**CSED211: Data Lab1 Report**

20230499

1. **Overview**

이번 랩 2 과제에서는 C언어를 이용해 정수 및 실수의 비트 표현을 연습하는 것을 연습하기 위해 6가지 문제를 해결하는 것이 목표이다. negate(int), isLess(int, int), float\_abs(unsigned), float\_twice(unsigned), float\_i2f(int), float\_f2i(unsigned) 총 6가지의 함수를 구현하여야 하며, 실수 자료형 대신 비트 연산을 적용하기 위해 unsigned 자료형을 사용한다.

1. **Question #1. negate**

2-1. Explanation.

* 해당 함수는 int형 정수를 받아, 양수일 경우 음수, 음수일 경우 양수로 바꾸어 반환하는 함수이다. 즉, int형 정수 x를 받아 -x 를 반환하여야 한다.
* 제약 조건: !, ~, &, ^, |, +, <<, >> 연산자만 최대 5회 사용할 수 있다.

2-2. Solution

* Int 형 정수의 2의 보수를 음수로 취급한다. –(–x) = x임과 같은 원리를 적용하면 음수의 2의 보수는 양수(절댓값)이 된다. 즉, -x는 x의 2의 보수를 계산함으로써 계산할 수 있다.
* 따라서 주어진 정수의 2의 보수를 계산하여 반환하여야 한다.
* 2의 보수를 구하는 방법은 다음과 같다.
  1. 모든 비트의 0과 1을 반전시킨다.
  2. 반전시킨 값에 1을 더한다.

2-3. Implementation.

* 위 해답과 같이, C언어를 이용하여 Listing 1과 같이 구현하였다.

Listing 1. Code of negate.

**negate(x){  
 return ~x+1;**

**}**

* ~ 연산자를 이용해 부호를 반전시킨 후, 1을 더하였다. 총 2개의 연산자를 이용해 구현할 수 있다.

1. **Question #2. isLess**

3-1. Explanation.

* 해당 함수는 두 int형 정수 x와 y 를 입력 받은 후, x보다 y가 크면 1, 아니면 0을 반환하는 함수이다.
* 제약 조건: !, ~, &, ^, |, +, <<, >> 연산자만 최대 24회 사용할 수 있다.

3-2. Solution.

* 수학적으로는 x – y가 음수인 경우, y가 더 큼을 확인할 수 있다.
* 그러나, 32비트를 이용하는 int 자료형의 한계로, x – y를 계산함에 있어서 오버플로우나 언더플로우가 발생할 가능성이 있다. 예를 들어, x = 2147483647, y = -10인 경우 x – y의 값은int형의 TMAX 값보다 커지게 된다.
* 이를 해결하기 위해서 Lab1 과제에서 오버플로우 / 언더플로우를 확인하는 함수에서 사용한 방법과 비슷한 방법을 사용할 수 있다. 오버플로우 등이 발생할 수 있는 경우와, 그렇지 않은 경우를 나누는 방법이다.
* x와 y의 부호가 같은 경우, 빼기 연산에서 오버플로우, 언더플로우가 발생할 가능성이 없다. 따라서 둘의 부호가 같다면 x – y를 직접 계산하여 대소관계를 확인한다. 문제가 발생할 가능성이 있는 경우, 즉 x와 y의 부호가 다른 경우에서는 x – y를 계산하여 비교하는 것 대신 둘의 부호를 직접 비교하여 대소관계를 확인할 수 있다. y가 양수이고 x가 음수인 경우가 이에 해당된다.
* 두 경우를 각각 계산하여 True(1), False(0)으로 표현한 후, OR 연산을 통해 x < y인지 결론지을 수 있다.

3-3. Implementation.

* 위 해답과 같이, C언어를 이용하여 Listing 2 와 같이 구현하였다.

Listing 2. Code of isLess.

**isLess(x,y){  
 // Two has different sign bit and y's sign bit is 0 (or x's sign bit is 1)**

**int sx = !!(x >> 31);**

**int sy = !!(y >> 31);**

**int If\_signbits\_are\_different = sx ^ sy;**

**int Case1\_Different = If\_signbits\_are\_different & !sy;**

**// Two has same sign bit and x - y is negative**

**int d = x + (~y+1);**

**int sd = !!(d >> 31);**

**int Case2\_Same = (!If\_signbits\_are\_different) & sd;**

**return Case1\_Different | Case2\_Same;**

**}**

* 처음 두 줄에서는 x와 y의 부호비트를 계산한다. 음수면 1, 양수나 0인 경우 0이다.
* 3번째 줄에서, x와 y의 부호비트가 다른지 확인한다.
* **4번째 줄에서 첫 번째 경우인 x와 y의 부호가 다른 경우에서 x < y인지 확인한다. 두 정수의 부호가 다르면서 y의 부호비트가 1 이 아닌 (y가 양수 또는 0, 따라서 x가 음수인) 경우 참이다.**
* 코드의 5~6번째 줄에서는 x – y와 차이의 부호비트를 계산한다.
* **7번째 줄에서 두 번째 경우에서 x < y인지 확인한다. 두 정수의 부호가 같으면서 x – y가 음수인 경우(부호비트가 1인 경우) y가 x보다 크다. 즉, 참이다.**
* 마지막으로, 두 경우를 OR 연산한 후 결과를 반환한다. 둘 중 하나가 참이거나, 둘 모두 거짓인 경우만 가능하므로 OR 연산하여 하나가 참인 경우 x < y인 경우를 확인할 수 있다.
* 총 18개의 연산자를 사용하여 구현하였다.

1. **Question #3. float\_abs**

4-1. Explanation.

* 해당 함수는 실수를 받아, 실수의 절댓값을 반환하는 함수이다.
* 제약 조건: 모든 상수와 연산자를 사용할 수 있다. AND(&&), OR(||), if, while 또한 사용할 수 있다.

4-2. Solution.

* Int형 정수에서의 절댓값 계산은, 정수가 음수일 경우 2의 보수를 계산하는 과정이 필요하였다.
* 그러나, float형 실수에서는, 음수를 표현하기 위해 2의 보수와 같은 방법을 적용하지 않는다. 32비트 실수형의 경우 부호비트 1개, 지수부(e) 비트 8개, 가수부(f) 비트 23개로 구성된다. 지수부 비트와 가수부 비트만을 이용해 절댓값을 계산하며, 그 수가 양수인지 음수인지는 부호 비트에 따라 결정된다. 즉, int형과는 다르게 부호비트만 0으로 만들면 절댓값을 계산할 수 있다.
* 실수 타입이 표현하는 값에 대한 수식을 보면 쉽게 이해할 수 있다. 이다. 여기서 s, M, E는 각각 부호비트, Significand, Exp – bias 값으로, M과 E의 값에 따라 절댓값을 계산할 수 있으며, 이때 부호비트와 관련되지 않는다.
* 단, 실수 표현 중 NaN인 경우는 매개 변수를 그대로 반환해야 하는 조건이 있다. 지수부(e) 비트가 모두 1이면서 가수부(f)가 모두 0이 아닌 경우가 이에 해당한다.
* 따라서, NaN인지 확인한 후 NaN이라면 그대로 반환, 아니라면 첫번째 비트만 0이고 나머지는 1로 채워진 mask와 AND(&) 연산하여 반환하면 된다.

4-3. Implementation.

* 위 해답과 같이, C언어를 이용하여 Listing 3과 같이 구현하였다.

Listing 3. Code of float\_abs.

**float\_abs(uf){  
 if ((uf & 0x7F800000) == 0x7F800000 && (uf & 0x007FFFFF) != 0) {**

**return uf;**

**}**

**return uf & 0x7FFFFFFF;**

**}**

* 0x7FFFFFFF는 0이 1개, 1이 31개 이어져 만들어진 부호비트 제외 마스크이다.   
  (0 – 11111111 – 11111111111111111111111)
* 0x7F800000는 0이 1개, 1이 8개, 0이 23개 이어져 만들어진 지수부 마스크이다.  
  (0 – 11111111 – 00000000000000000000000)
* 0x007FFFFF는 0이 9개, 1이 23개 이어져 만들어진 가수부 마스크이다.   
  (0 – 00000000 – 11111111111111111111111)
* 각 마스크를 실수에 AND(&)연산하면 마스크에서 비트가 1인 부분만을 각각 얻을 수 있다.
* 즉, uf와 지수부 마스크를 적용한 값인 지수부 부분이 모두 1이면서, 가수부 마스크를 적용한 가수부 부분 중 하나라도 1이 존재한 경우 NaN임을 판단하고 uf를 그대로 반환한다.
* NaN이 아니라면, uf에 부호비트를 제외하는 마스크를 적용함으로써 절댓값을 계산할 수 있다.

1. **Question #4. float\_twice**

5-1. Explanation.

* 해당 함수는 실수를 받아, 실수의 두배를 계산하여 반환하는 함수이다.
* 제약 조건: 모든 상수와 연산자를 사용할 수 있다. AND(&&), OR(||), if, while 또한 사용할 수 있다.

5-2. Solution.

* 실수의 두배를 계산하기 앞서, 몇 가지 예외를 고려해서 경우를 나누어야 한다.
* 만약 입력 값이 0이거나 -0인 경우, 그대로 반환한다.
* 입력 값이 NaN이거나 Inf인 경우, 그대로 반환한다.
* 입력의 지수부가 0이고 가수부가 0이 아닌 경우, 즉 denormalized number인 경우에는 부호 비트를 유지한 상태로 왼쪽으로 한 번 시프트 연산 처리함으로써 두 배를 계산할 수 있다. 이때 가수부는 1.xx… 형태의 이진법 소수 형태이므로, 왼쪽으로 시프트 연산을 하면 2배가 된 것 과 같다. 이는 10진법으로 생각할 때 0.09를 왼쪽으로 한번 시프트한 값인0.9가 0.09의 10배인것과 같은 원리이다.
* 그 이외의 나머지 경우에서는 지수부와 가수부가 모두 존재하는 실수이므로, 실수 값 계산 식 에 따라 지수부(E)에 1을 더해줌으로써 두 배를 계산할 수 있다.

5-3. Implementation.

* 위 해답과 같이, C언어를 이용하여 Listing 4와 같이 구현하였다.

Listing 4. Code of float\_twice.

**float\_twice(uf){  
 if(uf == 0 || uf == 0x80000000) return uf;**

**if((uf & 0x7F800000) == 0x7F800000) return uf;**

**if((uf & 0x7F800000) == 0x00000000) return (uf & (1<<31)) | (uf<<1);**

**return uf + (1 << 23);**

**}**

* 첫 번째 조건식은 입력 값이 0 또는 -0인지를 검사한다.
* 두 번째 조건식은 NaN또는 Inf인지 검사한다. 두 경우 모두 지수부가 1인 경우 발생하며 지수부가 모두 1일 때에는 오직 두가지 상태만 가능하므로, 지수부 비트가 모두 1인지만 확인하면 된다.
* 세 번째 조건식은 입력 값이 Denormalized된 상태인지 검사한다. 이 때에는 왼쪽으로 시프트 연산하며, 입력 값의 부호비트를 계산하여 OR처리함으로써 부호 상태를 유지한다.
* 그 이외의 경우에는 일반적인 소수이므로, 지수부에 1을 더해줌으로써 2배를 계산한다. 뒤의 23자리는 가수부이므로, 지수부에 1을 더하기 위해서 (1 << 23) 값을 더해준다.

1. **Question #5. float\_i2f**

6-1. Explanation.

* 해당 함수는 int형 정수를 받아, 32비트 float형 실수로 변환하여 반환하는 함수이다.
* 제약 조건: 모든 상수와 연산자를 사용할 수 있다. AND(&&), OR(||), if, while 또한 사용할 수 있다.

6-2. Solution

* 크게 네 가지 순서으로 나눌 수 있다.
* 첫 번째는 예외 처리로, 입력 값이 0이나 -2147483648인 경우 각각 0, 0xfc000000을 반환한다
* 두 번째는 부호비트 계산으로, 부호 비트 마스크 (0x80000000)를 이용하여 부호 비트를 계산한다. 이때 입력이 음수면 양수로 바꾸어서 계산하는데, 이는 실수의 음수 저장 방식이 2의 보수와 같은 방식이 아니라 부호 비트에 따라 부호가 결정되기 때문이다.
* 세 번째는 지수부 계산으로, 입력 값이 2로 몇 번 나누어지는지 확인한 후, bias값을 더해 지수부를 얻을 수 있다.
* 네 번째는 가수부 계산으로, 정수를 float 형으로 오차 없이 표현할 수 있는지 없는지에 따라 처리 방식을 달리 해야 한다. 오차 없이 표현할 수 있다면 그대로 사용하고, 오차가 생길 수밖에 없다면 Round-to-even 규칙에 따라 반올림한다.
* **구체적인 설명은 Implementation 항목에 서술하였다.**

6-3. Implementation

* 위 해답과 같이, C언어를 이용하여 Listing 5와 같이 구현하였다.
* ***코드가 지나치게 길어, 해당 항목에 기재하는 것이 아니라 본 보고서 제일 마지막 페이지에 따로 작성하였다.***
* 정수를 실수로 변환하기 위해서 총 4가지 부분과 3번의 마킹으로 나누어 작성하였다. 마킹이란 반환 값을 저장하는 변수인 result에 답을 반영하는 과정을 말하며, 코드에 각각 part. n기호와 Marking #n 기호를 이용하여 주석으로 표시해 두었다.
* 첫 번째 부분에서는 예외를 처리한다. 만약 입력이 0인경우 그대로 0을 반환하며, int형의 TMIN값이 입력된 경우 미리 정의된 상수인0xcf000000를 반환하도록 처리하였다.
* 두 번째 부분에서는 부호를 처리한다. 입력 값 x에 부호 마스크인 0x80000000를 씌워 부호비트를 얻은 후, x가 음수라면 양수로 바꾸어 지수부 및 가수부 계산을 준비한다. 해당 부분에서, result에 부호비트를 미리 반영해 둠으로써 부호 비트를 유지한다. (Marking #1)
* 세 번째 부분에서는 지수를 계산한다. 입력 값을 2로 몇 번 나눌 수 있는지 계산하며, bias와 더해 지수부를 계산할 수 있다.
* 네 번째 부분에서는 가수부를 계산한다. 입력된 정수를 오차 없이 정확하게 표현할 수 있다면, 즉 지수부의 값이 23이하라면 가수부를 그대로 왼쪽으로 시프트 하여 저장한다. (Marking #2-1)  
   그러나, 정확하게 표현할 수 없다면 반올림 처리한다. 이때 Round-to-even규칙에 따라 처리를 해야 한다. (Marking #2-2)
* 코드 중에서 right\_shift\_bits는 가수부 표현을 위한 시프트 비트 수, rounding\_bit\_pos는 반올림할 때 고려할 마지막 비트 위치를 나타낸다. 해당 값을 이용해 ev\_23, 24, 25를 계산하는데, 이는 각각 가수부의 마지막 비트, 마지막 다음인 반올림 대상 비트, 그리고 그 뒤인 25번째 이상의 부분이 0인지 아닌지 확인하는 비트이다. 해당 비트를 조합하여 Round-to-even 규칙을 적용할 수 있다. 만약 반올림 대상 비트가 1인 경우 (ev\_24가 1인 경우) 반올림을 수행하려고 시도하며 그 전 비트(ev\_23)과 그 이후 비트들(ev\_25)의 값에 따라 반올림을 수행하여 짝수로 반올림한다.
* 마지막으로, 반환 값에 지수부를 반영한 후 반환한다. (Marking #3)

1. **Question #6. float\_f2i**

7-1. Explanation.

* 해당 함수는 32비트 float형 실수를 받아, int형 정수로 변환하는 반환하는 함수이다.
* 제약 조건: 모든 상수와 연산자를 사용할 수 있다. AND(&&), OR(||), if, while 또한 사용할 수 있다.

7-2. Solution.

* 크게 3가지 부분으로 나눌 수 있다.
* 첫 번째는 입력 받은 실수의 부호비트, 지수부, 가수부를 분리하여 저장한다.
* 두 번째는 예외처리로, 수가 너무 작거나 큰 경우 지정된 값을 반환한다.
* 세 번째 부분에서는 실제 정수를 계산한다.
* 마지막으로, 가수부에 암묵적으로 존재하는 값을 더해준 후, 부호를 반영하여 반환한다.
* **구체적인 설명은 Implementation 항목에 서술하였다.**

7-3. Implementation.

* 첫 번째 부분에서는 세 부분을 분리하여 저장한다. 부호비트는 31번, 지수부는 0x7F800000로 마스킹한 후 23번, , 가수부는 0x007FFFFF로 마스킹하여 각 비트를 분리할 수 있다.
* 두 번째 부분은 예외처리이다. 지수부가 bias보다 작은 경우는 (은 자연수) 값이므로, 이는 정수 표현에서 매우 작은 소수 값이므로 0을 반환하여 처리한다. 반대로, 지수부에서 bias를 뺐을 때 31보다 크거나 같은 경우, 최소 이상이므로 int 범위로 표현할 수 없다. 수가 Inf거나 NaN인 경우에서도 지수부 비트가 11111111, 즉 255는 31이상이므로 이 또한 처리할 수 있다..따라서, 문제에서 지정한 대로 0x80000000u를 반환한다.
* 세 번째 부분에서는 실제 정수를 계산한다. **앞서 구한 가수부(fraction, mantissa) 비트를 그대로 int로 변환한다면, 입력된 실수의 꼴이 된다**. 예를 들어, 가수부가 1 두개, 0이 21개로 이루어진 11000000000000000000000라고 생각해보자. 그러면 해당 실수의 가수부의 Significand는 1.11(2), 1.75(10)이 된다. 해당 가수부를 비트를 유지한 채 정수형으로 취급한다면, 해당 값은 이므로, 이는 와 같다. 따라서, 앞서 구한 지수부가 23이라면 가수부 비트를 그대로 정수형으로 변환한 값이 결과 값이다. 즉, 지수부가 23이라면 가수부의 비트를 그대로 int형으로 취급하는 방법으로 값을 얻을 수 있고, 만약 지수부가 23보다 크거나 23보다 작은 경우에는 차이나는 만큼 2를 곱하거나 나누어 줌으로써 값을 얻을 수 있다.
* 앞서 수식에서 1을 뺀 이유는 다음과 같다. 가수부의 범위는 1보다 크거나 같고 2보다 작은 값을 나타내기로 정의되어 있으므로, 가수부 비트는 소수 부분만 표현하며 1은 암묵적으로 표시하지 않기로 정해져 있기 때문이다. 따라서 그 부분을 반영하기 위해서 만큼 더해준 후 부호를 처리하여 반환하였다.
* 위 해답과 같이, C언어를 이용하여 Listing 6와 같이 구현하였다.

Listing 6. Code of float\_f2i.

**float\_f2i(uf){**

**unsigned sign, exp, frac, bias = 127, result;**

**// part 1. calculating sign, exp, frac**

**sign = uf >> 31;**

**exp = (uf & 0x7F800000) >> 23;**

**frac = (uf & 0x007FFFFF);**

**// part 2. handling exception. Too small (including denormalized) and too large including not number**

**if(exp < bias) return 0;**

**if(exp - bias >= 31) return 0x80000000u;**

**// part 3. calculating integer.**

**// frac is (Mantissa - 1) \* 2^23 form already, so,**

**// if exp-bias is bigger than 23, multiply remain 2s**

**// if exp-bias is smaller than 23, divide inordinate 2s**

**exp -= bias;**

**if(exp > 23) result = frac << (exp - 23);**

**else if(exp < 23) result = frac >> (23 - exp);**

**// adding (1 << exp), because Mantissa is 1.xxx, but 1 is not represented in bits.**

**result += 1 << exp; // (2 ^ exp)**

**if(sign) result = -result;**

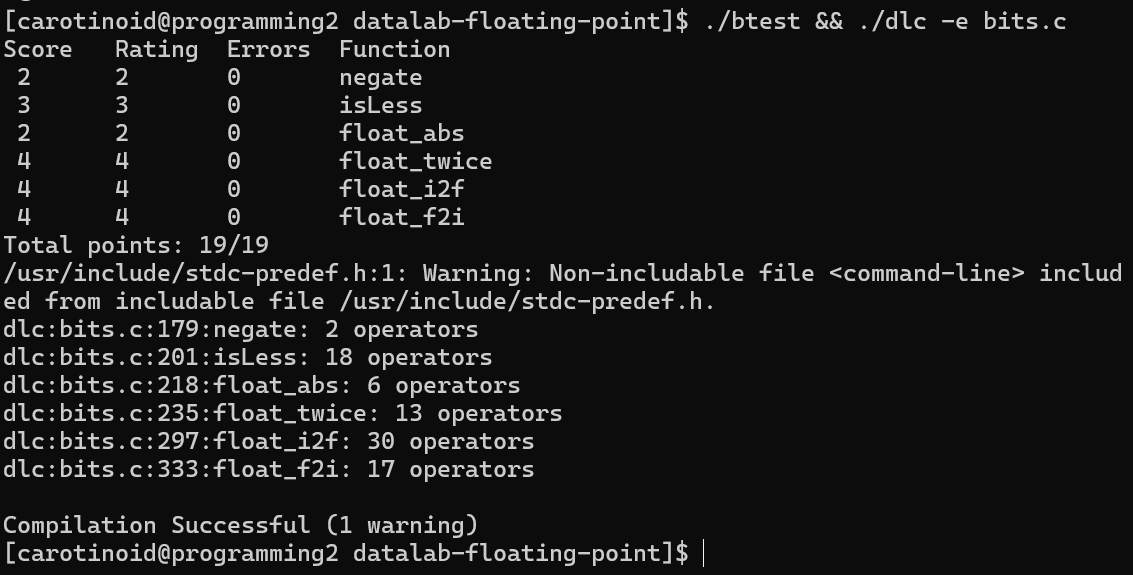
**return result;**

**}**

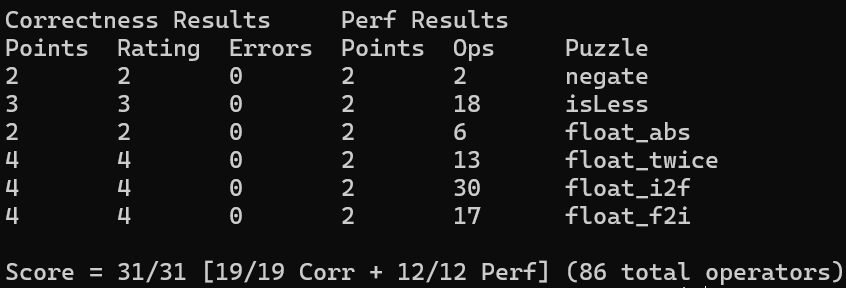
1. **Results**

Figure 1과 btest와 dlc의 결과를 나타내며, Figure 2는 driver.pl의 실행 결과를 보여준다. 둘 모두 감점 받지 않았으며, 총86개의 연산자를 사용하였다.

[Figure 1.]



[Figure 2.]



1. **References**

Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e), Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University (<https://csapp.cs.cmu.edu/3e/labs.html>)

1. **Attachment**

**float\_i2f(x){  
int sign, temp, bias = 127, exp = 127, mask;**

**int mantissa, shift\_amount, right\_shift\_bits, rounding\_bit\_pos, rounding\_mask, ev\_25, ev\_24, ev\_23, rounding, E;**

**unsigned result;**

**// part 1. handling exceptions.**

**if(x == 0) return 0;**

**if(x == 0x80000000) return 0xcf000000;**

**// part 2. calculating sign bit.**

**// Now, we think about only tx, tx is always positive.**

**sign = x & (0x80000000);**

**if(sign) x = -x;**

**/\*Marking #1\*/**

**result = 0 | sign;**

**// part 3. calculating Exponential**

**temp = x;**

**while(temp /= 2) exp ++;**

**E = exp – bias;**

**mask = (1 << E) - 1; // exp mask**

**// part 4. calculating Mantissa**

**mantissa = mask & x;**

**shift\_amount = 23 - E;**

**// Continue to next page**

Listing 5. Code of float\_i2f.

**// Cont’d**

**if(E <= 23) {**

**// part 4-1. if original (int) x can be represented in 23-bit without rounding.**

**// Just marking.**

**/\*Marking #2-1\*/**

**result += (mantissa << shift\_amount);**

**}**

**else {**

**// part 4-2.**

**right\_shift\_bits = - shift\_amount;**

**rounding\_bit\_pos = right\_shift\_bits - 1;**

**rounding\_mask = 1 << rounding\_bit\_pos;**

**ev\_23 = mantissa & (1 << right\_shift\_bits);**

**ev\_24 = mantissa & (rounding\_mask);**

**ev\_25 = mantissa & (rounding\_mask - 1);**

**rounding = mantissa >> right\_shift\_bits;**

**rounding += (ev\_24 && (ev\_25 || ev\_23));**

**/\*Marking #2-2\*/**

**result = result | rounding;**

**}**

**/\*Marking #3\*/**

**result += (exp << 23);**

**return result;**

**}**