Non-Case Sensitive Keyword Frequency and Indexing Data Counter for User Defined Statements

Daryl Thomas
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816-2362

***Abstract*— The purpose of this paper is to give a detailed evaluation on how this program was constructed to achieve the specified functionality. Through the manipulation of strings arrays, and registers, the use of system call directives, branching instructions and other MIPS instruction, the program can realize the desired functionality. Specifically, the program accepts a user defined statement (limit: 2000 characters), and a keyword to search for matches in the statement. The output provides user with index location and the number of times the phrase appears in the input statement, while disregarding case discrimination. Also is a detailed analysis of demand on the hardware utilized, in terms of instruction count and energy demand using single bit memory cells. Each of these memory cells utilize four latches resulting in varying energy consumption. Thes latches are used in various technology to provide soft error hardening. Comparison of the various latches support the use of DNU-Latch for memory storage which consumes the least amount of energy of latches analyzed.**

Keywords*— *assembly code, Double node upset (DNU), energy efficiency, indexing, Memory, MIPS, Single Event Upset (SEU), reliability, Triple Modular Redundancy (TMR), reliability,

I. Introduction

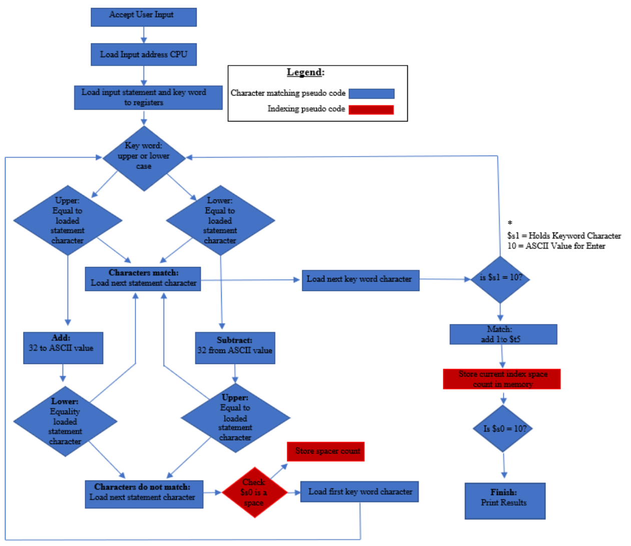
The coding strategy was implementing by utilizing a modular style featuring sections that performed a specific set of instructions to achieve design specifications.

A. Project Design

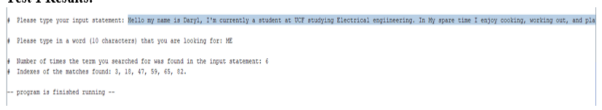
Design specification detailed that a user input statement and key word be entered. Using system call function "8" user defined input statement and key word can be accepted. Next addresses of input statement and key word are loaded in to registers. To manipulate the text for comparison the bytes were also loaded in registers $s0, $s1. For indexing locations to be stored a third address pointing to label "index_save" was also loaded to a register.
To satisfy design specification that key word not be case sensitive to be counted as a match, it was important to:

a. Define an instruction statement identifying if a character is upper or lower case

Figure1: Flowchart of the assembly


Accept User Input
Load Input address CPU
Load input statement and key word to registers
Legend:
Character matching pseudo code 🟦
Indexing pseudo code 🟥
Key word: upper or lower case
Upper: Equal to loaded statement character
Lower: Equal to loaded statement character
*
$s1 = Holds Keyword Character
10 = ASCII Value for Enter
Characters match: Load next statement character
Load next key word character
is $s1 = 10?
Add: 32 to ASCII value
Subtract: 32 from ASCII value
Match: add 1to $t5
Store current index space count in memory
Lower: Equality loaded statement character
Upper: Equal to loaded statement character
Is $s0 = 10?
Store spacer count
Characters do not match: Load next statement character
Check $s0 is a space
Load first key word character
Finish: Print Results

Test 1 Results:



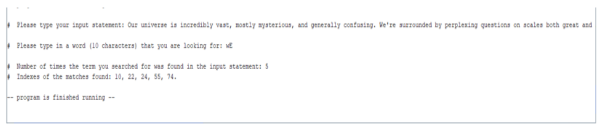
```
# Please type your input statement: Hello my name is Daryl, I'm currently a student at UCF studying Electrical engiineering. In My spare time I enjoy cooking, working out, and pla

# Please type in a word (10 characters) that you are looking for: ME

# Number of times the term you searched for was found in the input statement: 6
# Indexes of the matches found: 3, 18, 47, 59, 65, 82.

-- program is finished running --
```

Test 2 Results:



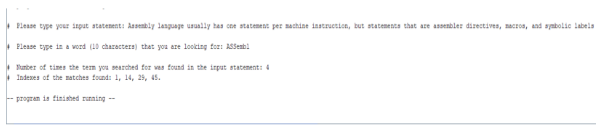
```
# Please type your input statement: Our universe is incredibly vast, mostly mysterious, and generally confusing. We're surrounded by perplexing questions on scales both great and

# Please type in a word (10 characters) that you are looking for: wE

# Number of times the term you searched for was found in the input statement: 5
# Indexes of the matches found: 10, 22, 24, 55, 74.

-- program is finished running --
```

Test 3 Results:



```
# Please type your input statement: Assembly language usually has one statement per machine instruction, but statements that are assembler directives, macros, and symbolic labels

# Please type in a word (10 characters) that you are looking for: ASSembl

# Number of times the term you searched for was found in the input statement: 4
# Indexes of the matches found: 1, 14, 29, 45.

-- program is finished running --
```

Figure 2: Program Output Results

This was executed using *slti* instruction. By comparing byte loaded in $s0 and 96 it and storing the value in a temporary register there is a set not set condition created. If set meaning value is one the character loaded in $s0 must be upper case. ASCII upper case values [65 - 90]. The unset case implies a lower-case value in $s0. [97 - 122]. After evaluation a decision to branch to "lowercase_check" or "uppercase_check" is executed.

b. Check equality of the key word character ASCII value with a test statement value currently loaded in $s1. A match is determined using MIPS instruction *beq*, which compares the integer values stored in byte address loaded in $s0 and $s1. If equality is found between these two registers program branches back to character check to check next character, however if equality is not found comparison for opposite case must be performed. In this program this was executed by adding (for upper-case characters) or subtracting (for lower-case character) by 32.

In the program these two partitions of instructions are labeled lowercase_check and uppercase_check which perform the appropriate instruction for each case respectively.

For program to effectively identify key word contained in the input statement, branch labels "*next_statement_character_nomatch"* and "*next_statement_character_match*" where created after discerning whether character case was upper or lower and matched or did not match. Each label implied a different set of program behaviors to advance through user input statement and key word, as well as operations to record match results for indexing.

next_statement_character_nomatch:

The processes to handle the programs behavior for characters that did not match was different from those that did. When characters loaded in $s0 (key word byte) and $s1 (statement byte) didn't match the first thing to do was to reset the loaded key word byte, signifying a restarting the check from the first character in the word. Meanwhile the program continues to advance thru the input statement, and count spaces.

next_statement_character_match:

While still advancing through characters in the user's input statement, when equality was found between characters the next step was to test if the next two characters were also equal. This again, was done by advancing through both input statement, and keyword. In MIPS this can be achieved by adding 1 to the register where the respective addresses are stored. In this case it was $t6 and $t7. This step followed by reloading the contents of the new address to temporary registers. Also, a word match was counted after a check confirming that there were no more characters to match in the keyword string.

The exit condition for the program was that all the characters in user input statement was equivalent to ASCII value 10. Instead of the null character, in this case 10 which is enter\next space in ASCII value signified that there were no more characters to check in both keyword and user input statement.

Indexing was accomplished by counting the spaces between each word in user input statement. A branch option checking the equivalency saved the number of spaces in temporary register $t5 after counting a space. After a keyword was found, the number stored un $t5 was stored in memory, so it could be recalled and printed in the "print results" portion of the code. Using spaces to count indexing progress could output incorrect results if more than one space is input between words. A solution to this would be to add an additional condition that the character following the index be something other than a space for it to count as an indexing space. This solution however would cost a dramatic increase in energy demand due to its requirement to repeatedly access memory. This feature was not specified in program design and was not added as a result.

B. Test Cases

Testing was done using the following test statements and keywords. Each test helped to confirm the program worked according to specification, by verifying the following design objectives:

1. Keyword is NOT case sensitive
2. Keyword can me contained and in a word and still be counted (Ex. Night in Nightmare counts as match)
3. Program accepts user input statement and 10-character keyword search
4. Program counts and outputs an accurate match count
5. Program exercises an accurate mechanism to count and output count index

Keyword: "ME"
*Test Statement 1***:**
"Hello, my name is Daryl, I'm currently a student at UCF studying Electrical engineering. In My spare time I enjoy cooking, working out, and playing basketball. I'm currently a junior and hope to graduate in 2020. I've been in school for a very long time so I hope to graduate as soon as possible, but at this moment I am a little exhausted, mentally. Often, I'm experiencing a lot of pressure. Navigating that, makes completing school work harder for me."

*Keyword***:** "wE"
Test Statement 2:
"Our universe is incredibly vast, mostly mysterious, and generally confusing. We're surrounded by perplexing questions on scales both great and small. We have some answers, for sure, like the Standard Model of particle physics, that help us (physicists, at least) understand fundamental subatomic interactions, and the Big Bang theory of how the universe began, which weaves together a cosmic story over the past 13.8 billion years. But despite the successes of these models, we still have plenty of work to do."

Keyword: "AsSembl"
Test Statement 3:
"Assembly language usually has one statement per machine instruction, but statements that are assembler directives, macros, and symbolic labels of program and memory locations are often also supported. Assembly code is converted into executable machine code by a utility program referred to as an assembler."

II. Memory Bit-Cells

With the advent of nanotechnology, processing capacity is expected to increase. This is largely an effect of the ability to improve transistor density in processing materials. However, with arrival of these new techniques arise new problems. One of those problems is an increased sensitivity to error which in affect raise reliability concerns. Single event upsets (SEUs) occur as a result of particle radiation. These radiated particles can stimulate transistor nodes, and cause voltage to peak, in affect producing a false signal. As that signal propagates through the circuit it can create issues with the output, memory and control signal.

Triple Modular Redundancy (TMR) provide protection from SEUs. As the name implies, they there are three devices signal is passed to which utilize a voter to verify unity in the signal. If there is a fault in one of the devices the vote cast by the remaining devices out vote the fault occurrence. In this way, TMR serves as a method of soft error tolerance. TMR however does have high spatial and energy demand (relative to alternatives) and is not totally fault proof [3]. A failure of the voter TMR uses could have a cascading effect on all outputs.

The dual interlocking storage cell (DICE) is SEU hardening latch configuration. Some of the benefits the design provides are identification and isolation of SEU vulnerable nodes [3]. It also provides a reduction in area consumption, output delay, and power consumption. A weakness of DICE design is it cannot provide multiple node faults occurring simultaneously. To circumvent this weakness one a combinational redundancy model utilizing two DICE latches, C-element, and a weak keeper has been discussed.[2].

Spin hall effect (SHE), Magnetic tunnel junction(MTI), non-volatile flip-flops (NVFF) , and CMOS based latches, are another way of addressing SEUs, and dual node upset (DNU) soft error. Each technology, and associated configuration focus on fault hardening in multiple areas while addressing energy consumption, spatial concerns, output delay and severity of fault occurrence at a variety of levels[1]. To address demand for high performance, low energy, and DNU tolerance CMOS based latches. An NVFF using SHE MTJs can be configured to provide fault tolerance while single , and multiple node upsets occur[1].

III. Results and Discussion

Assuming the following energy consumption profile for each respective instruction the MIPS assembly code uses. The following energy consumption for a single bit-cell memory design is detailed in Table I. Table II features energy for each design. These results reflect the program performance using baseline input statement and keyword. Using the number of instructions per type multiplied by energy consumed and added together the total energy consumption was calculated. Energy consumption is often time an indicant of soft error tolerance for that design.

Instruction Type	**Energy Concumed**
ALU	1fj
Branch	3fj
Jump	2fj
Other	5fj
Memory	Refer to Table I.

Table I: Energy consumption for a single bit-cell memory in the designs provided in [1-3].

Design	Energy consumption of a Single Bit-Cell Memory
SEU-Latch [1]	0.88 fJ
DNU-Latch [1]	0.28 fJ
TMR[2]	6.96 fJ
DNCS-SEU[3]	1.51 fJ

Table II: Total Energy consumption for the assembly program using designs provided in [1-3].

Design	Total Energy Consumption
SEU-Latch [1]	17362.44fj
DNU-Latch [1]	16589.64fj
TMR[2]	25193.48fj
DNCS-SEU[3]	18173.88fj

IV. Conclusion

Manipulation of branching, memory read and write, branching and arithmetic instructions were used to realize the program design. The program achieved all design specifications for multiple input phrases, and keywords regardless of case. Testing showed that keyword found in side of words as well as exact words were counted effectively. Testing also demonstrated that for longer statements and numerous keyword matches the program took more time to run, and used more energy. For this reason, it can be concluded that for this program the DNU-Latch can be utilized to not only provide hardening for single event upsets (SEU), but also dual node upsets. DNU-latch also uses the least amount of energy while performing read and write memory instructions. Memory bit-cells utilize multiple design features in combination and stand alone to achieve an optimal performance based on device priorities. Each design demonstrates a dual economy characterized by favorable soft error protection vs. performance, spatial, and energy tradeoffs. No one solution provides a universal solution, rather each design can be applied base upon hardware or software demand.

References

[1] F. S. Alghareb, R. Zand and R. F. Demara, "Non-Volatile Spintronic Flip-Flop Design for Energy-Efficient SEU and DNU Resilience," in IEEE Transactions on Magnetics, vol. 55, no. 3, pp. 1-11, March 2019, Art no. 3400611.

[2] H. Pourmeidani and M. Habibi, "Hierarchical defect tolerance technique for NRAM repairing with range matching CAM," 2013 21st Iranian Conference on Electrical Engineering (ICEE), Mashhad, 2013, pp. 1-6.

[3] K. Katsarou and Y. Tsiatouhas, "Double node charge sharing SEU tolerant latch design," 2014 IEEE 20th International On-Line Testing Symposium (IOLTS), Platja d'Aro, Girona, 2014, pp. 122-127.

[4] X. Wang, Y. Chen, H. Xi, H. Li, D. Dimitrov, "Spintronic memristor through spin-torque-induced magnetization motion", *IEEE Elect. Dev. Lett.*, vol. 30, no. 3, pp. 5-7, 127-129, Mar. 2009

[5] Jahanzeb Anwer, Marco Platzner, "Boolean Difference Based Reliability Evaluation of Fault-Tolerant Circuit Structures on FPGAs", *Digital System Design (DSD) 2016 Euromicro Conference on*, pp. 1-8, 2016.