CSAPP 2강 정수 연산, IEEE 754

정수의 연산

오늘의 퀴즈

컴퓨터의 연산에서 $0 < a <= b$ 라고 할 때
$a + b < 0$ 이 성립할 수 있을까?

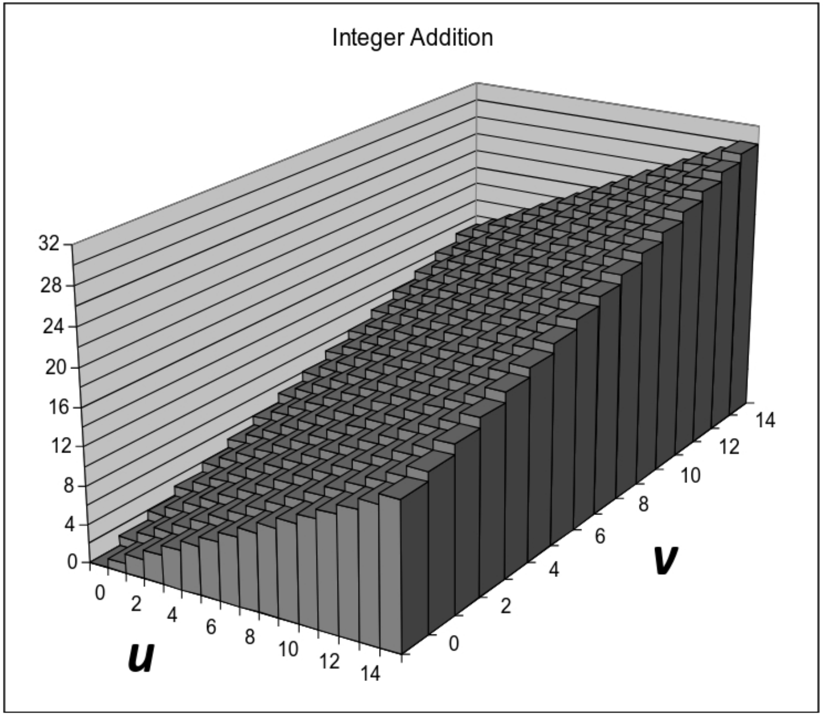
만약 그렇다면 $a + b = -1$을 만족하는 a, b의 값은?

(진짜) 정수 덧셈의 시각화

컴퓨터의 연산 말고 진짜 수학 연산!!
u와 v의 값에 따라 선형적으로 증가한다.

평평한 표면을 만든다.

$\mathbf{Add_4}(u , v)$

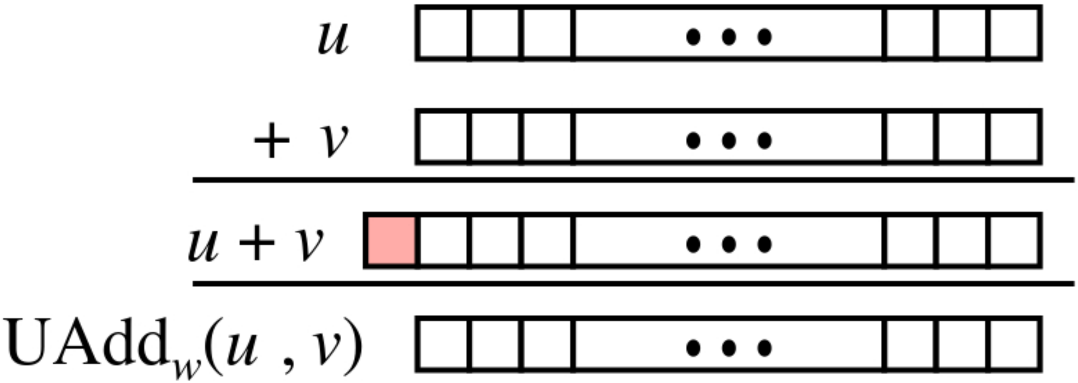


Unsigned 덧셈

w비트 길이를 가진 unsigned int u, v를 더해보자

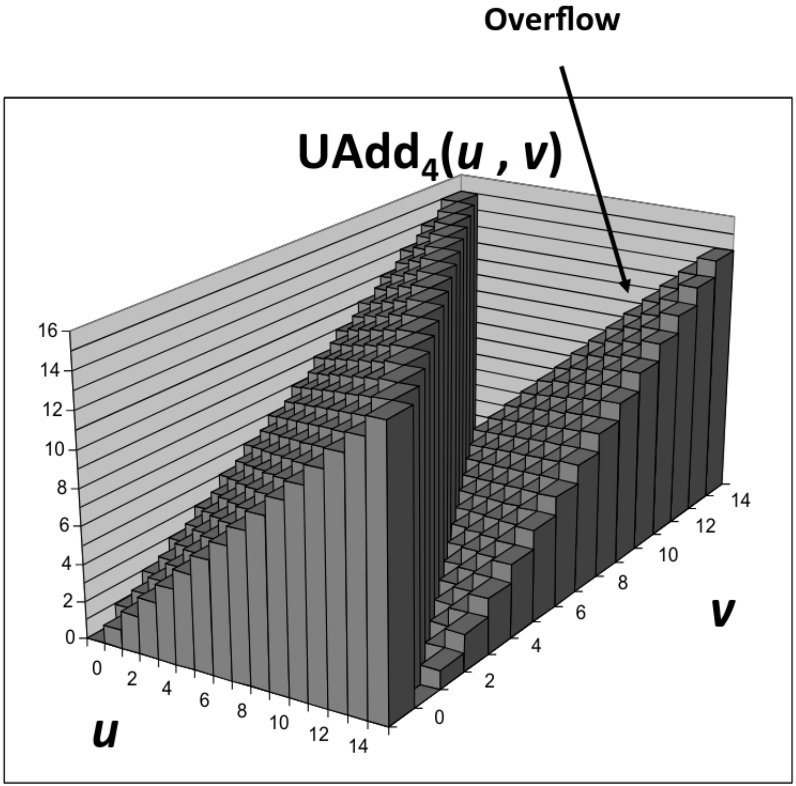
u + v의 값은 최대 w+1비트를 필요로 한다.
앞으로 실제 수학에서의 u + v값을 true sum이라고 하자.
그렇지만 컴퓨터의 정수 덧셈에서 w를 넘어가는 비트는 버려지게 된다.

$(u+v) mod 2^w$
즉, 모듈러 산술



Unsigned 덧셈의 시각화

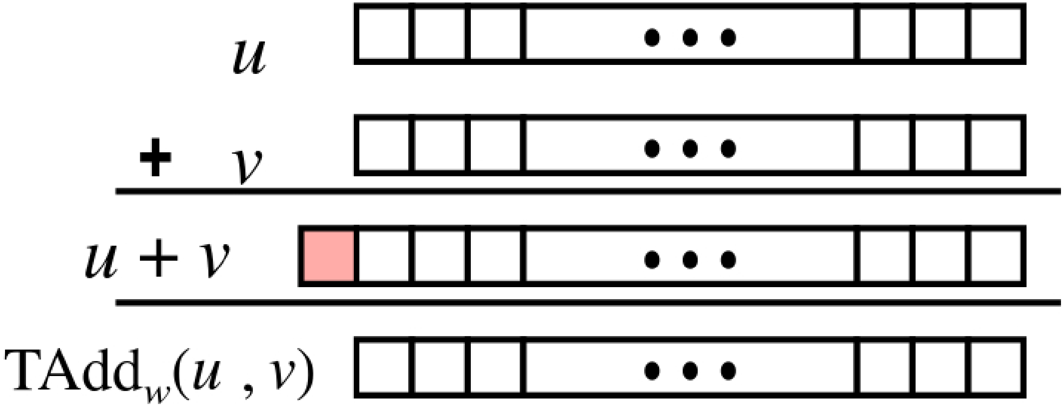
true sum이 2^w 이상이면 오버플로우가 발생한다.
오버플로우는 최대 한번



2의 보수 덧셈

2의 보수 u, v를 더해보자
true sum을 표현하려면 w+1비트가 필요

unsigned와 마찬가지로 w를 넘어가는 비트는 버려진다.



주의!

지난 주에 하나의 비트 벡터를 unsigned, 2의 보수 이렇게 두 방식으로 해석할 수 있다고 했다.
덧셈의 결과도 마찬가지.
unsigned를 더하나 signed를 더하나 비트 패턴은 동일하다.
결과를 어떻게 해석할지가 다른 것

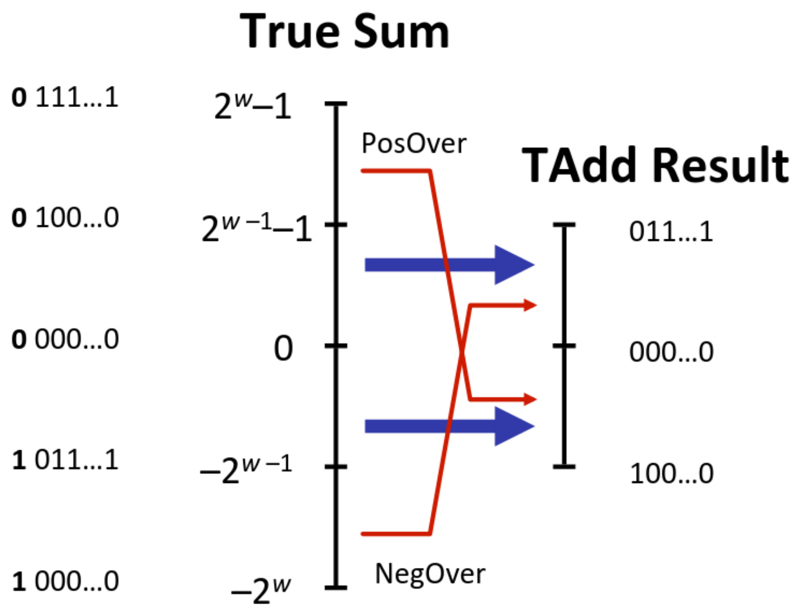
```c
int s, t, u, v;
s = (int) ((unsigned)u + (unsigned)v);
t = u + v

s == t // true
```

TAdd 오버플로우

w+1 비트에서 제일 큰 비트가 버려지기 때문에,
그 다음 비트에 따라 음수/양수가 결정된다.

같은 부호 덧셈을 해도 부호가 바뀌는 이유는 이것 때문.

남은 비트를 2의 보수로 해석하면 된다.



2의 보수 덧셈의 시각화

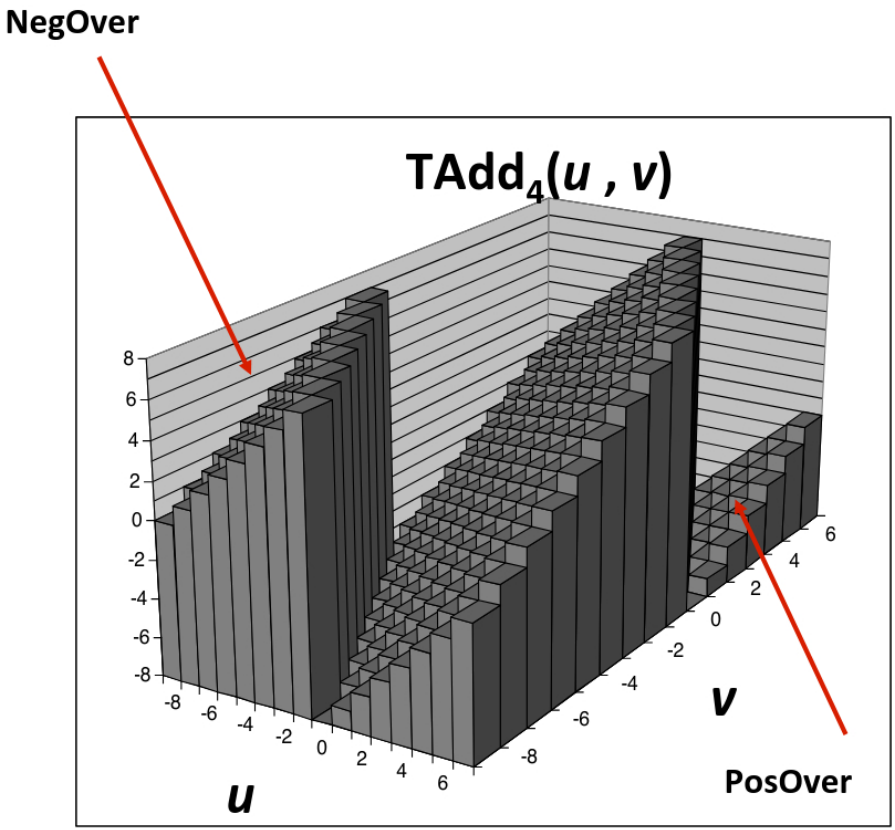
true sum >= 2^w-1 이면

- 음수가 된다.

  true sum < -2^w-1이면

- 양수가 된다.

  음의 오버플로우, 양의 오버플로우



곱셈

덧셈 다음에는 곱셈이죠?

w비트 x, y를 곱해봅시다.

덧셈은 true sum을 표현하는데 1 비트가 더 필요했다.
곱셈 역시 w비트로는 결과를 다 표현할 수 없다.

- unsigned의 경우: 최대 2w의 비트
    - $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - 왜 2w비트인지 설명 ㄱㄱ
- 2의 보수 최솟값의 경우: 2w-1 비트
    - $x * y \ge (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - 유도 방법
        - n 비트로 표현할 수 있는 최솟값은 -2^{n-1} 임을 지난 주에 확인했다.
        - w 비트 길이를 가지는 숫자 둘을 곱한 최솟값은 $-2^{2w-2} + 2^{w-1}$ 이다. (최솟값 * 최댓값)
        - $-2^{n-1} \le -2^{2w-2} + 2^{w-1}$ 을 만족하는 최소 n은 2w -1이다.

    - 따라서 $-2^{2w-2}+2^{w-1}$를 표현하기 위한 최소 비트 수는 2w-1이다.
- 2의 보수 최댓값의 경우: 2w 비트
  - $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - 2^{2w-2}는 표현하려면 2w-1비트가 필요함
  - 2의 보수에서 양수는 제일 앞 비트가 0이어야 하기 때문에 총 필요한 비트는 2w개

결과를 유지하려면...
비트가 매번 이렇게 확장 되어야 함

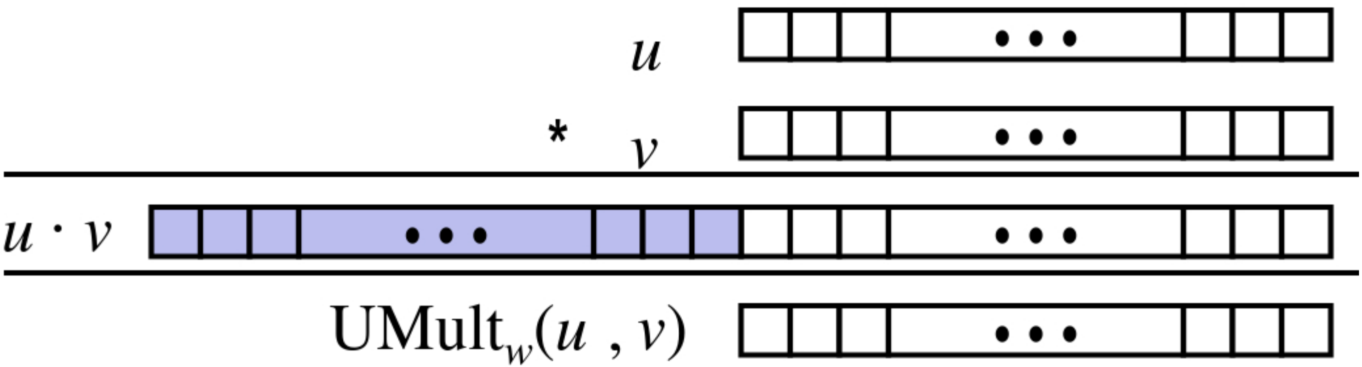
실제로 임의 정밀도 산술같은 경우 계산마다 word 크기가 확장됨 (bignum 산술)

C에서의 unsigned 곱셈

w길이를 가지는 u v를 곱할 때 실제 값을 true product라고 하자.

true product의 최대 길이는 아까 본 것 처럼 2w이다.
2w길이로 나온 결과의 앞 w비트를 버리면 끝!

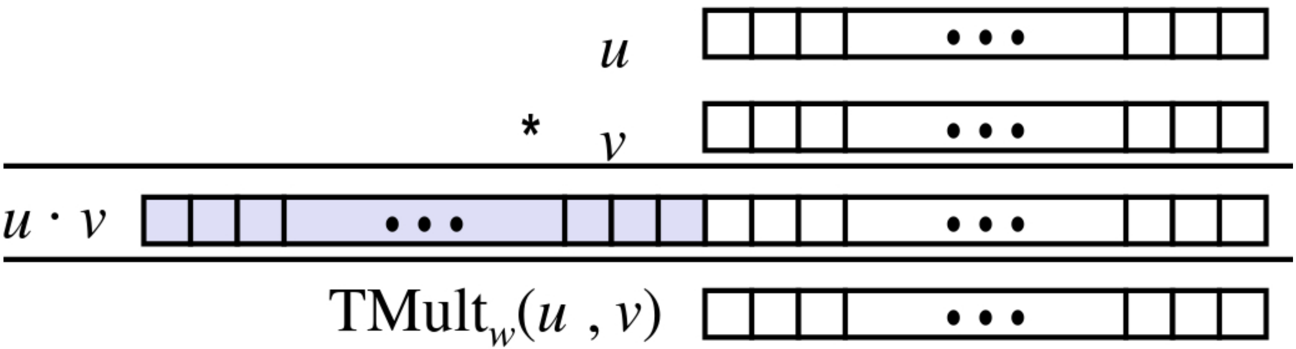
- 상위 w비트는 버려진다.
- 곱셈 역시 모듈러 산술
  - $UMult_w(u, v) = u * v \; mod 2^w$



C에서의 signed 곱셈

마찬가지로 true product의 최대 길이는 2w이다.
(Tmin * Tmin 일 경우)

- unsigned에서 했던 것 처럼 상위 w비트는 버려진다.
- unsigned와 결과가 다를 수 있다.
- 하위 비트는 같다.



시프트를 이용한 2의 거듭제곱 곱하기

u << k 연산은 u * 2k를 한 값이다.
시프트 방법은 지난 주에 다뤘으니 생략

w길이의 비트를 k만큼 시프트 한 걸 표현하려면 w+k비트가 필요하다.
이때 상위 k비트는 버린다.

거듭제곱 예시

- u << 3 == u * 8
- (u << 5) - (u << 3) == u * 24

대부분의 머신은 곱셈보다 시프트와 덧셈이 빠르다.

- 상수를 곱하는건 컴파일러가 자동으로 바꿔준다.

연습문제

x * 18를 시프트 연산과 덧셈으로 구현하면?

시프트를 이용해 2^k로 나누기 (unsigned)

거듭나눔이란 말은 없나봐.
$\frac{1}{2}$의 거듭제곱?

u >> k연산은 $[u/2^k]$ 가 된다. (가우스 or floor함수) 적용됨
unsigned이니 논리 시프트를 사용한다.

오른쪽으로 밀린 부분은 절삭되기 때문에 floor함수를 적용한 것 처럼 정수 결과만 남는다.

정수 연산은 끝

질문 있으면 하십쇼

Why should I use unsigned?

일단, 잘 모르겠으면 쓰지마십쇼 왜냐?

(코드 뭐가 문제인지 찾아보자.)

- 실수하기 쉽다고

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
```

- 미묘~하다고

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
    do_something();
```

unsigned로 반복 횟수를 정하려면?

아주 좋은 방법이 있다! (근데 직관적이진 않음..ㅎㅎ)

```
unsigned i;
for (i = cnt - 2; i < cnt; i--)
    a[i] += a[i+1];
```

C 표준은 unsigned의 덧셈이 모듈러 산술과 같이 동작함을 보장한다.
-> 0 - 1 은 음수가 아니라 UMax임

조금 더 개선하면?

```
size_t i;
for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
```

- size_t는 시스템의 word사이즈와 같은 길이를 가짐 (64비트 시스템의 경우 64비트)

- 만약 위의 unsigned 버전에서 cnt가 unsignunsigned 변수 여러개를 조합한다.ed의 범위보다 큰 수였다면?
  ex) unsigned long long 의 최댓값(UMax 등..)
- [다른 타입간의 비교에서 형 변환과 연산의 순서]

그래서 언제 unsigned를 써야 하냐구요

- 모듈러 산술
  - unsigned의 연산 결과는 항상 실제 값 mod 2^w 를 한 것과 같다.
  - signed의 경우 부호가 달라질 수 있음
- 다중 정밀도 산술
  - 큰 수를 표현할 때 여러개의 변수를 조합해서 계산한다.
  - 이 경우 unsigned를 쓰면 각 워드 연산이 자연스럽게 모듈러 산술이 되고, carry 처리만 해 주면 됨
- 비트로 집합을 표현할 때
  - 우측 시프트가 논리 시프트로 동작

부동 소수점..................

전 이게 정말 싫었어요..

1011.101_2 이거 계산 가능?

이거 계산 방법은 정수때랑 동일하다.

$$b_i\ b_{i-1}\ \ldots\ b_1\ b_0\ b_{-1}\ \ldots\ b_{-j}$$

라고 하면

$$\sum_{k=-1}^{i} b_k \times 2^k$$

를 하면된다.
(근데 이건 부동 소수점 표현이 아니야!)

비율 이진수 표현의 한계

제한 1: x/2^k 로 표현되는 숫자만 정확히 표기 가능

- 다른 숫자는 반복되는 비트 표현을 가진다. (무한소수처럼)
- `1/3 -> 0.01010101[01]...`
- `1/5 -> 0.001100110011[0011]...`

제한 2:
소수점이 고정되면 표현할 수 있는 범위가 매우 제한된다.

- 소수 부분이 길면 매우 작은 단위까지 표현할 수 있지만 전체 범위가 작아짐
- 정수 부분이 길면 전체 범위는 넓어지지만, 표현할 수 있는 소수 부분이 작아짐
- Fixed 클래스를 떠올려 보자!

IEEE Floating Point

IEEE 754: 부동 소수점을 표현하는 가장 널리 쓰이는 표준

근사, 오버플로우, 언더플로우 같은 요소들이 정밀하게 다뤄진다.
하드웨어에서 빠르게 구현하기는 어렵다.
성능보다는 수치적 안정성과 정확성에 중점

정밀도

단정밀도: 32비트
C언어의 float형에 대응

배정밀도: 64비트
C언어의 Double형에 대응

- Single precision: 32 bits

- Double precision: 64 bits

부동소수점 표현

$$(-1)^s \; M \; 2^E$$

- 부호 비트 s가 양수 음수 여부를 결정한다.
- 유효숫자 M은 비율 이진수로 1과 $2-\epsilon$사이 또는 0과 $1-\epsilon$ 사이의 값을 가진다.
- 지수 E는 2의 제곱으로 자리값을 제공한다.

인코딩

- s | exp | frac 순서
- exp와 frac은 각각 E와 M의 표현이지만, 둘의 값이 동일하지는 않다.

주어진 비트 표시로 인코딩 된 값은 exp 값에 따라 세 개의 다른 경우들로 나눌 수 있다.

case 1: 정규화 값

exp의 비트 패턴이 모두 0 또는 모두 1이 아닌 가장 일반적인 경우

이 경우는 지수 E는 exp - bias 가 된다.

- exp 필드의 unsigned value에 bias를 뺀 값
- bias는 $2^{k-1} - 1$, k는 지수 비트의 길이
    - 단정밀도의 bias = 127
    - 배정밀도의 bias = 1023

유효숫자 M은 1 + f 로 정의된다.
즉 frac 부분은 1.xxxx부분에서 xxxx부분만을 표현한다.

- frac=000...0 일 때 유효숫자의 최솟값 (M = 1.0)을 가진다.
- frac=111...1 일 때 유효숫자의 최댓값 (M = 2.0 - ϵ)
- 비트 하나를 공짜로 get!

case2: 비정규화 값

exp의 비트 패턴이 모두 0인 경우

이 경우 지수 E는 1-bias가 된다. (0-bias가 아님!! 주의!)

유효숫자 M은 0 + f로 정의된다.
즉 frac은 0.xxxx 부분에서 xxxx를 표현

비정규화 값을 쓰는 목적

- 0을 표현 (+0, -0이 존재한다.)
- 0.0에 매우 가까운 값 표현(0 근처에서 같은 간격을 가짐)

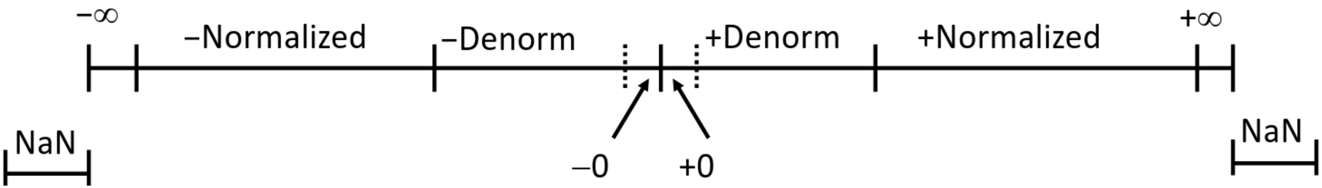
case3: 특수 값

exp의 비트 패턴이 모두 1인 경우

frac필드가 모두 0이면 무한대를 나타낸다.
0과 마찬가지로 부호 비트에 따라 $+\infty, -\infty$가 존재한다.

frac필드가 0이 아니면 NaN (Not a number)
이 값은 $\sqrt{-1}$ 또는 $\infty - \infty$같은 연산의 결과로 리턴된다.

부동 소수점의 시각화



비정규화 값과 정규화 값의 연결

```
부호 1비트
exp 4비트
frac 3비트

bias = ?

정규화 값 E = exp - bias
비정규화 값 E = 1 -  bias
--------------------------------------
E, M을 구하고 2^E * M 을 하여 값을 구해보자

0 0000 000
0 0000 001
0 0000 010
...
0 0000 111
0 0001 000
```

```
0 0001 001
...
0 1110 110
0 1110 111
0 1111 000
```

IEEE형식은 부동소수점 숫자들이 정수 절렬 루틴을 사용해서 정렬될 수 있도록 설계되었다.
비트 그대로 정수로 해석해서 비교 가능!

부동 소수점 연산

$$x +_f y = Round(x + y)$$

$$x \times_f y = Round(x \times y)$$

계산 방법

- 연산의 실제 결과를 계산하고, 원하는 정밀도에 맞춘다.
    - 오버플로우가 날 수 있음
    - frac에 맞추기 위해 근사가 일어날 수 있음

근사

IEEE 부동소수점 형식은 네 가지 근사 모드를 정의하고 있다.

1. 짝수근사법(default): 가까운 방향으로, 중간 값일 경우 짝수 방향으로
2. 영방향근사: 0에 가까운 방향으로
3. 하향근사: 모두 아래쪽으로 근사
4. 상향근사: 모두 위쪽으로 근사

	\$1.40	\$1.60	\$1.50	\$2.50	–\$1.50
Towards zero	\$1	\$1	\$1	\$2	–\$1
Round down (−∞)	\$1	\$1	\$1	\$2	–\$2
Round up (+∞)	\$2	\$2	\$2	\$3	–\$1
Nearest Even (default)	\$1	\$2	\$2	\$2	–\$2

짝수근사

근사의 기본 모드

- 어셈블리단의 시스템 코드를 고치지 않는 이상 바꾸기 어렵다..
- 다른 모드는 통계적인 편향을 가지게 된다.
- 짝수 방향으로 근사하게 되면 이런 편향을 대부분 회피 가능

정수가 아닌 실수로의 근사에도 사용 가능

마찬가지로 이진수 비율 숫자에도 적용될 수 있음..!

부동 소수점 곱셈

$(-1)^{s1}\ M1\ 2^{E1} \times (-1)^{s2}\ M2\ 2^{E2}$
결과:

- 부호(s): s1 ^ s2
- 유효숫자(M): M1 x M2
- 지수(E): E1 + E2\

이 결과를

- M>=2면, M >> 1 and E++
- E의 범위를 넘어가면 오버플로우
- M을 frac 정밀도에 맞게 근사

부동 소수점 덧셈

$(-1)^{s1}\ M1\ 2^{E1} + (-1)^{s2}\ M2\ 2^{E2}$

E1 > E2라고 가정

결과:

- E2를 E1에 맞추기 위해 M2를 적절히 시프트
- M1, M2를 부호에 맞게 덧셈/뺄셈
- 지수는 E1유지

결과 수정:

- M>= 2면 M >> 1 and E++
- M<1 면, M << k and E -= k
- E범위 넘어가면 오버플로우
- M은 정밀도에 맞게 근사

부동 소수점 덧셈의 수학적 성질

Unsigned / 2의 보수 덧셈(아벨 군)과의 비교

같은 점:

- 연산(덧셈)에 대해 닫혀있다(무한대와 NaN이 될 수 있음).
- 교환법칙이 성립한다.
- 항등원이 존재한다 (0).
- 역원이 존재한다 (무한대와 NaN만 빼고)

차이점:

- 결합법칙이 성립하지 않는다.

부동 소수점 곱셈의 수학적 성질

unsigned / 2의 보수 곱셈(가환환)과의 비교

같은점:

- 연산에 대해 닫혀있다(무한대와 NaN이 될 수 있음).
- 교환법칙이 성립한다.
- 항등원이 존재한다(1)

차이점:

- 결합법칙이 성립하지 않는다.
- 분배법칙이 성립하지 않는다.
    - `1e20 * (1e20-1e20) = 0.0`
    - `1e20 * 1e20 - 1e20 * 1e20 = NaN`

C에서의 부동소수점

단정밀도와 배정밀도를 각각 float, double로 제공한다.

자료형 간의 캐스팅:
(자료형 int가 32비트라고 가정시)

- int -> float: 근사될 수도 있다.
- int -> double: 정확한 수치 값이 보존된다.
- double -> float: 근사되거나 오버플로우할 수 있다.
- double/float -> int: 0 방향으로 근사된다.
    - NaN이나 범위를 벗어난 수는 정의되지 않음 (일반적으로는 TMin으로 변환됨)

마치며

최적화 그거 컴파일러가 해주는거 아냐? 라고 생각하셨다면,,

```
x = a + b + c;
y = b + c + d;
// 부동소수점 덧셈 4회
```

컴파일러가 위 코드를 아래처럼 최적화 할까?

```
t = b + c;
x = a + t;
y = t + d;
// 부동소수점 덧셈 3회
```

컴파일러는 사용자가 효율성과 정확성 사이에 어떤 절충을 원하는지 모름

-> 최적화 안함 ㅅㄱ

프로그래머가 코드를 잘 짜야하는 이유