CS 202 – Assignment #2 (Booth's Algorithm with sign-extended method)

Purpose: Learn simple class, multiple classes, multiple files implementation, I/O files
Points: 100

Assignment: **Video Link for Assignment 2: https://www.youtube.com/watch?v=PuL3MYliFbM**

Design a C++ class to do binary multiplication using booth's algorithm with the sign-extended method. In this assignment, the user will be asked to choose between reading data from an input file and reading data from the console window, as well as set/get the multiplicand and multiplier using accessor and mutator functions. The input.txt file contains the multiplicand, multiplier, and amount of decimal bits. When the user chooses to read data from a file, the users program should read the input values in decimal, convert them to signed binary, perform binary multiplication and write the result to an output file (output.txt) that includes all algorithm steps. When the user chooses manual entry or read data from the terminal, the program should read the decimal number, convert it to signed binary, perform binary multiplication, and output the result to the terminal using the booth's technique.

The provided boothMain() function will serve as a driver for the program and this assignment has *Multiplication* class and *FileClass* class. The *Multiplication* UML class diagram is as follows:

Multiplication
-manMcand: int
-manMplier: int
+multiplicand: int
+multiplier: int
+bits: int
+result: int
+Multiplication()
+Multiplication(int mc, int ml, int b)
+setMultiplicand(int mcand): void
+setMultiplier(int mplier): void
+getMultiplicand() const: int
+getMultiplier() const: int
+writeOutputToFile(std::ofstream &outputFile) const: void
+performBoothAlgorithm(std::ostream &output) const: void
+decimalToBinary(int num, int binary[], int length) const: void
+reverseArray(int arr[], int length) const: void
+printBinary(int num, int bits, std::ostream &output) const: void
+add(int accumulator[], int x[], int mplierLength) const: void
+complement(int accumulator[], int n) const: void
+arithShiftRight(int accumulator[], int mplierBinary[], int &mplierLSB, int mplierLength, std::ostream &output) const: void
+display(int accumulator[], int mplierBinary[], int mplierLength, std::ostream &output) const: void
+performBoothAlgorithmUserInput(): void
+binaryToDecimal(int accumulator[], int mplierBinary[], int mplierLength, std::ostream &output) const: void

You will need to develop implementation files based on the given UML diagrams. Your implementation files should be fully commented. Refer to the example execution for output formatting.

Function Descriptions:

The following are more detailed descriptions of the required functions.

class Multiplication

- Multiplication() is the default constructor and all the member variables have to be initialized
- *Multiplication*(**int** mc, **int** ml, **int** b)is the parameterized constructor to initialize the member variables according to the given parameters. Also initialize public variable *results* to zero
- **void** setMultiplicand(**int** mcand) is the setter function to set the member variables according to the given parameter
- **void** *setMultiplier(**int** mplier)* is the setter function to set the member variables according to the given parameter
- **int** getMultiplicand() **const** is to return multiplicand
- **int** getMultiplier() **const** is to return multiplier
- ***void** decimalToBinary(**int** num, **int** binary[], **int** length) **const***- This function is to convert the num to binary and store it in binary[]. Here length gives the number of bits in binary[].
- ***void** reverseArray(**int** arr[], **int** length) **const***- This function is to reverseArray.
- ***void** performBoothAlgorithm(std::ostream &output) **const***- Read the decimal inputs from input.txt file and perform booth's algorithm.
  - Declare arrays with size 10 for accumulator, multiplier, multiplicand, and tempArray
  - Declare local variables as needed to perform booth's algorithm.
  - You need to call the functions decimalToBinary(**parameters**) **const,** reverseArray(parameters) const, add(**parameters**) const, complement(parameters) const, arithShiftRight(parameters) const, display(parameters) const, printBinary(**parameters**) **const**; accordingly to fulfill the algorithm.
  - Initially the user reads the decimal values of multiplicand, multiplier and number of bits needed for that multiplication from input.txt file.
  - After performing the multiplication, write the result to output file (output.txt). Please see the sample output for more details.
  - Algorithm details:
  - num1 × num2
    → multiplicand × multiplier = product
  - Qn bit (Also known as Product LSB or multiplier LSB) and Qn+1 bit (Also known as Booth-bit (BB))

    00 -> no operation –
    shift product right (with sign extend)
    01 -> product = left side of product + multiplicand
    shift product right (with sign extend)
    10 -> product = left side of product - multiplicand
    shift product right (with sign extend)
    11 -> no operation
    shift product right (with sign extend)

    - Product is a combination of accumulator bits and multiplier bits
    - Left side of product (ls/prod) initially set to 0 (Accumulator)
    - Right side of product (rs/prod) is initially set to the multiplier (QR)

Example 1 (4-bit):
2 × 6 (0010 × 0110)

multiplicand = 0010 multiplier = 0110

step	action	multiplicand	product
0	initial	0010	0000 0110 0
1	a: 00 -> no operation	0010	0000 0110 0
	b: shift right product	0010	0000 0011 0
2	a: 10 -> prod = ls/prod - Mcand	0010	1110 0011 0
	b: shift right product	0010	1111 0001 1
3	a: 11 -> no operation	0010	1111 0001 1
	b: shift right product	0010	1111 1000 1
4	a: 01 -> prod = ls/prod + Mcand	0010	0001 1000 1
	b: shift right product	0010	0000 1100 0

- Note: In the above example, each step has two actions. However, while implementing the algorithm, I did arithmetic shift right operations in one row only when Qn and On+1 are equal (00 or 11)
- For 01 -> product = left side of product + multiplicand, you can copy the multiplicand to a tempArray and use add(**parameters**) const to perform addition operation. Then use arithShiftRight(parameters) const to shift the product right by one method using sign extend. Here sign-extend means the MSB of product is extended. For example, if you have the following scenarios:

  Scenario 1
  a: 10 -> prod = ls/prod - Mcand 0010 1110 0011 0
  b: shift right product 0010 1111 0001 1

  Scenario 2
  a: 01 -> prod = ls/prod + Mcand 0010 0001 1000 1
  b: shift right product 0010 0000 1100 0

- ***void*** add(**int** accumulator[], **int** x[], **int** mplierLength) **const** is to add (bit-wise operation) accumulator bits with multiplicand bits. Carry is forwarded to the next positions. However, if you get a carry (1) after performing last iteration that is MSB position, you ignore the carry. See the sample above and also watch the video for more information.
- ***void complement(int binary[], int n) const***– This function does the 1's complement to the array that is passed as a parameter. Here the parameter can be a multiplicand binary or accumulator binary. You can pass the required [] according to your logic.
- **void** arithShiftRight(**int** accumulator[], **int** mplierBinary[], **int** &mplierLSB, **int** mplierLength, std::ostream &output) **const** is to shift the product right by one method using sign extend. Here sign-extend means the MSB of product is extended. Please check the above scenarios listed.
- ***void display(int accumulator[], int mplierBinary[], int mplierLength,*** std::ostream &output) **const** is to print the accumulator binary array and multiplier binary array into an output file when reading data from input.txt else print data on console window for manual entry option.

- **void** performBoothAlgorithmUserInput()- This function works pretty much similar to ***void*** *performBoothAlgorithm(std::ostream &output)*. However, the user call this function through option 2 from boothMain.cpp. This function should prompt the messages like below. The prompt messages are shown in red color below only for a reference to understand easily. The numbers (decimal) in red are entered by the user on Terminal window.

  Number of multiplicand bits=4

  Enter the multiplicand in decimal=2

  Multiplicand in binary: 00010 (Hint: Use decimalToBinary())

  Number of multiplier bits=4

  Enter the multiplier in decimal=6

  Multiplier in binary: 00110 (Hint: Use decimalToBinary())

step	qn	q[n+1]	action	multiplicand	Product
			initial	0010	0000 0110
1	0	0	a:00->no operation;shiftRight	0010	0000 0011
2	1	0	a:10->prod=ls/prod-Mcand	0010	1110 0011
			b:shift right product	0010	1111 0001
3	1	1	a:11->no operation;shiftRight	0010	1111 1000
4	0	1	a:01->prod=ls/prod+Mcand	0010	0001 1000
			b:shift right product	0010	0000 1100

Result in binary=0000 1100
Result in decimal=12

- **void** *binaryToDecimal(**int** accumulator[], **int** mplierBinary[], **int** mplierLength, std::ostream &output)* ***const-*** is to convert binary to decimal.
  Steps to convert binary to decimal:
  Example 1: 0000 1101; If the MSB (sign-bit) is 0, then it is treated as a positive number. Check the following method.
  Excluding Sign-bit (0): $0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  $= 0 + 0 + 0 + 8 + 4 + 0 + 1 = 13$ in decimal

  Example 2: 1011 0010; If the MSB (sign-bit) is 1, then it is treated as a negative number. Check one of the possible methods for the conversion. You just add the weights where the bit positions are 0 only.
  Excluding Sign-bit (1):
  Bit positions that are 0: $2^0 + 2^2 + 2^3 + 2^6 = 1 + 4 + 8 + 64 = 77$
  Add '1' to the above step and so the value is 78. But consider it as -78 in decimal.

- ***void*** *writeOutputToFile(std::ofstream &outputFile)* ***const*** is to write outputs to a file. The user can check if outfile is open and then call performBoothAlgorithm*(outputFile)*. If the output file is not open, print an error message on terminal window "Failed to write to output file."

class fileInputOutput

FileClass
+inFile: ifstream
+outFile: ofstream
+fileError: bool
+fileInputOutput(string file_name)
+~fileInputOutput()

- *fileInputOutput(string file_name)* is a constructor with parameter (input filename). This constructor is to open input and output text files. If the input or output files are not found then print the error messages accordingly. Check the sample output for the error messages.
- *~fileInputOutput()* is the destructor and is to close the input and output files.

Global function: (1 function is listed in fileInputOutput.h)

- ***void** getData(std::vector<Multiplication> &multiplications, **const** std::string &filename))*- This function should open the input file. If input file is not open, print an error message "Failed to open the file." on terminal window. If the input file is open read multiplicand, multiplier, and bits amount (all lines) into the vector (pass by reference). Pass by reference makes the implementation of the code faster. [Hint: Use push_back()]

Linking Instructions:

A **make** file is provided. As projects get larger, compiling a series of source files by hand becomes tedious. Make is a utility that will read a make file (named **makefile** by default), compile applicable source(s), and build the executable file. Assuming the main file is named **`boothMain.cpp`**, and the implementation files are named as **`multiplicationImp.cpp`** and **`fileInputOutputImp.cpp.`** The following are the instructions to build the executable.

```
kishore@kishore-virtual-machine% make
```

Which will create an executable named **`booth_algorithm`**. The provided make file assumes these file names. As such, you should not change the file names for this assignment.

Following Files provided to Students:

- makefile, multiplication.h, fileInputOutput.h, boothMain.cpp, input.txt, Assignment2 pdf, sample_output file, and Booths_algorithm_with_sign-extended.pdf

Submission:

- All files must compile and execute on Ubuntu and compile with C++11.
- Submit source files via the on-line submission
  - Submit only multiplicationImp.cpp and fileInputOutputImp.cpp files only.
- Once you submit, the system(CodeGrade) will score the project and provide feedback.
  - If you do not get full score, you can (and should) correct and resubmit.
  - You can re-submit an unlimited number of times before the due date/time.
  - However, there is a limit to submit per hour (check that)
- Late submissions will be accepted for a period of 24 hours after the due date/time for any assignment. Late submissions will be subject to a ~2% reduction in points per an hour late. If you submit 1 minute - 1 hour late -2%, 1-2 hours late -4%, … , 23-24 hours late -50%. This means after 24 hours late submissions will receive an automatic 0.

Program Header Block

All program and header files must include your name, section number, assignment, NSHE number, input and output summary. The required format is as follows:

```
/*
Name: MY_NAME, NSHE, CLASS-SECTION, ASSIGNMENT
Description: <per assignment>
Input: <per assignment>
Output: <per assignment>
*/
```

Failure to include your name in this format will result in a loss of up to 5%.

Code Quality Checks

A C++ linter[1] is used to perform some basic checks on code quality. These checks include, but are not limited to, the following:

- Unnecessary 'else' statements should not be used.
- Identifier naming style should be either camelCase or snake_case (consistently chosen).
- Named constants should be used in place of literals.
- Correct indentation should be used.
- Redundant return/continue statements should not be used.
- Selection conditions should be in the simplest form possible.
- Function prototypes should be free of top level *const*.

Not all of these items will apply to every program. Failure to to address these guidelines will result in a loss of up to 5%.

Scoring Rubric

Scoring will include functionality, code quality, and documentation. Below is a summary of the scoring rubric for this assignment.

Criteria	Weight	Summary
Compilation	-	Failure to compile will result in a score of 0.
Program Header	5%	Must include header block in the required format (see above).
General Comments	10%	Must include an appropriate level of program documentation.
Line Length	5%	No lines should exceed more than eighty (80) characters.
Code Quality	5%	Must meet some basic code quality checks (see above)
Program Functionality (and on-time)	75%	Program must meet the functional requirements as outlined in the assignment. Must be submitted on time for full score.

1 For more information, refer to: https://en.wikipedia.org/wiki/Lint_(software)

Quick or Summary of Instructions that can help you with this Assignment:

- Please watch the video that is shared on the first page of this document
- Please read the instructions carefully before you start coding the assignment
- Do not make changes in files that are provided to you (makefile, multiplication.h, fileInputOutput.h, boothMain.cpp, input.txt, Assignment2 pdf and sample_output file)
- Implement all the functions in their respective implementation files as listed in their header files
- Implement the Global function in fileInputOutputImp.cpp
- Check the following guidelines or descriptions. The order of guidelines is not only the possible approach and it can be done in multiple ways.
  i. fileInputOutputImp.cpp can be implemented first.
  ii. Implement multiplicationImp.cpp
  iii. Do not alter boothMain.cpp
- In multiplicationImp.cpp, initially you can implement the default constructor and parameterized constructor.
- Implement setters (mutators) and getters (accessor) functions.
- Implement decimalToBinary(parameters), reverseArray(parameters), printBinary(parameters), complement(parameters), display(parameters), add(parameters), arithShiftRight(parameters), binaryToDecimal(parameters), performBoothAlgorithm(parameters), performBoothAlgorithmUserInput(parameters), writeOutputToFile(parameters)
- It can be easy when you implement the functions in the above order, however that might not be the only approach as suggested.
- You are not permitted to create new functions other than those provided.
- Please check the "Booths_Algorithm_with_sign-extend_method.pdf" to better understand the booth's algorithm. The algorithm does binary multiplication. Please read through the booth's algorithm provided in Function Descriptions to easily implement performBoothAlgorithm(parameters).
- Your output for option 1 must match with the given sample_output.txt file
- Your output for option 2 and 3 must match with the Example Execution given below Check the sample output.txt

Helpful Link to understand booth's algorithm:
https://www.youtube.com/watch?v=AcY43GfnON8

Example Execution:

Below is an example program execution.

```
kishore@kishore-VirtualBox:Ast2_F23$ make
g++ -Wall -Wextra -pedantic -std=c++11 -g -c boothMain.cpp -o boothMain.o
g++ -Wall -Wextra -pedantic -std=c++11 -g -c fileInputOutputImp.cpp -o
fileInputOutputImp.o
g++ -Wall -Wextra -pedantic -std=c++11 -g -c multiplicationImp.cpp -o
multiplicationImp.o
g++ -Wall -Wextra -pedantic -std=c++11 -g boothMain.o fileInputOutputImp.o
multiplicationImp.o -o booth_algorithm
kishore@kishore-VirtualBox:Ast2_F23$ ./booth_algorithm
Choose an option:
1. Read input from file, perform Booth's Algorithm, and write output files
2. Take input from User and perform Booth's Algorithm
3. Set multiplicand and multiplier using setters
4
Invalid option selected
kishore@kishore-VirtualBox:Ast2_F23$ ./booth_algorithm
Choose an option:
1. Read input from file, perform Booth's Algorithm, and write output files
2. Take input from User and perform Booth's Algorithm
3. Set multiplicand and multiplier using setters
1
File opened successfully.
kishore@kishore-VirtualBox:Ast2_F23$ ./booth_algorithm
Choose an option:
1. Read input from file, perform Booth's Algorithm, and write output files
2. Take input from User and perform Booth's Algorithm
3. Set multiplicand and multiplier using setters
2
You selected Option 2.

 Number of multiplicand bits=4

 Enter the multiplicand in decimal=5

 Multiplicand in binary: 00101

 Number of multiplier bits=4

 Enter the multiplier in decimal=7

 Multiplier in binary: 00111
step   qn     q[n+1]       action                          multiplicand  Product
                           initial                         0101          0000 0111
1      1      0            a:10->prod=ls/prod-Mcand        0101          1011 0111
                           b:shift right product           0101          1101 1011
2      1      1            a:11->no operation;shiftRight   0101          1110 1101
3      1      1            a:11->no operation;shiftRight   0101          1111 0110
4      0      1            a:01->prod=ls/prod+Mcand        0101          0100 0110
                           b:shift right product           0101          0010 0011
Result in binary=0010 0011
Result in decimal=35
kishore@kishore-VirtualBox:Ast2_F23$ ./booth_algorithm
Choose an option:
1. Read input from file, perform Booth's Algorithm, and write output files
2. Take input from User and perform Booth's Algorithm
3. Set multiplicand and multiplier using setters
3
You selected Option 3.
Mutator functions to set multiplicand and multiplier
Enter multiplicand= 34
Enter multiplier= 12
Accessor functions to get multiplicand and multiplier
Muliplicand=34
Multiplicand in binary: 00100010
Muliplier=12
Multiplier in binary: 00001100
```