CSED211: Data Lab1 Report

20200952 / Chaeyeon Jang

1. Overview

Data Lab1은 간단한 bitwise operation들을 구현하는 것이 목표이다. 이번 Lab은 총 5개의 문제가 있으며, 각각 bitNor, isZero, addOK, absVal, logicalShift 함수를 구현해야 한다.

2. Question #1. bitNor

2-1. Explanation.

- Question 1에서는 bitNor 함수를 구현해야 하는데, bitwise NOR 연산인 ~(x | y) 연산을 구현하면 된다.
- Legal ops: ~ &
- Max ops: 8, Rating: 1
- Table 1. shows the operation of the NOR operation.

	X = 0	**X = 1**
Y = 0	1	0
Y = 1	0	0

Table 1. Operation of NOR operation.

- 예를 들어, X = 0x6, Y = 0x5일 경우, bitNor(X, Y) = 0xFFFFFFF8이 된다.

2-2. Solution.

- 드모르간의 법칙에 따라, ~(x | y)는 ~x & ~y로 나타낼 수 있다.
  - 드모르간의 법칙: 논리 연산자의 and, or, not 연산 간의 관계를 설명하는 법칙으로, 아래 두 가지 법칙이 존재한다.
  - ¬(A∧B)=¬A∨¬B
  - ¬(A∨B)=¬A∧¬B

2-3. Implementation.

- We can implement the above equation with C as shown in Listing 1.

```c
int bitNor(int x, int y) {
  //Based on De Morgan's Law, ~(x|y) is equal to ~x | ~y
  return ~x & ~y;
}
```

Listing 1. Implementation of bitNor function.

3. Question #2. isZero

3-1. Explanation.

- Question 2에서는 isZero 함수를 구현해야 하는데, 함수의 input인 x가 0일 경우 1을, 0이 아닐 경우 0을 반환하도록 구현하면 된다.
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 2, Rating: 1
- 예를 들어, isZero(5) = 0, isZero(0) = 1이다.

3-2. Solution.

- 함수의 input x가 0과 같으면 x | 0 연산을 했을 때 결과가 0이 나오는 것을 알 수 있다. 반대로 x가 0이 아닌 수라면 x | 0 연산을 했을 때 0이 아닌 수(x 값)가 나오는 것을 알 수 있다.

- 이때 x가 0이면 1을 반환, 0이 아니면 0을 반환해야 하는데 x와 0의 bitwise or 연산 결과는0일 경우 1을, 0이 아닐 경우 0이 아닌 수를 도출하는 것을 볼 수 있다. 여기서 x와 0의 bitwise or 연산에 not을 붙이면 x가 0일 경우 1을 반환, 0이 아닐 경우 0을 반환하며 원하는 결과를 도출한다.

3-3. Implementation.

```
int isZero(int x) {
  /*
  by executing x | 0x0, we get non zero result if x is not eqaul to 0,
  and get zero if x == 0. By adding ! operation for negation, we can get
  appropriate result.
  */
  return !(x | 0x0);
}
```

Listing 2. Implementation of isZero function.

4. Question #3. addOK

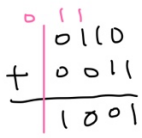
4-1. Explanation.

- Question3에서는 addOK 함수를 구현해야 하는데, 이는input인 x와 y에 대해 x + y를 overflow 없이 계산할 수 있는지 여부를 판단하여 결과를 반환하는 함수이다. Overflow가 발생할 경우 0을 반환하고, overflow가 발생하지 않을 경우 1을 반환한다.
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 20, Rating: 3
- 예를 들어, addOK(0x80000000, 0x80000000) = 0, addOK(0x80000000, 0x70000000) = 1이다.

4-2. Solution.

- Overflow가 발생하는 경우는 carry의 형태를 통해 알 수 있는데, 아래 두 가지 경우가 존재한다.
  - 앞의 두 carry가 01인 경우

    [Figure 1.]



  - 앞의 두 carry가 10인 경우

    [Figure 2.]

- 위의 overflow가 발생하는 경우를 통해, overflow가 발생하지 않는 경우는 아래 두 가지 경우임을 알 수 있다.
    - x와 y의 부호가 다를 경우
    - x와 y의 부호가 같지만, x + y의 부호와 x(또는 y)의 부호가 같을 경우
- overflow가 발생하지 않는 위 두 가지 경우를 구현하여 x + y가 위 두 가지 경우에 해당되는지 여부를 변수에 저장한 후, 두 가지 변수를 bitwise or 연산을 해주면 원하는 결과를 얻을 수 있다. 만약 x + y 계산이 위 두 가지 경우에 모두 해당되지 않아 overflow가 발생한다면 0 | 0 = 0이 되어 0을 반환할 것이고, x + y 계산이 위 두 가지 경우 중 하나라도 해당하여 overflow가 발생하지 않는다면 1을 반환할 것이다.

4-3. Implementation.

```
int addOK(int x, int y) {
  //below are the cases that overflow doesn't occur
  int case1 = ((x >> 31) & 1) ^ ((y >> 31) & 1); //case that sign of x and y
are different
  int case2 = !(((x >> 31) & 1) ^ (((x + y) >> 31) & 1)); //case that sign of x
and x + y are same
  return case1 | case2;
}
```

Listing 3. Implementation of addOK function.

- 이때, x부호와y 부호가 같은지 아닌지를 판단하기 위해 x를 31만큼 right shift 해주었고, 만약 x가 음수라면 arithmetic right shift 결과 앞의 31개 bit가 1로 채워질 것이므로 & 1 연산을 통해 앞의 31개 bit가 0이 되도록 해주었다. y도 같은 과정을 거쳐 LSB를 제외한 모든 bit가 0이 되도록 만들어주었다.
- 처리를 거친 x와 y의 부호(가장 마지막 bit)를 비교하기 위해 bitwise xor(^) 연산을 사용하여 부호가 같으면 0을, 부호가 다르면 1을 case1에 저장해주었다.
- Case2에서는 x의 부호와 x + y의 부호를 xor 연산을 사용하여 비교하였고, 부호가 같아야 overflow가 발생하지 않는 것이므로 xor 연산 결과에 ! 연산을 추가하여 부호가 같으면 1을, 부호가 다르면 0을 case2에 저장해주었다.
- 변수 case1과 case2 모두 overflow가 발생하지 않는 case에 해당되면 1, 해당되지 않으면 0을 저장하고 있으므로, case1 | case2 연산을 통해 둘 중 하나의 케이스라도 해당되면 overflow가 발생하지 않는 것이므로 1을, 둘 중 어느 케이스에도 해당되지 않으면 overflow가 발생하는 것이므로 0을 반환해주었다.

5. Question #4. absVal

5-1. Explanation.

- Question4에서는 absVal 함수를 구현해야 하는데, 이는 input x의 절대값을 반환하는 함수이다.
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 10, Rating: 4

- 예를 들어, absVal(-1) = 1이다.

5-2. Solution.

- Lecture 2 수업자료 p.33을 보면, ~x + 1 = -x 공식을 볼 수 있다. 이는 2's complement 방식에서 성립되는 공식이다.
- 위의 공식을 통해 음수가 들어왔을 경우 절댓값을 계산할 수 있다. x = ~(-x -1) 이므로 음수 input에 -1을 더해주고 그 결과값에 ~ 연산을 취해주면 된다. 이때, -1은 0xffffffff이므로 ~(input + 0xffffffff) 연산을 해주면 음수의 절댓값을 얻을 수 있다.
- 이때, 양수인 input이 들어올 경우 별도의 연산 없이 그대로 input을 반환해주면 되므로 input이 양수인 경우 x 값을 그대로 저장해주고, input이 음수인 경우 ~(input + 0xffffffff)의 연산을 해주어 결과값을 저장해주면 된다.

5-3. Implementation.

```
int absVal(int x) {
  int all1 = (1 << 31) >> 31;
  int isNeg = ~(x + all1) & (x >> 31);
  int isPos = ~(x >> 31) & x;
  return isNeg | isPos;
}
```

Listing 4. Implementation of absVal function.

- 우선, dlc 의 programming rule 에 의해 0xffffffff(= -1)는 사용할 수 없으므로 이를 만들어주기 위해 1 을 31 만큼 left shift 한 후, 이를 31 만큼 arithmetic right shift 해주어 all1 변수에 저장해주었다. 그러면 모든 bit 가 1 인 수를 얻을 수 있다.
- 이때, input x 가 음수라면 x 를 31 만큼 arithmetic right shift 했을 때 모든 bit 가 1 이 된다. 그래서 (x >> 31) & ~(x + all1) operation 을 통해 x 가 음수일 경우 isNeg 변수에 ~(x + all1) 연산 결과를 그대로 저장해준다. 반대로 x 가 양수라면 x 를 31 만큼 arithmetic right shift 했을 때 모든 bit 가 0 이 되어서 isNeg 에 0 이 저장된다.
- 또한, x 가 양수라면 x 그대로 반환해주면 되므로 ~(x >> 31) & x 를 통해 x 값을 그대로 isPos 변수에 저장해준다. x 가 양수면 ~(x >> 31)이 모든 bit 가 1 인 수가 되기 때문이다. 반대로 x 가 음수라면 ~(x >> 31)이 모든 bit 가 0 인 수가 되므로 0 이 isPos 에 저장된다.
- 최종적으로 isNeg | isPos operation 을 통해 x 가 음수면 isNeg 에 저장된 값을, x 가 양수면 isPos 에 저장된 값을 반환해준다.

6. Question #5. Logical shift

6-1. Explanation.

- Question5 에서는 logicalShift 함수를 구현해야 하는데, 이는 input x 와 input n 에 대해 x 를 n 만큼 logical right shift 한 결과를 반환하는 함수이다.
- Legal ops: ! ~ & ^ | + << >>

- Max ops: 20, Rating: 3
- 예를 들어, logicalShift(0x87654321, 4) = 0x08765432 이다.

6-2. Solution.

- Input x 가 양수일 경우는 MSB 가 0 이므로, arithmetic right shift 와 logical right shift 의 결과가 동일하다. 그러나 x 가 음수일 경우는 MSB 가 1 이므로, n 만큼 shift 했을 경우 arithmetic right shift 는 앞의 n 개의 bit 가 1 이 되고, logical right shift 는 앞의 n 개의 bit 가 0 이 된다.
- 그렇기 때문에 x 를 >>n(arithmetic right shift) 한 결과에 앞의 n 개의 bit 가 0 이고 나머지 비트가 1 인 수를 bitwise and(&) operation 하여 앞의 n 개의 bit 를 0 으로 만들어주고, 나머지 bit 는 x >> n 결과와 같도록 해주면 logical right shift 를 구현할 수 있다.

6-3. Implementation.

```
int logicalShift(int x, int n) {
  int frontN1 = ~(((1 << 31) >> n) << 1); //to create the integer whose n most
front bits are 1 and others are 0. e.g., if n = 3, 11100000...000
  return (x >> n) & frontN1;
}
```

Listing 5. Implementation of logicalShift function.

- 우선, input x 와 Input n 에 대해 앞의 n 개의 bit 가 1 이고, 나머지 bit 가 0 인 수를 만들어 frontN1 변수에 저장하였다. 먼저 1 을 31 만큼 left shift 하고, 이를 n 만큼 right shift 하였는데, 그러면 앞의 n + 1 개의 bit 가 1 이고 나머지가 0 인 수가 만들어지므로 다시 1 만큼 left shift 를 하였다.
- (x << n) & frontN1 연산을 통해 (x << n) 값에서 앞의 n 개의 bit 가 0 이 되도록 해주었다.

7. Results

Figure 3. shows the output of the programming rule checking program `dlc`. And Figure4. shows the output of the grading program `btest`. I get 12 points without violating programming rules.

[Figure 3.]

```
[jcy2749@programming2 datalab]$ ./dlc bits.c
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included from includab
le file /usr/include/stdc-predef.h.

Compilation Successful (1 warning)
```

[Figure 4.]

```
[jcy2749@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
```

8. References

[1] De Morgan’s Law. From ChatGPT.

드모르간의 법칙 설명

드모르간의 법칙은 두 가지 주요 형태로 정의됩니다. 이 법칙은 논리 연산자 `AND`와 `OR`에 대한 부정(NOT)의 분배를 설명합니다:

1. **첫 번째 법칙:**
   - 부정된 논리곱은 논리합의 부정과 같다.
   - 수식:
     $$\neg(A \land B) = \neg A \lor \neg B$$
   - 설명: 논리식 `A AND B`의 부정은 `A`의 부정과 `B`의 부정의 논리합(OR)과 같습니다.
2. **두 번째 법칙:**
   - 부정된 논리합은 논리곱의 부정과 같다.
   - 수식:
     $$\neg(A \lor B) = \neg A \land \neg B$$
   - 설명: 논리식 `A OR B`의 부정은 `A`의 부정과 `B`의 부정의 논리곱(AND)과 같습니다.