System Program – Assignment1

2020312552 소프트웨어학과 김민수

비트 조작: 0x???? 식의 16진수 표현과 &, |, ^ 등을 이용하여 비트 조작

전체적인 고려할 점: int라면 2의 지수 표현법으로 나타낸 뒤 sfp에 맞도록 비트 생성

float이라면 각 part를 분리하여 sfp 크기에 맞도록 조정

add: round to even, 각각의 frac part를 지수에 맞추어 위치를 조정하고 덧셈, 자리 수에 따른 exp 값 변경, sign, exp, frac 모두 반영하여 return

mul: round to even, 각 frac part끼리 곱하고, 지수에 맞추어 위치 조정, exp는 합하며, 자리 수가 늘어났다면 exp + 1, sign, exp, frac 모두 반영하여 return

특수 예외: input이 inf, -inf, NaN, 0(mul)인 경우

Output이 inf, -inf, NaN, 0 등인 경우

typedef unsigned short sfp; // 16비트 C언어 정수형 자료형을 floating point number로 해석할 것.

sfp int2sfp(int input);

=> int 자료형으로 저장된 input을 sign, exp, frac part로 나누어 해석하고, 이를 sfp 자료형에 저장하여 return

{

1. 예외 처리{

input == 0 -> return 0x0000;

input > 65535(0111 1011 1111 1111) -> return inf(0111 1100 0000 0000)

input < -65535(1111 1011 1111 1111) -> return -inf(1111 1100 0000 0000)

}

2. 부호 처리{

input < 0 이면 input = -input; SIGN = 0x8000 (1000 0000 0000 0000)

}

3. E 계산{

twop(power of two) = while 속에서 x2 반복

key = input의 E를 계산하기 위해 매번 1씩 커짐

twop와 input의 크기 비교를 통해 반복문 탈출

}

4. FRAC part 위치 조정{

key가 10 이하이면 input(int type)의 bit pattern을 sfp의 frac part로 좌측 shift

key가 10 초과면 input의 bit pattern을 sfp의 frac part 위치로 우측 shift.

이 때 round-to-zero => 절댓값이 작은 방향이므로 가장 우측 bit 버림

}

5. return value 만들기 및 return{

FRAC part의 implied leading 1 bit를 숨김(0000 0011 1111 1111과 & 연산)

EXP = EXP + 15; Bias: 2^(exp bit 개수 – 1) -1 = 2^(5-1) -1 = 15

sfp type return variable에 SIGN, (EXP << 10), FRAC을 |(or) 연산

= 모두 반영된 return value 완성 => return

}

}

int sfp2int(sfp input);

=> sfp 자료형 input을 sign, exp, frac part로 나누고, exp를 통해 E(실제 2의 지수)를 구한다. frac part을 E에 위치에 따라 조절하여 return

{

1. 예외 처리{

input == 0x0000(type sfp) -> return 0(type int);

input == 0x7C00(positive inf, 0111 1100 0000 0000)

-> return TMax(type int, 0x7FFFFFFF, 0111 1111 1111 1111 1111 1111 1111 1111)

input == 0xFC00(negative inf, 1111 1100 0000 0000)

-> return TMin(type int, 0x80000000, 1000 0000 0000 0000 0000 0000 0000 0000)

input == NaN : (input & 0x7C00) == 0x7C00 && (input & 0x03FF) != 0

-> exp part가 11111이면서 frac part에 0이 아닌 bit가 존재한다면

-> return TMin(0x80000000)

}

2. sfp의 sign, exp, frac part 만들기{

SIGN = (0x00008000 & input) << 16;

-> input(sfp)의 sign bit를 bit and를 통해 얻고 int type의 sign bit 위치로 옮김.

ret(return value) = 0x000003FF(0000 0000 0000 0000 0000 0011 1111 1111)

-> input(sfp)와 and 연산, 0x00000400(0000 0000 0000 0000 0000 0100 0000 0000)

과 or 연산 -> frac part와 implied leading 1 bit를 만듦.

sfp EXP = 0x7C00 & input; input과 0x7C00의 and 연산을 통해 exp part를 만듦.

정수형 해석을 위해 EXP >> 10, E(실제 지수)를 구하기 위해 – 15(Bias)

현재 sfp를 정수형으로 해석할 때 EXP == E(실제 지수)

}

3. int에 맞도록 위치 조정{

E가 10 이하면 (10–E)만큼 ret을 우측 shift(round-toward-zero = bit 버림)

E가 10 초과면 (E-10)만큼 ret을 좌측 shift

}

4. SIGN 고려 후 return{

과정 2에서 만든 SIGN가 음수면 ret = -ret

return ret; -> sign 고려함, exp로 위치 조정함, frac part가 bit pattern

}

}

sfp float2sfp(float input);

=> float 자료형으로 저장된 input의 sign, exp, frac part를 분리하고, 각각 sfp 크기에 맞도록 조절하여 sfp 자

료형에 저장하고 return

{

1. 예외 처리{

input == 0x00000000(float, +0.0) -> return 0x0000(sfp, +0.0)

input == 0x80000000(float, -0.0) -> return 0x8000(sfp, -0.0)

input > 65535 -> return 0x7C00(sfp, positive inf)

input < -65535 -> return 0xFC00(sfp, negative inf)

}

2. sign, exp, frac part 처리{

// unsigned int tmp = *(unsigned int*)&input : use casting

Casting 피하기 위해서 union 사용

Union의 float 변수에 input 저장 후, unsigned int 변수로 해석

int 변수를 이용해 exp part를 가져오고

>> 23(float의 frac part 크기)하여 정수형으로 해석

sfp형 exp를 저장할 변수에 (int형 exp변수 – Bias of float type)

(sfp형 exp 변수 + Bias of sfp type) << 10; // sfp exp part 위치로 옮김

sfp형 frac 변수에 float의 frac part를 int를 이용해 가져오고, 우측 13 shift하여 저장

shift시 bit 손실을 round-to-zero : 절댓값 작도록 bit 버림

sfp형 ret 변수 = SIGN | EXP of sfp type | FRAC of sfp type

}

3. return ret;

}

float sfp2float(sfp input);

=> sfp 자료형 input을 sign, exp, frac part로 나누고, exp를 bias에 따라 float 형으로 변경하고, sfp frac part를 float frac part위치로 옮겨서 return

{

unsigned int형 tmp 변수:

float에 맞춰진 SIGN, EXP, FRAC 변수들의 bit를 or 연산을 통해 만듦

union을 통해 unsigned int형 tmp, float 형 asFloat 만듦.

1. 예외 처리{

input == 0x0000(sfp, +0.0) -> return 0x00000000(float, +0.0)

input == 0x8000(sfp, -0.0) -> return 0x80000000(float, -0.0)

input == NaN(sfp) -> return 0x7FFFFFFF(float, nan)

input == 0x7C00(sfp, inf) -> return 0x7F800000(float, +inf) 0111 1111 1000 0000 …

input == 0xFC00(sfp, -inf) -> return 0xFF800000(float, -inf) 1111 1111 1000 0000 …

}

2. sfp를 float에 맞도록 조정{

sfp의 sign bit를 unsigned int 형으로 가져오고 << 16하여 float 위치에 맞춤

sfp의 exp part를 unsigned int 형으로 가져오고

- Bias of sfp, + Bias of float, << 23; // float exp part 위치로 맞춤

sfp의 frac part를 unsigned int 형으로 단순 복사 후 << 13

오른쪽 13개 bit는 0으로 채워짐

}

3. union 활용 return{

union 변수 선언 후 tmp(unsigned int)에 만든 bit 변수(tmp)를 저장

float ret = unionvar.asFloat;

return ret;

}

}

sfp sfp_add(sfp a, sfp b);

{

1. 예외 처리{

input == NaN : (input & 0x7C00) == 0x7C00 && (input & 0x03FF) != 0

-> return NaN (my example : 0111 1111 1111 1111)

input has inf or -inf

inf + inf -> return inf

inf + -inf -> return NaN

-inf + inf -> return NaN

-inf + -inf -> return -inf

inf + Normal value -> return inf

Normal value + inf -> return inf

-inf + Normal value -> return -inf

Normal value + -inf -> return -inf

0 + 0인 경우: exp와 frac part를 검사하여 둘 다 0인 경우 바로 0 return

}

2. 덧셈{

a, b의 exp part를 비교하여 diff 변수 만듦: diff = exp of a – exp of b

frac을 모두 왼쪽으로 미룸

위치에 맞는 tmp_diff 변수를 만듦(tmp_diff1 = ...11 1111 .., tmp_diff2 = ...10 0000 ...)

diff > 0: a의 exp part가 큰 경우

결과의 sign: a의 부호를 따름

b의 frac part를 |diff|만큼 우측 shift

이때 round-to-even:

잘릴 개수를 세고, 해당 위치에서 (>1/2) or (.5 && 잘릴 바로 앞 위치와 잘리는 첫번째 모두 1) 이면 +1, 나머지 그대로 shift

Ex) ???? ???? ???? ??1 | 1 => shift 전 +1

???? ???? ???? ??1 | 0 or 0 | 1 or 0 | 0 => 그냥 shift

결과의 exp part는 a가 크므로 a를 따름

diff < 0: b의 exp part가 큰 경우

결과의 sign: b의 부호를 따름

a의 frac part를 |diff|만큼 우측 shift

이때 위와 마찬가지로 round-to-even

결과의 exp part는 b가 크므로 b를 따름

diff == 0: a, b의 exp part가 같은 경우

frac part를 비교하여 절댓값이 큰 쪽의 부호를 따름

exp는 둘 중 아무거나 사용: 본인의 경우 a exp 사용

sign of A == sign of B: 부호가 같다면

결과의 frac part = frac of A + frac of B

부호가 다르면:

diff > 0:

결과의 frac part = frac of A – frac of B

diff < 0:

결과의 frac part = frac of B – frac of A

diff == 0:

결과의 frac part = 큰 frac – 작은frac == 차의 절댓값

frac == 0: return 0;

frac을 20자리 우측 쉬프트, 이 때 버려질 위치는 고정되었으므로 shift한 후 round to even 실행

for(i = 0 ~ 9, 10번)

frac & 0x0400 != 0:

implied leading 1 bit 자리에 최대 bit가 있다면 break;

최대 bit가 implied leading 1 bit에 없다면 자리수가 줄어듦

-> frac << 1, exp--;

frac을 *2하고 exp에서 1을 뺌.

}

3. ret 변수에 반영하고 return{

FRAC = FRAC & 0x03FF; // implied leading 1 bit 숨김

sfp ret 변수에 SIGN, (EXP << 10), FRAC을 모두 or 연산하여 만듦

만약 exp >= 31(<< 10 되지 않은 상태를 정수형으로 해석한 값)이면 inf

-> ret = 0x7C00 | SIGN; // sign 반영한 inf값 저장

return ret;

}

}

sfp sfp_mul(sfp a, sfp b);

{

1. 예외 처리{

input == NaN : (input & 0x7C00) == 0x7C00 && (input & 0x03FF) != 0

-> return NaN (my example : 0111 1111 1111 1111)

input has inf or -inf

(+, -)0 * (+, -)inf -> return NaN

(+, -)inf * (+, -)0 -> return NaN

inf * inf -> return inf

inf * -inf -> return -inf

-inf * inf -> return -inf

-inf * -inf -> return inf

inf * Normal value -> return inf

inf * -Normal value -> return -inf

Normal value * inf -> return inf

-Normal value * inf -> return -inf

-inf * Normal value -> return -inf

-inf * -Normal value -> return inf

Normal value * -inf -> return -inf

-Normal value * -inf -> return inf

(+, -)0 * (+, -)Normal value -> return 0;

(+, -)Normal value * (+, -)0 -> return 0;

}

2. 곱셈{

결과의 E = E of A + E of B;

Unsigned int 이용하여 곱셈

최대 비트 위치 체크, 얼마나 shift 해야 하는 지 loc 변수에 저장

loc만큼 round to even하여 우측 shift

While(FRAC < 0x0400) : 0000 0100 0000 0000:

frac의 최대 bit가 implied leading 1 bit 위치보다 우측에 있을 동안

implied leading 1 bit 위치에 최대 bit가 위치할 때까지 좌측 shift

}

3. ret 변수에 반영하고 return{

FRAC = FRAC & 0x03FF; // implied leading 1 bit 숨김

sfp ret 변수에 SIGN, (EXP << 10), FRAC을 모두 or 연산하여 만듦

만약 exp >= 31(<< 10 되지 않은 상태를 정수형으로 해석한 값)이면 inf

-> ret = 0x7C00 & SIGN; // sign 반영한 inf값 저장

return ret;

}

```
}

char* sfp2bits(sfp result);
{
        malloc 함수를 이용하여 길이 17인 char 배열 만듦
        sfp filter = 0x8000; 1000 0000 0000 0000 : 앞에서부터 확인하기 위한 filter
        for(i = 0 ~ 15, 16번)
                result(parameter)와 filter를 & 연산하여 값이 0이 아니면 answer[i] = '1' 저장
                        & 연산 후 결과가 0이면 answer[i] = '0' 저장
                for문 내에 증감문: i++, filter = filter << 1
                => filter를 다음 자리로 넘기고 answer에 저장할 위치인 i도 1씩 증가시킴
        for문 탈출
        answer[16] = '₩0'; // 마지막에 NULL 문자 삽입
        return answer
}
```