CMP 223

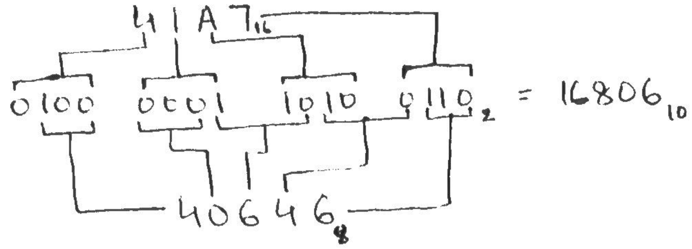
BCSF19A515 _ Assignment 01

Nofil Qasim

COAL ASSIGNMENT
NOFIL QASIM BCSF19A515

(1)

Problem 1 =



Since 1st bit is 0 answers will be same for sign-magnitude, 1s complement and 2s complement.

Problem 2 = In the case of 1s and 2s complement we can perform some prerequisite operations on the numbers and use the adder to both add and subtract numbers e.g. in 2s complement we can take NOT of negative number, increment it and then use the adder. The result will be the difference of the numbers.
This cannot be done in signed-magnitude notation so we need to design a seperate subtractor circuit for it just to be able to subtract.

Problem 3 =

C file also on bitbucket

```
#include <stdio.h>
#include <limits.h>
#define UCHAR_MIN 0  // Same for all unsigned
int main (void) {
      printf("Max char: %i\n", CHAR_MAX);
      printf(" Min char: %i\n", CHAR_MIN);
      printf(" Max unsigned char: %i\n", UCHAR_MAX);
      printf(" Min unsigned char: %i\n", UCHAR_MIN);
      printf(" Max short: %ui\n", SHRT_MAX);
      printf(" Min short: %ui\n", SHRT_MIN);
      printf(" Max unsigned short: %i\n", USHRT_MAX);
```

```
printf("Min unsigned short: %i\n", USHRT_MIN);
printf("Max int : %i\n", INT_MAX);
printf("Min int : %i\n, INT_MIN);
printf("Max unsigned int: %li\n", UINT_MAX);
printf("Min unsigned int: %i\n, UINT_MIN);
printf("Max long: %li\n", LONG_MAX);
printf("Min long: %li\n", LONG_MIN);
printf("Max unsigned long: %lu\n", ULONG_MAX);
printf("Min unsigned long : %i\n", ULONG_MIN);
printf("Max long long: %lli\n", LLONG_MAX);
printf("Min long long: %lli\n", LLONG_MIN);
printf("Max unsigned long long: %llu\n", ULLONG_MAX);
printf("Min unsigned long long: %i\n", ULLONG_MIN);
return 0;
}
```

Problem 4.

a) -65536 — 65535

b) -65535 — 65535

c) 0 – 131071

Problem 5:

a)

0x 86 =	1 0 0 0 0 1 1 0	CF = 1
0 x84 =	1 0 0 0 0 1 0 0	Cin = 0, Cout = 1
0 x0A =	0 0 0 0 1 0 1 0	XOR = 1

Negative Overflow

OF = 1

b) 0x7E = 0 1 1 1 1 1 1 0
0x70 = 0 1 1 1 0 0 0 0
0xEE = 1 1 1 0 1 1 1 0

Positive Overflow

CF=0
Cin=1 Cout=0
XOR=1
OF=1

c) 0xF6 = 1 1 1 1 0 1 1 0
0x7E = 0 1 1 1 1 1 1 0
0x74 = 0 1 1 1 0 1 0 0

No Overflow

CF=1
Cin=1 Cout=1
XOR=0
OF=0

Problem 6=

```
#include <stdio.h>
#include <limits.h>
#define UINT_MIN 0
int main(void){
    printf("Integer Positive Overflow: %i\n", INT_MAX+1);
    printf("Integer Negative Overflow: %i\n", INT_MIN-1);
    printf("Unsigned Integer Positive Overflow: %lu\n", UINT_MAX+1);
    printf("Unsigned Integer Negative Overflow: %lu\n", UINT_MIN-1);
    return 0;
}
```

C and C++, Java and JavaScript do not handle overflows by default. If an overflowed number is statically assigned the compiler will produce a warning. If the number is dynamically assigned there will be no warning rather the result produced will be unexpected.

In C# certain compilers will check for overflows and throw an exception on encountering an overflow. Most common compilers will still have the same behaviour as the other languages mentioned above.

In Python once an integer overflows its automatically promoted to an arbitrary length long which can be resized to take up all available memory space thus resulting in a type which is hard to overflow.

Problem 7 =

When a number outside the range of the integer type is attempted to be stored in a variable of integer data type an integer overflow occurs. Integer underflow is a rarely used term to describe an overflow which occurs from the negative end of the integer and makes it positive.

According to CWE Mitre integer overflows can be exploited to violate all three constituents of the CIA triad.

Integrity = Integer overflows can be used to bypass security methods used to protect against buffer overflows. A very large number can be passed to the program. If the number is being used in any way to allocate an array dynamically or any memory dynamically and it wraps around it may result in the allocated memory being smaller than the required amount to store data. This will result in a potential heap based buffer overflow allowing us to potentially access dynamic memory allocations we might not normally have access to and potentially edit the data there. This will result in a compromise of data integrity.

Even if we are not talking about dynamic memory, integer overflows may also be exploited to corrupt important values in the program resulting in unexpected behaviours.

Confidentiality = Integer overflows can be exploited to bypass checks against overflows in statically allocated memory forexample entering data into an array of predetermined size. This can result in a stack based buffer overflow. These are potentially even more dangerous than heap based overflows ~~since~~ if the stack is executable since the return

pointer of the function can be overwritten to point to memory of our own choosing which can let us execute code written by us. This is especially dangerous in programs where the setuid bit is set as it can allow us to get a root shell by pointing program flow towards shell code.
Even in programs where the setuid bit is not set, integer overflows can be exploited to result in buffer overflows which allow us to execute arbitrary code such as the ones used in Remote Code Execution exploits.

Availability = Integer overflows can be exploited to generate undefined behaviour in programs. Wraparounds can cause loops using counters based on user input to run infinitely.
Overwriting functions such as free() in C can also cause increased memory usage than normal. Overwriting important variables or instructions might also result in some part of the code not working which compromises availability.
According to CWE Mitre some potential protection mechanisms against this vulnerability can be:
Using languages such as Python or Ada which either make em arbitrarily long integers or throw exceptions when an overflow is encountered.
Performing validation checks to ensure data is entered within the bounds of the data type.
Other mitigations can be:
Using saturated arithmetic in places where it can be.
Assigning NaN status to values which exceed expected bounds (Explicit Propogation – Wikipedia)
Automated Static Analyses can be used to report on such vulnerabilit if they are found in code so they can be fixed before anyone has a chance to exploit them.

Problem 8 =

The Unicode Standard is a universal character encoding maintained by the Unicode Consortium. The encoding standard provides the basis for processing, storing and the interchange of text data in any language in all modern software and IT Protocols. (Unicode.Org)

The Unicode Standard covers all the characters used in all currently discovered writing scripts in the world as well as emojis.

The standard was created because previously encoding standards were found lacking in the scope of symbols they could represent and there was no single standardized system for encoding.

Unicode uses encodings classified under the name Unicode Transformation Format (UTF). There are several of these.

UTF7 = Uses 7 bits for each character. Designed to represent ASCII characters in emails encoded with Unicode.
Commonly used in MIME and IMAP.

UTF8 = Uses variable number of ~~bits~~ bytes to represent characters of different scripts.
Also has support for emojis and special characters.
The most common encoding used across most of the world wide web.

UTF16 = Supports up to 1.1 million characters.
was the default format for Windows for some time.
Supported and used by multiple languages such as Java, Python and JavaScript.

As of 2019 and 2020 Windows, Linux, MacOS and other Unix based distributions use and reccommend UTF8. (WikiPedia)

Problem 9 =

The IEEE 754 Standard for floating point arithmetic specifies interchange and arithmetic formats and methods for binary and decimal floating point arithmetic in computers (IEEE).
It was needed so different manufacturers had a single standard to work with instead of using their own.

a) 1 signed bit
8bit ~~mantissa~~ exponent
23 bit mantissa

Precision = $\pm 1.175 \times 10^{-38}$

b) 1 signed bit
11 bit exponent
52 bit mantissa

Precision = $\pm 2.23 \times 10^{-308}$

c) 1 signed bit
15 bit exponent
112 bit mantissa

Precision = $\pm 3.36 \times 10^{-4932}$

d) 1 signed bit
19 bit exponent
23 bit mantissa

Precision = $\pm 2.48 \times 10^{-78913}$

Problem 10 =

Biased exponent representation is used as it arranges the bits of the exponent in the same order of magnitude as the numbers they are suppoed to represent. This allows for simpler logic design when there is a need to compare two floating point numbers.

14 bit exponent   113 bit mantissa

$2^{14-1} - 1 = 8191$

$(2^{14} - 2) - 8191 = 8192$

Range of Exponent = -8191 to 8192

$(2 - 2^{-113}) \times 2^{8192} = 2.2 \times 10^{2466}$

Range = $\pm 2.2 \times 10^{2466}$

Problem 11 =

Exponent is placed first as it is the first determiner of the size of a number. Placing it first allows the computer to compare two floating point numbers more easily. Biased exponent is used as it arranges the bits of the exponent in the same order of magnitude as the number it is supposed to represent which makes comparison easier.

Problem 12 =

a) 75.07539

$\underbrace{0100\ 1011}_{+6}$ 00010011010011001

$127 + 6 = 133$

0 10000101 00101100010011010011001

Sign   Exponent   Mantisse

0x4296269A

0.07539 x 2
0.15078 x 2
0.30156 x 2
0.60312 x 2
1.20624 x 2
0.41248 x 2
0.82496 x 2
1.64992 x 2
1.29984 x 2
0.54968 x 2
1.19936 x 2
0.39872 x 2
0.79744 x 2
1.59488 x 2
1.18976 x 2
0.37952 x 2
0.75904 x 2
1.51808 x 2

b) − 128 · 25508

0·25508 × 2
0·51016 × 2
1·02032 × 2
0·04064 × 2
0·081128 × 2
0·16256 × 2
0·32512 × 2
0·65024 × 2
1·30048 × 2
0·60096 × 2
1·20192 × 2
0·40384 × 2
0·80768 × 2
1·61536 × 2
1·23072 × 2
0·46144 × 2
0·92288 × 2
1·84576 × 2
1·69152 × 2

1,0000000. 0100000010100100011

127 + 7 = 134

1 10000110 00000000100000010100110 1

Sign | Exponent | Mantissa

0xC300414D

Problem 13 =

41A42B43

0 1000001 1 010 0100 0010 1011 0100 0011

Sign | Exponent | Mantissa

131 − 127 = 4

1·010 0100 0010 1001 0100 0011

~~20.5211~~ 20·5211238861

Problem 14 =

```
#include <stdio.h>
# include < float.h>

int main (void) {
    printf (" Max Float: %f\n", FLT_MAX);
    Printf (" Min Float: %f\n", -FLT_MAX);
    printf(" Max Double: %lf\n", DBL_MAX);
    printf(" Min Double: %Lf\n", -DBL_MAX);
    printf(" Max Long Double: %Lf\n", LDBL_MAX);
    printf(" Max Long Double: %Lf\n", -LDBL_MAX);

    return 0;

}
```

In C float supports single precision.
Double supports double precision.
Long Double supports quadruple precision.

Problem 15=

```c
#include <stdio.h>
#include <float.h>
int main(void){
    printf(" Float Overflow: %f\n", FLT_MAX+FLT_MAX); //inf
    printf(" Float Underflow: %f\n", 0.0000125); //0.000013
    return 0;
```

Floating point imprecision is when a number is too small on the decimal side to be properly represented e.g 0.0000125 will be rounded to 0.000013 which is not the actual number.

Problem 16=

$$+0 = \underbrace{0}_{\text{Sign}}\ \underbrace{00000000}_{\text{Exponent}}\ \underbrace{00000000000000000000000}_{\text{Mantissa}}$$

$$-0 = \underbrace{1}_{\text{Sign}}\ \underbrace{00000000}_{\text{Exponent}}\ \underbrace{00000000000000000000000}_{\text{Mantissa}}$$

$$+inf = \underbrace{0}_{\text{Sign}}\ \underbrace{11111111}_{\text{Exponent}}\ \underbrace{00000000000000000000000}_{\text{Mantissa}}$$

$$-inf = \underbrace{1}_{\text{Sign}}\ \underbrace{11111111}_{\text{Exponent}}\ \underbrace{00000000000000000000000}_{\text{Mantissa}}$$

Problem 17 =

0 10000100 00000000100110001011001 = A
0 10001000 01100000000000101111010110 = B

A = 1.0000000 00 10011 0001011001 $\times 2^5$
B = 1.01100 00 00 000 0101111010110 $\times 2^9$

A = 0.00010000 000010011000 101 101 $\times 2^9$
B = 1.01100 000000001011110101 10 000 $\times 2^9$

1.011100000000 10101001101 1101 $\times 2^9$

$$0\ \underbrace{10001000}_{\text{Exponent}}\ \underbrace{01110000000001010100111100}_{\text{Mantissa}}$$

~~[illegible]~~
0x44380A9C

Bibliography:

techterms.com (For UTF)
cwe.mitre.org (Integer Overflow)
resources.infosecinstitute.com (Integer Overflow)
stackoverflow.com (Handling of Overflows)
standards.ieee.org