2진수와 그 기계적인 활용

2021 동탄고등학교 30925 전한결

April 7, 2021

Contents

1 10진수와 2진수

1.1 10진수

우리는 살면서 10진수를 통해 숫자를 표현한다. 인류가 굳이 10진수를 사용하는 이유는 정확히 밝혀지지 않았지만, 대부분의 인류학자들은 그것이 사람의 손가락으로 수를 세는 것에서 수가 기원했기 때문이라고 말한다.

10진수는 0부터 9까지의 10가지 모양으로 '없음'에 해당하는 수부터 '아홉'에 해당하는 수까지를 표현한다. 이 숫자를 '일의 자리 숫자'라고 한다. 만약 '아홉' 이상의 수를 표현해야 할 때에는 숫자 옆에 '하나'에 해당하는 수를 쓰고 숫자가 할 수 있는 경우의 수가 한 번 다 한 후에 몇이 더 큰지를 '일의 자리 숫자'에 나타낸다. 이 숫자를 '십의 자리 숫자'라고 한다. 십의 자리 숫자도 일의 자리 숫자와 같이 10가지의 경우의 수를 가질 수 있으며, 이 경우의 수를 모두 소진했을 때에는 또 옆에 숫자를 쓰고, 이 숫자를 '백의 자리 숫자'라고 한다. 우리가 사용하는 10진수에서는 이러한 진행을 이어가며 그 값을 나타낸다.

1.2 기계 장치에서 정보의 전달

컴퓨터는 기계적인 신호로 이루어져 있고, 기계적인 신호는 전압이 강한지 약한지를 통해 그 정보를 전달한다. 대부분의 기계 장치에서는 전력이 흐르지 않는, 즉 전압이 0인 지점 점과 전압이 최대[1] 인 지점의 두 가지를 1/2 또는 2/3지점에서 나누어 구분한다. 즉, 기계장치에서는 전압의 세기로 총 두 가지 경우를 나타낼 수 있는 것이다.

하지만 두 가지 경우로 의미있는 정보를 전달하기는 쉽지 않다. 수학자들은 당연히 이러한 문제를 알고 있었으며, 이 문제를 타파하기 위해 10진수에서 값을 표현하는 방식을 빌려왔다. 우리는 이것을 '2진수'라고 부른다.

1.3 2진수

10진수의 경우, 사용하는 10개의 글자 중 하나를 써서 표현할 수 있는 상황의 개수가 10개이기 때문에 10가지를 넘는 정보를 표현하기 위해서 숫자 옆에 다른 숫자를 써서 10가지를 초과하는 정보를 표현한다. 2진수도 이 방식을 적용해서 사용할 수 있다. 단지, 수를 구성하는 숫자가 2개밖에 없을 뿐이다.

2진수는 0부터 1에 해당하는 2가지 모양으로 '없음'에 해당하는 수부터 '하나'에 해당하는 수까지를 표현한다. 이 숫자를 10진수에서와 같이 '일의 자리 숫자'라고 한다. 만약 '하나' 이상의 수를 표현해야 할 때에는 숫자 옆에 '하나'에 해당하는 수를 써서 숫자가 할 수 있는 경우의 수가 한 번 다 한 후에 몇이 더 큰지를 '이의 자리 숫자'에 나타낸다. 이 숫자를 '이의 자리 숫자'라고 한다.[2] 이와 같은 방식으로 '사의 자리 숫자', '팔의 자리 숫자', '십육의 자리 숫자' 등을 정의하여 수를 표현한다.[3]

---

[1] 보통은 이 최대점이 어디인지 기계를 설계하는 사람이 정한다.

[2] 왜냐하면 2진수 10에 해당하는 수를 우리가 '이'라고 부르기 때문이다. 10진수의 경우도 10에 해당하는 수를 우리가 '십'이라고 부르기 때문에 십의 자리 숫자라고 하는 것이다.

[3] 2진수는 10진수 숫자와 헷갈릴 수 있기 때문에 그 숫자가 어떤 표기법을 통해 표현되었는지를 숫자 옆에 아랫첨자로 나타낸다. 사람에 따라 첨자 안의 숫자에 괄호를 치는 경우도 있고, 그냥 괄호로

$$11000_2$$

1.4 2진수 읽기

위에 11000_2라고 2진수로 표현한 수는 우리가 24라고 하면 아는 수와 같은 값을 나타낸다. 2진수를 보고 그 크기는 어떻게 가늠할 수 있을까?

우리가 10진수를 읽을 수 있는 것은 10진수에 대해 배웠기 때문이다. 2진수도 배운다면 누구나 읽어낼 수 있다. 따라서 일단 10진수에 대해서 다시 한 번 알아보자.

예를 들어, 우리 눈 앞에 세 자리 숫자가 있다고 하자. 이 숫자는 10진수로 표현되었다.

$$794$$

이 숫자는 백의 자리 숫자가 7, 십의 자리 숫자가 9, 일의 자리 숫자가 4이다. 이 값은 다음과 같이 표현될 수도 있다.

$$7 \times 100 + 9 \times 10 + 4 \times 1$$

또한 수학적으로는 다음과 같은 방식으로 표현할 수도 있다. 다음은 10에 대한 2차식이다.

$$7 \times 10^2 + 9 \times 10^1 + 4 \times 10^0$$

10진수는 가장 오른쪽에 있는 수(일의 자리 숫자)에다가 10^0을 곱하고, 그 값에 그 다음 자리에 있는 수(십의 자리 숫자)에다가 10^1을 곱한 값을 더하고, 이러한 과정을 숫자가 끝날 때까지 반복한 것이다. 이를 일반화하면 다음과 같다. n진수는 가장 오른쪽에 있는 수(n^0의 자리 숫자)에다가 n^0을 곱하고, 그 값에 그 다음 자리에 있는 수(n^1의 자리 숫자)에다가 n^1을 곱한 값을 더하고, 그 다음 자리에 있는 수(n^2의 자리 숫자)에다가 n^2을 곱한 값을 더하고... 결국 n에 대한 다항식으로 표현할 수 있는 수를 말한다. 그렇다면 2진수 11000은 다음과 같은 (2에 대한) 다항식으로 표현할 수 있다.

$$\begin{aligned}
&1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
=&1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1 \\
=&1 \times 16 + 1 \times 8 \\
=&16 + 8 \\
=&24
\end{aligned}$$

1.5 기계 장치에서 2진수의 활용

다시 수학자들의 이야기로 돌아와보자. 수학자들은 이렇게, n진수로 표현한 숫자가 결국 n에 대한 다항식으로 표현 가능한 수라는 것을 발견해내었다. 2진수 숫자를 하나의 전기 신호로 표현하고, 그 전기 신호를 여러개 작성하면 더 다양한 값을

나타내는 경우도 있다.

나태낼 수 있다는 사실을 알아낸 것이다. 전선을 여러개 사용하는 방법과 시간차를 두고 여러 값을 보내는 방식 중에서 수학자들은 시간차를 두고 여러 값을 보내는 방식을 선택했다. 컴퓨터와 같은 기계장치들은 시간차를 두고 전송되는 일련의 신호를 일련의 방식에 따라 이해하고 처리한다.

2 2진수 처리

컴퓨터가 2진수를 사용하여 숫자를 전송하고 처리한다는 사실을 알게 되었다. 하지만 컴퓨터는 우리가 생각하는 것만큼 그렇게 단순하게 작동하지 않는다.

우리가 '수'라고 할 때에 그 수는 보통 실수를 나타낸다. 실수에는 유리수와 무리수가 있고, 각각에는 0보다 큰 양수와 0보다 작은 음수, 그리고 0이 있다. 앞으로는 이러한 수들을 종류에 따라 컴퓨터에서 어떻게 표현하는지 알아보고, 컴퓨터에서 그 표현방식에 따라 어떤 연산을 취하는지를 알아보고자 한다.

2.1 자료 저장에 대한 컴퓨터 구조

컴퓨터 구조에 대해서 이야기하자면 양이 너무나 방대해 한 권의 책을 써야 할 정도이겠지만, 이번 장에서는 간단히 설명에 필요한 만큼만 서술하려고 한다.

컴퓨터의 구조는 크게 입력부분, 연산부분, 저장부분, 출력부분[4]으로 나눌 수 있다. 각각의 부분들은 그 이름에 맞게 작업을 수행하는데, 이번에 알아볼 부분은 '저장부분'이다. 컴퓨터는 2진수로 정보를 저장하고 전달하고 활용한다고 했다. 따라서 0과 1을 많이 저장할 수 있게 고안된 장치를 사용한다. 우리가 사용하는 컴퓨터에서 그 기능은 '주 기억 장치'와 '보조 기억 장치'가 수행한다.

주 기억 장치는 자주 책상에 비교되곤 한다. 우리가 업무를 볼 때에 당장 작업해야 하는 문서나 참고자료, 필통이나 도장 등은 책상에 놓고 사용하기 때문에 주 기억 장치가 넓을수록 한 번에 많은 작업을 수행할 수 있다.

반면에 보조 기억 장치는 책장에 비교되곤 한다. 당장 쓰지는 않지만 언젠가 쓸 일이 있을 때에 그때 그때 꺼내서 책상에 올려놓기 위해 사용되는 자료들을 보관해놓는 곳이다. 따라서 보조 기억 장치가 넓을수록 많은 정보를 기록해놓고 오래 볼 수 있다.

이 주 기억 장치와 보조 기억 장치의 크기는 '비트(bit)' 또는 '바이트(Byte)'라는 단위로 나타낸다. 기억 장치에 수용할 수 있는 2진수의 자릿수를 '비트(bit)'로 나타낸다. 예를 들어 지금 16개의 똑딱이 스위치가 있다고 치자. 이 스위치는 순서를 가지고 있고 각각 켜지거나 꺼진 상태를 나타낼 수 있다. 이 16개의 스위치들은 함께 해서 16자리 2진수 숫자를 나타낼 수 있으므로 16비트 기억장치라고 볼 수 있다. 바이트는 나중에 다시 설명하겠지만, 8개의 비트를 한 단위로 묶은 것이다. 왜 굳이 8개를 하나의 바이트로 묶었는지는 나중에 설명하게 될 것이다. 어쨌든 16비트 기억장치인 16개의 똑딱이 스위치는 2비트짜리 기억장치가 되는 것이다.

$$16 \text{ bit} = 2 \text{ Byte}$$

1바이트는 8비트이기 때문에 총 $2^8 = 256$개의 경우를 표현해낼 수 있다.

참고로 16진수는 0부터 9까지의 숫자에 A부터 F까지의 알파벳을 함께 하여 만든 수 체계로, 4비트를 한 글자로 표현할 수 있다. 2진수를 10진수로 변환하면

[4]이 이름은 설명을 위해 임의로 지은 것이다.

숫자와 변환된 숫자의 연관성이 나타나지 않기 때문에 불편했기 때문에 16진수 체계가 고안되었다. 16진수 표기법 따르면 다음 등식이 성립한다.

$$11010010_2 = \mathrm{D2}_{16}$$

16진수는 알파벳을 사용하는 이유로 8진수 체계를 사용하기도 했는데, 16진수가 표현할 수 있는 비트 수는 2의 거듭제곱 꼴로 나타낼 수 있지만 8진수는 3개의 비트를 나타내기 때문에 16진수를 사용하는 경우가 더 많다. 한 바이트는 8비트고 이는 2자리 16진수로 완벽히 표현가능한데, 이 역시 8진수보다 16진수를 선호하는 이유 중 하나이다.

2.2 정수

2.2.1 정수의 표현법

정수는 크게 음의 정수, 양의 정수, 0으로 나눌 수 있다. 음의 정수와 양의 정수는 각각 일대일 대응이 가능하고 순서가 있으므로 어떤 정수를 표현할 때에 몇 번째 정수인지를 표현하면 쉽게 이진수로 정수를 표현할 수 있다. 또한 음의 정수인지 양의 정수인지를 첫 번째 비트로 표현하면 모든 정수를 표현할 수 있게 된다. 이렇게 된다면 n비트 정수가 -2^{n-1}부터 2^{n-1}까지를 표현할 수 있게 된다. 이 방식을 '부호화 절댓값 표기법'이라고 한다.

음의 정수의 부호가 1이고 양의 정수의 부호가 0이라고 한다면 다음 등식이 성립한다.

$$10011000_2 = -24$$

하지만 이 방식을 사용하면 0이 2개가 생기는 문제가 발생한다. 이 방식에 따르면, 다음 두 4비트 정수는 같은 값을 가리킨다.

$$1000_2 = -0 = 0$$
$$0000_2 = +0 = 0$$

따라서 0이 겹쳐지는 문제를 해결하기 위해 컴퓨터에서는 '2의 보수 표기법'을 사용한다.

2의 보수 표기법은 사전적으로 '2의 보수 관계에 있는 두 2진수로 절댓값이 같고 부호가 다른 두 10진수를 표현하는 것'으로 정의된다. 이 표기법은 다음과 같은 특징을 가진다.

- 양의 정수를 표현할 때에는 부호 비트(가장 처음 비트)를 제외한 비트를 사용하여 표현한다.
- 음의 정수를 표현할 때에는 양의 정수를 비트 반전한 뒤 반전한 수에 1을 더한다. 이때 올림 비트는 무시한다.
- 음의 정수를 통해 양의 정수를 표현할 때에는 음의 정수를 비트 반전한 뒤 반전한 수에 1을 더한다. 이때 올림 비트는 무시한다.

예를 들어, 24를 8비트 이진수 2의 보수 표기법으로 표현한 것에 대해 다음 등식이 성립한다.

$$00011000_2 = 24$$

이때, -24를 표현하기 위해서는 양의 정수를 비트 반전한 뒤 반전한 수에 1을 더한다.

00011000_2	24
11100111_2	비트 반전
11101000_2	1을 더한다. -24 완성

이렇게 한다면 0의 표현법이 2개가 되는 문제가 해결된다.

00000000_2	0
11111111_2	비트 반전
00000000_2	1을 더한다.

이때 계산하는 수는 8비트 이진수이기 때문에 9번째 비트를 제외하고 생각한다. 이 방식을 사용하면 정수의 연산을 더욱 쉽게 할 수 있다.

2.2.2 2진수의 덧셈

2진수의 덧셈은 AND 연산과 XOR 연산으로 진행된다. 각각 연산에 대한 설명은 다음 표로 설명할 수 있다.

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

XOR 연산은 연산 후 수의 일의 자리 숫자를, AND 연산은 연산 후 수의 이의 자리 숫자를 나타낼 수 있다. 만약 이의 자리 숫자가 연산하는 숫자의 일의 자리 숫자에 해당한다면 그 덧셈의 이의 자리 숫자를 사의 자리 숫자로 적용하면 두 자리 이상의 연산도 가능하다.

		1	0	
+		1	1	
		0	1	XOR
	1	0		AND
	1	0	1	+

2.2.3 정수의 연산

2의 보수 표기법이 강력한 표기법인 이유는 연산을 쉽게 수행할 수 있기 때문이다. 2의 보수 표기법을 사용하면 별도의 추가적인 연산 없이 그냥 두 비트에서 더하기 연산을 수행해도 원하는 결과를 얻을 수 있다는 말이다. 예를 들어 $94 - 35$를 이진수로 수행한다고 해보자. 이때 사용하는 비트수는 8이다.

이때 94는 이진수로 01011110_2이고, 35는 이진수로 00100011_2이다. 하지만 35에는 빼기 연산을 적용하므로 -35를 더하는 연산을 수행해야 하기 때문에 -35를 2의 보수 표기법에 의거하여 이진수로 표현해보면 11011101_2이다. 이제 두 수를 더하기 연산하면 원하는 결과를 얻을 수 있다.

		0	1	0	1	1	1	1	0	
+		1	1	0	1	1	1	0	1	
		1	0	0	0	0	0	1	1	XOR
	0	1	0	1	1	1	0	0		AND
	1	0	0	1	1	1	0	1	1	+

하지만 우리는 8비트 연산을 수행하기 때문에 9번째 비트를 무시하여야 한다. 그러면 우리가 얻은 결과는 다음과 같다.

$$00111011_2 = 59$$

컴퓨터는 위와 같이 덧셈과 뺄셈 연산을 수행한다.

A와 B의 곱하기 연산은 A에다가 A를 더하면서 B를 1씩 빼는데, 이때 B가 0이 되면 연산을 종료하는 것으로 구현이 가능하다.[5]

A와 B의 나누기 연산은 A에서 B를 빼면서 A가 B보다 작아지는 지점이 몇 번을 뺀 후인지 알아보는 것으로 구현이 가능하다.

2.3 유리수

실수의 표기법에 대해 설명하기 전에, 컴퓨터와 이론의 차이에 대해서 간단히 짚고 넘어가겠다. 우리가 대표적인 무리수라고 하는 π의 값을 대부분 알고 있을 것이다.

$$\pi \simeq 3.14159265358979\ldots$$

하지만 π의 값을 숫자로 정확히 표현한 것을 눈으로 확인한 사람은 없을 것이다. 이는 모든 실수를 표현하는 데에 한계가 있는 우리 수 표기법 또는 실수의 특징의 탓이다. 즉, 숫자를 통해서는 모든 실수를 표현할 수 없다. 이는 컴퓨터에서도 마찬가지이다. 컴퓨터에서는 무리수를 표현할 수 없다. 무리수에 대한 이론적인 성질을 통해 무리수를 활용할 수는 있겠지만, 유리수를 대하는 것처럼 무리수를 대할 수가 없다는 것이다.

컴퓨터에서 실수(유리수)를 표현할 때에는 부동 소수점 방식과 고정 소수점 방식을 사용한다.

2.3.1 고정 소수점 방식 실수 표현

고정 소수점 방식은 부호화 절댓값 표기법과 비슷하다. 가장 처음 비트를 부호 비트로 사용하고 그 후 일정한 부분을 정수부, 그 후 일정한 부분을 소수부로 활용한다. 예를 들어, 고정 소수점 방식의 실수를 16비트 2진수로 표현한다고 하자. 이때 처음 비트는 부호 비트로 작용할 것이고 그 후 7개의 비트는 정수부를 나타내는 비트,

[5] $B < 0$인 경우 수가 0에 도달하지 않는데, 이때는 부호비트가 0으로 바뀔 때에 수 비트에 해당하는 모든 비트가 0으로 초기화되게 된다. 이것을 기준으로 연산을 종료한다. 이때 부호 비트 역시 덧셈의 결과에 적용되어야 한다.

그 후 8개는 소수부를 나타내는 비트가 될 것이다. 예의 예를 들어, 다음 2진수가 나타내는 실수는 다음 방정식을 만족한다.

$$1000011110100000_2$$

$$1 \mid 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \mid 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$$

$$\begin{aligned}
&1000011110100000_2 \\
&= -\left(1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3}\right) \\
&= -\left(4 + 2 + 1 + \frac{1}{2} + \frac{1}{8}\right) \\
&= -\frac{61}{8} \\
&= -7.625
\end{aligned}$$

이러한 고정소수점 방식은 구현하기 편리하지만 표현 가능한 수의 범위가 적고 정밀도가 떨어지기 때문에 정확도보다 구현의 용이성이 필요한 시스템에서 자주 사용한다. 이러한 문제점을 해결한 것이 부동 소수점 방식이다.

2.3.2 부동 소수점 방식 실수 표현

부동 소수점 방식은 더 많은 범위를 실수로 표현하기 위해서 고안된 실수 표기법이다. 가장 처음 비트를 부호 비트로 사용하고 그 후 일정한 부분을 '지수 부분(exponential bias)', 그 후 일정한 부분을 '기수(fraction 또는 mantissa)'로 사용한다. 예를 들어, 부동 소수점 방식의 실수를 16비트 2진수로 표현한다고 하자. 이때 처음 비트는 부호 비트로 작용할 것이고 그 후 7개의 비트는 지수부를 나타내는 비트, 그 후 8개는 가수부를 나타내는 비트가 될 것이다. 이 표기법으로 실수 -7.625를 표현하면 다음과 같다.

$$0000001011101002$$

이 표기법은 직관적이지 않은데, 이 수를 해석하는 방식은 다음과 같다.

$$0 \mid 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \mid 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0$$

1. 가수부의 숫자를 이진 소수점으로 두는 이진수 유리수를 하나 만든다. 이때 정수부는 언제나 1이다.
   1.11101_2

2. 지수부를 2의 보수 표기법으로 해석한 수를 e라고 할 때, **1.**의 숫자에 $10_2{}^e$를 곱한다.
   $1.11101_2 \times 10_2{}^{10_2}$

3. 이 수에 부호를 붙여 부호화 절댓값 표기법으로 해석한다.
   $-1 \times 1.1101_2 \times 10_2{}^{10_2}$

이 수를 해석하면 다음과 같다.

$$\begin{aligned} & -1 \times 1.11101_2 \times {10_2}^{10_2} \\ =& -1 \times 1.90625 \times 2^2 \\ =& -1.90625 \times 4 \\ =& -7.625 \end{aligned}$$

부동 소수점 표기법은 추가적인 연산이 많이 필요하고 고정 소수점 표기법에 비해 구현도 어렵지만 정확도가 뛰어나고 표현할 수 있는 수의 범위가 크기 때문에 대부분의 시스템에서 사용하는 방식이다.

2.4 문자

2진수로 표현하는 방식을 알아보기 전에 어떻게 표현하는지 가장 궁금했던 것이 바로 이 문자의 표기이다. 우리가 수학 시간에 배우는 숫자들로는 절대 메시지를 표현할 수 없을 것이라고 생각됐기 때문이다. 하지만 수학자들은 2진수만을 이용하여 문자, 심지어는 문자열까지 표현하는 방법을 고안해냈다.

위에서 '비트(bit)'와 '바이트(Byte)'의 차이점에 대해서 말했다. 바이트는 비트보다 2^8배 더 많은 경우의 정보를 저장할 수 있다. 여기서 2^8이라는 숫자는 문자의 표기를 설명하면서 빼놓을 수 없는 요소이다. 컴퓨터에서는 어떻게 문자를 표현하는 것일까?

살면서 '아스키(ASCII) 코드'라는 말은 들어본 적이 많을 것이다. ASCII는 American Standard Code for Information Interchange의 약자로, '정보 교환을 위한 미국 표준 부호'라는 뜻이다. 소프트웨어는 이 아스키라는 표준에 맞춰 글자를 표현한다.

아스키는 7개의 비트를 이용하여 컴퓨터에 명령을 내리거나 문자를 쓰기 위해 고안된 표준이다. 총 33개의 제어 문자와 95개의 출력 가능한 문자가 있는데, 이중 33개의 제어 문자는 대부분 현재는 사용되지 않는다. 이 아스키 표의 일부를 보면 다음과 같다.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

이 표는 영어 문자와 숫자를 대응시킨 것이다. 즉, 40_{16}행의 8이 'H'라는 말은 $40_{16} + 8_{16} = 48_{16} = 72$라는 숫자가 'H'에 대응된다는 뜻이다. 아스키 코드의 모든 문자는 7비트로 표현이 가능하므로 이는 다음과 같이 나타낼 수 있다.

$$1001000_2$$

만약 'HELLO'라는 말이 하고 싶다면 35개의 비트가 필요하다.

$$100100010001011001100100110010011111_2$$

그런데 8비트가 문자와 관련이 있다는 이유는 무엇일까? 아스키 코드에서는 7비트로 문자를 표현하지만 'Latin-1'이라는 인코딩 방식에서는 8비트로 문자를 표현하기 때문이다. 이 인코딩 방식에는 영어, 스페인어, 독일어 등까지 표현이 가능한 문자 목록이 포함되어 있다.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

이 방식으로 'Schrödinger'를 표현하면 다음과 같다.

$$53636872F6C4696E676572_{16}$$

하지만 시간이 지남에 따라 유럽 이외의 지역으로도 컴퓨터가 보급되면서 각국의 언어에 맞춰 인코딩 방식을 개발하는 시도가 늘어났고, 한글 조합형, 완성형 토론 등이 이뤄지기도 했다. 하지만 이 역시도 그 나라에서 사용하지 않는 소프트웨어를 설치하면 글씨가 깨져보이는 등의 문제가 발생해 사람들을 불편하게 했고, 결국 전세계적인 문자 표를 만들기 위한 '유니코드'가 출범한다. 이 유니코드 표에 따르면 완성형 한글은 AC00번부터 D7A3번까지 총 11172글자가 배당되어있다. 전 세계의 모든 글자를 담기에 256개는 너무 적은 수였기 때문에 그것을 넘어가는 글자들은 대부분 한 글자에 2바이트를 차지한다. 따라서 16진수로 나타낼 때에는 4글자로 나타내야 한다.[6] '한글'을 16진수로 나타내면 다음과 같다.

$$D55CAE00_{16}$$

이렇게 한글은 4글자, 유럽의 글자는 2글자의 16진수를 한 글자로 계산하는데, 이렇게 다양한 해석 방식에 표준을 정하고자 했으니, 그것이 바로 '인코딩 방식'이다. 이 인코딩 방식이 달라짐에 따라서 멀쩡한 글씨가 보이지 않거나 하는 일이 일어나기도 한다.

3 논리 연산

아까 정수의 덧셈에 대해서 알아보면서 AND연산과 XOR연산에 대해 보았다. 이러한 연산을 '논리 연산'이라고 한다. 하지만 논리 연산에 AND와 XOR만 있는 것은 아니다. 논리 연산은 NOT과 OR에서 시작한다.

[6] 2바이트가 표현할 수 있는 경우의 수는 $(2^8)^2 = 65,536$개이다.

3.1 논리합과 논리 부정

OR 연산은 입력 값 2개 중 하나라도 1이 있다면 1을 내보내는 연산이다. OR의 논리표는 다음과 같다.

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

OR은 기호로 $A|B$, $A+B$ 또는 $A \vee B$로 표현한다. 다른 말로는 '논리합'이라고 한다.

NOT 연산은 입력 값이 1개인 연산이다. 말 그대로 신호의 부정을 내보낸다. NOT의 논리표는 다음과 같다.

A	not A
0	1
1	0

NOT은 기호로 $!A$, $\overline{A}$ 또는 $\neg A$로 표현한다. 다른 말로는 '논리 부정'이라고 한다.

3.2 논리합과 논리 부정을 통한 기타 논리 연산의 구현

OR과 NOT, 이 두 연산이 구현되어있는 상황에서는 어떤 연산도 구현할 수 있다. 가령 AND의 경우,

$$A \times B = A \cdot B = \overline{\overline{A} + \overline{B}}$$

로 구현이 가능하다.

덧셈 연산에서 꼭 필요한 XOR은 다음과 같이 구현할 수 있다.

$$A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$$

이외에도 AND를 뒤집은 NAND, OR를 뒤집은 NOR, XOR을 뒤집은 NXOR 등도 있다.

$$A \text{ nand } B = \overline{A \times B}$$
$$A \text{ nor } B = \overline{A + B}$$
$$A \text{ nxor } B = \overline{A \oplus B}$$

결국 둘 중 하나가 켜져있는지와 거꾸로 하면 뭔지 알아보는 연산만으로 거의 모든 기계의 구현이 가능하다는 것이다.