CS1Q Computer Systems
Lecture 1

Prof. Chris Johnson

S141 Lilybank Gardens, Department of Computing Science, University of Glasgow, Scotland.
johnson@dcs.gla.ac.uk. http://www.dcs.gla.ac.uk/~johnson

Notes prepared by Dr Simon Gay

Aims

- To understand computer systems at a deeper level: general education for life in a technological society.
- Foundation for further CS modules:
  - Computer Systems 2
  - Operating Systems 3
  - Networked Systems Architecture 3
  - Computer Architecture 4

Books

- Essential: *Computer Science Illuminated* by N. Dale & J. Lewis.
- Supplementary notes will be produced.
- More detail on digital logic: *one* of
  - *Computers from Logic to Architecture* by R. D. Dowsing, F. W. D. Woodhams & I. Marshall (also useful for Level 2 CS)
  - *Digital Fundamentals* by T. Floyd

Other Reading

- *The New Turing Omnibus* by A. K. Dewdney
  - a tour through many and varied CS topics
- *Gödel, Escher, Bach: An Eternal Golden Braid* by D. Hofstadter
  - either love it or hate it: includes logic, computability, programming fundamentals, and much more
- *Code* by Charles Petzold
  - excellent explanation of computing fundamentals

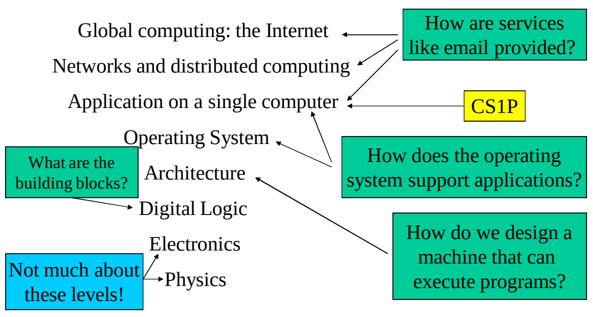
Course Plan

Preparatory reading will be assigned for each lecture - some is necessary, some is for general interest.

Look at Moodle (navigate from the Level 1 CS homepage) to find out the preparatory reading and other Information.

You will probably find it useful to make some notes during lectures.

A Hierarchical View


Global computing: the Internet
Networks and distributed computing
Application on a single computer
Operating System
Architecture
Digital Logic
Electronics
Physics
How are services like email provided?
CS1P
How does the operating system support applications?
How do we design a machine that can execute programs?
What are the building blocks?
Not much about these levels!

Information Processing

- Everything that computers do can be described as *information processing*.
- Information is also processed by other devices, e.g. CD player, television, video recorder, …
- Computers are *programmable*: the way they process information can be changed.
- Computers represent information *digitally*.

Digital Information

- *Digital* means *represented by numbers*. Ultimately, *binary* numbers (0, 1) are used.
- The alternative is an *analog* representation, meaning that information is represented by a *continuously variable physical quantity*.

Examples of Analog Devices

- Traditional clock with hands
- Car speedometer with a needle
- Video tape recorder
- Record player (remember those?)
- Radio and television
- Traditional film camera

Examples of Digital Devices

- Digital watch
- Car speedometer with a digital display
- DVD player/recorder
- CD or MP3 player
- Digital radio, digital television
- Digital camera

The Binary System

The *decimal* or *base 10* system uses digits 0,1,2,3,4,5,6,7,8,9.
The *column values* are powers of 10:

1000	100	10	1
2	4	7	6

means $2\times10^3+4\times10^2+7\times10^1+6\times10^0$

The *binary* or *base 2* system uses digits 0,1.
The *column values* are powers of 2:

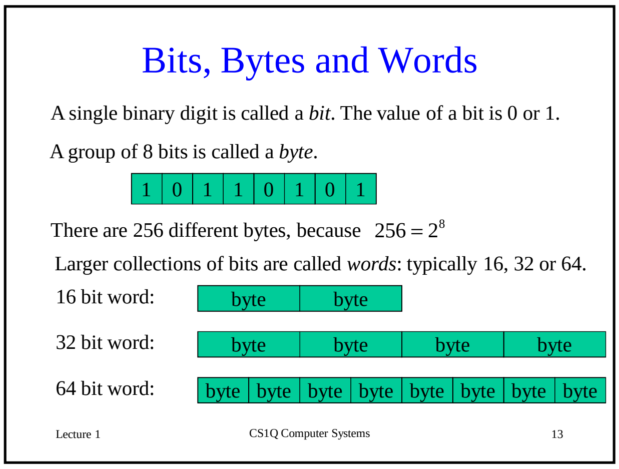
8	4	2	1
1	1	0	1

means $1\times2^3+1\times2^2+0\times2^1+1\times2^0=13$

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

Why is binary used?

- Because it's easy to distinguish between two states:
  - high or low voltage
  - presence or absence of electric charge
  - a switch in the on or off position



Bits, Bytes and Words

A single binary digit is called a *bit*. The value of a bit is 0 or 1.

A group of 8 bits is called a *byte*.

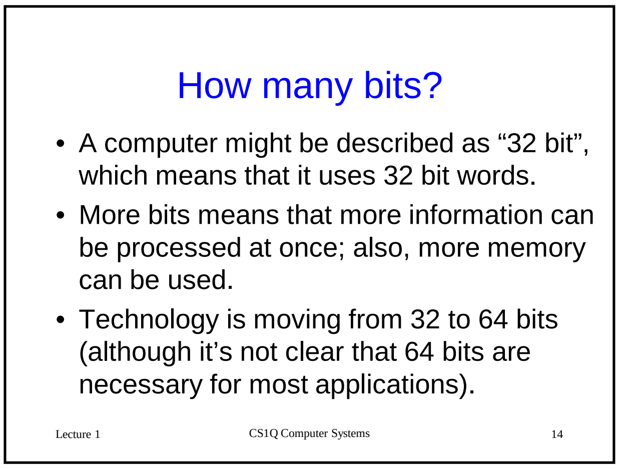
1	0	1	1	0	1	0	1

There are 256 different bytes, because $256 = 2^8$

Larger collections of bits are called *words*: typically 16, 32 or 64.

16 bit word:	byte	byte						
32 bit word:	byte	byte	byte	byte				
64 bit word:	byte	byte	byte	byte	byte	byte	byte	byte

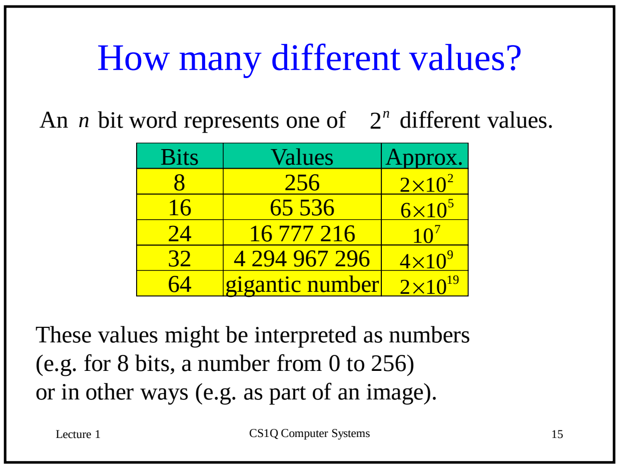
Lecture 1 CS1Q Computer Systems 13



How many bits?

- A computer might be described as "32 bit", which means that it uses 32 bit words.
- More bits means that more information can be processed at once; also, more memory can be used.
- Technology is moving from 32 to 64 bits (although it's not clear that 64 bits are necessary for most applications).

Lecture 1 CS1Q Computer Systems 14



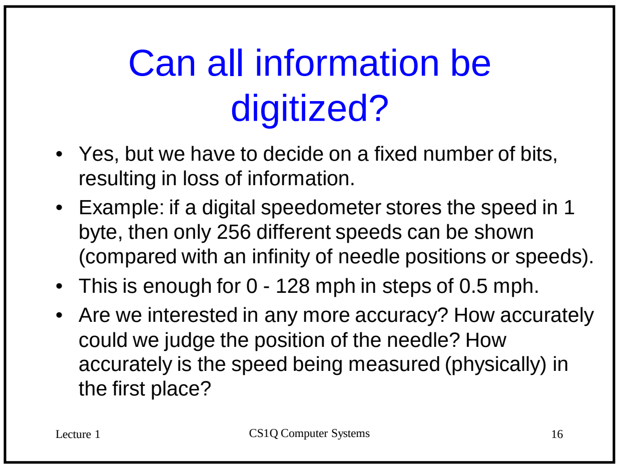
How many different values?

An n bit word represents one of 2^n different values.

Bits	Values	Approx.
8	256	2×10^2
16	65 536	6×10^5
24	16 777 216	10^7
32	4 294 967 296	4×10^9
64	gigantic number	2×10^{19}

These values might be interpreted as numbers (e.g. for 8 bits, a number from 0 to 256) or in other ways (e.g. as part of an image).

Lecture 1 CS1Q Computer Systems 15



Can all information be digitized?

- Yes, but we have to decide on a fixed number of bits, resulting in loss of information.
- Example: if a digital speedometer stores the speed in 1 byte, then only 256 different speeds can be shown (compared with an infinity of needle positions or speeds).
- This is enough for 0 - 128 mph in steps of 0.5 mph.
- Are we interested in any more accuracy? How accurately could we judge the position of the needle? How accurately is the speed being measured (physically) in the first place?

Lecture 1 CS1Q Computer Systems 16

Why Digital?

- So that all kinds of information can be stored and processed in a uniform way. Examples:
  - video and audio information can be stored on a DVD (or a magnetic disc, or a computer memory) and replayed using suitable software
  - any digital information can be compressed or encrypted using standard algorithms
- To exploit the distinguishability of 0 and 1
  - e.g. digital radio suffers less from interference, and bandwidth can be increased

Information Representation: Numerical

Positive integers: straightforward, use binary
• known range: easy, fixed number of bits for each number
e.g. 16 bits give us the range 0..65535
(*implemented in hardware; details later*)
• unlimited size: more complicated, work with sequences of bytes (*implemented in software*)

Positive and negative integers: use *two's complement* (later)

Real numbers (non-integers): complicated, some details next lecture.

Information Representation: Text

Fix the set of possible characters, decide on the appropriate number of bits, and assign a binary number to each character. Text is represented by a sequence of characters.

ASCII: the standard for many years. 128 characters, 7 bits each.
Later extended to an 8 bit format to include accents and more symbols.

ASCII is biased towards the English language, and is being replaced by Unicode, a 16 bit format with 65536 characters.

Documents are often represented in formats which are not plain text.
E.g. Microsoft Word files and PDF files contain formatting information, images, tables etc.

Data Compression

It is often useful to *compress* large data files. The book describes three kinds of compression:
• keyword encoding
• run length encoding
• Huffman encoding
Another is Lempel-Ziv compression: similar to keyword encoding, but all repeated strings become keywords.

These are all examples of *lossless* or *exact* compression: decompressing takes us back where we started.

Inexact or *lossy* compression is often used for image and sound files: decompression does not result in exactly the original information, but this can be acceptable if the differences are too small to notice.

Run Length Encoding Puzzle

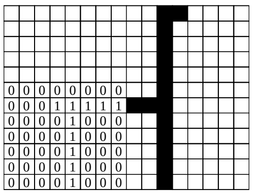
How does this sequence continue?

1
11
21
1211
111221
312211
13112221
…

Each line is a run length encoding of the line above. The increasing length of the lines illustrates the fact that run length encoding is not good at compressing data with many short runs.

Information Representation: Images

Straightforward idea: the *bitmap*, a rectangular grid of *pixels* (picture elements). One bit per pixel gives a black and white image.

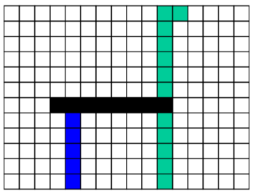


0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0

The whole image can be represented as a sequence of bytes: this part (in decimal) is 0, 31, 8, 8, 8, 8, 8.

Information Representation: Images

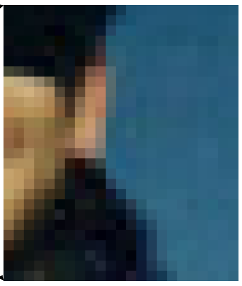
For more colours, use more bits per pixel. Here, for four colours, two bits are needed per pixel.



The more pixels we use, and the more colours, the more detailed and accurate the image can be (and the more space is needed to store it).

Digital Photography

An image is represented digitally by a grid of *pixels*.



The accuracy of the colours depends on the number of bits per pixel (but how many colours can the eye perceive?)

Image Formats

Up to 24 bits per pixel (giving over 16 million colours) is common. A compromise is to use a *colour palette*: for a particular image, select (say) 256 colours from the full range, then use 8 bits per pixel.

Compression is important because images can be large files. Some forms of lossless compression can work well: e.g. run length encoding is good if there are large blocks of the same colour; this is true for some kinds of image.

GIF (Graphics Interchange Format) uses a bitmap representation with 8 bit colour (using a palette) and Lempel-Ziv-Welch compression (lossless). It's particularly good for line drawings.

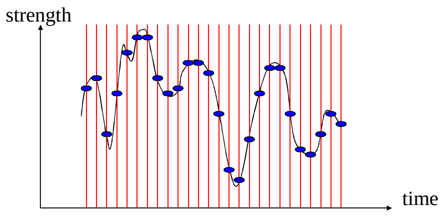
Image Formats

The JPEG (Joint Photographic Experts Group) format uses the *discrete cosine transformation* to convert a bitmap into a representation based on combinations of waveforms. This gives *lossy* but *adjustable* compression. A JPEG image must be converted back into a bitmap in order to be displayed on screen or printed.

JPEG is designed for good compression of images with smooth colour variations - for example, many photographs. The inaccuracy of the compression tends to smooth things out even more. JPEG is not so good for images with sharp edges, such as line drawings.

Information Representation: Sound

Sound is a waveform:



which can be sampled at some (suitably high) frequency and converted into a sequence of numbers (as accurately as we want). Playing the numbers through a digital-analog converter recovers sound.

Information Representation: Sound

The raw digital representation of the sound must be stored in a suitable format. Two formats are important at the moment:

Audio CD: an exact encoding, suitable for recording onto compact disk.

MP3 (Moving Picture Experts Group, audio layer 3): uses *lossy* compression to significantly reduce the size of audio files. The information lost during compression corresponds to parts of the sound which would not be (very) noticeable to the human ear. *Lossless* compression (Huffman encoding) is then used for further shrinkage.

MP3 representation is about one tenth of the size of audio CD.

CS1Q Computer Systems
Lecture 2

Binary Numbers

We'll look at some details of the representation of numbers in binary.
• unsigned integers (i.e. positive integers; this is probably revision)
• signed integers (i.e. positive and negative integers)
• fractions
• floating point numbers

It's important to understand the binary representation of unsigned and signed integers.

We won't be doing any work with floating point numbers, but it's interesting to see some of the complexities.

Converting Binary to Decimal

Converting binary numbers to decimal is easy: just add up the values of the columns which contain 1.

128	64	32	16	8	4	2	1	
1	0	1	1	0	1	0	1	= 181

128	64	32	16	8	4	2	1	
0	1	1	0	1	1	0	0	= 108

128	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	1	= 255

Converting Decimal to Binary
Method 1

First work out how many bits are needed.

If the number is at least 2^n but less than 2^{n+1} then $n+1$ bits are needed.

Examples:

103 is at least 64 but less than 128; it needs 7 bits

32 is at least 32 but less than 64; it needs 6 bits

257 is at least 256 but less than 512; it needs 9 bits

1000 is at least 512 but less than 1024; it needs 10 bits

Converting Decimal to Binary
Method 1

Work out the column values for the number of bits needed.
Example: 103, using 7 bits.

64	32	16	8	4	2	1
1	1	0	0	1	1	1

Starting from the left, enter a 1 if the number is at least as big as the column value; otherwise 0. If 1 is entered, subtract the column value. Repeat for each column.

More bits can be used: just put zeros in front.

128	64	32	16	8	4	2	1
0	1	1	0	0	1	1	1

Converting Decimal to Binary
Method 2

This method produces the binary digits from right to left. If the number is odd, enter 1 and subtract 1; if the number is even, enter 0. Divide the number by 2 and repeat.

Example: 237

128	64	32	16	8	4	2	1
1	1	1	0	1	1	0	1

Check: 128+64+32+8+4+1 = 237.

Hexadecimal

Hexadecimal, also known as *hex*, is base 16.
Each digit represents a number from 0 to 15.
The letters A to F (or a to f) are used for digits 10 to 15.

256	16	1
3	C	9

$= 3\times16^2 + 12\times16^1 + 9\times16^0 = 969$

Each hex digit corresponds to 4 bits:

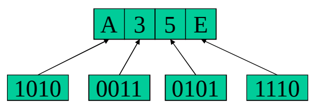
0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111

8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

A hex number is best thought of as an abbreviation for a binary number.

The number of bits can easily be seen (4 times the number of hex digits) but the number itself is shorter.



Addition in Binary

Just like in decimal: work from right to left, carrying to the next column if necessary.

```
 1  0  1  0  1  0  1  1          171
 0  1  0  0  1  1  1  0    +      78
-------------------------
 1  1  1  1  1  0  0  1          249
-------------------------
          1  1  1
                   carry
```

Working Within a Word Size

Usually a computer does addition on words of a fixed size.
A carry beyond the leftmost column is an *overflow*, which might be detectable, and the result is calculated *modulo 256* (with 8 bits).

```
    1  0  1  0  1  0  1  1          171
    0  1  0  1  1  1  1  0    +      94
   -------------------------
    0  0  0  0  1  0  0  1          9 instead of 265
   -------------------------
 1  1  1  1  1  1  1
overflow
```

265 divided by 256 = 1 remainder 9

With 16 bit words: addition modulo 65536, etc.

Unsigned or Signed?

Everything we have said so far applied to *unsigned* numbers: we are simply working with positive integers.

If we want to work with both positive and negative integers then we need to be able to distinguish between them: we need *signed* numbers.

We will now look at the representation of negative numbers in binary.

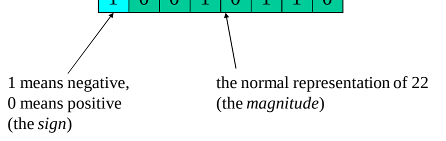
Negative Numbers in Binary

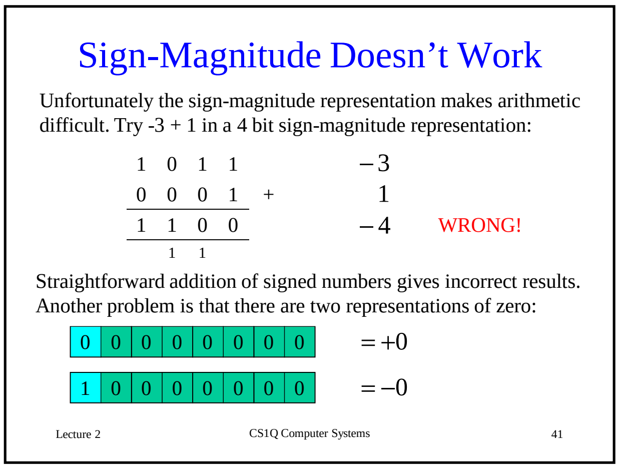
We need a representation of negative numbers in binary. On paper we might write

$$-10110_2 \quad \text{for} \quad -22_{10}$$

but how do we represent the minus sign within a byte or word?

The obvious idea is the *sign-magnitude representation*:


1 0 0 1 0 1 1 0
1 means negative, 0 means positive (the *sign*)
the normal representation of 22 (the *magnitude*)



Sign-Magnitude Doesn’t Work

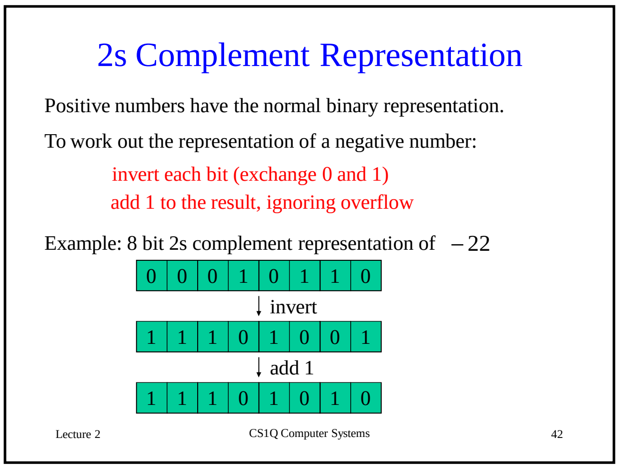
Unfortunately the sign-magnitude representation makes arithmetic difficult. Try -3 + 1 in a 4 bit sign-magnitude representation:

```
  1  0  1  1         −3
  0  0  0  1   +      1
  ----------
  1  1  0  0         −4     WRONG!
  ----------
     1  1
```

Straightforward addition of signed numbers gives incorrect results. Another problem is that there are two representations of zero:

0	0	0	0	0	0	0	0	= +0
1	0	0	0	0	0	0	0	= −0

Lecture 2 CS1Q Computer Systems 41



2s Complement Representation

Positive numbers have the normal binary representation.

To work out the representation of a negative number:

invert each bit (exchange 0 and 1)

add 1 to the result, ignoring overflow

Example: 8 bit 2s complement representation of -22

0	0	0	1	0	1	1	0

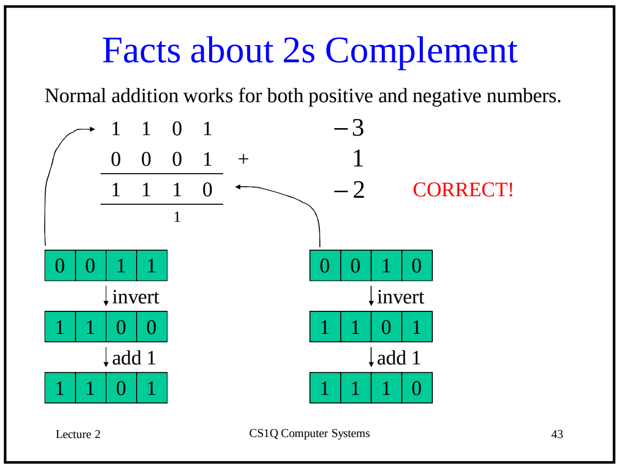
↓ invert

1	1	1	0	1	0	0	1

↓ add 1

1	1	1	0	1	0	1	0

Lecture 2 CS1Q Computer Systems 42



Facts about 2s Complement

Normal addition works for both positive and negative numbers.

```
  1  1  0  1         −3
  0  0  0  1   +      1
  ----------
  1  1  1  0         −2     CORRECT!
  ----------
        1
```

0	0	1	1

↓invert

1	1	0	0

↓add 1

1	1	0	1

0	0	1	0

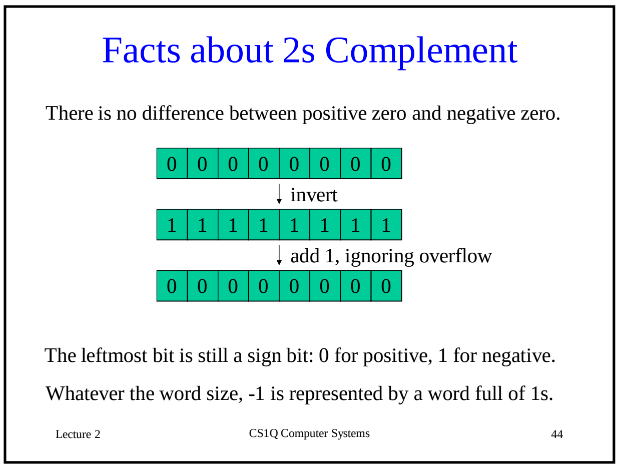
↓invert

1	1	0	1

↓add 1

1	1	1	0

Lecture 2 CS1Q Computer Systems 43



Facts about 2s Complement

There is no difference between positive zero and negative zero.

0	0	0	0	0	0	0	0

↓ invert

1	1	1	1	1	1	1	1

↓ add 1, ignoring overflow

0	0	0	0	0	0	0	0

The leftmost bit is still a sign bit: 0 for positive, 1 for negative.

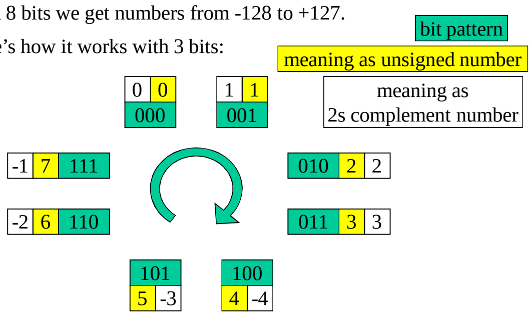
Whatever the word size, -1 is represented by a word full of 1s.

Lecture 2 CS1Q Computer Systems 44

Facts about 2s Complement

Half the bit patterns represent positive numbers, half negative.
With 8 bits we get numbers from -128 to +127.

Here's how it works with 3 bits:



Converting to and from binary

When converting from decimal to binary it is important to know whether we are producing a signed or unsigned representation. This is usually obvious: if we are given a negative decimal number then we must use the signed (two's complement) representation.

When converting from binary to decimal it is important to know whether the given number is signed or unsigned.

10101010 as an unsigned binary number means 170 in decimal.

10101010 as a signed binary number means -86 in decimal.

Some programming languages provide both signed and unsigned integer types, and confusion can result. (Example: C)

Real Numbers in Binary

Real numbers (i.e. non-integers) can be represented in binary in the same way as in decimal: the column values keep halving as we move to the right.

	128	64	32	16	8	4	2	1		1/2	1/4	1/8	1/16	
…	0	0	0	0	1	0	1	1	●	1	1	0	1	…

Example: $1011.1101_2 = 11.8125_{10}$

The familiar issues of decimal expansions also arise in binary: different numbers have expansions of different lengths, some have recurring or non-recurring infinite expansions, and so on.

Floating Point Numbers

For computational purposes we need a fixed-size representation of real numbers. Fixing the number of digits before and after the binary point would be too inflexible, so we use *floating point* numbers.

The basic idea is the same as standard *scientific notation* in decimal:

$2.5 \times 10^2 = 2500$
$3.4 \times 10^{-5} = 0.000034$

but we use powers of 2 instead of powers of 10, and express everything in binary:

$$\underbrace{1.01}_{\textit{mantissa or fraction}} \times \underbrace{2}_{\text{always 2}}^{\overbrace{100}^{\textit{exponent}}} \text{ (binary)} = 1.25 \times 16 \text{ (decimal)} = 20.$$

Floating Point Numbers

A particular floating point format will use a fixed number of bits for the mantissa, a fixed number of bits for the exponent, and one extra bit to represent the sign (0 for positive, 1 for negative) of the overall number.

Example: let's use a 2 bit mantissa and a 3 bit exponent

The 2 bit mantissa gives 4 possibilities: 00, 01, 10, 11 and we will interpret these as 0.00, 0.01, 0.10 and 0.11 (in binary),
i.e. 0, 0.25, 0.5 and 0.75 (in decimal).

The 3 bit exponent gives 8 possibilities and we will interpret these as -4 … +3.

Example Floating Point Format

	2^{exponent}							
mantissa	1/16	1/8	1/4	1/2	1	2	4	8
0	0	0	0	0	0	0	0	0
0.25	0.015625	0.03125	0.0625	0.125	0.25	0.5	1	2
0.5	0.03125	0.0625	0.125	0.25	0.5	1	2	4
0.75	0.046875	0.09375	0.01875	0.375	0.75	1.5	3	6

Points to note:
• the 32 combinations only give 18 different values
• the values are not evenly distributed

IEE Floating Point Format

The IEE floating point format avoids multiple representations, and represents some special values (NaN, ∞) to help with error detection. The exponent is interpreted differently, and the interpretation of the mantissa depends on the value of the exponent. Here's how it would look with a 2 bit mantissa and 3 bit exponent.

	exponent							
mantissa	000	001	010	011	100	101	110	111
00	0	0.25	0.5	1	2	4	8	∞
01	0.0625	0.3125	0.625	1.25	2.5	5	10	NaN
10	0.125	0.375	0.75	1.5	3	6	12	NaN
11	0.1875	0.4375	0.875	1.75	3.5	7	14	NaN

Floating Point Numbers

However many bits we use for the mantissa and exponent
(IEE single precision: 23 and 8; IEE double precision: 52 and 11)
the following points are always true:

Only a finite set of numbers is available, whereas in mathematical reality any range contains an infinite set of real numbers.

A real number is represented by the nearest floating point number; usually this is only an approximation.

Floating point arithmetic does not correspond exactly to mathematical reality: numbers of different sizes do not mix well.
E.g. in the IEE example, 12 + 0.25 = 12.

Usually it is possible to be accurate enough for a given purpose, but:

Floating Point Mathematics

Define the sequence a_i by

$$a_0 = \frac{11}{2} \qquad a_1 = \frac{61}{11} \qquad a_{n+1} = 111 - \frac{1130}{a_n} + \frac{3000}{a_{n-1}a_n}$$

In *any* floating point format, no matter how many bits are used, the sequence converges to 100.

In reality it converges to 6.

CS1Q Computer Systems
Lecture 3

Where we are

Global computing: the Internet

Networks and distributed computing

Application on a single computer

Operating System

Architecture

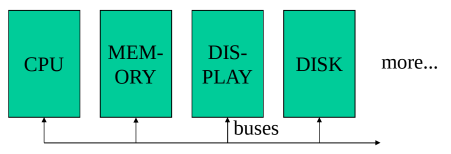
Digital Logic

Electronics

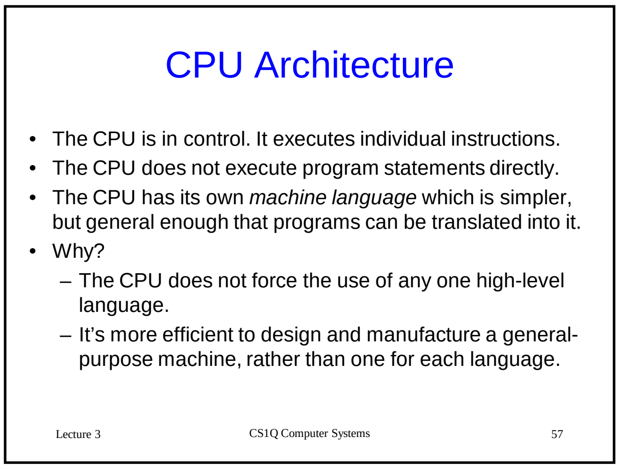
Physics

How do we design a machine that can execute programs?

Structure of a Computer

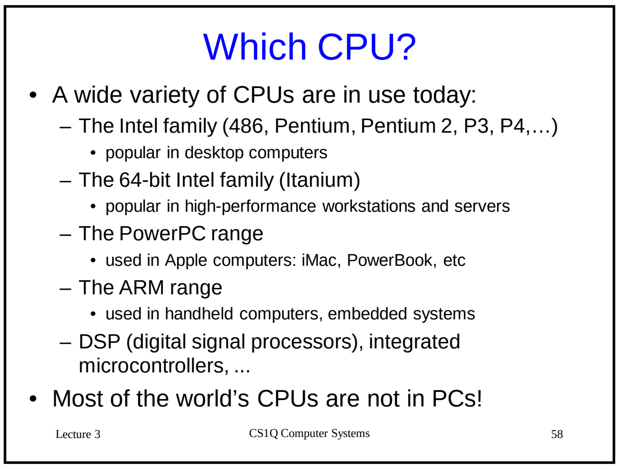


CPU - Central Processing Unit; microprocessor; e.g. Pentium 4
Memory - stores both programs and data
Peripherals - display, disk, keyboard, modem, printer, …
Disk - larger, slower and more permanent than memory
Buses - pathways for information



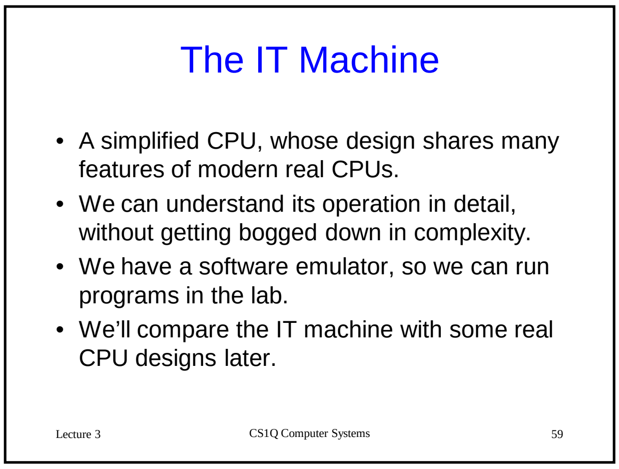
CPU Architecture

- The CPU is in control. It executes individual instructions.
- The CPU does not execute program statements directly.
- The CPU has its own *machine language* which is simpler, but general enough that programs can be translated into it.
- Why?
  - The CPU does not force the use of any one high-level language.
  - It's more efficient to design and manufacture a general-purpose machine, rather than one for each language.



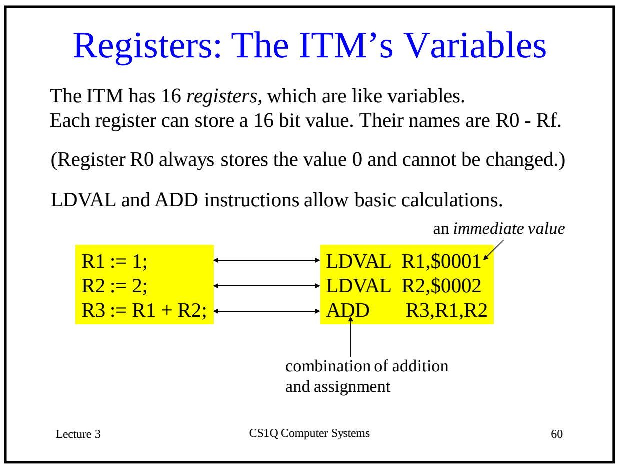
Which CPU?

- A wide variety of CPUs are in use today:
  - The Intel family (486, Pentium, Pentium 2, P3, P4,…)
    - popular in desktop computers
  - The 64-bit Intel family (Itanium)
    - popular in high-performance workstations and servers
  - The PowerPC range
    - used in Apple computers: iMac, PowerBook, etc
  - The ARM range
    - used in handheld computers, embedded systems
  - DSP (digital signal processors), integrated microcontrollers, ...
- Most of the world's CPUs are not in PCs!



The IT Machine

- A simplified CPU, whose design shares many features of modern real CPUs.
- We can understand its operation in detail, without getting bogged down in complexity.
- We have a software emulator, so we can run programs in the lab.
- We'll compare the IT machine with some real CPU designs later.



Registers: The ITM's Variables

The ITM has 16 *registers*, which are like variables.
Each register can store a 16 bit value. Their names are R0 - Rf.

(Register R0 always stores the value 0 and cannot be changed.)

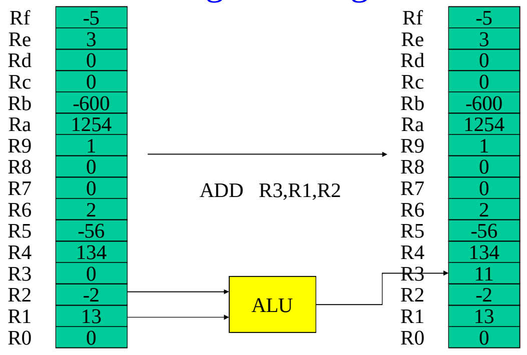
LDVAL and ADD instructions allow basic calculations.

an *immediate value*

```
R1 := 1;       <-->  LDVAL  R1,$0001
R2 := 2;       <-->  LDVAL  R2,$0002
R3 := R1 + R2; <-->  ADD    R3,R1,R2
```

combination of addition
and assignment

Working with Registers



	Before	After
Rf	-5	-5
Re	3	3
Rd	0	0
Rc	0	0
Rb	-600	-600
Ra	1254	1254
R9	1	1
R8	0	0
R7	0	0
R6	2	2
R5	-56	-56
R4	134	134
R3	0	11
R2	-2	-2
R1	13	13
R0	0	0

The ALU

- The Arithmetic and Logic Unit is a subsystem of the CPU.
- The ALU carries out operations such as addition, subtraction, comparisons, … when required by instructions such as ADD.

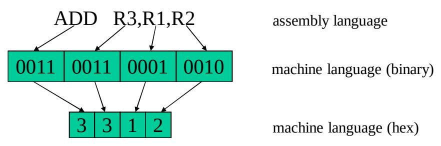
Memory

- The registers are not sufficient for storing large amounts of data. So the ITM has *memory*.
- The memory is like an array of 16 bit words. Each *location* (element) has an *address* (index).
- The ITM is a 16 bit machine, so a memory address is a 16 bit word. Therefore the maximum memory size is 65536 words.
- As well as storing data, the memory stores the instructions which make up a program.
- In practice, (most of) the memory is outside the CPU.

Assembly Language and Machine Language

Instructions such as ADD R3,R1,R2 are in *assembly language*.
Assembly language is a human-readable form of *machine language*.

Machine language is a binary representation of instructions.



It is the machine language form which is stored in memory.

The Stored Program Computer

- Storing the program in memory, in the same way as data, is one of the most important ideas in computing.
- It allows great flexibility, and means that programs which manipulate programs (e.g. compilers) are conceptually no different from programs which manipulate data.

Execution of a Program

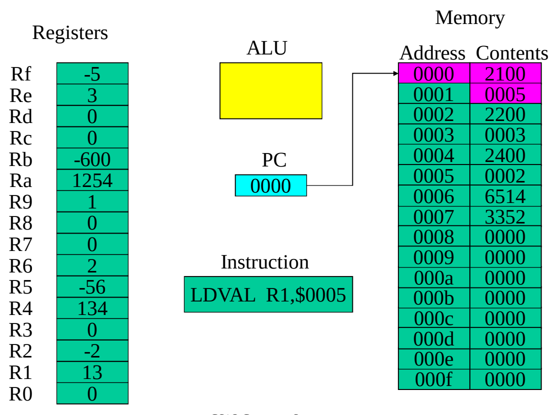
Instructions are executed in sequence, starting with the instruction in memory location 0.

A special register, the *program counter* (PC), stores the address of the instruction being executed.

Example:

```
R1 := 5;
R2 := 3;
R3 := 2*R1 + R2;
```

```
LDVAL  R1,$0005
LDVAL  R2,$0003
LDVAL  R4,$0002
MUL     R5,R1,R4
ADD     R3,R5,R2
```

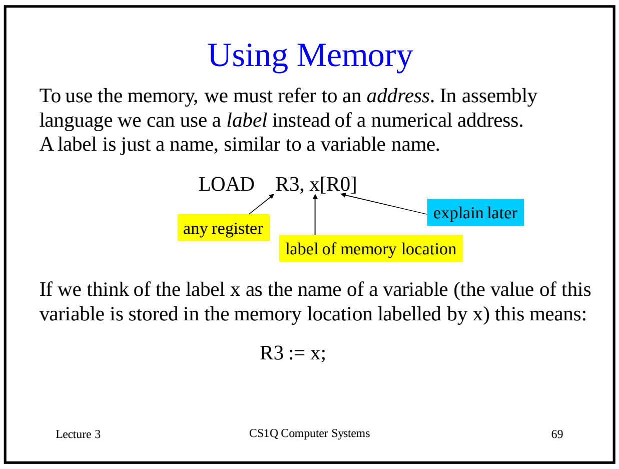


Registers	
Rf	-5
Re	3
Rd	0
Rc	0
Rb	-600
Ra	1254
R9	1
R8	0
R7	0
R6	2
R5	-56
R4	134
R3	0
R2	-2
R1	13
R0	0

Address	Contents
0000	2100
0001	0005
0002	2200
0003	0003
0004	2400
0005	0002
0006	6514
0007	3352
0008	0000
0009	0000
000a	0000
000b	0000
000c	0000
000d	0000
000e	0000
000f	0000

Assembly Language Programming

- It is rarely necessary to program in assembly language.
- Assembly language programs are produced by systematic (and automatic) translation of programs in high level languages (e.g. Ada).
- We will look at how some common high level constructs are translated.
- Compiler writers must understand assembly language.
- CPUs are designed with compilers in mind.



Using Memory

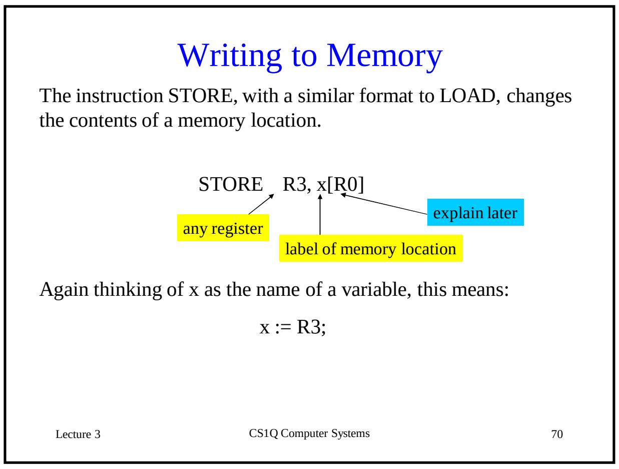
To use the memory, we must refer to an *address*. In assembly language we can use a *label* instead of a numerical address. A label is just a name, similar to a variable name.

```
LOAD   R3, x[R0]
```

any register
label of memory location
explain later

If we think of the label x as the name of a variable (the value of this variable is stored in the memory location labelled by x) this means:

R3 := x;



Writing to Memory

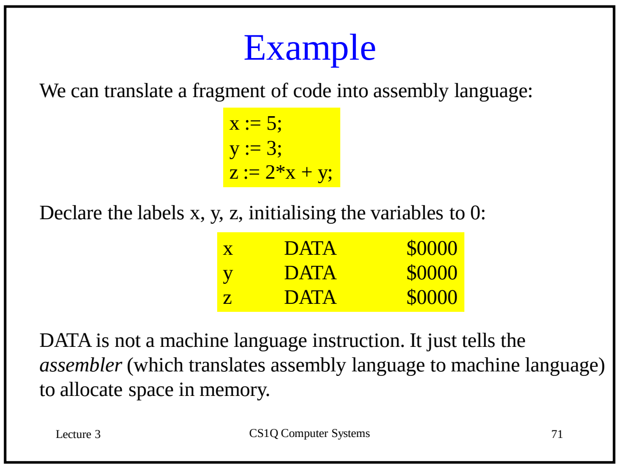
The instruction STORE, with a similar format to LOAD, changes the contents of a memory location.

```
STORE   R3, x[R0]
```

any register
label of memory location
explain later

Again thinking of x as the name of a variable, this means:

x := R3;



Example

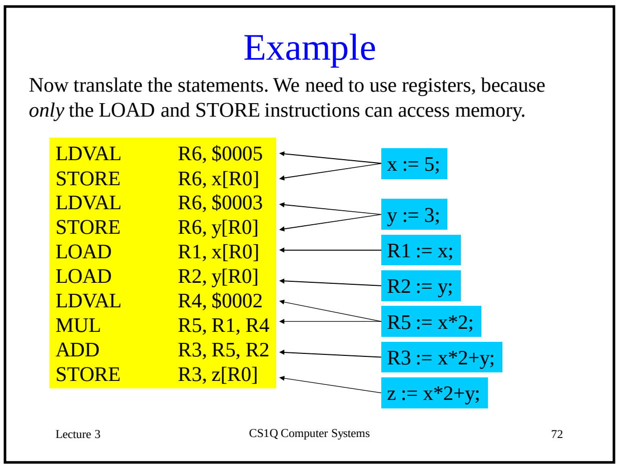
We can translate a fragment of code into assembly language:

```
x := 5;
y := 3;
z := 2*x + y;
```

Declare the labels x, y, z, initialising the variables to 0:

```
x    DATA    $0000
y    DATA    $0000
z    DATA    $0000
```

DATA is not a machine language instruction. It just tells the *assembler* (which translates assembly language to machine language) to allocate space in memory.



Example

Now translate the statements. We need to use registers, because *only* the LOAD and STORE instructions can access memory.

```
LDVAL    R6, $0005
STORE    R6, x[R0]
LDVAL    R6, $0003
STORE    R6, y[R0]
LOAD     R1, x[R0]
LOAD     R2, y[R0]
LDVAL    R4, $0002
MUL      R5, R1, R4
ADD      R3, R5, R2
STORE    R3, z[R0]
```

x := 5;
y := 3;
R1 := x;
R2 := y;
R5 := x*2;
R3 := x*2+y;
z := x*2+y;

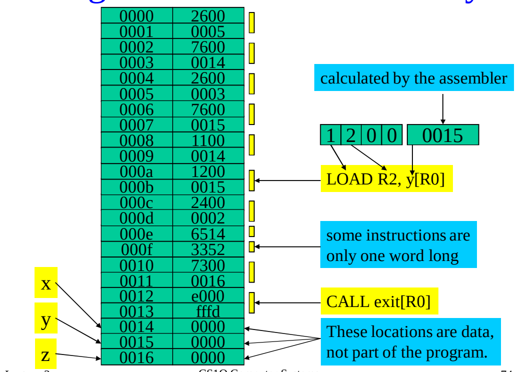
A Complete Program

```
        LDVAL     R6, $0005
        STORE     R6, x[R0]
        LDVAL     R6, $0003
        STORE     R6, y[R0]
        LOAD      R1, x[R0]
        LOAD      R2, y[R0]
        LDVAL     R4, $0002
        MUL       R5, R1, R4
        ADD       R3, R5, R2
        STORE     R3, z[R0]
        CALL      exit[R0]
x       DATA      $0000
y       DATA      $0000
z       DATA      $0000
```

stops execution

Lecture 3 CS1Q Computer Systems 73

Program and Data in Memory



0000	2600
0001	0005
0002	7600
0003	0014
0004	2600
0005	0003
0006	7600
0007	0015
0008	1100
0009	0014
000a	1200
000b	0015
000c	2400
000d	0002
000e	6514
000f	3352
0010	7300
0011	0016
0012	e000
0013	fffd
0014	0000
0015	0000
0016	0000

calculated by the assembler
1 2 0 0
0015
LOAD R2, y[R0]
some instructions are only one word long
CALL exit[R0]
These locations are data, not part of the program.
x
y
z

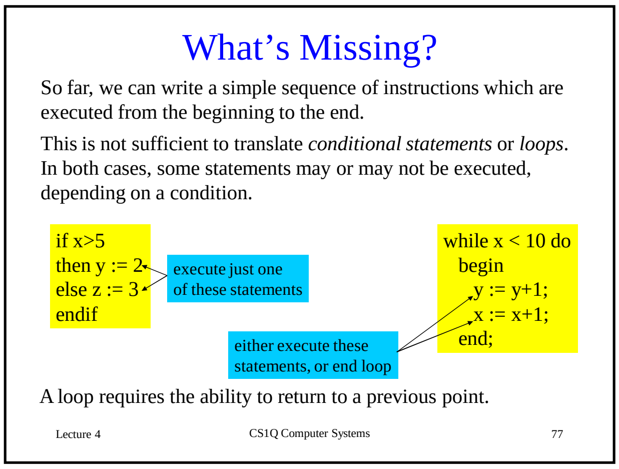
Lecture 3 CS1Q Computer Systems 74

Optimizations

- There are ways of improving this program by making it shorter.
- Compilers use a variety of techniques to produce *optimized* (as good as possible) code.
- We won't worry about this issue - we'll just concentrate on a straightforward and systematic translation of simple Ada statements into assembly language.

Lecture 3 CS1Q Computer Systems 75

CS1Q Computer Systems Lecture 4



What's Missing?

So far, we can write a simple sequence of instructions which are executed from the beginning to the end.

This is not sufficient to translate *conditional statements* or *loops*. In both cases, some statements may or may not be executed, depending on a condition.

```
if x>5
then y := 2
else z := 3
endif
```

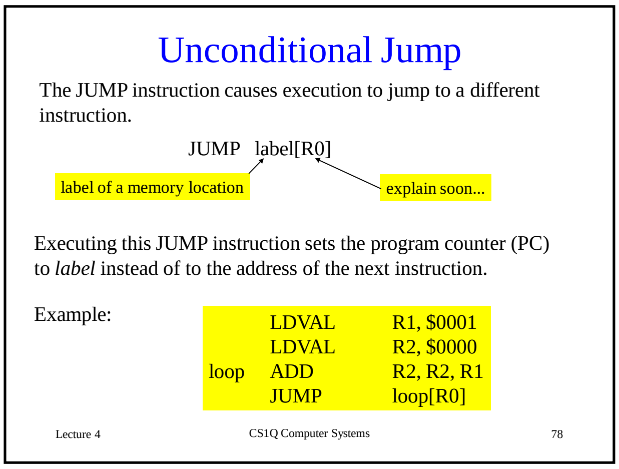
execute just one of these statements

```
while x < 10 do
  begin
    y := y+1;
    x := x+1;
  end;
```

either execute these statements, or end loop

A loop requires the ability to return to a previous point.

Lecture 4 CS1Q Computer Systems 77



Unconditional Jump

The JUMP instruction causes execution to jump to a different instruction.

```
JUMP   label[R0]
```

label of a memory location

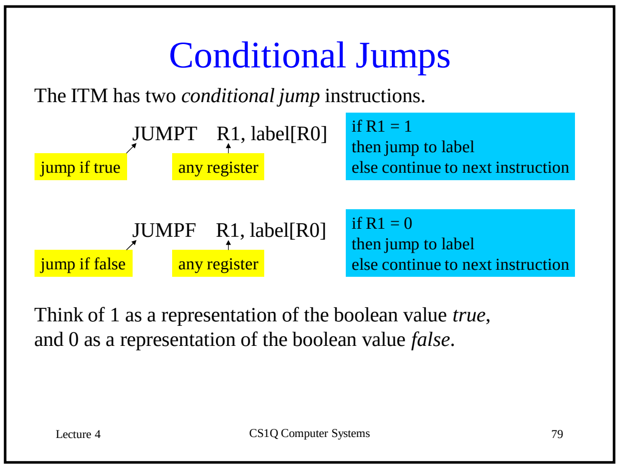
explain soon...

Executing this JUMP instruction sets the program counter (PC) to *label* instead of to the address of the next instruction.

Example:

```
        LDVAL      R1, $0001
        LDVAL      R2, $0000
loop    ADD        R2, R2, R1
        JUMP       loop[R0]
```

Lecture 4 CS1Q Computer Systems 78



Conditional Jumps

The ITM has two *conditional jump* instructions.

```
JUMPT    R1, label[R0]
```

jump if true

any register

if R1 = 1
then jump to label
else continue to next instruction

```
JUMPF    R1, label[R0]
```

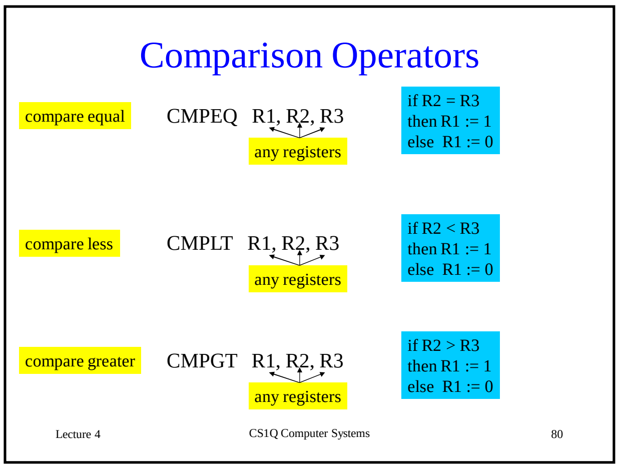
jump if false

any register

if R1 = 0
then jump to label
else continue to next instruction

Think of 1 as a representation of the boolean value *true*, and 0 as a representation of the boolean value *false*.

Lecture 4 CS1Q Computer Systems 79



Comparison Operators

compare equal

```
CMPEQ   R1, R2, R3
```

any registers

if R2 = R3
then R1 := 1
else R1 := 0

compare less

```
CMPLT   R1, R2, R3
```

any registers

if R2 < R3
then R1 := 1
else R1 := 0

compare greater

```
CMPGT   R1, R2, R3
```

any registers

if R2 > R3
then R1 := 1
else R1 := 0

Lecture 4 CS1Q Computer Systems 80

Translating if-then-else

Using a combination of conditional and unconditional jumps, we can translate an if-then-else statement into assembly language.

```
if  R1 < R2
then  statements1
else   statements2
end if;
more statements
```

```
          CMPLT         R3, R1, R2
          JUMPF         R3, else[R0]
          translation of statements1
          JUMP          end[R0]
else      translation of statements2
end       translation of more statements
```

if we shouldn't execute the *then* branch, jump past it

jump past the *else* branch

Translating a while loop

Again using conditional and unconditional jumps, we can translate a while loop into assembly language.

```
while  R1 < 10  loop
statements
end loop;
more statements
```

```
loop    LDVAL       R2, $000a
        CMPLT       R3, R1, R2
        JUMPF       R3, end[R0]
        translation of statements
        JUMP        loop[R0]
end     translation of more statements
```

if we shouldn't execute the loop body, jump past it

jump back to test the condition again

Example: Sum of Integers

The following code calculates, in *s*, the sum of the integers from 1 to *n*.

```
s := 0;
while n > 0 loop
          s := s + n;
          n := n - 1;
end loop;
```

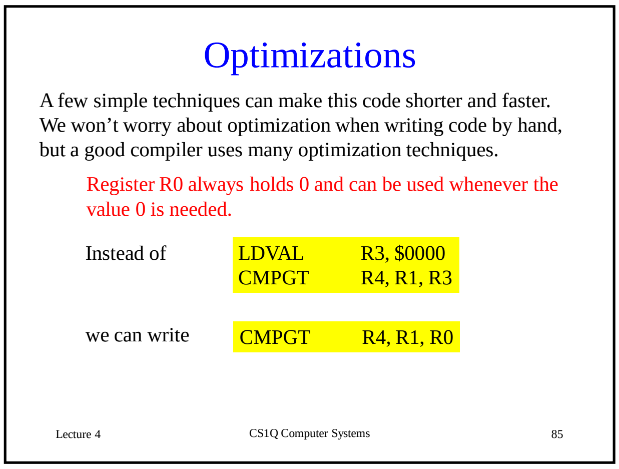
We can translate this code systematically into assembly language. First we'll do it using registers for the variables *s* and *n*.

Translating to Assembly Language

We will use register R1 for the variable *n*, and R2 for the variable *s*.

```
s := 0;
while n > 0 loop
          s := s + n;
          n := n - 1;
end loop;
```

```
        LDVAL       R2, $0000
loop    LDVAL       R3, $0000
        CMPGT       R4, R1, R3
        JUMPF       R4, end[R0]
        ADD         R2, R2, R1
        LDVAL       R5, $0001
        SUB         R1, R1, R5
        JUMP        loop[R0]
end
```



Optimizations

A few simple techniques can make this code shorter and faster. We won't worry about optimization when writing code by hand, but a good compiler uses many optimization techniques.

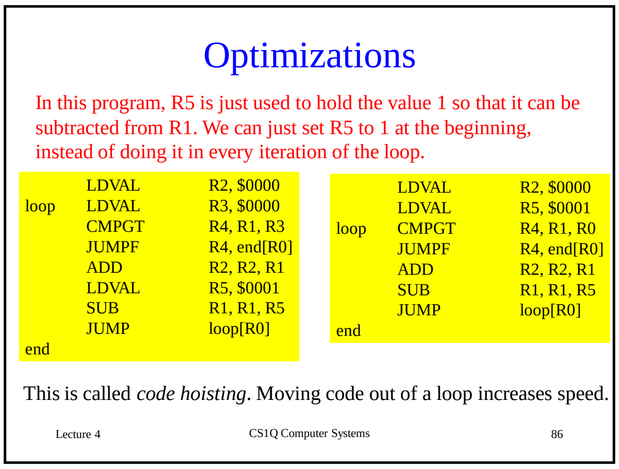
Register R0 always holds 0 and can be used whenever the value 0 is needed.

Instead of

```
LDVAL     R3, $0000
CMPGT     R4, R1, R3
```

we can write

```
CMPGT     R4, R1, R0
```



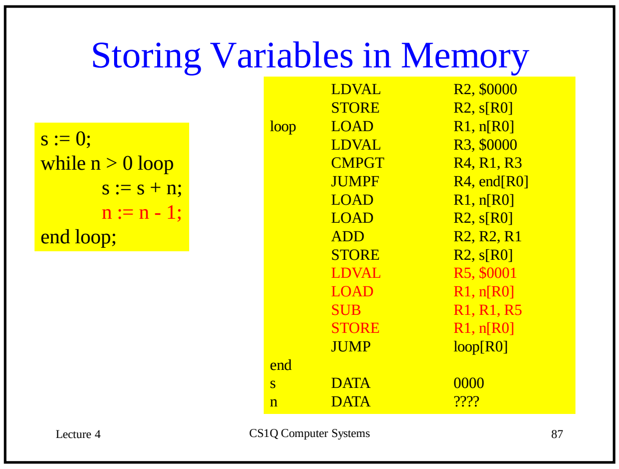
Optimizations

In this program, R5 is just used to hold the value 1 so that it can be subtracted from R1. We can just set R5 to 1 at the beginning, instead of doing it in every iteration of the loop.

```
        LDVAL     R2, $0000
loop    LDVAL     R3, $0000
        CMPGT     R4, R1, R3
        JUMPF     R4, end[R0]
        ADD       R2, R2, R1
        LDVAL     R5, $0001
        SUB       R1, R1, R5
        JUMP      loop[R0]
end
```

```
        LDVAL     R2, $0000
        LDVAL     R5, $0001
loop    CMPGT     R4, R1, R0
        JUMPF     R4, end[R0]
        ADD       R2, R2, R1
        SUB       R1, R1, R5
        JUMP      loop[R0]
end
```

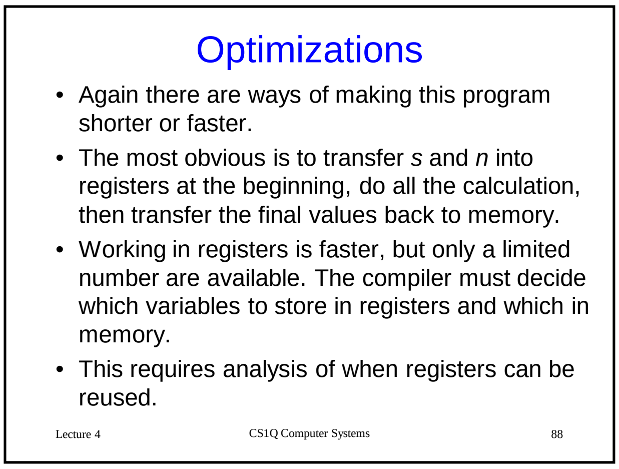
This is called *code hoisting*. Moving code out of a loop increases speed.



Storing Variables in Memory

```
s := 0;
while n > 0 loop
        s := s + n;
        n := n - 1;
end loop;
```

```
        LDVAL     R2, $0000
        STORE     R2, s[R0]
loop    LOAD      R1, n[R0]
        LDVAL     R3, $0000
        CMPGT     R4, R1, R3
        JUMPF     R4, end[R0]
        LOAD      R1, n[R0]
        LOAD      R2, s[R0]
        ADD       R2, R2, R1
        STORE     R2, s[R0]
        LDVAL     R5, $0001
        LOAD      R1, n[R0]
        SUB       R1, R1, R5
        STORE     R1, n[R0]
        JUMP      loop[R0]
end
s       DATA      0000
n       DATA      ????
```



Optimizations

- Again there are ways of making this program shorter or faster.
- The most obvious is to transfer *s* and *n* into registers at the beginning, do all the calculation, then transfer the final values back to memory.
- Working in registers is faster, but only a limited number are available. The compiler must decide which variables to store in registers and which in memory.
- This requires analysis of when registers can be reused.

Example: Multiplication

The ITM has an instruction for multiplication, but if it didn't, we could easily write a program for it.

To multiply *a* by *b*, leaving the result in *c*: (assuming *b* is positive)

```
c := 0;
while b > 0 loop
        c := c + a;
        b := b - 1;
end loop;
```

Multiplication is just repeated addition.

Multiplication

```
c := 0;
while b > 0 loop
        c := c + a;
        b := b - 1;
end loop;
```

```
% This is a comment
% R1 = a, R2 = b, R3 = c, R4 = 1
        LDVAL       R3, $0000 % c := 0
        LDVAL       R4, $0001 % R4 := 1
loop    CMPGT       R5, R2, R0 % R5 := (b > 0)
        JUMPF       R5, end % if not(b > 0) then exit loop
        ADD         R3, R3, R1 % c := c + a
        SUB         R2, R2, R4 % b := b - 1
        JUMP        loop[R0] % go to top of loop
end
```

Using Memory Locations

We have been writing references to memory locations in the form

`label[R0]`

Examples:

```
LOAD        R1, label[R0]
STORE       R1, label[R0]
```

to transfer data
to and from memory

```
JUMP        label[R0]
JUMPT       label[R0]
JUMPF       label[R0]
```

to jump to a different
point in the program

It's time to explain exactly what this means: why is R0 mentioned when we are just interested in the memory location label?

Indexed Addressing

The general form of a reference to a memory location is

`x[R]`

where x is a label and R is any register. This refers to the memory location at address x + R.

This is called *indexed addressing*. x is called the *base* and R is called the *index*.

Up to now we have just used R0, whose value is always 0. x[R0] just refers to the memory location at address x.

By using other registers, we can implement *arrays*.

Indexed Addressing and Arrays

```
        LDVAL   R1, $0005
        LDVAL   R2, $0002
        STORE   R1, a[R2]



a       DATA    $0000
        DATA    $0000
        DATA    $0000
        DATA    $0000
        ...
```

R1 := 5;
R2 := 2;
a[R2] := R1;

refers to address a+R2 = a+2

address is a+0
address is a+1
address is a+2
address is a+3

a sequence of memory locations, starting at address a

Array and While Loop

```
% R1 = i, R2 = 10, R3 = 1

        LDVAL   R1, $0000     % i := 0;
        LDVAL   R2, $000a     % R2 := 10;
        LDVAL   R3, $0001     % R3 := 1;
loop    CMPLT   R4, R1, R2    % R4 := (i < 10);
        JUMPF   R4, end[R0]   % if not (i < 10) then exit loop;
        STORE   R1, a[R1]     % a[i] := i;
        ADD     R1, R1, R3    % i := i + 1;
        JUMP    loop[R0]      % go to top of while loop;
end
```

```
i := 0;
while i < 10 loop
    a[i] := i;
    i := i + 1;
end loop;
```

Largest Element of an Array

Find the largest value in an array *a*, assuming that the end of the array is marked by the value -1.

```
max := a[0];
i := 1;
while a[i] <> -1 loop
        if a[i] > max
        then max := a[i];
        end if;
        i := i + 1;
end loop;
```

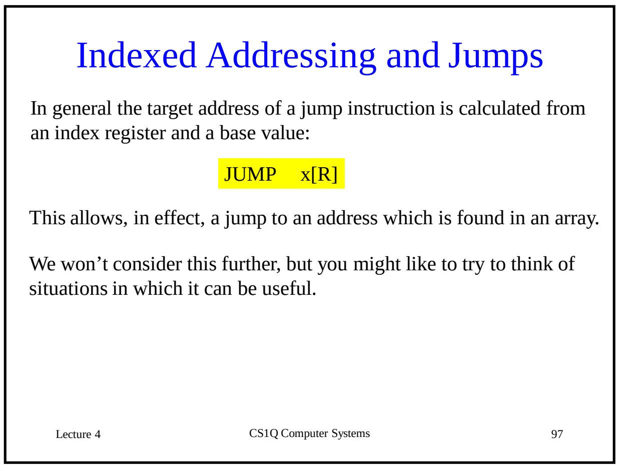
the first element is a[0]

max is the largest value found so far

Largest Element of an Array

```
% R1 = max, R2 = i, R3 = -1, R4 = 1, R5 = a[i]
        LDVAL   R3, $ffff   % R3 := -1
        LDVAL   R4, $0001 % R4 := 1
        LOAD    R1, a[R0]  % max := a[0]
        LDVAL   R2, $0001 % i := 1
loop    LOAD    R5, a[R2] % R5 := a[i]
        CMPEQ   R6, R5, R3 % R6 := (a[i] = -1)
        JUMPT   R6, end[R0] % if a[i] = -1 then exit loop
        CMPGT   R7, R5, R1 % R7 := (a[i] > max)
        JUMPF   R7, endif[R0] % if a[i] <= max then end if
        ADD     R1, R5, R0 % max := a[i] + 0
endif   ADD     R2, R2, R4 % i := i + 1
        JUMP    loop[R0] % go to top of while loop
end     CALL    exit[R0]  % stop
a       DATA    $0002  % values in array a
        DATA    $0005
        …
        DATA    $ffff     % indicates end of array a
```

```
max := a[0];
i := 1;
while a[i] <> -1 loop
        if a[i] > max
        then max := a[i];
        end if;
        i := i + 1;
end loop;
```



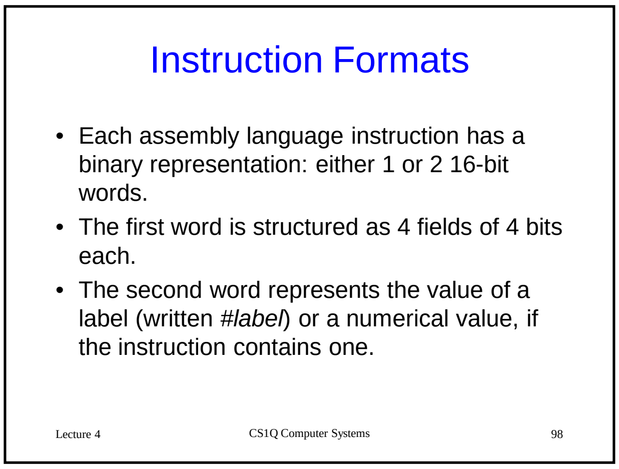
Indexed Addressing and Jumps

In general the target address of a jump instruction is calculated from an index register and a base value:

```
JUMP    x[R]
```

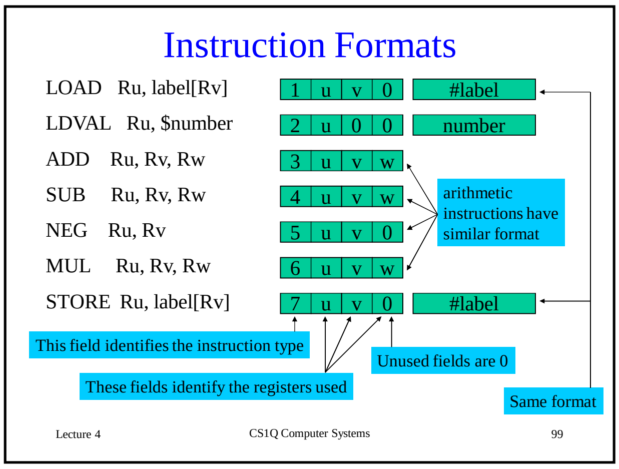
This allows, in effect, a jump to an address which is found in an array.

We won't consider this further, but you might like to try to think of situations in which it can be useful.



Instruction Formats

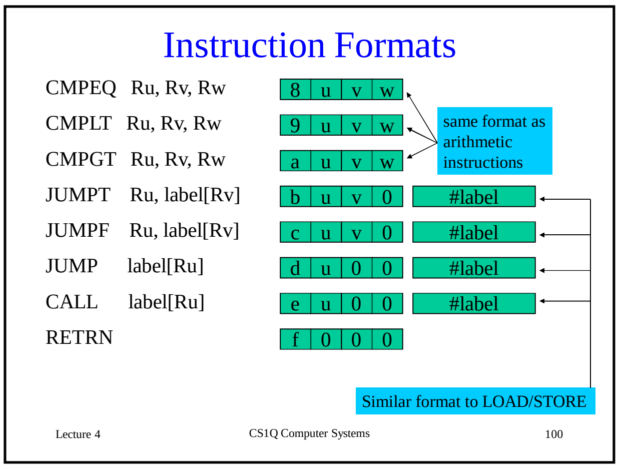
- Each assembly language instruction has a binary representation: either 1 or 2 16-bit words.
- The first word is structured as 4 fields of 4 bits each.
- The second word represents the value of a label (written *#label*) or a numerical value, if the instruction contains one.



Instruction Formats

LOAD Ru, label[Rv]	1	u	v	0	#label	
LDVAL Ru, $number	2	u	0	0	number	
ADD Ru, Rv, Rw	3	u	v	w		
SUB Ru, Rv, Rw	4	u	v	w		
NEG Ru, Rv	5	u	v	0		
MUL Ru, Rv, Rw	6	u	v	w		
STORE Ru, label[Rv]	7	u	v	0	#label	

arithmetic instructions have similar format
This field identifies the instruction type
Unused fields are 0
These fields identify the registers used
Same format



Instruction Formats

CMPEQ Ru, Rv, Rw	8	u	v	w	
CMPLT Ru, Rv, Rw	9	u	v	w	
CMPGT Ru, Rv, Rw	a	u	v	w	
JUMPT Ru, label[Rv]	b	u	v	0	#label
JUMPF Ru, label[Rv]	c	u	v	0	#label
JUMP label[Ru]	d	u	0	0	#label
CALL label[Ru]	e	u	0	0	#label
RETRN	f	0	0	0	

same format as arithmetic instructions
Similar format to LOAD/STORE

Program Execution

At the heart of the CPUs operation is a loop known as the

fetch-decode-execute cycle or the *fetch-execute cycle*

FETCH: transfer a word from memory (at the address indicated by the PC (program counter) into the CPU.

DECODE: work out which instruction it is, and which parts of the CPU must be used to execute it.

EXECUTE: activate the necessary parts of the CPU. Memory might be accessed again.

Then the PC must be updated: to point either to the next instruction in sequence, or to the target address of a jump.

A Bit of History

The first microprocessor was developed in the early 1970s, by Intel. Through the 1970s and 1980s, CPUs became more and more complex, along with developments in IC manufacturing technology.

By the late 1980s, instruction sets were enormously complex and therefore difficult to implement. But studies showed that most programs made little use of the more complex instructions, basically because it's hard for compilers to take advantage of special-purpose instructions.

This led to the development of RISC (reduced instruction set computer) CPUs, aiming to implement a small and simple instruction set very efficiently. The traditional designs were characterized as CISCs (complex instruction set computers).

The IT Machine vs. Real CPUs

The IT machine has many features typical of RISC designs:

- few instructions, following even fewer patterns
- regularity: all registers are interchangeable
- load/store architecture: the only instructions affecting memory are transfers to/from registers
- only one addressing mode: indexed addressing

In many ways the current Intel CPUs (Pentium x) are the culmination of the CISC approach, but they are becoming more RISC-like internally.

The problem of exploiting special-purpose instructions (e.g. MMX) in compiler-generated code still exists.

CS1Q Computer Systems
Lecture 5

Where we are

Global computing: the Internet

Networks and distributed computing

Application on a single computer

Operating System

Architecture ↑

Digital Logic

Electronics

Physics

Working upwards within the digital logic level, in order to understand architecture in more detail

Processing Digital Information

We'll start with some fundamental operations on binary digits and work up to more complicated operations.

AND

If *x* and *y* are binary digits (either 0 or 1) then

x AND y

is a binary digit, defined by

x AND *y* = 1 if *x* = 1 and *y* = 1
= 0 otherwise

The Truth Table for AND

If we think of the binary values as *true* and *false* instead of 1 and 0 then AND has its ordinary meaning:

x AND *y* is true if *x* is true and *y* is true

A *truth table* makes the meaning explicit:

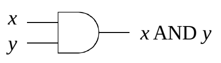
x	*y*	*x* AND *y*
f	f	f
f	t	f
t	f	f
t	t	t

x	*y*	*x* AND *y*
0	0	0
0	1	0
1	0	0
1	1	1

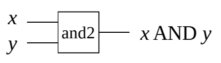
True/false, *high/low*, 1/0 are all alternatives. We will usually stick to 1/0 in truth tables.

Diagrammatic Representation

There is a conventional notation for diagrams in which the AND operation is represented by:



To make it easier to draw diagrams, we might just use a labelled box instead:



OR

If *x* and *y* are binary digits (either 0 or 1) then

x OR y

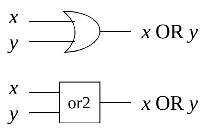
is a binary digit, defined by

x OR *y* = 1 if *x* = 1 or *y* = 1, or both
= 0 otherwise

Truth table:

x	*y*	*x* OR *y*
0	0	0
0	1	1
1	0	1
1	1	1

Diagram:



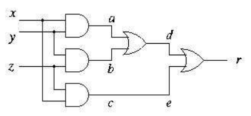
Example: Majority Voting

Imagine that three people have to vote either Yes (represented by 1) or No (represented by 0). The overall result is the majority decision.

If *x*, *y*, *z* stand for the three votes, and *r* stands for the result, then we can write

r = (*x* AND *y*) OR (*y* AND *z*) OR (*z* AND *x*)

Diagrammatically:



This can be viewed as a *circuit diagram* and implemented electronically. The components are then called *logic gates*.

Example: Majority Voting

We can use a truth table to check that the circuit works.

⇩ ⇩ ⇩ ⇩

x	*y*	*z*	*a*	*b*	*c*	*d*	*e*	*r*	
0	0	0	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	0	
0	1	0	0	0	0	0	0	0	
0	1	1	0	1	0	1	0	1	⇦
1	0	0	0	0	0	0	0	0	
1	0	1	0	0	1	0	1	1	⇦
1	1	0	1	0	0	1	0	1	⇦
1	1	1	1	1	1	1	1	1	⇦

The result *r* is 1 in the four cases when two of *x*, *y*, *z* are 1.

What We Have Defined

We have defined a function with three boolean (truth value) arguments (inputs) and a boolean result (output). Mathematically, we have

$$majority : B \times B \times B \to B$$

if *B* is the set {0,1}. The truth table (columns *x, y, z, r*) shows the result (an element of *B*) for each combination of inputs (each combination of inputs is an element of $B \times B \times B$).

The truth table defines a subset of $(B \times B \times B) \times B$ whose elements correspond to the rows: ((0,0,0),0), ((0,0,1),0), etc. It is a relation with attributes $B \times B \times B$ and *B*.

For each element *(x,y,z)* of $B \times B \times B$ the relation contains exactly one tuple whose input attributes match *(x,y,z)*. This property is what makes it into a function. The output attribute of this tuple is the result *r*.

Majority Voting

We can make use of the logical operations to express a majority voting function.

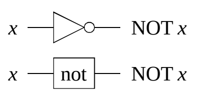
```
function Majority(x, y, z : Boolean) return Boolean is
begin
        return (x and y) or (y and z) or (z and x);
end Majority;
```

This gives a flavour of *hardware description languages*, which are used in preference to circuit diagrams for complex designs.

NOT

It turns out that AND and OR are not sufficient to define all functions on binary digits. (Because, any function constructed from AND and OR must output 1 if all inputs are 1.) The missing ingredient is NOT.

x	*NOT x*
0	1
1	0



Again, if we think in terms of truth values, NOT has its familiar meaning.

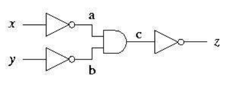
A NOT gate is often called an *inverter*.

(AND or OR) and NOT

By using AND, OR and NOT in combination, it is possible to define any desired function on binary numbers. We will see how to do this in a few lectures' time.

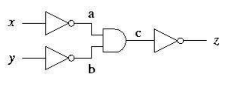
Perhaps surprisingly, we only need NOT and just *one* of AND and OR.

Exercise: work out the truth table for the following circuit and check that it is equivalent to the OR function.



OR from AND and NOT

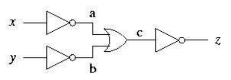
x	*y*	*a*	*b*	*c*	*z*
0	0				
0	1				
1	0				
1	1				



AND from OR and NOT

Exercise: check that this circuit is equivalent to the AND function.

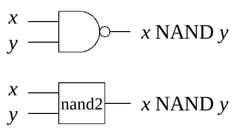
x	*y*	*a*	*b*	*c*	*z*
0	0				
0	1				
1	0				
1	1				



One Fundamental Operation

Even more remarkably, it is possible to build up all functions from combinations of just one operation: the NAND operation.

x	*y*	*x* NAND *y*
0	0	1
0	1	1
1	0	1
1	1	0



NAND is short for NOT AND. We can check that

x NAND *y* = NOT (*x* AND *y*)

NAND is Universal

Assuming that we have NAND, we can define the other operations:

NOT *x* = *x* NAND *x*
x AND *y* = NOT (*x* NAND *y*) = (*x* NAND *y*) NAND (*x* NAND *y*)
x OR *y* = (NOT *x*) NAND (NOT *y*) = (*x* NAND *x*) NAND (*y* NAND *y*)

Exercise: check these equations by constructing truth tables.

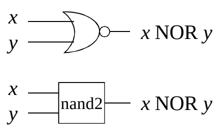
x	*x* NAND *x*
0	
1	

x	*y*	*x* NAND *y*	NOT(*x* NAND *y*)
0	0		
0	1		
1	0		
1	1		

Another Fundamental Operation

The NOR operation is also sufficient for building all functions.

x	*y*	*x* NOR *y*
0	0	1
0	1	0
1	0	0
1	1	0



NOR is short for NOT OR. We can check that

x NOR *y* = NOT (*x* OR *y*)

NOR is Universal

Assuming that we have NOR, we can define the other operations:

NOT *x* = *x* NOR *x*
x OR *y* = NOT (*x* NOR *y*) = (*x* NOR *y*) NOR (*x* NOR *y*)
x AND *y* = (NOT *x*) NOR (NOT *y*) = (*x* NOR *x*) NOR (*y* NOR *y*)

Exercise: check these equations by constructing truth tables.

x	*x* NOR *x*
0	
1	

x	*y*	*x* NOR *y*	NOT(*x* NOR *y*)
0	0		
0	1		
1	0		
1	1		

Exercise: prove that NAND and NOR are the only universal operations.

XOR

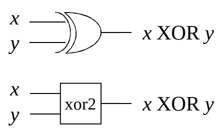
If *x* and *y* are binary digits then x XOR y is a binary digit, defined by

x XOR *y* = 1 if either *x* = 1 or *y* = 1, but not both
= 0 otherwise

Truth table:

x	*y*	*x* XOR *y*
0	0	0
0	1	1
1	0	1
1	1	0

Diagram:



XOR is *exclusive or*. OR is *inclusive or*.

Implication

Implication is a logical operation although it is not used in digital circuits. *x* => *y* means "*x* implies *y*" or "if *x* then *y*" or "if *x* is true then *y* is true".

x	*y*	*x* => *y*	Example
false	*false*	*true*	If 2+2=5 then the moon is made of cheese.
false	*true*	*true*	If 2+2=5 then Glasgow is in Scotland.
true	*false*	*false*	If Glasgow is in Scotland then 2+2=5.
true	*true*	*true*	If Glasgow is in Scotland then 2+2=4.

x => *y* is true if it is logically valid to deduce *y* from *x*. It is true in all cases *except* when *x* is true and *y* is false.

Note: *x* => *y* does not mean that *x* causes *y*.

CS1Q Computer Systems Lecture 6

Algebraic Notation

Writing AND, OR, NOT etc. is long-winded and tedious. We generally use a more compact notation:

xy	means	x AND y
$x+y$	means	x OR y
$\overline{x}$	means	NOT x
$x\oplus y$	means	x XOR y

The operations can be combined to form algebraic expressions representing logic functions.

Examples of Algebraic Notation

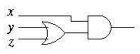
The majority voting function from the last lecture can be written

$$xy + yz + zx$$

The expression

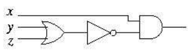
$$x(y+z)$$

means *x* AND (*y* OR *z*)

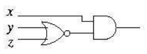


The expression $x\overline{(y+z)}$

means *x* AND NOT (*y* OR *z*)



and also *x* AND (*y* NOR *z*)



Exercise

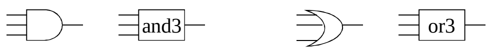
What is the meaning of this expression? Draw a circuit for this function, and calculate the truth table. Which logical operation is it?

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

Multi-Input Gates

The AND and OR operations can be generalized to take any number of inputs. Algebraically, we simply write *xyz* for the three-input AND of *x*, *y* and *z*. Similarly we write *x*+*y*+*z* for the three-input OR.

In circuit diagrams we use the same symbols as before, but with additional input wires:

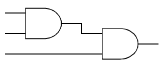


Definitions: AND is true if *all* the inputs are true; OR is true if *at least one* of the inputs is true.

NAND and NOR can also be defined for any number of inputs, in the obvious way.

Synthesis of Multi-Input Gates

An *n*-input AND or OR gate can be synthesized from 2-input gates of the same type.



Exercise: check this by using truth tables.

Exercise: how many 2-input AND gates are needed to synthesize an *n*-input AND gate?

Exercise: what happens if NAND or NOR gates are joined up like this?

Boolean Algebra

The algebraic properties of the logical operations were studied by George Boole (1815-1864). As a result we have *boolean algebra* and the datatype Boolean.

The laws of boolean algebra can be used to rewrite expressions involving the logical operations.

Negation is an involution	$\overline{\overline{x}} = x$	(1)
No contradictions	$x\overline{x} = 0$	(2)
AND is idempotent	$xx = x$	(3)

Laws of Boolean Algebra

Excluded middle	$x + \overline{x} = 1$	(4)
OR is idempotent	$x + x = x$	(5)
Zero law for AND	$x0 = 0$	(6)
AND is commutative	$xy = yx$	(7)
Unit law for AND	$x1 = x$	(8)
OR is commutative	$x + y = y + x$	(9)
Unit law for OR	$x + 0 = x$	(10)
Distributive law	$x(y+z) = xy + xz$	(11)

Laws of Boolean Algebra

One law for OR	$x + 1 = 1$	(12)
OR is associative	$x + (y+z) = (x+y) + z$	(13)
AND is associative	$x(yz) = (xy)z$	(14)
Distributive law	$x + yz = (x+y)(x+z)$	(15)

The associativity laws (13) and (14) justify writing *xyz* and *x*+*y*+*z* for the 3-input versions of AND and OR: it doesn't matter whether we interpret *xyz* as *x(yz)* or as *(xy)z*.

The laws can be verified by thinking about the ordinary meanings of AND, OR and NOT, or by truth tables.

Example

To verify that *x(y+z)* = *xy* + *xz* we construct the truth tables for the left and right hand sides of the equation, considering them both as functions of *x*, *y* and *z*.

x	*y*	*z*	*y*+*z*	*x*(*y* + *z*)	*xy*	*xz*	*xy* + *xz*
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Exercise

Using the laws of boolean algebra, show that *xy* + *x* = *x*.

Working out which law to use next requires some creativity. Truth tables provide a straightforward, systematic way to check equivalences.

Notice the similarity with the set membership tables used in the Information Management section to verify set identities.

De Morgan’s Laws

Two important laws relate AND, OR and NOT. They are named after Augustus De Morgan (1806-1871).

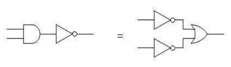
NOT(*x* AND *y*) = (NOT *x*) OR (NOT *y*)
NOT(*x* OR *y*) = (NOT *x*) AND (NOT *y*)

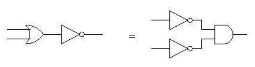
In algebraic notation:

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

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

“Break the line and change the sign.”





Boolean Algebra in Programming

The laws of boolean algebra apply *anywhere* that logical operations are used. For example, the code

```
if ((x=1) and (y=1)) or ((x=1) and (z=2)) then
        whatever
end if;
```

is equivalent to

```
if (x=1) and ((y=1)) or (z=2)) then
        whatever
end if;
```

Circuits from Truth Tables

- In Lecture 5 we constructed a logic circuit which computes the majority voting function.
- The function was defined by an English sentence, and I wrote down a logical expression and then a circuit by thinking about the ordinary meaning of the sentence.
- In general we need a more systematic approach.
- We'll use majority voting as an example, then progress to functions such as addition.
- Start with the truth table as the *definition* of the function to be implemented.

Majority Voting Systematically

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

For r to be 1, it must be the case that:

x=0 and y=1 and z=1
or
x=1 and y=0 and z=1
or
x=1 and y=1 and z=0
or
x=1 and y=1 and z=1

Majority Voting Systematically

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Alternatively, for r to be 1, it must be the case that:

$\overline{x}$=1 and y=1 and z=1
or
x=1 and $\overline{y}$=1 and z=1
or
x=1 and y=1 and $\overline{z}$=1
or
x=1 and y=1 and z=1

Majority Voting Systematically

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Alternatively, for r to be 1, it must be the case that:

$\overline{x}yz=1$
or
$x\overline{y}z=1$
or
$xy\overline{z}=1$
or
$xyz=1$

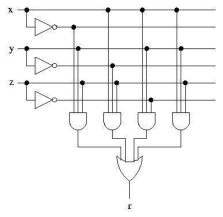
Majority Voting Systematically

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Rewriting one more time, we have discovered that:

$$r = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz$$

which gives the following circuit.



Majority Voting Systematically

$$r = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz$$

The expressions $\bar{x}yz$ etc. are called *minterms*.

The formula for r is said to be in *sum of products* form, for obvious reasons.

With n variables there are 2^n possible minterms. Each minterm involves all n variables, and each variable is either negated ($\bar{x}$) or not negated (just x).

Minterms and the Truth Table

Each minterm corresponds to one row of the truth table, i.e. to one combination of values (0 or 1) of the variables.

The minterm corresponds to the row in which the negated variables have value 0 and the non-negated variables have value 1.

The formula for r consists of the minterms corresponding to the truth table rows in which r = 1, ORed together.

x	y	z	r	
0	0	0	0	$\bar{x}\,\bar{y}\,\bar{z}$
0	0	1	0	$\bar{x}\,\bar{y}\,z$
0	1	0	0	$\bar{x}\,y\,\bar{z}$
0	1	1	1	$\bar{x}\,y\,z$
1	0	0	0	$x\,\bar{y}\,\bar{z}$
1	0	1	1	$x\,\bar{y}\,z$
1	1	0	1	$x\,y\,\bar{z}$
1	1	1	1	$x\,y\,z$

$$r = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz$$

Structure of the Circuit

- Notice the structure of the circuit: NOT gates to make negated inputs available, AND gates to produce the required minterms, an OR gate to produce the final output.
- In the same way we can construct a circuit for any function.
- With *m* inputs, and *n* rows with output value 1: *m* NOT, *n* *m*-input AND, 1 *n*-input OR.
- This circuit is more complex than the original majority voting circuit. We will have more to say about this later.

Equality Test

Suppose we want to design a circuit which implements the equality test function on two inputs. That is, we want to compute *r* as a function of *x* and *y*, where *r* will be 1 if *x* and *y* have the same value, and 0 if *x* and *y* have different values.

For two variables there are 4 possible minterms, which correspond to the rows of the truth table as follows.

x	y	r	
0	0	1	$\overline{x}\,\overline{y}$
0	1	0	$\overline{x}\,y$
1	0	0	$x\overline{y}$
1	1	1	$x\,y$

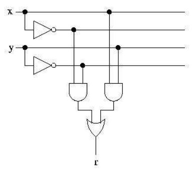
Equality Test

The formula for *r* is the OR of the two minterms corresponding to the rows in which *r* = 1.

$$\boxed{r = \overline{x}\,\overline{y} + xy}$$

x	y	r	
0	0	1	$\overline{x}\,\overline{y}$
0	1	0	$\overline{x}\,y$
1	0	0	$x\overline{y}$
1	1	1	$x\,y$

The circuit:



Parity

The *parity* of a binary word is determined by the number of 1s in it:
if it contains an odd number of 1s then the parity is 1 (or *odd*);
if it contains an even number of 1s then the parity is 0 (or *even*).

(Mathematically the parity of a number is sometimes said to be *odd* for odd numbers and *even* for even numbers. But for binary words, parity is based on the number of 1s.)

Example: 1010 has even parity. 1101 has odd parity.
11111111 has even parity. 00101010 has odd parity.

Parity

The parity function for a 3 bit word *xyz* is defined by the following truth table, which also shows the minterm for each row.

x	y	z	p	
0	0	0	0	$\overline{x}\,\overline{y}\,\overline{z}$
0	0	1	1	$\overline{x}\,\overline{y}\,z$
0	1	0	1	$\overline{x}\,y\,\overline{z}$
0	1	1	0	$\overline{x}\,y\,z$
1	0	0	1	$x\overline{y}\,\overline{z}$
1	0	1	0	$x\overline{y}\,z$
1	1	0	0	$x\,y\,\overline{z}$
1	1	1	1	$x\,y\,z$

The formula for *p* is the OR of the four minterms corresponding to the rows in which *p* = 1.

$$\boxed{p = \overline{x}\,\overline{y}\,z + \overline{x}y\overline{z} + x\overline{y}\,\overline{z} + xyz}$$

Exercises

1. Draw a circuit for the parity function, in the same way that we did for majority voting.

2. Find an equivalent circuit, which uses just two XOR gates. Prove that it is equivalent, both by truth tables and by using the laws of boolean algebra.

Applications of Parity

Parity checking can be used for error detection, for example in computer memory.

Suppose that each memory location stores an 8 bit word. A memory device with parity checking would actually store 9 bits per word, where the 9th bit is the parity of the original 8 bit word. The parity bit is calculated when a word is stored in memory.

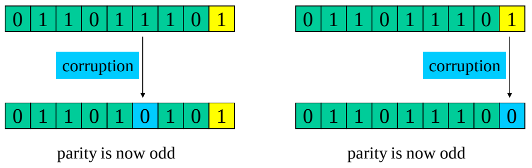
0	1	1	0	1	1	1	0	1

1	0	1	0	0	1	1	0	0

in both cases, the 9 bit word has even parity

Applications of Parity

When a 9 bit word is read from memory, its parity is calculated. If a single bit within the word has been corrupted (changed from 0 to 1 or from 1 to 0) then the parity of the word will be odd.



parity is now odd

parity is now odd

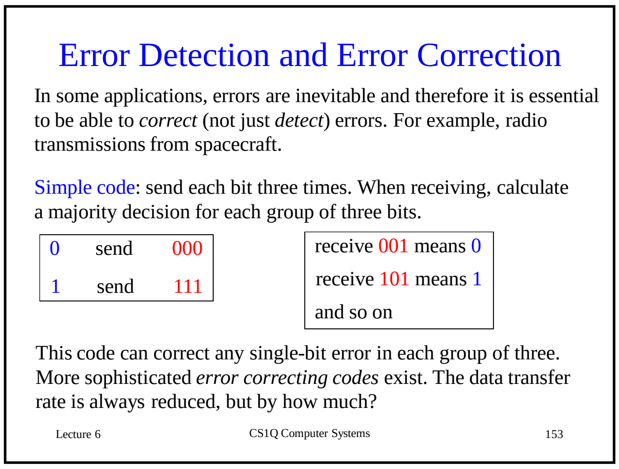
The computer can tell that a memory error has occurred (it could be because of a power fluctuation, for example) and do something (but what?)

Applications of Parity

The same idea can be used when transmitting data over a network. Instead of sending an 8 bit word, send a 9 bit word which includes a parity bit. The receiver can check the parity.

Parity checking cannot correct errors, because it is not possible to work out which bit was corrupted. In a networking application, the corrupted word would be retransmitted.

Parity checking can only detect single bit errors, because if two bits are changed then the parity remains the same. It might be acceptable to assume that the probability of two errors in the same word is very small.



Error Detection and Error Correction

In some applications, errors are inevitable and therefore it is essential to be able to *correct* (not just *detect*) errors. For example, radio transmissions from spacecraft.

Simple code: send each bit three times. When receiving, calculate a majority decision for each group of three bits.

0	send	000
1	send	111

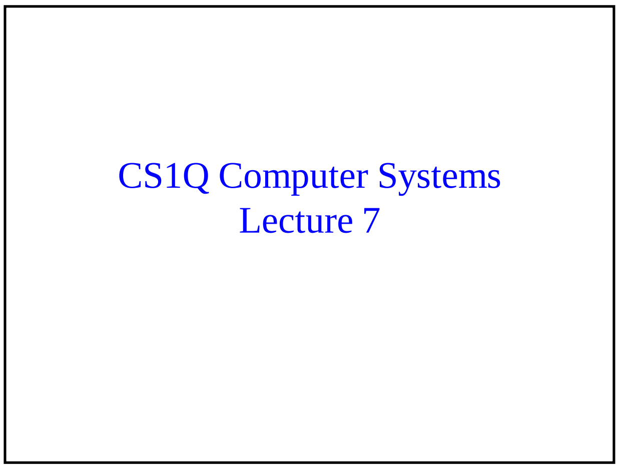
receive 001 means 0

receive 101 means 1

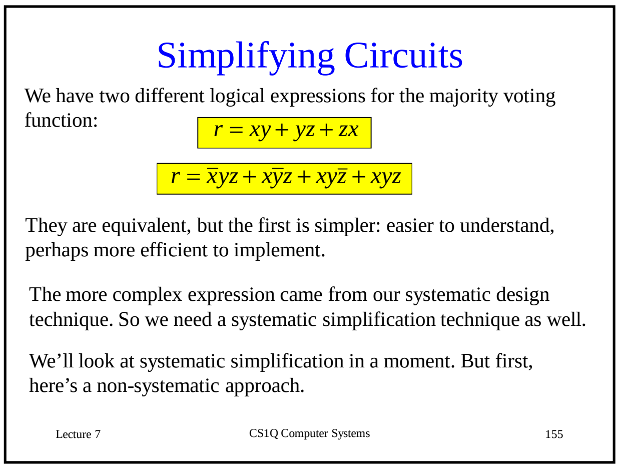
and so on

This code can correct any single-bit error in each group of three. More sophisticated *error correcting codes* exist. The data transfer rate is always reduced, but by how much?

Lecture 6 CS1Q Computer Systems 153



CS1Q Computer Systems
Lecture 7



Simplifying Circuits

We have two different logical expressions for the majority voting function:

$$\boxed{r = xy + yz + zx}$$

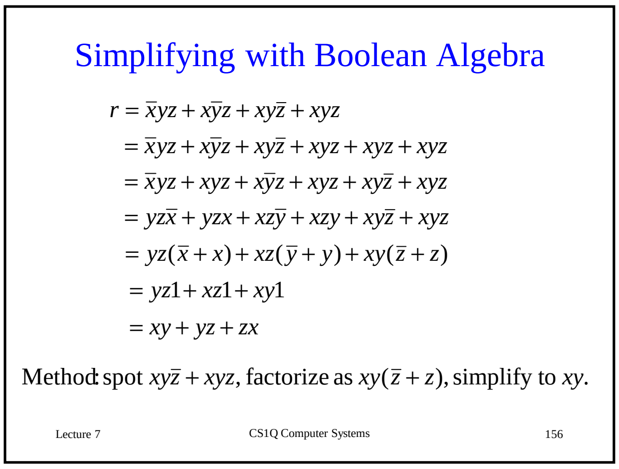
$$\boxed{r = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz}$$

They are equivalent, but the first is simpler: easier to understand, perhaps more efficient to implement.

The more complex expression came from our systematic design technique. So we need a systematic simplification technique as well.

We'll look at systematic simplification in a moment. But first, here's a non-systematic approach.

Lecture 7 CS1Q Computer Systems 155



Simplifying with Boolean Algebra

$$\begin{aligned} r &= \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz \\ &= \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz + xyz + xyz \\ &= \bar{x}yz + xyz + x\bar{y}z + xyz + xy\bar{z} + xyz \\ &= yz\bar{x} + yzx + xz\bar{y} + xzy + xy\bar{z} + xyz \\ &= yz(\bar{x} + x) + xz(\bar{y} + y) + xy(\bar{z} + z) \\ &= yz1 + xz1 + xy1 \\ &= xy + yz + zx \end{aligned}$$

Method: spot $xy\bar{z} + xyz$, factorize as $xy(\bar{z} + z)$, simplify to xy.

Lecture 7 CS1Q Computer Systems 156

Karnaugh Maps

A *Karnaugh map*, or K-map, is an alternative representation of a truth table, which makes it easy to spot when expressions of the form $x + \bar{x}$ can be eliminated.

Example: consider the function $r = xy + y$

and lay out its truth table as a 2 by 2 grid.

x	y	r
0	0	0
0	1	1
1	0	0
1	1	1

$$\longrightarrow \quad \begin{array}{cc|c|c|} & & \bar{y} & y \\ & & 0 & 1 \\ \hline \bar{x} & 0 & 0 & 1 \\ \hline x & 1 & 0 & 1 \\ \hline \end{array}$$

This grid is the Karnaugh map for *r*.

Karnaugh Maps

In the Karnaugh map, each square corresponds to one of the four combinations of values of *x* and *y*. The values of *x* and *y* are shown at the left hand side and along the top.

x	y	r
0	0	0
0	1	1
1	0	0
1	1	1

$$\begin{array}{cc|c|c|} & & \bar{y} & y \\ & & 0 & 1 \\ \hline \bar{x} & 0 & 0 & 1 \\ \hline x & 1 & 0 & 1 \\ \hline \end{array}$$

minterm
$\bar{x}\bar{y}$

The rows are labelled with $\bar{x}$ and x, and the columns with $\bar{y}$ and y, to show which axis corresponds to which variable and also to indicate which minterm corresponds to which square in the grid.

Karnaugh Maps

From the Karnaugh map, we can write down a formula for *r* by OR-ing together the minterms corresponding to the squares which contain 1.

$$r = \bar{x}y + xy$$

This can be factorised as

$$r = (\bar{x} + x)y$$

and therefore simplifies to

$$r = y$$

$$\begin{array}{cc|c|c|} & & \bar{y} & y \\ & & 0 & 1 \\ \hline \bar{x} & 0 & 0 & 1 \\ \hline x & 1 & 0 & 1 \\ \hline \end{array}$$

This is just what we did for the majority voting function, but now notice that the presence of $\bar{x} + x$ in the formula has a visual interpretation: there are two adjacent 1s in the *y* column, covering both the x and $\bar{x}$ squares.

Exercise

Draw a Karnaugh map for the function

$$r = x + \bar{x}\bar{y}$$

Simplification with K-Maps

Each square in the K-map corresponds to a minterm. Each 1 by 2 rectangle (either horizontal or vertical) corresponds to one of the variables, either negated or non-negated.

Any collection of squares and rectangles which cover all the 1s, corresponds to a logical formula for the function defined by the K-map.

By choosing a covering in which the rectangles are as large as possible (maybe overlapping), we obtain the simplest formula.

(What do we mean by “simplest”? We are trying to minimise the number of terms OR-ed together, and minimise the complexity of each term. This simplification process is often called *minimisation*.)

Simplification with K-Maps

Example: the function $r = x + \bar{x}y$ has this K-map:

		$\bar{y}$	y
		0	1
$\bar{x}$	0	0	1
x	1	1	1

Different coverings of the 1s give different formulae.

Three squares: $r = x\bar{y} + xy + \bar{x}y$

Square and horizontal rectangle: $r = x + \bar{x}y$

Square and vertical rectangle: $r = y + x\bar{y}$

Horizontal and vertical rectangles (shown): $r = x + y$

K-Maps for 3 Variables

The Karnaugh map for a function of 3 variables consists of a grid of 8 squares. Here is the K-map for the majority voting function.

	$\bar{y}$	y	y	$\bar{y}$
$\bar{x}$	0	0	1	0
x	0	1	1	1
	$\bar{z}$	$\bar{z}$	z	z

The 0s and 1s around the edges have been omitted. Remember that a negated label corresponds to 0 and a non-negated label to 1.

Notice that the negated *y*s appear in a different pattern from the negated *z*s. This means that again each square corresponds to one of the 8 minterms.

The three rectangles of 1s correspond to *xy*, *yz* and *xz*. OR-ing them together gives the simplified formula for majority voting:

$$xy + yz + zx$$

Labelling 3 Variable K Maps

It is essential to label the rows and columns correctly, otherwise the technique of finding overlapping rectangles does not work.

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

It must be the case that any two adjacent squares (including “wrapping round” from top to bottom) have labels which differ by negation of *exactly one variable*. There are several labelling schemes which have this property, but for safety you should *memorise the labelling which is used in the lecture notes*.

Another Example

We will use a Karnaugh map to minimise the formula

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

First we fill in the K-map. The terms with two variables correspond to 2 by 1 rectangles, and the other terms are just squares.

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

The remaining squares are 0.

Another Example

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

Now we can find collections of rectangles which cover the 1s.

Another Example

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

Now we can find collections of rectangles which cover the 1s.

Three horizontal 2 by 1 rectangles: $x\bar{z} + \bar{x}y + xz$

Another Example

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

Now we can find collections of rectangles which cover the 1s.

Three horizontal 2 by 1 rectangles: $x\bar{z} + \bar{x}y + xz$

2 by 2 square and two 1 by 1 squares: $y + x\bar{y}\bar{z} + x\bar{y}z$

Another Example

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

Now we can find collections of rectangles which cover the 1s.

Three horizontal 2 by 1 rectangles: $x\bar{z} + \bar{x}y + xz$

2 by 2 square and two 1 by 1 squares: $y + x\bar{y}\bar{z} + x\bar{y}z$

Combining the two 1 by 1 squares: $y + x\bar{y}$

Another Example

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

Now we can find collections of rectangles which cover the 1s.

Three horizontal 2 by 1 rectangles: $x\bar{z} + \bar{x}y + xz$

2 by 2 square and two 1 by 1 squares: $y + x\bar{y}\bar{z} + x\bar{y}z$

Combining the two 1 by 1 squares: $y + x\bar{y}$

4 by 1 and 2 by 1 rectangles: $x + \bar{x}y$

Another Example

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

Now we can find collections of rectangles which cover the 1s.

Three horizontal 2 by 1 rectangles: $x\bar{z} + \bar{x}y + xz$

2 by 2 square and two 1 by 1 squares: $y + x\bar{y}\bar{z} + x\bar{y}z$

Combining the two 1 by 1 squares: $y + x\bar{y}$

4 by 1 and 2 by 1 rectangles: $x + \bar{x}y$

4 by 1 rectangle and 2 by 2 square: $x + y$ (the simplest formula)

Exercise

In the same way, minimise the expression

$$xy + \bar{y}z + \bar{x}\,\bar{y}\,\bar{z}$$

K-Maps for 4 Variables

A Karnaugh map for a function of 4 variables *x*, *y*, *z*, *w* uses the following grid.

$$\begin{array}{c|cccc|c} & \bar{y} & y & y & \bar{y} & \\ \hline \bar{x} & & & & & \bar{w} \\ \bar{x} & & & & & w \\ x & & & & & w \\ x & & & & & \bar{w} \\ \hline & \bar{z} & \bar{z} & z & z & \end{array}$$

The left and right columns are adjacent. The top and bottom rows are adjacent. Larger K-maps can be constructed (e.g. for 5 variables, take 2 copies of this K-map, one labelled *v* and the other labelled $\bar{v}$) but are less useful because it is more difficult to spot rectangles.

Example: Gray Code

Gray code is an alternative binary counting sequence. The Gray code sequence for 3 bits is as follows:

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

At each step, exactly one bit is changed, and it is the rightmost bit such that a change produces a word which has not already occurred.

Exercise: use this rule to work out the Gray code sequence for other numbers of bits.

We will design a circuit to calculate the next 3 bit Gray code. Given a 3 bit input *xyz*, the 3 bit output *x'y'z'* is the word which follows *xyz* in the Gray code sequence. For input 100 the output is 000.

Gray Code Truth Tables

By combining three truth tables we can show *x'*, *y'* and *z'* as functions of *x*, *y* and *z*.

x	*y*	*z*	*x'*	*y'*	*z'*
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	1

Gray Code Karnaugh Maps

For each of *x'*, *y'*, *z'* we can draw a Karnaugh map and find a minimised formula.

For *x'*:

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

For *z'*:

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

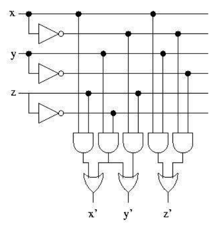
For *y'*:

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

$$x' = y\bar{z} + xz$$
$$y' = y\bar{z} + \bar{x}z$$
$$z' = xy + \bar{x}\,\bar{y}$$

Gray Code Circuit

Notice that the expression $y\bar{z}$ occurs twice, so we can reduce the size of the circuit by only calculating it once. Also notice that $z' = \overline{x \oplus y}$, which means that if XOR gates are available then the circuit can be simplified further.



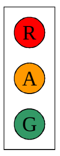
CS1Q Computer Systems
Lecture 8

Traffic Lights

Suppose we want to design a controller for a set of traffic lights. British traffic lights have three lights, coloured red, amber and green.

There are four possible combinations of the lights:

- Red
- Red and Amber
- Green
- Amber



The first step is to design a circuit which has an input representing which of the four combinations is required, and generates an output (1 or 0, representing on or off) for each of the three lights.

Traffic Lights

If we number the combinations 0 to 3, we can construct a truth table.

Number	Red	Amber	Green
0	1	0	0
1	1	1	0
2	0	0	1
3	0	1	0

How should the input be fed into the circuit? One way is to use four input wires, labelled d_0, d_1, d_2, d_3. To select combination *n*, we will input 1 on d_n and 0 on the other inputs.

Traffic Lights

Here is the truth table with the *d* inputs. (It is not a complete truth table because not all combinations of the inputs are listed.)

d_0	d_1	d_2	d_3	Red	Amber	Green
1	0	0	0	1	0	0
0	1	0	0	1	1	0
0	0	1	0	0	0	1
0	0	0	1	0	1	0

Exercise: try to spot simple definitions for Red, Amber and Green.

Reducing the Number of Inputs

Using 4 inputs to represent a choice of 4 combinations is inefficient. If we write the combination number in binary then only 2 bits are needed, and a 2 bit binary number corresponds to 2 input wires.

In general the difference is between n inputs and 2^n inputs for representing a choice between 2^n possibilities. As *n* becomes larger, this difference becomes more significant.

If the 2 bit binary input is $i_1\, i_0$ then the truth table becomes:

i_1	i_0	Red	Amber	Green
0	0	1	0	0
0	1	1	1	0
1	0	0	0	1
1	1	0	1	0

Exercise

Work out formulae for Red, Amber and Green.

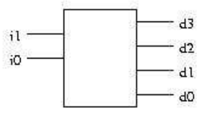
i_1	i_0	Red	Amber	Green
0	0	1	0	0
0	1	1	1	0
1	0	0	0	1
1	1	0	1	0

This can be done by using Karnaugh maps, but we can spot some shortcuts.

Decoders

A decoder is a circuit which has n inputs and 2^n outputs, and converts a binary number on the inputs into a 1 on just one of the outputs.

A 2-4 decoder:



and its truth table:

i_1	i_0	d_3	d_2	d_1	d_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

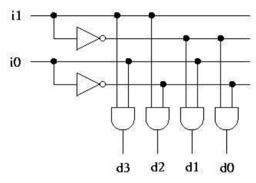
We can immediately see that each output corresponds to one of the four minterms:

$$d_3 = i_1 i_0 \qquad d_1 = \bar{i}_1 i_0$$

$$d_2 = i_1 \bar{i}_0 \qquad d_0 = \bar{i}_1 \bar{i}_0$$

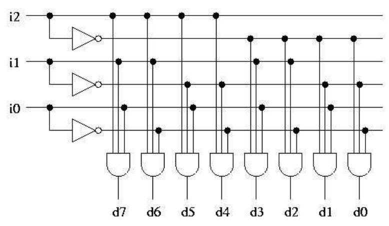
2-4 Decoder Circuit

The following circuit generates all four minterms from two inputs, and implements the 2-4 decoder.



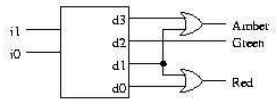
3-8 Decoder Circuit

Larger decoders can be implemented in the same way. Here is a 3-8 decoder.



Traffic Lights with a Decoder

Using a 2-4 decoder, the circuit which generates traffic light combinations is as follows.



We no longer have to think about the problem of invalid inputs.

To complete the traffic light controller, we just need to make the inputs cycle through the binary representations of the numbers 0,1,2,3. We will see how to do this later in the course.

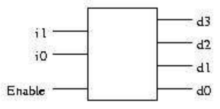
Exercises

The smallest possible decoder is a 1-2 (1 input, 2 outputs). How is this implemented?

How many components (inverters and AND gates) are needed to build an $n-2^n$ decoder? What if only 2-input (not larger) AND gates are used?

Decoders with Enable

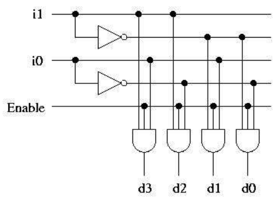
A standard decoder typically has an additional input called *Enable*.



If the *Enable* input is 1 then the component works as a decoder.
If the *Enable* input is 0 then the component is inactive. Exactly what this means depends on the details of the implementation, but for now we can interpret it as meaning that all the outputs are 0.

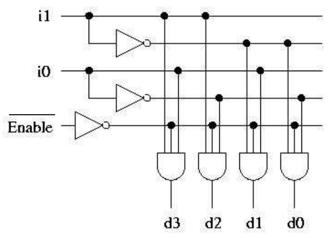
2-4 Decoder with Enable

The *Enable* input is fed into the AND gates which produce the outputs.



Many components have an *Enable* input which works in this way.
Sometimes the Enable input is *active high* (as in this case); sometimes it is *active low*.

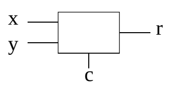
2-4 Decoder with Active Low Enable



Selecting Between Two Functions

Suppose we want a circuit which can do one of two things, depending on the value of a control input.

Example:



r = x OR y if c=0
= x XOR y if c=1

```
function r(c, x, y : Boolean) return Boolean is
begin
     if c then return x XOR y
          else return x OR y
     end if;
end r;
```

Our Standard Design Technique

We can design a circuit for *r* in the usual way:

c	*x*	*y*	*r*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$$\begin{array}{c|cccc} & \bar{x} & x & x & \bar{x} \\ \hline \bar{c} & 0 & 1 & 1 & 1 \\ c & 0 & 1 & 0 & 1 \\ \hline & \bar{y} & \bar{y} & y & y \end{array}$$

$$r = \bar{x}y + x\bar{y} + \bar{c}x$$

but there are several problems with this approach.

Problems

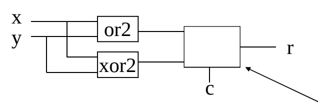
The final formula for *r* doesn't have the same structure as the original specification of the function. Where has the *x* OR *y* gone? If we wanted to change OR to AND in the specification of *r*, we would have to repeat the whole design process.

In a large system we might have complex circuits, computing functions *f* and *g*, say, instead of OR and XOR. We don't want to redesign *f* and *g* into a new circuit which includes the functionality of both.

In order to work with large and complex designs, it is essential to be able to treat parts of the design as *black boxes* which are combined in standard ways.

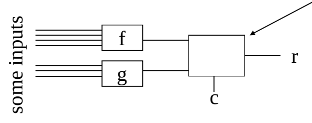
What We Really Want

We want to end up with a circuit which looks like this:



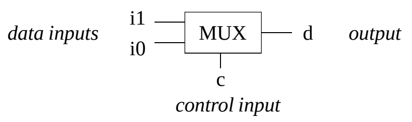
This is called a *multiplexer*.

or more generally like this:



The 2-1 Multiplexer

The 2-1 multiplexer has 2 data inputs, 1 output, and a control input.



Specification:

```
if c = 0
then d = i0
else d = i1
endif
```

The 2-1 Multiplexer

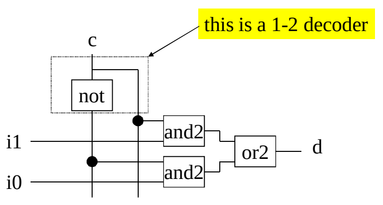
Using the usual technique:

c	*i1*	*i0*	*d*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$\begin{array}{c|cccc} & \bar{i}_1 & i_1 & i_1 & \bar{i}_1 \\ \hline \bar{c} & 0 & 0 & 1 & 1 \\ c & 0 & 1 & 1 & 0 \\ \hline & \bar{i}_0 & \bar{i}_0 & i_0 & i_0 \end{array}$$

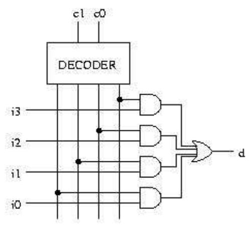
$$d = \bar{c}i_0 + ci_1$$

The 2-1 Multiplexer



A 4-1 Multiplexer

The 2-1 multiplexer is constructed from a 1-2 decoder, 2 AND gates and an OR gate. Using the same structure we can make a 4-1 multiplexer.



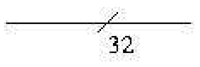
Larger multiplexers (in general, $2^n - 1$) are constructed similarly.

Multibit Multiplexers

The basic $2^n - 1$ multiplexer is a switch, allowing one of 2^n inputs to be connected to the output. Each input consists of a single bit.

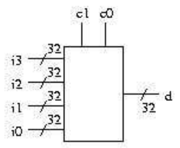
It is often necessary to consider a group of wires as a single signal. For example, in a 32-bit microprocessor, all data is handled in blocks of 32 bits, which means that 32 wires are needed to carry a value from one part of the circuit to another.

A collection of wires which form a single signal is called a *bus*. In circuit diagrams, a bus is represented by a single line with a short diagonal line across it, labelled to indicate the *width* of the bus.



Multibit Multiplexers

It is often necessary to use multiplexers to switch whole buses. In diagrams, we simply draw a multiplexer as usual, with buses of width as inputs and output. Bus notation may also be used to indicate the width of the control input signal.



This example shows a 4-1 multiplexer on a 32 bit bus. A 32 bit multiplexer can be implemented with 32 basic multiplexers, all sharing the same control inputs.

Multiplexers and Logic Functions

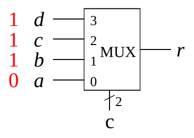
Any logic function of *n* inputs can be implemented with a $2^n - 1$ multiplexer. For example, for a 2 input logic function, call the inputs *x* and *y* and the result *r*, and let the truth table be: (*a, b, c, d* are each either 0 or 1)

x	*y*	*r*
0	0	*a*
0	1	*b*
1	0	*c*
1	1	*d*

Multiplexers and Logic Functions

x	*y*	*r*	OR
0	0	*a*	0
0	1	*b*	1
1	0	*c*	1
1	1	*d*	1

The following circuit implements this function, because *x* and *y*, when connected to the control inputs, select the correct row of the truth table.



Exercise

The previous slide shows how to implement any logic function of 2 inputs, by using a 4-1 multiplexer. It is actually possible to implement the AND and OR functions with a 2-1 multiplexer. Work out how to do this. Also work out how to use a 2-1 multiplexer to implement the NOT function.

Multiplexers and Logic Functions

Any logic function of 3 inputs can be implemented with a 4-1 multiplexer and an inverter, as follows.

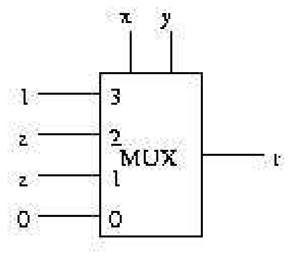
Let the inputs be *x*, *y*, *z*. Connect *x* and *y* to the control inputs of the multiplexer. For each combination of values of *x* and *y*, one of the following cases must apply.

- The output is 0, regardless of the value of *z*.
- The output is 1, regardless of the value of *z*.
- The output is equal to *z*.
- The output is equal to $\overline{z}$.

For each combination of values of *x* and *y*, the multiplexer input which is selected by that combination is connected to either 0, 1, *z* or $\overline{z}$, depending on which of the above cases applies.

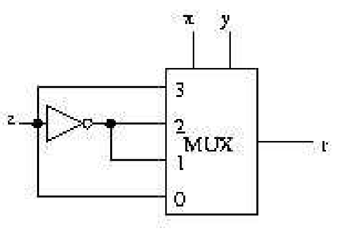
Example: Majority Voting

x	*y*	*z*	*r*	
0	0	0	0	0
0	0	1	0	0
0	1	0	0	*z*
0	1	1	1	*z*
1	0	0	0	*z*
1	0	1	1	*z*
1	1	0	1	1
1	1	1	1	1



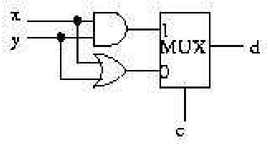
Example: Parity

x	*y*	*z*	*r*	
0	0	0	0	*z*
0	0	1	1	*z*
0	1	0	1	$\overline{z}$
0	1	1	0	$\overline{z}$
1	0	0	1	$\overline{z}$
1	0	1	0	$\overline{z}$
1	1	0	0	*z*
1	1	1	1	*z*



Multiplexer Applications

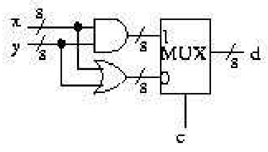
Using a multiplexer we can build a circuit which allows one of a number of operations to be chosen, and applied to the inputs (this is where we started). For example, here is a circuit which gives a choice between AND and OR.



For a choice between more operations, a larger multiplexer can be used. More generally, multiplexers are used to give a choice between a number of different sources of data, not necessarily a number of different operations on the same data.

Multiplexer Applications

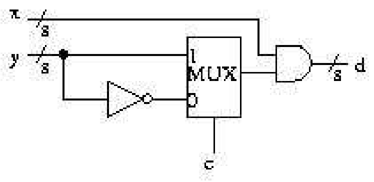
The same idea can be used for operations on multibit words. For example, using 8 bit words, we just replace every wire (except the control wire) by an 8 bit bus.



In this circuit, the AND operation is extended to 8 bit words by operating on each bit position independently (and similarly OR): e.g. 11010010 AND 01110110 = 01010010.

Multiplexer Applications

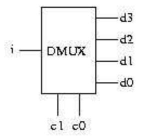
A similar example, which is relevant to the exercises in Lab 3, is calculating either *x* AND *y* or *x* AND (NOT *y*), where again *x* and *y* are multibit values.



These examples begin to show how the ALU of a microprocessor can be implemented. We'll see more details later.

Demultiplexers

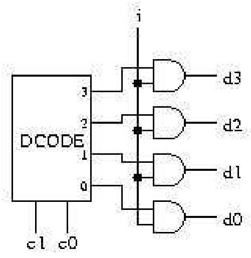
A demultiplexer is the opposite of a multiplexer. There is one data input, whose value appears on one of the data outputs, depending on the value of the control inputs. Here is a 1-4 demultiplexer.



If the control inputs *c1 c0* represent the number *n* in binary, then the value of *i* is copied to output *dn*. Depending on the details of the electronic implementation, the other outputs might be 0, or might be in a disconnected state.

Demultiplexers

It is straightforward to implement a demultiplexer. The circuit uses a decoder in a similar way to the implementation of a muliplexer.

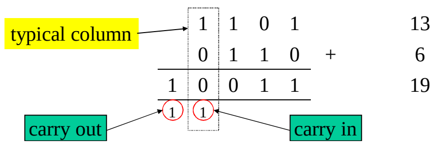


CS1Q Computer Systems
Lecture 9

Addition

We want to be able to do arithmetic on computers and therefore we need circuits for arithmetic operations. Naturally, numbers will be represented in binary. We'll start with addition.

Recall that addition in binary is just like addition in decimal:



Each column: add three bits (two from the original numbers, one carry input) and produce two bits (sum and carry output).

Designing an Adder

Here is the truth table for the single bit addition function. The bits being added are *x* and *y*. The carry input is *Cin*. The sum is *s* and the carry output is *Cout*.

x	*y*	*Cin*	*Cout*	*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

Notice that the *Cout* and *s* columns, interpreted as a 2 bit binary number, are simply the sum of the *x*, *y* and *Cin* columns.

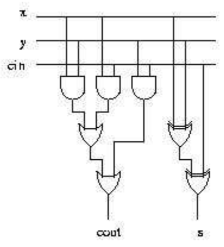
It turns out that *Cout* is the majority voting function from Lecture 5, and *s* is the parity function from Lecture 6.

Implementing the Adder

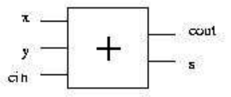
We now know that

$$s = x \oplus y \oplus c_{in}$$
$$c_{out} = xy + yc_{in} + c_{in}x$$

so we can construct a circuit:



A single bit adder is usually represented like this:



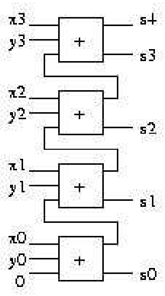
Multi-Bit Addition

Addition of multi-bit numbers is achieved by chaining single bit adders together. Here is a 4 bit adder. The inputs are *x3 x2 x1 x0* and *y3 y2 y1 y0*. The output is *s4 s3 s2 s1 s0* (a 5 bit number).

The carry out from each adder is fed into the carry in of the next adder. The carry in of the adder for the least significant bit is set to 0.

Note that the sum of two *n* bit numbers can always be expressed in *n+1* bits:
if $x < 2^n$ and $y < 2^n$ then

$$x + y < 2^n + 2^n = 2^{(n+1)}$$



Half Adders

In effect we have directly implemented addition of *three* binary digits. Let’s consider addition of just two digits, which is obviously more fundamental, even though it does not directly correspond to the original calculation.

Adding two bits *x* and *y* produces a sum *s* and a carry *c*:

x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

We can immediately see that

$c = xy$

$s = x \oplus y$

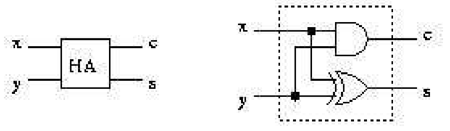
Half Adders

x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$c = xy$

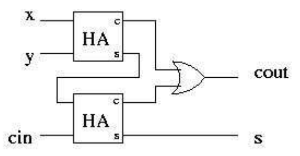
$s = x \oplus y$

The half adder consists of an AND gate and an XOR gate:



Two Halves Make a Whole

The following circuit uses two half adders to implement a full adder.



Exercise: use a truth table to check that this circuit is correct.

Ripple Carry

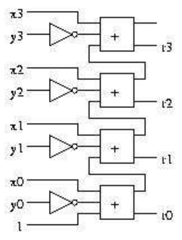
The electronic implementations of logic gates do not work instantaneously: when the inputs change there is a short delay, perhaps a few picoseconds, before the outputs change. In our multi-bit adder, these delays accumulate because the carry bits have to propagate all the way along the circuit. This adder design is called *ripple carry.* The more bits, the longer the delay.

Ripple carry delays would be very significant in a fast CPU. More sophisticated adder designs exist, which use various shortcuts to calculate carry bits without propagating them along the whole word. For more details, consult the books.

Subtraction

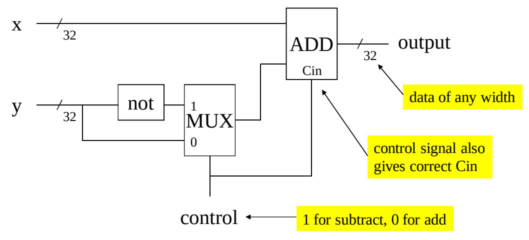
To calculate *x* - *y* we calculate *x* + *(-y)* where -*y* is calculated in the 2s complement representation by inverting all the bits of *y* and then adding 1. A modification of the addition circuit does the trick: NOT gates do the inversion, and the 1 can easily be added by connecting the rightmost carry input to 1 instead of 0.

The final carry output is ignored so that we get a 4 bit result. When working with 2s complement numbers, the final carry does not allow a 5 bit result to be produced.



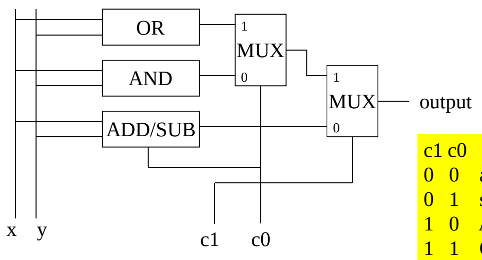
An Add/Subtract Unit

We can construct a circuit which either adds or subtracts, under the control of an input signal. A 2-1 multiplexer is used to select either plain or inverted values of the second input.


x
32
y
32
not
1
MUX
0
ADD
Cin
32
output
data of any width
control signal also gives correct Cin
control
1 for subtract, 0 for add

A Simple ALU

Using similar ideas, here is an ALU with 4 functions: add, subtract, AND, OR.



c1	c0	
0	0	add
0	1	sub
1	0	AND
1	1	OR

Other Mathematical Operations

There is a sequence of mathematical operations of increasing complexity:

addition/subtraction
multiplication
division
square root
transcendental functions (*log*, *sin*, *cos*, …)
…

Where is the hardware/software boundary?

Other Mathematical Operations

We have seen that integer addition and subtraction are easy to implement in hardware.

We have also seen that integer multiplication is easy to implement in software (e.g. in assembly language for the IT Machine). More complex mathematical operations can be implemented by more complex software.

For simple CPUs (e.g. microprocessors of the late 1970s/early 1980s, such as the 6502 or Z80) this is a natural place for the hardware/software boundary.

Modern microprocessors are more complex (e.g. Pentium 4 computes transcendental functions for 128 bit floating point in hardware).

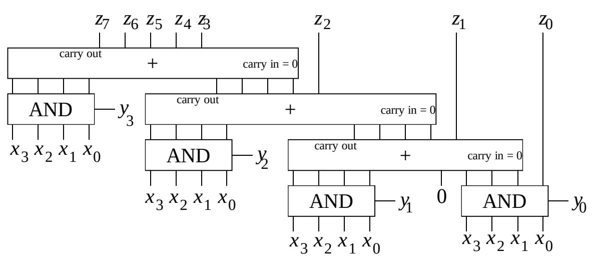
Multiplication

We can design a circuit for integer multiplication. If we multiply two 4 bit numbers $x = x_3x_2x_1x_0$ and $y_3y_2y_1y_0$ then the result is an 8 bit number $z_7z_6z_5z_4z_3z_2z_1z_0$.

$$\begin{aligned} x \times y_3y_2y_1y_0 &= x \times (y_3 \times 8 + y_2 \times 4 + y_1 \times 2 + y_0) \\ &= x \times y_3 \times 8 + x \times y_2 \times 4 + x \times y_1 \times 2 + x \times y_0 \\ &= (x \wedge y_3) \times 8 + (x \wedge y_2) \times 4 + (x \wedge y_1) \times 2 + x \wedge y_0 \end{aligned}$$

	$x_3 \wedge y_3$	$x_2 \wedge y_3$	$x_1 \wedge y_3$	$x_0 \wedge y_3$	0	0	0	
		$x_3 \wedge y_2$	$x_2 \wedge y_2$	$x_1 \wedge y_2$	$x_0 \wedge y_2$	0	0	
			$x_3 \wedge y_1$	$x_2 \wedge y_1$	$x_1 \wedge y_1$	$x_0 \wedge y_1$	0	
				$x_3 \wedge y_0$	$x_2 \wedge y_0$	$x_1 \wedge y_0$	$x_0 \wedge y_0$	+
z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0	

Multiplication



Multiplication

Any calculation which can be done in a fixed number of steps can be converted into a circuit in a similar way. Such a circuit is faster than a software solution (but not instant). But the circuit may be large: for multiplication, the size of the circuit is proportional to the *square* of the word length.

Key point: there's a trade-off between execution time, and space (area on the CPU chip). With older manufacturing technologies, space was at a premium, therefore hardware operations stopped at addition. Nowadays, time is more significant.

In practice, a circuit for a complex operation such as division is more likely to be designed as a *state machine* - more details later.

CS1Q Computer Systems
Lecture 10

Combinational Circuits

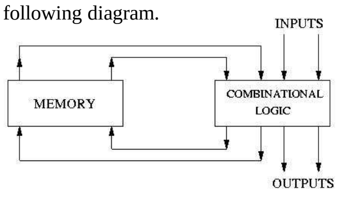
All the circuits we have seen so far are *combinational*, meaning that the output depends only on the present inputs, not on any previous inputs. Combinational circuits have no memory, no state information.

Some circuits which we might want to build are obviously not combinational.

- A traffic light controller must remember which point in the sequence has been reached.
- A CPU must remember which instruction it has to execute next. (Also the contents of all the registers. The RAM is further state information if we consider the computer as a whole.)

Sequential Circuits

Circuits with memory are called *sequential*. Their general structure is shown by the following diagram.



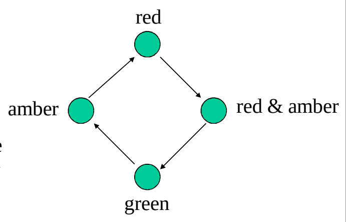
To predict the behaviour of a sequential circuit, we need to know which state it is in, and how the next state and the outputs depend on the current state and the inputs.

Abstract view: the *finite state machine*, a very important concept in CS.

Finite State Machines

A finite state machine is a system which can be in one of a finite number of states, and can change state. A change of state is called a *transition*.

Example: traffic lights.

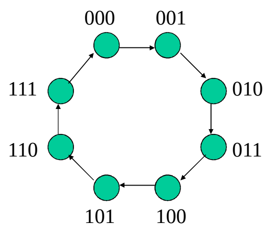


Here there are four states, labelled with the lighting combinations. We think of the transitions as being caused by an external timer or clock.

This is a *transition diagram*.

Finite State Machines

Example: 3 bit binary counter.



Usually the *initial state* is specified: in this case, probably 000.

Finite State Machines

A finite state machine is sometimes called a finite state *automaton* (plural: *automata*), and often abbreviated to FSM or FSA.

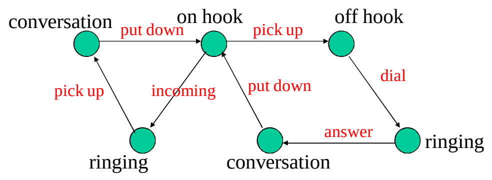
An FSM is an abstract description or specification of a system with several possible states: for example, a sequential circuit.

There are many variations of the basic idea. We can consider unlabelled transitions (as in the previous examples); labelled transitions in which the labels are viewed as inputs; outputs, which can be associated with either states or transitions; distinguished states with particular meanings.

FSMs pop up all over Computing Science. In fact, every computer is a FSM, although it is often convenient to pretend that computers have unlimited memory and an infinite number of possible states.

Finite State Machines

Example: telephone.



Transitions are labelled but we're not describing *how* each transition is activated.

Of course this example leaves out many details of the real telephone system!

Finite State Machines

Example: web site.

Any web site can be viewed as a finite state machine. Each state is a page, and each link is a transition to another state (page).

Exercise: pick a web site and start to draw the transition diagram for the FSM which describes its structure.

(Actually, many web sites contain dynamically generated pages which make it difficult to describe them as FSMs, but there is often an overall structure which can be thought of as an FSM.)

This idea could help to answer questions like: Are all pages reachable? Is it easy to return to the home page?

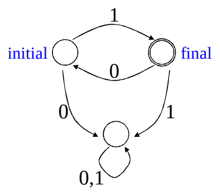
Finite State Machines as Accepters

A particular kind of FSM *accepts* or *recognises* certain input sequences.

Transitions are labelled with symbols from an *input alphabet*.
One state is the *initial* state and some states are *final* or *accepting* states.

If a sequence of input symbols is fed into the FSM, causing transitions, then the sequence is *accepted* if the last transition leads to a final state.

Example: accepting binary sequences of the form 10101…01.



Finite State Machines as Accepters

This is an important idea in Computing Science. Examples and applications occur in many places:
• searching for a particular string in a text file
• recognising programming language keywords, in a compiler
• studying the power of formal models of computation (which sets of strings can be recognised by a FSM?)

For more information, consult any book with “formal languages” or “automata” in the title.

The Mathematical Definition

Mathematically, an accepting finite state machine of the kind we have just illustrated, is defined by the following.

a finite set Q of *states*

a finite set Σ of symbols, called the *input alphabet*

a function $\delta : Q \times \Sigma \to Q$ called the *transition function*

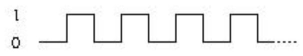
a state $q_0 \in Q$ called the *initial state*

a set $F \subseteq Q$ of *final states*

(You are not expected to know this for the exam; but it is important to be familiar with the informal idea of a FSM.)

Synchronous Systems

Sequential circuits are usually *synchronous*, which means that their behaviour is controlled by a clock. The clock is a signal which oscillates between 0 and 1.



Once per clock cycle the circuit changes state. The inputs are read, their values are combined with the state information to produce outputs and a new state, and the state is updated.

Typical microprocessors are synchronous. The clock speed (in MHz, now moved into GHz) is an often-quoted measure of the processor's performance, although it is not the only factor influencing overall execution speed. (1 MHz = 1 million cycles per second; 1GHz = 1 billion cycles per second.)

Asynchronous Systems

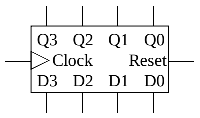
The alternative to a synchronous system is an *asynchronous* system. An asynchronous system has no clock; everything happens as quickly as possible. In principle, however rapidly the inputs change, the outputs will keep up; in practice there are physical limits on the speed.

Asynchronous systems are much more difficult to design, but they do have some advantages, such as low power consumption and low RF interference. Asynchronous microprocessors have been produced (e.g. the Amulet series from Manchester University) and are becoming of interest for application areas such as mobile telephones.

The design of asynchronous systems is an active research area. In this course we will only consider synchronous systems.

Registers

A basic component which allows state information to be stored in a circuit: the *register*. We have seen the use of registers in assembly language programming. Here is a 4 bit register as a component:



At each clock pulse (it is a *synchronous* register), the values of the inputs D3,D2,D1,D0 are stored in the register, replacing the previous stored values. The outputs Q3,Q2,Q1,Q0 are the stored values.
The Reset input sets the stored value to 0000, asynchronously.

Registers

Registers of any size work in the same way. A 32-bit CPU would use 32-bit registers, and so on.

The main memory of a computer (the RAM) can be thought of as a large number of registers, with additional circuitry to enable any desired register to be inspected or updated.

We'll assume for the moment that registers are available, without considering how they are implemented.

Design of Sequential Circuits

The systematic design of sequential circuits is not part of the syllabus of the course. However, looking at some examples will help us to understand the design of CPUs (coming later).

Also, we can emphasise the link between finite state machines and digital circuits.

Two examples:

1. a system which produces a sequence of outputs, driven by a clock
2. the accepting finite state machine from Slide 9.

The Prime Number Machine

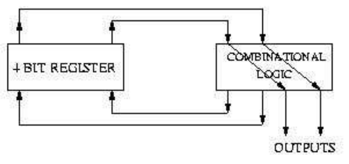
The first example is a circuit which outputs the sequence
2, 3, 5, 7, 11, 13 as 4 bit binary numbers. The circuit will be driven by a clock, so that each clock pulse causes the output to change to the next number in the sequence, returning to 2 after 13.

The sequence of outputs in binary is
0010, 0011, 0101, 0111, 1011, 1101

There are two possible approaches to the design, and we will look at them both.

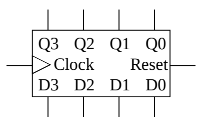
PNM First Design

Idea: store the output word in a 4 bit register.



PNM First Design

Assume that we have a 4 bit register as a standard component.
At each clock pulse, the values of the inputs D3,D2,D1,D0 are stored in the register, replacing the previous stored values. The outputs Q3,Q2,Q1,Q0 are the stored values. The Reset input sets the stored value to 0000, asynchronously.

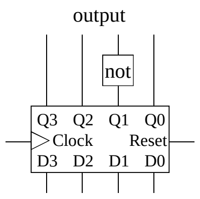


PNM First Design

The Reset input will set the stored value to 0000, but this is not one of the numbers in the sequence. Suppose we want Reset to make the output be 0010. A simple solution is to invert the Q1 output.

This means that the sequence of values for Q3,Q2,Q1,Q0 is

0000, 0001, 0111, 0101, 1001, 1111



PNM First Design

All we need to do now is design a combinational circuit which inputs Q3,Q2,Q1,Q0 and outputs D3,D2,D1,D0 (these are the values which will be stored in the register at the *next* clock cycle).

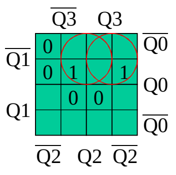
Q3	Q2	Q1	Q0	D3	D2	D1	D0
0	0	0	0	0	0	0	1
0	0	0	1	0	1	1	1
0	0	1	0	X	X	X	X
0	0	1	1	X	X	X	X
0	1	0	0	X	X	X	X
0	1	0	1	1	0	0	1
0	1	1	0	X	X	X	X
0	1	1	1	0	1	0	1

Q3	Q2	Q1	Q0	D3	D2	D1	D0
1	0	0	0	X	X	X	X
1	0	0	1	1	1	1	1
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	0	0	0	0

PNM First Design

Karnaugh maps are a convenient way of handling the *don't care* (X) values. Leaving the X squares blank, we can cover the 1s with rectangles which may also contain blank squares.

Karnaugh map for D3:

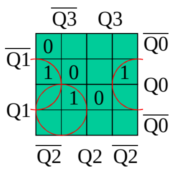


$$D_3 = \overline{Q}_1(Q_2 + Q_3)$$

PNM First Design

Karnaugh maps are a convenient way of handling the *don't care* (X) values. Leaving the X squares blank, we can cover the 1s with rectangles which may also contain blank squares.

Karnaugh map for D2:

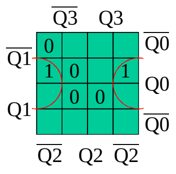


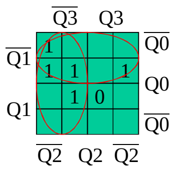
$$D_2 = Q_0\overline{Q}_2 + Q_1\overline{Q}_3$$

PNM First Design

Karnaugh maps are a convenient way of handling the *don't care* (X) values. Leaving the X squares blank, we can cover the 1s with rectangles which may also contain blank squares.

Karnaugh map for D1: Karnaugh map for D0:

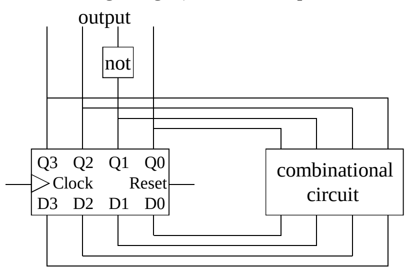




$D_1 = Q_0\overline{Q_2}$ $D_0 = \overline{Q_1} + \overline{Q_3}$

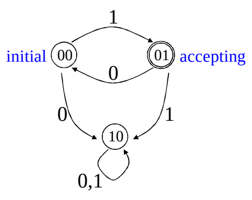
PNM First Design

We end up with the following design (exercise: complete the circuit).



Accepting FSM

Recall the transition diagram for the FSM which accepts binary sequences of the form 10101…01.



We'll use the same design technique as for the Prime Number Machine.

Accepting FSM

There are 3 states so we need 2 bits of state information. We'll use a 2 bit register with outputs (stored values) Q1,Q0 and inputs D1,D0.

There is another input: the current bit from the sequence. Call this I.

At each clock cycle, D1,D0 (which will be the next state) are calculated from Q1,Q0 and I. Here is the truth table:

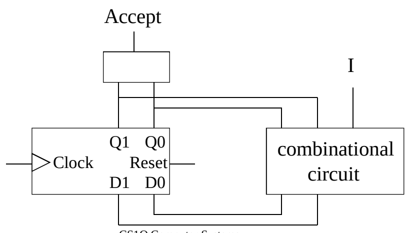
Q1	Q0	I	D1	D0
0	0	0	1	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	X	X
1	1	1	X	X

Exercise: work out formulae for D1,D0 as usual.

Accepting FSM

The final step is to add an output which will indicate whether or not the FSM is in an accepting state.

As the accepting state is state 01, we have $Accept = \overline{Q_1}Q_0$



Another Example: A Multiplier

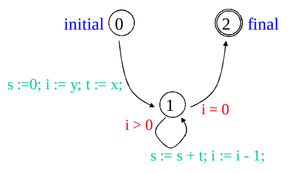
Suppose we want to multiply unsigned integers *x* and *y*, giving result *s*. The following code:

```
s := 0;
i := y;
t := x;
while i > 0 do
   s := s + t;
   i := i - 1;
end while;
```

can be converted into a finite state machine and then into a sequential circuit.

Multiplier

This transition diagram represents the control flow of the program (conditions, assignments):



Multiplier

Suppose that *x* and *y* are 4 bits each, so that the result *s* is 8 bits. The state of the circuit consists of

- a 4 bit register to store *i*
- a 4 bit register to store *x* (so we don’t have to assume that the input signal is maintained)
- an 8 bit register to store *s*
- a 2 bit register to store the state of the controlling FSM.

The combinational logic must update the registers, depending on the state:

- in state 0, load *y* into *i*, *x* into *t*, and 0 into *s*, and enter state 1
- in state 1 (if *i* > 0), load *i*-1 into *i* and *s*+*t* into *s*, and remain in state 1
- in state 1 (if *i* = 0), enter state 2
- in state 2, generate an output signal *finished*

Multiplier

Exercise (challenging): complete the design of this multiplier circuit.

In contrast to the combinational multiplication circuit, whose size is proportional to the square of the number of bits in the inputs, the size of this circuit is proportional to the number of bits in the inputs. However, the multiplication takes *y* +1 clock cycles to complete.

A better solution would be based on the following pseudocode:

```
s := 0; i := y; t := x;
while i > 0 do
  if odd(i) then s := s + t end if;
  i := i div 2; t := t * 2;
end while;
```

End of Part One

On to second half of the notes...