ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
IJARMATE Your uh-MATE Research Paper !!! Vol. 10, Issue 6June 2024

Design and implementation of 32-bit Arithmetic Logic Unit (ALU) with 32 operations

Mrs.G M G Madhuri
Department of Electronic and Communication Engineering
PSCMR College of Engineering and Technology
Vijayawada,India
gmgmadhuri@gmail.com

Prabhu Dasu Saikam
Department of Electronic and Communication Engineering
PSCMR College of Engineering and Technology
Vijayawada,India
prabhudas0123@gmail.com

Bhargavi Koyya
Department of Electronic and Communication Engineering
PSCMR College of Engineering and Technology
Vijayawada,India
bhargavikoyya102@gmail.com

Venkat Kandipalli
Department of Electronic and Communication Engineering
PSCMR College of Engineering and Technology
Vijayawada,India
venkatkandipalli93@gmail.com

Irfan Shaik
Department of Electronic and Communication Engineering
PSCMR College of Engineering and Technology
Vijayawada,India
iamirfan1919@gmail.com

Abstract-

This paper presents the design and implementation of 32-bit arithmetic logic unit capability of executing 32 distinct arithmetic and logic operation based on select lines. utilizing Verilog hardware description language (HDL) the ALU is synthesized on FPGA hardware offering a versatile solution for digital computing tasks. Selection of each operation is facilitated by 5-bit control input, enabling a broad spectrum of arithmetic and logic computations. The project emphasizes the ALU'S functionality and adaptability across various digital computing applications. North worthy features includes clock gating and power control optimization and efficiency utilization of hardware resources. The efficiency and correctness of ALU design are confirmed through stimulation and synthesis results, underscoring its utility and practical scenarios

Keywords: Arithmetic Logic Unit, Verilog HDL, FPGA, Digital Design, Arithmetic Operations, logic Operations, Clock Gating and Pipelining.

I. Introduction

The arithmetic logic unit (ALU) is a fundamental component of digital computing systems responsible for performing arithmetic and logic operations on binary data. ALUs are critical in executing various computational tasks within processes, ranging from basic arithmetic operation to complex logic comparisons [2]. Their significance extends across various computing domains, including digital signal processing, data manipulation and control logic. By efficiently processing binary data, ALUs enable the execution of complex algorithms and facilitate the core functionality of computing devices.

The scope of this project encompasses the design, simulation, synthesis and verification of the 32-bit ALU. Key tasks include defining the ALU architecture, implementing individual operations based on the select lines, ensuring proper clock gating for power optimization and conducting through testing to validate the functionality of each operation. The primary objective of this project is to design and implement a 32-bit ALU using Verilog HDL [5]. The ALU is intended to support a wide range of arithmetic, logical and bit wise operations providing essential functionality for a digital signal processing, data manipulation and control logic within a computer system. Through this endeavor, the objective is to deliver a versatile and efficient ALU solution capable of meeting the computational demands of modern computing systems [8].

The designed ALU module accepts two 32-bit operands (A and B), along with the control signal for selecting the desired operation.

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
IJARMATE Vol. 10, Issue 6June 2024

The ALU operates on a positive edge-triggered clock and delivers the computed results on a 64-bit output bus, ensuring compatibility with a wider range of digital computing system.

II. LITERATURE SURVEY

The literature review for the design project of 32-bit Arithmetic logic unit (ALU) encompasses several central areas pretending to digital logic circuitry and computer architecture. Firstly, a thorough exploration for ALU design technique Various methodologies and architectural approaches employed in constructing efficient ALUs. This includes investigating traditional architecture like ripple carry adders and carry lookahead adders, as well as modern designs incorporating pipelining and parallel processing for enhanced performance and thought put [1].
Moreover, an examination of fundamental digital logic circuits, such as logical gates and flip flops, sheds lights on the underlying components, utilized within ALU architectures. Combinational and sequential logic circuits are studied for their integration into ALU design, contributing to overall functionality and efficiency [7].
Within the context of computer architecture, the review delves into the role of ALU within the processor system and its interaction with other components like registers, control units and memory units. This encompasses an analysis of instruction set architectures (ISAs) and their influence on ALU design principles and the capabilities. Furthermore, optimization techniques aimed at improving ALU performance, area efficiency and power consumption are explored. This entails investigating design automation tools synthesis strategies and verification methodologies essential for the successful deployment and validation of alu designs.
Recent advanced and research trends in ALU design constitute another critical aspect of the literature review, highlight novel approaches and emerging technologies shaping the future of ALU architecture. By examining recent research papers and academic publications, insight into key challenges and future directions in ALU research, including scalability, reliability and support for specialized computation tasks, are gained. Moreover, the review encompasses an analysis of existing solutions and implementation including open-source projects and academic resources, providing reference designs and code examples for educational and research purposes. Through this comprehensive literature review, the project in is equipped with valuable knowledge and resource to inform their ALU design and implementation processes effectively [7]. Comparison of approaches
When comparing different approaches to ALU design, several factors come into play. Traditional approaches like ripple-carry adder offers simplicity and easy implementation, but may lack efficiency in terms of speed and area. On the other hand, carry-look ahead adder provides faster competition by reducing the propagation delay associated with carry propagation.
Modern design incorporating pipelining and parallel processing techniques aim to further enhance performance by breaking down operations into smaller stages or processing multiple operations simultaneously. These approaches often required more complex circuitry but offers significant improvement in throughput [8].
In terms of digital logic circuits. The choice between combinational and sequential logic depends on the nature of operation being performed. Combinational logic circuits are well-suited for operations where the output depends solely on the current input, such as arithmetic operations. Sequential logic circuits on the other hand, introduce memory elements like flip-flops to store intermediate results and maintain state information enabling more complex operations and control logic within the ALU [10].

III. METHODOLOGY

The methodology employed for developing the 32-bit Arithmetic Logic Unit (ALU) involves a systematic approach, processing through several key stages. Initially, the project team conducts an in-depth analysis of ALU requirements, encompassing supported operations, operand widths, and architectural constraints. This analysis informs the creation of a detailed functional specification document, outlining the ALU's intended behavior and capabilities.
Following the conceptualization phase, the design process commences with the definition of the ALU architecture. Factors such as operand width, supported operations, and data path organization are carefully considered. Critical components like adders, multipliers, and logic gates are then selected based on performance and efficiency criteria.
Subsequently, Register Transfer Level (RTL) descriptions of the ALU modules are developed using a hardware description language (HDL) like Verilog or VHDL, capturing the functional behavior and interconnections of the ALU components.
Moving to implantation, the RTL description are translated into synthesizable Verilog code, adhering to coding standards to ensure readability and maintainability. Testbenches are developed to verify the correctness and functionality of the ALU design through simulation-based validation. Various verification techniques, including functional and formal verification, are employed to ensure the design meets specified requirements.
Once verified, the RTL code undergoes logic synthesis to gee rate gate-level netlists, optimizing the design for target FPGA or ASIC technologies. During system design, the synthesized ALU IP core is integrated into the larger system environment, alongside processors, memory, and peripheral interfaces. System-level verification test are conducted to validate correct integration and functionality within the overall system context.

1.PASS

The ALU does not do any arithmetic or logic operations on operand A; instead, it merely passes the value to the output when select equals 5'b00000, which is the PASS operation. Usually, you would run a simulation or test where you change operand A's values and watch the data_out output to make sure the PASS operation is operating as intended. The value of operand A should be reflected in the output data_out when select is set to 5'b00000.

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
Vol. 10, Issue 6June 2024

For instance, if you choose 5'b00000 and set A to 32'h12345678, the output data_out should likewise be 32'h12345678. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can determine that the PASS operation has been confirmed if the output for each test case equals the expected value.

2.INC

Operand A's value is increased by 1 by the ALU when select equals 5'b00001, which is the INC (increment) operation. The result is then output.

You would normally run a simulation or test where you provide operand A different initial values and watch the data_out output to make sure the INC operation is operating as intended. The output data_out should equal the value of operand A multiplied by one when select is set to 5'b00001.

For instance, the output data_out should be 32'h00000002 if you set A to 32'h00000001 and choose 5'b00001. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the INC operation verified if the output for each test case equals the expected value.

3.DEC

Operand A's value is decreased by 1 by the ALU when select equals 5'b00010, which is the DEC (decrement) operation. The result is then output.

You would normally run a simulation or test where you provide operand A different initial values and watch the data_out output to make sure the DEC operation is operating as intended. The output data_out should equal the value of operand A decremented by 1 when select is set to 5'b00010.

For instance, the output data_out should be 32'h00000001 if you set A to 32'h00000002 and choose 5'b00010. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare that the DEC operation is validated if the output for each test case equals the expected value.

4.ADD

The ADD (addition) operation is represented by select equal to 5'b00011, at which point the ALU adds the values of operands A and B and output the results. Usually, you would run a test or simulation in which you set different initial values for operands A and B and watch the data_out output to confirm that the ADD operation is working as intended. The output data_out should equal the sum of operands A and B when select is set to 5'b00011.

For instance, the output data_out should be 32'h00000005 (2 + 3 = 5) if you set A to 32'h00000002, B to 32'h00000003, and choose to 5'b00011. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the ADD operation verified if the result for each test case equals the expected value.

5.SUB

The SUB (subtraction) operation is represented by select equal to 5'b00100. In this case, the ALU subtracts operand B's value from operand A and outputs the result. Usually, you would run a simulation or test in which you set different initial values for operands A and B and watch the data_out output to confirm that the SUB operation is operating as intended. The output data_out should equal the difference between operand A and operand B when select is set to 5'b00100.

For instance, the output data_out should be 32'h00000002 (5 - 3 = 2) if you set A to 32'h00000005 and B to 32'h00000003, and choose to 5'b00100. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the SUB operation verified if the output for each test case equals the expected value.

6.MUL

The MUL (multiplication) operation is represented by select equal to 5'b00101. In this case, the ALU multiplies the values of operands A and B before outputting the result. Usually, you would run a simulation or test where you change the initial values for operands A and B and watch the data_out output to make sure the MUL operation is working as intended. The output data_out should equal the product of operands A and B when select is set to 5'b00101.

For instance, the output data_out should be 32'h0000000F (5 * 3 = 15) if you set A to 32'h00000005 and B to 32'h00000003, and choose to 5'b00101. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can determine that the MUL operation is validated if the output for each test case equals the expected value.

7.DIV

The ALU divides operand A's value by operand B's value and outputs the quotient when select equals 5'b00110, which

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

IJARMATE
International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
Vol. 10, Issue 6June 2024

is the DIV (division) operation.

Usually, you would run a simulation or test in which you set different initial values for operands A and B and watch the data_out output to confirm that the DIV operation is operating as intended. The output data_out should equal the quotient of operand A divided by operand B when select is set to 5'b00110.

The output data_out should be 32'h00000003 (15 / 3 = 3), for instance, if you set A to 32'h0000000F and B to 32'h00000003, and choose to 5'b00110. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the DIV operation verified if the output for each test case equal to the expected value.

8.MODULUS

The ALU calculates and outputs the remainder that results from dividing the value of operand A by the value of operand B when select equals 5'b00111, which is equivalent to the MOD (modulus) operation.

Usually, you would run a simulation or test in which you set different initial values for operands A and B and watch the data_out output to confirm that the MOD operation is operating as intended. When select is set to 5'b00111, the output data_out should be equal to the remainder of operand A divided by operand B.

The output data_out should be 32'h00000000 (15 % 3 = 0), for instance, if you set A to 32'h0000000F and B to 32'h00000003, and choose to 5'b00111. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the MOD operation verified if the output for each test case equals the expected value.

9.ADC

The ALU computes and outputs the sum of operand A, operand B, and the carry-in Cin when select equals 5'b01000, which is equivalent to the ADC (Addition with Carry) operation.

In order to confirm that the ADC operation is operating as intended, you would usually run a test or simulation in which you change the initial values for operands A, B, and Cin and then watch the data_out output. The output data_out should equal the sum of operands A, B, and Cin when select is set to 5'b01000.

For instance, the output data_out should be 32'h0000000D (5 + 7 + 1 = 13) if you set A to 32'h00000005, B to 32'h00000007, Cin to 1 (signifying a carry-in), and select to 5'b01000. By simulating your Verilog module with varying input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the ADC operation verified if the output for each test case equals the expected value.

10.2’S COMPLIMENT

The ALU calculates operand A's two's complement and outputs the result when select equals 5'b01001, which is the TWOC (Two's complement) operation. In order to confirm that the TWOC operation is operating as intended, you would normally run a test or simulation in which you change operand A's initial values and watch the data_out output. The output data_out should equal operand A's two's complement when select is set to 5'b01001.

For instance, if you set A to 32'h00000005, then 32'hFFFFFFFB is the two's complement of 5, which is -5. Thus, 32'hFFFFFFFB should be the output data_out when select is set to 5'b01001. By emulating your Verilog module with various operand A input values and determining whether the output corresponds to the anticipated outcomes, you can confirm this behaviour. You can declare the TWOC operation verified if the output for each test case equals the expected value.

11.OR

The ALU computes the bitwise OR operation between operands A and B and outputs the result when select equals 5'b01010, which is the OR operation.

Usually, you would run a simulation or test and observe the data_out output after changing the initial values of operands A and B to confirm that the OR operation is working as intended. A bitwise OR operation between A and B should produce the output data_out when select is set to 5'b01010.

The outcome of the bitwise OR operation between A and B would be 32'h000000FF (binary: 0000 0000 0000 0000 0000 0000 1111 1111), for instance, if you set A to 32'h0000000F (binary: 0000 0000 0000 0000 0000 0000 0000 1111) and B to 32'h000000F0 (binary: 0000 0000 0000 0000 0000 0000 0000 1111 0000). Consequently, the output data_out should be 32'h000000FF when select is set to 5'b01010.

12.AND

The ALU computes the bitwise AND operation between operands A and B and outputs the result when select equals 5'b01011, which represents the AND operation. You would normally run a simulation or test where you provide operands A and B different initial values and watch

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
Vol. 10, Issue 6June 2024

the data_out output to make sure the AND operation is working as intended. A bitwise AND operation between A and B should produce the output data_out when select is set to 5'b01011.
For instance, if you set A to 32'h0000000F (binary: 0000 0000 0000 0000 0000 0000 0000 1111) and B to 32'h000000F0 (binary: 0000 0000 0000 0000 0000 0000 1111 0000), then 32'h00000000 (binary: 0000 0000 0000 0000 0000 0000 0000 0000) would be the outcome of the bitwise AND operation between A and B. Therefore, the output data_out should be 32'h00000000 when select is set to 5'b01011.

13.NOT

The ALU computes the bitwise NOT operation on operand A and outputs the result when select equals 5'b01100, which is equivalent to the NOT operation. In order to confirm that the NOT operation is operating as intended, you would usually run a test or simulation in which you change operand A's initial values and watch the data_out output. The output data_out should be the outcome of bitwise NOTing A when select is set to 5'b01100.
A bitwise NOT operation on A would result in 32'hFFFFFFF0 (binary: 1111 1111 1111 1111 1111 1111 0000), for instance, if you set A to 32'h0000000F (binary: 0000 0000 0000 0000 0000 0000 0000 1111). Thus, 32'hFFFFFFF0 should be the output data_out when select is set to 5'b01100.
.

14.NOR

The ALU computes the bitwise NOR operation on operands A and B and outputs the result when select equals 5'b01101, which is the NOR operation.
In a typical simulation or test, you would give operands A and B different initial values and watch the data_out output to make sure the NOR operation is working as intended. A bitwise NOR operation on A and B should produce the output data_out when select is set to 5'b01101.
The outcome of the bitwise NOR operation on A and B would be 32'hFFFFFF00 (binary: 1111 1111 1111 1111 1111 0000 0000) if, for instance, you set A to 32'h0000000F (binary: 0000 0000 0000 0000 0000 0000 0000 1111) and B to 32'h000000FF (binary: 0000 0000 0000 0000 0000 0000 1111 1111). Consequently, the output data_out should be 32'hFFFFFF00 when select is set to 5'b01101.

15.NAND

The ALU computes the bitwise NAND operation on operands A and B and outputs the result when select equals 5'b01110, which is the NAND operation.
Usually, you would run a simulation or test and observe the data_out output after changing the initial values of operands A and B to confirm that the NAND operation is working as intended. A bitwise NAND operation on A and B should produce the output data_out when select is set to 5'b01110.
The outcome of the bitwise NAND operation on A and B would be 32'hFFFFFF00 (binary: 1111 1111 1111 1111 1111 0000 0000) if, for instance, you set A to 32'h0000000F (binary: 0000 0000 0000 0000 0000 0000 0000 1111) and B to 32'h000000FF (binary: 0000 0000 0000 0000 0000 0000 1111 1111). Thus, 32'hFFFFFF00 should be the output data_out when select is set to 5'b01110.

16.EX-OR

The ALU computes the bitwise XOR operation on operands A and B and outputs the result when select equals 5'b01111, which is the XOR operation.
Usually, you would run a simulation or test and observe the data_out output after changing the initial values of operands A and B to make sure the XOR operation is working as intended. The output data_out should be the outcome of bitwise XORing A and B when select is set to 5'b01111.
For instance, if you set A to 32'h0000000F (binary: 0000 0000 0000 0000 0000 0000 1111 1111) and B to 32'h000000FF (binary: 0000 0000 0000 0000 0000 0000 1111 1111), then 32'h000000F0 (binary: 0000 0000 0000 0000 0000 0000 1111 0000) would be the outcome of the bitwise XOR operation on A and B. Thus, 32'h000000F0 should be the output data_out when select is set to 5'b01111.

17.XNOR

The ALU calculates the bitwise XNOR operation on operands A and B and outputs the result when select equals 5'b10000, which is the XNOR operation.

Usually, you would run a simulation or test and observe the data_out output after changing the initial values of operands A and B to make sure the XNOR operation is working as intended. The output data_out should be the outcome of bitwise XNORing A and B when select is set to 5'b10000.
When A and B are set to 32'h0000000F and 32'h000000FF, respectively (binary: 0000 0000 0000 0000 0000 0000 1111 1111), for instance, the outcome of the bitwise XNOR operation on A and B is 32'hFFFFFFFF (binary: 1111 1111 1111 1111 1111 1111 1111 1111). Thus, the output data_out should be 32'hFFFFFFFF when select is set to 5'b10000.

18.BOOLEAN AND

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
IJARMATE Vol. 10, Issue 6June 2024

The Boolean AND operation on operands A and B is computed by the ALU and output when select equals 5'b10001, which is the B_AND operation. You would normally run a simulation or test where you provide different initial values for operands A and B and watch the data_out output to make sure the B_AND operation is working as intended. A Boolean AND operation on A and B should produce the output data_out when select is set to 5'b10001.

For instance, setting A to 32'hFFFFFFFF (binary: 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111) and B to 32'h0000FFFF (binary: 0000 0000 0000 0000 1111 1111 1111 1111), for example, would result in 32'h0000FFFF (binary: 0000 0000 0000 0000 1111 1111 1111 1111 1111). This would be the outcome of the Boolean AND operation on A and B. Therefore, the output data_out should be 32'h0000FFFF when select is set to 5'b10001.

19.BOOLEAN NAND

The Boolean NAND operation is computed by the ALU on operands A and B, and the result is output when select equals 5'b10010, which is the B_NAND operation. You would normally run a simulation or test where you provide operands A and B different initial values and watch the data_out output to make sure the B_NAND operation is working correctly. The output data_out should be the outcome of a Boolean NAND operation on A and B when select is set to 5'b10010.

For instance, if you set A to 32'hFFFFFFFF (binary: 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111) and B to 32'h0000FFFF (binary: 0000 0000 0000 0000 1111 1111 1111 1111), then 32'hFFFF0000 (binary: 1111 1111 1111 1111 0000 0000 0000) would be the outcome of the Boolean NAND operation on A and B. Thus, 32'hFFFF0000 should be the output data_out when select is set to 5'b10010.

20.BOOLEAN OR

The Boolean OR operation is computed by the ALU on operands A and B, and the result is output when select equals 5'b10011, which is the B_OR operation.

In a typical simulation or test, you would give operands A and B different initial values and watch the data_out output to make sure the B_OR operation is working as intended. The output data_out should be the outcome of a Boolean OR operation on A and B when select is set to 5'b10011.

As an illustration, if you set A to 32'hFFFFFFFF (binary: 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111) and B to 32'h0000FFFF (binary: 0000 0000 0000 0000 1111 1111 1111 1111), then 32'hFFFFFFFF is the outcome of the Boolean OR operation on A and B. Thus, 32'hFFFFFFFF should be the output data_out when select is set to 5'b10011.

21.BOOLEAN NOR

The Boolean NOR operation is computed by the ALU on operands A and B, and the result is output when select equals 5'b10100, which is the B_NOR operation. In a typical simulation or test, you would give operands A and B different initial values and watch the data_out output to make sure the B_NOR operation is working as intended. The output data_out should be the outcome of a Boolean NOR operation on A and B when select is set to 5'b10100.

For instance, if you set A to 32'hFFFFFFFF (binary: 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111) and B to 32'h0000FFFF (binary: 0000 0000 0000 0000 1111 1111 1111 1111), then 32'h00000000 (binary) would be the outcome of the Boolean NOR operation on A and B. Hence, the output data_out should be 32'h00000000 when select is set to 5'b10100.

22.BOOLEAN XOR

The Boolean Exclusive OR (XOR) operation on operands A and B is computed by the ALU and output when select equals 5'b10101, which is equivalent to the B_EX_OR operation. You would normally run a simulation or test where you provide different initial values for operands A and B and watch the data_out output to make sure the B_EX_OR operation is working correctly. The output data_out should be the outcome of a Boolean XOR operation on A and B when select is set to 5'b10101.

For instance, setting A to 32'hAAAAAAAA (binary: 1010 1010 1010 1010 1010 1010 1010 1010) and B to 32'h55555555 (binary: 0101 0101 0101 0101 0101 0101 0101 0101), for instance, would result in 32'hFFFFFFFF (binary: 1111 1111 1111 1111 1111 1111 1111 1111). Thus, 32'hFFFFFFFF should be the output data_out when select is set to 5'b10101

23.BOOLEAN XNOR

The ALU computes the Boolean Exclusive NOR (XNOR) operation on operands A and B and outputs the result when select equals 5'b10110, which corresponds to the B_EX_NOR operation.

You would normally run a simulation or test where you provide different initial values for operands A and B and watch the data_out output to make sure the B_EX_NOR operation is working correctly. The output data_out should be the outcome of doing a Boolean XNOR operation on A and B when select is set to 5'b10110

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
Vol. 10, Issue 6June 2024

For instance, the outcome of the Boolean XNOR operation on A and B would be 32'h00000000 (binary: 0000 0000 0000 0000 0000 0000 0000 0000) if you set A to 32'hAAAAAAAA (binary: 1010 1010 1010 1010 1010 1010 1010 1010) and B to 32'h55555555 (binary: 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101). Therefore, the output data_out should be 32'h00000000 when select is set to 5'b10110.

24.LEFT SHIFT

[6] proposed a system, this paper presents an effective field programmable gate array (FPGA)-based hardware implementation of a parallel key searching system for the brute-force attack on RC4 encryption. The design employs several novel key scheduling techniques to minimize the total number of cycles for each key search and uses on-chip memories of the FPGA to maximize the number of key searching units per chip.. Compared to current designs, SEDAAF uses 25% less power and has a power delay product that is 17% lower.

25.RIGHT SHIFT

The ALU shifts the operand A's bits one position to the right using a logical right shift operation when select equals 5'b11000, which is the same as the R_SHIFToperation.

You can run a test or simulation in which you change the operand A's initial values and watch the data_out output to confirm that the R_SHIFT operation is operating as intended. The output data_out should be the outcome of moving the bits of A one position to the right when select is set to 5'b11000. For instance, the logical right shift operation on A would produce 32'h55555555 (binary: 0101 0101 0101 0101 0101 0101) if you set A to 32'hAAAAAAAA (binary: 1010 1010 1010 1010 1010 1010 1010 1010). Consequently, the output data_out should be 32'h55555555 when select is set to 5'b11000.

26.COMPARATOR

The COMP operation is represented by select equal to 5'b11001, at which point the ALU compares the operands A and B. You can run a test or simulation in which you change the values of operands A and B and watch the data_out output to confirm that the COMP operation is operating as intended. Whether or not A equals B should be indicated in the output data_out.

The output data_out should be 1'b1, for instance, if you set A to 32'h00000001 and B to 32'h00000001, showing that A and B are equal. In contrast, the output data_out should be 1'b0 if A is set to 32'h00000001 and B is set to 32'h00000000, indicating that A is not equal to B.

27.PARITY CHECKER

The ALU checks the operand A for parity when select equals 5'b11010, which is the result of the PARITY_CHECK operation.
In order to confirm that the PARITY_CHECK operation is operating as intended, you can run a test or simulation in which you can vary the value of operand A, which is a binary number, and then watch the data_out output. Whether or not there are even or odd number of set bits (ones) in A should be indicated by the output data_out.
For instance, the output data_out should be 1'b1 if you set A to 32'h00000001, which has one set bit, indicating that A has odd parity. In contrast, the output data_out should be 1'b0 if you set A to 32'h00000003, which has two set bits, indicating that A has even parity.

28.PARITY GENERATOR

The ALU creates a parity bit for the operand A when select equals 5'b11011, which is the result of the PARITY_GEN operation.
You can run a test or simulate the PARITY_GEN operation by changing the values of operand A, which is a binary number. This will allow you to confirm that the operation is operating as intended. You can see the output data_out, which should contain A followed by its generated parity bit, after setting the value of A.
The output data_out should be 33'h0000000101, indicating A followed by an additional bit (1'b1) indicating the generated parity, if, for instance, you set A to 32'h00000001, which has one set bit. Similarly, the output data_out should be 33'h0000000300, indicating A followed by 1'b0 as the generated parity, if you set A to 32'h00000003, which has two set bits.

29.BINARY TO GRAY

Operand A's binary representation is transformed into its corresponding Grey code representation by the ALU when select equals 5'b11100, which is associated with the BIN TO GRAY operation. You can run a test or simulate the BIN2GRAY operation by changing the values of operand A, which is a binary number. This will allow you to confirm that the operation is working as intended. You can see the output data_out, which should have the Grey code representation of A, after setting the value of A.

For instance, the output data_out should be 32'h00000001, indicating that the Grey code representation of 1 is also 1, if you set A to 32'h00000001, the binary representation of 1. Similarly, the output data_out should be 32'h00000011, indicating that the Grey code representation of 2 is 3, if you set A to 32'h00000010, which is the binary representation of 2.

30.GRAY TO BINARY

The Grey code representation of operand A is transformed into its corresponding binary representation by the ALU when select equals 5'b11101, which is associated with the GRAY2BIN operation.
Operand A is a Grey code number. You can run a test or simulation in which you enter different values for operand A to confirm that the GRAY2BIN operation is working as intended. You can see the output data_out, which should have the binary representation of the matching Grey code number, after setting the value of A.
For instance, the output data_out should be 32'h00000001, indicating that the binary representation of the Grey code 1 is also 1, if you set A to 32'h00000001, which is the Grey code representation of 1. Similarly, the output data_out should be 32'h00000010 if you set A to 32'h00000011, which is the Grey code representation of 3. This indicates that the binary representation of the Grey code 3 is 2.

31.CLOCK DIVIDER

Clock division is the behaviour when the select input is set to 5'b11110, which is equivalent to the CLK_DIV operation in the previously provided Verilog code. Starting Point:
The clock divider input, clk_div, is sent to the module.
There are additional inputs available, including clk, A, B, select, and Cin.
Logic of Clock Division:
The always block, which is responsive to the clock's positive edge (posedge clk), is where the clock division logic is implemented.
The clock division operation is enabled when 5'b11110, or CLK_DIV, is entered into the select input.
The code within the always block performs the clock division logic if the select input corresponds to clock division and the ALU is active (alu_clk_enable is high). The clock division logic is merely a placeholder (data_out <= {32'b0, A};) and not an actual implementation in the code that is provided. As a result, the input is effectively passed through without any clock division because the output data_out stays the same as the input A. Results:
If the select input is set to 5'b11110, the output data_out will be the same as the input A.

32.MANCHESTER ENCODING

Manchester coding is carried out by the Verilog code when the select input is 5'b11111, which is equivalent to the MANCHESTER operation. Manchester coding uses transitions inside the bit period to represent each bit in the encoding.
Starting Point:
Initialization is done for variables such as encoded_data, bit_index, and previous bit.
The encoded Manchester data is kept in a register called encoded_data.
The counter bit_index is used to record the bit that is currently being processed.
The value of the previously encoded bit is stored in previous_bit.
Manchester Encoding Theory:
[4] proposed a system, this paper presents an effective field programmable gate array (FPGA)-based hardware implementation of a parallel key searching system for the brute-force attack on RC4 encryption. The design employs several novel key scheduling techniques to minimize the total number of cycles for each key search and uses on-chip memories of the FPGA to maximize the number of key searching units per chip.

Every clock cycle, this process is repeated, encoding every bit of the input A data into Manchester-coded format.
Results:
Lastly, the encoded Manchester data that is kept in encoded_data is updated in the output data_out.

IV. DESIGNE AND IMPLENTATION OF ALU

In this section, we describe the design and implementation of the 32-bit ALU using Verilog HDL on Artix 7 FPGA. We first define the input and output ports of the ALU, then we explain the operation codes and the internal modules, and finally we present the test bench used to verify the functionality of the ALU.

A. Inputs and output ports

The ALU has two input ports and one output port, the input ports are data_in and select. The Output port is data_out. The data_in port is a 32-bit wire that represents the input operand for the ALU. The select port is a 5-bit wire that represents the operation code for the ALU. The clk port is a 1-bit wire that represents the clock signal for the ALU. The

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
Vol. 10, Issue 6June 2024

data_out port is a 64-bit reg that represents the output result of the ALU.

B. Operation cod

The ALU supports 32 different operations, as listed in Table 1. Each operation has a corresponding 5-bit operation code, which is assigned to the select port. The ALU is designed using a modular approach, where each operation is implemented as a separate case in a combinational always block. The always block is triggered by the rising edge of the clock port, and assigns the appropriate value to the data_out port, depending on the operation code.

TABLE 1: INSTRUCTION SET OF ALU

Select Line Input	Operation	Bit Width
00000	PASS	32
00001	INC	32
00010	DEC	32
00011	ADD	32
00100	SUB	32
00101	MUL	32
00110	DIV	32
00111	MODULUS	32
01000	ADC	32
01001	2'S COMP	32
01010	OR	32
01011	AND	32
01100	NOT	32
01101	NOR	32
01110	NAND	32
01111	EX-OR	32
10000	EX-NOR	32
10001	B_AND	32
10010	B_NAND	32
10011	B_OR	32
10100	B_NOR	32
10101	B_EX_OR	32
10110	B_EX_NOR	32
10111	LEFT SHIFT	32
11000	RIGTH SHIFT	32
11001	COMPARATOR	32
11010	PARITY CHECKER	32
11011	PARITY GENERATOR	32
11100	BIN TO GRAY	32
11101	GRAY TO BIN	32
11110	CLK_DIV	32
11111	MANCHESTER	32
Data_out	Output	64

RESULTS AND SIMULATIONS

1 UNIT TESTING:

Each operation supported by ALU module was individually tested to verify its correctness and adherence to the specified behaviour. Test cases were designed to cover different input combinations and edge cases for each operation, ensuring comprehensive coverage. Assertion and monitors were utilized within the Verilog Testbench to check the expected output against the actual output produced by the ALU module.

2 INTEGRATION TESTING:

The ALU module was integrated into layer system or environment to assess its compatibility and interoperability with other components. Integration test were conducted to evaluate the ALU's interaction with external modules, such as clock generators, input/output interfaces and memory units. Compatibility tests were performed to ensure seamless integration and communication between the ALU and the rest of the system.

3 FUNCTIONAL TESTING:

Comprehensive functional testing was conducted to validate the ALU compliance with the specified requirements and operation codes. Test cases covered a wide range of operations, including arithmetical operations (addition, subtraction, multiplication, division), logical operations (AND, OR, XOR), shift operations (left shift, right shift) and special operations (comparator, parity Checker). Input stimuli were provided to the ALU module through the test bench, and the resulting output were compared against expected values to verify correctness.

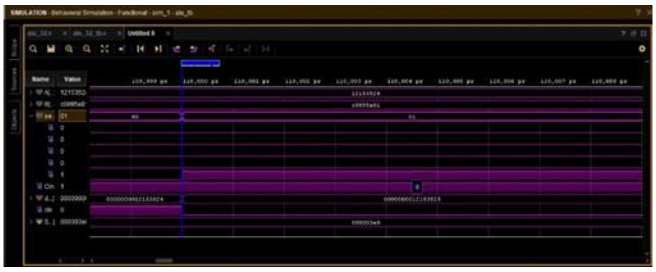
Fig: PASS AND INCREMENT

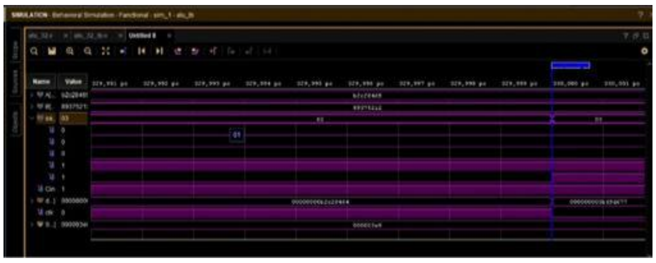
Fig: DECREMENT

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at
www.ijarmate.com

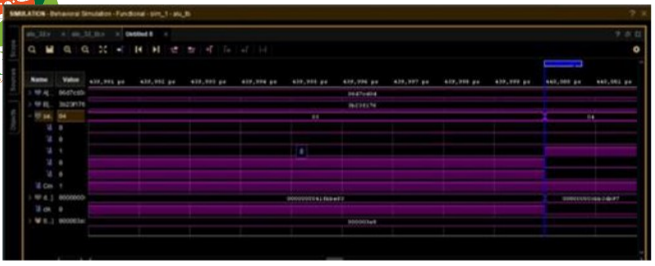

Fig: ADDITION

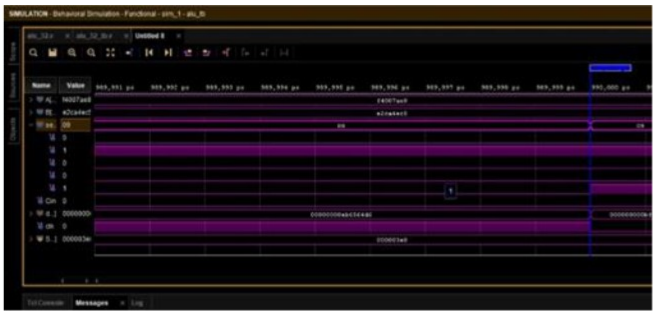

Fig: ADDITION WITH CARRY

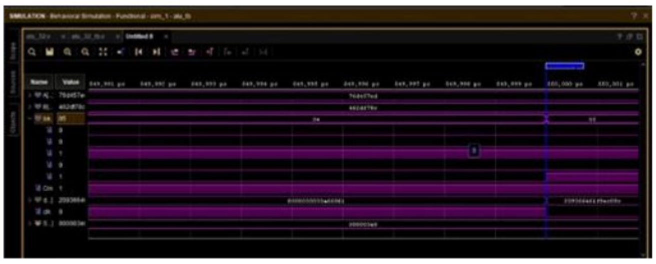

Fig: SUBTRACTION

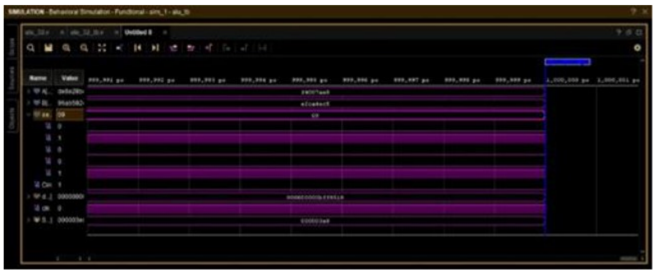

Fig: 2'S COMPLIMENT

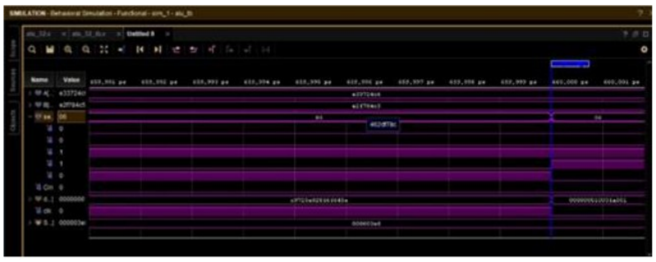

Fig: MULTIPLICATION

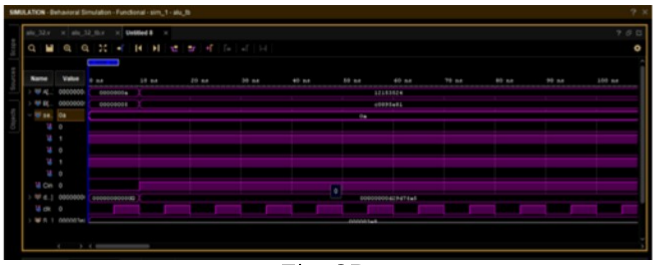

Fig: OR

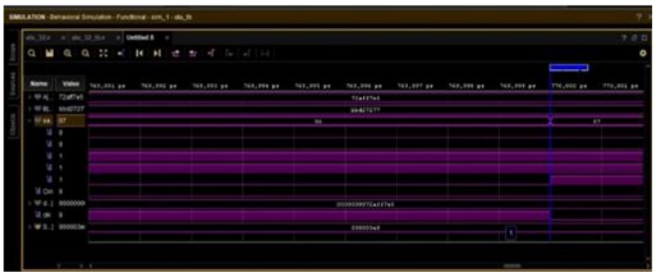

Fig: DIVISION

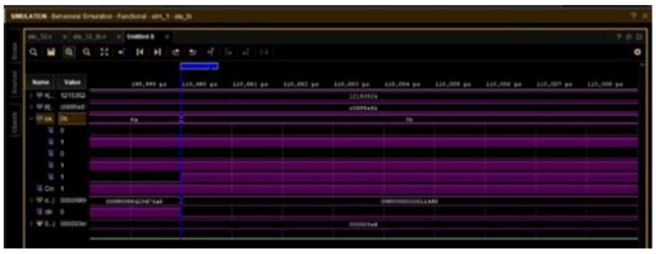

Fig: AND

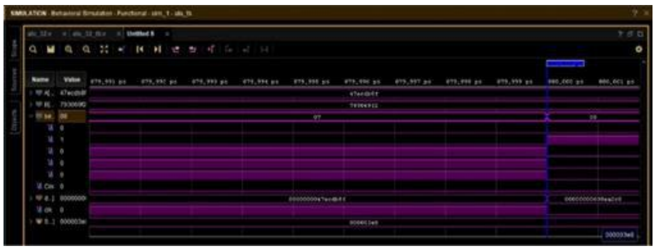

Fig: MODULUS

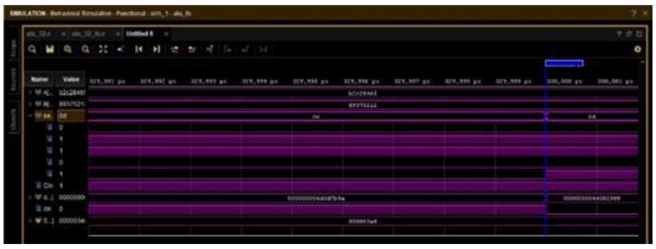

Fig: NOT

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at
www.ijarmate.com

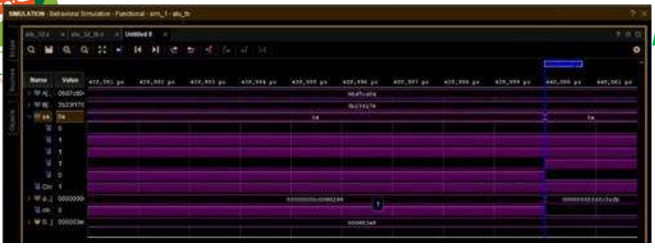
Fig: NOR

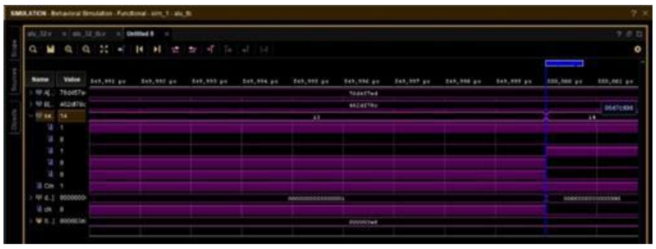
Fig: B_OR

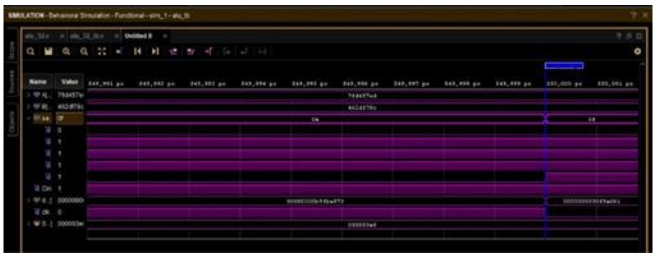
Fig: NAND

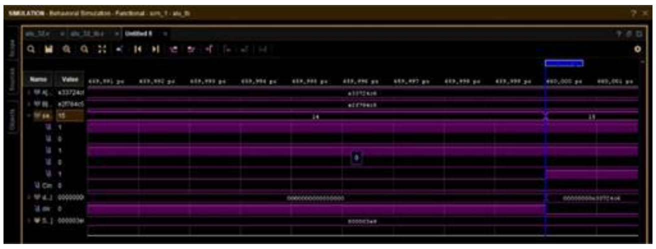
Fig: B_NOR

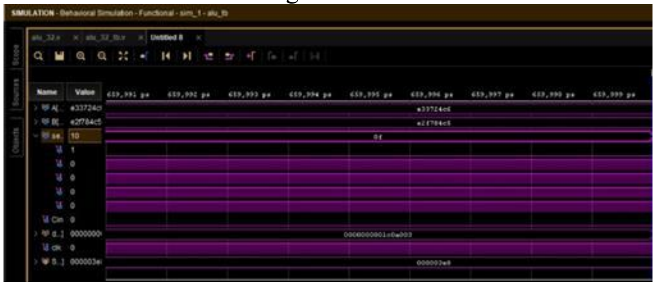
Fig: EX_OR

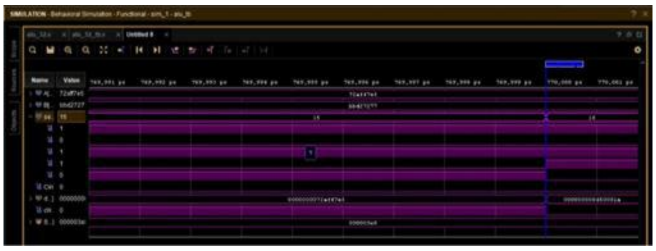
Fig: B_EX_OR

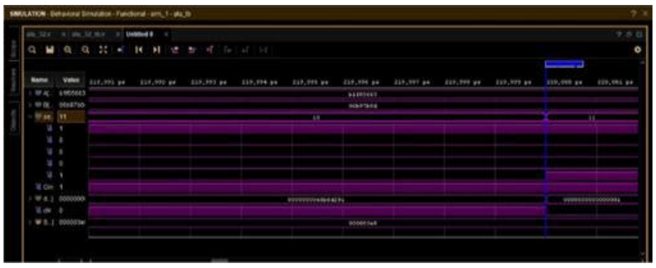
Fig: EX-NOR

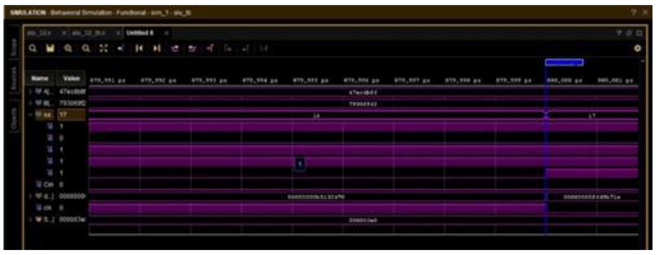
Fig: B_EX_NOR

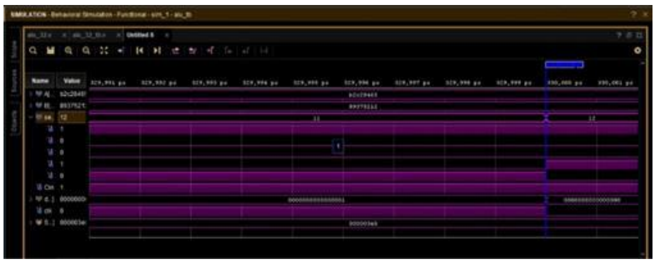
Fig: B_AND

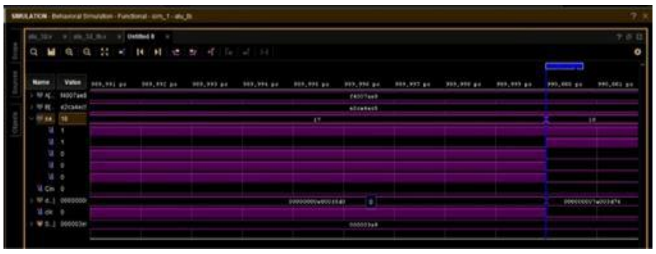
Fig: LEFT SHIFT

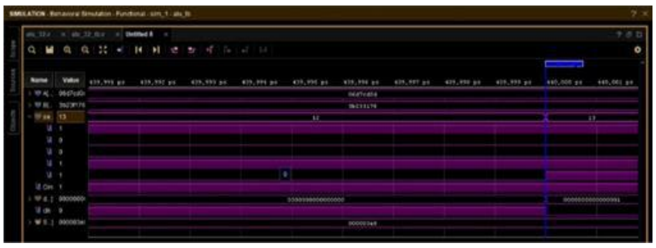
Fig: B_NAND

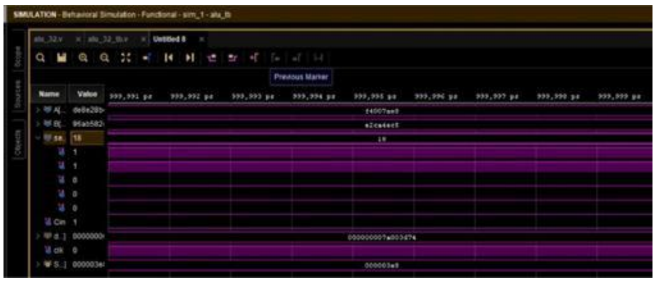
Fig: RIGHT SHIFT

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

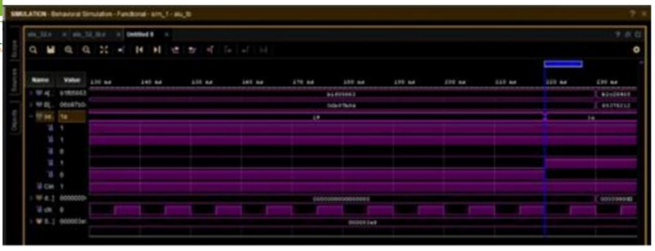
Fig: COMPARATOR

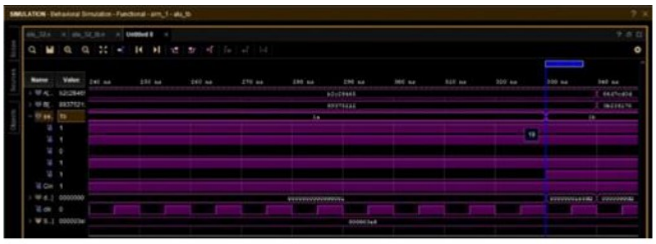
Fig: PARITY CHECKER

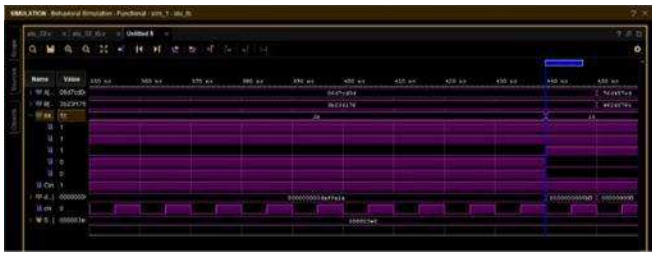
Fig: PARITY GENERATOR

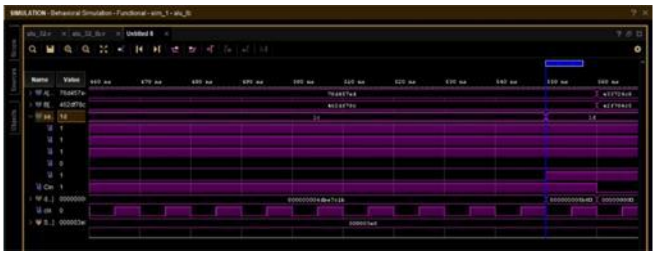
Fig: BINARY TO GRAY

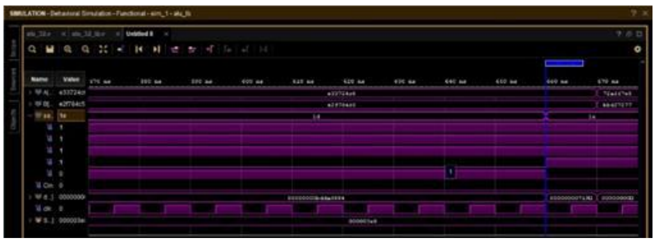
Fig: GRAY TO BINARY

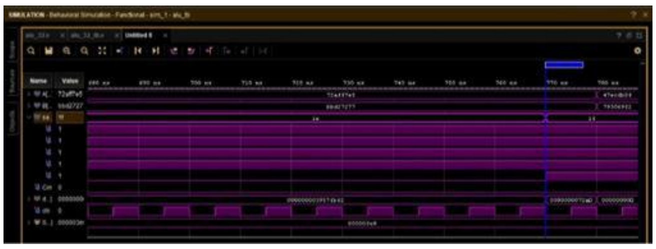
Fig: CLK_DIV

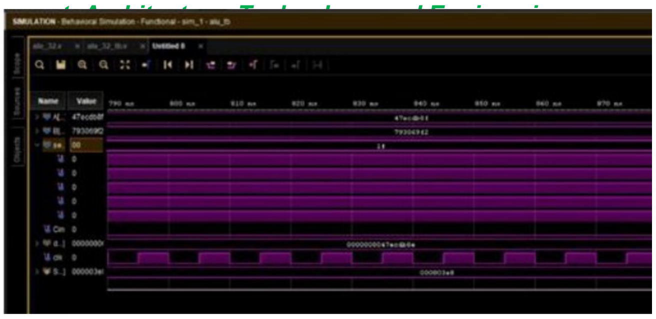
Fig: MANCHESTER

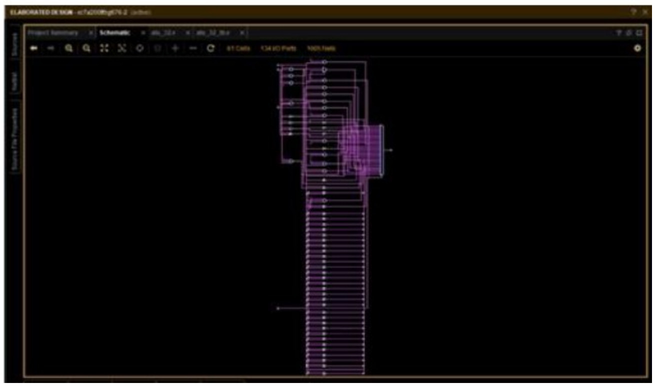
Fig: SCHEMATIC

In this research, we described the design and implementation of an Artix 7 FPGA-based 32-bit ALU using Verilog HDL. 32 distinct operations are supported by the ALU, some of which are new and beneficial to cryptographic processors. With a modular and hierarchical design, the ALU implements each operation as a distinct case within a combinational always block. Vivado 2018.1 is used to simulate and synthesise the ALU, and a test bench is used to confirm its functionality.

Based on the schematic results, the ALU has 1005 nets, 134 I/O ports, and 61 cells. The ALU can be used with the IEEE 754 floating-point standard.

mathematical situations, including overflow, underflow, and division by zero. Additionally, the ALU has flags for carry, negative, and zero outcomes. The suggested ALU design has a number of benefits over the current one, including minimal complexity and size, easy scalability and reuse, and support for a broad variety of operations and data types. Applications such as digital signal processing, wireless communication, and embedded systems that call for flexible and high-performance arithmetic and logic operations can all benefit from the suggested ALU design.

VI. CONCLUSION

In conclusion, the development of a 32-bit arithmetic logic unit presented in this paper showcases the efficiency and versatility of hardware-based solution for digital computing applications. By leveraging the capabilities of Verilog hardware description language and FPGA hardware, we have successfully implemented an alu capable of executing 32 distinct arithmetic and logical operations based on select lines.

ISSN (ONLINE): 2454-9762
ISSN (PRINT): 2454-9762
Available online at www.ijarmate.com

International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE)
Vol. 10, Issue 6June 2024

The flexibility provided by the 5-bit control input enables a wide range of arithmetic and logical computations, making the alu suitable for diverse digital computing task. Whether it's performing basic arithmetic operations, bitwise logic operations, or more complex competitions, the alu offers a reliable and efficiency solution.

Moreover, the incorporation of features such as clock gating for power optimization and efficient resource utilization highlights our commitment to enhance performance while minimising energy consumption hardware footprint.

Simulation and synthesis results have validated the correctness and efficiency of the ALU design, confirming its suitability for Practical deployment in various digital computing scenarios. This project underscores the importance of hardware-based implementation in meeting the computational demands of modern applications.

Looking ahead, further optimisation and refinement of the ALU design cloud yield even greater performance improvements and expand its applicability across a broader spectrum of digital computing domains. As technology continue to evolve, hardware solutions like ALU presented here will remain crucial component in driving innovation and progress in the field of digital computing.

VII. FUTURE WORKS

Future development on this topic could include the following:

- putting into practice the difficult conversion process from grey to binary, which calls for a different module.
- employing strategies like clock gating and pipelining to optimise the ALU design for speed and power usage.
- combining the ALU with additional parts, including a control unit, memory, and register file, to create a whole processor.

VIII. REFERENCES

[1] P. A. David and H.L. John, "Computer Organization and Design", Morgan Kaufmann, 2005.

[2] D. Michel, A. Murali and S. Per ,"Parallel Computer and Organization Design", Cambridge University Press, 2012.

[3] D. Neha, G. Nidhi, M. Anu "Hardware Efficient AES for Image Processing with High Throughput", IEEE conference on NGCT 2015, Sep 2015, pp 932-935.

[4] Christo Ananth, Muthamil Jothi.M, M.Priya, V.Manjula, "Parallel RC4 Key Searching System Based on FPGA", International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE), Volume 2, Special Issue 13, March 2016, pp: 5-12.

[5] G. Jeong and J. Park, "Design of 32-bit RISC Processor and efficient verification", Proceedings of the 7th Korea-Russia Infernational Symposium. KORUS, 2003, pp. 222-227.

[6] Christo Ananth, Muthamil Jothi.M, M.Priya, V.Manjula, "Parallel RC4 Key Searching System Based on FPGA", International Journal of Advanced Research in Management, Architecture, Technology and Engineering (IJARMATE), Volume 2, Special Issue 13, March 2016, pp: 5-12.

[7] Y. Kui and D. Yue-Hua., "32 Bit multiplication and division ALU design based on RISC structure", International Joint Conference on Artificial Intelligence, 2009, p.p. 761-764.

[8] P. Bhavina and S. Vandana. , "To implement Cryptographic model for secure communication on FPGA using 32-bit ALU unit", IEEE International Conference on Signal and Image Processing, p.p. 440-443.

[9] L. Khoi-Nguyen, D. Anh-Vu, D. Quoc-Minh and B. Trong-Tu, "RTL Implementation for a Specific ALU of the 32-bit VLIW DSP Processor Core" ,International conference on advanced technologies for communications (ATC'14), 2014, p.p.387-392.

[10] Y. Shao-Ying, L. Yuan-Te , L. Wei-Chi , and H. Terng-Yin "Cost-Efficient Frequency-Domain MIMO–OFDM Modem With an SIMD ALU-Based Architecture", IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 23, NO. 12, DECEMBER 2015, p.p. 2791-280