An Analysis on Tradeoffs between Speed and Power Dissipation within ALUs

Moses A. Quiliche | Jan G. Iglesias Morales
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816-2362

Abstract – Two key design aspects when it comes to designing ALUs are speed and power dissipation. Increasing the speed of ALUs is possible, but typically comes at the cost of higher power dissipation which can have negative repercussions such as data transfer delays within VLSI systems. An Analysis on Tradeoffs between Speed and Power Dissipation within ALUs gives a better idea of how both speed and power dissipation relate. A cipher program is implemented and tested onto MIPS to get a count for how many ALU instructions were used within the cipher program. Total power dissipation for three ALUs are then calculated to get an idea of how much energy is consumed and how fast each of the three configurations are. Each of the designs have 3 inputs of A, B, and C_{in} with two outputs of S and C_{out}. Within each design, the computational approach is different. Of the three configurations the spintronic design is found to be the best with a total energy consumption of 4,696 pJ. It also gives the consumer freedom to prioritize speed over energy consumption, or vice versa by varying the frequency.

Keywords — VLSI, Multi-bit Magnetic Adder, Non-Volatile Full Adder, Ferromagnetic Strip, Racetrack Memory

I. Introduction

A. Project Design

The purpose of this project was to create a cipher by using the "QuadMinMix" of two unsigned integers. The "QuadMinMix" of two numbers is simply the number generated by converting two numbers into hexadecimal and taking the smallest of each quad and placing each quad into a new number. The number generated by this process is the QuadMinMix. The overall program functions by generating the QuadMinMix of two numbers and then creating a frequency array of hexadecimal digits which will count the frequency of these digits in the QuadMinMix. The "QuadBitCipher" is the resulting cipher of this program. The QuadBitCipher is generated by outputting the entire frequency array of hexadecimal digits in descending order.

The program was designed by first designing the QuadMinMix subroutine and then designing the QuadBitCipher subroutine. The QuadBitCipher subroutine will call the QuadMinMix subroutine and then fill a frequency array with the result returned by the QuadMinMix subroutine. The QuadBitCipher subroutine does not return anything, but instead it prints the result to the screen. The overall design was first coded in C to test the viability of the design before being implemented in MIPS.

The QuadMinMix subroutine requires two parameters to be passed before the subroutine can be executed. These parameters are two unsigned integers. The parameters used will be referred to as X and Y. The instant the subroutine is executed three registers are pushed onto the stack. The return address pointer is pushed onto the stack in order allow for recursive capabilities in this program. Two save registers are used by this subroutine so they are pushed onto the stack in order to preserve them. Afterwards, a mask is used to extract the individual quads of X and Y. The mask used was given the value of 15 because 15 represents 1111 in binary. The chosen bitmask will allow for the extraction of a single quad from X and Y. A single quad is processed at a time. First, the rightmost quad of X and Y are processed and each stored into temporary registers. The smallest of these is then placed into the rightmost quad of the result. This process is then repeated for the second rightmost quad and onwards until all quads have been processed. This process can be done in a loop, but in the implementation chosen was done by unrolling loops in order to achieve a lower dynamic instruction count. The registers that were pushed onto the stack are now recovered by popping from the stack. The result is then returned by this subroutine.

The QuadBitCipher subroutine requires the same two parameters required by the QuadMinMix subroutine. This subroutine requires two unsigned integers which shall be referred to as X and Y. Several registers are saved onto the stack in order to preserve their contents. The return address register is pushed onto the stack along with two save registers. The QuadMinMix subroutine is called using X and Y as the parameters. The result of the QuadMinMix is then captured in a save register and shall be referred to as Z. A bitmask will be used to read the individual quads of Z. The same bitmask used for the QuadMinMix

subroutine is used for this subroutine. A frequency array was used for this subroutine. The frequency array was initialized to 0 and a reference to this array was passed to a save register. The quads in Z are extracted starting at the rightmost quad and moving to the left until all quads have been processed. When processing a quad, it is a read and stored in a temporary register. The temporary register is then shifted to the right until the quad extracted is the rightmost quad in the register. At this point, the temporary register is multiplied by 4 and used as an offset for accessing the array. This offset will yield the position of the array represented by the quad extracted from Z. The offset is then added to the register holding the base address of the array which will cause the new base address of the array to point to the desired value. The value is incremented and stored back into the register. The register holding the array is then subtracted by the value in the quad multiplied by 4 in order to return it to the proper base address. This process is repeated for all 8 quads in Z using unrolled loops to decrease the overall instruction count. Afterwards, the contents of the frequency array are printed to the screen in decreasing order. The printing of the array starts at the offset representing the hexadecimal digit 'F' and ends at the offset representing the hexadecimal digit '0'. Lastly, the registers that were pushed onto the stack are popped in order to restore them.

The project was given an efficient implementation which yielded a relatively low dynamic instruction count when given a specific input. When given inputs of X = 1792801454 and Y = 2016082984 the MIPS program had a dynamic instruction count of 211 instructions. The memory access was also optimized to require the least number of memory access instructions. A cache hit rate of over 90% was achieved with a minimum cache size of 256 Bytes.

B. Test Cases

Five test cases were used in order to test the functionality of the program. The first testcase used was one provided by the project instructions. The values in this testcase were random. The second testcase was designed and was simply composed of a random X and a random Y. The third testcase ensured that the program still worked as intended when X and Y were the same number. The fourth and fifth testcases ensured that the program worked when all the quads used to assemble the QuadMinMix comes from only one of the parameters. All these test cases were computed by hand and then confirmed using the C-language prototype. The MIPS implementation passed all these testcases and matched the output calculated by hand and generated by the C-language prototype code.

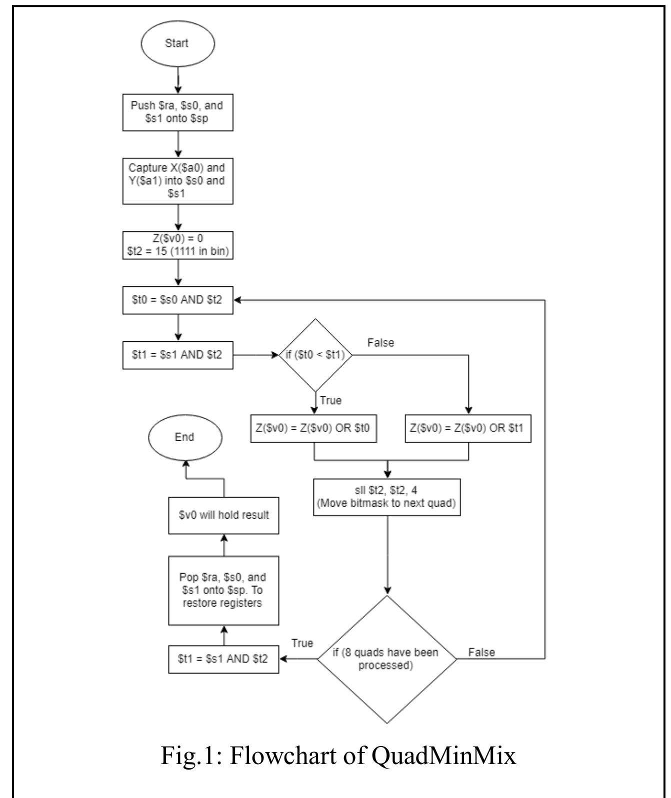

Fig.1: Flowchart of QuadMinMix

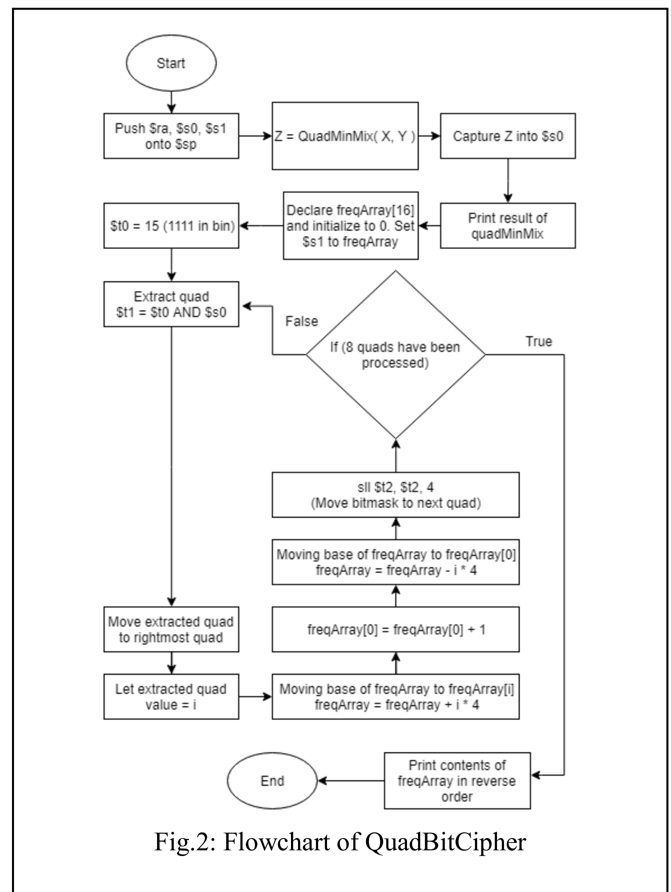

Fig.2: Flowchart of QuadBitCipher

```
1792801454
2016082984
QuadMinMixer = 0x682afa28
QuadBitCipher = 1000020201000200
-- program is finished running --

243647654
954354234
QuadMinMixer = 0x08824436
QuadBitCipher = 0000000201021101
-- program is finished running --

1234567890
1234567890
QuadMinMixer = 0x499602d2
QuadBitCipher = 0010002001010201
-- program is finished running --

715827882
305419896
QuadMinMixer = 0x12345678
QuadBitCipher = 0000000111111110
-- program is finished running --

305419896
715827882
QuadMinMixer = 0x12345678
QuadBitCipher = 0000000111111110
-- program is finished running --
```

Fig.3: Testcase inputs and outputs

II. FULL-ADDER CIRCUIT

A full adder in general has three inputs A, B, and C_I and two outputs for the sum and carry called S and C_O. There are many different designs for full adders out there, each with their own unique benefits and trades offs [4].

The key design aspect of full adders of the future are achieving a high sensing margin while keeping power dissipation low. This is especially useful in VLSI systems which tend to suffer from large power consumption which can cause delays due to memory retention [3]. Research in solving these issues has led to a couple possible solutions, one of them being the use of non-volatile memory, more specifically Racetrack memory. Due to Racetrack memory being a non-volatile memory, it can combine memory and logical circuits into one, reducing the fly time between memory and logic for full adders. The use of Racetrack memory in turn results in a decrease in the number of transistors needed within a given CPU [3].

The block diagram for the NVFA in question is displayed in Figure 5, along with the equations that define it right below.

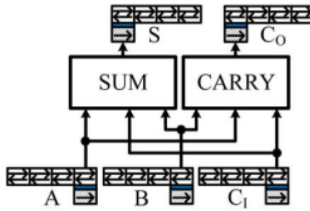

Figure 4: FA Block Diagram
Source: Adapted from [3]

$$S = ABC_I + A\overline{BC_I} + \bar{A}B\overline{C_I} + \overline{AB}C_I$$

$$C_O = AB + AC_I + BC_I$$

The "carry" component of the circuit compares the resistance of the two branches shown within the schematic below. The branch on the left corresponds to $\overline{C_O}$ which is equivalent to a logical output of 0, while the right side corresponds to C_O which is equivalent to a logical output of 1 [3].

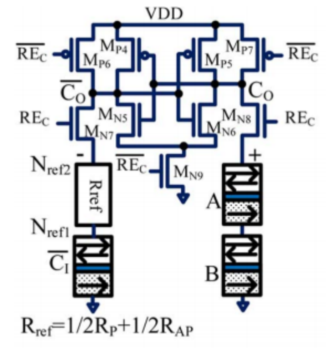

Fig 5: Carry circuit
Source: Adapted from [3]

A higher resistance results in a logical output of 1, while lower resistance results in a logical output of 0. This is because the higher of the two currents within the branch will be redirected to ground, while the weakest current resulting from higher resistance will head towards VDD [3].

Three MTJ's corresponding to A, B, and C_I form the three inputs for a full adder. If MTJ's A and B were both on, then they would cause a higher resistance on the right side, resulting in C_O being the output which is equivalent to a logical output of one. If only MTJ A is on, it would have a lower resistance due to C_I and R_{ref}. This would result in a logical output of 0 [3].

The sum component of the circuit which can be seen below uses the exact same principles of the

carry when it comes to the output of either S or $\bar{S}$. However, the value of R_{ref} changes depending on the value of the carry C_O. If the value of C_O is found to be 1, then the resistance on the left branch is the greatest and will cause $\bar{S}$ to be the output [3]. The only case in which the right branch will have a higher resistance will be when both MTJ's A and B are on.

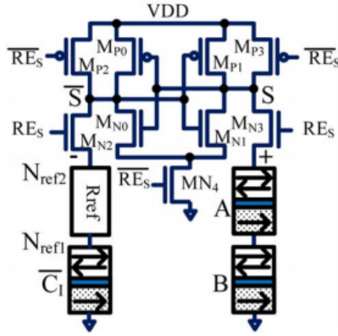

Fig 6: The Sum Circuit
Source: Adapted from [3]

Since both the sum and carry share the same inputs, there is a time delay between the two to calculate the value of C_O based off inputs A, B, and C_I in order to adjust the value of R_{ref} accordingly. Afterwards the data is loaded onto a single ferromagnetic strip, creating a 1-bit full adder. For a multi-bit adder these can be cascaded with the C_O of the previous adder as the input for the C_I of the next, allowing for a single ferromagnetic strip to still be a viable way of displaying data since there'd be no need to fetch carries from previous adders [3].

The circuit displayed in Figure 7 also uses resistance to decide the logical output to be 1 or 0, however its logic is flipped from the previous circuit. If the R_{left} is higher, then it results in a logical output of 1. If R_{right} is higher, then it results in a logical output of 0 [2]. Like the previous circuit, this implementation uses Racetrack memory and MTJs for inputs A, B, and C_i.

Within this design though two separate magnetic strips are used to write and read data [2]. For the write, both possibilities of the carry being either C_O or $\overline{C_O}$ are computed via the PCSA circuit. The decision is based off the current flowing through the magnetic strip for C_O. It will reverse directions if the data being written does not match with what is being read [2].

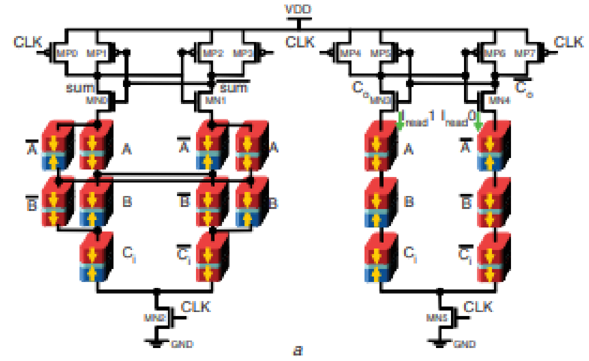

Fig 7: FA Circuit
Source: Adapted from [2]

The next circuit uses spintronics along with domain wall motion to realize a full adder by converting analog signals to digital. To realize the full adder there are two modes this circuit can take on, those two being ADC mode and Logic-in-Memory mode [1].

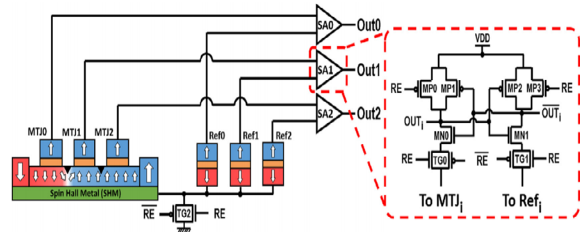

Fig 8: Spintronic Circuit
Source: Adapted from [1]

ADC mode is used to convert analog signals into digital. Within ADC mode the circuit is reset by moving the DW to the beginning of its domain in order to prepare it to receive a signal. From here the analog signal is used as an input to move the DW. Given the signal has enough amplitude, it will output a digital signal representing a logic of 1, if not it will output a 0 [1]. This will allow for a reading of inputs A, B, and C from their corresponding MTJ's.

Logic-in-Memory mode is used to compute the resulting digital logic. It starts off by resetting the circuit as mentioned in ADC mode. The currents coming from the MTJ's that correspond with inputs A, B, and C are then used to move the domain wall accordingly to perform logical operations [1]. Allowing for the data from ADC mode to be implemented into logic.

III. RESULTS AND DISCUSSION

Each different ALU design discussed incurred a different energy cost. While some types of ALUs have much better speed performance than others these faster ALUs usually incur a much higher energy cost. The different cost of ALU instructions for each design is shown in Table I.

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

Design	Energy Consumption For Each ALU Instruction
[1]	0.6 pJ
[2]	6.3 pJ
RTM-based [3]	1.67 pJ
STT-based [3]	1.61 pJ

Given these different ALU costs and a standard instruction cost shown below:

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

The total energy consumption that each ALU design would incur for the QuadBitCipher project is shown in Table II.

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

Design	Total Energy Consumption
[1]	4696 pJ
[2]	5380 pJ
RTM-based [3]	4824.4 pJ
STT-based [3]	4817.2 pJ

Based off Tables I and II Design 1 has the least energy consumption, while Design 2 has the most. Design 2 may in some cases be the fastest of the three, but in order to reach such speeds large power dissipation is nearly inevitable. This is one of the main reasons cooling is key, because if power dissipation is not kept in check, then damage can be inflicted onto the CPU. Design 1 on the other hand is efficient and wouldn't be much of a burden to cool, however depending on the configuration speed requirements may not be met. Design 2 and 3 provide a nice middle ground where speed is improved, but not at the cost of greatly increasing the power dissipation or energy consumption.

IV. CONCLUSION

Project three introduced the possibility of reaching computing powers beyond the Von Neumann threshold. This is executed by cutting out the fly time between memory and logic by integrating the two into one. To realize this, an analysis on power dissipation, overall energy consumption, and speed between the three designs that integrated memory and logic was performed on all three designs. Between all three designs, Design 1 was seen to be the best with the lowest overall energy consumption. Design 1 can also be adjusted to suit certain needs as well, while keeping the energy per bit the same at 0.6pJ. If speed is key you can adjust the input of the analog signal to be 1GHz to have a delay of 2ns, and power dissipation that's about that of the Racetrack memory. If power dissipation is needed to be kept low, it can be cut nearly in half by decreasing the frequency to 500MHz. This does come at the cost of cutting the speed in half, but as stated before you can cater this design to meet both power dissipation or speed. Overall this Project was a great opportunity to understand more about the 6 following topics.

- Racetrack Domain Wall memory
- Integrating Logic and Memory into 1
- Relationships between energy, power dissipated, and speed within ALUs.
- Manipulation of the stack pointer in MIPS to allow for the implementation of recursion and to backup save registers.
- Implementation of MIPS subroutines that take in input parameters and return an output variable.
- Usage of techniques for lowering dynamic instruction count such as unrolling loops and minimizing redundant instructions.

References

[1] S. Salehi and R. F. DeMara, "SLIM-ADC: Spin-based Logic-In-Memory Analog to Digital Converter Leveraging SHE-enabled

Domain Wall Motion Devices," Microelectronics Journal, Vol. 81, pp. 137–143, November 2018.

[2] H. P. Trinh et al., “Magnetic adder based on racetrack memory,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 60, no. 6, pp. 1469–1477, Jun. 2013.

[3] K. Huang, R. Zhao and Y. Lian, "A Low Power and High Sensing Margin Non-Volatile Full Adder Using Racetrack Memory," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 62, no. 4, pp. 1109-1116, April 2015.

[4] “The Full Adder,” *The Full-Adder*. [Online]. Available: http://hyperphysics.phyastr.gsu.edu/hbase/Electronic/fulladd.html. [Accessed: 05-Apr-2019].