CSCE 231, Fall 2019
Lab 2: Integer Lab

Code Due: 11:59pm on the day prior to your assigned lab section

In this assignment, you will become more familiar with bit-level representations of integers. You'll do this by implementing integer arithmetic for 16-bit signed and unsigned integers using only bitwise operators.

1 Getting Started

Download *lab2.zip* or *lab2.tar* from Canvas and copy it to a directory on the *csce.unl.edu* server. Once copied, unzip the file. Three of the four files (*alu.h*, *alu.c*, and *integerlab.c*) contain the starter code for this assignment. The fourth file (*Makefile*) tells the `make` utility how to compile the code. To compile the program, type:

```
make
```

This will produce an executable file called `integerlab`. If you're using your own computer and you don't have `make` available to you, then you can compile the program by typing:

```
gcc -std=c99 -Wall -o integerlab integerlab.c alu.c
```

2 Description of IntegerLab Files and Tasks

2.1 alu.h

Do not edit *alu.h*.

This header file contains the function declarations for **add()**, **subtract()**, **multiply()**, and **divide()**. It also declares a global variable:

is_signed This boolean is used to indicate whether the functions should treat the values as signed integers or as unsigned integers.

Finally, it contains three type defintions for arithmetic results:

addition_subtraction_result This structure has two fields. The **result** field is to store the sum or difference (as appropriate). The **overflow** field should be set to be `true` if the full answer does not fit in the 16-bit **result** and `false` if the full answer does fit.

multiplication_result This structure has three fields. The **product** field is to store the lowest 16 bits of the product. The **full_product** field is to store the full 32-bit product. The **overflow** field should be set to be `true` if the full answer does not fit in the 16-bit **product** and `false` if the full answer does fit.

division_result This structure has three fields. The **quotient** field is to store the integer quotient, and the **remainder** field is to store the integer remainder. Mathematically, $dividend \div divisor = quotient + \frac{remainder}{divisor}$. The **division_by_zero** field should be set to `true` if the **divide()** function cannot compute the quotient because the divisor is 0 and **false** otherwise.

2.2 integerlab.c

Do not edit *integerlab.c*.

This file contains the driver code for the lab. It parses the input, calls the appropriate arithmetic function, and displays the output.

2.3 alu.c

This file contains stubs for the four functions you need to edit. Add your name in comments as indicated, and write the code. In addition to the four functions, you may add helper functions to make your code more modular; you may only place these helper functions in *alu.c*.

When you implement these functions, you may NOT use C's arithmetic operators: $+$ $-$ $*$ / %. *You will receive no credit for the assignment if you use a prohibited operator.* You may only use bitwise and &, bitwise or |, bitwise exclusive-or ˆ, bitwise complement ˜, and bit shifts << >>.

Exception: You MAY use C's addition operator once to update one loop variable in **add()**.

add()

Takes two 16-bit integers and adds them. The sum should be stored as a 16-bit value in **result**. If **is_signed** is true, treat all values as signed integers; otherwise, treat all values as unsigned integers. If addition overflowed, set **overflow** to `true`.

subtract()

Takes two 16-bit integers and subtracts the second from the first. The difference should be stored as a 16-bit value in **result**. If **is_signed** is true, treat all values as signed integers; otherwise, treat all values as unsigned integers. If subtraction overflowed, set **overflow** to `true`.

multiply()

Takes two 16-bit integers and multiplies them. The lowest 16 bits of the product should be stored in **product**, and the full product should be stored in **full_product** as a 32-bit value. If the full product doesn't fit in the 16-bit **result** then set **overflow** to `true`.

- Only implement multiplication for unsigned integers. You do not need to implement multiplication for signed integers.
- Your multiplication algorithm MUST be polynomial in the number of bits. *You will receive no credit for multiplication if your algorithm is superpolynomial.* The brute-force approach of repeatedly adding **multiplicand** to itself **multiplier** times is a $\mathcal{O}(2^n)$ algorithm, where n is the number of bits.
- For full credit, be able to multiply any two non-negative integers that fit in 16 bits; for partial credit, be able to multiply by a power-of-two.

divide()

Takes two 16-bit integers and divides the first by the second. The integer quotient should be stored in **quotient**, and the remainder should stored in **remainder**. If the divisor is zero, then set **division_by_zero** to `true` and provide any value as the quotient and remainder.

- Only implement division for unsigned integers. You do not need to implement division for signed integers.
- Your Division algorithm MUST be polynomial in the number of bits. *You will receive no credit for division if your algorithm is superpolynomial.* The brute-force approach of repeatedly subtracting **divisor** from **dividend** is a $\mathcal{O}(2^n)$ algorithm, where n is the number of bits.
- For full credit, be able to dividy by a power-of-two; for bonus credit, be able to divide by an arbitrary non-negative integer.

3 Running IntegerLab

After you've compiled the program, you can run it as `./integerlab unsigned` to perform arithmetic on unsigned integers or as `./integerlab signed` to perform arithmetic on signed integers. you will be prompted to input a simple two-operator arithmetic expression. After you do so, the result of the computation will be printed and then you'll be prompted to enter another arithmetic expression. For example:

```
Input a simple two-operator arithmetic expression: 50+3
50 + 3 = 53
Input a simple two-operator arithmetic expression:
```

This will continue until you enter a blank line, at which point the program will terminate.

You can enter the inputs as either decimal or as hexidecimal. If at least one input is hexidecimal, then the output will be hexidecimal. For example:

```
Input a simple two-operator arithmetic expression: 55 + 0x4
0x37 + 0x4 = 0x3b
```

I suspect that you'll mostly use decimal inputs/outputs; however, being able to use hexidecimal inputs/outputs may help you with debugging. I have provided you with another tool for testing your lab solution. While logged into the *csce.unl.edu* server, there is a reference solution that you can use by typing `~cse231/lab2/reference unsigned` (or `signed`). If you're unsure whether your solution is providing the correct answer, you can compare it to the reference solution's answer.

4 Grading

This assignment is worth 40 points. You will receive no credit if you use C's arithmetic operators for anything other than updating one loop variable.

Run `./integerlab unsigned`

_____ **+2** Satisfies additive identity; for example, 5+0 = 5

_____ **+2** Performs addition; for example, 32+10 = 42

_____ **+2** Sums between 2^{15} and $2^{16} - 1$ do not overflow; for example, 30000+5000 = 35000

_____ **+2** Sums greater than $2^{16} - 1$ do overflow; for example, 60000+6000 = 464 and reports Overflow

_____ **+2** Satisfies subtractive identity; for example, 5-0 = 5

_____ **+2** Performs subtraction; for example, 40000-300 = 39700

_____ **+2** Differences of zero do not overflow; for example, 10-10 = 0

_____ **+2** Negative differences do overflow; for examlple, 2-3 = 65535 and reports Overflow

_____ **+1** Satisfies multiplicative identity; for example, 3*1 = 3

_____ **+1** Satisfies multiplicative zero; for example, 3*0 = 0

_____ **+1** Performs multiplication when multiplier is a power of two; for example, 3*4 = 12

_____ **+1** Performs multiplication when multiplier is not a power of two; for example, 3*5 = 15

_____ **+1** Products less than 2^{16} do not overflow when multiplier is a power of two; for example, 3000*16 = 48000

_____ **+1** Products less than 2^{16} do not overflow when multiplier is not a power of two; for example, 3000*20 = 60000

_____ **+1** Products greater than $2^{16} - 1$ do overflow when multiplier is a power of two; for example, 3000*32 = 30464 and reports Overflow with the full answer 0x17700

_____ **+1** Products greater than $2^{16} - 1$ do overflow when multiplier is not a power of two; for example, 3000*25 = 9464 and reports Overflow with the full answer 0x124f8

_____ **+1** Satisfies divisive identity; for example, 8/1 = 8

_____ **+1** A value divides itself once; for example, 8/8 = 1

_____ **+1** Satisfies divisive zero; for example, 0/8 = 0

_____ **+1** Reports division by zero; for example, 8/0 reports Division by Zero

_____ **+1** Divides a power of two by another power of two; for example, 32/4 = 8

_____ **+1** Divides an arbitrary non-negative integer by a power of two; for example, 30/4 = 7 remainder 2

_____ **Bonus +1** Divides an arbitrary non-negative integer by one of its factors; for example, 30/5 = 6

_____ **Bonus +1** Divides an arbitrary non-negative integer by an arbitrary integer; for example, 32/5 = 6 remainder 2

Run `./integerlab signed`

_____ **+1** Satisfies additive identity; for example, 5+0 = 5

_____ **+1** Performs addition with positive values; for example, 32+10 = 42

_____ **+1** Sums less than 2^{15} do not overflow; for example, 30000+2000 = 32000

_____ **+1** Sums greater than $2^{15} - 1$ do overflow; for example, 30000+3000 = -32536 and reports Overflow

_____ **+1** Performs addition with a negative value; for example, -2+3 = 1

_____ **+1** Satisfies subtractive identity; for example, 5-0 = 5

_____ **+1** Performs subtraction; for example 200-50 = 150

_____ **+1** Can subtract a greater value from a lesser without overflowing; for example, 10-20 = -10

_____ **+1** Can subtract from a negative value; for example, -10-10 = -20

_____ **+1** Differences beyond -2^{15} overflow; for example, -30000-3000 = 32536 and reports Overflow

Search *alu.c* for arithmetic operators (+ - * / %), and examine algorithms.

_____ **-8** Uses superpolynomial multiplication algorithm, such as brute-force repeated addition

_____ **-8** Uses superpolynomial division algorithm, such as brute-force repeated subtraction

_____ Score assignment as 0 if arithmetic operators are used for anything other than updating one loop variable