1 PA2

In this assignment, you will write a few more programs in ARM assembly language by using the branching and control-flow instructions that you have learned about in class.

2 Division/Modulo

The ARMv7 architecture does not have any instructions for the quotient or remainder of two numbers, so we'll have you implement it instead. Please complete the file `divmod.s` with a simple ARM assembly implementation of integer division. There will be two expected inputs: the dividend in register `r0` and the divisor in register `r1`. The expected output will also have two values: the quotient in register `r0` and the remainder of the two numbers in register `r1`. A sample output would look like the following:

```
$ pa2-runner divmod.bin divmod 7 3
7 / 3 = 2
7 % 3 = 1
```

For this assignment, you do not need to worry about performance or division by zero.

You may test your solution with two steps. First, we've provided a command for you to turn your assembly into binary (it uses `as`, the built-in assembler):

```
$ make divmod.bin
```

This will create a file called `divmod.bin`, or report any errors the assembler found while reading the file `divmod.s`.

Then you can run your code by using the provided runner executable:

```
$ pa2-runner build/divmod.bin divmod 7 3
```

You *cannot* use the instructions `sdiv` or `udiv` in this part, or any other part, of the assignment.

3 A Simple Cipher

In this part of the exercise, you will implement a simple cipher that encrypts and decrypts ASCII messages using a 16-bit key. Your cipher will operate on one character at a time; after encrypting/decrypting a single character, it will update the key used for the rest of the message.

3.1 Encryption

In `encrypt.s`, you'll be writing the code to encrypt the character. There will be two inputs for this algorithm: the character that will be encrypted and the key used for encryption, which you can assume to be in registers `r0` and `r1` respectively. After your program runs, the encrypted character should be in `r0` and the updated key should be in `r1`.

First, the character needs to be encrypted, following the steps below:

1. Save the original input character in a register for later use, as you'll need it when generating the new key.
2. Inspect the most significant bit of the key. For big-endian, the most significant bit would be the rightmost bit, as it reads from higher memory to lower memory, but it's the opposide for little-endian; its most significant bit in the leftmost bit, as it reads from lower memory to higher memory. Because ARM uses little-endian, the most significant bit of the key would be the leftmost one.
3. The process to encrypt the character may cause some odd ASCII characters to appear, so you'll have to do some checks to prevent that from happening using some constants we'll provide for youe. First, AND the character with $\#0x1F$ and check if the resulting value is $\#0x7f$. If it is, replace the value with $\#0x0A$.
4. EOR the input character with either the upper byte of the key (when the MSB is 1) or the lower byte in order to get the newly encrypted character.

After encrypting the character, the key needs to be updated using the following steps:

1. The new key will use an integer n such that the encrypted character modulo n is equal to 5. In other words, you'll need to find a value for n that will get a remainder of 5 when dividing the character.

   Note that, if the newly encrypted character has a value less than 11, there is no n value that satisfies the condition; in this case, use the encrypted character as the n value.
2. Multiply the old key by 4.
3. Add n.
4. Divide by 2.
5. Subtract the original input character, which you saved at the start.
6. Shift the register logically right until there are only two bytes left (the upper half word is zero).

7. Reverse the value of the most significant bit. (If it was 1, it becomes 0, and vice versa.)

At this point, you'll have the encrypted character in `r0`, and the updated key in `r1`. To test your encryption, build with:

```
$ make encrypt.bin

$ pa2-runner encrypt.bin encrypt <input-file> <output-file>
```

You should not write `<input-file>` and `<output-file>` exactly, but instead use real filenames. We recommend making some simple text files containing a few characters, and trying on those. For example, you might open a file named `hello.txt`, save the contents `Hello!` into it, and then run:

```
$ pa2-runner encrypt.bin encrypt hello.txt hello-encrypted.txt
```

After running the command, the runner will place the encrypted text in the specified file. Open the file to check if your output is as expected. Below are some examples of inputs and expected output which you can test your code against:

- insert examples -

3.2 Decryption

You should also write a decryption algorithm that inverts the process in the file `decrypt.s`. Be sure to also reverse the check against $\#0x7F$ from the encryption algorithm in order to return the character to its original state. The following commands will build and test decryption:

```
$ make encrypt.bin

$ pa2-runner encrypt.bin encrypt <input-file> <output-file>
```

If your encryption algorithm works as expected, you can test the decryption by using the encrypted file as input to see if you get the original text back.

4 README

In addition to your code, please include a README file which contains your answers to the following questions:

1. The encryption and decryption algorithms are similar, though parts are reversed. Identify one common piece of code across your implementations, and one piece that's different. Describe in a sentence or two each why the common piece can be the same and why the different pieces cannot be the same.

5 Commenting and Style Guide

Every section of code, loop, or branch command needs to be commented, although you are free to include other comments. Lines of code should not exceed 80 characters.

On including your name in files: To detect instances of academic integrity violations in programming assignments we may use 3rd party software. We recommend you only include your class lab account ID (not your name or PID) in your submissions. Including your name and/or PID will disclose that information to the 3rd party.

6 Handin

Commit and push the four files to the Github repository that was created for you by 11:59PM on Tuesday, October 17. You can push up to one day late for a 20% penalty. After you push, make sure to check on Github that the files are actually there; we will mark all of the repositories for grading a few minutes after midnight and grade precisely what is there.

Your handin should include:

- Two assembly files for the cipher process, (`encrypt.s`, `decrypt.s`), and one assembly file for the divison/module function, (`divmod.s`);
- A single README.txt file that has answers to the open-ended questions above, as specified.

Note that you do not need to hand in the `.bin` files.