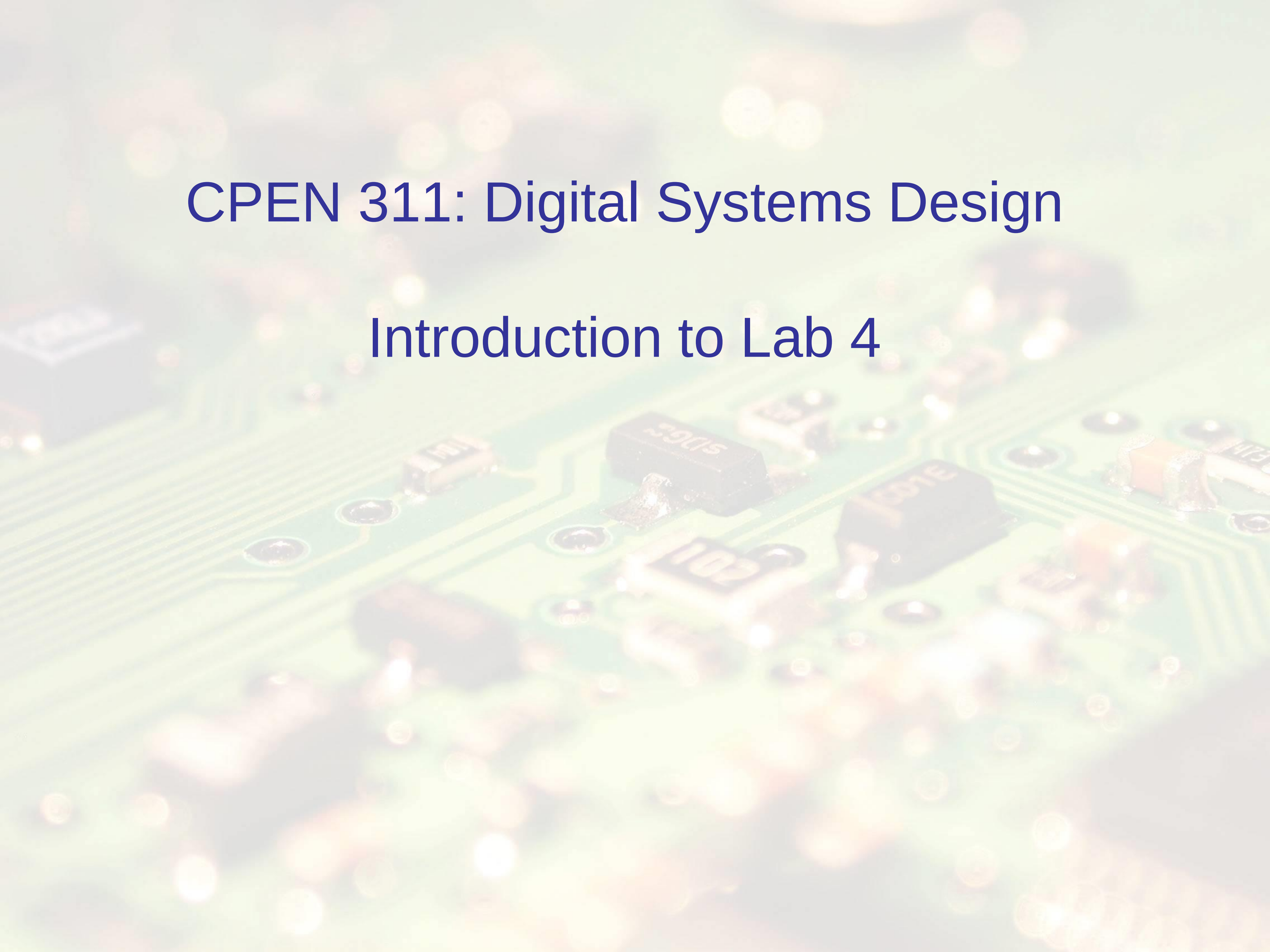
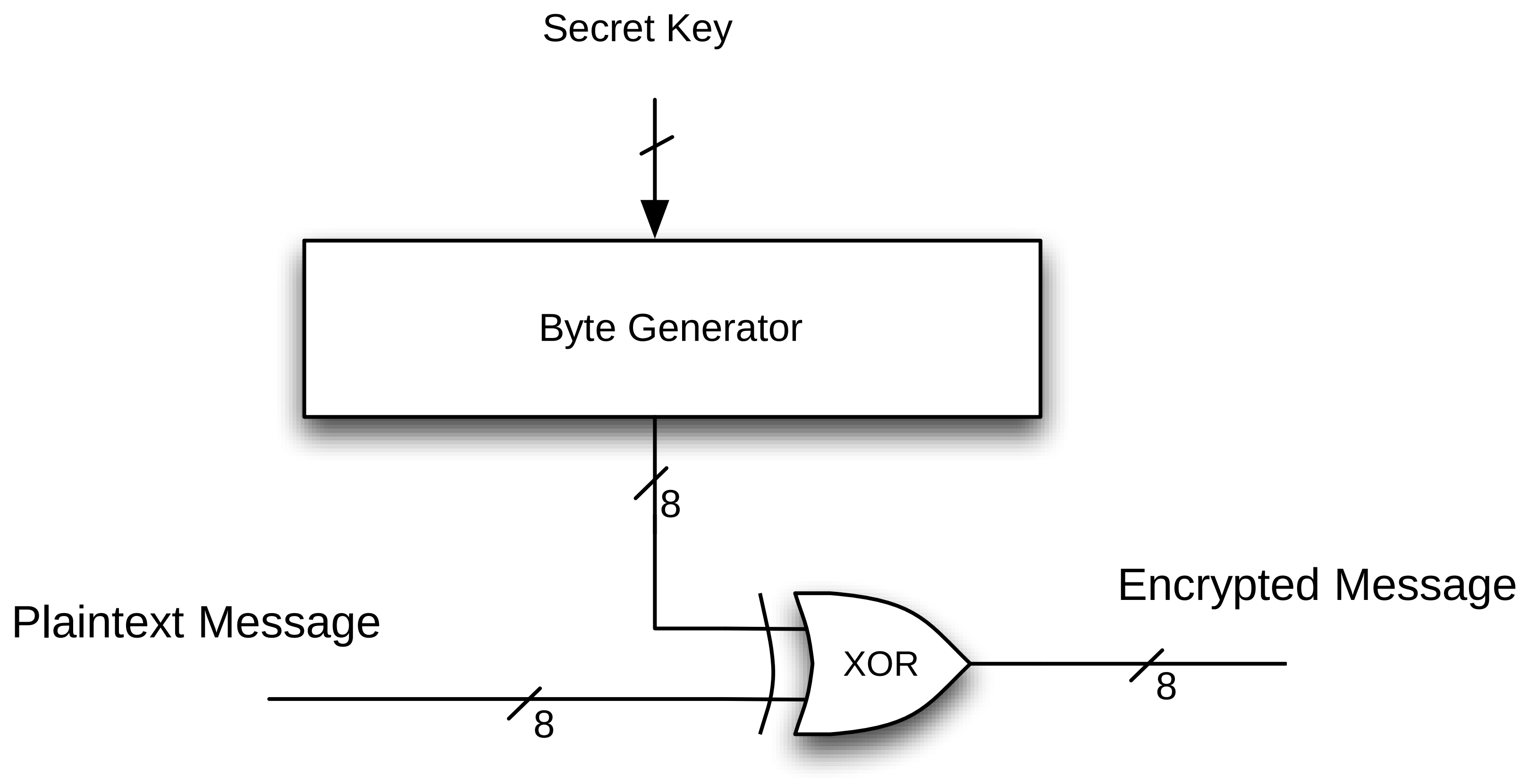


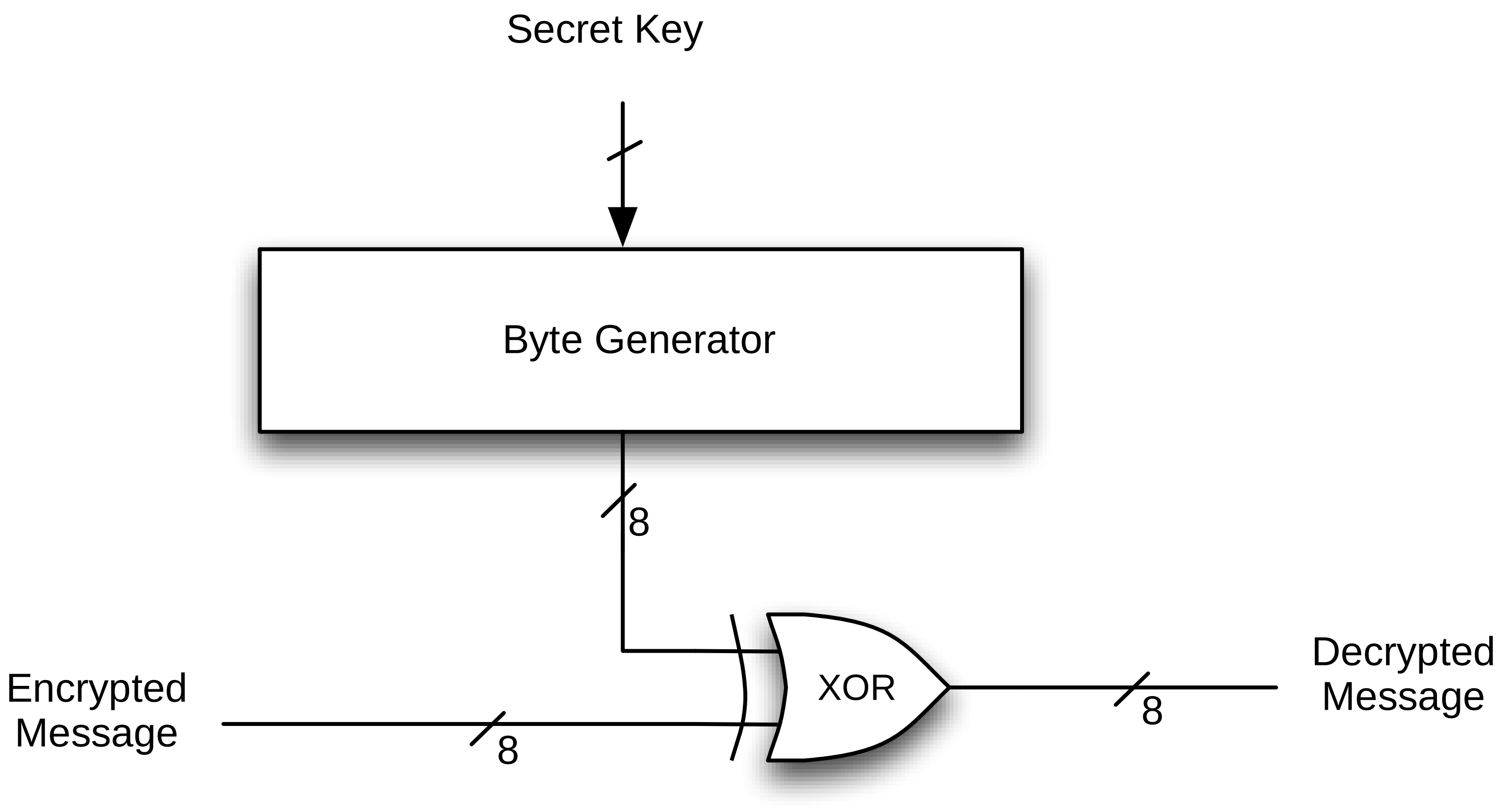
CPEN 311: Digital Systems Design

Introduction to Lab 4

RC4 Stream Cipher Encryption



RC4 Stream Cipher Decryption



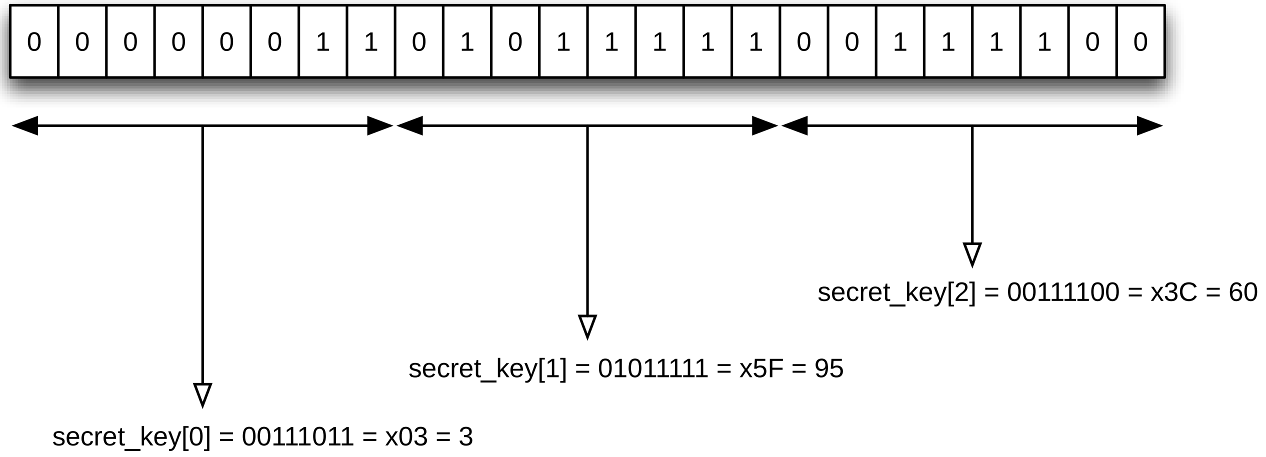
RC4 Overview

- RC4 is a widely used cipher but is not considered secure anymore
- RC4 is an example of Symmetric Key Cipher because the encryption and decryption keys are the same
- It is a Stream Cipher because each byte is encoded individually (unlike a block cipher where bytes are encoded as a block)
- One of the advantages of RC4 is that it has simple and fast hardware implementations, so it has been used in phone telephony
- Many other ciphers have a similar structure and operation

```
// 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 “Secret Key”



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

Simplifications

- However, if you define the variables i, j, and f as 8-bit registers, and since s[i] and s[j] are also 8-bit numbers, then due to the "automatic" modulo-256 that we get by having an 8-bit number, we can drop all the "mod 256"

- This is a significant simplification since the modulo operations could be synthesized by the compiler using divisions, which we know takes up a lot of resources.

- We still need to implement "i mod keylength". The Verilog keyword for mod is "%"

Simplified Equivalent Algorithm

```
// 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] )  //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
        j = j+s[i]
        swap values of s[i] and s[j]
        f = s[ (s[i]+s[j]) ]
        decrypted_output[k] = f xor encrypted_input[k]   // 8 bit wide XOR function
}
```

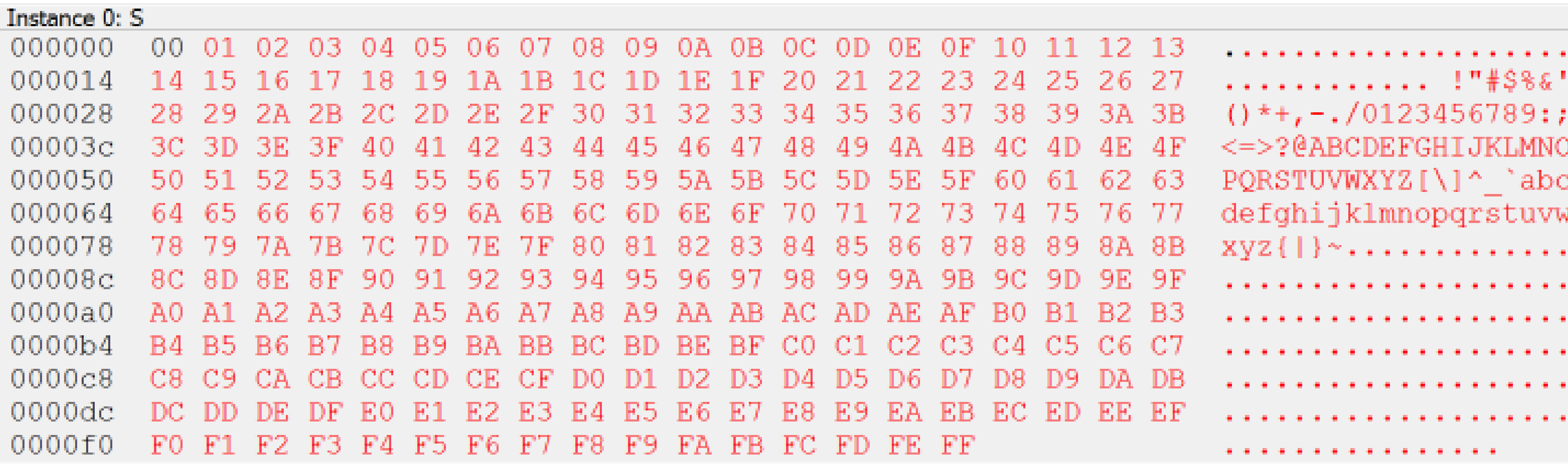
Task 1

Implement first loop of algorithm.

This will give you practice:

- Instantiating memories using MegaWizard
- Writing into memories
- Using the In-System Memory Content Editor

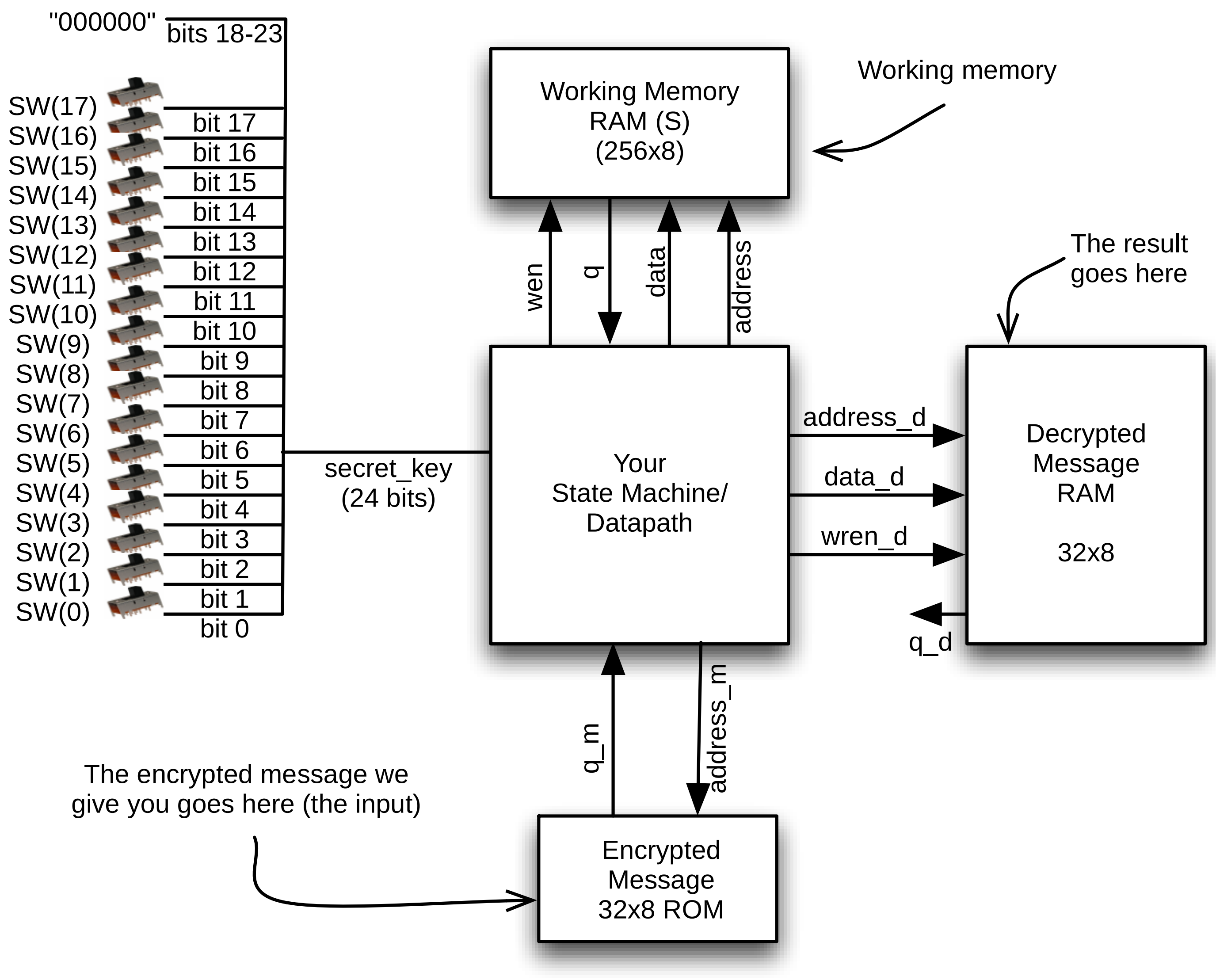
```
for i = 0 to 255 {
        s[i] = i;
}
```



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               ................
```

Task 2



"000000"
bits 18-23
SW(17)
SW(16)
SW(15)
SW(14)
SW(13)
SW(12)
SW(11)
SW(10)
SW(9)
SW(8)
SW(7)
SW(6)
SW(5)
SW(4)
SW(3)
SW(2)
SW(1)
SW(0)
bit 17
bit 16
bit 15
bit 14
bit 13
bit 12
bit 11
bit 10
bit 9
bit 8
bit 7
bit 6
bit 5
bit 4
bit 3
bit 2
bit 1
bit 0
secret_key (24 bits)
Working Memory RAM (S) (256x8)
Working memory
wen
q
data
address
Your State Machine/ Datapath
The result goes here
address_d
data_d
wren_d
q_d
Decrypted Message RAM 32x8
q_m
address_m
The encrypted message we give you goes here (the input)
Encrypted Message 32x8 ROM

Task 3

RC-4 Cracking:

Cycle through all keys. For each key, if the message is something readable, you have cracked the message!

To make this feasible, you will assume 24 bit keys

- My implementation takes about 10 minutes to cycle through
- In the lab, bits 24 and 23 are both 0 -> 2.5 minutes

Scaling to a real implementation: 40 bits

$(10 \text{ min}) * 2^{(40-24)} = 455 \text{ days}$

We can probably fit about 8 of these on a chip -> 56 days

If the 20 groups in a lab got together -> 2.8 days

Challenge Task

- Implement multiple (4 or more) “cores” on the chip
- When one core finds the solution, all cores should stop
- This qualifies for 10 points of the 15 point bonus mark

Two Approaches

- One approach, adopted by Prof. Wilton who originally designed the lab, is to solve it using one FSM that is continually enhanced
- Another approach, which I took, is to solve the lab using the modular FSM approach along with the start-finish protocol to communicate between FSMs
- I used about 6 different FSMs to solve this lab
- Both approaches are valid and have their advantages and disadvantages
- The important thing is to design the FSM(s) in a reliable manner

Don’t freak out

- Do not be fooled by the complexity of this lab.
- Although it is more complicated than previous labs, we are still talking about an algorithm that can be expressed in a flowchart (or, in the case of the modular FSM approach, multiple flowcharts).
- You already know from the lectures and Lab 2 how to reliably convert any proper FSM flowchart to an FSM.
- In essence, therefore, this is not fundamentally different from Lab 2 except that the algorithm and flowcharts will be much more complicated.

Don’t freak out (2)

- Implementing “for” loops just means implementing a counter and checking its value
- You already did this for lab 2
- Multiple “for” loops mean multiple counters
- Remember that you just need to control those counters from your state machines, i.e. control resetting them and incrementing them
- Implementing reads from memory is something you also did in Lab 2. Here we need writes too.
- Here we are using embedded memories, so reading and writing from the memories is almost trivial and does not necessarily require using a dedicated FSM (though I did do that in my modular approach)

Working in groups of 2

- For this lab you may work in groups of 2, who must be from the same lab section, IF both group members have an average of 80% or more in labs 1 to 2.
- If you choose to work in a group of 2, then you must use the modular FSM approach with a standardized communications protocol to solve the lab
- I am adding the 2-person group option because:
  - I want to maximize your chances of success
  - I want to promote good modular hierarchical team design practices in this course
- If you work in groups of 2, you must use good design practices as discussed in the course, otherwise you will find that it is less efficient and less productive than groups of 1.

Final thoughts

- I suggest that you start work on this lab immediately. You know everything you need to know to in order to be able to work on it.
- Again, do not be fooled by the complexity of this lab: it is just another algorithm. You know how to convert algorithms to FSMs.
- Incremental design is extremely important in this lab.
- Good Luck!!!