University
of Windsor

School of Computer Science
Faculty of Science

COMP-2650: Computer Architecture I: Digital Design
Winter 2021

Lab#	Date	Title	Due Date	Grade Release Date
Lab05	Week 05	**Number Systems (Signed-Magnitude)**	Feb. 09, 2021 Tuesday Midnight AoE Wednesday 7 AM EDT	Feb. 15, 2021

This lab's objectives will be to master the topics in number systems, especially arithmetic, by implementing the algorithms with a programming language, herein, C/C++.

Step 1. Environment Setup

Our programming environment is the same as the first lab (Lab 01). In this lab, we want to extend our Lab 03 to support arithmetic on binary numbers. For instance, we want to calculate the addition or subtraction of two binary numbers in signed-magnitude.

As we discussed in the lectures, there are different ways to represent negative and positive numbers. In signed-magnitude, we consider the highest significant position as the sign. Hence, it loses its value as a power of r in base-r. However, in signed-radix-complement, we use positive numbers to show the negative numbers. So, there is no position for the sign.

In C/C++, a variable can be defined as **unsigned** to indicate that the variable does not support to show negative numbers. This is similar to when we have number system with no negative numbers:

```c
01 #include <stdio.h>
02 int main(void) {
03
04     setbuf(stdout, NULL);
05     unsigned int a;
06
07     printf("Enter an unsigned integer number:\n");
08     scanf("%u", &a);
09
10     printf("The number is: \n");
11     //printf("Binary: %b \n", a); There is no option for binary!
12     printf("Octal: %o \n", a);
13     printf("Decimal: %u \n", a);
14     printf("Hexadecimal: %x \n", a); //Alphabet in small letters
15     printf("HEXAdecimal: %X \n", a); //Alphabet in capital letters
16     return 0;
17 }
```

As shown in lines# 05, we defined an **unsigned** integer variable filled with an **unsigned** decimal value by the user in line# 08. Please pay attention to the format specifier for **unsigned** variables in **scanf** and **printf,** which is **"%u".** An example run would be:

```
Enter an integer number:
15
```

```
The number is:
Octal: 17
Decimal: 15
Hexadecimal: f
```

You may know that C/C++ are so-called ==weakly-typed== languages. This is because C/C++ does not raise exceptions or error, neither at *compile-time* nor *run-time*, in cases that a value of a type is assigned to a variable of a different type due to ==implicit casting==. For example, another run of our previous code could be:

```
Enter an unsigned integer number:
-45
The number is:
Octal: 37777777723
Decimal: 4294967251
Hexadecimal: fffffd3
HEXAdecimal: FFFFFFD3
```

As seen, the user enters a negative decimal number -45. Our program stores this value in the variable 'a' whose type is **unsigned**. However, no error or exception raised by C/C++. Indeed, C/C++ did an implicit cast and converted the type of variable 'a' to **signed** integer first, and then stored the negative value. You can see this in the output in octal and hexadecimal bases. In the decimal base, however, you may wonder why -45 became 4294967251 (Why?).

Also, you can define a variable that can store negative and positive numbers as **signed**. Indeed, any numeric variable in C/C++ is in signed-2's-complement by default, and ==using the keyword **signed** is optional,== as seen below! Please pay attention to the format specifier for **signed** variables in **scanf** and **printf,** which is **"%d"**.

```
01 #include <stdio.h>
02 int main(void) {
03
04     setbuf(stdout, NULL);
05     signed int a; //you can drop 'signed' keyword
06
07     printf("Enter an integer number:\n");
08     scanf("%d", &a);
09
10     printf("The number is: \n");
11     //printf("Binary: %b \n", a); There is no option for binary!
12     printf("Octal: %o \n", a);
13     printf("Decimal: %d \n", a);
14     printf("Hexadecimal: %x \n", a); //Alphabet in small letters
15     printf("HEXAdecimal: %X \n", a); //Alphabet in capital letters
16     return 0;
17 }
```

Regarding overflow, C/C++ does not raise an error or exception when an overflow happens either in **signed** or **unsigned** numeric variables. We explain this in the following program:

```
00 #include <limits.h>
01 #include <stdio.h>
02 int main(void) {
03
04     setbuf(stdout, NULL);
05     unsigned int a;
```

```
06
07     printf("Enter an unsigned integer number between %u and %u:\n", 0, UINT_MAX);
08     scanf("%u", &a);
09
10     printf("The number is: \n");
11     //printf("Binary: %b \n", a); There is no option for binary!
12     printf("Octal: %o \n", a);
13     printf("Decimal: %u \n", a);
14     printf("Hexadecimal: %x \n", a); //Alphabet in small letters
15     printf("HEXAdecimal: %X \n", a); //Alphabet in capital letters
16     return 0;
17 }
```

From the library `<limits.h>`, we can find the minimum and maximum for a given type in C/C++. Here, we used UINT_MAX for the maximum **unsigned** integer, which is equal to 4294967295 in C/C++:

```
Enter an unsigned integer number between 0 and 4294967295:
4294967295
The number is:
Octal: 37777777777
Decimal: 4294967295
Hexadecimal: ffffffff
HEXAdecimal: FFFFFFFF
```

As seen, we enter the maximum number, and the program prints out this value in octal, decimal, and hexadecimal bases. What would be this number in base-2? It is 32 bits of 1 (Why?).

Now, let's create an overflow by entering the maximum number + 1:

```
Enter an unsigned integer number between 0 and 4294967295:
4294967296
The number is:
Octal: 0
Decimal: 0
Hexadecimal: 0
HEXAdecimal: 0
```

You see that the program did not raise any error or exception about an overflow. This number would be 33 bits with the highest bit equal to 1 and all other remaining bits equal to 0. Why? Simply increment the maximum **unsigned** integer by one unit in base-2. However, the program dropped the bit 1 in the 33rd bit and only stored the 0s since there are only 32 bits for **unsigned** integer in C/C++.

You can try overflow in **signed** integer by running the following code and see that C/C++ does not raise any error or exception for an overflow.

```
00 #include <limits.h>
01 #include <stdio.h>
02 int main(void) {
03
04     setbuf(stdout, NULL);
05     int a;
06
07     printf("Enter a signed integer number between %d and %d:\n", INT_MIN, INT_MAX);
08     scanf("%d", &a);
09
10     printf("The number is: \n");
```

```
11      //printf("Binary: %b \n", a); There is no option for binary!
12      printf("Octal: %o \n", a);
13      printf("Decimal: %d \n", a);
14      printf("Hexadecimal: %x \n", a); //Alphabet in small letters
15      printf("HEXAdecimal: %X \n", a); //Alphabet in capital letters
16      return 0;
17 }
```

```
Enter a signed integer number between -2147483648 and 2147483647:
2147483648
The number is:
Octal: 20000000000
Decimal: -2147483648
Hexadecimal: 80000000
HEXAdecimal: 80000000

Enter a signed integer number between -2147483648 and 2147483647:
-2147483649
The number is:
Octal: 17777777777
Decimal: 2147483647
Hexadecimal: 7fffffff
HEXAdecimal: 7FFFFFFF
```

Can you explain what the program printed out in octal, decimal, and hexadecimal? As seen, the minimum negative number minus one became the maximum positive number given 32 bits. Why?

Regarding arithmetics, all calculations happen in signed-2's-complement in C/C++. No built-in feature has been included for other number systems such as signed-magnitude due to their inefficient representation of negative numbers and arithmetic, as we discussed in our lectures.

In this lab, we are going to add arithmetic in signed-magnitude to our program. Also, we want to let the user know whether an overflow happens.

Step2. Writing Modular Programs

As we did in Lab04, let's add a new header file arithmetic.h and code (source) file arithmetic.cpp to implement all functions related to arithmetic in signed-magnitude.

arithmetic.h
```
void func_signed_mag_addition(int a[], int b[], int result[]);
void func_signed_mag_subtraction(int a[], int b[], int result[]);
```

arithmetic.cpp
```
#define MAX 8//Byte = 8 bits
void func_signed_mag_addition(int a[], int b[], int result[]){...}
void func_signed_mag_subtraction(int a[], int b[], int result[]){...}
```

As seen, header files contain only the signatures of the functions and not the bodies. Please look at the ';' in the end of each function. Now we are ready to add the headers to our main program and use the functions in each separate file:

```
01 #include <stdio.h>
02 #include "arithmetic.h"
03
04 #define MAX 8//Byte = 8 bits
05 int main(void) {
```

```
06      setbuf(stdout, NULL);
07
08      int x[MAX];
09      int y[MAX];
10
11      printf("Enter the first binary number:\n");
12      for(int i=0; i < MAX; i = i + 1){
13          scanf("%d", &x[i]);
14      }
15      printf("Enter the second binary number:\n");
16      for(int i=0; i < MAX; i = i + 1){
17          scanf("%d", &y[i]);
18      }
19
20      int z[MAX];
21      func_signed_mag_addition(x, y, z);
22      printf("The first number AND second binary yield:\n");
23      for(int i=0; i < MAX; i = i + 1){
24          printf("%d", z[i]);
25      }
26
27    return 0;
28}
```

Lab Assignment

You should complete the above program under the name of a project COMP2650_Lab05_UWinID that firstly outputs a menu of commands as follows:

```
Enter the command number:
0) Exit
1) Addition in signed-magnitude
2) Subtraction in signed-magnitude
```

Based on the user's chosen number of commands, the program should then ask for the two inputs. After that, the program asks to what base the user wants to see the results. Then, it applies the command and prints out the result in the requested base. For instance, if a user selects (1), the program should accept two inputs as follows:

```
Enter the first binary number:
x0 =
x1 =
...
x7 =

Enter the second binary number:
y0 =
y1 =
...
y7 =
```

When the user enters the two binary numbers, the program asks for a base number to print out the result:

```
Enter the output base:
1) Binary
2) Octal
3) Decimal
4) Hexadecimal
```

Then the program applies the 1) `Addition in signed-magnitude` command on the input x and y and prints the result on the selected base and comes back to the main menu. Other commands should follow the same flow. If the user selects (0), the program ends. Please restrict the user to enter inputs within the range {0,1}. For instance, if the user enters 2, -1, ..., print out an error message and come back to ask for correct inputs. <mark>Also, print an error/warning message whether an overflow happened during the arithmetic.</mark> It is required to write a modular program.

Deliverables

You will prepare and submit the program in one single zip file `COMP2650_Lab05_UWinID.zip` containing the following two items:

1. The entire project folder `COMP2650_Lab05_UWinID` including the code file (`main.c` or `main.cpp`) and executable file (`main.exe` in windows or `main` in mac)

2. The result of the commands in the file `COMP2650_Lab05_Results_UWinID.pdf`. Simply make a screenshots of the results and save (print) them <mark>into a single pdf.</mark>

2. [Optional and if necessary] A readme document in a txt file `COMP2650_Lab05_ReadMe_UWinID.txt`. It explains how to build and run the program as well as any prerequisites that are needed. *Please note that if your program cannot be built and run on our computer systems, you will lose marks.*

In sum, your final `COMP2650_Lab05_UWinID.zip` file for the submission includes 1 folder (entire project folder), 1 image (results snapshot) and 1 txt (report). *Please follow the naming convention as you lose marks otherwise.* Instead of UWinID, use your own UWindsor account name, e.g., mine is hfani@uwindsor.ca, so,

```
COMP2650_Lab05_hfani.zip
```
- **(60%)** `COMP2650_Lab05_hfani`=>(15%) Addition, (15%) Subtraction, (15%) Overflow, (15%) Different Bases in Output.
  - o `conversion.cpp`
  - o `conversion.h`
  - o `arithmetic.cpp`
  - o `arithmetic.h`
  - o `main.c or main.cpp` => Must be compiled and built with no error!
  - o `main.exe or main`
- **(10%)** `COMP2650_Lab05_Results_hfani.pdf`
- **(Optional)** `COMP2650_Lab05_ReadMe_hfani.txt`

(10%) `Modular Programming (using separate header and source files)`
(10%) `Files Naming and Formats`
(10%) `Folder Structure`