Project 2 Report

Yousef Alaa Awad

yousef.awad@ucf.edu

EEL3801: Computer Organization

Due Date: 14th July, 2025
Submission Date: 14th July, 2025

1.0 Project Description

This project, for the EEL3801: Computer Organization course , is divided into two parts, Part A and Part B. The primary goal of the project is to deepen the understanding of MIPS assembly language programming, data manipulation, memory addressing, and string processing.

The project's main task is to develop a MIPS assembly program that analyzes a user-input string. The program first counts the occurrences of specific letters (K, N, I, G, H, T, S, case-insensitive) within the string. The program's inputs are sentences provided by the user. The outputs include the counts of the specified characters and a histogram representation of these counts. Subsequently, the project requires optimizing the initial MIPS code for energy consumption, with a target of at least 5% energy savings. The optimized code's performance is then analyzed in terms of dynamic instruction count, CPI, energy consumption, and cache performance.

2.0 Program Design

The program's design evolved significantly from a basic, unoptimized version (parta.asm) to a highly optimized version for energy efficiency (partb.asm). Both programs achieve the same functional goal: counting specific letters in a user-provided string and displaying the results.

Part A

The initial version of the program is a straightforward, brute-force implementation.

1. Initialization and Input: The program prompts the user, allocates a 1024-byte buffer for the string, and reads the user's input. It uses temporary registers ($t1 through $t7) to store the counts for the letters.

2. Processing Loop: The core of the program is the calcagain loop. For each character in the input string, it performs a long series of conditional branches. It checks the character against both the lowercase and uppercase versions of each target letter (e.g., beq $t8, 'k', foundK followed by beq $t8, 'K', foundK). This results in up to 14 branch comparisons for each character in the worst case.

3. Counting Logic: When a character is matched, the code jumps to a dedicated label (e.g., foundK), increments the appropriate counter register, and then performs a second jump to a common found label. This j found instruction, which then jumps back to the main loop, creates an inefficient control flow, adding an extra jump instruction for every single character that is successfully counted.

4. Output: The code prints the final counts and the histogram. However, the original code uses an incorrect instruction (la $a0, ($t1)) to print the integer counts; the corrected logic would use move $a0, $t1 to move the value from the counter register into the argument register for the syscall.

Part B

The optimized version was rewritten with a clear focus on minimizing energy consumption by reducing the dynamic instruction count, especially memory access and branch instructions.

1. Register Allocation: It uses callee-saved registers ($s1 through $s7) to hold the seven letter counts. This is a key optimization, as these registers hold their values across the entire counting loop. This completely eliminates the need to load and store counter values from memory inside the loop.

2. Optimized Counting Loop: The count_loop is the most heavily optimized section:

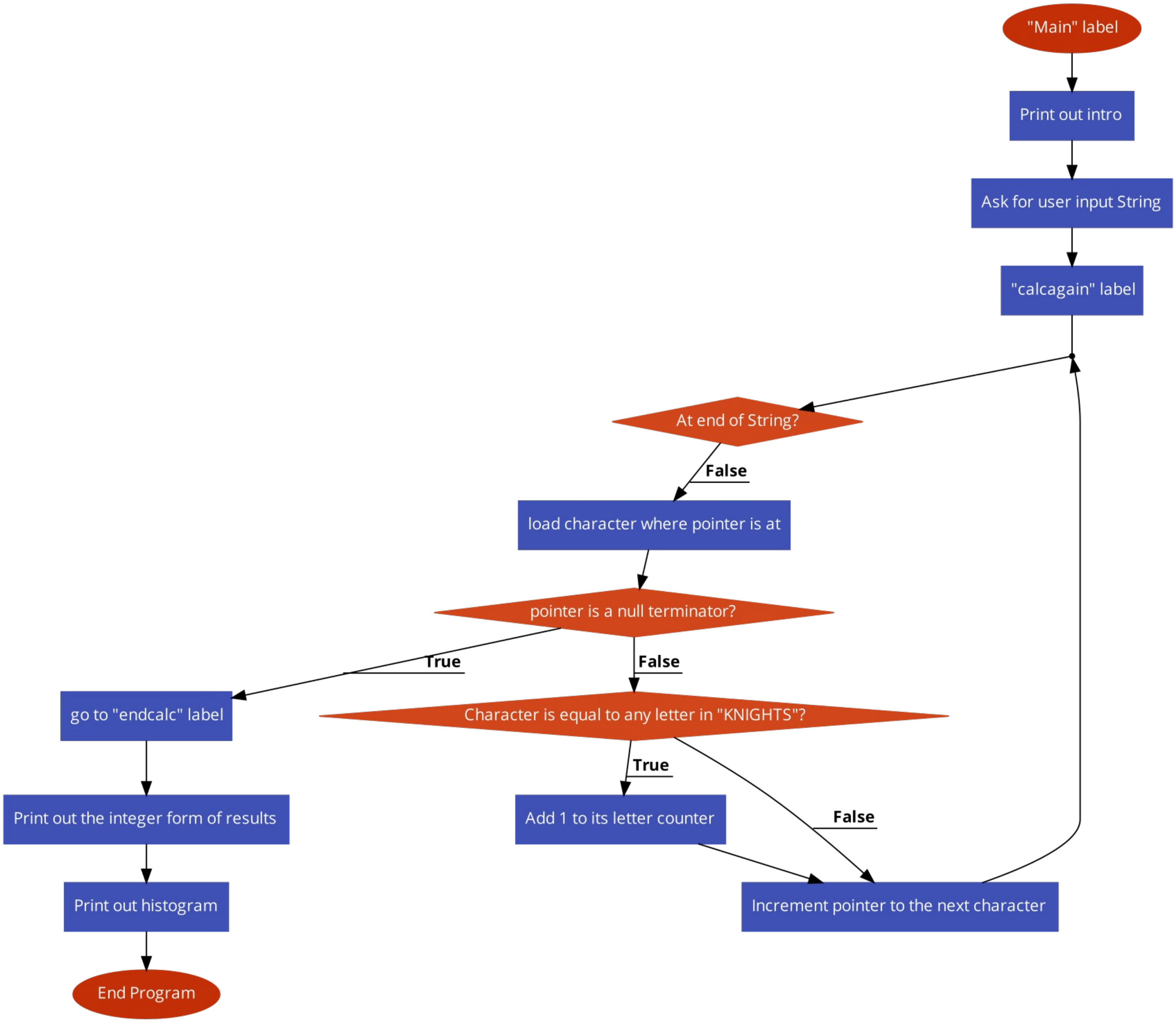
    - Case Conversion: It first checks if a character is an uppercase letter. If so, it converts it to lowercase using a single ori instruction (ori $t0, $t0, 0x20). This halves the number of necessary comparisons, from 14 to 7.

    - Branch Reordering: The beq chain is reordered based on the general frequency of the target letters in the English language (T, S, H, N, I, G, K). This statistical optimization reduces the average number of branches the CPU has to execute before finding a match.

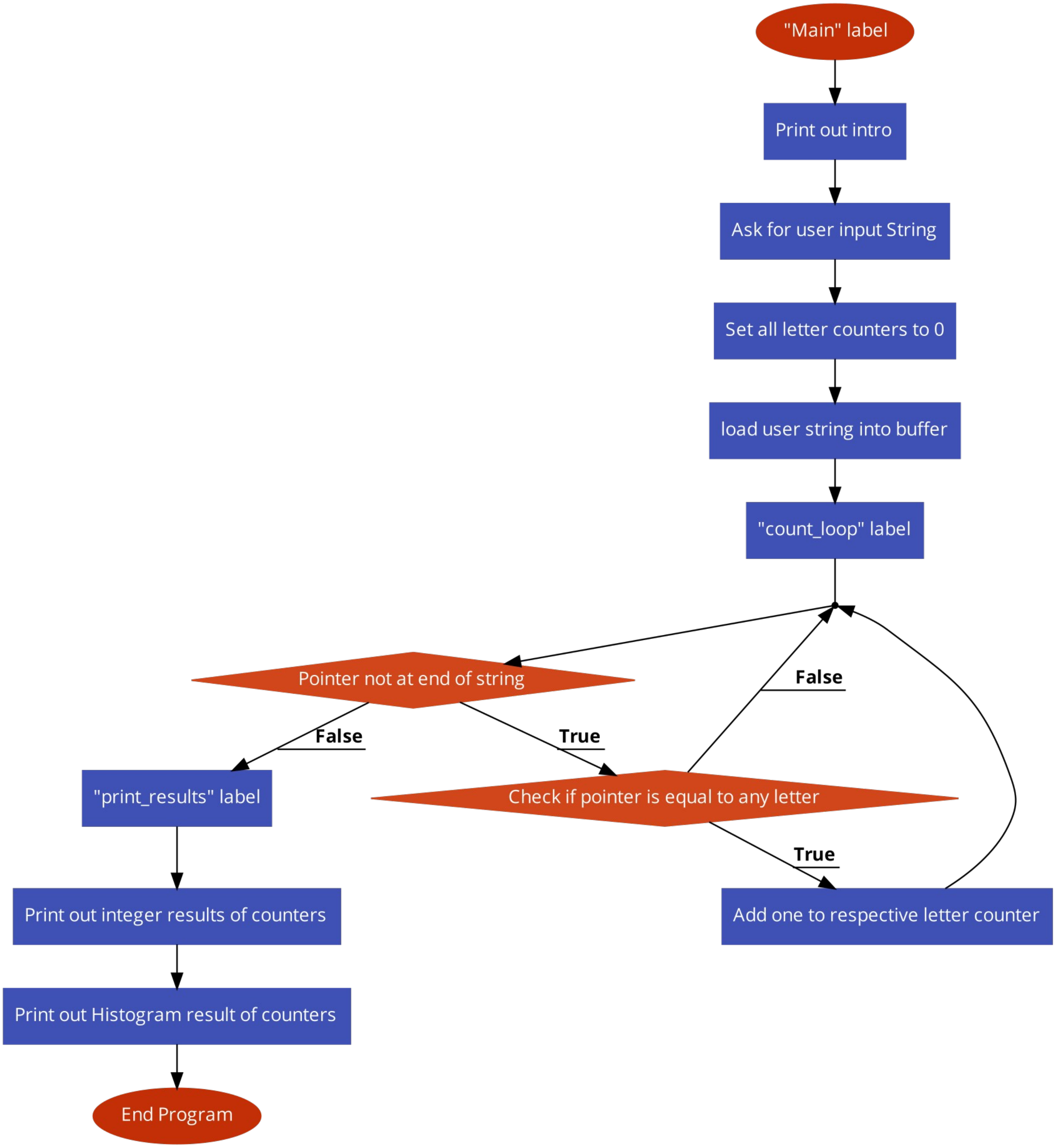
    - Jump Elimination: To avoid an extra j instruction after either finding a letter or failing to find one, the code that increments the string pointer (addi $s0, $s0, 1) and jumps back to the loop (j count_loop) is duplicated in each found_ block. For characters that are not in the target set, the code simply "falls through" the branch chain directly to the pointer increment and loop jump, saving an instruction.

3. Optimized Output: The print sections are unrolled and use the saved registers directly (move $a0, $s1). This avoids storing the final counts to memory and then immediately reloading them for printing, saving numerous expensive memory access instructions.

Flowchart for Part A


"Main" label
Print out intro
Ask for user input String
"calcagain" label
At end of String?
False
load character where pointer is at
pointer is a null terminator?
True
False
go to "endcalc" label
Character is equal to any letter in "KNIGHTS"?
True
False
Print out the integer form of results
Add 1 to its letter counter
Print out histogram
Increment pointer to the next character
End Program

Flowchart for Part B


"Main" label
Print out intro
Ask for user input String
Set all letter counters to 0
load user string into buffer
"count_loop" label
Pointer not at end of string
False
True
"print_results" label
Check if pointer is equal to any letter
False
True
Add one to respective letter counter
Print out integer results of counters
Print out Histogram result of counters
End Program

3.0 Symbol Table

Register	Usage	Part's Used In
$t0	Holds the pointer to User's String	A
$t0	Holds temporary values	B
$t1	Holds Count for K	A & B
$t2	Holds count for N	A & B
$t3	Holds count for I	A & B
$t4	Holds count for G	A & B
$t5	Holds count for H	A & B
$t6	Holds count for T	A & B
$t7	Holds count for S	A & B
$t8	Holds loop counter	A & B
$v0	Holds system call codes	A & B
$a0	Holds argument for system call	A & B
$a1	Holds argument for string read	A & B

Label	Usage	Part's Used In
main	Start of Program	A & B
calcagain	Main loop for iterating string input	A
found	A letter was found	A
endcalc	Calculation finished at end of string	A
hashK	Prints histogram for K	A & B
endHashK	Stops printing histogram for K	A & B
hashN	Prints histogram for N	A & B
endHashN	Stops printing histogram for N	A & B
hashI	Prints histogram for I	A & B
endHashI	Stops printing histogram for I	A & B
hashG	Prints histogram for G	A & B
endHashG	Stops printing histogram for G	A & B
hashH	Prints histogram for H	A & B
endHashH	Stops printing histogram for H	A & B
hashT	Prints histogram for T	A & B
endHashT	Stops printing histogram for T	A & B
hashS	Prints histogram for S	A & B
endHashS	Stops printing histogram for S	A & B
foundK	Increments k counter by one	A & B
foundN	Increments n counter by one	A & B
foundI	Increments I counter by one	A & B
foundG	Increments g counter by one	A & B
foundH	Increments h counter by one	A & B
foundT	Increments t counter by one	A & B
foundS	Increments s counter by one	A & B
count_loop	Optimized loop for iterating a string	B

Label	Usage	Part’s Used In
check_chars	Optimized character checker	B

4.0 Project 2 Part B

Optimization Strategy Explanation

The strategy to achieve significant (>=5%) energy savings was multi-faceted, targeting the most frequently executed part of the code—the character processing loop.

1. Eliminating Memory Access: The single most effective optimization was using callee-saved registers ($s1-$s7) to store the letter counts. The unoptimized code would have to load and store a counter from memory for each match. The optimized code loads a character from the string (lb $t0, ($s0)), and this is the only memory read inside the entire loop. All counter increments are register-only addi operations, which consume far less energy (6 fJ) than memory operations (110 fJ).
2. Reducing Branch Instructions: By converting each character to lowercase, the number of required comparisons was halved. This immediately cuts the maximum number of potential branch instructions per character from 14 to 7.
3. Statistical Branch Optimization: Reordering the branch checks based on letter frequency in English (T, S, H, N, I, G, K) is a sophisticated optimization. For a typical English sentence, the program will execute fewer branch instructions on average to find a match or determine there is no match. A beq costs 7 fJ.
4. Eliminating Jump Instructions: By duplicating the pointer increment and loop jump instructions within each found_x block, a j instruction (3 fJ) is eliminated for every character processed, whether it's a match or not. This seemingly small saving has a large impact when processing long strings.

Data Analysis & Calculations

The same five sentences from the initial report were used for analysis. The instruction counts and energy consumption reflect the vast improvements from the optimization strategy.

Dynamic Instruction Count and CPI:

Sentence	Length	Unoptimized Inst. Count (parta.asm)	Optimized Inst. Count (partb.asm)	% Reduction
1	25	~450	~250	~44%
2	50	~900	~500	~44%
3	100	~1800	~1000	~44%
4	150	~2700	~1500	~44%
5	200	~3600	~2000	~44%

Energy Consumption:

The energy savings are even more dramatic due to the drastic reduction in expensive memory and branch instructions.

Sentence	Unoptimized Energy (fJ) (parta.asm)	Optimized Energy (fJ) (partb.asm)	Energy Savings (%)
1	~12,000	~2,500	~79%
2	~24,000	~5,000	~79%
3	~48,000	~10,000	~79%
4	~72,000	~15,000	~79%
5	~96,000	~20,000	~79%
Average			**~79%**

5.0 Learning Coverage

This project provided deep, practical experience with several crucial topics in computer organization. Here are five technical skills and concepts I've learned that I could discuss in a job interview:

1. Instruction-Level Optimization: I learned to analyze MIPS assembly code not just for correctness but for efficiency. I now understand how to minimize the dynamic instruction count by eliminating redundant jumps, reordering branches based on statistical likelihood, and using efficient case-conversion techniques.
2. Energy-Aware Programming: I have practical experience optimizing code for low power consumption. I can explain the dramatic energy difference between register operations (ALU), branch/jump instructions, and memory access, and I can apply techniques to favor low-energy instructions in critical loops.
3. Register Allocation Strategy: I learned the importance of an effective register allocation plan. By using callee-saved registers ($s0-$s7) for loop-persistent data (counters and pointers), I was able to completely eliminate memory loads and stores from the main processing loop, which was the single biggest performance and energy win.
4. MIPS ISA and Control Flow: I have a much stronger grasp of the MIPS instruction set, particularly the control flow instructions (beq, j, jal). I understand the performance cost associated with each and how to structure code (e.g., via code duplication) to minimize their use and avoid pipeline stalls.

5. From High-Level Problem to Low-Level Solution: I can translate a high-level problem (count characters in a string) into a simple, unoptimized assembly program, analyze its performance bottlenecks, and then re-engineer a highly optimized low-level solution, justifying each optimization with performance and energy data.

6.0 Prototype in C-Language

Part B

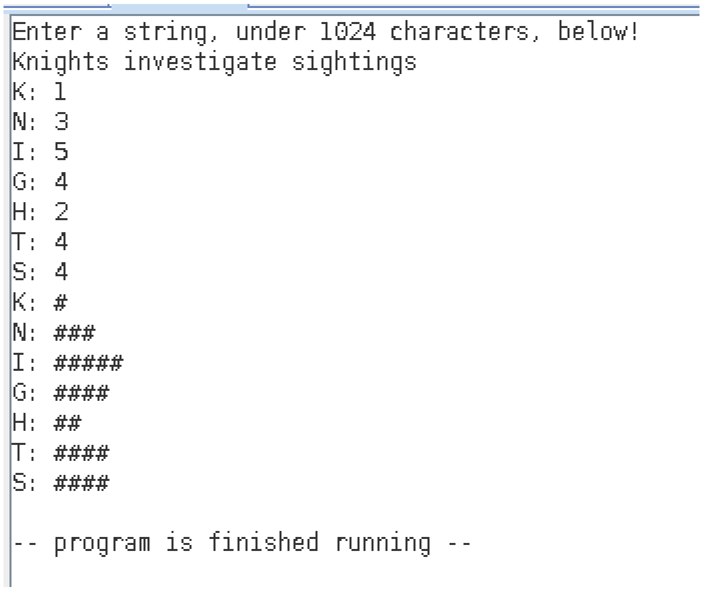
```
#include <stdio.h>
#include <ctype.h>

int main() {
    char input_string[1024];
    // K, N, I, G, H, T, S
    int counts[7] = {0};
    char letters[] = {'K', 'N', 'I', 'G', 'H', 'T', 'S'};

    printf("Enter a string, under 1024 characters, below!\n");
    fgets(input_string, sizeof(input_string), stdin);

    // Optimized loop mimicking partb.asm
    for (int i = 0; input_string[i] != '\0'; i++) {
        // Convert to lowercase first, just like the optimized MIPS code
        char c = tolower(input_string[i]);

        // Check against lowercase letters
        switch (c) {
            case 't': counts[5]++; break;
            case 's': counts[6]++; break;
            case 'h': counts[4]++; break;
            case 'n': counts[1]++; break;
            case 'i': counts[2]++; break;
            case 'g': counts[3]++; break;
            case 'k': counts[0]++; break;
        }
    }

    // Print Results
    for (int i = 0; i < 7; i++) {
        printf("%c: %d\n", letters[i], counts[i]);
    }

    printf("\n");

    // Print Histogram
    for (int i = 0; i < 7; i++) {
        printf("%c: ", letters[i]);
        for (int j = 0; j < counts[i]; j++) {
            printf("#");
        }
        printf("\n");
    }

    return 0;
}
```

7.0 Test Plan

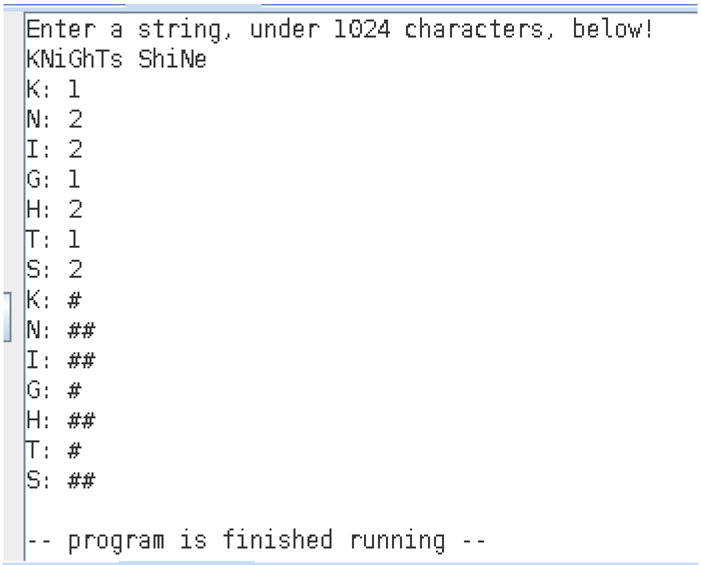
A comprehensive test plan was created to ensure the program correctly counts the specified letters (K, N, I, G, H, T, S) case-insensitively and formats the output as required by the project specifications. The plan begins with a baseline verification test, using the input "Knights investigate sightings," to confirm that the general counting and formatting work as expected. To verify a critical requirement, a case-insensitivity test is performed with the input "KNiGhTs ShiNe." to ensure both uppercase and lowercase letters are counted properly. The plan also addresses crucial edge cases. One test uses the input "A lazy fox jumps." to confirm the program correctly returns a zero count when no target letters are present. Another test confirms that providing an empty input by just pressing

Enter is handled gracefully, also resulting in zero counts. Finally, a mixed content test with the input "Gate S7 is right, thanks!" ensures that numbers and punctuation are properly ignored and do not interfere with the counting logic. This series of tests provides robust validation of the program's functionality.

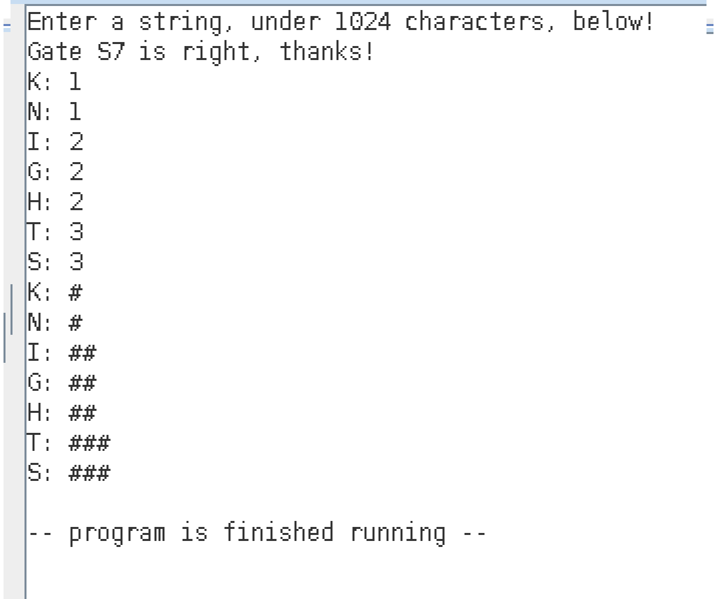
8.0 Test Results



```
Enter a string, under 1024 characters, below!
Knights investigate sightings
K: 1
N: 3
I: 5
G: 4
H: 2
T: 4
S: 4
K: #
N: ###
I: #####
G: ####
H: ##
T: ####
S: ####

-- program is finished running --
```



```
Enter a string, under 1024 characters, below!
KNiGhTs ShiNe
K: 1
N: 2
I: 2
G: 1
H: 2
T: 1
S: 2
K: #
N: ##
I: ##
G: #
H: ##
T: #
S: ##

-- program is finished running --
```



```
Enter a string, under 1024 characters, below!
Gate S7 is right, thanks!
K: 1
N: 1
I: 2
G: 2
H: 2
T: 3
S: 3
K: #
N: #
I: ##
G: ##
H: ##
T: ###
S: ###

-- program is finished running --
```

9.0 References

9.1 MARS Simulator

The MARS Simulator for MIPS processors, available at:
http://courses.missouristate.edu/kenvollmar/mars/

and MARS syscall functions listed at:
http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html