18차시

Bitmasking

Bit

- 컴퓨터는 0과 1로 이루어져 있다
- 0과 1을 저장하는데, 0 또는 1 데이터 하나, 또는 저장하는 공간의 단위: Bit
- 1 bit: 0, 1
- 2 bit: 00, 01, 10, 11
- 3 bit: 000, 001, 010, 011, 100, 101, 110, 111
- 4 bit: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, ...

자료형

종류	자료형	크기
정수형 signed/unsigned	char	1 Byte
	bool	1 Byte
	short	2 Byte
	int	4 Byte
	long	4/8 Byte
	long long	8 Byte
실수형 (부동소수점)	float	4 Byte
	double	8 Byte

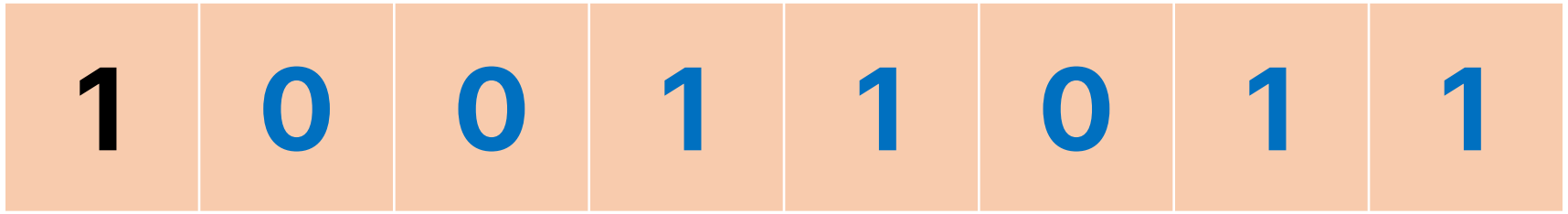
정수를 나타내는 법을 배우자

- 정수의 경우 절대값를 나타내는 부분과 부호를 나타내는 부분으로 나뉜다
- 부호의 경우, 제일 앞에 비트 하나를 가지고 0인 경우 양수로, 1인 경우 음수로 표현한다

1	0	0	1	1	0	1	1

정수를 나타내는 법을 배우자

- 부호비트를 제외한 나머지의 부분은 절대값을 표시한다
- 비트는 2진수로 계산된다
- 오른쪽부터 왼쪽으로 점점 커진다(10진수와 동일)



정수를 나타내는 법을 배우자

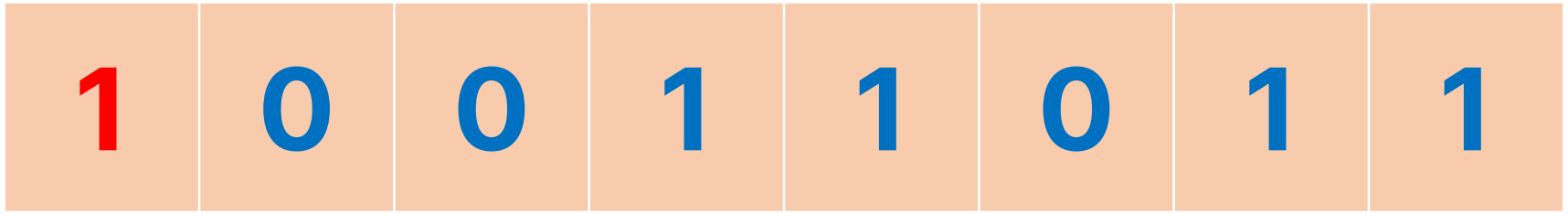
- 10진수가 오른쪽 자리부터 10^0, 10^1, 10^2, 10^3, 10^4, …을 나타내는 것과 동일하다
- 2진수에서는 오른쪽 자리부터 2^0, 2^1, 2^2, 2^3, 2^4, …을 나타낸다
- 따라서 아래의 수는 다음과 같이 계산할 수 있다

$$0\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$

1	0	0	1	1	0	1	1

정수를 나타내는 법을 배우자

- 따라서 비트가 k개일 때, $-2^{k-1} \sim 2^{k-1} - 1$의 범위를 나타낸다
- 양수의 범위에 1이 빠진 이유는 범위에 0을 포함하기 때문에 양수의 범위가 1 작다
- 간단하게 0와 -0이 공존하는데 0은 그대로 사용하며, -0을 음수의 제일 작은 수로 사용하는 것이다



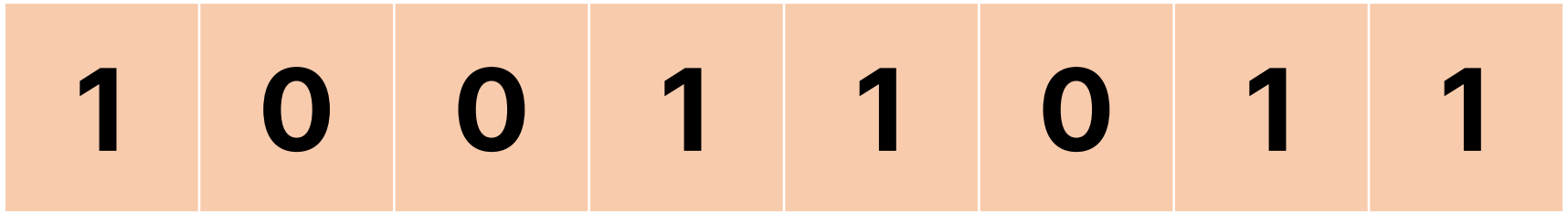
정수를 나타내는 법을 배우자

- 몇몇 변수들은 부호가 필요 없는 자료형이 존재한다
- 정확히는 음수가 필요 없는 자료형들이다
- unsigned 자료형들이 대표적인 예시들이다

$1_?$	0	0	1	1	0	1	1

정수를 나타내는 법을 배우자

- 양수만 나타내는 자료형의 경우, 부호 비트를 사용하지 않으므로 0부터 시작해 비트 전체를 이용해 절대값을 표현한다
- $0 \sim 2^k-1$의 범위를 표현한다(k는 비트의 수)
- int의 경우 $-2^{31} \sim 2^{31}-1$의 범위를 나태지만, unsigned int의 경우 $0 \sim 2^{32}-1$의 범위를 표현한다



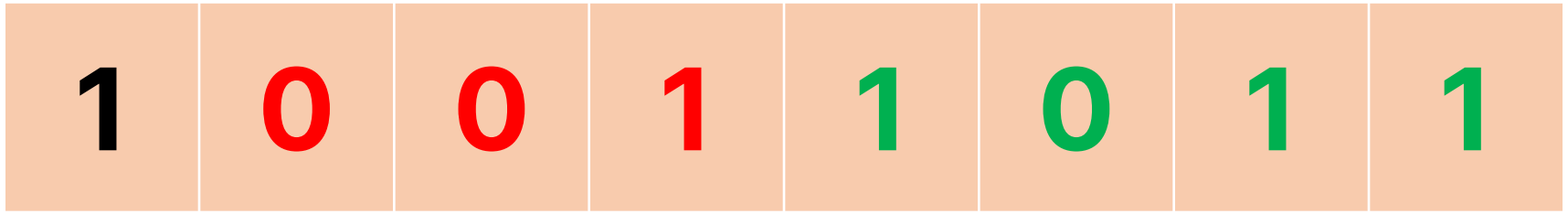
소수를 나타내는 법을 배우자

- Floating Point, 소수점의 경우 주로 사용하지 않으므로 간단하게 개념만 배우고 넘어가자
- 소수의 경우, 부동소수점 방식이라 표현하는데 이는 소수점이 움직인다는 뜻이다
- 정수와 마찬가지로 제일 왼쪽 비트는 부호를 나타낸다

1	0	0	1	1	0	1	1

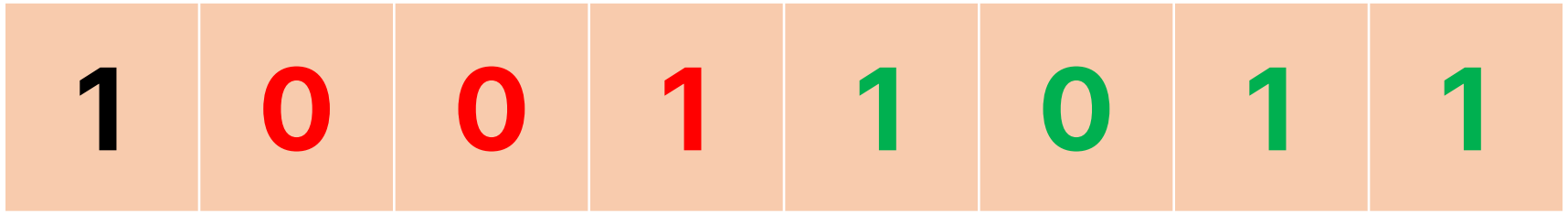
소수를 나타내는 법을 배우자

- 그 부호비트를 제외하고 나머지 비트를 두 범위로 나눈다
- 아래 예시의 경우 3개와 4개로 나누었는데, 이는 나타내려는 범위, 자세한 정도에 따라 바뀔 수 있다



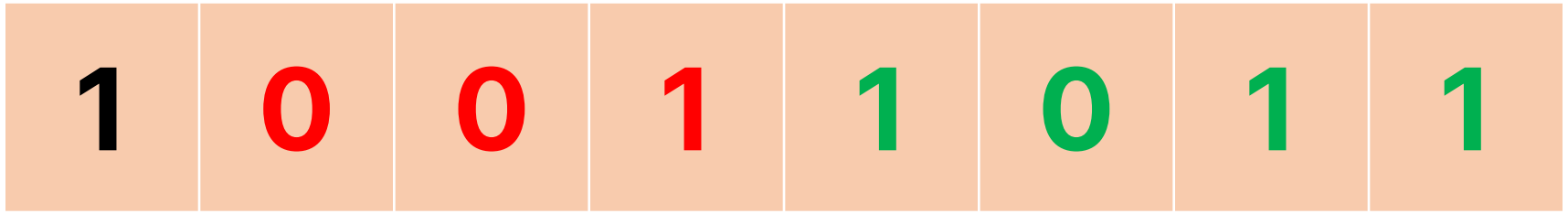
소수를 나타내는 법을 배우자

- 뒤의 초록색 부분부터 살펴보면 소수점 이후의 부분을 나타낸다
- 이진수로 1.X라는 기본 형태를 전제로 가지며, 아래의 예시의 경우 1.1011_2을 나타낸다
- 만약 초록색 부분이 0000인 경우, 1.0000을 나타내는 것이다



소수를 나타내는 법을 배우자

- 가운데 빨간색 부분은 소수점을 얼마나 이동시킬지 선택한다
- 일반적으로 전부 양수로 보며 시작 값은 0이 아닌 bias라는 특정 값이 지정되어 있다.
- 일반적으로 bias는 음수로 지정되어 있으며 이 둘을 합친 값만큼 움직이게 된다



소수를 나타내는 법을 배우자

- bias가 -4인 경우, 빨간 부분은 1을 나타내므로 -4 + 1, 즉 -3칸만큼 소수점을 움직이게 된다.
- 음수의 경우 작아지는 쪽으로 3칸을 움직이므로 1.1011_2의 소수점을 왼쪽으로 3칸 움직여 0.0011011_2를 나타내게 된다

1	0	0	1	1	0	1	1

소수를 나타내는 법을 배우자

- 하지만 기본 모양이 1.x인 형태이므로 0을 정확하게 표현할 수 없다
- 따라서 가운데 빨간 부분이 전부 0이거나 전부 1인 경우, special case를 만들어 0 또는 NaN(Not a Number), 무한 등을 나타낸다
- 소수점이 이동하는 형태이므로 덧셈이나 곱셈 등 연산이 더 복잡하다
- 직접적으로 많이 쓰이지 않으므로, 이런 식으로 해석한다 정도만 알고 있자

음수

- 지금까지 나타낸 방식은 전부 양수에 관련된 표현 방식이다
- 일반적으로 절대값을 나타내고, 부호를 붙여 아래의 예시를 -27이라고 표현할 수도 있지만 음수의 경우 연산의 편의성을 위해 보수 표현을 사용하고 있다
- 음수를 지금 배우지는 않지만 실제 컴퓨터에서는 다른 방식으로 음수를 사용한다고 알고 있자

1	0	0	1	1	0	1	1

비트 연산

- 일반적으로 우리가 수학을 배울 때, 가장 기초적인 연산을 사칙연산이라고 한다
- 사칙연산을 기반으로 더 어려운 미적분을 배우는 것처럼 비트에서 여러가지 사칙연산을 할 수 있지만 비트에서 사칙연산은 어려운 부분이다
- 비트에서 가장 기본이 되는 연산을 비트 연산이라고 한다
- 컴퓨터에서 숫자의 덧셈과 뺄셈 등은 비트 연산의 조합으로 만들어지게 된다

AND

- 논리곱
- 두 비트가 모두 1인 경우 1, 그 외의 경우 0을 출력한다.
- 일반적으로 코드에서는 &를 사용한다

A	∧	B	=	Result
0		0		0
0		1		0
1		0		0
1		1		1

OR

- 논리합
- 두 비트 중 하나라도 1인 경우 1, 다른 경우 0을 출력한다.
- 일반적으로 코드에서는 |를 사용한다

A	V	B	=	Result
0		0		0
0		1		1
1		0		1
1		1		1

NOT

- 논리 부정
- 값을 뒤집으려는 경우 사용
- 일반적으로 코드에서는 !를 사용한다
- 다른 비트 연산자들은 두 비트를 합쳐서 사용하지만 유일하게 비트 하나에 대해 직접 연산

! A	= Result
0	1
1	0

XOR

- 배타적 논리합
- 두 비트가 같은 경우 1, 다른 경우 0을 출력한다.
- 일반적으로 코드에서는 ^를 사용한다

A ⊕	B =	Result
0	0	0
0	1	1
1	0	1
1	1	0

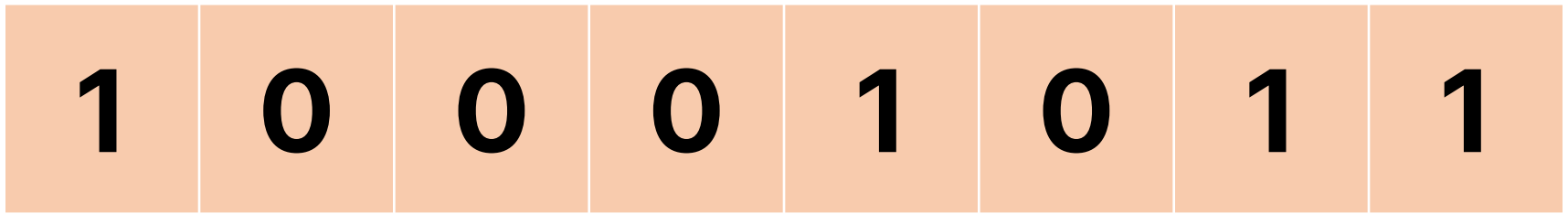
Shift 연산자

- 비트를 이동시키는 연산자
- 방향에 따라서 << 와 >>를 사용한다
- ‘움직일 비트 또는 변수' ‘연산자(<< 또는 >>)’ ’움직일 거리’ 순서로 적는다

  ex) tmp << 1; tmp 변수의 비트를 전부 왼쪽으로 한 칸 움직인다

Shift 연산자

- 8비트 변수를 예시로 들어보자
- 기존에는 10001011이라는 값을 가지고 있다



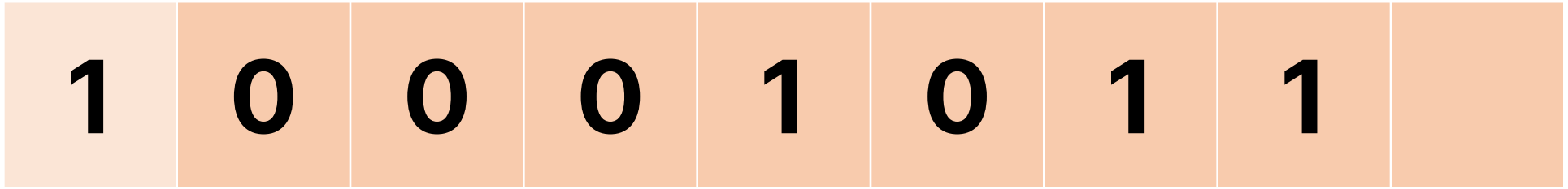
Shift 연산자

- 이 변수를 왼쪽으로 한 칸 움직이려고 한다
- tmp << 1;

1	0	0	0	1	0	1	1

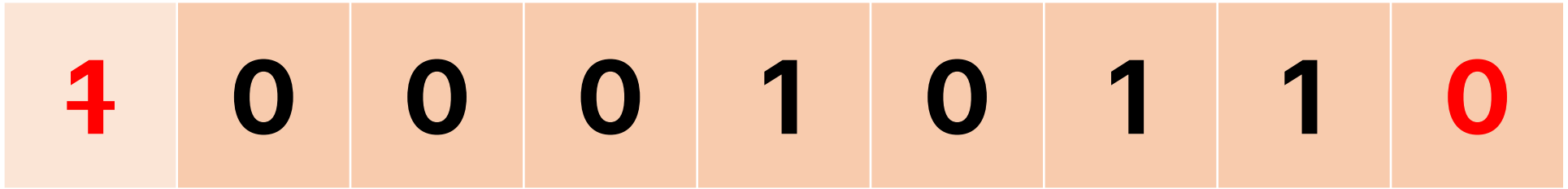
Shift 연산자

- 한 칸을 왼쪽으로 밀면 제일 왼쪽에 있던 1은 기존의 8칸에서 벗어나게 되고 오른쪽에는 빈 공간이 생기게 된다



Shift 연산자

- 밀려난 공간에 있는 1은 지워지며, 비어 있는 공간은 0으로 채운다



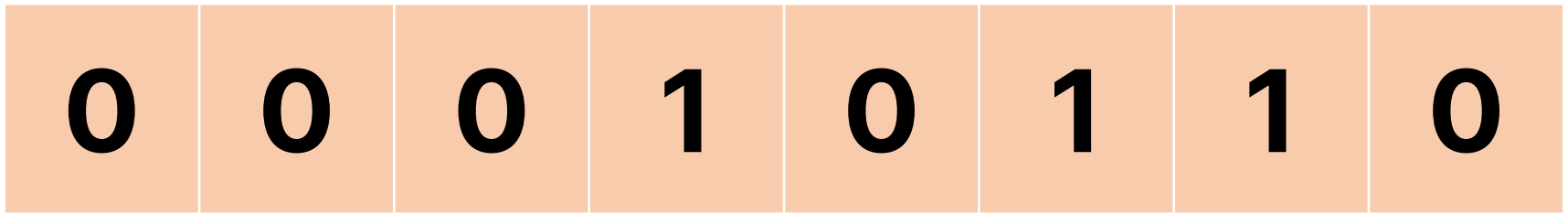
Shift 연산자

- 원하는 만큼 밀 때까지 반복한다

0	0	0	1	0	1	1	0

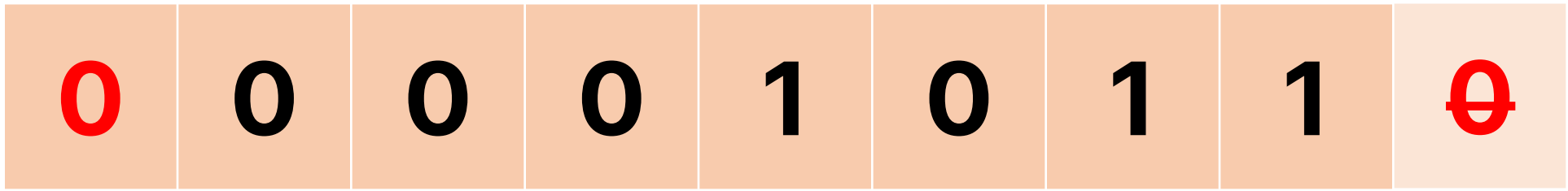
Shift 연산자

- 반대로 오른쪽으로 shift할 때도 자리를 넘어가는 비트는 지워지며 새로운 칸에는 0이 생겨난다



Shift 연산자

- 오른쪽으로 shift하면서 왼쪽 끝에 생겨난 빈 공간에는 0이 채워지며 오른쪽 끝에 나간 비트 0은 지워진다



Shift 연산자(Unspecified behavior)

- 지금부터 나오는 내용은 정확하게 정의 되지 않은 내용이다
- C++에서 정확하게 명시해두지 않은 내용은 어떻게 작동할지 보장할 수 없으며, 따라서 visual studio에서는 잘 됐는데~ 하는 식의 코딩은 지양한다
- 컴파일러마다 다르게 작동할 수 있으므로, 이러한 내용이 다른 컴파일러에서는 오류를 발생시킬 수 있음을 인지하고 피하도록 하자
- ex) int를 반환하는 함수에서 어떠한 값도 return 하지 않을 때, 함수 값을 사용하고자 한다면 어떠한 값이 나올지 보장하지 않는다

Shift 연산자(Unspecified behavior)

- 일반적인 int와 같은 숫자 변수의 제일 왼쪽 비트는 부호를 의미한다
- 0인 경우 양수를 의미하며 1인 경우 음수를 의미한다

- 반면 char 또는 unsigned 자료형의 경우, 항상 양수만 나타내므로 부호 비트 없이 모든 비트를 수를 나타내는 데 사용한다

Shift 연산자(Unspecified behavior)

- 따라서 int의 경우 -2,147,483,648 ~ 2,147,483,647를 나타내며
  unsigned int의 경우 0 ~ 4,294,967,295를 나타낸다
- 둘 다 범위의 경우 42억을 나타내는 것은 동일하지만 음수의 여부가 다르다

- 문제는 부호 비트에서 발생한다

Shift 연산자(Unspecified behavior)

- 음수를 오른쪽으로 shift하는 경우 부호 비트에 빈자리가 생긴다
- 일반적으로 배운 shift 연산의 경우 빈 자리는 0으로 채운다
- 이러한 경우 음수는 기존에 부호비트가 1인데 오른쪽으로 shift를 하면서 부호비트를 0으로 채우게 된다
- 양수가 음수로 바뀌게 된다는 문제점이 존재한다
- 따라서 이 부분에 따라 shift가 두가지로 나뉘게 된다

논리 Shift

- 기존에 배운 Shift가 논리 Shift다
- 비트에 어떠한 의미를 부여하지 않으며 정의에 기반하여 그저 비트를 움직이고 빈자리를 0으로 채우는 것을 논리 shift라고 한다

산술 Shift

- 산술 Shift는 부호비트를 고려하여 생겨났다
- 차후에 배울 내용이지만 우측으로 shift를 하는 것은 한 칸 움직일 때마다 값이 ½로 줄어드는 연산이다
- 이러한 수의 관점에서 볼 때 음수의 우측 shift를 통하여 부호 비트가 0으로 바뀌는 것은 잘못된 연산이다

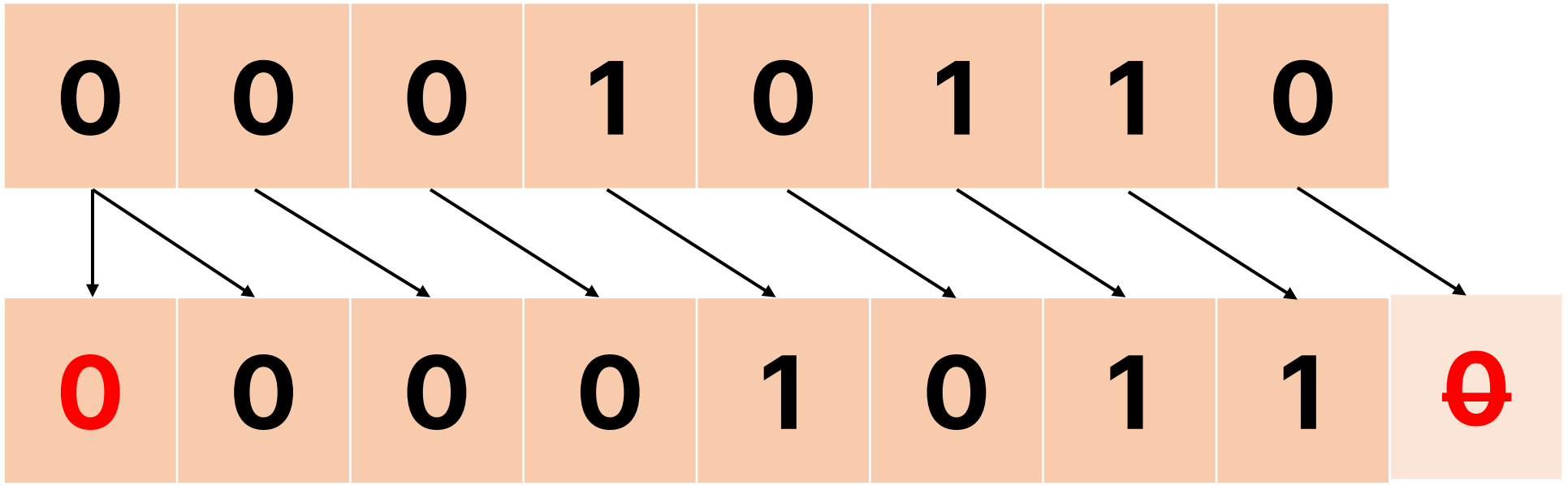
산술 Shift

- 따라서 양수인지, 음수인지에 따라 왼쪽 빈공간에 채우는 비트를 0 또는 1로 하는 것이 산술 shift이다
- 양수의 경우 기존과 동일하게 0을 채우며, 음수의 경우 1을 채워 부호비트를 꾸준히 1을 유지하도록 한다(기존의 부호비트를 이용해 빈자리를 채운다)

- 직접 예시를 봐보자

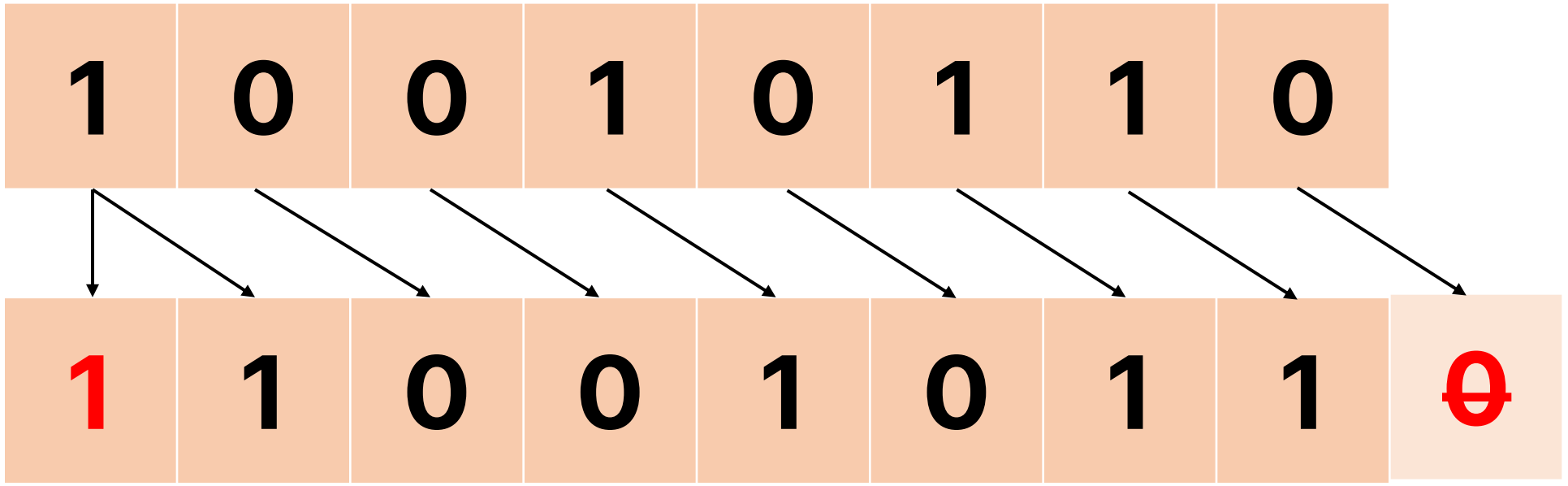
양수 산술 Shift

- 양수는 기존과 동일하게 부호 비트 0을 채운다



음수 산술 Shift

- 음수는 부호비트가 1이므로 1로 빈자리를 채운다



Shift 연산자(Unspecified behavior)

- 논리 Shift와 산술 Shift는 기존에 정의되어 있던 내용이다
- 하지만 문제는 부호가 있는 정수 자료형에서 논리 Shift인지 산술 Shift인지 정확하게 명시되어 있지 않다
- 이 부분은 어떤 컴파일러를 사용하느냐에 따라서 논리 Shift인지 산술 Shift인지 결정되게 된다
- MSVC와 g++를 사용하면 산술 Shift가 작동한다

Shift 연산자(Unspecified behavior)

- 따라서 일반적으로 int로 비트 연산을 사용하는데, shift를 사용할 때 부호비트를 사용하지 않은 쪽으로 사용하자
- 최대한 양수 위주로 사용을 하자

- 이 부분의 경우 컴파일러를 정확하게 알고 사용한다고 하여도, 어느 순간 바뀔 수 있기 때문에 조심하자

비트 연산

- 지금까지 나온 비트 연산의 경우, 비트 하나에 대해서 사용하는 내용이다. (Shift 제외)
- 우리가 사용하는 자료형의 경우, 수십 개의 비트이므로 어떻게 적용시켜야 할까?

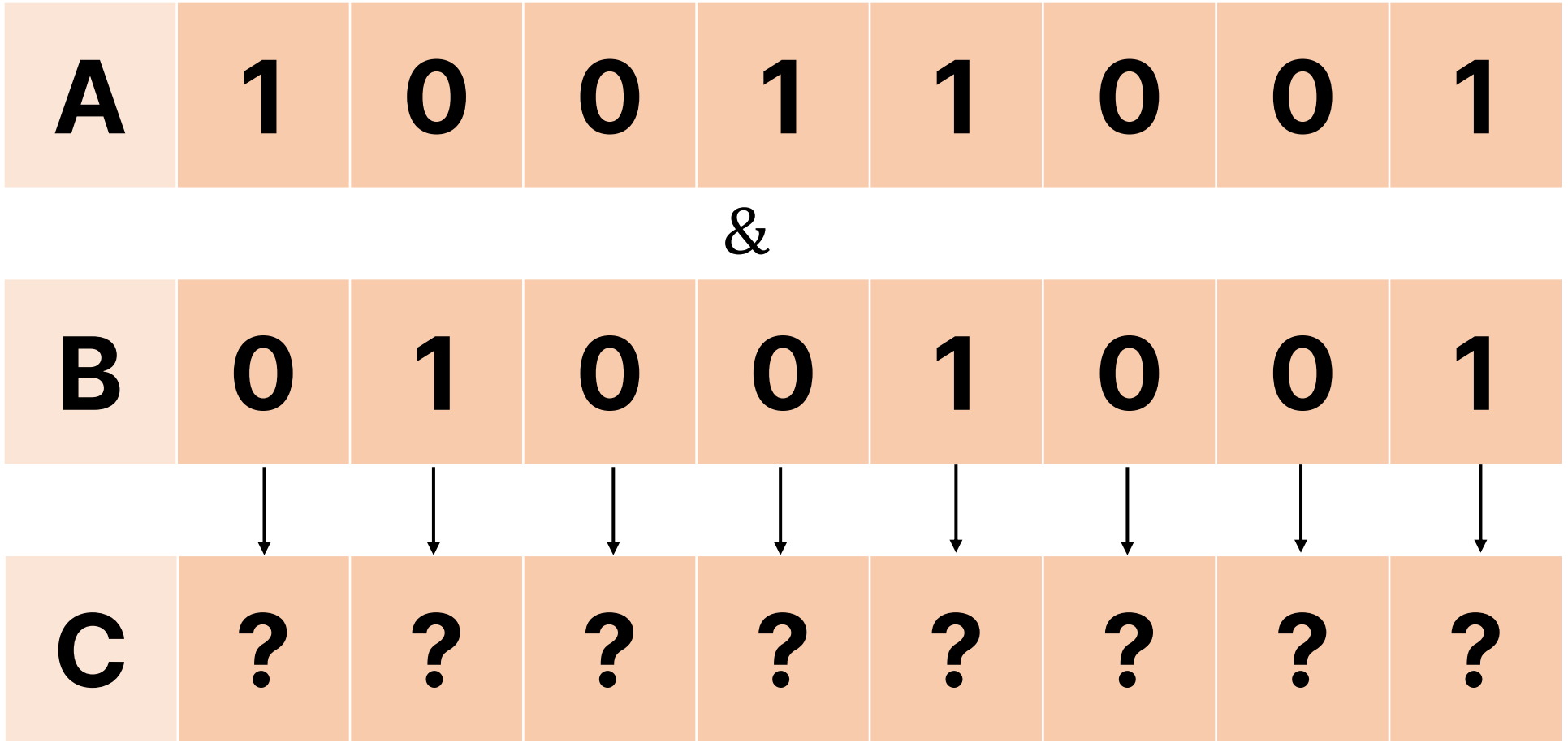
- **여러 개의 비트가 있을 때 각 해당되는 자리에 대해 비트 연산을 수행한다**

비트 연산

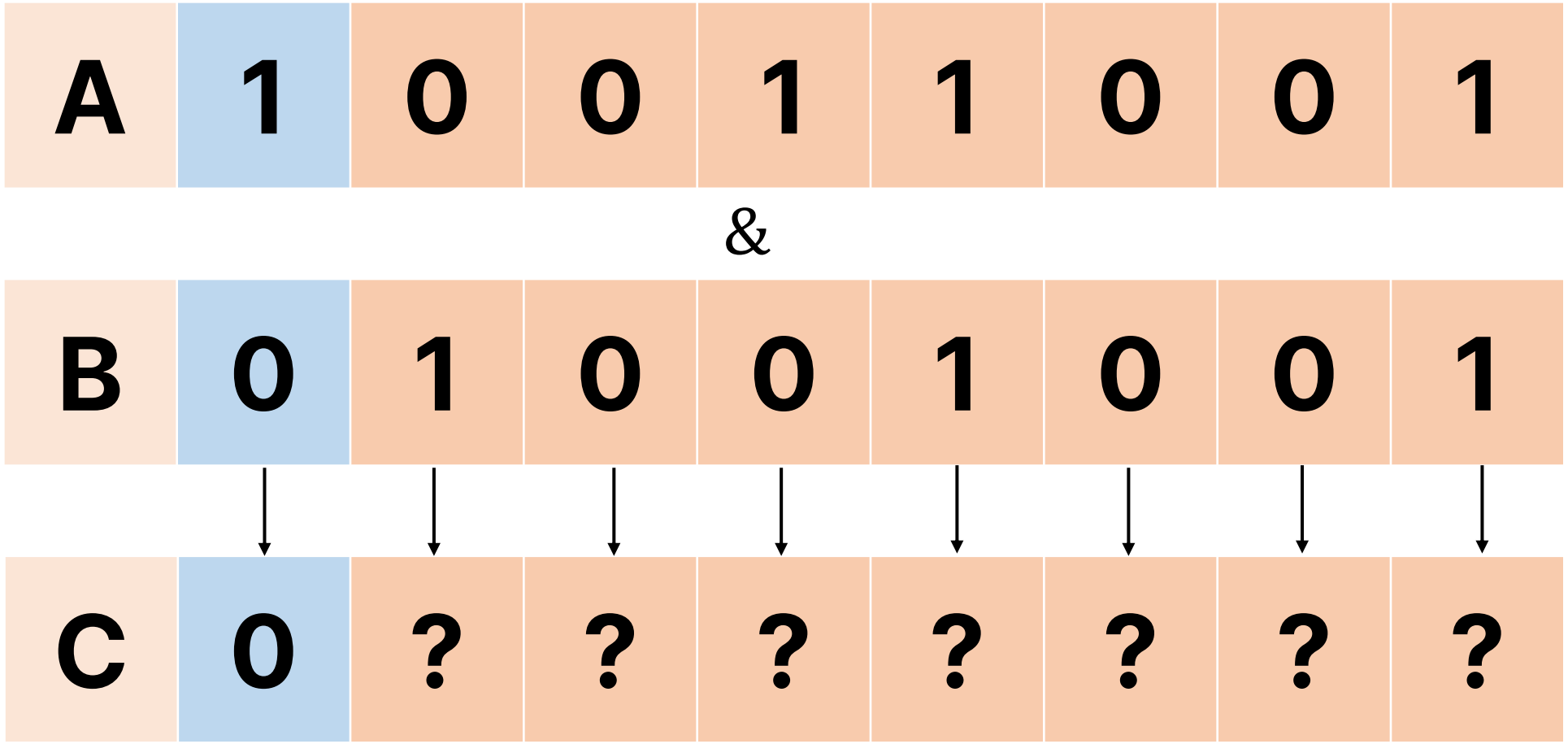
- A와 B는 32비트로 이루어진 자료형이라 하자(int)
- A에는 1번째 비트, 2번째 비트, ..., 32번째 비트까지 있을 것이며 B도 마찬가지이다
- 각 자리의 비트를 A_1, A_2, A_3, ... A_{32} 그리고 B_1, B_2, B_3, ... B_{32} 로 표현하자

- **비트 연산의 결과를 C라고 할 때, A_k <비트연산> B_k = $C_k(1 \leq k \leq 32)$이다**

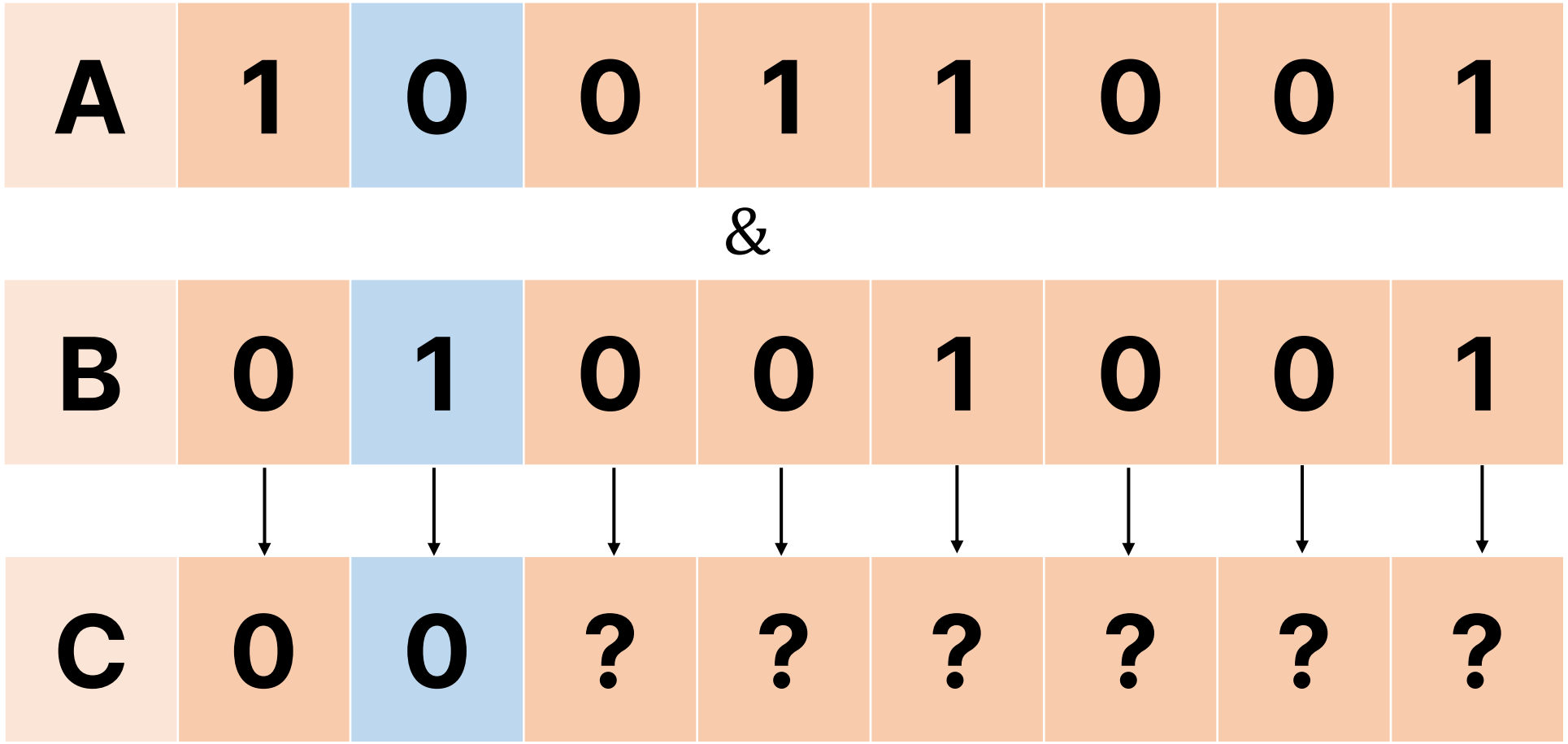
비트 연산



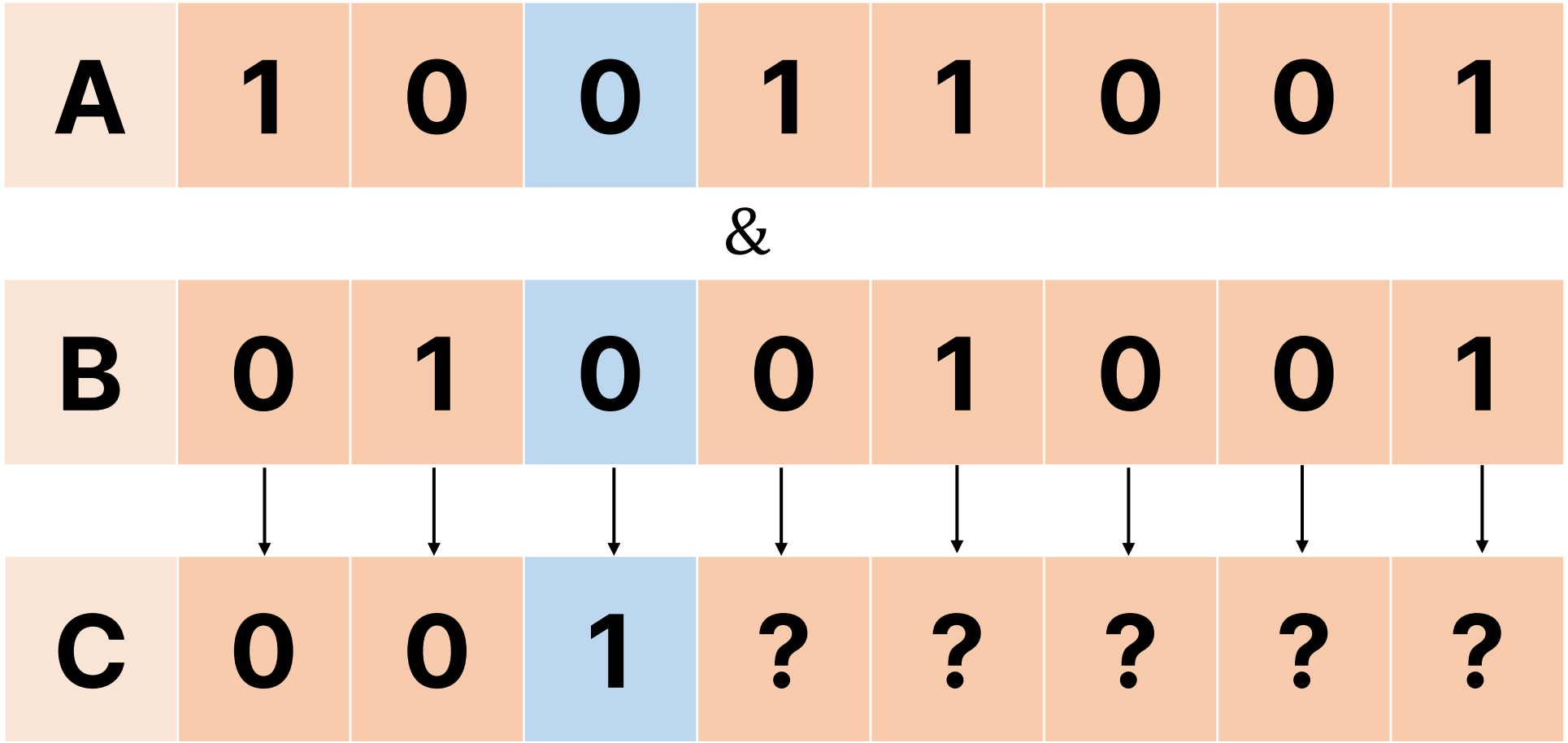
비트 연산



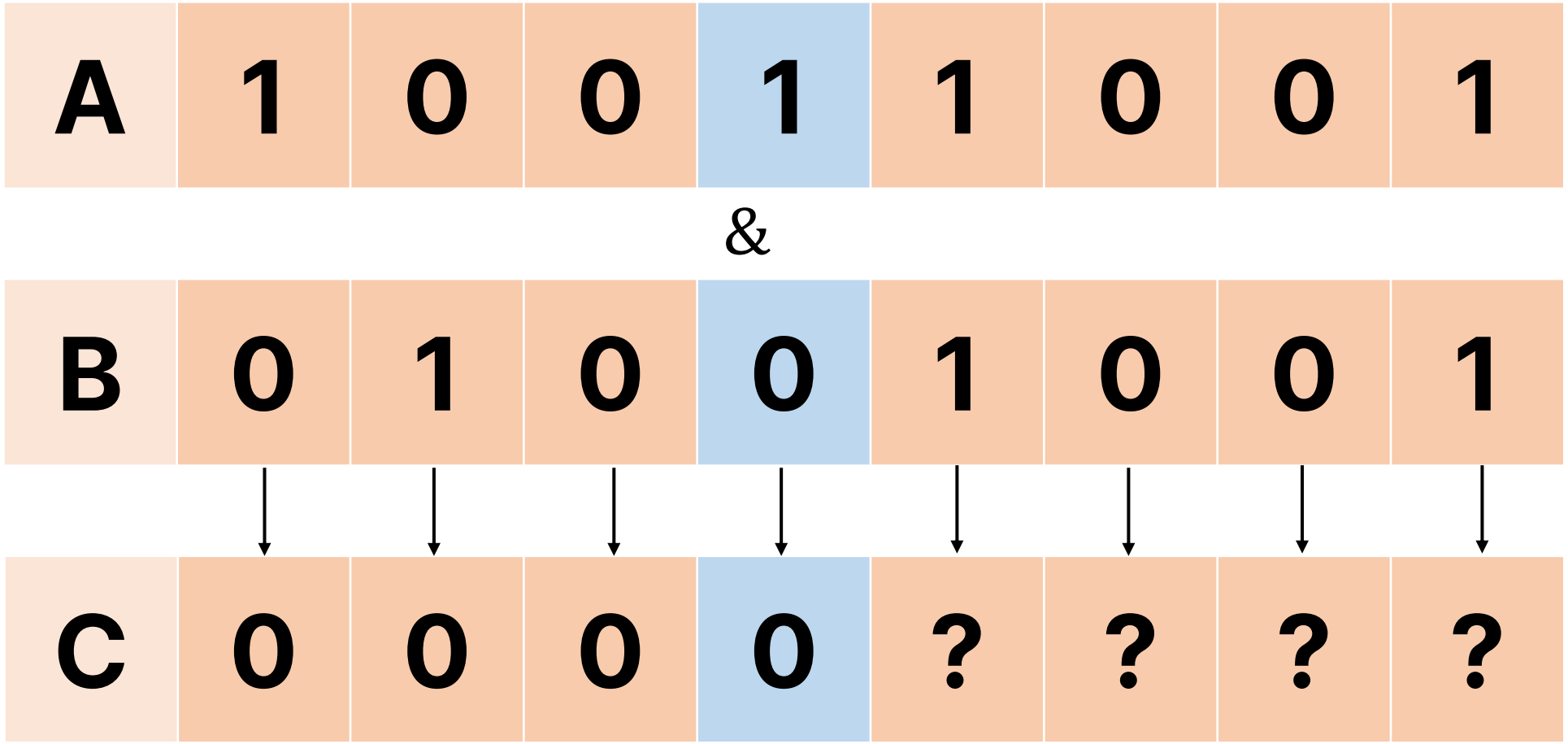
비트 연산



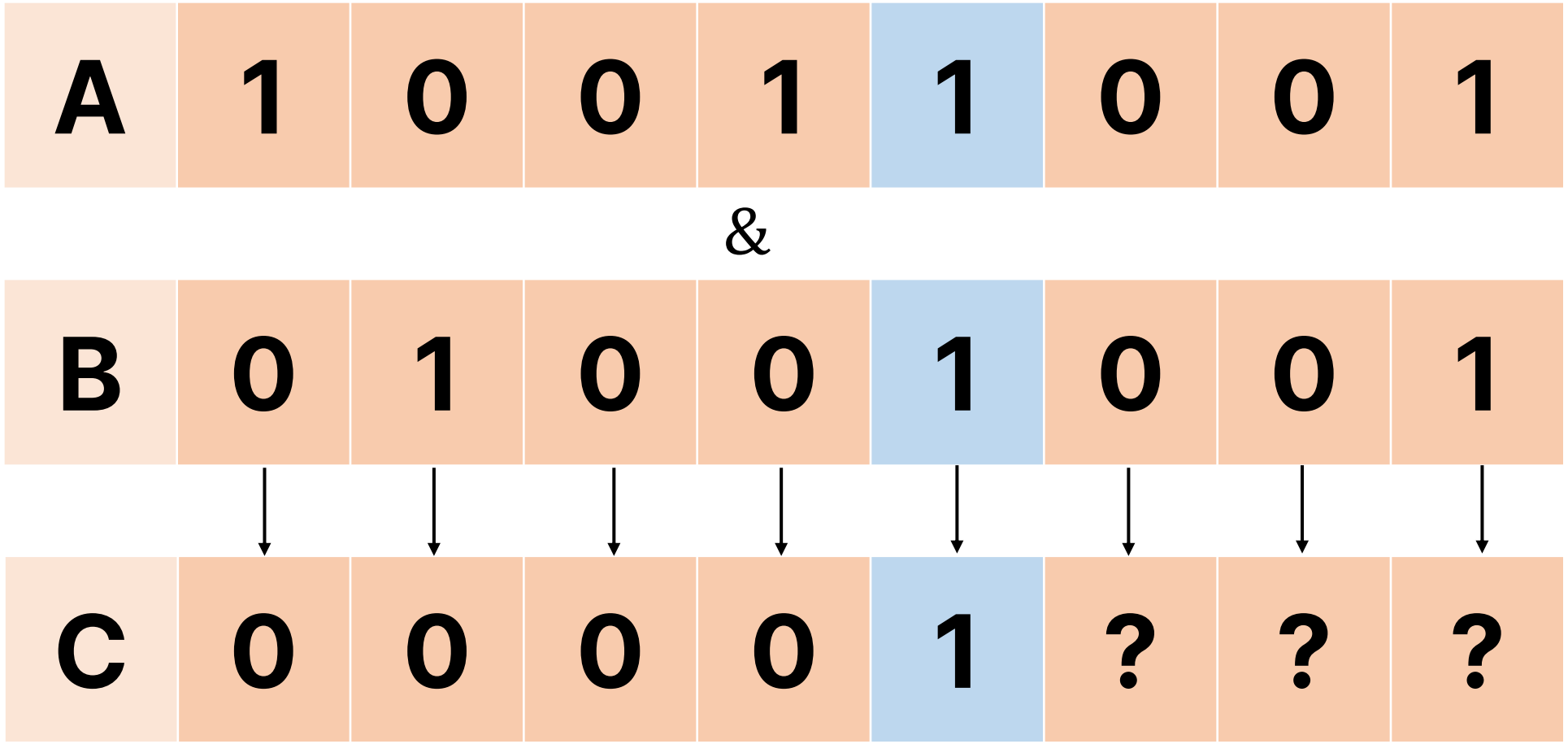
비트 연산



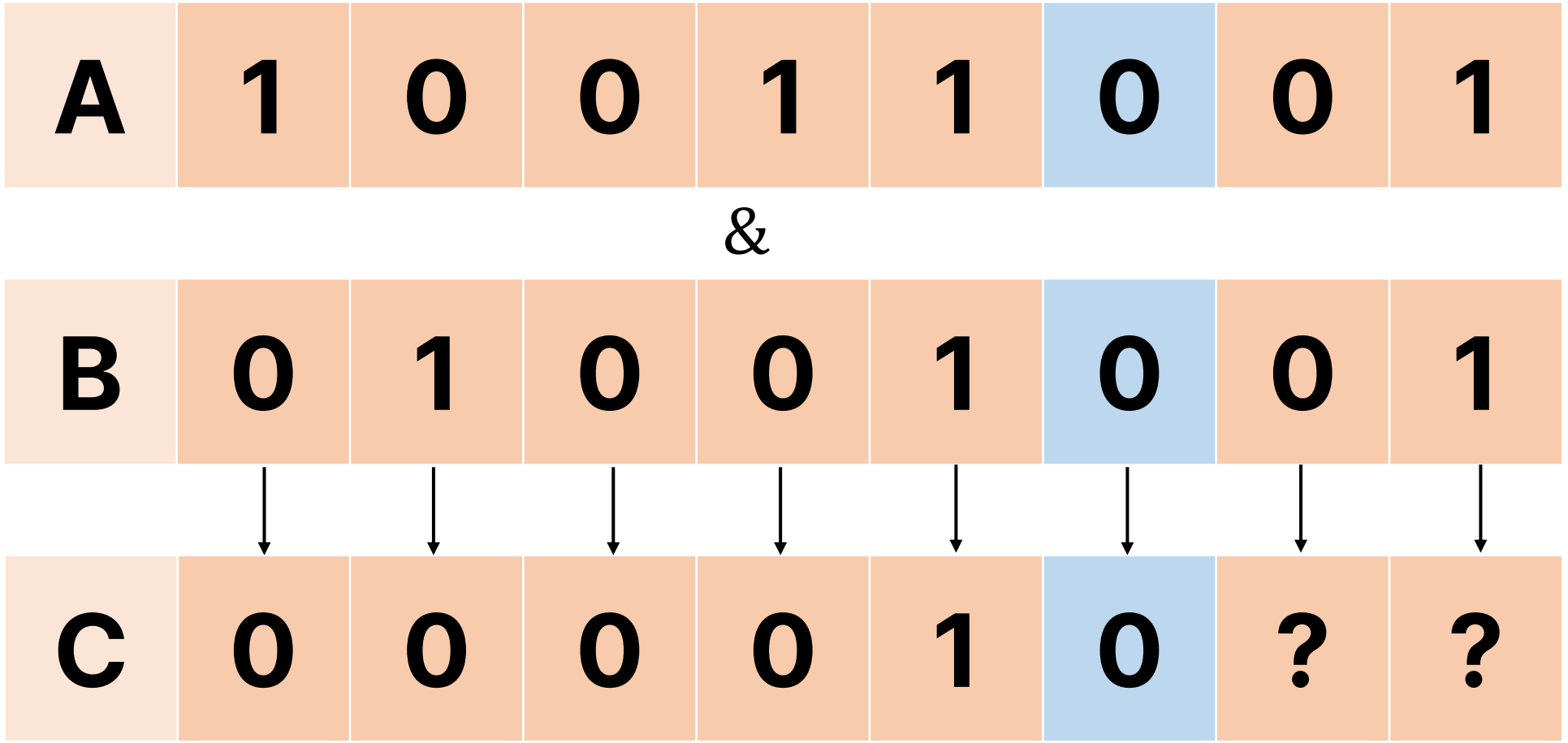
비트 연산



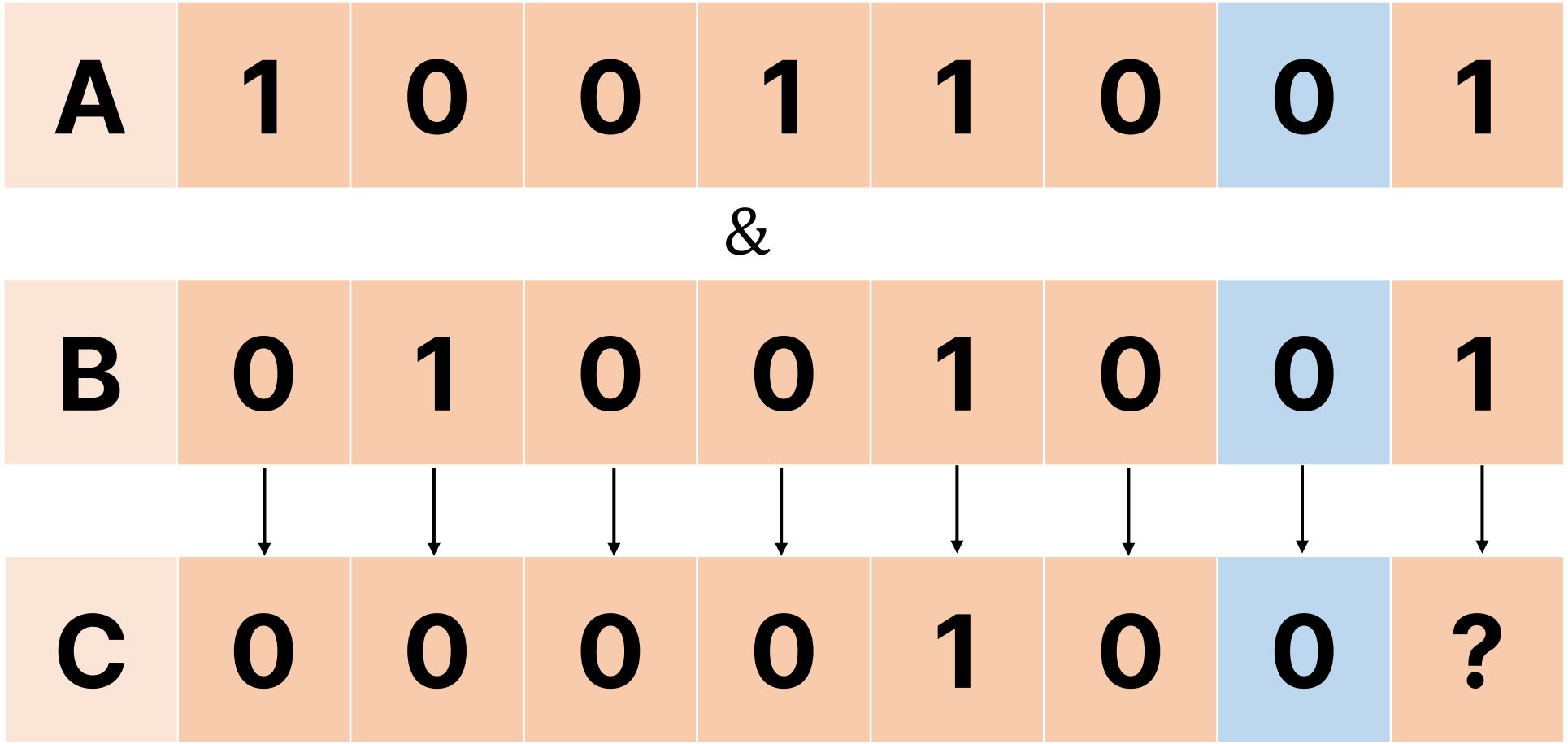
비트 연산



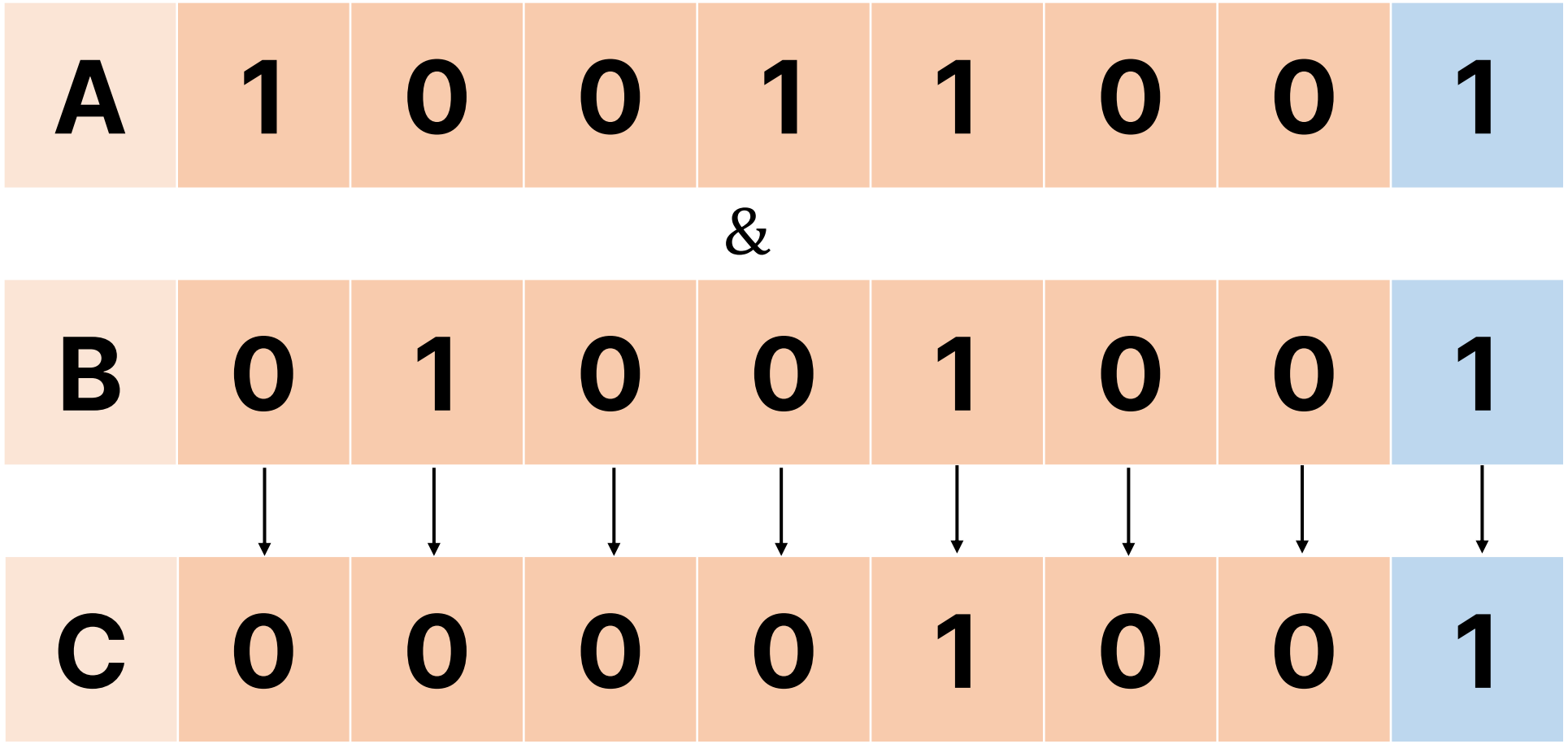
비트 연산



비트 연산



비트 연산



비트 연산

- 예시에서는 하나씩 순차적으로 진행됐지만, 실제로는 모든 비트가 동시에 진행된다

AND 연산의 의미

- AND는 특정한 내용을 남기도록 주로 사용한다
- AND의 결과를 살펴보자
- B가 0인 경우 반드시 0이 나오며,

  B가 1인 경우 A가 나오게 된다
- 이것을 이용하여 A중에서 원하는 부분만 꺼낼 수 있다

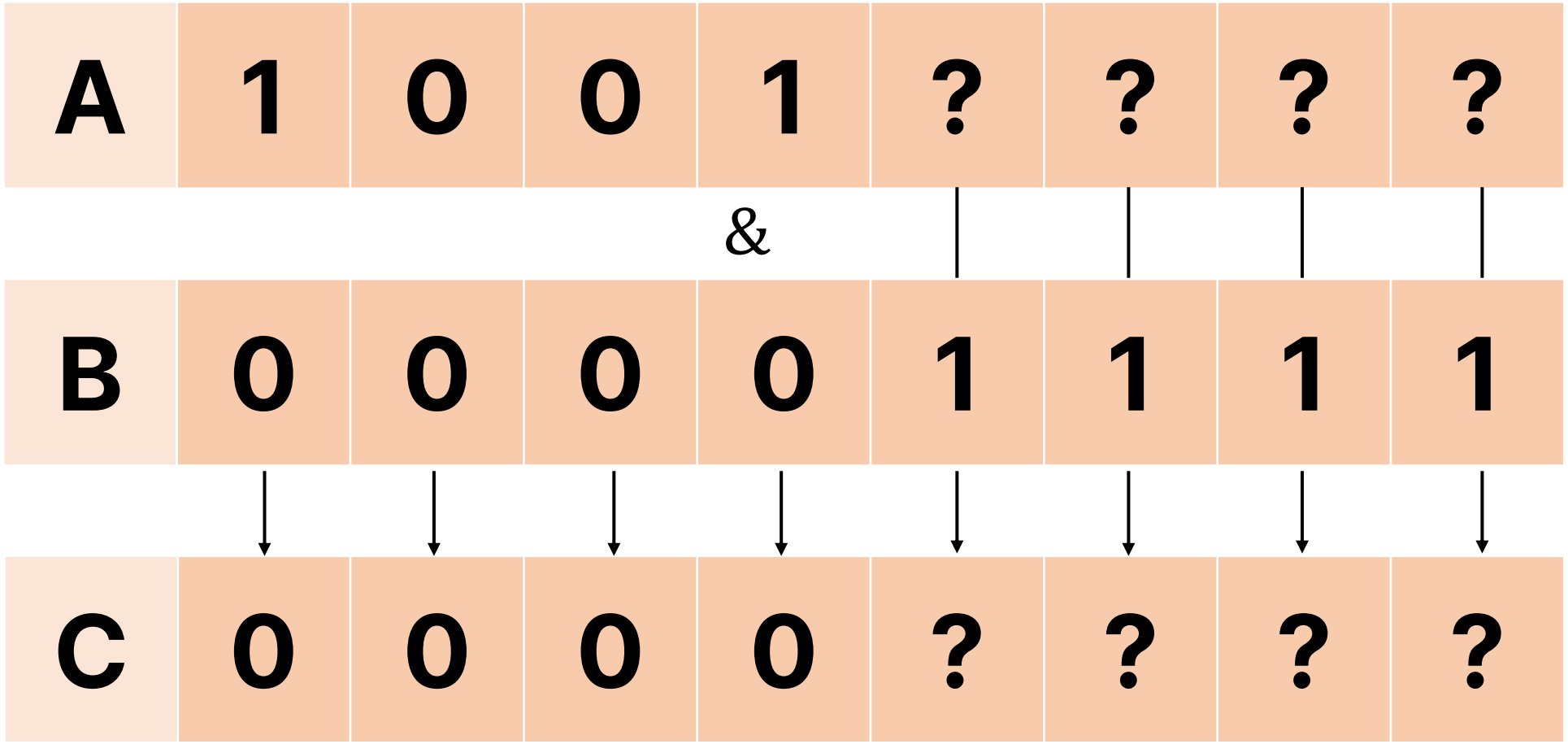
A	∧	B	=	Result
0		0		0
0		1		0
1		0		0
1		1		1

AND 연산의 의미

- 8비트짜리 A가 있다고 가정하자
- A중에서 뒤의 4비트를 알고 싶다고 하자

- 어떻게 하면 될까?
- 0000 1111을 AND연산 하면 된다

AND 연산의 의미



AND 연산의 의미

- 이러한 방식으로 보존하고 싶은 부분을 1, 지우고 싶은 부분을 0으로 정한 B를 AND연산해주게 되면 1로 정한 부분은 A를 그대로 가져오고 그 외의 부분은 0으로 정할 수 있다

- 네트워크에서 주로 사용한다

AND 연산의 의미

- 네트워크에서 말하는 ip주소 127.0.0.1
- 이들은 8비트 숫자 4개이다
- (8비트). (8비트). (8비트). (8비트)

- ip주소 중 제일 뒤에 있는 숫자를 가져오고 싶다면

  (0000 0000). (0000 0000). (0000 0000). (1111 1111)을 AND 연산하면 된다

OR 연산의 의미

- OR는 두 정보를 합치기 위해 주로 사용한다
- OR의 결과를 살펴보자
- A와 B가 둘 다 0인 경우에만 0이 나오는 것을 알 수 있다
- 이를 이용하여 둘 다 갖고 있지 않은 경우 0

  둘 중 하나라도 가지고 있는 경우 1로 만들 수 있다

A	∨	B	=	Result
0		0		0
0		1		1
1		0		1
1		1		1

OR 연산의 의미

- 두 정보를 합치는 것은 이후 비트마스킹 알고리즘에서 좀 더 자세히 살펴보자
- 대표적인 사용 예시는 어떠한 숫자를 만들 때 사용한다
- ex) 1, 2, 4, 5번째 자리의 비트가 1인 수를 만들고 싶다

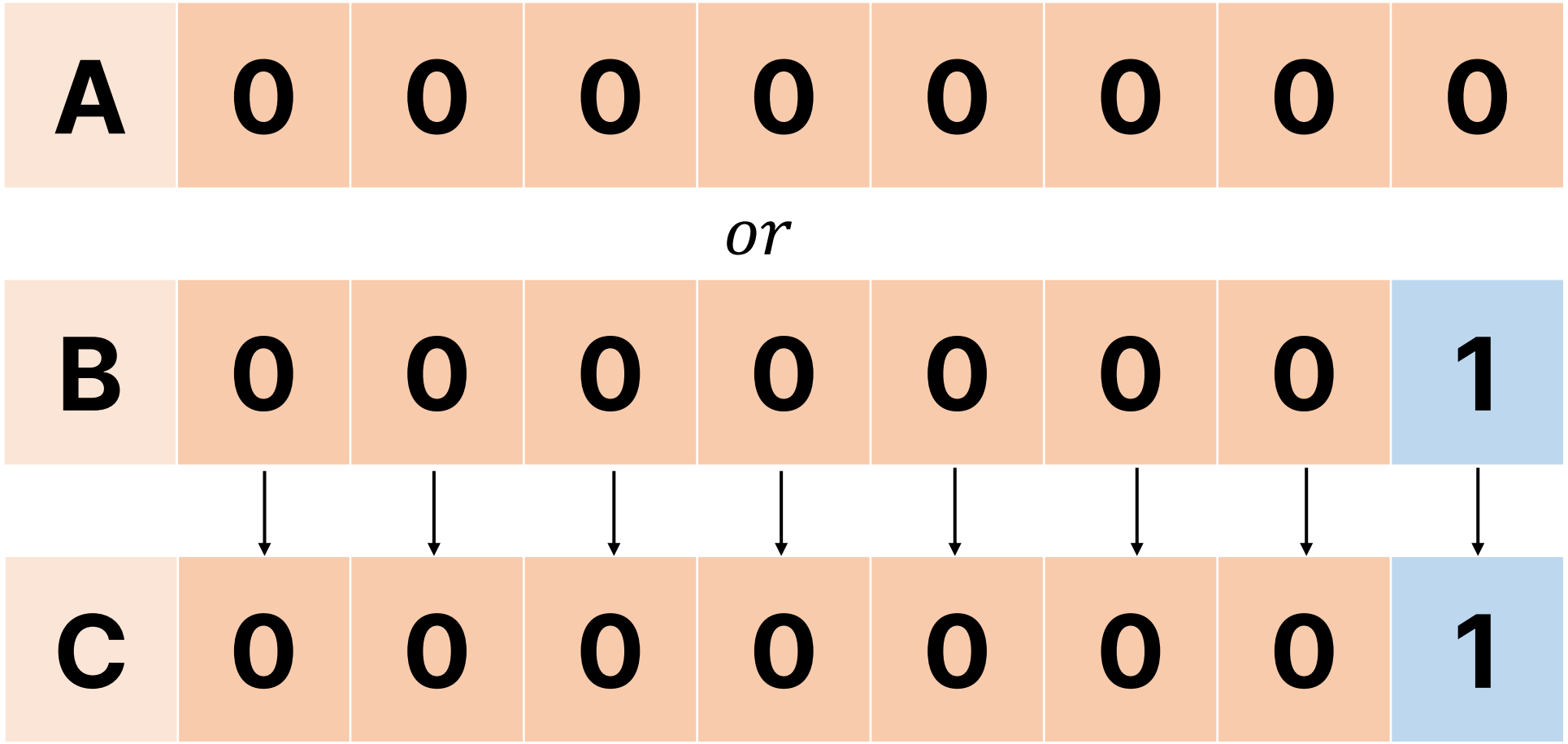
OR 연산의 의미

- 두 정보를 합치는 것은 이후 비트마스킹 알고리즘에서 좀 더 자세히 살펴보자
- 대표적인 사용 예시는 어떠한 숫자를 만들 때 사용한다
- ex) 1, 2, 4, 5번째 자리의 비트가 1인 수를 만들고 싶다

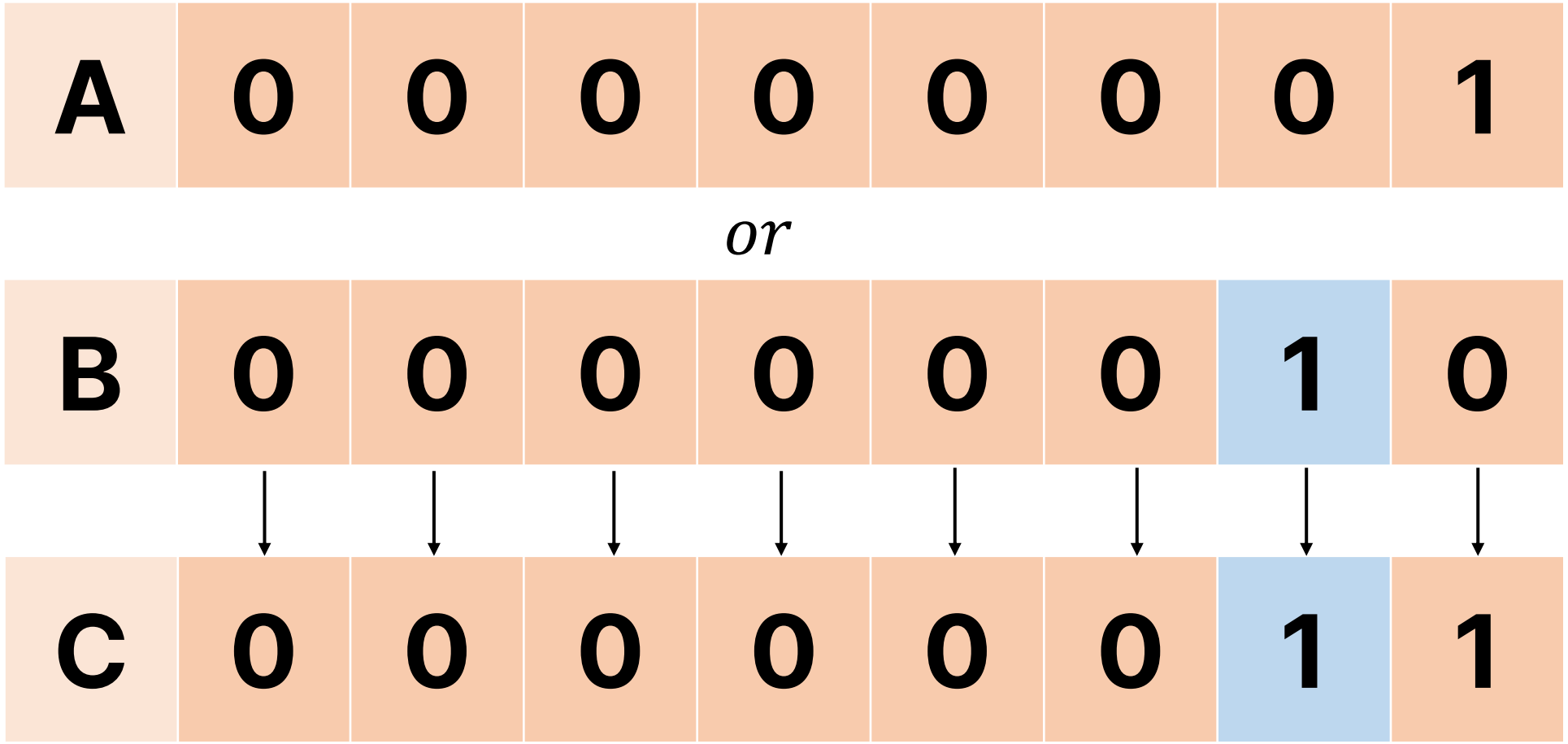
OR 연산의 의미

- 1을 왼쪽으로 Shift하면 k번째 자리의 비트가 1인 수를 만들 수 있다
- 이것을 이용해 1, 2, 4, 5번째 자리의 비트가 1인 수, 총 4개를 만든 후 이것을 or 연산을 이용해 하나로 합치자

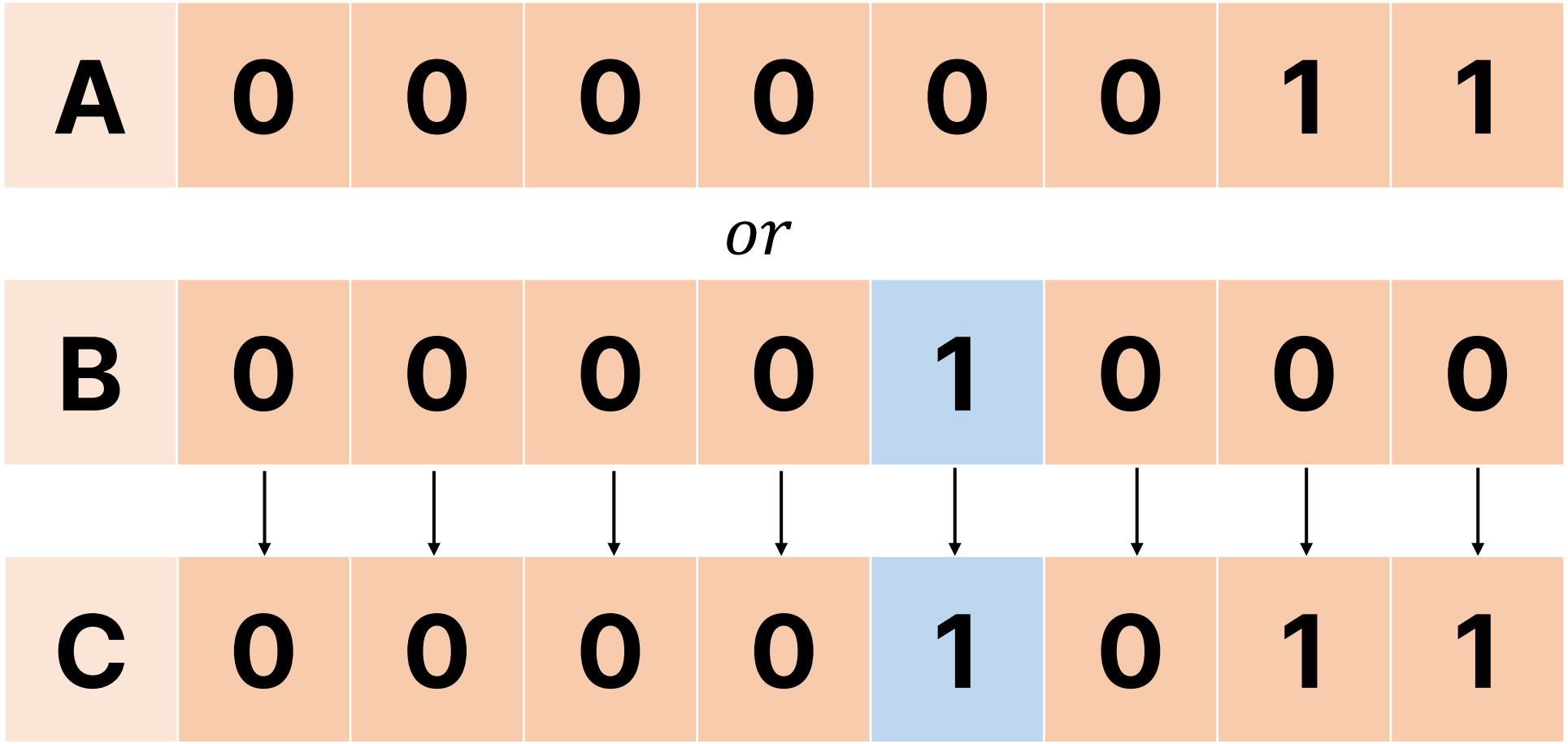
OR 연산의 의미



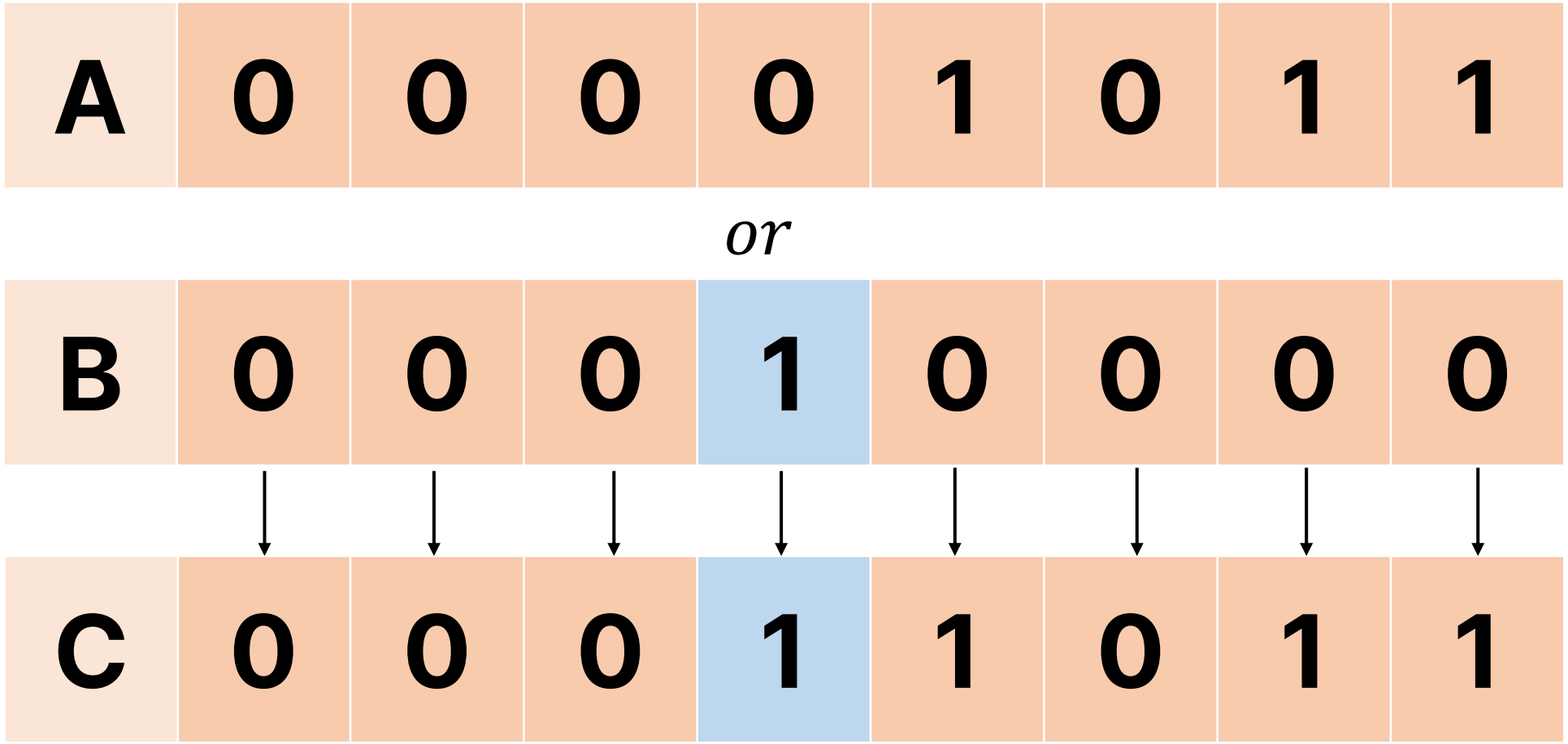
OR 연산의 의미



OR 연산의 의미



OR 연산의 의미



XOR 연산의 의미

- XOR는 특정 비트를 뒤집기 위해 사용된다
- XOR의 결과를 살펴보자
- B가 0인 경우 A가 그대로 나오지만

  B가 1인 경우 !A가 나오는 것을 볼 수 있다
- 이를 이용하여 특정 비트를 뒤집을 수 있다

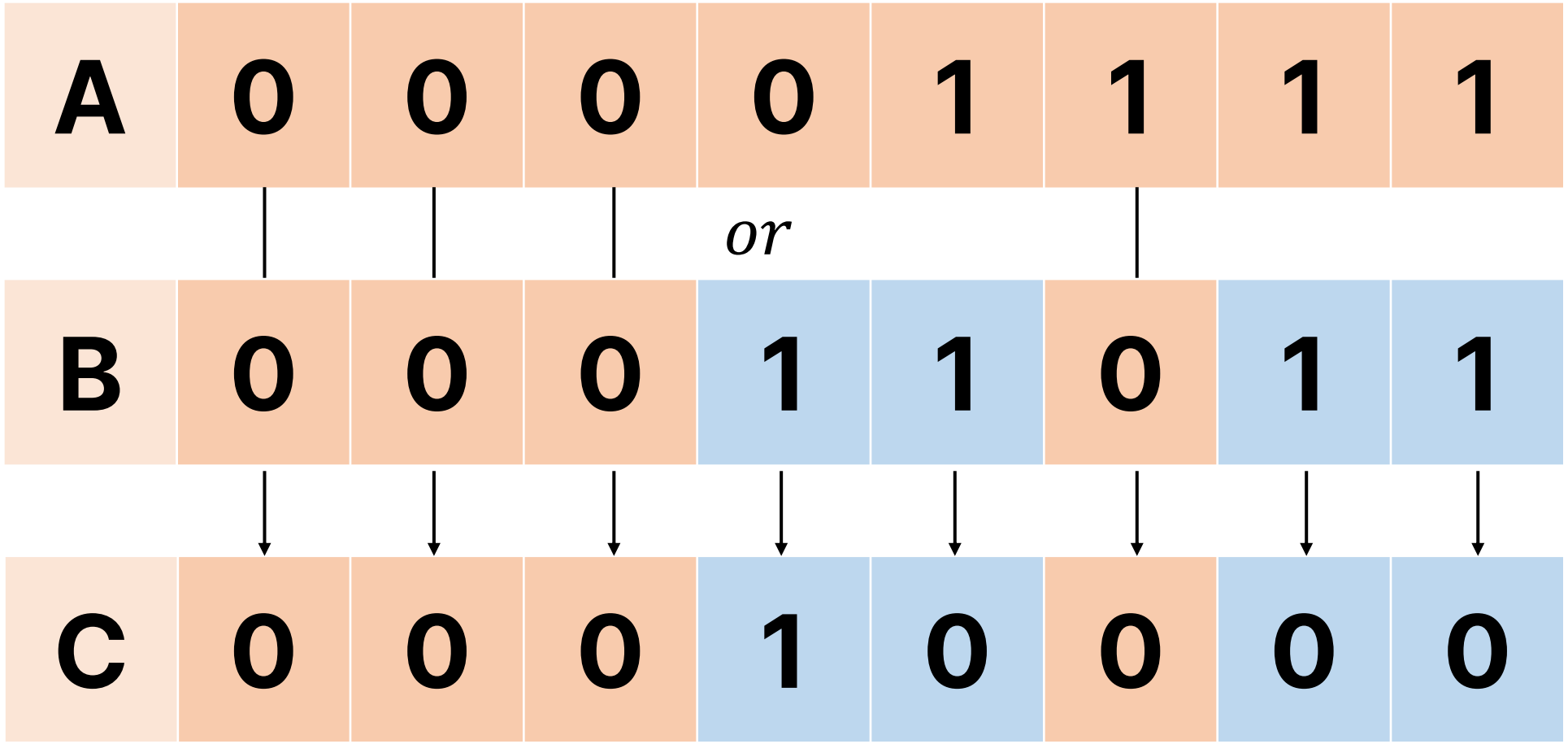
A ⊕	B =	Result
0	0	0
0	1	1
1	0	1
1	1	0

XOR 연산의 의미

- 정보를 뒤집어보자
- ex) 1, 2, 4, 5번째 자리의 비트가 1이면 0으로 0이면 1로 바꾸고 싶다
- 뒤집고 싶은 자리의 비트는 1, 다른 자리는 0인 B를 만들어 XOR 하면 뒤집을 수 있다

  00011011을 XOR 하면 된다

XOR 연산의 의미



비트연산으로 무엇을 할 수 있을까

- 기존에는 산술연산을 통해 얻던 것을 빠르게 비트를 이용해 구할 수 있다
- 대소문자 변환
- 2의 배수의 나눗셈, 나머지 연산
- ...
- 이들은 아주 매우 조금 빨라지므로 매우 작은 프로그램에서는 체감하지 못할 수 있다
- 비트를 이용해 이런 것들도 가능하다고 알고 있자

대소문자도 쉽게 바꿀 수 있다

- char는 문자를 나타내는 자료형
- 아스키 코드로 정의된 값을 나타냄

문자	char 변수 값(아스키 코드)
A	65
Z	90
a	97
z	122

대소문자도 쉽게 바꿀 수 있다

- 대문자의 범위: 65 ~ 90
- 소문자의 범위: 97 ~ 122

문자	char 변수 값(아스키 코드)
A	65
Z	90
a	97
z	122

기존 방식

- 대문자와 소문자는 변수 값이 32만큼 차이난다

Upper(대문자로 변환)

- 소문자에서 32만큼 뺀다

Lower(소문자로 변환)

- 대문자에서 32만큼 더한다

기존 방식

- assert는 없어도 무관

```
char upper(char ch) {
    assert('a' <= ch and ch <= 'z');
    return ch - 32;
}

char lower(char ch) {
    assert('A' <= ch and ch <= 'Z');
    return ch + 32;
}
```

기존 방식 개선

- 하나로 합쳐보자
- 개선보다는 하나로 합쳤다는 의미

- 대문자인 경우, 소문자로 변환 후 return
- 소문자인 경우, 대문자로 변환 후 return
- 대소문자를 바꾸려는 목적이므로, 영문을 제외한 다른 문자는 들어오지 않는다 가정

기존 방식 개선

- if문으로 범위를 검사했으므로 assert는 필요 없다

```
char upper(char ch) { return ch - 32; }

char lower(char ch) { return ch + 32; }

char change(char ch) {
    if ('a' <= ch and ch <= 'z') // 소문자인 경우 대문자로 변환
        return upper(ch);
    else                         // 대문자인 경우 소문자로 변환
        return lower(ch);
}
```

기존 방식 개선

- 다음처럼 간단한 함수를 합칠 수도 있다.
- 하지만 코드가 거대해지는 경우, 매우 많은 else if문은 가독성을 떨어트리므로 함수를 사용하는 것을 권장(클린코드)(PS식 코딩과는 대조됨)

```
char change(char ch) {
    if ('a' <= ch and ch <= 'z') // 소문자인 경우 대문자로 변환
        return ch - 32;
    else                         // 대문자인 경우 소문자로 변환
        return ch + 32;
}
```

비트를 생각해보자

- char는 1바이트 자료형, 즉 8개의 비트로 이루어진 자료형이다
- 각 문자들의 특징을 보자

대문자 비트

문자(값)	비트	문자	비트	문자	비트
A(65)	0100 0001	J(74)	0100 1010	S(83)	0101 0011
B(66)	0100 0010	K(75)	0100 1011	T(84)	0101 0100
C(67)	0100 0011	L(76)	0100 1100	U(85)	0101 0101
D(68)	0100 0100	M(77)	0100 1101	V(86)	0101 0110
E(69)	0100 0101	N(78)	0100 1110	W(87)	0101 0111
F(70)	0100 0110	O(79)	0100 1111	X(88)	0101 1000
G(71)	0100 0111	P(80)	0101 0000	Y(89)	0101 1001
H(72)	0100 1000	Q(81)	0101 0001	Z(90)	0101 1010
I(73)	0100 1001	R(82)	0101 0010		

소문자 비트

문자(값)	비트	문자	비트	문자	비트
a(97)	0110 0001	j(106)	0110 1010	s(115)	0111 0011
b(98)	0110 0010	k(107)	0110 1011	t(116)	0111 0100
c(99)	0110 0011	l(108)	0110 1100	u(117)	0111 0101
d(100)	0110 0100	m(109)	0110 1101	v(118)	0111 0110
e(101)	0110 0101	n(110)	0110 1110	w(119)	0111 0111
f(102)	0110 0110	o(111)	0110 1111	x(120)	0111 1000
g(103)	0110 0111	p(112)	0111 0000	y(121)	0111 1001
h(104)	0110 1000	q(113)	0111 0001	z(122)	0111 1010
i(105)	0110 1001	r(114)	0111 0010		

비트를 비교해보자

- 대문자와 소문자는 같은 문자에 대해서 뒤에서 6번째 비트만 제외하고 모든 비트가 동일하다.
- 대문자는 뒤에서 6번째 비트가 0이며, 소문자는 뒤에서 6번째 비트가 1이다

- 왜?

비트를 비교해보자

- 변환 과정을 생각해보자
- 대문자랑 소문자로 변환할 때 32라는 차이가 존재했다
- 32를 비트로 나타내면 0010 0000이다
- 뒤에서 6번째 비트가 의미하는 것은 2^5, 즉 32를 의미한다

비트를 비교해보자

- 기존에 대문자가 32를 나타내던 자리는 0이었음
- 소문자로 변환하기 위해서는 32를 더해야 하므로 뒤에서 6번째 자리에 1을 더해야 한다
- 하지만 기존에 0이었으므로, 1을 더해도 다른 자리가 바뀌지 않는다
  (100에서 10의 자리에 9를 더한다 해도 190이 되면서 10의 자리 숫자만 바뀔 뿐 다른 자리까지 영향을 미치지 않는 것과 동일)

비트를 비교해보자

- 소문자를 대문자로 바꿀 때도 마찬가지
- 대문자로 변환하기 위해서는 32를 빼야 하므로 뒤에서 6번째 자리에 1을 빼줘야 한다.
- 기존에 1이었으므로, 1을 빼도 다른 자리가 바뀌지 않는다

종합해보면

- 대문자를 소문자로 바꾸기 위해서는, 또는 소문자를 대문자로 바꾸기 위해서는 뒤에서 6번째 자리의 비트를 바꾸면 된다
- 기존에 0인 경우 1로, 1인 경우 0으로 바꾸면 된다.

- 기존과 다른 숫자로 바꾸는 비트 연산이 존재한다: 비트 1과 XOR

XOR

- 배타적 논리합
- 두 비트가 같은 경우 1, 다른 경우 0을 출력한다.
- 일반적으로 코드에서는 ^를 사용한다

A ⊕	B =	Result
0	0	0
0	1	1
1	0	1
1	1	0

종합해보면

- 뒤에서 6번째 자리 비트에 1을 XOR해주면 대소문자를 바꿀 수 있다
- 뒤에서 6번째 자리가 1이며 다른 자리가 0인 수는 32이므로 32를 XOR 해주면 된다

```
char change(char ch) { return ch ^ 32; }
```

정수에서 비트를 이용한 트릭

- 정수에서 나눗셈은 몫을 취하고 나머지를 버리는 나눗셈이다
- 우리가 나눗셈을 할 때 비트를 그대로 가져갈 수 있는 방법을 생각해보자

정수에서 비트를 이용한 트릭

- 우리가 5678이라는 수에서 제일 쉽게 몫을 알 수 있는 나눗셈은 무엇일까
- 10진수 이므로 10의 배수로 나누면 자리수를 옮기는 것과 동일하다
- 10으로 나누면 5678 중에 567이 몫이며 100으로 나누면 56이 몫이다

- 비트도 동일하다

정수에서 비트를 이용한 트릭

- 컴퓨터는 2진수 체계이므로 2의 배수로 나누는 것이 직관적으로 몫이 보인다
- 2를 나눈다 생각해보자

0	0	0	1	1	0	0	1

정수에서 비트를 이용한 트릭

- 00011001은 10진수로 25이다
- 2로 나눈다면 몫은 12, 나머지는 1이다

0	0	0	1	1	0	0	1

정수에서 비트를 이용한 트릭

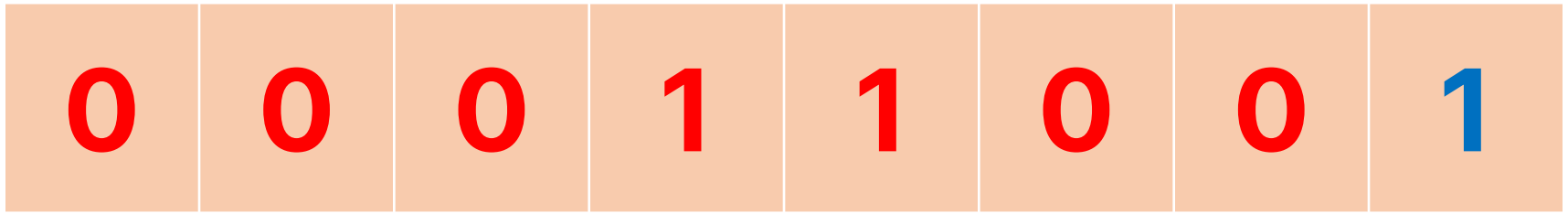
- 10진수에서 10으로 나누는 것처럼 자리로 생각해보자
- 2로 나눈다면 오른쪽에서 2번째 자리부터 왼쪽으로 남은 것이 몫이다

0	0	0	1	1	0	0	1

정수에서 비트를 이용한 트릭

- 2로 나눈다면 다음 그림에서 빨간색이 몫이며 파란색이 나머지이다
- 그렇다면 빨간색 부분만 남기고 파란색은 버리는 연산이 무엇이 있을까?

  >> 우측 Shift를 이용하면 된다



정수에서 비트를 이용한 트릭

- 따라서 2^k로 나누려 할 때 k번 만큼 shift하면 된다
- 값만 얻으려는 경우 tmp >> k, 변수에 대입하려는 경우 tmp >>= k로 사용하면 된다

- 일반적으로 제일 많이 사용하는 것이 나누기 2다
- 이를 이용해서 어떤 변수를 2로 나누려 할 때 tmp / 2 를 tmp >> 1로,
  tmp /= 2를 tmp >>= 1로 사용할 수 있다

정수에서 비트를 이용한 트릭

- 나누기 2와 유사한 연산이 또 무엇이 있을까
- 2로 나눈 나머지를 사용하는 경우가 많다
- 제일 대표적인 예시가 홀수 짝수 판별
- %2를 하여 1이 남으면 홀수 0이 남으면 짝수로 판별할 수 있다

- 이것을 비트연산으로 살펴보자

정수에서 비트를 이용한 트릭

- 홀수와 짝수를 어떻게 정의할 수 있을까
- 어떠한 자연수 k에 대하여 짝수를 곱해준다면 그 수는 반드시 짝수이다
- 그렇다면 짝수 중 제일 작은 수는 2이다

- 이런저런 말들이 많지만 2의 배수는 전부 짝수이다
- 짝수에 +1을 해주면 홀수이다

정수에서 비트를 이용한 트릭

- 2의 배수를 비트에서 나타내면 어떤 의미일까?
- 나눗셈과 마찬가지로 생각해보자
- 10진수에서 왼쪽으로 한 칸 미는 것은 10을 곱한 것이다
- 예를 들어 5678이 있을 때 56780으로 왼쪽으로 한 칸 미는 것은 10을 곱한 것이다

정수에서 비트를 이용한 트릭

- 2진수에서도 왼쪽으로 한 칸 밀면 2를 곱한 것과 마찬가지임을 알 수 있다
- 아래에 2진수에서 빨간색 부분은 어떤 수에 2를 곱한 것과 마찬가지임을 알 수 있다

0	0	0	1	1	0	0	1

정수에서 비트를 이용한 트릭

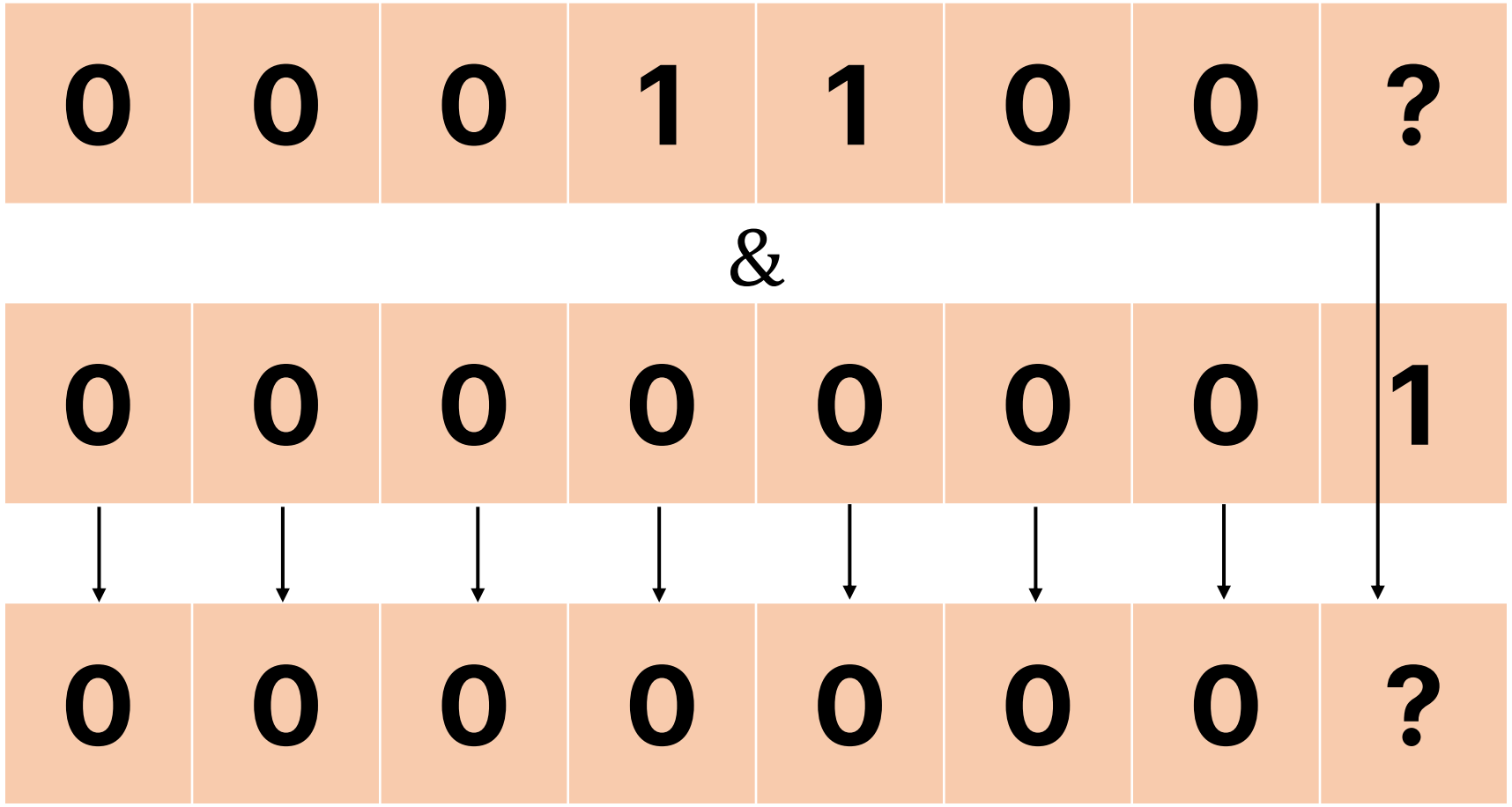
- 그렇다면 빨간색 부분은 2의 배수, 즉 짝수이므로 제일 오른쪽 끝 비트 하나만 확인하면 홀수와 짝수를 판별할 수 있다
- 따라서 오른쪽 제일 끝 비트 하나만 남기고 모두 지워보자(AND 연산을 사용해보자)

0	0	0	1	1	0	0	1

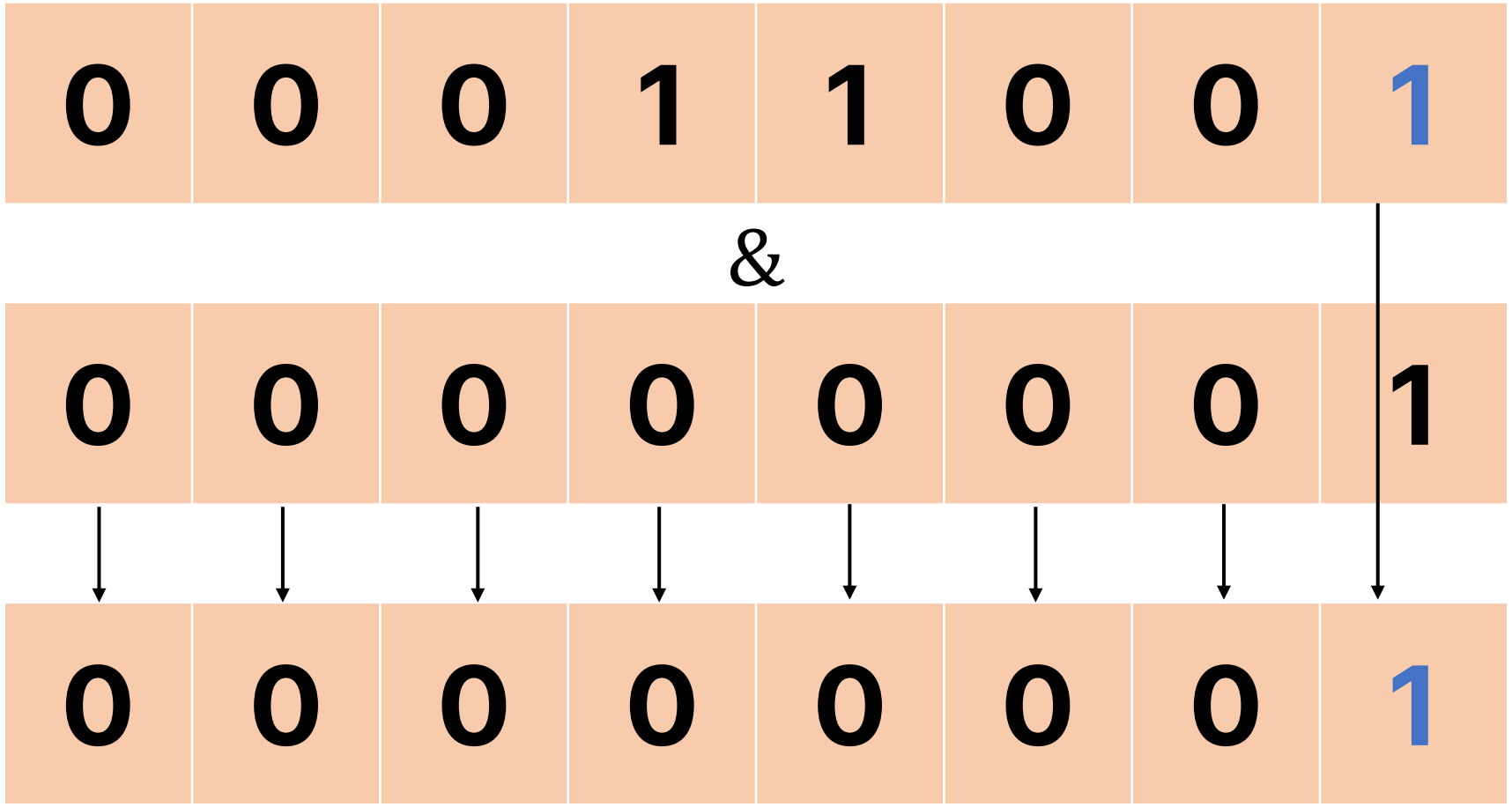
정수에서 비트를 이용한 트릭

- 00000001을 AND연산하는 경우 앞 7자리는 모두 0으로 사라진다.
- 그리고 끝 비트에 따라 0 또는 1이 나오게 된다

정수에서 비트를 이용한 트릭



정수에서 비트를 이용한 트릭



정수에서 비트를 이용한 트릭

- 이러한 방식을 이용하여 1을 AND연산 해준 결과가 1인 경우 홀수, 0인 경우 짝수임을 알 수 있다

```
if ((num & 1) == 1) {
    // 홀수
} else {
    // 짝수
}
```

Bitmasking Algorithm

- 지금까지 배운 내용은 비트를 다루는 방법과 비트로 각 데이터를 다루는 예시를 배웠다
- 비트를 직접적으로 다루면 더 빠르고, 더 적은 메모리를 사용한다는 장점이 있다

- 그렇다면 알고리즘에서는 어떻게 사용할까?

Bitmasking Algorithm

- 직접적으로 비트를 사용해 연산을 하기도 하지만, **비트에 의미를 부여한다**
- 아까 나온 나누기 2 또는 홀수 짝수 판별 등 프로그램을 더 빠르게 만드는 것에 기여하기도 하지만, 각각의 비트에 내가 어떤 의미를 부여해 사용할 수 있다
- 즉, int변수를 int 하나로만 사용하지 않아도 된다는 의미이다

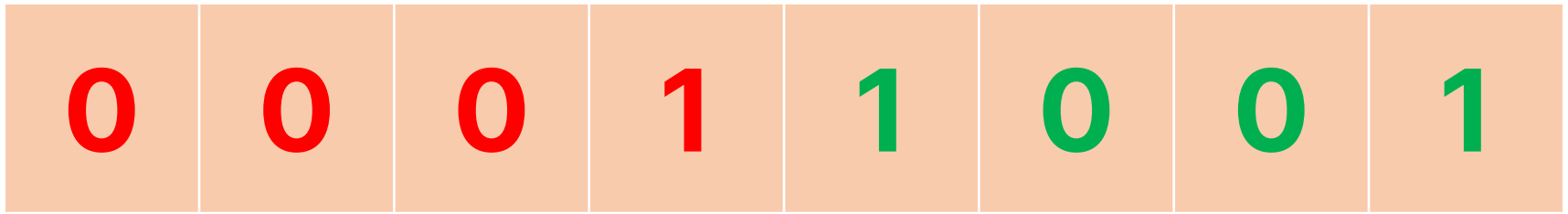
Bitmasking Algorithm

- 가장 간단한 예시를 보자.
- x좌표와 y좌표를 나타내는데 각각 1~10을 나타낸다 하자
- 지금까지 사용하는 방법은 int 변수 2개를 만들어서 저장하는 방식이다.

```
int x;
int y;
```

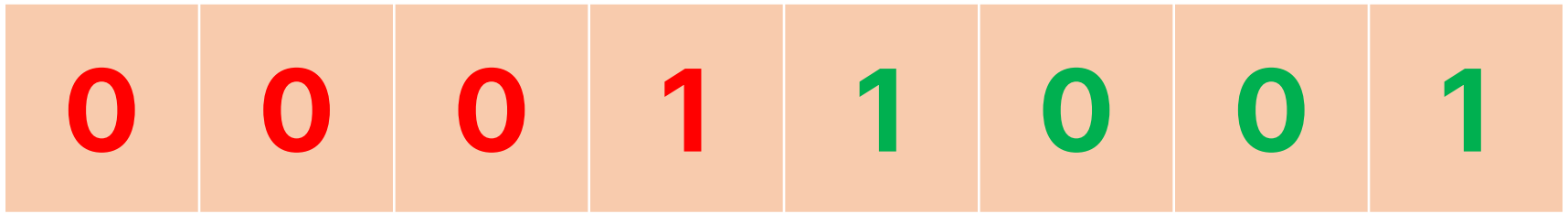
Bitmasking Algorithm

- 또 돌아온 8비트 예시이다
- 여기서 반을 나누어 생각해보자
- 비트마스킹 알고리즘은 내가 원하는 **의미**를 비트에 부여하는 것이다



Bitmasking Algorithm

- 내가 원하는 **의미**를 부여해보자
- 앞의 빨간 비트 4개는 X좌표를 의미하며, 뒤 초록 비트 4개는 Y좌표를 의미한다고 내가 의미를 부여하자



Bitmasking Algorithm

- 00011001을 단순히 사용하면 25라는 숫자이다
- 방금 부여한 의미로 생각해보면 25라는 숫자는 0001 1001로 나뉘며

  이것은 X좌표 1과 Y좌표 9를 의미하는 값으로 사용할 수 있다

0	0	0	1	1	0	0	1

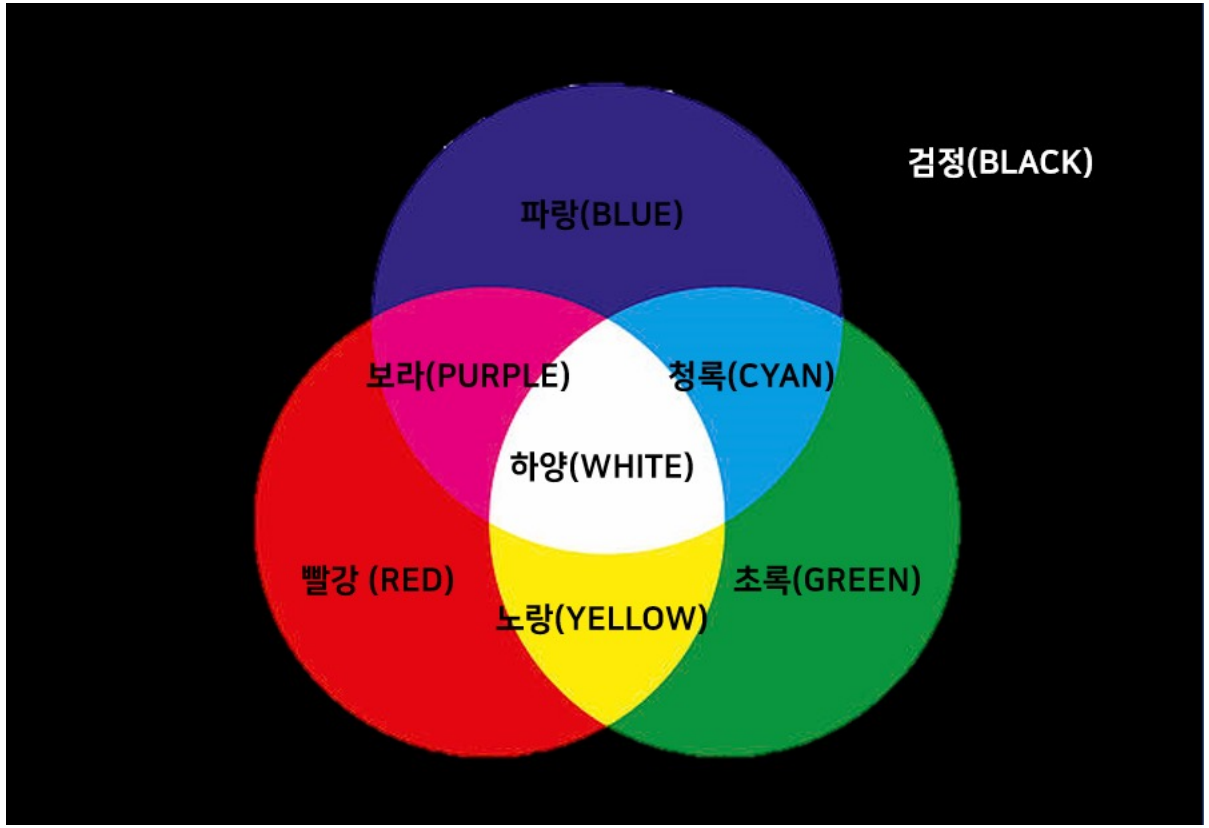
Bitmasking Algorithm

- 이러한 방식으로 내가 원하는 **의미**를 비트에 부여해 사용하는 방식이 비트마스킹 알고리즘이다
- 앞 예시처럼 4비트씩 묶어서 의미를 부여할 수도 있지만, 제일 자주 사용하는 것은 있다/없다(1/0)로 나뉘는 값을 각 비트에 부여하여 사용하는 방식을 제일 많이 사용한다

빛의 3원색

- 빛은 빨강, 초록, 파랑 3가지 색으로 많은 색을 표현할 수 있다
- 총 8가지 색이 존재한다 하자
- 아무런 빛이 없을 때 검정
- 한가지 색이 표현할 수 있는 빨강, 초록, 파랑
- 두가지 색의 조합으로 표현할 수 있는 보라, 청록, 노랑
- 세가지 색이 합쳐졌을 때 나오는 하양

빛의 3원색(출처 boj 16853)



빛의 3원색

- 색의 조합을 어떻게 표현할 수 있을까
- 가장 직관적인 방법을 사용해보자
- 각 색에 번호를 부여한다
- 그리고 조건문을 이용해 색의 합을 표현해보자

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

- 각 색을 먼저 정의

```
#define BLACK  0
#define RED    1
#define GREEN  2
#define BLUE   3
#define PURPLE 4
#define CYAN   5
#define YELLOW 6
#define WHITE  7
```

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

```
int color(int a, int b) {
    if (a == BLACK) {
        return b;
    }
    if (a == RED) {
        if (b == BLACK or b == RED)
            return RED;
        if (b == GREEN or b == YELLOW)
            return YELLOW;
        if (b == BLUE or b == PURPLE)
                return PURPLE;
        if (b == CYAN or b == WHITE)
            return WHITE;
    }
    //... too many code
}
```

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

- 색의 조합을 어떻게 표현할 수 있을까
- 가장 직관적인 방법을 사용해보자
- 각 색에 번호를 부여한다
- 그리고 조건문을 이용해 색의 합을 표현해보자

-> **단순 조건문은 너무 코드가 복잡해진다**

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

- 구조체를 사용해보자
- 각 빛이 존재하는 지 여부를 담아보자

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

```
struct Color {
    bool red;
    bool green;
    bool blue;
};
```

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

```
struct Color {
    int get_color() {
        if (!red and !green and !blue)
            return BLACK;
        if (red and !green and !blue)
            return RED;
        if (!red and green and !blue)
            return GREEN;
        if (!red and !green and blue)
            return BLUE;
        if (red and !green and blue)
            return PURPLE;
        // more code
    }
};
```

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

```
Color color(Color a, Color b) {
    Color ret;
    // a와 b중에서 하나라도 색상이 있다면
    // ret의 색상은 true
    ret.red = (a.red or b.red);
    ret.green = (a.green or b.green);
    ret.blue = (a.blue or b.blue);

    return ret;
}
```

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

- 구조체를 사용해보자
- 구조체에 색상을 나타내는 함수를 만들면서,
  색상을 합쳐주는 함수가 굉장히 간결해졌다
- 더 간단하게 할 수 있을까?

색	번호
검정	0
빨강	1
초록	2
파랑	3
보라	4
청록	5
노랑	6
하양	7

빛의 3원색

- 지금까지 색을 나타낼 때 빨간색이 있는지 없는지, 초록색이 있는지 없는지, 파란색이 있는지 없는지로 8가지 색상을 나타냈다
- 비트마스킹의 방식으로 표현하기 쉽다

  의미(색상)를 부여하고, 각 의미가 있다 없다로 표현이 가능하다

빛의 3원색

- 빨강, 초록, 파랑을 각 비트에 의미로써 부여해보자
- 끝 3비트를 제외하고 나머지 비트는 사용할 필요가 없다
- 각 자리에 의미(색상)를 부여하며, 만일 R이 1인 경우 빨강색이 존재, 0인 경우 존재하지 않는다는 의미를 부여하자

0	0	0	0	0	R	G	B

빛의 3원색

- 색을 합치는 연산은 두 색 중에 둘 중 하나라도 해당 색을 가지고 있으면 합친 색에서도 나타난다
- 즉 둘 다 없을 때는 0, 둘 중 하나 이상 있다면 1로 돌려주면 된다
- OR연산을 사용하면 됨을 알 수 있다

0	0	0	0	0	R	G	B

빛의 3원색

- 비트 연산에 사용할 int 하나를 넣어두자

```
struct Color {
    int color;
    // char도 가능
};
```

빛의 3원색

- 색을 합칠 때는 OR연산을 사용하면 되므로 두 색을 합치는 것은 OR 연산을 사용해 돌려주도록 하자

```
Color color(Color a, Color b) {
    Color ret;

    ret.color = a.color | b.color;

    return ret;
}
```

빛의 3원색

- 색을 나타내주는 get_color함수를 다시 정의하자

- 이전의 예시를 다시 살펴보자

Bitmasking Algorithm

- 00011001을 단순히 사용하면 25라는 숫자이다
- 방금 부여한 의미로 생각해보면 **25라는 숫자는 0001 1001로 나뉘며**

  이것은 X좌표 1과 Y좌표 9를 의미하는 값으로 사용할 수 있다

0	0	0	1	1	0	0	1

빛의 3원색

- 예시에서 25라는 숫자가 X좌표 1과 Y좌표 9를 의미하는 것처럼 각 숫자가 의미하는 색상을 다시 설정해보자
- 각 비트의 의미와 0과 1 값을 생각해 다시 골라보자

빛의 3원색

- 이처럼 수정함으로 숫자 자체가 색상을 의미하게 나타낼 수 있다

색	비트	번호
검정	000	0->0
빨강	100	1->4
초록	010	2->2
파랑	001	3->1
보라	101	4->5
청록	011	5->3
노랑	110	6->6
하양	111	7->7

빛의 3원색

```
#define BLACK  0 // 000
#define RED    4 // 100
#define GREEN  2 // 010
#define BLUE   1 // 001
#define PURPLE 5 // 101
#define CYAN   3 // 011
#define YELLOW 6 // 110
#define WHITE  7 // 111
```

색	비트	번호
검정	000	0->0
빨강	100	1->4
초록	010	2->2
파랑	001	3->1
보라	101	4->5
청록	011	5->3
노랑	110	6->6
하양	111	7->7

Bitmasking Algorithm

- 이런 예시들처럼 비트에 의미를 부여함으로 데이터 압축, 편리함 등을 챙길 수 있다
- 비트마스킹은 공식보다는 내가 특정한 의미를 부여하는 것이므로, 정확하게, 그리고 명확하게 의미를 표현하는 것이 중요하다

문제

- 사격 내기 BOJ 27960
- 막대기 BOJ 1094
- 데스스타 BOJ 11811
- 외판원 순회 BOJ 2098