Lab 4

Integer Representation and Arithmetic Lab

Due: Week of February 14, before the start of your lab section

This is an individual-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with the professor and the TAs. Sharing code with or copying code from another student or the internet is prohibited.

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.

The instructions are written assuming you will edit and run the code on your account on the *csce.unl.edu* Linux server. If you wish, you may edit and run the code in a different environment; be sure that your compiler suppresses no warnings, and that if you are using an IDE that it is configured for C and not C++.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Apply bit operations in non-trivial applications
- Illustrate ripple-carry binary addition
- Recognize whether integer overflow has occurred
- Explain the relationship between multiplication, division, and bit shifts
- Express multiplication as an efficient use of other functions

Continuing Forward

You used bit operations for toy applications in KeyboardLab; in IntegerLab you will use bit operations to re-implement integer arithmetic. This increased familiarity with bit operations will pay off handsomely in next lab and also in the I/O labs. Your increased understanding of integer operations will improve your performance on the exam.

During Lab Time

During your lab period, the TAs will review ripple-carry addition, overflow for unsigned and signed integers, and bit shift-based multiplication and division. During the remaining time, the TAs will be available to answer questions.

No Spaghetti Code Allowed

In the interest of keeping your code readable, you may *not* use any **`goto`** statements, nor may you use any **`continue`** statements, nor may you use any **`break`** statements to exit from a loop, nor may you have any functions **`return`** from within a loop.

1 Scenario

All work at the Pleistocene Petting Zoo has stopped while Archie tries to find a ~~gullible~~ reasonable insurance company. Rather than furloughing staff, he's asked everybody to help out with his other startup companies for a week or two. He specifically asked that you help out with Eclectic Electronics.

Herb Bee, the chief engineer, explains that Eclectic Electronics is developing a patent-pending C-licon tool that will convert C code into an integrated circuit that has the same functionality as the original C code. To test it out, he tasked you with writing the code to implement an Arithmetic Logic Unit (ALU). Your task will be to implement integer addition, subtraction, multiplication, and division. Because bitwise operations and bit shift operations have been implemented, you will be able to use C's bitwise and bit shift operators, but because arithmetic operations have not yet been implemented, you cannot use C's arithmetic operators. Because C library functions generally make use of arithmetic operations (which have not yet been implemented), you cannot use library functions.

2 Getting Started

Download *integerlab.zip* or *integerlab.tar* from Canvas or ~cse231 on *csce.unl.edu* and copy it to your account on the *csce.unl.edu* Linux server. Once copied, unpackage the file. Three of the five files (*alu.h*, *alu.c*, and *integerlab.c*) contain the starter code for this assignment. The fourth file (*integergrader.c*) contains code to run your code through the lab's rubric. The last 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`.

3 Description of IntegerLab Files and Tasks

3.1 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.

3.2 integerlab.c

Do not edit *integerlab.c*.

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

3.3 integergrader.c

Do not edit *integergrader.c*.

This file contains alternate driver code for the lab. It generates inputs for each of the test cases, calls the appropriate arithmetic function, and displays the result. After all test cases have been run, an initial score will be calculated (this score is subject to change due to violating the assignment's requirements).

3.4 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 may not use floating-point operators as a substitute for integer operators. You may use only functions that you write yourself). If you use `printf` statements for debugging instead of using a debugger, remove them before turning in your code. *You will receive no credit for functions that use a prohibited operator or function.* You may only use bitwise and &, bitwise or |, bitwise exclusive-or ˆ, bitwise complement ∼, and bit shifts << >>.

Hints:

- The value 0x8, if right-shifted one position becomes 0x4 which is logically `true`. If right-shifted by one position a second time, the value becomes 0x2 which is logically `true`. If right-shifted by one position a third time, the value becomes 0x1 which is logically `true`. If right-shifted by one position a fourth time, the value becomes 0x0 which is logically `false`. If you generalize this idea, you may find a way to control a loop without an arithmetic operator.
- After you have written the **`add()`** function, you may use it in other functions to control loops and for other other purposes.

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`.

- Addition must work for both signed and unsigned integers.
- You may find it beneficial for another part of the lab if you implement a 32-bit full adder; you can have **`add()`** call the 32-bit full adder.

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`.

- Subtraction must work for both signed and unsigned integers.

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 divide by a power-of-two; for bonus credit, be able to divide by an arbitrary non-negative integer.

4 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
```

We suspect that you'll mostly use decimal inputs/outputs; however, being able to use hexidecimal inputs/outputs may help you with debugging.

Turn-in and Grading

When you have completed this assignment, upload *alu.c* to Canvas.

This assignment is worth 40 points.

Run `./integerlab unsigned` (or run `./integergrader`)

_____ **+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` (or run `./integergrader`)

_____ **+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

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

_____ **-13** The **`add()`** function (or a helper function for **`add()`**) uses an arithmetic operator (+ - * / %) or a function you did not write

_____ **-13** The **`subtract()`** function (or a helper function for **`subtract()`**) uses an arithmetic operator (+ - * / %) or a function you did not write

_____ **-8** The **`multiply()`** function (or a helper function for **`multiply()`**) uses an arithmetic operator (+ - * / %) or a function you did not write, *and/or* **`multiply()`** uses a superpolynomial multiplication algorithm, such as but not limited to brute-force repeated addition

_____ **-6** (also no bonus) The **`divide()`** function (or a helper function for **`divide()`**) uses an arithmetic operator (+ - * / %) or a function you did not write, *and/or* **`divide()`** uses a superpolynomial division algorithm, such as but not limited to brute-force repeated subtraction

_____ **-1** for each **`goto`** statement, **`continue`** statement, **`break`** statement used to exit from a loop, or **`return`** statement that occurs within a loop.

Epilogue

Herb smiles as he hands you the the test results from the latest integrated circuit fab batch. "C-licon successfully turned your code into an ALU. Nicely done!" I think maybe it's time to use it to introduce some optimizations into the Floating Point Unit (FPU) on our experimental microprocessor.

To be continued...