Unit 3: Logic & numbers

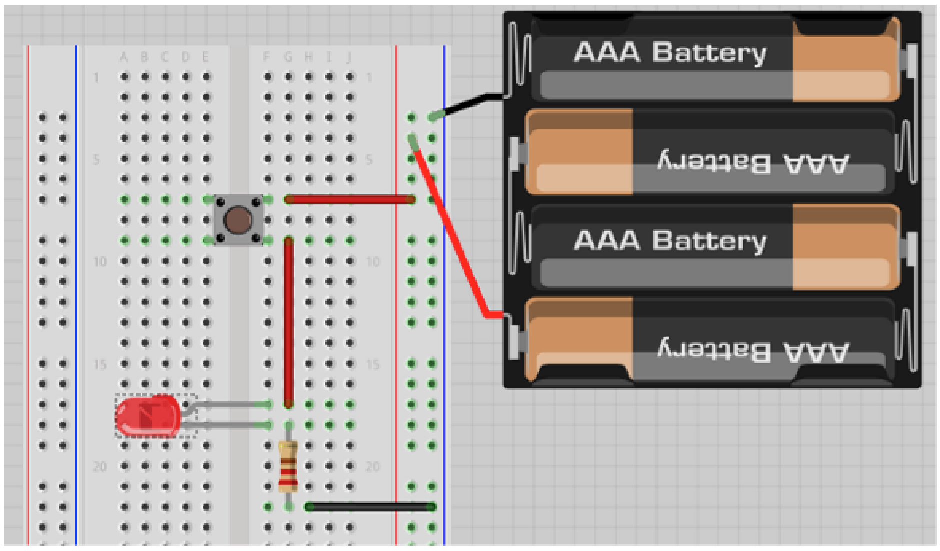
0. Using transistors for logic

Ultimately everything in a computer is reduced to either the presence or absence of an electrical charge. This electrical charge inside transistors is scaled up to form basic circuits that can be used to remember information (ie: act as memory) and perform calculations. A modern Intel CPU has about 1.75 billion transistors in a piece of silicon the size of a fingernail, or 17.185 million transistors per square millimetre. (1)

A transistor is effectively a simple switch. How is it possible to build something as seemingly complex as a computer out of on/off switches? Hopefully this activity will give you a small window into how. We are going to see how logic gates can be physically constructed from simple transistors.

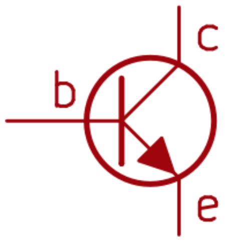
Simple electronics basics

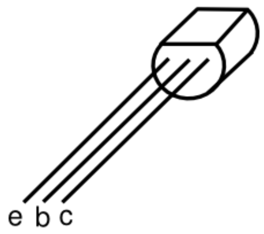
(TODO - further explanation needed. Intro to circuits; wires; breadboards; reading schematics;)



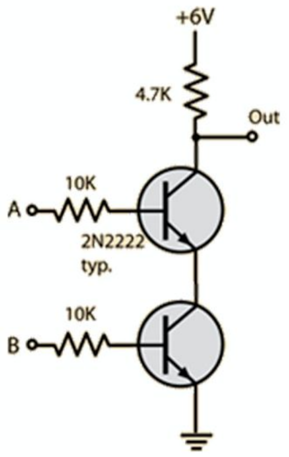
Before going any further you should build the simple push-button and LED scenario sketched out above to test your understanding of how the breadboard functions. Once you have completed the LED/button exercise, it is time to attempt to wire up some logic gates.

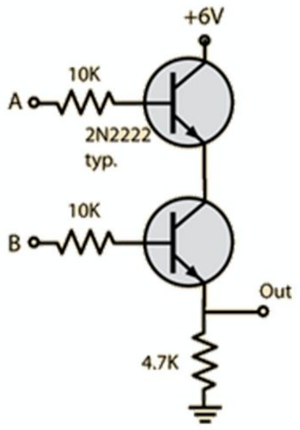
The following shows you how the diagram symbol for a transistor correlates to its physical appearance. You will need to get the order of the pins correct for your circuit to function.

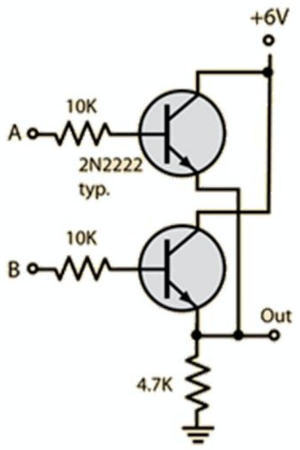


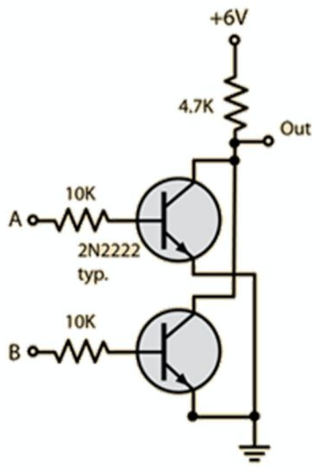


The following are the electrical wiring diagrams for four "logic gates": We have the AND, OR, NAND and NOR gate represented here. Can you hypothesis which is which?


+6V
4.7K
Out
10K
A
2N2222
typ.
10K
B


+6V
10K
A
2N2222
typ.
10K
B
Out
4.7K


+6V
10K
A
2N2222
typ.
10K
B
Out
4.7K


+6V
4.7K
Out
10K
A
2N2222
typ.
10K
B

1. Logic gates & converting circuits to truth tables

Learning objectives

- 1.3.1 - use logic gates to create electronic circuits
- 1.3.1 - understand and define the functions of NOT, AND, OR, NAND, NOR and XOR (EOR) gates, including the binary output produced from all the possible binary inputs (all gates, except the NOT gate, will have 2 inputs only)
- 1.3.1 - draw truth tables and recognise a logic gate from its truth table
- 1.3.1 - recognise and use the following standard symbols used to represent logic gates

Introduction

Read chapter 3 of the textbook

Relays and Logic Gates - How to Make a Computer: Part I (6:30) -
https://www.youtube.com/watch?v=fB85NrUBBhQ

Discussion & learning items

This video introduced you to logic gates. This is the level of complexity from the transistor. We use multiple transistors to build logic gates. Multiple logic gates can then be used in clever patterns to create memory and perform calculations. Once we have the ability to store values in memory, and to be able to perform calculations on those values, we then have the basic building blocks of every computer.

There are 6 logic gates we will study in this course.

- NOT gate
- AND gate
- OR gate
- NAND gate
- NOR gate
- XOR gate

These gates are effectively switches, where the state of the output (whether it is on or off) is determined by the combination of the inputs and the rule of the gate.

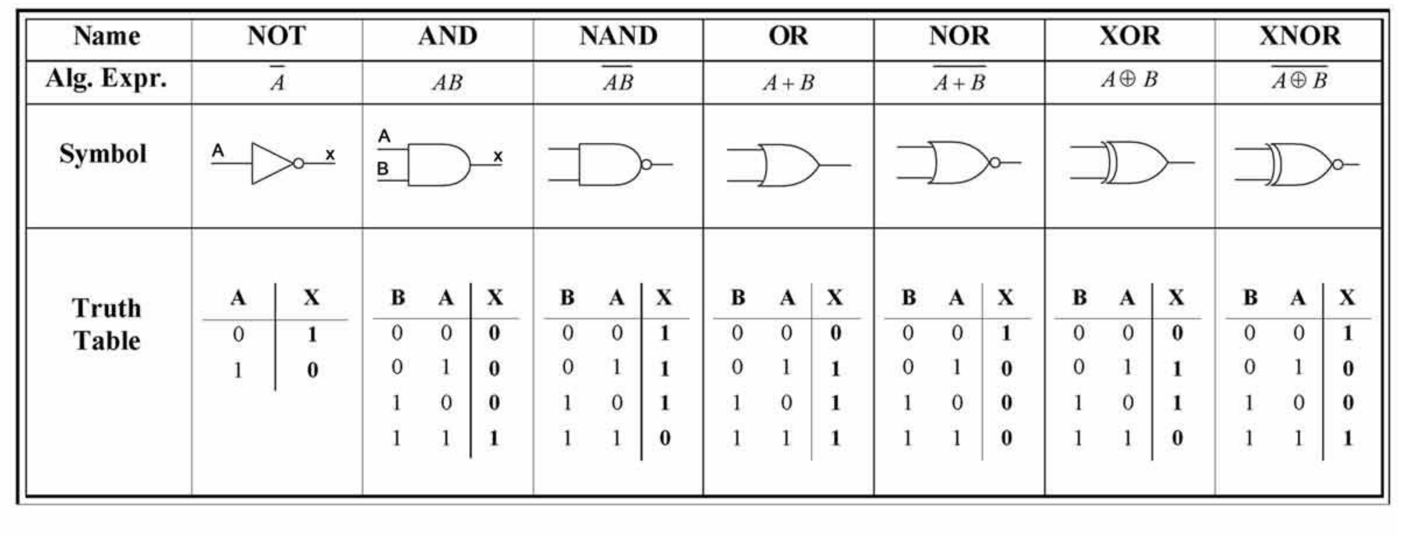
- The NOT gate has one input and one output. If the input is on, the output is off. If the input is off, the output will be on.
- The AND gate has two inputs and one output. If both inputs are on, the output will be on. In all other cases, the output is off.
- The OR gate has two inputs and one output. If either one of the inputs is on, the output will be on. If both inputs are off, the output will be off.
- The NAND gate is a concatenation of an AND gate where the output goes through a NOT gate. This means if both inputs are on, the output will be off. In all other cases, the output is on.
- The NOR gate is a concatenation of an OR gate where the output goes through a NOT gate. This means if either input is on, the output will be off. If both inputs are off, the output will be on.

- The XOR gate is known as the eXclusive OR gate. With this gate if one of the inputs are on, the output is on. However, if both inputs are on, the output remains off. Likewise if both inputs are off, the output remains off.

Clarification: While it is possible to have 3 input AND and OR gates, the iGCSE course is limited to gates of 2 inputs

We will use combinations of these logic gates to create logic circuits. To enable us to do this easily each gate has a symbol by which it can be represented in a diagram, and there are also a couple of notations available to represent them in the form of written equations.

The logic gate symbols are shown below and are also in page 26 of your text (the XNOR gate is not part of your course but it should be an intuitive extension to determine what it is...?)



Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A\oplus B$	$\overline{A\oplus B}$
Symbol	A x	A B x					

Truth Table	NOT		AND			NAND			OR			NOR			XOR			XNOR		
	A	X	B	A	X	B	A	X	B	A	X	B	A	X	B	A	X	B	A	X
	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1
	1	0	0	1	0	0	1	1	0	1	1	0	1	0	0	1	1	0	1	0
			1	0	0	1	0	1	1	0	1	1	0	0	1	0	1	1	0	0
			1	1	1	1	1	0	1	1	1	1	1	0	1	1	0	1	1	1

As you can also see, we can simplify our understanding of rules for each gate by using a table, known as a truth table, to document the circumstances in which a gate is on or off. Rather than using the "on" and "off" terminology, we use binary where 0 represents off and 1 represents on.

Exercises

Logic gates are combined together to form logic circuits. To gain an understanding of what an individual circuit may do, we create truth tables for them as well.

- Complete the walk through example, converting a logic diagram to a truth table, as a class (3.5.1 on page 29/30).
- Individually produce the truth tables for the logic diagram questions in activity 3.2 on page 32.
- Complete these additional practice questions: Logic equations to circuits

2. Logic equations to circuits

Learning objectives

- 1.3.1 - produce truth tables for given logic circuits,
- 1.3.1 - produce a logic circuit to solve a given problem or to implement a given written logic statement

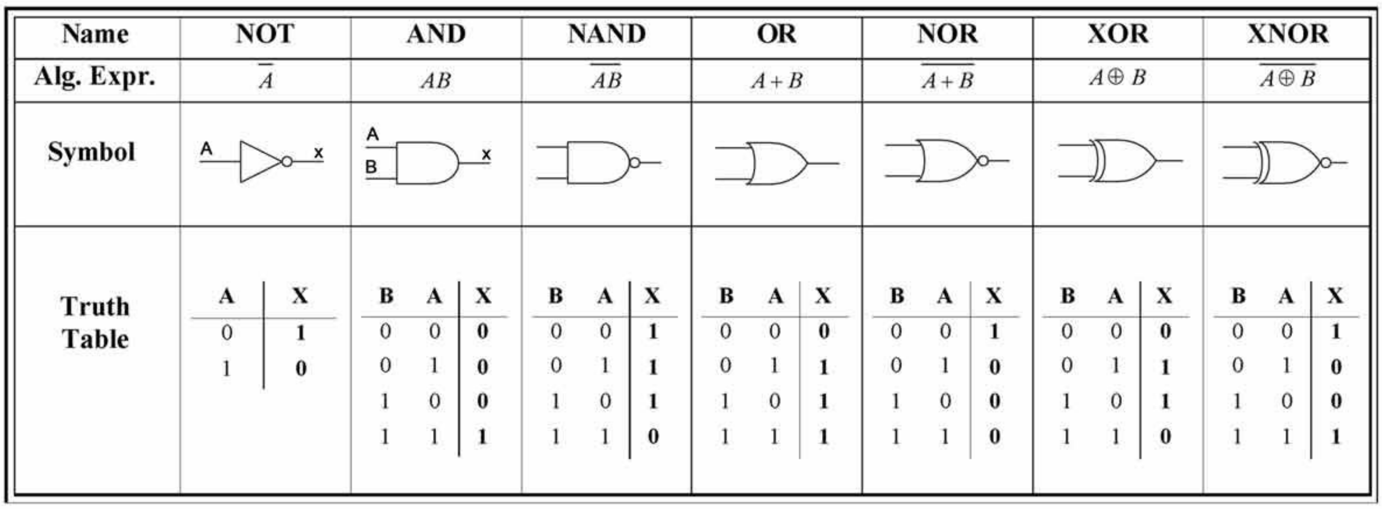
Introduction

- Chapter 3

Discussion & learning items

Any logic circuit can be expressed as a diagram or as an equation. You need to be able to convert one to the other.

If we review this table again, we'll see that there were arithmetic symbols provided for gate operation.



Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A\oplus B$	$\overline{A\oplus B}$
Symbol	A x	A B x					
Truth Table	A | X	B A | X	B A | X	B A | X	B A | X	B A | X	B A | X
	0 | 1	0 0 | 0	0 0 | 1	0 0 | 0	0 0 | 1	0 0 | 0	0 0 | 1
	1 | 0	0 1 | 0	0 1 | 1	0 1 | 1	0 1 | 0	0 1 | 1	0 1 | 0
		1 0 | 0	1 0 | 1	1 0 | 1	1 0 | 0	1 0 | 1	1 0 | 0
		1 1 | 1	1 1 | 0	1 1 | 1	1 1 | 0	1 1 | 0	1 1 | 1

This means instead of having to draw a circuit, we can write an equation that represents a circuit.

For instance:

- an AND gate with inputs A and B that outputs to X, we could write this as either X = A AND B, or just X = AB.
- an OR gate with inputs A and B that outputs to X, we could write this as either X = A OR B, or X = A + B.
- a NOT gate with input A and output X, that could be written as X = NOT A, or X = !A, or X = $\bar{A}$ (notice the line over the A).

Once we start writing equations, one small but crucial detail to know is the order of precedence. Intuitively from maths we may understand that brackets are resolved first, but what about other situations? For instance given X = NOT A AND B... is it (NOT A) AND B.... or is it NOT (A AND B)... ?

The order of precedence is 1st parenthesis, 2nd NOT, 3rd AND, 4th OR.

Exercises

Create circuit diagrams from these logic equations, then produce the truth tables.

1. Draw a logic circuit for $(A + B)C$
2. Draw a logic circuit for $A + BC + \overline{D}$
3. Draw a logic circuit for $AB + CD$
4. Draw a logic circuit for $\overline{(A + B)}(C + D)\overline{C}$

Check your answers at http://sandbox.mc.edu/~bennet/cs110/boolalg/gate.html

Now from logic equation to truth table without the circuit diagram.

1. Draw a truth table for $A + BC$
2. Draw a truth table for $A(B + D)$
3. Draw a truth table for $(A + B)(A + C)$
4. Draw a truth table for $\overline{W}(X + Y)$
5. Draw a truth table for $P\overline{T}(P + Z)$

Check your answers at http://sandbox.mc.edu/~bennet/cs110/boolalg/truthtab.html

3, 4. Solving written problems

Learning objectives

- 1.3.1 - produce truth tables for given logic circuits,
- 1.3.1 - produce a logic circuit to solve a given problem or to implement a given written logic statement

Introduction

- Chapter 3

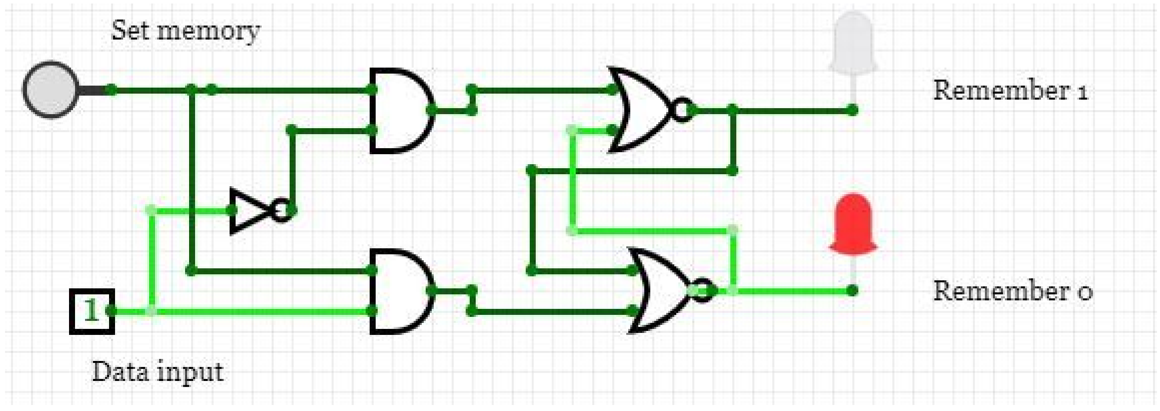
Discussion & learning items

You may be presented with a written scenario from which you need to discern the equation and logic circuit.

So we've been building these things called logic circuits. So what do they have to do with computers?

The internal operations of your CPU is reduced down to transistors that form logic gates that form logic circuits.

For an example: this is a circuit known as a d-latch. It will "store" one bit of "memory". Whatever value is on the data wire when the set wire is activated, will be "remembered" until the next time you activate the set wire.



Exercises

- Complete the walk through example as a class 3.5.3 on page 35.
- Individually complete the practice question in activity 3.4 on page 38.
- Complete the remaining activities in chapter 3.
- To demonstrate just how an entire computer system is built from nothing but logic gates, this website allows you to build a virtual computer beginning from nothing but a NAND gate.
  http://nandgame.com/
  Complete at least the first 6 levels, then go as far as you wish beyond that.

5. Binary numbers & converting to/from decimal

Learning objectives

- 1.1.1 - recognise the use of binary numbers in computer systems
- 1.1.1 - convert positive denary integers into binary and positive binary integers into denary (a maximum of 16 bits will be used)
- 1.1.1 - show understanding of the concept of a byte and how the byte is used to measure memory size
- 1.1.1 - use binary in computer registers for a given application (such as in robotics, digital instruments and counting systems)

Introduction

- Read 1.1, 1.2, 1.3, 1.4 of the textbook
- https://www.youtube.com/watch?v=NRKORzi5tnM

Discussion & learning items

This absence of electricity needs to be simplified for computer scientists to effectively scale it to the complexity of modern computers. For this reason we think of it as True and False which is then further simplified into 1 and 0.

This most simple form of data, that is a 1 or 0 is known as a bit. The number system based on using 0 and 1 is known as the binary number system.

Counting in binary

To count up in binary, works the same as counting in decimal except we only have two values for each place. For each column, we start from zero, count up until we have hit the highest possible value, in this case 1, and add a new significant figure.

Starting from zero, it would look like...

```
  0
  1
 10
 11
100
101
110
111
```

Questions:
- If I wanted to add binary 100 and binary 1000, what would it be?
- If I wanted to add binary 100 and binary 1100, what would it be?

Dealing with lots of bits at a time isn't always practical from our human perspective, so we create an abstraction to allow us to simplify the process. The first level of complexity we introduce is to group 8 bits together into a byte.

Number of bits	1	2	3	4
Possible values	0 1	00 01 10 11	000 001 010 011 100 101 110 111	0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
Total possibilities	2 2^1	4 2^2	8 2^3	16 2^4

A byte consists of 8 bits, how many possible values does a byte have?

BInary size units

Memory size	Number of bits	Equivalent denary value	Memory size	Number of bits	Equivalent denary value
Kilobyte (kB)	10^3	1,000	Kibibyte (KiB)	2^10	1,024
Megabyte (MB)	10^6	1,000,000	Mebibyte (MiB)	2^20	1,048,576
Gigabyte (GB)	10^9	1,000,000,000	Gibibyte (GiB)	2^30	1,073,741,824
Terabyte (TB)	10^12	1,000,000,000,000	Tebibyte (TiB)	2^40	1,099,511,627,776
Petabyte (PB)	10^15	1,000,000,000,000,000	Pebibyte (PiB)	2^50	1,125,899,906,842,624

Uses of binary numbers

Integers

```
for n in range(256):
    b = bin(n)
    print(f" Decimal { n :3}, in binary is { b :>10}")
```

There are two different ways that integers might be stored with binary numbers - signed and unsigned - referring to whether or not to have a positive/negative sign as part of the number.

For instance, if we were to use an 8bit binary number to store integers. Without storing the sign, we can store 2^8 possible numbers giving a range of 0 to 256. If we used 1bit to store positive/negative, that means our positive range is reduced to 2^7. We'd actually still end up with 2^8 possibilities, it's just that half of them would be negative numbers, so the range would be -128 to +127. When using a signed integer, the first bit is used to indicate sign and is set to 0 for positive, 1 for negative.

To see how it works, let's count down...

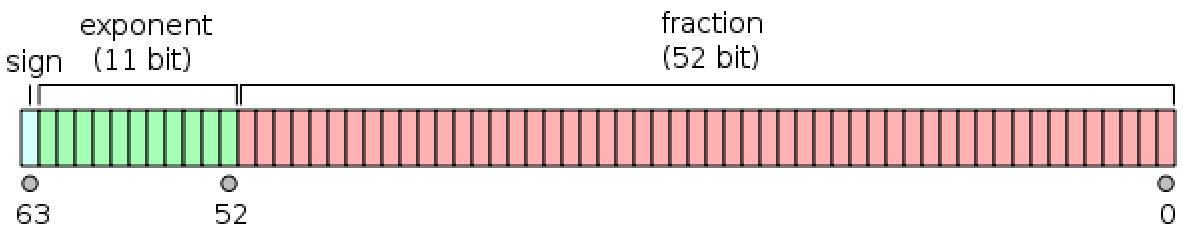
```
0000 0100 = 4
0000 0011 = 3
0000 0010 = 2
0000 0001 = 1
0000 0000 = 0
1111 1111 = -1
1111 1110 = -2
1111 1101 = -3
1111 1100 = -4
```

Having said all that - this course only requires you to understand positive integers.

Floating point numbers

How are floating point numbers stored internally? By storing two integers, one representing the significant figures, the other representing the exponent for the number. In this way we can store very large and very small numbers but with a limited degree of accuracy.

In the same way that we might think of the speed of light in decimal notation as being represented by the number 3 and 8.... to represent 3 x 10^8, floating point numbers use the same approach.



The 64 bit floating point number uses 1 bit for the sign (positive/negative), 8 bits for the exponent, and 55 bits for the significant number.

Everything is stored in binary rather than decimal. For the significant figures portion, this means the first bit represents 1/2, the second bit represents 1/4, the third will represent 1/8 and so forth. This poses some challenges for seemingly common numbers.

For instance, look at the following output from Python

```
>>> 0.3
0.3
>>> 0.1
0.1
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>>
```

The number 0.1 in binary is actually an infinitely recurring decimal, so when we add several together we are getting a rounding effect occurring. This is analogous to decimal 1/3rd being 0.3333333 recurring.

To illustrate: How would you get to the value of 0.1 from binary fractions?

1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625
0	0	0	1	1	0	0	1

1/2, 1/4 and 1/8 are too high. 1/16 + 1/32 gets you close, but adding 1/64 or 1/128 takes you over. The pattern is actually: `00011 0011 0011 0011 0011 recurring...`

It is important to remember that floating point numbers are useful, but they are not designed for high level precision after multiple mathematical operations. They were designed for scientific applications, not financial.

Try the floating point calculator... https://www.h-schmidt.net/FloatConverter/IEEE754.html

Characters & strings

To store the values of characters (which are combined together to create strings) a universally agreed upon lookup table was created. In its simplest form, this is known as the ASCII table (we'll discuss Unicode at a later point). You can see part of the ASCII table here, http://sticksandstones.kstrom.com/appen.html or a more complete version here, https://www.rapidtables.com/code/text/ascii-table.html

An example of bytes of ASCII being used to represent a string...

0100 1000	0110 0101	0110 1100	0110 1100	0110 1111
H	e	l	l	o

Try it for yourself here, https://www.rapidtables.com/convert/number/ascii-to-binary.html

Booleans

Since boolean variables are True or False, this only requires 1 bit to store.

That said, it will occupy at least 1 byte of memory as it is simply not efficient or practical for RAM to be managed on a per-bit basis.

Registers

A register is a collection of bits within the CPU with a designated purpose. They may be a temporary holding area for an instruction about to be executed, or a value that is about to be used. They are the very short term working memory of the CPU for the exact job it is doing at that moment.

You have probably heard the phrase 32 bit computing or 64 bit computing. This refers to, amongst other things, the size of the registers within the CPU. If a CPU has 32 bit registers, then it can cope with, at most 2^32 memory locations in RAM (about 4GB), and calculations on numbers of up to 4 billion. Modern computers are now 64 bit.

Another example of using registers is to control the outputs of a computer system. The textbook contains an example of a computer controlling motors through setting the bits within a register.

Read section 1.4, complete activity 1.3

Exercises

Lesson intro: Watch video Binary - How to Make a Computer: Part II (7:15)

- https://www.youtube.com/watch?v=NRKORzi5tnM

Lesson outline:

- Converting binary to decimal
- Converting decimal to binary

Practice questions:

- Activity 1.1, page 3, Convert binary to decimal
- Activity 1.2, page 3, convert decimal to binary

Some additional questions:

- Binary 0000 0100 is what in decimal?
- Binary 1100 1100 is what in decimal?
- Binary 0111 0001 is what in decimal?
- Binary 0010 1001 is what in decimal?
- Binary 0111 0101 is what in decimal?
- Binary 0111 0101 is what in decimal?
- Binary 1010 1010 0111 0101 is what in decimal?
- Binary 0011 1011 0000 0101 is what in decimal?
- Binary 1100 0111 1100 1100 is what in decimal?
- Binary 1001 0110 1010 1010 is what in decimal?
- Binary 1100 0011 1100 0011 is what in decimal?
- Decimal 98 is what in binary?
- Decimal 195 is what in binary?
- Decimal 156 is what in binary?
- Decimal 245 is what in binary?
- Decimal 116 is what in binary?
- Decimal 1160 is what in binary?
- Decimal 2235 is what in binary?
- Decimal 2424 is what in binary?
- Decimal 12235 is what in binary?
- Decimal 54321 is what in binary?

6,7. Hexadecimal numbers & converting to/from binary

Learning objectives

- 1.1.2 - represent positive numbers in hexadecimal notation
- 1.1.2 - show understanding of the reasons for choosing hexadecimal notation to represent numbers
- 1.1.2 - convert positive hexadecimal integers to and from binary (a maximum of 16 bit binary numbers will be required)
- 1.1.2 - convert positive hexadecimal integers to and from denary (a maximum of four hexadecimal digits will be required)
- 1.1.2 - represent numbers stored in registers and main memory as hexadecimal
- 1.1.2 - identify current uses of hexadecimal numbers in computing, such as defining colours in Hypertext Markup Language (HTML), Media Access Control (MAC) addresses, assembly languages and machine code,debugging

Introduction

- Read 1.5, 1.6 of the textbook

Discussion & learning items

Binary is great, it's the language our computers function in, but it's not exactly human friendly. They quickly become overwhelmingly long, especially now in the days of 64 bit computing.

For instance, this is the "Mac address" of the laptop I'm writing this on in binary...
`111001001011001100011000101111011111001110011111`.

If humans had to deal with these numbers regularly there would be too many errors.

Sure, we can convert between base-2 and base-10 numbers but it's not particularly convenient, and it disguises binary patterns. Due to the linkage with logic gates specific bits might have particular significance. For instance, you can't quickly tell just by looking at these numbers which bit they all have in common can you... 124, 68, 30?

The solution is to come up with another number system. One that is relatively human friendly and binary friendly at the same time. What computer scientists came up with is the Hexadecimal number system. Hexadecimal is base-16, meaning there are 16 different values per significant figure. 16 is useful for binary because it is 4 bits. This means a full byte can be represented with just 2 characters instead of 8.

Obviously our number system only has 10 symbols, but we now require 16, so what to do? We use letters.

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

The following is some simple Python code that will generate all the binary, decimal and hexadecimal values from 0 to 255.

```
for n in range(256):
    b = bin(n)
    h = hex(n)
    print(f" Decimal { n :3 }, in binary { b :>10 }, in hex { h :>4 }")
```

Common uses of hex

Because it is a convenient way of grouping 4 bits together, hexadecimal is used a lot in the computer industry to represent binary data.

One usage you will have seen before is hexadecimal colour codes. Modern computers run RGB (red-blue-green) pixel displays where each of the three colours can be set to a range from 0 (off) to 255 (fully on). So, every pixel requires 3 bytes of colour information. But rather than describing red as `111111110000000000000000`, we commonly see things like `ff0000` instead. This is hexadecimal at work.

Network addresses are another common place you will see Hexadecimal values being used. Earlier I said my computer Mac address was `111001001011001100011000101111011111001110011111` but you would normally see it written as `e4b318bdf39f` instead.

If you are curious to know your computer's Mac address...

- On Windows, press Windows key + X at the same time. Select the Command Prompt or PowerShell. In the terminal that opens, enter ipconfig /all.
- Or try this little bit of Python...

```
import uuid
print( hex(uuid.getnode()) )
```

Hexadecimal numbers are also commonly used in other places such as assembly programming, and when we are debugging output from a program. We could, for instance, have the computer "dump* the contents of memory to a file so we can inspect it and diagnose faulty behaviour. This dump would typically get written out into hexadecimal.

Converting

Converting between binary and hex is quite straightforward, as each group of 4 bits and be substituted with a hex number.

The only trick is to make sure you start with the least significant digits first, or put enough leading zeros in place you are working with a number of bits that is divisible by 4. For instance, `101100` should convert to `2C` rather than `B0`.

Exercises

- Textbook activity 1.4 - Convert from binary to hexadecimal
- Textbook activity 1.5 - Convert from hexadecimal to binary
- Textbook activity 1.6 - Convert from hexadecimal to denary
- Textbook activity 1.7 - Convert from denary to hexadecimal

8. Quiz