2019-2학기

컴퓨터 SW시스템 개론

Lab #1

20180085 송수민

#1 Previous Week

I. Why bits?

-> 두 상태의 정보를 쉽게 저장하기 위해, 부정확하거나 잡음이 있는 전선에서도 원활하게 전달하기 위해.

1 byte = 8 bits

II. Bit-level manipulations

> &(And) A=1 and B=1 -> A&B=1

|(Or) A=1 or B=1 -> A|B=1

~(Not) ~A=1 -> A=0

^(Xor) A=1 or B=1, but not both -> A^B=1

Logical Operators와 차이점은 Logical Operators는 언제나 return값이 0 또는 1이라는 것이다.

Left Shift & Right Shift Operations

i) x<<y(Left Shift) : y만큼 x의 왼쪽을 버린 후, x의 오른쪽에 y만큼 0으로 채워 넣는다.

ii) x>>y(Logical Shift) : y만큼 x의 오른쪽을 버린 후, x의 왼쪽에 y만큼 0으로 채워 넣는다.

iii) x<<y(Arithmetic Shift) : y만큼 x의 오른쪽을 버린 후, x의 왼쪽에 x의 MSB로 y만큼 채워 넣는다.

III. Signed, Unsigned, Two's Complements

Unsigned와 Two's Complements의 표현 범위는 서로 다르며 다음과 같은 식이 성립한다.

$$-\mathrm{UMax} = 2\mathrm{TMax} + 1$$

$$-|\mathrm{TMin}| = TMax + 1$$

Unsinged와 Signed 자료형 사이에는 Implicit casting이 적용되며, Unsigned와 Signed가 섞여서 비교가 되어진다면 Signed 값은 Unsigned로 Implicitly하게 Casting된다.

IV. Sign Extension

> 만약 크기가 작은 자료형을 크기가 더 큰 자료형에 대입한다면, 증가된 bit에 원래 MSB가 복사 된다.

> 그와 반대로 크기가 큰 자료형을 크기가 작은 자료형에 대입하고자 하면, 큰 숫자에 경우 예기치 못한 결과가 나올 수 있다.

#2 Source Code

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

Bitor 함수에 대해서는 드모르간의 법칙을 사용하였다. ($A \cup B = (A^C \cap B^C)^C$)

위 법칙을 사용하여 구현하였다.

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

logicalShift 함수에 대해서는 하나의 임시변수를 활용하여 구현하였다. C언어에서는 >>가 Arithmetic Shfit로 구현되는데, 이를 LogicalShift의 결과값으로 바꾸기 위해서는 새로 생성된 부분을 모두 0으로 바꾸어 주어야 한다. util이라는 임시의 11111/0000000값을 만든다. /의 기준은 n에 의해 결정된다. 이 값에 ~을 취해주면 00000/11111111...값이 만들어지고, 이와 shift된 x를 &연산을 취해주면 /오른쪽 부분은 0이였으면 0&1 = 0이고, 1&1이면 1이므로 원래의 값이 유지되고, /왼쪽 부분은 0&X이므로 반드시 0이 되므로 logicalShfit가 최종적으로 구현된다.

```
int bitCount(int x) {
int forTest=0x1;
int result=0;
forTest|=forTest<<8;
forTest|=forTest<<16;
result=forTest&x;
result+=forTest&(x>>1);
result+=forTest&(x>>2);
result+=forTest&(x>>3);
result+=forTest&(x>>4);
result+=forTest&(x>>5);
result+=forTest&(x>>6);
result+=forTest&(x>>7);
result+=result>>16;
result+=result>>8;
result&=0xFF;
  return result ;
}
```

bitCount는 막연히 생각하면 한자리 씩 shfit하여 검사할 수 있지만, divide-conquer 알고리즘을 사용하여 4비트 단위로 검사를 진행할 수 있다. 위 방법을 적용하면 매번 검사해야 할 과정을 줄여서 검사할 수 있으므로 효율적이다.

```
int negate(int x) {
  return ~x+1;
}
```

Negate 함수는 2의 보수를 이용하여 구현하였다. (CSED211-2019-02, slide pg. 36)

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

AddOK 함수는 두 수를 더하였을 때, overflow가 일어나는지에 대한 함수이다. Overflow가 일어나면 0을 리턴, 일어나지 않으면 1을 리턴하도록 하여야 한다. Overflow의 발생여부는 결국 MSB와 직접적인 관련이 있다. 합과 x,y의 MSB를 구하여 이를 이용하여 판단한다. 예를 들어, 4자리의 2진수가 있고 왼쪽자리는 MSB이라고 하자. 만약 6+7연산을 한다면 0110+0111=1101이고, 위 코드대로 따라간다면 0000,0000,1111이라고 할 수 있다. return값 식에 다음 값을 대입하여 연산해보면 Overflow라는 결과가 나오고 실제로 위 연산은 Overflow이다. 일반적으로 2의 보수를 이용한 더하기 연산에서는 MSB 위에 있는 Carry와 그 옆의 왼쪽의 Carry가 서로 다르면 Overflow가 발생한다. (2019-1학기 디지털시스템설계 Chapter 5)