logo.png

Indian Institute of Information Technology
Design and Manufacturing, Kancheepuram
Chennai – 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
Computational Engineering

Instructor
N.Sadagopan

Objectives:

- To learn to work with machines (computers, ATMs, Coffee vending machines).
- To understand how machines think and work.
- To design instructions which machines can understand so that a computational task can be performed.
- To learn a language which machines can understand so that human-machine interaction can take place.
- To understand the limitations of machines: what can be computed and what can not be computed using machines.

Outcome: To program a computer for a given computational task involving one or more of arithmetic, algebraic, logical, relational operations.

1 Evolution and Base Conversion

Why Machines ?
We shall now discuss the importance of automation; human-centric approach (manual) versus machines. Although machines are designed by humans, for many practical reasons, machines are superior to humans as far as problem solving is concerned. We shall highlight below commonly observed features that make machines powerful.

- Reliable, accurate and efficient.
- Can do parallel tasks if trained.
- Efficient while peforming computations with large numbers. If designed correctly, then there is no scope for error.
- Good at micro-level analysis with precision.
- Consistent

It is important to highlight that not all problems can be solved using machines (computers). For example, (i) whether a person is happy (ii) whether a person is lying (not speaking the truth) (iii) reciting the natural number set (iv) singing a song in a particular *raga*.

Computer: A computational machine In this lecture, we shall discuss a computational machine called *computer*. The notion *computation* is a process/science that solves a problem in a systematic way using computational machines such as calculators and computers. For example, to perform the addition of two numbers using a calculator, we usually perform the following;

(a) Input: feeding numbers into the machine, say, 2 and 3.
(b) Computation: addition arithmetic, select the appropriate operator.
(c) Output: machine displays the result of the computation.

Essentially, any scientific process consists of a set of inputs, one or more computations and a set of outputs. The *computational engineering* focuses on implementation aspects of the above scientific process using computers (any computational machine). This calls for a study on
(i) how are inputs given to computers.
(ii) how are inputs represented and stored by the computer internally.
(iii) how is the computation done using digital circuits.
(iv) how is the result of a computation displayed to the user.

Observe that humans are trained to think about numbers in the decimal system $(0, 1, \ldots, 9)$, and perform computations in the decimal system itself. However, the scenario is very different with machines. Note that any electronic circuit (digital circuit) always operates at two levels; Logic 1 - signifies the presence of a voltage, Logic 0 - signifies the absence of a voltage. Thus, we naturally obtain Base-2 system (0 or 1), whereas, the human computational model works with Base-10 system $(0, 1, \ldots, 9)$.
This shows that to facilitate human-computer interaction, we need an *encoding scheme* that converts a decimal number into a binary number before the interaction takes place, and similarly, a *decoding scheme* that converts a binary number into a decimal number post the interaction. Apart from arithmetic operations, one can perform other computations such as playing a song, store/manipulation of an image and recording of some activity. For all these operations, an appropriate encoding/decoding is necessary to facilitate human-computer interaction. Interestingly, the tasks performed by modern-day computers digitally have one-one correspondence with tasks performed by humans naturally. The following table summarizes a few tasks;

`Task`	`Tools in Digital Computers`	`Tools in Human Computers`
Record/Capture an activity	Cameras	Eyes
Sensing	Sensors	Nose/Tongue
Talking	Speakers	Mouth
Hearing	Microphones	Ears
Reading/Writing	Keyboard/Printer	Hands
Computing	Central Processing Unit (CPU)	Human brain
Storage	Random Access Memory (RAM), Hard disk	Short-term/Long-term memory

The data captured by digital components such as cameras, microphones, etc., follow an appropriate encoding/decoding for internal storage which is summarized below.

`Data from`	`Encoding (Storage format)`	`Internal Representation`
Keyboard	ASCII (American Standard Code for Information Interchange)	Input text consists of *alphabets, symbols, numbers* and are mapped to decimal numbers. Further, such decimal numbers are converted into binary for representation.
Microphone	MP3, Wav	Sound signal is converted into a sequence of decimal numbers using *sampling technique*. Decimal numbers represent *amplitude* of the signal. Further, such decimal numbers are converted into binary.
Cameras (Image)	bmp, jpeg	Image is viewed as a matrix and each cell represents a decimal number termed as *pixel value.* Pixel values represent the contribution of colors Red, Green and Blue, the fundamental color set. Further, such decimal numbers are converted into binary.
Cameras (Video)	MP4	Video consists of a sequence of images coupled with an audio component. Hence, the above two strategies are used for representation.

While retrieving the stored data, the decoding technique converts the stored binary data into a text or a sound or an image. Having highlighted the importance of binary system and how any data can be stored in binary format, we shall now discuss decimal to binary and binary to decimal conversion procedures.

2 Base Conversion: Decimal-Binary-Decimal

We shall first discuss the conversion for unsigned numbers. In the latter half, we shall focus on signed numbers and their conversion procedures.

Decimal to Binary: Given a decimal number x, the approach is to divide x repeatedly by two (base in binary) and record the remainder at each stage. The approach terminates, when the quotient from Stage i is one. The binary equivalent of decimal number x is precisely the quotient recorded at Stage-i followed by remainders recorded in Stage-i, Stage-$(i-1)$,...,Stage-1. We shall now illustrate with two examples.

(i) $x = 71$

Stage number	Repeated Division	Quotient (Q)	Remainder (R)
1.	$71 \div 2$	35	1
2.	$35 \div 2$	17	1
3.	$17 \div 2$	8	1
4.	$8 \div 2$	4	0
5.	$4 \div 2$	2	0
6.	$2 \div 2$	1	0

Thus, the binary equivalent of 71 is, $(71)_{10} = (1000111)_2$. Note that the notation $(y)_r$ says that the number y is represented in Base r system.

(ii) $x = 127$

Stage number	Repeated Division	Quotient (Q)	Remainder (R)
1.	$127 \div 2$	63	1
2.	$63 \div 2$	31	1
3.	$31 \div 2$	15	1
4.	$15 \div 2$	7	1
5.	$7 \div 2$	3	1
6.	$3 \div 2$	1	1

Thus, the binary equivalent of 127 is, $(127)_{10} = (1111111)_2$.

Approach 2: Given a number x, to find the binary equivalent of x, perform the following: (i) Identify the largest integer k such that $\frac{x}{2^k}$ leaves a quotient one. (ii) Let $y = x - 2^k$. Identify the next largest integer $k' < k$ such that $\frac{y}{2^{k'}}$ leaves a quotient one. (iii) Repeat the above task until $y = 0$. The binary equivalent of x is given by the vector of size $k+1$ such that integer $l \in [0..k]$ is set to '1' if in the above process, the division by 2^l leaves a quotient one, otherwise it is set to '0'. Note that in the $(k+1)$ vector, the bit positions are Bit 0, Bit 1, ..., Bit k.

(i) $x = 71$

`Stage number`	`Identifying the largest integer`	`Subtraction`	`Setting a bit in the output vector`
1.	Largest integer k such that $\frac{71}{2^k}$ leaves a quotient one is $k = 6$	$71 - 2^6 = 7$	Bit 6 is set to '1'
2.	Since $\frac{7}{2^2}$ leaves a quotient one, $k' = 2$	$7 - 4 = 3$	Bit 5,4,3 are set to 0 and Bit 2 is set to '1'
3.	Since $\frac{3}{2^1}$ leaves a quotient one, $k' = 1$	$3 - 2 = 1$	Bit 1 is set to '1'
4.	As $\frac{1}{2^0}$ leaves a quotient one, $k' = 0$	$1 - 1 = 0$	Bit 0 is set to '1'

Thus, the binary equivalent of 71 is

Position Values:	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary equivalent of 71:	1	0	0	0	1	1	1

Binary to Decimal: Approach 2 presented above, in particular, position values help in obtaining a decimal number from a given binary number. For example; given a binary number $(1\ 0\ \ 0\ 0\ \ 1\ 1\ 1)_2$, to obtain the equivalent decimal number, multiply each bit value (0 or 1) with its position value, and the sum of these products of all bits is precisely the decimal number.

Decimal Equivalent of $(1\ 0\ \ 0\ 0\ \ 1\ 1\ 1)_2$ is

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 64 + 4 + 2 + 1 = (71)_{10}$$

Decimal Equivalent of $(1\ 1\ \ 1\ 1\ \ 1\ 1\ 1)_2$ is

$$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 64 + 32 + 16 + 8 + 4 + 2 + 1 = (127)_{10}.$$

Generalization: Base r Representation
One can generalize the above arithmetic and discuss a conversion strategy for decimal to r-ary number, and vice versa.
Decimal to Base r: Recall that to get a binary number, we performed repeated division by 2. To get a base r number, perform repeated division by r. The remainder is in the range $[0..(r-1)]$, and hence each bit takes the value $l \in [0..(r-1)]$.
Base r to Decimal: Position values are $r^0, r^1, \ldots, r^k$, and to obtain the corresponding decimal number, multiply bit value with its position value and the sum of these products gives the decimal number.

Note that $r = 8$ and $r = 16$ are `Octal Number System` and `Hexa-decimal Number System`, respectively.
`Base 3 system:` We perform repeated division by 3 and the remainder obtained in each stage written in reverse order yields the number in base 3. For example, $(789)_{10} = (1\ 0\ 0\ 2\ 0\ 2\ 0)_3$.
`Base 8 system (octal):` For $(789)_{10} = (1425)_8$; this is obtained by repeatedly dividing the number by 8.
`Base 16 system (hexa):` It is important to highlight that the remainders are in the range [0..15] and while representing the number in hexa notation, the remainders 10 to 15 are represented using alphabets A through F. This is done so that if the remainder value is 10, it should not be read as 1 followed by 0. For example; $(171)_{10} = (AB)_{16}$ and $(789)_{10} = (315)_{16}$.

Claim: Given a decimal number x, the binary representation of x is unique.

Proof: We present a proof by contradiction. Suppose, to the contrary, there exist two representations for x. Let B_1 and B_2 are two such representations. Assume that the number of bits in B_1 and B_2 are same. If not, the smaller one is prefixed with zeros. That is, if $|B_1| = p, |B_2| = q, q < p$, then prefix B_2 with $p - q$

zeros. Since the bit values in B_1 and B_2 are different, there exists a bit position b_i such that $b_i = 1$ in B_1 and $b_i = 0$ in B_2. Let i be the largest index satisfying this property. This shows that the value in $b_j, j \in [i+1..p]$ is same for both B_1 and B_2.
Let $K = \sum_{j=i+1}^{p} b_j \times 2^j$.
X : Decimal equivalent of B_1 is $K + 1 \times 2^i + \sum_{j=0}^{i-1} b_j \times 2^j$.
Y : Decimal equivalent of B_1 is $K + 0 \times 2^i + \sum_{j=0}^{i-1} b_j \times 2^j$.
Note that $\sum_{j=0}^{i-1} b_j \times 2^j \leq 2^i - 1 < 2^i$.
Observe that $X \geq K + 2^i$, whereas $Y \leq K + 2^i - 1$. This implies that $X \neq Y$, and hence they both are two different decimal numbers, a contradiction to the given premise. Therefore, our assumption that there exist two binary representations B_1 and B_2 is wrong. Hence, the claim follows.

Claim: Given a binary number B, the decimal equivalent of B is unique.

Proof: On the contrary, assume that there exist two decimal numbers W and Z. Since $W \neq Z$, when we use Approach 1 to compute the binary equivalent of W and Z, we see that for some Stage i, the remainder is '1' in one case, whereas it is '0' in the other case. This shows that $b_i = 1$ in the binary representation of W and $b_i = 0$ in the binary representation of Z. Thus, the underlying binary representations are different, a contradiction to the premise.

3 Arithmetic in Binary System

Addition of two binary numbers is performed at bit level, similar to decimal system. If the result of bit-wise addition exceeds the range, then the excess value is carried to the next bit as a carry.
Addition Rules: $0+0=0$, $1+0=1$, $0+1=1$ and $1+1=0$ with carry 1. For example,
(i) $(1\ 0\ 1\ 1\ 1\ 1)_2 + (0\ 1\ 0\ 1)_2$

```
Carry: 0 1 1 1 1 0
       1 0 1 1 1 1    (47)10
       0 0 0 1 0 1    (5)10
       -----------
       1 1 0 1 0 0    (52)10
```

Subtraction Rules: $0-0=0$, $1-0=1$, $1-1=0$ and $0-1=1$ with borrow 1. Subtraction is undefined if 'borrow' can not be performed.
(i) $(1\ 0\ 1\ 1\ 1\ 1)_2 - (0\ 1\ 0\ 1)_2$

```
Borrow: 0 0 0 0 0 0
        1 0 1 1 1 1    (47)10
        0 0 0 1 0 1    (5)10
        -----------
        1 0 1 0 1 0    (42)10
```

(ii) $(1\ 0\ 0\ 0\ 1\ 1\ 1\ 1)_2 - (0\ 0\ 0\ 1\ 0\ 0\ 1\ 1)_2$

```
Borrow: 0 1 1 1 0 0 0 0
        1 0 0 0 1 1 1 1
        0 0 0 1 0 0 1 1
        ---------------
```

```
      0 1 1 1 1 1 0 0
```

Observe that in the above subtraction, at Bit 4, we had to borrow '1' from the neighboring bit. Since Bits 5 and 6 are '0', a '1' is borrowed from Bit 7 via Bits 5 and 6. Therefore, borrow bits of Bit 4,5 and 6 are set to '1'. Note that position value of Bit 7 is 128 and when we borrow '1' from Bit 7, we actually borrow 128. Due to borrow, binary value at Bit 6 becomes 1 0 whose decimal equivalent is $2 * 64$, i.e., decimal equivalent(1 0) $\times$ position value. Since 128 can not be stored at Bit 6, it retains 64 and gives the other 64 to Bit 5. Due to this, borrow bit of Bit 6 is set to 1. Similarly, when Bit 5 receives '1', it is actually $2 * 32$. Bit 5 retains 32 and gives the other to Bit 4. Now, Bit 4 can perform the subtraction which is $1\ 0 - 1 = 1$ or $32 - 16 = 16$.

Multiplication Rules: $0 \times 0 = 0$, $1 \times 0 = 0$, $1 \times 1 = 1$ and $0 \times 1 = 0$. For example,
(i) $(1\ 0\ 1\ 1\ 1\ 1)_2 \times (0\ 1\ 0\ 1)_2$

```
   1 0 1 1 1 1  ×  0 1 0 1
--------------------------
      1 0 1 1 1 1
    0 0 0 0 0 0
  1 0 1 1 1 1
0 0 0 0 0 0
--------------------------
0 1 1 1 0 1 0 1 1
--------------------------
```

Note that $(0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1)_2 = (235)_{10}$, which is $(1\ 0\ 1\ 1\ 1\ 1)_2 = (47)_{10} \times (0\ 1\ 0\ 1)_2 = (5)_{10}$. As part of multiplication, we perform bit-wise multiplication similar to decimal arithmetic and perform binary addition on the partial products (column-wise addition).

Division: The simplest approach to division is repeated subtraction. To compute $x \div y$, assuming $x > y$, we perform $x - y$ repeatedly until the result of the subtraction z is less than y. The number of iterations is the *quotient* and z is the *remainder*.

`Addition in Base 3 system:` Addition rules; $0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 2, 0 + 2 = 2, 1 + 2 = 0$ with carry 1 and $2 + 2 = 1$ with carry 1. For example;
$(182)_{10} + (156)_{10} = (338)_{10}$;
$(182)_{10} = (\ \ 2\ 0\ 2\ 0\ 2)_3$
$(156)_{10} = (\ \ 1\ 2\ 2\ 1\ 0)_3$
$(338)_{10} = (1\ 1\ 0\ 1\ 1\ 2)_3$.

`Mulitplication in Base 3 system:` $0 \times 0 = 0$, $1 \times 0 = 0$, $1 \times 1 = 1$, $0 \times 2 = 0, 1 \times 2 = 2, 2 \times 2 = 1$ with carry 1. For example,
$(2\ 0\ 2\ 0\ 2) \times (1\ 2\ 2\ 1\ 0)$

```
------------------------------
         0 0 0 0 0
       2 0 2 0 2
   1 1 1 1 1 1
 1 1 1 1 1 1
 2 0 2 0 2
---------------------------
1 1 0 2 2 2 1 1 2 0
---------------------------
```

4 Non-negative Real Numbers: Decimal Number with a Fraction

In the previous section, we have seen the representation of decimal numbers without fractions (natural numbers or non-negative integers). We shall now discuss an approximate binary representation of non-negative real numbers.
To represent a number of the form $(x.y)_{10}$, in binary, the simplest scheme is to represent both x and y in binary with a dot separating the two. That is, $(x.y)_{10}$ = (`binary equivalent of` x `. binary equivalent of` $y)_2$. However, in practice, the number of places after the dot is restricted, usually six places. This approximation is acceptable as the contribution from places after six is relatively insignificant. We present a strategy which gives an approximate representation for $x.y$ and the approximation is on y.

(i) $(17.123)_{10}$.
The number 17 is represented as $(1\ 0\ 0\ 0\ 1)_2$, whereas, .123 represented as follows;
Perform repeated multiplication:
$0.123 \times 2 = 0.246$
$0.246 \times 2 = 0.492$
$0.492 \times 2 = 0.984$
$0.984 \times 2 = 1.968$
$0.968 \times 2 = 1.936$
$0.936 \times 2 = 1.872$

The binary equivalent of .123 is $(0\ 0\ 0\ 1\ 1\ 1)_2$. That is, the integer part from each stage of the multiplication written in sequence is the desired approximate value of 0.123. Thus, $(17.123)_{10} = 1\ 0\ 0\ 0\ 1.0\ 0\ 0\ 1\ 1\ 1$.

(ii) $(7.25)_{10}$.
To represent 0.25, perform the following;
$0.25 \times 2 = 0.5$
$0.5 \times 2 = 1.0$, stop the process as the fractional part is 0.
Thus, $(7.25)_{10} = (0111.010000)$.

Binary with dot representation to Decimal with fraction:
$(1\,0\,0\,0\,1.0\,0\,0\,1\,1\,1)_2 = 1{\times}2^4{+}0{\times}2^3{+}0{\times}2^2{+}0{\times}2^1{+}1{\times}2^0{+}0{\times}2^{-1}{+}0{\times}2^{-2}{+}0{\times}2^{-3}{+}1{\times}2^{-4}{+}1{\times}2^{-5}{+}1{\times}2^{-6}$.
Observe that the above summation does not yield $(17.123)_{10}$, in particular, the fractional part is not 0.123 since the approximate value is being stored as part of the representation.

5 Integer Representation

Integers can be represented in three ways: (i) signed representation (ii) 1's complement representation (iii) 2's complement representation.
Signed: To represent signed decimal numbers in binary system, we use a sign bit along with the binary representation of the magnitude. For a negative number, sign bit is '1', and for a positive number, it is '0'. For example; to represent -11, we use 5 bits, the first bit is the sign bit and the next four bits correspond to the magnitude of 11. That is, $(-11)_{10} = (\mathbf{1\ 1\ 0\ 1\ 1})_{\mathbf{2}}$. Similarly, $(17)_{10} = (\mathbf{0\ 1\ 0\ 0\ 0\ 1})_{\mathbf{2}}$.
The following table gives the range of values for a 3-bit system in signed representation.

Integer	Representation
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

The most significant bit (bit 3) represents the sign bit. Further, '0' has two representations; 000 (+0) and 100 (-0). Thus, in a 3-bit signed binary system, the integers represented are in the range -3 to 3, $-(2^{3-1}-1)$ to $2^{3-1}-1$. In general, a r bit signed binary system can represent numbers in the range $-(2^{r-1}-1)$ to $2^{r-1}-1$.
We shall next discuss addition/subtraction in signed system. Addition is performed if sign bits of both the input numbers are same and if they are different, then subtraction is performed. As long as the result of the addition is within the range, the addition is performed following binary addition rules and the sign bit of the resultant is '1' if both the numbers are negative and the sign bit is '0' if both the numbers are positive. For subtraction, if $|a| > |b|$, then sign bit of $a - b$ is '0', otherwise, it is '1'. For example; the sign bit of $4 - 2$ is '0' and $2-4$ is '1'. If the result of an operation goes out of the range, then overflow/underflow is said to occur.

1's complement: In this system, negative numbers are represented in complement form, that is, 1's are flipped to 0's and vice-versa. For, positive numbers, the 1's complement representation is same as the binary representation without any flipping of bits. Finding 1's complement of a k-bit binary number is equivalent to subtracting that number from $2^k - 1$.
For example; to represent -11, we use 5 bits, the first bit is the sign bit (implicit) and the next four bits correspond to the magnitude of 11. That is, $11 = 01011$, 1's complement of $11 = 10100$, which is the binary representation of -11 in 1's complement system. Similarly, $(17)_{10} = (\mathbf{0\ 1\ 0\ 0\ 0\ 1})_{\mathbf{2}}$.
The following table gives the range of values for a 3-bit system in 1's complement representation.

Integer	Representation	Reason
0	000	
1	001	
2	010	
3	011	
-3	100	Binary(3)=011, 1's(3)=100
-2	101	
-1	110	
-0	111	Binary(0)=000, 1's(0)=111

Observe that in 1's complement system, the most significant bit (bit 3) is '1' for all negative numbers and it is '0' for all positive numbers. This implicitly represents the sign bit. Further, similar to signed system, '0' has two representations; 000 (+0) and 111 (-0). Thus, in a 3-bit 1's complement binary system, the integers represented are in the range -3 to 3, $-(2^{3-1}-1)$ to $2^{3-1}-1$. In general, a r bit 1's complement binary system can represent numbers in the range $-(2^{r-1}-1)$ to $2^{r-1}-1$.
We shall next discuss arithmetic (addition and subtraction) in 1's complement system, assuming 3-bit system. While performing arithmetic, if there is a carry from the most significant bit, then this carry is not ignored, instead, added with the result. For example;

(i) $-2-1$
$-2 =$ 1 0 1
$-1 =$ 1 1 0
———————-
0 1 1
1 (carry from the MSB bit)
———————-
1 0 0 (which is -3, the desired answer).

The above scenario happens when two negative numbers are added. Similar to signed arithmetic, overflow and underflow can occur if the resultant does not fit in the specified range. For example;

(i) 14 + 2
Note that we need at least 5 bits to represent 14 and at least 3 bits to represent 2.
$14 = 01110$
$2 = 00010$
$------$
$= 10000$

Observe that the result of 14+2 is -0, and not 16. This is due to the fact that the result, 16, can not be represented in a 5-bit 1's complement system. If the result of adding two positive numbers yields a negative number, then the system is in underflow. Similarly, if the result of adding two negative numbers results in a positive number, then the system is in overflow.
(i) -14 - 2 = -14 + (-2)
Note that we need at least 5 bits to represent 14 and at least 3 bits to represent 2. Binary(14)=01110, 1's complement(14)=10001; Binary(2)=00010, 1's complement(2)= 11101.
$-14 = 10001$
$-2 = 11101$
$------$
$= 01110$

In the above addition, the result is 14 instead of -16, results in overflow. That is, -16 can not be represented in a 5-bit 1's complement system as the range is -15 to 15.

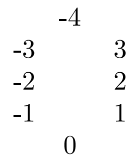
Variant of 1's complement: 2's complement: In this system, negative numbers are represented in complement form, that is, we first find 1's complement, and then add '1' to the result to obtain 2's complement representation. For, positive numbers, the 2's complement representation is same as the binary representation without any flipping of bits.
For example; to represent -11, we use 5 bits, the first bit is the sign bit (implicit) and the next four bits correspond to the magnitude of 11. That is, $11 = 01011$, 1's complement of $11 = 10100$, 2's complement of $11 = 10100 + 00001 = 10101$, which is the binary representation of -11 in 1's complement system. Similarly, $(17)_{10} = (\mathbf{0\ 1\ 0\ 0\ 0\ 1})_{\mathbf{2}}$.
The following table gives the range of values for a 3-bit system in 2's complement representation.

Integer	Representation	Reason
0	000	
1	001	
2	010	
3	011	
-4	100	Binary(4)=100, 1's(4)=011, 2's(4)=100
-3	101	
-2	110	
-1	111	Binary(1)=001, 1's(1)=110, 2's(1)=111

Observe that in 2's complement system, the most significant bit (bit 3) is '1' for all negative numbers and it is '0' for all positive numbers. This implicitly represents the sign bit. Further, in this system, '0' has a unique representation and -4 can be represented in a 3-bit system. Thus, the range of 2's complement system is one more than the range of 1's complement system. Thus, in a 3-bit 2's complement binary system, the integers represented are in the range -4 to 3, $-(2^{3-1})$ to $2^{3-1}-1$. In general, a r bit 2's complement binary system can represent numbers in the range $-(2^{r-1})$ to $2^{r-1}-1$.
We shall next discuss addition in 2's complement system. While performing addition, if there is a carry from MSB, then the carry is ignored. This happens when two negative numbers are added.
(i) 14 + 2
Note that we need at least 5 bits to represent 14 and at least 3 bits to represent 2.
$14 = 01110$
$2 = 00010$
$------$
$= 10000$

Observe that the result of 14+2 is -16, and not 16. This is due to the fact that the result, 16, can not be represented in a 5-bit 2's complement system. If the result of adding two positive numbers yields a negative number, then the system is in underflow. Similarly, if the result of adding two negative numbers results in a positive number, then the system is in overflow.
(i) -14 - 2 = -14 + (-2)
Note that we need at least 5 bits to represent 14 and at least 3 bits to represent 2. Binary(14)=01110, 1's complement(14)=10001, 2's complement(14)=10010; Binary(2)=00010, 1's complement(2)= 11101, 2's complement(2)=11110.
$-14 = 10010$
$-2 = 11110$
$------$
$= 10000$

In the above addition, the result is -16 which can be represented in a 5-bit 2's complement system as the range is -16 to 15.



Range of numbers in 3-bit 2's complement system

Remarks: (i) -4-1=3 in 2's complement system, in the above figure, this can be visualized as clock wise movement by one position with respect to -4. Similarly, 3+1=-4, which is anti-clock wise movement by one position with respect to 3. In general, the range of numbers represented by this system can be arranged in a circular fashion. Accordingly, if the result of adding positive numbers exceeds the range, then an appropriate number from the range of negative numbers will be output as the result. Similarly, for negative numbers.

6 Code Complexity

For a given problem, it is natural to think of more than one logic and program in solving the problem. This calls for a study on how much time (clock cycles) each program takes on a reasonable sized input. As part of theoretical study, we shall fix the following model to analyze/compare programs.

- Basic Arithmetic; addition, subtraction - incurs 1 clock cycle
- Basic Arithmetic; multiplication, division - incurs 2 clock cycles
- Simple assignment such as a=b, a=5 incurs one clock cycle
- Relational, and logical operations incur 1 clock cycle

It is important to note that the model we work with must be close to the reality and consistent with the implementation aspect of the program. For example, one should not assume, multiplication takes less clock cycles than addition.

```
1.

void main()
{

int a, b;  // Simple assignment incur 1 cc

c=a+b; // addition incurs 1 cc, followed by 1 more for assignment, overall, 2 cc

print c; // 1 cc for display

// This program incurs 1+2+1=4 cc

}
```

```
2.

if(condition)  // 1 cc for check, multiple cc if it is a compound expression
{
block of statements // analyze cc for each statement following our model
}

else  // same as cost of if(condn), the compiler does the condn check first before it switches to else
{
block of statements // analyze cc for each statement following our model
}

Overall cost is ,
MAX( cost of condn check + cost of IF block, cost of condn check + cost of ELSE block)

3.

if(marks >=90)        // 1 cc
   print grade 'S'    // 1 cc
elseif (marks >=80)   // 1 cc for if + 1 cc for elseif
   print grade 'A'    // 1 cc
else       // 1 cc for if + 1 cc for elseif
   print grade 'B'    // 1 cc


4. for(int i=0; i<10; i++) // this statement is executed for i=0,...,9 and i=10, overall 11 cc
     st; // whenever for is true, this st is executed, overall, 10 cc.

when i=10, for statement is executed once and condn check terminates
and hence for block will not be executed.
```

7 Cyclomatic Complexity

Verification of a program is important to ascertain whether the program is giving the right output for all valid inputs and error for all invalid inputs. There are many input test cases for a given program and the focus of cyclomatic complexity is to identify the number of independent test cases and corner test cases. Consider the following code;

```
void main()
{
short int a,b;
a=a+b;
b=a-b;
a=a-b;
}
```

Corner cases for the above program include (i) $a = 32767, b = 32767$ (ii) $a = -32768, b = -5$, those inputs that trigger overflow/underflow. Independent test cases include (i) both a and b are positive integers (ii) both a and b are negative integers (iii) a is negative and b is positive (iv) both are zeros. Invalid test cases include chars, special chars and float.

For if-else structure, we need one test input that satisfies if condition and executes the if block, and another test input that fails to satisfy if condition and executes else block. On the similar line, we need three independent test cases for if-elseif-else structure. For switch case with p number of case labels and default, we need p independent test cases.

8 Floating Point Representation

In integer system, while representing the numbers such as 1234, -345, there is no decimal point (dot) or the dot is fixed. That is, 1234 is equivalent to 1234.0, the dot is always at the end and thus yield a fixed-point number system. Interestingly, one can allow the dot to float, thus yielding floating-point representation. For a 32-bit system, the range of numbers represented by float is much higher than the integer. Further, float offers a good precision for the fractional part of the number.

Consider the decimal number, 10.125, if we allow the dot to float, then the following are equivalent;
$10.125 = 10.125 \times 10^0$
$10.125 = 1.0125 \times 10^1$
$10.125 = 1012.5 \times 10^{-2}$
On the similar line, one can allow the dot to float in binary system as well.
$(10.125)_{10} = (1010.001)_2 = 1010.001 \times 2^0 = 10.10001 \times 2^2 = 10100.01 \times 2^{-1}$.

`Normalized Representation:` In normalized representation, all float numbers except zero begin with 1.. That is, before the dot it is just one digit. An unnormalized number can be converted into a normalized number by shifting the dot with appropriate increase/decrease in the exponent.
$1010.001 \times 2^0 = 1.010001 \times 2^3$.
$0.00001010 \times 2^0 = 1.010 \times 2^{-5}$.
Floating point numbers consist of three components, namely exponent, sign and fraction (mantissa). For the number 1.010×2^{-5}, 010 is mantissa (number after the dot), -5 is the exponent, the sign bit of the exponent is '1' and the sign bit of mantissa is '0'. In a 32-bit system, we represent them as per the following distribution;

1-bit each reserved for sign bit of the exponent and sign bit of the mantissa.
7-bit for the exponent and 23 bits for mantissa.

Since the number is represented in normalized form, 1. is ignored and not represented. Thus, we get,

sign mantissa(1-bit)	sign exponent(1-bit)	exponent(7 bits)	mantissa (23 bits)

$1.010 \times 2^{-5} =$

	0	1	0000101	01000000000000000000000	.

$(255)_{10} = (11111111)_2 = 11111111.0 \times 2^0 = 1.1111111 \times 2^7$

	0	0	0000111	11111110000000000000000	.

`Smallest float number:`

	1	0	1111111	1111111111111111111111	.

$= -(2-2^{-23}) \times 2^{127}$. Note that $1.11 = 1.75 = 2-0.25 = 2-2^{-2}$, similarly $1.1111111111111111111111 = 2-2^{-23}$.
`Largest float number:`

	0	0	1111111	1111111111111111111111	.

$= (2-2^{-23}) \times 2^{127}$. `Special Numbers:` To represent special numbers such as 0.0, ∞, $-\infty$, $\sqrt{-1}$, NAN

(not a number), we work with 0 to 126 in the exponent and 127 is reserved for this purpose. Further, we make use of sign bits and representation are as follows;
(i) ∞ with exponent=127, mantissa= all 1's, sign bits of exponent and mantissa are 0
(ii) $-\infty$ with exponent=127, mantissa= all 1's, sign bit of exponent=0 and mantissa=1
(iii) NAN, $\sqrt{-1}$ with exponent=127, mantissa=0

(iv) 0.0 with mantissa=0, sign bit of exponent=1

9 Interesting stuff with loops

```
int a=1, x, counter=0;
while(a < x)
{ a=a*2;
  counter++;
}
```

Let us analyze what is the value of counter if $x = 32$.

Iteration:1	$1 < 32$	counter=1, a=2
Iteration:2	$2 < 32$	counter=2, a=4
Iteration:3	$4 < 32$	counter=3, a=8
Iteration:4	$8 < 32$	counter=4, a=16
Iteration:5	$16 < 32$	counter=5, a= 32
Iteration:6	$32 < 32$, condition fails	no update

The value of the counter is 5 and in general, for arbitrary x, the counter is $\lceil \log_2 x \rceil$.

```
int a=1, x, counter=0;
while(a < x)
{ a=a*3;
  counter++;
}
```

Let us analyze what is the value of counter if $x = 32$.

Iteration:1	$1 < 32$	counter=1, a=3
Iteration:2	$3 < 32$	counter=2, a=9
Iteration:3	$9 < 32$	counter=3, a=27
Iteration:4	$27 < 32$	counter=4, a=81
Iteration:5	$81 < 32$ condition fails	no update

The value of the counter is 4 and in general, for arbitrary x, the counter is $\lceil \log_3 x \rceil$.

```
int a=1, x, counter=0;
while(a < x)
{ a=a*k;
  counter++;
}
```

For arbitrary x, the counter is $\lceil \log_k x \rceil$.

```
int a, x, counter=0;
while(a < x)
{ a=a*2;
  counter++;
}
```

For arbitrary x and a, the value of counter can be calculated as follows; the update on a follows the pattern, $a, 2a, 2^2a, 2^3a, ...$ and we are interested in the largest l such that $2^l a > x$. This implies that l (the value of the counter) is $\lceil \log_2 \frac{x}{a} \rceil$.

```
int a, x, counter=0;
while(a < x)
{ a=a*k;
  counter++;
}
```

For arbitrary x and a, the value of counter can be calculated as follows; the update on a follows the pattern, $a, ka, k^2a, k^3a, ...$ and we are interested in the largest l such that $k^l a > x$. This implies that l (the value of the counter) is $\lceil \log_k \frac{x}{a} \rceil$.

10 Expression Evaluation, Precedence and Associativity

An expression involve operators such as arithmetic $(+, *)$, relational $(<, <=)$, logical (&&), and hence there is a need for fixing the order among operators. The notion *precedence* fixes the priority among operators and the *associativity* defines the order of evaluation if multiple operators of same precedence appear in an expression. The following table summarizes the precedence and associativity for commonly used operators listed in increasing order of priority.

Precedence (priority)	Operators	Associativity
1	Unary operators $-, +, ++, --,$, sizeof()	right to left
2	Binary $*, /, \%$	left to right
3	Binary $+, -$	left to right
4	Shift $<<, >>$	left to right
5	relational $<, >, <=, >=,$	left to right
6	comparison $==, ! =,$	left to right
7	bitwise AND, &,	left to right
8	bitwise XOR,	left to right
9	bitwise OR, $\|$,	left to right
10	logical AND &&,	left to right
11	logical OR $\|\|$,	left to right
12	equality,	left to right
13	comma,	left to right

1. $x = x + 1, y = x$ can be represented succinctly as $y = + + x$. The unary operator $+ + x$ is called as pre-increment operator which says that increment the value of x by one before assigning to y.

2. $y = x, x = x + 1$ can be represented succinctly as $y = x + +$. The unary operator $x + +$ is called as post-increment operator which says that assign the value of x to y followed by increment x by one.

3. Pre/post increment is appropriate if it is part of an expression/printf, otherwise, their behavior is same as simple increment. That is, the output of $x + +; + + x, x = x + 1$ are same as these are simple sequence statements.

4. Suppose $x = 5$, the result of executing $-x{+}{+}$ is -6. Note that this expression has two unary operators $-$ and $++$ and thus the order of evaluation is right to left. $x + +$ increments x by one, subsequently, unary $-$ is assigned to $x + +$.

5. Suppose $x = 5$, the expression $++x++$ evaluates to 7. Since both pre and post operators have same precedence, as per associativity rule, the expression is evaluated from right to left. $x++$ gives 6 and further, $++x++$ updates x to 7.

6. The operator $\tilde{}$(tilde) performs 1's complement arithmetic. That is, if $y = \ x$ and $x = 7$, then $y = 8$. The binary representation of 7 = 0111 and its 1's complement is 1000 which is 8 in unsigned system and -8 in 2's complement system.

7. The operator sizeof() returns the number of bytes occupied by the variable in the primary memory. The number of bytes varies depending upon the data type.

   (a) char - 1 byte
   (b) short int - 2 bytes
   (c) int - 4 bytes
   (d) float - 4 bytes
   (e) long int - 8 bytes
   (f) double - 8 bytes
   (g) long double - 16 bytes

8. `Sequence Point:` Consider the expression $++x+x++$. The binary operator $+$ looks at two operands, namely $++x$ and $x++$ and the compiler realizes that there are two updates to x in the same atomic expression. An atomic expression consists of a single binary operator and two operands. A sequence point is defined for each atomic expression which defines the start and end of evaluation. For the expression $(++x+x++)$, the sequence point starts at $'('$, and ends at $')'$. If a variable is updated more than once in a sequence point, then the result of the expression depends on the implementation of the compiler. For the above expression, if $x = 5$, then the possible evaluation schemes are (a) $5+5$, (b) $6+6$, (c) $7+5$ (d) $7+6$.

9. We shall now discuss how simple assignments and expressions involving unary operator gets evaluated in printf statements. This is again compiler dependent and we shall present one version based on gcc compiler.

   (a) `printf("%d %d %d %d %d", x=x+1, x=x+2, x+5, x++, ++x)`
   Suppose the value of $x = 7$, then the evaluation proceeds as follows; (i) expressions appear in printf are evaluated from right to left (ii) simple expressions without assignments such as $x+5$ are evaluated and output with respect to the current value of x (iii) expressions involving assignments such as $x = x + 2$ are evaluated with respect to current x, x is also updated to $x + 2$, however, the updated x will not be output immediately. The updated x will be used for other expressions involving x while scanning left from the current place. If there are multiple assignments to x in a single printf statement, then the output of all expressions is same as the value of x from the last assignment statement.

   (b) For this example, the compiler updates $++x$, since it is pre-increment, $x = 7+1 = 8$ and outputs 8. Then, it scans $x++$ and outputs 8 as it is post increment. Further, $x = 8 + 1 = 9$. The next expression $x+5$ is a simple expression without assignment, system outputs $9+5 = 14$. Note that x is still 9.

   (c) The next assignment encountered while scanning from right to left is $x = x + 2$, it is evaluated to $x = 9+2 = 11$, however, x is not output now. Although, x is not output, it is updated implicitly to 11. The last statement which is again an assignment is evaluated to $x = 11+1 = 12$. Now, the compiler outputs the value of x as 12 for all assignment statements. Thus we get the following as the output.

   (d) `12 12 14 8 8`

(e) `printf("%d %d %d %d %d %d", x=x+0, x=x*1, x=x, x=x+2-2, x=15, x)`
Since compiler does optimization, expressions $x = x + 0, x = x * 1, x = x + 2 - 2$ all refer to x and hence, it will be treated as simple assignment. Therefore, 15 is the last assignment in the above expression and no further updates to x. Thus, the output is

(f) `15 15 15 15 15 15`

10. We shall now focus on binary arithmetic. Consider the expression; $a * b/c * d\%e * f$. Since $*$, % have same precedence, the order of the evaluation is from left to right as per associativity rule. Therefore, the order of the evaluation is (i) $a * b$ (ii) $(a * b)/c$ (iii) $((a * b)/c) * d$ (iv) $(((a * b)/c) * d)\%e$ (v) $((((a * b)/c) * d)\%e) * f$. Note that the innermost parenthesized expression is executed first followed by next innermost, and so on.

11. Consider the expression; $a + b - c + d$. Since $+$ and $-$ have same precedence, as per associativity, the expression is executed from left to right. The order of the evaluation is (i) $(a + b)$ (ii) $(a + b) - c$ (iii) $((a + b) - c) + d$.

12. For the expression $a+b*c-d/f*e$, the order of the evaluation is from left to right; $(b*c), (d/f), ((d/f)*e)$. Thus we get, $a + (b * c) - ((d/f) * e)$. As per precedence, $+, -$ get the preference and executed as per the order (i) $(a + (b * c))$ (ii) $(a + (b * c)) - ((d/f) * e)$.

13. We shall next discuss shift left $<<$ and shift right $>>$ operators. Shift operators work at bit level and hence we work with the underlying binary representation.

    (a) The expression $5 << 1$ means shift the binary representation of 5 left by one position. The last bit (lsb) is suffixed with 0. That is, $5 = 0101$, $5 << 1 = 1010$, equals integer 10. Similarly, $5 << 2 = 10100$, equals integer 20. In general, if an integer m is shifted left by n positions, then the binary representation of m is suffixed with n positions which is the integer $m \times 2^n$.

    (b) The expression $32 >> 1$ means shift the binary representation of 32 right by one position. The last bit (lsb) is discarded and the most significant bit (msb) is prefixed with 0. That is, $32 = 100000$, $32 << 1 = 010000$, equals integer 16. Similarly, $32 << 2 = 0010000$, equals integer 8. In general, if an integer m is shifted right by n positions, then the binary representation of m is pre-fixed with n positions which is the integer $\frac{m}{2^n}$. For $17 << 2$, the result is $10001 << 2 = 00100$, which is $\lfloor \frac{17}{4} \rfloor$.

    (c) In signed system, while performing shift right, we prefix the sign bit (0 or 1). For example, $-16 << 2$, $10000 << 2 = 11100$, which is -4 in a 5-bit signed system.

    (d) The expression $a << 2 << 1 >> b$ will be evaluated from left to right as $(((a << 2) << 1) >> b)$.

14. Relational Operators: The next operator in the precedence hierarchy is the relational operator. The expression $5 > 2 < 1 >= 0 < -1$ is executed as per the order $((((5 > 2) < 1) >= 0) < -1)$. Note that each atomic expression returns either 0 or 1. That is, $(5 > 2)$ returns 1 and the expression reduces to $(((1 < 1) >= 0) < -1)$, further it becomes, $((0 >= 0) < -1)$. Finally, $(1 < -1) = 0$.

15. Comparison Operators: The expression $a == b! = c == 4$ is evaluated as $(((a == b)! = c) == 4)$

16. Bit-wise operators: AND (&), XOR(), OR (|);
    5&7 = 0101&0111 = 0101,
    12 xor 3 = 01100 xor 00011 = 01111
    6|4 = 0110|0100 = 0110.

17. Logical Operators: AND (&&), OR(||), the expression a && b || $c||d\&\&e$ will be evaluated as $(a\&\&b)||c||(d\&\&e)$, further, $(((a\&\&b)||c)||(d\&\&e))$. Logical AND returns '1' if both the operands are non-zero and logical OR returns '1' if at least one of the operands is non-zero. Logical AND returns '0' if at least one of the operands is zero and logical OR returns '0' if both the operands are zero. For example, $(5||0) = 1$ and $(14\&\&98) = 1$

18. Assignment operator: $a = b = c$ is evaluated as $(a = (b = c))$, the evaluation is from right to left.

19. Comma Operator: The expression $a = (50, a + 2, 40)$ is evaluated from left to right, and a is assigned the value 40.

20. Consider the expression; $a == b > c + 4 \&\& d\%e * f >> 2 ! = 0 \& -1 < 90$, is evaluated in the following order

    (a) Binary $\%, *$ get higher priority. $((d\%e) * f)$.

    (b) Binary $+$, $(c + 4)$

    (c) Shift right; $((d\%e) * f) >> 2$.

    (d) Relational operators; $(b > (c + 4))$, $(-1 < 90)$

    (e) Comparison; $(a == (b > (c + 4)))$, $(((d\%e) * f) >> 2)! = 0$.

    (f) bitwise AND; $((((d\%e) * f) >> 2)! = 0) \& (-1 < 90)$

    (g) logical AND; $(a == (b > (c + 4))) \&\& (((((d\%e) * f) >> 2)! = 0) \& (-1 < 90))$

11 Logical Puzzles

1. It is a game between two persons, A and B. A asks B to think of a number between 1 and 10, say x. B is asked to perform the following in sequence (i) multiply two with x, with the resultant add two. (ii) Further, multiply 5 and add 5. B discloses the result to A and A identifies the number. Model this using a C program.

2. Think of a number game: The game proceeds as follows; (i) Person B asks Person A to think of a number, say, 3457. (ii) Then, B tells A to convert the number into a single digit, i.e., 3+4+5+7=19, further 1+9=10, 1+0=1. At the end of the process of converting into a single digit, you find the resultant 'R1' in the range [0..9]. (iii) B tells A to multiply the resultant R with 9 and again, convert the resultant 'R2' into a single digit. (iv) From now on, B can tell A any arithmetic (add, sub, mul, div, fac, square, square root, etc.). (v) Finally, B guess the result of any sequence of arithmetic operations. We shall give an example scenario as follows;
   (a) B tells A to add 6 to R2. If the original number was 3457, then R1 is 1 and R2 is 1*9=9. The result of this operation is 9+6=15.
   (b) B tells A to multiply 10. The result is 15*10=150.
   (c) B tells A to divide by 30. The result is 150/30=5.
   (d) B tells A to find the factorial. 5!=120.
   (e) B tells A to add 80, the result is 120+80=200.
   (f) B tells A to multiply 10. the result is 200*10=2000.

   B tells A that he wants to end the game and he tells A the final answer was 2000. 'A' is surprised. How did B guess the answer. As a programmer, implement this game assuming you as A and the computer as B. Can we implement this game by assuming computer as A and the end user as B. Model this problem properly and understand the technical intricacies involved before implementation.

3. Given x, print x^2 number of prime numbers.

4. In a group of 7 people, can we claim, there exists either two enemies or two friends. If this claim is true, how do we verify using a C program.

5. Finding odd coin out of a box of n coins.