ← [MBR] + D0

10. [D0] ← ALU

Assembly Language

C Programming

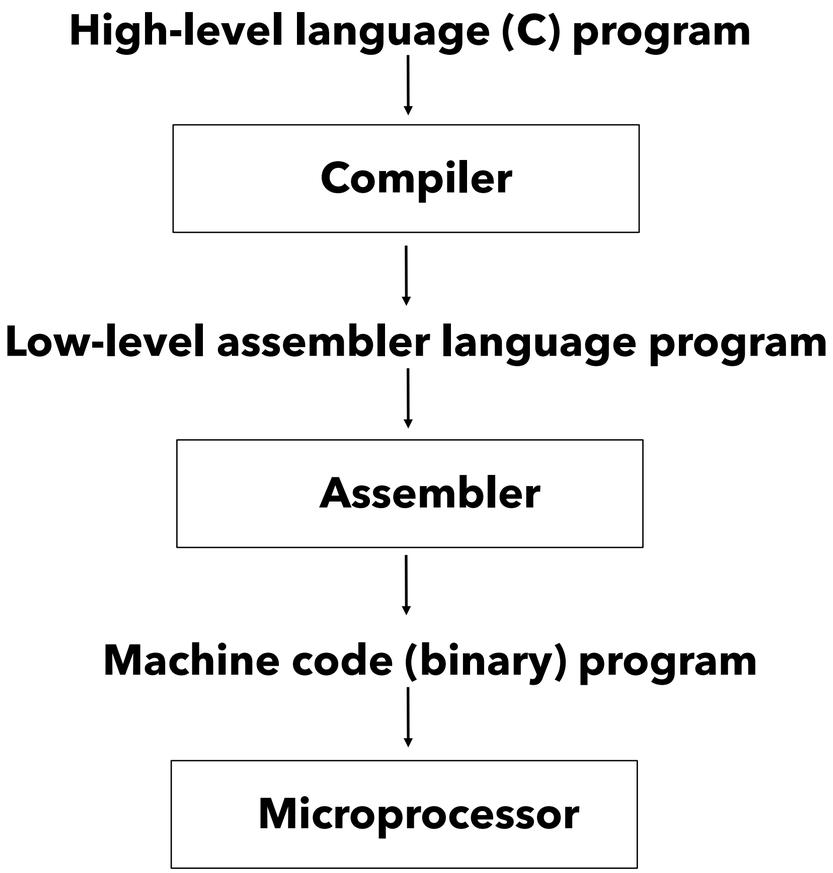
You can work at the hardware-software interface by using the C programming language

C is compiled to assembler - the low level instruction set of the microprocessor

Assembler programs are assembled in order to generate loadable binary

We concentrate on understanding the instruction set

Compiler, Assembler and Microprocessor



Assembly Language

Rather than programming in machine code, i.e., placing numbers in memory locations, we prefer to program at a slightly higher level, i.e., in an assembler language

Assembler language uses easily remembered mnemonics for each instruction, e.g., MOVE D0, D1

Assembler language also allows memory locations and constants to be given symbolic names

A point in a program can be referred to by its name rather than a numeric address

Machine and Assembler Codes

Microprocessors "understand" programs of 0's and 1's

For example:

1010 1001 A9

0000 0101 05

Hex is an aid, but what does the following small program do?

D8 A2 FF 9A 18 A9 05 69 07 8D 11 00

This is why we use assembly languages

Assembler Format

Assembly languages vary but typically have a similar format:

<LABEL>: <OPCODE> <OPERAND(S)> | COMMENT

For example:

```
START:  move.b   #5, D0     | load D0
                            | with 5
```

Example Assembler Program

Understanding assembler programs is about recognising each small operation and how it fits as part of a larger sequence of operations

```
        ORG         $4B0        | this program starts at hex 4B0
        move.b #5, D0           | load D0 with 5
        add.b       #$A, D0     | add 10 to D0
        move.b D0, ANS          | store result in ANS
                                |
ANS:    DS.B        1           | leave 1 byte of memory empty
                                | and give it the name ANS
```

Numbers:

 # : indicates a constant. A number without # prefix is an address

Default number base is DECIMAL

$: means in HEX

% : means in BINARY

Assembler Directives:

ANS: A label, i.e., a symbolic name, that must be terminated by a colon

DS (Define Storage): instructs the assembler to reserve some memory

ORG (Origin): tells the assembler where in memory to start putting the instructions or data

68008 Instruction Set

There are two aspects to the 68008 instruction set:

Instructions - The commands that tell the processor what operations to perform

Addressing Modes - The ways in which the processor can access data or memory locations, i.e., the ways in which addresses may be calculated by the CPU

68008 Instructions

The 68008 instruction set is made up of five categories of instructions:

1. Data Movement
2. Arithmetic
3. Logical
4. Branch
5. System Control

We will look briefly at the first four groups

Form of 68008 Assembler Instructions

Assembler instructions are written in the form:

operation.datatype source, destination

The operation can be on one of three data types:

byte	**.b**	**(8 bits)**
word	**.w**	**(2 bytes)**
long word	**.l**	**(4 bytes)**

The data type and dot may be omitted if the data type is word

1. Data Movement Instructions

```
move.b    D0, D1      | [D1(0:7)] ← [D0(0:7)]
moveb  D0, D1         | the same
move.w    D0, D1      | [D1(0:15)] ← [D0(0:15)]
move   D0, D1         | the same
move.l  $F20, D3      | [D3(24:31)] ← [MS($F20)]
                      | [D3(16:23)] ← [MS($F21)]
                      | [D3( 8:15)] ← [MS($F22)]
                      | [D3( 0:7)] ← [MS($F23)]
                      | (Big-Endian mean this way around)
exg.b   D4, D5        | exchange
swap    D2            | swap lower and upper words
lea   $F20, A3        | load effective address [A3] ←[$F20]
```

2. Arithmetic Instructions

The 68008 does not have hardware floating point support – instructions operate on integers

```
add.l    Di, Dj        | [Dj] ← [Di] + [Dj]
addx.w   Di, Dj        | also add in x bit from CCR
sub.b    Di, Dj        | [Dj] ← [Dj] - [Di]
subx.b   Di, Dj        | also subtract x bit from CCR
mulu.w   Di, Dj        | unsigned multiplication:
                       | [Dj(0:31)] ← [Di(0:15)] * [Dj(0:15)]
muls.w   Di, Dj        | signed multiplication
divu.b   Di, Dj
divs.l   Di, Dj
```

3. Logical Instructions

These instructions perform bit-wise operations on data and include AND, OR, EOR, NOT

For example, if register D3 contains 1010 0101, then...

AND.B #%11110000, D3

... will produce the result 1010 0000 in the least significant byte of register D3

Examples of Logical Instructions - Logic

Once you understand a few of these operations the operation of the rest becomes much clearer

Logical AND:

```
AND.B    #$7F, D0
```

D0	1	1	0	1	1	0	1	0
7F	0	1	1	1	1	1	1	1
D0	0	1	0	1	1	0	1	0

(keep just the lower 7 bits and ignore the MSB)

Logical OR:

```
OR.B    D1, D0
```

D1	1	0	0	0	0	0	1	0
D0	0	0	0	1	0	1	1	0
D0	1	0	0	1	0	1	1	0

Examples of Logical Instructions - Shifts

Shift operations are often fundamental to many computational tasks- remember we achieved multiplication by 2 using a left-shift operation

- **Logical Shift (L - left, R - right)**

(C and X are in Condition Code Register)

- **Arithmetic Shift**

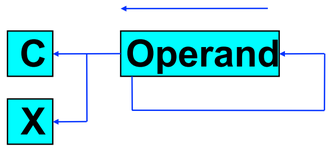
(ASR preserves sign bit)

Examples of Logical Instructions - Rotates

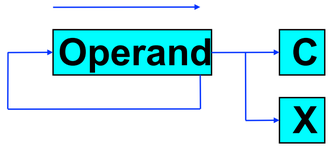
Note the difference between rotating (RO) and rotating through extend bit (ROX) - what does ROX achieve?

- **ROtate**

ROL

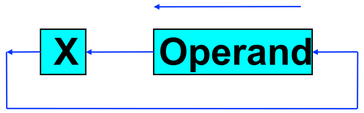


ROR

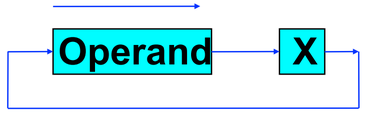


- **ROtate through eXtend bit**

ROXL



ROXR



4. Branch Instructions

Branch instructions cause the processor to branch (jump / GOTO) the labelled address

The instruction can test the state of the CCR bits and branch if a certain condition is met

CCR flags are set by the previous instruction

Form: Bcc <label> (where cc is a condition code)

If a branch is taken, [PC] ← label

Examples of Branch Instructions

15 branch instructions, including:

BRA branch unconditionally
BCC branch on carry clear $(\overline{C})$
BCS branch on carry set (C)
BEQ branch on equal (Z)
BGE branch on greater than or equal $(N.V + \overline{N}.\overline{V})$
…
BPL branch on plus (ie positive) $(\overline{N})$
BVC branch on overflow clear $(\overline{V})$
BVS branch on overflow set (V)

Subroutines and Stacks

Subroutines

Subroutines are useful for frequently used sections of a program

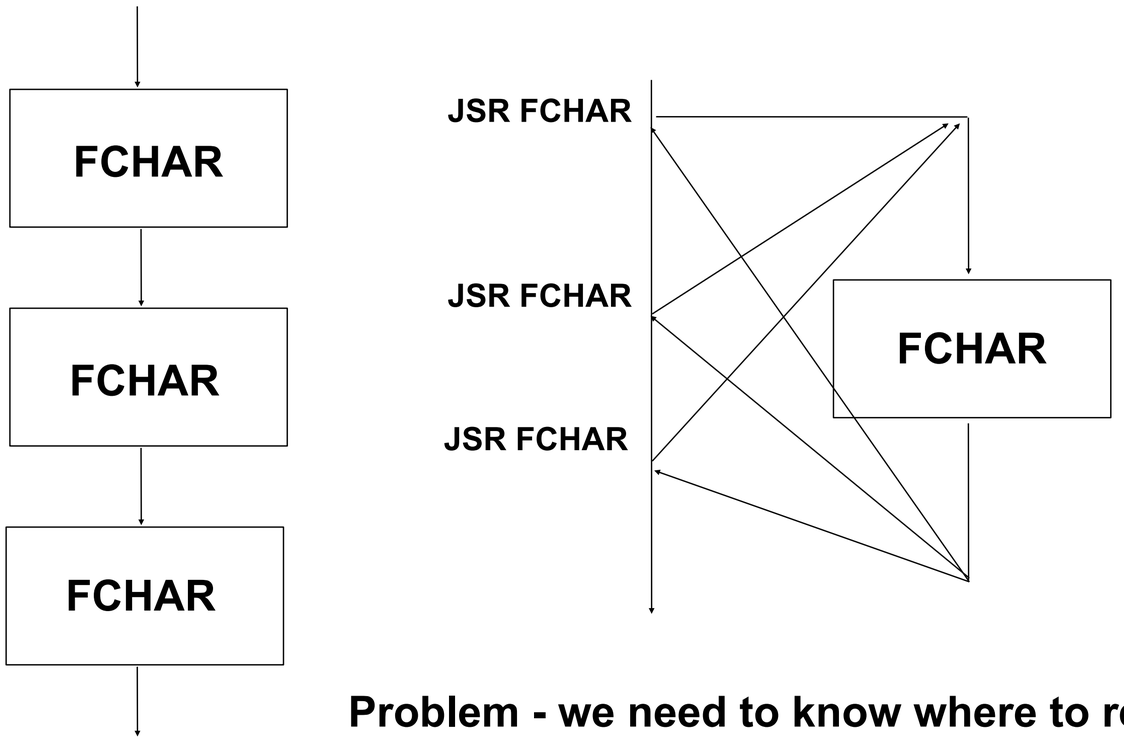
Write (and debug) a subroutine once, and use that program code whenever it is needed

- Reduces program size
- Improves readability

JSR <label>	**Jump to SubRoutine**
RTS	**Return from SubRoutine**

Subroutine Example

Can you see how subroutines work here - what assumption do you make when you return from a subroutine?



Problem - we need to know where to return

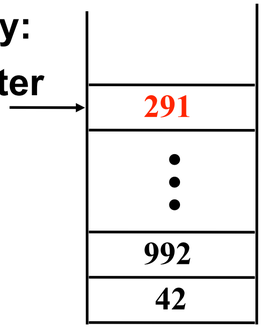
Subroutines and Stacks

A stack can be used to capture the last in first out (LIFO) aspect of the assumption we were making

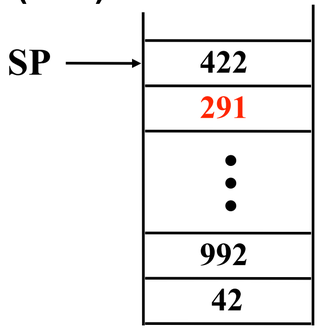
The stack is a last in first out structure. As before, A7 points to the Top Of Stack.

Say, initially:

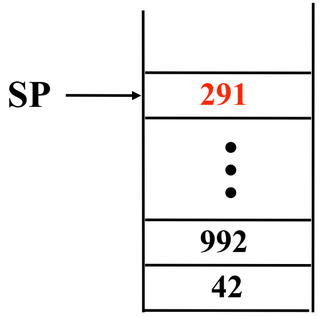
Stack Pointer (A7)



Then push (add) new value:



Then pop (remove) value:



Use of Stack for Subroutine Calls

Subroutine Call (JSR)

- Saves (pushes) the contents of the PC on the stack
- Puts start address of subroutine in PC

Return from Subroutine (RTS)

- Restores (pops) the return address from the stack and puts it in PC

So the stack stores where to return from a subroutine

A subroutine can call another subroutine - we have multiple return addresses on the stack in this case

Addressing Modes

Addressing Modes

How we tell the computer where to find data it needs

Need to organise application data

- Some data never changes, some is variable and some needs to be located within a data structure, e.g., list, table or array

In 68008 systems data can be located in a data register, within the instruction itself or in external memory

Addressing modes have expressive power

- Provide data directly
- Specify exactly where data is
- Specify how to go about finding data

Human “Addressing Modes”

“Here’s £100” (literal value)

“Get the cash from Room 119” (absolute address)

“Go to Room 26 and they’ll tell you where to get the cash” (indirect address)

“Go to Room 42 and get the cash from the fifth room to the right” (relative address)

68008 Addressing Modes

1. Data or Address Register Direct
2. Immediate Addressing
3. Absolute Addressing
4. Address Register Indirect - five variations
5. Relative Addressing

1. Data or Address Register Direct

Perhaps the simplest addressing mode to comprehend if you understand the 68008 architecture

The address of an operand is specified by either a data register or an address register

Example 1:

```
        move    D3, D2
```

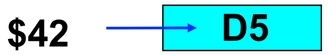
Example 2:

```
        move    D3, A2
```

2. Immediate Addressing

The operand forms part of the instruction and remains constant throughout the execution of a program

move.b #$42, D5



The above puts the hex value 42 into register D5

Note the # symbol!

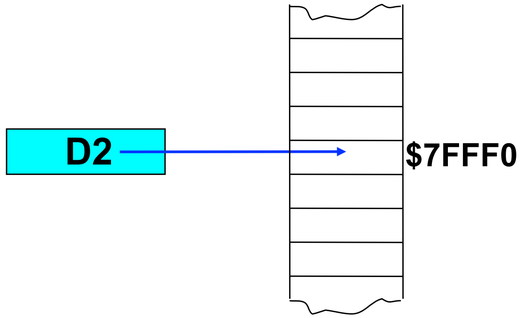
[D5] ← $42

3. Absolute Addressing

The operand specifies the location in memory explicitly, meaning that no further processing required

move.l D2, $7FFF0

[MS(7FFF0)] ← [D2]



Absolute addressing does not allow position independent code because a program will consistently use the same memory address

Note there is no # symbol

4. Address Register Indirect

Address Register Indirect

```
move    (A0), D3
```

Address Register Indirect with offset

```
move    7F(A1), D3
```

Post-Incrementing Address Register Indirect

```
move.b  (A0)+, D3
```

4. Address Register Indirect

Pre-Decrementing Address Register Indirect

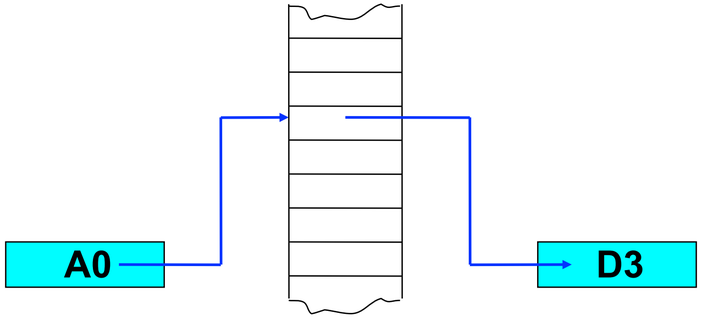
move.b -(A0), D3

Indexed Addressing

move.l 1F(A0, A1), D3

Address Register Indirect

move (A0), D3



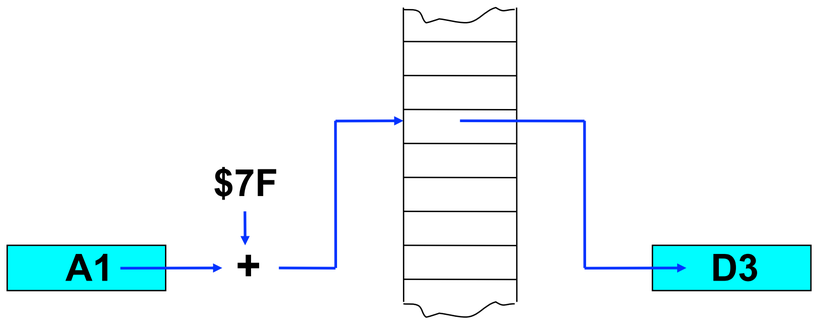
Means take the contents of address register A0 and use this number as the address at which the data will be found. Move this data to register D3

General form: **move (Ai), <ea>**

Remember <ea> is an effective address and can be Dj, (Aj), etc...

Address Register Indirect with Offset

move 7F(A1), D3



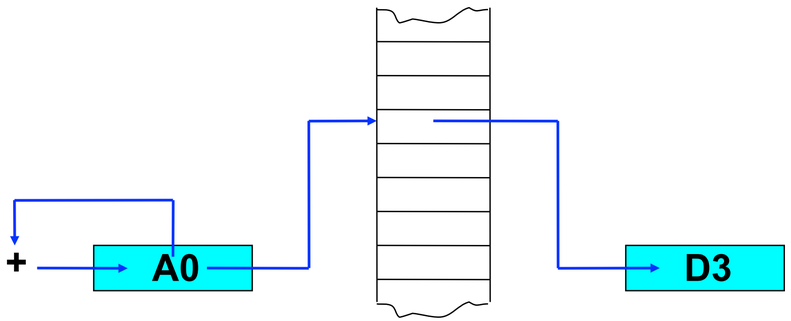
Means take the contents of address register A1, add to this number a (16-bit two's complement) constant and use this result as the address at which the data will be found. Move this data to register D3

General form: **move d16(Ai), <ea>**

This is where d16 is a 16-bit two's complement number

Post-Incrementing Address Register Indirect

move.b (A0)+, D3



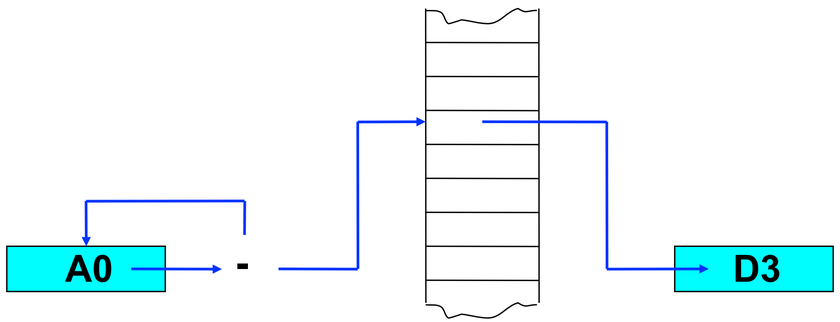
Means take the contents of address register A0, use this number as the address at which the data will be found. Move this data to register D3 and increment A0

General form: **move (Ai)+, <ea>**

Amount by which address register is incremented depends on the type of data being moved, i.e., 1 for bytes, 2 for words, 4 for long words

Pre-Decrementing Address Register Indirect

move.b -(A0), D3



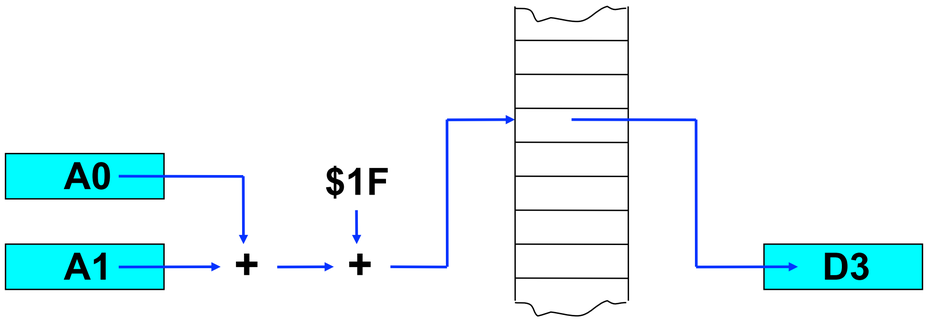
Means decrement the contents of address register A0, and use this number as the address at which the data will be found. Move this data to register D3.

General form: **move -(Ai), <ea>**

Amount by which the address register is decremented depends on the type of data being moved

Indexed Addressing

move.l $1F(A0, A1), D3



Means add the contents of address register A0 to A1, add to this the (8-bit two's complement) hex constant 1F and use this number as the address at which the data will be found. Move this data to register D3.

General form: **move d8(Ai, Xj), <ea>**

Xj can be an address or data register, **d8** is an 8-bit two's complement constant

Indexed Addressing Example

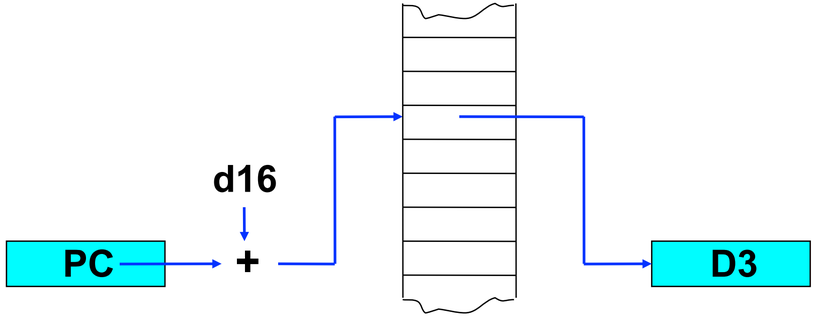
Imagine you're using a 2D-array to track the number of lectures you have, e.g., the number of lectures on Friday Week 2 is located at DIARY+2*7+5

	0 (Sun)	1 (Mon)	2 (Tue)	3 (Wed)	4 (Thu)	5 (Fri)	6 (Sat)
Week 0	*0*	*6*	*7*	*3*	*5*	*8*	*0*
Week 1	*0*	*4*	*5*	*2*	*6*	*6*	*0*
Week 2	*0 (location DIARY+14)*	*6*	*6*	*3*	*5*	*7 (location DIARY+19)*	*0*
Week 3	*0*	*7*	*4*	*1*	*7*	*5*	*0*

```
| pre:  week number in D0
| post: D1 contains number of lectures on Tuesday in week D0
lecsOnTue:   lea       DIARY, A0       | A0 points to start of DIARY
             mulu      #7, D0          | D0 is now a no. of days
             move.b    #2(A0,D0),D1    | D1 is now a no. of lectures
             rts
DIARY:       DS.B      28              | reserve 28 bytes (4 weeks)
                                       | (data initialised elsewhere)
```

5. Relative Addressing

move d16(PC), D3



Code like this contains no absolute addresses, i.e., it uses only addresses relative to the current program counter, and therefore can be placed anywhere in memory

This addressing mode can be used to write “position independent code”

Be careful - small addressing errors can have big consequences

The Accumulator Example

The example below shows how the use of an alternative addressing mode can reduce the work we have to do in developing elegant assembler programs

Original (pseudo code)

```
int N
int W[100]
int i, sum

sum = 0
for i = 0 to N-1
       sum = sum + W[i]
```

Assembly language version

```
        move.l   N, D1          | load D1 with number of items
        movea.l  #W, A2         | load A2 with addr. of 1st item
        clr.l    D0             | D0 used to accumulate sum
loop:   add.w    (A2), D0       | add next number from array
        adda.l   #2, A2         | increment A2: point to next word
        subq.l   #1, D1         | decrement counter
        bgt      loop           | if D1 > 0 then branch to loop
        move.l   D0, sum        | store result in sum
N:      DS.L     1
W:      DS.W     100            | (data initialised elsewhere)
sum:    DS.L     1
```

or, the two boxed lines could be the more optimal…

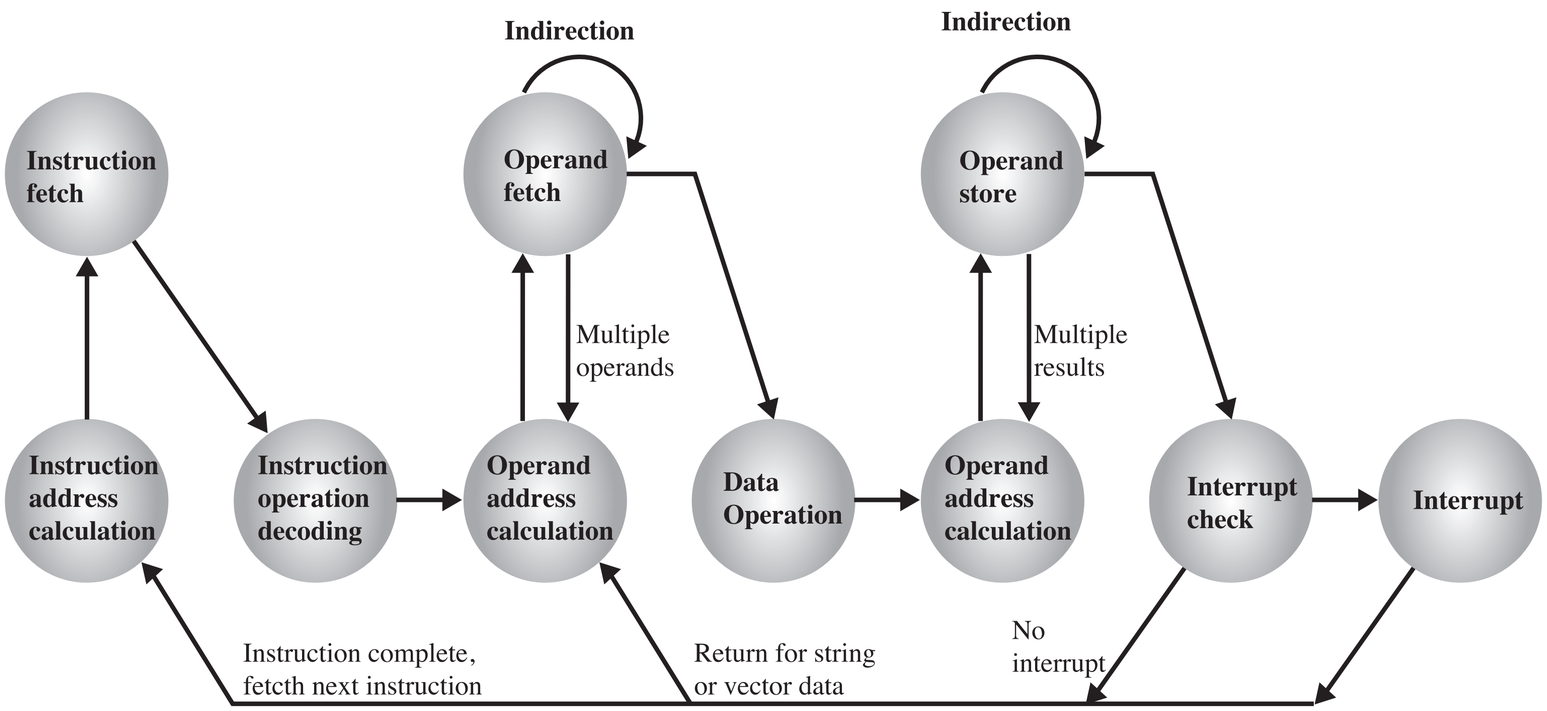
```
loop:  add.w   (A2)+, D0     | also increments A2
```

Fetch-Decode-Execute Revisited

Having studied assembler and addressing modes, we can consider the fetch-decode -execute cycle in it's full form

Elements of this might not fit together perfectly until we've explored memory system and input-output mechanisms

We'll be back!!



Reproduced from: W.Stallings, "Computer Organization and Architecture", 9th Edition, Pearson, 2013

Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 4 - Memory Systems

Session Outline

The memory hierarchy, including motivation and structure

Organising memory cells for addressing

Detecting and correcting errors in unreliable memory and network communications

The design and implementation of common memory components, including hard disks and optical discs

The Memory Hierarchy

The Choice of Memory Technology

To construct a memory system, we must think carefully about the properties of each storage approach

Only condition is that a memory system must facilitate data modification (writing) and data detection (reading)

Many factors influence the choice of memory technology

- Frequency of access
- Access time
- Capacity required
- Cost, e.g., cost per bit

The Designer's Dilemma for Memory Systems

The designer's dilemma

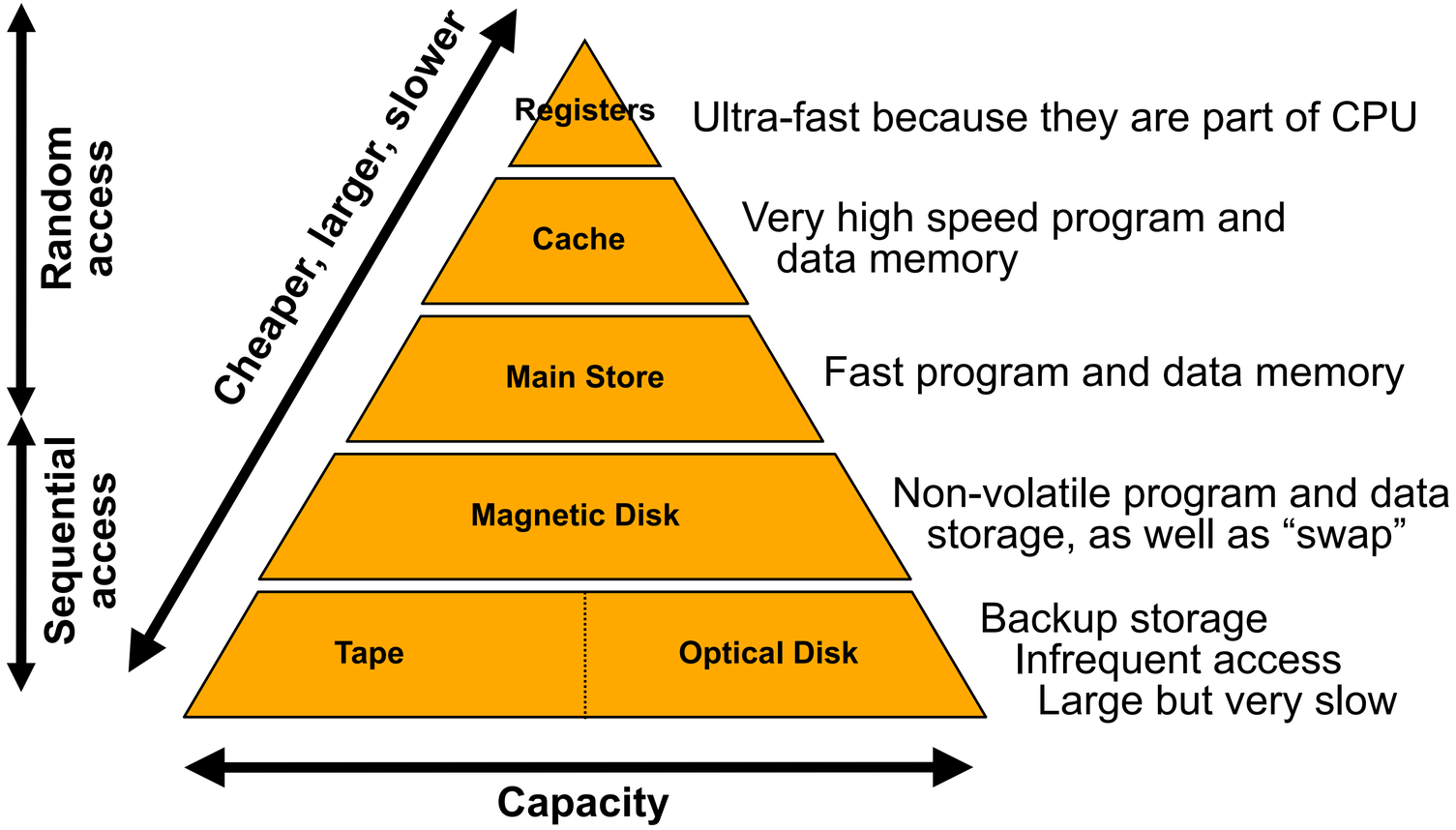
Desire low cost and high capacity - leads to choice of cheap per bit, high capacity and slow access time

Desire high performance - leads to choice of expensive per bit, low capacity and fast access time

To solve the problem we choose both by organising memory into a hierarchy

The Memory Hierarchy

A general view of the memory system that underpin modern computer systems



Why Have a Memory Hierarchy?

Economics!

Small sequential loops, subroutines, operations on arrays, tables and other structures mean that the locations in memory which will be accessed is somewhat predictable

Temporal locality - If a particular memory location is referenced, it is likely that the same location will be referenced again in the near future

Spatial locality - If a particular memory location is referenced, it is likely that nearby memory locations will be referenced in the near future

Reasonable assumption - It has been shown that 90% of memory accesses are within ± 2 kilobytes of previous PC position

Semiconductor Memory Types

Particularly interested in memory that provides random access

RAM is most common type of semiconductor memory, though the term is misleading since all types shown above allow random access

<table>
<tr><th>Memory Type</th><th>Category</th><th>Erasure</th><th>Write Mechanism</th><th>Volatility</th></tr>
<tr><td>Random Access Memory (RAM)</td><td>Read-write</td><td>Electrically at byte-level</td><td>Electrically written</td><td>Volatile</td></tr>
<tr><td>Read-only Memory (ROM)</td><td rowspan="2">Read-only</td><td>Not possible</td><td>Mask written</td><td rowspan="6">Non-volatile</td></tr>
<tr><td>Programmable ROM (PROM)</td><td>Not possible</td><td rowspan="4">Electrically written</td></tr>
<tr><td>Erasable PROM (EPROM)</td><td rowspan="3">Read-mostly</td><td>UV-light at chip-level</td></tr>
<tr><td>Electrically Erasable PROM (EEPROM)</td><td>Electrically at byte-level</td></tr>
<tr><td>Flash Memory</td><td>Electrically at block-level</td></tr>
</table>

Cache Memory

What is a Cache?

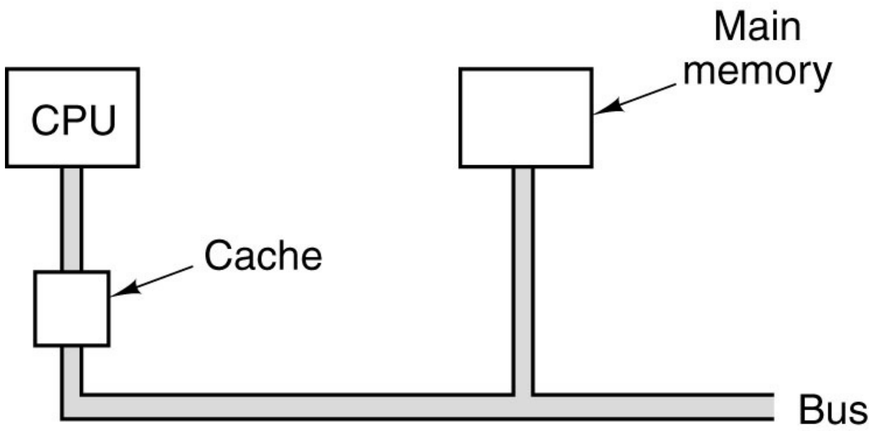
"90% of memory accesses within only 2 Kbytes"

So store those 2 Kbytes in a small, fast "cache" memory

If data required by CPU is in the cache (a "cache hit"), big speed improvement

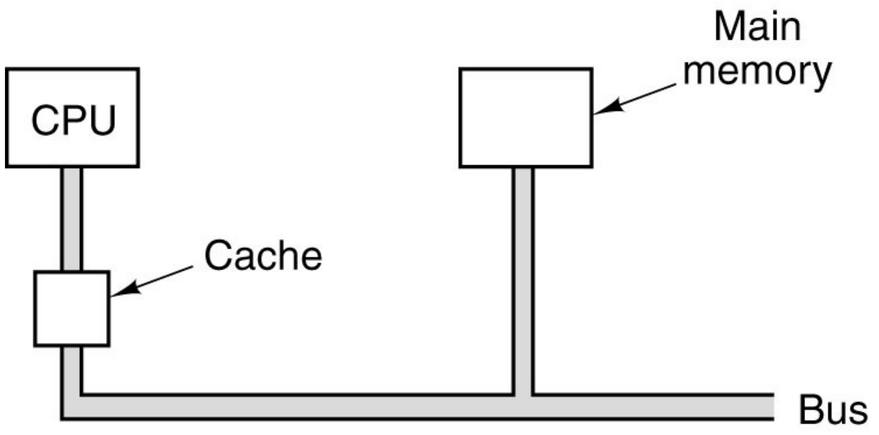
Cache is small to limit cost

Note that "cache" means "hiding place" - transparent to programmer



What is a Cache?

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

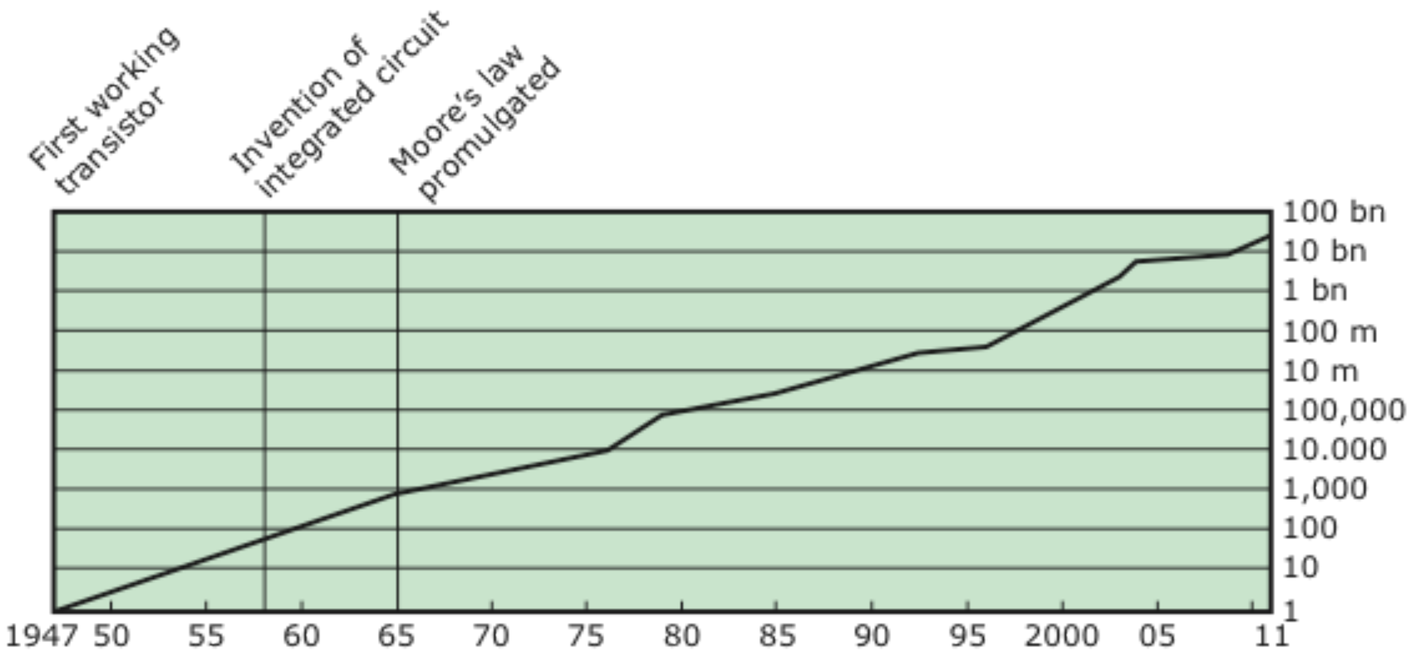


Moore's Law

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years." ***- Gordon Moore (co-founder of Intel), Electronics, Volume 38, No. 8, April 1965***

Now more commonly expressed as "The number of transistors on a memory chip doubles every 18 months"

Moore's Law



Reproduced from: W.Stallings, "Computer Organization and Architecture", 9th Edition, Pearson, 2013

Moore's Law

The cost of computer logic and memory circuitry has fallen at a dramatic rate

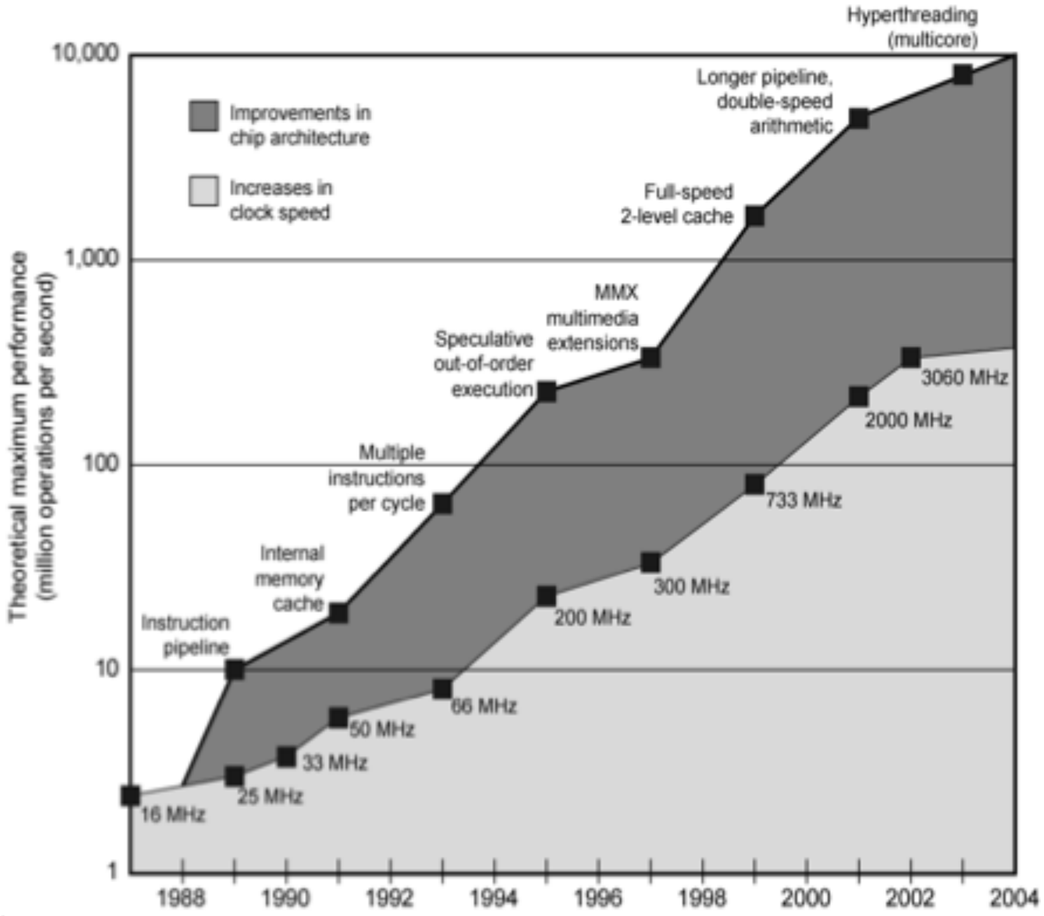
As logic and memory is placed closer together on more densely packed chips, the electrical path is shortened, increasing operating speed

Computers have become smaller, making them more convenient for use in a variety of environments, especially embedded systems

Reduction in power and cooling requirements

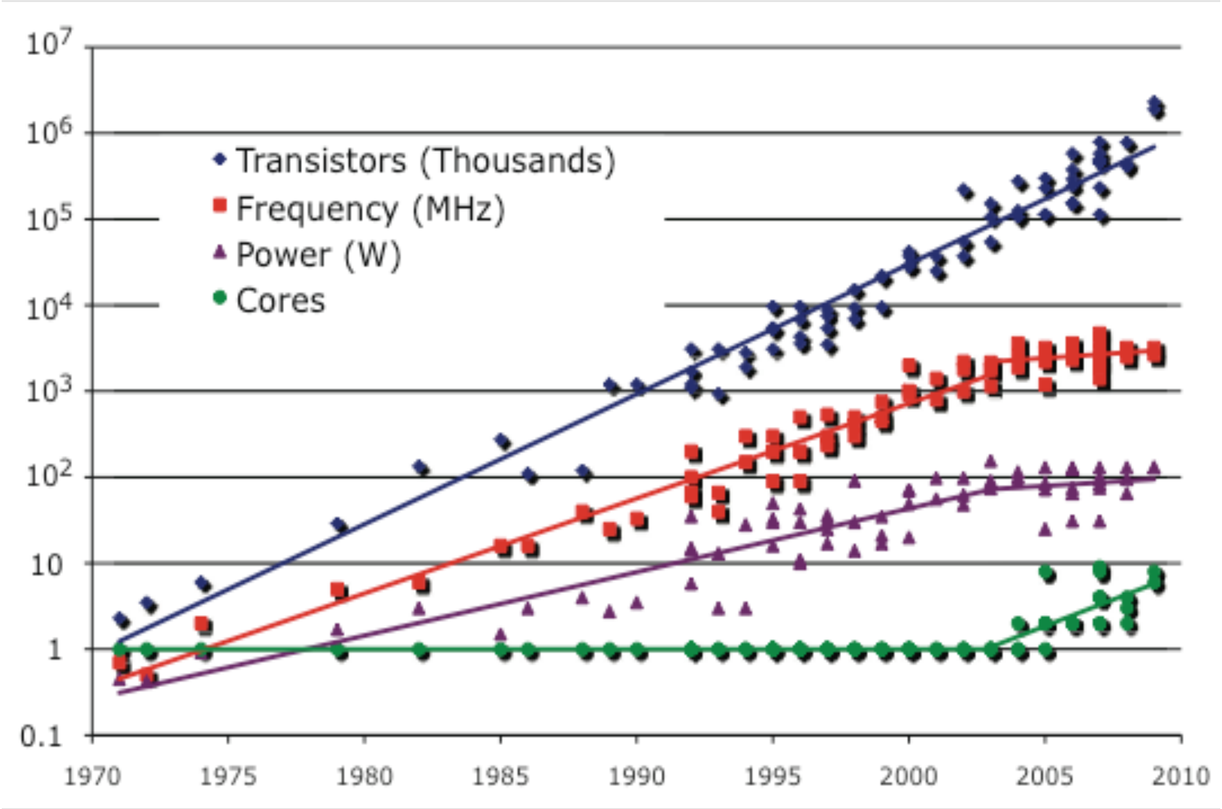
Interconnections on an IC are more reliable than solder connections, thus having fewer off-chip connections increases reliability

Intel Microprocessor Performance



Reproduced from: W.Stallings, “Computer Organization and Architecture”, 8th Edition, Pearson, 2010

Current Outlook - Hardware

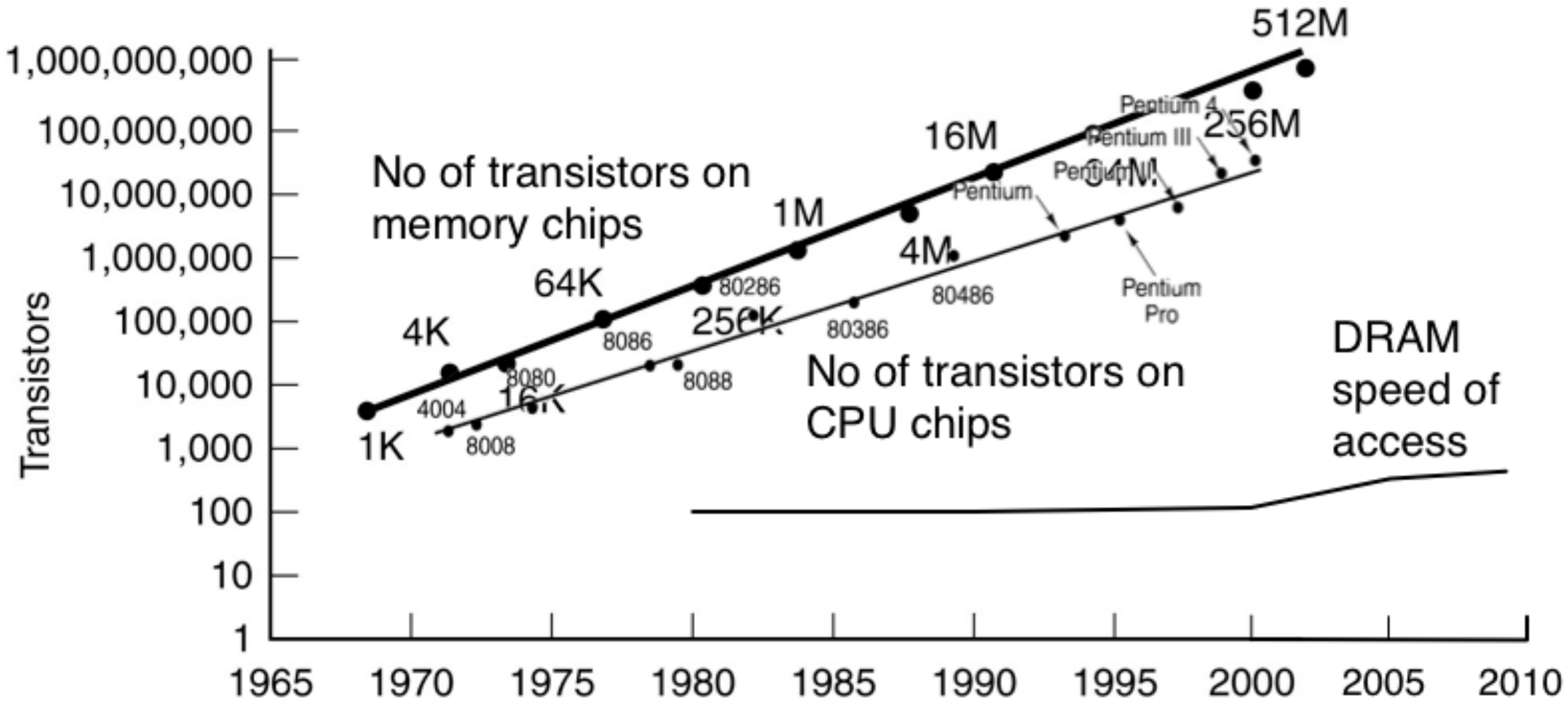


Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

Why Have a Cache?

Memory size is increasing according to Moore's Law

Memory access speed is improving much more slowly



Cache Concepts

Caching read-only data is relatively straightforward

We don’t need to consider the possibility that items will change, hence copies across the memory hierarchy will remain consistent

Two general strategies can be adopted when caching writes, both of which can employ a write buffer to enhance performance

Write Through - Updates the item in cache and writes through to update lower levels of the memory hierarchy

Write Back - Only updates copy in cache, ensuring that blocks are copied back to memory when when they are to be replaced

Types of Cache Miss

Measures of cache performance include hit rate (h) and miss rate (1-h)

$$h = \frac{\text{Number of times the words are in cache}}{\text{Total number of memory references}}$$

We can understand cache misses by sorting all misses into categories

Compulsory - Misses that would occur regardless of cache size, e.g., the first access to a block can not be in cache (regardless of cache size) so the block must be retrieved

Capacity - Misses occurring because a cache is not large enough to contain all blocks needed during the execution of a program

Conflict - Misses occurring as a result of the placement strategy for blocks not being fully associative, which means that a block may be discarded and retrieved

Coherency - Misses occurring due to cache flushes in multiprocessor systems

Measuring Cache Performance

Performance measures based solely on hit / miss rates don't factor in the cost of a cache miss - the real performance issue!

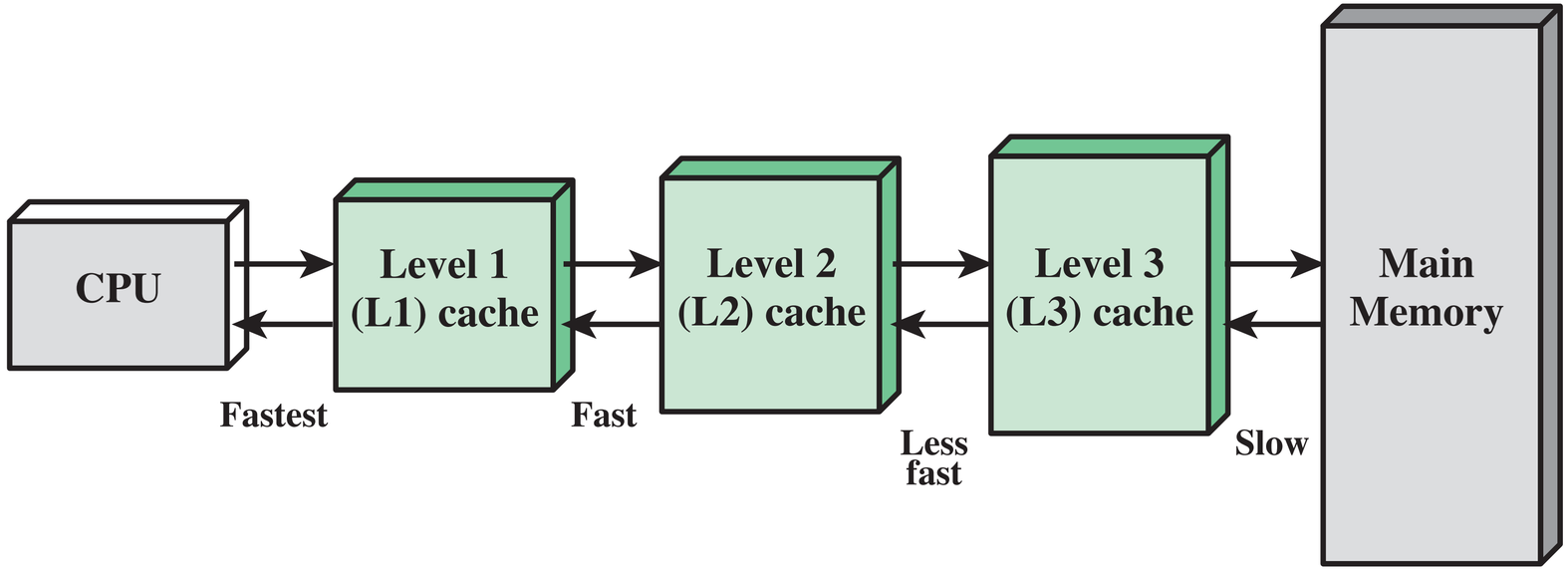
Average memory access time is an alternative measure that accounts for the cost of a cache miss

$$\text{Average memory access time} = \text{Hit time} + (\text{Miss rate} \times \text{Miss Penalty})$$

This is where hit time is the time to hit in the cache and the miss penalty is the time taken to replace the block from memory

Even average access time is still an indirect measure of performance - execution time can often be a better measure in real-world situations

Multilevel Cache



Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

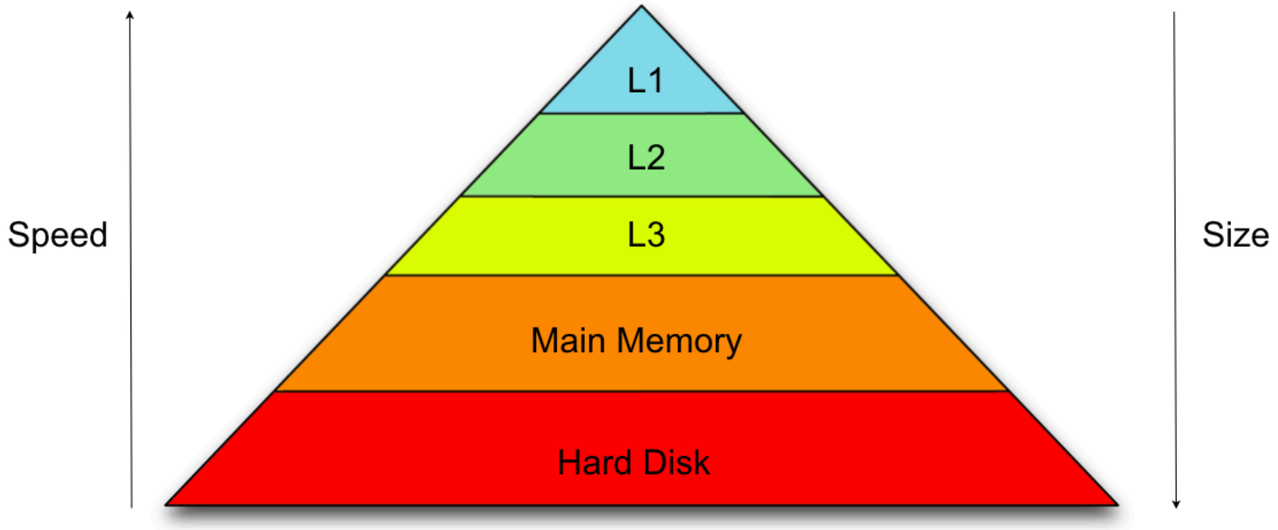
Relating the Memory Hierarchy to Cache

Data storage is a balance of size and speed:

Memory closer to the CPU (Cache - L1, L2, L3) is typically faster but has a far smaller capacity.

Main memory is further and slower but has a far greater capacity

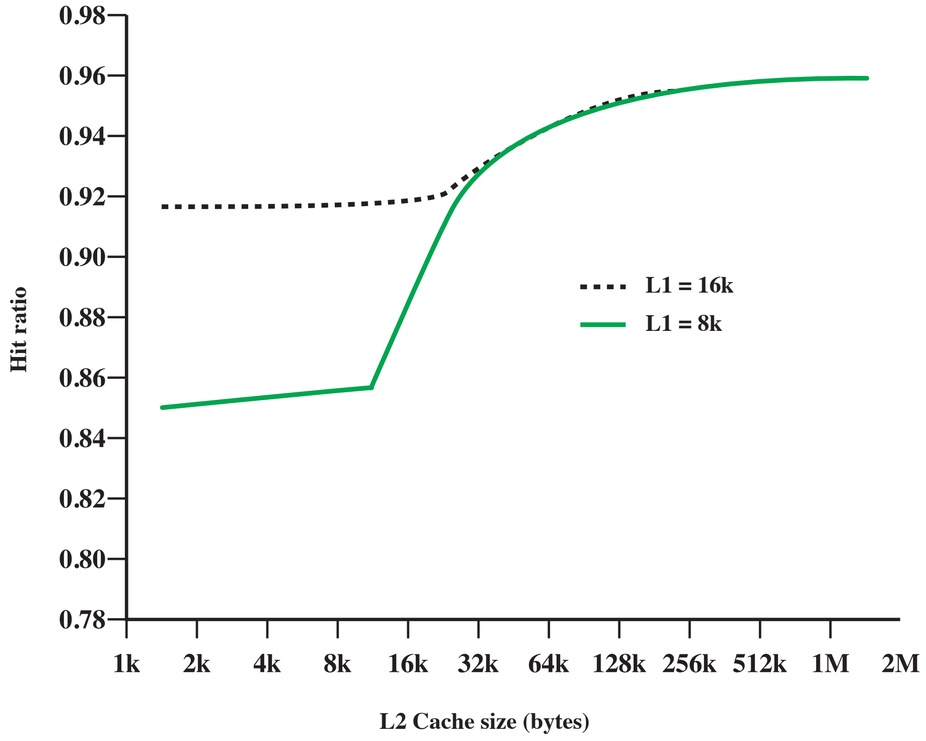
Hard disks have the greatest capacity, and do not need continuous power, but have the slowest access times.



Optimisation: We want to use fastest memory possible for high performance, but problems may large datasets.

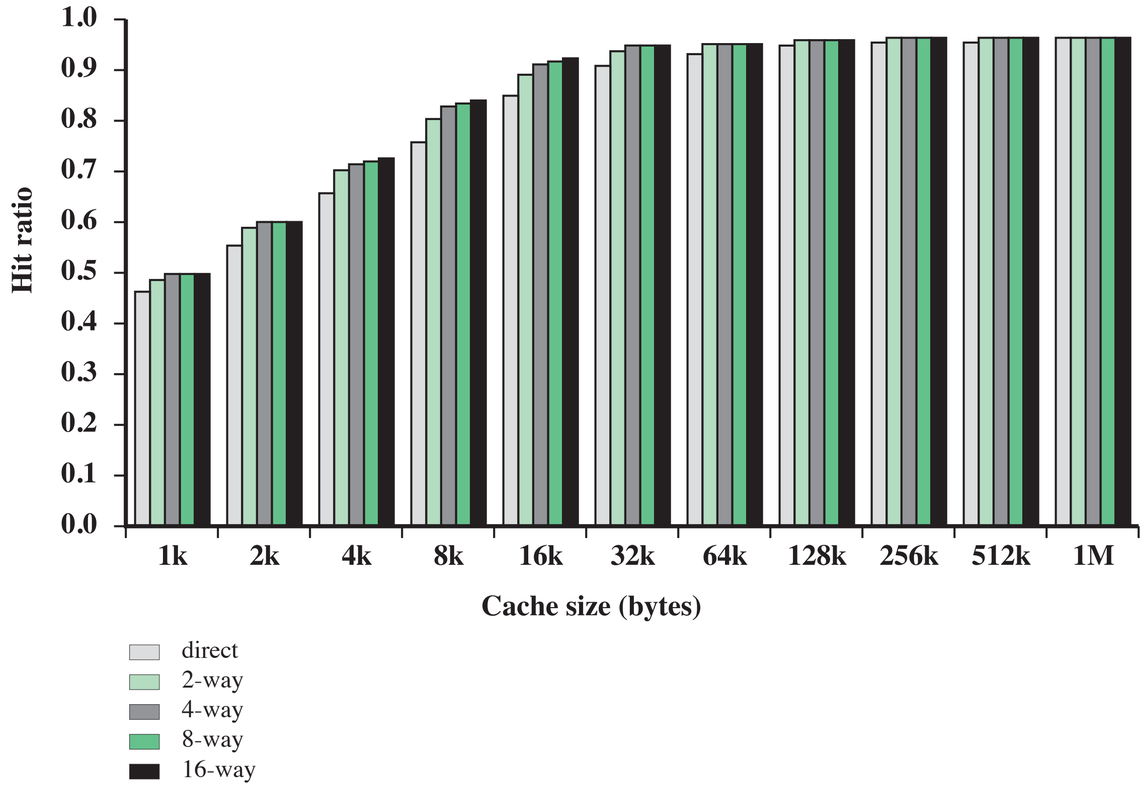
Solution: Reuse data already in cache as much as possible and prefetch data from main memory into cache before it is needed, masking load times

L1 Cache Size and Complexity



Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

L1 Cache Size and Complexity



Reproduced from: W.Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013

Memory Cell Organisation

Semiconductor Memory

Most common form of main store

RAM (Random Access Memory)

Two main technologies

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

SRAM use a flip-flop as storage element for each bit

DRAM for each bit, use the presence or absence of charge in a capacitor to denote a 1 or 0

- Capacitor charge leaks away over time - requires periodic refreshing (more complex) but is cheaper than SRAM so is more commonly used

Static RAM (SRAM)

SRAM stores data using configurations of flip-flops and logic gates

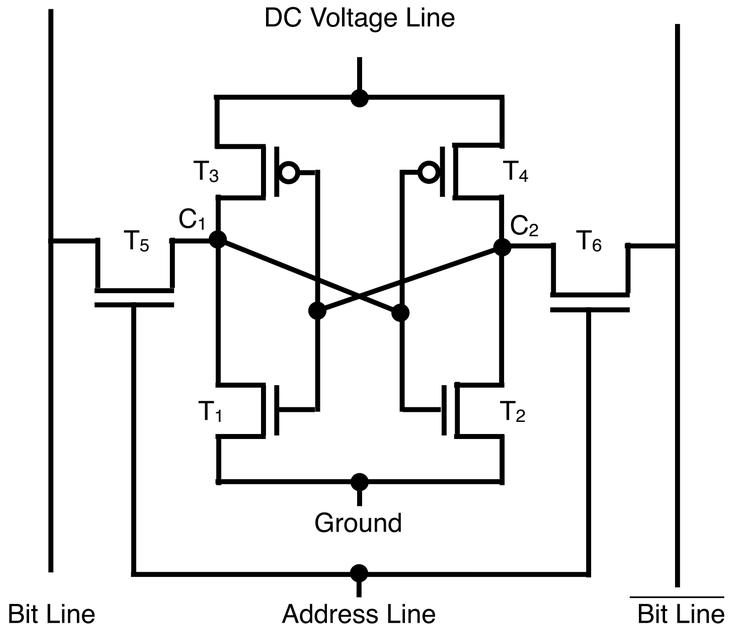
Memory cells hold data as long as power is supplied

Typical SRAM cell uses four cross-coupled transistors to establish a stable logic state

When storing logical 1, C1 is high and C2 is low (T1 and T4 off, T2 and T3 on)

When storing logical 0, C1 is low and C2 is high (T1 and T4 on, T2 and T3 off)

Address line controls two transistors (T5 and T6), where both transistors being on permits read and write



Write operation by applying bit value to Bit Line, which sets T1, T2, T3 and T4 to the appropriate bit value

Read operation by reading Bit Line

Dynamic RAM (DRAM)

DRAM memory cells store data as charge on capacitors

Capacitors naturally discharge, hence DRAM cells require periodic charge to maintain data storage

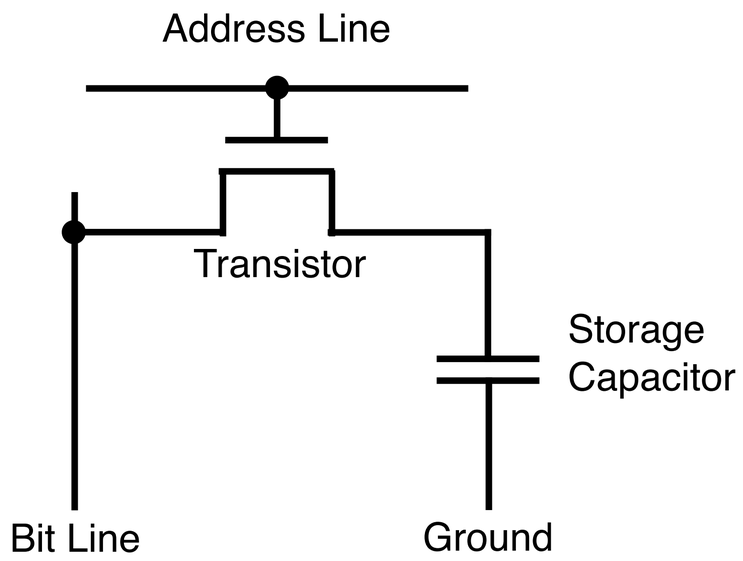
The term 'dynamic' refers to the natural leakage of charge that takes place even when power is applied

Presence or absence of charge is interpreted as a logical 0 or 1

We can appreciated the operation of DRAM by considering a 1-bit DRAM memory cell

Address line is activated when the bit value from the cell is to be read or written

A transistor acts as a switch that is closed (allowing current flow) if a voltage is applied to the address line and open if a voltage is not applied to the address line (no flow)



DRAM Operation

Write operation:

Voltage applied to the bit line (high for logical 1, low for logical 0)

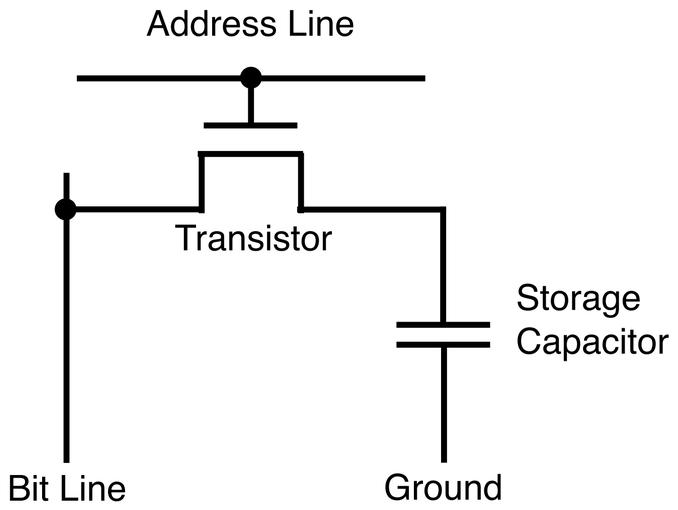
A signal is then applied to the address line, allow a charge to be transferred to the capacitor (data store)

Read operation:

Transistor turns on when address line is selected, allowing the stored charge to be fed out to the bit line and a sense amplifier

Sense amplifier compared the capacitor voltage to a predetermined threshold value to determine if the cell contains a logic 0 or 1

The read process discharges the capacitor, which means that it must be restored (as well as refreshed) following the read operation



Comparing SRAM and DRAM

Both are volatile - power must be continuously applied

Dynamic memory cells are generally simpler and more compact

- Allows greater memory cell density (more cells per unit area)
- Cheaper to produce than equivalent SRAM memory

Refresh circuitry incurs one-off cost that is only compensated for by larger memory capacities

SRAM are typically provide better read and write times than DRAM

- Cache memory (on and off chip) is often implemented as SRAM, whilst DRAM is commonly used for main memory

Advanced DRAM Organisations

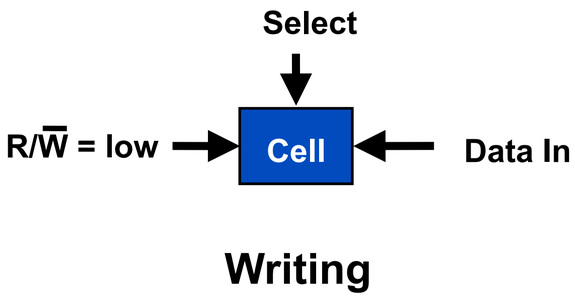
- The interface to main memory is a critical performance bottleneck
  - Motivation for multiple high-speed SRAM caches between a processor and DRAM main memory
  - To achieve better system performance we can also consider advanced DRAM organisations
    - Synchronous DRAM (SDRAM)
    - Rambus DRAM (RDRAM)
    - DDR SDRAM
    - Cache DRAM (CDRAM)

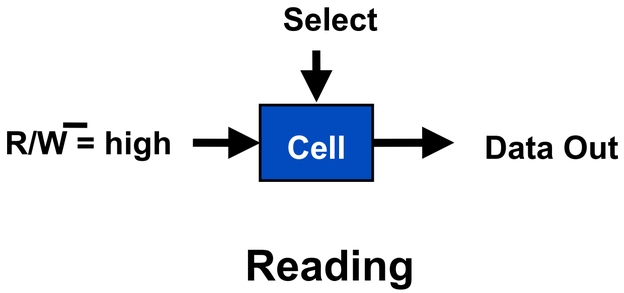
Memory Organisation

Basic element is a memory cell

Two states 1, 0

Capable of being written and read





Select – Enable a particular memory cell

R/W – Indicate read or write

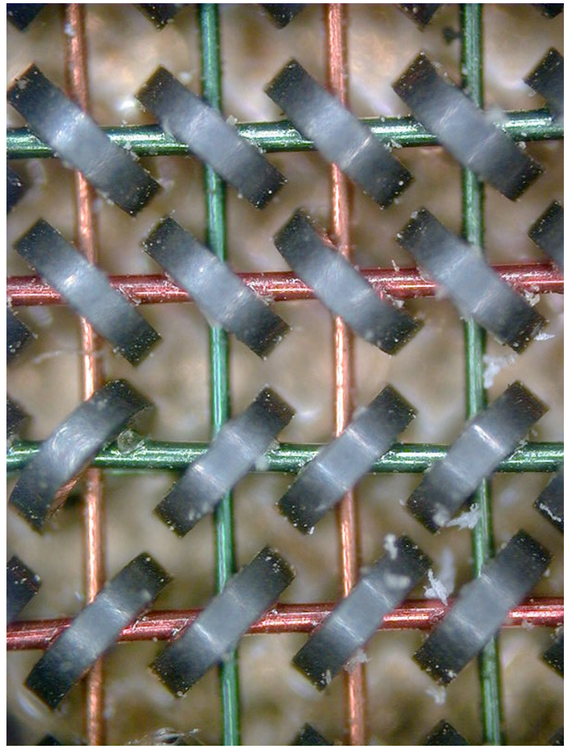
Data – Input when writing or output when reading

Core Memory

An early form of RAM where data is stored through the magnetisation of tiny metal rings – one bit per ring

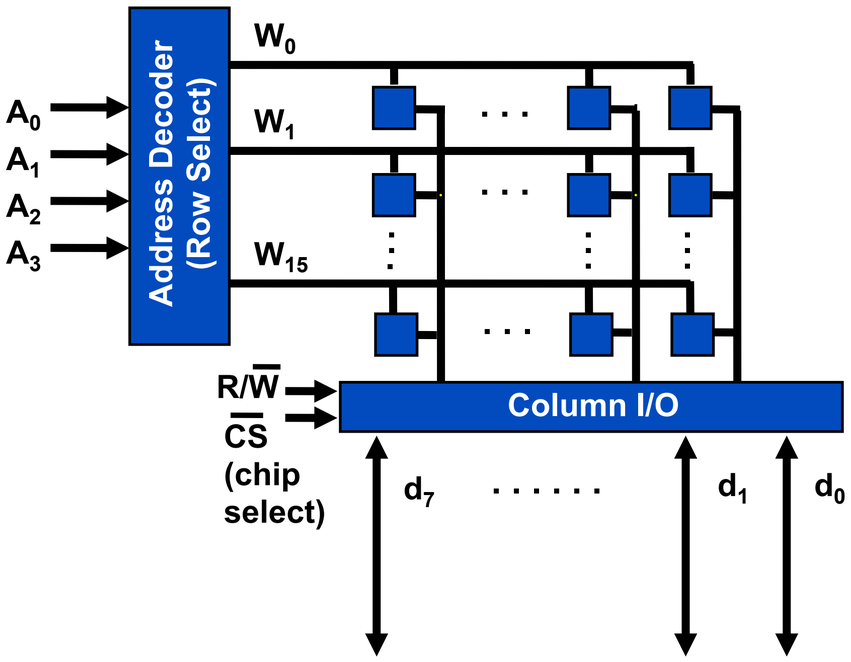
Rings are organised into a grid array

Individual bits are accessed by energising row and column wires



Organisation of a Memory Chip

As you might expect, it is critical to maintain a structured approach to the organisation of on-chip memory



Example of a 16x8 memory integrated circuit

16x8 denotes
W x B, i.e., words x bits
16 words, each of 8 bits

Choices in Memory Chip Organisation

IC design for 16x8 = 128 cells required 4 address inputs (= $\log_2$ W, where W = no of words) and 8 data lines (=B, where B = word size)

Consider a 1Kbit =1024 cells device

Possible to organise as a 128 x 8 memory cell array

Requires 7 address pins (log2 128) and 8 data pins

A total of 15 I/O pins, plus power, etc...

Poor Choices in Memory Chip Organisation

Alternative, it is possible to organise as a 1024 x 1 array

Requires 10 address pins ($\log_2 1024$) and 1 data pin

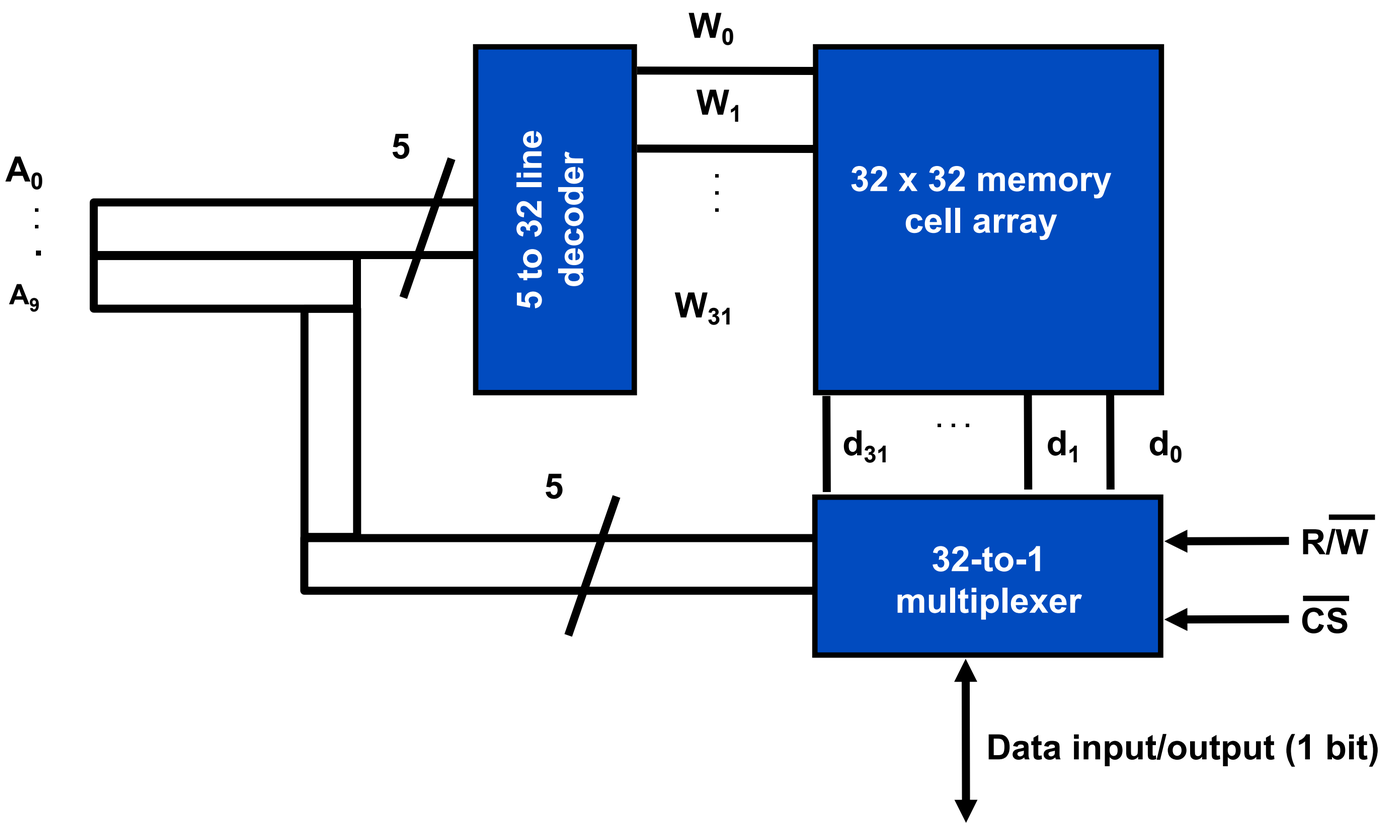
A total of only 11 I/O pins, plus power, etc...

Results in a long, narrow cell array with a large address decoder, which is an inefficient use of space

Minimising Address Decoding Space

Divide the address inputs into 2 parts - row address and column address

This minimises the space required for address decoding and hence maximises space used for memory cells



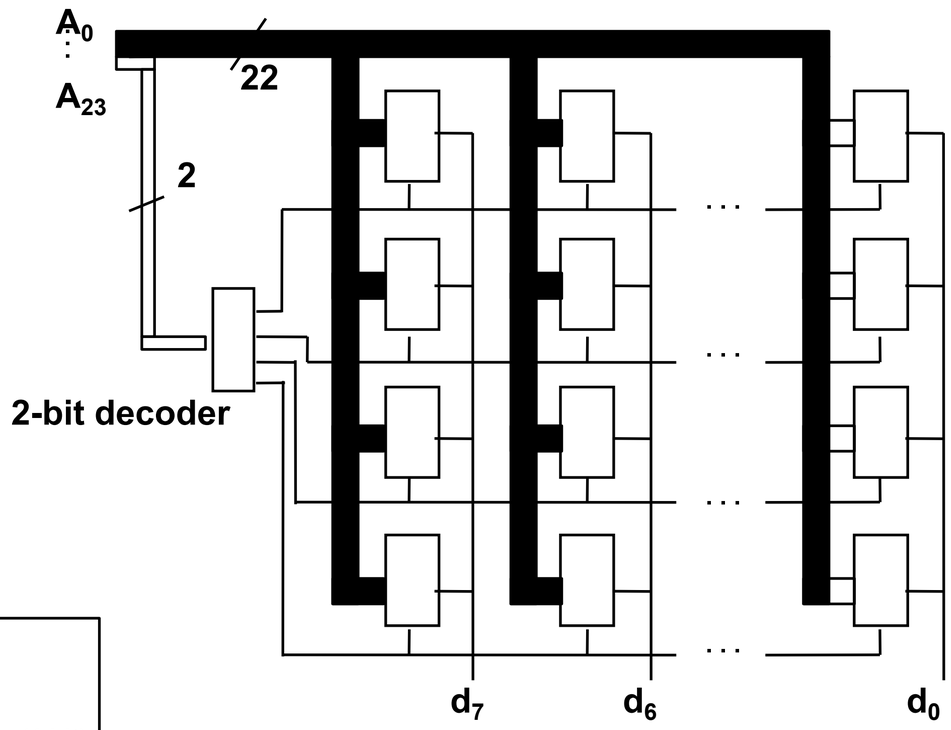
System Memory Organisation

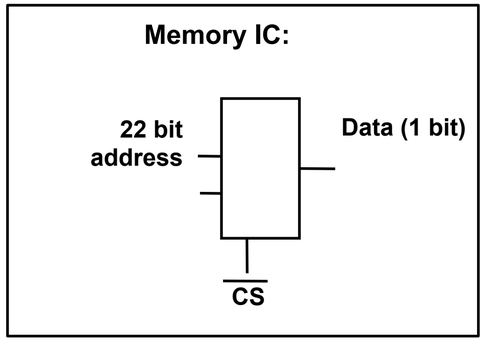
We can use the same idea to organise multiple memory ICs

Here, A_{22} and A_{23} select 1 row of 8 chips, each providing one bit of the 8-bit word

Each chip is 4M x 1 (22 address pins, 1 data pin)

System has 4 rows, 8 chips per row, making the system total a 16M x 8 memory

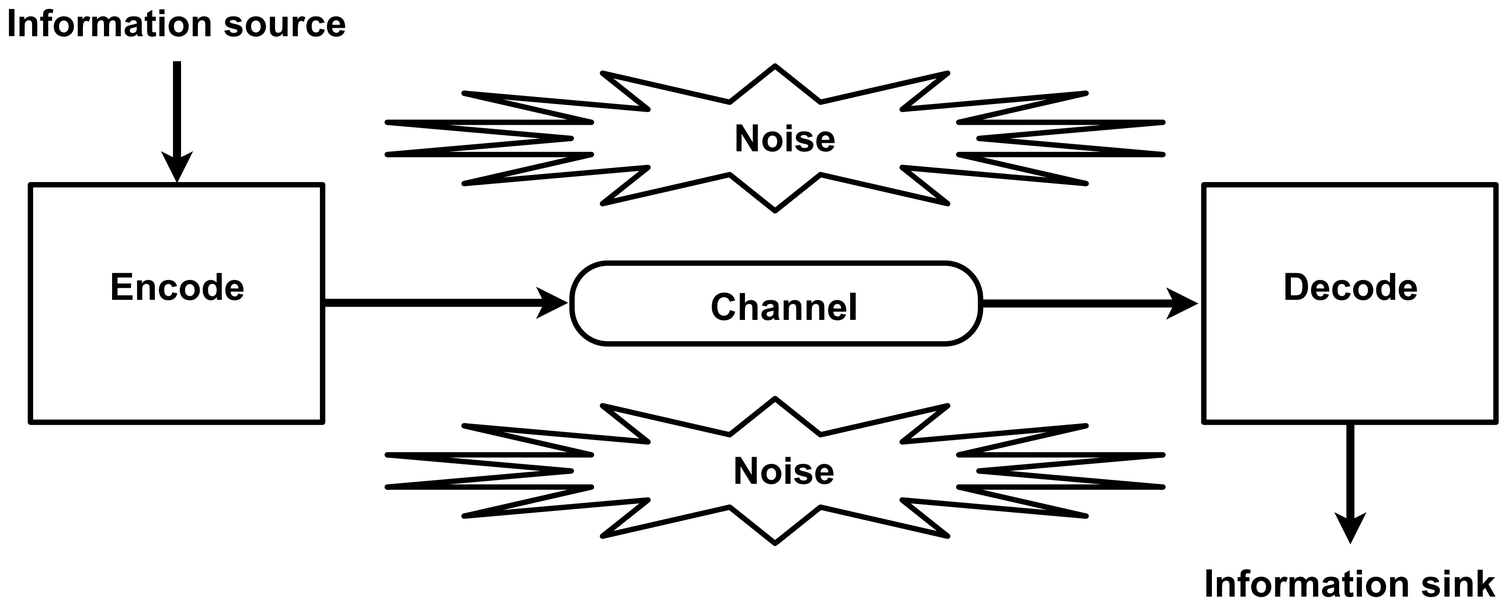




Detecting and Correcting Errors

Error Detection and Correction

Errors occur within a computer system, e.g., in system memory, and in the communication between systems, e.g., in the transmission of messages



Noise

Noise is unwanted information

It comes in various forms, but is always present, and is one of the limiting factors in computer systems

Noise arises from the physical properties of devices

- Thermal noise
- Noise of electronic components
- Noise of transmission circuits

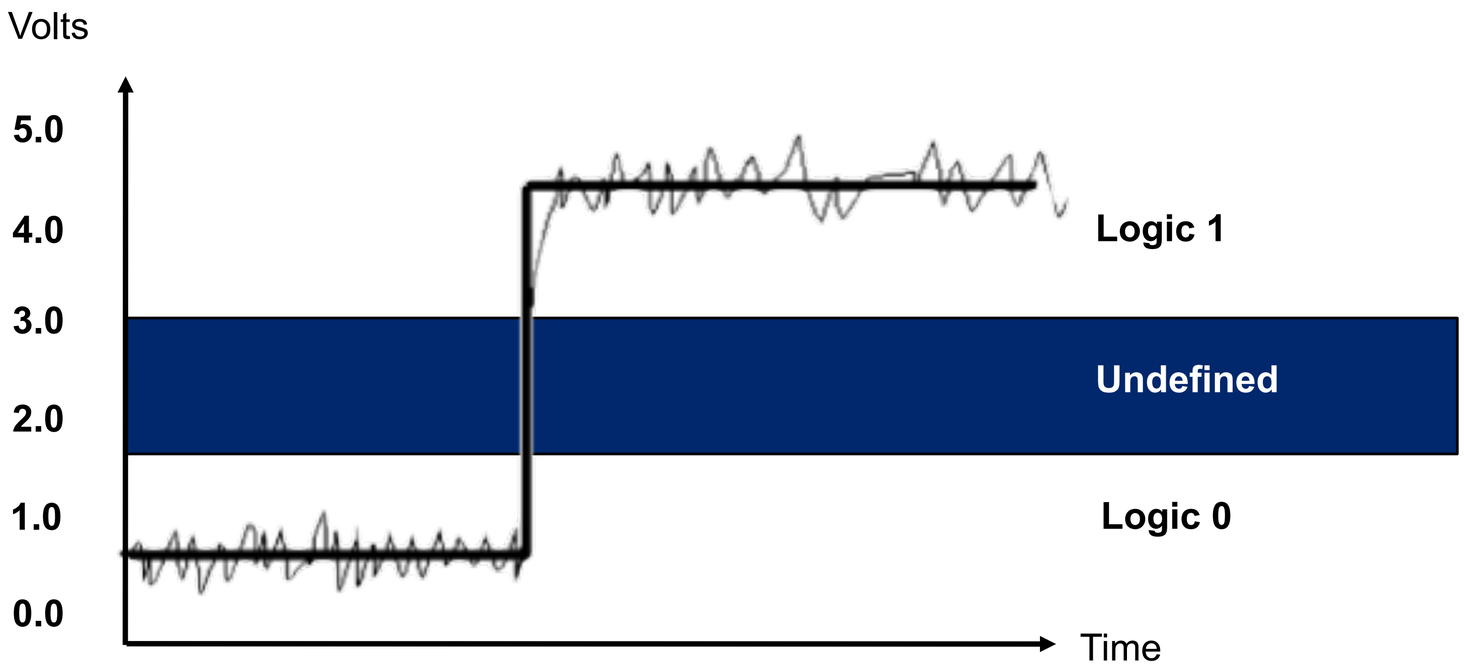
Magnetic media also have a classic form of noise due to random alignment of magnetic fields.

- Decreased area to store a bit means noise gets worse, making errors likely

Noise and Digital Logic Devices

Digital logic gives us a high degree of noise immunity

Immunity collapses once the noise reaches a certain magnitude - our aim is to not let this happen



Detecting Single, Isolated Errors

These are considered to occur at random, usually due to noise

We could send the message three times and take a vote

- Very expensive

Observation that, if the probability of an error in the channel is low, the probability of two errors close together is even lower

Thus, we can add a “parity” bit to the message every so often, which “summarises” a property of the message that we can check to determine whether the message has been altered

- Much cheaper and adequate in many situations

Parity

Parity adds an extra bit

There are two types of parity system.

Even parity system - The value of the extra bit is chosen to make the total number of logic 1s an even number

Odd parity system - Make the total number of logic 1s odd

Parity bit value can be computed in software or hardware

7-bit ASCII for 'A':	**A = 0100 0001**
Using even parity system:	**A = 0100 0001**
Using odd parity system:	**A = 1100 0001**

Parity in Software

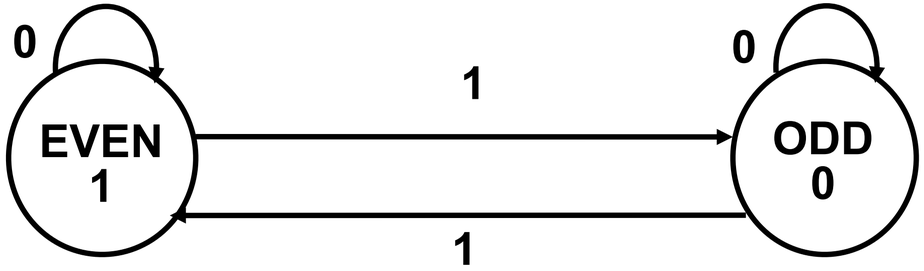
Odd parity system - Start at EVEN

Even parity system - Start at ODD

Each 1 in the message causes a transition of state

Final state at the end of the message is the resulting parity bit

Parity may be computed using a two state Finite Automaton



Parity in Software - Example

For example, try sending 110 using an even parity system

Start with the ODD state, then

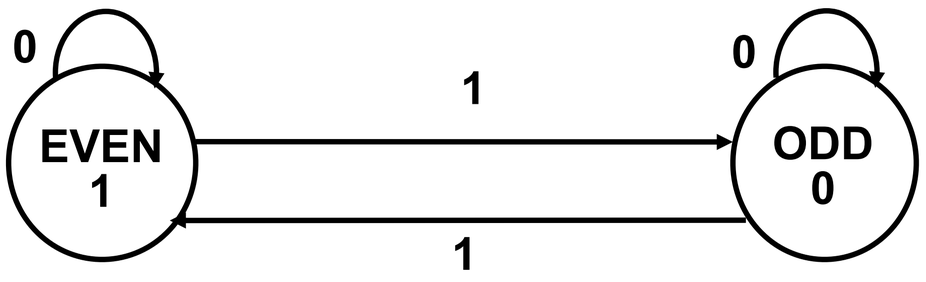
Bit 0 = 0, so go from ODD to ODD

Bit 1 = 1, so go from ODD to EVEN

Bit 2 = 1, so go from EVEN to ODD

Therefore the parity bit is 0 and the message to be sent is 0110

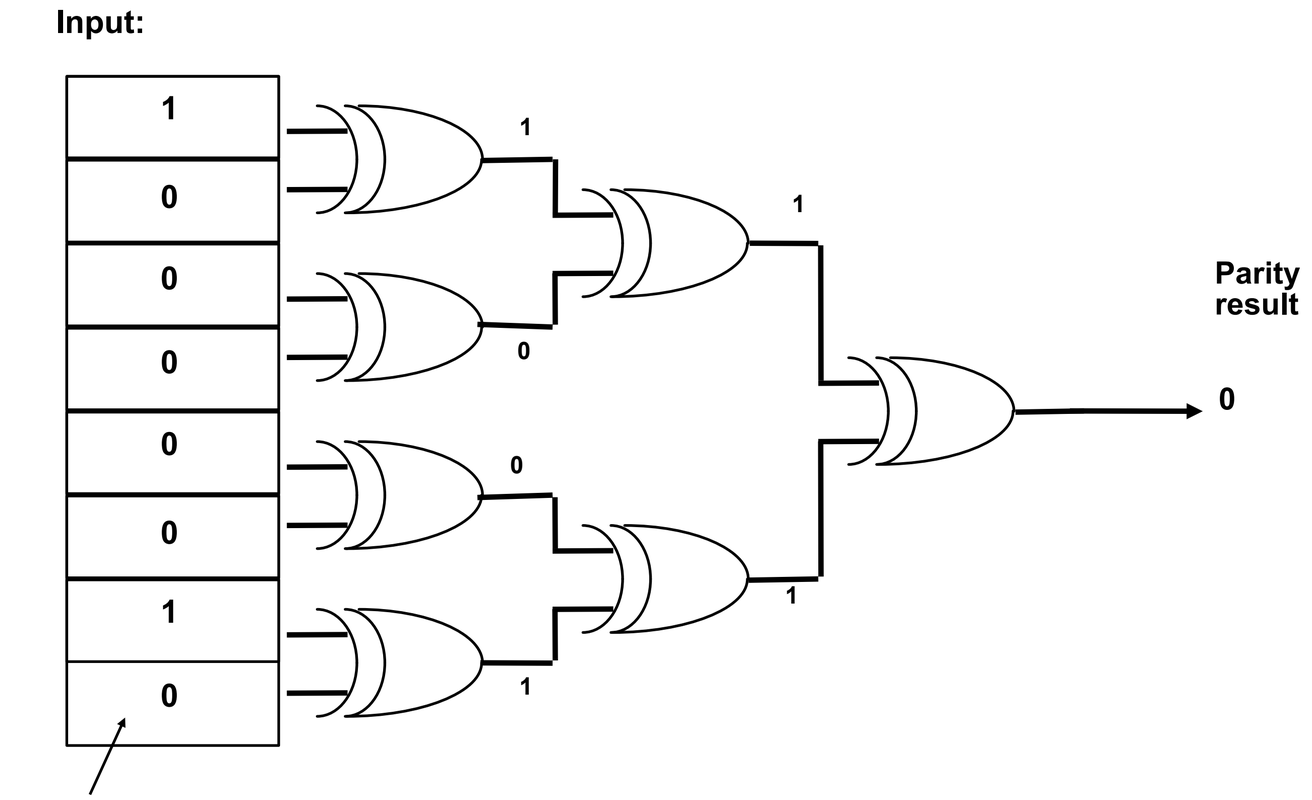
Parity may be computed using a two state Finite Automaton



Parity in Hardware

Computing parity in hardware is popular

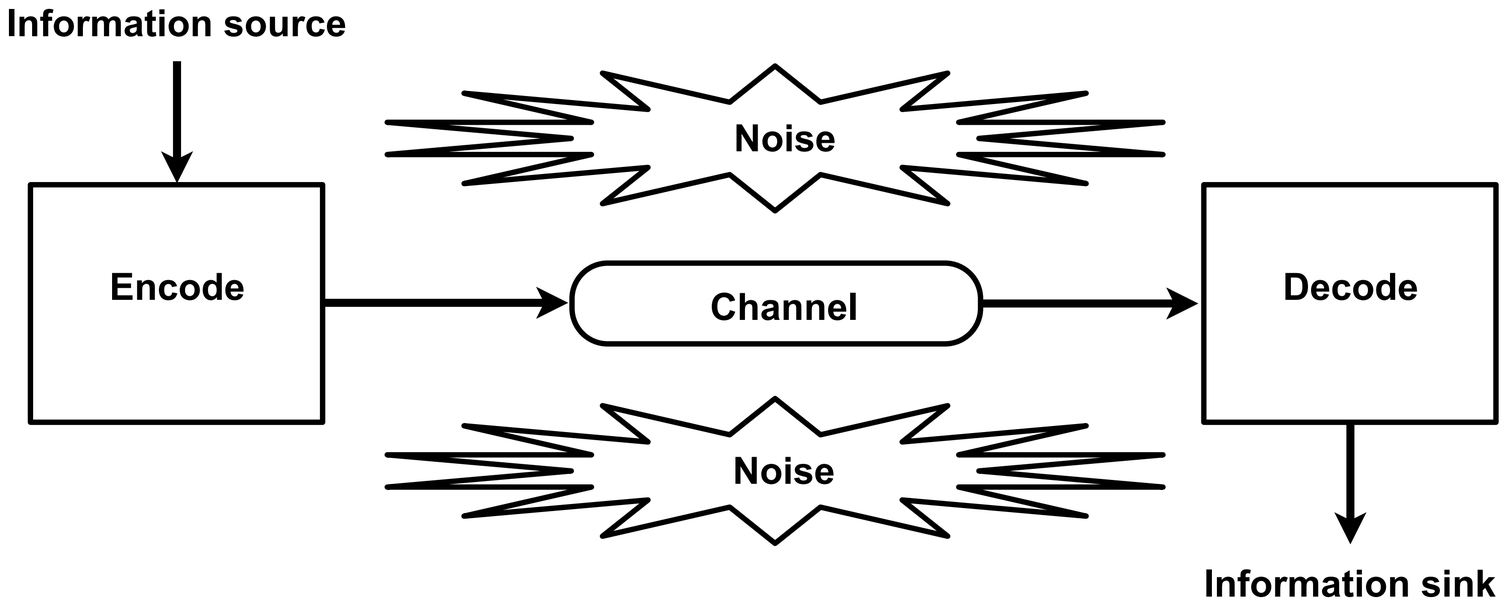
Hardware implementations generally perform better than software implementations, despite being a relatively simple digital logic circuit



Dummy input parity bit:
always 0 in even parity system (1 in odd parity system)

Overview of Parity

Parity provides error detection without the need to significantly change our model for communication



Error Detection Using Parity

At our transmitter a message, e.g., 7 bits, is encoded

Compute parity (P) and append parity to message

Send modified message across channel

At our receiver the message, now 8-bits including parity, is decoded

Compute parity (Q) from data bits of received message

Compare computed parity Q with received parity P in message and flag ERROR if these are not consistent

Error Detection Using Parity - Example

Transmitter and receiver have been designed to use even parity

Message: 100 0101

Transmitter computes P=1

Transmits: 1100 0101

An error in the channel changes one bit

Received: 1101 0101. Receiver notes received P=1

Receiver uses the data bits (101 0101) to compute Q=0

Receiver compares received P and computed Q.

An error has been detected, thus receiver will ask for retransmission

Burst Errors

Remember that, if the probability of an error in the channel is low, then the probability of two errors close together is even lower

In practice, multiple errors often occur in bursts, e.g., a 1ms dropout on a 56k baud modem link can error up to ~58 bits

Worse on broadband, e.g, ADSL

Parity can detect single isolated errors but an even number of errors close together leaves parity unchanged, e.g., 0100 0001 => 0100 1101 => parity unchanged!

The form of parity we have seen is inadequate for burst errors

A possible solution is to compute “checksum” values, i.e., the parity of each bit-column over multiple bytes

Detecting Burst Errors - Example

Checksums vastly improve our error detection capabilities and allow for the detection of burst errors

1. To send the following message over a channel:

   (NB 7-bit ASCII without parity)

M	100 1101
e	110 0101
s	111 0011
s	111 0011
a	110 0001
g	110 0111
e	110 0101

2. Calculate bit-column parity "checksum" values, e.g., using even parity

   100 1011 = ASCII '**K**'

3. Send "Message**K**" instead of "Message"

This system detects all burst errors < 14 bits
Fails if an even number of errors occurs in a bit-column

Error Correcting Codes

The checksum codes we have seen (those with bit-column parity) could detect some burst errors

We can extend to an Error Correcting Code (ECC) by calculating both character and bit-column parity

M	**0**	100 1101
e	**0**	110 0101
s	**1**	111 0011
s	**1**	111 0011
a	**1**	110 0001
g	**1**	110 0111
e	**0**	110 0101
K	**0**	**100 1011**

1. ← character parity

↓ 2. bit-column parity

ECCs - Example

At transmitter we computer and transmit

Compute character parity bits, then bit-column parity value, then transmit message with these added bits

At receiver we re-compute, compare and correct if needed

Compute character parity bits, ignoring received parity bits

For each character, compare re-computed character parity with received parity

If not equal, ERROR somewhere in row

Re-compute bit-column parity value

Compare re-computed value with received value

If not equal, ERROR somewhere in column

If a single error detected, correct by inverting bit in message at (row, column)

ECCs - Graphically

A graphical representation allows us to think clearly about the potential issues and capabilities of a particular ECC

✓	0	**0**	100	1101	**M**
✓	0	**0**	110	0101	**e**
✓	1	**1**	111	0011	**s**
✗	0	**1**	110	0011	**c**
✓	1	**1**	110	0001	**a**
✓	1	**1**	110	0111	**g**
✓	0	**0**	110	0101	**e**
		0	**100**	**1011**	
		0	101	1011	
			✓✓✗	✓✓✓✓	

Key:

1101	– received message bits
0	– bit changed by noise
1011	– received parity bits
1011	– re-computed parity bits
✓✓✗	– comparisons

This ECC can detect and correct single errors

It can also detect, but not correct, certain combinations of multiple error

Common Memory Components

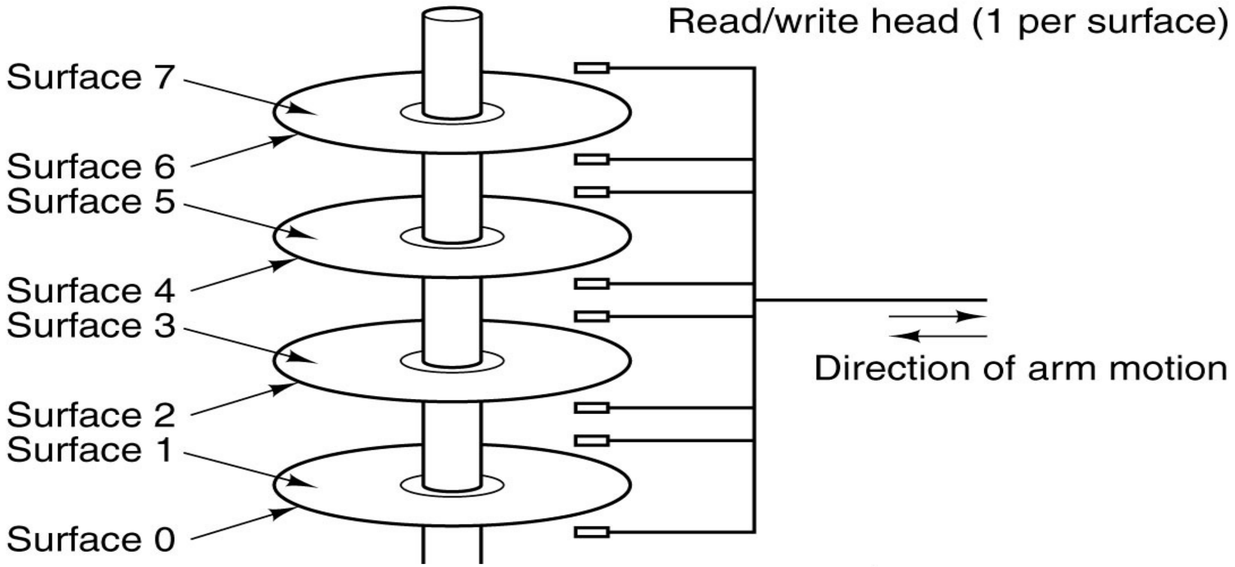
Magnetic Hard Disks - Platters, Surfaces and Heads

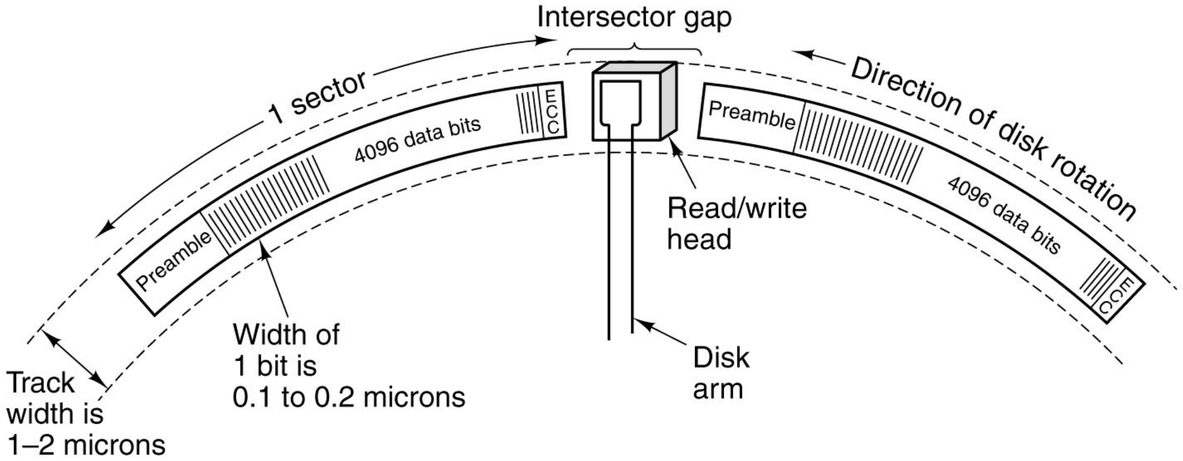
Record data by magnetising a thin film of ferromagnetic material on a disk

One track contains many sectors

Each sector is separated by an inter-sector gap.

Sector contains preamble to allow head to be synchronised before read/ write, data and ECC





Magnetic Hard Disks - Performance - What Can We Learn?

Seek times - time to move arm to correct track

- Typically about 5ms between different tracks but can be below 1ms for consecutive tracks

Rotational latency

- Constant angular velocity, i.e., motor spins at constant speed
- Drives available with rotational speeds of 5400 RPM, 7200 RPM or 10800 RPM
- Average delay (half a rotation) = 3 to 6ms

Sector read time is about 0.013ms, which means that seek time and rotational latency dominate access time.

Formatting (preambles, ECC etc...) reduces capacity by about 15%

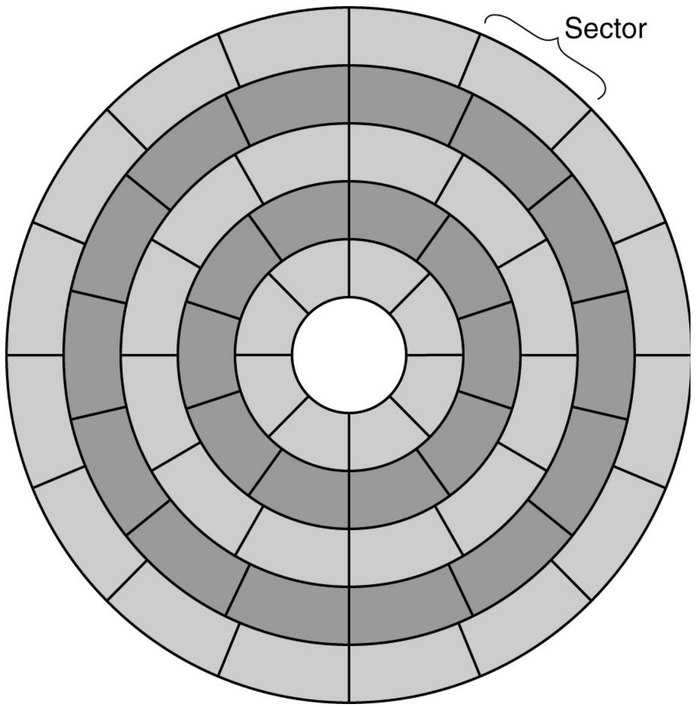
Hard Disks - Increasing Capacity with Zones - More To Learn?

Older drives have a constant number of sectors per track, which meant that, due to constant angular velocity, bit density is lower on outer tracks

Most Recent disks are zoned (10-30 zones), with more sectors in the longer (outer) tracks

- Many tracks per zone
- More sectors gives higher capacity

Graphic shows a disk with 5 zones



Optical Disc - Spiral Track

Originally designed to hold music

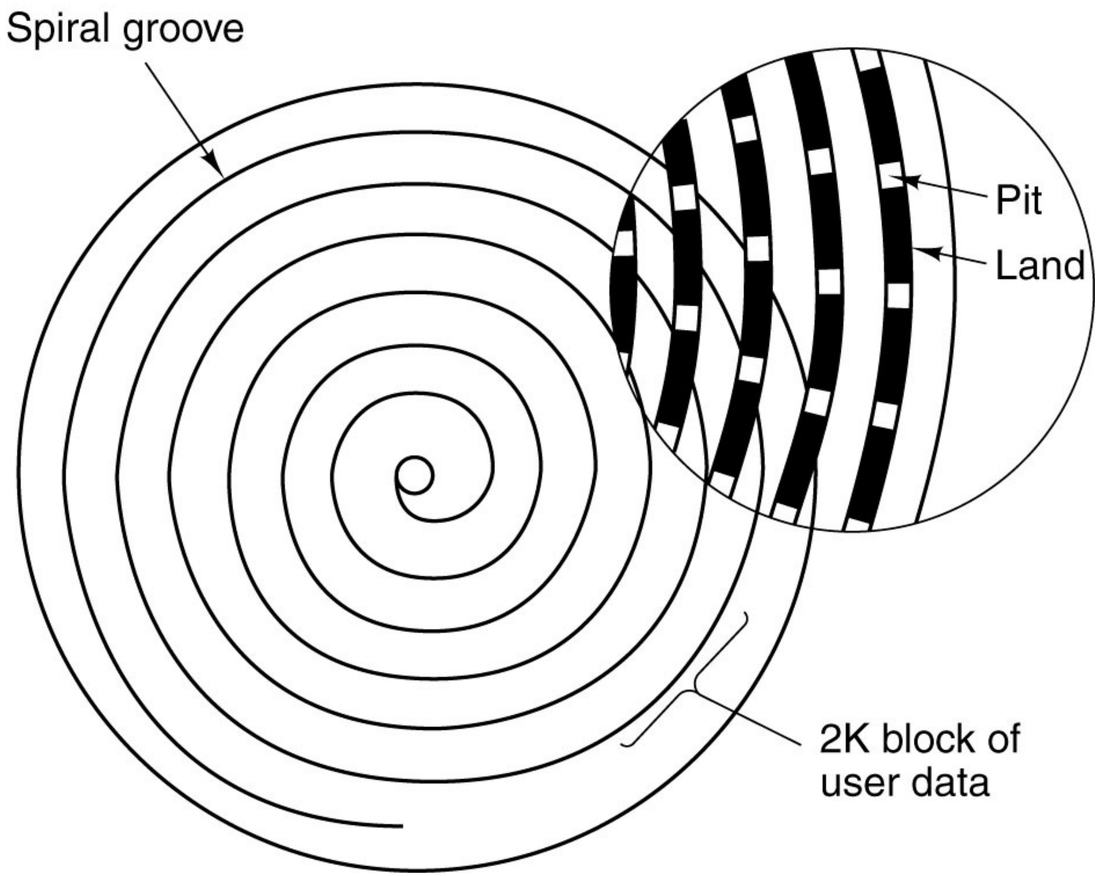
Max 74 mins - Beethoven's Ninth Symphony!

Spiral is 3.5 miles long if unwound

6,000 tracks/cm

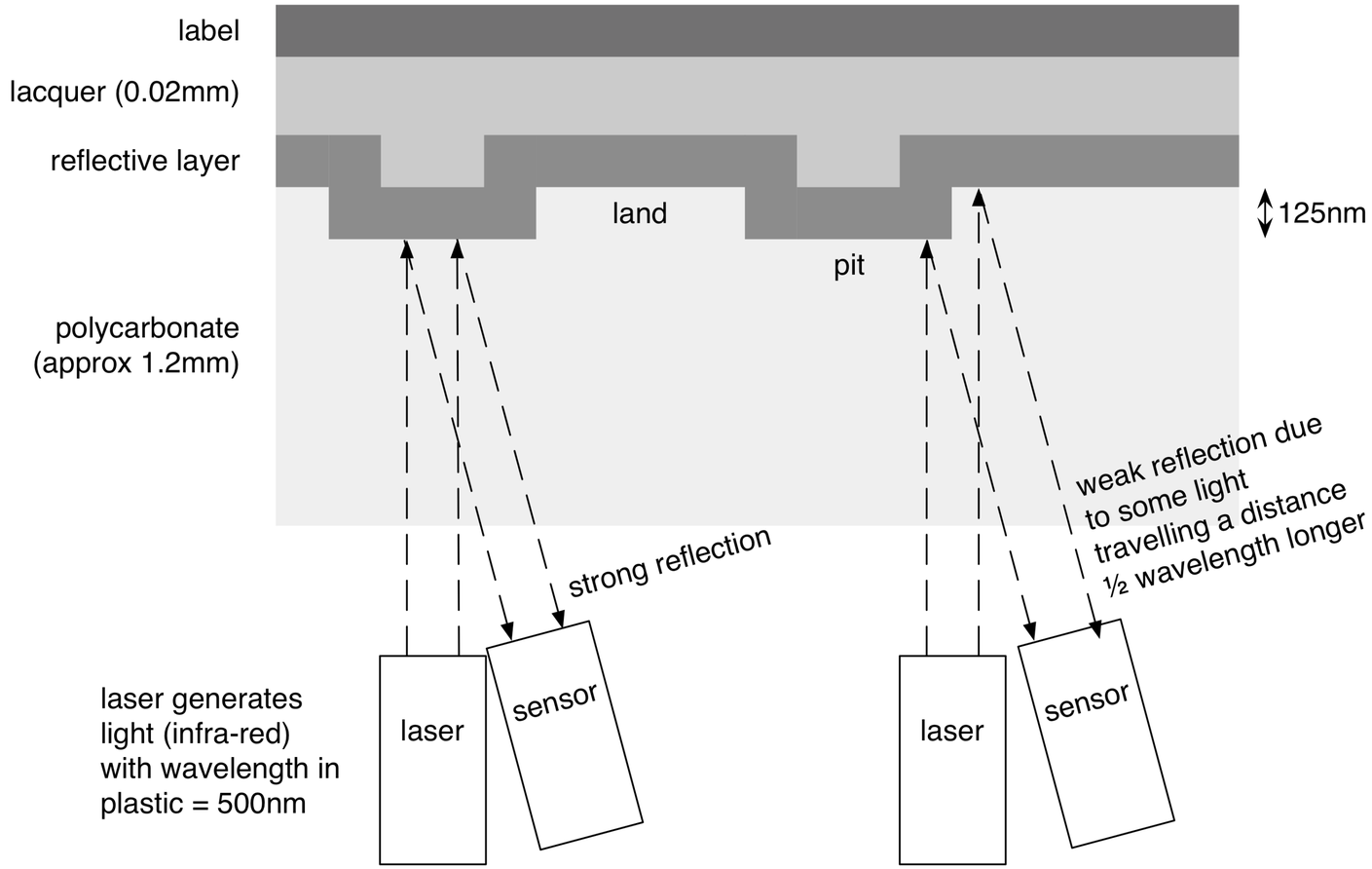
Data encoded as "pits" and "lands"

15,000 pits and lands/cm



Optical Disc - Spiral Tack Detail

Consider a much closer look at how op



Optical Disc - Data Encoding

A change in state (pit/land, land/pit) signifies data is a 1

Data represented as bytes, but each 8 bit byte is mapped to a 14 bit symbol (first level of CD-ROM ECC)

Symbols are chosen such that there are always two 0s between 1s and a max of ten 0s in succession

Of the 214 possibilities, only 267 values fit the rules

- This is more than sufficient for the 256 values (for the 8 bits) that are to be represented

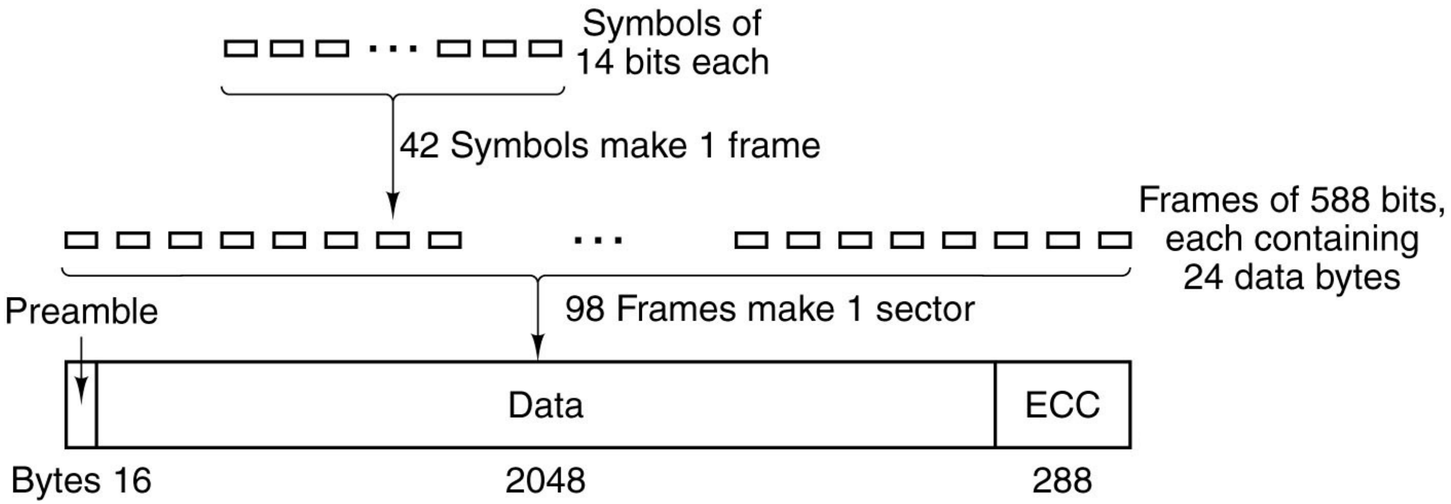
42 symbols make a frame (containing 24 bytes of data)

98 frames make a sector (containing 2048 bytes)

Optical Disc - Data Encoding Graphically

Multiple separate error correcting schemes

- Single bit errors corrected at symbol level
- Short burst errors corrected at frame level
- Any residual errors corrected at sector level



Optical Disc - Error Prevention

Most errors are burst and local, e.g., those caused by scratches

Cross Interleaved Reed-Solomon (CIRC) code is used to spread data out over the surface of the disk.

Burst errors of up to 2mm track can be corrected

Data accounts for only 28% of bits stored, thus we pay a heavy price for great reliability

Optical Disc - Performance

Constant linear velocity, i.e., motor spins at variable speed, to keep the pits and lands moving over the laser at a constant speed

For a “1x” drive, i.e., audio CD speed we have 120 cm/sec (data rate of 150kbytes / sec) and 200 RPM (outer) – 530 RPM (inner part of spiral)

An “8x” drive spins 8 times faster (1600-4000 RPM)

Above 12x speed, CAV is usually used, and the “x” number denotes data transfer rate at the outer edge of the disc, i.e., the max rate

Seek times averages are typically 80ms, which is at least an order of magnitude worse than hard disks

Computer Systems and Professions Practice Week 4 Checklist

To help you make the most of the first few weeks of the module, the checklist below provides a list of activities that you should have completed by the end of Week 4 of Term 1. The list is purely to help you plan your time so do not worry if you have or have not done everything already.

1. **Explore the Computer Systems and Professions Practice Canvas page.**

   https://canvas.bham.ac.uk/courses/77845

   Why? Everything you need for the module can be found on Canvas webpage, including lecture videos, lecture slides, lab sheets, continuous assignment and the contact details for all staff on the module.

2. **Watch the video lectures for the 01 Introduction and 02 Data Representation topics.**

   Why? The first lecture topic helps you understand what we will be doing in the module, the work you will be doing in labs and the arrangements for the coursework assignments. The second lecture topic is fundamental to lots of the content will cover in the next few weeks.

3. **Attend your assigned lectures.**

   Why? It is good to get used to attending lectures, since they are a primary means of engaging with academic materials. Moreover, they are an excellent supplement the video lectures and they provide great way to understand of what will be expected for your continuous assessment and examination.

4. **Complete the first C programming lab.**

   Why? The labs are an important complement to lectures and a great way to prepare for the continuous assessment. The labs are completed in your own time, which means that you can do them whenever you like, but if you're doing get them all done by the end of Week 6 then it's be a good idea to start one of them done at an early stage.

5. **(Optional) Get C working on your own computer.**

   Why? It will be much easier to complete coursework assignments with C working on your own computer. The module webpage provides suggestions for how to go about getting C working. This will be especially important if you are working remotely, making it a useful focus for your first lab if you have not done it by then.

6. **(Optional) Attempt the quiz for the 02 Data Representation topic.**

   Why? You can use the quizzes provided for each topic to gauge how much you have understood after watching each video lecture. You do not submit them for credit. You can use them for revision and we will talk about them when we are preparing for your exam. If you found the Data Representation quiz difficult then you might like to try working through the Data Representation worksheet - the additional resources are there to help you.

Whether you do or do not manage to get everything above completed by the end of Week 4, it is important not to panic. You have time and I will always be pleased to hear from you and happy to help you get up to speed.

Hope that helps,
Matt

Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 1 - I/O Mechanisms

Topic Outline

Memory mapped I/O

Synchronisation and transferring data with an I/O device

Busy-wait polling techniques

Handshaking protocols for data synchronisation

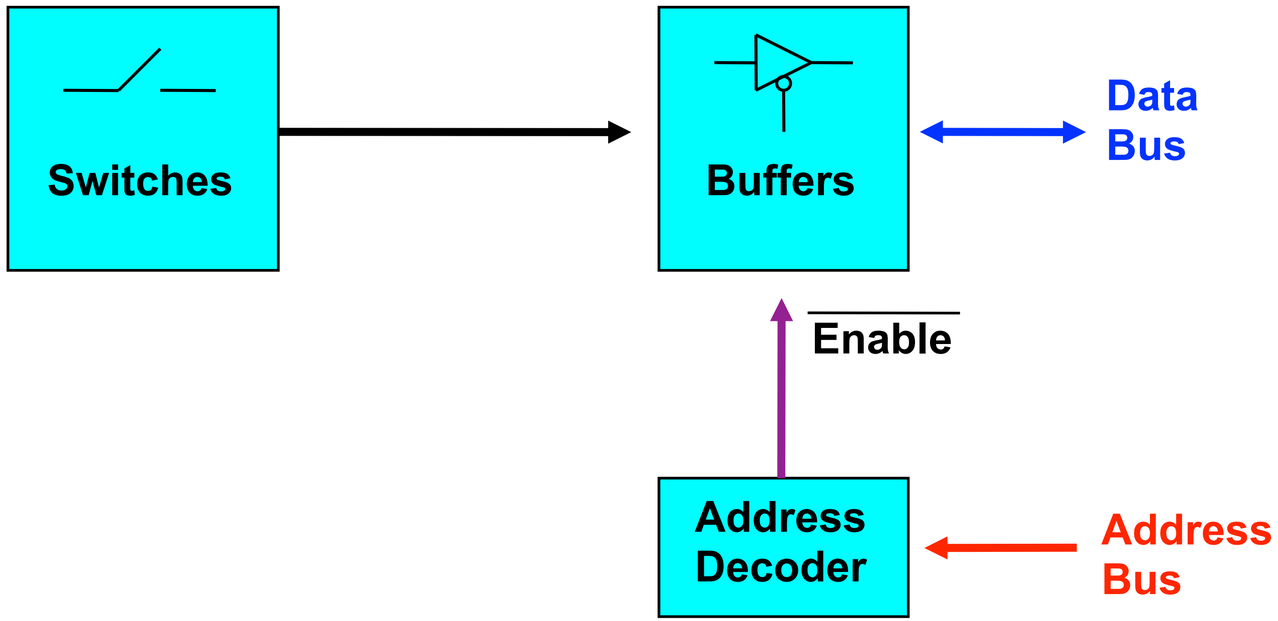
Interrupts and the principles of interrupt driven I/O

The differences between I/O DMA data transfer and programmed I/O

Memory Mapped I/O

Input Example

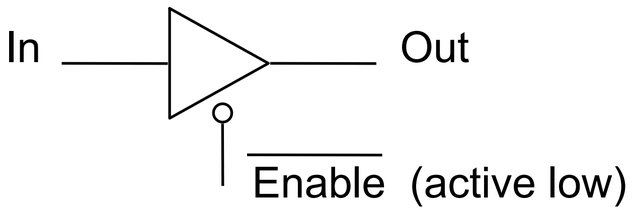
We've previously seen the principles underpinning the diagram below in relation to digital logic circuits



Reminder - Three-state Logic

The logic components considered so far have two possible logic states, i.e., 1 or 0

There is a further gate, termed a three-state buffer, whose output can be placed in a third state, known as UNCONNECTED



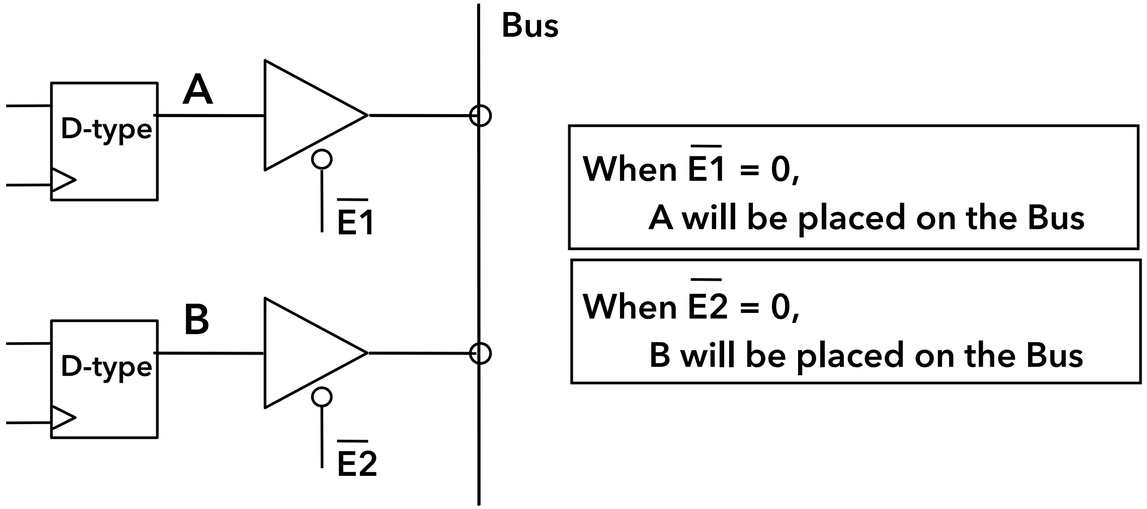
When $\overline{\text{Enable}}$ is high, the input is disconnected from the output

When $\overline{\text{Enable}}$ is low, the input is connected to the output

Reminder - Three-state Buses

Can use three state buffers to allow different sources of data onto a common BUS

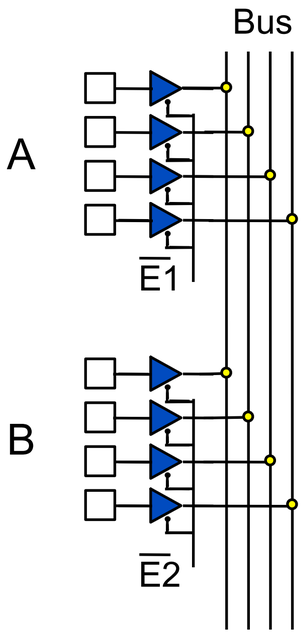
For example, consider a 1-bit wide bus



N.B. Should not have $\overline{E1}$=0 and $\overline{E2}$ = 0 at the same time!

Reminder - Multi-bit Bus

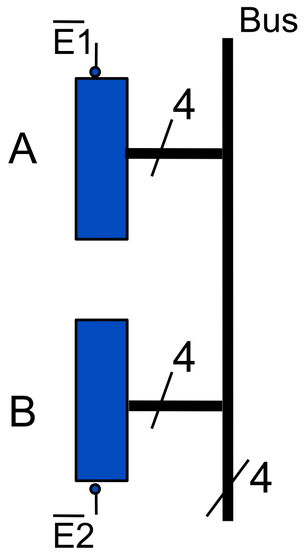
We can consider a 4-bit bus



When $\overline{E1}$ = 0,
A will be placed on the Bus

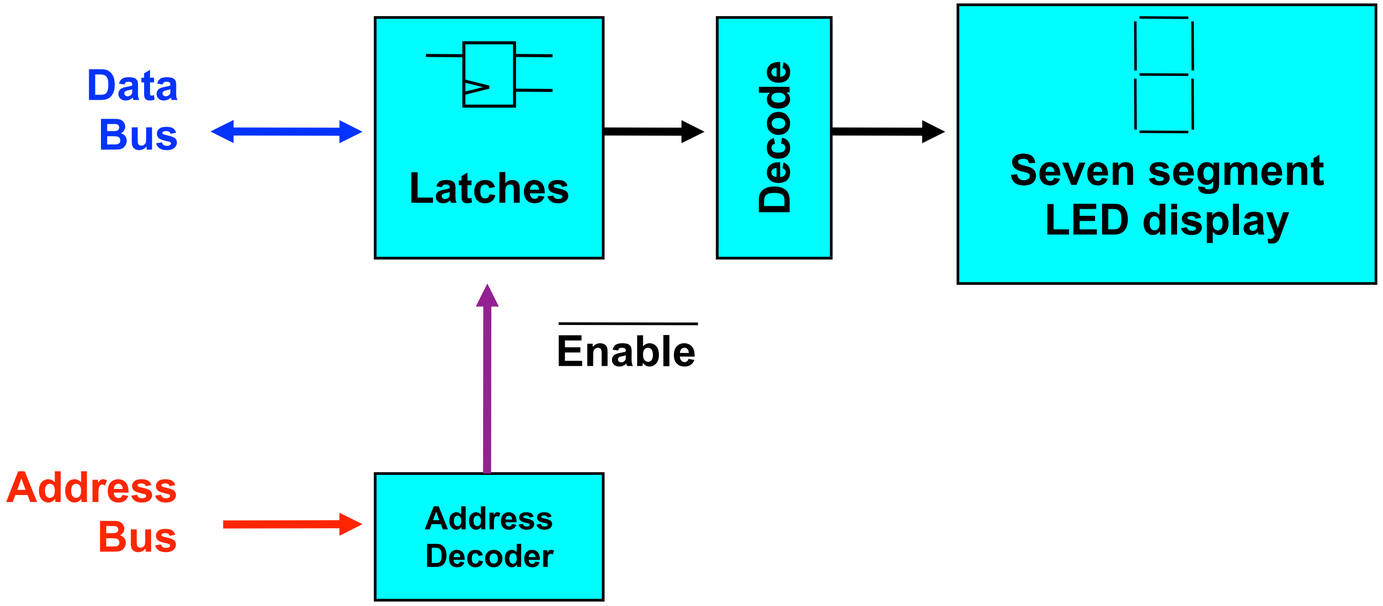
When $\overline{E2}$ = 0,
B will be placed on the Bus

Or, can represent this more concisely:



Output Example

Again, we've previously seen the principles underpinning the diagram below in relation to digital logic circuits



Memory-mapped I/O

The same address bus is used to address both memory and I/O devices

Memory (including registers) on I/O devices are mapped to address values

An I/O device can then operate as designed on the data given

When a CPU accesses a memory address, that memory address may be in physical memory (typically RAM) or be associated with the memory of an I/O device

Memory-mapped I/O Advantages

- Simpler than many alternatives
  - Particularly compared to port I/O (where a dedicated class of instructions are defined for performing I/O)
  - CPU requires less internal logic, which can help make the design and fabrication of a CPU cheaper
- Use of general purpose memory instructions
  - All addressing modes supported by a CPU are available to I/O

Memory-mapped I/O Disadvantages

Portions of memory address space must be reserved

Less of a concern as 64-bit processors, hence address spaces, have come to market

Still relevant where 16-bit (and sometimes 32-bit) processors are used, e.g., embedded and legacy systems

Synchronising with I/O Devices - Polling

Motivating Synchronisation

Most I/O devices are much slower than the CPU

Several factors to consider

- Read - Is there data to be read from the device?
- Write - Is the device ready to accept data?

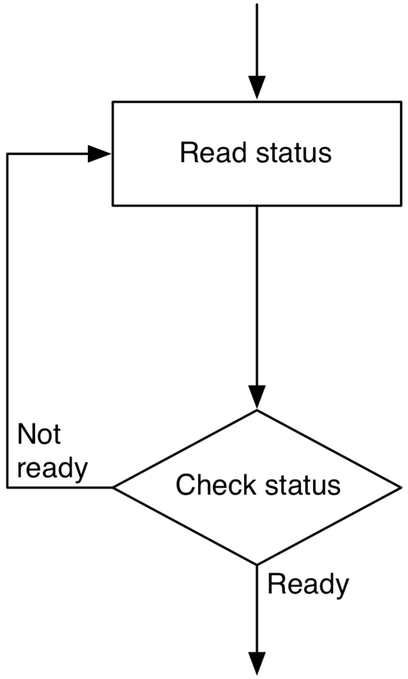
Consider a simple printer that takes a finite amount of time to print a character

- This time very much greater than the processor instruction time

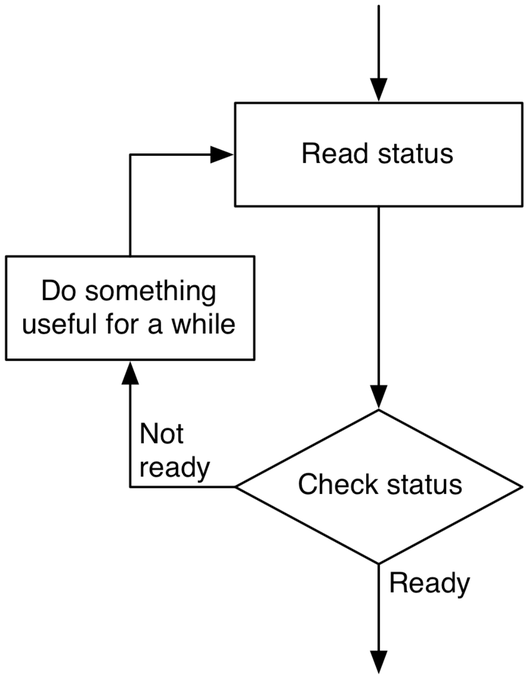
Polling and “Busy-wait” Polling

What are the practical advantages of these polling techniques?

What might we want to do whilst waiting for change in status?



Busy-wait polling



Polling, interleaved with another task

Polled I/O Advantages

- Simple software
  - A looping construct paired with some known checks
- Simple hardware
  - Support for notion of “ready” is all that is required

Polled I/O Disadvantages

Busy-wait polling wastes CPU time and consumes power

- Imagine you have a power constrained device, would this be where you want to spend your power?

Polling when interleaved with other tasks can lead to a significantly delayed response to device

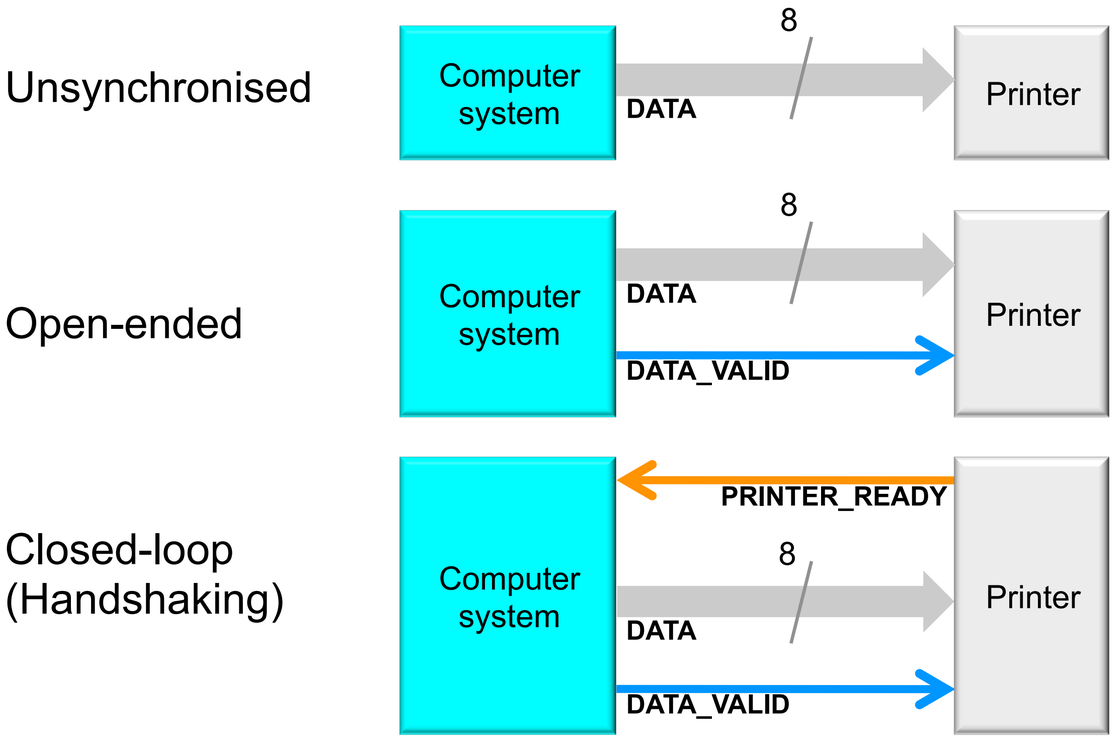
- Not a problem for some I/O situations but a serious issue if you're working in a hard real-time context

But sometimes it is good enough!

Synchronising with I/O Devices - Handshaking

Motivating Handshaking

It seems that we're adding complexity - why?

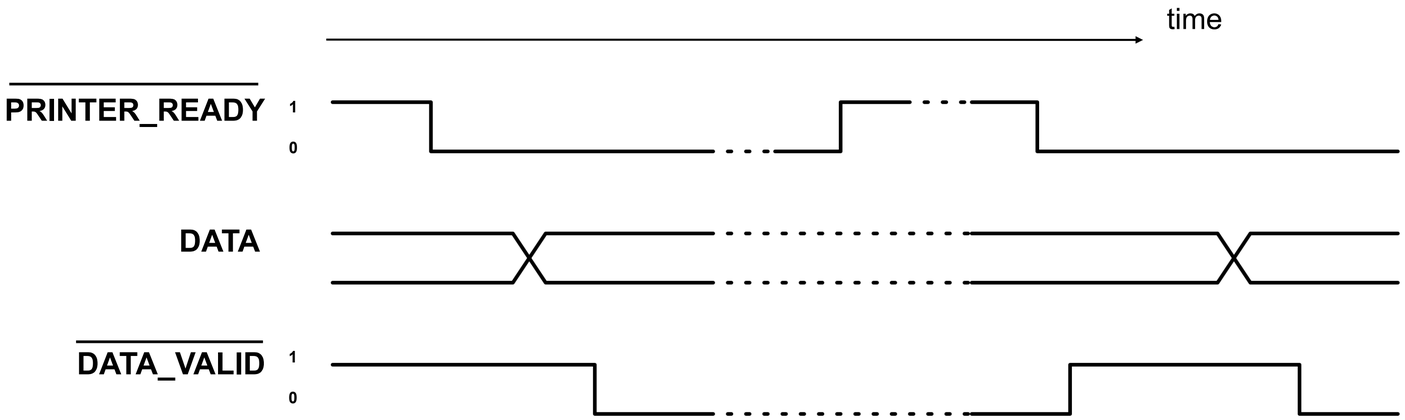


Handshaking - Timing Diagram

Computer system responds to the printer being ready by placing a new character on the data bus and signalling DATA_VALID

We could write code to get the CPU to do this

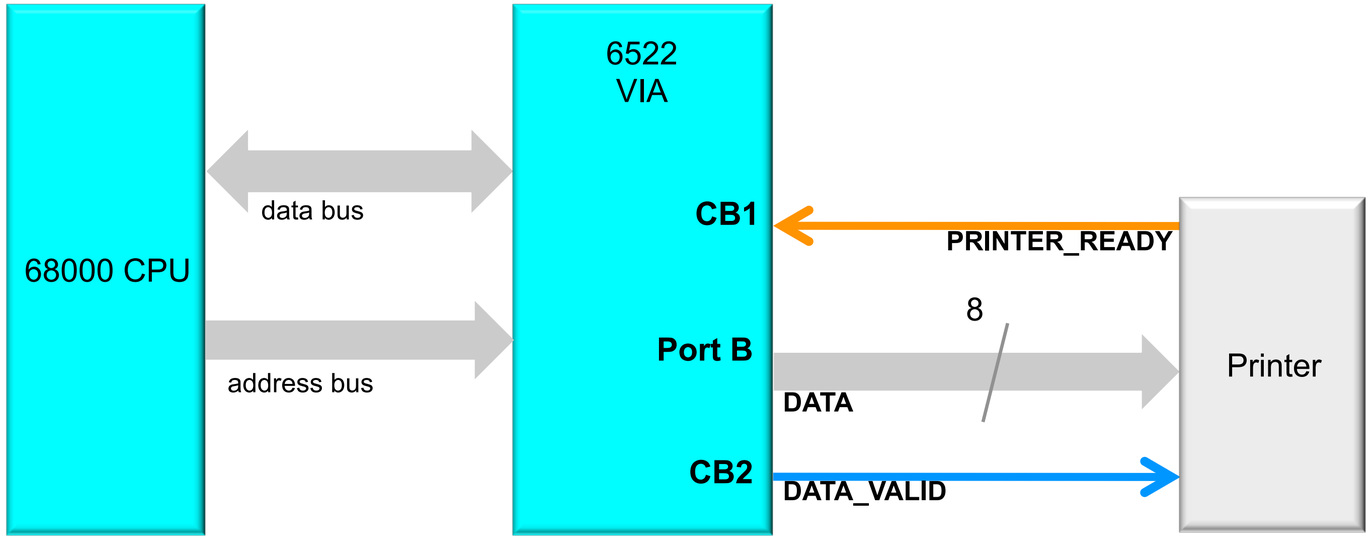
Alternatively, we could use the specialised hardware, which often requires fewer CPU instructions



Example - Using the 6522 VIA for Handshaking

The 6522 Versatile Interface Adapter (VIA) is a (relatively old) I/O IC that supports handshaking - this is just an example!

This is more intuitive than it might seem - consider how you'd act out handshaking



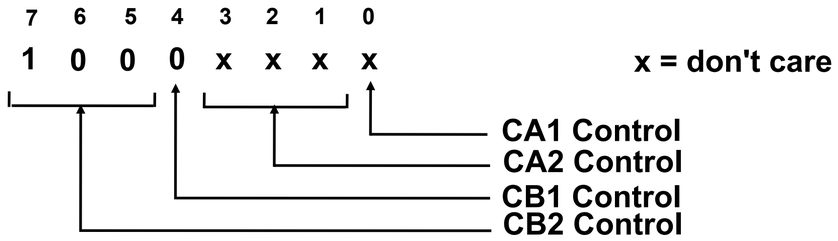
Setup for 6522 VIA Handshaking

There is a direct relationship between the diagram below and the block diagram we analysed previously

Note how setting bit values in particular locations allows us to determine function

Use: **PORT B as Output**
CB1 control line as PRINTER_READY
CB2 control line as DATA_VALID

Need to set up Peripheral Control Register (PCR) on VIA



How the 6522 VIA Performs Handshaking

PCR 1000xxxx

"CB2 handshake output mode"

"CB1 Interrupt Flag set by negative transition on CB1"

When PRINTER_READY is asserted, i.e., goes high to low

VIA sets IFR4, which can be later read by CPU

VIA sets DATA_VALID high, i.e., invalid

When CPU writes a value to ORB

VIA sets DATA_VALID to low (i.e., valid)

In this example, the CPU will repeatedly poll IFR to see if PRINTER_READY - we are not (yet) using interrupts.

Busy-wait Printing Program with Handshaking

```
| Print characters A to Z on hypothetical printer connected as per previous slides.
MAIN:     clr.b     VIA_DDRA                    | set PORT A = inputs
          move.b    #%11111111,  VIA_DDRB       | set PORT B = outputs
          move.b    #%10000000,  VIA_PCR        | initialise PCR
          move.b    #%01111111,  VIA_IER        | disable VIA interrupts
          move.b    #$41, D0                    | start with ASCII 'A' = $41

| now… when PRINTER_READY goes high to low, VIA sets DATA_VALID=invalid, and also sets IFR bit 4: check if so…
LOOP:     move.b    VIA_IFR, D1                 | read VIA interrupt flag byte, put it in D1
          and.b     #%00010000, D1              | is bit 4 set?
          beq       LOOP                        | if clear then not PRINTER_READY: loop
          move.b    D0, VIA_ORB                 | PRINTER_READY, so output character
                                                | (this write to ORB causes VIA to set DATA_VALID=valid)
                                                | printer will eventually set PRINTER_READY high again
          add.b     #1, D0                      | increment to next character
          cmp.b     #$5B, D0                    | are we at the end of the alphabet…
     bne  LOOP                                  | …(ASCII 'Z' = $5A)? If not, do it again
```

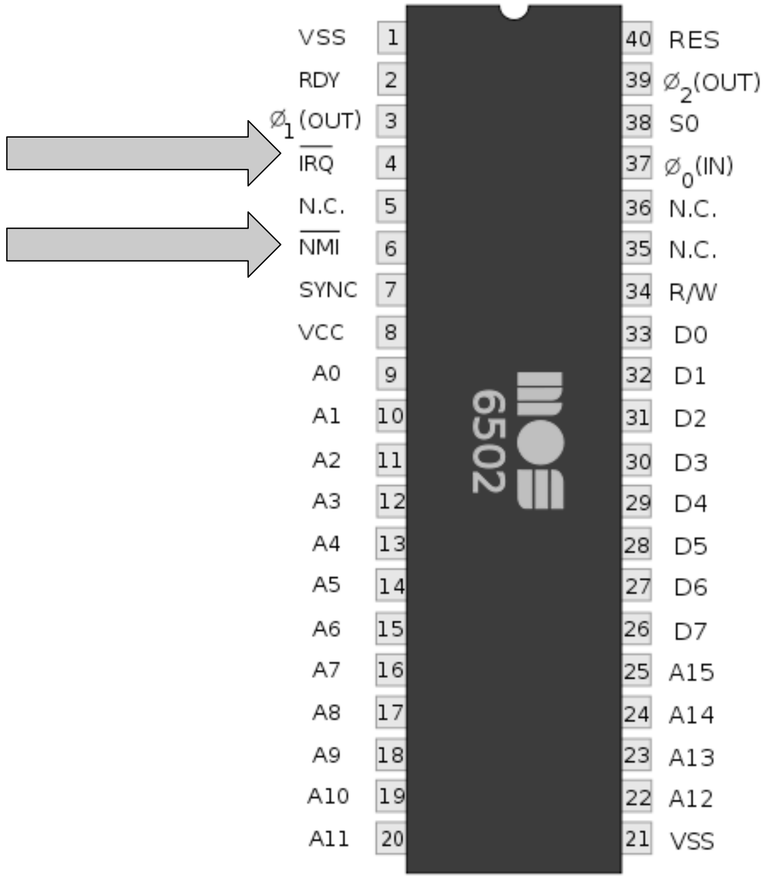
Synchronising with I/O devices - Interrupts

Interrupts - An Input on the CPU

Consider the 6502 processor, designed in 1975, used in the BBC micro and others

IRQ (Interrupt Request) and NMI (Non-maskable Interrupt) inputs

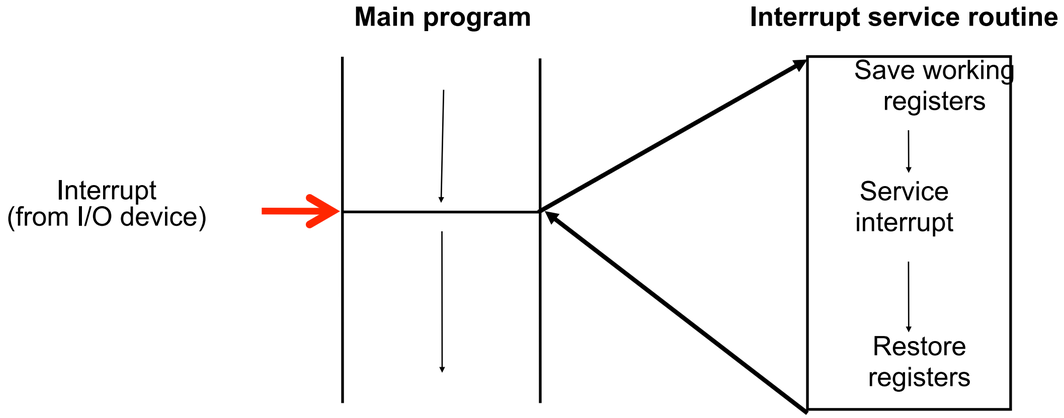
Code can disable response to IRQ, as it is only a request, but NMI cannot be disabled or "masked out"



Effect of an Interrupt Input

CPU normally executes instructions sequentially, unless a jump or branch is made

An interrupt input can force CPU to jump to a service routine



Can therefore make it appear (to humans) that the CPU is performing two or more tasks "simultaneously"

Interrupt Handling Sequence

1. External device signals interrupt

INTERRUPT RESPONSE

1. CPU completes current instruction
2. Push PC onto Stack
3. Push Status Register(s) onto Stack
4. Load PC with address of Interrupt Handler

Context switch

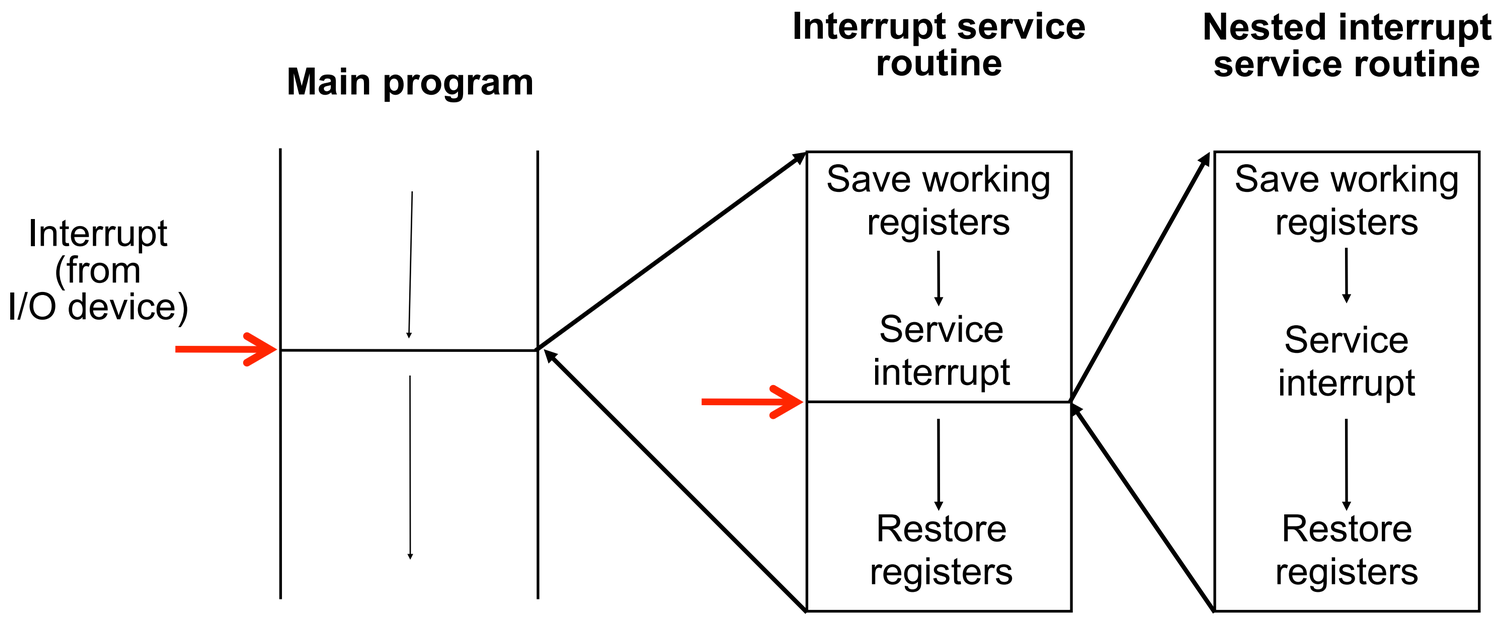
RETURN FROM INTERRUPT

1. Pop PC from Stack
2. Pop Status Register(s) from Stack
3. Load PC with popped return address

Context switch

Nested Interrupts

An interrupt service routine can be interrupted



Interrupts for I/O Examples

Switches can be connected to IRQ

If using multiple switches we can OR them all together to form IRQ and then check the individual switch states within the service routine

A hard drive can generate an interrupt when data, requested some time earlier, is ready to read

A timer can generate an interrupt every 100ms and the service routine can then read an sensor input

A printer can generate an interrupt when it is ready to receive the next character to print, as we will see

Interrupts - Advantages and Disadvantages

Advantages

Fast response

No wasted CPU time / battery power

Disadvantages

All data transfers still controlled by CPU

More complex hardware and software

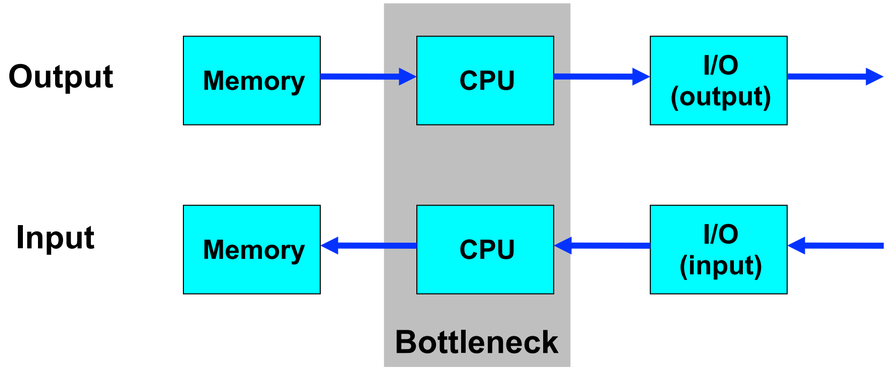
Understanding I/O mechanisms is vital - no single best!

Direct Memory Access (DMA)

Motivating Direct Memory Access (DMA)

The CPU is a bottleneck for I/O

Programmed I/O techniques, as in all the examples we've seen so far, are slowed down by the CPU



DMA avoids the CPU bottleneck, thus speeding up the transfer of data to memory

Why Use DMA?

DMA is used where large amounts of data must be transferred at high speed

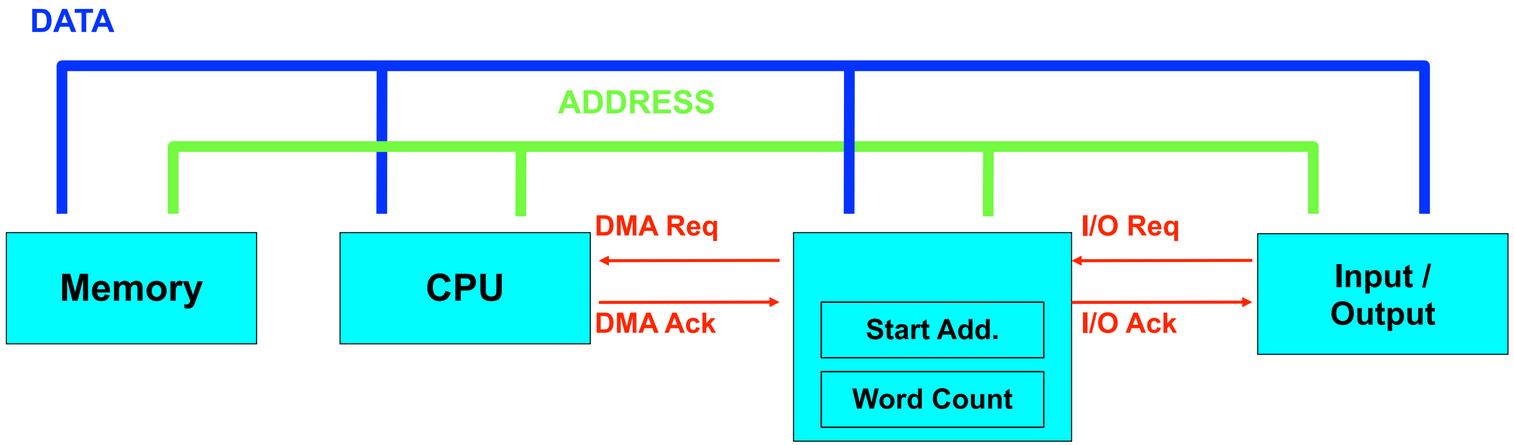
Control of the system buses is surrendered by the CPU to a DMA Controller (DMAC).

- The DMAC is a dedicated device that controls the three system buses during the data transfer.
- The DMAC is optimised for one operation, i.e., data transfer.
- The CPU is more general purpose, it both transfers data and is a processor of information.

DMA-based I/O can be more than 10 times faster than CPU-driven I/O

DMA Operation

1. DMA transfer requested by I/O
2. DMAC passes request to CPU
3. CPU initialises DMAC
   i) Input or Output,
   ii) Start address → DMAC Address Reg,
   iii) Number of words to transfer → Count Reg,
   iv) CPU enables DMAC
4. DMAC requests use of system buses
5. CPU responds with DMA Ack when it's ready to surrender buses



DMA Modes of Operation

Cycle Stealing

DMAC uses the system buses when they are not being used by the CPU - usually by “grabbing” available memory access cycles not used by the CPU

Burst Mode

DMAC requires system buses for extended transfer of large amount of data at high speed and “locks” the slower CPU out of using the system buses for a fixed time or until the transfer is complete or the CPU receives an interrupt from a device of greater priority

DMA Organisation

Possible to consider multiple organisations for the incorporation of DMA

Single bus, detached DMA

All modules (DMA, I/O, memory and processor) share the same system bus

The DMA module, mimicking the processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module

This implementation is straightforward to implement for most bus-based systems but is inefficient

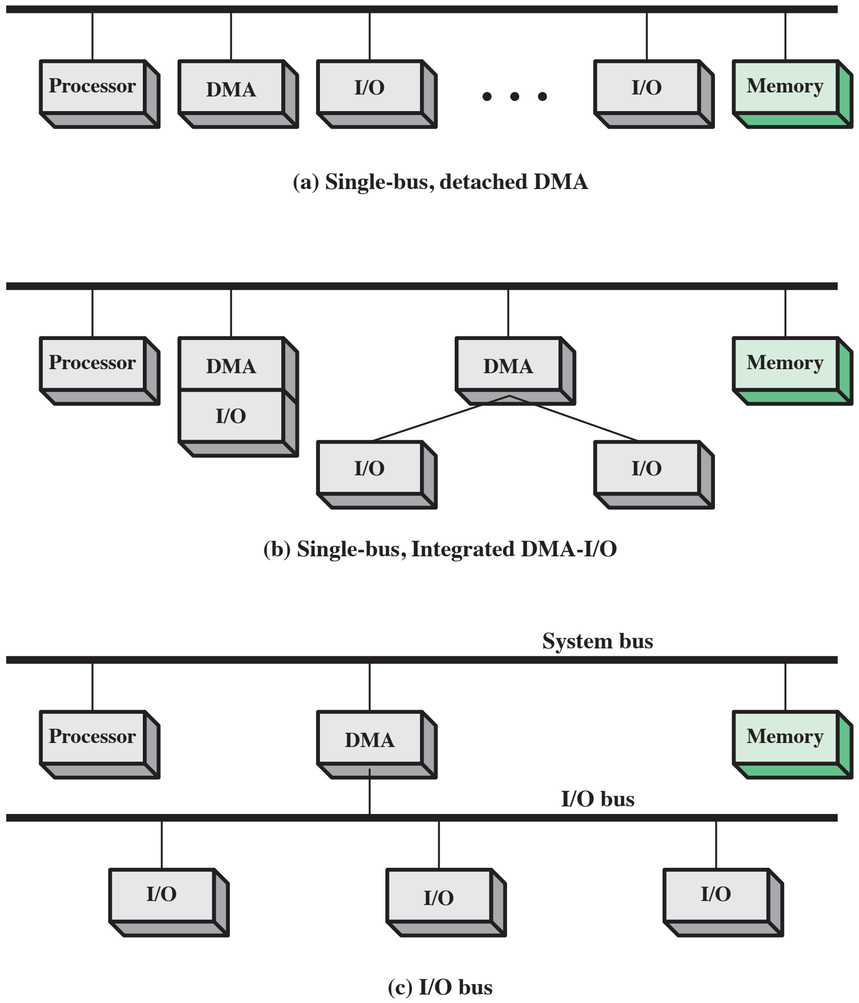
As with processor-controlled I/O, each word transfer takes two bus cycles

DMA Organisation

I/O Bus

Reduces number of I/O interfaces in the DMA module to one

System bus the DMA module shares with the processor and memory is used only to transfer data to and from memory



Reproduced from: W. Stallings, "Computer Organization and Architecture", 9th Edition, Pearson, 2013

I/O Mechanisms

Memory-mapped input and output devices can be accessed in the same way as RAM, at special address locations

Polled I/O is a technique for scheduling input and output, where the CPU repeatedly checks if this is necessary

I/O devices are usually much slower than the CPU, so **handshaking techniques** must be used to coordinate CPU and device to synchronise the transfer of data

Interrupts are a way to avoid polled I/O by diverting the CPU to a special I/O routine – quickly and only when necessary

If necessary, a **DMA** controller can be used instead of the CPU to transfer I/O data into/from memory - this can be faster than the CPU but at additional hardware cost

Which technique to use? Depends on amount of data, required transfer speed, budget, power, physical size, complexity - this is **computer system design**