Energy consumption of Assembly Language Code in MIPS using Deep Belief Networks

Helen Diaz
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816-2362

***Abstract*—The scope of this paper is to examine the project design of an assembly language program used to count the number of occurrences of a word inputted by a user in a given statement, and that also outputs the indexes of the matches found. The total energy consumed by this program is also calculated by using the energy consumed for its branch instruction from four different Deep Belief Networks circuits designs. The four final results are compared to determine the total energy consumption of the best design, which is the design with the least amount of energy consumed, Design 1.**

***Keywords*—Deep Belief Network, Restricted Boltzmann Machines, Probabilistic Generative Models, Dynamic Instructions, Energy Consumption, Syscalls, Assembler Directives, MIPS.**

I. Introduction

String manipulation has become an important functionality that allows the user to perform more tasks on saved data. Over the years it has proven to be a useful tool to parse through data, compare it, and extract information from it.

A program was written to find the number of occurrences of a word in a given statement. Furthermore, the index of where each word match is found within the statement should be returned to the user, taking the first word of the text as index '1'.

For this program, a sample statement was hardcoded, from which the user was prompted to input one or two different words -with less than ten characters each. The program should output an integer number describing how many times the word(s) was used in the statement. Additionally, the indexes of the matches found should be outputted to the screen.

This task was accomplished by applying selected MIPS Assembly language instructions, assembler directives and system calls sufficient enough to handle string manipulation tasks.

A. Project Design

The program design for counting the number of occurrences of the word(s) inputted by the user was based on a main routing with loops and branch statements. The code was hardcoded to have an initial statement, which was stored as a string and then copied to a register. The first step was to

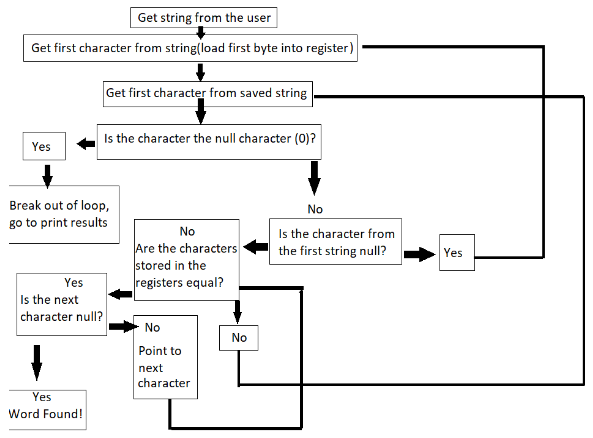
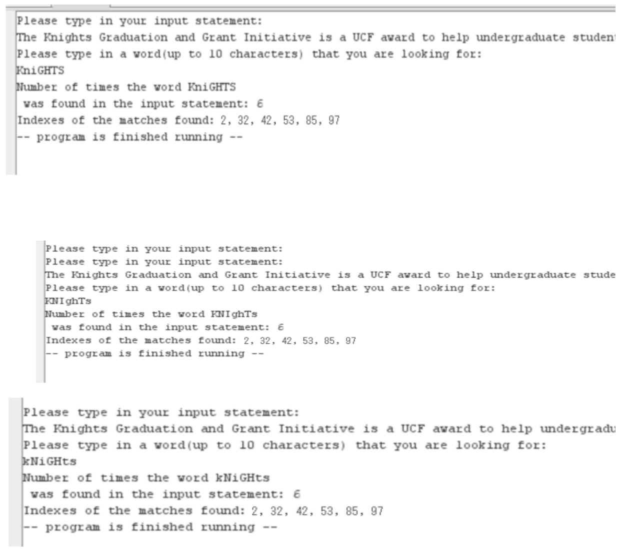

Get string from the user
Get first character from string(load first byte into register)
Get first character from saved string
Is the character the null character (0)?
Yes
Break out of loop, go to print results
No
Is the character from the first string null?
Yes
No
Are the characters stored in the registers equal?
Yes
Is the next character null?
No
Point to next character
No
Yes
Word Found!

Fig.1: Flowchart of the assembly program.



```
Please type in your input statement:
The Knights Graduation and Grant Initiative is a UCF award to help undergraduate studen
Please type in a word(up to 10 characters) that you are looking for:
KniGHTS
Number of times the word KniGHTS
 was found in the input statement: 6
Indexes of the matches found: 2, 32, 42, 53, 85, 97
-- program is finished running --
```

```
Please type in your input statement:
Please type in your input statement:
The Knights Graduation and Grant Initiative is a UCF award to help undergraduate stude
Please type in a word(up to 10 characters) that you are looking for:
KNIghTs
Number of times the word KNIghTs
 was found in the input statement: 6
Indexes of the matches found: 2, 32, 42, 53, 85, 97
-- program is finished running --
```

```
Please type in your input statement:
The Knights Graduation and Grant Initiative is a UCF award to help undergradu
Please type in a word(up to 10 characters) that you are looking for:
kNiGHts
Number of times the word kNiGHts
 was found in the input statement: 6
Indexes of the matches found: 2, 32, 42, 53, 85, 97
-- program is finished running --
```

Fig.2: Sample outputs of the assembly program.

successfully read a string with the desired words from the user, which was achieved by applying the corresponding syscall. After that, the string was copied into a different register so operations could be performed, and the information wouldn't be lost. Because the main goal of the project was to count the amount of times that the desired word appeared within the string, lb (load byte) instruction was used to analyze character by character.

This process was contained inside of the main loop of the program: after the user typed in the string, the first byte was loaded and this procedure would continue until the null character was found (use a branch statement for this, when "null" break out of the loop). Once inside the loop and through the use of multiple branch statements, the byte being analyzed was compared to the characters of the initial statement which were stored in a different register. If it each character from the string the user inputted matched the characters stored in the original string, it would branch to a label where a register, holding the counting number for the match word, would be incremented and it would also increment the register holding the string by 1, pointing to the next byte. If the character of the inputted string was not equal to the character from the saved string, then it would increment the register holding the string by 1, pointing to the next byte. After reaching the null character, the appropriate syscalls were made for printing the desired word(s) with its number of occurrences. Per project's instructions, the code should not be case sensitive, therefore additional branch statements were added to convert all the characters to upper case before the comparison.

To find the indexes at which the found words were located an array was created by using '.word'. Each word from the initial hardcoded text received an index and a flag was used to mark the index of a word every time the match word was found. The flowchart for this code is found in Figure 1.

B. *Test Cases*

To properly test this code, different combinations of the word "Knights" have been used. These three cases have been selected because they provide enough variation to ensure that the code is working properly and to confirm that the program is not case sensitive, as it should. Test cases and expected output are listed in Figure 2.

II. DBN Circuit

Deep Belief Networks are probabilistic generative models [4] consisting of undirected and directed layers of latent variables. These layers contain stacked Restricted Boltzmann Machines (RBMs) that learn through pre-training of the network and backpropagation mechanisms. The hidden layer of one RBM is the visible layer of succeeding layer. Each layer learns its entire input.

To describe its structure an analogy of a person opening his eyes can be used. When a person, that has been with his eyes closed in a dark room for a long period of time, steps outside and opens his eyes, the image does not come all at once, instead his eyes slowly adapt to the new environment to form the whole picture. Deep Belief Networks are trained a similar way; the hidden layer of the first RBM becomes the visible layer for the second one, which is trained using the outputs from the first one. This method continues until all the layers are trained which further allows pattern recognition in the data. Deep Belief Networks (DBN) have been used in an array of research areas: identifying hand-written digits, reducing the dimensionality of data and creating motion capture data [3].

An example of a DBN could be a 784×200×10 DBN for MNIST pattern recognition tasks. Its inputs could be digits, and its outputs, voltages of the neurons where each neuron represents an output class [1].

III. Results and Discussion

In this section, the energy consumption of the assembly program introduced in *Project Design* is calculated using the below energy consumption per instruction values:

1. *ALU = 2 pJ*
2. *Branch = Refer to Table I*
3. *Jump = 4 pJ*
4. *Memory = 100 pJ*
5. *Other = 5 pJ*

It is assumed that the branch prediction is implemented using DBN circuit. In this paper, the energy consumption values reported in references [1-4] are used to calculate the total energy consumption of the assembly program. The below table lists the required energy consumption to perform each branch prediction based on the different technologies proposed in [1-4].

Table I: Energy consumption for a branch instruction in the designs provided in [1-4].

Design	Energy Consumption For Each Branch Instruction
[1]	0.2 pJ
[2]	$0.03e^{+5}$ pJ
[3]	$6e^{+5}$ pJ
[4]	$5e^{+5}$ pJ

Table II contains the total energy consumption for the assembly program using designs provided in [1-4]. This table uses the energy values mentioned in Table I and the dynamic instruction count obtained in the MIPS Instructions (MARS4.5➔Tools➔Instruction Statistics, Figure 3). To calculate the total energy, the number of different types instructions is multiplied by its corresponding energy consumption per instruction value and all the values are added.

Design [1]

- Energy = (2 x 2972) + (0.2 x 2683) + (4 x 1306) + (100 x 797) + (5 x 11) = 91.5 nJ

Design [2]

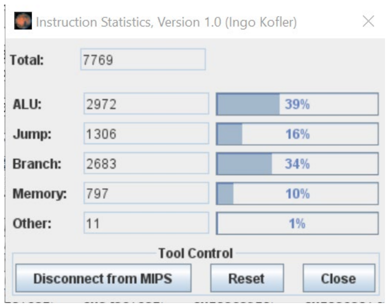
- Energy = (2 x 2972) + ($0.03e^{5}$ x 2683) + (4 x 1306) + (100 x 797) + (5 x 11) = $8.1e^{+3}$ nJ

Design [3]

- Energy = (2 x 2972) + (6e^5 x 2683) + (4 x 1306) + (100 x 797) + (5 x 11) = 1.6e^{+6} nJ

Design [4]

- Energy = (2 x 2972) + (5e^5 x 2683) + (4 x 1306) + (100 x 797) + (5 x 11) = 1.3e^{+6} nJ



Total:	7769	
ALU:	2972	39%
Jump:	1306	16%
Branch:	2683	34%
Memory:	797	10%
Other:	11	1%

Fig 3: Instruction statistics from Assembly language program

Table II: Total Energy consumption for the assembly program using designs provided in [1-4].

Design	Total Energy Consumption
[1]	91.5 nJ
[2]	8.1e^{+3} nJ
[3]	1.6e^{+6} nJ
[4]	1.3e^{+6} nJ

The results from Table II indicate that Design 1 used the least amount of energy, 91.5 nJ, while Design 3 used the most, 1.6e^{+6} nJ. Analyzing the findings from Table I and II, it can be observed that, while the energy consumed for all other instructions, the greatest energy consumption for each branch instruction, the greatest the total energy that will be consumed by the program. Table I also shows that Design 1 had the most efficient DBN circuit for having consumed the least energy for its branch instruction.

IV. Conclusion

Over the years, string manipulation have become an essential factor in handling data. It has made tasks easier and more efficient than ever before. A useful practice can be an assembly language code that counts the number of occurrences of a word inputted by a user in a given statement. Through the use of assembler directives and syscalls it was possible to write such a program in MIPS.

This paper also evaluates the total energy consumed by such program when calculated by using the energy consumed for its branch instruction from four different Deep Belief Networks circuits designs. Deep Belief Networks are various Boltzmann Machines stacked together. A Boltzmann Machine is nothing else than a learning algorithm used for numerous fields.

The four final results were compared to determine the total energy consumption of the best design, which is the design with the least amount of energy consumed, Design 1.

References

[1] H. Pourmeidani, S. Sheikhfaal, R. Zand, and R. F. DeMara, "Probabilistic Interpolation Recoder for Energy-Error-Product Efficient DBNs with p-bit Devices," IEEE Transactions on Emerging Topics in Computing, 2020.

[2] A. Roohi, S. Sheikhfaal, S. Angizi, D. Fan, and R. F. DeMara, “ApGAN: Approximate GAN for Robust Low Energy Learning from Imprecise Components,” IEEE Transaction on Computer, vol. 69, no. 3, 2020.

[3] D. Le Ly and P. Chow, “High-performance reconfigurable hardware architecture for restricted boltzmann machines,” IEEE Transactions on Neural Networks, vol. 21, no. 11, pp. 1780–1792, 2010.

[4] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, “Vlsi implementation of deep neural network using integral stochastic computing,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, pp. 2688–2699, 2017.

[5] Hinton, G. E. (n.d.). Deep belief networks. Retrieved from http://scholarpedia.org/article/Deep_belief_networks

[6] Generative Model. (n.d.). Retrieved from https://www.sciencedirect.com/topics/engineering/generative-model