School of Computer Science
Faculty of Science

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

Lab#	Date	Title	Due Date	Grade Release Date
Lab 06	Week 06	**Number Systems (2's Complement)**	Feb. 23, 2021 Tuesday Midnight AoE Wednesday 7 AM EDT	March. 01, 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 05 to support arithmetic on binary numbers. Particularly, we want to calculate the addition or subtraction of two binary numbers in signed-2's-complement system.

As we discussed in the lectures, there are different ways to represent negative and positive numbers. In signed-radix-complement, we use positive numbers to show the negative numbers. So, there is no position for the sign. However, we can show that the bit in the highest significant position in signed-2's-complement binary system acts *like* sign bit (not the same though!). This is because given n bits, the maximum number divided by 2 is $(2^n-1) \div 2 = 2^{n-1}$ which is equal to the largest number given n-1 bits, i.e., `0111...111`. So, all binary numbers equal or below this number is positive, having `0` in the n-th bit. This number plus `1` becomes `1000...000`. All numbers equal or above this are negative and have 1 which is non-zero in the n-th bit. In summary, positive numbers has `0` and negative numbers has `1` in n-th bit. The only non-zero value in base-2 is `1`. So, the n-th bit indicate the sign similar to the signed-magnitude. *This is the nice property of base-2, not other bases!*

In C/C++, 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
```
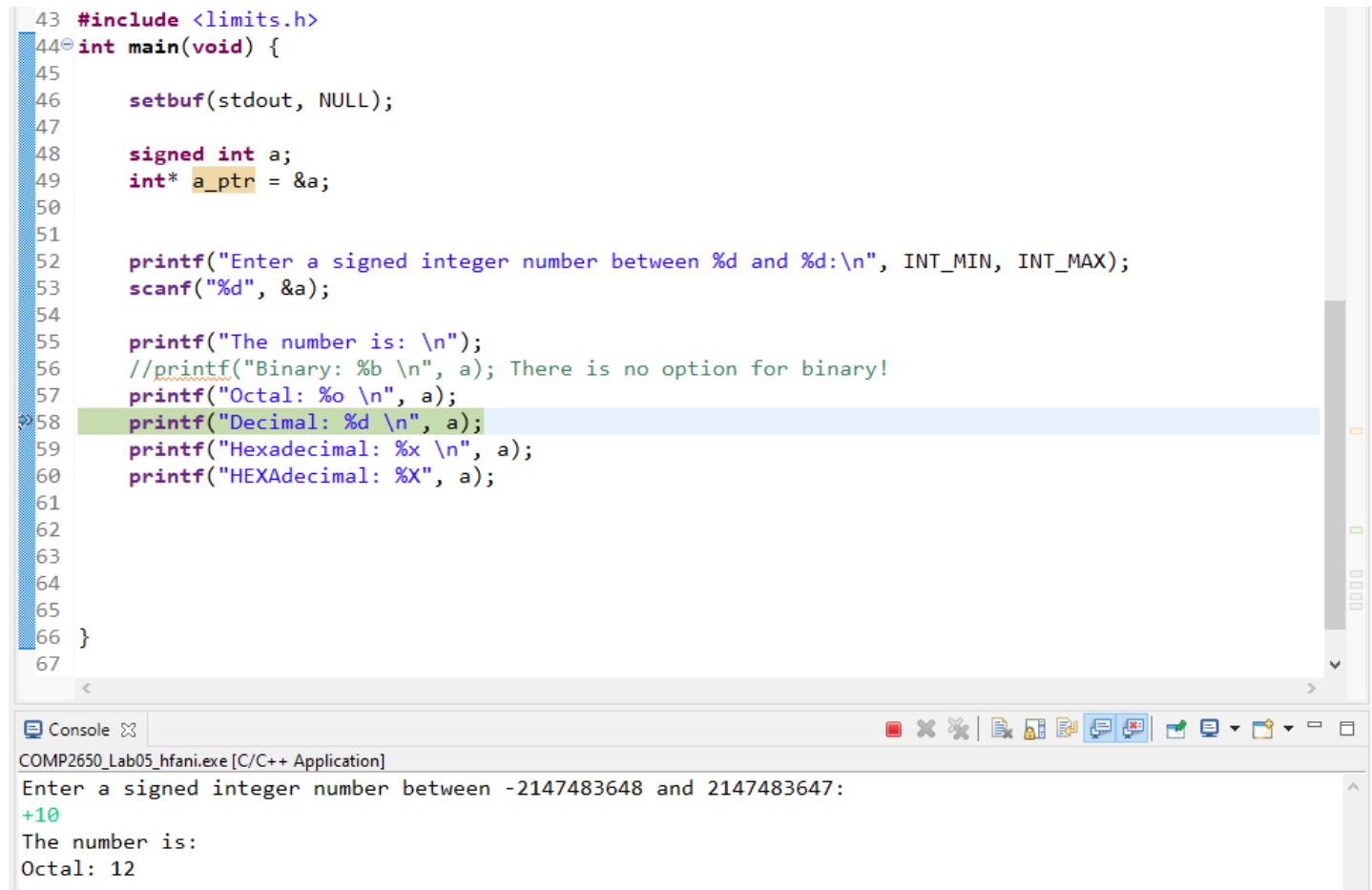
```
16      return 0;
17 }
```

You can check how the positive and negative numbers are stored in C/C++ using signed-2's-complement by looking at memory locations. This can be done in Eclipse in `Debug` mode by using `Memory` and `Monitors`. To `Debug` your code, you can click on [※] and run each line step-by-step using [↻] . You might be asked to `locate` the source code. Then, select the main.cpp from the `src` folder.

For instance, I ran the program in `Debug` mode and put a `breakpoint` when the program wants to print the output:

```
43 #include <limits.h>
44⊖ int main(void) {
45
46      setbuf(stdout, NULL);
47
48      signed int a;
49      int* a_ptr = &a;
50
51
52      printf("Enter a signed integer number between %d and %d:\n", INT_MIN, INT_MAX);
53      scanf("%d", &a);
54
55      printf("The number is: \n");
56      //printf("Binary: %b \n", a); There is no option for binary!
57      printf("Octal: %o \n", a);
58      printf("Decimal: %d \n", a);
59      printf("Hexadecimal: %x \n", a);
60      printf("HEXAdecimal: %X", a);
61
62
63
64
65
66 }
67
```

```
Console ⊠
COMP2650_Lab05_hfani.exe [C/C++ Application]
Enter a signed integer number between -2147483648 and 2147483647:
+10
The number is:
Octal: 12
```
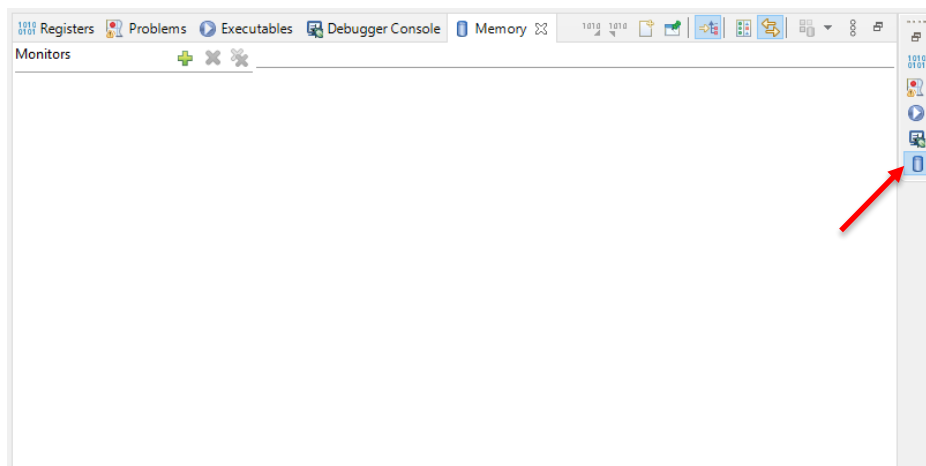
Basically, the program execution is stopped at the line with a `breakpoint` and the program is waiting for you to click on [↻] to execute the next statement. At this point, you are able to open the `Memory` to see the actual memory location for variables and the content. The `Memory` option is usually available in the right side of the Eclipse and looks like [0] .

When the `Memory` panel opens, you have the option to enter an address of a memory location to see the content in the `Monitors` area.
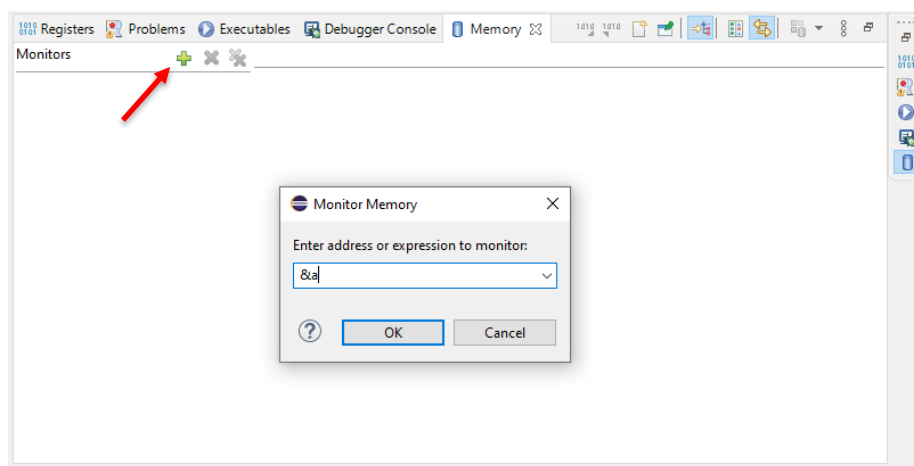
In our sample program, we ask the user to enter a signed number and we store the number is a variable called `a` in line# 08. So, in order to see what the actual value in the memory location for the variable `a` is, we need to find the address of the variable `a`. This can be done by `&` operator in C/C++ language.
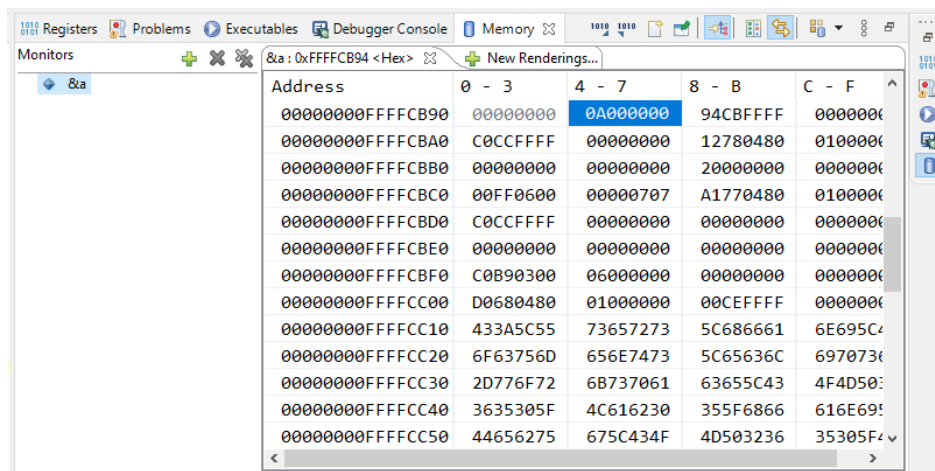
You can define different `renderings` of the memory locations by clicking on the `New Renderings`. I did so and chose `Traditional`:
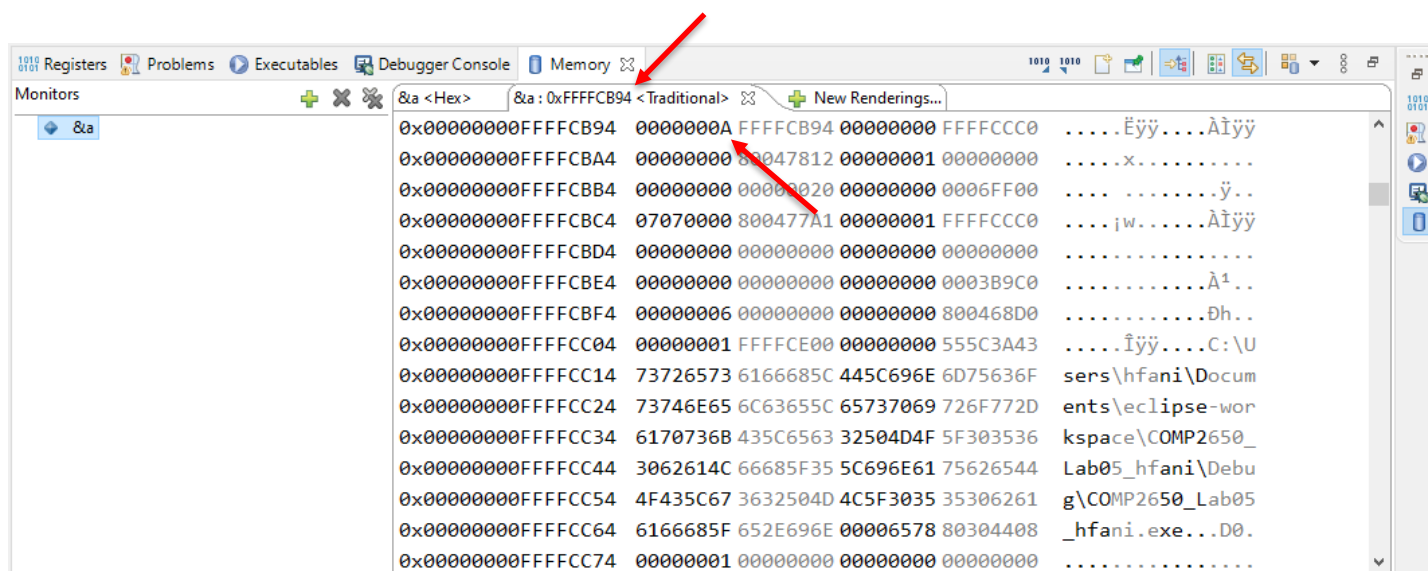
You can see that at the address **&a** (**0xFFFFCB94**), the content of memory is **0000000A** in hexadecimal system which is equal to **+10** in decimal system. Unfortunately, Eclipse does not allow you to see the bits since it would be very long stream of bits (32 bits). Nonetheless, we already knew that **0000000A** is equal to **00…0001010** in binary system.

Let's run the program for −10:
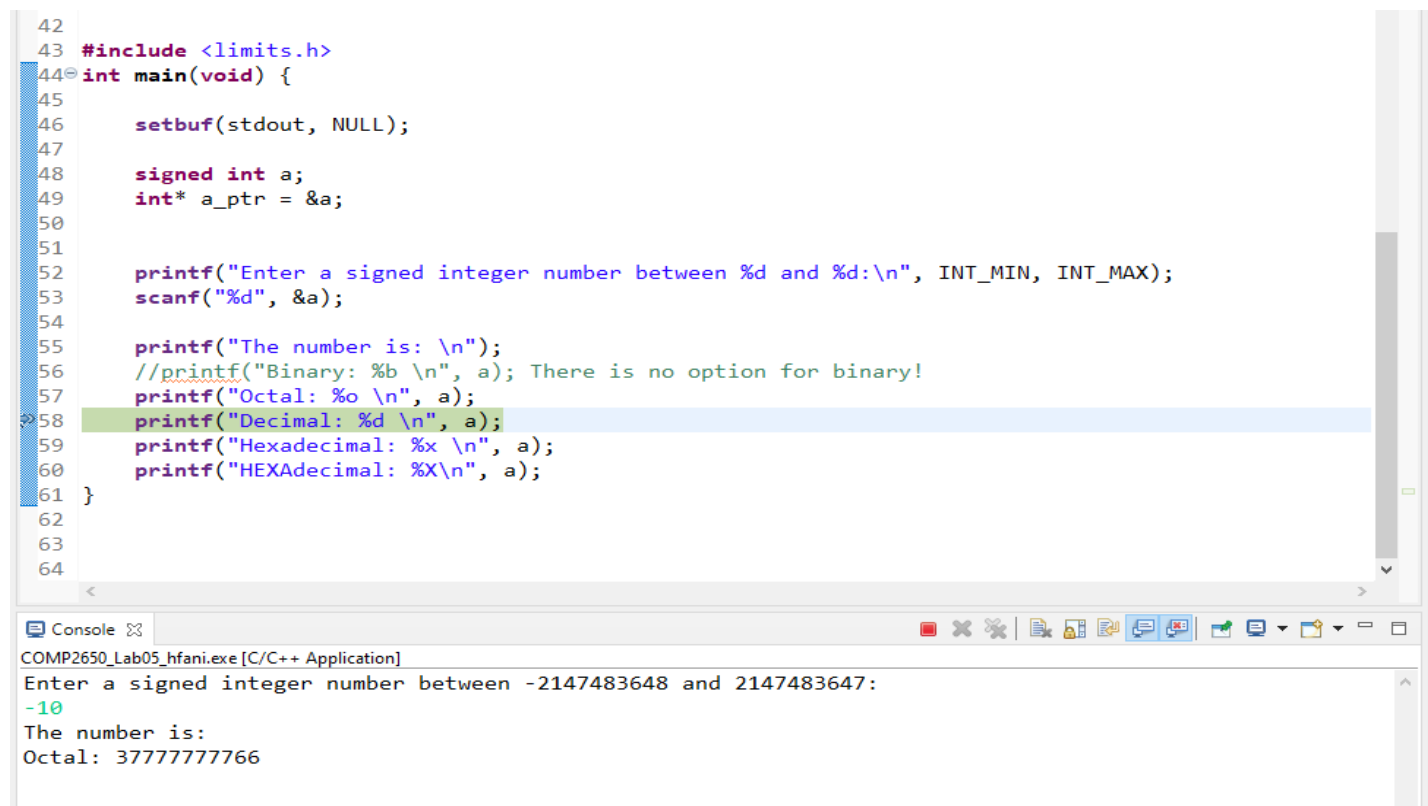
In the second run, the variable `a` has the same memory location[1]. As you can see, −10 is represented as `FFFFFFF6` or `11…11110110`. We expect that as in C/C++ the signed numbers are in the signed-2's-complement system. So, −10 is represented as the 2's-complement of +10:
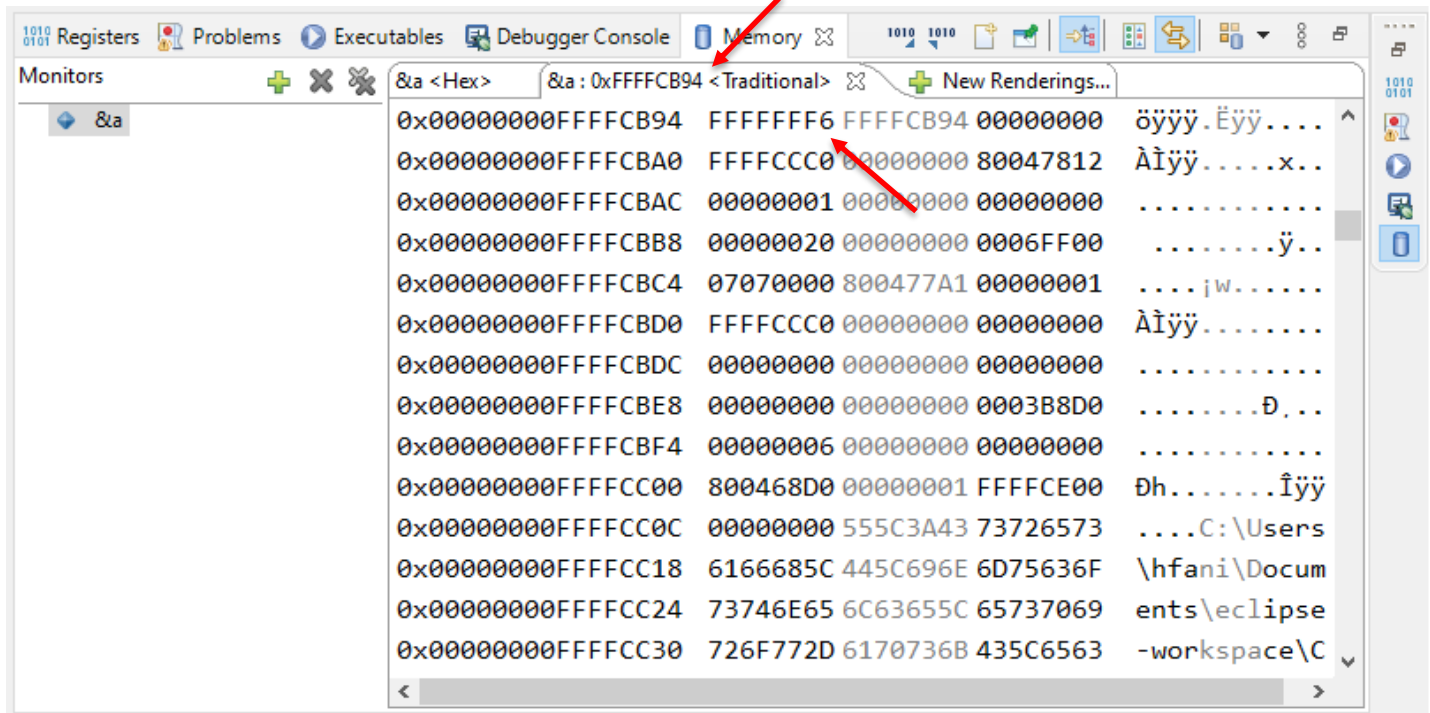
2's-comp(`00…000001010`) = `11…1110110`

You may be tempted to explore what has been stored in other memory locations. Probably, peeking at other programs running in the memory. Is it possible? In the Operating System course, you learn how an OS gives memory to a program to run and more.

Regarding overflow, C/C++ does not raise an error or exception when an overflow happens in **signed** 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     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 }
```

---

[1] In this course, we can assume that this is a coincidence that different runs of the same program has same memory locations. However, in Operating System course, you will learn that this is not a coincidence.

From the library `<limits.h>`, we can find the minimum and maximum for a given type in C/C++. Here, we used `INT_MAX` for the maximum and `INT_MIN` for the minimum **signed** integer, which is equal to 2147483647 and -2147483648 in C/C++:

```
Enter a signed integer number between -2147483648 and 2147483647:
2147483647
The number is:
Octal: 37777777777
Decimal: 2147483647
Hexadecimal: 7fffffff
HEXAdecimal: 7FFFFFFF

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

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

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

This number would be 32 bits with the highest bit equal to 1 and all other remaining bits equal to 0. Why? Simply increment the maximum **signed** integer by one unit in base-2. However, this number is equal to -2147483648 which is the negative number `1000…0000` or `800…000` in C/C++. The addition of two positive numbers becomes a negative number in signed-2's-complement which is an overflow. You see that the program did not raise any error or exception about an overflow though.

Let's create an overflow by entering the minimum number – 1:
```
Enter an unsigned integer number between -2147483648 and 2147483647:
2147483647
The number is:
Octal: 17777777777
Decimal: 2147483647
Hexadecimal: 7fffffff
HEXAdecimal: 7FFFFFFF
```

The minimum signed number (e.q. minimum negative number) minus 1 is equal to (given 32 bits):
```
= 100...0000 – 1
= 100...0000 + 2's-comp(00...0001)
= 100...0000 + 11...1111 (sum of two negative number)
= (carry=1)011...1111
= ignore carry → 011...1111 (positive number)
```

As seen, the result is the maximum positive number. Sum of a two negative number (note: after the 2's complement and change the subtraction to addition) becomes a positive number. This is an instance of overflow, but the program did not raise any error and exception.

In this lab, since our program stores the input bits in an array of integers, we cannot use C/C++'s signed-2's-complement directly. We have to either *i)* implement the signed-2's-complement algorithm for addition and subtraction, or *ii)* convert the input bits stored in the integer arrays to an integer variable and then use the built-in addition or subtraction in C/C++. Also, we want to let the user know whether an overflow happens.

Step2. Writing Modular Programs

In Lab 04, we added a header `arithmetic.h` and a source file `arithmetic.cpp` for arithmetic. Let's add new functions to the header file and the source file to implement all functions related to arithmetic in signed-2's-complement number systems.

arithmetic.h
```
void func_signed_2s_addition(int a[], int b[], int result[]);
void func_signed_2s_subtraction(int a[], int b[], int result[]);
```

arithmetic.cpp
```
#define MAX 8//Byte = 8 bits
void func_signed_2s_addition(int a[], int b[], int result[]){...}
void func_signed_2s_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. A useful hint for subtraction of a from b is to use the addition function to add but with 2's-comp. of b. You already have 2's-complement function in `complement.h` and `complement.cpp`.

Now we are ready to add the headers to our main program and use the functions in each separate file:
```
00 #include <stdio.h>
01 #include "arithmetic.h"
02
03 #include "complement.h"
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
22
23     func_signed_2s_addition(x, y, z);
24     printf("The first number AND second binary yield:\n");
25     for(int i=0; i < MAX; i = i + 1){
26         printf("%d", z[i]);
27     }
28
29     return 0;
30 }
```

Lab Assignment

You should complete the above program under the name of a project COMP2650_Lab06_UWinID that firstly outputs a menu of commands as follows:

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

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 in signed-2's-complement 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-2's-complement` 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. Also, print an error/warning message whether an overflow happened during the arithmetic.

It is required to write a *modular* program. For arithmetic in signed-2's-complement, you can re-use the 2's-complement function in `complement.h` or any other functions you already developed.

Deliverables

You will prepare and submit the program in one single zip file `COMP2650_Lab06_UWinID.zip` containing the following two items:

1. The entire project folder `COMP2650_Lab06_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_Lab06_Results_UWinID.pdf`. Simply make a screenshots of the results and save (print) them into a single pdf.

2. [Optional and if necessary] A readme document in a txt file `COMP2650_Lab06_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_Lab06_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_Lab06_hfani.zip
```
- **(60%)** `COMP2650_Lab05_hfani`=>(15%) Addition, (15%) Subtraction, (15%) Overflow, (15%) Different Bases in Output.
  - o `conversion.cpp`
  - o `conversion.h`
  - o `complement.h`
  - o `complement.cpp`
  - 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_Lab06_Results_hfani.pdf`
- **(Optional)** `COMP2650_Lab06_ReadMe_hfani.txt`

(10%) `Modular Programming (using separate header and source files, ==use 2's comp. function of complement.h for subtraction)==`
(10%) `Files Naming and Formats`
(10%) `Folder Structure`