CS 발표 - 2단원 데이터

Chapter 02 - 데이터

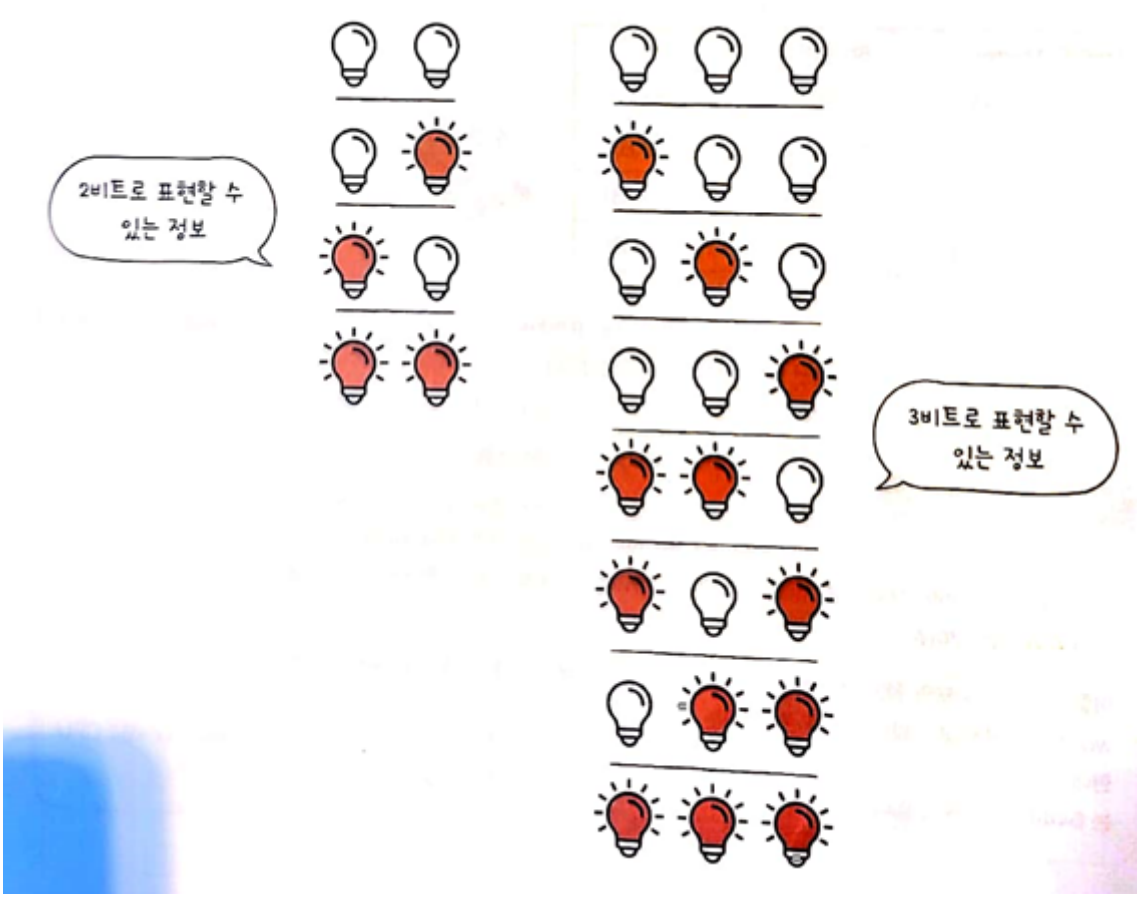
- 컴퓨터는 사람과 달리 모든 것을 0과 1로 표현한다.
- 이번 장에서는 0과 1로 데이터를 표현하는 방법에 대해 학습 (숫자, 문자)

◆ 0과 1로 숫자를 표현하는 방법

◎ 정보단위

- 컴퓨터가 이해하는 가장 작은 정보 단위로 0과 1을 나타내는 **비트(bit)**가 있다.
- **1비트**는 0 또는 1, 두 가지 정보를 표현 가능

◎ (문제) 그럼 2비트랑 3비트는 몇 개의 정보를 표현할 수 있을까?



→ 총 4개와 8개

- **즉 n비트는 2^n가지 정보를 표현 가능**

◎ 바이트, 킬로, 메가, 기가, 테라

- **바이트** : 여덟 개의 비트를 묶은 단위, 비트보다 한 단계 큰 단위
- **1바이트**는 8비트와 같으니 2^8(256)개의 정보 표현 가능
- 1바이트 1000개 묶은 단위 ⇒ **1 킬로바이트 (1kb)**

1바이트(1byte)	8비트 (8bit)
1킬로바이트(1kB)	1000바이트 (1000byte)

1메가바이트(1MB)	1000킬로바이트(1000kB)
1기가바이트(1GB)	1000메가바이트(1000MB)
1테라바이트(1TB)	1000기가바이트(1000GB)

바이트 크기 v • d • e • h					
SI 접두어		전통적 용법		이진 접두어	
기호(이름)	값	기호	값	기호(이름)	V값
kB (킬로바이트)	$1000^1 = 10^3$	KB	$1024^1 = 2^{10}$	KiB (키비바이트)	2^{10}
MB (메가바이트)	$1000^2 = 10^6$	MB	$1024^2 = 2^{20}$	MiB (메비바이트)	2^{20}
GB (기가바이트)	$1000^3 = 10^9$	GB	$1024^3 = 2^{30}$	GiB (기비바이트)	2^{30}
TB (테라바이트)	$1000^4 = 10^{12}$	TB	$1024^4 = 2^{40}$	TiB (테비바이트)	2^{40}
PB (**페타바이트**)	$1000^5 = 10^{15}$	PB	$1024^5 = 2^{50}$	PiB (페비바이트)	2^{50}
EB (엑사바이트)	$1000^6 = 10^{18}$	EB	$1024^6 = 2^{60}$	EiB (엑스비바이트)	2^{60}
ZB (제타바이트)	$1000^7 = 10^{21}$	ZB	$1024^7 = 2^{70}$	ZiB (제비바이트)	2^{70}
YB (요타바이트)	$1000^8 = 10^{24}$	YB	$1024^8 = 2^{80}$	YiB (요비바이트)	2^{80}

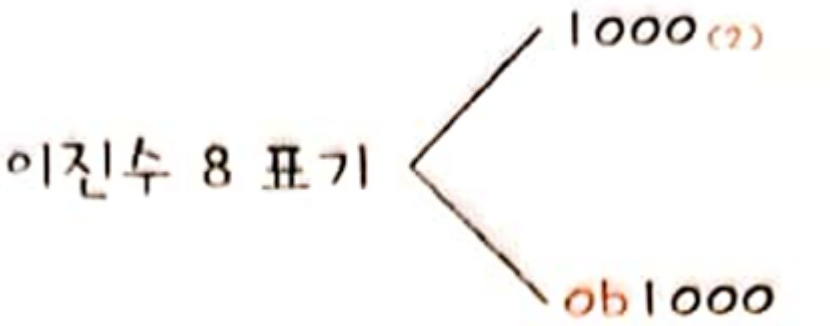
◎ 이진법

- 수학에서 0과 1만으로 모든 숫자를 표현하는 방법
- 우리는 일상적으로 십진법을 쓴다.
- 이진법으로 표현한 수를 **이진수**, 십진법으로 표현한 수를 **십진수**
- 0과 1 밖에 모르는 컴퓨터에 어떤 숫자를 알려주려면 십진수가 아닌 **이진수**로 알려줘야 한다.

- 십진수 8을 컴퓨터에게 알려주려면? ⇒ 이진수 1000

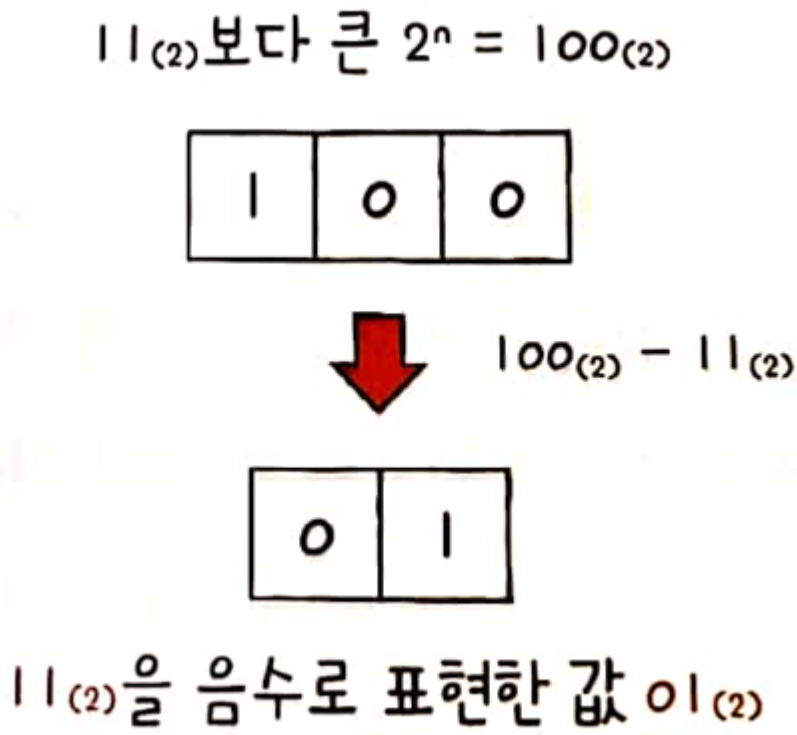
◎ 그런데 숫자 10을 십진수로 읽는지, 이진수로 읽는지, 어떤 진법으로 표현된 수인지 알 수 없다.

- 이러한 혼동 예방을 위해 이진수 끝에 **아래첨자(2)** 혹은 이진수 앞에 **0b**를 붙임



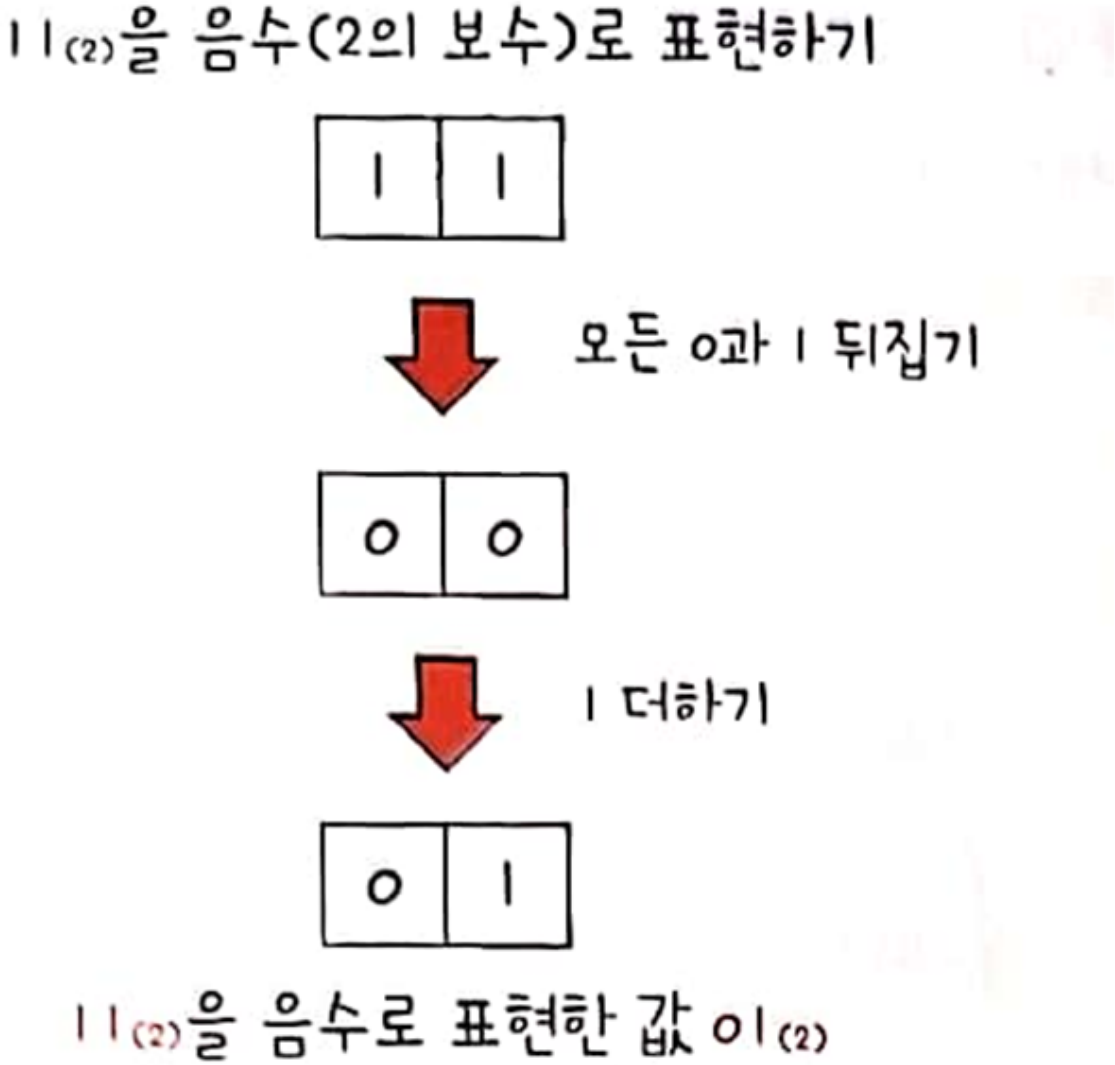
◎ 이진수의 음수 표현

- 십진수는 음수 표현할 때 숫자 앞에 마이너스 부호만 붙이면 됨
- 그렇다면 이진수는? ⇒ **2의 보수**
- **2의 보수** : 어떤 수를 그보다 큰 2^n에서 뺀 값



◎ 하지만 이렇게 하면 헷갈린다. 쉽게 풀자

- **2의 보수** : 모든 0과 1을 뒤집고, 거기에 1을 더한 값



◎ 그렇다면 $-1011_{(2)}$을 표현하기 위한 음수로서의 $0101_{(2)}$와 십진수 5를 표현하기 위한 양수로서의 $0101_{(2)}$을 어떻게 구별하나?

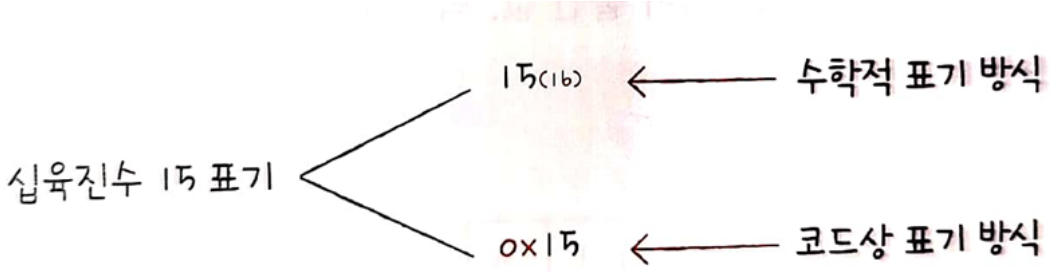
→ **플래그**를 사용한다.

- 컴퓨터 내부에서 어떤 값을 다룰 때 부가 정보가 필요한 경우 **플래그**를 사용
- 04장에서 자세하게 다룸 (p105 ~ 106) - 연산 결과에 대한 추가적인 상태 정보

플래그 종류	의미	사용 예시
부호 플래그	연산한 결과의 부호를 나타낸다.	부호 플래그가 1일 경우 계산 결과는 음수, 0일 경우 계산 결과는 양수를 의미한다.
제로 플래그	연산 결과가 0인지 여부를 나타낸다.	제로 플래그가 1일 경우 연산 결과는 0, 0일 경우 연산 결과는 0이 아님을 의미한다.
캐리 플래그	연산 결과 올림수나 빌림수가 발생했는지를 나타낸다.	캐리 플래그가 1일 경우 올림수나 빌림수가 발생했음을 의미하고, 0일 경우 발생하지 않았음을 의미한다.
오버플로우 플래그	오버플로우가 발생했는지를 나타낸다.	오버플로우 플래그가 1일 경우 오버플로우가 발생했음을 의미하고, 0일 경우 발생하지 않았음을 의미한다.
인터럽트 플래그	인터럽트가 가능한지를 나타낸다. 인터럽트는 04-3절에서 설명한다.	인터럽트 플래그가 1일 경우 인터럽트가 가능함을 의미하고, 0일 경우 인터럽트가 불가능함을 의미한다.
슈퍼바이저 플래그	커널 모드로 실행 중인지, 사용자 모드로 실행 중인지를 나타낸다. 커널 모드와 사용자 모드는 09장에서 설명한다.	슈퍼바이저 플래그가 1일 경우 커널 모드로 실행 중임을 의미하고, 0일 경우 사용자 모드로 실행 중임을 의미한다.

◎ 십육진법

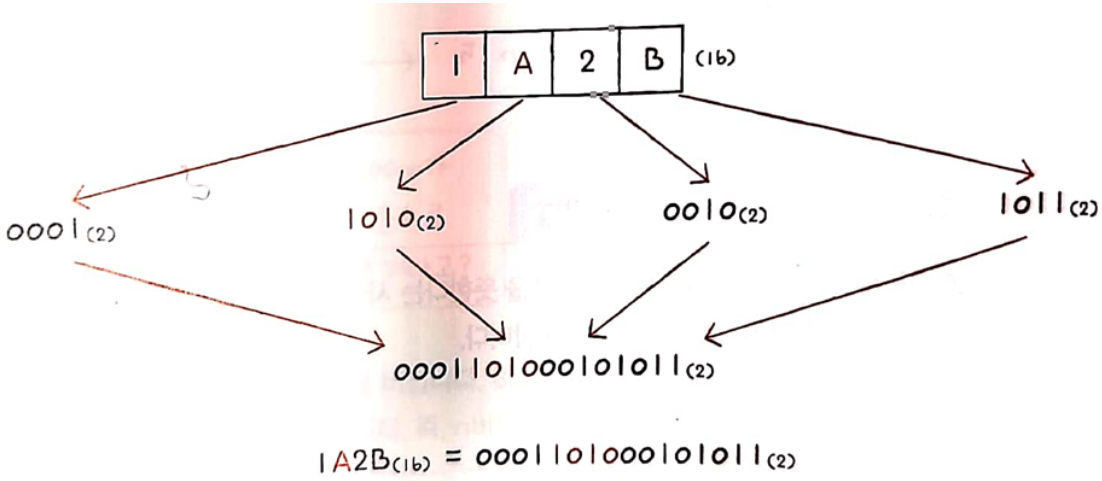
- 데이터를 표현할 때 이진법 이외에도 **십육진법**도 자주 사용
- **십육진법** - 수가 15를 넘어가는 시점에 자리 올림을 하는 숫자 표현 방식
- 십진수 10, 11, 12, 13, 14, 15를 각각 A, B, C, D, E, F로 표기
- 아까 십진수와 이진수 구분할 때처럼, 십육진수도 표기 방식이 있다.



- 그럼 왜 십육진법을 사용할까?
- 이진수를 십육진수로, 십육진수를 이진수로 변환하기 쉽기 때문

◎ 십육진수를 이진수로 변환하기

- 십육진수를 이진수로 표현하려면 몇 비트가 필요할까? 4비트 (2^4)



◎ 이진수를 십육진수로 변환하기

- 반대로 변환할 때는 이진수 숫자를 네 개씩 끊고, 끊어준 네 개의 숫자를 하나의 십육진수로 변환한 후 이어붙이면 된다.

✚ 여기서 잠깐 **코드에 십육진수를 직접 넣는 사례**

십육진수는 프로그래밍할 때 이진수와 더불어 자주 사용되므로 기억해 두는 것이 좋습니다. 하드웨어와 밀접하게 맞닿아 있는 개발 분야에서는 아래와 같이 코드에 십육진수를 직접 쓰는 경우도 더러 있기 때문입니다.

```
offset = __mem_to_opcode_arm(*(u32 *)loc);
offset = (offset & 0x00ffffff) << 2;
if (offset & 0x02000000)
offset -= 0x04000000;
offset += sym->st_value - loc;
```

위 코드가 무슨 뜻인지 알 필요는 없습니다. 프로그래밍할 때 십육진수를 직접 써넣는 경우도 많다는 사실만 인지하면 됩니다.

◎ 5가지 키워드 핵심 포인트

- 비트 : 0과 1로 표현할 수 있는 가장 작은 정보 단위
- 바이트, 킬로, 메가, 기가, 테라바이트는 비트보다 더 큰 정보 단위
- 이진법은 1을 넘어가는 시점에 자리 올림을 하여 0과 1만으로 수 표현하는 방법
- 이진법에서 음수는 2의 보수로 표현
- 십육진법은 15를 넘어가는 시점에 자리 올림하여 수를 표현하는 방법

◆ 0과 1로 문자를 표현하는 방법

- 컴퓨터는 0과 1만 이해할 수 있다고 했는데, 문자는 어떻게 이해하고 출력하는 걸까?

◎ 문자 집합과 인코딩

- **문자 집합** : 컴퓨터가 인식하고 표현할 수 있는 문자의 모음
- 컴퓨터는 **문자 집합**에 속해 있는 문자를 이해할 수 있고, 그에 속해 있지 않는 문자는 이해할 수 없다.
- 예를 들어 문자 집합 {a, b, c, d}인 경우 컴퓨터는 4개의 문자를 이해하고, e나 f 같은 문자는 이해할 수 없다.

- 문자 집합에 속한다고 해서 컴퓨터가 그대로 이해하는 것이 아닌, 문자를 0과 1로 변환할 때 비로소 컴퓨터가 이해 가능
- **문자 인코딩** : 문자 → 0과 1로 변환
- **문자 디코딩** : 0과 1로 이루어진 문자 코드 → 사람이 이해할 수 있는 문자로 변환

- 파이썬에서도 비슷한 chr() , ord() 함수가 있었다.
- **ord(문자)** : 아스키 코드를 반환해준다. 문자를 아스키 코드로
- **chr(숫자)** : 숫자에 맞는 아스키 코드를 찾아 문자를 반환

◎ 아스키 코드

- 초창기 문자 집합 중 하나
- 영어 알파벳과 아라비아 숫자, 그리고 일부 특수문자 포함
- 0부터 127까지 총 128개의 숫자 중 하나의 고유한 수에 1대1 대응

아스키 코드표

십진수	십육진수	문자
0	0	Null
1	1	Start of Header
2	2	Start of Text
3	3	End of Text
4	4	End of Transmission
5	5	Enquiry
6	6	Acknowledge
7	7	Bell
8	8	Backspace
9	9	Horizontal Tab
10	A	Line Feed
11	B	Vertical Tab
12	C	Form Feed
13	D	Carriage Return
14	E	Shift Out
15	F	Shift In
16	10	Data Link Escape
17	11	Device Control 1
18	12	Device Control 2
19	13	Device Control 3
20	14	Device Control 4
21	15	Negative Acknowledge
22	16	Synchronous Idle
23	17	End of Trans. Block
24	18	Cancel
25	19	End of Medium
26	1A	Substitute
27	1B	Escape
28	1C	File Separator
29	1D	Group Separator
30	1E	Record Separator
31	1F	Unit Separator

십진수	십육진수	문자
32	20	Space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

십진수	십육진수	문자
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

십진수	십육진수	문자
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	Del

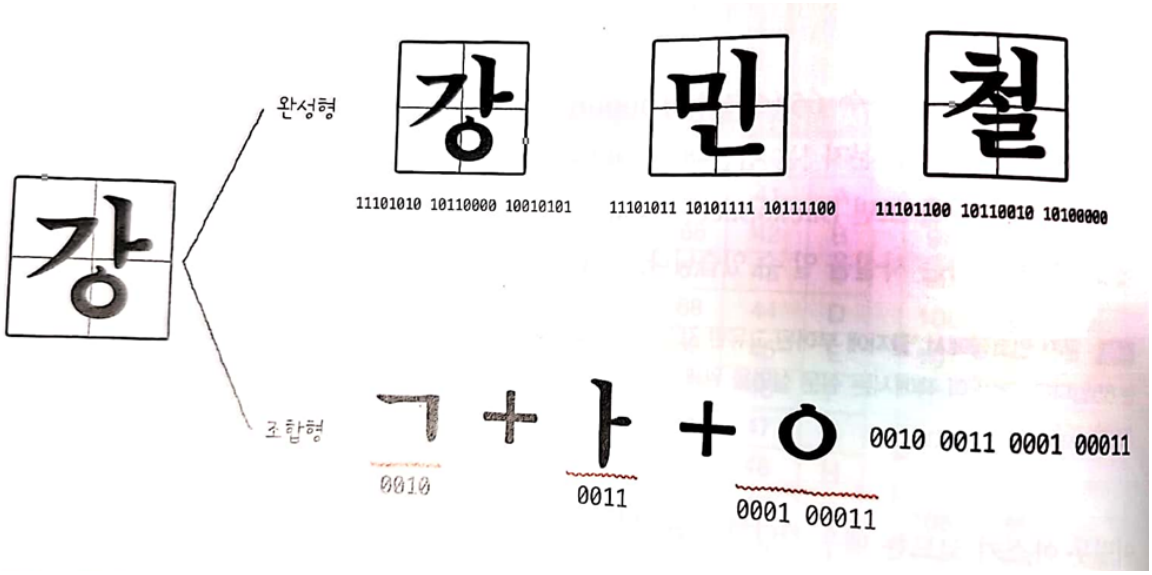
→ 한글을 표현할 수 없다는 단점 (그 밖의 아스키 코드란 문자 집합 외의 문자, 특수문자 포함)

- 확장 아스키 → 턱없이 부족

- 한국을 포함한 영어권 외의 나라들이 자신들의 언어를 0과 1로 표현할 수 있는 고유한 문자 집합과 인코딩 방식이 필요하다고 생각!

◎ EUC-KR

- 우선 한글 인코딩 방식에 대해 알아보자
- 한글 인코딩 방식은 **완성형**과 **조합형**이 존재
- **완성형 인코딩** : 초성, 중성, 종성의 조합으로 이루어진 완성된 하나의 글자에 고유한 코드 부여
- **조합형 인코딩** : 초성을 위한 비트열, 중성을 위한 비트열, 종성을 위한 비트열을 할당하여 그것을 조합하여 하나의 글자 코드 완성



- **EUC-KR**은 KS X 1001, KS X 1003이라는 문자 집합을 기반으로 하는 대표적인 **완성형 인코딩** 방식
- 초성, 중성, 종성이 모두 결합된 한글 단어에 2바이트 크기의 코드를 부여
- **EUC-KR**로 인코딩된 한글 한 글자를 표현하려면 16비트가 필요 ⇒ 네자리 십육진수로 나타낼 수 있음
- EUC-KR 인코딩 방식으로 총 2,350개 정도의 한글 단어 표현 가능

- 뷁, 뷓, 믜 같은 글자는 문자 집합에 정의되지 않아 EUC-KR로 표현 불가능

- 웹사이트 한글 깨지는 문제가 발생하자 이를 조금이나마 해결하기 위해 등장한 것이 마이크로 소프트 사의 **CP949**
- **CP949**는 EUC-KR의 확장된 버전
- 하지만 이마저도 한글 전체 표현하기에 넉넉한 양이 아님

◎ 유니코드와 UTF-8

- 한국어 외에도 영어, 일본어, 중국어, 독일어를 지원하는 웹사이트가 있다면 5개 언어의 인코딩 방식을 모두 이해하고 지원해야 한다. (깨지는 문제가 발생하지 않기 위해)
- 모든 언어를 아우르는 문자 집합과 통일된 표준 인코딩 방식이 있다면 언어 별로 인코딩 할 필요가 없다.

◎ 유니코드

- EUC-KR보다 훨씬 다양한 한글을 포함하며, 대부분의 나라의 문자, 특수문자, 화살표나 이모티콘까지도 코드로 표현할 수 있는 통일된 문자 집합
- https://www.unicode.org/charts/PDF/UAC00.pdf

◎ UTF-8

- 아스키 코드, EUC-KR은 글자에 부여된 값을 그대로 인코딩 값으로 삼음
- 유니코드는 글자에 부여된 값 자체를 인코딩된 값으로 삼지 않고, 이 값을 다양한 방법으로 인코딩한다.
- 이러한 인코딩 방법에는 크게 UTF-8, UTF-16, UTF-32 등이 있다.
- **UTF-8, UTF-16, UTF-32**는 유니코드 문자에 부여된 값을 인코딩하는 방식
- **UTF** : Unicode Transformation Format
- 그 중 가장 대중적인 것이 **UTF-8**
- 단 한 가지 인코딩 방식만 존재하는 **단일 인코딩 방식**이며, 인터넷 사이트에서 가장 많이 쓰이는 인코딩이다.

- 유니코드를 지원하는 대부분의 프로그램들은 일단 UTF-8을 디폴트 상태로 지정해 주는 경우가 많다.
- UTF-8은 통상 1바이트부터 4바이트 까지의 인코딩 결과를 만들어 낸다.
- **UTF-8**로 인코딩한 값의 결과는 **1바이트 ~ 4바이트 사이의 값**이 된다.

첫 코드 포인트	마지막 코드 포인트	1바이트	2바이트	3바이트	4바이트
0000	007F	0XXXXXXX			
0080	07FF	110XXXXX	10XXXXXX		
0800	FFFF	1110XXXX	10XXXXXX	10XXXXXX	
10000	10FFFF	11110XXX	10XXXXXX	10XXXXXX	10XXXXXX

- 유니코드 문자에 부여된 값의 범위가 0부터 $007F_{(16)}$까지는 1바이트로 표현
- 유니코드 문자에 부여된 값의 범위가 $0080_{(16)}$부터 $07FF_{(16)}$까지는 2바이트로 표현
- 유니코드 문자에 부여된 값의 범위가 $0800_{(16)}$부터 $FFFF_{(16)}$까지는 3바이트로 표현
- 유니코드 문자에 부여된 값의 범위가 $10000_{(16)}$부터 $10FFFF_{(16)}$까지는 4바이트로 표현

그렇다면 '한글'은 몇 바이트로 구성될까요? '한'에 부여된 값은 $D55C_{(16)}$, '글'에 부여된 값은 $AE00_{(16)}$이었죠? 두 글자 모두 $0800_{(16)}$와 $FFFF_{(16)}$ 사이에 있네요. 따라서 '한'과 '글'을 UTF-8로 인코딩하면 3바이트로 표현될 것을 예상할 수 있습니다.

- 저 위의 X표가 있는 곳에 유니코드에 문자에 부여된 고유한 값이 들어간다.
- '한'과 '글'에 부여된 값은
- '한' : $D55C(16)$ → 1101 0101 0101 1100 (2)
- '글' : $AE00(16)$ → 1010 1110 0000 0000 (2)
- 따라서 '한'과 '글'의 UTF-8 방식으로 각각 인코딩하면
- 1110**1101** 10**010101** 10**011100** (2)
- 1110**1010** 10**111000** 10**000000** (2)