컴퓨터 SW 시스템 개론
LAB 2

20190650 김관호

1. 배운 것

integer에 이어서 floating point의 표현 방법을 배웠다. 길이가 제한된 상황을 극복하기 위해 Exponent, Mentissa를 도입해 수를 **(-1)^s*M*2^E**로 표현하는 방법인 IEEE Floating Point Standard를 배웠다. 고정된 길이에서 최대한 다양한 수를 표현할 수 있게 되었고, Mentissa의 할당 길이를 23bits로 길게 하여 0에 가까이 갈수록 더욱더 정교하게 표현할 수 있게 되었다. 0에 근접한 부분을 정교하게 표현할수록 이 수를 2^E 배 했을 때도 옆 수와의 차이가 적어질 수 있다.

float가 나타낼 수 있는 수의 범위는 -2^127~2^128로 int의 범위를 충분히 커버할 수 있음에도 **floating은 모든 int를 담지 못한다.** 그 **이유**는 float의 Mentissa를 표현하는 **fraction part가 23bits이기 때문**이다. 앞서 float는 수를 (-1)^s*M*2^E로 표현한다고 했다. s는 sign만 정해주고, 2^E는 bit vector를 shift해주는 역할만 한다. 즉, M이 float가 표현되는 bit vector를 구성한다. 하지만, **M은 hidden bit를 포함해 24bits까지만 표현이 가능**하다. **반면, int는 32bits**이다. 그래서 int의 **남은 8bits를 표현하지 못하게 된다.**
그렇다면 이런 표현하지 못하는 상황은 언제 나오게 될까? shift를 24번 넘게 하는 경우로, E가 24보다 크면 fraction part를 넘어가 bit vector 끝부분에는 0만 추가될 것이다. 반대로 생각하면 대략 2^24보다 작은 경우에서는 float는 모든 int를 표현할 수 있다고 볼 수 있다. 또한 2^24보다 커진다고 해도 몇몇개만 표현할 수 없게 될 것이다. 이런 경우, rounding(round to even)을 통해 표현할 수 있는 가장 가까운 int로 변환된다.

2. 구현 설명

negate(int x)

간단히 bit level에서 -x의 정의를 생각하면 된다. -x = ~x+1이 된다.

isLess(int x, int y)

x - y < 0 인지 확인하는 함수로, overflow를 조심해야 할 것 같지만 고려할 필요 없게 짤 수 있다. 두 가지를 고려할 것이다.
결과가 곧바로 정해지는 경우를 먼저 생각해보면, x, y의 부호가 반대일 때이다. 이 때는 계산할 필요도 없다.
다음으로 x, y의 부호가 같을 때를 생각해보자. 이 때 **합이 아닌, 차를 구하는 것**이므로

overflow가 일어날 일이 없다. 부호가 같은 경우에서, 그 차이는 항상 int 범위 이내일 것이기 때문이다. x < y 이려면, 부호에 관계없이 x - y < 0이 되어야 한다. 이는 negate()를 이용해 차이를 구하고 이 때의 sign bit를 확인해보면 알 수 있다.

float_abs(unsigned uf)

IEEE standard 형태로 들어오는 bit vector의 sign bit를 건드려주면 된다. 그 전에, 입력값이 NaN인지 INF인지 확인해주자. 두 경우는 sign part, fraction part 상관없이 exponent part가 [1111 1111]인지 보면 된다. 덧붙이자면, fraction이 0이면 INF, 0이 아니면 NaN이다.
만약 NaN, INF가 아니라면 sign bit만 complement를 취해주면 된다.

float_twice(unsigned uf)

IEEE standard 형태로 들어오는 bit vector를 두 배 해줘야 한다. 이 때도 마찬가지로, 입력값이 NaN, INF일 때 먼저 처리해준다. 또한, 두 배 했을 때 NaN, INF가 되는 경우도 생길 것이므로 생각해야 한다. NaN인 경우에는 0x7FC00000를 리턴한다. 반면, NaN을 거르고 난 후엔 fraction part가 0으로 되어있을 것이므로 INF인 경우에 따로 처리할 필요 없이 두 배한 bit vector를 똑같이 반환하면 된다.

2배를 해도 float 범위 안에 드는 정상적인 경우에서는 exponent part에 1을 더해주면 된다. 이게 가능한 이유는 exponent part는 2의 지수로 들어가기 때문에 1을 더한 것은 2를 곱해준 효과와 동일해지기 때문이다.

float_i2f(int x)

int로 들어온 값을 IEEE float 형식에 맞게 변환하고 리턴하면 된다. 이를 위해서는 크게 세 가지를 고려해야 한다.

1. exponent part 찾기

exponent part를 찾기 위해 들어온 bit vector 중 1이 나오는 가장 큰 자릿수를 알아야 한다. bit vector 1001 1101이 있을 때 이를 IEEE float 형식으로 변환하기 위해 1.0011101 * 2^7 와 같이 바꿔주는 과정으로 생각하면 쉽다. 이 때 E = 7이고 expoenent part = E + bias가 된다.
위와 같이 exponent part를 찾기 위해 본 과제에서는 exponent_part = 127 + 31로 잡고, 1이 나올 때까지 left shift하며 exponent_part를 1씩 깎았다. exponent_part를 127 + 31로 초기화한 이유를 먼저 설명하자면, 127은 bias이고 left shift를 이용할 것이므로 31 - left shift한 횟수 = E가 될 것이기 때문이다.

2. fraction part 찾기

fraction part는 exponent part에서 이어진다. left shift한 bit vector(제출한 코드에서는 ux)는 MSB가 1로 시작되는 bit vector일 것이다. 여기서 MSB 뒤의 23bits를 읽어주면 fraction part가 된다. 하지만, 뒤쪽의 남은 8bits까지 고려해줘야 한다. 이는 rounding에서 이어나가겠다.

3. rounding

left shift한 bit vector의 앞선 24bits의 MSB를 제외한 23bits는 fraction part에 들어가고, 나머지 8bits는 rounding을 고려하기 위해 round_bits에 넣어준다.

round 조건은 아래 두 경우이다.

1. round_bits가 0.5보다 클 때
2. round_bits가 0.5이고, fraction part의 LSB가 1일 때(round to even)

두 경우 중 하나를 만족한다면 rounding이 일어나 fraction part++를 해준다. 여기서도 또 고려해줘야 할 것이 있는데, fraction part에서 overflow가 일어난 경우다. 이 때는 1.111에서 10.00과 같이 자릿수가 올라간 경우로, exponent part에 1 더해주고 fraction part는 0으로 바꿔준다.

이제 결과를 모두 합쳐 반환하면 된다.

return sign_part | (exponent_part << 23) | fraction_part;

float_f2i(unsigned uf)

float를 int로 바꿔줘야 한다. 이 때 0.xx와 같이 정수가 아니거나, int의 범위를 벗어나는 경우를 잘 생각해 줘야한다. 특이한 경우는 아래와 같다.

1. exponential part가 1111 1111로, NaN, INF인 경우
2. exponential part가 0인 경우엔 fraction part가 어떻든지간에 0.xxx로 나와 int가 표현하지 못함. 0을 리턴해야 함.
3. exponent part < bias 일 때, 앞선 경우와 비슷하게 0.xxx가 되므로 0을 리턴해야 함.
4. exponent part – bias = E > 30 일 때, TMAX를 초과하게 된다.

이 외 정상적인 상황에서는 fraction part의 MSB 앞 부분, 그러니까 24번째 bit에 hidden bit인 1을 넣어 normalize 한다. exponent part가 0인 경우를 위의 특이 케이스에서 이미 처리했기 때문에, exponent part != 0인 경우만 남아 normalize 할 수 있다. 다음으로 normalize한 fraction part를 E만큼 shift해주면 된다. 이 때 fraction part는 23bits임을 인지하고 shifting 해야 한다. E < 23이라면 23 – E 만큼을 right shift 해야 하고, E >= 23이라면 E – 23 만큼을 left shift 해야 한다. 이렇게 해야 23bits짜리 fraction part를 적절히 길이 조절 할 수 있게 된다.

3. 느낀 점

int -> float 에서는 rounding을 float -> int 에서는 0.xxx, 범위 초과를 고려해주는 것이 포인트였다. (float) (int)같은 형변환 안에 상당한 bit level operator가 깔려있음을 조금이나마 알게 됐다.