Name : Muhammad Nauman Sheikh

Roll No : BSEF19M504

Computer Organization
&
Assembly Language

Assignment # 1

Instructor : Prof Muhammad Arif Butt

Q. No. 1 Convert into binary, decimal and octal

$68AC_{(16)}$

in binary convertion each digit ~~its~~ is converted into its respective binary notation

6	8	A	C
0110	1000	1010	1100

$= 0110\ 1000\ 1010\ 1100_{(2)}$

in octal

$\underbrace{000110}_{0\ 6}\ \underbrace{1000}_{4}\ \underbrace{1010}_{2\ 5}\ \underbrace{1100}_{4}$

$64254_{(8)}$

In the given binary converted number the most significant bit is 0. Hence the number is unsigned.

$0\times2^{15} + 1\times2^{14} + 1\times2^{13} + 0\times2^{12} + 1\times2^{11} + 0\times2^{10} + 0\times2^{9} + 0\times2^{8} + 1\times2^{7} + 0\times2^{6} + 1\times2^{5} + 0\times2^{4} + 1\times2^{3} + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$

$= 0 + 16384 + 8192 + 0 + 2048 + 0 + 0 + 0 + 128 + 32 + 0 + 8 + 4 + 0 + 0$

$= 16384 + 8192 + 2048 + 128 + 32 + 8 + 4$

$= 26796$

Q. No. 2 Different hardware

Sol: a) Sign Magnitude

We need different hardware for subtraction if using sign magnitude encoding, because, if ~~because~~ the numbers have different signs in addition then the most significant bit of the number of greater magnitude is used for the sign of ~~the~~ answer while the remaing magnitude is added simply. But in case of same MSBs, that bit is used in the answer. Both operations cannot be performed through same hardware.

b) 1's Complement vs 2's Complement

In 1's complement, the numbe 0 has two different representations and the range of both methods differ. While performing addition through 1's Complement the carry is further added to the answer, while in case of 2's complement the carry is ignored. Thus different hardware is need for each representation scheme.

Q. No. 3

```
# include<iostream>
# include <cmath>
using namespace std;
int main (){
    cout<< "Range of char: "<<-(int)pow(2,(sizeof(char)*8)-1);
    cout<< " to "<< (int) pow (2, (sizeof(char)*8)-1)-1 << endl;
    cout << "Range of unsigned char: 0 to ";
    cout << (int) pow (2, sizeof (char)*8) - 1 << endl;
    cout<< "Range of short: "<<-(int)pow(2,(sizeof(short)*8)-1);
    cout<< " to "<< (int) pow (2,(sizeof(short)*8)-1)-1 << endl
    cout<< "Range of unsigned short: 0 to ";
    cout<< (int)pow (2, sizeof(short)*8)-1 << endl;
    cout<< "Range of int : "<< +(int)pow(2, sizeof(int)*8);
    cout << " to " << (int) pow( 2, sizeof (int)*8)-1 << endl;
    cout << "Range of unsigned int: 0 to ";
    cout << (unsigned int) pow (2, sizeof (int)*8)-1 << endl;
    cout << "Range of long :"<<(long) pow(2, sizeof(long)*8);
    cout << " to " << (long) pow (2, sizeof(long)*8)-1;
    cout << " Range of unsigned long: 0 to ";
    cout << (unsigned long)pow (2,sizeof(long)*8)-1 << endl;
    cout << "Range of long long:"<< (long long) pow(2, sizeof(long long)*8)
    cout<< " to " << (long long) pow (2, sizeof (long long)*8)-1);
    cout<< endl << "Range of unsigned long long: 0 to ";
    cout<< (unsigned long long) pow (2,sizeof (long long)*8)-1;
    cout<< endl;
}
```

The hardware can effect on the range of different data types. The maximum number of a data type depends on the number of data lines of the processor. In a 16 bit arthitecher the range of integer is -2^{15} to $2^{15}-1$ for signed and 0 to $2^{16}-1$ for unsigned

Q. No. 4 Range of numbers (16-bits)

① 2's complement signed number

-32768 to $+32767$

$-(2)^{15}$ to $(2)^{15}-1$

② 1's complement signed number

-32767 to $+32767$

$-(2)^{15}+1$ to $(2)^{15}-1$

③ Unsigned number

0 to 65535

0 to $2^{16}-1$

Q. No. 5 Check for overflow and carry flag

(i) 0x 86 + 0x 84

Given the given hexadecimal numbers, convert into binary

$86_{(16)}$ $84_{(16)}$

1000 0110 1000 0100

Adding these numbers

$$\begin{array}{r} 1000\ 0110 \\ 1000\ 0100 \\ \hline 1\ 0000\ 1010 \end{array} = 0000\ 1010_{(2)} = 0x\,0A$$

Carry

Since the given numbers are 8 bits in size and a carry 1 is generated hence carry flag is raised. Since the MSB's of both binary numbers are same and the MSB of the added result is different hence overflow has occured.

MBs of both binary number is 1 and the numbers are in form of 2s compliment signed numbers so both numbers are negative and the obtained result has MSB 0 hence it is positive, then by definition

If two -ve binary numbers are added to get a positive number then this overflow is negative overflow.

b) 0x 7E + 0x 70

in binary

$7E_{(16)}$ + 70

0111 1110 + 0111 0000

$$\begin{array}{r} 0111\ 1110 \\ +\ 0111\ 0000 \\ \hline 1110111 0 \end{array}$$

No carry is generated, hence carry flag is not raised
MSB's of both numbers are same and MSB of result is different, hence overflow flag is raised

Since both numbers are +ve binary integers and the result is -ve 2's complement binary integer, then by definition

If two positive numbers are added to get a negative number the such overflow is positive overflow.

c) 0x F6 + 0x 7E

in binary

$F6_{(16)}$ + $7E_{(16)}$

$1111\ 0110_{(2)} + 0111\ 1110_{(2)}$

$$\begin{array}{r} 1111\ 0110 \\ +\ 0111\ 1110 \\ \hline 0111\ 0100 \end{array}$$

No carry bit is generated, hence carry flag is not raised
~~MBs~~ MSB's of both numbers are different hence no overflow occurs and ~~of~~ overflow flag is not raised

Q. No. 6 C program

```
# include <iostream>
using namespace std;
int main()
{
    int a,b;
    unsigned int m,n;
    cout << "Enter 2 signed integers: ";
    cin >> a >> b;
    cout << "Enter 2 unsigned integers: ";
    cin >> m >> n;
    if(a>0 && b>0 && a+b<0)
    {
      cout << "Positive overflow occured" << endl << a+b << endl;
    }
    else if (a<0 && b<0 && a+b>0)
    {
      cout << "Negative overflow occured" << endl << a+b << endl
    }
    if (m+n< m || m+n<n)
    {
      cout << "Overflow occured" << endl << m+n << endl;
    }
    return 0;
}
```

In case of positive or negative overflow in c/c++

The count starts from the opposite limit.

In case of Python or C# an exception is thrown in case of overflow

In javascript positive overflow ~~stores~~ returns max integer value and minimum integer value in case of negative overflow.

In java, there may be an undefined value or junk value in case of overflow.

Q. No. 7 Integer overflow and underflow vulnerabilities.

Sol: Integer overflow and underflow vulnerabilities are caused due to unsafe conversion between different variables types or logical miscalculations. The reason is that when converting from unsigned to sign in some cases gives wrong result which no one wants.

Exploitation

Integer overflow and underflow vulnerabilities are useful for hackers in many ways. These vulnerabilities can invalidate differt verification checks which were already use to protect a greater system from other vulnerabilities.

For example in an ATM transaction, there are certain criteria for performing different types of transactions, like making sure that the amount of withdraw should be less then the amount of balance in the account such that, their difference is greater than 0. In case of using unsigned integer variables the above condition will always return true and would cause unauthorized withdrawal transactions. This may lead to further problems.

Protection

In most cases, integer overflow and underflow vulnerabilities are caused by non secure or misused type conversion between signed and unsigned variables ~~of~~ or variables of smaller and larger size. So protection against these vulnerabilities can be avoided by the use of languages which allow dynamic variable typing like Python and Javascript. It is also the duty of the developer to make sure the variable types are specified explicitly.

Q. No. 8. What is IEEE-754 standard for floating point representation.

Ans. IEEE-754 is a technical standard for floating point representation established in 1985.

It contains 3 basic components

- The sign of mantissa
- The biased exponent
- The normalized mantissa

a) 32 bits representation

Sign	Exponent	Mantissa
1 bit	8 bits	23 bits

Precision: Approx 7 decimal digits

b) 64 bits representation

Sign	Exponent	Mantissa
1 bit	11 bits	52 bits

Precision: Approx 15 decimal digits

c) 128 bits representation

Sign	Exponent	Mantissa
1 bit	15 bits	112 bits

Precision: Approx 33 decimal digits

d) 256 bits representation

Sign	Exponent	Mantissa
1 bit	19 bits	236 bits

Precision: Approx 71 decimal digits

Q. No. 9 Why are biased exponents used.

Sol Biased exponents are used because it allows to have different numbers of +ve and -ve exponents

For 128 bit representation (Quadruple Precision)

There are 15 bits where 1 bit is sign bit

Inorder to calculate the biased exponent for 128 bit representation. we use the formula

$$2^{(n-1)} - 1$$

where n is the number of bits for exponent

$n = 15$

$2^{15-1} - 1$

$= 2^{14} - 1$

$= 16384 - 1$

$= 16383$

Q. No. 10 Placement of exponent

Sol : a) In IEEE-754 representation, the bits of the biased exponents are placed before the mantissa, making it easier to compare the two floating point numbers.

b) Biased vs 2's complement

In the IEEE-754 representation, the biased exponents are stored in lexical order. This would not be possible by the use of 2's complement representation. And sorting can also be carried out by IU in biased exponents as they are the most significant bit.

Q. No. 11 Convert into IEEE-754 hexadecimal format

a) $15.07539_{(10)}$

+ve sign = 0

Convert into pure binary

$15_{(10)} = 1111$

$0.07539 = .000100110100110011 00$

So $15.07539 = 1111.000100110100110011 00_{(2)}$

After normalizing

$= 1.1110001001101001100 1100 \times 2^{3}$

True exponent = 3

Biased exponent = 127 + 3 = 130

$= 10000010_{(2)}$

So final result becomes

Sign	Exponent	Mantissa
0	10000010	$11100010011010011001100_{(2)}$

0100 0001 0111 0001 0011 0100 1100 1100

4 1 7 1 3 4 C C

Convert int hexadecimal number system we get

$417134CC_{(16)}$

$0.07539 \times 2 = 0.15078$
$0.15078 \times 2 = 0.30156$
$0.30156 \times 2 = 0.60312$
$0.60312 \times 2 = 1.20624$
$0.20624 \times 2 = 0.41248$
$0.41248 \times 2 = 0.82496$
$0.82496 \times 2 = 1.64992$
$0.64992 \times 2 = 1.29984$
$0.29984 \times 2 = 0.59968$
$0.59968 \times 2 = 1.19936$
$0.19936 \times 2 = 0.39872$
$0.39872 \times 2 = 0.79744$
$0.79744 \times 2 = 1.59488$
$0.59488 \times 2 = 1.18976$
$0.18976 \times 2 = 0.37952$
$0.37952 \times 2 = 0.75904$
$0.75904 \times 2 = 1.51808$
$0.51808 \times 2 = 1.03616$
$0.03616 \times 2 = 0.07232$
$0.07232 \times 2 = 0.14464$

b) -68.2508

-ve sign = 1

Convert to pure binary

$68 = 1000100_{(2)}$

$0.2508 = .0100000000110100 1_{(2)}$

So

$68.2508 = 1000100.01000000001101001_{(2)}$

After normalizing

$= 1.0001000100000000110100 1 \times 2^{6}$

Biased exponent = 127 + 6 = 133

$= 10000101$

So our final result is

Sign	Exponent	Mantissa
1	10000101	00010001000000000110100 1

C 2 8 8 8 0 6 9

convert into hexadecimal we get

$C2888069_{(16)}$

$0.2508 \times 2 = 0.5016$
$0.5016 \times 2 = 1.0032$
$0.0032 \times 2 = 0.0064$
$0.0064 \times 2 = 0.0128$
$0.0128 \times 2 = 0.0256$
$0.0256 \times 2 = 0.0512$
$0.0512 \times 2 = 0.1024$
$0.1024 \times 2 = 0.2048$
$0.2048 \times 2 = 0.4096$
$0.4096 \times 2 = 0.8192$
$0.8192 \times 2 = 1.6384$
$0.6384 \times 2 = 1.2768$
$0.2768 \times 2 = 0.5536$
$0.5536 \times 2 = 1.1072$
$0.1072 \times 2 = 0.2144$
$0.2144 \times 2 = 0.4288$
$0.4288 \times 2 = 0.8576$
$0.8576 \times 2 = 1.7152$

Q. No. 12 Find decimal equivalent of IEEE-754 representation

$41A41B4C_{(16)}$

convert into binary

$0100\ 0001\ 1010\ 0100\ 0001\ 1011\ 0100\ 1100_{(2)}$

According to IEEE-754 representation

$\underbrace{0}_{\text{sign}}\ \underbrace{1000\ 0011}_{\text{Exponent}}\ \underbrace{01001000001101101001100}_{\text{Mantissa}}$

Biased exponent $= 10000011_{(2)}$
$= 131$

True exponent $= 131 - 127 = 4$

We get normalized binary

$1.01001000001101101001100 \times 2^4$

$10100.1000001101101001100_{(2)}$

$10100_{(2)} = 1 \times 2^4 + 1 \times 2^2 = 20_{(10)}$

Sign bit = 0 = +ve

The numbers after decimal point equate to = ~~1048575~~ 531276

The ~~total~~ maximum number of the binary combination = 1048575

The decimal conversion of the floating point number is 0.5066647593

Thus

$41A41B4C_{(16)} = 20.5066647593$

Q. No. 13

```
# include<iostream>
# include<cmath>
using namespace std;
int main()
 {
    cout << "Range of float: " << -(2-pow(2,-23))*pow(2,pow(2,8-1)-1);
    cout << " to " << (2-pow(2,-23))*pow(2,pow(2,8-1)-1);
    cout << endl << endl;
    cout << "Range of double: " << -(2-pow(2,-52))*pow(2,pow(2,11-1)-1);
    cout << " to " << (2-pow(2,-52))*pow(2,pow(2,11-1)-1);
    cout << endl << endl;
    cout << "Range of long double: ";
    cout << -(2-pow(2,-112))*pow(2,pow(2,15-1)-1) << " to ";
    cout << (2-pow(2,-112))*pow(2,pow(2,15-1)-1) << endl;
    return 0;
 }
```

In C/C++ :

- float data type offers single precision
- double data type offers double precision
- long double data type offers quadruple precision

Q. No. 14

```
# include <iostream>
# include <cmath>
using namespace std;
int main ()
{
    cout << "In case of float overflow ";
    cout <<( (float)(2-pow(2,-52))* pow(2,pow(2,10)-1);
    cout << endl << endl;
    cout << " In case of float underflow ";
    cout << (float) pow(2,-150);
    cout << endl;
    return0;
}
```

Float Imprecision

Float imprecision is the problem which occurs when trying to store an ~~fraction~~ irrational number in binary format with a finite amount of bits.

Q. No. 15 Representation of ±0 and ± infinity in IEEE-754

Sol. For +0

Sign	Exponent	Mantissa
0	00000000	00000000000000000000000

For -0

Sign	Exponent	Mantissa
1	00000000	00000000000000000000000

Note : Even though +0 and -0 have distinct representations, they ~~are~~ both compare as equal

For + infinity

Sign	Exponent	Mantissa
0	11111111	00000000000000000000000

For - infinity

Sign	Exponent	Mantissa
1	11111111	00000000000000000000000

Q. No. 16 Add and Subtract

01000010 0 00000000100110001011001

$+1.00000000100110001011001 \times 2^5$

$= 100000.000100110001011001_{(2)} \Rightarrow 32.0745582581_{(10)}$

And 01000100 0 01100000 000010111010110

$+1.01100000000010111010110 \times 2^9$

$= 1011000000.00010111010110_{(2)} \Rightarrow 704.09118652 3_{(10)}$

a) Adding both numbers we get

$736.1657447811_{(10)} \Rightarrow \underbrace{0100}_{4}\,\underbrace{0100}_{4}\,\underbrace{0011}_{3}\,\underbrace{1000}_{8}\,\underbrace{0000}_{0}\,\underbrace{1010}_{A}\,\underbrace{1001}_{9}\,\underbrace{1100}_{C}{}_{(2)}$

$= 44380A9C_{(16)}$

b) Subtracting both numbers we get

$-672.0166282649 \Rightarrow \underbrace{1100}_{C}\,\underbrace{0100}_{4}\,\underbrace{0010}_{2}\,\underbrace{1000}_{8}\,\underbrace{0000}_{0}\,\underbrace{0001}_{1}\,\underbrace{0001}_{1}\,\underbrace{0000}_{0}{}_{(2)}$

$= C4280110$