CSED211: Data Lab1 Report

20230024 / 문요준

Overview

이 Lab 은 정수에 대한 bit-level 연산을 위한 C 프로그래밍 퀴즈이고, bitNor 연산 함수, 정수가 0 인지 확인하는 함수, 오버플로우 없이 더하기 연산이 되는지 확인하는 함수, 절댓값을 반환하는 함수, Logical right shift 를 수행하는 함수 퀴즈를 푸는 것이 목표이다.

1. Question #1. bitNor

2-1. Explanation.

bitNor 함수는 두 정수 x 와 y 에 대해 논리적 NOR 연산을 수행한다. NOR 연산은 수학적으로 ~(x | y)로 정의되며, 여기서는 드 모르간의 법칙을 활용해 x | y = ~(~x & ~y) 식을 활용해 연산한다.

2-2. Solution.

| 연산자는 사용할 수 없으므로, 드모르간의 법칙을 이용해 ~x & ~y 로 OR 연산을 대체한다

2-3. Implementation.

```
int bitNor(int x, int y) {
  //to be implemented
  return ~x & ~y;
}
```

2. Question #2. bitNor

2-1. Explanation.

isZero 함수는 정수 x 가 0 인지 확인하고, 0 일 경우 1 을 반환하고, 0 이 아닐 경우 0 을 반환하는 함수이다.! 연산자는 부정 연산자로, 특정 값이 참(1)이면 거짓(0)을 반환하고, 거짓(0)이면 참(1)을 반환하기 때문에 이를 이용하여 x 가 0 인지 아닌지를 확인할 수 있다.

2-2. Solution.

!x 를 이용하면 x 가 0 인지 아닌지를 확인할 수 있다

2-3. Implementation.

```
int isZero(int x) {
  //to be implemented
  return !x;
}
```

3. Question #3. addOK

3-1. Explanation.

addOK 함수는 두 정수 x 와 y 를 더할 때 **오버플로우(overflow)**가 발생하는지 여부를 확인하는 함수이다. 오버플로우는 부호가 동일한 두 정수를 더했을 때, 결과값의 부호가 달라지는 경우 발생한다. 즉, 두 정수 x 와 y 의 부호가 같고, 그 합의 부호가 다르면 오버플로우가 발생한 것이다. 이 함수는 오버플로우가 발생하지 않았을 경우 1 을 반환하고, 오버플로우가 발생하면 0 을 반환한다.

3-2. Solution.

부호 비트는 정수의 최상위 비트(31 번째 비트)로, 부호 비트를 통해 오버플로우 여부를 판단할 수 있다. x 와 y 의 부호를 비트 연산으로 추출한 후, 이를 더한 값의 부호와 비교하면 오버플로우 여부를 판단할 수 있다.

1. sign_x = x >> 31: x 의 최상위 비트를 추출하여 부호를 확인한다. 음수면 1, 양수면 0 이다.
2. sign_y = y >> 31: y 의 부호를 추출한다.
3. sum_xy = x + y: x 와 y 를 더한 값을 계산한다.
4. sign_sum_xy = sum_xy >> 31: 두 값의 합인 sum_xy 의 부호를 추출한다.

오버플로우가 발생하는 경우는 x 와 y 의 부호가 동일하지만, 합의 부호가 다른 경우이고, XOR(^) 연산을 사용하여 부호가 다른지 여부를 확인할 수 있다. !(~(sign_x ^ sign_y) & (sign_x ^ sign_sum_xy))는 두 수의 부호가 같고, 합의 부호가 다르면 0(오버플로우 발생)을 반환하고, 그렇지 않으면 1(오버플로우 없음)을 반환한다.

3-3. Implementation.

```
int addOK(int x, int y) {
  //to be implemented
  int sign_x = x >> 31;
  int sign_y = y >> 31;

  int sum_xy = x + y;
  int sign_sum_xy = sum_xy >> 31;

  return !(~(sign_x ^ sign_y) & (sign_x ^ sign_sum_xy));
}
```

x 와 y 의 부호를 추출한 후, 두 수의 부호가 같고 결과값의 부호가 달라졌는지를 판단하여 오버플로우 여부를 확인한다.

- sign_x ^ sign_y 는 x 와 y 의 부호가 다르면 1, 같으면 0 이 된다.

- sign_x ^ sign_sum_xy 는 x 와 합계의 부호가 다르면 1, 같으면 0 이 된다.

- 두 조건을 결합한 !(~(sign_x ^ sign_y) & (sign_x ^ sign_sum_xy))를 통해 오버플로우가 발생하지 않으면 1, 발생하면 0 을 반환하게 된다.

4. Question #4. absVal

4-1. Explanation.

absVal 함수는 정수 x 의 절대값을 반환하는 함수이다. 따라서 양수는 그대로 반환되고 음수는 그 부호를 반전시켜 양수로 반환된다. 인자로 받은 x 의 부호를 판단한 후, 음수인 경우 부호를 반전시켜 절대값을 계산하는 방식으로 동작한다.

4-2. Solution.

이 함수는 정수 x 가 음수일 경우 부호를 반전시키고, 양수일 경우에는 그대로 반환한다. 이때, 조건문을 사용하지 않고 비트 연산만으로 절대값을 계산해야 한다.

1. sign = x >> 31 을 통해 x 의 부호 비트를 추출한다. x 가 음수이면 sign 은 -1(즉, 모든 비트가 1)이 되고, x 가 양수이면 sign 은 0 이 된다.

2. x + sign 은 x 가 음수일 때 x - 1 이 되며, 양수일 때는 그대로 x 가 된다.

3. (x + sign) ^ sign 을 통해 부호를 반전시켜 절대값을 계산한다. sign 이 -1 이면 x 의 비트가 반전되어 절대값이 되고, sign 이 0 이면 원래 값이 그대로 유지된다.

4-3. Implementation.

```
int absVal(int x) {
  //to be implemented
  int sign = x >> 31;
  return (x + sign) ^ sign;
}
```

부호 비트를 추출한 후, 비트 연산을 통해 음수인 경우 부호를 반전시켜 절대값을 계산한다. x >> 31 은 x 의 부호 비트를 추출하며, 음수일 경우 -1(모든 비트가 1), 양수일 경우 0 이 된다. 그 후, x + sign 과 sign 의 XOR 연산을 통해 부호를 반전시켜 절대값을 반환한다.

5. Question #5. logicalShift

5-1. Explanation.

logicalShift 함수는 정수 x 를 오른쪽으로 n 비트만큼 논리적 시프트하는 함수이다. 논리적 시프트는 비트를 오른쪽으로 이동시키며, 최상위 비트가 0 으로 채워지는 방식이다. C 언어에서 >> 연산자는 산술적 시프트를 수행하므로, 논리적 시프트를 수행하기 위해 마스크(mask)를 사용해 최상위 비트를 0 으로 설정한다.

5-2. Solution.

x >> n 을 통해 오른쪽으로 n 비트만큼 시프트하면 산술적 시프트가 일어나므로, 마스크를 생성해 최상위 비트를 0 으로 설정하는 논리적 시프트를 구현한다.

value = ~(((1 << 31) >> n) << 1)은 n 비트만큼 시프트된 후 남은 비트를 0 으로 설정하는 마스크를 생성한다. 1 << 31 은 최상위 비트가 1 인 값을 생성하고, 이 값을 n 만큼 오른쪽으로 시프트한 후 다시 1 비트 왼쪽으로 시프트하면, n 비트만큼은 0 이 되고 나머지 비트는 1 인 비트 마스크가 만들어진다. 마지막으로 ~ 연산을 사용하여 비트를 반전시키면, n 비트 이후의 모든 비트가 0 으로 설정된 마스크가 생성된다. x >> n 은 오른쪽으로 n 비트만큼 시프트하지만, 부호 비트가 유지된 채로 시프트된다. 따라서 & value 를 통해 최상위 비트를 0 으로 만들어 논리적 시프트를 구현한다.

5-3. Implementation.

```
int logicalShift(int x, int n) {
  //to be implemented
  int value = ~(((1 << 31) >> n) << 1);
  return (x >> n) & value;
}
```

1 << 31: 최상위 비트가 1 인 값을 생성한다. (ex: 10000000 00000000 00000000 00000000) 이를 n 비트만큼 오른쪽으로 시프트하면, 최상위 n 비트는 0 이 되고 나머지는 그대로 유지된다. 예를 들어, n = 4 일 경우, 값은 00001000 00000000 00000000 00000000 이 된다. 다시 1 비트 왼쪽으로 시프트하면 n 비트 이후가 모두 1 이 되며, 나머지는 0 이 된다. 이를 반전시키면 n 비트만큼은 0 이고, 나머지는 모두 1 이 되는 마스크가 생성된다. 최종적으로, x 를 오른쪽으로 n 비트 시프트하면, 부호 비트가 유지된다. 그리고, 마스크를 적용하여 최상위 n 비트를 0 으로 만들고, 논리적 시프트를 구현한다.

Results

```
[yojun313@programming2 datalab]$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitNor
 1      1       0       isZero
 3      3       0       addOK
 4      4       0       absVal
 3      3       0       logicalShift
Total points: 12/12
```