Assignment 1: Legacy Fortran (25%)

LUCIFER - A CRYPTOGRAPHIC ALGORITHM

This assignment deals with a program for a cipher, LUCIFER, which is a direct predecessor of the DES algorithm. A description of the algorithm can be found in this paper:

- Sorkin, A., "*Lucifer, A Cryptographic Algorithm*", Cryptologia, 8(1), pp.22.41 (1984).

LUCIFER was designed by IBM and was an iterative block cipher which used Feistel rounds. This means LUCIFER scrambled a block of data by performing an encipherment step on that block several times. The step used involved taking the key for that step and half of the block to calculate an output which was then applied by exclusive-OR to the other half of the block. The halves of the block were then swapped allowing both halves of the block to be modified an equal number of times.

The paper provides a number of Fortran programs for the algorithm:

- A simple Fortran subroutine **`lucifer()`**, which implements Lucifer.
- A Fortran **main program** which calls Lucifer.
- Two subroutines **`expand()`** which expand input bytes into bit array format, and **`compress()`** which compresses bit array format back into byte format.

The program takes a key and plaintext word in hexadecimal as input (using the formatting in the Fortran **read** statement).

TASK

Read over the code provided (see Appendix A, B, C attached), and also the algorithm as described in the paper. The series of programs provided are in a version of Fortran prior to Fortran 90. This code will compile using **gfortran** as per the "Sample Output".

Perform the following tasks:

TASK 1:

This task involves re-engineering the legacy Fortran code. No changes need to be made to the algorithm.

1. Re-engineer and refactor all four pieces of Fortran code - **lucifer()**, **expand()**, **compress()**, and the main program - provided to F95+ standards. This means removing any legacy artifacts (see "Re-engineering" below). The program file should be named **luc.f95**.

2. Modify the main program to print the ciphertext before it is decrypted.
3. Provide a good user interface to make the program easy to use.
4. Document your program using the information in the paper, and the comments in the original program.

TASK 2:

This task involves adding the functionality to allow the user to input an ASCII word which is converted to hexadecimal, and then processed as normal.

1. Copy the working program to create **luchex.f95**.
2. Add a module named **hex.f95**, which will be used by the program **luchex**, and includes the following three subprograms:
   2.1. A subprogram, **readWord()**, to obtain input for a word to be encrypted by the user.
   2.2. A subprogram, **word2hex()**, to convert the user input to hexadecimal, which can then be input to the **lucifer()** subroutine.
   2.3. A subprogram **printhex()**, to print the hex version of the word.
3. Modify **luchex.f95** in an appropriate manner to achieve the following:
   3.1. The user is able to enter a word 8-10 characters in length, instead of the AAA...BBB sample of Task 1.
   3.2. The program then converts this to hexadecimal, and prints out the hexadecimal version of the code.
   3.3. The program then submits the word as normal to the subroutine **lucifer()**.

Note that the only thing that changes in this version of the program is instead of entering the hexadecimal AAA...BBB (see Sample Output below), the user can encrypt/decrypt a real word.

Subprograms dependencies:

Here are the list of required subprograms. You may add any other additional functions you require.

Program: `luc.f95`:

`lucifer(d,k,m)`
- A subroutine which implements the Lucifer algorithm, allowing for both encryption and decryption. The parameters are as follows:
  - **d** = encipher (0) or decipher (1) (in)
  - **k** = key (array of values, 0 or 1) (in)
  - **m** = message (array of values, 0 or 1) (in/out)

`expand(a,b,l)`
- A subroutine which expands input bytes into binary array format. The parameters are as follows:
  - **a** = array in bit array format (out)
  - **b** = array in byte format (in)
  - **l** = length of array in hexdigits (in)

`compress(a,b,l)`

- A subroutine which compresses array format back into byte format. The parameters are as follows:
  - **a** = array in bit array format (in)
  - **b** = array in byte format (out)
  - **l** = length of array in hexdigits (in)

Module: **`hex.f95`**:

`readWord(w)`

- A function which takes input from the user in the form of a word to be encrypted, and returns the word to the calling program. The parameters are as follows:
  - **w** = user submitted word in ASCII (out)

`word2hex(w,h,l)`

- A function which converts the ASCII word obtained using readWord(), to hexadecimal, before it is passed to the subroutine expand(). The parameters are as follows:
  - **w** = word (in)
  - **h** = word in hexadecimal (out)
  - **l** = the length of the word entered (out)

`printhex(h,l)`

- A function which prints the word in hexadecimal. The parameters are as follows:
  - **h** = word in hexadecimal (out)
  - **l** = the length of the word entered (out)

Re-engineering:

Make sure to convert/remove any structures which are relevant/irrelevant, via refactoring and restructuring. Your code should be clean and easy to understand (unlike the existing code). The program may contain structures which should be modified or removed to make the program more maintainable. Some examples include: GOTO statements, equivalence statements, arithmetic IF statements, and label-enabled looping structures. Structures such as arrays, and strings should also be modified to newer constructs. Note that to remove the **equivalence** statement requires the arrays to be reshaped (using the function **reshape()**).

COMPILING

Please <u>do not include</u> a Makefile, and make sure your program compiles in the following manner:

Task 1:

```
> gfortran -Wall luc.f95
```

Task 2:

```
> gfortran -Wall luchex.f95 hex.f95
```

SAMPLE OUTPUT

The original program `lucifer.for` as coded produces the following output (input is shown in **bold**):

```
$ ./a.out
 key
0123456789ABCDEFFEDCBA9876543210
 plain
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
 key
 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1
 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1
 1 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1
 1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1
 1 1 1 1 1 1 1 0 1 1 0 1 1 1 0 0
 1 0 1 1 1 0 1 0 1 0 0 1 1 0 0 0
 0 1 1 1 0 1 1 0 0 1 0 1 0 1 0 0
 0 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0
 plain
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 plain
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 key
 0123456789ABCDEFFEDCBA9876543210
 plain
 AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
```

The modified program of Task 1 should produce output which looks something like this (you should use the bit output to verify your algorithm works as you are reengineering it, but do not have to include it in the final output):

```
 key
0123456789ABCDEFFEDCBA9876543210
 plaintext
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
 Encrypted message
 ciphertext
7C790EFDE03679E4BF28FE2D199E41A0
 Decrypted message
 key
0123456789ABCDEFFEDCBA9876543210
 plaintext
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
```

The modified program of Task 2 should produce output which looks something like this:

```
 key
0123456789ABCDEFFEDCBA9876543210
 Enter word:
hello
 Hexadecimal word:
68656C6C6F
ciphertext
FA6B0BD120FB28D5829BF9813A3ECD63
 key
0123456789ABCDEFFEDCBA9876543210
 plaintext
68656C6C6F0000000000000000000000
```

WRITE-UP INFORMATION

REFLECTION REPORT

Discuss your program in a one (1) page (single-spaced) **reflection report**, explaining the decisions you made in the process of reengineering your program. It should provide a synopsis of your experience with Fortran.

You should attempt to answer the following questions:

- What were the greatest problems faced while re-engineering the algorithm in Fortran?
- What particular features make Fortran a good language? (In comparison to C for instance)
- Would it have been easier to write the program in a language such as C?
- Given your knowledge of programming, was Fortran easy to learn?

DELIVERABLES

The submission should consist of the following items:

- The reflection report (PDF).
- The code (well documented and styled appropriately of course):
  `luc.f95, luchex.f95, hex.f95`

- Both the code and the reflection report can be submitted as a ZIP, TAR, or GZIP file.

SKILLS

- Fortran programming, re-engineering, program comprehension, ability to review specifications, ability to add on additional functionality, ability to write a Fortran program.

APPENDIX A: MAIN PROGRAM

```
      program luc

      implicit integer (a-z)
      data handle/0/
      dimension k(0:7,0:15),m(0:7,0:7,0:1)
      dimension key(0:127), message(0:127)
      equivalence (k(0,0),key(1)),(m(0,0,0),message(1))

      dimension kb(0:31),mb(0:31)
      write(*,1003)
      read(*,1004) (kb(i),i=0,31)

      write(*,1005)
      read(*,1006) (mb(i),i=0,31)

      call expand(message,mb,32)
      call expand(key,kb,32)

      write(*,1000) (key(i), i=0,127)
      write(*,1001) (message(i), i=0,127)

      d=0
      call lucifer(d,k,m)

      d=1
      call lucifer(d,k,m)

      write(*,1001) (message(i),i=0,127)

      call compress(message,mb,32)
      call compress(key,kb,32)
      write(*,1003)
      write(*,1007) (kb(i),i=0,31)
      write(*,1005)
      write(*,1007) (mb(i),i=0,31)

1000  format(' key '/16(1x,i1))
1001  format(' plain '/16(1x,i1))
1002  format(' cipher '/16(1x,i1))
1003  format(' key ')

1004  format(32z1.1)
1005  format(' plain ')
1006  format(32z1.1)
1007  format(1x,32z1.1)
      end
```

APPENDIX B: SUBROUTINE LUCIFER

```
      subroutine lucifer(d,k,m)
      implicit integer(a-z)
      dimension m(0:7,0:7,0:1),k(0:7,0:15),o(0:7)

      dimension sw(0:7,0:7),pr(0:7),tr(0:7),c(0:1)
      dimension s0(0:15),s1(0:15)
      equivalence (c(0),h),(c(1),l)

c     diffusion pattern
      data o/7,6,2,1,5,0,3,4/

c     inverse of fixed permutation
      data pr/2,5,4,0,3,1,7,6/

c     S-box permutations
      data s0/12,15,7,10,14,13,11,0,2,6,3,1,9,4,5,8/
      data s1/7,2,14,9,3,11,0,4,12,13,1,10,6,15,8,5/

      h0=0
      h1=1

      kc=0
      if (d .eq. 1) kc=8

      do 100 ii=1,16,1

      if (d.eq.1) kc=mod(kc+1,16)
      ks=kc

      do 200 jj=0,7,1
      l=0
      h=0

      do 400 kk=0,3,1
      l=l*2+m(7-kk,jj,h1)
400   continue
      do 410 kk=4,7,1
      h=h*2+m(7-kk,jj,h1)
410   continue

      v=(s0(l)+16*s1(h))*(1-k(jj,ks))+(s0(h)+16*s1(l))*k(jj,ks)

      do 500 kk=0,7,1
      tr(kk)=mod(v,2)
      v=v/2
500   continue

      do 300 kk=0,7,1
      m(kk,mod(o(kk)+jj,8),h0)=mod(k(pr(kk),kc)+tr(pr(kk))+
     + m(kk,mod(o(kk)+jj,8),h0),2)
300   continue
      if (jj .lt. 7 .or. d .eq. 1) kc=mod(kc+1,16)
200   continue

      jjj=h0
      h0=h1
      h1=jjj
```

```
100   continue

      do 700 jj=0,7,1
      do 800 kk=0,7,1
      sw(kk,jj)=m(kk,jj,0)
      m(kk,jj,0)=m(kk,jj,1)
      m(kk,jj,1)=sw(kk,jj)
800   continue
700   continue

      return
      end
```

APPENDIX C: EXTRA SUBROUTINES

```
      subroutine expand(a,b,l)
      implicit integer (a-z)
      dimension a(0:*),b(0:*)
      do 100 i=0,l-1,1
      v=b(i)
      do 200 j=0,3,1
      a((3-j)+i*4)=mod(v,2)
      v=v/2
200   continue
100   continue
      return
      end

      subroutine compress(a,b,l)
      implicit integer (a-z)
      dimension a(0:*),b(0:*)
      do 100 i=0,l-1,1
      v=0
      do 200 j=0,3,1
      v=v*2+mod(a(j+i*4),2)
200   continue
      b(i)=v
100   continue
      return
      end
```