# 시스템 프로그래밍

강의 4 : 2.4 실수의 표현 및 처리

2010년 9월 15일

김 형신

http://eslab.cnu.ac.kr

# 전달사항

- 질문 : 1991년 걸프전쟁에서 왜 패트리엇 미사일이 스커드 미사일을 격추하지 못했는가 ?

# Floating point 퍼즐

- 아래의 C 프로그램을 보고 :
  - 모든 값에 대해 항상 성립하는 관계인지 생각해 보라
  - 그렇지 않은 이유는 무엇인지 설명해 보라

```
int x = …;

float f = …;

double d = …;
```

d 나 f는 NaN 은 아니다

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` ⇒ `((d*2) < 0.0)`
- `d > f` ⇒ `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

# IEEE Floating Point

## IEEE 표준 754

- 실수(floating point) 연산을 위한 단일 표준으로 1985년에 제정
  - 이전까지는 다양한 형태가 존재
  - 인텔사의 지원으로 8087 프로세서 개발목적으로 추진
- 모든 주요 CPU에서 IEEE Floating Point라는 이름으로 지원됨

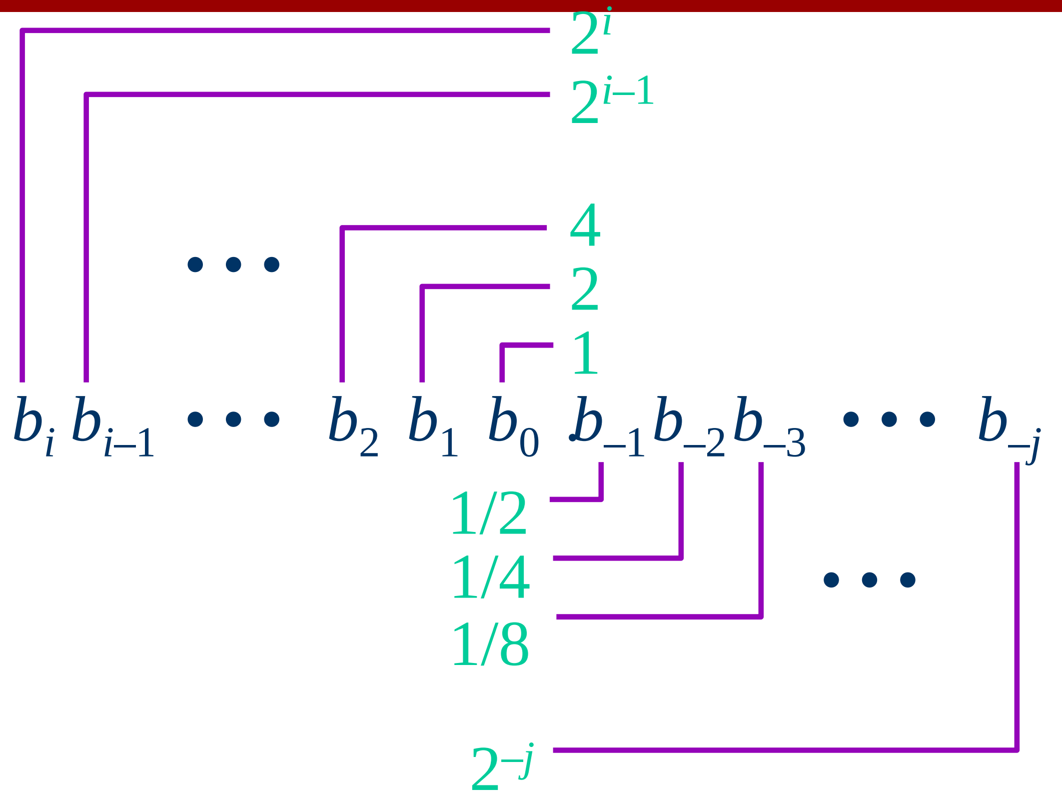
## 수치해석적인 측면을 고려하여 정의됨

- rounding, overflow, underflow 등에 유용함
- 빠른 연산을 수행하기가 어려운 단점이 있다
  - 표준 정의시 수치해석자들의 수가 하드웨어 연구자들 보다 많았다

## Floating point 와 프로그래머

- 무관심 , 흥미상실, 이해하기 어려운 내용
- 우아하고 이해할만한 내용

# 2진 소수




**표현방법**

- **이진 소수점 우측의 비트들은 2의 분수제곱을 의미**
- **소수를  다음과 같이 표시:**

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

# 2진 소수 예제

| 값 | 2진 소수 표시 |
|---|---|
| 5 와 3/4 | $101.11_2$ |
| 2 와 7/8 | $10.111_2$ |
| 63/64 | $0.111111_2$ |

관찰

- 우측으로 쉬프트 하면 **2**로 나눈 효과를 얻음
- 좌측으로 쉬프트하면 **2**를 곱하는 효과를 얻음
- **1.0** 에 근접하는 $0.111111\ldots_2$ 과 같은 수들은 다음과 같이 표시한다
  - $1/2 + 1/4 + 1/8 + \cdots + 1/2^i + \cdots \rightarrow 1.0$
  - $1.0 - \varepsilon$

# 표시 가능한 수

한계

- $x/2^k$ 형태 만 정확히 표시가능
- 비트들이 무한 반복되는 경우는 정확히 표시할 수 없다
- **5 x 2**

- **$5 \times 2^{100}$** 은 어떻게 표시되겠는가**?**
  - 큰 수의 표시는 이렇게 자리값을 이용해 표시하는 방법을 사용할 수 없다

| 값 | 표시 |
|---|---|
| **1/3** | $0.0101010101[01]\ldots_2$ |
| **1/5** | $0.001100110011[0011]\ldots_2$ |
| **1/10** | $0.0001100110011[0011]\ldots_2$ |

- 대신에,
  - 큰 수의 표시는 이렇게 자리값을 이용해 표시하는 방법을 사용할 수 없다
  - $x \times 2^y$ 의 형태로 수를 표시하고, **x, y**를 이용해 표시하는 방법을 이용

# IEEE Floating Point 표준

## 수식적 표현

- $(-1)^s\ M\ 2^E$
  - 부호비트 ***s*** 는 양수/음수 여부를 표시
  - 유효숫자 ***M*** 은 **[1.0,2.0)** 또는 **[0.0, 1.0)** 사이의 실수 값을 표시
  - 지수 ***E*** 은 **2**의 지수제곱을 표시

## 인코딩

| s | exp | frac |
|---|---|---|

- **MSB** 는 부호 비트
- exp 필드는 ***E*** 를 인코딩
- frac 필드는 ***M*** 를 인코딩

# Floating Point의 정밀도

인코딩

| s | exp | frac |
|---|---|---|

- **MSB** 는 부호 비트
- exp 필드는 ***E*** 부분을 인코딩
- frac 필드는 ***M*** 부분을 인코딩

크기

- **Single** 정밀도**(float ): 8** exp 비트**, 23** frac 비트
  - 총 **32** 비트
- 더블**(double): 11** exp 비트**, 52** frac 비트
  - 총 **64** 비트
- 확장**(Extended precision): 15** exp 비트**, 63** frac 비트
  - **Intel** 호환 컴퓨터에서만 사용
  - **80** 비트표시
    - **1** 비트는 사용하지 않음

인코딩은 **exp** 값에 따라 세 가지의 경우로 달라진다
**(Normalized, Denormalized, Special values)**

# 정규화된 값(Normalized values)

| s | exp | frac |
|---|---|---|

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

조건

- exp ≠ 000…0 그리고 exp ≠ 111…1 인 경우에 사용

지수부는 ***biased* value** 를 형식으로 표시된다

- $E$ = $exp - bias$
  - ***Exp* : unsigned value**
  - ***Bias* : Bias value**
    - **Single precision: 127 (*Exp*: 1…254, *E*: −126…127)**
    - **Double precision: 1023 (*Exp*: 1…2046, *E*: −1022…1023)**
    - 일반적으로 : ***Bias* = $2^{e-1}$ − 1,** 여기서 **e** 는 지수비트의 갯수임

유효숫자는 묵시적으로 **1**로 시작하는 것으로 간주한다

- $M$ = $1.xxx\cdots x_2$
  - xxx…x: frac 부분
  - 000…0 일 때 최소(***M* = 1.0**)
  - 111…1 일 때 최대(***M* = 2.0** − ε)
  - 이와 같이 함으로써 **1**비트를 무료로 표시할 수 있게 된다

# 정규화 인코딩 예제

## Value

`Float F = 15213.0;`

- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \text{ X } 2^{13}$

## 유효숫자(Significand)

$M$ = $1.\underline{1101101101101}_2$

frac= $\underline{1101101101101}0000000000_2$ <= 23 비트

## 지수(Exponent)

$E$ = 13

$Bias$ = 127

$Exp$ = 140 = $10001100_2$ <= 8 비트

Floating Point Representation:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hex: | 4 | 6 | 6 | D | B | 4 | 0 | 0 |
| Binary: | 0100 | 0110 | 0110 | 1101 | 1011 | 0100 | 0000 | 0000 |
| 140: | 100 | 0110 | 0 | | | | | |
| 15213: | | | *1*110 | 1101 | 1011 | 01 | | |

# 비정규화 인코딩(Denormalized value)

적용 조건

- exp = 000…0

인코딩

- 지수(Exponent value) $E = 1 - Bias$
- 유효숫자(Significand value) $M = 0.\mathrm{xxx}\cdots\mathrm{x}_2$
  - xxx…x: bits of frac

두 가지 목적으로 이용

- **case I** : exp = 000…0, frac = 000…0
  - **0**을 표시
  - **+0** 과 –**0** 의 경우 표시가 다르다는 점에 주의
- case II : exp = 000…0, frac ≠ 000…0
  - **0.0**에 매우 근접한 소수값을 표시
  - “점차적인 언더플로우” 특성 : **0.0** 부근의 숫자들이 동일 간격으로 분포한다

# 특별 값(Special Values)

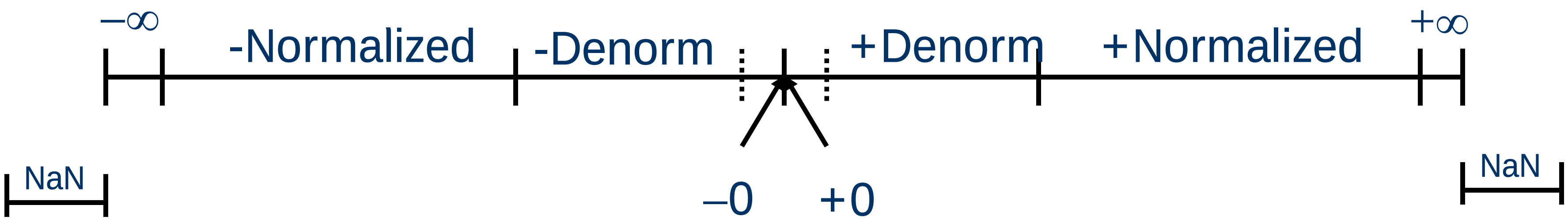
적용조건

- exp = 111…1

두 가지 경우

- exp = 111…1, frac = 000…0
  - 무한대 ∞ (infinity)를 표시(+/-)
  - 오버플로우를 표시할 수 있다.
  - E.g., 1.0/0.0 = –1.0/–0.0 = +∞, 1.0/–0.0 = –∞
- exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - 숫자로 표시할 수 없는 결과를 나타낼 때 사용
  - E.g., sqrt(–1), $\infty - \infty$, $\infty * 0$

# 실수 표현 요약

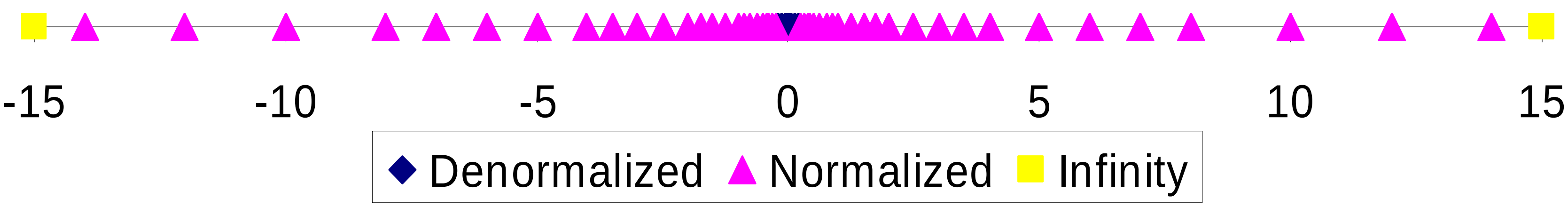


# 값의 분포 – 6비트 체계

## 6 비트 IEEE 유사한 포맷의 경우

- e = 3 비트 지수
- f = 2 비트 소수
- 바이어스 = 3

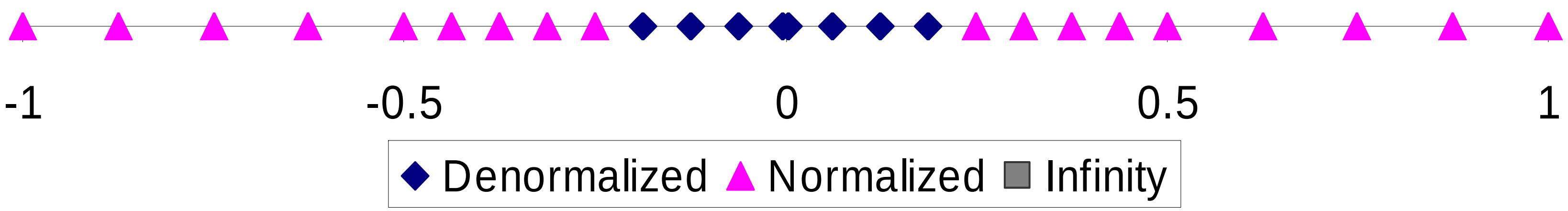
## 0 부근에서 왜 분포가 조밀해 지는지 파악해 보자



# 값의 분포(확대)

## 6 비트 IEEE 유사한 포맷의 경우

- e = 3 비트 지수
- f = 2 비트 소수
- 바이어스 = 3



# 간단한 Floating Point 시스템 예제

## 8-bit Floating Point 표시

- 부호비트는 **MSB**로 표시.
- 다음 **4**비트는 바이어스 값 **7**로 지수부로 사용.
- 마지막 **3**비트는 frac

## ● IEEE Format 과 동일한 일반적인 형태

- 정규화, 비정규화 표시 사용
- **0, NaN, infinity** 등 표시 사용

| 7 | 6 … 3 | 2 … 0 |
|---|---|---|
| s | 지수(exp) | 소수(frac) |

# 지수 값에 관련된 값들

| Exp | exp | E | $2^E$ | |
|---|---|---|---|---|
| 0 | 0000 | -6 | 1/64 | (비정규화) |
| 1 | 0001 | -6 | 1/64 | |
| 2 | 0010 | -5 | 1/32 | |
| 3 | 0011 | -4 | 1/16 | |
| 4 | 0100 | -3 | 1/8 | |
| 5 | 0101 | -2 | 1/4 | |
| 6 | 0110 | -1 | 1/2 | |
| 7 | 0111 | 0 | 1 | |
| 8 | 1000 | +1 | 2 | |
| 9 | 1001 | +2 | 4 | |
| 10 | 1010 | +3 | 8 | |
| 11 | 1011 | +4 | 16 | |
| 12 | 1100 | +5 | 32 | |
| 13 | 1101 | +6 | 64 | |
| 14 | 1110 | +7 | 128 | |
| 15 | 1111 | n/a | | (inf, NaN) |

# 8비트 시스템 – 표현 가능 수(양수)

| | s | exp | frac | E | Value | |
|---|---|---|---|---|---|---|
| 비정규화 수 | 0 | 0000 | 000 | -6 | 0 | |
| | 0 | 0000 | 001 | -6 | 1/8*1/64 = 1/512 | ← 0 에 가장 가까운 값 |
| | 0 | 0000 | 010 | -6 | 2/8*1/64 = 2/512 | |
| | … | | | | | |
| | 0 | 0000 | 110 | -6 | 6/8*1/64 = 6/512 | |
| | 0 | 0000 | 111 | -6 | 7/8*1/64 = 7/512 | ← 최대 비정규화 수 |
| 정규화 수 | 0 | 0001 | 000 | -6 | 8/8*1/64 = 8/512 | ← 최소 정규화 수 |
| | 0 | 0001 | 001 | -6 | 9/8*1/64 = 9/512 | |
| | … | | | | | |
| | 0 | 0110 | 110 | -1 | 14/8*1/2 = 14/16 | |
| | 0 | 0110 | 111 | -1 | 15/8*1/2 = 15/16 | ← 1 에 가장근접한 수 |
| | 0 | 0111 | 000 | 0 | 8/8*1 = 1 | |
| | 0 | 0111 | 001 | 0 | 9/8*1 = 9/8 | ← 1에 가장근접한 수 |
| | 0 | 0111 | 010 | 0 | 10/8*1 = 10/8 | |
| | … | | | | | |
| | 0 | 1110 | 110 | 7 | 14/8*128 = 224 | |
| | 0 | 1110 | 111 | 7 | 15/8*128 = 240 | ← 최대 정규화 수 |
| | 0 | 1111 | 000 | n/a | inf | |

# C 에서의 Floating Point 숫자

| 설명 | exp | frac | Numeric Value |
|---|---|---|---|
| 0 | 00…00 | 00…00 | 0.0 |
| 최소 비정규화 수(+) | 00…00 | 00…01 | $2^{-\{23,52\}} \times 2^{-\{126,1022\}}$ |
| • Single ≈ $1.4 \times 10^{-45}$ | | | |
| • Double ≈ $4.9 \times 10^{-324}$ | | | |
| 최대 비정규화 수 | 00…00 | 11…11 | $(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$ |
| • Single ≈ $1.18 \times 10^{-38}$ | | | |
| • Double ≈ $2.2 \times 10^{-308}$ | | | |
| 최소 정규화 수(+) | 00…01 | 00…00 | $1.0 \times 2^{-\{126,1022\}}$ |
| • 최대 비정규화 수보다 아주 조금 크다 | | | |
| 1 | 01…11 | 00…00 | 1.0 |
| 최대 정규화 수 | 11…10 | 11…11 | $(2.0 - \varepsilon) \times 2^{\{127,1023\}}$ |
| • Single ≈ $3.4 \times 10^{38}$ | | | |
| • Double ≈ $1.8 \times 10^{308}$ | | | |

# 인코딩의 특징

## FP에서의 0 은 정수에서의 0 과 동일하다

- 모든 비트들 = 0

## 비부호 정수 비교를 사용할 수 있다(거의)

- 먼저 부호비트를 비교해야 한다
- -0 = 0 인 점을 고려해야 한다
- NaN의 경우는 문제가 생긴다
  - 다른 모든 값들보다는 클 것인가
  - 비교의 결과는 무엇인가?
- 그렇지 않은 다른 경우는 OK
  - 비정규화 vs. 정규화
  - 정규화 vs. 무한대

# Floating Point 근사(Rounding)

개념적인 설명

- 먼저 정확한 값을 계산한다
- 그 결과를 원하는 정밀도로 조정한다
  - 만일 지수부가 너무 크다면 오버플로우일 수 있다
  - frac 에 맞출 수 있도록 반올림 한다

근사 방법

| | $1.40 | $1.60 | $1.50 | $2.50 | –$1.50 |
|---|---|---|---|---|---|
| 0 방향 | $1 | $1 | $1 | $2 | –$1 |
| 내림(-∞) | $1 | $1 | $1 | $2 | –$2 |
| 올림 (+∞) | $2 | $2 | $2 | $3 | –$1 |
| 인접짝수 (default) | $1 | $2 | $2 | $2 | –$2 |

Q. **웬 짝수** ?

Note:

1. 버림: 근사값은 참값보다 클 수 없다.
2. 올림: 근사값은 참값보다 작을 수 없다.

# 인접 짝수 모드(Round-To-Even)

## 기본(Default) 근사모드

- 어셈블리 언어 수준까지 이해하지 않고는 설명할 수 없다
- 다른 근사 모드들은 모두 통계적으로 편향되어 있다
  - 여러개의 근사한 양수들의 합은 과대평가 하게 된다

## 십진수의 다른 자리에 근사법의 적용

- 두 개의 가능한 값의 정 중간 값인 경우
  - 최소 자리값이 짝수가 되도록 근사
- E.g., 0.01의 자리로 근사하는 경우

| | | |
|---|---|---|
| 1.2349999 | 1.23 | (정 중간 보다 작다) |
| 1.2350001 | 1.24 | (정 중간보다 크다) |
| 1.2350000 | 1.24 | (정 중간 값 – 올림) |
| 1.2450000 | 1.24 | (정 중간 값— 내림) |

# 이진수에서의 근사법

이진 소소의 표현

- "짝수" 는 최소 유효 값이 0 인 경우를 말함
- 근사하는 자리 바로 오른쪽의 비트가 = $100\cdots_2$

## Examples

- **1/4** 자리로 근사하는 경우(**2** 진 소숫점이하 **2**번째 자리로)

| 값 | 이진수 | 근사값 | 연산 | 근사값(10진수) |
|---|---|---|---|---|
| **2 3/32** | $10.00011_2$ | $10.00_2$ | (<1/2—down) | 2 |
| **2 3/16** | $10.00110_2$ | $10.01_2$ | (>1/2—up) | 2 1/4 |
| **2 7/8** | $10.11100_2$ | $11.00_2$ | (1/2—up) | 3 |
| **2 5/8** | $10.10100_2$ | $10.10_2$ | (1/2—down) | 2 1/2 |

# FP 곱셈

피연산자

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

정확한 결과

$(-1)^{s}\ M\ 2^{E}$

- **Sign *s*: *s1* ^ *s2***
- 소수 ***M*** **:** ***M1*** ***** ***M2***
- 지수 ***E*****:** ***E1*** **+** ***E2***

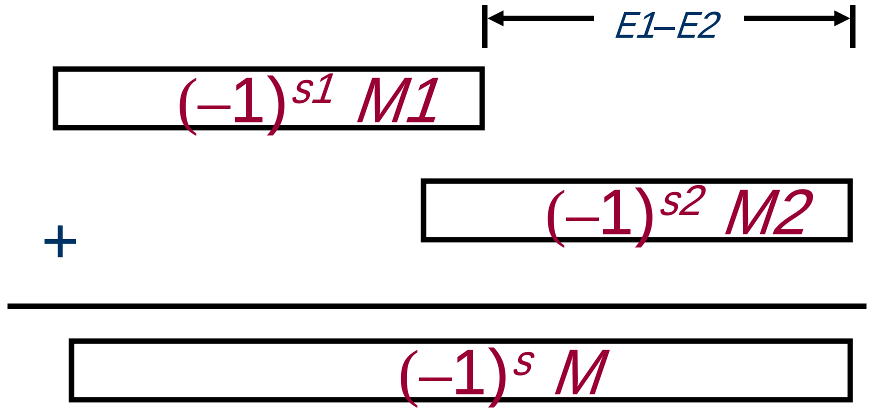
정정

- 만일 ***M*** ≥ **2,** ***M*** 을 우측으로 **shift,** ***E*** 를 **1** 증가
- 만일 ***E*** 값이 범위를 벗어나면, 오버플로우
- ***M*** 을 frac 정밀도에 맞추어 근사

구현

- 가장 중요한 부분은 소수들 간의 곱셈

# FP 덧셈




- 피연산자
  - $(-1)^{s1}\ M1\ \ 2^{E1}$
  - $(-1)^{s2}\ M2\ \ 2^{E2}$
  - *E1* > *E2* 을 가정
- 정확한 결과
  - $(-1)^{s}\ M\ \ 2^{E}$
  - 부호 *s*, 소수 *M*:
    - 열 맞춤후의 덧셈의 결과
  - 지수 *E*: *E1*
- 정정
  - 만일 $M \geq 2$, *M* 을 우측으로 shift, *E* 를 1 증가
  - 만일 $M < 1$, *M* 을 좌측으로 *k* 자리 shift, *E* 를 *k* 만큼 감소
  - 만일 *E* 가 범위를 벗어나면 오버플로우
  - *M* 을 근사시켜서 frac 정밀도에 맞춤

# Floating point 수 만들기

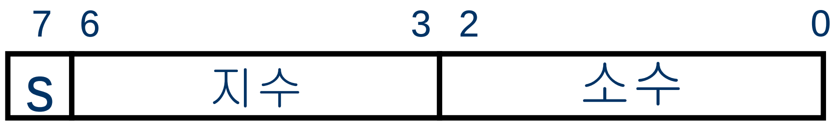
## 과정

- 1 이 맨 앞에 올 수 있도록 정규화(Normalize)
- 소수점 자리(fraction)에 맞도록 반올림
- 반올림 효과를 반영하기 위한 정규화 후처리(Postnormalize)

## 사례

- 8-bit 비부호형 수를 작은 floating point format 으로 표시해보자
- Example Numbers

| | |
|---|---|
| 128 | 10000000 |
| 15 | 00001101 |
| 33 | 00010001 |
| 35 | 00010011 |
| 138 | 10001010 |
| 63 | 00111111 |



# 정규화

| 7 | 6 … 3 | 2 … 0 |
|---|---|---|
| s | exp | frac |

요구사항

- 이진 소수의 위치를 이동해서 **1.xxxxx** 의 형태가 되도록 한다
- 자리수 만큼 **0** 을 채운다
  - 지수값을 소수점 위치에 맞도록 조정

| Value | Binary | 소수 | 지수 |
|---|---|---|---|
| 128 | 10000000 | 1.0000000 | 7 |
| 15 | 00001101 | 1.1010000 | 3 |
| 17 | 00010001 | 1.0001000 | 5 |
| 19 | 00010011 | 1.0011000 | 5 |
| 138 | 10001010 | 1.0001010 | 7 |
| 63 | 00111111 | 1.1111100 | 5 |

# 근사

1.BBGRXXX

Guard bit: 결과값의 LSB

Round bit: 없어지는 첫번째 비트

Sticky bit: 기타 나머지 비트들

반올림 조건

- Round = 1, Sticky = 1 ➔ > 0.5
- Guard = 1, Round = 1, Sticky = 0 ➔ 인접짝수

| Value | 소수 | GRS | 올림? | 근사값 |
|---|---|---|---|---|
| 128 | 1.0000000 | 000 | N | 1.000 |
| 15 | 1.1010000 | 100 | N | 1.101 |
| 17 | 1.0001000 | 010 | N | 1.000 |
| 19 | 1.0011000 | 110 | Y | 1.010 |
| 138 | 1.0001010 | 011 | Y | 1.001 |
| 63 | 1.1111100 | 111 | Y | 10.000 |

# 정규화후처리

## 이슈

- 근사화 결과 오버플로우가 발생했을지 모른다
- 우측으로 한 자리 shift 하고, 지수값을 1 증가시킨다

| Value | 근사값 | 지수 | 조정후 | 결과 |
|---|---|---|---|---|
| 128 | 1.000 | 7 | | 128 |
| 15 | 1.101 | 3 | | 15 |
| 17 | 1.000 | 4 | | 16 |
| 19 | 1.010 | 4 | | 20 |
| 138 | 1.001 | 7 | | 134 |
| 63 | 10.000 | 5 | 1.000/6 | 64 |

# C 언어에서의 Floating Point

C 에서는 2개의 정밀도를 제공

- float **single precision**
- double **double precision**

변환

- int, float, double 간의 **casting** 은 값을 변화시킨다
- Double, float 에서 int 로의 변환
  - 소수 부분을 제거
  - **0** 방향으로의 근사와 동일
  - 무한대 또는 **NaN** 의 경우에는 정의 되어 있지 않음
    - 일반적으로 **TMin**
- int 에서 double 로
  - 정확한 변환, 정수가 **53**비트 보다 짧은 경우에만
- int 에서 float 로
  - 근사 모드에 따라 근사화

# Floating Point Puzzles

- 다음의 C 수식에서 :
  - 다음의 수식이 항상 참인지 판별하라
  - 참이 아니라면, 그 이유를 설명하라

```
int x = …;
float f = …;
double d = …;
```

d , f 는 NaN 가 아님

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` ⇒ `((d*2) < 0.0)`
- `d > f` ⇒ `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

# Ariane 5

- 이륙후 37초 후 폭파
- 위성체 손실규모 5천억원

## 이유

- 수평 속도를 floating point 형으로 계산
- 16-bit 정수형으로 변환
- Ariane 4 로켓에서는 정상동작
- Ariane 5 로켓에서는 오버플로우
  - 동일한 소프트웨어를 사용해서

# Summary

## IEEE Floating Point 표준은 명쾌한 수학특성을 갖는다

- $M$ X $2^E$ 형태의 수 표현
- 실제 구현 방법과 관계없이 연산을 적용할 수 있다
  - 마치 완벽한 정밀도로 계산한 후에 근사화 한 것 처럼
- 실제 연산과 일치하는 것은 아니다
  - 교환/배분 법칙에 위배되는 경우가 있다
  - 컴파일러 개발자나 고급 프로그래머들에게는 큰 위협

## 다음주 예습 숙제 : pp.193-199, 3.2-3.2.2