ASSIGNMENT 1: MIPS (using MARS)

- SIDDHARTH PALOD (IMT2022002)

-RAHUL MUKUNDHAN (IMT2022518)

Q1) Assembly programming (Simple encryption and decryption)

Using the logic of an encryption key and a XOR gate ([A^K] ^K=A, where A is the input and K is the Key (^ is XOR)), the following code is presented below with its output.

(explanation shown through comments in the code)

mips1.asm

```
.data
plaintext: .asciiz "Enter Plain text: "
ciphertext: .asciiz "Cipher text: "
deciphertext: .asciiz "Deciphered text: "
data: .space 40
Key: .asciiz "mykey002"  #added key as my rollno

.text
main:
    li $t3, 0 # 0 for encryption, 1 for decryption (change as needed)
    la $a0, plaintext
    li $v0, 4
    syscall

    # Read the input
    la $a0, data
    la $a1, 40   # Maximum 40 characters can be read
    li $v0, 8
    syscall

    # Load the fixed key
    la $t4, Key

    # Encryption or Decryption Loop
    li $t0, 0      # Initialize loop counter

encrypt_decrypt_loop:
    lb $t2, 0($a0)   # Load a character from the input
    beqz $t2, end_encrypt_decrypt  # Terminate when the end of the string is reached

    # Encryption or Decryption
    lb $t1, 0($t4)   # Load a character from the key
    xor $t2, $t2, $t1   # XOR the character with the key
```

```
    sb $t2, 0($a0)   # Store the character (encrypted or decrypted) back
    addi $a0, $a0, 1   # Move to the next character
    addi $t4, $t4, 1   # Move to the next character in the key
    j encrypt_decrypt_loop

end_encrypt_decrypt:
    # Display the result (encrypted or decrypted text)
    la $a0, ciphertext
    li $v0, 4
    syscall

    # Print the result (encrypted or decrypted text)
    la $a0, data
    li $v0, 4
    syscall

    # Reset the key pointer for decryption
    la $t4, Key
    li $t3, 1  # Set the mode to decryption

    # Decryption Loop
    la $a0, data
    j decrypt_data_loop

decrypt_data_loop:
    lb $t2, 0($a0)   # Load a character from the data buffer
    beqz $t2, end_decryption  # Terminate when the end of the string is reached

    # Decryption
    lb $t1, 0($t4)   # Load a character from the key
    xor $t2, $t2, $t1   # XOR the character with the key
```

```
decrypt_data_loop:
    lb $t2, 0($a0)    # Load a character from the data buffer
    beqz $t2, end_decryption  # Terminate when the end of the string is reached

    # Decryption
    lb $t1, 0($t4)    # Load a character from the key
    xor $t2, $t2, $t1    # XOR the character with the key

    sb $t2, 0($a0)    # Store the character (decrypted) back
    addi $a0, $a0, 1    # Move to the next character
    addi $t4, $t4, 1    # Move to the next character in the key
    j decrypt_data_loop

end_decryption:
    # Display the result (decrypted text)
    la $a0, deciphertext
    li $v0, 4
    syscall

    # Print the result (decrypted text)
    la $a0, data
    li $v0, 4
    syscall

    # Terminate program
    li $v0, 10
    syscall
```

```
Mars Messages    Run I/O

Enter Plain text: RAM
Cipher text: ?8&oDeciphered text: RAM
```

output of the code)

Q2) Assembler (using Python)

```python
opcodes = {
    "li": "001001",
    "la": "001111",
    "syscall": "001100",
    "lb": "100000",
    "sb": "101000",
    "addi": "001000",
    "beqz": "000100",
    "j": "000010",
    "xor": "000000",
} #opcodes of the MIPS cmds used
registers = {
    "$zero": "00000",
    "$t0": "01000",
    "$t1": "01001",
    "$t2": "01010",
    "$t3": "01011",
    "$t4": "01100",
    "$t5": "01101",
    "$a0": "00100",
    "$a1": "00101",
    "$v0": "00010",
    "$0": "00000",
}#various registers used
label_addresses = {"plaintext":"0001000000000001",
                   "data":"0001000000000001",
```

```python
label_addresses = {"plaintext":"0001000000000001",
                    "data":"0001000000000001",
                    "Key" : "0001000000000001",
                    "ciphertext":"0001000000000001",
                     "deciphertext":"0001000000000001",
                     "40":"0000000000101000",
                     "8":"0000000000001000"}
#use of variables in the .data segment
jump_label = {"encrypt_decrypt_loop":"00001000000000000000001101","decrypt_data_loop":"00001000000000000000100011"}
#conditions to jump

def assemble(instruction):
    parts = instruction.replace(',', '').split() #split an instruction into words
    if not parts:
        return ""
    if parts[-1].endswith(":"): #a potential function
        label_name = parts[0][:-1]
        label_addresses[label_name] = len(label_addresses)
        return ""
    opcode = opcodes.get(parts[0]) #get opcode
    if opcode is None:
        return ""
    if parts[0] == "li": # assign regs to 0
        rd = registers.get(parts[1])
        if rd is not None:
```

```python
            imm = format(int(parts[2]), '016b')
            return opcode +"00000"+ rd + imm
    elif parts[0] == "la": #load address to a a0
        rd = registers.get(parts[1]) # get 5 bit rd
        if rd is not None:
            label_name = parts[2]
            if label_name in label_addresses:
                if label_name == "40": #use of constant, makes it similar to li
                    address = label_addresses[label_name]
                    return opcodes["li"]+"00000"+rd+address
                address = label_addresses[label_name]
                return opcode + "00000" + rd + address
    elif parts[0] in ["addi", "beqz"]: #i type instructions of addi,if a==0
        rd = registers.get(parts[1])
        rs = registers.get(parts[2])
        if rd is not None and rs is not None: #addi
            imm = format(int(parts[3]), '016b')
            return opcode + rs + rd + imm
        if rs is None and parts[2] == "end_encrypt_decrypt": #beqz instuctions
            imm =  "0000000000000110" # end_encrypt_decrypt
            rs = "00000"
            return opcode+rs+rd+imm
        if rs is None and parts[2]=="end_decryption":
            imm =  "0000000000000110" # end_decryption
            rs = "00000"
            return opcode+rs+rd+imm
```

```python
    elif parts[0] in ["lb", "sb"]:
        rt = registers.get(parts[1])
        if rt is not None:
            imm = format(int(parts[2].split('(')[0]), '016b')
            rs = registers.get(parts[2].split('(')[1][:-1])
            if rs is not None:
                return opcode + rs + rt + imm
    elif parts[0] == "j": #use of jump via the jumplabel dictionary
        target_label = parts[1] #where it jumps too
        if target_label in jump_label:
            target_address = jump_label[target_label]
            #imm = format(target_address, '026b')
            return opcode + target_address # return functiom

    elif parts[0] == "xor": #r type with rd = rs^rt
        rd = registers.get(parts[1])
        rs = registers.get(parts[2])
        rt = registers.get(parts[3])
        if rd is not None and rs is not None and rt is not None:
            funct = "100110"
            return "000000" + rs + rt + rd + "00000" + funct

    elif parts[0] in ["end_encrypt_decrypt", "end_decryption"]: #
        end_opcode = opcodes.get(parts[0])
        if end_opcode is not None:
```

```python
                return end_opcode

        elif parts[0] == "syscall": #syscall always occurs at 0x0000000c
            return  "00000000000000000000000000"+opcode
        return ""

assembly_code = [
    ".text",
    "main:",
    "li $t3, 0",
    "la $a0, plaintext",
    "li $v0, 4",
    "syscall",
    "la $a0, data",
    "la $a1, 40",
    "li $v0, 8",
    "syscall",
    "la $t4, Key",
    "li $t0, 0",
    "encrypt_decrypt_loop:",
    "lb $t2, 0($a0)",
    "beqz $t2, end_encrypt_decrypt",
    "lb $t1, 0($t4)",
    "xor $t2, $t2, $t1",
    "sb $t2, 0($a0)",
```

```python
    "addi $a0, $a0, 1",
    "addi $t4, $t4, 1",
    "j encrypt_decrypt_loop",
    "end_encrypt_decrypt:",
    "la $a0, ciphertext",
    "li $v0, 4",
    "syscall",
    "la $a0, data",
    "li $v0, 4",
    "syscall",
    "la $t4, Key",
    "li $t3, 1",
    "la $a0, data",
    "j decrypt_data_loop",
    "decrypt_data_loop:",
     "lb $t2, 0($a0)",
    "beqz $t2, end_decryption",
    "lb $t1, 0($t4)",
    "xor $t2, $t2, $t1",
    "sb $t2, 0($a0)",
    "addi $a0, $a0, 1",
    "addi $t4, $t4, 1",
    "j decrypt_data_loop",
    "end_decryption:",
    "la $a0, deciphertext",
```

```python
    "li $v0, 4",
    "syscall",
    "la $a0, data",
    "li $v0, 4",
    "syscall",
    "li $v0, 10",
    "syscall",
]

machine_code = ""
for line in assembly_code:
    machine_instruction = assemble(line)
    if machine_instruction:
        machine_code += machine_instruction

# Split the machine code into groups of 32 bits (8 characters) for better readability
machine_code_groups = [machine_code[i:i + 32] for i in range(0, len(machine_code), 32)]

for group in machine_code_groups:
    print(group)
```

Instruction Set and Registers: The code defines a set of MIPS instructions in the opcodes dictionary and a set of registers in the registers dictionary. These dictionaries map mnemonic instructions and register names to their binary representations.

Label Addresses and Jump Labels: The label_addresses dictionary is used to store the addresses of various labels in the code, such as "plaintext," "data," "Key," etc. The jump_label dictionary is used to specify jump targets for certain instructions like "encrypt_decrypt_loop" and "decrypt_data_loop."

Assembly Function: The assemble function takes an assembly instruction as input and converts it into its corresponding machine code. It first splits the input instruction into its parts and checks for various conditions to determine the appropriate binary representation.

Handling Different Instruction Types: The code handles different types of instructions, including R-type, I-type, jump, syscall, and label definitions. For example, it correctly translates instructions like "li," "la," "beqz," "xor," and others into their binary representations.

Main Assembly Code: The assembly_code list contains the high-level MIPS assembly instructions for a program. The code iterates through this list, calls the assemble function for each instruction, and concatenates the resulting machine code into a single string. Finally, it splits this machine code into 32-bit groups for readability and prints the result.