LOGIC I

1. Bitwise logic

Bitwise logical operations view an integer word in memory as an array of bools and perform a logical operation on all these bits simultaneously.

The following are bitwise logic operations in C++:

1. & and
2. | or
3. $\wedge$ exclusive or
4. $\sim$ not
5. << shift left (i.e. multiplying by a power of 2)
6. >> shift right (i.e. dividing by a power of 2)

The two lines

```
int n;
x = x << n;
```

has the effect of shifting the bits in x to the left n steps, padding new entries with zeros. Overflowing bits are discarded. This is equivalent to multiplying with 2^n.
Similarly,

```
int n;
x = x << n;
```

shift bits to the right, which is equivalent to dividing by 2^n and discarding the remainder.

On the assembly language level there are a other bitwise operations which are not implemented in C++, for instance rotate left, rotate right etc.

Given an integer x, where you want to clear bit number 4, you can write

```
x = x & (1 << 4);
```

If you want to set all bits except number 5

```
x = x | ~(1 << 5);
```

You can toggle bit 7 with

```
x = x ^ (1 << 7);
```

You can multiply a number by 8 by

```
x = x << 3;
```

since $8 = 2^3$.

We end with a more complicated exampl. We clearing the least significant bit

```
y = x & (x-1);
```

Lets look at how it works. First, if When $x = 0$, the expression evaluates to 0. Assume x is of the form $b..b10...0$ with an arbitrary number of zeros to the right of the first 1. Then $x-1$ is $b..b01...1$ and $x\&(x-1)$ is $b..b00...0$.

2. Two's complement

We can check if an integer is negative by looking at the most significant bit in an integer. If it is set then the number is negative, if not, it is positive. In binary, the operation $y = -x$ is given by $y = (\sim x) + 1$. Doing two's complement a second time, i.e. on y gives back the original number x.

Lets look at some examples with four and 16 bit numbers. First the two's complement of $0 = 0000$ is again 0, which is what you expect. Now, -1 is obtained by taking the two's complement of 0001 which is 1111. Similar for larger integers, e.g. in 16 bits, where -1 is $0xFFFF$. The number -2 is the two's complement of 0010 which is 1110 with four bits or $0xFFFE$ in 16 bits. So the negative numbers go down

from 1111 to 1000 with four bits or $0xFFFF$ to $0x8000$ for 16 bits.

There is a small issue with the smallest negative number $-16 = 1000$ (or $0x8000$) as the two's complement of these integers leave the number unchanged. So you cannot take the negative of the four bit number -16, or in other words: it is not possible to represent the positive number 16 with a four bit signed integer. This actually causes issues with the abs() function in some programming languages.

You might wonder why negative numbers are represented in this way, and I believe the main answer is that substraction and adding of negative numbers now be done with the same logical circuitry as usual addition. I give an example showing the calculation.

Example 1. *We work with four bit integers and compute* $5 - 6$*. The number* 5 *is represented by* 0101 *and* 6 *is* 0110*. The Two's complement of* 6 *is* 1010*. We add* $0101 + 1010$ *as usual and get* 1111*. The two's complement of* 1111 *is* 0001*, so* 1111 *represents* -1*.*

3. The Perceptron

A bool is usually represented by the smallest unit of addressable memory (and that is usually one byte). Of course a byte can be viewed as a number as well. As integers $(1 < 2)$ is the number 1 whereas $(1 > 2)$ is the number 0.

It therefore makes sense to for instance do the following.

```
(1<2)*2+4;
```

which would evaluate to 6.

The if statement

```
float w1,w2.w3;
bool u,v;

if(w1*u + w2*v>w3) {
  \\ stuff
}
```

can be made to be equal to any of the following if statements

```
if(u)
if(!u)
if(u && v)
if(!u && v)
if(!u && !v)
if(u || v)
if(!u || v)
if(!u || !v)
```

by adjusting the weights w1 and w2. Not all logical expressions in u and v can be realised. In particular it is not possible to do exclusive or. However, if we increase the depth of the expression by one, for instance

```
if((w1*u + w2*v>w3) * w4 + ((w5*u + w6*v>w7)*w8 > w9 )
```

we would be able to express any logical expression, for instance via its disjunctive normal form, including exclusive-or.

3.1. **The perceptron.**

```
class Perceptron
{
     float weights[3];
     float  lc = 0.01;
     int feedforward(float* input, int target);
     void train(float* inputs, int target)
}

int feedforward(float* input, int target)
{
      if(input[0]*weight[0] + input[1]*weight[1]+
            input[2]*weight[2]>0) return 1;
      else return -1;
}


void train(float* inputs, int target)
{
      int current = feedforward(inputs);
      float delta = lc * (target - current);
      weights[0] += delta * input[0];
      weights[1] += delta * input[1];
      weights[2] += delta * input[2];
 }
```

Exercises.

Exercise. 1. *Let x be an 16-bit integer. Do the following operation on x (with one line of code):*

a) *Set bit number* 4.
b) *Clear bit number* 5.
c) *Set bit number* 4 *and* 6.
d) *Clear bit number* 5 *and* 10.
e) *Set all bits except bit number* 10 *and* 11.
f) *Clear all bits except bit numer* 5 *and* 15.
g) *Toggle bit number* 8.
h) *Toggle all bits except bit* 9.

Exercise. 2. *Let x be an 16-bit integer. Do the following operation on x (with one line of code):*

a) *Set all bits in the least significant byte.*
b) *Clear all bits in the most significant byte.*
c) *Toggle all bits in the least significant byte.*

Exercise. 3. *Let x and y be 16-bit integers. Find out what the following code does. Explain!*

```
a) y = x & (x + 1);
b) y = x | (x + 1);
c) y = x | (x-1);
d) y = x ^ (x + 1);
```

Exercise. 4. *Let x, y and r be 8-bit integers. Find out what the following code does. Explain!*

```
a) r = y ^ ((x ^ y) & -(x < y));
b) r = x ^ ((x ^ y) & -(x < y));
```

Exercise. 5. *Let x and y be 16-bit integers. Find out what the followng three line code does to x and y. Explain!*

```
x ^= y;
y ^= x;
x ^= y;
```

Exercise. 6. *Find weights so that the if statement*

```
if((w1*u + w2*v>w3) * w4 + ((w5*u + w6*v>w7)*w8 > w9 )
```

expresses

a) $u \oplus v$ *(exclusive or)*

b) $(\overline{u} \vee v) \wedge \overline{(u \vee \overline{v})}$

Answers:

Exercise 1.

```
a) x = x | (1<<4);
b) x = x & (1<<5);
c) x = x | (5 << 4);
or x = x | (1 << 4) | (1<<6);
d) x = x | ~(3 << 10);
or x = x | ~(1 << 10 | 1 << 11);
e) x = x & ~(1 << 16 | 1 << 5);
f) x  = x & (1<<5 | 1<< 15);
g) x = x ^ (1 << 8);
h) x = x ^ ~(1<<9);
```

Exercise 2.

```
a) x = x | 0xFF;
b) x = x & 0xFF;
c) x = x ^ 0xFF;
```

Exercise 3.

a)

If x has the form $b...b0$, then $x+1$ is $b...b1$ and $x\&(x+1)$ is x. If x is of the form $b...b01..1$ then $x+1$ is $b..b10..0$ and $x\&(x+1)$ is $b...b00...0$.

In other words, the operation clears all bits set to 1, starting with the least significant bit up to the first bit set to zero.

b)

If x has the form $b...b0$, then $x+1$ is $b...b1$ and $x|(x+1)$ is has the form $b...b1$ If x is of the form $b...b01..1$ then $x+1$ is $b..b10..0$ and $x\&(x+1)$ is $b...b11...1$. If x is $0xFFFF$, then $x+1$ is zero, and $x|(x+1)$ is x.

In other words, the operation sets the least significant zero bit (if it exists).

c)

If x has the form $b...b1$ then $x-1$ is $b...b0$ and $x|(x-1)$ is x. If x has the form $b...b10...0$ then $x-1$ is of the form $b...b01...1$ and $x|(x-1)$ is of the form $b...b11..1$.

In other words, the operation sets all bits to 0, starting with the least significant bit and up to the first bit set to one.

d)

If x has the form $b...b0$, then $x+1$ is $b...b1$ and $x^(x+1)$ is 0..01. If x is of the form $b...b01..1$ then $x+1$ is $b..b10..0$ and $x\&(x+1)$ is 0..011...1.

In other words, the operation sets the least significant bit which is set to zero, and clears all higher bits and keeps all lower bits.

Exercise 4.

a) r = min(x,y)

If $x < y$ then $-(x < y)$ is -1 which is $0xFFFF$. So r evaluates to $y \oplus x \oplus y$ which is x since $y \oplus y = 0$. If $x \geq y$ then $-(x < y)$ is 0. So r evaluates to $y \oplus 0x000 = y$.

b) r = max(x,y)

If $x \leq y$ then $-(x < y)$ is -1 which is $0xFFFF$. So r evaluates to $x \oplus x \oplus y$ which is y since $x \oplus x = 0$. If $x > y$ then $-(x < y)$ is 0. So r evaluates to $x \oplus 0x000 = x$.

Exercise 5.

In order to make the explanation below a little clearer, lets rewrite the code as follows.

```
z = x ^ y;
w = y ^ z;
v = z ^ w;
```

and analyse step by step. After the first line

```
z = x^y;
```

z contains the following information. A bit is set if the value of x and y where different (01 or 10), and cleared if the value was the same (11 or 00). After the second line

```
w = y^z;
```

If a bit is set in w, it means that the values of y and z are different. If the bit in y is one, then the bit in z is zero, which means that the bit in x must be one. If the bit in y is zero, then the bit in z is one, which means that the bit in x must be one.

If a bit is cleared in w, it means that the values of y and z were the same. If the bit in y is zero, then the bit in z is zero and so the bit in x is zero. If the bit in y is one, then the bit in z is one, and so the bit in x is zero.

In either case, the bit is a copy of the corresponding bit in x, and so the whole of w is equal to x.

After the third line

```
v = z^w;
```

If a bit is set in v, then the values of the corresponding bits of z and $w = x$ are different. If the bit in x is 1, then the bit in z is zero, and so the bit in y is one. If the bit in x is 0, then the bit in z is 1, and so the bit in y is one.

If a bit is cleared in v, then the values of the corresponding bits of z and x are equal. If the bit in x is 1, then the bit in z is 1 and so the bit in y is zero. If the bit in x is zero, then the bit in z is zero, and so the bit in y is zero.

So the whole of v is a copy of y. The code

```
z = x ^ y;
w = y ^ z;
v = z ^ w;
```

has the effect of swapping x and y (without the use of a temporary variable).

Exercise 6.

a)

We write $u \oplus v$ as

$$(\overline{u} \wedge v) \vee (u \wedge \overline{v}).$$

The corresponding weights are for instance

```
w1 = -0.5
w2 = 0.3
w3 = 0.2
w4 = 0.5
w5 = 0.3
w6 = -0.5
w7 = 0.2
w8 = 0.5
w9 = 0.3
```

b)

Using boolean algebra, we rewrite to $(\overline{u} \vee v)$. A choice of weights are for instance

```
w1 = -0.2
w2 = 0.2
w3 = -0.1
w4 = 1.0
w5 = 0.0
w6 = 0.0
w7 = 0.0
w8 = 0.0
w9 = 0.0
```