Computer Fundamentals

Lecture ١٠

Data Representation

Negative Binary Numbers

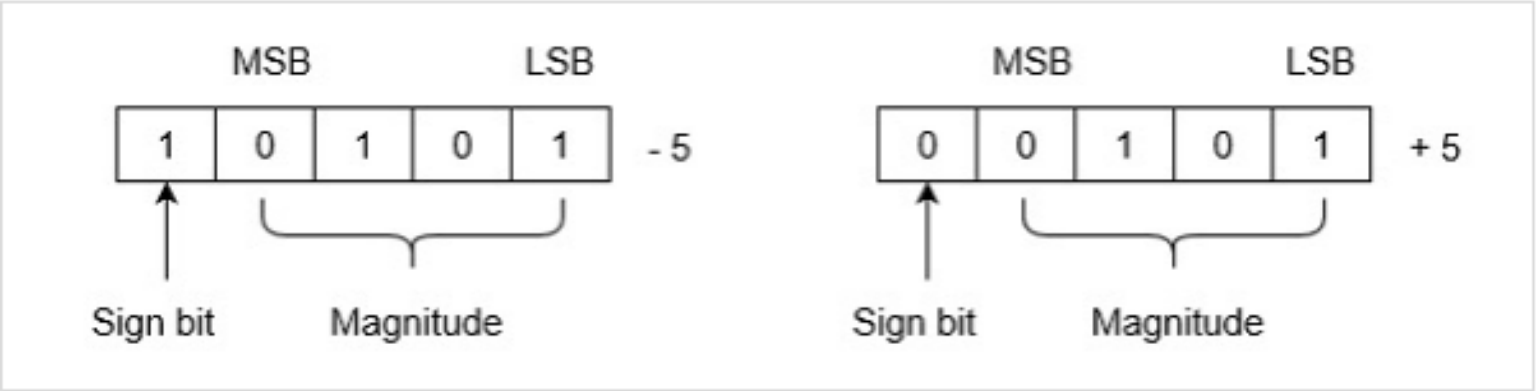
- To store negative numbers, we need a bit to indicate the sign
  - 0 = positive, 1 = negative
- It is not possible to add minus or plus symbol in front of a binary number
- number may be represented in one of three possible ways:
  - Sign-Magnitude method
  - 1’s Complement method
  - 2’s complement method

Sign-Magnitude method

- In this method, number is divided into two parts: Sign bit and Magnitude.
- If the number is positive then sign bit will be **0** and if number is negative then sign bit will be **1**.
- Magnitude is represented with the binary form of the number to be represented.

Sign-Magnitude method - Example

- **Example:** Let we are using 5 bits register. The representation of -5 to +5 will be as follows:



Sign-Magnitude method - Example

- Convert the following decimal values into signed binary numbers using the sign-magnitude format:

-15_{10} as a 6-bit number ⇒ $١٠١١١١_{٢}$

$+23_{10}$ as a 6-bit number ⇒ $٠١٠١١١_{٢}$

-56_{10} as a 8-bit number ⇒ $١٠١١١٠٠٠_{٢}$

$+85_{10}$ as a 8-bit number ⇒ $٠١٠١٠١٠١_{٢}$

-127_{10} as a 8-bit number ⇒ $١١١١١١١١_{٢}$

Sign-Magnitude method (cont'd)

- **Range of Numbers:** For k bits register, MSB will be sign bit and (k-1) bits will be magnitude. Positive largest number that can be stored is $(2^{(k-1)}-1)$ and negative lowest number that can be stored is $-(2^{(k-1)}-1)$.

- For example: if we have **4 bits** to represent a signed binary number, (1-bit for the Sign bit and 3-bits for the Magnitude bits), then the actual range of numbers we can represent in sign-magnitude notation would be:

$$-2^{(4-1)} - 1 \quad \text{to} \quad +2^{(4-1)} - 1$$

$$-2^{(3)} - 1 \quad \text{to} \quad +2^{(3)} - 1$$

$$-7 \quad \text{to} \quad +7$$

- **Note** that drawback of this system is that 0 has two different representation one is -0 (e.g., 1000 in four bit register) and second is +0 (e.g., 0000 in four bit register).

2’s Complement Method

- Positive numbers are the same as in our previous approach
- Negative numbers need to be converted, to do this:
  - NOT (flip) all of the bits (1 becomes 0, 0 becomes 1)
  - Add 1
  - Example: -57 in 8 bits
    - +57 = 32 + 16 + 8 + 1 = **00111001**
    - -57 = NOT(00111001) + 1 = 11000110 + 1 = **11000111**

2’s Complement Method - Examples

- **Convert -15 to binary (as 1 byte):**
  - the number +15 will be:
  - 8 + 4 + 2 + 1 = **00001111** (as 1 byte)

  We flip all the bits::
  - Not(**00001111**) = **11110000**

  Then add 1

  11110000 + 1 = **11110001**

-15_{10} = 11110001_2

2’s Complement Method - Examples

- **Convert -48 to binary (as 1 byte):**
  - +48 = 32 + 16 = **00110000**
  - Not(**00110000**) = **11001111**
  - **11001111** + 1 = **11010000**
  - Then: –48 = **11010000**

2’s Complement Method - Range

- **Range of Numbers:** For k bits register:
- positive largest number that can be stored is $(2^{(k-1)}-1)$
- negative lowest number that can be stored is $-(2^{(k-1)})$.

- The method of 2’s complement arithmetic is commonly used in computers to handle negative numbers

Character Representations

Character Representations

- We need to invent a represent to store letters of the alphabet (there is no natural way)
- The codes that represent numbers, alphabetic letters and special symbols are called alphanumeric codes.
- A complete set of necessary characters include :
  - 26 lower case letters (a to z)
  - 26 upper case letters (A to Z)
  - 10 numeric digits (0 to 9).
  - about 25 special characters such as /, #, $ ….. etc.
- These total up to about 87 symbols.
- To represent these 87 symbols with some binary code will require at least 7-bits.

Character Representations

- Three character codes have been developed
  - **EBCDIC** – (Pronounced **ebsa-dik)** used by IBM mainframes
  - **ASCII** – the most common code, 7 bits – 128 different characters (add a 0 to the front to make it 8 bits or 1 byte per character)
  - **Unicode** – expands ASCII to 16 bits to represent over 65,000 characters

EBCDIC

- EBCDIC - **Extended Binary-Coded Decimal Interchange Code**
- Pronounced **ebsa-dik**
- It was one of the first widely-used computer codes that supported upper and lowercase alphabetic characters, in addition to special characters.
- EBCDIC used in IBM equipment.
- It uses 8 bits for each character.
  - contains 256 different combinations.

- **It has not been popular.**

ASCII Code

- ASCII (American National Standard Code for Information Interchange), pronounced **ask-ee**.
- In ASCII, each character (letter, number, symbol or control character) is represented by a binary value.
- ASCII encoding maps each character to 1 byte with the leading bit set to 0, and other 7 bits representing the code point of the character.
  - which gives us 128 different combinations.
  - With the 52 letters, 10 digits, and various symbol, there is room left over for other characters.

ASCII Code

- For example, in ASCII (Unicode, and many other character sets):
  - code numbers 65D (41H) to 90D (5AH) represents 'A' to 'Z', respectively.
  - code numbers 97D (61H) to 122D (7AH) represents 'a' to 'z', respectively.
  - code numbers 48D (30H) to 57D (39H) represents '0' to '9', respectively.

ASCII Code - Control Characters

- The first 32 characters of ASCII table are used for control
- Control character codes = 00 to 1F (hex)
- Examples of Control Characters
  - Character 0 is the NULL character ⇒ used to terminate a string
  - Character 9 is the Horizontal Tab (HT) character
- One control character appears at end of ASCII table
  - Character 7F (hex) is the Delete (DEL) character

ASCII Code - Control Characters

DEC	HEX	Meaning	
0	00	NUL	Null
1	01	SOH	Start of Heading
2	02	STX	Start of Text
3	03	ETX	End of Text
4	04	EOT	End of Transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledgment
7	07	BEL	Bell
8	08	BS	Back Space '\b'
9	09	HT	Horizontal Tab '\t'
10	0A	LF	Line Feed '\n'
11	0B	VT	Vertical Feed
12	0C	FF	Form Feed 'f'
13	0D	CR	Carriage Return '\r'
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Datalink Escape

DEC	HEX	Meaning	
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Negative Ack.
22	16	SYN	Sync. Idle
23	17	ETB	End of Transmission
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	IS4	File Separator
29	1D	IS3	Group Separator
30	1E	IS2	Record Separator
31	1F	IS1	Unit Separator
127	7F	DEL	Delete

ASCII printable characters

DEC	HEX	BIN	Symbol
48	30	00110000	**0**
49	31	00110001	**1**
50	32	00110010	**2**
51	33	00110011	**3**
52	34	00110100	**4**
53	35	00110101	**5**
54	36	00110110	**6**
55	37	00110111	**7**
56	38	00111000	**8**
57	39	00111001	**9**

DEC	HEX	BIN	Symbol
32	20	00100000	
33	21	00100001	**!**
34	22	00100010	**"**
35	23	00100011	**#**
36	24	00100100	**$**
37	25	00100101	**%**
38	26	00100110	**&**
39	27	00100111	**'**
40	28	00101000	**(**
41	29	00101001	**)**
42	2A	00101010	*****
43	2B	00101011	**+**
44	2C	00101100	**,**
45	2D	00101101	**-**
46	2E	00101110	**.**
47	2F	00101111	**/**

DEC	HEX	BIN	Symbol
65	41	01000001	**A**
66	42	01000010	**B**
67	43	01000011	**C**
68	44	01000100	**D**
69	45	01000101	**E**
70	46	01000110	**F**
71	47	01000111	**G**
72	48	01001000	**H**
73	49	01001001	**I**
74	4A	01001010	**J**
75	4B	01001011	**K**
76	4C	01001100	**L**
77	4D	01001101	**M**
78	4E	01001110	**N**
79	4F	01001111	**O**
80	50	01010000	**P**
81	51	01010001	**Q**
82	52	01010010	**R**
83	53	01010011	**S**
84	54	01010100	**T**
85	55	01010101	**U**
86	56	01010110	**V**
87	57	01010111	**W**
88	58	01011000	**X**
89	59	01011001	**Y**
90	5A	01011010	**Z**

DEC	HEX	BIN	Symbol
97	61	01100001	**a**
98	62	01100010	**b**
99	63	01100011	**c**
100	64	01100100	**d**
101	65	01100101	**e**
102	66	01100110	**f**
103	67	01100111	**g**
104	68	01101000	**h**
105	69	01101001	**i**
106	6A	01101010	**j**
107	6B	01101011	**k**
108	6C	01101100	**l**
109	6D	01101101	**m**
110	6E	01101110	**n**
111	6F	01101111	**o**
112	70	01110000	**p**
113	71	01110001	**q**
114	72	01110010	**r**
115	73	01110011	**s**
116	74	01110100	**t**
117	75	01110101	**u**
118	76	01110110	**v**
119	77	01110111	**w**
120	78	01111000	**x**
121	79	01111001	**y**
122	7A	01111010	**z**

Extended ASCII Code

- ASCII was extended to 8 bits to represent $2^8 = 256$ characters.
- The extended ASCII codes are the codes used in most of the microcomputers.
- Extended ASCII code holds some of the special characters and this extended set will represent character codes form 128 - 255.

Extended ASCII Code - Example

DEC	HEX	BIN	Symbol
128	80	10000000	€
130	82	10000010	‚
131	83	10000011	*ƒ*
132	84	10000100	„
133	85	10000101	…
134	86	10000110	†
135	87	10000111	‡
136	88	10001000	ˆ
137	89	10001001	‰
138	8A	10001010	Š
139	8B	10001011	‹
140	8C	10001100	Œ
142	8E	10001110	Ž
145	91	10010001	‘
146	92	10010010	’
149	95	10010101	•

ASCII and EBCDIC

- Neither ASCII nor EBCDIC has any representations for letters outside of the English alphabet.

UNICODE

- **Unicode** is also known as the **universal character encoding standard.**
- Using ASCII, we can only represent the basic English characters, but with the help of Unicode, we can represent characters from all languages around the World.
  - Currently: 21 bits code space
  - How many diff. characters?
- In UNICODE, the first 128 characters are the same as in ASCII.
- The Unicode standard now encompasses 144,076 characters as of version 13.1.
  - It includes emoji, currency symbols, mathematical symbols as well as characters used in almost every language on the planet.

The Unicode Character Sets

- Encoding forms:
  - UTF-8: each Unicode character represented as 1, 2, 3, or 4 bytes
  - UTF-16: one or two 16-bit code units
  - UTF-32: a single 32-bit code unit

* UTF = Unicode Transformation Format

UTF-8 encoding

You can think of unicode as a **map of every possible character to a number**. A character in the broader sense, e.g. an emoji is also a character.

Examples :

A = 65

µ = 181

ż = 380

╬ = 9820

北 = 20000

☠ = 9760

😄 = 128516

UTF-8 encoding

Now you think, wait, I know how to transform a number into binary:

A = **65** = **1100101**

😄 **= 128516 = 11111011000000100**

The problem becomes evident if you try to put together two characters:

A😄 = **65128516** = **11001011111101100000100**

↑ ↑

here the new character starts,

but it could also be read as one big number

instead of two numbers.

So we need a way to separate two numbers.

UTF-8 encoding

. Let's encode A☠😄 in UTF-8!

```
01000001 11100010 10011000 10100000 11110000 10011111 10011000 10000100
|______| |_____________________________| |_____________________________________|
   |                    |                                |
   A                    ☠                                😄
```

```
01000001110001010011000101000001111000010011111100110001000010000100
```

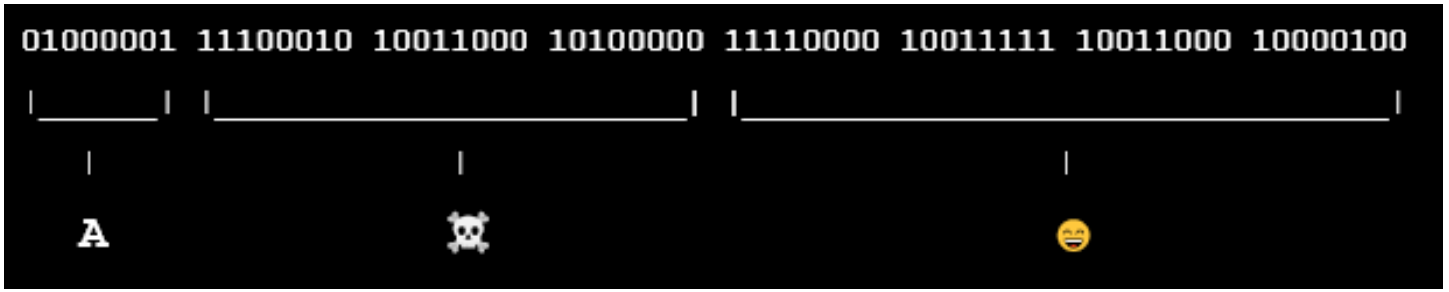
How does the computer know where one character stops and the next starts?

UTF-8 encoding

The trick is: Not all bits are used to encode the actual characters, some bits are used to encode how many **bytes belong** to a character instead!

Those “header bits” are **all those up to the first zero in each byte** and they are to be read like this:

- **0** means the entire char is contained in **one byte**.
- **110** means there are **two bytes** that belong to this character, so the one where `110` is found and the next one.
- **1110** means there are **three bytes** that belong to this character, so the one where `1110` is found and the next two.
- **11110** means there are **four bytes** that belong to this character, so the one where `11110` is found and the next three.

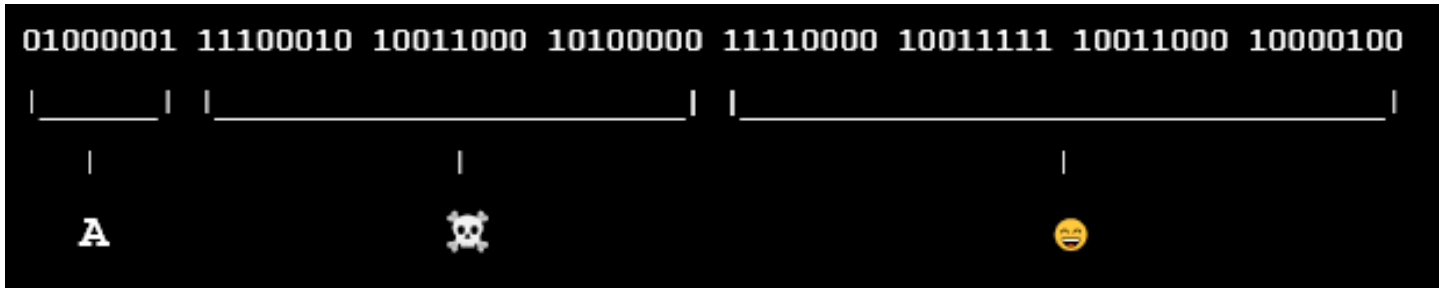


UTF-8 encoding

Bits of code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4
7	1	0xxxxxxx			
11	2	110xxxxx	10xxxxxx		
16	3	1110xxxx	10xxxxxx	10xxxxxx	
21	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

one byte for ascii **characters** (the first 128 **unicode** values are the same as asci
UTF-8 is capable of encoding all **1,112,064** valid character code points in Unicode using one to four one-byte (8-bit) code units

UTF-8 encoding



- A: `01000001` => `0` means everything in **one byte**, so after dropping the `0` we get: `1000001 = 65 = A`
- ☠: We continue reading and find `1110`. This means there are **three** bytes that belong to this character so let's collect them: `11100010 10011000 10100000`. Now that we have those, let's drop all the header bits: `0010 011000 100000 = 9760 =` ☠
- 😁: After having read in all those bits, we're now arriving at the next byte: `11110000`. looks like a **four** byte character since the header is `11110`! So we'll collect the bytes: `11110000 10011111 10011000 10000100`, then drop the header bits and get: `000 011111 011000 000100 = 128516 =` 😁