Report: Small Floating Point

2016314786 수학과 김호진

이번 과제에서 구현해야 할 SFP는 1-bit sign(s), 5-bit exponent(exp), 10-bit significant (frac)으로 이루어져 있습니다. 그렇기 때문에 다음과 같이 비트 필드(bit-field)와 공용체(union)를 함께 사용하여 SFP를 나타내는 구조체를 만들어 활용했습니다.

```
struct sfpbits {
    union {
        struct {
            sfp F : 10;  // F(Fraction bits)에 해당하는 10비트 크기 할당
            sfp E : 5;  // E(Exponential bits)에 해당하는 5비트 크기 할당
            sfp S : 1;  // S(Sign bit)에 해당하는 1비트 크기
        }; // 합계 16비트
        sfp SEF;
    };
};
```

sfp int2sfp(int input), int sfp2int(sfp input), sfp float2sfp(float input), float sfp2float(sfp input), sfp sfp_add(sfp in1, sfp in2), sfp_mul(sfp in1, sfp in2), char* sfp2bits(sfp result) 함수 설명을 하기에 앞서 추가적으로 구현한 함수 sfp return_E(int input)와 sfp return_exp(int input)에 대해 간단하게 설명하겠습니다.

sfp return_E(int input)는 SFP 부동소수점에서 E를 구하기 위한 함수입니다.

```
sfp return_E(int input) {
    sfp E = 0;

    while (input > 1) {
        input = input / 2;
        E++;
    }

    return E;
}
```

SFP에서 Bias = $2^{5-1}-1$ = 15이므로 normalized number의 경우, Exp = E + Bias = E + 15입니다. sfp return_exp(int input)는 이를 이용해 normalized number의 Exp를 구해주는 함수입니다.

```
sfp return_exp(int input) {
    sfp E;
    E = return_E(input);

    return 15 + E;
}
```

[sfp int2sfp(int input)]

sfp int2sfp(int input)는 정수형을 SFP로 변환시키는 함수입니다.

0 11110 1111111111 → $(-1)^0 * (2047 / 1024) * 2^{15} = 65504$ (largest normalized number)

1 11110 1111111111 → $(-1)^1 * (2047 / 1024) * 2^{15} = -65504$ (smallest normalized number)

위 과정을 통해 SFP의 범위가 -65504 ~ 65504임을 확인했습니다. Input의 값이 이러한 SFP의 범위를 넘어서게 된다면 Overflow가 발생하게 되므로 input이 65504보다 클 때 +∞를, -65504보다 작을 때 -∞를 나타내는 SFP를 반환하도록 하였습니다.

```
if (input > 65504) {
    k.F = 0;    // Fraction bits = 0000000000
    k.E = 31;   // Exponential bits = 11111
    k.S = 0;    // Sign bit = 0
}

else if (input < -65504) {
    k.F = 0;    // Fraction bits = 0000000000
    k.E = 31;   // Exponential bits = 11111
    k.S = 1;    // Sign bit = 1
}
```

정수형에서 Denormalized number는 0만 존재하기 때문에 0의 경우, +0.0을 나타내는 SFP를 반환하도록 다음과 같이 구현하였습니다.

```
else if (input == 0) {
    k.F = 0;
    k.E = 0;
    k.S = 0;
}
```

이 밖의 경우는 모두 Normalized numbers이기 때문에 다음과 같이 구현이 가능합니다.

```
    else {
        if (input > 0) {
            k.F = input;
            for (sfp i = 0; i < 10 - return_E(input); i++) {  //  Fraction part의 나머지 부분에 0을 할당하는 과정
                k.F *= 2;
            }
            k.E = return_exp(input);
            k.S = 0;
        }
        else {
            k.F = -input;
            for (sfp i = 0; i < 10 - return_E(-input); i++) {  //  Fraction part의 나머지 부분에 0을 할당하는 과정
                k.F *= 2;
            }
            k.E = return_exp(-input);
            k.S = 1;
        }
    }

    return k.SEF;
}
```

[int sfp2int(sfp input)]

int sfp2int(sfp input)은 SFP를 정수형으로 변환시키는 함수입니다.

문제 조건에 의해서 우리가 우선적으로 신경 써줘야 하는 것은 Input이 +∞, -∞ 그리고 NaN일 경우입니다. +∞는 int의 최대범위 값을, -∞와 NaN은 int을 최소범위 값을 return 해줘야 하므로 #include <limits.h>을 추가하여 각각의 경우에서 INT_MAX와 INT_MIN을 return하도록 했습니다.

```
if (k.E == 31) {
    if (k.F == 0) {
        if (k.S == 0) {  // 0 11111 0000000000
            output = INT_MAX;  // +∞ -> int 최대범위값
        }
        else {  // 1 11111 0000000000
            output = INT_MIN; // -∞ -> int 최소범위값
        }
    }
    else {
        output = INT_MIN; // NaN -> int 최소범위값
    }
}
```

exponent part가 00000의 경우에는 round-toward-zero에 의해 0이 return됩니다.

```
else if (k.E == 0) {
    output = 0;
}
```

그 이외의 경우, 모두 Normalized Values이므로 $v=(-1)^S M*2^E$를 이용해여 return 하였습니다. 이때 모든 Normalize Values의 SFP에는 숨겨진 bit 1이 fraction part 앞에 위치함을 주의해야 하며 따라서 다음과 같이 구현하였습니다.

```
else {
    float frac = (float)(1024 + k.F) / 1024;  // Normalized value의 fraction bit 내 hidden bit 고려
    int exp = k.E - 15; // E = Exp - Bias(15)

    if (exp > 0) {
        for (int i = 0; i < exp; i++) {
            frac *= 2;
        }
    }
    else if (exp < 0) {
        for (int i = 0; i < -exp; i++) {
            frac /= 2;
        }
    }

    if (k.S == 0) {
        output = (int)(frac);
    }
    else {
        output = (int)(-frac);
    }
}
```

[sfp float2sfp(float input)]

sfp float2sfp(float input)은 실수형을 SFP로 변환시키는 함수입니다.

앞서 SFP의 범위가 -65504 ~ 65504을 확인했습니다. 마찬가지로 Input의 값이 이러한 SFP의 범위를 넘어서게 된다면 Overflow가 발생하게 되므로 input이 65504보다 클 때 +∞를, -65504보다 작을 때 -∞를 나타내는 SFP를 반환하도록 하였습니다.

```c
if ( input > 65504 ) {
    k.F = 0;
    k.E = 31;
    k.S = 0;
}

else if ( input < -65504 ) {
    k.F = 0;
    k.E = 31;
    k.S = 1;
}
```

정수형과 다르게 실수형에서는 0 이외에 Denormalized number인 경우가 존재합니다. 0 이상의 가장 작은 Normalize number는 0 00001 0000000000으로, 2^{-14} = 0.000061..이므로 round-to-zero를 고려하여 input이 -0.000061보다 크고 0.000061보다 작은 경우 denormalized number가 됨을 알 수 있습니다. Denormalized number의 경우, E = 1 – bias = -14인 점을 고려하여 구현해주면 다음과 같습니다.

```c
else if ((-0.000061 < input) && (input < 0)) {
    k.E = 0;
    k.S = 1;
    float Real = -input;

    Real *= -16384;  //  E = 1 - Bias = -14

    for (sfp i = 0; i < 10; i++) {  //  소수부분 이진변환
        if (Real * 2 >= 1) {
            int k_Real = 1;
            for (sfp j = 0; j < 9 - i; j++) {
                k_Real *= 2;
            }
            k.F += k_Real;
            Real = 2 * Real - 1;
        }
        else {
            Real *= 2;
        }
    }
}

else if ((0 <= input) && (input < 0.000061)) {
    k.E = 0;
    k.S = 0;
    float Real = input;

    Real *= 16384;  //  E = 1 - Bias = -14

    for (sfp i = 0; i < 10; i++) {  //  소수부분 이진변환
        if (Real * 2 >= 1) {
            int k_Real = 1;
            for (sfp j = 0; j < 9 - i; j++) {
                k_Real *= 2;
            }
            k.F += k_Real;
            Real = 2 * Real - 1;
        }
        else {
            Real *= 2;
        }
    }
}
```

그 이외의 경우는 Normalized number입니다. 정수 부분과 소수 부분을 각각 나누어 이진 변환을 한 뒤 SFP로 변환해줘야 하므로 코드는 다음과 같습니다. 단, 앞서 구현한 함수 sfp return_E에서 input $< 1 = 2^0$인 경우에 대해서는 고려하지 않았기 때문에 value를 $(-1)^S M 2^E$ 형태로 표현했을 때 $E < 0$가 되는 경우를 추가하여 구현했습니다.

```
int num = (int)(input);  // float input의 정수부분
float Real = input - num;  // float input의 소수부분

k.S = 0;
k.F = num;

for (sfp i = 0; i < 10 - return_E(num); i++) {
    k.F *= 2;
}  // 정수부분 fractional part에 입력

for (sfp i = 0; i < 10 - return_E(num); i++) {
    if (Real * 2 >= 1) {
        int k_Real = 1;
        for (sfp j = 0; j < 9 - return_E(num) - i; j++) {
            k_Real *= 2;
        }
        k.F += k_Real;
        Real = 2 * Real - 1;
    }
    else {
        Real *= 2;
    }
}  // 소수부분 fractional part에 입력

if (input < 1) {
    int E = 0;
    while (input < 1) {
        input *= 2;
        E--;
    }
    k.E = 15 + E;

    for (sfp o = 0; o < -E; o++) {
        k.F *= 2;
    }
}

else {
    k.E = return_exp(num);
}
```

[float sfp2float(sfp input)]

float sfp2float(sfp input)은 SFP를 실수형으로 변환시키는 함수입니다.

이를 구현하기 위해 input이 Denormalized number인 경우와 Normalized number인 경우를 나누었습니다.

SFP에서 Denormalized number는 Exponential part가 00000인 경우입니다. 이때, 앞서 말했듯이 E = 1- Bias = -14이므로 이를 이용하여 $(-1)^S M 2^{-14}$의 계산 값을 return 해주면 됩니다.

```c
    struct sfpbits k = { 0, };

    k.SEF = input;
    int exp = k.E - 15;

    if (k.E == 0) {
        float frac = (float)(k.F) / 1024;

        for (int i = 0; i < 14; i++) {    // E = 1 - Bias(15) = - 14
            frac = frac / 2;
        }

        if (k.S == 0) {
            output = frac;
        }
        else {
            output = -frac;
        }
    }
```

그 이외의 경우는 Normalized number입니다. 앞서 말했듯이, Normalized number에서 SFP의 Fraction part의 앞에는 bit 1이 숨겨져 있으므로 이를 적용하여 계산을 해야 합니다. $V = (-1)^S M 2^E$임을 이용하여 return해주면 다음과 같이 구현할 수 있습니다.

```c
    else {
        float frac = (float)(k.F + 1024) / 1024;

        if (exp >= 0) {
            for (int i = 0; i < exp; i++) {
                frac = frac * 2;
            }

            if (k.S == 0) {
                output = frac;
            }
            else {
                output = -frac;
            }
        }

        else {
            for (int i = 0; i < -exp; i++) {
                frac = frac / 2;
            }

            if (k.S == 0) {
                output = frac;
            }
            else {
                output = -frac;
            }
        }

    }
```

[sfp sfp_add(sfp a, sfp b)]

sfp sfp_add(sfp a, sfp b)는 SFP의 덧셈 기능을 수행하는 함수입니다.

우선적으로 NaN, +∞, -∞을 return하는 경우들을 다음과 같이 처리했습니다.

```c
struct sfpbits A = { 0, };
struct sfpbits B = { 0, };
struct sfpbits O = { 0, };

A.SFF = a;
B.SFF = b;

if ((A.E == 31 && A.F != 0) || (B.E == 31 && B.F != 0)) {  //  a == NaN or b == NaN
    output = 31745;  // output = NaN
}

else if (((A.S == 0 && A.E == 31 && A.F == 0) && (B.S == 1 && B.E == 31 && B.F == 0)) || ((A.S == 1 && A.E == 31 && A.F == 0) && (B.S == 0 && B.E == 31 && B.F == 0))) {  //  (a == ∞ && b == -∞) or (a == -∞ && b == ∞)
    output = 31745;  //  output = NaN
}

else if ((A.S == 0 && A.E == 31 && A.F == 0) || (B.S == 0 && B.E == 31 && B.F == 0)) {  //  a == ∞ || b == ∞
    output = 31744;  // output = ∞
}

else if ((A.S == 1 && A.E == 31 && A.F == 0) || (B.S == 1 && B.E == 31 && B.F == 0)) {  //  a == -∞ || b == -∞
    output = 64512;  // output = -∞
}
```

부동소수점에서 덧셈 혹은 뺄셈은 4단계 과정을 통해 진행됩니다.

1. 두 수의 지수를 같게 조정합니다. (이때, 지수가 작은 쪽을 큰 쪽에 맞춥니다.)
2. 연산을 실행합니다.
3. 계산 결과의 가수를 정규화 합니다.
4. 정규화 시킨 만큼 지수를 조절합니다.

그런데 Normalized number와 Denormalized number는 구조상 차이가 있기 때문에 우선적으로 Denormalized number가 포함되어 있는 경우에 대해 고려했습니다. 그 중에서 두 인자가 모두 Denormalized number인 경우를 인자의 부호에 따라 나누고 각 인자의 절댓값간 대소관계를 추가적으로 비교한 뒤 위 4단계 과정을 활용해 구현했습니다.

```c
else if ((A.E == 0) || (B.E == 0)) {
    if (A.E == B.E) {  //  a와 b가 모두 Denormalized number인 경우
        if (A.S == 0 && B.S == 0) {  //  a >= 0 && b >= 0
            sfp Of = A.F + B.F;
            O.F = Of;
            O.S = 0;

            if (Of > 1023) {
                O.E = 1;
            }
            else {
                O.E = 0;
            }
        }

        else if (A.S == 0 && B.S == 1) {  //  a >= 0 && b < 0
            if (a - b + 32768 >= 0) {
                O.F = A.F - B.F;
                O.E = 0;
                O.S = 0;
            }
            else {
                O.F = B.F - A.F;
                O.E = 0;
                O.S = 1;
            }
        }

        else if (A.S == 1 && B.S == 0) {  //  a < 0 && b >= 0
            if (b - a + 32768 >= 0) {
                O.F = B.F - A.F;
                O.E = 0;
                O.S = 0;
            }
            else {
                O.F = A.F - B.F;
                O.E = 0;
                O.S = 1;
            }
        }

        else {  //  a < 0 && b < 0
            sfp Of = A.F + B.F;
            O.F = Of;
            O.S = 1;

            if (Of > 1023) {
                O.E = 1;
            }
            else {
                O.E = 0;
            }
```

그 다음으로는 Normalized number와 Denormalized number간의 연산을 고려했습니다. Normalized number의 지수가 Denormalized number의 지수보다 항상 크거나 같기 때문에 Denormalized number의 지수를 Normalized number의 지수에 맞춰줘야 합니다. 이때 Denormalized number의 경우 E = 1 - Bias이고, Normalized number의 경우 E = Exp – Bias라는 차이가 있다는 점과 Normalized number의 경우만 fraction part 앞에 bit 1이 숨겨져 있다는 점을 주의해서 구현해줘야 합니다. 또한, 부동소수점의 덧셈과정을 따라 구현하되 두 인자의 부호가 같을 때 Overflow가 발생하는 경우와 두 인자의 부호가 다를 때 Normalized number가 return 되는 경우를 예외적으로 처리해줘야 합니다. 인자의 부호에 따라 경우를 나누고 각 인자의 절댓값간 대소관계를 추가적으로 비교한 뒤 구현하였으며 기본적인 틀은 다음과 같습니다.

```
else {  //  a와 b중 하나만 Denormalized number인 경우

    if (A.S == 0 && B.S == 0) {  // 두 인자의 부호가 같을 때
        O.S = 0;

        if (A.E >= B.E) {
            shift = A.E - B.E - 1;  //  (Exp - Bias) - (1 - Bias) = Exp - 0 - 1

            for (int i = 0; i < shift; i++) {  //  지수를 같게 조정
                Bf /= 2;
            }

            sfp Of = Af + 1024 + Bf;  // Normalized number의 Fraction part 앞에 존재하는 hidden bit 고려

            while (Of > 2047) {  // 정규화 과정
                Of /= 2;
                Ae += 1;
            }

            if (Ae >= 31) {  // Overflow가 발생하는 경우
                O.E = 31;
                O.F = 0;
            }
            else {
                O.E = Ae;
                O.F = Of;
            }
        }
```

```
else if (A.S == 0 && B.S == 1) {  // 두 인자의 부호가 다른 경우
    if (a - b + 32768 >= 0) {  // 두 인자 간의 절댓값 비교를 통한 부호 결정
        O.S = 0;

        shift = A.E - B.E - 1;

        for (int i = 0; i < shift; i++) {
            Bf /= 2;
        }

        sfp Of = Af + 1024 - Bf;

        if (A.E == 1) {  // Denormalized number가 return 되는 경우
            if (Of < 1024) {
                O.E = 0;
            }
            else {
                O.E = A.E;
            }
            O.F = Of;
        }

        else {
            while (Of < 1024) {  // 정규화 과정
                Of *= 2;
                A.E -= 1;
            }
            O.E = A.E;
            O.F = Of;
        }
    }
```

이 밖의 경우는 Normalized number 간의 연산만 남게 됩니다. Normalized number와 Denormalized number 간의 연산과 매우 유사하게 진행되지만 두 인자 모두 E = Exp – Bias이고 Fraction part 앞에 hidden bit 1이 존재함을 고려해야 합니다. 또한, 이 경우 역시 부동소수점의 덧셈과정을 따라 구현하되 두 인자의 부호가 같을 때 Overflow가 발생하는 경우와 두 인자의 부호가 다를 때 Normalized number로 return되는 경우를 고려해줘야 합니다. 인자의 부호에 따라 경우를 나누고 각 인자의 절댓값간 대소관계를 추가적으로 비교한 뒤 구현하였으며 기본적인 틀은 다음과 같습니다.

```
sfp Af = A.F;
sfp Bf = B.F;
sfp Ae = A.E;
sfp Be = B.E;

if (A.S == 0 && B.S == 0) {  // a와 b의 부호가 같을 경우
    O.S = 0;

    if (A.E >= B.E) {
        shift = A.E - B.E;
        Bf += 1024;  //  b의 fraction part에 존재하는 hidden bit 고려

        for (int i = 0; i < shift; i++) {  //  지수를 같게 조정하고, 가수를 그에 맞게 조정
            Bf /= 2;
        }

        sfp Of = Af + 1024 + Bf;  //  a의 fraction part에 존재하는 hidden bit 고려

        while (Of > 2047) {
            Of /= 2;
            Ae += 1;
        }

        if (Ae >= 31) {  //  Overflow가 발생하는 경우
            O.E = 31;
            O.F = 0;
        }
        else {
            O.E = Ae;
            O.F = Of;
        }
    }
```

```
else if (A.S == 0 && B.S == 1) {  // a, b의 부호가 다른 경우
    if (a - b + 32768 >= 0) {  // a와 b의 절댓값 간 비교를 통해 부호 결정
        O.S = 0;

        shift = A.E - B.E;
        Bf += 1024;

        for (int i = 0; i < shift; i++) {  // 지수를 같게 조정하고, 가수를 그에 맞게 조정
            Bf /= 2;
        }

        sfp Of = Af + 1024 - Bf;

        while (Of < 1024) {  // 정규화 과정
            if (A.E == 1) {
                if ((0 <= Of) && (Of < 1024)) {  // Denormalized number로 return 되는 경우
                    A.E = 0;
                    break;
                }
            }
            else {
                Of += 2;
                A.E -= 1;
            }
        }
        O.E = A.E;
        O.F = Of;
    }
```

[sfp sfp_mul(sfp a, sfp b)]

sfp sfp_mul(sfp a, sfp b)는 SFP의 곱셈 기능을 수행하는 함수입니다.

우선적으로 NaN, +∞, -∞을 return하는 경우들에 대해 처리했습니다.

```c
sfp output;

struct sfpbits A = { 0, };
struct sfpbits B = { 0, };
struct sfpbits O = { 0, };

A.SEF = a;
B.SEF = b;

if ((A.E == 31 && A.F != 0) || (B.E == 31 && B.F != 0)) {  //  a == NaN or b == NaN
    output = 31745;  // output = NaN
}

else if (((A.E == 31 && A.F == 0) && (B.E == 0 && B.F == 0)) || ((A.E == 0 && A.F == 0) && (B.E == 31 && B.F == 0))) {  // (a == +-∞, b == 0) or  (a = 0, b = +-∞)
    output = 31745;  // output = NaN
}

else if (((A.S == 0 && A.E == 31 && A.F == 0) && B.S == 0) || ((A.S == 1 && A.E == 31 && A.F == 0) && B.S == 1) || (A.S == 0 && (B.S == 0 && B.E == 31 && B.F == 0)) || (A.S == 1 && (B.S == 1 &
    output = 31744; // output = ∞
}

else if (((A.S == 0 && A.E == 31 && A.F == 0) && B.S == 1) || ((A.S == 1 && A.E == 31 && A.F == 0) && B.S == 0) || (A.S == 0 && (B.S == 1 && B.E == 31 && B.F == 1)) || (A.S == 1 && (B.S == 0 &
    output = 64512; // output = -∞
}
```

부동소수점에서의 곱셈은 다음의 과정을 통해 이루어집니다.

1. 가수끼리 곱합니다.
2. 지수끼리 더합니다.
3. 계산 결과의 가수를 정규화 시켜줍니다.

앞서 말했듯이 Normalized number와 Denormalized number는 구조상 차이가 있기 때문에 우선적으로 Denormalized number가 포함되어 있는 경우에 대해 고려해야 합니다. 그 중에서 두 인자가 모두 Denormalized number인 경우, 가장 큰 Denormalized number끼리 곱해도 0이 return되므로 항상 0을 return함을 알 수 있습니다.

```c
else if (A.E == 0 || B.E == 0) {  //  연산에 Denormalized number가 인자로 존재하는 경우
    if (A.E == B.E) {  //  두 인자가 모두 Denormalized number인 경우
        O.E = 0;
        O.F = 0;

        if (A.S == B.S) {
            O.S = 0;
        }
        else {
            O.S = 1;
        }
    }
```

그 다음으로는 Normalized number와 Denormalized number간의 연산을 고려했습니다. 만약 인자 중 0이 포함되어 있다면 0을 return하는 것은 자명한 사실입니다. 이 경우, 모든 Denormalized number는 1보다 작으므로 overflow가 발생할 일은 없지만 return값이 Normalized number인 경우와 Denormalized number인 경우를 구분하여 구현해야 합니다. 부동소수점 곱셈의 3단계 과정을 따라 구현했을 때 기본적인 틀은 다음과 같습니다.

```
else if (A.E == 0) {  // 두 인자 중 하나(a)만 0인 경우
    int Af = A.F;
    int oE = bE - 15;
    int oF = A.F * (B.F + 1024);  // b의 fraction part에만 hidden bit 1이 존재함

    if (A.F == 0) {  // a가 0이라면 곱셈의 결과는 항상 0이 나옴
        O.E = 0;
        O.F = 0;

        if (A.S == B.S) {
            O.S = 0;
        }
        else {
            O.S = 1;
        }
    }

    else {
        oF /= 1024;

        for (int i = 0; i < oE; i++) {
            oF *= 2;
        }

        if (oF < 1024) {  // Denormalized number로 return하는 경우
            O.E = 0;
            O.F = oF;

            if (A.S == B.S) {
                O.S = 0;
            }
            else {
                O.S = 1;
            }
        }

        else {  // Normalized number로 return하는 경우
            int j = 0;
            while (oF > 1023) {
                oF /= 2;
                j++;
            }

            O.E = j + 1;
            O.F = oF;

            if (A.S == B.S) {
                O.S = 0;
            }
            else {
                O.S = 1;
            }
        }

    }
    output = O.SEF;
```

마지막으로는 Normalized간의 연산에 대해 구현했습니다. 이 경우에는 return값이 Denormalized number, Normalized number, Overflow가 될 경우가 모두 존재하므로 이를 유의하여 부동소수점 곱셈의 3단계 과정을 따라 다음과 같이 기본적인 틀을 만들었습니다.

```
    else {   //  Normalized numbers 간의 곱셈 연산
        int oE = aE + bE - 15;   //  곱셈에서 지수의 연산
        int oF = (A.F + 1024) * (B.F + 1024);   //  곱셈에서 가수의 연산

        while (oF > 2096128) {  //  정규화 과정
            oF /= 2;
            oE++;
        }

        while (oF > 2047) {
            oF /= 2;
        }

        if (oE < 0) {  //  Denormalized number로 return할 경우
            while (oE < 0) {
                oF /= 2;
                oE++;
            }
            O.E = oE;
            O.F = oF / 2;
        }

        else if (oE >= 31) {  //  Overflow가 발생할 경우
            O.E = 31;
            O.F = 0;

            if (A.S == B.S) {
                O.S = 0;
            }
            else {
                O.S = 1;
            }
        }

        else {  //  Normalized number로 return할 경우
            if (A.S == B.S) {
                O.S = 0;
            }
            else {
                O.S = 1;
            }
            O.E = oE;
            O.F = oF;
        }
        output = O.SEF;
    }
```

[char* sfp2bits(sfp result)}]

char* sfp2bits(sfp result)는 SFP를 문자열로 변환해 비트 단위로 출력하기 위한 함수입니다.

Malloc을 이용해 문자열의 저장공간을 확보하고 memset을 통해 초기화를 시켜줍니다. SFP는 총 16비트이기 때문에 이를 유의하여 이진변환을 통해 구현하였습니다.

```
char* sfp2bits(sfp result) {

    int i, index = 1;
    char* bit = (char*)malloc(sizeof(char) * 17);

    memset(bit, '0', 17 * sizeof(char));
    bit[16] = '\0';         // the end of string

    for (i = 15; i >= 0; i--) {
        if (result % 2 == 0) {
            bit[i] = '0';
        }
        else {
            bit[i] = '1';
        }
        result /= 2;
    }

    return bit;
}
```