Introduction to Computing

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2014 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. All rights reserved.

Computer Architecture: The von-Neumann architecture

An abstraction of sequential (sometimes called serial) computing has been available since the mid 40s in the form of the **von-Neumann model/architecture** of computation.

Under this architecture a central processing unit, also known as the CPU, is responsible for computations. The program that is being executed and its relevant data both reside in a separate memory (usually called **main memory**). Thus memory stores both program and data, the CPU retrieves from memory at every cycle either program (in the form of an instruction) or data or writes back into memory data that were computed at the CPU by one of its units.

Thus von-Neumann introduced the concept that program and data are held together in the same memory space and the CPU interacts with the memory for all operations involving program or data.

We provide some notation that will disambiguate several concepts that are used interchangeably in the literature. A CPU (i.e. central processing unit) used to be a collection of chips (also known as microchips, IC, or integrated chips). However since the mid 70s most if not all of the functions of a CPU were able to be housed into a single microchip. That microchip is known as the microprocessor.

Although in its barest form the von-Neumann architecture is quite intuitive and nowadays everybody uses its abstraction, this was not the case in the early days of computing (early 1940s to mid 1950s).

An alternative architecture, the so called **Harvard model/architecture** of computation as influenced by (or implemented into) the Harvard Mark IV computer for USAF (1952) was also prevalent. In the Harvard architecture, programs and data are stored separately and the CPU maintains distinct access paths to obtain pieces of a program or its associated data. In that model, a concurrent access of a piece of a program and its associated data is possible. This way an instruction and its relevant data can reach the CPU faster.

Harvard architectures with their separation of program from data proved complex, expensive and less popular than the von-Neumann oriented architectures in the early days of computing.

Note. We use the terms `model` and `architecture` interchangeably. We prefer to use `architecture` when we talk about computer units and the more general `model` otherwise.

(c) Copyright A. Gerbessiotis. All rights reserved.

Yet every time one faces with certain limitations of the von-Neumann model, a Harvard-based approach is used to overcome or ameliorate such limitations. The concepts of pipelining, instruction and data-level caches can be considered Harvard-architecture intrusions into von-Neumann models. Most modern microprocessor architectures are using them. We could call such architectures hybrid Harvard but the abstraction that is being made available to the end user and hides all low-level details is that of the von-Neumann model.

In most modern designs, a modified Harvard (or hybrid Harvard architecture) is available for specific purposes. Since the mid to late 80s microprocessor speeds have been much faster than memory speeds. A 25Mhz microprocessor of 1990 required a memory of $1000/25 = 40$ nanoseconds (ns) to keep up with computations. An Mhz is a megahertz (mega is one million), that is 1,000,000 Hz (Hertz), and 1 Mhz is roughly 1,000,000 of clock ticks (or operations) per second, and a gigahertz (Ghz) is 1000Mhz. A nanosecond is a billionth of a second. Light and electronic signals can travel approximately one foot (or one third of a meter) in a nanosecond. Yet most memories of that time operated at 80-100ns. More recently microprocessors operate routinely at the 1-3Ghz (or 1000-3000Mhz) level yet memory is still quite slow at 30-50ns.

Thus to bridge the gap between a fast CPU (microprocessor) and a slow memory some new tricks need to be used (or invented). Almost every microprocessor has available a **cache** which is a very fast, very expensive, and consequently very small-size auxiliary memory. The intent of making available a cache is that a CPU uses it (the cache) most of the time rather than the (slower) main memory. The CPU speculatively decides which instructions or data will use in the next few cycles repeatedly, it transfers them once (slowly) from main memory into the cache, and then it uses the cache to retrieve and modify them (fast access/rewrites). Speculating about the behavior of a program or a user's intentions is one thing. Implementing an architecture that considers it is another. The end result is that modern architectures accommodate multiple cache memories. They are labeled level-1 (the closest to the CPU, the fastest, and the smallest in size), level-2 (intermediate) and more recently, level-3.

Among them, level-1 cache might be the only one with separate memory banks (modules) for storing instructions (programs) and data. This thus becomes a Harvard-based architectural intrusion into an otherwise von-Neumann architecture. The end-user still views such an architecture as a fundamentally von-Neumann model abstraction. Deep down in hardware a hybrid Harvard architecture or a modified von-Neumann architecture has been realized.

(c) Copyright A. Gerbessiotis. All rights reserved.

A level-1 cache is traditionally on-die within the CPU core. It operates at the speed of the CPU core. Level-1 caches are traditionally Harvard-hybrid architectures. There is an instruction (i.e. program) cache, and a data-cache separately available. Its size is kept small due to the complexity of the design and intended purpose. 8-16kiloBytes (more on what a kiloByte is, later) is the minimum, and 64kB-256kB a maximum size per cache (instruction or data).

Since roughly the early 90s several microprocessors have become available utilizing secondary level-2 caches. In the early years those level-2 caches were available on the motherboard or on a chip next to the CPU core (the microprocessor core along with the level-2 cache memory were sometimes referred to as the microprocessor slot). Several more recent microprocessors have level-2 caches on-die as well. A typical size of a level-2 cache is several Megabytes (noting the capitalization, a Megabyte is one more way to denote memory size).

Level-3 caches are not unheard of nowadays in multiple-core systems/architectures. A **multiple-core** system (also known as **multi-core**, or **many-core**, with dual-core or quad-core referring to systems with specifically 2 or 4 cores) is one in which multiple-CPU cores are available on the same die, and can perform computations independently of each other as if the system contained multiple CPUs.

All cores share however the same main memory. Each core might have its own level-1 caches. However level-2 caches are shared if not by all, usually by several cores. In architectures that utilize level-3 caches, these caches are shared by all cores. The main memory is naturally shared by all cores as well.

Note. A `kB` might mean kiloByte. Note that the `k` in kilo is lower case and the `B` for byte is upper case. A `kilo` traditionally means 1000 units. However 1kB might mean 1000 bytes or 1024 bytes. More about bits and bytes and kilobytes and kilobits later.

Note. We will formally introduce `bits, bytes` later in this subject. For now a `bit` is the elementary piece of digital information, and a byte is a bigger bite of memory. For all practical purposes, nowadays **one byte is equal to eight bits**.

(c) Copyright A. Gerbessiotis. All rights reserved.

In computing and in electronics, digital information appears in one of two states, on and off, or high and low, or as we usually write, 0 and 1. Thus number two (2) becomes ubiquitous. Logarithms and exponentials that utilize 2 as a base become part of thinking digitally.

1.1 Exponentials. All exponentials in computing are two base 2. Thus 2^x is 2 raised to the x-th power or simply 2 raised to x. The end effect is to multiply 2 by itself a number of x times. Several times one might write $2 * *x$ or $2\hat{}x$ for 2^x ($\hat{}$ is the hat/caret symbol usually co-located with the numeric-6 keyboard key). All three notations indicate the same thing i.e. $2^x = 2 * *x = 2\hat{}x$. If you are using a programming language and can't figure out how to raise 2 to the x-th power as in 2^x, then the alternative approaches with the stars or the caret (hat) might work! Experiment.

1.1.1 (Definition.) $2^x = 2 * *x = 2\hat{}x$. (Product of x 2s.)

1.2 Properties of exponentials.

1.2.1 (Multiplication.) $2^x \cdot 2^y = 2^x\ 2^y = 2^{x+y}$. (Dot $\cdot$ optional.)

1.2.2 (Division.) $2^x/2^y = 2^{x-y}$. (The symbol / is the slash symbol)

1.2.3 (Exponentiation.) $(2^x)^y = 2^{x \cdot y}$.

1.2.4 (Backslash Division.) $2^x \backslash 2^y = 2^y/2^x = 2^{y-x}$. (The symbol \ is the backslash symbol)

Examples.
$2^1 = 2$; $2^8 = 256$; $2^{10} = 1024$. (The kilo in a kilobyte is 1024 or 1000.)

How much is 2^{20}? Well $20 = 2 \cdot 10$ thus $2^{20} = 2^{2 \cdot 10} = (2^{10})^2$ and by (2.3) and the previous examples, we have $2^{20} = (2^{10})^2 = (1024)^2 = 1048576$. (The Mega in a Megabyte can sometimes refer to 1048576 or sometimes confusingly to just 1000000.)

How much is 2^{30}? Well $2^{30} = 2^{20} \cdot 2^{10}$ by Property (2.1) or equal to $(2^{10})^3$ by (2.3). Either of them gives $1024 \times 1024 \times 1024$ which is very close to $1000 \times 1000 \times 1000$, which is a billion. Thus either 2^{30} or 1,000,000,000 can serve as the definition of what we call Gigabyte.

I hope that you had noticed that a kilobyte is written with a `k` (lower case) as in a kilogram whereas a Megabyte or Gigabyte are written with an upper case `M` or `G`. This is because lower case `m,g` letters refer to subdivisions of a unit, not multiples (and more confusingly, a milligram is one thousandth of a gram). More on this later.

(c) Copyright A. Gerbessiotis. All rights reserved.

1.3 Logarithms. The logarithm base two of a number b is denoted by $\lg b$ or $\log_2 b$. In the latter case log indicates a generic logarithm and thus the base 2 must be explicitly written as a subscript to log. If one writes $\log b$, the writing will be ambiguous. Mathematicians or engineers might assume that it is to the base 10, i.e. $\log_{10} b$ or they can confuse it with $\ln b$. One can also write $\ln b$ to denote $\log_e b$, where e is the base of the Neperian logarithms (i.e. of $\ln b$ type logarithms), where $e = 2.7172\ldots$.

In computing the writing lg **IMPLIES that the logarithm is to base 2, and thus IT IS NOT a misspelling of log in which an oh! is missing.**

1.3.1 (Definition.) $a = \lg b$ if and only if $b = 2^a$, (i.e. $\lg b$ is the power we need to raise two to to get b, and this power is a).

1.4. Properties of logarithms.

1.4.1 (Multiplication.) $\lg(x \cdot y) = \lg x + \lg y$.

1.4.2 (Division.) $\lg(x/y) = \lg x - \lg y$.

1.4.3 (Exponentiation.) $\lg(x^y) = y \cdot \lg x$.

1.4.4 (Change of bases.) $a^{\lg b} = b^{\lg a}$.

If $a_1 = \lg b_1$ and $a_2 = \lg b_2$, then $2^{a_1} = b_1$ and $2^{a_2} = b_2$.

Let $b_3 = b_1 \cdot b_2 = 2^{a_1} \cdot 2^{a_2} = 2^{a_1+a_2} = 2^{a_3}$, where $a_3 = a_1 + a_2$. Thus, since $b_3 = 2^{a_3}$ we have that $a_3 = \lg b_3$.

$\lg(b_1 \cdot b_2) = \lg b_3 = a_3 = a_1 + a_2 = \lg b_1 + \lg b_2$. Thus the log of the product of $b_1 \cdot b_2$ (the product is b_3) is the sum of the logs of b_1 and b_2.

Examples.

How much is $\lg 2$? Since $2^1 = 2$, we have $\lg 2 = 1$. How much is $\lg 16$? Since $2^4 = 16$, $\lg 16 = 4$. How much is $\lg 1000$ or better $\lg 1024$? Since $1024 = 256 \cdot 4 = 16 \cdot 16 \cdot 4 = 2^4 \cdot 2^4 \cdot 2 \cdot 2$, we have that $\lg 1024 = \lg 2^4 + \lg 2^4 + \lg 2 + \lg 2 = 4+4+1+1 = 10$.

How much is $\lg 256$? Since $256 = 16 \cdot 16$ by 4.1 we have $\lg 256 = \lg(16 \cdot 16) = \lg 16 + \lg 16 = 4 + 4 = 8$. We could also have used the Exponentiation rule with $x = 16$ and $y = 2$. Or just note that $2^8 = 256$.

How many bits in (ie. to represent) a million (in binary)? It suffices to find $\lg 1000000$. Since 2^{20} is approximately $1,000,000$ or more precisely $1,048,576 = 1024 \cdot 1024$ we could say 20-21 bits in a million.

(c) Copyright A. Gerbessiotis. All rights reserved.

1.5 Relational operators. The symbols $<, >, \leq, \geq, \neq$ are called relational operators and used to compare two same-type objects (eg. numbers) the obvious way. They are the less-than, greater-than, less-than-or-equal-to, greater-than-or-equal-to, and not-equal-to symbols respectively.

1.6 $\forall, \exists, \in$. The symbol $\forall$ is the for-all operator/symbol and it reads `for-all`. $\exists$ is the existential operator/symbol and it reads `exists, or there exists`. And $\in$ is the `belongs-to` operator (to indicate than an element belongs to a set of elements).

(c) Copyright A. Gerbessiotis. All rights reserved.

I hope that you noticed that a kilobyte is written with a `k` (lower case) whereas a Megabyte or Gigabyte are written with an upper case `M` or `G`. According to the SI (International System of Units), a `k, M, G` (note the capitalization or lack of) refer to multiples of 1000, $1,000,000$ and $1,000,000,000$ of a unit. On the other hand, an `m` refers to a submultiple $1/1000$ of a unit. A `g` is just a gram (a mass unit), i.e. a thousandth of a kilogram (`kg`).

According to the SI system, if one wants to say unambiguously refer to 1024 or $1024 \times 1024 \times 1024$ bytes, one needs to say 1KiB or 1GiB respectively with KiB pronounced as `kibi-bee` and GiB `gibi-bee` for `kilobyte-binary` and `gigabyte-binary` respectively. Note also that for uniformity `KiB` uses an upper-case `K`. Then 1MiB is a `mebi-bee`, for a megabyte-binary.

1.7 Metric system notation for binary multiples

```
  SI (metric system)
Symbol Factor       Name    Symbol   Factor           Name
 k   = 1000         kilo        Ki = 1024            kibi  kilobinary
 M   = 1000 000     mega        Mi = 1024x1024       mebi  megabinary
 G   = 1000 000 000 giga        Gi = 1024x1024x1024  gibi  gigabinary
 T   = 10**12       tera        Ti = 1024**4         tebi  terabinary

 bit for bit never a b.   B for bytes

Thus  1MiB=             is 1048576B  or 1048576bytes
                        but never 1048576b (this latter is nonsense)

      1b                is nonsense
      1bit              is 1 bit
      10bit or 10 bits  is 10bits

      1B                is 1byte   (normally 8 bits)
     10B                is 10bytes
```

1.8 Colloquial use.

1.8.1 The terms `kB, MB, GB` with a capital B but lower case k or capitcal case M or capital case G and more recently capital case T or P (for terabyte and petabyte) have context specific confusing meanings.

1.8.2 When they describe main memory they denote capacity in terms of kilobyte binary, megabyte binary, and gigabyte binary. Thus.

```
Main Memory (DRAM/RAM/cache) : kilobyte/megabyte/gigabyte binary meaning for kB,MB,GB
1kB = 1024B,
1MB = 2**20B
1GB = 2**30B
```

1.8.3 When they describe the capacity of a disk drive (e.g. in the manufacturer's warranty) they denote capacity in terms of multiples of 10.

```
Bytes of Disk Drive capacity (manufacturer's specifications)
 1MB = 1,000,000B
 1GB = 1,000,000,000B
 1TB = 1,000,000,000,000B
```

1.8.4 It is thus imperative that you use the SI notation for 1KiB, 1MiB, 1GiB, 1TiB or ambiguity might cause confusion.

(c) Copyright A. Gerbessiotis. All rights reserved.

1.9 Bits and bytes!. (The capitalization is English grammar imposed and intended as an unintentional joke!)

1.9.1 Bit. A `bit` is the minimal amount of digital information. The term `bit` originates from `BInary digiT` to indicate that a bit can exist in one of two states: `1` or `0`. Sometimes those two states are indicated as `High, Low`, `Up, Down` , `True, False` , or `T, F`. A collection of 8 bits is called a `byte` (at least in this course).

The memory of a computer or more formally, the main memory, stores information in the form of bytes, not bits. Thus if one needs to store 2 bits of information, one must utilize one full byte.

1.9.2 Byte. Thus a `byte` is the smallest amount of data a computer could `bite` out of its memory (if you are interested in the etymology of the word byte). Memory size is usually expressed in bytes or larger aggregates. For example we might say that a memory is of size 10 bytes, rather than 80 bits.

In the not so distant past a byte used to be 4-bits, and sometimes 6-bits as well (for example CDC computers used to have 60-bit words that could be viewed as ten 6-bit bytes). For that reason the term `octet` has been used to unequivocally define an 8-bit quantity such as the one we usually refer to as a byte. Sometimes the word byte was used to refer to a `word`, a traditionally larger aggregation of bits than a byte. If this discussion is confusing, you may feel free to ignore it as long as you remember that for us in this course (and for you later on at NJIT) a byte is 8 bits (unless otherwise explicitly re-defined).

A `word` is (usually) defined to be 32-bits, i.e. it is 4B long.

The main memory of a computer is a linear storage fast random access space. Thus if we represent the memory with `MEM`, the first byte is `MEM[0]` and the second byte in memory is `MEM[1]`. Thus the i-th byte of memory will be `MEM[i-1]`. This is by the linearity of memory. Random-access means that the time it takes to retrieve one or the other byte is identical and does not depend on whether we are retrieving the first or the third byte of memory for example.

1.9 Bit and Byte sizes.

```
1 bit   =  2 states (0 or 1)       1 byte = 8 bits  (nowadays)
1 short =  1 wide-byte = 16 bits   1 word = 4B = 32 bits (nowadays)
1 int   = 4B or 8B (32-bit or 64-bits)      (integer number)
1 float  = 32-bits                          (real number)
1 double=  64-bits                          (real number)
1 kilobyte  is ambiguous ; however 1KiB =  1024 bytes
1 kB for a kilobyte might mean 1024 bytes
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Radix and base representation of (natural) integers

When we write number `13`, implicit in this writing, is that the base used to represent it is base-10 that uses digits `0-9` to represent all **natural numbers**.

A natural number is a positive integer as in $N = N^* = \{1, 2, 3, \ldots\}$ or sometimes a non-negative integer as in $N_0 = \{0, 1, 2, \ldots\}$. The set of natural numbers is N. If we want to unambiguously exclude 0 we use the N^* notation with the $*$ having the meaning that "zero is not included"; we unambiguously include zero by using the set notation N_0.

For **integer numbers** we might use one or two signs `+,-` for noting positive and negative integer numbers. The set of all integer numbers is Z. The set of non-negative integer numbers is Z_+ (that includes the zero), of non-positive is Z_-, the set of non-zero integer numbers is Z_*, and Z_+^* and Z_-^* are defined accordingly as the set of positive and negative integers.

The set of **real numbers** is R (with variations for positive, negative, etc defined similarly to Z) and we denote them by using additionally the decimal sign `.`

In computing, we use base-2 representation of numbers that are called binary numbers. So when we write 10, most people read it as a ten, but in computing or in base-2 this is integer two. To disambiguate such writing we write the base as a subscript to the right of the number. A more formal way to say base-10 is to say radix-10, where **radix** is another name (alias) for **base**. Thus 10_{10} is our ten, but 10_2 is the binary (i.e. base-2/radix-2) notation for the decimal integer (i.e. base-10/radix-10) two.

Note. When we say base-10 or radix-10 the base or radix indicated is ALWAYS base-ten.

(c) Copyright A. Gerbessiotis. All rights reserved.

1.10 Base-10 or radix-10 natural number representation.
When we use the symbols `0-9` to write $\texttt{123456}_{10}$ or simply `123456` we can convert this base-10 representation into a natural number by the simple (implicit conversion) that identifies, the units (right-most digit), the tens (second right-most), the hundreds (third right-most), the thousands etc, in the writing.

The **least-significant digit** (lsd) or right-most digit is 6 and is the number of units, and the **most-significant digit** (msd) is 1 and is the number of hundred-thousands.

Of course 123456 and 000123456 and 0123456 represent the same number. The second representation has three `leading` zeroes and the third one-leading zero. One should avoid this writing; yet the `most-significant digit` in all three is still an `1` although the **left-most** digit in the former case is an `1` and in the other two case is a `0`.

10^5		10^4		10^3		10^2		10^1		10^0	
$\cdot$	$+$	$\cdot$	$+$	$\cdot$	$+$	$\cdot$	$+$	$\cdot$	$+$	$\cdot$	
1		2		3		4		5		6	$=$
$1 \cdot 10^5$	$+$	$2 \cdot 10^4$	$+$	$3 \cdot 10^3$	$+$	$4 \cdot 10^2$	$+$	$5 \cdot 10^1$	$+$	$6 \cdot 10^0$	$=$
$100,000$	$+$	$20,000$	$+$	$3,000$	$+$	400	$+$	50	$+$	6	$= 123,456$

Note. In several programming languages a number starting with a zero such as 010, is an indication that that the number is written in octal (i.e. base-8). Thus 010 in such a case is 10_8 which is 8_{10} or 1000_2.

(c) Copyright A. Gerbessiotis. All rights reserved.

Base-2 (binary) natural integers

Binary (natural) numbers behave similarly to decimal (natural) numbers. Base-2 addition or subtraction, and similarly multiplication or division follow the same pattern as the elementary school version of the operations on natural numbers.

The only change is when a carry-bit is generated in addition or borrowed in subtraction. For example a carry-bit is generated if the sum of the two digits is 2 or more (i.e. 3) as opposed to a sum of 10 or more in decimal-based operations. Note that for decimal numbers a 9 plus 9 addition with a prior carry one, generates a sum of 19, i.e. a digit 9 and forward carry of 1. For binary numbers a digit 1 and a digit 1 with prior carry digit 1 can generates a sum of 3, i.e. a digit 1 and forward carry 1. Thus binary addition is simpler that decimal addition.

1.10.1 Base-2 to Base-10 conversion of natural numbers.

If we are given a binary number in base-2 we can similarly convert it in base-10, by using instead of 10, the designated base 2 in the calculations involved.

For example binary natural number (from now on, simply binary number) $\texttt{101011}_2$ also written as `101011` if it is unambiguously binary, represents the decimal integer `43`.

Instead of using units, tens, hundreds etc, which are powers of ten as in $10^0,\ 10^1,\ 10^2,$ in attempting to read the bit-sequence we use powers of two $2^0,\ 2^1,\ 2^2, \ldots$ with the smallest power of two (units) corresponding to the least-significant digit (also known as the **least-significant bit**) and progressing towards the most-significant digits (also know as the **most-significant bit**)

2^5		2^4		2^3		2^2		2^1		2^0	
$\cdot$	$+$	$\cdot$	$+$	$\cdot$	$+$	$\cdot$	$+$	$\cdot$	$+$	$\cdot$	
1		0		1		0		1		1	$=$
$1 \cdot 2^5$	$+$	$0 \cdot 2^4$	$+$	$1 \cdot 2^3$	$+$	$0 \cdot 2^2$	$+$	$1 \cdot 2^1$	$+$	$1 \cdot 2^0$	$=$
32	$+$	0	$+$	8	$+$	0	$+$	2	$+$	1	$= 43_{10} = 43$

This method works as long as we know how many digits the binary number has. We need to generate all power of two starting with 2^0 all the way to 2^l, where l is the number of bits in the number minus one. For our example, the number of bits is 6, and therefore $l = 6 - 1 = 5$, and therefore the largest power generated is 2^5.

Example A Base-10 to Base-2 conversion for powers of two is easy. $2^3 = 8_{10}$ is just `1000`, i.e. an one followed by three-zeroes. Thus the exponent determines the number of zeroes following the most-significant bit that is set to one. Similarly $2^7 = 128_{10}$ is `10000000`.

(c) Copyright A. Gerbessiotis. All rights reserved.

1.10.2 Base-2 to Base-10 conversion of natural numbers.

A simpler computational method (also called an `algorithm`) is one that scans the input number bit by bit from the most-significant bit (msb) to the least-significant-bit (lsb).

A temporary space (variable) called `RES` for result is used to hold the currently computed value. When a new bit, call it `b` is encountered, then the contents of `result` are first doubled and to this new value the value of `b` is added (obviously we only add if `b` is 1, since an addition of a zero value won't change the result).

Note that this `algorithm` works even if we do not know how many digits the binary number has. (You can feed the digits to the algorithm one by one starting with the least-significant one.)

```
Algorithm Base-2-to-Base-10              Old RES    b   New RES     . shows bit that is being read

RES=0;                                      0       -   -          1  0  1  0  1  1

repeat until all bits are read        2 *   0   +   1 = 1         .1  0  1  0  1  1
  read_next_bit b;                    2 *   1   +   0 = 2          1 .0  1  0  1  1
  double RES;                         2 *   2   +   1 = 5          1  0 .1  0  1  1
  add b to RES;                       2 *   5   +   0 = 10         1  0  1 .0  1  1
                                      2 *  10   +   1 = 21         1  0  1  0 .1  1
                                      2 *  21   +   1 = 43         1  0  1  0  1 .1
```

Definition of algorithm. An **algorithm** is a well-defined sequence of computational steps that takes some value or set of values as input and generates some value or set of values as an output.

In the example of an algorithm above, the input is a number in base-2 notation (a binary number) and the output is its base-10 value expressed as a decimal number.

(c) Copyright A. Gerbessiotis. All rights reserved.

Binary Numbers and Hexadecimal numbers

Binary number are to base two. Hexadecimal numbers are to base 16. Octal numbers are to base 8. Decimal numbers are to base 10. It is easy to represent binary numbers, we borrow two of the digits used in decimal notation, that is 0 and 1. It also easy to represent octal numbers: we use `0-7` as digits. However with hexadecimal numbers we are out of luck. 0-9 offer us only 10 choices; the remaining six are alphabetic characters `A-F`. One easy way to find the representation of a number in hexadecimal form is to get its binary form, and then group its bits into groups of 4. A group of 4 bits takes $2 \times 2 \times 2 \times 2 = 16$ different values since each bit takes at most two different values. Thus the following table gives all 4 bit sequences in binary, decimal and hexadecimal notation. We can even group bits into groups of three. Since three bits take 8 different values a base-8 representation results: ann integer is thus written in octal notation!

Binary	Decimal	Hexadecimal	Octal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	10
1001	9	9	11
1010	10	A	12
1011	11	B	13
1100	12	C	14
1101	13	D	15
1110	14	E	16
1111	15	F	17
10000	16	10	20
10001	17	11	21

```
Notes
Octal 17 is  written as      0o17
    1*(8**1)+ 7*(8**0)= 8+7 = 15

Hexadecimal 11 is written as 0x11
    1*(16**1)+1*(16**0)=16+1 = 17


Hexadecima to Decimal  use A10.1
Octal      to Decimal
```

(c) Copyright A. Gerbessiotis.

1.11. Base-10 to Base-2 conversion.

Converting a decimal (base-10) integer into binary is as simple as the second algorithm (A10.2) presented for base-2 to base-10 conversion.
We start with the input number, we call it `INPUT` generically, and we want to convert it into binary. We generate bits starting with lsb (least-significant-bit) first. If `INPUT` is odd then the bit that will be generated will be an 1, and we follow this by a subtraction of this 1 value from `INPUT`, and then division of the result `INPUT` by two. If `INPUT` on the other hand is, even the bit generated is a 0, and then we just divide `INPUT` by two. (There is no point subtracting 0.) We repeat this while `INPUT` is still greater than zero, i.e. our method (algorithm) terminates as soon as `INPUT` becomes 0.

```
Algorithm Base-10-to-Base-2(INPUT)
repeat until INPUT is 0
  1. INPUT is odd;
     generate   1;
     subtract 1 from INPUT;
     divide INPUT by 2;
  2. INPUT is even;
     generate   0;
     divide INPUT by 2;
```

o/e	INPUT		b		New INPUT	Bits generated
o	43	-	1	/2=	21	1
o	21	-	1	/2=	10	1 1
e	10	-	0	/2=	5	0 1 1
o	5	-	1	/2=	2	1 0 1 1
e	2	-	0	/2=	1	0 1 0 1 1
o	1	-	1	/2=	0	1 0 1 0 1 1
	0					

^ o/e= ODD/EVEN for the number on its right

(c) Copyright A. Gerbessiotis. All rights reserved.

Natural numbers aka non-negative integers aka unsigned integers. In the previous examples all integers used were natural integers. Thus we did not require some **sign** information to identify the number as positive or negative (or zero) when written in decimal or converted into binary. We often use the term **unsigned integer** to denote non-negative numbers (i.e. **natural numbers** and the zero).

An n-bit unsigned integer. An n-bit unsigned integer represents all non-negative integers in the range from 0 through $2^n - 1$.

MATLAB and n-bit unsigned integer. Matlab supports $n = 8$, $n = 16$, $n = 32$ and $n = 64$ unsigned integers, that is 1B, 2B, 4B, and 8B unsigned integers respectively. They are explicitly named `uint8, uint16, uint32, uint64`. MATLAB and us usually refer to them as 8-bit, 16-bit, 32-bit and 64-bit unsigned integers. When one wants to represent 5 in MATLAB as 16-bit unsigned integer, one should write explicitly `uint16(5)`. Then 2 bytes will be used to store the 5; the first byte is `00000000` and the second one is `00000101`.

Positive and negative integers and zero. In order to able to represent negative integers we need to somehow represent the sign of the number as positive (say +) or negative (say −). We somehow need to incorporate this sign with the binary digits of the number. The simplest way to accomplish this is to borrow one of the bits of a binary number and use the `0` value of it to represent a positive number and the `1` value of it to represent a negative number. That is too easy. The problem that we might face is zero: Will its sign be positive or negative, i.e. a `0` or a `1`?

How we handle zero, is going to affect the discussion that follows. But let us resolve first one more issue. What bit of a bit sequence will denote the sign? The answer to this question is rather easy: the left-most bit. Note that we use the term left-most rather than the dangerous (in this setting) most-significant digit or most-significant bit.

In addition, when we deal with binary (positive and negative) integers we also need to be explicit about the number of digits in the representation.

MATLAB conversion functions. MATLAB provides the following functions for converting from one base to another `bin2dec , dec2bin, hex2dec, based2dec` . For example, base2dec('1237',8) will conert the octal integer 1237 into decimal. In this case the integer in octal is presented as a string and enclosed in single quotation marks.

Signed mantissa and one's and two's complement

The are three major representation of (negative and non-negative) integers.

1. **Signed mantissa.** An n-bit integer in signed mantissa has one sign bit (the leftmost one) and $n-1$ bits for the value. Thus integers in the range $-2^{n-1}, \ldots, 2^{n-1}$ can be represented with two zeroes, a negative and a positive one available.

2. **One's complement.** It is similar to the signed mantissa representation for range and number of zeros. Thus an n-bit one'complement integer can represent all integers in the range $-2^{n-1}, \ldots, 2^{n-1}$ with two zeroes (the positive zero being an all-zero sequence and the negative zero an all-one bit sequence).

3. **Two's complement.** An n-bit integer in two's complement has a left-most bit used as a sign bit, and the remaining $n-1$ bits for its value. There is only one zero (all bits are zero) and the integers represented are in the range $-2^{n-1}, \ldots, 2^{n-1}-1$. MATLAB uses this representation and supports `int8, int16, int32, int64` two's complement integers with 8, 16, 32, 64 bits or equivalently with 1B, 2B, 4B, and 8B.

   For this two's complement representation note the asymmetry between positive and negative numbers. There are 128 negative values represented and 128 non-negative values represented. Among the 128 non-negative values, 127 are positive, 1 of them is the zero.

```
8-bit two's complement  i.e. int8 in MATLAB
negative values  :  -128 , ... ,  -1        128 values total
zero             :                 0        1   value  total
positive values  :  1    , ... , 127        127 values total
```

(c) Copyright A. Gerbessiotis. All rights reserved.

1.12. Signed-mantissa representation of n-bit integer in binary.

The simplest way to represent a negative integer number is to use one bit to denote the sign of the number (positive or negative) and the remaining bits to represent the (positive) value of the number, i.e. its absolute value that we call `mantissa`. This way $+43$ and -43 will only differ in the sign bit. A `1` sign indicates a negative number and a `0` sign a positive number. If we use an n-bit representation, only $n-1$ bits can be used for the mantissa (and one for the sign). These $n-1$ mantissa bits can represent 2^{n-1} different values ranging from 0 to $2^{n-1}-1$ inclusive. Therefore including the sign, the range of values represented with n-bit signed-mantissa is $-2^{n-1}+1 \ldots 0 \ldots +2^{n-1}-1$.

One might have observed a problem with this representation: there are two zeroes. For example for 8-bit signed-mantissa, 10000000 and 00000000 are both the negative and positive zero.

Positive integer. The signed-mantissa n-bit representation of a positive integer x is the integer itself in binary.

Negative integer. The signed-mantissa n-bit representation of a negative integer x is the positive integer $-x$ represented in binary with its leftmost bit turned from zero into a one to indicate the sign (i.e. negative number).

Zero. The representation accepts two zeroes, a positive and a negative one with all bits being zero in the positive zero and only the left-most bit being one in a negative zero.

1.12.1 Example.

Thus for 8-bit integers, $+43$ becomes `00101011` and -43 becomes `10101011`. The left-most (cautiously most-significant-bit) is the sign and differs in $+43$ from -43; the remaining 7 bits are the same `0101011` and correspond to natural number 43 including the leading (left-most) insignificant zero. Note that one NEEDS TO TELL US THAT `10101011` is in signed-mantissa representation. Otherwise we might think that this a natural (i.e. `unsigned`) integer whose value is 171.

We also note the following. If `10101011` is viewed as a natural number its decimal value is $128+43=171$. Thus if x is a positive integer number, then we can represent $-x$ in signed-mantissa 8-bit notation by considering $128+x$, and writing down its bits as if it was a natural number rather than $-x$.

Thus when we are given a binary representation `10101011` we can view it as an `unsigned` integer number representing integer 171, or in some other way. One such other way is signed-mantissa. In such a case this bit sequence becomes a -43. This is not the only way we can view this bit sequence. More ways are shown later.

(c) Copyright A. Gerbessiotis. All rights reserved.

Fixed-point and Floating-point numbers

At this point we are going to skip details on how to deal with real numbers in binary. The Electrical and Computer Engineering Department has plenty of courses discussing the subject in computer organization or computer architecture. One easy way to incorporate real numbers is to assume that in fixed-point n-bit representation, some of the bits represent the integer part (say a of them) and the remaining $n - a$ bits the decimal If one does not want to commit that many bits, a floating-point representation is used where the decimal point is not fixed to a digits or $n - a$ decimals.

```
                   Integer  Fixed-Point   Fixed-Point
                            abcd.efgh     abcde.fgh
  00011100           28       1.75          3.50

abcd.efgh = a x 8 + b x 4 + c x 2 + d x 1 + e x 1/2 + f x 1/4 + g x 1/8 + h x 1/16
abcde.fgh = a x16 + b x 8 + c x 4 + d x 2 + e x 1   + f x 1/2 + g x 1/4 + h x 1/8
```

Floating-point representation and MATLAB: `single` and `double`. MATLAB supports 32-bit floating-point numbers called **single** and 64-bit floating-point numbers called **double**. Not all real numbers can be represented in `single` or `double` notation. Thus a 5 thus it to mean 5.0 is written `single(5)` in 32-bit floating-point representation, and `double(5)` in 64-bit floating-point representation in MATLAB. If you want an 8-bit unsigned integer instead write `uint8(5)`, but a 32-bit two's complement would require `int32(5)`.

Default representation of a number in MATLAB. When you write 5 in MATLAB, MATLAB always interprets it as `double(5)` i.e. it represents 5 not as an integer but as a 64-bit floating-point real number, i.e. as if you had written 5.0. Thus the `double(5)` is optional as it is the default interepretation for 5.

MATLAB realmax, realmin, intmax, intmin, NaN, Inf. The realmax and realmin return the largest and smallest (positive) real number representable. For intmax, intmin the largest positive and smallest negative integer numbers are returned (for 32-bit integers expect $2^{31} - 1$ and -2^{31} respectively by default. An intmax('int8') can show values for other integer types. Similarly for example, realmax('single'). A `NaN` is not a number (usually the result of operations involving division or multiplication of infinity by infinity or 0). An infinity in MATLAB is `Inf`.

MATLAB accuracy. Although `cos(pi/2)` or `sin(pi)` are expected to be zero, they are not (necessarily) in MATLAB. Expect non-zero values that are very small indeed. MATLAB function eps (for epsilon) shows the limitations of real numbers in MATLAB. All numbers for example between 5 and less than 5+eps(5) are considered indistinguishable from 5. Thus you would be surprised by the result `(5+eps(5)/2) == 5`.

(c) Copyright A. Gerbessiotis. All rights reserved.

ASCII

In the preceding pages we viewed several 8-bit or so sequences in a variety ways, as positive or negative numbers in various representations. An 7-bit bit-sequence can also be viewed as an ASCII character code. When we talk about characters we store them in a byte (remember that a byte is the minimum storage that can be used in a byte). For ASCII characters that extra bit is 0 or an error-correction parity bit. The table below contains all 128 ASCII character codes from 0 to 127 arranged in 8 rows (0-7 in octal or hexadecimal) and 16 columns (0-F in hexadecimal). The ASCII code for a character can be retrieved by concatenating the row index (code) with the column index code. For example `A` is in row `4` and column `1` i.e. its hexadecimal code is `0x41` (the prefix `0x` means hexadecimal). Its row index `4` in binary is `100` and `1` in decimal or binary is `0001`. Thus the code for `A` is `1000001` which is `65` in decimal or `0x41` in hexadecimal. The first two rows (0-31) contain Control Characters represented by the corresponding mnemonic code/symbol. Code `32` or `0x20` is the space symbol.

The Unicode character set uses 2 bytes (1 wide character) to represent one character. The Unicode character for an ASCII character remains the same if one adds extra zeroes. Thus `DEL` which is `0x7F` in ASCII has UNICODE code `0x007F`.

\	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

NUL = null	BS = Backspace	DLE = Datalink escape	CAN = cancel
SOH = start of heading	TAB = horizontal tab	DC1 = Device control 1	EM = end of medium
STX = start of text	LF = linefeed/newline	DC2	SUB = substitute
ETX = end of text	VT = vertical TAB	DC3	ESC = escape
EOT = end of transmission	FF = form feed/newpage	DC4	FS = file separator
ENQ = enquiry	CR = carriage return	NAK = negative ACK	GS = group separator
ACK = acknowledge	SO = shift out	SYN = synchronous idle	RS = record separator
BEL = bell	SI = shift in	ETB = end of trans. block	US = unit separator

ASCII. An ASCII character is 7-bits but it is stored in 1B as a byte is the minimal amount of information in a computer.

Unicode and MATLAB. MATLAB supports 2B i.e. 16-bit UNICODE **char** characters.

Find the Unicode or ASCII value of a character in MATLAB. Type ’A’+0 to confirm that 65 is the (decimal) code for an A. ’ ’+0 gives 32 for the single space (between the quotation marks, if it is not clear).

(c) Copyright A. Gerbessiotis. All rights reserved.