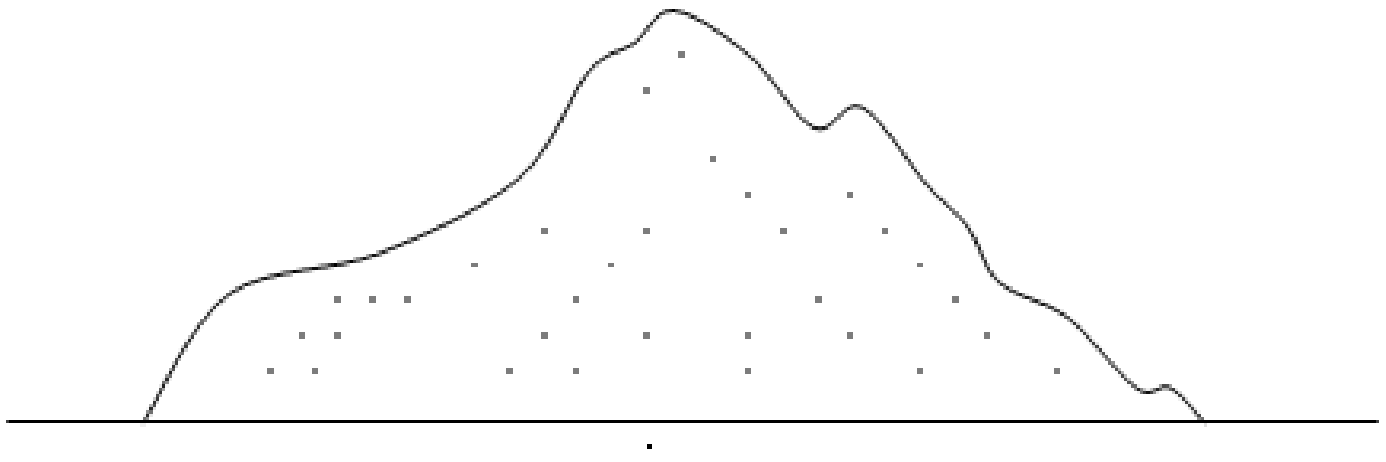
CHAPTER 2

REPRESENTATION OF DATA IN BINARY

- A ‘pile of bits’ is not useful!



We need to organize memory with:

- Addressing

and

- Representation

Number systems

- A *number system* can be defined as a system of *representation* of numbers using numeric characters
  - Unary number system
  - Positional / place value system
  - Binary system

Unsigned Integers

- Unsigned (non-negative) integers make up the data type which can be represented in binary using the simplest possible technique

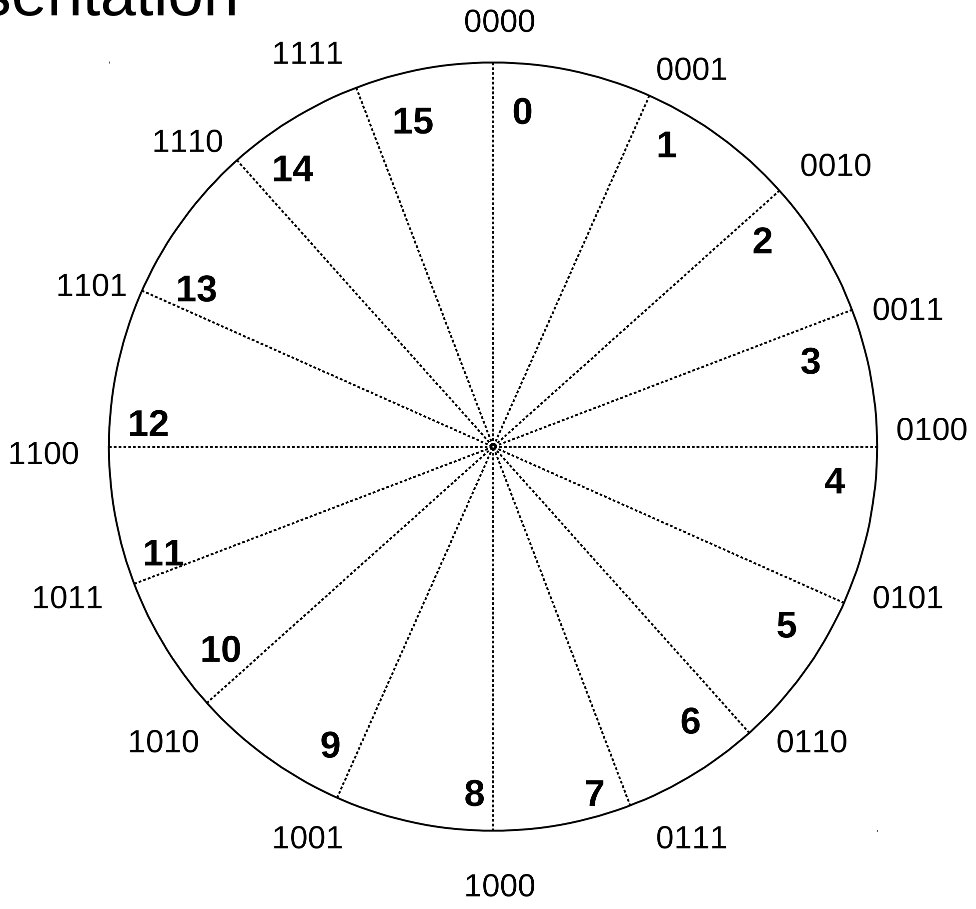
- For numeric value, the general bit pattern will be:

$$b_{n-1}b_{n-2}\ldots\ldots b_3b_2b_1b_0$$

- b_0 is referred to as the *least significant bit* (or LSB), and b_{n-1} is referred to as the *most significant bit* (or MSB).

- An *n*-bit unsigned integer can represent a number from 0 to 2^n-1

- With four bits, the sixteen unsigned numbers which can be represented run from 0 to 15, shown below in a circle
- outside the circle: the binary bit patterns
- inside the circle: corresponding unsigned numbers in decimal representation



- Example: To find the value of 3+4 on this circle, we first locate 3 on the circle, and then count off four more numbers on the circle in the clockwise direction, to arrive at the correct answer 7.

- To find the value of 8+10, we locate 8, and then count off ten more steps in the clockwise direction
  - But our answer comes out as 2, rather than 18 (after 15, our count becomes 0 rather than 16)
  - Reason: Decimal 18 has the binary equivalent 10010, which <u>cannot</u> be represented in four bits.

- In the process of addition, the leftmost ‘1’ fell off, and we were left with the answer 0010, which is decimal 2.

- Since the leftmost ‘1’ in 10010 has place value 16, this ‘dropped bit’ means a ‘loss’ of 16. More technically, the answer is obtained by taking the remainder of 18/16, which is 2.

- This type of remainder is also known as *modulo* operation, so that we can say that the result is:

  (8+10) *modulo* 16 = 2.

- The technical name for this fallen off ‘1’ bit is *carry* or *overflow*.

- For signed arithmetic, there is a difference between *carry* and *overflow*. For unsigned integer arithmetic, these two have the same meaning.

Signed Integers

- Sign and Magnitude representation

  Provide an extra *sign bit* in front of the (unsigned) number.

    ‘0’ in this sign bit represents a positive number, and

    ‘1’ represents a negative number.

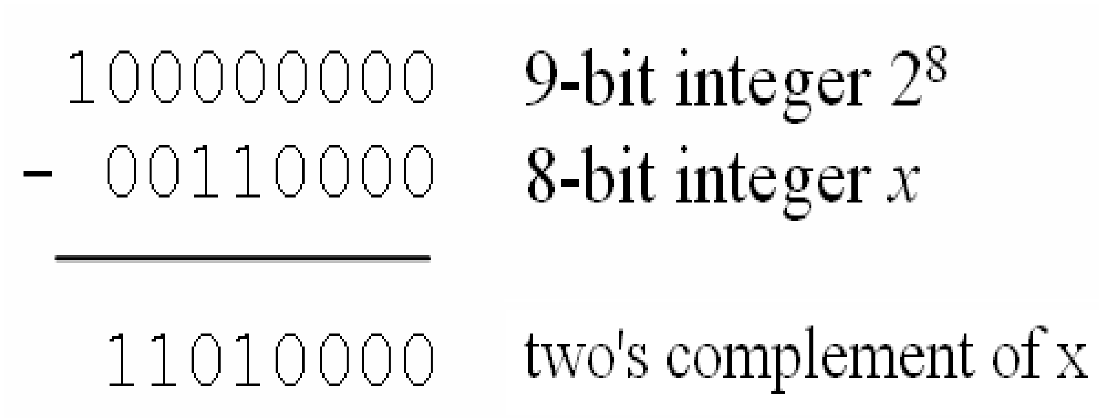
  In designing hardware circuits for integer arithmetic operations, *sign and magnitude* representation is not much used.

Bit-wise complement

- One’s complement
  - ‘0’ and ‘1’ are the bit-wise complements of each other
  - Example: In eight bits, 00110000 equals decimal 48.
  - Therefore, in one’s complement, 11001111 represents the decimal value -48
  - this representation is also not much used in computer hardware

- Two’s complement
  - *Two’s complement* of an *n*-bit binary quantity *x* is defined as the *n*-bit binary quantity given by 2^n-*x*

- For example:
  - The eight-bit integer *x* = 00110000 has decimal value 48.
  - Taking *n* = 8, we get the two's complement of this 8-bit integer as the integer 11010000, as shown in the binary subtraction below:



100000000	9-bit integer 2^8
− 00110000	8-bit integer x
11010000	two's complement of x

- As an unsigned 8-bit integer, 11010000 represents decimal 208.

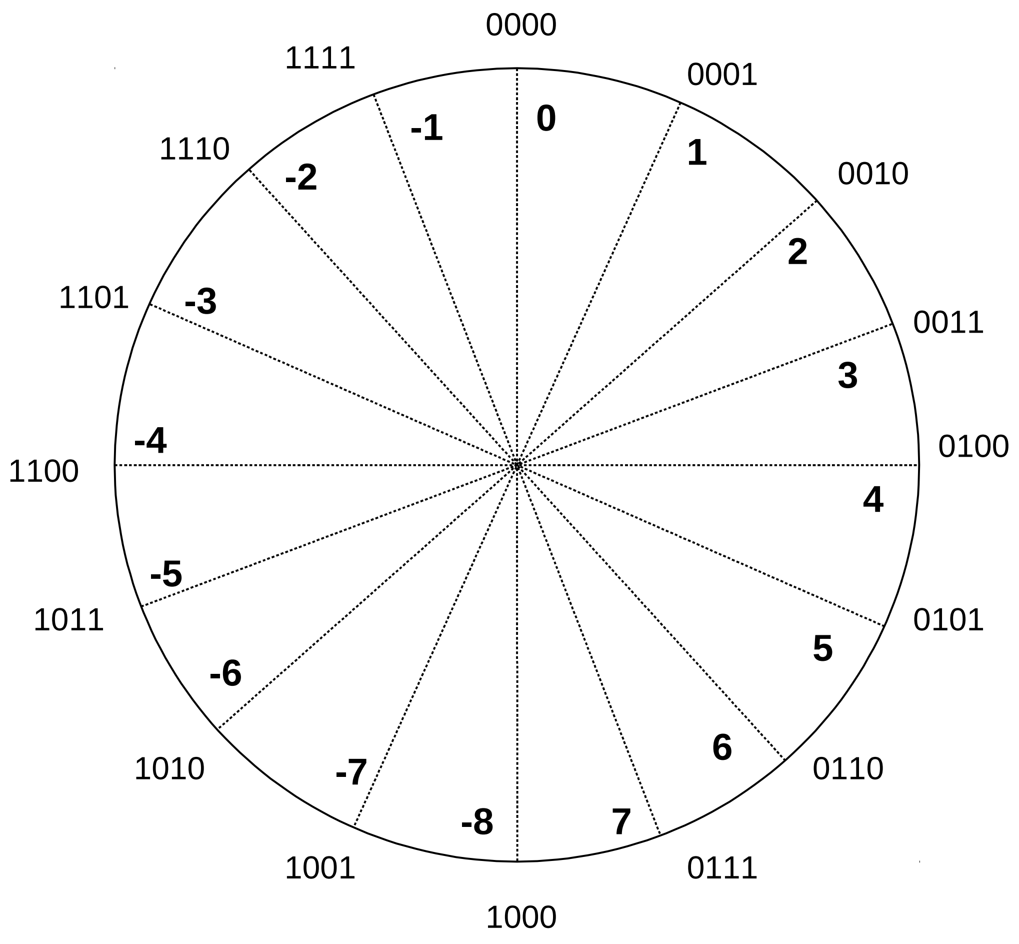
- As *signed two's complement* 8-bit integer, this same bit pattern represents the decimal value

  -48 rather than 208.

- What a particular bit pattern represents depends on the program which processes the bit pattern!

- Two's complement of (two's complement of *x)* equals *x*.

- In the next slide, the circle shows four-bit integers in signed two’s complement representation.
  - Outside the circle, binary bit patterns are shown.
  - Inside the circle, the corresponding signed integers are shown in decimal representation.
  - The range of integers represented runs from -8 (at the bottom), in the clockwise direction, to +7.
  - The leftmost bit remains 1 for negative numbers and 0 for non-negative numbers – and therefore the leftmost bit is considered to be the *sign bit*.


0000
0
0001
1
0010
2
0011
3
0100
4
0101
5
0110
6
0111
7
1000
-8
1001
-7
1010
-6
1011
-5
1100
-4
1101
-3
1110
-2
1111
-1

- To find the value of 3+4 on this circle, we first locate 3 on the circle, and then count off four more numbers on the circle in the clockwise direction, to arrive at the correct answer 7.

  But if we try to find 3+5, we get the answer -8 rather than 8!

- This is because the discontinuity in numeric values is now occurring at the bottom of the circle:

  +7 is followed by -8, rather than by +8. This is a jump of +16 or -16, depending on the direction in which we move.

- With four-bit signed integers, the range of representation is from -8 to +7.

- With *n* bit representation, the range is from -2^{n-1} to $2^{n-1}-1$.

  e.g.16 bit signed integers would have the range -32768 to +32767.

Carry & Overflow in signed integers

- With signed integer representations, *carry* and *overflow* refer to two different arithmetic conditions.
- In two's complement form, when we add 1 to 7, the resulting bit pattern 1000 represents the number -8 rather than +8; and a similar thing happens when we subtract 1 from -8.
- This condition is known as *overflow*, since the correct results (+8 and -9 above) are out of the range of representation.
- *Carry* occurs when we move between 1111 and 0000 (in either direction) on the circle
- *Overflow* occurs when we move between 0111 and 1000 (in either direction) on the circle.

- For an integer *x*, its *two's complement* can also be found in two alternate simple ways.

  (i) Find the bit-wise complement of *x*, and then add 1 to it.

  (ii)

  - Starting from the rightmost (LSB) bit of *x*, scan the bits from right to left.
  - Leave unchanged all the initial 0's on the right, as well as the first 1 coming from right to left.
  - Complement bit-wise all the remaining bits in *x*.

- Two’s complement representation is used in computer systems
  - Because, with this representation, the arithmetic operations of addition and subtraction remain unchanged between signed and unsigned integers.

- Sign & magnitude representation, and one’s complement representation, both suffer from the problem of ‘two zeros’!

Floating Point Numbers in Decimal

- *Range* and *precision* are two important aspects of numbers when we look for a suitable binary representation

- The *range* of a number representation signifies the smallest and the largest numbers which can be represented.

- *Precision* in a number representation is defined in terms of *significant digits* that can be accommodated.

- Take the example of two quantities 0.7291×10^{-24} and 0.7291×10^{24}.

- In both cases the *precision* is signified by the four digits 7291.

- The numerical *range* between these two quantities – from 10^{-24} to 10^{+24} – is signified by the two *exponents* -24 and +24.

- If base 10 and the position of the decimal point are made implicit, then these two numbers can also be written in the form of pairs of numbers, say <7291,-24> and <7291,+24>.

[Decimal point is assumed to be to the left of the digits 7291]

- The first element in each pair contributes to *precision*, while the second number contributes to *range*.

  The first number in the pair is known as *mantissa*, and the second number is known as *exponent*.

- The resulting number representation allows us to achieve the required range as well as precision.
  - Note that this number representation must also provide for negative numbers.

- Known as *Floating Point* number representation
  - Because a change in the exponent value causes the implied decimal point to effectively move left or right within the significant digits contained in the mantissa.

IEEE Floating Point Standard

- International standard for binary floating point number representation, adopted in 1985

- The two standard floating point number representations in common use, known as *single-precision* and *double precision*, make use of 32 and 64 bits respectively.

32 bit floating point number

- The stored part of mantissa has 23 bits, with the implicit ‘binary point’ to the left of it.
- An extra ‘hidden’ bit is also implicitly present to the left of the binary point. In a normalized number, this bit is always ‘1’.
  - Suppose the stored mantissa of a binary floating point number is 111100000..0, i.e. four ‘1’s followed by all ‘0’s.
  - The implicit ‘binary point’ is to its left, so that the decimal numeric value of the stored mantissa is:

    .111100000..0 = $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 0.9375$
  - With an extra hidden ‘1’ to the left of the binary point, the value of mantissa becomes 1.9375.

- The radix (base) in IEEE standard floating point number is 2, and in a single-precision number, 8 bits are used for the exponent.

- Thus the unsigned numerical range of this stored 8-bit stored exponent varies from 0 to 2^8-1 = 255.

- Since negative exponent values are also required, the actual exponent is taken as the stored 8-bit exponent <u>minus 127</u>.

- Stored exponent values of 0 and 255 are reserved for zero and the special ‘infinity’ and ‘not a number’ symbols.

- Therefore the actual exponent value lies in the range from the smallest value 1 - 127 = -126 to the largest value 254 - 127 = +127.

- Since the stored value of e exponent is 127 greater than its arithmetic value, this exponent is said to be in *excess 127* format.

- <u>Example</u>: Represent the number 15.5 in 32 bit floating point:

  Note that 15.5 = 1.9375 x 8 = 1.9375 x 2^3

  Since the actual exponent is 3, the stored value in the 8-bit exponent must be made 127+3 = 130. In binary, this value is 10000010

  We have already seen this value of mantissa.

  Sign bit for a positive number is ‘0’.

  Thus, the full 32-bit floating point number is:

$$\underset{\textbf{sign bit}}{\overset{\mathbf{0}}{\uparrow}}\quad \underbrace{\mathbf{10000010}}_{\textbf{8-bit exponent}}\quad \underbrace{\mathbf{11110000000000000000000}}_{\textbf{23-bit stored mantissa}}$$

- Excluding the sign bit, we have a total of 31 bits in single precision representation.
- Mantissa contributes to precision, while exponent contributes to range.
- The choice of 23-bit mantissa and 8-bit exponent involves a *trade-off* between precision and range
  - i.e. more bits can be assigned to one only at the expense of the other, their sum in single-precision being restricted to 31

- Numerically, the largest possible mantissa is the one in which all bits are ‘1’, and this mantissa has numerical value of almost 2.0, after we add 1 for the hidden bit.

- The largest possible exponent is 254-127 = 127.

- The number $2x2^{127}$ is approximately equal to 10^{38} which is thus the largest number we can represent in single precision.

- The smallest exponent is 1-127 = -126, and correspondingly the smallest normalized number we can represent in single precision is approximately equal to 10^{-38}.

- With 23 bits and one hidden bit in mantissa, single-precision standard provides a precision of approximately $24/\log_2 10$ = seven decimal digits.

- <u>64 bit *double precision format*</u>
  - Has 52 bit mantissa and 11 bit exponent, with the same concepts of normalization, hidden bit, etc.
  - Used in applications requiring higher precision & range.

- Precision of double precision floating point numbers is about 16 decimal digits, while the approximate range is from 10^{-308} to 10^{+308}.

Limits of Representation

- In 32-bits, we can represent 2^{32} distinct integers or real numbers – since that is the total number of distinct bit patterns possible.
- Although 2^{32} is a fairly large number – about four billion – it is finite.
- Thus the infinite sets of integers and real numbers of mathematics cannot be represented in 32 bits.
- In unsigned integer representation, with *k* bits for an integer value, the range of representation is from 0 to 2^{k-1}. Any possible result outside this range generates *overflow*.
- In signed two's complement representation, the range is from -2^{k-1} to $+2^{k-1}-1$, and an out of range result generates *overflow*.

- In the case of floating point number representation, the size of the exponent determines the range of representation.

- For single-precision numbers, the range is approximately 10^{-38} to 10^{+38}, whereas for double-precision numbers, it is approximately 10^{-308} to 10^{+308}.

- Irrational numbers such as π or $\sqrt{3}$, and fractions such as 1/3 – in which the denominator is not a power of 2 – cannot be represented exactly in single or even double precision.

- But, as long as the *round-off error* is acceptable, practical computations can go ahead.

Octal and hexadecimal representation

- For achieving compactness in displaying binary data, it is convenient to define digits which are ‘more expressive’ than the two binary digits ‘1’ and ‘0’.

- If we group three binary digits together, we create an *octal* digit, while if we group four binary digits together, we create a *hexadecimal* digit.

- Let us say the eight-bit binary quantity 11010111 is to be represented in octal as well as hexadecimal.

- For octal representation, <u>starting from the right</u>, we divide the given eight bits into groups of three each, adding an extra ‘0’ to the left, as shown:

$$\underline{0}11\quad 010\quad 111$$

- Thus the given eight bits have octal representation 327.

- For hexadecimal representation, <u>starting from the right</u>, we divide the given eight bits into groups of four bits each, as shown:

1101 0111

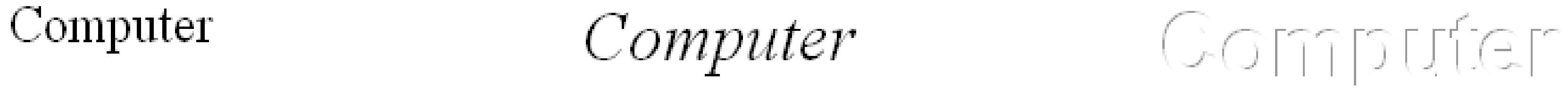
- We encode groups of four bits into sixteen *hexadecimal digits* ‘0’ to ‘9’ and then ‘A’ to ‘F’:

  - 0000 becomes hex digit ‘0’,
  - 0001 becomes hex digit ‘1’, and so on …
  - 1010 becomes hex digit ‘A’ and …
  - 1111 becomes hex digit ‘F’.

- With this encoding, the given eight bits have hexadecimal representation D7.

Characters

- ‘Text’ in any language is the sequence of characters which make up a piece of writing.
- Below we see the word ‘Computer’ displayed in three different combinations of *fonts*, *font sizes* and *font styles*.



Times Roman font, size 12	Arial font, size 16, in italics	Arial font, size 20, with ‘engraved’ appearance

- Set of characters we need to encode in binary:
  - 26 letters of the English alphabet - in upper case and lower case, giving 52 characters
  - Ten decimal digits, ‘space’ (or ‘blank’) character, and various punctuation marks, quotation marks, and three different pairs of ‘open bracket’ and ‘close bracket’ symbols
  - Basic arithmetic and other symbols such as the plus sign, asterisk, ampersand, percent sign, and so on.

- The total number of characters comes to just under a hundred, which means that seven bits are needed to encode this set of characters. (since $2^7 = 128$)

- Certain *control characters* also need to be included with the set to help a computer system to:
    - (a) send and receive data from another system, and
    - (b) control the operations of an output device

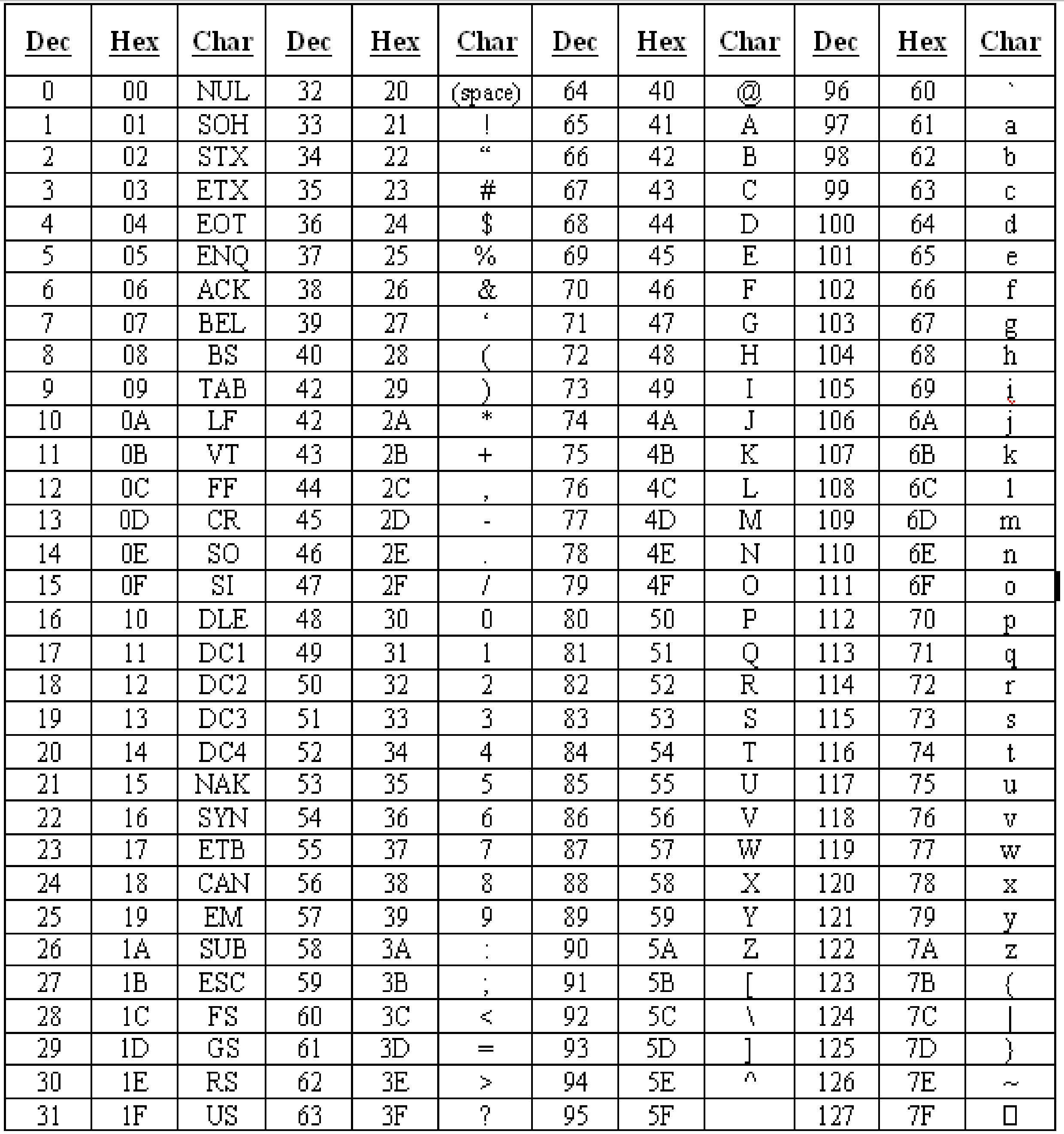
- Taking all this into account, a seven bit character code known as the **American Standard Code for Information Interchange (ASCII)** has been defined since the early years of computer industry.

- The binary value zero is reserved for a character known as *null* having no assigned function.
  - A programmer is free to assign it a role depending on specific needs of the software.
  - In several programming languages, the *null* character is used to indicate the end of a sequence of non-*null* characters.

- The next slide shows the table of all 128 ASCII characters. It has 32 rows and12 columns.

- The columns in the table are arranged in four groups of three columns each.

- In each such group of three columns, the header row has the three headings **Dec**, **Hex** and **Char**. Thus each character is shown in decimal and hex.



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	(space)	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	“	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	‘	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	TAB	42	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F		127	7F	□

- For languages such as Chinese and Japanese, however, an eight bit code does not suffice.
- Number of characters to be represented in these languages runs into thousands
- To accommodate such writing systems, and also many alphabet-based languages such as Hindi and Telugu, a code known as Unicode has been standardized.
- Using two byte code, Unicode can accommodate up to $2^{16} = 65536$ characters.
- There is also provision for extending the code to three bytes.

Audio, Video and Graphics

- To store and process sound, and to send it over a network, sound signal is converted to electronic form and *digitized*.
- i.e. the original sound signal (‘audio’) is converted to a sequence of electronic ‘1’s and ‘0’s. Higher quality of digitized sound has higher storage requirements.
- Video data differs from audio data in the following ways

(a) Visual information may be required either in *static* or in *animated* form – i.e. still pictures or moving pictures, whereas there is no such thing as a *static* sound signal.

(b) One second of digitized video signal requires a much large number of bits to represent – i.e. *encode* – than one second of sound.

Error Detection and Correction

- Electronic signals carrying data are subject to several forms of unwanted disturbances of random nature, which are collectively known as *noise*.
- Due to this noise, some bits in transmitted data may be in *error* when they are received at the other end.
- To reduce probability of errors:
  - Circuits and data paths are shielded from the physical effects of noise, and
  - In binary encoding of data, care is taken so that, at the receiving end, errors in the data may be *detected* and *corrected*

- Assume that you have a message of length *k* bits.
- Another *r* redundancy bits – which are calculated from the *k* message bits – are added to the message.
- Thus total *k*+*r* bits are transmitted. At the receiving end, we can re-check whether the *r* redundancy bits are consistent with the message bits.
- If so, we assume error-free reception; otherwise we have error detection (and also possibly correction)

Summary

- Number systems
- Unsigned integers
- Signed integers
- Floating point numbers
- IEEE Floating Point Standard
- Limits of representation
- Octal and hexadecimal representation
- Characters
- Audio, video and graphics
- Error detection and correction