Analyzing RNG Computation-Based Designs and Calculating the Simulated Energy Consumption

Nicholas Alban
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816-2362

***Abstract*—This paper explores Project 3 - Analyzing RNG Computation-Based Designs and Calculating the Simulated Energy Consumption, which involved creating a program in MIPS to count the number of occurrences of an inputted word string within a larger inputted statement string. The program would then output the total number of occurrences and the indexes these occurrences appeared. The fundamental metrics of energy consumption were measured based on different design option that all utilized random number generation and stochastic computation to more efficiently perform ALU instructions. The general topic of random number generation was explored, and the energy consumption of the program created for Project 3 was calculated using each presented design method. The design that used p-bit MRAM cells to produce random numbers was the most energy efficient as it had the lowest energy consumption of all other design alternatives, consuming 323521.68 pJ.**

Keywords— Random Number Generator (RNG), Probabilistic Spin Logic Device (p-bit), Processing In Memory (PIM), Memristor, Phase Change Memory (PCM), Stochastic Number Generator (SNG), True Random Number Generator (TRNG), Volatile Memory, Gaussian digital-to-analog converter (GDAC), Linear Feedback Shift Register (LFSR)

I. Introduction

The objective of this program was to search a given statement for a given word, count the number of occurrences, and show the indexes of these occurrences. The program accomplished this by parsing through every character of the statement string and checking if any series of characters matched the desired word. Every time a sequence of characters in the statement matched the sequence of characters in the word, it would increment the count and save the index. If the sequence of characters were interrupted with a character that did not belong, the program would restart the sequence checking for the first character in the word. The program would end once the null terminator of the statement string was reached.

Test cases were chosen to test basic functionality and tricky edge cases. The first test case was the given test case of the lab, which acted as a benchmark for the program and passing it would prove program functionality. The other testcases used tricky sequences of characters to test the program's ability to parse and properly follow the required steps given a wide variety of unique character sequences.

A. Project Design

This program starts by loading the Statement string and the desired Word string into address registers. The program parses through the word to remove any newline character that might have been added to the word string when the user pressed the "Enter" key to input the word. The word counter and word character counter are initialized to 1. Once this is done, the program loads the word and statement strings into registers and then begins its loop to search the statement for occurrences of the word.

First, a character from the word is loaded into a register and the program checks if the ascii value is equal to zero, which would be a NULL character. If this is true, the program branches; the last character of the word has been visited and the program has found a sequence of characters in the statement which matched the sequence of characters making up the word. The address pointer of the word is brought back to the first character by subtracting the word character counter, the word character counter is set to zero, and the word counter in incremented by 1. However, if this is not true, it will continue to the next step of loading the character from the Statement string address into a register.

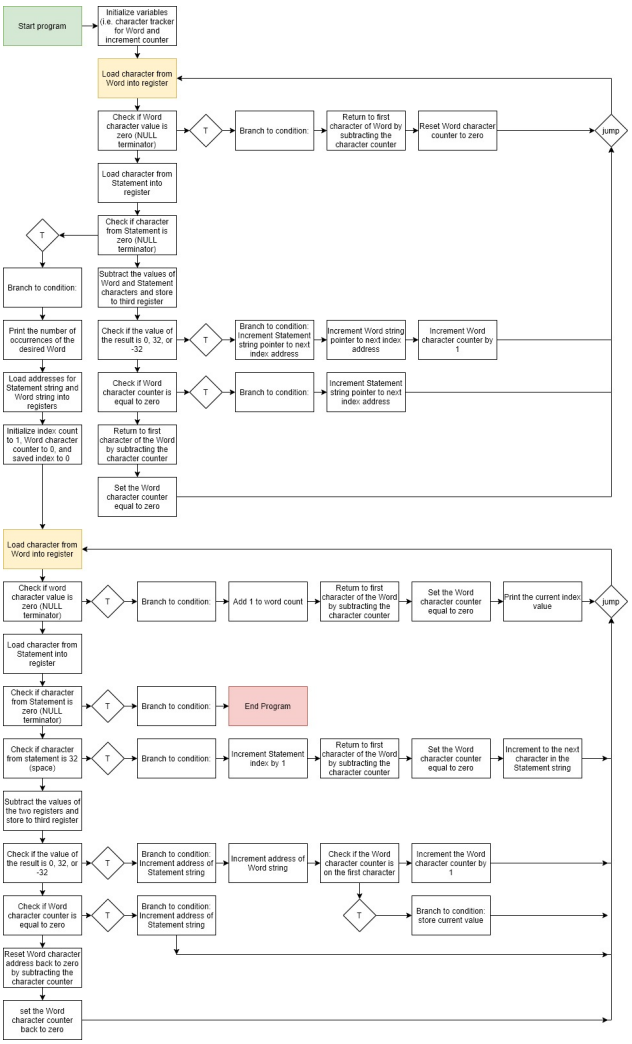
After this, it will check if the ascii value of the character from the statement string is zero. If this is true, it will branch, as the program has visited all characters in the statement. If this is not true, it will subtract the ascii value from the two characters currently stored in different registers and store the resulting difference in a third register. The value of this register is checked whether it is 0, 32, or -32 , which would mean these two characters were either the same, or one was an uppercase and the other was a lowercase. If this is true, the program will branch to another line to increment the pointer to the next character address of the statement string, do the same for the word string, and increment the word character counter by 1, then jump back to the start of the loop. If none of these three conditions are true, branch if the word character counter is equal to zero. At this branch, increment the statement string pointer and jump to the top of the loop.

If not true, continue to return to the first character in the word by subtracting the word character counter from the address pointer, and set the word character counter back to zero. Then, jump to the top of the loop. If the condition to branch was met where the last character of the statement string was found, then

it will branch to the next loop to do the same process, except outputting the indexes as the program continues. First, it outputs the number of word occurrences, then load the first addresses back into address registers, and initialized index count to 1, word character count to zero, and the saved index to 0.

Starting the second loop, the program loads the character from the word into a register and branches if this character's ascii value is zero. If true, it adds 1 to the word count, returns to the first character of the word, sets the word character count to zero, and prints the current saved index value. Then, it jumps back to the top of the loop. The program loads the ascii value of the character from the statement string into a register and branches if it is equal to zero. If true, it branches to the end of the program, and the program terminates. Otherwise, it branches if the value of this is 32 – a space. If true, it increments the statement index address by 1, returns to the first character of the word, sets the word character counter equal to zero, and increment to the next character address in the statement string. Then, jump to the top of the loop.

Fig.1: Flowchart of the assembly program.



true, increment the address of the statement string, check if the word character counter is on the first character still. If true, branch and store the current value of the index into a register and jump to the top of the loop, otherwise, increment the word character counter by 1.

If the two characters did not match, then check if the word character counter is equal to zero. If true, branch, increment the address of the statement string, and jump to the top of the loop. Otherwise, reset the word address back to the first character by subtracting the word character counter, set the word character counter back to zero, and jump back to the top of the loop. At the end of this second loop, there should be a series of indexes printed and the program will be terminated.

Fig.2: Sample outputs of assembly program.

Testcase #1:

```
Please type in your input statement:

The Knights Graduation and Grant Initiative is a UCF award to help undergraduat

Please type in a word (up to 10 characters) that you are
looking for (e.g. Knight or KnIGhT, knight, ...):

Knight


Number of times the word "Knight" was found in the input statement: 6

Indexes of the matches found: 2, 32, 42, 53, 85, 97,
```

Testcase #2:

```
Indexes of the matches found: 2, 32, 42, 53, 85, 97,
-- program is finished running --

Please type in your input statement:

JIMJiji mmmjImmmm jjjiM jiiim Mij j i m. jIMJImjin j im jiim JiM.

Please type in a word (up to 10 characters) that you are
looking for (e.g. Knight or KnIGhT, knight, ...):

jim


Number of times the word "jim" was found in the input statement: 6

Indexes of the matches found: 1, 2, 3, 9, 9, 13,
```

Testcase #3:

```
Indexes of the matches found:
-- program is finished running --

Please type in your input statement:

The quick** "BROWN" cat! %^ jumped (into the blue$) jeans123--quick & effortlessly?<>()cat.

Please type in a word (up to 10 characters) that you are
looking for (e.g. Knight or KnIGhT, knight, ...):

cAt


Number of times the word "cAt" was found in the input statement: 2

Indexes of the matches found: 4, 12,
```

B. Test Cases

The first test case was given by the lab. This included a long statement string set up with ordinary sentence structure and the program was specifically asked to look for the word "knights". This testcase proved the functionality of the program in more ordinary circumstances that would normally be encountered in regular sentence structure, therefore if it worked, it would prove general functionality.

The second test case was focused on testing the program's ability to reset checking the sequence and counting the number of occurrences appropriately, and also its ability to match the word to strings of varying capitalization. There were many of the same characters in particular sequences that would just barely match the sequence of characters of the word but slightly different. This was to test the program's ability to run progress part way through matching the word to characters in the string but stopping part way, without failing and losing accuracy.

The third test was to test the program's ability to essentially only pay attention to the alphabetical characters and not get stuck from these. The third test case included many random characters sprinkled among the statement string, testing the ability of the program to parse through these non-important characters. Also, special circumstances such as

All of these testcases were tested with multiple different inputs with varying capitalization, e.g. "Knight", "KNIgHt", "knight", etc. All tests were successful, as proven in the screenshots in Fig. 2.

II. P-BIT CIRCUIT

Random Number Generation (RNG) has important applications in multiple fields of computer science and architecture, ranging from security to processing [1][2][3]. Processing in Memory (PIM) is a stochastic based computer architecture that implements complex computations in main memory, providing massive parallelism and higher energy efficiency [2].Stochastic based computing has been found to be extremely relevant in brain-inspired computer architecture [1]. RNG is critical for an effective stochastic based computer system, and there are several ways to produce stochastic numbers with randomness. One example is Memristor neural networks, which uses volatile memory and switches to a low-resistance state under a voltage after a random time delay. Using this time delay can provide stochastic source for RNG [4] and output a randomly generated bitstream.

Another possibility is using Phase Change Memory (PCM) which utilizes cells of special material which can exist in one of two states of matter, crystalline or amorphous each one representing SET (bitwise 1) or RESET (bitwise 0) respectively. Introducing a reference voltage and current can cause a 50% chance of cells already in SET to change to a RESET state. By starting with an array of cells in SET, this can be leveraged to control the PCM and design a Stochastic Number Generator (SNG) [2] which given an input of N bits can produce an output of 2^N stochastic bitstream. This is implemented in the Gaussian digital-to-analog converter (GDAC), which in superior to the Linear Feedback Shift Register (LFSR). Another suggested method for SGN is the use of a probabilistic spin logic device (p-bit). The p-bit acts as a neuron and using voltages and currents to alter the energy barrier, a bitstream can be generated with randomness.

The final method explored is a Full-Entropy true random number generator (TRNG), which takes inputs from multiple independent entropy sources and creates one TRNG bitstream. Overall, RNG is critical for stochastic based PIM architectures. There are a variety of designs to realize this functionality to varying degrees of energy consumption.

III. RESULTS AND DISCUSSION

In this section, it is assumed an RNG circuit is used in the implementation of the ALU. This implies that the energy consumption of any ALU instruction in the program will be calculated with the energy consumption amounts from the different designs reported in references [1-3]. The energy consumption of each design can be found in Table I.

Using the given energy consumption per non-ALU instruction values:

1) ALU = refer to Table I
2) Branch = 4 pJ
3) Jump = 3 pJ
4) Memory = 100 pJ
5) Other = 5 pJ

Table I: Energy consumption for a single ALU Instruction in the designs provided in [1-3].

Design	Energy Consumption For Each ALU Instruction
[1]	0.08 pJ
LFSR [2]	0.9 pJ
GDAC [2]	90 pJ
[3]	36 pJ
Memristors [4]	23 pJ

For Testcase 1, there were a total of 19342 instructions, with a total of 6496 ALU instructions, 1428 jump instructions, 8552 branch instructions, 2844 memory instructions, and 22 other types. Assuming that for all designs the jump, branch, memory, and other type instructions all had the same energy consumption, the energy consumption of these four types could be calculated and added to each design's energy consumption for each ALU instruction.

For the non-ALU instruction energy consumption:

(4 pJ)*(8552) + (3 pJ)*(1428) + (100 pJ)*(2844) + (5 pJ)*(22) = 323002 pJ

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

Design	Total Energy Consumption
[1]	323521.68 pJ
LFSR [2]	328848.4 pJ
GDAC [2]	907642 pJ
[3]	556858 pJ
Memristors [4]	472410 pJ

The total energy consumption for each design can be seen in the Table II.

IV. Conclusion

Implementation of RNG has the potential to improve processing efficiency and speed and reduce energy consumption. This can be achieved through several different methods involving RNG using a variety of methods involving probabilistic bit generation. This probabilistic generation is utilized to generate stochastic numbers for stochastic based computer architecture systems which closer represent brain architecture. This architecture has the potential to more efficient energy consumption for ALU instructions than traditional computer systems.

List of technical topics explored in this project:

1. RNG and the critical importance of random number generation
2. Phase Change Memory devices and the functionality of them
3. Stochastic computing and its importance in brain-inspired computer architecture
4. P-bits and how to use their properties to generate random numbers
5. Effectiveness of brain-inspired computer hardware design

The program designed for this project had approximately a third of its total instructions as ALU instructions. Therefore, the design with the lowest ALU power consumption would have a significant impact on the program's overall power consumption. This means that the design from paper [1] using p-bits as stochastic neurons, consuming a total of 323521.68 pJ.

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] S. M. Shivanandamurthy, I. G. Thakkar, and S. A. Salehi, "Work-in-Progress: A Scalable Stochastic Number Generator for Phase Change Memory Based In-Memory Stochastic Processing," in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion, pp. 1-2, October 2019.

[3] S. K. Mathew, D. Johnston, S. Satpathy, V. Suresh, P. Newman, M. A. Anders, H. Kaul, A. Agarwal, S. K. Hsu. G. Chen, and R. K. Krishnamurthy, "μRNG: A 300–950 mV, 323 Gbps/W all-digital full-entropy true random number generator in 14 nm FinFET CMOS," in IEEE Journal of Solid-State Circuits, vol. 51, no. 7, pp. 1695-1704, 2016.

[4] Jiang, H., Belkin, D., Savel'ev, S.E. *et al.* A novel true random number generator based on a stochastic diffusive memristor. *Nat Commun* **8,** 882 (2017). https://doi.org/10.1038/s41467-017-00869-x.

[5] Yang, K. et al. A 23Mb/s 23pJ/b fully synthesized true-random-number generator in 28 nm and 65 nm CMOS. *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)* 280–281 (San Francisco, CA, USA, 2014).

[6] Srinivasan, S. et al. 2.4GHz 7mW all-digital PVT-variation tolerant true random number generator in 45nm CMOS. *Symposium on VLSI Circuits* 203–204 (Honolulu, HI, USA, 2010).