# Computer Systems 1
## Dr. John T. O'Donnell

*Joao Almeida-Domingues**

*University of Glasgow*

*January 7th, 2019 – May 24th, 2019*

## Contents

*2334590D@student.gla.ac.uk

# 1    Analogue and Digital Representation

## 1.1    *Computer Systems*

**1.1 definition.  Digital Circuits** Electronic systems which use very large numbers of only a few types of components, which when connected the right way create incredibly complex , useful behaviour

**1.2 definition.  Machine Language** Programming language executed directly by the computer hardware, which serves as the interface between low-level circuits and high-level software. Therefore, it is simple enough so that a digital circuit can be designed to execute it, yet also powerful enough that high-level languages can be converted into it

## 1.2    *Data Representation*

There are several digital data types which need somehow be represented in hardware. This is done by encoding electrical signals (voltage). It is by manipulating these voltages in the hardware that computations are able to be performed. There are 2 ways of achieving this:

1. **Analogue**

    **1.3 definition.**   The variables in the calculation are *analogue* to the physical signals being measured (e.g. $1 \iff 3v$)

    There are some advantages to this method, such as fast calculation of differential equations. Unfortunately, it also has significant drawbacks: limited precision; errors accumulate; difficult to represent non-numerical data

2. **Digital**

    **1.4 definition.**   As indicated by the root of the word, it makes use of digits. It is by counting that calculations can be performed.

    This addresses some of the drawbacks of analogue computing. For example, one can always add more digits when higher precision is required; Handles noise and errors better; Easy representation of most data types

**1.5 definition. Bit** Unit of information used in digital circuits. It represents either a 0 or a 1

**1.6 remark.**  Different circuits use different voltages, but the more simpler and reliable ones tend to use just two clearly distinct voltages to represent each value $(0, 1)$

**1.7 definition. Flip Flop** The simplest circuit with memory. Basic element of computer memory

**1.8 definition. Byte** 8 bits

Note that just like in any other positional number system, the number of possible representations grows exponentially with the number of digits added. In the binary system we only have $0, 1$. But by using basic knowledge of combinatorics, one observes that for every new position $n$ the number of possible orderings doubles. It follows, for example, that for an 8 bit binary number one can represent up to $2^8 = 256$ different values ( $0 \rightarrow 255$ ). In general, for a $k$ bit number:

$$2^k \text{ possible values. From 0 until } 2^k - 1$$

**1.9 notation.** By convention, 4 bits are separated by a space for readability

$$0000\ 0000$$

**1.10 definition. Word** For convenience, for larger numbers of bits, the term *word* is used. The specific number of bits which a word represent changes. For this class: $16, 32, 64 =$ short w., w., long w.

*Lecture 2*
*January $10^{th}$, 2019*

## 2 Number Systems

### 2.1 *Conversion*

Conversions between different bases are easily achieved, by keeping in mind the simple fact that each position is able to represent a minimum value, and a maximum value, every time that value is exceeded , we need another "slot" to reset the counting with the "symbols" (i.e the digits) available in that system. To make it clear, let's first look at the decimal system, as first learned in primary school. $19_{10}$ can be thought of $10 + 9$ , which in turn can be thought of 9 single units + 1 single "10" unit. Since 9 is the highest "unit symbol" available in the decimal number system we need another slot "the tenths". Transitioning from 9 to "10". Note how the "counter" was reset to 0 and we now start counting again from 1, but in the tenths and in the units slot. We don't need to add another slot until all digits, in all possible combinations have been used.

Similarly, binary/base 2, simply means that for every "slot" we have only two "symbols" at our disposal, 0 and 1. And instead of thinking of the slots as $10^{ths}, 100^{ths}, \ldots$ we think in terms of powers of 2.

| 128 | 64 | 32 | 16 | | 8 | 4 | 2 | 1 | |
|-----|-----|-----|-----|---|-----|-----|-----|-----|---|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
| 0 | 0 | 0 | 0 | | 0 | 1 | 1 | 1 | $= 4 + 2 + 1 = 7_{10}$ |

From decimal to binary the process is similar, but in reverse. If a primary school child was given a decimal table, and was asked to fill it with 150 , she would obviously not start by the units and start incrementing 1-by-1. She would notice that she'll need at least the 1 unit of "100" ($10^3$) and immediately put a 1 in the 3rd slot, and then notice that she'll need 5 "10s" and so on. This is trivial, and similarly trivial it is for base 2. Say you're given $74_{10}$. What is the highest slot you can immediately fill? Well, 128 it's too high, the one before it is $64 = 2^7$ (position 7). That works, but it still leaves us with 10 units to be represented. Repeating the process, we see that we'll need an $8 = 2^3$ (position 3) and a 2 (position 2) until all units have been distributed amongst the "slots". Giving us:

$$0100\ 1010$$

This process is less straightforward for other bases, simply because we're used to think in base10. But the method is similar for all. Another useful base is **base 16**, where we use the letters A - F to represent the numbers from $10 \rightarrow 15$. This has the obvious advantage of reducing the number of "symbols"/characters needed for representing a binary number. For example: an 8 bit number can be easily represented by just 2 characters, with plenty leftover, since $16^2 = 256$ possible values.

**2.1 notation.** For this class "$" is used to represent hexadecimal numbers

The simplest way to convert from Binary to Hex is simply to note that, we need 4 bits to represent 1 hexadecimal bit in its entirety (i.e. 0 - F) . Hence we split the number into 4 bit chunks and convert them into the corresponding hex character. Such that the first 4 bits correspond to the hex uni bit , the next 4 bits to the "$16^{ths}$" , etc.

$$\lfloor\ 0\ |\ 1\ |\ 0\ |\ 0\ |\ \ |\ 1\ |\ 0\ |\ 1\ |\ 0\ \rfloor$$
$$\underbrace{\qquad}_{4}\qquad\ \ \underbrace{\qquad}_{A}$$

The reverse is done in the same way as in the decimal above, but again working only in 4 bit chunks

**2.2 remark.** Note however that it is always more natural to filter through decimal instead of thinking "$2^4 - 1 = F$"

### 2.2 *Operations*

Multiplication and Addition are performed much like in the same way we perform operations in decimal, again keeping in mind the digits available to work within each system, every time we exceed the limit we must carry onto the next.

Subtraction can be seen as a special case of addition with negative numbers, i.e. $3 - 2 = 3 + (-2)$. Binary numbers cannot represent negative numbers however, so we use **2's complement**. A 2c number is negative if its leftmost digit is 1

**2.3 remark.** Note that this does not affect the number of possible values that can still be represented, but it reduces the possible positive number representations in half, since for every positive number there will be a corresponding negative.

The general conversion method is straight forward:

1. Invert the bits

2. Add 1

It can be useful to think of the rightmost digit of an $n$ bit 2c number as always representing the sum of the highest possible negative number $(-2^{(n-1)})$ with an $n-1$ bits positive number. The $n^{th}$ bit is just the sign bit

**2.4 remark.** Note that, by adding all the positives we'll get 0, in which case the sign bit is turned off. We can think of 0 as belonging to the positives in this case, since it has the sign bit off. But when not considering it, even though we can represent $2^{(n-1)}$ negative numbers we can only represent $2^{(n-1)} - 1$. Since the $"n^{th}"$ would in effect be 0.

**2.5 example.**
$$\begin{array}{c|c|c|c} 1 & 0 & 1 & 0 \\ -8 & +0 & +2 & +0 \end{array} = -6$$

Since it is a 2c number we know we can represent up to $2^3 = 8$ different values in each "direction" (considering 0 a positive number). So we have $-8$ and then we need to find out the 3bit binary. $(0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 = 2)$. And then add them

It is clear from the example above how subtraction and division can then be performed just like addition

## 3   Logic Gates
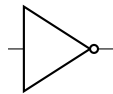
**3.1 definition. Logic Gate** A physical, basic component, which takes 1+ input(s) and performs a boolean function (i.e the result is either T/F)

1. **Inverter**

   **3.2 definition.** Takes an input and returns its opposite

   **3.3 notation.** inv

*alternatively $\bar{p}$*
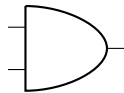
| $p$ | $\neg p^*$ |
|-----|-----|
| 1   | 0   |
| 0   | 1   |

2. **2-Input AND**

   **3.4 definition.** Returns true *iff* the two inputs are true

*the 2 used in the gates' names just means that 2 inputs are passed*

   **3.5 notation.** and2

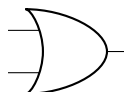| $p$ | $q$ | $p \wedge q$ |
|-----|-----|-----|
| 1   | 0   | 0   |
| 1   | 1   | 1   |
| 0   | 0   | 0   |
| 0   | 1   | 0   |

   **3.6 remark.** Note that for a $n$ input truth-table, there are $2^n$ possible input combos

   **3.7 remark.** An easy way to construct truth tables is to start half(1)-half(0), then $\frac{1}{4}$, $\frac{1}{8}$, $\cdots$. So for a 3 input table, one first put 4 1s, 4 0s. Then, half of that, i.e. 2 1s 2 0s 2$\times$, and finally 1, 0 , 8$\times$

3. **2-input OR**

   **3.8 definition.** Returns true if any one of the two inputs are true, or if both are true
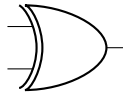
   **3.9 notation.** or2

| $p$ | $q$ | $p \vee q$ |
|-----|-----|-----|
| 1   | 0   | 1   |
| 1   | 1   | 1   |
| 0   | 0   | 0   |
| 0   | 1   | 1   |

4. **2-input XOR**

   **3.10 definition.** Returns true *iff* one of the two inputs is true

   **3.11 notation.** xor2

*⊕

| $p$ | $q$ | $p \underline{\vee} * q$ |
|---|---|---|
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |

### 3.1 *Combinational Circuits*

**3.12 definition. Combinational Circuits** Their output depends only on their current input. There's no feedback loop, or "memory" state

For complex circuits it is hard to keep track of every input and output by just connecting single logic gates. It is useful to create *black box circuits* to abstract some of the complexity out. These components, regardless of what logic gates are used to built them*, always perform the same operation (e.g. choosing between two inputs).
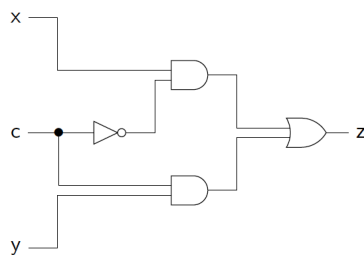
*hence the name black box

1. **Multiplexer**

   **3.13 notation.** mux1

   **3.14 definition.** Hardware equivalent of the *"if-then-else"* statement. Chooses between 2 values $(x, y)$ given a third $(c)$. If $c$ $y$, else $x$

   **3.15 remark.** Every decision a computer makes comes down to a multiplexer

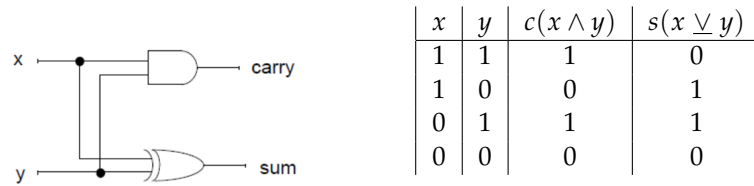| $c$ | $x$ | $y$ | $z$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |

2. **Half-Adder**

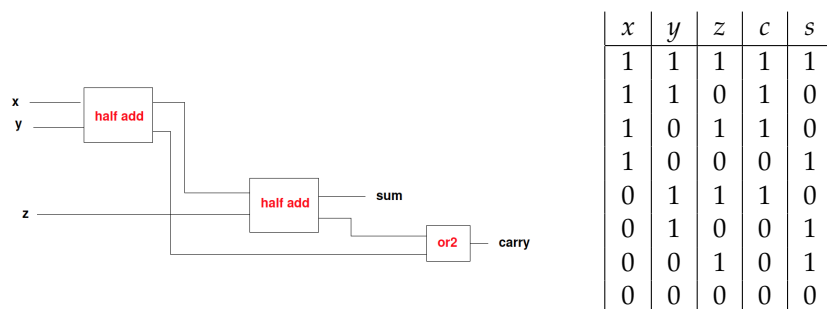**3.16 definition.** Adds two bits using a 2-bit (carry,sum) representation.

Note that there's only a carry if both inputs are 1, hence we represent this by an and2 gate, while the sum can be represented by an xor2

| $x$ | $y$ | $c(x \wedge y)$ | $s(x \veebar y)$ |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |

3. **Full-Adder**

**3.17 definition.** Adds three bits (inputs + carry) using a 2-bit (carry,sum) representation.

Note that there's only a carry when 2+ inputs are 1, and there's only a sum iff there are an odd number of inputs equal to 1

| $x$ | $y$ | $z$ | $c$ | $s$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 |

# 4   Boolean Algebra & Arithmetics

### 4.1   *Laws*

**4.1 definition.  Idempotence** operation that can be applied several times without changing the result

$$x \vee x = x \qquad x \wedge x = x$$

**4.2 definition.  Commutative** operations can be performed in any order

$$x \vee y = y \vee x \qquad x \wedge y = y \wedge x$$

**4.3 definition.  Associative** the order in which the operations are performed does not matter as long as the sequence of the operands is not changed

$$x \vee (y \vee z) = (x \vee y) \vee z$$
$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

*Proof.*

| x | y | z | y ∨ z | x ∨ y | x ∨ (y ∨ z) | (x ∨ y) ∨ z |
|---|---|---|-------|-------|-------------|-------------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

QED

**4.4 definition.  Distributive & Absorption**

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \wedge z)$$

$$x \wedge (x \vee y) = x$$
$$x \vee (x \wedge y) = x$$

**4.5 remark.**  Useful for prove of correctness

**4.6 definition.** **4-bit Ripple Carry Adder** Uses 4 full adders to add two 4-bit numbers
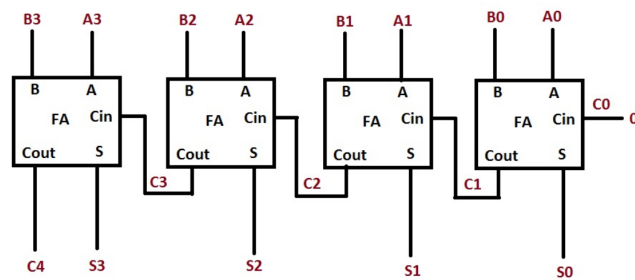


**4.7 remark.** Note how the carry from each previous operation is passed onto the next, just like how one does addition manually

**4.8 remark.** Subtraction can be done much in the some way, by first applying 2's c to one of the inputs

## REFERENCES

**wikipedia**

URL: https://en.wikipedia.org/wiki/Wikipedia.

**Basic Electronics Tutorials and Revision**        **basic˙electronics**

*Basic Electronics Tutorials and Revision.* URL: https://www.electronics-tutorials.ws/.

**Petzold: Code: the hidden language of computer hardware and software**

**petzold˙2001**

Charles Petzold. *Code: the hidden language of computer hardware and software.* Microsoft Press, 2001.

**Velleman: How to prove it: a structured approach**        **velleman˙2009**

Daniel J. Velleman. *How to prove it: a structured approach.* Cambridge University Press, 2009.