CSE 30 Fall 2022 Programming Assignment #8 (Vers 1.3a)
Due Tuesday November 22 , 2022 @ 11:59PM

Remember: 'Start Early'

Grading (50 points total) What You Need To Do

You must complete all of the following items by the due date to get all points.
The last day to turn in is Friday Nov 25, 2022 at 11:59 PM.

There are **50 points available** for this PA that are distributed as follows:

- ☐ Up to 5 points for following the C (1 point) and Arm (4 points) style guidelines. https://docs.google.com/document/d/1jItOTaolKNfF7QTnCnJQ9KxgkcllDJ_rD86u7nHnHZ4/edit?usp=sharing https://docs.google.com/document/d/1dJlBwofIcSfhEhvIFPfPOS7tYDb3gxEP2vfs_CeZ8YI/edit?usp=share_link
- ☐ Up to 10 points (at our discretion) if the following files compile/assemble without warnings. This has to be a meaningful attempt to write the code that meets the specifications of the program. Random C, ARM assembly statements or empty files will not be awarded any points. **These are the only files that are part of PA8:**

```
Cencrypt.c (2 points)
Cdecrypt.c (2 points)
encrypt.S (3 points)
decrypt.S (3 points)
```

- ☐ Up to 20 points for passing the tests in the included test harness
- ☐ Up to 10 points for passing the gradescope tests (these will be run after the late deadline).
- ☐ Up to 3 points canvas checkpoint quiz (Available Tues Nov 15, 11:59 PM, due Saturday 11/19 11:59 PM). No slip days or late submissions.
- ☐ **Take the ungraded CSE30 Post Assessment quiz** Given during the week8 (F)/week 9(M) discussion (you can attend any discussion section you want to complete this item) **+2 points if taken**

Assignment – Stack Frames, Bitwise, and Passing Arguments

PA8 is the first part of a two part project where you will be working on a program called ***cipher***. ***Cipher*** is a file encryption/decryption program that uses a variation of what is called a book cipher. ***Cipher*** will both **encrypt** and **decrypt** **any type of file** **(text and binary)** using a ***combination*** of **per byte bit manipulation** (using bitwise operations) and an **exclusive or** (EOR in arm ^ in C) encoding using sequence of **bytes from a book file** (traditionally a real book, but it can be any text or binary file).

cipher reads the file to be processed from standard input, either encrypts or decrypts the file, and then writes the result to standard output. Cipher has two required arguments:

1. A flag that specifies on of ***either*** encrypt (**-e**) or decrypt (**-d**) operational **mode**
2. A flag **-b *<filename>*** that specifies the name of the book file that contains the encoding keys

`Synopsis: ./cipher (-d|-e) -b <bookfile>`

Flag	Description
`-d`	Sets the program to decrypt. **Exactly 1 of -d OR -e must be provided, but not both.**
`-e`	Sets the program to encrypt. **Exactly 1 of -d OR -e must be provided, but not both.**
`-b` *`bookfile`*	The file path to the input bookfile. This is required.

Error Messages and Return from main()

All error messages are output in main() (In the below, argv0 is argv[0])

When the write to stdout fails:

```
printf(stderr, "%s: write failed\n", argv0);
```

When either the read of stdin or bookfile fails or the bookfile is too short in length:

```
fprintf(stderr, "%s: read failed\n", argv0);
```

Command line options are handled for you by setup() (in setup.c). Any errors detected by setup are sent to stderr and setup() returns EXIT_FAIL (see cipher.h). **No other errors need to be detected.**

`main() returns EXIT_SUCCESS` when the program completes successfully (no read or write I/O errors), and `EXIT_FAILURE otherwise.`

How the program works

The program is called with a decrypt or encrypt flag, a bookfile flag and a file path for a bookfile. Command line option handling is done for you in the supplied function setup() (written in C in the file setup.c).

Inputs:

- Decrypt flag OR encrypt flag
- Bookfile flag and file path of the bookfile
- Standard input (usually redirected as

```
./cipher -e -b in/BOOK < input_file >output_file
```

Outputs:

- When encrypting, the output goes to **`stdout`**
- When decrypting, the output goes to **`stdout`**
- Options handling and the opening of the bookfile is handled for you in setup(). If setup fails (returns EXIT_FAIL as defined in cipher.h), cipher exits with EXIT_FAILURE.

Obtaining the Key

If you do not know what a key in cryptography is (specifically, symmetric cryptography), it is essentially similar to how a physical key and a lock works. The key is used to close the lock (or encrypt a message), and the same key is required to unlock the lock (or decrypt the encrypted message).

In practice, this almost always uses the exclusive-or operation, or XOR. This is because XOR has the wonderful identity and self-inverse properties, meaning that $A \oplus 0 = A$ ($\oplus$ is the XOR symbol), and $A \oplus A = 0$.

Thus, if we have the message M and we XOR it with key K, we can XOR the key again to reobtain M.

Example: $M \oplus K \oplus K = M \oplus (K \oplus K) = M \oplus (0) = M$

The bookfile is simply a file to obtain keys from. This is based on book ciphers, in which the plaintext of a book is used as a key to a cipher. This is more convenient than carrying around specific keys, as books are public and easily accessible. In the starter code, the bookfile is just a plaintext file of The Adventures of Sherlock Holmes by Arthur Conan Doyle. However a book file does not have to be a text file, as any file can be used as long as it has the same or more bytes (same length or longer) in it than the input file.

How the program obtains keys from this bookfile is reading bytes from the book file. The first key is the very first character of the input, which in the case of the starter code is `'T'` (the first line of the file being `"The Adventures of Sherlock Holmes"`). The first key is used to encrypt the first byte of the input file (read from standard input). However, the next time we obtain a key, we will increment the location by one byte, meaning that the next key would be `'h'`. This second byte of the book file, the 'h' is used to encrypt the second byte of the input file and so on until we reach EOF on the input file. The only requirement for the bookfile is that the number of bytes in the bookfile is equal to or greater than the number of bytes in the input file. If EOF is reached on the book file before reaching EOF on the input file, this is an error and the program terminates with EXIT_FAILURE. In the starter code, all the I/O and filling of the two buffers is handled by rdbuf() (see the description of rdbuf.c below).

Encryption Algorithm

There are two main steps to the encryption algorithm: reversing the bits for each in**put file one byte at a time**, then XORing this with a one byte key from the book file. Remember that the encryption algorithm is performed one byte at a time (byte is 8 bits or the sizeof of a char) in a buffer of bytes (char).

Step 1: Reversing **the bits (bit order) in one byte**

Let's begin with the example of encrypting a byte containing the hexadecimal value of 0x61. In binary it is the following (6 = 0110, 1 = 0001):

0	1	1	0	0	0	0	1

The first step of the algorithm is to reverse the order of the bits in the byte. After this procedure, the result will be the following:

1	0	0	0	0	1	1	0

This process can be done any number of ways using shifts and bitwise operations. However, one thing to note is that at this point in the process, the encryption would result in the hexadecimal value of 0x86 (1000 0110). This approach will work with any byte value (ASCII text or binary).

Step 2: XORing the Key

Once we have performed the first step, a single bitwise operation needs to be done on the byte. Using a single byte key (how to obtain the key is described above), the next step of the algorithm is to exclusive-or (XOR) this **single byte key** with the **single byte result** from the first step.

For example, let's use the letter `'T'` as the key. In ASCII, `'T'` has the hexadecimal value of 0x54. This means in binary it is the following (5 = 0101, 4 = 0100):

0	1	0	1	0	1	0	0

XOR 0101 0100 (the key) with 1000 0110 (the input), and obtain the final result:

1	1	0	1	0	0	1	0

This gives hexadecimal 0xd2 (1101 0010) as the output byte.

Decryption Algorithm

The decryption algorithm is the exact inverse of the encryption process.

Step 1: XORing the Key
Step 2: Reversing the bits of a byte

Starter Code Overview

The starter files are obtained from github. These files will be used for both PA8 and PA9.

```
cs30fa22@pi-cluster-153:~ % git clone https://github.com/cse30-fa22/PA8_starter.git PA8
cs30fa22@pi-cluster-153:~ % cd PA8
cs30fa22@pi-cluster-153:~ % chmod 0755 runtest in/cmd*
```

In the starter code you are supplied the following

File	Function name	Description
Cmain.c	main	The main() function is written in C. The source is complete and you do not need to edit this file. This allocates space for local variables and calls the function setup(), rdbuf(), fwrite(), encrypt() decrypt(), fclose() and fprintf()
main.S	main	**This is part of PA9.** The main function version you will write in assembly based on the C version above. Starter code in assembly is provided. You have to finish the setup of the stack frame design for local variables (matching the supplied C version layout exactly), call the functions setup(), rdbuf(), encrypt(), decrypt(), fwrite(), fprintf(), and fclose(), and implement the code as shown in the provided C version in assembly. Follow the comments in the starter file to help you write the code.
setup.c	setup	The setup() function is written in C. The source is complete and you do not need to edit this file or create an assembly language version of it. setup() handles the command line options and opens the bookfile. It returns 0 on success and EXIT_FAIL on any error. When EXIT_FAIL is returned from setup(), cipher exits with an EXIT_FAILURE.
Crdbuf.c	rdbuf	The C version of the function rdbuf(). The source is complete and you do not need to edit this file. **You will write the assembly language version of this file in PA9.** rdbuf() fills two buffers using the stdio function fread(): iobuf with bytes

		from the input and the same number of bytes into the buffer bookbuf from the book file. See man 3 fread for more details on fread(). On entry, rdbuf() first checks for EOF state condition on the input, FILE *in using eof(in). If it is at EOF, it returns 0 signaling to main() that the end of the input has been reached. It next checks for an IO error condition on FILE *in using ferror(in). If there is an error, it returns EXIT_FAIL. Then rdbuf reads up to a block of bytes (BUFSZ as defined in main.h) from FILE *in into the array iobuf with a single call to fread(). fread() returns the number of bytes actually read (which is <= BUFSZ, see main.h). When fread() returns 0, the end of file has been reached and it returns 0 to main(). Next rdbuf() uses fread() on the FILE *book to read exactly the same number of bytes into the bookbuf. For example if the fread() returns 100 bytes from the input file, this function using fread() then reads exactly 100 bytes from the bookfile. If rdbuf() does not fill both buffers with the same number of bytes it exits with EXIT_FAIL. Otherwise it returns the number of bytes read.
rdbuf.S	rdbuf	**This is part of PA9.** The rdbuf() function version is written in assembly based on the C version above. Starter code in assembly is provided. You have to implement the code as shown in the C version in assembly in PA9.
Cdecrypt.c	decrypt	**This is part of PA8.** The decrypt function is written in C by you. Decrypt is passed two buffers that contain the same number of bytes. It decrypts the bytes in iobuf one byte at a time until all the bytes are processed. Each byte is processed (**one byte** at a time) first by performing an bitwise exclusive or (^ in C) with the corresponding byte (same index offset) in the second buffer bookbuf. Then it reverses the order of the bits in the **one byte**. Finally it stores the result of the bit reversal of the one byte back into iobuf, overwriting the original encrypted version of the byte. Make sure that you use **unsigned int variables when processing the single bytes** as it will be similar to using registers in the assembly language version.

decrypt.S	decrypt	**This is part of PA8.** The decrypt function is written in assembly by you that is based on the C version you wrote in Cdecrypt.c. The function setup and return is provided for you already, you just write the main body of the function. The comments in the starter file describe the parameters passed and which registers you can freely use. You will not need to make any function calls as this routine is not that complicated.
Cencrypt.c	encrypt	**This is part of PA8.** The encrypt() function is written in C by you. Encrypt() is passed two buffers that contain the same number of bytes. It encrypts the bytes in iobuf one byte at a time until all the bytes are processed. First it reverses the order of the bits in the **one byte.** Then it performs a bitwise exclusive or (^ in C) with the corresponding byte (same index offset) in the second buffer bookbuf. Finally it stores the **one byte result** of the bitwise exclusive or back into iobuf, overwriting the original encrypted version of that one byte. Make sure that you use **unsigned int variables when processing the single bytes** as it will be similar to using registers in the assembly language version.
encrypt.S	encrypt	**This is part of PA8.** The encrypt function is written in assembly by you that is based on the C version you wrote in Cencrypt.c. The function setup and return is provided for you already, you just write the main body of the function. The comments in the starter file describe the parameters passed and which registers you can freely use. You will not need to make any function calls as this routine is not that complicated.
libpa8.a	Encrypt & decrypt	Contains working object code versions of the functions: encrypt and decrypt, for you to use until you get your C and then assembly language versions written. The file SELVERS.h is used to select which version to use, the C, assembly or the library version.
Makefile	Compilation and assembly control	Same Makefile format as previous PA's/ alltest: runs test1-test7
runtest	Test harness execution	Same test harness as previous PA's.
cipher.h	Function prototypes and	Function prototypes for the routines encrypt() and decrypt() in C and .externs for assembly. Also some #defines for the operational mode (encrypt or decrypt) and return values.

	common definitions	
main.h	Function prototypes and common definitions	Macros for BUFSZ and defines for EXIT_FAILURE and EXIT_SUCCESS for use by the assembly language version of main()
setup.h	Function prototypes and common definitions	Function prototypes for the routine setup() in C and .extern for assembly.
SELVERS.h	Control of compilation	Selects which versions of the files (C, assembly or library) are used when compiling cipher

Modifying SELVERS.h to Control Compiling & Assembling

In the starter file there is a file called SELVERS.h, which has four sections of control, labeled (A) through (D) below.

(A) In the section below, select whether you want to use the C version, the assembly version or the library solution version for the function decrypt.

```
// at most one of the following two should be uncommented
// if both are commented out, use the solution code
#define MYDECRYPT_C       // when defined will use your decrypt.c for PA8
//#define MYDECRYPT_S       // when defined will use your decrypt.S for PA8
```

(B) In the section below, select whether you want to use the C version, the assembly version or the library solution version for the function encrypt.

```
// at most one of the following two should be uncommented
// if both are commented out, use the solution code
#define MYENCRYPT_C       // when defined will use your encrypt.c for PA8
//#define MYENCRYPT_S       // when defined will use your encrypt.S for PA8
```

(C) In the section below, select whether you want to use the C version (for PA8), the assembly version (for PA9) for main().

```
// only one of the following two must be uncommented
```

```
#define MAIN_C              // Must be uncommented to use Cmain.c for PA8
//#define MYMAIN_S            // when defined will use your main.S for PA9
```

(D) In the section below, select whether you want to use the C version (for PA8), the assembly version (for PA9) for rdbuf().

```
// only one of the following two must be uncommented
#define RDBUF_C             // Must be uncommented to use Crdbuf.c for PA8
//#define MYRDBUF_S           // when defined will use your rdbuf.S for PA9
```

Other code in SELVERS.h (not shown above) will cause an compilation/assembly error if the rules described above are not met. Be careful when editing this file as described above.

Compiling and assembling the starter code

```
 % make
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o Cmain.o Cmain.c
Cmain.c:11:1: note: ‘#pragma message: WARNING - "using Cmain.c"’
   11 | TODO("using Cmain.c");
      | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o Cdecrypt.o Cdecrypt.c
Cdecrypt.c:20:1: note: ‘#pragma message: WARNING - "using solution decrypt"’
   20 | TODO("using solution decrypt");
      | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o Cencrypt.o Cencrypt.c
Cencrypt.c:20:1: note: ‘#pragma message: WARNING - "using solution encrypt"’
   20 | TODO("using solution encrypt");
      | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o Crdbuf.o Crdbuf.c
Crdbuf.c:8:1: note: ‘#pragma message: WARNING - "using Crdbuf.c"’
    8 | TODO("using Crdbuf.c");
      | ^~~~
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o setup.o setup.c
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o main.o main.S
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o decrypt.o decrypt.S
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o encrypt.o encrypt.S
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h   -c -o rdbuf.o rdbuf.S
gcc -I. -ggdb -Wall -Wextra -Werror   -include SELVERS.h Cmain.o Cdecrypt.o Cencrypt.o
Crdbuf.o setup.o main.o decrypt.o encrypt.o rdbuf.o -L ./libpa8.a -o cipher
```

In the output above you can see what a typical compile/assembly output looks like with the starter code just cloned from github. The TODO warnings tell you which versions of the files are being used

based on the setting in SELVERS.h. These are not errors. Observe these messages carefully as you edit SELVERS.h to make sure that you are using the versions of the files that you expect. The TODO warnings do not work in the assembly files as that pragma is for C only (so you will get messages from the C versions even when they are not being used). Gcc is used to assemble the ARM 32 .S files. Gcc calls cpp to expand the #include files in the assembly source files (note the condition checks for assembly in several of the .h files), then calls gas (the GNU assembler) to assemble the file.

Running the Tests - What Passing the tests Looks like

```
$ make alltest

****** starting test1 *******
./runtest 1
----- Starting test number 1 -----
Running in/cmd1 < in/test1 > out/out1 2> out/err1
cmd1 is: ./cipher -e -b in/BOOK
./cipher returned EXIT_SUCCESS
    Comparing exp/out1 to out/out1
**** Standard out   on test number 1 passed ****
    Comparing exp/err1 to out/err1
**** Standard error on test number 1 passed ****
----- Ending   test number 1 -----
******* All Done *******

****** starting test2 *******
./runtest 2
----- Starting test number 2 -----
Running in/cmd2 < in/test2 > out/out2 2> out/err2
cmd2 is: ./cipher -d -b in/BOOK
./cipher returned EXIT_SUCCESS
    Comparing exp/out2 to out/out2
**** Standard out   on test number 2 passed ****
    Comparing exp/err2 to out/err2
**** Standard error on test number 2 passed ****
----- Ending   test number 2 -----
******* All Done *******

****** starting test3 *******
./runtest 3
----- Starting test number 3 -----
Running in/cmd3 < in/test3 > out/out3 2> out/err3
```

```
cmd3 is: ./cipher -e -b in/BOOK
./cipher returned EXIT_SUCCESS
    Comparing exp/out3 to out/out3
**** Standard out   on test number 3 passed ****
    Comparing exp/err3 to out/err3
**** Standard error on test number 3 passed ****
----- Ending   test number 3 -----
******* All Done *******

****** starting test4 *******
./runtest 4
----- Starting test number 4 -----
Running in/cmd4 < in/test4 > out/out4 2> out/err4
cmd4 is: ./cipher -d -b in/BOOK
./cipher returned EXIT_SUCCESS
    Comparing exp/out4 to out/out4
**** Standard out   on test number 4 passed ****
    Comparing exp/err4 to out/err4
**** Standard error on test number 4 passed ****
----- Ending   test number 4 -----
******* All Done *******

****** starting test5 *******
./runtest 5
----- Starting test number 5 -----
Running in/cmd5 < in/test5 > out/out5 2> out/err5
cmd5 is: ./cipher -e -b in/BOOK
./cipher returned EXIT_SUCCESS
    Comparing exp/out5 to out/out5
**** Standard out   on test number 5 passed ****
    Comparing exp/err5 to out/err5
**** Standard error on test number 5 passed ****
----- Ending   test number 5 -----
******* All Done *******

****** starting test6 *******
./runtest 6
----- Starting test number 6 -----
Running in/cmd6 < in/test6 > out/out6 2> out/err6
cmd6 is: ./cipher -d -b in/BOOK
./cipher returned EXIT_SUCCESS
    Comparing exp/out6 to out/out6
**** Standard out   on test number 6 passed ****
    Comparing exp/err6 to out/err6
```

```
**** Standard error on test number 6 passed ****
----- Ending   test number 6 -----
******* All Done *******

****** starting test7 *******
./runtest 7
----- Starting test number 7 -----
Running in/cmd7 < in/test7 > out/out7 2> out/err7
cmd7 is: ./cipher -e -b in/SHORTBOOK
./cipher returned EXIT_FAILURE
    Comparing exp/out7 to out/out7
**** Standard out   on test number 7 passed ****
    Comparing exp/err7 to out/err7
**** Standard error on test number 7 passed ****
----- Ending   test number 7 -----
******* All Done *******
```

Since the output of the encryption tests (odd tests: test1, test3, test5, test7) is not readable text and the input to the decryption tests (even tests: test2, test4, test6) are not reliable tests, the files are only compared on a byte to byte comparison. You will have to use tools like hexdump to examine the encrypted files.

You should edit SELVERS.h to test your four routines with the test fixture to make sure everything works properly. For example, here are some (but not all) combinations you might want to try:

1. Your Cencrypt.c
2. Your Cdecrypt.c
3. Your decrypt.S
4. Your encrypt.S
5. Your decrypt.S & encrypt.S

You can run the program back to back with a linux shell pipe. So if the file in/test1 contains:

```
this is a test
```

The sequence of back-to-back encrypt and decrypt produces the clear text contents of in/test1. The following will take the stdout output from cat and pipe it into cipher with the -e flag, and pipe that stdout to cipher with the -d flag, resulting in the original text, unchanged.

```
% cat in/test1 | ./cipher -e -b in/BOOK | ./cipher -d -b in/BOOK
this is a test
```

Writing the replacement routines

1. **Write ONLY the following replacement functions for PA8. It is suggested you write the C versions of encrypt() and decrypt() first.**

```
encrypt          in file: Cencrypt.c
decrypt          in file: Cdecrypt.c
encrypt          in file: encrypt.S  // arm32 assembly
decrypt          in file: decrypt.S  // arm32 assembly
```

2. **These (and SELVERS.h) are the only files that you should modify for PA8**
3. Make sure that you remove the comment for the correct versions you want to test in SELVERS.h

Please examine Cmain.c and read how the program works.

When you are designing your assembly language functions, once you have settled on an algorithm, your next task is to decide which registers are going to be used for what purpose. On entry to a function, r0-r3 are used to pass the first four arguments. It is ok to modify these as you see fit in your code. However, if you are going to make a call to another function (PA 9 only), remember that the called function has the right to change these. So if you will still need the parameters after making a function call, either copy them to a protected register or save them on the stack (this is a step you will only take in PA9, you do not have to do this in PA8).

When designing your assembly functions, decide which registers you are going to use for each local variable first. If the function you are working on does not call any function that has output parameters, you can put all the variables in registers (combination of preserved and scratch registers). Try to use the scratch registers first, remembering that function calls that you make (PA9 only) can change the values in r0-r3.

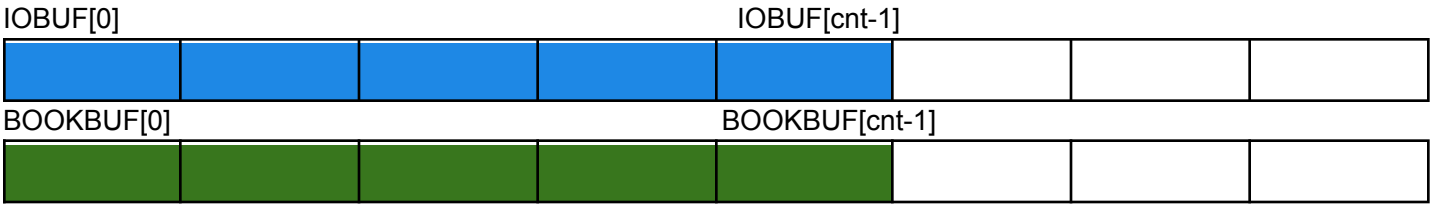
One way to keep track of the registers is to create a table either in your code inside a comment at the top of the function or elsewhere (like a sheet of paper) that describes how you have decided to use each register relative to your algorithm. Example:

```
// register use
// r0 on entry contains the address of iobuf (char * pointer)
// r1 on entry contains the address of bookbuf (char * pointer)
// r2 on entry contains cnt
// r3 is loop counter i
// r4 contains the shifted byte
```

Carefully look at how return values are used from the functions in the C code. Make sure you return the same values in your assembly language equivalent programs.

int encrypt(char *iobuf, char *bookbuf, int cnt) - C version
encrypt() is passed pointers to two buffers, the iobuf which contains the bytes to be encrypted and bookbuf the cipher key buffer. Each buffer contains the exact same number of bytes (cnt) in them.

You will need an outer loop to process each byte in the buffers until all cnt bytes have been processed. Starting with byte 0 in iobuf, write the C instructions to load a byte into an unsigned int local variable. There are many ways to reverse the bit order of the lower 8-bits (the byte) of this unsigned int variable into another unsigned int local variable. Typical approaches include using an inner loop to reverse the bits one at a time, or a sequence of bit-wise operations with shifts.



Copy the book buffer byte into an unsigned int variable. Finally you do an exclusive or ^ of the reversed bit variable and the bookbuf local variable. Finally you store the result back into byte 0 of iobuf. Repeat this for each byte (up to cnt bytes) in the two buffers. Return the value of cnt passed to the function. To avoid issues with sign extension it is important that you do all shifting and bitwise operations on unsigned int variables (they are the same size as registers on the 32-bit arm).

int encrypt(char *iobuf, char *bookbuf, int cnt) - assembly version
encrypt() is passed pointers to two buffers, the iobuf which contains the bytes to be encrypted and bookbuf the cipher key buffer. Each buffer contains the exact same number of bytes (cnt) in them. You need an outer loop that processes each byte in the buffers until all cnt bytes have been processed. Starting with byte 0 in iobuf, you need to write the assembly language instructions to load a byte into a register. There are many ways to reverse the bit order of the lower 8-bits (the byte) into a register. Typical approaches include using an inner loop to reverse the bits one at a time, or a sequence of bit-wise operations with shifts. Next you load byte 0 in bookbuf into another register and eor the two registers. Finally you store the result back into byte 0 of iobuf. Repeat this for each byte (up to cnt bytes) in the two buffers. Return the value of cnt passed to the function.

int decrypt(char *iobuf, char *bookbuf, int cnt) - both versions
Same processing as encrypt except do the eor ^ with the bookbuf first, then reverse the order of the bytes in the byte.

Coding Requirements

1. You will write your code only in C and hand-generated arm32 assembly
2. You are not allowed to use any automated tools that generate C source or assembly language source.
3. Your code must run in a Linux environment on the pi-cluster.
4. Do not use recursive solutions in your code.
5. In PA8 do not use any helper functions when writing Cencrypt.c, Cdecrypt.c, encrypt.S or decrypt.S as we have not covered how to call functions in assembly yet (and the code is too simple to require the use of a helper function).
6. **You cannot use any downloaded software** (other than what is in the github repository **for this PA**). **If it is not already installed on the pi-cluster by ITS, you cannot use it.**
7. Make sure you pass all the tests before submitting to gradescope.

Turning in Files For Your Grade

Before submitting your program to gradescope, make sure at a minimum that your program passes the supplied tests in the test harness.

Submit to gradescope under the **assignment titled PA8**

ONLY submit the following files (total of four (4) files):

```
Cencrypt.c
Cdecrypt.c
encrypt.S
decrypt.S
```

You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder.

You can also zip all the files and upload the .zip to the assignment. Ensure that the files you submit are not in a nested folder.

Appendix 1 Bit and Shift Operations C and Assembly

Bitwise <op> description	C Syntax	*Arm <op> Syntax Op2: either register or constant value*	*Operation*
Bitwise **AND**	`a & b`	`and` R_d, R_n, Op2	R_d = R_n & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	`a & ~b`	`bic` R_d, R_n, Op2	R_d = R_n & ~Op2
Bitwise **OR**	`a | b`	`orr` R_d, R_n, Op2	R_d = R_n | Op2
Exclusive **OR**	`a ^ b`	`eor` R_d, R_n, Op2	R_d = R_n ^ Op2

Instruction	*Syntax*	*Operation*	*Notes*	*Diagram*
Logical Shift Left int x; or unsigned int x x << n;	LSL R_d, R_m, *const5* LSL R_d, R_m, R_s	$R_d \leftarrow R_m$ << *const5* $R_d \leftarrow R_m$ << R_s	Zero fills shift: 0 - 31	
Logical Shift Right unsigned int x; x >> n;	LSR R_d, R_m, *const5* LSR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Zero fills shift: 1 - 32	
Arithmetic Shift Right int x; x >> n;	ASR R_d, R_m, *const5* ASR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Sign extends shift: 1 - 32	
Rotate Right unsigned int x; x = (x>>n)|(x<<(32-n)) ;	ROR R_d, R_m, *const5* ROR R_d, R_m, R_s	$R_d \leftarrow R_m$ ror *const5* $R_d \leftarrow R_m$ ror R_s	right rotate rot: 0 - 31