컴퓨터 시스템 일반

Sangwook Lee
Deogi High School

I 정보의 표현

1 수의 체계

2 디지털 정보의 연산

3 디지털 정보의 표현

2 디지털 정보의 연산

1. 수치 데이터의 표현
2. 2진수의 연산

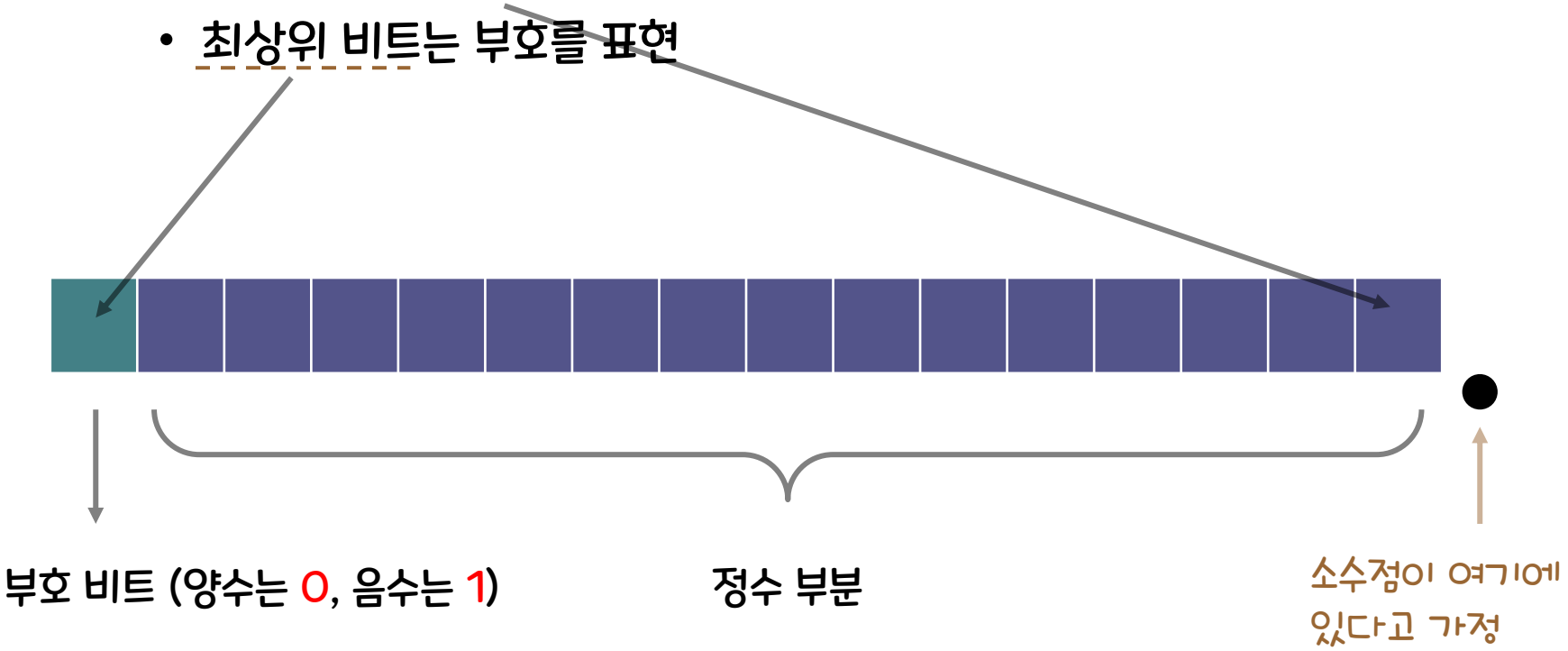
1. 수치 데이터의 표현

〈 학습 목표 〉

- 수치 데이터의 표현 원리를 설명할 수 있다.

1. 수치 데이터의 표현 (p.19)

- 2진수 정수를 표현하는 방법 (메모리에 저장하는 방법)
  - 고정 소수점(fixed point) 형식 사용
    - 소수점이 최하위 비트 오른쪽에 있다고 가정하고 표현하는 방식
    - 최상위 비트는 부호를 표현

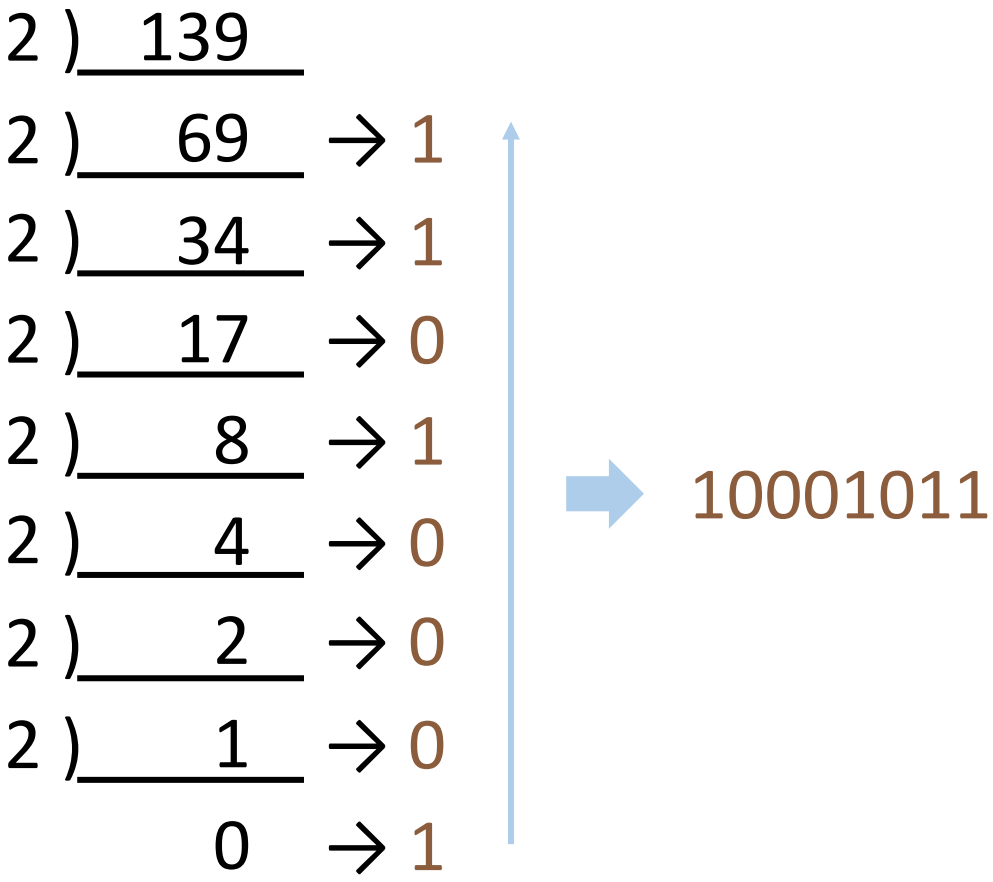


<2바이트 고정 소수점 데이터 형식>

1. 수치 데이터의 표현 (p.19)

예제) -139를 2바이트 고정 소수점 형식으로 표현

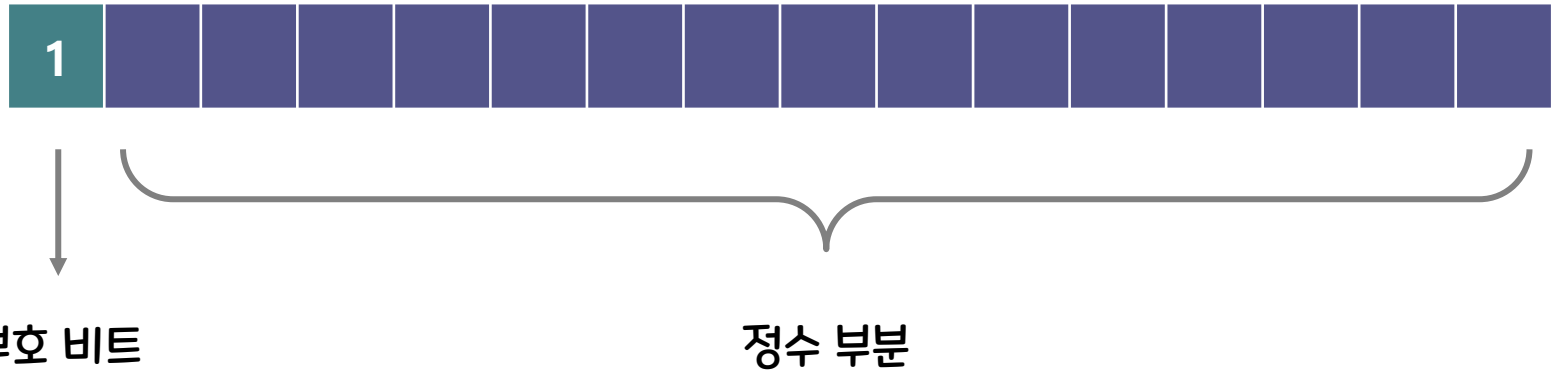
① 10진수 139를
2진수로 변환



1. 수치 데이터의 표현 (p.19)

예제) -139를 2바이트 고정 소수점 형식으로 표현

② 최상위 비트를 음수(1)로 표현



1. 수치 데이터의 표현 (p.19)

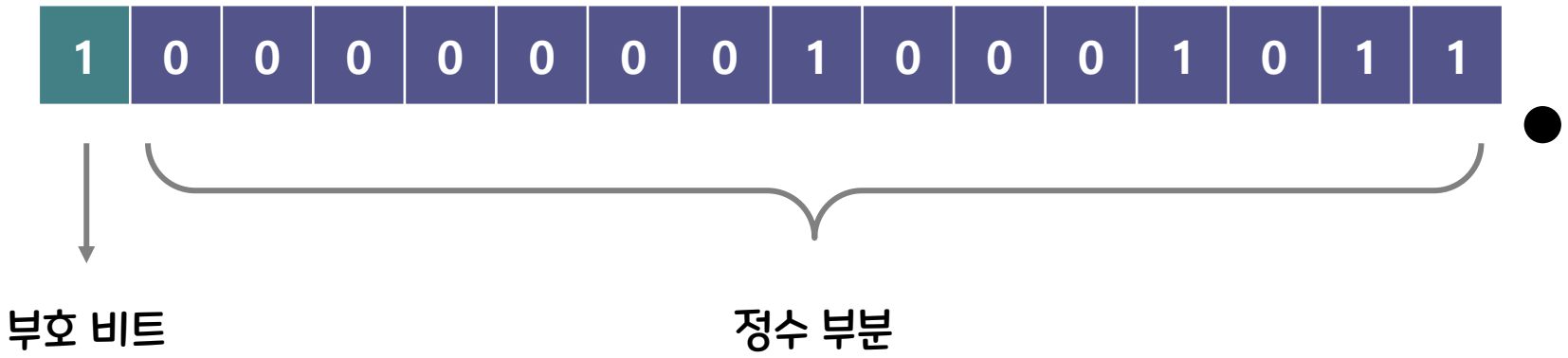
예제) -139를 2바이트 고정 소수점 형식으로 표현

③ 10001011을 정수 부분의 오른쪽(최하위 비트)부터 표현

1. 수치 데이터의 표현 (p.19)

예제) -139를 2바이트 고정 소수점 형식으로 표현

④ 나머지 부분을 0으로 채움



1. 수치 데이터의 표현 (p.19)

예제) -139를 2바이트 고정 소수점 형식으로 표현

따라서, $-139_{(10)}$를

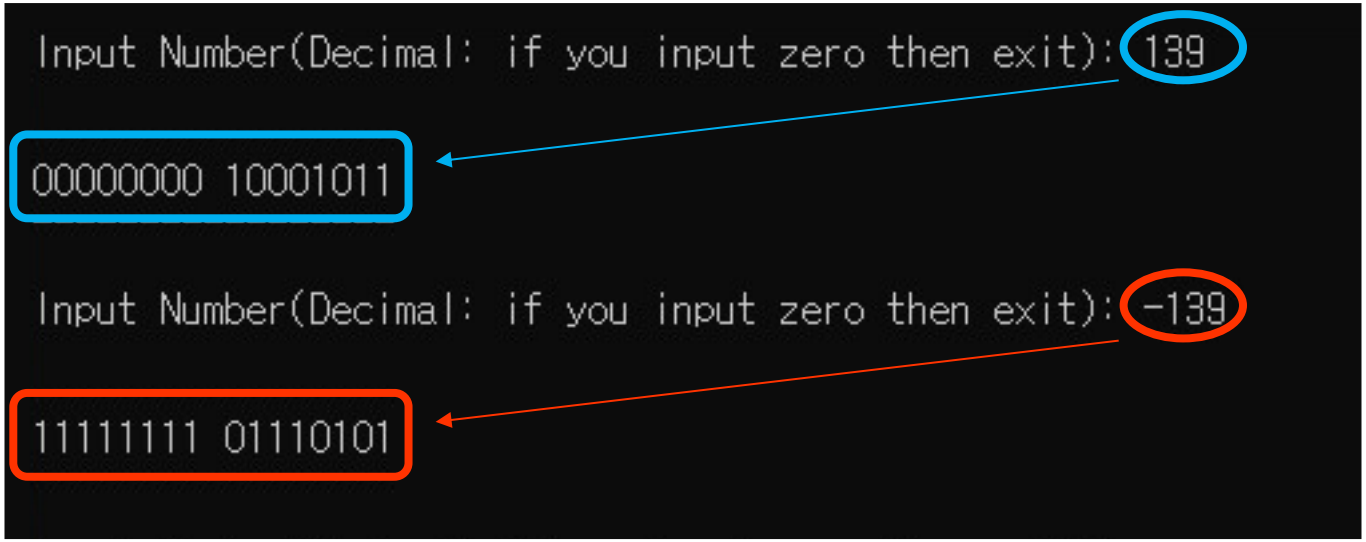
2바이트 고정 소수점 데이터 형식으로 표현하면..

$10000000\ 10001011_{(2)}$

1. 수치 데이터의 표현 (p.19)

예제) -139를 2바이트 고정 소수점 형식으로 표현

$10000000\ 10001011_{(2)}$

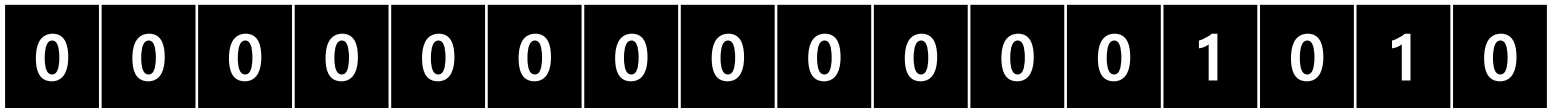


1. 수치 데이터의 표현 (p.19)

- 2진수의 음수를 표현하는 3가지 방법
  - 부호 절댓값 표현법
    - 양수 값의 최상위 자리 값을 1로 바꾸고, 나머지는 같게 표현하는 방법
  - 1의 보수 표현법
    - 양수 값의 0은 1로, 1은 0으로 바꾸어 표현하는 방법
  - 2의 보수 표현법
    - 양수 값의 0은 1로, 1은 0으로 바꾼 후 1을 더하여 표현하는 방법

1. 수치 데이터의 표현 (p.19)

예제) $1010_{(2)}$을 2바이트로 표현하면 아래와 같다.

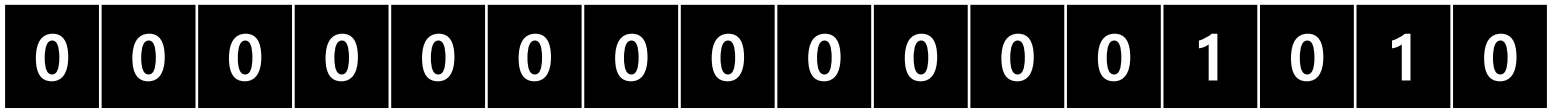


① $-1010_{(2)}$을 부호 절댓값 방식으로 표현하시오.

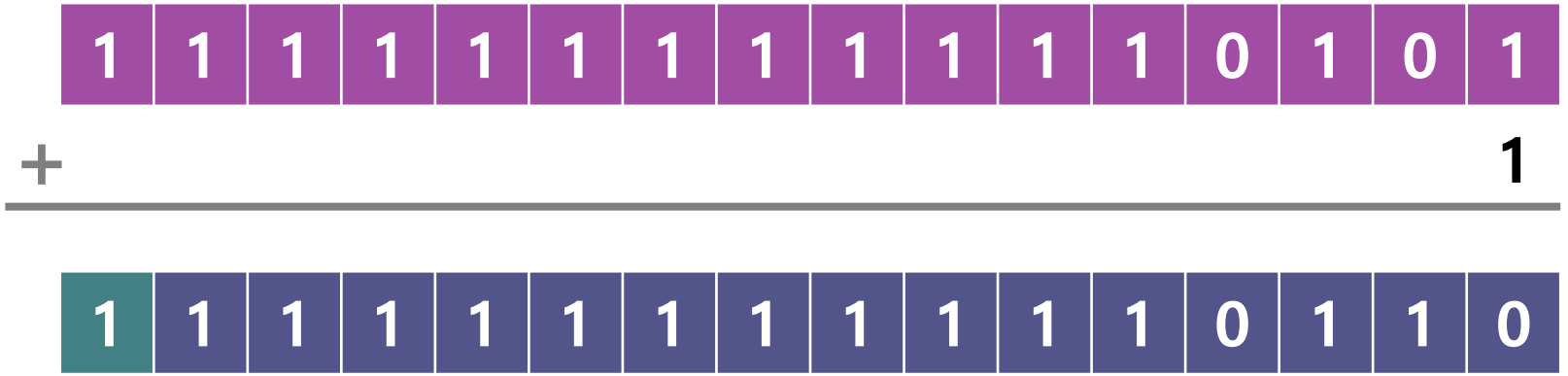
② $-1010_{(2)}$을 1의 보수 방식으로 표현하시오.

1. 수치 데이터의 표현 (p.19)

예제) $1010_{(2)}$을 2바이트로 표현하면 아래와 같다.



③ $-1010_{(2)}$을 2의 보수 방식으로 표현하시오.



음수 표현 방식 비교

- 3비트로 양수만 표현

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

표현 범위? 0 ~ 7

※ N 비트일 경우 표현 범위: 0 ~ (2^N-1)

음수 표현 방식 비교

- 3비트로 양수, 음수 표현 방법 ① 부호 절댓값 표현법

0	1	2	3	-0	-1	-2	-3
000	001	010	011	100	101	110	111

표현 범위? −3 ~ −0, 0 ~ 3

의미하는 값의 크기 순으로 재배치

-3	-2	-1	-0	0	1	2	3
111	110	101	100	000	001	010	011

※ N 비트일 경우 표현 범위: $-(2^{N-1}-1) \sim (2^{N-1}-1)$

음수 표현 방식 비교

- 3비트로 양수, 음수 표현 방법 ② 1의 보수 표현법

0	1	2	3	-3	-2	-1	-0
000	001	010	011	100	101	110	111

표현 범위? −3 ~ −0, 0 ~ 3

의미하는 값의 크기 순으로 재배치

-3	-2	-1	-0	0	1	2	3
100	101	110	111	000	001	010	011

※ N 비트일 경우 표현 범위: $-(2^{N-1}-1) \sim (2^{N-1}-1)$

음수 표현 방식 비교

- 3비트로 양수, 음수 표현 방법 ③ 2의 보수 표현법

0	1	2	3	-4	-3	-2	-1
000	001	010	011	100	101	110	111

표현 범위? -4 ~ 3

의미하는 값의 크기 순으로 재배치

-4	-3	-2	-1	0	1	2	3
100	101	110	111	000	001	010	011

※ N 비트일 경우 표현 범위: $-(2^{N-1}) \sim (2^{N-1}-1)$

부호 절댓값 방식의 단점

1. 표현할 수 있는 수의 종류가 적다.

부호 절댓값 방식

$-(2^{N-1}-1) \sim (2^{N-1}-1)$

-3 ~ 3

0	1	2	3	-0	-1	-2	-3
000	001	010	011	100	101	110	111

1의 보수 방식

$-(2^{N-1}-1) \sim (2^{N-1}-1)$

-3 ~ 3

0	1	2	3	-3	-2	-1	-0
000	001	010	011	100	101	110	111

2의 보수 방식

$-(2^{N-1}) \sim (2^{N-1}-1)$

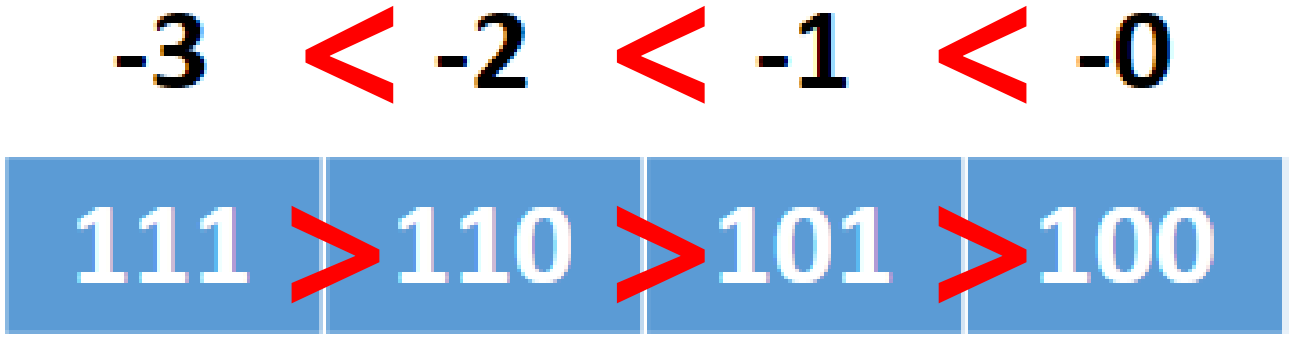
-4 ~ 3

0	1	2	3	-4	-3	-2	-1
000	001	010	011	100	101	110	111

부호 절댓값 방식의 단점

2. 음수의 크기 비교가 비효율적이다

메모리에 저장된 비트열의 크기 순서와
비트열이 의미하는 음수의 크기 순서가 상이

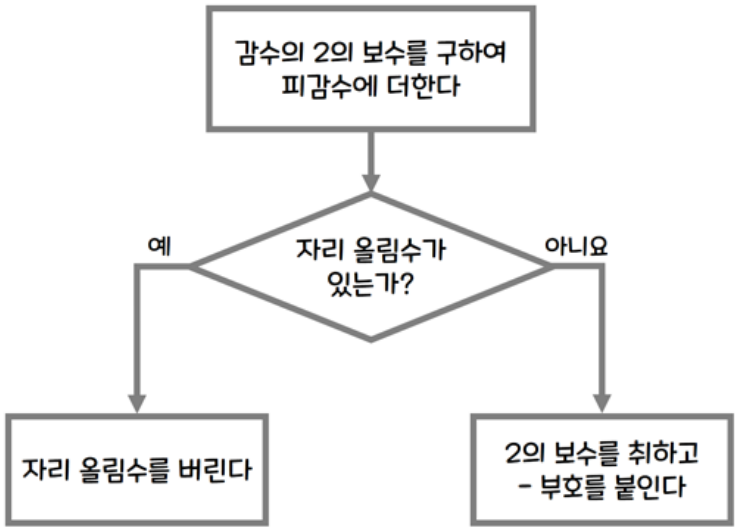


이렇게 되면, 양수의 크기를 비교하는 회로를 음수의 크기 비교 시 사용 못함

부호 절댓값 방식의 단점

3. 뺄셈 작업이 비효율적이다

☞ 1의 보수나 2의 보수 방식에서는 덧셈 회로를 사용하여 뺄셈이 가능하지만 부호 절댓값 방식에서는 불가능



※ 2의 보수에 의한 뺄셈 과정 (I-2-2 단원에서 배움)

1의 보수 방식과 비교 시, 2의 보수 방식의 장점

1. 더 많은 수를 표현할 수 있다

왜냐면, -0을 표현하지 않아도 되기 때문에

☞ 부호 절댓값 방식에서는 -0이 존재

-3	-2	-1	-0	0	1	2	3
111	110	101	100	000	001	010	011

☞ 1의 보수 방식에서도 -0이 존재

-3	-2	-1	-0	0	1	2	3
100	101	110	111	000	001	010	011

1의 보수 방식과 비교 시, 2의 보수 방식의 장점

2. 뺄셈 작업이 더 효율적이다

1의 보수를 사용하여 뺄셈을 할 경우

감수의 1의 보수와 피감수를 더했을 때 자리 올림수가 발생하면

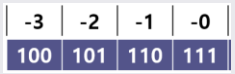
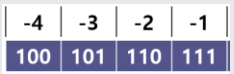
자리 올림수를 더해야 되지만

2의 보수를 사용하여 뺄셈을 할 경우에는

자리 올림수를 버리면 됨

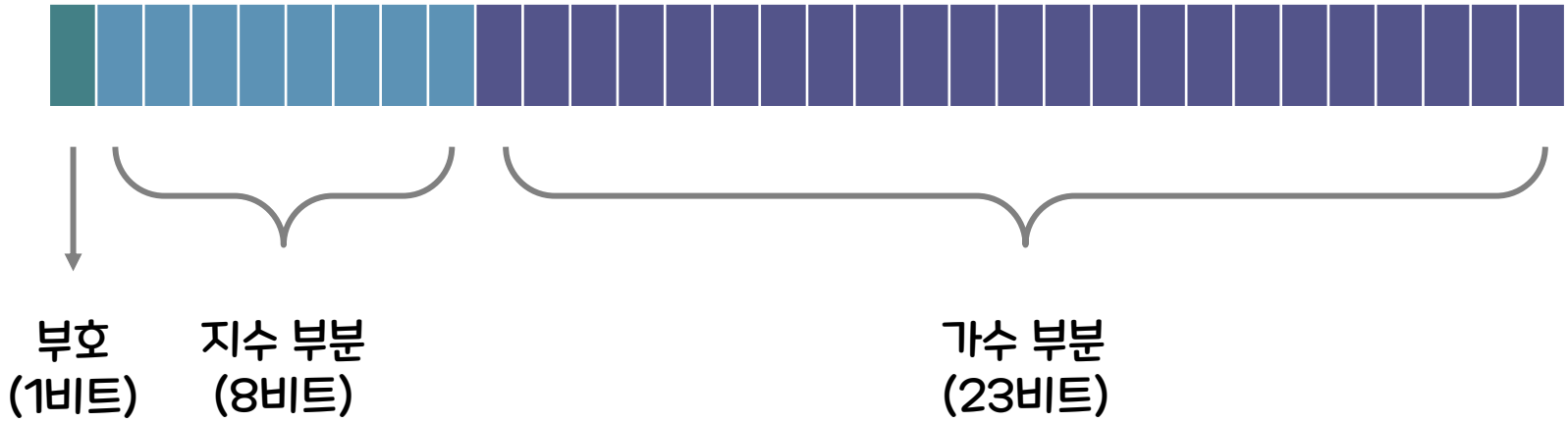
※ 뺄셈은 I-2-2 단원(2진수의 연산)에서 배움

음수 표현 방식 비교

	부호 절댓값	1의 보수	2의 보수
표현 가능한 수의 가짓수	$-(2^{N-1}-1) \sim (2^{N-1}-1)$ ★★	$-(2^{N-1}-1) \sim (2^{N-1}-1)$ ★★	$-(2^{N-1}) \sim (2^{N-1}-1)$ ★★★
음수 크기 비교 효율성	 ★	 ★★★	 ★★★
뺄셈 작업 용이성	★	★★	★★★
결론	★★★ ★	★★★★ ★★★	★★★★★ ★★★★

1. 수치 데이터의 표현 (p.20)

- **2진수 실수를 표현(저장)하는 방법**
  - 부동 소수점(floating point) 형식 사용
    - 실수를 정규화한 후, 보호, 지수, 가수(소수)로 나누어 표현하는 방식
    - 4바이트로 표현할 경우 부호는 1비트, 지수는 8비트, 가수는 23비트



<4바이트 부동 소수점 데이터 형식>

1. 수치 데이터의 표현 (p.20)

- **2진수를 부동 소수점 형식으로 표현하는 과정**
  - ① 정규화하기
    - 실수를 1.△△△… × 2□ 형태로 변환
      예) 2진수 -111.101을 정규화하면 -1.11101×2^2
  - ② 부호 부분 구하기
    - 실수가 음수(-)이면 → 1
    - 실수가 양수(+)이면 → 0
  - ③ 지수 부분 구하기
    - 정규화된 형태에서 □ 부분 + $(2^{n-1}-1)$
  - ④ 가수 부분 구하기
    - 정규화된 형태에서 △△△… 부분

?

n은 지수부 비트 수

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

① 10진수 123.25를 2진수로 변환

$$123_{(10)} = 1111011_{(2)}$$

$$0.25_{(10)} = 0.01_{(2)}$$

따라서,

$$-123.25_{(10)} = -1111011.01_{(2)}$$

(I-1-3 진법 변환 참조)

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

② 정규화

$$-1111011.01 = -1.11101101 \times 2^{6}$$

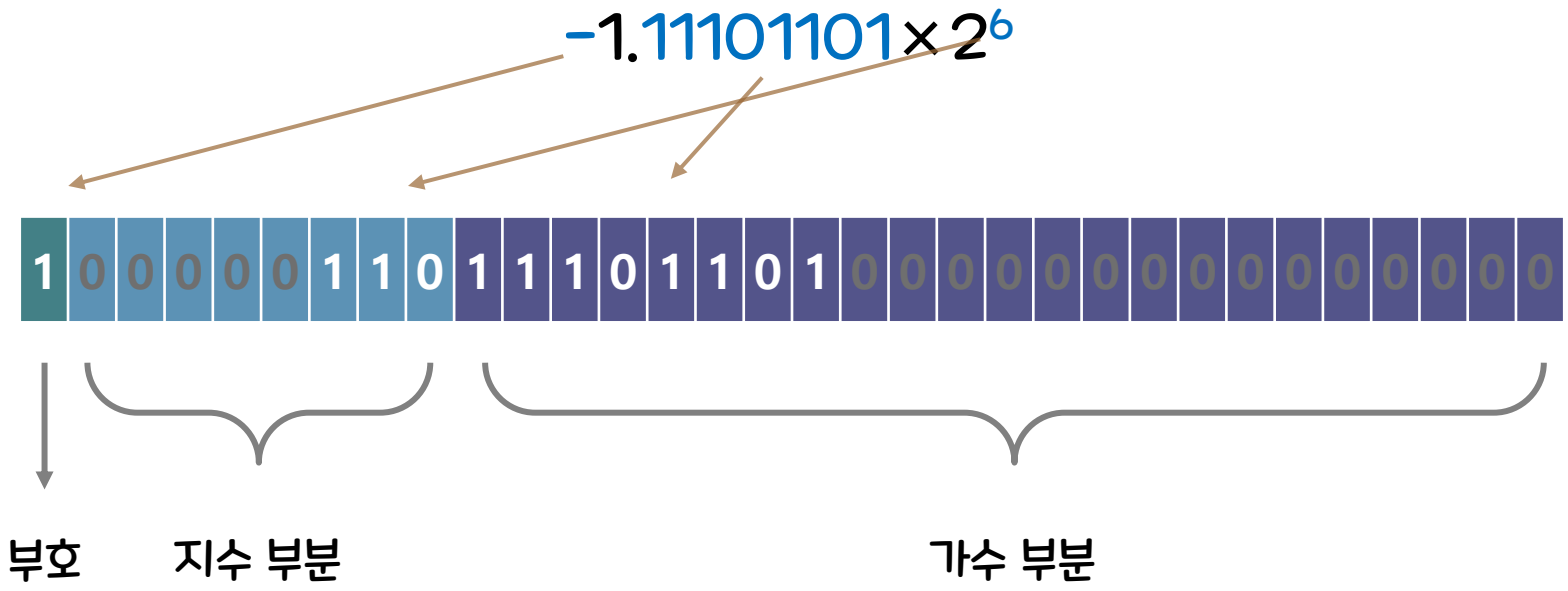
③ 부호, 지수, 가수 부분 2진수 값 계산

- 부호 - → 1
- 지수 6 → 110 (6) + 1111111 (127) = 10000101 (133)
- 가수 11101101 → 그대로 저장

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

④ 4바이트로 표현



지수 부분은 오른쪽부터 채우고, 가수 부분은 왼쪽부터 채움

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

따라서, $-123.25_{(10)}$를
4바이트 부동 소수점 데이터 형식으로 표현하면..

1 00000110 11101101000000000000000

지수에 127을 더하지 않았음!

지수 값 범위

- 8비트 크기의 지수부로 표현 가능한 지수 값 개수는?

$$2^8 = 256 \text{ (개)}$$

- 256개의 지수 값 범위를 정하면?

0 ~ 255

-155 ~ 100

-128 ~ 127

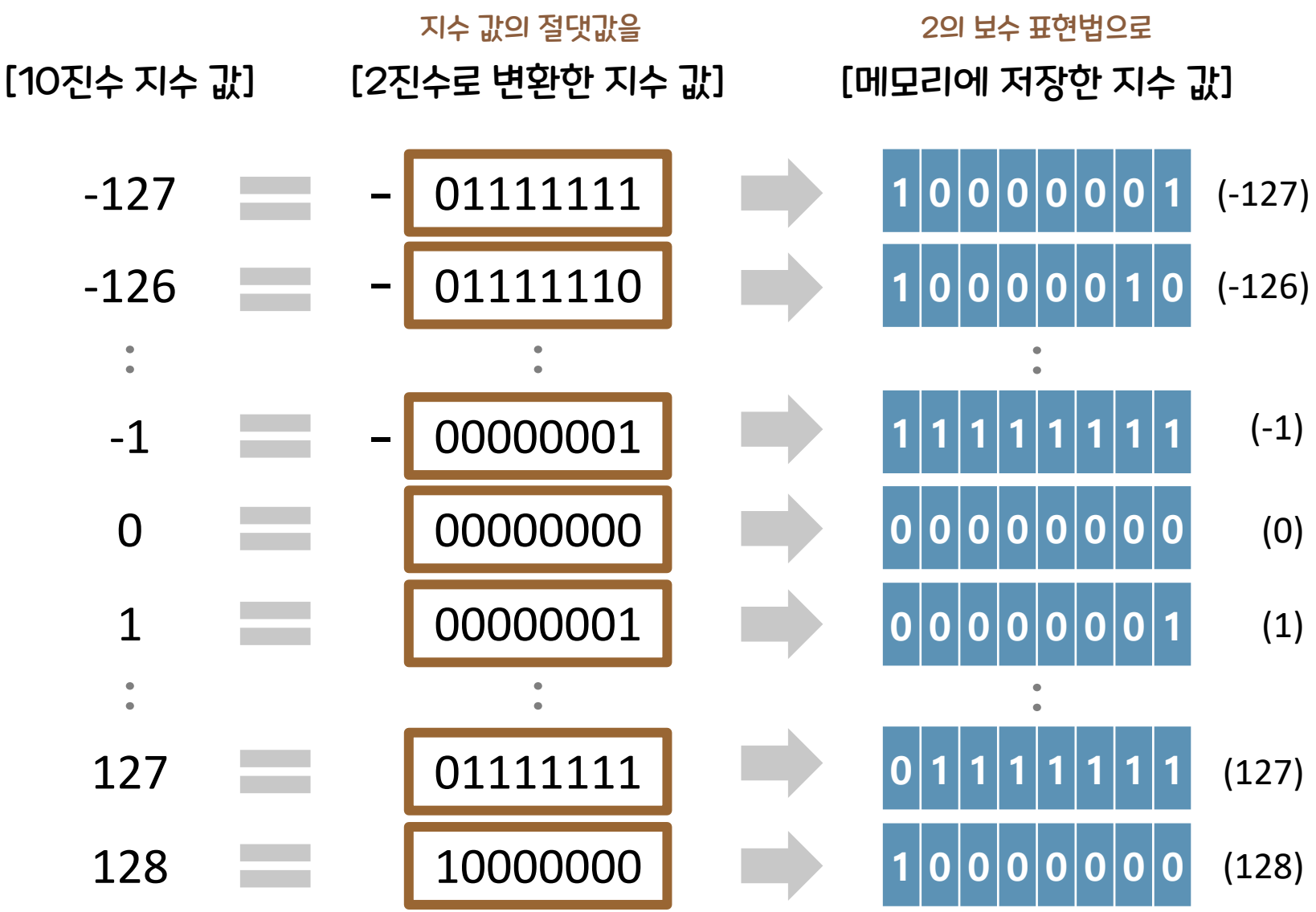
Q. 문제점?

A. 음수 지수를 표현하지 못하기 때문에
1보다 작은 2진수를 표현하지 못함

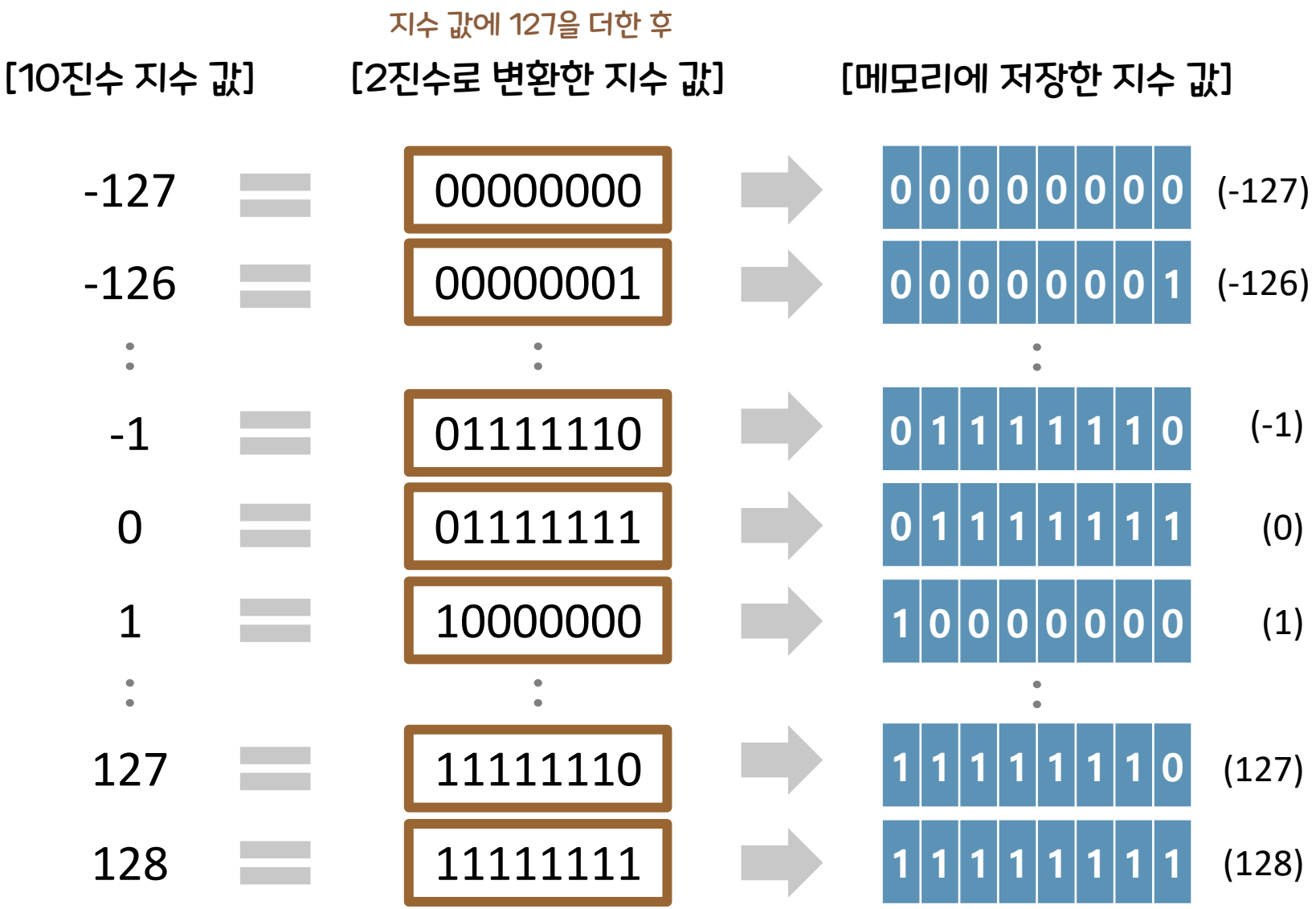
-127 ~ 128

표준으로 채택!

지수 값을 그대로 메모리에 저장한 경우



127을 더한 지수 값을 메모리에 저장한 경우



127을 더한 지수 값을 메모리에 저장한 경우

지수 값에 127($1111111_{(2)}$)을 더하여 저장하면..

최소 지수 -127부터 최대 지수 128까지가

메모리에

00000000부터 11111111까지

순서대로 표현됨

바이어스(bias) 상수

지수 부분이 n비트인 부동 소수점 표현 방식에서

최소 지수 $-(2^{n-1}-1)$부터 최대 지수 2^{n-1}까지가

메모리에

0……0부터 1……1까지

순서대로 표현되도록 지수에 더하는 값

$$2^{n-1}-1$$

바이어스 상수를 더한 지수를 바이어스된 지수(biased exponent)라고 함

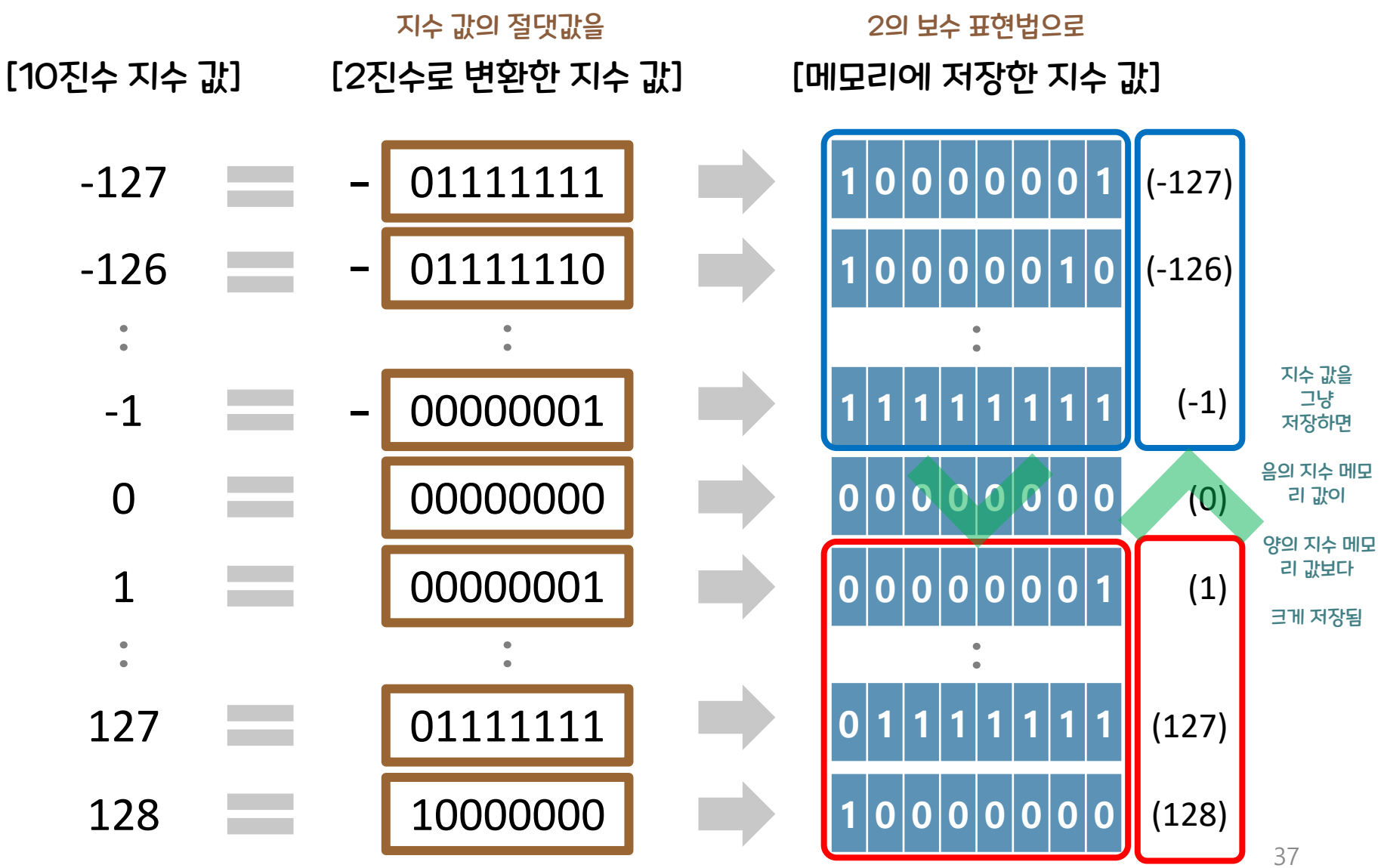
바이어스된 지수를 사용하는 목적

지수 값을 그대로 메모리에 저장하면
음의 지수의 메모리 값이 양의 지수의 메모리 값보다 크기 때문에
의미하는 값의 크기 순서와 저장된 값의 크기 순서가 불일치

이렇게 되면..

부호가 다른 두 지수의 크기를 비교할 때
양의 지수끼리(또는 음의 지수끼리) 비교하는 방식으로
메모리 값의 크기를 비교할 수 없음

바이어스된 지수를 사용하는 목적



바이어스된 지수를 사용하는 목적

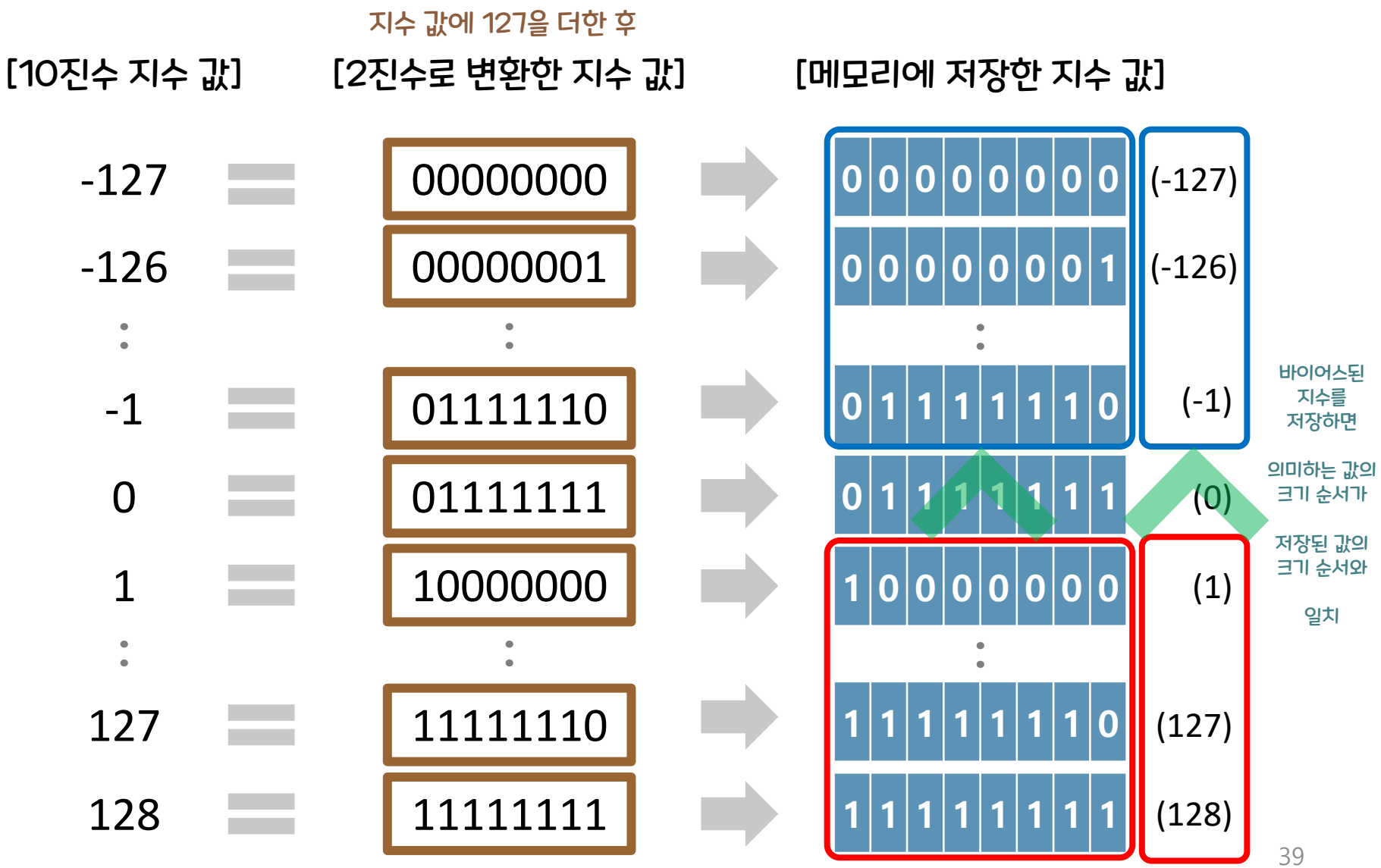
바이어스된 지수를 사용하여
의미하는 값의 크기 순서와 저장된 값의 크기 순서를 일치시키면

두 지수의 부호가 다르더라도
양의 지수끼리(또는 음의 지수끼리) 비교하는 방식으로
두 지수의 크기를 비교할 수 있기 때문에

실수의 크기 비교를 효율적으로 할 수 있음

“
양수 실수끼리 크기를 비교할 때는 지수부가 크면 더 큰 수,
음수 실수끼리 비교할 때는 지수부가 작으면 더 큰 수
”

바이어스된 지수를 사용하는 목적



1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

① 10진수 -123.25를 2진수로 변환

$$123_{(10)} = 1111011_{(2)}$$

$$0.25_{(10)} = 0.01_{(2)}$$

따라서,

$$-123.25_{(10)} = -111011.01_{(2)}$$

(1.1.3 진법 변환 참조)

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

② 정규화

$$-1111011.01 = -1.11101101 \times 2^{6}$$

③ 부호, 지수, 가수 부분 2진수 값 계산

- 부호 - → 1
- 지수 6 + 127 → 110 (6) + 1111111 (127)
- 가수 11101101 → 그대로 저장

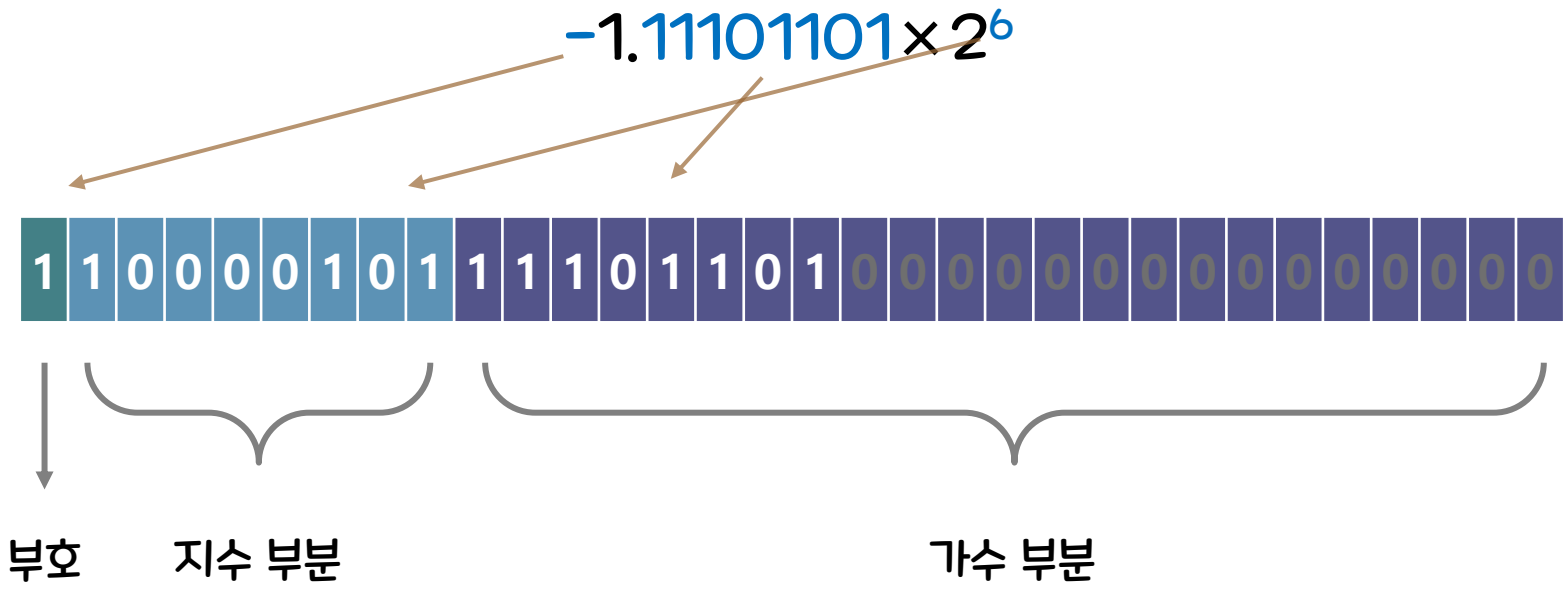
$$\begin{array}{r} 1111110 \\ 110 \\ + \quad 1111111 \\ \hline 10000101 \end{array}$$

(133)

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

④ 4바이트로 표현



지수 부분은 오른쪽부터 채우고, 가수 부분은 왼쪽부터 채움

1. 수치 데이터의 표현 (p.19)

예제) -123.25를 4바이트 부동 소수점 형식으로 표현

따라서, $-123.25_{(10)}$를
4바이트 부동 소수점 데이터 형식으로 표현하면..

1 10000101 11101101000000000000000

2 디지털 정보의 연산

1. 수치 데이터의 표현
2. 2진수의 연산

2. 2진수의 연산

〈 학습 목표 〉

- 2진수의 덧셈 연산 처리 원리를 설명할 수 있다.
- 2진수의 뺄셈 연산 처리 원리를 설명할 수 있다.
- 2진수의 곱셈 연산 처리 원리를 설명할 수 있다.

2. 2진수의 연산 (p.23)

- **덧셈**
  - 최하위 자리부터 같은 자리의 수끼리 더함
    - ☞ 더한 결과가 2이상이면 자리 올림 수 발생
      예) $1_{(2)} + 1_{(2)} = 10_{(2)}$ 이 되어 바로 위 자리로 1이 올라감
    - ☞ 최상위 자리까지 같은 자리의 합을 구함

자리 올림 수(carry)란..

어떤 숫자 위치(digit position)의 합 또는 곱이
그 위치에서 표시할 수 있는 최고 큰 수를 넘을 때,
상위 위치에서의 처리를 위해 보내지는 수

2. 2진수의 연산 (p.23)

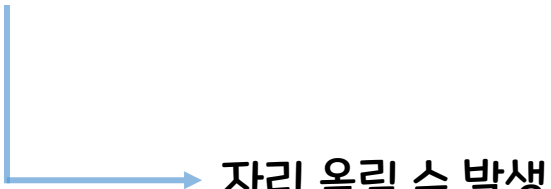
- 덧셈
  - 2진수의 덧셈 규칙

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10



2. 2진수의 연산 (p.23)

예제) 2진수 1011과 0101 덧셈

$$
\begin{array}{r}
{\scriptstyle 1\ \ 1\ \ 1\ \ 1} \\
1011 \\
+)\ \ 0101 \\
\hline
10000
\end{array}
$$

$\therefore\ 1011_{(2)} + 0101_{(2)} = 10000_{(2)}$

2. 2진수의 연산 (p.24)

- **1의 보수를 사용한 뺄셈**

  ① 감수의 1의 보수를 구함

감수와 피감수란..

감수는 빼는 수, 피감수는 뺌을 당하는 수

(예)

0111 - 0100

피감수 감수

0100의 1의 보수는?

1011

2. 2진수의 연산 (p.24)

- 1의 보수를 사용한 뺄셈

  ② 감수의 1의 보수와 피감수를 더함

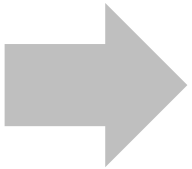
```
  1 1 1 1
    0111
+)  1011
-------
   10010
```

2. 2진수의 연산 (p.24)

- 1의 보수를 사용한 뺄셈

  ③ 자리 올림 수가 발생하면 자리 올림수를 더함
  (※ 발생하지 않으면 1의 보수를 취하고 '-' 를 붙임)

$$\begin{array}{r} 0111 \\ +)\ 1011 \\ \hline 10010 \end{array}$$



$$\begin{array}{r} 0010 \\ +)\ \quad 1 \\ \hline 0011 \end{array}$$

$$\therefore 0111_{(2)} - 0100_{(2)} = 0011_{(2)}$$

2. 2진수의 연산 (p.24)

(예제) $0100_{(2)} - 0111_{(2)} = ?$

풀이〉

① 감수 0111의 1의 보수를 구함

0111의 1의 보수는 1000

② 피감수와 감수의 1의 보수를 더함

```
  0 0 0 0
    0100
+ )  1000
-------
   01100
```

자리 올림수 발생하지 않음

2. 2진수의 연산 (p.24)

(예제) $0100_{(2)} - 0111_{(2)} = ?$

풀이〉

③ 자리 올림수가 없기 때문에 1100의 1의 보수를 구함

1100의 1의 보수는 0011

④ '-' 를 붙임

-0011

$$\therefore 0100_{(2)} - 0111_{(2)} = -0011_{(2)}$$

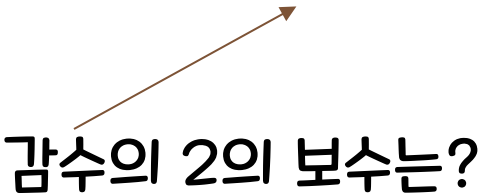
2. 2진수의 연산 (p.24)

- **2의 보수를 사용한 뺄셈**

  ① 감수의 2의 보수를 구함

(예)

0111 - 0100



감수의 2의 보수는?

0100의 1의 보수(1011) + 1 = 1100

2. 2진수의 연산 (p.24)

- **2의 보수를 사용한 뺄셈**

  ② 감수의 2의 보수와 피감수를 더함

```
  1 1 0 0
    0111
+ ) 1100
-------
   10011
```

2. 2진수의 연산 (p.24)

- **2의 보수를 사용한 뺄셈**

  ③ 자리 올림 수가 발생하면 버림

  (※ 발생하지 않으면 2의 보수를 취하고 '-' 를 붙임)

$$\begin{array}{r} 0111 \\ +)\ 1100 \\ \hline 10011 \end{array}$$

$$\therefore 0111_{(2)} - 0100_{(2)} = 0011_{(2)}$$

2. 2진수의 연산 (p.24)

(예제) $0100_{(2)} - 0111_{(2)} = ?$

풀이〉

① 감수 0111의 2의 보수를 구함

1000(1의 보수) + 1 = 1001

② 피감수와 감수의 2의 보수를 더함

```
  0 0 0 0
    0100
+ )  1001
-------
   01101
```

자리 올림수 발생하지 않음

2. 2진수의 연산 (p.24)

(예제) $0100_{(2)} - 0111_{(2)} = ?$

풀이〉

③ 자리 올림수가 없기 때문에 1101의 2의 보수를 구함

0010(1의 보수) + 1 = 0011

④ '-' 를 붙임

- 0011

$$\therefore 0100_{(2)} - 0111_{(2)} = -0011_{(2)}$$

2. 2진수의 연산 (p.25)

- **곱셈**
  - ① 승수의 최하위 자리부터 피승수와의 곱을 구함
    - ☞ 곱한 결과의 최하위 자리를 승수의 자리와 맞춤
  - ② 승수의 자리별로 구한 곱들을 더함

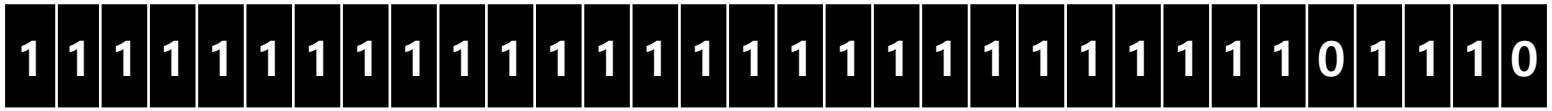
```
    1001
×)   101
------
    1001
   0000
  1001
------
  101101
```

Backup Slides

[예제] 정수 $-17_{(10)}$을 4바이트로 표현하시오

① 양수 $17_{(10)}$을 2진수로 표현

② 0은 1, 1은 0으로 바꿈 (☞ 1의 보수 표현법)



[예제] 정수 $-17_{(10)}$을 4바이트로 표현하시오

③ 1을 더함 (☞ 2의 보수 표현법)

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

\+ 1

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

-17이 메모리에 저장된 모양을
눈으로 직접 확인해 보고 싶은데…

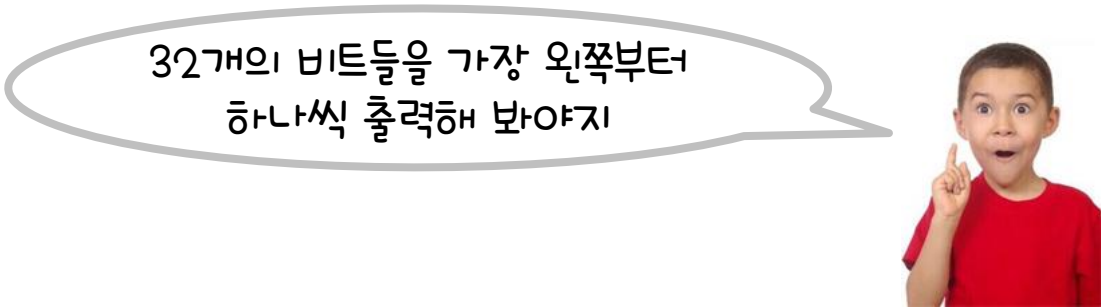
정수의 비트 열을 출력하는 C언어 소스코드

```
#include <stdio.h>

main()
{
            int input = 0;
            int mask;

            while (1)
            {
                        printf("정수: ");
                        scanf("%d", &input);

                        int i;
                        for (i = 31; i >= 0; i--)
                        {
                                    mask = 1 << i;
                                    printf("%d", input & mask ? 1 : 0);
                        }
                        printf("\n\n");
            }
}
```



정수의 비트 열을 출력하는 C언어 소스코드

```
#include <stdio.h>

main()
{
        int input = 0;
        int mask;

        while (1)
        {
                printf("정수: ");
                scanf("%d", &input);

                int i;
                for (i = 31; i >= 0; i--)
                {
                        mask = 1 << i;
                        printf("%d", input & mask ? 1 : 0);
                }
                printf("\n\n");
        }
}
```

printf 소스코드와 scanf 소스코드를 가져옴

사용자가 입력한 값을 저장할 변수 input 만듦

출력할 비트의 위치를 저장할 변수 mask 만듦

입력 받은 정숫값을 input에 저장

32개의 비트를 출력하기 위한 반복문

출력할 비트 위치 지정 연산

지정된 위치의 비트 출력

32번 반복하여 출력하면, input 값의 전체 비트가 됨

C언어 삼항 연산자

- 문법

조건식 ? 반환값1 : 반환값2

물음표 앞의 조건식의 결괏값이
참이면 반환값1을 반환하고, 거짓이면 반환값2를 반환

- 예제 1

```
int num1 = 15;
int num2 = 8;
int result;

result = (num1 > num2) ? num1 : num2;
printf("둘 중에 더 큰수는 %d입니다.\n", result);
```

num1 〉num2 값이 참이면 num1 값을 반환
num1 〉num2 값이 거짓이면 num2 값을 반환

<실행 결과>

```
둘 중에 더 큰수는 15입니다.
```

C언어 삼항 연산자

- 예제 2

```
int num1 = 5;
int num2;

if (num1)
    num2 = 100;
else
    num2 = 200;

printf("%d\n", num2);
```

num1이 참이면 num2에 100을 할당
num1이 거짓이면 num2에 200을 할당

<실행 결과>

```
100
```

- 예제 3

```
int num1 = 5;
int num2;

num2 = num1 ? 100 : 200;

printf("%d\n", num2);
```

num1이 참이면 num2에 100을 할당
num1이 거짓이면 num2에 200을 할당

<실행 결과>

```
100
```

input & mask ? 1 : 0 결괏값

(입력 받은 값은 -17이라고 가정)

- i = 31일 때,

```
input = -17     ☞ 11111111 11111111 11111111 11101111
mask = 1 << i   ☞ 10000000 00000000 00000000 00000000
-------------------------------------------------------
input & mask    ☞ 10000000 00000000 00000000 00000000
```

여기서 **input & mask** 논리값은? **참**

(※ 0이면 거짓, 0이 아니면 참)

따라서 **input & mask ? 1 : 0** 값은? **1**

input & mask가 참이면 1
input & mask가 거짓이면 0

input & mask ? 1 : 0 결괏값

(입력 받은 값은 -17이라고 가정)

- i = 4 일 때,

```
input = -17     ☞ 11111111 11111111 11111111 11101111
mask = 1 << i   ☞ 00000000 00000000 00000000 00010000
-------------------------------------------------------
input & mask    ☞ 00000000 00000000 00000000 00000000
```

여기서 **input & mask** 논리값은? 거짓 (※ 0이면 거짓, 0이 아니면 참)

따라서 **input & mask ? 1 : 0** 값은? 0

결론적으로..
mask의 1의 위치에 있는 input 비트가 1이면 결괏값은 1
mask의 1의 위치에 있는 input 비트가 0이면 결괏값은 0

input & mask ? 1 : 0 결괏값

mask의 1의 위치와
동일 위치에 있는 **input 비트**가 **1** 이면
input & mask ? 1 : 0 값은 **1** 이 됨

mask의 1의 위치와
동일 위치에 있는 **input 비트**가 **0** 이면
input & mask ? 1 : 0 값은 **0** 이 됨

결론적으로
mask 값이 2^{31}부터 2^{0}까지 32번 변하는 동안
input & mask ? 1 : 0 값은 **input 비트 열**이 됨

[예제] $4.5_{(10)}$를 4바이트로 표현하시오

① 2진수로 변환

$$4_{(10)} = 100_{(2)}$$

$$0.5_{(10)} = 0.1_{(2)}$$

따라서,

$$4.5_{(10)} = 100.1_{(2)}$$

[예제] $4.5_{(10)}$를 4바이트로 표현하시오

② 정규화

$$100.1 = +1.001 \times 2^2$$

③ 부호, 지수, 가수 부분 2진수로 변환

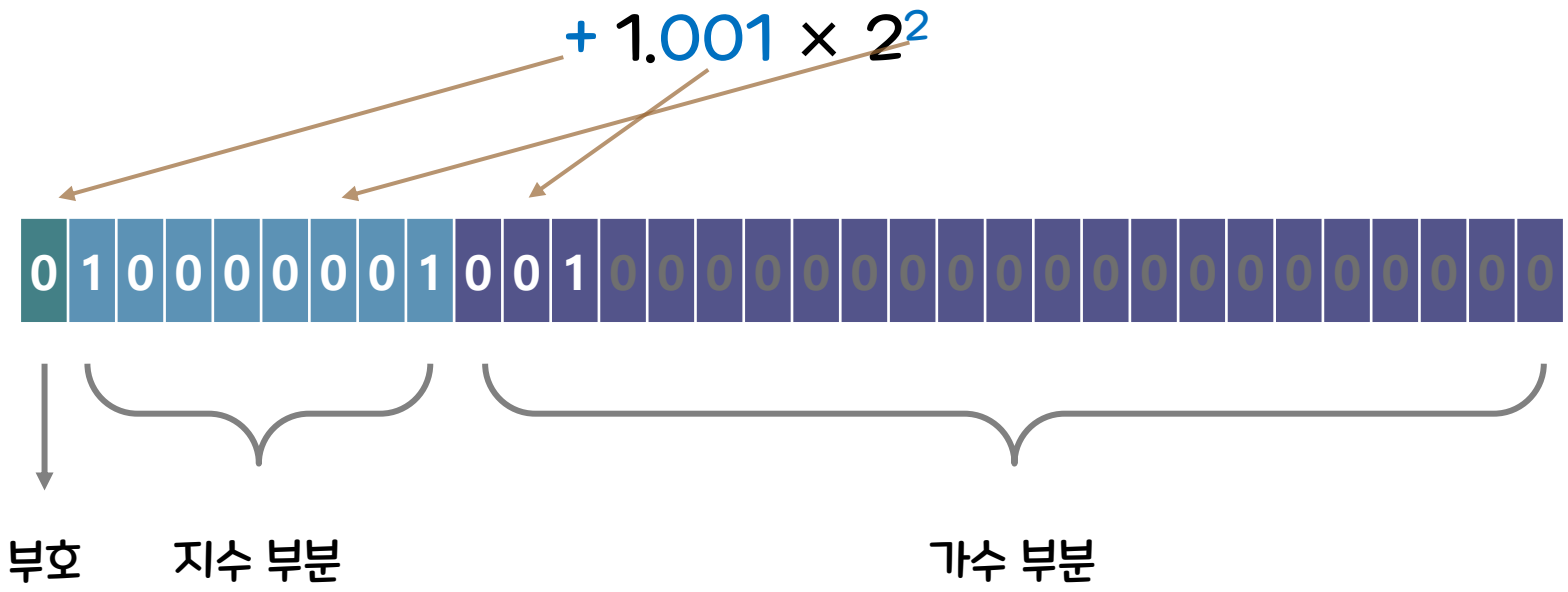
- 부호 + → 0
- 지수 2 → $10_{(2)} + 1111111_{(127)} = 10000001_{(129)}$
- 가수 001 → 001

$$\begin{array}{r} 1111110 \\ 10 \\ +\ 1111111 \\ \hline 10000001 \end{array}$$

(129)

[예제] $4.5_{(10)}$를 4바이트로 표현하시오

④ 4바이트로 표현



지수 부분은 오른쪽부터 채우고, 가수 부분은 왼쪽부터 채움

실수의 비트 열을 출력하는 소스코드

```
#include <stdio.h>

main()
{
            float input = 0;
            int mask;

            while (1)
            {
                        printf("실수: ");
                        scanf("%f", &input);

                        int i;
                        for (i = 31; i >= 0; i--)
                        {
                                    mask = 1 << i;
                                    printf("%d", input & mask ? 1 : 0);

                                    if (i==31 | i==23) printf(" ");
                        }
                        printf("\n\n");
            }
}
```

입력 받은 실숫값을
input 변수에 저장하는 코드

? 부호, 지수부, 가수부 사이를
한 칸 띄우기 위한 코드

C언어로 프로그램을 개발할 때 필요한 도구

- **코드 편집기**
  - Visual Studio Code
  - Vim

역할 → **코딩 (coding)**

프로그래밍 언어를 사용하여 프로그램을 작성하는 과정

- **컴파일러**
  - GCC
  - Clang

역할 → **컴파일 (compilation)**

고수준언어(C, Java 등)로 작성된 프로그램을 기계어로 변환하는 과정

- **통합 개발 환경 (IDE)**
  - Visual Studio
  - Dev-C++

역할 → **코딩+컴파일**

C언어 컴파일러

- GCC (GNU Compiler Collection)
  - 리눅스 기본 컴파일러 (윈도우/macOS도 지원)
  - Clang보다 플랫폼 호환성 좋음
  - Clang보다 컴파일 속도 느리고, 에러 메시지가 이해하기 어려움
- Clang ("클랭")
  - macOS 기본 컴파일러 (윈도우/리눅스도 지원)
  - GCC보다 플랫폼 호환성 나쁨
  - GCC보다 컴파일 속도 빠르고, 에러 메시지가 이해하기 쉬움
- CL (cl.exe)
  - Visual Studio에서 사용하는 C 컴파일러
  - 보통 "Visual C++ Compiler", 또는 "MSVC 컴파일러"라고 함