Data Lab: Manipulating Bits (Floating Points)

20220127 임유진

이번 Lab 에서는 정수와 부동 소수 데이터의 bit-level 표현에 익숙해질 수 있도록 특정 연산자만을 이용하여 정수와 부동 소수 데이터에 특정한 작업을 수행하는 함수를 구현해보는 것을 목표로 한다. 문제 별 풀이는 다음과 같다.

problem 1 – negate(x)

Problem 1 은 인자로 받은 int 형 데이터 x 에 대해 -x 를 반환하는 negate(int x) 함수를 – 연산자를 사용하지 않고 구현하는 것이다.

Sol)

Two's complement 에서 -x == ~x+1 이므로, 함수가 ~x+1 을 반환하도록 구현하였다.

problem 2 - isLess(x, y)

Problem 2 는 인자로 받은 int 형 데이터 x 와 y 에 대해 x < y 인 경우 1 을, 그렇지 않은 경우 0 을 반환하는 isLess(int x, int y) 함수를 구현하는 것이다.

Sol)

x < y 를 판단하기 위해 x 와 y 의 부호가 같은 경우와 부호가 같지 않은 경우로 나누어 생각해볼 수 있다.

우선 x 와 y 의 부호가 다른 경우 비음수가 음수보다 항상 크다는 것을 통해 x 가 비음수, y 가 음수인 경우 함수가 0 을, 반대의 경우 1 을 반환하도록 구현할 수 있을 것이다.

x 와 y 의 부호가 같은 경우 x – y < 0 이면 x < y 라는 사실을 이용할 수 있지만, x – y 의 연산을 하는 과정에서 오버플로우가 발생하게 된다면 잘못된 결과를 얻을 수 있으므로 이에 대한 확인이 필요하다.

int 형 데이터의 경우 -2^{31} 와 $2^{31}-1$ 사이의 값을 가질 수 있으므로 x 와 y 가 모두 비음수일 때, $0 \le x \le 2^{31}-1,\ -2^{31}+1 \le -y \le 0$ 이고, 따라서 $-2^{31}+1 \le x-y \le 2^{31}-1$ 이다. 다음으로 x 와 y 가 모두 음수일 때는, $-2^{31} \le x < 0,\ 0 < -y \le 2^{31}$이므로, $-2^{31} < x-y < 2^{31}$이 성립한다. 이로부터 x 와 y 의 부호가 같은 경우 x – y 의 연산에서 오버플로우가 발생하지 않음을 확인할 수 있다. 따라서 $x-y = x+(-y) = x+(\sim y+1)$ 의 연산 결과가 음수인 경우 1 을, 비음수인 경우

0 을 반환하도록 하였다. 코드의 구체적인 설명은 아래와 같다.

sign_x_y 변수가 (x >> 31) ^ (y >> 31)의 연산을 통해 x 와 y 의 부호가 같은 경우 00...0, 다른 경우 1...1 의 값을 가지도록 하였고, ~y + 1 의 연산을 통해 -y 의 값을 가지는 negate_y 변수를 사용하였다. int 형 변수 x_sub_y 는 (x + negate) >> 31 의 연산을 통해 x + (-y), 즉 x - y 의 연산 결과가 음수인 경우 11...1 을, 비음수인 경우 00...0 의 값을 가지도록 하였다. 마지막으로 int 형 변수 result 는 (sign_x_y & ~(x >> 31)) | (~sign_x_y & ~x_sub_y)의 연산을 통해 x 와 y 의 부호가 같은 경우 x 가 음수일 때 00...0, 비음수일 때 11...1 의 값을, x 와 y 의 부호가 다른 경우 x - y < 0 일 때 00...0, x - y >= 0 일 때 11...1 의 값을 가지도록 하였다. 마지막으로 !result 를 반환하여 x < y 일 때 1 이, 아닌 경우 0 이 반환되도록 isLess 함수를 구현할 수 있었다.

problem 3 – float_abs(uf)

Problem 3 는 single-precision floating-point 자료형, 즉 float 타입에 대해 절댓값을 계산하는 함수를 구현하는 것을 목적으로 한다. 이때 함수의 인자와 반환 값은 unsigned 이지만 float 의 비트 표현으로 해석하여 연산할 수 있게 float_abs(unsigned uf)를 구현해본다.

Sol)

인자로 주어진 uf 가 Nan 값을 가지는 경우를 확인하여 해당 경우에 uf 를 반환하고, Nan 값이 아닌 경우에 대해서는 uf 가 single-precision floating point 의 비트 표현으로 해석했을 때 음의 값인 경우 부호 비트의 값만 0 으로 바꿔주고, 비음수 값인 경우 uf 를 그대로 반환함으로써 절댓값이 반환되도록 구현할 수 있다. 코드의 구체적인 설명은 아래와 같다.

float 타입에서 exponent field 에 해당하는 비트들이 모두 1, 다른 비트는 0, 즉 0x7f800000 의 값을 가지는 변수 e 와 fractional field 에 해당하는 비트들이 모두 0, 다른 비트는 1, 즉 0xff800000 의 값을 가지는 변수 f 를 사용하였다. uf & (1 << 31)의 연산을 통해 sub 변수가 uf 가 float 의 비트 표현으로 해석했을 때 음수인 경우 MSB 만 1 이고 나머지 비트는 0 인 값을, 비음수인 경우 0 의 값을 가지도록 하였다. 그후 (e & uf) == e(uf 의 float 비트 표현 해석에서 exponent field 에 해당하는 비트가 모두 1)이면서, (f | uf) != f(fractional field 의 비트가 모두 0 이 아닌 경우) ,즉 uf 가 float 의 비트 표현 해석에서 Nan 값인지 확인하여 Nan 값이면 uf 를 그대로 반환하도록 하였다. Nan 값이 아닌 경우에는 uf & ~sub 의 연산을 통해 uf 가 float 비트 표현에서 음수인 경우 부호 비트를 0 으로 바꿔서, 비음수인 경우 uf 가 그대로 반환되도록 float_abs 함수를 구현할 수 있었다.

problem 4 – float_twice(uf)

Problem 4 는 single-precision floating-point 자료형, 즉 float 타입 데이터에 대해 2 를 곱한 값을 계산하는 함수를 구현하는 것을 목적으로 한다. 이때 함수의 인자와 반환 값은 unsigned 이지만 float 의 비트 표현으로 해석하여 연산할 수 있게 float_twice(unsigned uf)를 구현해본다.

Sol)

problem 3 에서와 마찬가지로 0x7f800000 의 값을 가지는 변수 e 와 0xff800000 의 값을 가지는 변수 f 를 사용하였으며, (e & uf) == e 인 경우 Nan 또는 무한대인데 이 경우 Nan 값은 인자 그대로 반환되고, 무한대의 경우 2 배를 하더라도 그대로 무한대이므로 인자를 반환하도록 하였다.

또한 부호, 지수, 비율 부분만 1 나머지는 0 인 값 들과의 &연산을 통해 uf 의 부호, 지수, 비율 부분만 저장하는 변수 uf_s, uf_e, uf_f 의 변수를 사용하였다.

$(-1)^S M 2^E$의 표현에서 Normalized value 에 대해 $1 \leq M < 2$, 즉 $2 \leq 2 * M < 4$ 이므로 2 를 곱한 값에서도 $1 \leq M < 2$이 만족하도록 $(-1)^S M 2^E$의 표현 방식을 사용하기 위해서는 지수 E 가 반드시 1 증가해야 함을 알 수 있고, 이로 인해 M 의 값은 변함없게 된다. 따라서 uf_e 에 0x00800000 을 더하여 지수 부분을 1 증가시켜주었으며, uf_s + uf_e + uf_f 를 통해 f * 2 의 연산 결과를 얻을 수 있다. 또한 exp 가 1111 1110 인 경우 2 를 곱해주게 되면 float 에서 표현할 수 있는 범위를 벗어나 무한대가 되도록 처리하였다.

Denormalized value 에 대해서는 $0 \leq M < 1$, 즉 $0 \leq 2 * M < 2$ 이다. 이때 $0 \leq 2 * M < 1$ 인 경우 지수 E 가 증가하지 않아도 되므로 fractional field 는 기존에서 한 비트 left shift 가 이루어지면 된다. $1 \leq 2 * M < 2$인 경우에는 Normalized value 의 범위 안으로 들어가$(-1)^S M 2^E$에서 M 이 $1 \leq M < 2$을 만족해야 하고, exp 가 1 로 증가하게 되지만 exp 가 0 일 때와 1 일 때 지수 E 의 값은 같으므로 fractional field 는 기존에서 한 비트 left shift 가 이루어지면 된다. 따라서 uf_f 를 한 비트 left shift 해주었으며, 이때 $1 \leq 2 * M < 2$이 되는 경우 uf_f 에서 지수 부분의 LSB 를 나타내는 비트가 1 이 되는데, 이는 uf_s + uf_e + uf_f 의 연산에서 결국 exp 를 1 증가시켜준 효과로 나타남을 알 수 있다. 따라서 마찬가지로 uf_s + uf_e + uf_f 를 통해 f * 2 의 연산결과를 얻을 수 있다.

problem 5 – float_i2f(x)

Problem 5 는 인자로 주어진 int 형 데이터에 대해 float 형으로 타입 캐스팅한 결과를 반환하는 함수를 구현하는 것을 목적으로 한다. 타입 캐스팅한 결과의 float 형 비트 표현을 가지는 unsigned 값을 반환하도록 float_i2f(int x)를 구현해본다.

Sol)

floating-point 타입에서는 음수가 같은 절댓값을 가지는 비음수형 데이터에 대해 부호 부분의 비트만 1 로 바뀐 형태로 나타난다. 따라서 int 형 데이터를 floating point format 으로 전환하기에 앞서 절댓값을 먼저 구해주어 absval 와 absval2 변수에 저장하였다. 또한 int 타입 x 를 float 으로 타입 캐스팅했을 때 각각 부호, 지수, 비율 부분의 값을 저장하고 나머지 부분에 대해서는 0 의 값을 갖는 x_s, x_e, x_f 의 변수를 사용하고, 마지막으로 x_s + x_e + x_f 를 통해 (float) x 를 구할 수 있도록 하였다.

우선 int 타입의 데이터 x 를 $(-1)^S M 2^E$과 같은 format 으로 전환하기 위해 while 문을 이용해 binary point 를 왼쪽으로 shift 하면서 1.xxx...의 형태를 가질 때까지 E 를 1 씩 증가시키는 연산을 수행하였다. 이렇게 얻은 E 값에 대해 absval 을 32 – E 만큼 left shift 시켜 32bit 의 fractional part 를 구하였다. 이때 Rounding 을 위해 0xff & absval, 0x100 & absval 을 통해 sticky 와 round 의 값을 각각 구하였고, 그후 absval 을 다시 9 bit 만큼 right shift 하여 23 bit 의 Rounding 하기 전 fractional field 의 값을 구하였다. round bit 가 1 인 경우에 대해 sticky 가 0 인 경우 Round to even 을 위해 x_f 의 LSB, 즉 Guard bit 가 1 인 경우에 대해서만 1 이 더해 Guard bit 를 증가시켰고, sticky 가 0 이 아닌 경우엔 1 을 더해 Guard bit 를 증가시켜주었다. 이때 오버플로우가 발생할 수 있지만 오버플로우가 발생하는 경우 x_f 에서 지수 부분의 LSB 가 1 이 되고, fractional field 는 모두 0 임을 고려할 때 x_s + x_e + x_f 의 연산을 통해 exponent 가 1 증가하고 Rounded 값이 1 bit right shift 된 결과를 얻게 됨을 알 수 있다. 따라서 오버플로우의 여부와 상관없이 x_s + x_e + x_f 연산을 통해 (float) x 의 비트 표현을 가지는 unsigned 데이터를 얻을 수 있다.

problem 6 – float_f2i(uf)

Problem 6 은 single-precision floating-point 자료형, 즉 float 형 데이터에 대해 int 형으로 타입 캐스팅한 결과를 반환하는 함수를 구현하는 것을 목적으로 한다. 이때 함수의 인자는 unsigned 타입이지만 float 의 비트 표현으로 해석하여 연산할 수 있게 float_i2f(unsigned uf)를 구현해본다.

Sol)

uf & 0x80000000, uf & 0x7f800000, uf & 007fffff 의 연산을 통해 인자로 받은 uf 를 float 의 비트 레벨 표현으로 해석했을 때 각각 부호, 지수, 비율 부분의 값만을 저장하는 uf_s, uf_e, uf_f 의 변수를 사용하였다. 그리고 $(-1)^S M 2^E$ format 에서의 E 값을 나타내는 변수 E 와 int 로 전환한 절댓값을 나타내는 변수 val 을 저장하였다.

우선 uf_e 가 0x7f800000(지수 부분이 모두 1), 즉 uf 가 Nan 또는 infinity 일 때 0x80000000u 를 반환하도록 하였다. 그리고 만약 uf_e 가 0(지수 부분이 모두 0)이라면 E 의 값을 1-bias 의 값인 0xffffff82 로 해주었다. 나머지 경우, 즉 uf 가 Normalized value 를 나타내는 경우에는 (uf_e >> 23) + 0xffffff81(exp – bias)을 E 의 값으로 해주었다.

그렇게 구한 E 가 만약 0 보다 작은 경우 0 으로 캐스팅되므로 uf_f 를 0 으로 해주었으며, 0 이상 23 이하의 값을 가지는 경우와 24 이상 31 미만의 값을 가지는 경우에 대해 (1 + f) * 2^E 의 값을 계산하기 위해 각각 (uf_f + 0x00800000) >> (23 – E), (uf_f + 0x00800000) << (E – 23)의 계산을 수행해 val 에 저장하였다. E 가 31 이상인 경우는 -2^{31} 인 경우를 제외하고는 int 의 표현 범위를 넘어서는 수가 되지만 -2^{31} 의 16 진수 표현은 0x80000000 이므로 E 가 31 인 이상일 때는 Nan 또는 infinity 일 때와 마찬가지로 0x80000000u 를 반환한다. 마지막으로 val 값은 int 로 전환한 절댓값을 나타내게 되므로, 부호에 맞게 올바르게 나타내기 위해 ((uf_s >> 31) & (~val + 1)) | (~(uf_s >> 31) & val)의 연산을 수행하여 반환해주도록 float_f2i 를 구현하였다.