Data Lab: Manipulating Bits

Report

20220127 임유진

이번 Lab에서는 정수의 bit-level 표현에 익숙해지기 위해 bitwise operator(~, &, |, ^), shift operator(〉〉, 〈〈), +, !의 연산자만 사용하여 정수에 대해 특정한 작업을 수행할 수 있는 함수를 작성해보았다. 문제 별 풀이는 다음과 같다.

problem 1 - bitNor(x, y)

Problem 1은 ~와 &의 연산자만을 이용하여 인자로 받은 x, y 값에 대해 Nor 연산, 즉 ~(x | y)을 수행하는 함수 bitNor(int x, int y)을 구현하는 것이다.

풀이)

논리 연산에서 $\overline{(A + B)} = \bar{A} \cdot \bar{B}$ 이므로, x, y에 대해 bit 단위로 Nor의 논리 연산을 수행한 결과, 즉 ~(x | y) = ~x & ~y임을 알 수 있다. 따라서 bitNor이 ~x & ~y의 값을 반환하도록 함수를 구현하였다.

problem 2 - isZero(x)

Problem 2는 bitwise operator(~, &, |, ^), shift operator(〉〉, 〈〈), +, !의 연산자만을 이용하여 인자로 받은 x의 값이 0인지 판단하는 함수 isZero(int x)를 구현하는 것이다. 만약 인자로 받은 x의 값이 0인 경우 1을, 0이 아닌 경우 0을 반환하게 된다.

풀이)

논리 연산에서 자기 자신과의 Xor 연산의 값이 0($x \oplus x = 0$) 임을 이용하면, bit 단위로 Xor의 논리 연산을 수행했을 때 같은 값을 가지는 수끼리의 bitwise Nor 연산의 결과는 0이 될 것임을 알 수 있다. 또한 ! 연산자는 0이 아닌 인자들에 대해서는 1을, 0에 대해서는 0을 반환한다는 것을 이용할 수 있다.

따라서 인자로 받은 x와 0의 값을 가지는 변수 zero와의 bitwise Nor 연산 결과(x ^ zero)에 ! 연산자를 적용한 값을 반환하도록 함수를 구현하였다.

problem 3 - addOK(x,y)

Problem 3는 bitwise operator(~, &, |, ^), shift operator(〉〉, 〈〈), +, !의 연산자만을 이용하여 인자로 받은 x, y의 값에 대해 x + y 연산 시 오버플로우의 발생 여부를 판단하는 함수인 addOK(int x, int y)를 구현하는 것이다. 만약 인자로 받은 x, y에 대하여 x + y 연산이 오버플로우 없이 수행될 수 있다면 1을, 그렇지 않다면 0을 반환한다.

풀이)

signed int 타입은 4byte, 즉 32bit이며 따라서 -2^{31} 와 $2^{31}-1$ 사이의 값을 가질 수 있다. 따라서 signed int 타입의 두 정수의 합이 -2^{31}보다 작아지거나 $2^{31}-1$보다 커지게 되면 오버플로우가 발생하게 된다.

그리고 signed int 타입의 음이 아닌 데이터는 $0 \sim 2^{31}-1$ 사이의 값을, 음수 데이터는 $-2^{31} \sim -1$ 의 값을 가진다는 것을 고려할 때 signed int 타입에서 음이 아닌 데이터와 음수 데이터 간의 덧셈을 하게 되면 오버플로우가 발생하지 않을 것이라는 사실을 알 수 있다.

두 데이터의 부호 비트가 같은 경우에는 오버플로우가 발생할 수도 발생하지 않을 수도 있는데 이를 확인하기 위해 부호 비트를 사용하였다. 두 데이터의 합이 TMax를 넘어가는 경우 음수로 오버플로우가 발생하며, TMin보다 작아지는 경우 양수로 오버플로우가 발생한다. 따라서 두 데이터와 두 데이터의 합의 부호 비트가 달라지게 될 것이라는 것을 알 수 있다.

이를 이용하여 x + y의 값을 저장하는 sum 변수를 만들고, x와 y의 부호 비트가 같은 경우 0xFFFFFFFF, 다른 경우 0x00000000의 값을 가지는 ~((x 〉〉 31) ^ (y 〉〉 31))의 연산 결과와 sum과 x의 부호 비트가 같은 경우 0x00000000, 다른 경우 0xFFFFFFFF의 값을 가지는 ((sum 〉〉 31) ^ (x 〉〉 31))의 연산 결과와의 bitwise and 연산을 실행한 결과를 overflow 변수에 저장하였다. 따라서 overflow의 값은 오버플로우가 발생할 때 0xFFFFFFFF, 오버플로우가 발생하지 않을 때 0x00000000의 값을 갖게 되며, overflow에 ! 연산자를 적용한 값을 반환하여 오버플로우가 발생하면 0, 발생하지 않으면 1의 값을 반환할 수 있게 된다.

problem 4 - absVal(x)

Problem 4는 bitwise operator(~, &, |, ^), shift operator(〉〉, 〈〈), +, !의 연산자만을 이용하여 인자로 받은 x의 절댓값을 반환하는 함수 absVal(int x)를 구현하는 것이다.

풀이)

만약 인자로 받은 x의 값이 음이 아닌 정수라면 x의 절댓값은 x와 같으며, x의 값이 음의 정수라면 x의 절댓값은 -x가 될 것이다. 2's complement에서 -x = ~x + 1임을 고려할 때 즉 x가 음수일 때는 ~x + 1이 x의 절댓값이 됨을 알 수 있다. 따라서 x가 음수일 때 ~x + 1의 값을, 음수가 아닐 때 0x00000000의 값을 가지는 ((x 〉〉 31) & (~x + 1))의 연산 결과와 x가 음수가 아닐 때 x, 음수일 때 0x00000000의 값을 가지는 (~(x 〉〉 31) & x)의 연산 결과와의 bitwise or 연산을 통해 함수가 x의 절댓값을 반환하도록 구현하였다.

problem 5 – logicalShift(x, n)

Right shift에는 좌측 끝을 0으로 채우는 Logical shift와 좌측 끝을 sign bit로 채우는 Arithmetic shift가 존재하지만, 대부분의 C언어 컴파일러/컴퓨터 조합에서는 signed 타입 데이터의 Right shift에 Arithmetic shift을 적용하고 있다. Problem 5는 bitwise operator(~, &, |, ^), shift operator(〉〉, 〈〈), +, !의 연산자만을 이용하여 인자로 받은 signed int x에 대해 오른쪽으로 n bit만큼 logical shift한 결과를 반환하는 함수 logicalShift(int x, int n)을 구현하는 것이다.

풀이)

signed int 타입의 데이터에서 음이 아닌 값의 경우 부호 비트가 0, 음의 값인 경우 부호 비트가 1이므로, n bit 만큼 right shift을 수행한 결과 arithmetic shift에 의해 음이 아닌 값의 경우 좌측의 n bit가 0으로, 음의 값의 경우 1로 채워짐을 알 수 있다. 따라서 좌측을 무조건 0으로 채우는 logical shift의 결과를 얻기 위해서는 음이 아닌 수의 경우 right shift의 결과를 그대로 사용할 수 있다. 음수의 경우 좌측 n bit를 0으로 바꿔줘야 하며, right shift의 실행한 결과와 좌측 n bit만 0, 나머지 bit는 1의 값을 가지는 수와의 bitwise and 연산을 통해 해당 결과를 얻을 수 있다. 이때 1의 값을 가지는 변수 one에 대하여 (((one 〈〈 31) 〉〉 n) 〈〈 1)을 수행하여 좌측 n bit만 1, 나머지 bit는 0의 값을 구하였고, 이 값과 (x 〉〉 31)과의 bitwise and 연산을 수행하여 x가 음이 아닌 수일 때는 0x00000000, 음수일 때는 좌측 n bit만 1, 나머지 bit는 0의 값을 가지는 변수 sub을 구하였다. 마지막으로, 인자로 받은 x를 오른쪽으로 n bit만큼 arithmetic shift한 결과와 ~sub와의 bitwise and 연산을 통해 x를 오른쪽으로 n bit만큼 logical shift한 결과를 반환하는 함수를 구현할 수 있었다.