Lab 4

Integer Representation and Arithmetic Lab

Due: Week of February 19, before the start of your lab section

During your scheduled lab time, you may, ***but are not required to****, form a partner group of 2 students. When necessary, there may be a group of 3 students. During your scheduled lab time, and only during your scheduled lab time, you may discuss problem decomposition and solution design with your lab partner. After your scheduled lab time, 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.*

Obtaining the starter code Download integerlab.zip or integerlab.tar from Canvas, or copy integerlab.zip or integerlab.tar from ~cbohn2/csce231 on *cse-linux-01.unl.edu*.

Runtime environment We will grade this assignment by compiling and running it on the *cse-linux-01.unl.edu* Linux server; you should compile and test your code on the *cse-linux-01.unl.edu* Linux server before turning it in.

Submitting your work When you have completed this assignment, submit *basetwo.c* and *alu.c* to the assignment in Canvas.

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 the *cse-linux-01.unl.edu* Linux server. If you wish, you may develop your 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++.

Contents

Learning Objectives

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

- Recognize several powers of two
- Apply bit operations in non-trivial applications
- Illustrate ripple-carry binary addition
- Express common boolean and comparison operations in terms of other functions

- Recognize whether integer overflow has occurred
- Explain the relationship between multiplication, division, and bit shifts
- Express multiplication as an efficient use of other functions

During Lab Time

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

During your lab period (and only during your lab period), you may discuss problem decomposition and solution design with your lab partner. *Be sure to add your name and your partner's name to the top of your source code files.* To receive full credit for the work you and your partner do during lab time, you must be an active participate in the partnership. In accordance with the School of Computing's Academic Integrity Policy, we reserve the right to adjust your calculated grade if you merely "tag along" and let your partner do all the thinking.

If you worked with a partner, then before leaving lab, upload *basetwo.c* and *alu.c*to Canvas to create a record of what you and your partner worked on during lab time. After you have finished the full assignment, you will upload your completed work as well. If you did not work with a partner, you do not need to upload your files until you have completed the assignment.

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.

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. Even though high-level languages' *logical*

boolean operations normally are not part of an ALU, Herb wants you to include these in the ALU to see if that can make some programs run faster. 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.

1 Assignment Summary

In this assignment, you will implement a 16-bit ALU. You will implement logical boolean operations and arithmetic using only bitwise operations, bit shift operations, a small amount of starter code, and other code that you will write. Specifically, the ALU's functions are:

- Logical NOT
- Logical AND
- Logical OR
- Comparisons, both equality and inequality
- Addition, both signed and unsigned values
- Subtraction, both signed and unsigned values
- Multiplication, unsigned values only
- Division, unsigned values only, and only when the divisor is a power of two

The operands for the arithmetic and comparison functions will be 16-bit integers. The arguments for the logical boolean functions are declared to be 32-bit values because that may make some of your code easier to implement; however, we will only grade these functions with 16-bit operands. The utility and "building block" functions that you will implement along the way will have arguments sized according to the functions' needs.

1.1 Constraints

You may not use C's built-in arithmetic and logical boolean operations. Specifically, you may not use addition (+), increment (++), subtraction(-), decrement(--), multiplication(∗), division(/), modulo (%), logical NOT (!), logical AND (&&), or logical OR (||). You may not use arithmetic-and-assignment operators (+=, -=, ∗=, /=, %=). You may not use C's comparators; specifically, you are not permitted to use equals (==), not-equals (!=), less-than (<), less-than-or-equal-to (<=), greater-than-or-equal-to (>=), or greater-than (>). You may not use floating-point operators as a substitute for integer operators.

You may not use arrays as truth tables. You may not use inline assembly code.

You may not use any libraries. This prohibition includes, but is not limited to, C's *math* library. If you find yourself **`#include`**ing any header files other than *alu.h*, you are violating this constraint.

The time-complexity of all operations must be polynomial (or constant) in the number of bits required to represent the values. Inefficient solutions, such as (but not limited to) implementing multiplication by repeatedly adding the multiplicand to itself *multiplier* times, or implementing division by repeatedly subtracting the divisor from the dividend, are prohibited.

You may not change the signatures of any functions that you are required to complete for this assignment. If you write helper functions that are not required, then they may have any signature that you deem necessary (your helper functions must also comply with the assignment's constraints).

You are permitted to use C's bitwise operations and bit shift operations, including bitwise complement ($\sim$), bitwise AND (&), bitwise inclusive OR (|), bitwise XOR ($\hat{}$), left shift ($<<$), and arithmetic and logical right shift ($>>$), and their corresponding assignment operators. You may cast between integer types. You may use loops, conditionals, function calls, structs, and (except as noted above) arrays. And, of course, you may use any of the provided starter code and any code that you write by yourself for this assignment.

You can check whether you're using a disallowed arithmetic, logical, or comparison operator by running the constraint-checking Python script:

```
python constraint-check.py integerlab.json
```

1.2 Problem Decomposition

The logical operations offer little opportunity for decomposition: determine whether each operand should be interpreted as true or false, and find a way to return the appropriate boolean value.

Inequality comparison requires that you use a mathematical operation to deduce which operand is greater than the other. Equality comparison can be decomposed either into using that mathematical operation and deducing that neither operand is greater than the other, or it can be decomposed into determining whether the two operands have the same bit pattern.

Arithmetic, however, is a little more complex. Before you can add sixteen bit positions, you must be able to add one bit position. By chaining the carry bits for each bit position, you can add an arbitrary number of bit positions. You must also be able to detect both signed and unsigned overflow. Finally, you must be able to use an adder to perform subtraction.

Before you can multiply arbitrary values together, you must be able to multiply a value by a power of two. You must also be able to add those intermediate products together; because the multiplicand and multiplier are 16-bit values, you need to be prepared to add 32-bit values as part of this multiplication.

Implementing division requires that you be able to determine whether the divisor is 0 and also that you be able to divide a value by a power of two.

1.3 Suggestion for Loops

Some of the functions in this assignment will require loops that execute a predetermined number of iterations. The obvious approach, the one that you were surely taught, is to set a loop counter to 0 and then increment the counter until it reaches the desired number of iterations. Because you cannot use C's arithmetic operations in this assignment, you cannot use this approach until *after* you have implemented addition.

An alternative that will work within this assignment's constraints is to use bit shifts. Notice that a 16-bit unsigned integer initially set to 1 will become 0 after it has been left-shifted 16 times. Similarly, a 32-bit unsigned integer initially set to 1 will become 0 after it has been left-shifted 32 times. You might find that this alternative "loop counter" will be a useful bitmask in some of your functions.

(If you need to iterate an arbitrary number of times, then an unsigned integer initially set to 2^{n-1} will become 0 after it has been right-shifted n times, assuming that the integer type you use has at least n bits. We do not anticipate that you will need to iterate an arbitrary number of times.)

2 Getting Started

Download *IntegerLab.zip* or *IntegerLab.tar* from Download integerlab.zip or integerlab.tar from Canvas, or copy integerlab.zip or integerlab.tar from ~cbohn2/csce231 on *cse-linux-01.unl.edu*. and copy it to the *cse-linux-01.unl.edu* Linux server. Once copied, unpackage the file. Three of the five files (*alu.h*, *basetwo.c*, and *alu.c* Most of the remaining files contain driver code or test code for this assignment. *Makefile* tells the `make` utility how to compile the code. To compile the program, type:

```
make
```

This will produce an executable file called *integerlab*.

When you run the program with the command *`./integerlab`*, you will be prompted:

```
Enter a one- or two-operand logical expression,
    a two-operand comparison expression, a two-operand arithmetic expression,
    "lg <value>" or "exponentiate <value>" to test your powers-of-two code,
    "is_negative <value>" to determine if 2's complement value is negative,
    "add1 <binary_value1> <binary_value2> <carry_in>" for 1-bit full adder,
    "add32 <hex_value1> <hex_value2> <carry_in>" for 32-bit ripple-carry adder,
    "mul2 <hex_value> <hex_power_of_two>" for power-of-two-multiplier,
    or "quit":
```

When you enter a value, if it is prepended with *`0x`* then the parser will parse it as a hexadecimal value; otherwise, except as noted in Sections 4.1 and 4.2, the parser will treat it as a decimal value.

For now, type *`quit`* to exit the program.

2.1 Description of IntegerLab Files

2.1.1 alu.h

Do not edit *alu.h*.

This header file contains two type definitions:

one_bit_adder_t is a structure to hold the 1-bin inputs (`a`, `b`, `c_in`) and 1-bit outputs (`sum`, `c_out`) of a one-bit full adder.

alu_result_t is a structure to hold the outputs from an arithmetic logic unit. Its fields are:

- `result`, a 16-bit bit vector that is considered "the" result of the computation
- `supplemental_result`, a 16-bit bit vector that stores additional result data from instructions that place their results in two registers
- `unsigned_overflow`, a 1-bit flag to indicate whether overflow occurred when interpreting the source operands as unsigned values
- `signed_overflow`, a 1-bit flag to indicate whether overflow occurred when interpreting the source operands as signed values
- `divide_by_zero`, a 1-bit flag to indicate whether there was an attempt to divide by zero.

The header file also contains two macros, **`is_zero()`** and **`is_not_zero()`** to bootstrap your ALU code. These macros act like functions and return a boolean value to indicate whether an integer is 0 or not.[1]

The header file also contains several function declarations. The requirements for these functions will be discussed later in this assignment.

2.1.2 basetwo.c

This is the first of two files that you will edit.

There are two functions in *basetwo.c* that will allow you to demonstrate an understanding of powers-of-two and/or an understanding of some uses of bit shifts.

lg() returns the base-2 logarithm of its argument, assuming its argument is a positive power-of-two; if the argument is 0 or is not a power-of-two, then there are no guarantees about the function's return value

exponentiate() creates a power-of-two by raising 2 to the provided exponent, assuming the exponent is a non-negative value strictly less than 32; if the argument is negative or is greater than 31, then there are no guarantees about the function's return value

[1] The astute student will quickly realize that **`is_not_zero()`** is not necessary and, with a little thought, will realize that they can write **`is_zero()`** as a function within the constraints of this assignment.

These functions are inverses of each other: $x = \log_2 2^x$, and $y = 2^{\log_2 y}$.

Strictly speaking, you can write your ALU code without these functions; however, some students in the past had difficulty finding solutions for their ALU code without obtaining a base-2 logarithm and/or calling a function to create a power-of-two. Rather than tempt you to violate one of the assignment's constraints by calling the *math* library's **`log2()`**, **`exp2()`**, and/or **`pow()`** functions, we now have you write your own code for these functions.

2.1.3 alu.c

This file will contain most of the code that you write, and the functions in *alu.c* are in the order in which you will likely write them.

- A simple check

  is_negative() returns a boolean value to indicate whether the argument, when interpreted as a signed integer, is negative

- Equality comparisons

  equal() returns `true` if and only if $value1 = value2$

  not_equal() returns `true` if and only if $value1 \neq value2$

- Logical operations

  logical_not() returns the logical inverse of the argument

  logical_and() returns the logical conjunction of the two arguments

  logical_or() returns the logical disjunction of the two arguments

- Addition and subtraction

  one_bit_full_addition() performs addition for one bit position, determining both the sum bit and the carry-out bit

  ripple_carry_addition() adds two 32-bit values to each other and to a carry-in bit

  add() adds two 16-bit values to each other

  subtract() subtracts a 16-bit value from another

- Signed Inequality comparisons

  less_than() returns `true` if and only if $value1 < value2$ when these values are signed integers

  at_most() returns `true` if and only if $value1 \leq value2$ when these values are signed integers

  at_least() returns `true` if and only if $value1 \geq value2$ when these values are signed integers

greater_than() returns `true` if and only if *value*1 > *value*2 when these values are signed integers

- Unsigned multiplication and division

  multiply_by_power_of_two() multiplies the first argument by the second, assuming that the second argument is zero or a power of two; there are no guarantees if this assumption is not satisfied

  unsigned_multiply() multiplies two 16-bit values to each other, if the arguments are interpreted as unsigned integers

  unsigned_divide() divides a 16-bit value by another, if the arguments are interpreted as unsigned integers

- Signed multiplication and division (bonus credit)

  signed_multiply() multiplies two 16-bit values to each other, if the arguments are interpreted as signed integers

  signed_divide() divides a 16-bit value by another, if the arguments are interpreted as signed integers

2.1.4 Other files

There are a few other files that serve as driver code and code to help you test your code. Do not modify these files.

3 Utility Functions, Equality Comparisons, and Logical Boolean Operations

( ) Open *basetwo.c* in your editor. You will see the stubs of two functions there.

3.1 exponentiate()

This function produces a power of two. Treating its argument as an exponent, it returns the value $2^{exponent}$ when $0 \leq exponent < 32$. If $exponent < 0$ or $exponent \geq 32$, the function must return *something*, but we do not require that it return any specific value.

A characteristic of powers of two is that when represented in binary, exactly one bit is 1 and all others are 0.

( ) Implement the **`exponentiate()`** function.

You should be able to implement this function with a single line of code, but you may use more than one line.

3.2 lg()

This function is the inverse of **exponentiate()**: it produces the base-2 logarithm of its argument. Assuming its argument is a power of two, then if $power\_of\_two = 2^{exponent}$, the function will return *exponent*. If the argument is not a power of two, the function must return *something*, but we do not require that it return any specific value.

There are some very short solutions that will work if you have already implemented arithmetic. Since you haven't, the function stub suggests an alternative: use a **switch** statement, enumerating the 32 possible cases and returning the appropriate value in each case.

( ) Implement the **lg()** function.

Check your work

( ) Compile and run *./integerlab*, trying a few values.

- Note that you will receive a warning for an unused variable in **ripple_carry_addition()**; this is okay for now

For example:

```
Enter a one- or two-operand logical expression,
    a two-operand comparison expression, a two-operand arithmetic expression,
    "lg <value>" or "exponentiate <value>" to test your powers-of-two code,
    "is_negative <value>" to determine if 2's complement value is negative,
    "add1 <binary_value1> <binary_value2> <carry_in>" for 1-bit full adder,
    "add32 <hex_value1> <hex_value2> <carry_in>" for 32-bit ripple-carry adder,
    or "quit": exponentiate 10
expected: 2**10 == 0x00000400 == 1024
actual:   2**10 == 0x00000400 == 1024

Enter ... "lg <value>" or "exponentiate <value>" ... or "quit": lg 1024
expected: log2 1024 == log2 0x00000400 == 10
actual:   log2 1024 == log2 0x00000400 == 10

Enter ... "lg <value>" or "exponentiate <value>" ... or "quit": lg 0x0400
expected: log2 1024 == log2 0x00000400 == 10
actual:   log2 1024 == log2 0x00000400 == 10
```

The expected results come from the *math* library's **exp2()** and **log2()** functions. The actual results come, of course, from the code you wrote.

( ) Check your code with other values, comparing your actual results with the expected results.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

( ) Open *alu.c* in your editor. You will see the stubs of several functions there.

3.3 is_negative()

Real ALUs typically have hardware dedicated to quickly determining whether a value is 0 or not, and *alu.h* includes the macros **`is_zero()`** and **`is_not_zero)()`** to serve this purpose. Real ALUs also typically have hardware dedicated to quickly determine whether an integer, when treated as a signed value, is negative.

( ) Implement **`is_negative()`** to determine whether its argument, when interpreted as a signed value, is negative.

The function shall return 1 when the value is negative, and 0 when it is non-negative. You should be able to implement this function in a single line of code (but you may use more, provided you comply with the assignment's constraints).

3.4 equal() and not_equal()

The general approach to comparing two values requires arithmetic, as discussed in Section 5. If you do not anticipate testing the equality of two values in your arithmetic, then you can postpone implementing **`equal()`** and **`not_equal()`** until later. On the other hand, if you think that you might need to test for equality as part of your arithmetic functions, there is a simple test for equality that does not require arithmetic.

To implement each of the **`equal()`** and **`not_equal()`** functions, you will need one 2-operand bitwise operation, either bitwise AND, bitwise OR, or bitwise XOR. Recognize that so far you have only three tests you can make on the output of that bitwise operation: **`is_zero()`**, **`is_not_zero()`**, and **`is_negative()`**.

( ) Consider what the output of each of those three bitwise operations would be if the two operands were the same, and what the output would be if the two operands were different.

One of those six possibilities will have a predictable output that can be evaluated with one or more of the three existing tests.

( ) Implement **`equal()`** to return `true` if and only if its two arguments are the same value.

( ) Implement **`not_equal()`** to return `true` if and only its two arguments are not the same value.

Check your work

( ) Compile and run *./integerlab*, trying a few values.

- Note that you will receive a warning for an unused variable in **ripple_carry_addition()**; this is okay for now

For example:

```
Enter ... "is_negative <value>" ... or "quit": is_negative 1
expected: 1 (0x0001) is not negative
actual:   1 (0x0001) is not negative

Enter ... "is_negative <value>" ... or "quit": is_negative -1
expected: -1 (0xFFFF) is negative
actual:   -1 (0xFFFF) is negative

Enter ... a two-operand comparison expression ... or "quit": 1 == 1
expected: (1 == 1) = 1
actual:   (1 == 1) = 1

Enter ... a two-operand comparison expression ... or "quit": 1 != 1
expected: (1 != 1) = 0
actual:   (1 != 1) = 0

Enter ... a two-operand comparison expression ... or "quit": 1 == -1
expected: (1 == -1) = 0
actual:   (1 == -1) = 0

Enter ... a two-operand comparison expression ... 1 != -1
expected: (1 != -1) = 1
actual:   (1 != -1) = 1
```

( ) Check your code with other values, comparing your actual results with the expected results.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

Implementing logical NOT, logical AND, and logical OR is not quite as simple as applying the corresponding bitwise operations, but it is very nearly so.

3.5 logical_not()

When is a value considered to be `false`? From among the tests that you have available, one of these will return `true` when that condition is satisfied, and `false` when it is not.

( ) Implement **`logical_not()`** to return `true` if and only if its two arguments are considered to be `true`.

3.6 logical_and() and logical_or()

When is a value considered to be `true`? From among the tests that you have available, one of these will return `true` when that condition is satisfied, and `false` when it is not. Specifically, it will return a 1 or a 0, as appropriate.

You cannot simply apply bitwise AND and bitwise OR to the original values because their bits might not line up – for example, `0x5 & 0xA == 0x0`. After you reduce these values to a 1 or a 0, then their bits will line up, and tjen you can apply a bitwise operation to the results of the aforementioned test.

( ) Implement **`logical_and()`** to return `true` if and only if its two arguments are considered to be `true`.

( ) Implement **`logical_or()`** to return `true` if and only if at least one of its two arguments is considered to be `true`.

Note: you are not required to preserve C's "shortcut evaluation" of the logical AND and logical OR operations. Indeed, you cannot because the semantics of C's functions requires that both arguments to **`logical_and()`** and **`logical_or()`** be evaluated before your code has the opportunity to determine their truth values.

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

- Note that you will receive a warning for an unused variable in **`ripple_carry_addition()`**; this is okay for now

For example:

```
Enter a one- or two-operand logical expression or "quit": !0
expected: !0 = 1
actual:   !0 = 1

Enter a one- or two-operand logical expression or "quit": !1
expected: !1 = 0
actual:   !1 = 0
```

```
Enter a one- or two-operand logical expression or "quit": 0 && 42
expected: 0 && 42 = 0
actual:   0 && 42 = 0

Enter a one- or two-operand logical expression or "quit": 0 || 73
expected: 0 || 73 = 1
actual:   0 || 73 = 1
```

( ) Check your code with other values, comparing your actual results with the expected results.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

4 Addition and Subtraction

Now that you've warmed up to bitwise operations and implementing operations without using C's built-in operations, let us turn your attention to arithmetic. Before you can add two n-bit values, you must be able to add two 1-bit values.

4.1 One Bit Full Adder

In the **`one_bit_full_addition()`** function, you will implement a 1-bit full adder; that is, an adder that takes two operand bits and a carry-in bit, and it produces a sum bit and a carry-out bit.

The function takes one argument, a structure containing five fields. As described in Section 2.1.1, these five fields are the operand bits `a` and `b`, the carry-in bit `c_in`, the sum bit `sum`, and the carry-out bit `c_out`. When the structure is passed in to the function, only `a`, `b`, and `c_in` are populated. Your task is to populate the `sum` and `c_out` fields, and return the structure.

( ) Implement a 1-bit full adder using bitwise operations.

Because the fields are guaranteed to be strictly 1 or 0, you do not need to apply any of the test functions to reduce them to 1 or 0.

Check your work

( ) Compile and run *`./integerlab`*, trying all possible values.

- Note that you will receive a warning for an unused variable in **`ripple_carry_addition()`**; this is okay for now

When you enter the inputs for your 1-bit full adder, only the least significant bit of each operand will be used. For example:

```
Enter ... "add1 <binary_value1> <binary_value2> <carry_in>" ...: add1 0 0 0
expected: 0 + 0 + 0 = 0 carry 0
actual:   0 + 0 + 0 = 0 carry 0

Enter ... "add1 <binary_value1> <binary_value2> <carry_in>" ...: add1 0 0 1
expected: 0 + 0 + 1 = 1 carry 0
actual:   0 + 0 + 1 = 1 carry 0
```

Check your code with all eight possible inputs, comparing your actual results with the expected results.

4.2 Ripple-Carry Adder

( ) Use your 1-bit full adder to implement a 32-bit ripple-carry adder.

As a reminder, in a ripple-carry adder, the carry-out bit from bit position n becomes the carry-in bit for bit position $n + 1$.

Use whatever code that you need, that does not violate any of this assignment's constraints, to populate the input fields of a `one_bit_adder_t` variable and pass that variable to **`one_bit_full_addition()`**. Use the `sum` bit to contribute to the 32-bit sum and the `c_out` bit as the `c_in` bit of the next bit position. Repeatedly do this until you have added all 32-bit positions, resulting in the 32-bit sum. (Discard the final carry-out.)

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

When you enter the inputs for your 32-bit adder, the operands will be interpreted as hexadecimal values even if you omit the leading "0x", and only the least-significant bit of the carry-in will be used. For example:

```
Enter ... "add32 <hex_value1> <hex_value2> <carry_in>" ...: add32 0x1a 0x22 0
expected: 0x0000001A + 0x00000022 + 0 = 0x0000003C
actual:   0x0000001A + 0x00000022 + 0 = 0x0000003C

Enter ... "add32 <hex_value1> <hex_value2> <carry_in>" ...: add32 1a 22 1
expected: 0x0000001A + 0x00000022 + 1 = 0x0000003D
actual:   0x0000001A + 0x00000022 + 1 = 0x0000003D
```

( ) Check your code with other values, comparing your actual results with the expected results.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

When you test your 32-bit adder, don't forget to test larger values, too, such as:

```
Enter ... "add32 <hex_value1> <hex_value2> <carry_in>" ...:
                                              add32 0x76543210 0x89ABCDEF 0
expected: 0x76543210 + 0x89ABCDEF + 0 = 0xFFFFFFFF
actual:   0x76543210 + 0x89ABCDEF + 0 = 0xFFFFFFFF

Enter ... "add32 <hex_value1> <hex_value2> <carry_in>" ...:
                                              add32 0x76543210 0x89ABCDEF 1
expected: 0x76543210 + 0x89ABCDEF + 1 = 0x00000000
actual:   0x76543210 + 0x89ABCDEF + 1 = 0x00000000
```

4.3 16-Bit Addition

The **`add()`** function, along with the other arithmetic functions, returns an `alu_result_t` structure.

Having implemented a 32-bit adder, you can use it for your 16-bit addition function. Following the convention that the most significant bit is bit_{31}, and the least significant bit is bit_0 then:

- The 16-bit sum will be in $bits_{15..0}$, the lower 16 bits of the 32-bit adder's sum.
- Viewed from the perspective of 16-bit addition:
  - The 32-bit sum's bit_{15} is the 16-bit value's most significant bit, and bit_0 is the 16-bit value's least significant bit.
  - The 32-bit sum's bit_{16} is the final carry-out of 16-bit addition.

( ) Use the 32-bit adder to add `addend` to `augend` (*i.e.*, calculate $augend + addend$).

( ) Place the 16-bit sum in the `alu_result_t` variable's `result` field.

( ) Assume that the operands are unsigned 16-bit integers and determine whether overflow occurred; set the `alu_result_t` variable's `unsigned_overflow` flag accordingly.

( ) Assume that the operands are signed 16-bit integers and determine whether overflow occurred; set the `alu_result_t` variable's `signed_overflow` flag accordingly.

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

For example:

```
Enter ... a two-operand arithmetic expression... or "quit": 3 + 15
UNSIGNED ADDITION
    expected result (hexadecimal): 0x0003 + 0x000F = 0x0012
    expected result (unsigned):    3 + 15 = 18 overflow: false
    actual result (hexadecimal):   0x0003 + 0x000F = 0x0012
    actual result (unsigned):      3 + 15 = 18 overflow: false
SIGNED ADDITION
    expected result (hexadecimal): 0x0003 + 0x000F = 0x0012
    expected result (signed):      3 + 15 = 18 overflow: false
    actual result (hexadecimal):   0x0003 + 0x000F = 0x0012
    actual result (signed):        3 + 15 = 18 overflow: false

Enter ... a two-operand arithmetic expression... or "quit": 0x6000 + 0x3000
UNSIGNED ADDITION
    expected result (hexadecimal): 0x6000 + 0x3000 = 0x9000
    expected result (unsigned):    24576 + 12288 = 36864 overflow: false
    actual result (hexadecimal):   0x6000 + 0x3000 = 0x9000
    actual result (unsigned):      24576 + 12288 = 36864 overflow: false
SIGNED ADDITION
    expected result (hexadecimal): 0x6000 + 0x3000 = 0x9000
    expected result (signed):      24576 + 12288 = -28672 overflow: true
    actual result (hexadecimal):   0x6000 + 0x3000 = 0x9000
    actual result (signed):        24576 + 12288 = -28672 overflow: true
```

If you are performing this lab on the *cse-linux-01.unl.edu* Linux server, then the expected overflow flags are obtained directly from flags set in the processor's ALU and are authoritative.[2]

( ) Check your code with other values, comparing your actual results with the expected results.

- Use positive and negative operands.
- Generate sums that produce signed overflow, sums that produce unsigned overflow, and sums that produce neither.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

4.4 16-Bit Subtraction

Having implemented a 32-bit adder, you can use it for your 16-bit subtraction function.

[2]If you are not performing this lab on the *cse-linux-01.unl.edu* Linux server and receive the compile-time warning "Some of the code to determine the *expected* supplemental_result and *expected* flags is not yet defined" then the expected overflow flags should not be trusted.

( ) Use the 32-bit adder to subtract `subtrahend` from `menuend` (*i.e.*, calculate *menuend*−*subtrahend*).

- Use the adder using the technique discussed in Chapter 3 and in lecture
- Apply a `0xFFFF` bitmask to your arguments when you call **`ripple_carry_addition()`** to make sure that only the 16-bit values are passed to **`ripple_carry_addition()`**![3]

( ) Place the 16-bit difference in the `alu_result_t` variable's `result` field.

( ) Assume that the operands are unsigned 16-bit integers and determine whether overflow occurred; set the `alu_result_t` variable's `unsigned_overflow` flag accordingly.

( ) Assume that the operands are signed 16-bit integers and determine whether overflow occurred; set the `alu_result_t` variable's `signed_overflow` flag accordingly.

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

For example:

```
Enter ... a two-operand arithmetic expression... or "quit": 15 - 25
UNSIGNED SUBTRACTION
    expected result (hexadecimal): 0x000F - 0x0019 = 0xFFF6
    expected result (unsigned):    15 - 25 = 65526 overflow: true
    actual result (hexadecimal):   0x000F - 0x0019 = 0xFFF6
    actual result (unsigned):      15 - 25 = 65526 overflow: true
SIGNED SUBTRACTION
    expected result (hexadecimal): 0x000F - 0x0019 = 0xFFF6
    expected result (signed):      15 - 25 = -10 overflow: false
    actual result (hexadecimal):   0x000F - 0x0019 = 0xFFF6
    actual result (signed):        15 - 25 = -10 overflow: false

Enter ... a two-operand arithmetic expression... or "quit": 0x100 - 0x7F
UNSIGNED SUBTRACTION
    expected result (hexadecimal): 0x0100 - 0x007F = 0x0081
    expected result (unsigned):    256 - 127 = 129 overflow: false
    actual result (hexadecimal):   0x0100 - 0x007F = 0x0081
    actual result (unsigned):      256 - 127 = 129 overflow: false
SIGNED SUBTRACTION
    expected result (hexadecimal): 0x0100 - 0x007F = 0x0081
    expected result (signed):      256 - 127 = 129 overflow: false
```

[3] A subtle, normally-desirable, rule in the bitwise operator's semantics will cause 1s to be placed in $bits_{31..16}$. For our specific use, this is undesirable and so you need to force $bits_{31..16}$ to have 0s.

```
actual result (hexadecimal):   0x0100 - 0x007F = 0x0081
actual result (signed):        256 - 127 = 129 overflow: false
```

As with addition, if you are performing this lab on the *cse-linux-01.unl.edu* Linux server, then the expected overflow flags are obtained directly from flags set in the processor's ALU.

( ) Check your code with other values, comparing your actual results with the expected results.

- Use positive and negative operands.
- Generate differences that produce signed overflow, differences that produce unsigned overflow, and differences that produce neither.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

5 Signed Inequality Comparison Functions

In general, comparing two signed values to determine which (if either) is greater can be achieved with subtraction. Consider, for example, the less-than comparison. A simple application of algebra tells us that

$$value1 < value2 \Leftrightarrow value1 - value2 < 0$$

Similarly,

$$value1 = value2 \Leftrightarrow value1 - value2 = 0$$

As we noted in Section 3.3, real ALUs dedicate hardware to quickly compare values to 0. This is an example of why that is so. By performing this subtraction and determining the truth values of **`is_zero()`** and **`is_negative()`**, you have sufficient data to determine whether `value1` is **`less_than`**, **`at_most`**, **`at_least`**, and/or **`greater_than`** `value2`.

( ) Implement the four signed inequality functions.

Note: processor instructions using "greater than" and "less than" inequalities generally operate with ***signed*** values, as is the case in this assignment. The corresponding unsigned inequalities tend to go by names such as "above"/"below" (x86) or "higher"/"lower" (ARM).

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

For example:

```
Enter ... a two-operand comparison expression ... or "quit": 4 < 5
expected: (4 < 5) = 1
actual:   (4 < 5) = 1

Enter ... a two-operand comparison expression ... or "quit": 4 >= 5
expected: (4 >= 5) = 0
actual:   (4 >= 5) = 0
```

( ) Check your code with other values, comparing your actual results with the expected results.

- Use positive and negative operands.
- Generate both `true` and `false` results for each of the four signed inequality functions.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

6 Unsigned Multiplication and Division

Before you can perform long multiplication of arbitrary values, you must be able to multiply by a power of two.

6.1 Multiplication by a Power of Two

Strictly speaking, the **`multiply_by_power_of_two()`** function needs to do a little bit more than multiplying by a power of two; it also needs to be able to multiply by zero.

( ) Add code to **`multiply_by_power_of_two()`** so that if the `power_of_two` argument is 0, then the function returns 0.

( ) Add code to **`multiply_by_power_of_two()`** that assumes any other `power_of_two` is in fact a power of two, and apply the fast multiplication technique for powers of two discussed in Chapter 3 and in lecture.

Note that the second argument is the power of two value, such as 0x0040 (64_{10}) or 0x2000 (8192_{10}) and *not* the exponent of two, such as 6 or 13. Be sure to remember that **`multiply_by_power_of_two()`** returns a 32-bit value.

Your solution for `multiply_by_power_of_two()` should be a constant-time solution. If your solution includes a loop or recursion, please review the Chapter 3 material.

Check your work

( ) Compile and run *./integerlab*, trying a few values.

When you enter the inputs for your power-of-two multiplier, the operands will be interpreted as hexadecimal values even if you omit the leading "0x". For example:

```
Enter ... "mul2 <hex_value> <hex_power_of_two>" ...:  mul2 5 4
expected: 0x0005 * 0x0004 = 0x00000014
actual:   0x0005 * 0x0004 = 0x00000014

Enter ... "mul2 <hex_value> <hex_power_of_two>" ...:  mul2 5 0
expected: 0x0005 * 0x0000 = 0x00000000
actual:   0x0005 * 0x0000 = 0x00000000

Enter ... "mul2 <hex_value> <hex_power_of_two>" ...:  mul2 0xFFFF 0x8000
expected: 0xFFFF * 0x8000 = 0x7FFF8000
actual:   0xFFFF * 0x8000 = 0x7FFF8000
```

( ) Check your code with other values, comparing your actual results with the expected results.

6.2 General Unsigned Multiplication

The distributive property of multiplication tells us that if $multiplier = \sum_{i=0}^{31} multiplier_i \times 2^i$ then

$$\begin{aligned} multiplicand \times multiplier &= multiplicand \times \sum_{i=0}^{31} multiplier_i \times 2^i \\ &= \sum_{i=0}^{31} multiplicand \times multiplier_i \times 2^i \end{aligned}$$

In the **`unsigned_multiply()`** function,

( ) Use each of the `multiplier`'s bits, in turn, as the `power_of_two` argument to **`multiply_by_power_of_two()`** to multiply `multiplicand`.[4]

( ) Add each of these intermediate products to arrive at the 32-bit product of *multiplicand*× *multiplier*.

[4]You must implement long multiplication by calling **`multiply_by_power_of_two()`**. Alternate algorithms, such as Russian peasant multiplication, are prohibted. You should not even consider superpolynomial algorithms, such as repeatedly adding the multiplicand to itself *multiplier* times.

When multiplying two 16-bit operands, a real ALU will spread the 32-bit full product across two 16-bit registers. High-level languages will only access the register containing the 16-bit product when assigning the result to the destination variable. Assembly language, however, lets programmers access both registers.

( ) Place the 16-bit product, the lower 16 bits of the full product, in `product`'s `result` field.

( ) Place the upper 16 bits of the full product in `product`'s `supplemental_result` field.

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

(Note that unless and until you implement signed multiplication, your "SIGNED MULTIPLICATION" actual results will differ from the expected results. *You are **not** required to implement signed multiplication.*)

For example:

```
Enter ... a two-operand arithmetic expression... or "quit": 3 * 5
UNSIGNED MULTIPLICATION
    expected result (hexadecimal): 0x0003 * 0x0005 = 0x0000'000F
    expected result (unsigned):    3 * 5 = 15 (15)
    actual result (hexadecimal):   0x0003 * 0x0005 = 0x0000'000F
    actual result (unsigned):      3 * 5 = 15 (15)
SIGNED MULTIPLICATION
    expected result (hexadecimal): 0x0003 * 0x0005 = 0x0000'000F
    expected result (signed):      3 * 5 = 15 (15)
    actual result (hexadecimal):   0x0003 * 0x0005 = 0x0000'0000
    actual result (signed):        3 * 5 = 0 (0)

Enter ... a two-operand arithmetic expression... or "quit": 0x234 * 0x345
UNSIGNED MULTIPLICATION
    expected result (hexadecimal): 0x0234 * 0x0345 = 0x0007'3404
    expected result (unsigned):    564 * 837 = 13316 (472068)
    actual result (hexadecimal):   0x0234 * 0x0345 = 0x0007'3404
    actual result (unsigned):      564 * 837 = 13316 (472068)
SIGNED MULTIPLICATION
    expected result (hexadecimal): 0x0234 * 0x0345 = 0x0007'3404
    expected result (signed):      564 * 837 = 13316 (472068)
    actual result (hexadecimal):   0x0234 * 0x0345 = 0x0000'0000
    actual result (signed):        564 * 837 = 0 (0)
```

If you are performing this lab on the *cse-linux-01.unl.edu* Linux server, then the expected results (including the upper 16 bits) come directly from the two registers used by processor's ALU and are authoritative.

( ) Check your code with other values, comparing your actual results with the expected results.

- Generate products that fit within the lower 16 bits and products that require more.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

6.3 Unsigned Division by a Power of Two

As discussed in Chapter 3 and in lecture, there is a fast division technique when the divisor is a power of two. In the **`unsigned_divide()`** function,

( ) If the divisor is 0 then set the `divide_by_zero` flag.

( ) Otherwise, use that fast technique to implement division by a power of two.

Do not implement general division.

When dividing integers, a real ALU will place the quotient in one register and the remainder in another. When assigning the result to the destination variable, a high-level language will only access the register containing the quotient or the register containing the remainder, depending on whether the program called for division or the modulo operator. Assembly language, however, lets programmers access both registers.

( ) Place the quotient in `quotient`'s `result` field.

( ) Place the remainder in `quotient`'s `supplemental_result` field.

Your solution to determine the quotient should be a constant-time solution. If your solution includes a loop or recursion, please review the Chapter 3 material.

Check your work

( ) Compile and run *`./integerlab`*, trying a few values.

(Note that unless and until you implement signed division, your "SIGNED DIVISION" actual results will differ from the expected results. You are not required to implement signed division.)

For example:

```
Enter ... a two-operand arithmetic expression... or "quit": 70 / 8
UNSIGNED DIVISION
    expected result (hexadecimal): 0x0046 / 0x0008 = 0x0008    0x0046 % 0x0008 = 0x0006
    expected result (unsigned):    70 / 8 = 8    70 % 8 = 6
    actual result (hexadecimal):   0x0046 / 0x0008 = 0x0008    0x0046 % 0x0008 = 0x0006
    actual result (unsigned):      70 / 8 = 8    70 % 8 = 6
```

```
SIGNED DIVISION
    expected result (hexadecimal): 0x0046 / 0x0008 = 0x0008    0x0046 % 0x0008 = 0x0006
    expected result (signed):      70 / 8 = 8    70 % 8 = 6
    actual result (hexadecimal):   0x0046 / 0x0008 = 0x0000    0x0046 % 0x0008 = 0x0000
    actual result (signed):        70 / 8 = 0    70 % 8 = 0

Enter ... a two-operand arithmetic expression... or "quit": 0x29B / 0x40
UNSIGNED DIVISION
    expected result (hexadecimal): 0x029B / 0x0040 = 0x000A    0x029B % 0x0040 = 0x001B
    expected result (unsigned):    667 / 64 = 10    667 % 64 = 27
    actual result (hexadecimal):   0x029B / 0x0040 = 0x000A    0x029B % 0x0040 = 0x001B
    actual result (unsigned):      667 / 64 = 10    667 % 64 = 27
SIGNED DIVISION
    expected result (hexadecimal): 0x029B / 0x0040 = 0x000A    0x029B % 0x0040 = 0x001B
    expected result (signed):      667 / 64 = 10    667 % 64 = 27
    actual result (hexadecimal):   0x029B / 0x0040 = 0x0000    0x029B % 0x0040 = 0x0000
    actual result (signed):        667 / 64 = 0     667 % 64 = 0

Enter ... a two-operand arithmetic expression... or "quit": 53 / 0
UNSIGNED DIVISION
    expected result: divide-by-zero
    actual result:   divide-by-zero
SIGNED DIVISION
    expected result: divide-by-zero
    actual result (hexadecimal):   0x0035 / 0x0000 = 0x0000    0x0035 % 0x0000 = 0x0000
    actual result (signed):        53 / 0 = 0    53 % 0 = 0
```

If you are performing this lab on the *cse-linux-01.unl.edu* Linux server, then the expected results (including the upper 16 bits) come directly from the two registers used by processor's ALU and are authoritative.

( ) Check your code with other values, comparing your actual results with the expected results.

- Remember that the divisor must be either 0 or a power of two.

( ) Run the constraint checker: `python constraint-check.py integerlab.json`

7 Signed Multiplication and Division (Bonus Credit)

You have the opportunity to earn a small amount of bonus credit.

Addition uses the same assembly code instruction for both signed and unsigned integers, as does subtraction. Indeed, these instructions perform exact same actions regardless of whether the integers will be interpreted as signed or unsigned, which is why the overflow conditions for both are flagged.

Multiplication and division, however, have separate instructions for signed and unsigned integers. This is because the logic for unsigned multiplication and division only produce correct results for positive numbers, and so the unsigned implementations cannot be used for negative integers. The signed implementations cannot be used for unsigned integers because they treat half of the possible unsigned integers as though they were negative, yielding incorrect results.

A simple patch would be to keep track of which operands are negative, negate those operands so that they are positive, apply the unsigned implementation, and then negate the result as necessary. *Using that patch technique will not earn you bonus credit.* To earn bonus credit, you must address the underlying reason that the signed implementations need to be different.

7.1 Signed Multiplication

If we only cared about the 16-bit product, the lower 16 bits of the full 32-bit product, then the unsigned implementation works for both signed and unsigned integers. The upper 16 bits, however, differ when **`is_negative()`** is true. For example:

```
Enter ... a two-operand arithmetic expression... or "quit": -3 * 5
UNSIGNED MULTIPLICATION
    expected result (hexadecimal): 0xFFFD * 0x0005 = 0x0004'FFF1
    expected result (unsigned):    65533 * 5 = 65521 (327665)
    ...
SIGNED MULTIPLICATION
    expected result (hexadecimal): 0xFFFD * 0x0005 = 0xFFFF'FFF1
    expected result (signed):      -3 * 5 = -15 (-15)
    ...
```

If you chose to implement signed multiplication then step through your unsigned multiplication to see if you can find where it breaks down for negative operands.

( ) For bonus credit, implement **`signed_multiply()`** to correctly handle negative numbers when the arguments are interpreted as signed integers.

- *Reminder: you may not change the signatures of any functions declared in* alu.h*; however, you may implement other helper functions if you wish.*

( ) Check your work with several values, both great and small.

7.2 Signed Division

Recall that the semantics of integer division are that the fractional portion of the quotient be truncated; that is, the quotient is rounded toward zero. The fast division technique for powers of two, however, has the effect of rounding toward negative infinity. This is fine for positive quotients, but it rounds negative quotients in the wrong direction.

For example, if we used unsigned fast division for signed division then we would see this:

```
Enter ... a two-operand arithmetic expression... or "quit": -14 / 4
...
SIGNED DIVISION
    expected result (hexadecimal): 0xFFF2 / 0x0004 = 0xFFFD    0xFFF2 % 0x0004 = 0xFFFE
    expected result (signed):      -14 / 4 = -3    -14 % 4 = -2
    actual result (hexadecimal):   0xFFF2 / 0x0004 = 0xFFFC    0xFFF2 % 0x0004 = 0x0002
    actual result (signed):        -14 / 4 = -4    -14 % 4 = 2
```

If you chose to implement signed division then in your implementation of **`signed_divide()`**, whenever the dividend is negative you need to introduce a bias toward positive infinity. This bias needs to be sufficient so that when the fast division technique rounds non-integer quotients toward negative infinity, it ends up rounding to the correct quotient – but do so without overcorrecting. The other precaution you need to take is to ensure that when you apply the fast division technique, you preserve the sign bit.

( ) For bonus credit, implement **`signed_division()`** to correctly handle negative dividends.

( ) Check your work with several values, both great and small.

8 Turn-in and Grading

When you have completed this assignment, submit *basetwo.c* and *alu.c* to the assignment in Canvas.

No Credit for Uncompilable Code

If the TA cannot create an executable from your code, then your code will be assumed to have no functionality.[5] Before turning in your code, be sure to compile and test your code on the *cse-linux-01.unl.edu* Linux server with the original driver code, the original header file(s), and the original *Makefile*.

Late Submissions

This assignment is due before the start of your lab section. The due date in Canvas is five minutes after that, which is ample time for you to arrive to lab and then discover that you'd forgotten to turn in your work without Canvas reporting your work as having been turned in late. We will accept late turn-ins up to one hour late, assessing a 10% penalty on these late submissions. Any work turned in more than one hour late will not be graded.

[5] At the TA's discretion, if they can make your code compile with *one* edit (such as introducing a missing semicolon) then they may do so and then assess a 10% penalty on the resulting score. The TA is under no obligation to do so, and you should not rely on the TA's willingness to edit your code for grading. If there are multiple options for a single edit that would make your code compile, there is no guarantee that the TA will select the option that would maximize your score.

Rubric

This assignment is worth 40 points.

_____ **+1** The **`exponentiate()`** function produces the correct powers of two.

_____ **+1** The **`lg()`** function produces the correct base-2 logarithms.

_____ **+1** The **`is_negative()`** function correctly determines whether its argument has a negative value when interpreted as a signed integer.

_____ **+1** The **`equal()`** function correctly determines whether its arguments are equal to each other.

_____ **+1** The **`not_equal()`** function correctly determines whether its arguments are not equal to each other.

_____ **+1** The **`logical_not()`** function correctly produces the logical inverse of its argument.

_____ **+1** The **`logical_and()`** function correctly produces the logical conjunction of its arguments.

_____ **+1** The **`logical_or()`** function correctly produces the logical disjunction of its arguments.

_____ **+1** The **`one_bit_full_addition()`** function correctly determines the `sum` and `c_out` bits for a 1-bit full adder.

_____ **+5** The **`ripple_carry_addition()`** function correctly implements a 32-bit ripple-carry adder.

_____ **+2** The **`add()`** function correctly performs 16-bit integer addition.

_____ **+2** The **`add()`** function correctly detects unsigned integer overflow and signed integer overflow.

_____ **+3** The **`subtract()`** function correctly performs 16-bit integer subtraction.

_____ **+2** The **`subtract()`** function correctly detects unsigned integer overflow and signed integer overflow.

_____ **+1** The **`less_than()`** function correctly determines whether its first argument is strictly less than its second argument.

_____ **+1** The **`at_most()`** function correctly determines whether its first argument is less than or equal to its second argument.

_____ **+1** The **`at_least()`** function correctly determines whether its first argument is greater than or equal to its second argument.

_____ **+1** The **`greater_than()`** function correctly determines whether its first argument is strictly greater than its second argument.

_____ **+3** The **`multiply_by_power_of_two()`** function correctly multiplies its first argument by its second argument when the second argument is 0 or is a power of two.

_____ **+3** The **`unsigned_multiply()`** function correctly provides the 16-bit product when it multiplies its first argument by its second argument when they are interpreted as unsigned integers.

_____ **+2** The **`unsigned_multiply()`** function correctly provides the 32-bit full product spread across the `supplemental_result` and `result` fields.

_____ **+3** The **`unsigned_divide()`** function correctly provides the 16-bit quotient when it divides its first argument by its second argument (or correctly reports division by zero) when they are interpreted as unsigned integers and the second argument is 0 or is a power of two.

_____ **+2** The **`unsigned_divide()`** function correctly provides the 16-bit remainder.

_____ **Bonus +1** The **`signed_multiply()`** function correctly performs signed integer multiplication by addressing the underlying reason that signed and unsigned multiplication need to be different.

_____ **Bonus +1** The **`signed_divide()`** function correctly performs signed integer division by addressing the underlying reason that signed and unsigned division need to be different.

Penalties

_____ **-1** For each of these functions that violates an assignment constraint: **`exponentiate()`**, **`lg()`**, **`is_negative()`**, **`equal()`**, **`not_equal()`**, **`logical_not()`**, **`logical_and()`**, **`logical_or()`**, **`less_than()`**, **`at_most()`**, **`at_least()`**, **`greater_than()`**.

_____ **-15** If **`one_bit_full_addition()`** violates an assignment constraint.

_____ **-14** If **`one_bit_full_addition()`** does not violate an assignment constraint but **`ripple_carry_addition()`** does.

_____ **-4** If **`one_bit_full_addition()`** and **`ripple_carry_addition()`** do not violate an assignment constraint but **`add()`** does.

_____ **-5** If **`one_bit_full_addition()`** and **`ripple_carry_addition()`** do not violate an assignment constraint but **`subtract()`** does.

_____ **-8** If **`multiply_by_power_of_two()`** violates an assignment constraint.

_____ **-5** If **`multiply_by_power_of_two()`** does not violate an assignment constraint but **`unsigned_multiply()`** does.

_____ **-5** If **`unsigned_divide()`** violates an assignment constraint.

Assignment constraint violations by helper functions will be assessed against the required function(s) that they help.

no bonus If **`signed_multiply()`** or **`signed_divide()`** violate an assignment constraint or fail to address the underlying reason that the signed and unsigned implementations need to be different.

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

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 C-licon to see if we can improve the Floating Point Unit (FPU) on our experimental microprocessor.

To be continued...