Implementation of MIPS microprocessor architecture using a fast adder ALU circuit

Michael Judrick Aringo, Henson Adrian Lee, Steven Joseph Obico, Jared Craig Ong, Sean Patrick Tan

Department of Electronics and Computer Engineering
De La Salle University
2401 Taft Avenue, Manila, Metro Manila, Philippines
{michael_aringo, sean_patrick_y_tan, steven_obico, henson_lee, jared_craig_ong}@dlsu.edu.ph

Abstract - **This paper presents the design and implementation of a MIPS microprocessor architecture enhanced with a fast adder ALU circuit to improve performance. The MIPS architecture, a widely used implementation of RISC, is optimized by replacing conventional ripple carry adders (RCA) with a carry look-ahead adder (CLA) to reduce carry propagation delay and enhance processing speed. The processor was developed and tested using Vivado IDE, employing both structural and dataflow modeling. Results from the simulations demonstrate that the CLA significantly improves execution speed, with a measurable 4.5% increase in performance over traditional designs. This enhancement, though modest, contributes to more efficient computation in digital systems, especially in high-performance computing environments where time efficiency is critical.**

Index Terms - MIPS architecture, fast adder, carry look-ahead adder, ripple carry adder, FPGA implementation

I. Introduction

The RISC architecture is a computer architecture designed to perform a small optimized set of instructions as fast as possible. Compared to CISC architecture computers, RISC has a significantly reduced and less specialized set of instructions, meaning that more complex instructions take up several operations. The RISC architecture however makes up for this by being able to execute its instructions faster than a typical CISC processor. This allows RISC architecture processors to have a simpler construction and more power efficient design compared to CISC processors (ARM, n.d.). Among the many implementations of the RISC architecture is the MIPS instruction set, which is a popular computer architecture used for educational purposes and implemented in embedded systems.

Though RISC processors are designed to execute instructions quickly, it is possible to improve the underlying hardware architecture to achieve increased performance, such as in modifying the ALU. The ALU, or Arithmetic Logic Unit is the component of the CPU that performs all calculations and logical operations. It performs all arithmetic operations such as addition and logical operations such as AND, OR and NOR in the processor, with more complex arithmetic and logical operations arising from a few basic operations (Ferguson & Hebels, 2003). among the many components of the ALU are the adder circuits. ALUs often contain half and full adder circuits in order to perform arithmetic operations (CircuitVerse, n.d.). While these circuits serve their function well, it is possible to improve upon the ALU by implementing fast adder circuits instead of regular adder circuits.

The fast adder circuit or carry look ahead circuit, is a circuit that reduces the propagation delay of a carry in an adder circuit by introducing additional hardware that uses logic gates in order to quickly determine the carry logic in the circuit (Geeksforgeeks, 2023). Regular adder circuits typically make use of a ripple carry in their circuits in order to implement the carry. While this type of circuit is able to produce accurate results without issue, the propagation delay that results from the ripple of the carry from bit to bit can slow down the speed of the ALU(Geeksforgeeks, 2023). As such, a fast adder circuit can be used in place of typical ripple adders in an ALU to speed up arithmetic operations and improve the response time of the CPU and speed up the execution of instructions.

This paper shall see the construction of a MIPS microprocessor that implements fast adder circuits in order to increase the performance of the ALU and reduce the execution time for addition instructions using the ALU.

II. Computer Architecture Specifications and Application

A. Application Area

As previously stated, the RISC architecture is a microprocessor architecture design that focuses on speed and simplicity in order to increase performance with a small instruction set, as opposed to CISC which has a larger amount of instructions saving memory space, while sacrificing execution speed and complexity (Geeksforgeeks, 2024). The RISC architecture was chosen as its simplicity and focus on fast execution times make it well suited to improvement using a fast adder circuit, as instructions are not as bottlenecked through the pipeline as is the case with some CISC architecture processors. The MIPS instruction set implementation meanwhile was chosen as it is a popular implementation of the RISC architecture making use of pipelining of instructions, with numerous available online resources.

The microprocessor shall consist of a 32-bit pipelined RISC architecture processor with fast adder circuits, where the fast adder component shall be implemented in place of full adder circuits within the processor, namely in the ALU, ensuring the most utility out of implementing said circuit design. This will allow faster processing time per instruction , and improve throughput, and thus the performance of the processor, as instructions such as calculating memory address & arithmetic will be sped up due to the use of a fast adder circuit.

The carry look ahead adder used is a modified version based on a paper by Balasubramanian & Mastorakis (2022), who developed a faster and more energy efficient implementation of a carry look ahead adder, which greatly reduces power consumption and critical path delay compared to conventional carry lookahead adders through changes in the circuit that allow the carry bit to propagate quickly.

The processor shall be designed and developed with the use of the Vivado IDE. The design is to be developed based on a 32-bit MIPS architecture, modified to use fast adder circuits, and perform multiplication and division using the ALU. Most of the components in the processor shall be programmed using structural modeling, specifying the connections between different components by declaring modules within each component and declaring connections between the input and output pins or logic gates of each component. The ALU meanwhile uses dataflow modeling, which states the flow of data within a component in order to model its functionality. Once the system has been developed it will then be simulated using testbench files in order to verify its functionality and perform an analysis of performance compared to processors using full adder circuits.

B. Instruction Set Architecture

The instruction set architecture or ISA is the set of all instructions that can be executed by the processor. These instructions are written in assembly, and implemented as sequences of bits in machine code. The ISA serves as the bridge between hardware and software, as it provides a set of instructions that the software can use to dictate instructions for the hardware to perform, similar to a menu in a restaurant. The proposed architecture makes use of a basic instruction set with 10 basic instructions. This allows the processor to have a simple yet effective design as it reduces the complexities needed to decode instructions, and increases performance of the processor for simple instructions.

Instruction	Instruction format	operation
ADD	ADD rd, rs, rt	rd = rs +rt

BNE	BNE rs, rt, imm16	if (rs != rt){
		PC = PC + 4 + imm16<<2}
		
		else{PC = PC+4}

J	J target	PC = {PC[31:28,target, 00}
JR	JR rs	PC = rs
LW	LW rt,imm16(rs)	rt = Mem[rs] + imm16

SLT	SLT, rd,rs,rt	if (rs < rt){
		rd = 00000001}
		
		else{rd = 00000000}

SUB	SUB, rd,rs,rt	rd = rs - rt
SW	SW rt,imm16(rs)	Mem[rs] + imm16 = rt
XORI	XORI, rt, rs, imm16	rt = rs XOR imm16
ADD Fast	ADDF rd, rs, rt	rd = rs +rt

Figure 1: instruction set of the proposed architecture

The instruction set contains instructions for all the basic operations necessary for a processor to perform most basic operations performed on a CPU. Each instruction is represented by a 32 bit code, The opcode is composed of 6 bits found in the function portion of the entire 32 bit instruction and 2 bits of the Opcode portion of the instruction. These are then concatenated and sent to the ALU control in order to send out a 3 bit long control signal to the ALU to determine what operation to perform. The remaining bits contain fields for variables such as immediates or register addresses depending on the operation.

The ISA contains 2 different ADD operations: ADD and ADDF, with ADDF being the instruction used for using the fast adder ALU, while ADD is simply used for troubleshooting purposes. ADDF has an opcode of 110000, whileADD has an opcode of 10000, and both functions make use of registers declared as rs and rt as operands and register rd to store the result. The SUB instruction also works similarly to the ADD operation, performing subtraction using the ALU with similar register referencing. XORI meanwhile works similarly to ADD or SUB, however referencing rs as an operand and uses register rt to store the result; it does however use an immediate variable as its other operand. The operation is used to perform XOR operations on the register value referenced in rs and an immediate data.

The ISA also includes 2 different Jump instructions, with J jumping to an immediate address specified in the instruction, and JR jumping to an address specified in the register rs. BNE and SLT meanwhile are decision instructions that allow branching paths in code. BNE or branch not equal, uses registers rs, rt, and an immediate operand; if the value in rs is not equal to rt, then the instruction will jump to an address specified in the immediate operand, if rs= rt however then the program will continue onto the next instruction. SLT or set less than meanwhile uses registers rd, rs, and rt similar to the AND instruction.if rs is less than rt then the value in rd is set to 1, else, the value in rd is set to 0.

Finally, the SW and LW operations, or store word and load word respectively, are used to read or write data into memory. both operations make use of registers rt, and rs and an immediate operand representing the memory address. LW fetches the data stored in the the location specified by the value stored in rs + the immediate and places it in register rt, while SW takes the data in rt and stores it in the memory address specified by rs + immediate.

C. *Computer Performance Testing*

The performance testing of the MIPS processor with fast adder circuits aims to evaluate the speed and efficiency enhancements. The focus will be on checking the execution time of arithmetic and logical operations within the ALU, comparing the original processor and the modified processor.

The design will be implemented and tested using Vivado to carry out performance testing. The processor will be constructed using structural modeling, while the ALU will be designed using dataflow modeling to reflect the data flow and logic accurately. Testbenches will be created to test various arithmetic and logical operations, including addition, subtraction, multiplication, and division. These programs will be executed on the processors to measure and compare their performance.

Criteria for evaluation will include execution time, throughput, and latency. Execution time will be measured to determine the time to complete each operation. Throughput will be evaluated by assessing the number of instructions executed per unit time. Lastly, latency will be analyzed to understand the impact of the fast adder circuits on processor responsiveness.

Simulation of testbench files will show detailed data of the system. Comparative analysis will be performed to quantify the improvements in speed and efficiency. It is expected that replacing the full adder circuits with fast adder circuits in the ALU will reduce execution times for arithmetic operations, increase throughput due to faster instruction execution, and decrease latency, resulting in a more responsive processor.

III. Objectives

This engineering paper primarily aims to develop and implement a fast-adder ALU circuit based on the several specifications and operations within the MIPS or the Microprocessor without Interlocked Pipeline Stages architecture. The secondary objectives of this project are the following.

- To Simulate the MIPS architecture through the environment of Vivado.
- To Implement and add the fast adder ALU circuit to the MIPS architecture
- To Implement three sorting algorithms in the created architecture.
- To verify the results through simulations in the timing diagram window.

IV. Design Operation

As the processor is based on the MIPS architecture, it includes pipelining features that separate the processor into stages, that allow multiple instructions to be processed at once, with 1 instruction executing per stage at any given moment. This allows maximum utilization of the CPU at any given point in time, as ideally there would be 1 instruction being executed at each stage unless a hazard or jump occurs. The 5 stages of the pipeline are the instruction fetch, Instruction decode, execute, memory access, and write back stage.

The instruction fetch stage includes the memory file as well as the program counter. This stage of the pipeline uses the program counter to generate instruction addresses sequentially and getch said instructions from the instruction memory to be sent to the next stage. This stage aslo takes in address inputs from the execution stage in the event of a jump operation which changes the address of the next operation to be performed.

The instruction decode stage meanwhile takes in the instructions from the instruction fetch and decodes the opcodes and variables in order to generate control signals to be sent to the ALU,and register files. As the proposed architecture contains 2 ALUs (1 for regular operations and 1 for fast adder operations), the control signals are multiplexed to determine what ALU to use. The execute stage consists of the ALUs, which take in control signals from the control unit and operands from registers or immediates, and output the result to the next stage.

The memory access stage is where memory read or write operations occur, this stage is where the output of the ALU is either stored in memory or used to fetch memory addresses. The final stage is write back, where the result of the ALU or memory access is stored in a register within the register file.

A typical full adder or ripple carry adder circuit uses propagation for adding a carry bit; i.e., it uses cascading 1-bit full adder circuits in order to propagate carry bits. This propagation can cause delays and slow down the ALU, especially once the size of the ALU increases (Hasan, et al., 2021). Fast adder circuits such as carry look ahead circuits meanwhile can allow for faster addition operations with no propagation in the carry bit to cause delays. The proposed processor uses a carry look ahead adder (CLA) modified for improved speed and power efficiency.

The CLA circuit can calculate multiple carries at once and reduce carry propagation by pre-calculating carry signals for each bit. The To achieve this, the CLA uses 2 signals: generate (G) and Propagate (P), which are defined as fiollows:

$Gi=Ai * Bi$
$Pi=Ai + Bi$

where Ai and Bi are inputs at bit position i.

The carry (C_i+1) for each bit position is calculated with the following formula:

$Ci + 1 = Gi + Pi * Ci$

this allows carry signal processing to be performed in parallel for all bits, significantly reducing propagation delay (Parhami, et al., 2000).

V. Test Data

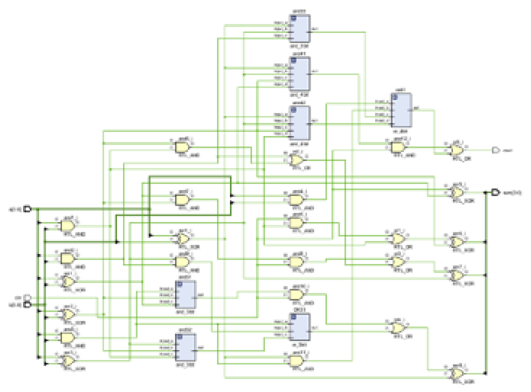


Figure 1: 4-bit Implemented CLA schematic

The figure above shows the schematic diagram of the 4-bit CLA., the inputs of which are a[3:0], b[3:0], and cin. It can be seen that the implemented CLA is a collection of AND gates and XOR gates, the outputs of which are directly connected to the 3-bit AND and 3-bit OR gate modules. The outputs of which will be directly placed in the sum[3:0] variable along with the cout, which will house the carry-out variable.

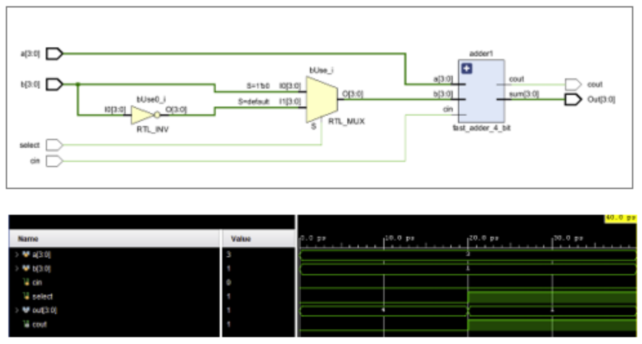


Figure 2: 4-bit Implemented adder/subtractor schematic and timing diagram

The figure above shows the 4-bit adder or subtractor schematic diagram along with the timing diagram that shows the operations. The variable a[3:0] was directly linked to the 4-bit adder while variable b[3:0] was placed firstly into a multiplexer along with a buffer. The output of which will then be connected to the 4-bit adder. The results of which will be placed inside the variable out [3:0] along with the carry out.

In the timing diagram, it can be seen that the results operate the subtraction operation since the initial value a is 3 while b stored a value of 1. The result of which was the addition operation of 4 that was stored inside the out[3:0] variable.

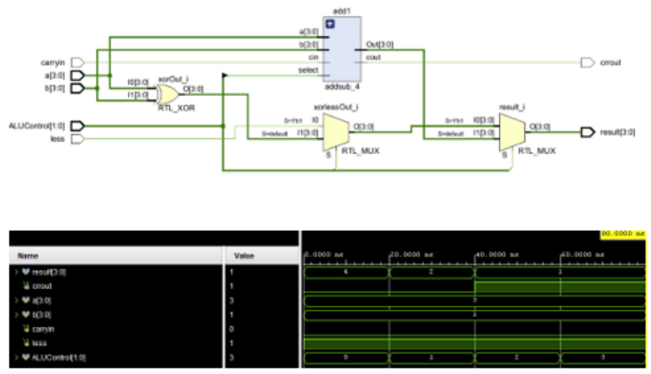
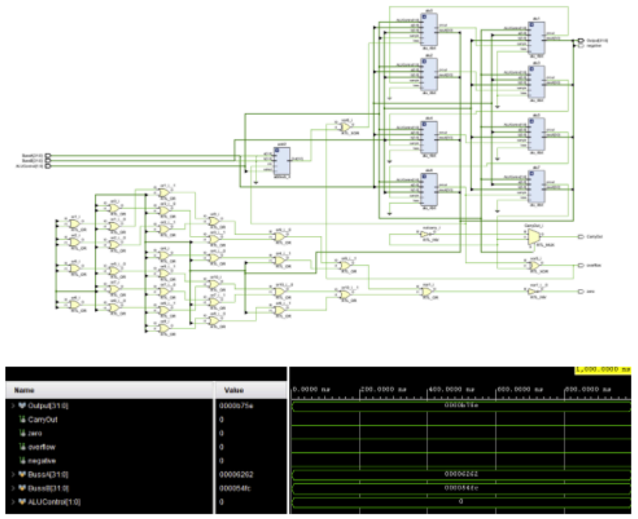


Figure 3: 4-bit Implemented ALU schematic and timing diagram

The figure above shows the implemented 4-bit ALU schematic along with its timing diagram. The schematic shows the addsub module along with XOR gates and different multiplxers. The variables a and b will be firstly moving into the ALU, while at the same time their data will also be transported to the multiplexer. After going through the different gates and modules the results of which will be placed inside the result[3:0] variable.

In the timing diagram, it can be seen that the results operate the subtraction operation since a and b when added results in a 4, while when subtracting them will result in a value of 2. This coincides with the results shown in the result variable. Since the value of result shows a value of 4 then 2, which proves that the created ALU works accordingly.



Name	Value
Output[31:0]	0000b75e
CarryOut	0
zero	0
overflow	0
negative	0
BussA[31:0]	00006262
BussB[31:0]	000054fc
ALUControl[1:0]	0

Figure 4: 32-bit Implemented ALU schematic and timing diagram

The figure above shows the implementation of the 32-bit adder along with its schematic and timing diagram. In the previous figure, it shows only a 4-bit ALU which can operate only up to 4 bits meaning from 0-16. The 32-bit ALU however, can hold up to 4,294,967,296. This greatly increases the strength and capabilities of the ALU within the architecture. In the schematic it can be seen that in order to create a 32-bit ALU, it will need a collection of 8 total 4-bit ALU, meaning that when making a 64-bit ALU, you would need a total of 16 4-bit ALU in order to build it. It can also be seen that the two variables BussA and BussB show the two initial values along with the ALUcontrol. Then it shows multiple orgates connected to an inverter, where the output will be placed in the zero variable. Lastly, after the two inputs were moved inside the 32-bit ALU, the results will then be placed into the output[31:0] variable that can house 32 bits worth of data.

In the timing diagram, it shows that a carries a value of 00006262 which in base 10 is 25186, and b, which has a value of 000054fc which is 21756. In the results shows a hex value of 0000b75e which is 46942 in decimal. This means that the addition operation was done to the two variables since adding a and b will get you 46942, the same results shown inside the output[31:0] variable.

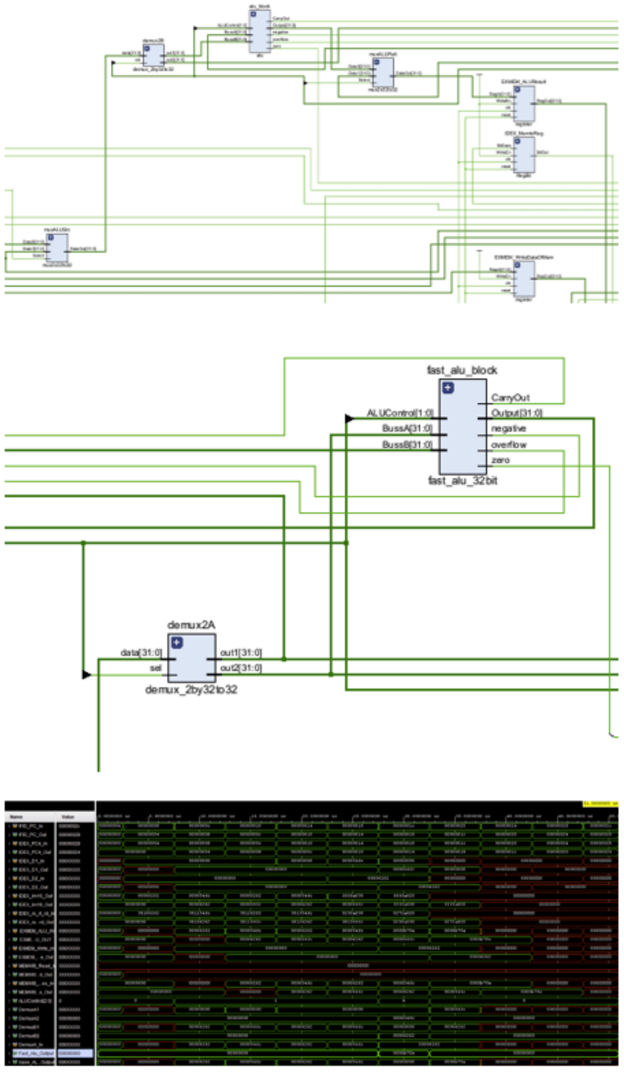


Figure 5: Final system schematic and timing diagram

The figure above shows the final schematic and timing diagram of the total system. The schematic shows ALU block, ALUSrc, demux, ALURsit, memory register, write data memory, and lastly the fast ALU block. In total it clearly shows the full implementation and creation of the different modifications done to the entire architecture.

In the timing diagram, it shows the entire process of the demultiplexer, memory register, ALU control, IFID, IDEX , and the fast ALU. This shows that the entire architecture works accordingly and with the added modifications, improved it in such a way that it will now be able to handle more complex computations due to the added applicability of its ALU.

VI. Results

Test Code

```
xori s0, zero, 0x6262 ; 25186
xori s1, zero, 0x54FC ;21756
xori s2, zero, 0x6262 ;25186
xori s3, zero, 0x54FC ;21756
addf s4, s0, s1
add s5, s2, s3
```

Above shows the test code the group used for testing. It is devised with different XORI operations, add operation, and addf operation. The addf operation, as can be seen, is denoted by the opcode 0x30h, with the addf operation being an instruction implemented in the fast ALU. However, it should be noted that the subtraction operation was not implemented, as it was found out during simulation and testing of the individual components, at larger values, the result is incorrect due to the implementation of obtaining the LSB of the output being determined solely by an exclusive or operation between Cin, input A and input B, which results in some errors when a borrow operation is required. The addition operation proved to be as accurate as the ripple carry adder ALU found within the original implementation of this MIPS pipeline.

Table 1
Recorded Test Data

Adder Impleme ntation	Test 1 time taken	Test 2 time taken	Test 3 time taken	Average
RCA	5.58 ps	5.00 ps	5.55 ps	5.38 ps
CLA	5.20 ps	5. 00 ps	5.21 ps	5.14 ps

Table 2
Calculated Percent Difference

Difference	Percent Difference
0.24 ps	4.46%

The table shown above shows the recorded test data for the RCA and CLA. it can be incurred that on average the time it takes to finish is on average 5.38 ps on the RCA and 5.14 ps on the CLA. This shows that it incurred a 0.24 ps difference. While only minuscule, it still clearly shows that, on average, CLA is 0.24 ps faster than RCA, making a remarkable 4.46 percent difference between them.

VII. Conclusion

The successful simulation of a High-Speed Energy-Efficient Carry Look-Ahead Adder (CLA) using Vivado marks an important step forward in digital computation. This project tackles key challenges in traditional adder designs, especially the delays found in Ripple Carry Adders (RCA). By pre-calculating carry signals, the CLA design enables parallel processing, which significantly cuts down the time required for arithmetic operations. Our simulations showed that this design could achieve about a 4.5% speed improvement over the traditional RCA. While this improvement might seem small, it can make a meaningful difference in more complex systems where every millisecond counts. Additionally, this CLA design shows promise for reducing power consumption, which is crucial in today's energy-conscious world. The results highlight the potential for integrating this efficient design into a wide range of digital and embedded systems. This work emphasizes the need for continued innovation in digital circuit design to further boost performance and energy efficiency in future technologies.

VIII. Recommendations

Advancing Power Efficiency: With the increasing emphasis on energy-efficient electronics, it is imperative to prioritize the reduction of the CLA's power usage. Exploring strategies like Dynamic Voltage and Frequency Scaling (DVFS) and power gating can lead to significant gains in energy efficiency, aligning the system with global sustainability goals.

Integrating into Next-Generation Systems: To fully assess the CLA's performance, it should be incorporated into advanced digital platforms, including cutting-edge microprocessors and specialized architectures. This integration will provide valuable insights into its effectiveness in real-world applications, such as digital signal processing and encryption systems.

Exploring Higher-Radix Number Systems: Research should continue to explore the potential benefits of higher-radix number systems, which can dramatically reduce calculation times for certain operations. Leveraging these systems could further improve the CLA's speed and efficiency, making it a more attractive solution for high-performance computing tasks.

Optimizing and Scaling the Design: Ongoing efforts should focus on optimizing and scaling the CLA design, particularly by increasing the adder's bit-width capabilities. Through careful segmentation and precise mapping of the CLA logic on FPGA structures, the system can achieve greater performance while maintaining a balance between speed and power consumption.

Acknowledgment

The authors would like to express their deepest gratitude to Sir Cesar Llorente, our esteemed professor in Computer Architecture, for his invaluable guidance, support, and encouragement throughout this project. His expertise and insights have been instrumental in shaping our understanding and approach to the implementation of the High-Speed Energy-Efficient Carry Look-Ahead Adder. We are truly grateful for his mentorship and the opportunity to learn under his tutelage.

References

[1] M. King, B. Zhu, and S. Tang, “Optimal path planning,” *Mobile Robots*, vol. 8, no. 2, pp. 520-531, March 2001.

[2] ARM. (n.d.). *Reduced Instruction Set Computer (RISC)*. Retrieved from https://www.arm.com/glossary/risc#:~:text=A%20Reduced%20Instruction%20Set%20Computer,typically%20found%20in%20other%20architectures.

[3] Clements, A. (2003). *MIPS: A microprocessor architecture*. ScienceDirect. Retrieved from https://www-sciencedirect-com.dlsu.idm.oclc.org/science/article/pii/B9781876938604500124

[4] CircuitVerse. (n.d.). *Combinational Logic - Arithmetic Logic Unit (ALU)*. Retrieved from https://learn.circuitverse.org/docs/comb-lsi/alu.html

[5] GeeksforGeeks. (n.d.). *Carry Look-Ahead Adder*. Retrieved from https://www.geeksforgeeks.org/carry-look-ahead-adder/

[6] GeeksforGeeks. (n.d.). *Computer Organization: RISC and CISC*. Retrieved from https://www.geeksforgeeks.org/computer-organization-risc-and-cisc/

[7] Balasubramanian, P., & Mastorakis, N. E. (2022). High-speed and energy-efficient carry look-ahead adder. Journal of Low Power Electronics and Applications, 12(3), 46. https://doi.org/10.3390/jlpea12030046.

[8] Hasan, M., Muhammad Saddam Hossain, Abdul Hasib Siddique, Hossain, M., Zaman, H. U., & Islam, S. (2021). A high-speed 4-bit Carry Look-Ahead architecture as a building block for wide word-length Carry-Select Adder. Microelectronics, 109, 104992–104992. https://doi.org/10.1016/j.mejo.2021.104992

[9] Parhami, Behrooz. Computer Arithmetic : Algorithms and Hardware Designs. New York: Oxford University Press, 2000. Print.