CSED211: Lab 1

손량(20220323)

Last compiled on: Monday 18th September, 2023, 19:56

1 개요

비트 연산을 사용하여 몇 가지 간단한 정수 관련 함수를 구현해 본다.

2 코드 설명

2.1 `bitNor` – Bitwise NOR

드 모르간의 법칙에 따르면, $\neg(p \vee q) = (\neg p) \wedge (\neg q)$이다. 이를 C언어의 비트 연산자로 옮겨, `(~x) & (~y)`라는 코드를 작성하였다.

2.2 `isZero` – Check If Zero

C언어는 0을 `false`, 0이 아닌 값을 `true`로 판정하는 것을 이용, logical not 연산자 `!`를 사용하여 문제를 해결하였다.

2.3 `addOK` – Checking Overflows

오버플로우 없이 두 signed int를 더할 수 있는지 판정해야 한다. 우선 `x`와 `y`의 부호가 다를 경우는 언제나 오버플로우 없이 더할 수 있고, 두 값의 부호가 같다면 덧셈 과정에서 most significant bit의 carry in과 carry out이 같은지 검사하였다. Carry in과 carry out이 같은 상황이라면 오버플로우가 일어나지 않은 것이고, 다르다면 오버플로우가 발생한 상황이다. C언어 표준에 따르면 signed int 연산 중의 오버풀로우는 undefined behavior이기 때문에, `x_top2`, `y_top2` 변수에 `x`, `y` 각각의 상위 2비트를 저장하고, 하위 30비트와 상위 2비트를 따로 계산하여 carry in, carry out을 구하는 코드를 작성하였다.

2.4 `absVal` – Absolute value of integer

절댓값을 구해야 한다. 2의 보수 표기법에서는 `-x = ~x + 1`임에 주목하자. 만약 `if`문과 비교 연산자가 사용 가능하다면 다음과 같은 코드를 작성할 것이다.

```
if (x >= 0)
  return x;
else
  return (~x) + 1;
```

비교 연산자 대신 most significant bit를 검사하여 음수 판정을 할 수 있다.

```
int msb = (1 << 31) & x;
if (!msb)
  return x;
else
  return (~x) + 1;
```

한편, XOR 연산은 1이 항등원이자 자기 자신의 역원이기 때문에, bitwise not 대신 다음과 같이 쓸 수 있다.

```
int msb = (1 << 31) & x;
if (!msb)
  return 0 ^ x;
else
  return (0xffffffff ^ x) + 1;
```

그런데, C언어에서의 signed int의 right shift는 arithmetic shift이기 때문에, `msb`를 31만큼 right shift하면, `msb`가 0인 경우 그대로 0이고, `msb`가 1인 경우 모든 bit가 1인 값, 즉 `0xffffffff`가 될 것이다. 이를 코드로 옮기면 다음과 같다.

```
int msb = (1 << 31) & x;
if (!msb)
  return (msb >> 31) ^ x;
else
  return ((msb >> 31) ^ x) + 1;
```

여기에다, `isZero`를 생각해 보면 `msb`가 0이 아닐 때 1, 0일 때 0을 얻기 위해서는 `!!msb`라고 적을 수 있을 것이다.

```
int msb = (1 << 31) & x;
if (!msb)
  return ((msb >> 31) ^ x) + (!!msb);
else
  return ((msb >> 31) ^ x) + (!!msb);
```

이제 `if`문을 필요 없게 된다.

```
int msb = (1 << 31) & x;
return ((msb >> 31) ^ x) + (!!msb);
```

2.5 `logicalShift` – Logical Right Shift

`x`를 n만큼 arithmetic right shift하면, 상위 $n+1$ 비트는 `x`의 most significant bit와 같게 채워질 것이다. Logical shift를 했다면 이 중에서 앞의 n비트는 0으로 채워야 하므로, `x`의 most significant bit만 남기고 나머지는 0인 값을 얻은 다음 이를 $n-1$만큼 right arithmetic shift를 해 앞서 `x`를 arithmetic shift한 값과 XOR하면 된다. `x`의 most significant bit가 0인 경우에는 arithmetic shift와 logical shift 결과가 같고, most significant bit를 shift한 결과가 0이기 때문에 정확한 결과가 나온다. `x`의 most significant bit가 1인 경우에는 most significant bit를 shift한 결과의 상위 n비트는 1이고, 나머지는 0이기 때문에 XOR 결과 logical shfit를 한 것과 같은 결과가 나온다. 이 문제에서는 뺄셈 연산자를 사용할 수 없기 때문에, $n-1$만큼 shift하는 대신 n만큼 right shfit하고 1만큼 left shift하였다.