Fixed point

박정식

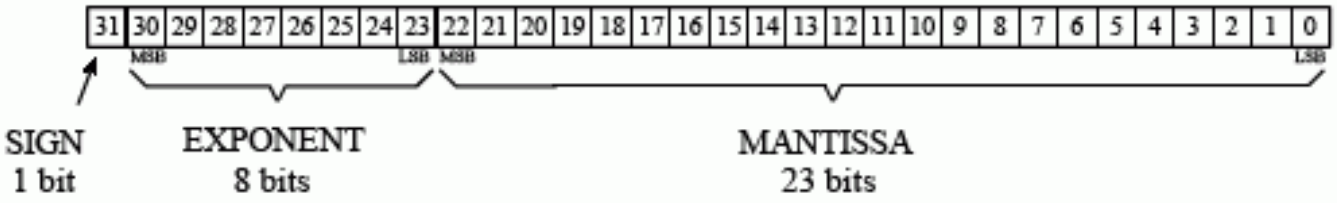
nangsik@mr.hanyang.ac.kr

C언어의 실수 연산

- C언어에서는 실수 표현 및 연산을 위해 floating point를 사용
  - float, double
  - float, double type을 위한 수학함수 제공

Floating point

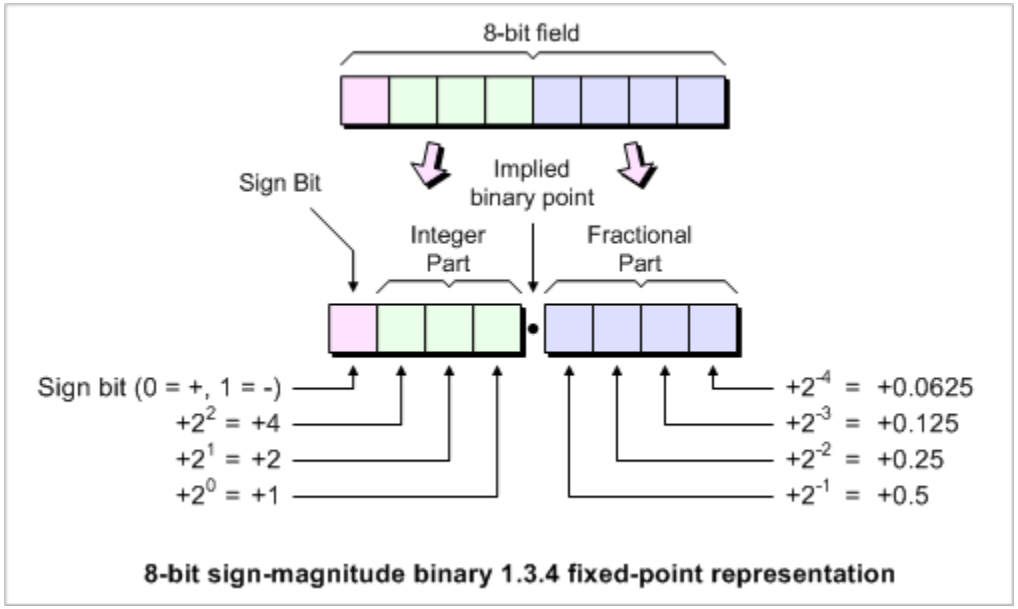
- 부호비트, 지수부, 가수부로 구성



- 특징
  - 표현 가능한 수의 범위가 큼
  - 정수 연산에 비해 연산량이 많음
  - 빠른 연산 속도를 위해 FPU 필요

Fixed point

- Bit열을 부호비트, 정수부, 소수부로 나눔



8-bit sign-magnitude binary 1.3.4 fixed-point representation

- 특징
  - Integer 연산으로 구현 가능하므로 하드웨어 cost가 낮음

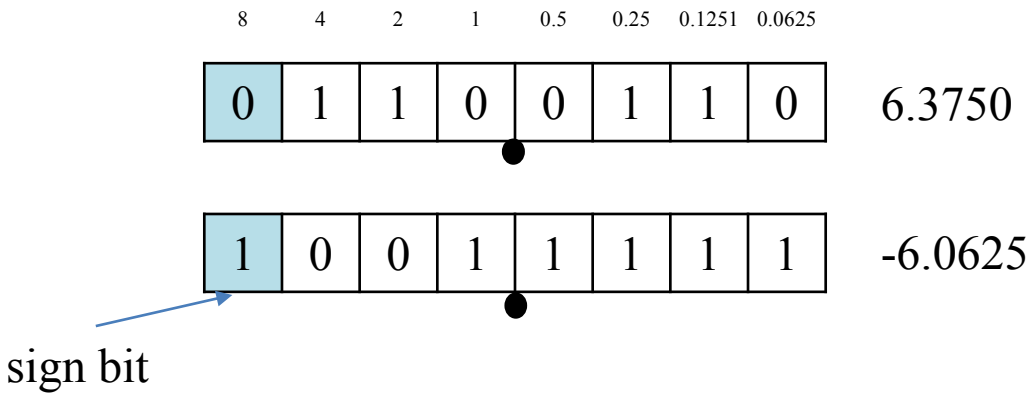
Fixed point가 사용되는 곳

- FPU가 없는 경우
  - 시스템
    - 임베디드 시스템
      - 마이크로 프로세서
      - 모바일 GPU
  - 알고리즘
    - 신호처리
      - 영상, 음성
    - 그래픽스

FIXED POINT C언어 구현

데이터 구조

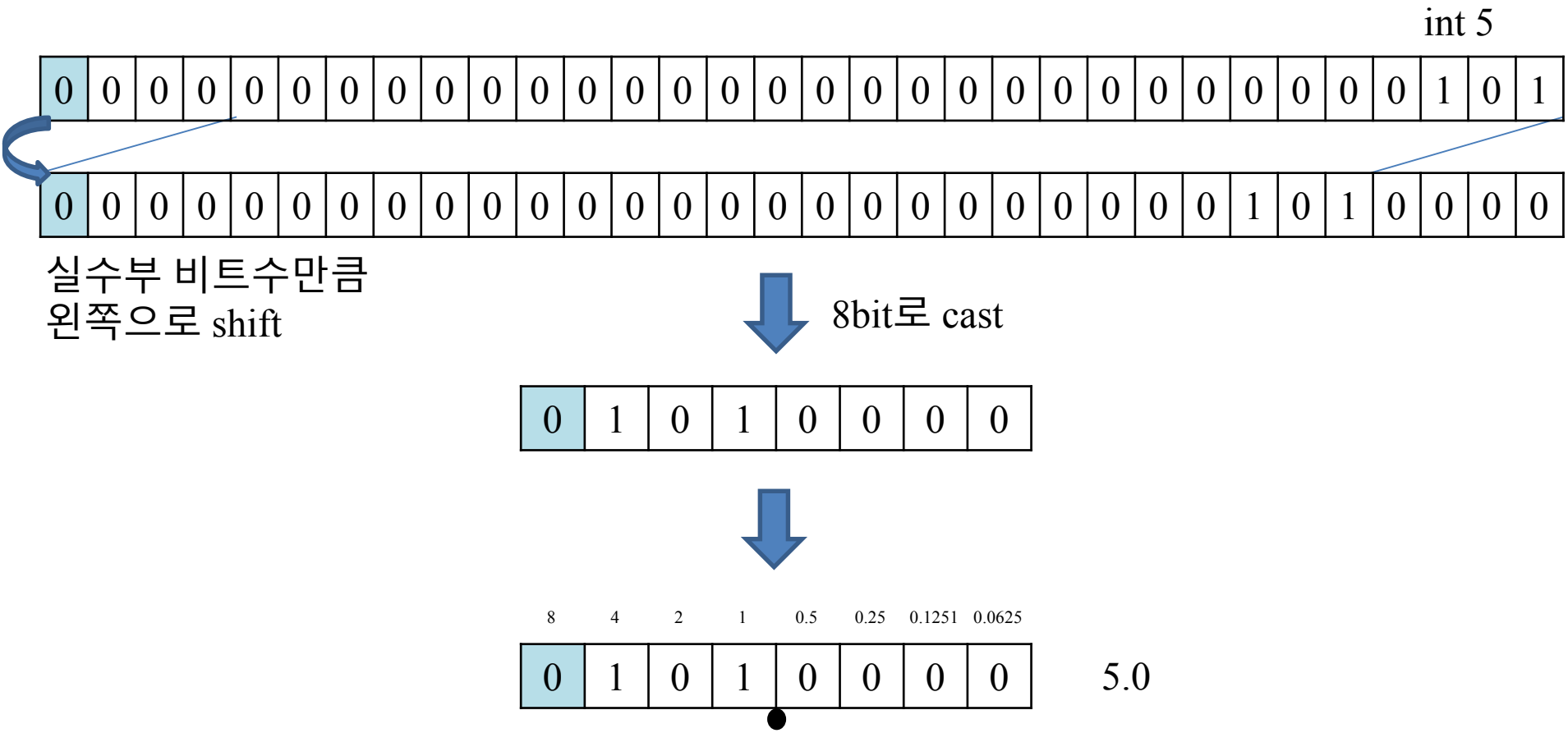
- 예시
  - 8bit, signed
  - 4bit를 실수부에 할당



  - 값의 범위: 0b10000000 ~ 0b01111111 (-8 ~ 7.9375)

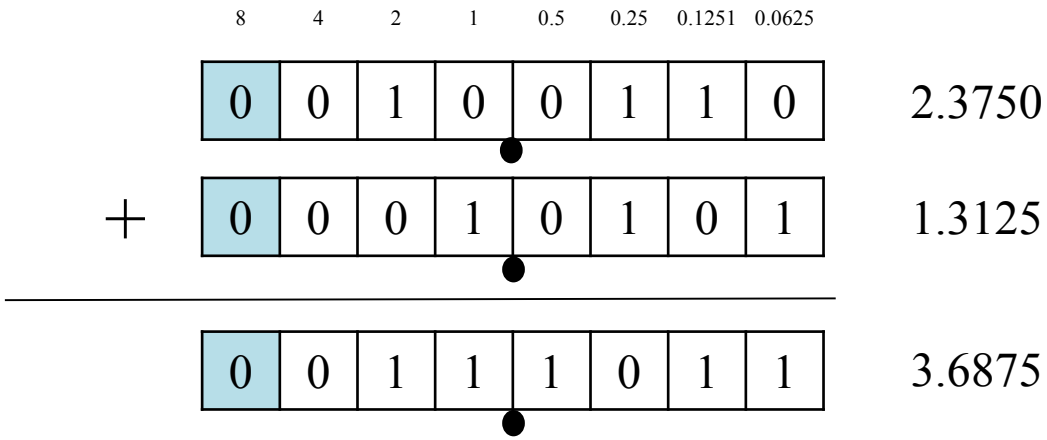
연산 과정

- 정수로부터 변환 (예시: 8bit)



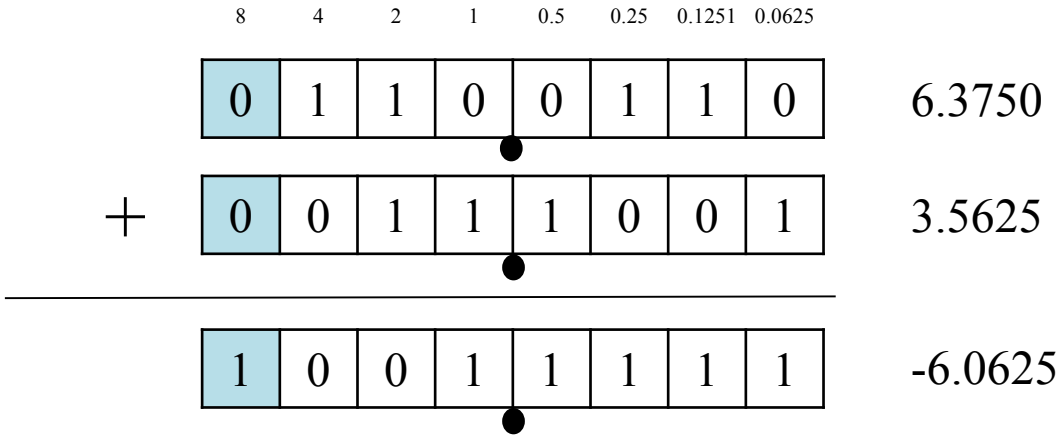
연산 과정

- 덧셈 (예시: 8bit)



연산 과정

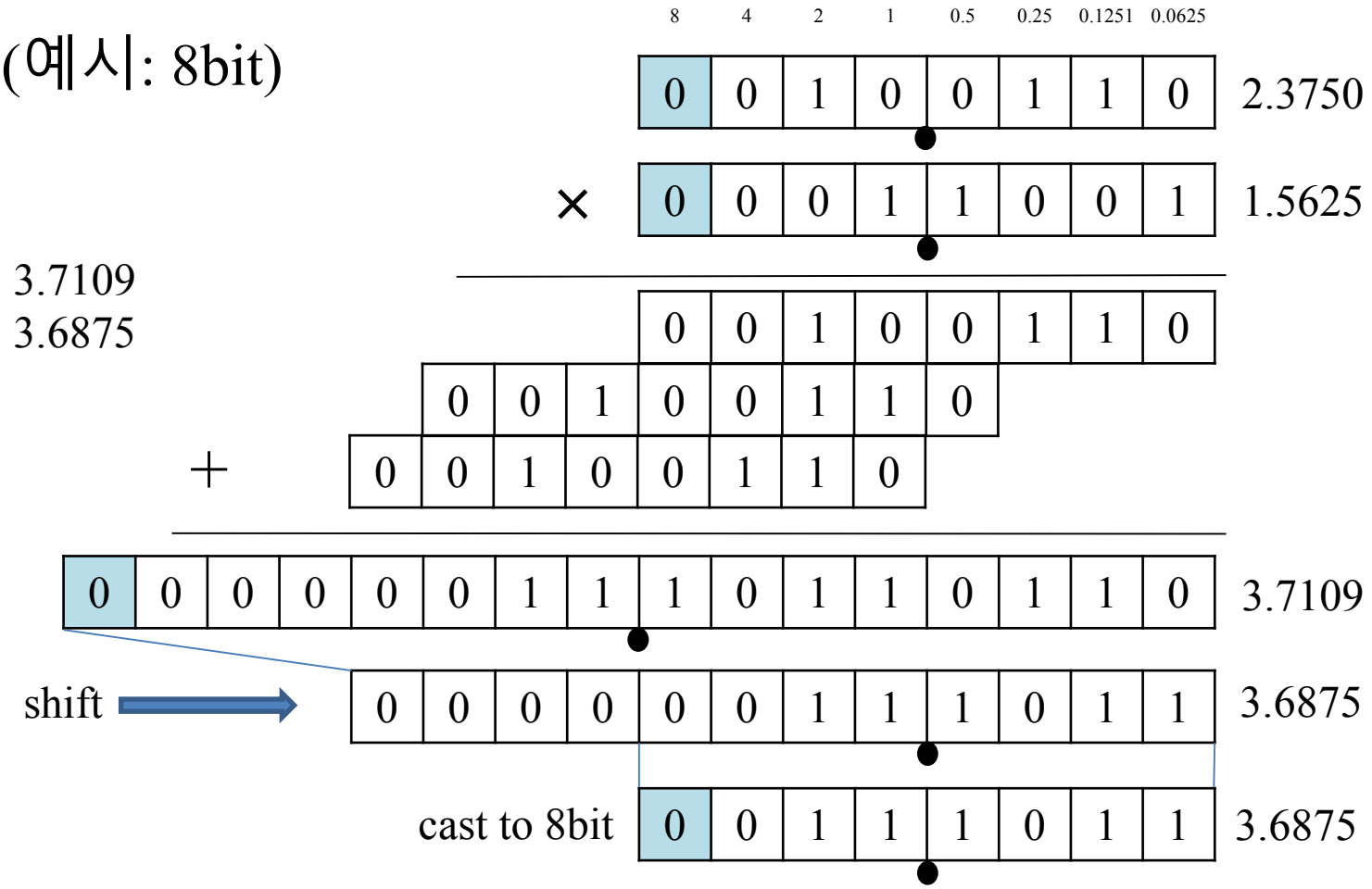
- 덧셈 (예시: 8bit)
  - overflow 발생



연산 과정

- 곱셈 (예시: 8bit)

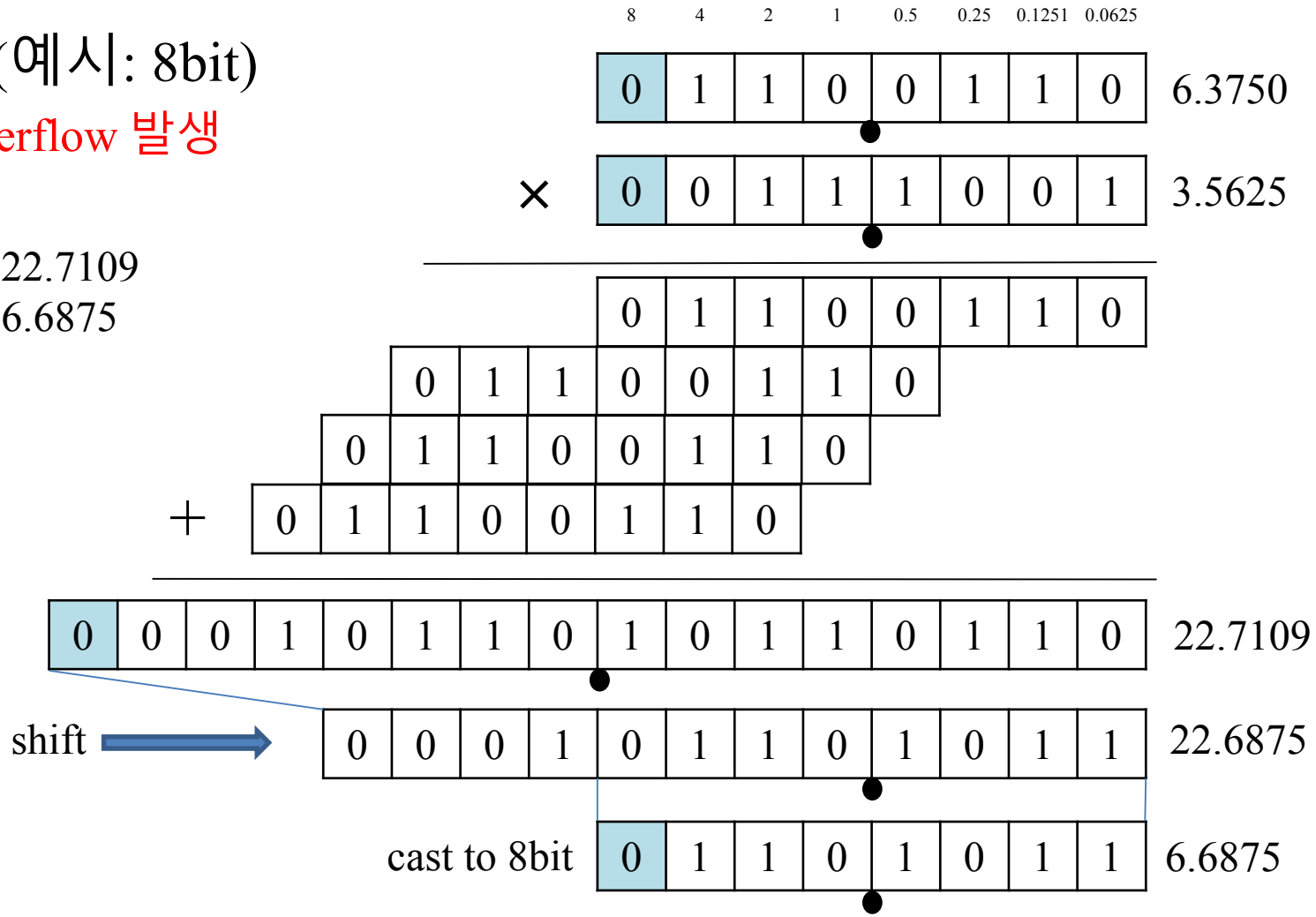
참값: 3.7109
연산 결과: 3.6875



연산 과정

- 곱셈 (예시: 8bit)
  - Overflow 발생

참값: 22.7109
연산 결과: 6.6875



데이터 구조

- 소수점 위치를 특정 위치로 고정시키는 경우
  - int 형을 typedef
  - 소수점 위치는 define 매크로로 지정

- 소수점 위치를 원하는 위치에 두고 사용하는 경우
  - 구조체로 선언
    - 예

```
struct FixedPoint
{
        int fractionBits;
        int value;
}
```

구현 예

```
typedef int FP;
#define FP_FRAC_BITS 23
static inline FP FP_add(FP A, FP B)
{
return A + B;
}
static inline FP FP_mul(FP A, FP B)
{
return (((long long int)A * (long long int)B) >>
FP_FRAC_BITS);
}
```

- 사용이 불편함
  - Built-in type처럼 사용불가(별도 함수 필요)

FIXED POINT C++ 구현

C++ 구현의 장점

- C 구현에 비해 직관적이고 편하게 사용 가능
  - Built-in type처럼 사용 가능
    - 연산자 오버로딩
  - 소수점 위치를 원하는 위치로 지정 가능하며, 소수점 위치가 서로 다른 두 숫자의 연산이 가능
    - template class 활용

데이터 구조

```
template<unsigned PREC>
class FP
{
public:
        typedef int T;
        template<unsigned PP>
        friend class FP;
private:
        T v;
};
```

- PREC: fraction 비트의 수
- 소수점 위치가 서로 다른 경우에도 연산이 가능하도록 friend class로 선언

생성자

- 생성자(Built-in type으로부터 변환)

```
FP():v(0){}
FP(const FP& x):v(x.v){}
FP(int x):v(x*(1<<PREC)){}
FP(float x):v(x*(1<<PREC)){}
FP(double x):v(x*(1<<PREC)){}
```

값 손실 발생 가능

- Built-in type으로 변환

```
operator int () const {return (int)(v>>PREC);}
operator float () const {return ((float)v)/(1<<PREC);}
operator double () const {return ((double)v)/(1<<PREC);}
template<unsigned PP> operator FP<PP> () const {
        if (PP>PREC) {return (v>>PP-PREC);}
        else {return (v<<PREC-PP);}
}
```

할당 연산자

```
FP& operator=(const int& x){
        v = (T)(x*(1<<PREC));
        return *this;}
FP& operator=(const float& x){
        v = (T)(x*(1<<PREC));
        return *this;}
FP& operator=(const double& x){
        v = (T)(x*(1<<PREC));
        return *this;}
FP& operator=(const FP& x){
        v = x.v;
        return *this;}
template<unsigned PP>
FP& operator=(const FP<PP>& x){
        if (PP>PREC) {v = (x.v >> (PP-PREC));}
        else {v = (x.v << (PREC-PP));}
        return *this;}
```

값 손실 발생 가능

덧셈 연산자

```
FP& operator+=(const FP& x){
        v += x.v;
        return *this;}
FP operator+(const FP& x) const{
        FP res(*this);
        return res += x;}
template<typename TT>
FP& operator+=(const TT& x){
        this+=FP(x);
        return (*this);}
template<typename TT>
FP operator+(const TT& x){
        FP res(*this);
        return res+=FP(x);}
```

- 뺄셈 연산은 덧셈 연산으로부터 응용

곱셈/나눗셈 연산자

- 곱셈

```
FP& operator*=(const FP& x) {
        typedef long long int T2;
        v = (T)(((T2)v*(T2)x.v)>>PREC);
        return *this;}
FP operator*(const FP& x) const {
        FP res(*this);
        return res*=x;}
template<typename TT>
FP& operator*=(const TT& x){
        *this*=FP(x);
        return (*this);}
template<typename TT>
FP operator*(const TT& x){
        FP res(*this);
        return res*=FP(x);}
```

- 나눗셈

```
FP& operator/=(const FP& x) {
        typedef long long int T2;
        v =
(T)((((T2)v)<<PREC)/((T2)(x.v)));
        return *this;}
FP operator/(const FP& x) const {
        FP res(*this);
        return res/=x;}
template<typename TT>
FP& operator/=(const TT& x){
        *this/=FP(x);
        return (*this);}
template<typename TT>
FP operator/(const TT& x){
        FP res(*this);
        return res/=FP(x);}
```

스트림 연산자

- 값의 출력에 사용

```
template<typename OUT>
friend OUT& operator<<(OUT& out,const FP& f){
                        return f.print(out);}
private:
template<typename OUT>
OUT& print(OUT& out) const{
            return ( out << ((double)v)/(1<<PREC)
;}
```

Test code #1

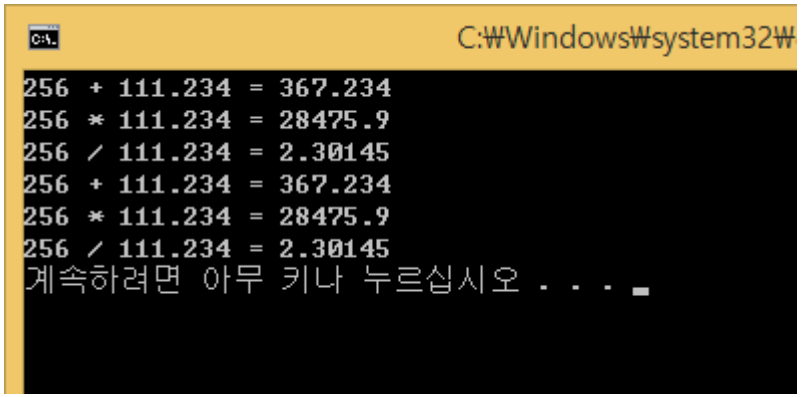
소수부로 16비트 사용시

```
FP<16> a = 256;
FP<16> b = 111.234;

std::cout << a << " + " << b << " = " << a+b << std::endl;
std::cout << a << " * " << b << " = " << a*b << std::endl;
std::cout << a << " / " << b << " = " << a/b << std::endl;

float c = a;
float d = b;

std::cout << c << " + " << d << " = " << c+d << std::endl;
std::cout << c << " * " << d << " = " << c*d << std::endl;
std::cout << c << " / " << d << " = " << c/d << std::endl;
```



Test code #1

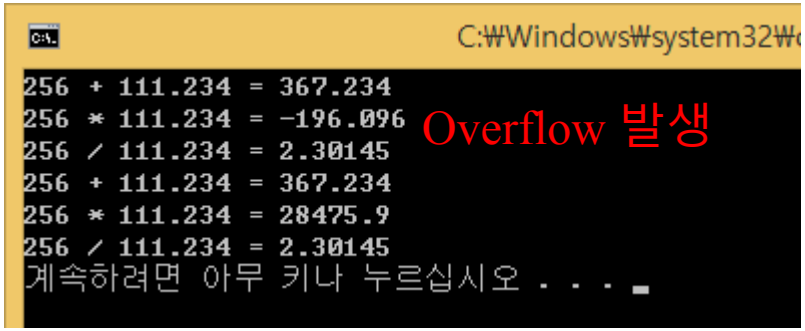
소수부로 20비트 사용시

```
FP<20> a = 256;
FP<20> b = 111.234;

std::cout << a << " + " << b << " = " << a+b << std::endl;
std::cout << a << " * " << b << " = " << a*b << std::endl;
std::cout << a << " / " << b << " = " << a/b << std::endl;

float c = a;
float d = b;

std::cout << c << " + " << d << " = " << c+d << std::endl;
std::cout << c << " * " << d << " = " << c*d << std::endl;
std::cout << c << " / " << d << " = " << c/d << std::endl;
```



Fixed point 사용시 주의점

- Overflow 발생
  - 다루는 데이터의 값의 범위와 연산 결과의 값의 범위를 고려
  - 자료형 크기를 증가
    - 예를 들면 long long int 혹은 int 두개를 사용
    - 단, 추가적인 연산으로 인한 오버헤드 발생

- 예외처리
  - NaN, ±∞ 에 대한 처리

Math function

- C/C++는 math.h, cmath에서 float point에 대한 여러 함수를 제공
  - abs, floor, ceil, 삼각함수, exp, pow, sqrt 등
  - 수치해석으로 구현 가능, 하드웨어 설계로 고속 연산

- Fixed point에도 이러한 함수의 제공이 필요
  - 소프트웨어적으로 계산 → 수치해석

삼각함수

- 삼각함수를 구하는 방법은 여러 가지가 있을 수 있음
  - 가장 간단한 방법:
    - Taylor series

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}.$$

    - Look-up table
    - 두 방법을 혼용
  - Floating point의 경우는 속도와 정확도를 위해 cordic algorithm 등의 방법을 이용

Taylor series를 이용한 sine 함수

- sine 함수

```
friend FP sin(FP x)
{
        FP xx = x * x;
        FP sx = -xx + (FP)(6*7);
        sx *= xx;
        sx -= ((FP)(4*5*6*7));
        sx *= xx;
        sx += ((FP)(2*3*4*5*6*7));
        sx *= x;
        sx /= ((FP)(2*3*4*5*6*7));
        return sx;
}
```

- 테스트 코드

```
float a = 0.5;
FP<16> b = a;
float c = sin(a);
FP<16> d = sin(b);
std::cout << "float sin(" << a << ") = " << c << std::endl;
std::cout << "fixed sin(" << b << ") = " << d << std::endl;
std::cout << "error = " << (float)d - c << std::endl;
```

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} = x \cdot \left(1 - x^2 \cdot \left(\frac{1}{3!} + x^2 \cdot \left(\frac{1}{5!} - \frac{x^2}{7!}\right)\right)\right) = \left(\left(\left(-\frac{x^2}{7!} + \frac{1}{5!}\right) \cdot x^2 - \frac{1}{3!}\right) \cdot x^2 + 1\right) \cdot x$$

$$= \left(\left(\left(-x^2 + \frac{7!}{5!}\right) \cdot x^2 - \frac{7!}{3!}\right) \cdot x^2 + 7!\right) \cdot \frac{x}{7!}$$

Taylor series를 이용한 sine 함수

- 숫자가 커질수록 에러가 증가하는 경향
  - 해결책
    - Taylor series의 차수를 증가
    - 주기성과 대칭성을 이용 (sin, cos)
    - 구간별로 Taylor series와 look-up table을 혼용

```
float sin(0.5) = 0.479426
fixed sin(0.5) = 0.479416
error = -9.65595e-006

float sin(1) = 0.841471
fixed sin(1) = 0.841461
error = -9.77516e-006

float sin(1.5) = 0.997495
fixed sin(1.5) = 0.997391
error = -0.000104249

float sin(2) = 0.909297
fixed sin(2) = 0.907928
error = -0.00136894

float sin(2.5) = 0.598472
fixed sin(2.5) = 0.588531
error = -0.00994062

float sin(2.75) = 0.381661
fixed sin(2.75) = 0.35849
error = -0.023171

float sin(3) = 0.14112
fixed sin(3) = 0.0910645
error = -0.0500555

float sin(3.14) = 0.00159255
fixed sin(3.14) = -0.073288
error = -0.0748805
```

Reference

- Oliver Schloesser, “Implementing a C++ Fixed-Point Class for Embedded Systems,” IME internal paper of master thesis, University of Applied Sciences, NW Switzerland, 2013.