Intro. to Computer SW Systems Lab Report

[Data Lab: Manipulating Bits]

20220100 박기현

kihyun@postech.ac.kr

명예 서약 (Honor Code)

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

Problem 1. bitNor(x, y): ~(x | y) using only ~ and &

```
int bitNor(int x, int y) {
  return (~x) & (~y);
}
```

Problem 1은 ~(Not)과 &(And)를 이용하여 ~(x | y)(Nor)을 반환하는 함수를 구현하는 문제이다.

De Morgan's Law에 의해 ~(x | y) = (~x) & (~y)가 성립하므로 ~(Not)과 &(And)만을 이용하여 구현할 수 있다.

Problem 2. isZero(x): return 1 if x == 0, and 0, otherwise

```
int isZero(int x) {
  return !x;
}
```

Problem 2는 x가 0이면 1을, 그렇지 않으면 0을 반환하는 함수를 구현하는 문제이다.

!(Logical Negation)은 값이 True이면 False를, False이면 True를 반환한다.

x가 0이면 False, 그렇지 않으면 True이므로 !(Logical Negation)을 이용하여 x가 0(False)일 때 1(True)을, 이외의 값(True)일 때 0(False)을 반환하도록 구현할 수 있다.

Problem 3. addOk(x, y): determine if can compute x + y without overflow

```
int addOK(int x, int y) {
  return (((x ^ y) >> 31) & 1) | !((((x + y) ^ x) >> 31) & 1);
}
```

Problem 3은 x + y를 overflow 없이 계산할 수 있는지 결정하는 함수를 구현하는 문제이다.

우선, overflow가 발생하는 경우에 대해서 알아야 한다.

overflow는 MSB(Most Significant Bit)의 Carry-in과 Carry-out이 다를 때 발생한다.

이때, x와 y의 부호, 즉 MSB(Most Significant Bit)가 다르면 무조건 Carry-in과 Carry-out이 같아지므로 overflow는 발생하지 않는다.

따라서 이 경우에 대해 ^(XOR)을 이용하여 x ^ y의 MSB(Most Significant Bit)가 1이 되도록 하고, 이를 >>(Right Shift)와 &(And)를 이용하여 최종적으로 1이 반환될 수 있도록 구현한다.

다음으로는 x와 y의 부호, 즉 MSB(Most Significant Bit)가 같은 경우에 대해서 고려해야 한다.

x와 y의 MSB(Most Significant Bit)가 같은 경우에서 Carry-in과 Carry-out이 다른 경우는 x+y의 MSB(Most Significant Bit)가 x 또는 y의 MSB(Most Significant Bit)와 다른 경우이다.

따라서 이 경우에 대해서도 마찬가지로 ^(XOR)을 이용하여 (x + y) ^ x(또는 y)의 MSB(Most Significant Bit)가 1이 되도록 하고, 이를 >>(Right Shift)와 &(And)를 이용하여 최종적으로 1이 반환될 수 있도록 구현한다.

하지만 이는 overflow가 발생하는 경우이므로 !(Logical Negation)을 이용하여 옳은 값을 반환한다.

최종적으로 두 경우를 |(Or)을 이용하여 어느 경우에도 만족할 수 있도록 구현할 수 있다.

Problem 4. absVal(x): absolute value of x

```
int absVal(int x) {
  return ((x >> 31) & (~x + 1)) | (~(x >> 31) & x);
}
```

Problem 4는 x의 절댓값을 반환하는 함수를 구현하는 문제이다.

만약 x가 음수이면 절댓값은 ~(Not)을 이용하여 비트를 반전시킨 후 1을 더해야 한다.

만약 x가 양수이면 절댓값은 x값과 같다.

따라서 x가 음수인지 양수인지를 판단하고, 그에 따른 연산을 진행해야 한다.

x의 부호를 판단하기 위해 MSB(Most Significant Bit)가 1인지 0인지 알아야 하므로, >>(Right Shift)를 이용한다.

이때, >>(Right Shift)는 Arithmetic Shift를 지원하기 때문에 만약 x가 음수이면 (x >> 31)의 모든 비트가 1이 되고, x가 양수이면 (x >> 31)의 모든 비트가 0이 된다.

따라서 음수인 경우 (x >> 31) & (~x + 1)를 통해, 양수인 경우 ~(x >> 31) & x를 통해 절댓값을 반환할 수 있다.

최종적으로 두 경우를 |(Or)을 이용하여 어느 경우에도 만족할 수 있도록 구현할 수 있다.

Problem 5. logicalShift(x, n): shift x to the right by n, using a logical shift

```
int logicalShift(int x, int n) {
  return ~(((1 << 31) >> n) << 1) & (x>>n);
}
```

Problem 5는 x를 n만큼 Logical Shift를 이용하여 비트를 오른쪽으로 이동시키는 함수를 구현하는 문제이다.

우선, >>(Right Shift)는 Arithmetic Shift를 지원하므로 MSB(Most Significant Bit)가 1인 경우, 왼쪽 비트가 1로 채워진다.

이를 해결하기 위해 상위 n비트를 0으로 만들어 &(And)로 상위 n비트가 0이 될 수 있도록 구현해야 한다.

먼저, 1을 <<(Left Shift)를 이용하여 MSB(Most Significant Bit)에 위치시키고, n만큼 >>(Right Shift)를 이용하여 상위 비트에 1을 채운다.

하지만 이때, 상위 n + 1비트가 1이므로 다시 왼쪽으로 1만큼 이동시켜 상위 n비트가 1, 나머지 비트가 0이 될 수 있도록 구현한다.

이후 ~(Not)을 통해 비트를 반전시키고, &(And)를 함으로써 상위 n비트는 0, 나머지 비트는 x가 n만큼 오른쪽으로 이동한 비트에 맞게 반환되도록 구현할 수 있다.