SCHOOL OF ENGINEERING SCIENCE
SIMON FRASER UNIVERSITY
ENSC 350 – Digital Systems Design
Spring 2018

Lab5: Memory, Scheduling, and Decryption

Working week: Mar 12-16, 2018
Marking week: Mar 19-23, 2018

In this lab, you will get experience creating a design that contains several on-chip memories. This is an opportunity to practice what you learned in the lecture on on-chip memory; be sure to review and understand that slide set before starting this lab.

The circuit you will create is an RC4 Decryption circuit. RC4 is a popular stream cypher, and until recently was widely used in encrypting web traffic among other uses. RC4 has now been deemed insecure and has been replaced by several variants. Still, RC4 is an important algorithm and provides a good vehicle for studying digital circuits that made extensive use of on-chip memory. It also provides a basis for implementing some of these variants that are more secure.

In this lab, you will first design an RC4 decryption circuit. The secret key will be obtained from a bank of switches on your DE2 board, and the encrypted message will be given to you as a ROM initialization file. In Task 1 and 2, you will build the basic decryption circuit. In Task 3, you will extend this to build an RC4 cracking circuit; the circuit will implement a ‘brute-force’ attack on RC4 by cycling through the entire keyspace and stopping when a successful decryption is performed. Those of you that want to go further can consider building multiple functional units, each of which cycles through a portion of the keyspace in parallel for faster cracking (Challenge Task).

Background: RC4 Decryption

This section describes the RC4 decryption algorithm. You can find more information by doing a Google Search, but the information here should be sufficient to complete this lab. Interestingly, the same algorithm is used for both encryption and decryption, but we will only use it for decryption in this lab.

RC4 is a stream cipher. Based on a key, the algorithm generates a series of bytes. Each byte is XOR’ed with one byte of a message to produce the decoded text.

The basic RC4 algorithm is shown on the following page:

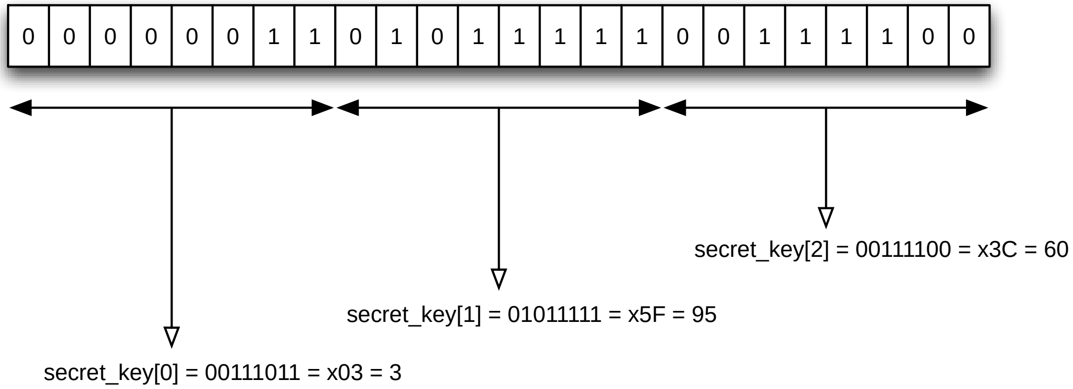
```
// Input:
//          secret_key [] : array of bytes that represent the secret key.  In our implementation,
//                          we will assume a key of 24 bits, meaning this array is 3 bytes long
//          encrypted_input []:  array of bytes that represent the encrypted message.  In our
//                          implementation, we will assume the input message is 32 bytes
// Output:
//          decrypted _output []:  array of bytes that represent the decrypted result.  This will
//                          always be the same length as encrypted_input [].

// initialize s array.  You will build this in Task 1
for i = 0 to 255 {
         s[i] = i;
}

// shuffle the array based on the secret key.  You will build this in Task 2
j = 0
for i = 0 to 255 {
         j = (j + s[i] + secret_key[i mod keylength] ) mod 256  //keylength is 3 in our impl.
         swap values of s[i] and s[j]
}

// compute one byte per character in the encrypted message.  You will build this in Task 2
i = 0, j=0
for k = 0 to message_length-1 {   // message_length is 32 in our implementation
         i = (i+1) mod 256
         j = (j+s[i]) mod 256
         swap values of s[i] and s[j]
         f = s[ (s[i]+s[j]) mod 256 ]
         decrypted_output[k] = f xor encrypted_input[k]   // 8 bit wide XOR function
}
```

The length of the secret key can vary in different applications, but is typically 40 bits (5 bytes). In our implementation, we will assume a 24 bit (3 byte) key. We are using a smaller key to ensure that you can “crack” the implementation in Task 3 in a reasonable amount of time. Note that in the second loop above, secret_key[0] refers to the first *byte* of the key, secret_key[1] refers to the second *byte* of the secret key, and secret_key[2] refers to the third *byte* of the secret key. This is illustrated below; in this example, the 24-bit secret key is 000000110101111100111100 = x035F3C. The diagram shows how each of secret_key[0], secret_key[1], and secret_key[2] are found.



0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 0 0
secret_key[2] = 00111100 = x3C = 60
secret_key[1] = 01011111 = x5F = 95
secret_key[0] = 00111011 = x03 = 3

The encrypted message (the input) consists of 32 bytes, and in the above pseudo-code, **encrypted_input[k]** refers to the kth byte in the encrypted input. The decrypted message (the output) will also be 32 bytes; in the above pseudo-code, **decrypted_output[k]** refers to the kth byte in the encrypted output.

Task 1: Creating a memory, instantiating it, and writing to it (2 marks)

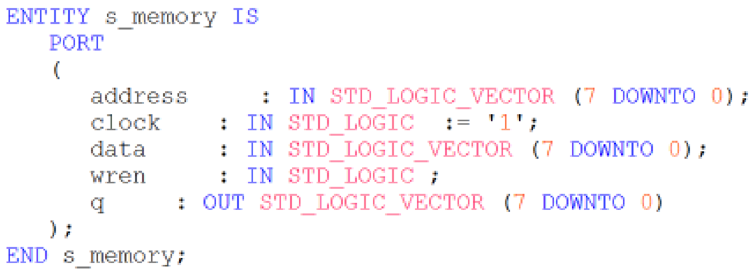
In this task, you will get started by creating a RAM using the Wizard in the IP Catalog, creating circuitry to fill the memory, and observing the memory contents using the In-System Memory Content Editor.

a) Creating a RAM block using the Wizard

First, create a new Quartus II project. Then, choose **Tools->IP Catalog.** In the IP Catalog, select **Basic Functions->On Chip Memory->RAM: 1-Port.** A window **Save IP Variation** will appear. Modify the file name to be **s_memory.vhd** in your working directory. Hit **OK**. In the next few panels, customize your IP core as follows:

How wide should the 'q' output bus be? 8 bits
How many 8-bit words of memory? 256 words
What should the memory block type be? M9K
Set the maximum block depth to be: auto words
What clocking method would you like to use? Single clock
Which ports should be registered: Make sure 'q' output port is *unselected*
Create one clock enable signal… : do not select
Create an 'aclr' asynchronous clear… : do not select
Create a 'rden' signal…: do not select
What should the q output be when reading… : New Data
Get X's for write mask bytes… : selected
Do you want to specify the initial contents? No
Allow In-System Memory Content Editor to capture and update.. Select this option
The 'Instance ID' of this RAM is S
Generate netlist: do not select
Output files: select s_memory.cmp (VHDL Component declaration file)
Do you want to add the Quartus II File to the project? Yes

When you finish this, you will find the file **s_memory.qip** in your project file list. Click on the triangle beside **s_memory.qip** to expand, and you will see **s_memory.vhd**. Open this file and examine it. Near the top of your file, you will find the Entity declaration for **s_memory**, which will look something like this:



```
ENTITY s_memory IS
   PORT
   (
      address      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      clock     : IN STD_LOGIC   := '1';
      data      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      wren      : IN STD_LOGIC ;
      q      : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
   );
END s_memory;
```

Be sure your declaration matches the above. This is the entity you will include as a component in your design.

b) Creating a VHDL module that writes to your memory

To get started, examine **ksa.vhd** from the task1 subdirectory in the distribution zip file. You will see that this declares the component that you should have created using the Wizard. Add this file to your project.

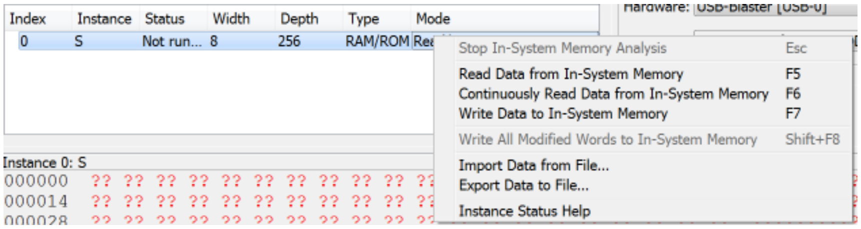
Examine this file. You are to add code to implement the following algorithm (this is the very first part of RC4). Be sure to study the slide set from the lectures before writing your code. After the memory is filled, go to a state called DONE that does nothing but stay in DONE (a endless loop).

```
for i = 0 to 255 {
        s[i] = i;
}
```

Import your pin assignments (as usual) and compile. Be sure your design compiles without errors before going on.

c) Running your code.

Download your design and run it as usual. The circuit will fill the on-chip memory with the numbers 0-255. To examine the contents of the memory, you can choose **Tools->In System Memory Content Editor** (while your circuit is running). If you see "No Devices Detected" in the JTAG Chain configuration, select the hardware pulldown and select USB Blaster. This will open a window that shows the memory contents. In the instance manager window (left) you should see a list of memories in your design (in this case, it is only **S**). Right click **S**, and choose **Read Data from In-System Memory**.

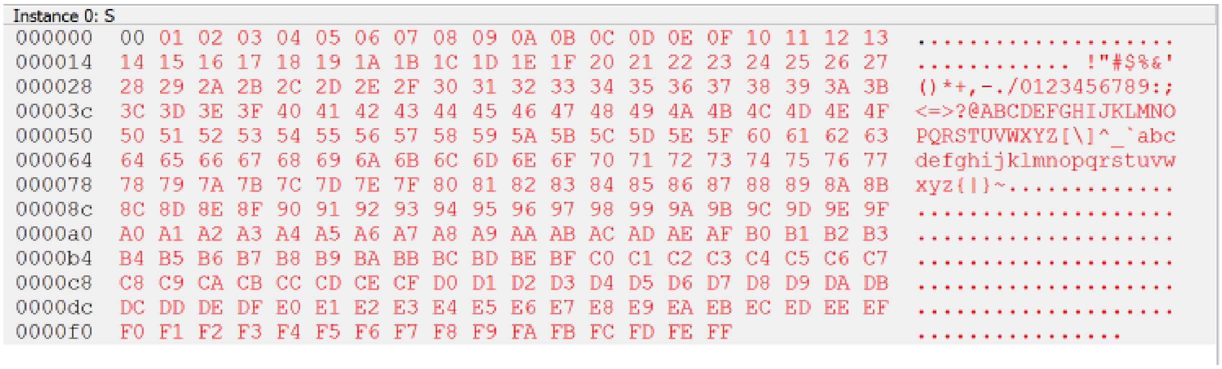


Index	Instance	Status	Width	Depth	Type	Mode
0	S	Not run...	8	256	RAM/ROM	Rea

```
Instance 0: S
000000  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
000014  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
000028  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
```

Hardware: USB-Blaster [USB-0]
Stop In-System Memory Analysis Esc
Read Data from In-System Memory F5
Continuously Read Data from In-System Memory F6
Write Data to In-System Memory F7
Write All Modified Words to In-System Memory Shift+F8
Import Data from File...
Export Data to File...
Instance Status Help

This memory contents will be read and displayed. You should see something like:



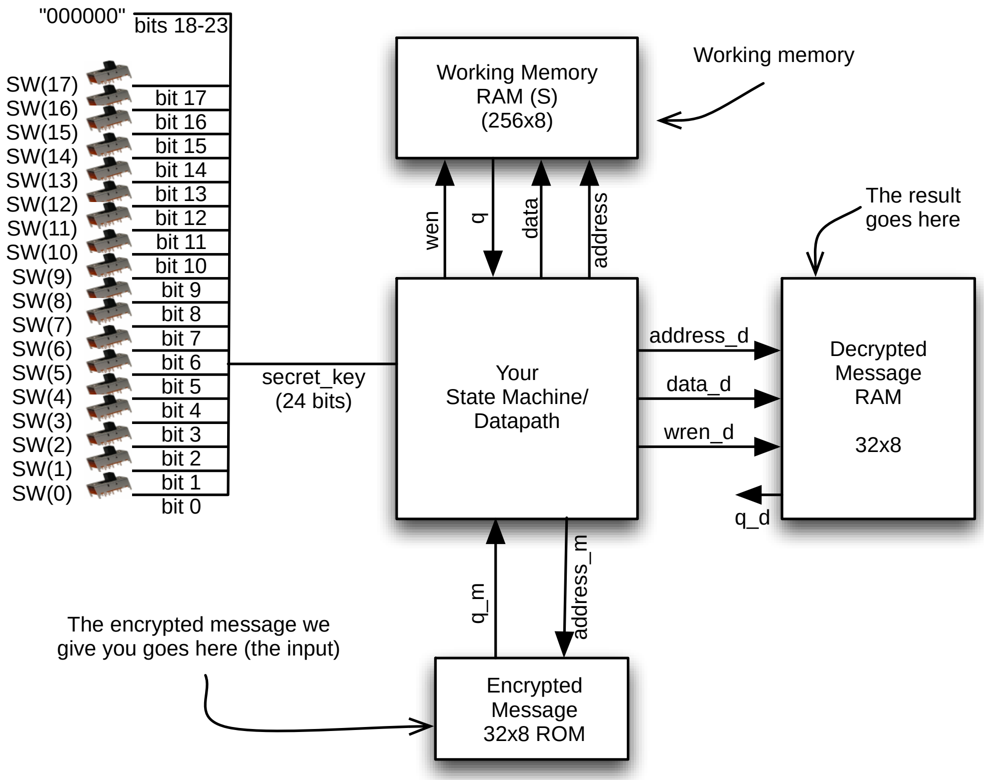
```
Instance 0: S
000000  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13  ....................
000014  14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27  ............ !"#$%&'
000028  28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B  ()*+,-./0123456789:;
00003c  3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F  <=>?@ABCDEFGHIJKLMNO
000050  50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63  PQRSTUVWXYZ[\]^_`abc
000064  64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77  defghijklmnopqrstuvw
000078  78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B  xyz{|}~.............
00008c  8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F  ....................
0000a0  A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3  ....................
0000b4  B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7  ....................
0000c8  C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB  ....................
0000dc  DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF  ....................
0000f0  F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF              ................
```

As you can see, the memory has been filled. Each location *i* contains the value *i*. Be sure that your program has filled the memory as expected. You will use the In-System Memory Content Editor to help you debug your lab in Task 2, so be sure you are able to get this working before moving on.

Task 2: Building a Single Decryption Core (4 marks)

In this task, you are to continue your design from Task 1 to implement a single decryption core. Given a 24 bit key, and an encrypted message in a ROM, your algorithm will decrypt the message and store the result in another memory. You will then use the In-System Memory Content Editor to read the result to ensure the encryption occurred as expected.

As shown below, your system will consist of three memories and a state machine/datapath. The initial encrypted message is stored in a 32-word ROM (which you will initialize using a .mif file when you compile the design). The result is stored in a 32-word RAM (which you can examine using the In-System Memory Content Editor after the decryption is complete. In addition, you will use a 256x8 bit S memory (the same memory you used in Task 2). The secret key is set using the slider switches. Note that the secret key should be 24 bits long, but you only have 18 slider switches on the DE2 board. *For this task only* (not Task 3), you can hardwire the upper order six bits of the secret key to 0.

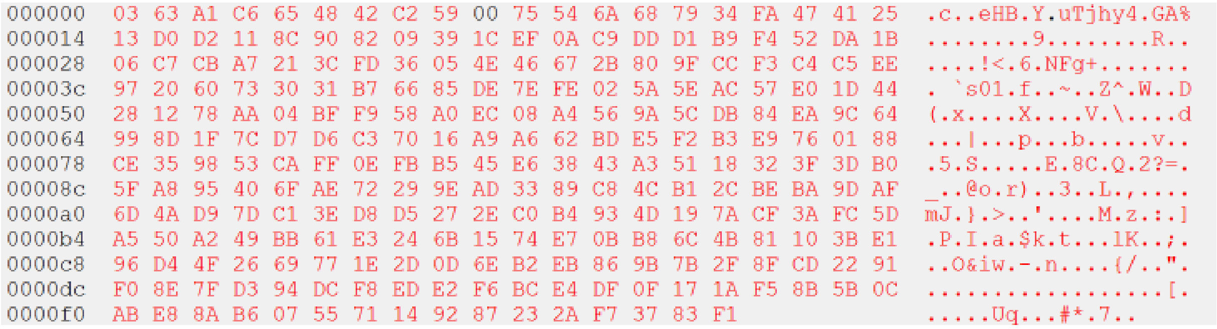


Task 2a) Second Loop in algorithm

Starting with your Task 1 code, add hardware to implement the second loop from the algorithm on the second page of this handout. That is, add code to implement

```
j = 0
for i = 0 to 255 {
        j = (j + s[i] + secret_key[i mod keylength] ) mod 256  //keylength is 3 in our impl.
        swap values of s[i] and s[j]
}
```

This code does not use the encoded message ROM or the decoded message RAM (you will add that in part b below). Test your code as follows. Set the secret key to 00000011 01011111 00111100 = x035F3C (remember you can set the lower order 18 bits of secret key using the slider switches as in the above diagram) and examine the **s** array using the in-system memory content editor as before. You should see the following. If you do, your code is likely correct; if not, you have some debugging to do. Don't move on until you have this right.



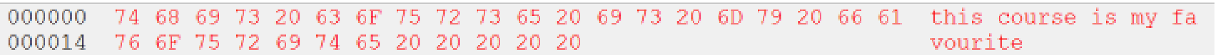
```
000000  03 63 A1 C6 65 48 42 C2 59 00 75 54 6A 68 79 34 FA 47 41 25  .c..eHB.Y.uTjhy4.GA%
000014  13 D0 D2 11 8C 90 82 09 39 1C EF 0A C9 DD D1 B9 F4 52 DA 1B  ........9........R..
000028  06 C7 CB A7 21 3C FD 36 05 4E 46 67 2B 80 9F CC F3 C4 C5 EE  ....!<.6.NFg+.......
00003c  97 20 60 73 30 31 B7 66 85 DE 7E FE 02 5A 5E AC 57 E0 1D 44  . `s01.f..~..Z^.W..D
000050  28 12 78 AA 04 BF F9 58 A0 EC 08 A4 56 9A 5C DB 84 EA 9C 64  (.x....X....V.\....d
000064  99 8D 1F 7C D7 D6 C3 70 16 A9 A6 62 BD E5 F2 B3 E9 76 01 88  ...|...p...b.....v..
000078  CE 35 98 53 CA FF 0E FB B5 45 E6 38 43 A3 51 18 32 3F 3D B0  .5.S.....E.8C.Q.2?=.
00008c  5F A8 95 40 6F AE 72 29 9E AD 33 89 C8 4C B1 2C BE BA 9D AF  _..@o.r)..3..L.,....
0000a0  6D 4A D9 7D C1 3E D8 D5 27 2E C0 B4 93 4D 19 7A CF 3A FC 5D  mJ.}.>..'....M.z.:.]
0000b4  A5 50 A2 49 BB 61 E3 24 6B 15 74 E7 0B B8 6C 4B 81 10 3B E1  .P.I.a.$k.t...lK..;.
0000c8  96 D4 4F 26 69 77 1E 2D 0D 6E B2 EB 86 9B 7B 2F 8F CD 22 91  ..O&iw.-.n....{/..".
0000dc  F0 8E 7F D3 94 DC F8 ED E2 F6 BC E4 DF 0F 17 1A F5 8B 5B 0C  ..................[.
0000f0  AB E8 8A B6 07 55 71 14 92 87 23 2A F7 37 83 F1              .....Uq...#*.7..
```

Task 2b) Rest of algorithm

Now you will complete the rest of the algorithm (the third for loop in the pseudo-code on Page 2 of this handout). This will require adding two memories: one to store the encrypted message (this could be a ROM since the message to decrypt will be compiled with your hardware) and one to store the decrypted message (this could be a RAM since your circuit must write the result to it). To create the ROM and RAM, it is suggested that you use the Wizard (as you did for Task 1).

When creating a ROM using the Wizard from the IP catalog, you will have the ability to indicate an initialization file name. Use the name **message.mif**. You can examine an example **message.mif** from the distribution zip file (under the folder **secret_message_1**); you will need to make sure this file is in your working directory before you create your component with the Wizard. During compilation, the contents in this file is then read (notice: it is read at compile time), and is used as the initial contents of the memory. There are several other settings you will need to specify in the Wizard; you should know enough now that you can figure out what to use for each of these settings.

Test your design. If you decrypt the first message (in **secret_message_1**) in the distribution zip file (with secret key 00000011 01011111 00111100 = x035F3C) you should see the following if you use the In System Memory Content editor to view the contents of the Decrypted Message RAM after your algorithm completes (note: *not* the s memory; that will continue to contain pseudo-random bytes). As you can see, the decrypted message is an ASCII string that you can read.



```
000000  74 68 69 73 20 63 6F 75 72 73 65 20 69 73 20 6D 79 20 66 61  this course is my fa
000014  76 6F 75 72 69 74 65 20 20 20 20 20                          vourite
```

Check the other messages in the distribution zip file (each message has its own secret key). For each message, you will need to replace **message.mif** in your working directory. Due to a bug in Quartus II, you must also delete the **db** directory in your working directory before recompiling (the db directory acts as a cache, and caches your most recent **message.mif** file). You may find that you cannot delete the db directory while Quartus II is open; if that is the case for you, then quit Quartus II, delete the db directory, and then restart Quartus II. Remember that the contents of the initial memory contents file is read *at compile time*, not run time. Of course, the slider switches are read at run-time. Be sure to use the slider switches to set the appropriate secret key (each of the three messages in the distribution zip file has its own secret key).

Finally, you may see this common error: Watch out for a warning such as:

> *Critical Warning (127003): Can't find Memory Initialization File or Hexadecimal (Intel-Format) File ./db/message_mod.mif -- setting all initial values to 0*

If you see this warning, it did not find your **message.mif** file during compilation. Make sure it is in your project directory, and that you specified it properly when you made your component.

When you can read all the message, you can move on to Task 3. Be sure to save your code; if you don't get Task 3 working, you can get part marks for demoing what you have here.

Task 3: Cracking RC-4 (3 marks)

In this task, you are going to modify your design from Task 2 to "crack" a message that has been encrypted using RC4. In this case, you have the encrypted message, but you do not know the key. You will implement a brute-force algorithm that cycles through all possible keys. For each key, if the result string is a readable message, then you will assume the string is correct. More precisely, your circuit will search through the key space until it finds a result in which, for every character in the decoded output, the character is in the range [97,122] (corresponding to the characters 'a' to 'z') or the value 32 (which is a space). All characters in the correct output will either be a lower case letter or a space; the correct message will contain no upper character letters or punctuation. Note that it is *not* enough that just one character is a lower case letter or space; *all* 32 characters in the decoded output must be lower case letters or a space before you deem it correct. Of course, there is a chance that even though a key leads to an ASCII string, the string is not actually correct, but if you work out the math you will see that the odds of this are extremely low.

My implementation of this Task takes about 10 minutes to cycle through the entire key space. To make your life easier in the lab, I will tell you that, for every test case we will give you, the two most significant bits in the correct key are 0. This reduces the size of the keyspace you have to search, meaning you should be able to search the entire keyspace in a few minutes.

When your circuit finds a correct message, it should put the message in the ROM and enters an endless loop in which it does nothing. You can use the In System Memory Content editor to view the output to make sure it is correct. You should also turn on an LED to indicate that the search has complete. If you search the entire keyspace and do not find a correct message, then you should turn on a different LED.

I strongly urge you to also display the key that is being considered on the LEDs as well, so that you can observe that the circuit is making progress.

You can extract some encoded messages from the distribution zip file. You can test your design on these messages; you should be able to "crack" each test case. For your demonstration, the TA will give you a new encoded message that you have never seen before, so make sure your design works before demoing!

If you have implemented the LCD you need to be careful to make it work with this task: don't display a partial message until you know it is the right message.

Challenge Task: Multi-Core Cracking (1 mark)

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 1 mark. If you don't demo the challenge task, the maximum score you can get on this lab is 9/10 (which is still an A+).

In the challenge task, you will accelerate the keyspace search by using multiple instantiations of the decryption circuit (core) in Task 3, and using each one to search a subset of the keyspace simultaneously. For example, if you have two cores, one core could search all even keys and one could search all odd keys. If you have 4 cores, each could search ¼ of the search space. When one unit finds a correct message, all units should stop.

For the challenge task, modify your design so that it contains at least four decryption cores that operate simultaneously.

Bonus Mark: LCD Display of Decoded Message (1 mark)

This bonus task can be completed in addition to the challenge task (for a maximum grade of 11 out of 10). It can only be added to your system after having completed task 2b.

In the bonus task, you have the chance to reuse one of your previous labs. Specifically, you can reuse your LCD driver core to write the contents of the Result RAM to the LCD. This will make debugging easier for both Task 2b and Task 3. This can be done in conjunction with or in place of the challenge task.

Marking: What to demo depends on how far you get.

If you get the challenge task done, demo that, and explain to the TA how you parallelized the code across multiple cores. The TA will likely give you a new encrypted message for you to solve. If your implementation is successful, you do not need to demo Tasks 2 or 3.

If you get Task 3 done, but not the challenge task, demo a working Task 3. The TA will likely give you a new encrypted message for you to solve. In this works, you do not need to demo Task 2.

If you only get Task 2 done, demo that. If you have Task 2 plus parts of Task 3 done (but not a complete working system), demo Task 2 and show the TA what you have for Task 3 for possible part marks.

If you only get Task 1 done, demo that. If you have Task 1 plus parts of Task 2 done (but not a complete working system), demo Task 1 and show the TA what you have for Task 2 for possible part marks.

If you didn't get Task 1 done, show the TA what you have for possible part marks.