

# 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.

```
6 struct sfpbits {
7     union {
8         struct {
9             sfp F : 10; // F(Fraction bits)에 해당하는 10비트 크기 할당
10            sfp E : 5; // E(Exponential bits)에 해당하는 5비트 크기 할당
11            sfp S : 1; // S(Sign bit)에 해당하는 1비트 크기
12        }; // 합계 16비트
13        sfp SEF;
14    };
15};
```

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.

```
17 sfp return_E(int input) {
18     sfp E = 0;
19
20     while (input > 1) {
21         input = input / 2;
22         E++;
23     }
24
25     return E;
26 }
```

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

```
28 sfp return_exp(int input) {
29     sfp E;
30     E = return_E(input);
31
32     return 15 + E;
33 }
```

## [sfp int2sfp(int input)]

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

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

$1\ 11110\ 1111111111 \rightarrow (-1)^1 * (2047 / 1024) * 2^{15} = -65504$  (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  $+\infty$ 를, -65504보다 작을 때  $-\infty$ 를 나타내는 SFP를 반환하도록 하였습니다.

```
41     if (input > 65504) {
42         k.F = 0;    // Fraction bits = 0000000000
43         k.E = 31;   // Exponential bits = 11111
44         k.S = 0;    // Sign bit = 0
45     }
46
47     else if (input < -65504) {
48         k.F = 0;    // Fraction bits = 0000000000
49         k.E = 31;   // Exponential bits = 11111
50         k.S = 1;    // Sign bit = 1
51     }
```

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.

```
60     else {
61         if (input > 0) {
62             k.F = input;
63             for (sfp i = 0; i < 10 - return_E(input); i++) { // Fraction part의 나머지 부분에 0을 할당하는 과정
64                 k.F += 2;
65             }
66             k.E = return_exp(input);
67             k.S = 0;
68         }
69         else {
70             k.F = -input;
71             for (sfp i = 0; i < 10 - return_E(-input); i++) { // Fraction part의 나머지 부분에 0을 할당하는 과정
72                 k.F += 2;
73             }
74             k.E = return_exp(-input);
75             k.S = 1;
76         }
77     }
78
79     return k.SEF;
80 }
```

### [int sfp2int(sfp input)]

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

문제 조건에 의해서 우리가 우선적으로 신경 써줘야 하는 것은 Input이  $+\infty$ ,  $-\infty$  그리고 NaN일 경우입니다.  $+\infty$ 는 int의 최대범위 값을,  $-\infty$ 와 NaN은 int의 최소범위 값을 return 해줘야 하므로 #include <limits.h>을 추가하여 각각의 경우에서 INT\_MAX와 INT\_MIN을 return하도록 했습니다.

```
89     if (k.E == 31) {
90         if (k.F == 0) {
91             if (k.S == 0) { // 0 1111 0000000000
92                 output = INT_MAX; // +∞ -> int 최대범위값
93             }
94             else { // 1 1111 0000000000
95                 output = INT_MIN; // -∞ -> int 최소범위값
96             }
97         }
98         else {
99             output = INT_MIN; // NaN -> int 최소범위값
100         }
101     }
```

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

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

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

```
107     else {
108         float frac = (float)(1024 + k.F) / 1024; // Normalized value의 fraction bit 내 hidden bit 고려
109         int exp = k.E - 15; // E = Exp - Bias(15)
110
111         if (exp > 0) {
112             for (int i = 0; i < exp; i++) {
113                 frac *= 2;
114             }
115         }
116         else if (exp < 0) {
117             for (int i = 0; i < -exp; i++) {
118                 frac /= 2;
119             }
120         }
121
122         if (k.S == 0) {
123             output = (int)(frac);
124         }
125         else {
126             output = (int)(-frac);
127         }
128     }
```

### [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  $+\infty$ 를, -65504보다 작을 때  $-\infty$ 를 나타내는 SFP를 반환하도록 하였습니다.

```
137         if (input > 65504) {
138             k.F = 0;
139             k.E = 31;
140             k.S = 0;
141         }
142
143         else if (input < -65504) {
144             k.F = 0;
145             k.E = 31;
146             k.S = 1;
147         }
```

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,  $E = 1 - \text{bias} = -14$ 인 점을 고려하여 구현해주면 다음과 같습니다.

```
149         else if ((-0.000061 < input) && (input < 0)) {
150             k.E = 0;
151             k.S = 1;
152             float Real = -input;
153
154             Real += -16384; // E = 1 - Bias = -14
155
156             for (sfp i = 0; i < 10; i++) { // 소수부분 이진변환
157                 if (Real + 2 >= 1) {
158                     int k_Real = 1;
159                     for (sfp j = 0; j < 9 - i; j++) {
160                         k_Real += 2;
161                     }
162                     k.F += k_Real;
163                     Real = 2 * Real - 1;
164                 }
165                 else {
166                     Real += 2;
167                 }
168             }
169         }
170
171         else if ((0 <= input) && (input < 0.000061)) {
172             k.E = 0;
173             k.S = 0;
174             float Real = input;
175
176             Real += 16384; // E = 1 - Bias = -14
177
178             for (sfp i = 0; i < 10; i++) { // 소수부분 이진변환
179                 if (Real + 2 >= 1) {
180                     int k_Real = 1;
181                     for (sfp j = 0; j < 9 - i; j++) {
182                         k_Real += 2;
183                     }
184                     k.F += k_Real;
185                     Real = 2 * Real - 1;
186                 }
187                 else {
188                     Real += 2;
189                 }
190             }
191         }
```

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  $\text{input} < 1 = 2^0$ 인 경우에 대해서는 고려하지 않았기 때문에 value를  $(-1)^S M 2^E$  형태로 표현했을 때  $E < 0$ 가 되는 경우를 추가하여 구현했습니다.

```
159 int num = (int)(input); // float input의 정수부분
160 float Real = input - num; // float input의 소수부분
161
162 k.S = 0;
163 k.F = num;
164
165 for (sfp i = 0; i < 10 - return_E(num); i++) {
166     k.F *= 2;
167 } // 정수부분 fractional part에 입력
168
169 for (sfp i = 0; i < 10 - return_E(num); i++) {
170     if (Real * 2 >= 1) {
171         int k_Real = 1;
172         for (sfp j = 0; j < 9 - return_E(num) - i; j++) {
173             k_Real *= 2;
174         }
175         k.F += k_Real;
176         Real = 2 * Real - 1;
177     }
178     else {
179         Real *= 2;
180     }
181 } // 소수부분 fractional part에 입력
182
183 if (input < 1) {
184     int E = 0;
185     while (input < 1) {
186         input *= 2;
187         E--;
188     }
189     k.E = 15 + E;
190
191     for (sfp o = 0; o < -E; o++) {
192         k.F *= 2;
193     }
194 }
195
196 else {
197     k.E = return_exp(num);
198 }
```

### [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  $E = 1 - \text{Bias} = -14$ 이므로 이를 이용하여  $(-1)^S M 2^{-14}$ 의 계산 값을 return 해주면 됩니다.

```
287 struct sfpbits k = { 0, };
288
289 k.SEF = input;
290 int exp = k.E - 15;
291
292 if (k.E == 0) {
293     float frac = (float)(k.F) / 1024;
294
295     for (int i = 0; i < 14; i++) { // E = 1 - Bias(15) = -14
296         frac = frac / 2;
297     }
298
299     if (k.S == 0) {
300         output = frac;
301     }
302     else {
303         output = -frac;
304     }
305 }
```

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 return해 주면 다음과 같이 구현할 수 있습니다.

```
308 else {
309     float frac = (float)(k.F + 1024) / 1024;
310
311     if (exp >= 0) {
312         for (int i = 0; i < exp; i++) {
313             frac = frac * 2;
314         }
315
316         if (k.S == 0) {
317             output = frac;
318         }
319         else {
320             output = -frac;
321         }
322     }
323     else {
324         for (int i = 0; i < -exp; i++) {
325             frac = frac / 2;
326         }
327
328         if (k.S == 0) {
329             output = frac;
330         }
331         else {
332             output = -frac;
333         }
334     }
335 }
336
337 }
```

### [sfp sfp\_add(sfp a, sfp b)]

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

우선적으로 NaN,  $+\infty$ ,  $-\infty$ 을 return하는 경우들을 다음과 같이 처리했습니다.

```
346 struct sfpbits A = { 0, };
347 struct sfpbits B = { 0, };
348 struct sfpbits O = { 0, };
349
350 A.SEF = a;
351 B.SEF = b;
352
353 if ((A.E == 31 && A.F != 0) || (B.E == 31 && B.F != 0)) { // a == NaN or b == NaN
354     output = 31745; // output = NaN
355 }
356
357 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 == ∞)
358     output = 31745; // output = NaN
359 }
360
361 else if ((A.S == 0 && A.E == 31 && A.F == 0) || (B.S == 0 && B.E == 31 && B.F == 0)) { // a == ∞ || b == ∞
362     output = 31744; // output = ∞
363 }
364
365 else if ((A.S == 1 && A.E == 31 && A.F == 0) || (B.S == 1 && B.E == 31 && B.F == 0)) { // a == -∞ || b == -∞
366     output = 64512; // output = -∞
367 }
```

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.

```
359 else if ((A.E == 0) || (B.E == 0)) {
360     if (A.E == 0 && B.E == 0) { // a & b are both denormalized (add number) case
361         if (A.S == 0 && B.S == 0) { // a & b are both positive
362             sfp OF = A.F + B.F;
363             OE = 0;
364             OS = 0;
365             if (OF > 1023) {
366                 OE = 1;
367             }
368             else {
369                 OE = 0;
370             }
371         }
372         else if (A.S == 0 && B.S == 1) { // a > 0 & b < 0
373             if (a - b + 32768 > 0) {
374                 OF = A.F - B.F;
375                 OE = 0;
376                 OS = 0;
377             }
378             else {
379                 OF = B.F - A.F;
380                 OE = 0;
381                 OS = 1;
382             }
383         }
384         else if (A.S == 1 && B.S == 0) { // a < 0 & b > 0
385             if (b - a + 32768 > 0) {
386                 OF = B.F - A.F;
387                 OE = 0;
388                 OS = 0;
389             }
390             else {
391                 OF = A.F - B.F;
392                 OE = 0;
393                 OS = 1;
394             }
395         }
396         else { // a < 0 & b < 0
397             OF = A.F + B.F;
398             OE = 0;
399             OS = 1;
400             if (OF > 1023) {
401                 OE = 1;
402             }
403             else {
404                 OE = 0;
405             }
406         }
407     }
408 }
```

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  $E = 1 - \text{Bias}$ 이고, Normalized number의 경우  $E = \text{Exp} - \text{Bias}$ 라는 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.

```

429 else { // a와 b중 하나만 Denormalized number인 경우
430
431     if (A.S == 0 && B.S == 0) { // 두 인자의 부호가 같을 때
432         O.S = 0;
433
434         if (A.E >= B.E) {
435             shift = A.E - B.E - 1; // (Exp - Bias) - (1 - Bias) = Exp - 0 - 1
436
437             for (int i = 0; i < shift; i++) { // 지수를 같게 조정
438                 Bf /= 2;
439             }
440
441             sfp Of = Af + 1024 + Bf; // Normalized number의 Fraction part 앞에 존재하는 hidden bit 고려
442
443             while (Of > 2047) { // 정규화 과정
444                 Of /= 2;
445                 Ae += 1;
446             }
447
448             if (Ae >= 31) { // Overflow가 발생하는 경우
449                 O.E = 31;
450                 O.F = 0;
451             }
452             else {
453                 O.E = Ae;
454                 O.F = Of;
455             }
456         }
457         else if (A.S == 0 && B.S == 1) { // 두 인자의 부호가 다른 경우
458             if (a - b + 32768 >= 0) { // 두 인자 간의 절댓값 비교를 통한 부호 결정
459                 O.S = 0;
460
461                 shift = A.E - B.E - 1;
462
463                 for (int i = 0; i < shift; i++) {
464                     Bf /= 2;
465                 }
466
467                 sfp Of = Af + 1024 - Bf;
468
469                 if (A.E == 1) { // Denormalized number가 return 되는 경우
470                     if (Of < 1024) {
471                         O.E = 0;
472                     }
473                     else {
474                         O.E = A.E;
475                     }
476                     O.F = Of;
477                 }
478                 else {
479                     while (Of < 1024) { // 정규화 과정
480                         Of *= 2;
481                         A.E -= 1;
482                     }
483                     O.E = A.E;
484                     O.F = Of;
485                 }
486             }
487         }
488     }
489 }

```



이 밖의 경우는 Normalized number 간의 연산만 남게 됩니다. Normalized number와 Denormalized number 간의 연산과 매우 유사하게 진행되지만 두 인자 모두  $E = \text{Exp} - \text{Bias}$ 이고 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.

```

665     sfp Af = A.F;
666     sfp Bf = B.F;
667     sfp Ae = A.E;
668     sfp Be = B.E;
669
670     if (A.S == 0 && B.S == 0) { // a와 b의 부호가 같을 경우
671         O.S = 0;
672
673         if (A.E >= B.E) {
674             shift = A.E - B.E;
675             Bf += 1024; // b의 fraction part에 존재하는 hidden bit 고려
676
677             for (int i = 0; i < shift; i++) { // 지수를 같게 조정하고, 가수를 그에 맞게 조정
678                 Bf /= 2;
679             }
680
681             sfp Of = Af + 1024 + Bf; // a의 fraction part에 존재하는 hidden bit 고려
682
683             while (Of > 2047) {
684                 Of /= 2;
685                 Ae += 1;
686             }
687
688             if (Ae >= 31) { // Overflow가 발생하는 경우
689                 O.E = 31;
690                 O.F = 0;
691             }
692             else {
693                 O.E = Ae;
694                 O.F = Of;
695             }
696         }

```

```

724     else if (A.S == 0 && B.S == 1) { // a, b의 부호가 다른 경우
725         if (a - b + 32768 >= 0) { // a와 b의 절댓값 간 비교를 통해 부호 결정
726             O.S = 0;
727
728             shift = A.E - B.E;
729             Bf += 1024;
730
731             for (int i = 0; i < shift; i++) { // 지수를 같게 조정하고, 가수를 그에 맞게 조정
732                 Bf /= 2;
733             }
734
735             sfp Of = Af + 1024 - Bf;
736
737             while (Of < 1024) { // 정규화 과정
738                 if (A.E == 1) {
739                     if ((0 <= Of) && (Of < 1024)) { // Denormalized number로 return 되는 경우
740                         A.E = 0;
741                         break;
742                     }
743                 }
744                 else {
745                     Of += 2;
746                     A.E -= 1;
747                 }
748             }
749             O.E = A.E;
750             O.F = Of;
751         }

```

### [sfp sfp\_mul(sfp a, sfp b)]

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

우선적으로 NaN,  $+\infty$ ,  $-\infty$ 을 return하는 경우들에 대해 처리했습니다.

```
901 sfp output;
902
903 struct sfpbits A = { 0, };
904 struct sfpbits B = { 0, };
905 struct sfpbits O = { 0, };
906
907 A.SEF = a;
908 B.SEF = b;
909
910 if ((A.E == 31 && A.F != 0) || (B.E == 31 && B.F != 0)) { // a == NaN or b == NaN
911     output = 31745; // output = NaN
912 }
913
914 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 == +-∞)
915     output = 31745; // output = NaN
916 }
917
918 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 && B.E == 31 && B.F == 0)) {
919     output = 31744; // output = ∞
920 }
921
922 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 && B.E == 31 && B.F == 1)) {
923     output = 64512; // output = -∞
924 }
```

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.

```
928 else if (A.E == 0 || B.E == 0) { // 연산에 Denormalized number가 인자로 존재하는 경우
929     if (A.E == B.E) { // 두 인자가 모두 Denormalized number인 경우
930         O.E = 0;
931         O.F = 0;
932
933         if (A.S == B.S) {
934             O.S = 0;
935         }
936         else {
937             O.S = 1;
938         }
939     }
940 }
```

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.

```
933 else if (A.E == 0) { // 두 인자 중 하나(a)만 0인 경우
934     int Af = A.F;
935     int oE = bE - 15;
936     int oF = A.F * (B.F + 1024); // b의 fraction part에만 hidden bit 10이 존재함
937
938     if (A.F == 0) { // a가 0이라면 곱셈의 결과는 항상 0이 나옴
939         O.E = 0;
940         O.F = 0;
941
942         if (A.S == B.S) {
943             O.S = 0;
944         }
945         else {
946             O.S = 1;
947         }
948     }
949
950     else {
951         oF /= 1024;
952
953         for (int i = 0; i < oE; i++) {
954             oF *= 2;
955         }
956
957         if (oF < 1024) { // Denormalized number로 return하는 경우
958             O.E = 0;
959             O.F = oF;
960
961             if (A.S == B.S) {
962                 O.S = 0;
963             }
964             else {
965                 O.S = 1;
966             }
967         }
968
969         else { // Normalized number로 return하는 경우
970             int j = 0;
971             while (oF > 1023) {
972                 oF /= 2;
973                 j++;
974             }
975
976             O.E = j + 1;
977             O.F = oF;
978
979             if (A.S == B.S) {
980                 O.S = 0;
981             }
982             else {
983                 O.S = 1;
984             }
985         }
986     }
987 }
988 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.

```
1050 else { // Normalized numbers 간의 곱셈 연산
1051     int oE = aE + bE - 15; // 곱셈에서 지수의 연산
1052     int oF = (A.F + 1024) * (B.F + 1024); // 곱셈에서 가수의 연산
1053
1054     while (oF > 2096128) { // 정규화 과정
1055         oF /= 2;
1056         oE++;
1057     }
1058
1059     while (oF > 2047) {
1060         oF /= 2;
1061     }
1062
1063     if (oE < 0) { // Denormalized number로 return할 경우
1064         while (oE < 0) {
1065             oF /= 2;
1066             oE++;
1067         }
1068         O.E = oE;
1069         O.F = oF / 2;
1070     }
1071
1072     else if (oE >= 31) { // Overflow가 발생할 경우
1073         O.E = 31;
1074         O.F = 0;
1075
1076         if (A.S == B.S) {
1077             O.S = 0;
1078         }
1079         else {
1080             O.S = 1;
1081         }
1082     }
1083
1084     else { // Normalized number로 return할 경우
1085         if (A.S == B.S) {
1086             O.S = 0;
1087         }
1088         else {
1089             O.S = 1;
1090         }
1091         O.E = oE;
1092         O.F = oF;
1093     }
1094     output = O.SEF;
1095 }
```

**[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.

```
929 char* sfp2bits(sfp result) {
930
931     int i, index = 1;
932     char* bit = (char*)malloc(sizeof(char) * 17);
933
934     memset(bit, '0', 17 * sizeof(char));
935     bit[16] = '\0'; // the end of string
936
937     for (i = 15; i >= 0; i--) {
938         if (result % 2 == 0) {
939             bit[i] = '0';
940         }
941         else {
942             bit[i] = '1';
943         }
944         result /= 2;
945     }
946
947     return bit;
948 }
```