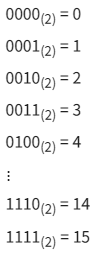
먼저 가장 간단한 2진수 표현법을 알아보자. 아마 2진수라고 하면 아래와 같은 수들을 생각할 것이다.



여기서 한 가지 큰 문제점이 있다. '음수를 표현할 수 없다'는 것이다.

**1. 부호 절대값(sign-magnitude)**

가장 쉽게 생각할 수 있는 방법은 최상위 비트(가장 왼쪽 비트)를 이용하는 방법이다. 최상위 비트는 MSB(Most Significant Bit)라고도 부른다.

예를 들어 5를 표현한다고 가정해보자. 이진수로는 아래와 같다.



만약 -5를 표현한다면 최상위 비트를 1로 설정하는 것이다. 즉, 최상위 비트가 0이면 양수, 1이면 음수를 의미한다.

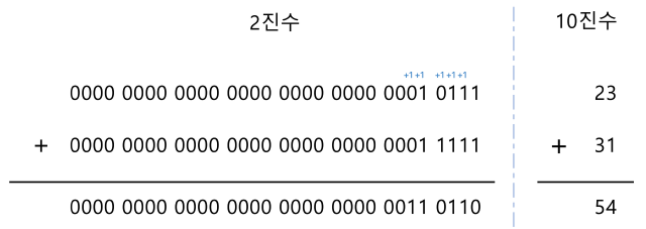


이 방식을 이용해 4bit로 음수와 양수를 표현해보면 아래와 같다.



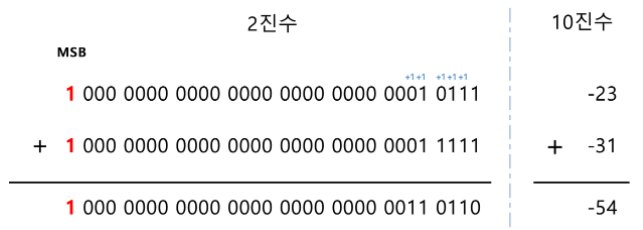
최상위 비트를 부호 절대값으로 사용하는 방식은 매우 직관적이면서 쉽다. 하지만 컴퓨터 입장에서 몇 가지 불편한 점이 있다. 일단 위 표를 보면 직관적으로 0이 양수와 음수로 나뉜다는 단점이 있다. 하지만 **가장 큰 단점은 뺄셈을 위해 각 수가 음수 또는 양수냐에 따라 고려해 구현해야 할 것이 많아진다는 것**이다.

기본적으로 이진수의 덧셈은 아래와 같다.



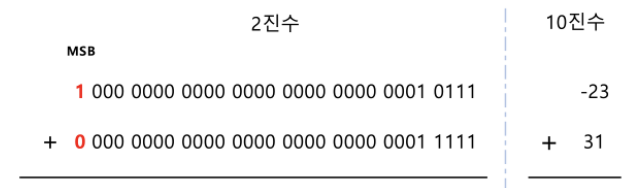
양수끼리의 덧셈은 문제가 없어 보인다. 그럼 음수의 덧셈은 어떻게 할까?

두 수의 덧셈에서 음수가 있는 경우는 3가지가 있다. 두 수 모두 음수인 경우, 첫 번째 수만 음수인 경우, 두 번째 수만 음수인 경우이다. 먼저 두 수 모두 음수인 경우를 살펴보자.



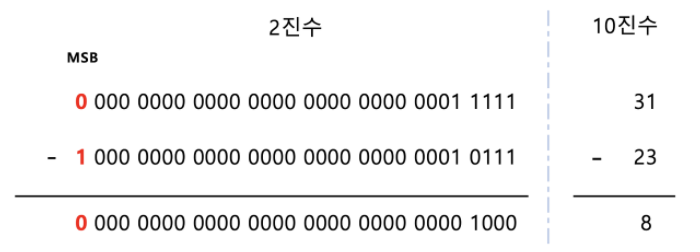
이 부분은 쉽게 구현할 수 있을 것 같다. 두 수의 최상위 비트가 같다면 결과값의 최상위 비트도 같아지게 된다. 나머지 비트들을 덧셈한 뒤 최상위 비트만 그대로 내려오면 되는 것이다.

문제는 다음부터 생긴다. 먼저 첫 번째 수가 음수인 경우를 살펴보자.



위에서 이진수의 덧셈을 하면 54가 나온다. MSB에 따라 +, -를 붙여줘도 올바른 답을 얻을 수가 없다. 그렇다고 31을 빼면 -23에서 최상위 비트를 제외한 나머지 비트에서 31을 빼줘야하는데 그러면 뺄셈을 따로 구현해야 한다.

위와 같은 경우에는 '절대값이 큰 수가 첫 번째 수'가 되도록 하면 된다.



이 방식은 사람한테는 쉬운 것처럼 보이지만 회로로 만들려고 하면 모든 경우의 수를 고려해야 하므로 쉬운 방식은 아니다. 즉 장단점을 정리하면 아래와 같다.

[장점]

1. 최상위 비트만 고려하면 되므로 직관적이다.

[단점]

1. +0과 -0 둘 다 존재하므로 둘 다 0으로 인식하도록 설정해야 한다.

2. 연산에서 고려해야할 것이 많아져 회로가 복잡해진다. (최상위 비트와 절대값을 각각 계산해야 한다.)

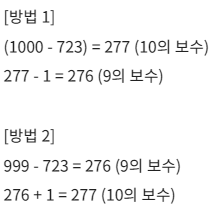
**2. 1의 보수 (One's Complement)**

1의 보수에 대해 알아보기 전에 보수가 무엇인지 알아보자. 보수란 '보충해주는 수'이다. 이때 2진수에서 보수는 **어떤 수를 만들기 위해 필요한 수**를 의미한다. 그리고 보수는 각 n진법마다 모두 존재하는데, n진법에는 n의 보수와 n-1의 보수가 쓰인다. 10진법을 예로 들어보자.

3에 대한 '10의 보수'는 '3에서 10을 만들기 위해 필요한 수'를 의미한다. 즉, 7이된다. 12에서 '10의 보수'라고 하면 88이 된다.

쉽게 말해 'n의 보수'는 **'어떤 수에 대해 n의 제곱수가 되도록 만드는 수'**라고 볼 수 있다. 그리고 'n-1 보수'는 (n의 보수-1)이 된다. 즉, 17의 10의 보수는 83이지만 9의 보수는 82라는 것이다.

위 관계를 다시 생각해보면 9의 보수에 +1한 값이 10의 보수가 된다는 것과 같은 의미이다. 즉, 10진법에서 723의 10의 보수와 9의 보수를 구한다고 하면 아래처럼 구할 수 있다.



지금까지 보수에 대해 알아보았다. 그럼 보수를 사용하는 이유가 뭘까? 앞서 부호 절대값을 이용하여 계산하는 경우 고려해야 할 점이 많았다. 하지만 보수를 사용하면 컴퓨터 입장에서 좀 더 쉽고 일관되게 쓸 수 있다.

앞서 부호 절대값에서 문제가 되었던 -23+32를 예로 들어보자. 음수인 -23을 보수로 취하고 풀이하면 아래와 같이 답을 얻을 수 있다.

{(100-23)+31}+100 = (77+31)-100 = 108-100 = 8

수학적으로는 위와 같이 풀이하여 답을 구할 수 있다. 이를 조금 응용하여 '보수'만 구해주는 방식으로 변형할 수 있다. 위에서는 보수를 구하기 위해 100이 더해진 만큼 다시 빼주었는데 그냥 보수를 구하기만 하고 다시 빼지 않는다는 것이다.

(100-23)+31 = 77+31 = 108

여기서 가장 왼쪽 값 1은 올림으로 발생한 수이기 때문에 버린다. 이를 최상위비트(MSB)에서 '자리 올림'이 되었다는 의미로 '캐리 발생'이라고 한다. 이때 중요한 점은 캐리가 발생하는 경우는 '양수'라는 의미고, 캐리가 발생하지 않은 경우 '음수'라는 의미이다.

정리하면 108에서 1이라는 수를 버린 08, 즉 8이 정답이 된다. 해당 과정을 단계별로 정리하면 아래와 같다.

(1) 음수에 대해 보수를 구한다: 100-23 = 77

(2) 구한 보수 값에 나머지 수를 더한다: 77+31 = 108

(3) 올림이 발생할 경우 해당 수는 버린다: 08

만약 올림이 발생하지 않는 경우를 살펴보자. -31+23을 예시로 들어보자.

(100-31)+23 = 69 + 23 = 92

캐리가 발생하지 않았다면, 그 수의 보수를 다시 구해서 얻어진 값에 음수 부호를 붙이면 된다.

100-92 = 8 => -8

두 수 모두 음수인 경우를 살펴보자. -23-31을 예시로 보자. 이때는 보수가 두 개 있기 때문에 캐리 여부 또한 2번 확인해야 한다.

(100-23)+(100-31)

= 77+69

= 146 (캐리 발생, 왼쪽자리를 버린다.)

= 46 (캐리가 발생하지 않는다.)

= 100-46 = 54

= -54

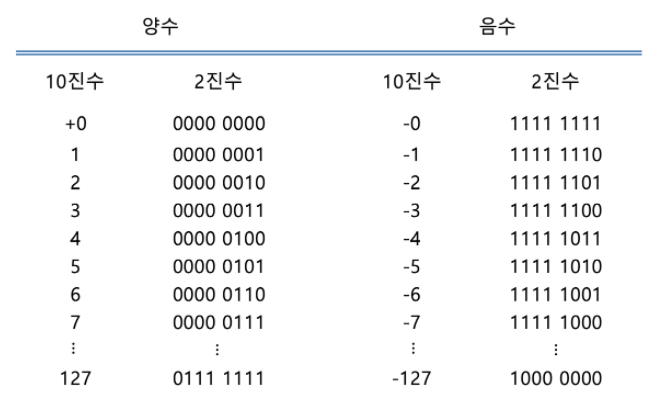
위와 같이 계산을 이전보다 편하게 할 수 있다.

그래서 조금만 생각해보면 왜 보수가 필요하고, 1의 보수와 2의 보수가 있는지 알 수 있다. 즉, 2진법의 계산을 위해 보수를 구하려고 있는 것이다.

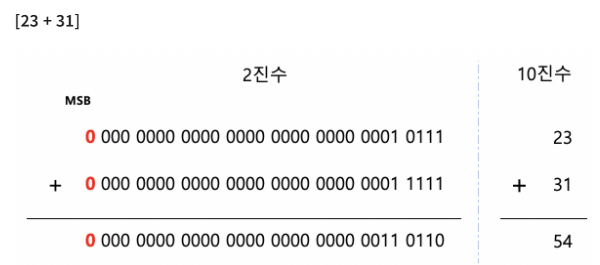
10진법을 제외한 다른 진법은 n-1보수를 구하는 것이 더 편하다. 이유는 아래와 같다. 2진수는 7자리로 이루어지면 최상위 비트는 MSB로 총 8bit이다.

0000 0011(=3)의 2의 보수는 1 0000 0000 - 0000 0011 = 1111 1101가 된다. 비트가 한정되어 있는 경우 보수를 구하는 과정에서 키트 한 칸이 더 필요하다. 즉, 실질적으로 9bit가 필요하다. 하지만 1의 보수의 경우 1111 1111 - 0000 0011 = 1111 1100으로 더 편리하게 1의 보수를 구할 수 있다.

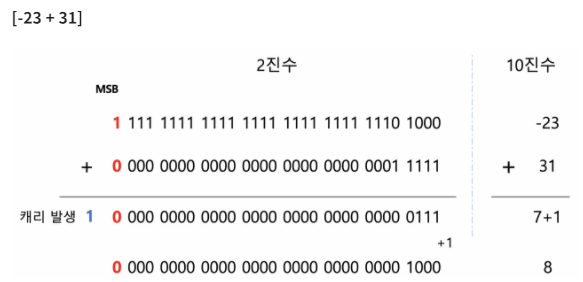
여기서 가장 중요한 점은 1의 보수를 구하면 알겠지만 1의 보수 방식에서 음수는 양수의 비트를 반전시킨 값이다. 3을 이진수로 나타내면 0000 0011이고, -3은 1111 1100이 된다. 아래 표를 참고하자.



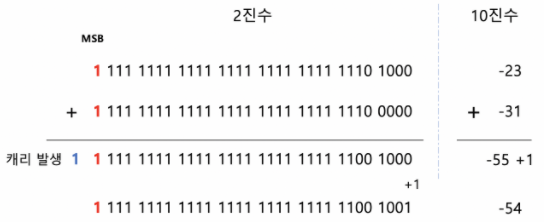
즉, 1의 보수를 이용하면 비트만 반전시키면 된다. 부호의 절대값을 따로 계산할 필요가 없고 뺄셈 대신 음수를 더하기만 하면 되는 것이다. 다만 1의 보수에도 단점은 존재한다.



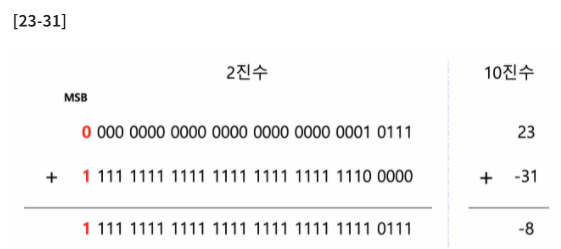
양수끼리의 덧셈은 어렵지 않게 된다.



위 계산을 보면 -23의 비트를 더해주니 MSB자리를 넘어서 올림이 발생한다. 즉, 캐리가 발생한 것이다. 1의 보수에서 캐리가 발생하면 +1을 해주면 된다.



이 경우도 캐리가 발생하므로 +1을 해준다.



이 경우에는 캐리가 발생하지 않으므로 바로 올바른 정답이 나온다.

계산 후 MSB가1일 경우는 음수, 0일 경우는 양수를 유지하고 캐리가 발생할 경우에만 +1을 해주면 되므로 편리하다. 하지만 여전히 남아있는 문제가 있다. 캐리가 발생할 경우를 처리해야 한다는 점과 +0과 -0이 존재하는 것이다. 1의 보수 방식의 장단점은 아래와 같다.

[장점]

(1) 비트만 반전시키면 음수값을 얻을 수 있다.

(2) MSB의 성질이 유지된다.

(3) 덧셈만으로 뺄셈을 구현할 수 있어 비교적 회로가 단순해진다.

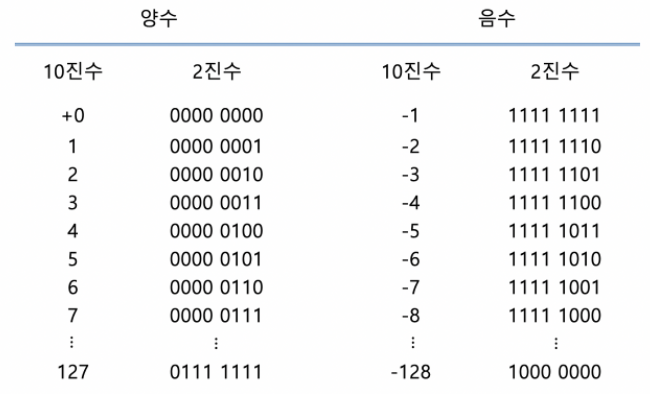
[단점]

(1) 캐리가 발생하는 경우를 처리해줘야 한다.

(2) -0과 0을 모두 인지하도록 처리해야 한다.

**3. 2의 보수 (Two's Complement)**

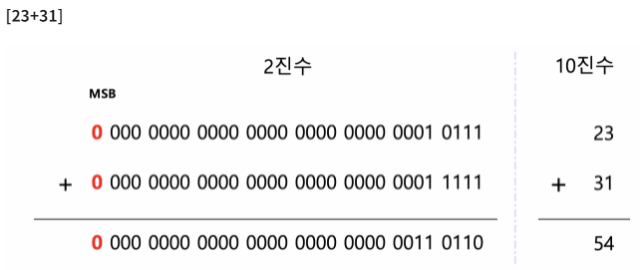
1의 보수의 -0과 +0이 존재하는 문제점을 해결하려면 단순히 -0을 없애면 된다. 즉, 음수 영역에서 각 대응되는 수를 -1씩 대응시키는 것이다.



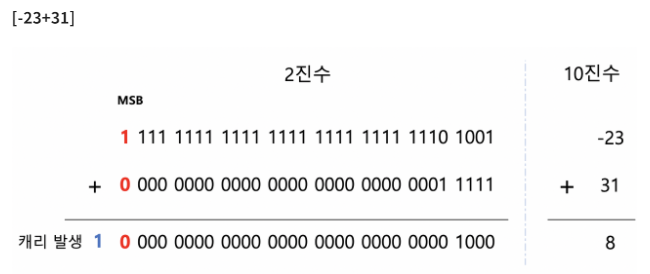
음수의 영역에서 각 대응되는 수를 -1씩 대응시켰다는 말은 a에 대한 2의 보수는 1의 보수 +1이라는 것과 같은 말이다. 예를 들어보자. 이전에 1에 대해 -1을 표현하기 위해 1의 보수를 이용하면 비트를 반전시키면 된다고 했다. 즉, 2-3 연산을 하려고 할 때, 뺄셈 대신 음수를 더하는 방식을 사용하기 위해 보수를 이용하는데, 3에 대한 1의 보수는 0000 0011 -> 1111 1100이 된다.

하지만 2의 보수를 이용한 음수를 표현하는 표에서 1111 1100은 -4이다. -3을 표현하려면 -4에 +1을 해야한다.

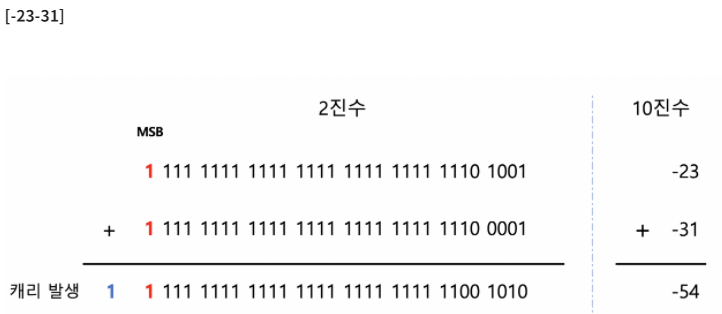
즉, **어떤 수의 부호를 바꾸고자 한다면 비트를 반전시킨 뒤 1을 더하면 된다**는 것이다. 다시 4가지 경우를 보도록 하자.



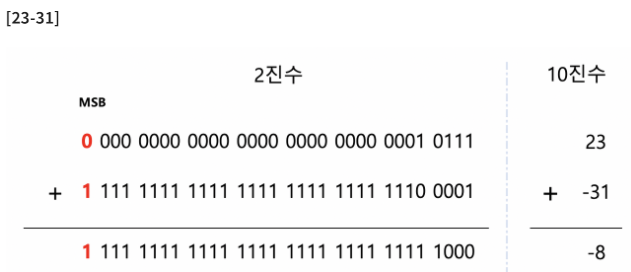
양수끼리의 덧셈은 문제될 것이 없다.



캐리가 발생해도 올바른 값이 나온다.



마찬가지로 캐리가 발생하지만 별 다른 조치 없이도 올바른 수가 나온다.



올바른 값이 바로 출력된다.

즉, 2의 보수법을 이용하면 캐리 발생시 아무런 조치를 취하지 않아도 올바른 값이 나오는 것을 볼 수 있다. 이렇게 2진수에서는 컴퓨터가 연산을 더욱 편리하게 하기 위해 2의 보수를 활용하여 쓰는 것이다.

실제로 대부분의 컴퓨터는 2의 보수를 활용한 방식을 택하고 있다. 2의 보수 방식의 장점을 정리하면 다음과 같다.

[장점]

(1) 1의 보수에 +1을 하면 음수값을 얻을 수 있다.

(2) MSB의 성질이 유지된다.

(3) 덧셈만으로 뺄셈을 구현할 수 있어 회로가 단순해진다.

(4) 1의 보수의 단점이 모두 해결된다.