2018 시스템 프로그래밍
- Lab 03 -

제출일자	2018.10.16
분 반	02
이 름	진승언
학 번	201404377

실습 1 [bitAnd]

```
168  * bitAnd - x&y using only ~ and |
169  *   Example: bitAnd(6, 5) = 4
170  *   Legal ops: ~ |
171  *   Max ops: 8
172  *   Rating: 1
173  */
174 int bitAnd(int x, int y) {
175     return ~(~x | ~y);
176 }
```

bitAnd는 ~와 | 만 사용해서 두 수의 and비트연산을 하는 함수다. 디지털논리회로때 배웠던 것을 이용해서 X+Y(=XandY)에 드모르간을 두 번 씌우고 한번 풀어주면 nXnY에 위에 ㅡ(불대수)가 있는 상태이다. 이걸 그대로 표현하면 ~(~x | ~y)가 된다.

[getByte]

bits.c (~/datalab-handout) - VIM

```
177 /*
178  * getByte - Extract byte n from word x
179  *   Bytes numbered from 0 (LSB) to 3 (MSB)
180  *   Examples: getByte(0x12345678,1) = 0x56
181  *   Legal ops: ! ~ & ^ | + << >>
182  *   Max ops: 6
183  *   Rating: 2
184  */
185 int getByte(int x, int n) {
186   return (x >> ( n << 3 )) & (0xFF);
187 }
```

(n<<3)에서 8bit * n 만큼, 즉 n바이트를 의미한다. 그걸 x>>(n<<3) 해서 입력값 x를 n 이트만큼 우측 shift시킨다. 그럼 내가 추출할려는 n바이트의 데이터가 LSB부분에 온다. 그리고 내가 추출할려는 바이트 부분 남게 하고 상위에 있는 나머지 비트들은 버려주면 된다. 이것은 위에서 계산한 것에 0xFF와(1111 1111) &비트연산을 해서 구현할 수 있다. 내가 추출할려는 바이트 단위의 데이터는 1111 1111과 &비트 연산을 함으로 그대로 보존되고 나머지 상위비트들은 0과 &비트연산을 하는 것이므로 다 0으로 되게 된다. 이렇게 내가 원하는 바이트 부분을 얻을 수 있다.

[logicalShift]

bits.c (~/datalab-handout) - VIM

```
188 /*
189  * logicalShift - shift x to the right by n, using a logical shift
190  *   Can assume that 0 <= n <= 31
191  *   Examples: logicalShift(0x87654321,4) = 0x08765432
192  *   Legal ops: ! ~ & ^ | + << >>
193  *   Max ops: 20
194  *   Rating: 3
195  */
196 int logicalShift(int x, int n) {
197   return (((x&(0x8<<28))>>n) <<1)^(x>>n);
198 }
```

x= 0x87654321

(0x8<<28) => 1000 0000 0000 0000 0000 0000 0000 0000

(0x8<<28)을 하는 이유는 n비트씩 오른쪽으로 시프트를 하는 함수이고 최상위 비트가 0이냐 1이냐에 따라 우측 시프트 연산했을 때 최상위비트 부분이 1로채워지냐 0으로 채워지냐가 결정되기 때문에 필요하다.

x가 1000 0111 0110 0101 0100 0011 0010 0001이므로

(0x8<<28) 1000 0000 0000 0000 0000 0000 0000 0000 과 &연산하므로

x& (1<<31) => 1000 0000 0000 0000 0000 0000 0000 0000 된다

이것은 이제 n bit만큼 우측shift 할 때의 기준으로 할 수 있다.

만약 n = 4라면 1111 1000 0000 0000 0000 0000 0000 0000 된다

위의 것에서 <<1을 해야한다. 왜냐하면 기존에 최상위비트에 1이 있었으므로

<<1을 안하면 4비트가아니라 5비트를 우측shift한 게 되기 때문이다.

그래서 <<1을 하여 1111 0000 0000 0000 0000 0000 0000 0000 된다.

이것을 (x>>n)과 ^(xor)연산을 해주면 된다.

(x>>n) => 1111 1000 0111 0110 0101 0100 0011 0010 ,0xF8765432

이것과 => 1111 0000 0000 0000 0000 0000 0000 0000 를

^(XOR)연산해주면 0000 1000 0111 0110 0101 0100 0011 0010이 됨으로서

4비트가 우측 시프트 된 결과가 나온다.

[bitCount]

bits.c (~/datalab-handout) - VIM

```
201 *     Examples: bitCount(5) = 2, bitCount(7) = 3
202 *     Legal ops: ! ~ & ^ | + << >>
203 *     Max ops: 40
204 *     Rating: 4
205 */
206 int bitCount(int x) {
207   int result;
208   int y = 0x1;
209   y |= y<<8;
210   y |= y<<16;
211   result = x & y;
212   result += (x>>1) & y;
213   result += (x>>2) & y;
214   result += (x>>3) & y;
215   result += (x>>4) & y;
216   result += (x>>5) & y;
217   result += (x>>6) & y;
218   result += (x>>7) & y;
219   result += result >> 16;
220   result += result >> 8;
221   return (result & 0xFF);
222 }
```

이것은 x를 2진수로 표현 했을 때 1의 개수를 출력하면 되는 문제이다. 먼저 개수를 담을 result변수를 먼저 선언하고(코드 중간에 선언하면 왜 그런지 모르겠지만 에러가 난다.) y라는 변수를 선언해서 0x1에서 8비트 왼쪽시프트해서 OR연산 해주고 저장하고 이것을 다시 16비트 왼쪽시프트를 해주어서 y에 0000 0001 0000 0001 0000 0001 0000 00001을 저장하게 만들었다.(이것도 처음부터 int y = 0x01010101로 선언 했을 때 ./btest는 되는데 ./driver.pl 했을 때 에러가 났다.) 그리고 먼저 선언한 result 변수에 입력값 x와 y를 &연산을 함으로서 바이트 단위로 나눠서 봤을 때 1씩 더해지게 만들어서 저장시켰다. 그리고 x를 우측시프트를 한 칸씩 하면서 이 과정을 바이트 8자리 모두 검사할 수 있게 x와 y의 &연산을 총 8번 반복하였다. 그럼 모든 자리의 비트1의 개수가 바이트단위로 나뉘어서 result에 저장되있을 것이다. 이 문제는 4바이트를 기준으로 했으니까 그럼 result는 1바이트 4개로 나뉘어서 개수가 저장되있는 상태이다. 그럼 총 개수를 구하기MSB부분에 있는 바이트들을 LSB바이트로 옮겨서 개수를 구하게 만들어야한다. 이것은 처음에 1바이트를 기준으로 0000 0001을 만들었으니까 현재 4바이트로 개수가 담겨져있는 것을 1바이트로 묶어주면 개수가 나온다. 그래서 reuslt를 먼저 >>16해서 반으로 나눠 더한 후 (16비트 길이의 데이터가 남음), 이것을 한번 더 반으로 나눠서 result에 저장하면 네 바이트로 나눠서 저장 되있던 개수가 한 바이트로 합해지고 LSB부분으로 옴으로 이 자체로 총 개수를 구할 수 있게 된다.

[isZero]

bits.c (~/datalab-handout) - VIM

```
209 //#include "bang.c"
210 //#include "tmin.c"
211 /*
212  * isZero - returns 1 if x == 0, and 0 otherwise
213  *   Examples: isZero(5) = 0, isZero(0) = 1
214  *   Legal ops: ! ~ & ^ | + << >>
215  *   Max ops: 2
216  *   Rating: 1
217  */
218 int isZero(int x) {
219   return !(x);
220 }
```

입력값 x가 0이면 1이 나오게 해야 하므로 0 일 때 !(논리반전)하면 1이 나오게 된다. 논리반전은 해당값이 0이면 1 아닌 경우는 0이 되게 한다. 그러므로 x가 0이 아닌 경우는 1이 나오게 된다.

[isEqual]

bits.c (~/datalab-handout) - VIM

```
221 /*
222  * isEqual - return 1 if x == y, and 0 otherwise
223  *   Examples: isEqual(5,5) = 1, isEqual(4,5) = 0
224  *   Legal ops: ! ~ & ^ | + << >>
225  *   Max ops: 5
226  *   Rating: 2
227  */
228 int isEqual(int x, int y) {
229   return !(x^y);
230 }
```

입력값 x와 y를 ^(XOR)하면 둘이 같은 비트부분은 0이 나올 거고 다른 비트부분은 1이 나올 것이다. 그럼 둘이 비트 전체가 같다면 모든 비트가 0이 될 것이다. 즉, equal인 경우는 0일 것이고 equal하지 않은 경우는 0이 아니다. 하지만 이 문제에서 equal한 경우는 1을 반환해야하고 아닌 경우는 0을 반환해야한다. 그럼 isZero에서 했던 거를 응용 해주면 된다. 여기에 !(논리반전)을 붙여서 0이면 1이 나오게 하고 0이 아닌 경우는 0이 반환되게 하면 된다.

[fitBits]

bits.c (~/datalab-handout) - VIM

```
245 /*
246  * fitsBits - return 1 if x can be represented as an
247  *  n-bit, two's complement integer.
248  *   1 <= n <= 32
249  *   Examples: fitsBits(5,3) = 0, fitsBits(-4,3) = 1
250  *   Legal ops: ! ~ & ^ | + << >>
251  *   Max ops: 15
252  *   Rating: 2
253  */
254 int fitsBits(int x, int n) {
255     int tmp = 32 + (~n+1);
256     int result = x << tmp;
257     result = (result >> tmp)^x;
258     return !result;
259 }
```

먼저 이 함수는 n비트 2의 보수로 표현 할 수 있는 범위안에 x가 포함되어 있는지를 판단한다. 포함되있으면 1을 리턴한다.

예를 들어 fitsBits(5,3)이면 3비트 2의 보수로 표현 할 수 있는 범위는 -4부터 3까지인데 5는 포함되지 않으므로 0을 반환한다. fitsBits(-4,3)인 경우에는 3비트 2의 보수 범위가 -4부터 3인데 -4는 범위안에 포함됨으로 1을 리턴한다.

먼저 tmp에 shift할 횟수를 저장하게한다. 그다음 result에 x를 왼쪽으로 tmp만큼 shift시킨다. 그리고 result를 전 코드와 반대로 tmp만큼 오른쪽으로 shift시키고 이것을 shift연산을 하기 전인 입력 x값과 같은지 ^(XOR)연산으로 비교한다. 같다면 0이 리턴 될 것이다. 같다는 것은 x가 -(n+1)~(n)범위에 있다는 것을 뜻한다. 마지막으로 현재 범위에 있다면 1이 리턴되는 상태이므로 !연산을 이용해서 0이 아닌 1이 리턴 되게 한다.

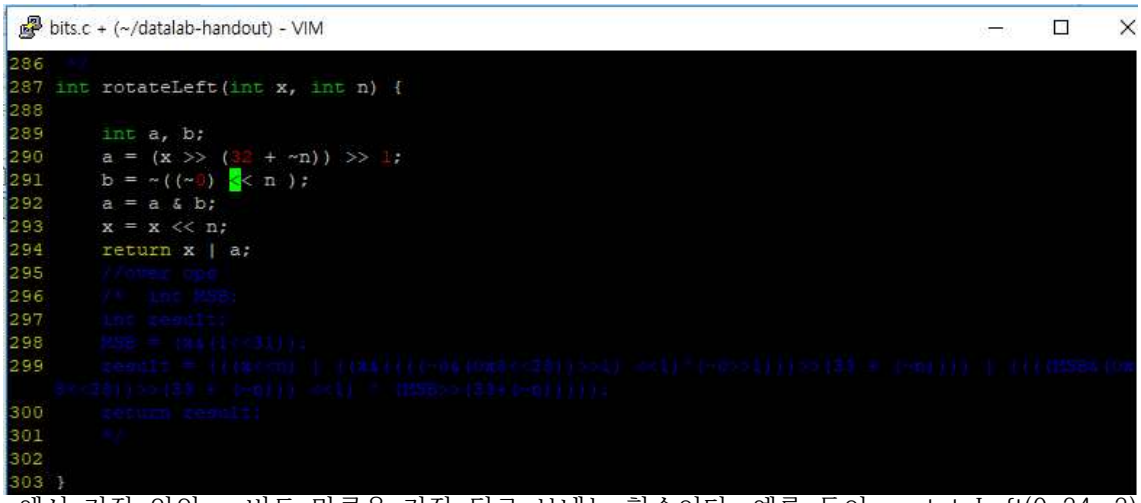
[isLessOrEqual]

bits.c (~/datalab-handout) - VIM

```
260 //#include "divpwr2.c"
261 //#include "negate.c"
262 //#include "isPositive.c"
263 /*
264  * isLessOrEqual - if x <= y  then return 1, else return 0
265  *   Example: isLessOrEqual(4,5) = 1.
266  *   Legal ops: ! ~ & ^ | + << >>
267  *   Max ops: 24
268  *   Rating: 3
269  */
270 int isLessOrEqual(int x, int y) {
271     int a = (x>>31) & 1;
272     int b = (y>>31) & 1;
273     int t1 = y + (~x + 1);
274     int t2 = (t1 >> 31) & 1;
275
276     int result = (a & !b) | (!(a ^ b) & !t2);
277     return result;
278 }
```

x가 y보다 작거나 같으면 1을 반환하고 반대면 0을 반환하는 함수이다. 변수 a에 입력값 x를 우측으로 31만큼 shift시킨 후 1과 AND연산을 한후 저장하고, b에도 입력값 y를 x와 같은 방법으로 저장한다. 이 연산을 통해 x, y의 부호를 안다. t1에는 x와 y를 뺀 값을 저장한다. 그리고 t2에 처음 x, y의 부호를 구해 a,b에 저장하는 방법과 똑같이 x와y 뺀 값(=t1)을 오른쪽으로 31번 shift하고 1과 AND연산을 해서 결과값의 부호를 저장한다. x<=y일 때 2가지의 경우의 수가 있는데 x가 음수, y가 양수일 때와 서로 같은 부호일 때 y가 더 클 때이다. 이 두 가지의 경우를 모두 해당될 수 있게 result에 |(OR)연산을 통해 구현하였다.

[rotateLeft]



x에서 가장 앞의 n 비트 만큼을 가장 뒤로 보내는 함수이다. 예를 들어 roatateLeft(0x34, 3)을 하면 0011 0100에서 가장 앞 비트인 3개(0011 0100)를 맨 뒤로 보내서 1010 0001이 된다. 16진수로 바꿔서 0xA1이 출력된다.

먼저 밑에 주석은 앞서 구현한 LogicalShift를 이용해서 구현해서 답은 맞았으나 연산자가 3개 오버 되서 위와 같이 코드를 바꾸었다. 아까워서 안 지웠다.

여기서 a 변수에서 (32 + ~n)이란 시프트할 횟수를 의미한다.(이 횟수는 32-n만큼을 의미 하는데뒤에서 입력 x값에서 n비트만큼 오른쪽으로 shift해야 rotate되는 n비트가 rotate된 자리에 올 수 있기 때문에 필요하다.) 그 후 a변수는 1칸 더 오른쪽으로 shift해준 값을 저장한다. b는 LSB부분을 위한 즉 n비트만큼 rotate한 자리를 위한 마스크로 만든다. 그리고 a와 b를 &(AND)연산시켜 b에 다시 저장하는데 a에 저장하는데 a에는 rotate된 n비트만 rotate된 위치에 되있는 상태이다. 그리고 입력값 x를 n비트만큼 왼쪽으로 shift하여 이 결과를 MSB에 있다가 LSB로 rotate된 비트들만 담고있는 a와 |(OR)연산을 하면 x의 n비트만큼의 LSB부분의 빈 부분에 rotate된 값이 채워지므로 roateLeft가 구현된다.