Lab Exercises 2 – Multiplication, Division and Branching

(50 points)

This lab is in two parts. The first consists of a series of coding exercises to familiarize you with the use of the instructions you are learning. The second contains a series of short answer questions which you must complete for full credit. You will submit files containing job output for the exercises in Part A and the answers to Part B separately. For simplicity, a basic code skeleton has been provided for you containing both storage declarations for you to use where appropriate and in-stream program data.

Before you submit your job output for the coding exercises (which will include the source code), be absolutely certain that you have *thoroughly* reviewed the requirements set forth in the "Coding and Documentation Standards" section on pages 2 & 3 of this document. Failure to follow them will result in substantial loss of credit on this assignment.

Part A: Coding Exercises (30 points)

1. **(5 points).** Pick two registers and load the constant values 10 and 128, then multiply them together. You'll need to pay attention to the even-odd register pair convention and pick your registers accordingly. You may use as many registers as you need. Declare storage for and store the (relevant 32-bit portion of the) result, then XDUMP it (*not the registers!*).
2. **(5 points).** Do the same thing again to multiply -28 by 300, but this time use no more than 2 registers (i.e., only the even-odd pair). Declare storage for and store the (relevant 32-bit portion of the) result, then XDUMP it (*not the registers!*).
3. **(5 points).** Pick three registers (an even-odd pair for the dividend and a third for the divisor), and perform the computation 200 / 15. Remember to properly set up the even-odd register pair for the division operation! Convert and place the quotient and remainders in the output buffer (the provided `OQUOT` and `OREMAIN` fields, respectively), then send the buffered line (`S3PRLINE`) to output.
4. **(5 points).** Write a top-driven loop with a priming read to load all integer values in the input data (refer to the in-stream program data in the provided JCL skeleton) into a sequence of fullwords (i.e., array) in storage. You'll need to write the *single declaration statement* to reserve an appropriate amount of storage. You'll also need to use relative addressing in the (single) `ST` instruction in the loop. `XDUMP` the storage for the entire sequence.
5. **(5 points).** Write a bottom-driven loop to accumulate (add together) the values you stored in the sequence from Step 4. Use the address of the current location in the sequence to decide when to stop so that your program doesn't go past the end of the array! Declare storage for and store the result, then `XDUMP` it (*not the registers!*).
6. **(5 points).** Divide the accumulated value from Step 5 by 2 *using a literal.* If the result is **even**, send the message "The accumulated result is even!" to job output, otherwise if it is **odd** send the message "The accumulated result is odd!" instead. Be sure to follow the proper conventions and declare your print lines to contain 133 bytes, making sure that the first is the carriage control character that generates *double-spacing.*

Part B: Short Answer (20 points)

1. **(5 points).** Describe a method to compute powers of 2 using only `LA` (to set an initial value) and `MR`, and only the required even-odd register pair.
2. **(5 points).** Suggest a way to determine if the divisor used in a modulus computation is negative. Your method may only inspect the result, and you must describe the logic used in the determination (e.g., comparison) itself.
3. **(10 points).** Write a code skeleton implementing the following pseudocode:

```
if (x != y) then
    ...
else if (x < y) then
    ...
else
    ...
```

   Make sure your code skeleton includes the instructions to set up for the comparisons, as shown in the class slides. Your code skeleton does not need to include any JCL or the assembler instructions to establish or end a control section, or return to the caller.

Coding and Documentation Standards

Assuring that your code follows proper coding conventions and documentation standards is a critical habit in the workplace, and can make the difference between keeping or losing your job. Your work will therefore be graded, in part, on your ability to adhere to the following coding and documentation standards:

Coding standards

1. **Available registers.** You may only use registers 2 through 11 (inclusive) for loading values, addresses and performing math operations. Registers 0–1 and 12–15 are reserved for special use.
2. **Loading constant values.** Unless otherwise instructed, when you are told to load a *constant* integer value into a register you must use the `LA` instruction. You may *not* declare the value in program storage and then load it with `L`.
3. **Creating arrays.** When told to declare an array to store a sequence of values in program storage, you must declare it using a *single* `DS` or `DC` instruction prefixed with a label.
4. **Accessing arrays.** For this assignment, when you are told to access elements (locations) in the array for loading or storing values, you must first load the array's base address into a register using `LA` and then use that register with an appropriate displacement in a relative address as the second operand of your `L` or `ST` instruction.

Documentation standards

1. **Documentation box.** Your program code must contain a standard documentation box (or docbox) immediately following the opening JCL statements:

```
*****************************************************************
* CSCI 360-2           LAB EXERCISE n              SPRING 2022 *
*                                                               *
* NAME: (Your name)                                             *
* DATE: (Date of submission)                                    *
*                                                               *
* (Brief Description)                                           *
*                                                               *
* REGISTER USAGE:                                               *
*   Ra - (Usage description)                                    *
*   Rb - (Usage description)                                    *
*   ...                                                         *
*                                                               *
*****************************************************************
```

   n – Number of the current lab exercise (i.e., 1)
   a – Lowest register number
   b – Next register number

2. **Inline documentation.** Roughly 80% of your lines of code should have comments separated from and following the operands of the Assembler instructions.

If your submitted code violates *any* of these standards, then you will automatically lose 50% of the points ***you would have received***, ***AFTER*** your program's functionality and output are assessed.