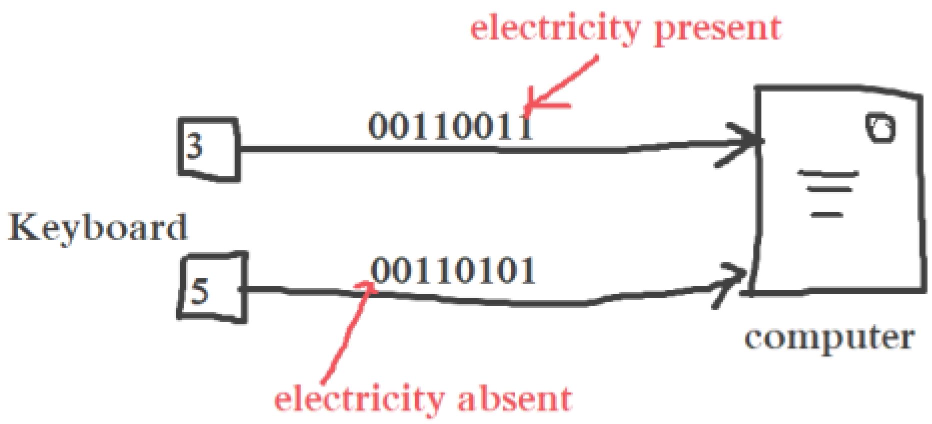
Number systems

- Computer doesn't understand 0,1 it understands the presence and absence of current
- Presence of current - 1, absence of current - 0
- Voltage from 0v - 0.8v -> 0 and 2v - 5v ->1
- 0.8v – 2.0v –>uncertain area



- **American National Standard Institute** proposed (ANSI) to use ASCII
- This was proposed in order for all to use the same standard when representing characters (Requirement of a character set)
- Character set is a standard way to represent characters.
- There are a couple of character representation methods
  - BCD (Binary Coded Decimal)
  - EBCDIC (Extended Binary Coded Decimal Interchange Code)
  - ASCII (American Standard Code for Information Interchange)
  - Unicode

BCD

- **4 bit** representation (`2**4 = 16`)
- total number of characters that can be represented - **16** (`2**4 = 16`)
- used to represent numbers (0-9)
  - 1 -> 0001 | 8 -> 1000
- But for digits with two numbers 8 bits are used.
  - 10 -> 0001 0000 | 12 -> 0001 0010

ASCII

- 8 bit representation, However, the last bit (first one from left) is used as the **check digit**, so the representable bits are **7**
- This last bit is used to check the type of the entered character (whether it's a number, special character, function key or a letter etc.)
- total number of characters that can be represented - **128** (`2**7 = 128`)

- Originally proposed by ANSI
- **IBM personal computers** use ASCII

EBCDIC

- typically used by **IBM mainframe computers**
- **8 bit** representation
- total number of characters that can be represented - **256** (`2**8 = 256`)

Unicode

- 16 bit representation
- total number of characters that can be represented - **65536** (`2**16 = 65536`)

	Advantage	Disadvantage
BCD	• Easy to encode and decode decimals into BCD and vice versa. • Simple to implement a hardware algorithm for the BCD converter. • It is very useful in digital systems whenever decimal information is given either as inputs or displayed as outputs. • Digital voltmeters, frequency converters and digital clocks all use BCD as they display output information in decimal.	• Not space efficient. • Difficult to represent the BCD form in high speed digital computers in arithmetic operations, especially when the size and capacity of their internal registers are restricted or limited. •Require a complex design of Arithmetic and logic Unit (ALU) than the straight Binary number system. •The speed of the arithmetic operations slow due to the complete hardware circuitry involved.
ASCII	• Uses a linear ordering of letters. • Different versions are mostly compatible. • compatible with modern encodings	• Not Standardized. • Not represent world languages.
EBCDIC	• uses 8 bits while ASCII uses 7 before it was extended.	• Does not use a linear ordering of letters.
	• Contained more characters than ASCII.	• Different versions are mostly not compatible. • Not compatible with modern encodings
UNICODE	• Standardized. • Represents most written languages in the world • ASCII has its equivalent within Unicode.	• Need twice memory to store ASCII characters.

Data Representation

There is a special system to represent characters with ASCII

- Representation of characters
  If we press A, the `A` gets the ASCII value for it which is `65` Then it's its converted to binary which is `1000001` and sent to interpret.

`A` -> `65` -> `1000001`

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

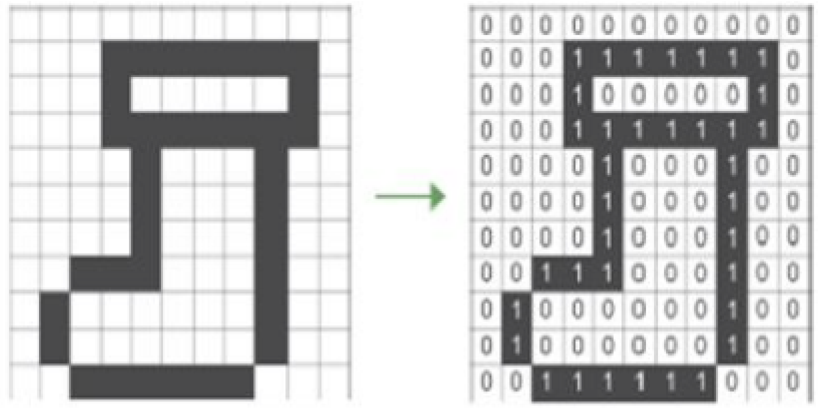
- Representation of Images

First the image is divided into rows and columns and a **bitmap** is made. If 2 colors are used to represent the image (I.e black and white images) black is represented by 1 and white is represented by 0.

Since there are 2 colors only 1 bit is needed `2**1 = 2`

1 -> black
0 -> white

1	0	1	1
0	1	1	1
1	0	1	1
0	1	1	1



If we need to represent 4 colors, 2 bits are needed `2**2=4`

00 -> white
11 -> black
01 -> red
10 -> blue

01	11	00	11
01	11	00	11
11	01	00	10
01	11	00	11

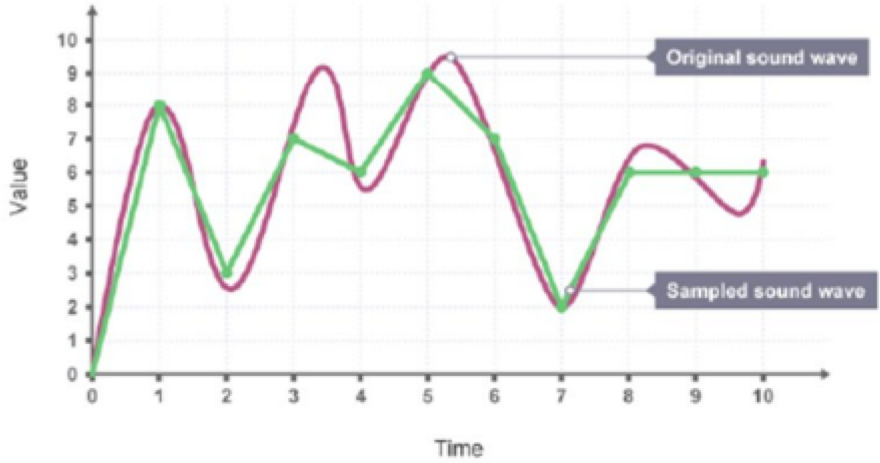
- Representation of Videos

Videos are separated into frames, and then made together with a specific frame size (fps) These frames are represented just like images

- Representation of Audios

Audio is a continues analog signal. Computers can't understand analog signals. So the analog signals are covered to digital signals.

We cannot digitize all the analog values into Digital values. Because Analog signal has an infinite number of values. So, we take sample values then digitize them.



Time sample	1	2	3	4	5	6	7	8	9	10
Decimal	8	3	7	6	9	7	2	6	6	6
Binary	1000	0011	0111	0110	1001	0111	0010	0100	0110	0110

Conversion between fractional numbers

Fractions to binary

- Multiply the given decimal fraction by 2.
  - It's multiplied by 2 because its binary, if octal multiply by 8,if hexadecimal by 16
- Multiply by 2 until the decimal part becomes 0.
- Write the values in front of decimal point from top to bottom.

E.g.:- convert 0.3125_{10} to binary

	0.3125	x2
0	.625	x2
1	.25	x2
0	.50	x2
1	.00	

$0.3125_{10} = 0.0101_2$

```
0.3125 --> 0.0101
0.625 --> 0.101
0.25 --> 0.01
0.50 --> 0.1
```

5.3125_{10}

$.3125_{10}$ to binary

101

$.3125 \times 2$

$0.3125_{10} = 0.0101_2$

$.6250 \times 2$

1.2500×2

0.5000×2

1.0000

101.0101_2

Same theory goes for octal and hex

- Octal

Converting fractions to Octal

- Multiply the given decimal fraction by 8.
- Multiply the decimal by 8 until it becomes 0.
- Write from the beginning to end, the values in front of the decimal point.

E.g. :- convert 0.3125_{10} to binary

0	0.3125	x8
2	.50	x8
4	.0	x8

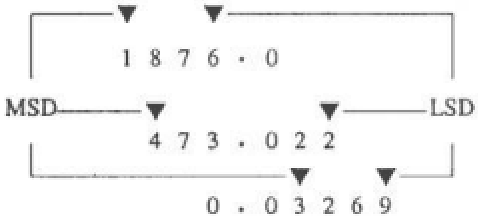
$0.3125_{10} = 0.24_8$

02 - Most Significant Digit (MSD) and Least Significant Digit (LSD)

Most Significant Digit (MSD) and Least Significant Digit (LSD)

- **MSD - The Digit that contain the most positional value in a number.**
- **LSD - The Digit that contains the least positional value in a number.**

Number	MSD	LSD
2975.0	2	5
56.034	5	4
0.03145	3	5
0031.0060	3	6



With binary or octal or hex, you need to get the position of the MSB and LSB and then raise to power of the position

100100
-> MSB = `2**6` (1 in the left-hand side in the 6th position)
-> LSB = `2**0` (0 in the right-hand side in the 0th position)

Here, 0 is considered because if another 0 is added in the end, the value of the number changes. But with decimal numbers (32.41) this doesn't matter. Even if we add another 0 at the end, the value **doesn't change**

Octal and hexadecimal number systems are there for **human convenience**. This helps to compress data and make it short so that's easy to read and interpret.

Signed Integers

To **represent negative numbers, these signed integers are used**. There are 3 ways to do this.

1. Signed Magnitude Representation
2. 1's Complement
3. 2's Complement

Signed Magnitude Representation

The left most bit is used as the singed bit

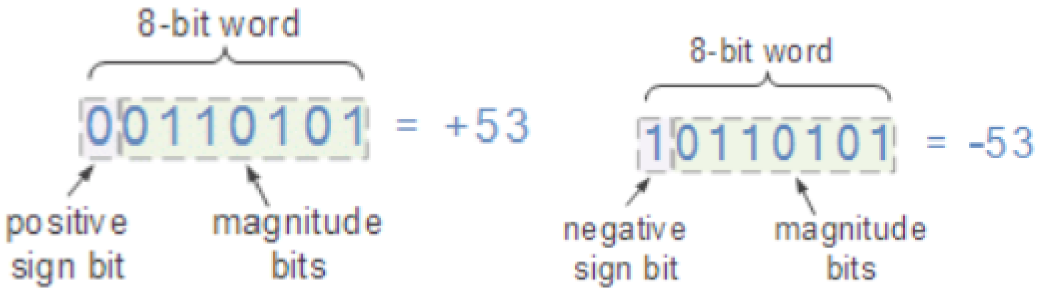
- Used leftmost bit for the sign.

Mathematical representation	Binary representation
3	0011
-3	1011

Here the 0 is left last bit tells that the number is a positive number. If it's 1 its a negative

0 -> positive
1 -> negative

The conventional bit length is 8 (ASCII format)



If its a 4 bit computer, we can have 16 possibilities, But here we don't represent numbers from 0-15. Here 7 bits are given for positive numbers and 7 bits for negative. And other 2 numbers are for 2 zeros.

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

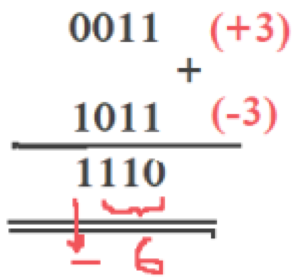
**** If you want to extend this range you need more bits.

Problems of sign magnitude

- One problem in this is that **it has 2 zeros**. A +0 (0000) and a -0 (1000) This is mathematically wrong.
- **Subtraction (other calculations too) of negative values can't be done**. The computer can't do other calculations other than addition (that's why its called adding machine). Every other calculation like -, * and % is done by addition

To do all 4 mathematical operations in decimal number system, can do by using adder.
Ex:
3+5 = 8
5-3 =2 → 5+(-3) = 2
5*3 =15 → 5+5+5 =15
15/3=5→15-5-5-5=5

A problem arises when we try to add a positive number to a negative number



-3 + 3 should be 0 but here it's giving -6

1's Complement

This affects to negative numbers. Here we flip the numbers for the negative numbers
For a 1 we use 0 and for a 0 we use 1

1011 is -3, since its a negative we do a 1's Complement to the positive of it (3). Even though we are doing the 1's compliment to the -3 we do the flip to the positive value of the digit which in this case is 3. (if we wanted to do 1's compliment to -5 we use 5 (0101) and then flip it `0101 -> 1010`)

```
3 = 0011 -> do 1's Complement -> 1100
```

Now we do the calculation

```
3 + (-3) = 0011 + 1100 (this is the value for doing 1's complement for 3)

0011 + 1100 = 0
```

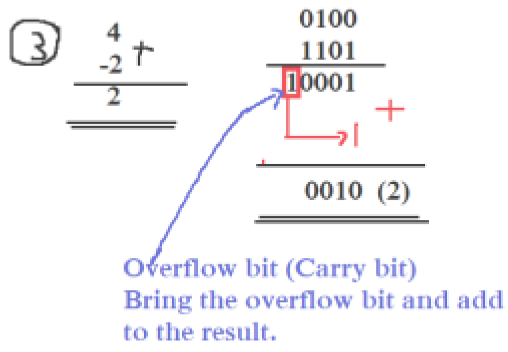
- I.e 3+ (- 5)
  1. get binary for 3 and 5
     - `3 = 0011, 5 = 0101`
  2. do 1's complement for 5 since it's a negative used.
     - `0101 -> 1010 ; (-5) -> 1010 (not the real value for -5)`
  3. Then do the calculation
     - `0011 + 1010 = 1101`

But here is `1101 = -2`? No! If the answer is a negative, first we need to do 1's complement to the value and flip it. Then its converted to decimal

```
1101 -> 0010 = 2
```

- Problems with 1's compliments
  - We still have the 2 zeros problem
- The second problem we had is solved though. We now can do calculations with negative numbers

If the answer overflows the max number of bits, the overflowing bit is called the **Carry bit**. We need to add this carry bit to the answer itself (LSB of the answer - last bit) to get the accurate answer.



2's Complement

This is **done to negative numbers** too. Here the same process happens like the 1's complement. The difference is we have to add 1 after the 1's complement is done.

2's complement for -3

```
3 = 0011 -> 1's complement -> 1100
```

Add one to the answer of 1's complement

```
1100 + 0001 = 1101 = -3 (This can't be reverted like 1's complement)
```

We can confirm our answer from this table

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

- In 4-bit computer, we can have 2^4 =16 combinations.
- 1 for 0 other 15 for other numbers.
- 15 cannot divide into same parts.

So here, no +8, only have +7 to -8 including one 0.

If you want to represent +8 then you have to increase the number of bits

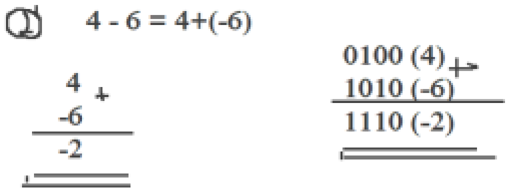
+8 can't be represented with 4 bits according to this, you have to use 5 bits to represent that.

Now that we have that, if this is correct, if we add +3 to -3 it should result in 0 ryt. Let's see.

"/IT/Images/Pasted image 20220910122102.png|300" could not be found.

No here one bit get's overflown. Therefore, we just discard that bit. (In 1's compliment we add it to the LSB, here we just discard it) So as the final answer, we get `0000` which is 0

Let's see another example of `4 - 6`



Here we get the answer as `1110` but this is not `-2` So what we have to do to get the correct decimal value is as follows.
1. See what the sign bit is.
2. if its `0` (positive answer) then no problem, keep it as same and convert to decimal.
3. If it's `1` (negative answer), flip the bits `1110 -> 0001` and then add one to the end (LSB) `0001 + 0001 = 0010`
4. Get the final answer as `0010` which is the binary equivalent to `2` and if the sign bit was `0`, no problem, keep it as it is. But if it's `1` the answer is negative. (The sing checking it done to the value we get before flipping. In this case for `1110`)
5. Since the check bit of the answer is `1` the answer is negative which is `-2`

NOTE: One thing to remember here is that, we **can't subtract** values from the answer. So we **can't subtract 1 from the answer and then do the flip**. What we have to do is that first we need to **do the flip and then add 1 to the answer**

```
answer - 1 -> do the flip ------- WRONG

do the flip -> answer + 1 ------- Correct
```

Nonetherless the answer with both ways is going to be the same but the first method is wrong!

Here since the sign bit is `1`, the answer becomes negative which is `-2`

Advantages of 2's compliment
- Operations are simpler.
- 2 zero problem is gone.
- In modern computers, this method is mostly used.
- Makes it possible to build low cost, high speed hardware

Usage of sign magnitude, 1's complement and 2's complement

	Usage
Sign Magnitude	Used only when we do not add or subtract the data. They are used in analog to digital conversions. They have limited use as they require complicated arithmetic circuits.
One's Complement	Simpler design in hardware due to simpler concept.
Two's Complement	Makes it possible to build low-cost, high-speed hardware to perform arithmetic operations.

Maximum and Minimum values of these encodings (in 8 bit representation)

Encoding	Min	Max
1's Complement	0	127
2's Complement	0	128

Uses basic arithmetic and logic operations on binary numbers

NOT

1. NOT operation

A	NOT A
0	1
1	0

E.g. :- **NOT** 0111_2 (7_{10}) = 1000_2 (8_{10})

Here only one bit stream is needed to do the operation

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

E.g. :- 0101_2 (5_{10}) **AND** 0011_2 (3_{10})

$$\begin{array}{r} 0\,1\,0\,1_2 \\ \underline{0\,0\,1\,1_2} \\ 0\,0\,0\,1_2\ (1_{10}) \end{array}$$

Therefore 0101_2 **AND** 0011_2 is 0001_2

With AND (and all the other operations) 2 bit streams are used.

OR

3. Bitwise OR operation

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

E.g. :- 0101_2 (5_{10}) **OR** 0011_2 (3_{10})

$$\begin{array}{r} 0\,1\,0\,1_2 \\ \underline{0\,0\,1\,1_2} \\ 0\,1\,1\,1_2\ (7_{10}) \end{array}$$

Therefore 0101_2 OR 0011_2 is 0111_2

XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

E.g. :- 0010_2 (2_{10}) **XOR** 1010_2 (10_{10})

$$\begin{array}{r} 1\,0\,1\,0_2 \\ \underline{0\,0\,1\,0_2} \\ = 1\,0\,0\,0_2\ (8_{10}) \end{array}$$

Therefore 0010_2 **XOR** 1010_2 is 1000_2

If the same inputs are given, the answer is `0`. If different inputs are given, the answer is `1`