Computer Organization
AND
Assembly Language (CMP-223)

Assignment-01

Name: Ali Raza

Roll No: BCS-F19-M-513

Problem 1:

$$(41A7)_{16}$$

Binary:

$$(41A7)_{16} = (4\times16^3) + (1\times16^2) + (10\times16^1) + (7\times16^0)$$

$$= 16384 + 256 + 160 + 7$$

$$= (16807)_{10}$$

Now

$$(0100000110100111)_2$$

2	16807
2	8403 - 1
2	4201 - 1
2	2100 - 1
2	1050 - 0
2	525 - 0
2	262 - 1
2	131 - 0
2	65 - 1
2	32 - 1
2	16 - 0
2	8 - 0
2	4 - 0
2	2 - 0
	1 - 0

Decimal:

$$= (16807)_{10}$$

Octal:

$$(\overline{100}\,\overline{000}\,\overline{110}\,\overline{100}\,\overline{111})_2$$

$$(4\ 0\ 6\ 4\ 7)_8$$

Note: As the number is +ve (it using signed value), so it'll be same for 2s, 1s and Signed magnitude methode.

Problem:- 02:

In signed magnitude methode, MSb is reserved for sign and remaing bits are used for magnitude. It works well for representing +ve and -ve integers (although the two 0s are bother some). But it doesn't work well in computation. It produces undesired results while subtraction (addition of +ve and -ve numbers).

For example:

(13) ⟶ 1101

+ (-6) 1110

7 11011 ⟶ which isn't correct answers.

So we need a seperate hardware for subtration is signed magnitude.

1s and 2s complement methods don't require any additional hardware for subtration. But 1s complement methode

has two different representations for zero. (+ve 0: 0000) and (-ve 0: 1111). So this produce some undesire results.

Mon Tue Wed Thu Fri Sat

Problem 03:

Date: __/__/20

```
#include <stdio.h>
#include <limits.h> // A Library that have
int main(){     //macros for data types limits.
 printf("Signed char minimum = %d\n", SCHAR_MIN);
 Printf("Signed char maximum = %d\n", SCHAR_MAX);
 printf("Unsigned char maximum = %d\n", UCHAR_MAX);
 printf("Signed Short minimum = %d\n", SHRT_MIN);
 printf("Signed short maximum = %d\n", SHRT_MAX);
 printf("Unsigned short maximum = %d\n", USHRT_MAX);
 Printf("Signed int minimum = %d\n", INT_MIN);
 printf("Signed int maximum = %d\n", INT_MAX);
 printf("Unsigned int maximum = %d\n", UINT_MAX);
 printf("Long signed minimum = %ld\n", LONG_MIN);
 printf("Signed Long maximum = %ld\n", LONG_MAX);
 Printf("Unsigned Long maximum = %lld\n", ULONG_MAX);
 printf("Signed Long Long min = %lld\n", LLONG_MIN);
 Printf("Signed Long Long max = %lld\n" LLONG_MAX);
 printf("Unsigned Long Long max = %llu\n" ULLONG_MAX);
 return 0;
}
```

Yes, the value that can a variable hold depends on hardware. It depends on word size or register size of the machine.

For example in c/c++ size of long-int is 4-byte in 32-bit machine. While it is of 8-byte in 64-bit machine. The compile is restricted with register size of installed OS (32-bit or 64bit).

Problem 04:

a) 2's complement: (signed)

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

-32768 to 32767

Unsigned:-

0 to 2^{16}

0 to 65536

b) 1's complement: (signed.)

c) also to signed magnitude:-

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

-32767 to 32767

Problem 05:-

a) 0x86 + 0x84

$(86)_{16} \rightarrow (1000\ 0110)_2$

$(84)_{16} \rightarrow (1000\ 0100)_2$

1 0000 1010

As carry in and carry out from MSb is different so this is overflow.

Carry Flag = 1 negative

Overflow flag = 1

Mon Tue Wed Thu Fri Sat

Date: __/__/20

b) 0x7E + 0x70

7E ⟶ 0111 1110

+ 70 0111 0000

1 1110 1110

As carry in and carry out are different, There is overflow.

Carry flag = 0

Overflow flag = 1

c) 0x F6 + 0x 7E

F6 ⟶ 1111 0110

+ 7E 0111 1110

1 0111 0100

As carry in and carry out are same so Their is no overflow.

Carry flag = 1

Overflow flag = 0

Mon Tue Wed Thu Fri Sat

Date: __/__/20__

Problem 06:

```
#include <stdio.h>
#include < limits.h>
int main(){
    int n1;
    unsigned int n2;
    n1 = INT_MAX; // stored maximum value
    n1 += 1       // Added 1 to test overflow
    printf(" n1 = %d\n", n1);
    n2 = UINT_MAX // stored max unsigned
    n2 +=1        // To test overflow
    printf ("n2 = %u \n", n2);
    return 0;
}
```

output

```
n1 = -2147483648
n2 = 0
```

In C/C++, when overflow occure, it will wraped arround the minimum value. For signed numbers, it will stard ~~form~~ -ve value. For example, int has 2147483647 max value. When we add 1 to it, it will become -2147483648

which is the minimum value of signed int.

And it will become 0, if there is unsigned int.

However, C/C++ [redacted] does not produce any error or warning for overflow.

In some programing languages like Python and Rubby, they detect the overflow and allocate larger size memory to prevent data from losing. SWIFT language produces error if we didn't tell comput what to do in case of overflow.

Mon Tue Wed Thu Fri Sat

Date: __/__/20

Problem 07:-

Integer overflow vulnerabilities are occured when a value is moved into a variable type too small to hold it.

One example is "Down Casting" from a long (which uses 64 bits) to an int (which uses 16 or 32 bits). This is accomplished by cutting the value down to a small eough size that it fits in the small value. If any of the bits that are droped are non-zero, then the value suddenly becomes a lot smaller.

Bascically, underflow term is closely related to floating point numbers. But, specifically for integes, integer underflow happen when a value beaomes too small to fit. When this occurs, the value wraps around from

the minimum value that can be stored to the maximum.

Integer overflow and underflow vulnerabilities are useful to hack in a number of ways. The main reason for this is that these vulnerabilities can invalidate checks made to protect against other classes of vulnerablities.

For example, a buffer overflow vulnerability is created when a developer fails to check the length of user-controlled input befor placing it in a preallocated memory buffer.

Another application of integer overflow and underflow is defeating checks to determine if a value meets certain criteria, like ensuring that an account has a certain minimum balance before initiating a withdrawal.

Date: __/__/20__

Integer overflow and underflow vulnerablities are caused by misuse of variable type conversions. Protecting against these vulnerablities is fairly simple:

↳ use a language with dynamic variable typing (like python).

↳ Make sure that variable types are explicitly specified and the correct type for the job.

Mon Tue Wed Thu Fri Sat

Date: __/__/20__

Problem 08: Unicode

Unicode Standard is a character coding system designed to support the worldwide interchange processing and display of the written texts of the diverse languages and technical disciplines of the modern world. In addition, it supports classical and historical texts of many written languages. It afford simplicity and consistancy of ASCII, even correspond characters are same.

Some common encoding formates used by Unicode are **UTF-8, UTF-16, UTF-32** and several other encodings are in use. These are common implementation used now-a-days.

Windows uses UTF-16LE for all things Unicode (except for MultiByteToWideChar() and

WideCharToMultiByte(), which support UTF-7, UTF-8, and UTF-16, amongst other charaseter installed in the OS).

In Linux, the most general list of world characters is usually encoded in UTF-8.

In Mac OS, Simredo 3.31 uses default formate which is UTF-16. it can convert over 100 encodings to and from unicode (UTF-8 and big and little endian UTF-16).

Problem # 09:

The concept of floating point binary numbers were introduced in the mid 1950s. There was no uniformity in the formates used to represent floating point numbers and programs were not portable from one manufacture's computer to another.

So a standards committee was formed by the institute of Electrical and Eleetronics Engineers (IEEE) to standardize how floating point binary numbers would be represented in computers. In addition, the standard specified uniformity in rounding numbers, treating exception conditions such as

Tue Wed Thu Fri Sat

Date: 1/20

attempting to divide by 0, and representation of 0 and infinity (∞).

Part-ii	expo-Bais	E-bits	D-bits	E-min	E-Max
a) 32-bit	$2^7-1=127$	8	23	-126	+127
b) 64-bit	$2^{10}-1=1023$	11	52	-1022	+1023
c) 128-bit	$2^{14}-1=16383$	15	112	-16382	+16383
d) 256-bit	$2^{18}-1=262143$	19	236	-262142	+262143

Problem 10:

For flexibility to choose whether they required more +ve ore more -ve exponent numbers, IEE designers used baised notation.

Moreover in basic notation, the exponents are in lexical order and allow easy comparison of Floating Point numbers.

If 14 bits are used to store the exponent, the baising will be:

$$2^{14-1} - 1 = 2^{13} - 1$$

$$= 8192 - 1 = 8191$$

And Range of exponent will be:

Exp: $16382 - 8192 = 8191$

Range: $+/- *(2 - 2^{-113}) \times 2^{+8191}$

$= +5.25176 1E+2431$

Date: __/__/20

Problem 11:

Exponent in IEE-754 is placed before mantissa so that comparison of floating point number is easily performed. The baise exponant are in lexical order. In this way, for baised exponent it's simple to compare values. While in 2's complement, it's hard to compare the numbers because of negative and positive values.

Tue Wed Thu Fri Sat

Date: __/__/20__

Problem 12:

a) $(75.07539)_0$

Binary = $(1001011.00010011010011001111)_2$

Normalizing:

$(1.00101100010011010011001111 \times 2^6)$

Adding Baising to exponent:

$6 + 127 = 133 = 10000101$

Sign Bit = 0

Now

adding last bit.

0 10000101 00101100010011010011001111

$(01000010100101100010011010011010)_2$

↓ IEE-754 Binary

↳ $(4296269A)_{16}$

b) $(-128.25508)_{10}$

Binary = $-(10000000.01000001010011001111)_2$

Normalizing = $(1.00000000100000010100110011111 \times 2^7)$

Adding Baising to exponent:-

$7 + 127 = 134 = 10000110$

sign-Bit = 1

1 10000110 00000000100000101001101111

1 10000110 00000000100000101001101

Mon Tue Wed Thu Fri Sat

Date: / /20

Problem 13:

$(41A42B43)_{16}$

Binary:

$(0100\ 0001\ 1010\ 0100\ 0010\ 1011\ 0100\ 0011)_2$

Now

(0 10000011 01001000010101101000011)

Exponent = $10000011 = (131)_{10}$

Removing baising = $131 - 127 = 4$

Sign = 0

So

$1.01001000010101101000011 \times 2^4$

$10100.10000101011010000011$

$20.(1X2^{-1} + 1X2^{-6} + 1X2^{-8} + 1X2^{-10} + 1X10^{-11} + 1X10^{-13} + 1X10^{-78} + 1X10^{-9})$

$= 20.052111238861$

Problem 14:-

```
#include<stdio.h>
#include <float.h>
int main(){
 printf("Precision of float = +-(%e)\n", FLT_MIN);
 printf("Range of float = +-(%e)\n", FLT_MAX);
 Printf("precision of double = +-(%e)\n", DBL_MIN);
 printf("Range of double = +-(%e)\n", DBL_MAX);
 printf("Precision of long double = +-(%e)\n", LDBL_MIN);
printf("Range of long double = +-(%e)\n", LDBL_MAX);
return 0;
}
```

In C/C++:

float = single precision

double = double precision

long double = quarduple precision.

Problem 15:

```
#include <stdio.h>
#include <float.h>
int main(){
    float n = FLT_MAX;
    n *= 1.5; // To check overflow
printf("n = %e\n", n);
    n = FLT_MIN;
    n /= 1e20; // To check underflow
printf("n = %e\n", n);
```

Floating point imprecision:-

Floating point impreecision stems from the problem of trying to store number like 1/10 or (0.10) in a computer with binary number system with a finite amount of ~~zero~~ numbers. In other words we can say recurring numbers.

Mon Tue Wed Thu Fri Sat

Date: __/__/20__

Problem 16:

	sign	Expo	Mantissa
+/- 0 =	0/1	00000000	00000000000000000000000

	sign	Expo	Mantissa
+/- (∞) =	0/1	11111111	00000000000000000000000

Problem 17.

i) Addition:

a = 01000010 00000000 01001100 01011001

b = 01000100 00010000 00000101 11010110

Now a → sign = 0

expo = 10000100 = 132

after baising removed = 132 − 127 = 5

Mentissa = $1.00000000100110001011001 \times 2^5$

Similarly b → sign = 0

expo = 10001000 = 136

after baising removed = 136 − 127 = 9

so $1.01100000000101111010110 \times 2^9$

For addition → normalizing a:

$0.0001\ 00000000\ 10011000\ 10110 \times 2^9$

$+\ 1.01100000000101111010110 \times 2^9$

$1.0111\ 00000000\ 10100101\ 01011 \times 2^9$

Now moving back to IEEE-754.

$q+127 = 136 = 10001000$

so $\boxed{0\ 10001000\ 01110000000010101001011}$

or $0x\ 44380A9B$

Subtraction:

As b is ~~greater~~ greater that a:

so $b-a$:

$1.011000000000010111011 \times 2^9$

$0.000100000000100111000101 \times 2^9$

$1.010100000000010000100001 \times 2^9$

again moving back to IEEE-754

$q+127 = 136 = 10001000$

so

$\boxed{0\ 10001000\ 01010000000010000100001}$

or $0x\ C4280111$