|  |
| --- |
| 1. Bitxor 연산 |
| 먼저 Xor 연산은 두 개의 피 연산자가 다를 경우에만 1을 리턴한다.  그래서 &를 통해서 두 개의 피 연산자의 비트표현에서 둘다 1인 자리들을 1로 지정 후 나머지를 0으로 표현한 pos를 구한다  그 후 두 개의 피 연산자에 ~를 취하여 0인 비트를 1로 1인 비트를 0으로 반전시킨 후 & 연산자를 통해 기존의 피 연산자들의 비트표현에서 둘 다 0인 자리를 1로 지정 후 나머지를 0으로 표현한 neg를 구한다  그러면 ~pos는 피 연산자들의 비트표현에서 둘 다 1인경우 제외하고 모두 1이다.  또한 ~neg는 피 연산자들의 비트표현에서 둘 다 0인경우 제외하고 모두 1이다.  그러면 ~pos & ~neg 를 취하면 둘다 1인경우, 0인경우 제외하고 모두 1이 된다. 즉 xor연산이 실행이 된다. |
| 1. tmin ( tmin 출력 ) |
| Tmin은 비트표현으로 [100…000] 으로 표현이 가능하다.  또한 int의 크기는 32비트이니 1을 왼쪽으로 31번 시프트를 수행하면 tmin의 비트표현을 구할 수 있다. |
| 1. Istmax ( tmax 인지 아닌지 확인 ) |
| 입력 받은 값이 tmax인지 아닌지 논리 연산을 실행하면 된다.  먼저 논리연산의 특징은 0이 아닌 인자들은 참 = 1, 0인 인자들을 거짓 =0으로 리턴을 한다.  즉 조건에서는 tmax이면 1, 아니면 모두 0을 리턴을 해야 한다.  Tmax의 비트표현을 생각해보면 +1을 하면 [100..000]이 되고 이것을 2배하면 오버 플로우가 발생하여 [000…000]이 된다. 하지만 이 방식은 -1 [111…111] 에서도 적용이 된다.  그러면 이제 tmax와 -1만 구분을 하면 된다. -1은 ~연산을 취하면 모든 비트가 0이된다. ( 다른 숫자들은 무조건 비트표현에서 최소한 한 개의 1인비트가 존재 )  이 두 개의 특징을 사용해 리턴 값을 생각해보자.  먼저 다른 숫자들로부터 tmax와 -1을 구분하는 방법은 x+x+2를 사용하면 된다. 이 식을 1번식이라 하자.  이때 입력 받은 수가 tmax또는 -1일 경우에는 0을 반환하고 나머지경우에는 1을 반환한다.  그리고 -1을 구분하는 방법은 !(~x)을 사용하면 된다. 이 식을 2번식이라 하자  이때 입력 받은 수가 -1일 경우에는 1을 아닐 경우에는 0을 반환한다.  즉 두 가지 모든 조건을 만족시키며 1을 1번식부정 & 2번식부정 을 연산하면 된다.  연산 횟수를 줄이기 위해 (1번식 | 2번식 )전체부정을 사용하면 결과는 같다. |
| 1. Alloddbits ( 홀수번째 비트가 모두 1인지 아닌지 확인 ) |
| 중요한 건 홀수 번째 비트 묶음이니 홀수 번째 비트만 고려해보자.  홀수 번째 비트 중 임의의 한 비트가 0이라고 가정해보자  그리고 입력 받은 변수를 오른쪽으로 16번 시프트 한 것을 shift1이라고 하자  그러면 shift1과 입력 받은 변수를 &연산을 실행하면 0번째부터 15번째비트 사이의 홀수 번째 비트에 0인 비트가 생기게 된다. ( 만약 가정에서 31번째 비트가 0일경우에는 16번쨰부터 31번째비트 까지 모두 0이 된다. )  방금 수행한 연산의 결과를 가지고 또 오른쪽으로 8번 시프트 한 것을 shift2이라고 하자 그 후 방금 수행한 연산의 결과와 shift2를 &연산하면 0번째부터 7번째 사이의 홀수 번째 비트에 0인 비트가 생기게 된다. ( 만약 가정에서 31번째 비트가 0일 경우에는 8번째부터 31번쨰 비트까지 모두 0이 된다. )  이러한 연산을 계속 반복하면  결국 1번쨰 비트와 0번째 비트 사이의 홀수 번째 비트에 0일 비트가 생기게 된다. 즉 1번째 비트가 0이된다.. ( 만약 가정에서 31번째 비트가 0일경우에는 모든 비트가 0 )  이후 마지막 결과를 오른쪽으로 1번 쉬프트 한 결과와 [00000….0001]을 &연산 취하게 되면 결과는 0이 된다.  이러한 가정을 통해 홀수 번째 비트가 모두 1이면 결과값은 1이 나오게 된다. |
| 1. Int negative ( -x 리턴하기 ) |
| 2의 보수에서 덧셈에 대한 역원을 구하기 위해서는 입력 받은 x에 ~연산을 취한 후 +1을 하면 된다. |
| 1. isAsciiDigit ( 아스키 코드 값이 0x30 ~ 0x39 이면 1리턴하는 함수 ) |
| 아스키 코드 값이 0x30~0x39임을 구분하기 위해서는 먼저 입력 받은 수의 비트 표현이 0x3@이 인지 확인해보자  이를 위해 thirty = 0x30으로 두고 입력 받은 수 x와 thirty를 ^연산을 실행해보자.  그럼 조건이 참이면 연산의 결과는 0번째부터 3번쨰 비트를 제외한 나머지 비트들은 모두 0으로 변하게 될 것이다. 이것을 check1로 두자  그럼 이제 0번째부터 3번째 비트가 0~9사이의 값을 표현하고 있는지 확인하자  즉 비트표현이 0000, 0001, … 0111, 1000, 1001 이중 하나가 되어야 한다.  이때 check1과 &~0x7연산을 수행하면 이 값은 2번째 바이트가 0x3이고 1번째 바이트가 0x0~0x7인 경우에만 0을 반환한다 이것을 check2라 하자 이제 남은 것은 첫 번째 바이트가 1000과 1001경우를 고르는 것 이다.  또 check1과 &~0x9연산을 수행하면 이 값은 2번쨰 바이트가 0x3이고 1번째 바이트의 형태가 \_00\_인 경우에만 0을 반환한다. 즉 1000과 1001인 경우가 포함된다 또한 \_00\_일 경우 표현 가능한 비트 표현은 0000, 0001, 1000, 1001 이므로 모두 0x30 ~ 0x39 사이에 포함된다. 이것을 check3이라 하자  즉 결과를 !check2 | !check3 = (check2 & check3)! 으로 설정하면 된다. |
| 1. Conditional ( x ? y : z ) 와 같은 함수를 만들어라 |
| X가 0이 아니면 y를 0이면 z를 출력하는 함수이다.  X가 0인지 아닌지 확인하기 위해서는 !x를 사용한다. !x를 통해서 0이면 1을 아니면 0을 x값에 저장한다.  그리고 구한 결과를 왼쪽으로 31번 시프트 후 오른쪽으로 31번 시프트 하면 32개의 비트가 모두 1또는 0이 된다. 이렇게 구한 것을 또 x에 저장 하면 된다.  지금 까지 과정을 요약하자면 x가 0이 아니면 위의 연산을 통해 x의 비트표현은 0000..0000 이고 0이면 x의 비트표현은 1111…..11111이 된다.  이때 ~x&y를 생각해보면 x가 0이 아닐 때 연산 결과는 y가 되고 0일떄는 0이 된다.  또한 x&z를 생각해보면 x가 0이 아닐 때 연산 결과는 0이 되고 0일때는 z가 된다.  이러한 특징을 사용하여  반환 값을 ~x&y | x&z 라고 설정하면 x가 0이 아닐 때는 y 0일때는 z를 반환한다. |
| 1. Islessorequal ( x <=y 일 경우 1을 반환 아닐 경우 0을 반환 하는 함수 ) |
| 일반적으로 x-y가 음수 또는 0이면 x<=y는 참이다 하지만 중요하게 생각해봐야 할 것은 오버 플로우의 발생이다 만약 x=tmax 이고 y가 -1이면 x-y=tmin으로 x-y <=이 된다.  즉 우리가 확인해야 할 조건은 x,y의 부호에 따른 x-y의 부호이다.  경우 1 y>x>=0 -> x-y <0 즉 x,y의 31번비트 0 그리고 x-y 31번 비트 1  경우 2 y>=0>x -> x-y < 0 즉 x의 31번비트 1, y의 31번 비트 일 경우는 trivial  경우 3 0>y>x -> x-y >=0 즉 x,y의 31번 비트 1 그리고 x-y 31번 비트 1  경우 4 x=y -> x-y =0 x-y의 모든 비트 0  위 경우들을 정리해보면 x,y의 31번 비트가 같으면 x-y의 31번 비트와는 1이어야 한다.  Xneg 변수를 선언하고 x를 31번 오른쪽으로 시프트 한 값과 1을 &연산취하면 x가 음수일 경우 1을 가진다.  비슷한 논리로 Yneg을 구한다. 그 후 cal변수를 선언하고 x + ~y +1 ( = x-y )값을 넣어준다.  그리고 Cneg = ((cal>>31)&1) | (!(cal^0)) 이렇게 선언하면 cal이 음수일 때 또는 0일 때 1의 값을 가진다.  그후 반환 값을 정할 때 경우 1,3,4 와 2를 구분하자  경우2는 (!Yneg & Xneg) 를 하면 표현 가능하다.  이 연산은 y가 양수 x가 음수일 경우만 1을 반환한다.  경우1,3,4는 x와 y부호 같을 때 = !(Xneg^Yneg) 부호 다르면 0 같으면 1을 반환  그리고 뒤에 &Cneg을 추가하면 부호가 다르고 cal이 음수 또는 0일경우 1을 반환한다.  즉 반환값은 (!(Xneg^Yneg)&Cneg) | (!Yneg & Xneg) 으로 설정하면 된다. |
| 1. logicalNeg ( ! 논리연산자 사용 하지 않고 참일 경우 0을 반환 거짓일 경우 1을 반환 ) |
| !을 사용 하지 않고 입력 받은 값이 0인지 0이아닌지 확인하는 과정이 필요하다.  비트표현을 생각해보면 0이면 모든 비트가 0이고 0이 아니면 임의의 비트에서는 무조건 1이 존재 한다.  4번 alloddbit 에서 사용한 방법을 사용해 임의의 비트에 존재하는 1을 0번째 비트로 옮겨보자 (이 과정을 수행하면 모든 비트가 0일경우에는 변화가 없고 1인 비트가 있으면 그 비트는 0번쨰 비트로 옮길 수 있다. )  입력 받은 값을 x라고 하자 이때 x는 임의의 n번째 비트가 1이라고 가정한다.  그리고 x를 16번 왼쪽 쉬프트 한 값을 shift1이라하자 그 후 x|shift1 연산을 하게 되면 결과 값은 0번~15번 비트 사이에 비트 값이 1인 임의의 비트가 생성된다. 이 결과 값을 x에 저장하고 다시 x를 왼쪽 8번 쉬프트를 한 값을 shift2이라고 하자 위와 같은 |연산을 하게 되면 결과 값은 0번~7번 비트 사이에 비트 값이 1인 임의의 비트가 생성된다. 이러한 과정을 반복하게 되면 0번~1번 비트 사이에 비트 값이 1인 임의의 비트가 생성되고 한번 더 연산을 수행하면 0번째 비트 값이 1이 된다. 그리고 이 결과값을 x라고 하자.  지금 까지 의 연산은 비트표현이 [000…000]이 아닌 모든 비트들에 대해서 0번쨰 비트를 1로 만드는 과정이다. 이 연산을 [000..000]에 수행을 하면 변화는 없다.  그 후 참일 경우 0을 반환하고 거짓일 경우 1을 반환해야 하니깐 result = x ^ 1연산을 해서 0번째 비트를 반전시켜 준다.  이 연산을 시행하면 초기에 입력 받은 값이 0이면 비트표현은 [000….001]이 되고 0이 아니면 [….0]이 된다.  그리고 리턴 값으로 result & 1 을 수행하면 입력 받은 값이 0이면 비트표현은 [00…001] 즉 1을 반환하고 입력 받은 값이 0이 아니면 [000…0000]이 된다. 즉 0이 반환된다. |
| 1. Howmanybits ( 입력 받은 수 2의보수형태로 비트표현 하기 위해 필요한 최소 비트개수 ) |
| 양수일 경우에는 비트 값이 1인 비트 중에서 가장 큰 비트를 찾으면 된다.  음수일 경우에는 비트 값이 0인 비트 중에서 가장 큰 비트를 찾으면 된다.  이 두 가지를 구분하지 않고 찾기 위해 양수일 경우에는 비트를 반전시키지 않고 음수일 경우에만 비트를 반전시켜보자.  이를 위해서 ^의 특징을 사용할 것이다. x^[000…000]은 비트가 그대로 유지되고 x^[111…111]은 비트가 반전이 된다. 또한 31번 비트는 양수이면 0 음수면 1이므로 이 두 가지 특징을 사용하면 된다.  Sign 변수에 (x>>31) 값을 저장하면 x가 양수면 [000…000] 음수면 [111…111]이 저장된다.  그 후 x^sign을 수행하면 양수이면 비트유지, 음수이면 비트 반전이 된다.  그리고 비트들을 확인을 해야 하는데 4번 alloddbit와 9번 logicalneg 에서 사용했던 방법과 비슷한 방법을 사용해보자  우리의 목표는 비트 값이 1인 비트 중에서 가장 큰 비트를 찾는 것이다.  !(x>>16) 이 연산을 수행하면 x의 16~31번째 비트가 [000…000] 아닌지 확인 가능하다. 만약 [000…000]이면 1을 반환하고 아니면 0을 반환할 것이다.  [000…000]이면 우리는 0~15번 비트를 확인해야 하고 아니면 16~31번 비트를 확인 해야 한다. 즉 [000…000]이면 왼쪽으로 0번 시프트 아니면 왼쪽으로 16번 시프트를 수행 해야 한다.  이러한 논리를 위해 !(x>>16)에 !을 더 추가해 [000…000]이면 0을 반환하고 아니면 1을 반환하게 하자. 그 후 !!(x>>16)을 오른쪽으로 4번 shift한 값을 shift1이라 하자. Shift1의 값은 만약 x의 16~31번 비트가 [000…000]이면 16의 값을 아니면 0의 값을 가지게 된다. 그러면 x = x>>shift1을 수행하면 우리가 원하던 필요 없으면 생략하고 필요한 부분이면 유지하는 과정을 수행 가능하게 된다. 이 과정을 계속 반복해보자.  Shift2 = !!(x>>8) << 3, x = x >> shift2, …. Shift4 = !!(x>>2) << 1, x = x >> shift4  여기까지 수행하면 현재 x는 n~n+1번째 비트가 [00]인지 아닌지 확인을 하고 [00]일 경우 n-2번째 비트를 확인하기 위해 왼쪽으로 0번 아닐 경우 n+2번 비트 확인을 위해 왼쪽으로 2번 shift하였다. 이제 마지막으로 shift5 = !!(x>>1)), x= x>>shift5 까지 하면 x에는 확인한 자리가 0이면 0 1이면 1의 값을 가지게 된다.  여기서 shift1~shift5를 확인해보면 각각 확인한 자리들이 [000…000]이면 0의 값을 아니면 시프트 한 횟수를 저장 하고 있다.  결국 shift1 + … + shift5 + x를 하면 비트 값이 1인 비트 중에서 가장 큰 비트가 몇 번 째인지 확인이 가능하게 된다. 그리고 2의 보수의 형태이므로 +1을 해주면 입력 받은 수 2의보수형태로 비트표현 하기 위해 필요한 최소 비트개수를 구할 수 있게 된다. |
| 1. Floatscale2 ( 배정도형 \*2 수행하기 ) |
| 입력 받은 부동소수점을 uf라 하자  우선 부동소수점의 비트표현에 대해서 생각해보자  부동소수점의 비트표현은 31번은 부호비트 23~30은 지수비트 0~22은 비율비트로 구분이 된다.  부동소수점은 크게 지수비트 기준으로 특수 값 아니면 정규화 또는 비정규화 구분이 가능하다.  그저 지수부분에 1을 추가하면 비정규화 값에 대해서는 2배가 아닌 엄청난 차이를 만들게 되어서 다른 방법을 생각해야 한다. 또한 간단히 uf를 왼쪽으로 shift할 경우에는 지수부분에 값이 있으면 비율부분뿐만 아니라 지수부분도 2배가 되어 총 8배가 될 것이다. 이를 구분하기 위해서 비정규화와 정규화를 구분 해야 한다. 그리고 조건에 따라 리턴 방법을 달리 해야 하므로 if구문을 사용하자. 구분 방법은 23~30번 비트가 0000 0000가 맞으면 비정규화 아니면 정규화 또는 특수 값으로 구분이 가능하다. 이것을 식으로 서술하면 uf & 0x7f000000 ==0으로 표현 가능하다. 지수비트가 0000 0000이 아닌 경우 좌변의 식은 모든 비트가 0이 될 것이다. 비정규화 값을 2배하기위해서는 그저 uf를 왼쪽으로 1번 shift 하면 충분하다. 또한 왼쪽으로 1번 shift를 하면 부호비트가 사라지므로 0x80000000 & uf를 통해서 부호 비트를 확인해 | 연산으로 부호 비트를 전달해 주어야 한다.  이제는 특수 값과 정규화 값을 구분해야 한다. 특수 값은 uf & 0x7f000000 을 수행하면 0x7f000000이 결과인 반면 정규화 값은 지수 부분 중 임의의 한 비트는 항상 0이기 때문에 0x7f000000과 다른 값을 가지게 된다. 이 방법을 통해 else if 구문으로 구분을 하자.정규화 값을 2배하는 방법을 생각하면 된다. 그저 지수부분에 1을 추가하면 우리가 원하는 값을 구한수 있을 것이다. 이것을 식으로 표현하면 uf + 0x00800000을 수행 하면 된다. 특수 값은 그대로 리턴하면 되는 조건이 있으므로 따로 else구문을 적을 필요가 없다. |
| 1. Floatfloat2int ( 부동소수점 표현을 int로 표현하기 ) |
| 부동소수점 표현은 (M/2^23)\*2^E 로 표현이 가능하다. 이때 값을 가장 크게 변화시키는 것은 E의 값이다 E의 값에 따라 특수 값, 정규화 값, 비 정규화 값으로 나눌 수 있다. 이번에는 E의 범위 따라 정수표현으로 바뀔 때 달라지는 return값을 생각해보자.  부동소수점 표현을 정수로 바꾸기 위해서 구분해야 할 것은 4가지이다.  첫 번째는 조건에 나온 것 과 같이 표현이 불가능할 경우, 두 번째는 값이 1미만일경우, 세 번째는 E이 23미만일 때, 마지막으로 E가 23이상일 때  첫 번째를 나눈 이유는 조건에 나온 것처럼 정수로 표현 불가능하거나 특수 값일 경우 0x80000000u로 리턴을 하면 된다. 정수 변수의 크기는 32비트이며 그 중 한 개는 부호 비트로 사용한다. 즉 부동소수점에서 E가 31을 넘어버리면 정수로 나타낼 수가 없다. 왜냐하면 E가 31일 경우 양수인 부동소수점으로 표현 가능한 숫자는 2^31이상 2^32 미만이다. 즉 31을 넘어버리면 표현 가능 숫자가 2^32를 넘어버리게 되어 정수 자료형의 표현 범위를 넘게 된다.  두 번째 구분 방법은 E가 음수인지로 확인 가능하다. 만약 E가 음수이게 되면 trivial하게 부동소수점의 표현은 -1과 1사의 값을 가지게 되고 이 값을 정수형으로 표현하게 되면 버림을 실행하여 0의 값을 가지게 된다.  세 번째 구분을 한 이유는 비트표현을 생각해보면 X/2^n은 X을 오른쪽으로 n번 시프트 한 결과이다. 그리고 X\*2^n을 비트표현을 생각해보면 X를 왼쪽으로 n번 시프트 한 결과이다. 고로 (M/2^23)\*2^E는 E가 23이상이면 M을 (E-23)만큼 왼쪽으로 시프트한 값과 같고, E가 23미만이면 M을 (23-E)만큼 오른쪽으로 시프트한 값과 같게 된다. 이제 마지막으로 고려해야 할 것은 3번째 4번쨰의 결과값을 가지고 부호에 따라 비트의 표현이 달라지는 것이다. 즉 세 번째와 네 번째에서 구한 비트표현에 대해 처음에 입력 받은 uf가 음수이면 비트표현을 반전 시킨 후 1을 추가해주고 양수이면 구한 값 그대로 리턴하면 된다.  이 과정을 수행하기 위해서 필요한 값들을 구하는 과정들은 아래와 같다.  우선 부동소수점은 31벙 비트는 부호비트, 30번부터 23번 비트까지는 지수비트, 22번부터 0번까지는 비율비트이다. 부호 비트를 구하기 위해서는 uf를 오른쪽으로 31번 시프트하고 &1 연산을 수행하면 음수일 경우 1의 값을 양수일 경우 0의 값을 결과로 가질 것이다. 그리고 지수 비트는 uf를 오른쪽으로 23번 시프트 하고 &0xff연산을 통해 남길 수 있다. 비슷하게 비율 비트는 &0x7fffff연산을 통해 가능하다. 그리고 E는 구한 지수비트에 bias를 빼줌으로 구할 수 있다. 이때 bias는 (2^7)-1 즉127이 된다. 마지막으로 M은 특별히 방금 구한 비율비트 부분과 &0x800000을 통해 구해준다. 이유는 우리는 M을 오직 세 번째 네 번째 경우에서만 사용할 것이고 이 경우들은 E의 값이 0이상이기 때문이다. |
| 1. Floatpower2 ( 입력받은 정수를 x라 할 때 2^x를 부동소수점으로 표현하기 ) |
| 부동소수점의 표현을 생각해보면 지수비트가 0이 아니고 비율비트가 [000…000]일 경우 2^x꼴로 나타낼 수 있다. 즉 입력 받은 정수를 지수비트로 표현하고 그 후 왼쪽으로 23번 shift하면 우리가 원하는 답을 구할 수 있다. 이때 조건에서 입력 받은 x가 음수이면 0을 리턴 하고 표현 불가능하면 +inf를 리턴 하라고 설명한다.  부동소수점에서 지수비트의 표현은 실제 값 + bias로 나타낸다. 즉 우리가 수행해야 할 것은 bias를 구해 입력 받은 x만큼 더하고 왼쪽으로 shift를 하면 부동소수점으로 표현 가능하다 이때 우리가 표현 할 수 있는 지수비트의 크기는 8비트이므로 255가 최대이다 즉 bias+x가 255이상이 되어버리면 무한대로 간주하려 +inf를 리턴 해야 한다. 이 경우는 if ( x+bias >=255 ) return 0xff << 23 으로 가능하다.  그리고 마지막으로 2^x표현은 x+bias << 23으로 표현 가능하다. |