CSED211: Data Lab1 Report

20240832 / 채승현

1. Overview

이번 랩에서는 컴퓨터에서 숫자가 비트들로 저장되어 있으며, 여러 가지 비트 수준의 연산(~, &, |, ^, <<, >>)을 할 수 있다는 것을 상기하였다. 특히 이번 랩에서 주어진 문제는 주어진 비트 연산자를 한정된 수 이내로 이용하여 함수를 구현하는 것으로, 총 5문제가 있었다. 이후 내용에서 이 문제들에 대해 상세히 서술할 예정이다.

2. Question #1. bitNor

2-1. Explanation.

- 주어진 int형 변수 x, y에 대해 bitwise NOR 연산을 하여 그 값을 반환하는 함수 bitNor을 구현해야 한다. 이때 ~, & 연산을 최대 8번 사용할 수 있다.
- NOR은 OR 연산의 부정(NOT)으로, 단일 비트에 대한 bitwise NOR 연산의 결과는 다음과 같다.

	a = 0, b = 0	a = 0, b = 1	a = 1, b = 0	a = 1, b = 1
a NOR b	1	0	0	0

2-2. Solution.

- NOR은 NOT (x OR y)로 나타낼 수 있다. 하지만 이는 OR 연산이 사용된다.
- 이때 드모르간 법칙을 이용해, NOR을 NOT, AND만 사용하여 나타낼 수 있다.
    - 드모르간 법칙 : 논리곱(합)의 부정은 각각 부정의 논리합(곱)과 같다는 법칙 [1]
    - 이때 부정은 NOT, 논리곱은 AND, 논리합은 OR이다.
- 따라서 x NOR y는 NOT(x OR y)와 같고, 이는 드모르간 법칙에 따라 (NOT x) AND (NOT y)와 같다.
- (NOT x) AND (NOT y)를 비트 연산자로 표현하면 (~x) & (~y)이며, 이는 ~, & 연산을 3번만 사용한 것이다.

2-3. Implementation.

- bitNor을 c언어에서 다음과 같이 구현할 수 있다.

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

}
```

Listing . Code of bitNor.

- 앞선 2-2에서 설명한 바와 같이 int형 변수 x, y에 대한 bitwise NOR operation은 (~x) & (~y)로 표현할 수 있으며, 이를 그대로 반환해주었다.

3. Question #2. isZero

3-1. Explanation.

- 주어진 int형 변수 x가 0이면 1을 반환하고, 그렇지 않으면 0을 반환하는 함수 isZero를 구현해야 한다. 이때 , ~, &, ^, |, +, <<, >> 연산을 최대 2번 사용할 수 있다.
- 는 논리 연산자 NOT으로, 0(false)에 대해서는 1(true), 0이 아닌 다른 모든 숫자(true)에 대해서는 0(false)을 반환한다.

3-2. Solution.

- 3-1에서 보았던 논리 연산자 를 활용하면 isZero를 단순하게 나타내는 것이 가능하다.
- isZero는 x로 표현할 수 있다. 이는 연산을 1번만 사용한 것이다.

3-3. Implementation.

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

}
```

Listing 2. Code of isZero.

- 앞선 3-2에서 설명한 바와 같이 int형 변수 x에 대한 isZero의 결과값은 x 로 표현할 수 있으며, 이를 그대로 반환해주었다.

4. Question #3. addOK

4-1. Explanation.

- 주어진 int형 변수 x, y에 대해 x+y가 overflow 없이 계산된다면 1을, 그렇지 않다면 0을 반환하는 함수 addOK를 구현해야 한다. 이때 , ~, &, ^, |, +, <<, >> 연산을 최대 20번 사용할 수 있다.
- overflow는 정수형 데이터가 주어진 범위를 벗어났을 때 발생한다. int의 경우, 32개의 비트로 구성되어 있는데, 그 중 맨 첫 번째 비트(가장 왼쪽 비트)가 부호를 결정한다. 편의상 이 비트를 부호 비트라고 부르겠다. 부호 비트가 0인 경우가 양수, 1인 경우가 음수가 된다. 여기서 양수에 어떤 양수를 더하여 주어진 범위를 벗어나게 되면(positive overflow), 부호 비트가 1로 바뀌며 음수가 된다. 마찬가지로 음수의 경우 어떤 음수를 더하여 주어진 범위를 벗어나게 되면(negative overflow), 부호 비트가 0으로 바뀌며 양수가 된다. [2]
- overflow가 나는 경우는 양수끼리 더했을 때, 음수끼리 더했을 때로 국한되어 있다.
- >> 연산(right shift)은 전체 비트를 오른쪽으로 밀고, 빈 공간을 부호 비트로 채운다. int형의 경우 >> 31을 통해 모든 비트를 부호 비트로 바꿀 수 있다. (양수인 경우 모두 0, 음수인 경우 모두 1)

4-2. Solution.

- 4-1에서 보았던 부호 비트와 비트 연산자을 활용하여, addOK 함수를 구현할 수 있다.
- overflow가 일어나는 경우는 4-1에서 설명한 아래 두 가지 경우 밖에 없다.
- 음수끼리의 덧셈과 양수끼리의 덧셈의 경우를 나누어서 구현한다.
    - 음수끼리의 덧셈의 경우 x와 y가 모두 음수인지 확인하고, 이후 x+y가 양수인지 확인한다. 이는 4-1에서 언급한 >> 연산을 통해 구현할 수 있다. (x >> 31) & (y >> 31)은 x와 y가 모두 음수일 때만 모든 비트가 1이 된다. 또한 ~((x+y) >> 31)는 x+y가 양수일 때 모든 비트가 1이 된다. 이 둘을 &로 묶으면 음수끼리 더하여 양수가 되었을 때 모든 비트가 1이 되고, 그렇지 않은 경우엔 0이 된다. 이 결과값에 논리 연산자 를 적용하면, overflow가 일어나지 않은 경우 1, 그렇지 않은 경우 0이 된다.
        - ((x >> 31) & (y >> 31) & (~((x + y) >> 31)))
    - 양수끼리의 덧셈의 경우도 마찬가지로 x와 y가 모두 양수인지 확인하고, 이후 x+y가 음수인지 확인한다. 이는 4-1에서 언급한 >> 연산을 통해 구현할 수 있다. (~(x >> 31)) & (~(y >> 31))은 x와 y가 모두 양수일 때만 모든 비트가 1이 된다. 또한 (x+y) >> 31는 x+y가 음수일 때 모든 비트가 1이 된다. 이 둘을 &로 묶으면 양수끼리 더하여 음수가 되었을 때 모든 비트가 1이 되고, 그렇지 않은 경우엔 0이 된다. 이 결과값에 논리 연산자 를 적용하면, overflow가 일어나지 않은 경우 1, 그렇지 않은 경우 0이 된다.
        - (~(x >> 31) & (~(y >> 31)) & ((x + y) >> 31))
- 두 경우 모두에서 overflow가 일어나지 않았다면 1이, 그렇지 않으면 0이 반환되어야 하므로 이 두 경우를 &로 묶어 반환하면 addOK 함수가 구현된다. 이는 , >>, &, ~, +를 18번 사용한 것이다.
    - ((x >> 31) & (y >> 31) & (~((x + y) >> 31))) & (~(x >> 31) & (~(y >> 31)) & ((x + y) >> 31))

4-3. Implementation.

```
int addOK(int x, int y){
    return !((x >> 31) & (y >> 31) & (~((x + y) >> 31)))
& !((~(x >> 31)) & (~(y >> 31)) & ((x + y) >> 31));

}
```

Listing 3. Code of addOK.

- 앞선 4-2에서 설명한 바와 같이 addOK 함수를 구현할 수 있다.

5. Question #4. absVal

5-1. Explanation.

- 주어진 int형 변수 x의 절댓값을 반환하는 함수 absVal을 구현해야 한다. 이때 , ~, &, ^, |, +, <<, >> 연산을 최대 10번 사용할 수 있다.
- -x = ~x + 1이다.

5-2. Solution.

- 4-1에서 보았던 부호 비트와 비트 연산자을 활용하여, addOK 함수를 구현할 수 있다.
- Question #3과 같이 양수인 경우와 음수인 경우를 나누어 구현한다.
    - x & (~(x >> 31))를 통해 x가 양수인 경우에 x가, x가 음수인 경우에 0이 나오도록 한다. 이는 ~(x >> 31)이 x가 양수일 때 모든 비트가 1, 음수일 때 0이기 때문에 가능하다.
    - (~x + 1) & (x >> 31)를 통해 x가 음수인 경우에 -x가, x가 양수인 경우에 0이 나오도록 한다. 이는 (x >> 31)이 x가 음수일 때 모든 비트가 1, 양수일 때 0이기 때문에 가능하다.
- 이 두 가지 경우를 | 로 묶어 반환하면 absVal 함수가 구현된다. 이는 &, ~, >>, |, +를 8번 사용한 것이다.
    - (x & (~(x >> 31))) | ((~x + 1) & (x >> 31))

5-3. Implementation.

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

}
```

Listing 4. Code of absVal..

- 앞선 5-2에서 설명한 바와 같이 absVal을 구현할 수 있다.

6. Question #5. logicalShift

6-1. Explanation.

- 주어진 int형 변수 x, n에 대하여 logical shift로x를 n번 right shift한 결과를 반환하는 함수 logicalShift를 구현해야 한다. 이때 , ~, &, ^, |, +, <<, >> 연산을 최대 20번 사용할 수 있다.
- logical shift는 shift하며 생긴 빈 공간의 비트를 0으로 채우는 것이다.
- 기본적으로 c에서 지원하는 shift 연산은 arithmetic shift이다. 이는 right shift를 했을 때 빈 공간의 비트를 부호 비트로 채우고, left shift를 했을 때 빈 공간의 비트는 0으로 채운다. 즉, arithmetic shift로 left shift한 결과는 logical shift의 결과와 같다. [2]
- c언어에서 int 형 변수에 left/right shift를 한 번에 최대 31번 할 수 있다. (32로 나눈 나머지만큼) [3]
- -x = ~x + 1에서, -x - 1 = ~x 이다.

6-2. Solution.

- 6-1에서 보았던 left shift와 ~x = -x - 1을 이용하면 logicalShift 함수를 구현할 수 있다.
- 먼저, arithmetic shift로 x를 n번 right shift 한다.
- 이때 맨 왼쪽부터 차례대로 n개의 비트가 부호 비트로 채워지게 되며, 이를 0으로 변경하면 된다. 즉, 맨 왼쪽부터 차례대로 n개의 비트가 0이고 나머지 비트가 전부 1인 수와 & 연산을 하면 된다.
    - x >> n
- 이 수를 만들기 위해 ~0을 하여 모든 비트를 1로 바꾼다. 이후 left shift를 32 - n번 하여 맨 왼쪽부터 차례대로 n개의 비트가 1, 나머지 비트가 전부 0인 수를 만들고, 이 수에 ~연산을 취하면 된다. 이때 주의할 점은 n이 0인 경우 left shift를 한 번에 32번 진행하므로, shift 연산이 진행되지 않는다는 것이다. 이를 해결하기 위해 먼저 32 + (~n)번 left shift를 한 후에 다시 한 번 left shift를 진행한다. 이는 ~n = -n - 1인데, 0 <= n <= 31이기 때문에 0 <= 32 + (~n) <= 31이 되므로 가능하게 된다.
    - ~(((~0) << (32 + (~n))) << 1)
- 이 두 가지 결과 나온 수를 & 로 묶어 반환하면 logicalShift 함수가 구현된다. 이는 >>, &, ~, << , +를 8번 사용한 것이다.
    - (x >> n) & (~(((~0) << (32 + (~n))) << 1))

6-3. Implementation.

```
int logicalShift(int x, int n){
    return (x >> n) & (~(((~0) << (32 + (~n))) << 1));

}
```

Listing 5. Code of logicalShift..

- 앞선 6-2에서 설명한 바와 같이 logicalShift를 구현할 수 있다.

7. Results

```
[cotmdgus@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
[cotmdgus@programming2 datalab]$ ./dlc -e bits.c
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included from includable file /usr/include/stdc-predef.h.
dlc:bits.c:28:bitNor: 3 operators
dlc:bits.c:41:isZero: 1 operators
dlc:bits.c:55:addOK: 18 operators
dlc:bits.c:69:absVal: 8 operators
dlc:bits.c:83:logicalShift: 8 operators

Compilation Successful (1 warning)
```

Figure 1. Output of the grading program btest and programming rule checking program dlc.

Figure 1에서 확인할 수 있듯이 프로그래밍 규칙을 어기지 않으며, btest의 결과로 만점인 12점을 얻었다. dlc의 결과로 나오는 경고는, 내가 건드리지 않았던 부분에서 발생하였기 때문에 무시하였다.

8. References

[1]https://namu.wiki/w/%EB%93%9C%EB%AA%A8%EB%A5%B4%EA%B0%84%20%EB%B2%95%EC%B9%99

[2] Lecture Note of CSED211, “[CSED211 2025 Fall] Lecture 2 Bits, Bytes, Integers”

[3] https://stackoverflow.com/questions/7401888/why-doesnt-left-bit-shift-for-32-bit-integers-work-as-expected-when-used