Term Project:

MIPS Assembly Implementation of an Encryption Algorithm: Phase I

Ege Kaan Özalp

Sabancı University

CS401: Computer Architecture

Professor Atıl Utku Ay

May 20, 2024

1. Non-Linear Function

a. Small Tables Implementation:

```
.data
# 4 S-Boxes:
S0: .byte 0x2, 0xF, 0xC, 0x1, 0x5, 0x6, 0xA, 0xD, 0xE, 0x8, 0x3, 0x4, 0x0, 0xB, 0x9, 0x7
S1: .byte 0xF, 0x4, 0x5, 0x8, 0x9, 0x7, 0x2, 0x1, 0xA, 0x3, 0x0, 0xE, 0x6, 0xC, 0xD, 0xB
S2: .byte 0x4, 0xA, 0x1, 0x6, 0x8, 0xF, 0x7, 0xC, 0x3, 0x0, 0xE, 0xD, 0x5, 0x9, 0xB, 0x2
S3: .byte 0x7, 0xC, 0xE, 0x9, 0x2, 0x1, 0x5, 0xF, 0xB, 0x6, 0xD, 0x0, 0x4, 0x8, 0xA, 0x3

# 100 randomly generated 16-bit numbers:
X: .word 0x1F74, 0xB380, 0x7571, 0xF851, 0x143B, 0x4A7A, 0xA3CD, 0x0835, 0xFF2E, 0x9406, 0x5287, 0xA676, 0x9AED, 0xA271, 0x2C75, 0x94BC, 0x15F5, 0x

# Output message for the result of each iteration.
Message: .asciiz "The value gathered from the S-Boxes: "
Termination: .asciiz "The void S(X) function is returned."
```

Image 1.1.: Data part of the implementation.

In the data part of the program, there exists 4 S-Boxes which were given in the Term Project explanation paper and an array (i.e., X) which consists of 100 random integers that were created using Python. In addition to that, the data part includes two different strings: one for printing the result and another for printing the termination of the function S(x).

```
.text
main:
        la $a0, X               # The address of X[0] is in $a0.
        jal S                   # Execute the S(X) function. This function assumes that len(X) is 100.
        la $a0, Termination     # Load the termination message into $a0.
        li $v0, 4               # Put the system call code for printing a string into $v0.
        syscall                 # Make the system call.
        li $v0, 10              # Put the system call code for exit into $v0.
        syscall                 # Make the system call to exit the program.
```

Image 1.2.: Main of the program.

The main part of the program loads the address of the array X to $a0 and passes it as an argument to the "void S(X)" function – which prints the results to the console, therefore it is implemented as void (optional). After the void S(X) function returns, the program prints "The void S(X) function is returned to the console" and terminates.

```
S:
        add $t0, $t0, $zero # $t0 = i = 0. $t0 is the loop counter.
        add $t1, $a0, $zero # The address of X[0] is in $t1.
        j S_Loop            # Jump to the loop part of the method.
```

Image 1.3.: The entrance of the S(X) method.

Once the program jumps to the S(X) method, it first sets $t0 as the loop counter (i). Then it stores the address of X[0] in $t1. Lastly, it jumps to the loop part of S(X).

```
S_Loop:
        # Loop condition (while loop counter < 100):
        addi $t2, $zero, 100
        beq $t0, $t2, S_End

        # Access to the 16-bit number for the current iteration:
        add $t2, $t0, $t0    # $t2 = 2*$t0.
        add $t2, $t2, $t2    # $t2 = 4*$t0.
        add $t2, $t1, $t2    # $t2 = &X[i] = &X[$t0].
        lw $t2, 0($t2)       # $t2 = X[i] = X[$t0].

        # Parse the 16-bit number into 4 4-bit numbers:
        # $t2[15:12]:
        andi $t3, $t2, 0xF000 # Isolate the [15:12] bits of the number.
        srl  $t3, $t3, 12      # Shift it right to place it to the lower 4 bit.
        la $t4, S0             # Load &S0[0] to $t4.
        add $t4, $t4, $t3      # Find the address of S0[$t2[15:12]].
        lb $t3, 0($t4)         # $t3 = S0[$t2[15:12]]

        # $t2[11:8]:
        andi $t4, $t2, 0x0F00 # Isolate the [11:8] bits of the number.
        srl  $t4, $t4, 8       # Shift it right to place it to the lower 4 bit.
        la $t5, S1             # Load &S1[0] to $t5.
        add $t5, $t5, $t4      # Find the address of S1[$t2[11:8]].
        lb $t4, 0($t5)         # $t4 = S1[$t2[11:8]]

        # $t2[7:4]:
        andi $t5, $t2, 0x00F0 # Isolate the [7:4] bits of the number.
        srl  $t5, $t5, 4       # Shift it right to place it to the lower 4 bit.
        la $t6, S2             # Load &S2[0] to $t6.
        add $t6, $t6, $t5      # Find the address of S2[$t2[7:4]].
        lb $t5, 0($t6)         # $t5 = S2[$t2[7:4]]
```

Image 1.4.: The loop part of the S(X) function.

```
        # $t2[3:0]:
        andi $t6, $t2, 0x000F # Isolate the [3:0] bits of the number.
        la $t7, S3             # Load &S3[0] to $t7.
        add $t7, $t7, $t6      # Find the address of S3[$t2[3:0]].
        lb $t6, 0($t7)         # $t6 = S3[$t2[3:0]]

        # Concatenate $t3 || $t4 || $t5 || $t6:
        sll $t3, $t3, 12       # Shift $t3 left by 12 bits to place it in [15-12].
        sll $t4, $t4, 8        # Shift $t4 left by 8 bits to place it in [11-8].
        sll $t5, $t5, 4        # Shift $t5 left by 4 bits to place it in [7-4].
                               # No shifting for $t6 since it is in the correct place.
        or $t3, $t3, $t4       # $t3 = $t3 || $t4.
        or $t3, $t3, $t5       # $t3 = $t3 || $t4 || $t5.
        or $t3, $t3, $t6       # $t3 = $t3 || $t4 || $t5 || $t6.

        # Print out the result to the console.
        add $t4, $a0, $zero    # Preserve the address of X[0] which is in $a0.
        la $a0, Message        # Load address of the Message into $a0.
        li $v0, 4              # Put the system call code for printing a string into $v0.
        syscall                # Make the system call.

        add $a0, $t3, $zero    # Load concatenated integer $t3 into $a0.
        li $v0, 34             # Put the system call code for printing an integer into $v0.
        syscall                # Make the system call.

        li $a0, 10             # Load the ASCII code for newline into $a0.
        li $v0, 11             # Put the system call code for printing a character into $v0.
        syscall                # Make the system call.

        # Increase the loop counter by 1 (i++) and recover the original value of $a0:
        addi $t0, $t0, 1       # Increase the loop counter.
        add $a0, $t4, $zero    # Recover the original value of $a0.
        j S_Loop
```

Image 1.5.: The loop part of the S(X) function (cont'd).

Inside of the S_Loop part of the S(X) method, it first checks if the loop counter value equals to 100 or not (since the function assumes length of the array X is always 100) and if it is not the case, it continues to the loop. Inside of the loop, the method first gathers the i^{th} integer inside of the array X, then it parses the 16-bit number into 4 4-bit numbers (which will be used as the indices) by performing logical masking and shifts each one of the parsed values to the right-most. While doing so, it retrieves the 4-bit value from the corresponding S-Box using the parsed value (recall that parsed values represent the index). After retrieving each one of the 4-bit values, the method shifts them to the left based on the order specified in the Term Project explanation paper ($t3 || $t4 || $t5 || $t6) using “sll” instruction. For example, since the value inside $t3 will be the left-most 4-bit of the 16-bit concatenated number, it is shifted by 12 and since the $t6 will be the right-most sequence, it is not shifted at all. To concatenate the shifted values, the method performs “or” instruction: $t3 or $t4 or $t5 or $t6.

After finding the concatenated value, it first prints out the string “The value gathered from the S-Boxes:”, then prints the concatenated value and then prints a newline. Lastly, it increases the loop counter (i) by 1 and jumps back to the loop.

```
S_End:
		jr $ra

```

Image 1.6.: S_End part of the S(X) function.

Once the loop counter value (i) gets the value 100, the method jumps to the S_End part in which it just returns to the main.

b. Performance of the Small Tables Method Using Different Cache Parameters:

Max Cache Size: 8 KB = 2^{13} Bytes = 8192 Bytes.

The Found Parameters for the Minimum Miss Amount (1) Using a Full 8 KB Cache:

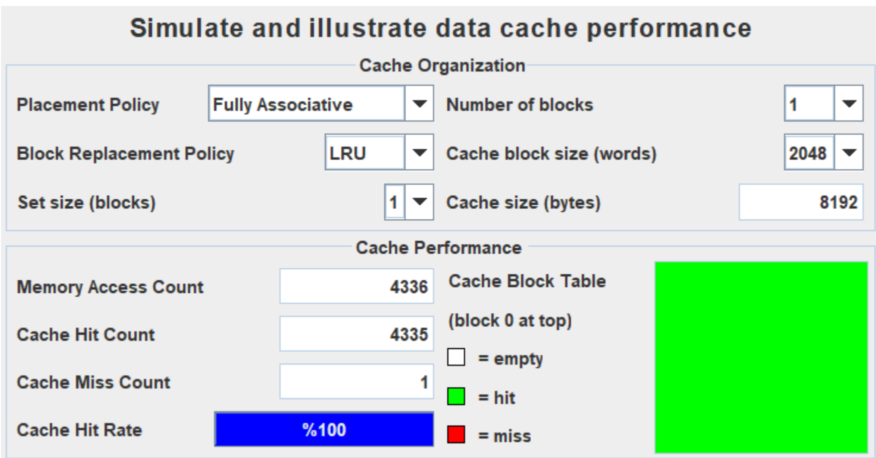

Image 1.7.: The parameters that are found which give the minimum miss rate of 1.

Simulation shows that once the number of blocks is 1 and the cache block size is 2048 words (yields to an 8 KB Cache), there exist only a single miss which is the compulsory miss. However, it is not the best option to continue with since we did not consider the memory optimization but only focused on the hit rate.

Better Parameter Selection for the Cache for Small Tables Method:

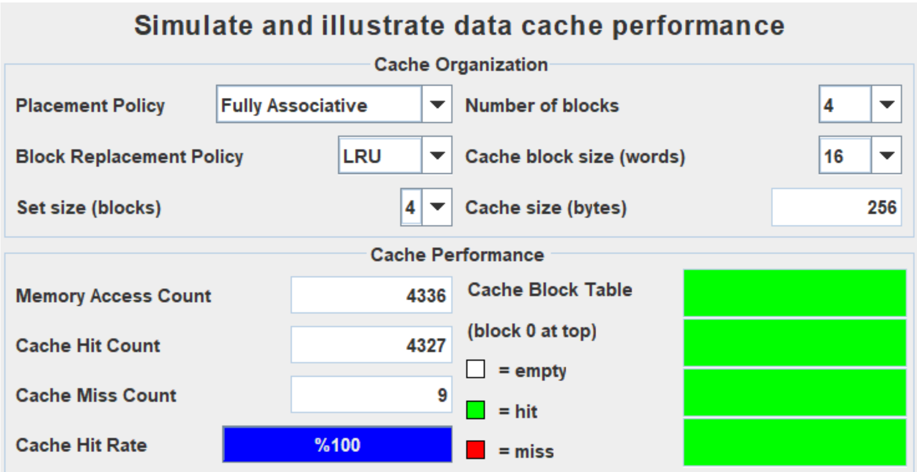

Image 1.8.: Better parameter selection (organization) for the cache.

It is shown that an 8 KB fully associative cache with LRU and a single block may provide the cache hit rate of 100% and the cache miss amount of 1. However, it is possible to keep the same cache hit rate (approximately, since we increase the cache miss for a negligible amount) while using a cache that has a size of 256 bytes (Image 1.8.). Considering the improvement in the memory (2^{13} / 2^8 = 32 times less memory space), it would be a better practice to use the cache with organization shown in Image 1.8. considering solely this S-Box program with "Small Tables Method".

c. Large Tables Implementation

The implementation of the "Large Tables Method" is merely the same with the "Small Tables Method" with some minor differences.

```
.data
# 1 S-Box Containing all other S-boxes:
S0: .byte 0x2, 0xF, 0xC, 0x1, 0x5, 0x6, 0xA, 0xD, 0xE, 0x8, 0x3, 0x4, 0x0, 0xB, 0x9, 0x7, 0xF, 0x4, 0x5, 0x8, 0x9, 0x7, 0x2, 0x1, 0xA, 0x3, 0x0, 0x
```

Image 1.9.: Single S-Box in data part.

Rather than storing 4 S-Boxes in the memory, this implementation stores 1 large table that contains all the other tables (Image 1.9.).

```
        srl  $t3, $t3, 12      # Shift it right to place it to the lower 4 bit.
        la $t4, S0             # Load &S0[0] to $t4.
        add $t4, $t4, $t3      # Find the address of S0[$t2[15:12]].
        lb $t3, 0($t4)         # $t3 = S0[$t2[15:12]]

        # $t2[11:8]:
        andi $t4, $t2, 0x0F00 # Isolate the [11:8] bits of the number.
        srl  $t4, $t4, 8       # Shift it right to place it to the lower 4 bit.
        la $t5, S0             # Load &S0[0] to $t5.
        addi $t5, $t5, 16      # $t5 = &S1[0] now.
        add $t5, $t5, $t4      # Find the address of S1[$t2[11:8]].
        lb $t4, 0($t5)         # $t4 = S1[$t2[11:8]]

        # $t2[7:4]:
        andi $t5, $t2, 0x00F0 # Isolate the [7:4] bits of the number.
        srl  $t5, $t5, 4       # Shift it right to place it to the lower 4 bit.
        la $t6, S0             # Load &S0[0] to $t6.
        addi $t6, $t6, 32      # $t6 = &S2[0] now.
        add $t6, $t6, $t5      # Find the address of S2[$t2[7:4]].
        lb $t5, 0($t6)         # $t5 = S2[$t2[7:4]]

        # $t2[3:0]:
        andi $t6, $t2, 0x000F # Isolate the [3:0] bits of the number.
        la $t7, S0             # Load &S0[0] to $t7.
        addi $t7, $t7, 48      # $t7 = &S3[0] now.
        add $t7, $t7, $t6      # Find the address of S3[$t2[3:0]].
        lb $t6, 0($t7)         # $t6 = S3[$t2[3:0]]
```

Image 1.10.: Additional “addi” instruction for offset adjustment (shown with orange arrows).

Another difference is that this implementation requires 3 additional “addi” instruction to adjust the offset for S1, S2 and S3.

d. Performance of the Large Table Method Using Different Cache Parameters:

The Found Parameters for the Minimum Miss Amount (1) Using a Full 8 KB Cache:

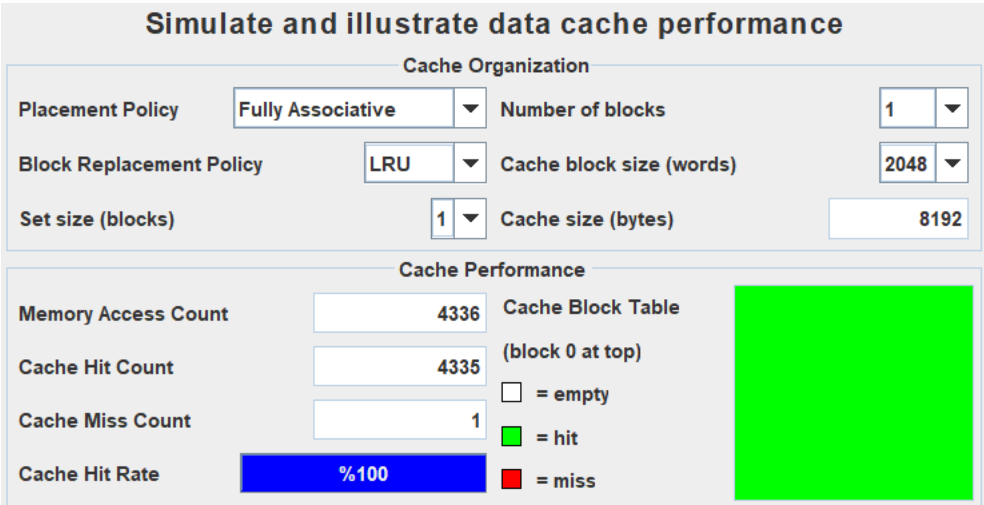

Image 1.11.: The parameters that are found which give the minimum miss rate of 1.

Better Parameter Selection for the Cache for Large Tables Method:

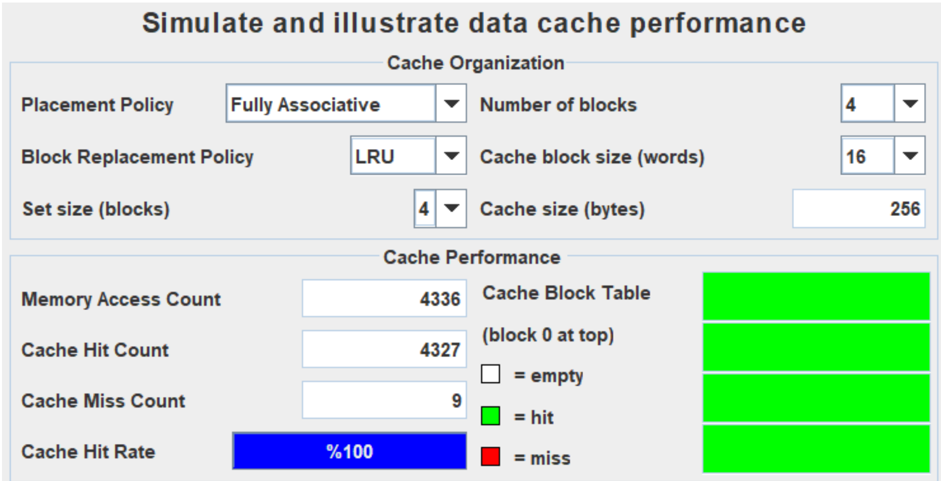

Image 1.12.: Better parameter selection (organization) for the cache.

e. **Result:**
For both the “Small Tables Method” and the “Large Tables Method”, it is shown that a cache with a single block and a block size of 8192 bytes would yield a single compulsory miss and 100% cache hit rate.

However, it is possible to use a cache size of 256 bytes (which is 32 times less than the first option) and sustain approximately the same cache hit rate of 100% (with 9 misses). Hence, using a cache with the size of 256 bytes would be the better option to optimize both the cache hit rate and memory space for this specific S-Box method.

2. Linear Function

a. Implementation:

```
.data
# A random 16-bit X value:
X: .word 0xbbaa
Result_Message: .asciiz "The result is: "

```

Image 2.1.: The data part of the implementation.

In the data part of the program, there exist a 16-bit X value and a string to use while printing the result.

```
.text
main:
        lw $a0, X               # $a0 = X
        jal L
        add $t0, $v0, $zero

        # Print the result message:
        la $a0, Result_Message
        li $v0, 4
        syscall

        # Print the result itself:
        add $a0, $t0, $zero
        li $v0, 34
        syscall

        # Terminate the program:
        li $v0, 10
        syscall

```

Image 2.2.: The main part of the program.

The program first loads the 16-bit number X as an argument and passes it to the function “int L(X)”. Then once the “int L(X)” function returns, the main prints the result message and the result. Lastly, it terminates the program since the program is finished.

```
L:
        add $t0, $a0, $zero    # $t0 = $a0 = X

        # <<< (Circular Left):
        andi $t1, $t0, 0xFC00 # Find the first 6 bits by masking.
        srl $t1, $t1, 10       # Send them to the end.
        andi $t2, $t0, 0x03FF # Find the last 10 bits.
        sll $t2, $t2, 6        # Send them to the start.
        or $t3, $t1, $t2       # Find the circular left result.

        # >>> (Circular right):
        andi $t1, $t0, 0x003F # Find the last 6 bits.
        sll $t1, $t1, 10       # Send them to the start.
        andi $t2, $t0, 0xFFC0 # Find the first 10 bits.
        srl $t2, $t2, 6        # Send them to the end.
        or $t4, $t1, $t2       # Find the circular right result.

        # XOR Operations:
        xor $t0, $t0, $t3      # $t0 = X ⊕ (X <<< 6).
        xor $t0, $t0, $t4      # $t0 = X ⊕ (X <<< 6) ⊕ (X >>> 6).

        # Return the value:
        add $v0, $t0, $zero    # Return $t0.
        jr $ra
```

Image 2.3.: The L(X) function.

The L(X) method first sets $t0 as X. Then it applies the circular left logic. The logic is simple: Find the first 6 bits of X by masking, shift these bits to the right-most (which is stored in $t1), find the last 10 bits by masking again and send them to the left-most (which is stored in $t2). Then, store “$t1 or $t2” in $t3 – by doing so, $t3 now stores X <<< 6.

The method applies a similar logic for circular right: Find the last 6 bits by masking, shift them to the left-most (which is stored in $t1), find the last 10 bits by masking X again, shift them to the right-most (which is stored in $2). Then, store “$t1 or $t2” in $t4 – by doing so, $t4 now stores X >>> 6.

The method then performs X ⊕ (X <<< 6) ⊕ (X >>> 6) and returns it to the main via $v0.

b. Sample Run:

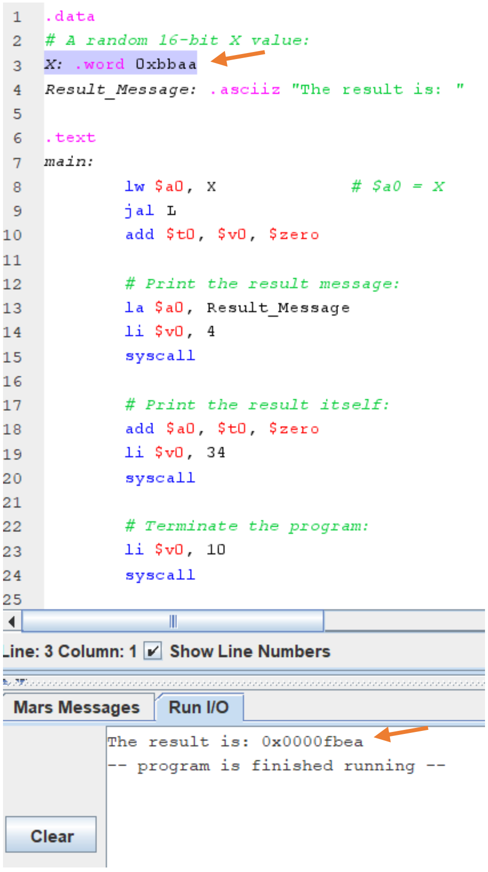

Image 2.4.: The method returns 0xfbea when X is 0xbbaa.

```
.data
# A random 16-bit X value:
X: .word 0x1111
Result_Message: .asciiz "The result is: "

.text
main:
        lw $a0, X               # $a0 = X
        jal L
        add $t0, $v0, $zero

        # Print the result message:
        la $a0, Result_Message
        li $v0, 4
        syscall

        # Print the result itself:
        add $a0, $t0, $zero
        li $v0, 34
        syscall

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

Line: 3 Column: 16 ☑ Show Line Numbers

Mars Messages | Run I/O

```
The result is: 0x00001111
-- program is finished running --
```

Clear

Image 2.5.: The method returns 0x1111 when X is 0x1111.

3. Permutation:

a. Implementation:

```
.data
# Random Number (8-bit):
Num: .word 0xD6

# Result Message:
Result_Message: .asciiz "The result is: "
```

Image 3.1.: Data part of the implementation.

The data part of the program stores a random 8-bit number and a string for printing the result.

```
.text
main:
        #Load Num1:
        lw $a0, Num
        jal P

        # Store the result of p(x) for Num1:
        add $t0, $v0, $zero

        # Print "The result is: ":
        la $a0, Result_Message
        li $v0, 4
        syscall

        # Print the result:
        add $a0, $t0, $zero  # Recover the result of p(x)
        li $v0, 34
        syscall

        # Exit from the program.
        li $v0, 10
        syscall
```

Image 3.2.: Main part of the implementation.

The main part of the code first loads the Num value and passes it to the “int p(x)” function as an argument. Once the function returns, it then stores the returned value in $t0 to print the result message. After that, it recovers the returned value from $t0 and prints it (using hexadecimal representation). Lastly, it terminates the program since the program is finished.

```
P:
        add $t0, $a0, $zero  # Assign $a0 to $t0.

        # Access to the 0th bit of the "Num":
        andi $t1, $t0, 0x80  # Access to the 0th bit.
        srl $t1, $t1, 5      # Send the 0th bit to the 5th bit.

        # Access to the 1st bit of the "Num":
        andi $t2, $t0, 0x40  # Access to the 1st bit.
        srl $t2, $t2, 6      # Send the 1st bit to the 7th bit.

        # Access to the 2nd bit of the "Num":
        andi $t3, $t0, 0x20  # Access to the 2nd bit.
        srl $t3, $t3, 1      # Send the 2nd bit to the 3rd bit.

        # Access to the 3rd bit of the "Num":
        andi $t4, $t0, 0x10  # Access to the 3rd bit.
        srl $t4, $t4, 1      # Send the 3rd bit to the 4th bit.

        # Access to the 4th bit of the "Num":
        andi $t5, $t0, 0x08  # Access to the 4th bit.
        sll $t5, $t5, 2      # Send the 4th bit to the 2nd bit.

        # Access to the 5th bit of the "Num":
        andi $t6, $t0, 0x04  # Access to the 5th bit.
```

Image 3.3.: The implementation of the p(x) function.

```
        srl $t6, $t6, 1       # Send the 5th bit to the 6th bit.

        # Access to the 6th bit of the "Num":
        andi $t7, $t0, 0x02  # Access to the 6th bit.
        sll $t7, $t7, 5      # Send the 6th bit to the 1st bit.

        # Access to the 7th bit of the "Num":
        andi $t8, $t0, 0x01  # Access to the 7th bit.
        sll $t8, $t8, 7      # Send the 7th bit to the 0th bit.

        or $v0, $t1, $t2
        or $v0, $v0, $t3
        or $v0, $v0, $t4
        or $v0, $v0, $t5
        or $v0, $v0, $t6
        or $v0, $v0, $t7
        or $v0, $v0, $t8

        jr $ra
```

Image 3.4.: The implementation of the p(x) function (cont'd).

Since the permutation logic is fixed, I did not put any permutation table to the data part of the program. Rather, I implemented it inside of the p(x) method. The basic logic of the function is that it accesses to the i^{th} bit of the Num value by utilizing masking and then shifts either left or right based on the value specified in the permutation table in Term Project explanation paper. It then puts all the reordered bits together to create an 8-bit number by using "or" instructions and returns the reordered value.

b. Sample Runs:

```
.data
# Random Number (8-bit):
Num: .word 0xD6

# Result Message:
Result_Message: .asciiz "The result is: "

.text
main:
        #Load Num1:
        lw $a0, Num
        jal P

        # Store the result of p(x) for Num1:
        add $t0, $v0, $zero

        # Print "The result is: ":
        la $a0, Result_Message
        li $v0, 4
        syscall

        # Print the result:
        add $a0, $t0, $zero  # Recover the result of p(x)
        li $v0, 34
        syscall
```

Line: 3 Column: 1 Show Line Numbers

Mars Messages | Run I/O

```
The result is: 0x0000004f
-- program is finished running --
```

Clear

Image 3.5.: The method returns 0x4F when Num is 0xD6.

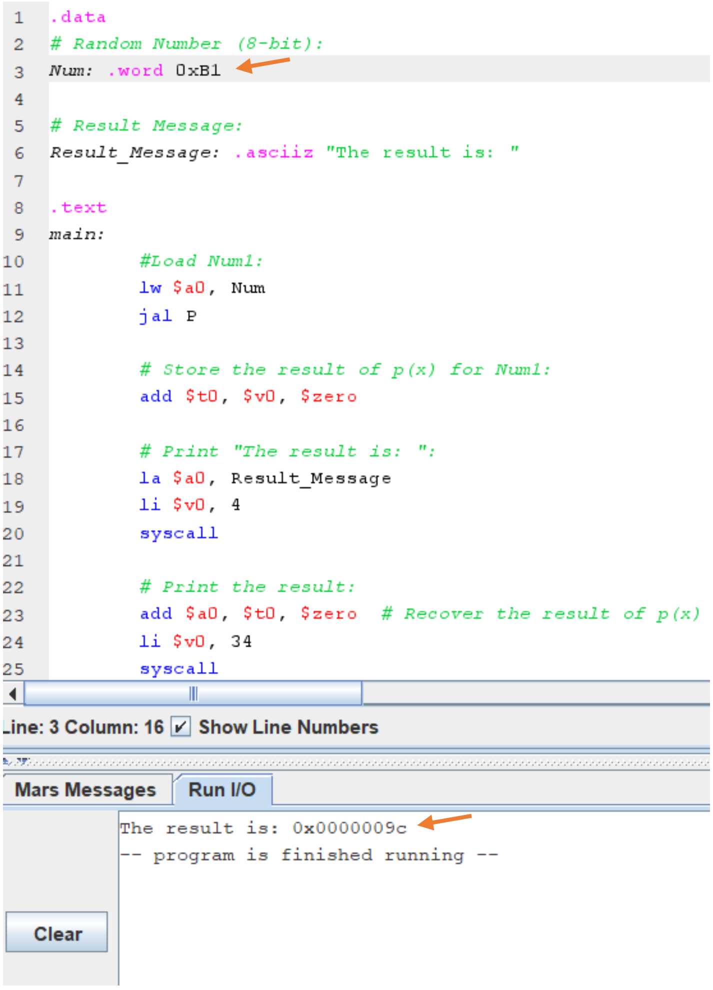

Image 3.6.: The method returns 0x9C when Num is 0xB1.