University of Windsor
Faculty of Science
School of Computer Science

COMP-2650
Computer Architecture I: Digital Design
Winter 2022

Lab#	Date	Title	Due Date	Grade Release Date
Lab 12	Week 12	**Shift Operators**	April 13, 2022, Wednesday 4 AM EDT	April. 18, 2022

This lab's objectives will be to master the topics in logic circuit design 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 explore operations related to our knowledge of Boolean logic obtained in this course. In C/C++, there are three categories of operations, aside from the normal arithmetic operations, dealing with Boolean values. They are:

1) Logical Operation: &&, ||, !
2) Relational Operation: <, <=, >, >=, ==, !=
3) Bitwise Operators: &, |, <<, >>, ~, ^

Bitwise operators are the focus of this lab, and, in particular, we want to work with the >> (shift right) and << (shift left) operators. We know that the binary numbers are stored in flip-flops that can store a binary digit 0 or 1. Additionally, we know that the digits of binary numbers have significance based on the powers of 2. For instance, 101 is $1\times2^2 + 0\times2^1 + 1\times2^0 = 5$ in decimal system. If we are given a byte, we have 8 flip-flops, e.g., D flip-flops, to store this number as 00000101. What happens if I shift the numbers in these flip-flops one time to the right:

1) Remove 1 in the lowest significant bit
2) Shift all other bits in the higher significant bits to the right by one time
3) Repeat the highest significant bit to preserve the sign of the number

This is called *arithmetic* shift right as it preserves the sign of the number after shift operation. So, 00000101 becomes 00000010. It means that we decreased the significance of bits by division by 2. The bit in 3rd position that has the significance of 2^2 now has 2^1 significance. Practically, we divided the number by 2 and kept the integer quotient, which is 5/2 = 2.

If we do another shift to the right, 00000010 becomes 00000001, which is 2/2=1.
If we do another shift to the right, 00000001 becomes 00000000, which is 1/2=0.

As seen, division by 2 can be done via shifting to the right one time. In general, division by 2^i is equal to shift the bits to the right *i* times, as shown below:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
	setbuf(stdout, NULL);

	int a;

	printf("First Operand:");
	scanf("%d", &a);

	printf("%d/2 = %d \n", a, a >> 1 );
	printf("%d/4 = %d \n", a, a >> 2 );
	printf("%d/8 = %d \n", a, a >> 3 );
}
```

Sample runs would be:

```
First Operand:64
Normal divisions:
64/2 = 32
64/4 = 16
64/8 = 8
Divisions by arithmetic shifts to right:
64/2 = 32
64/4 = 16
64/8 = 8

First Operand:35
Normal divisions:
35/2 = 17
35/4 = 8
35/8 = 4
Divisions by arithmetic shifts to right:
35/2 = 17
35/4 = 8
35/8 = 4
```

How about negative numbers?

```
First Operand:-35
Normal divisions:
-35/2 = -17
-35/4 = -8
-35/8 = -4
Division by arithmetic shifts to right:
-35/2 = -18
-35/4 = -9
-35/8 = -5

Normal divisions:
-32/2 = -16
-32/4 = -8
-32/8 = -4
Divisions by arithmetic shifts to right:
-32/2 = -16
-32/4 = -8
-32/8 = -4
```

As seen, the results of divisions are the floor of the quotient, e.g., $-35/2 = \lfloor -17.5 \rfloor = -18$. The results are different by 1 unit for divisions that have remainder compared to when you normally divide a negative number by powers of 2. Why? This is not the case for -32, though.

The division operation is a costly action that requires multiple clock pulses. However, shifting to the right can be done in only 1 clock pulse. So, whenever a division by 2 or powers of 2 is needed, shifting is a preferable and wise choice to have an efficient (fast) program.

What happens if we do shift left (<<)?

Lab Assignment

You should implement that outputs a menu of commands as follows:

```
Enter the command number:
0) Exit
1) Powers of 2
```

```
   2) Divisions of 2
```

If a user selects (1), the program asks for a decimal number and a number to power:

```
Enter a decimal number:20
Enter a power number:4
```

Then, the program should print out the result of 20×2^4 *by the arithmetic shift to the left* as shown below:

```
20*(2^4)  = 320
```

and come back to the main menu. Also, same procedure but divisions of 2 for option (2). If the user selects (0), the program ends.

Deliverables
You will prepare and submit the program in one single zip file `lab12_{UWinID}.zip` containing the following items:
1. The code files and executable file (`main.exe` in windows or `main` in unix/mac)
2. The result of the commands in the file `results.png/jpg`. Simply make a screenshot of the results.
3. [Optional and if necessary] A readme document in a txt file `readme.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.*

`Lab12_hfani.zip`
- (70%) `main.c =>` Must be compiled and built with no error!
- (05%) `main.exe or main`
- (10%) `results.jpg/png`
- (Optional) `readme.txt`

(10%) `Modular Programming (using separate functions)`
(05%) `Files Naming and Formats`

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, `lab11_hfani.zip`.