1 PA2: Encrypt and Decrypt with Loops and Bitwise Operations

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

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, division by zero, or negative numbers.

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

Encryption is about making messages indecipherable without the use of a key. In this assignment we'll implement a simple encryption scheme for text files. You'll implement both the encryption side, which takes a key and a text file and produces a file full of gibberish, and the decryption side, which takes a file generated by the encryption and uses the same key to produce the original text again.

NOTE: This is a bad encryption algorithm from a security point of view. You shouldn't actually use it to keep things secret. In general, you shouldn't write your own cryptography and put it in a product; there are well-known algorithms and libraries for encryption. But, for exploring some uses of bit-twiddling operations, this simple cipher is quite fun!

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.

ASCII is a representation of characters as numbers. You can find plenty of ASCII tables on the web with a quick search that show the value for each letter. ASCII is quite Anglo-centric, since it focuses on the Latin alphabet, and has stayed around for a while because it was an encoding used in early systems. The input to your encryption algorithm will come as single-byte ASCII characters, and you can write text files containing ASCII for encryption. We use ASCII to keep the input space small for this sample program while still working for interesting inputs; a more robust example would scale to an encoding like UTF-8.

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`.

The key is 16 bits (two bytes) long, and is stored in bits 0-15 of `r1`.

The character is 8 bits (one byte) long, and is stored in bits 0-7 of `r0`.

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 – the one that corresponds to the 1 in `0x00008000`.

3. If that bit is 1, use the upper byte of the key for the next step, if it is 0, use the lower byte.

4. AND the key with `0x1F`.

5. EOR the input character with the key that results from this AND. This is the encrypted character to return in `r0`. We'll also refer to this encrypted character in the key update below.

For example, let's use `0x42` as the input character, and `0x4849` as the input key. In binary, the key would be `0100 1000 0100 1001`. The leftmost bit is a 0, so you would use the right byte of the key. First, you'll need to AND the right half with `0x1F`:

original:	0 1 0 0	1 0 0 1
0x1F:	0 0 0 1	1 1 1 1
AND result:	0 0 0 0	1 0 0 1

Although the original is still unchanged, the half key we are using now holds the value `0x09`. Notice how, by using `0x1F`, the left half byte has changed to have a value of `0x0`, and the right half byte is unchanged. Next, EOR the new key with the input character, `0x42`.

key:	0 0 0 0	1 0 0 1
character:	0 1 0 0	0 0 1 0
EOR result:	0 1 0 0	1 0 1 1

You would end up with `0x4B` as your 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 of the key (that is, bit 15, the highest bit in the 2-byte key). By reverse we mean if it was 1, it becomes 0, and vice versa.

Continuing from the previous example, we have an encrypted character `0x4B` which originally held the value of `0x42`, and a key `0x4849`. So, in order to update the key, it would go through this process:

```
0x4849 * 4 = 74020
74020 + 10 = 74030
74030 / 2 = 37015
37015 - 0x42 (66) = 36949
```

```
36949 converted to hexadecimal is 0x9055
0x9055 in binary is 1001 0000 0101 0101
As it only takes up two bytes, no shift is required
Flipping the MSB gives 0x1055
```

After this completes, the n value would be `0xA`, and the new key would be `0x1055`.

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 <key> <input-file> <output-file>
```

Where `<key>` is any two ASCII characters.

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 ke hello.txt hello-encrypted.txt
```

After running the command, the runner will place the encrypted text in the specified file. If you open the output file in an editor, it will most likely look like gibberish! These are the characters that were produced in `r0` by your algorithm, in order. You can inspect them with `xxd` if you want to look at the hex values that are stored in the file. This can be useful if you want to write a very small two-or-three character test that you write out by hand first, and know the expected hex values for.

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 <key> <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:

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.

Keep your answer to under 200 words total.

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.