Estudo de Implementação de uma Unidade Lógica Aritimética em VHDL

Gustavo G. Macedo[1], **Igor S. P. Paschoalino** [2]

[1]Instituto de Ciências Exatas e Informática – Pontifícia Universidade Católica de Minas Gerais (PUCMG)
Caixa Postal, 1.686. CEP 30535.610 - Belo Horizonte - Minas Gerais - Brasil

Abstract. *Arithmetic Logic Units (ALUs) are fundamental components in processors and digital systems, responsible for performing essential arithmetic and logical operations such as addition, subtraction, AND, OR, among others. This paper proposes the implementation of an ALU in VHDL for a RISC microprocessor focused on integer numbers, supporting operations like addition, subtraction, AND, OR, SET LESS THAN (SLT), and BEQ (Branch if Equal). The implementation was carried out in a modular manner, facilitating the understanding and maintenance of the design. After implementation, tests were conducted in the EDA Playground environment, where the ALU was subjected to various simulations. The results were satisfactory, showing 100% accuracy in all tested operations. It is concluded that the developed ALU meets the expected accuracy and functionality requirements, proving to be an effective solution for RISC microprocessors. The use of VHDL for modeling and simulation proved to be adequate, allowing detailed verification of the hardware behavior before its physical implementation.*

Resumo. *As Unidades Lógicas Aritméticas (ULAs) são componentes fundamentais em processadores e sistemas digitais, responsáveis por realizar operações aritméticas e lógicas essenciais, como adição, subtração, AND, OR, entre outras. Este artigo propõe a implementação de uma ULA em VHDL para um microprocessador RISC focado em números inteiros, suportando operações de soma, subtração, AND, OR, SET LESS THAN (SLT) e BEQ (Branch if Equal). A implementação foi realizada de forma modular, facilitando o entendimento e a manutenção do design. Após a implementação, foram realizados testes no ambiente EDA Playground, onde a ULA foi submetida a diversas simulações. Os resultados obtidos foram satisfatórios, apresentando 100% de precisão em todas as operações testadas. Conclui-se que a ULA desenvolvida atende aos requisitos de precisão e funcionalidade esperados, demonstrando ser uma solução eficaz para microprocessadores RISC. A utilização de VHDL para a modelagem e simulação mostrou-se adequada, permitindo a verificação detalhada do comportamento do hardware antes da sua implementação física.*

1. Introdução

A Unidade Lógica Aritmética (ULA) é um importante componente de processadores e sistemas digitais, que tem função de realizar operações aritméticas como soma e subtração, e operações lógicas como as operações computacionais AND e OR. Uma ULA é de extrema importância na área da arquitetura e está presente em diversos processadores e pode

ser representada com implementações diferentes entre si, dependendo da necessidade do projeto.

Para realizar um estudo a respeito de uma ULA, é necessário definir uma maneira de implementa-la. Para este trabalho, foi decidido o uso de uma linguagem de descrição de hardware (HDL), que são linguagens especializadas utilizadas para modelar e descrever o comportamento e a estrutura de circuitos e sistemas digitais. A utilização de tal tipo de linguagem permitirá a simulação do sistema de uma ULA ao nível de hardware.

A unidade lógica que proposta será capaz de realizar um conjunto de operações aritméticas e lógicas, sendo elas adição, subtração, AND, OR e SET LESS THAN (SLT), que são operações frequentemente requeridas em aplicações de processamento de dados. A implementação ainda contará com um decodificador para operações e também com detecção de "overflow". Os detalhes da ULA serão apresentados com o decorrer do artigo.

Desta forma, o objetivo desse artigo é estudar e demonstrar a implementação de uma ULA de 32 bits de uma forma mais próxima de uma implementação em hardware por meio da utilização de uma linguagem de descrição de hardware. Nessa implementação utilizaremos a HDL "Very high speed integrated circuit Hardware Description Language" (VHDL) para criação da ULA, para um entendimento melhor da HDL recomenda-se o Livro "VHDL: Programming by Example "[Perry 2002]. Essa implementação será realizada seguindo um formato modularizado, isto é, dividida em diversos módulos que serão integrados na implementação final, o que facilita o entendimento e auxilia na realização de um estudo a respeito do funcionamento da mesma.

O artigo está organizado da seguinte forma: Na Seção 2 vão ser apresentados trabalhos correlatos. Na Seção 3 será apresentada a proposta de implementação da ULA. Na Seção 4 serão apresentados os resultados do trabalho. E o trabalho será concluído na Seção 5.

2. Trabalhos Correlatos

A implementação de Unidades Lógicas e Aritméticas (ULA) em VHDL tem sido tema de diversos estudos, cada um explorando diferentes abordagens e técnicas para otimizar o desempenho e a eficiência dessas unidades. A seguir, são apresentados três artigos relevantes que abordam diferentes aspectos da implementação de ULAs.

Inicialmente, em [Reaz et al. 2002], é apresentado um design de ULA de ponto flutuante pipeline utilizando VHDL. A novidade deste trabalho reside na utilização do conceito de pipeline para aumentar o desempenho, onde ao ser utilizada, ela permite que múltiplas instruções sejam executadas de forma sobreposta. No design proposto, quatro módulos aritméticos (adição, subtração, multiplicação e divisão) subdivididos em módulos menores, juntamente com um método para seleção de dois bits para definir qual operação será feita são combinados para formar a Unidade Lógica Aritmética. Cada módulo é subdividido em módulos menores, sendo ela inteiramente feita em VHDL.

Em [Ritpurkar et al. 2015], também temos uma ULA em VHDL, sendo o tamanho da sua palavra 32 bits. O artigo discute a emulação de ASICs (Application Specific Integrated Circuits) usando VHDL, destacando a reprogramabilidade e rápida comercialização dos FPGAs (Field Programmable Gate Arrays). Apresenta o design de uma CPU RISC baseada em MIPS, incluindo conjunto de instruções, arquitetura e dia-

grama de temporização. Aborda a conversão de números de ponto flutuante para fixos, utilizando um módulo específico. O design, síntese e simulação foram realizados com o simulador Xilinx ISE 13.1i.

Já em [Verma 2020], é proposta uma ULA de baixo consumo de energia destinada a arquiteturas de processadores centrados em IoT. A ULA, sendo a principal unidade de computação em muitas arquiteturas de processadores e controladores utilizados em placas IoT, enfrenta alta comutação que leva à significativa dissipação de energia. Para mitigar isso, a arquitetura proposta utiliza técnicas de clock gating e one hot coding (CGOH), que reduzem a atividade de comutação e garantem a seleção eficiente de operações distintas. A ULA é codificada em VHDL e testada usando o Xpower Analyser no Xilinx ISE 14.1 para diversas arquiteturas de processadores centrados em IoT. Os testes, realizados em FPGA Virtex, mostraram melhorias substanciais no consumo de energia, especialmente em frequências mais altas.

3. Proposta de Arquitetura para ULA

A ULA de 32 bits foi implementada em VHDL de maneira modularizada, isso significa que a ULA foi dividida em diversas partes que se unem em um ”container”formando assim a ULA completa e funcional. A Figura 1 mostra o modelo da Unidade Lógica, que será apresentado com detalhes com o decorrer do artigo.

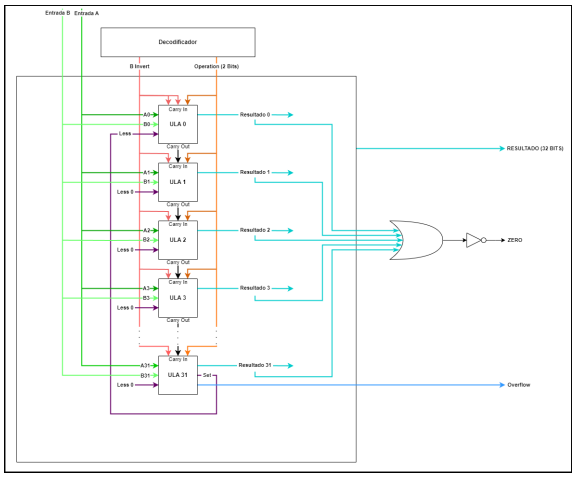

Figura 1. Modelo completo da Unidade Lógica

A arquitetura da ULA proposta contém três entradas padrão, duas de 32 bits, sendo elas os valores de entrada que serão computados, e uma de 3 bits enviada a um decodificador. A operação que será realizada é definida pelo decodificador com base nesta entrada de 3 bits. Para esta implementação, será utilizada a versão comportamental do decodificador, como mostrado na Figura 2.

```
entity decodificador is
    port(a, b, c : in std_logic;
         s1, s2, s3 : out std_logic);
end decodificador;

architecture arch_decodificador of decodificador is

begin

--Comportamental
    s1 <= '1' when (a = '0' and b = '1' and c = '1') else
          '1' when (a = '1' and b = '1' and c = '0') else
          '1' when (a = '1' and b = '1' and c = '1') else
          '0';

    s2 <= '1' when (a = '0' and b = '1' and c = '0') else
          '1' when (a = '0' and b = '1' and c = '1') else
          '1' when (a = '1' and b = '1' and c = '0') else
          '1' when (a = '1' and b = '1' and c = '1') else
          '0';

    s3 <= '1' when (a = '0' and b = '0' and c = '1') else
          '1' when (a = '0' and b = '1' and c = '1') else
          '1' when (a = '1' and b = '0' and c = '1') else
          '1' when (a = '1' and b = '1' and c = '1') else
          '0';


--Estrutural
--    s1 <= ((b and c) or a);
--    s2 <= b;
--    s3 <= c;


end arch_decodificador;
```

Figura 2. Arquitetura decodificador

Definida a operação, os valores de entrada vão para a ULA propriamente dita. A ULA de 32 bits é formada por uma sequência de Unidades Lógicas de 1 bit cada, que recebem o bit relativo da entrada e o valor da entrada de comando. As entradas passam pela sequência de 32 Unidades Lógicas de 1 Bit que realizam individualmente, bit a bit, a operação desejada. O projeto desta ULA implementa uma porta AND, OR e um somador completo para realizar suas operações. A saída de cada ULA do seguimento é 1 bit de resultado e o carry out, que será o valor de entrada do carry in da próxima unidade do seguimento. A Figura 3 apresenta o modelo de uma ULA de 1 bit padrão.

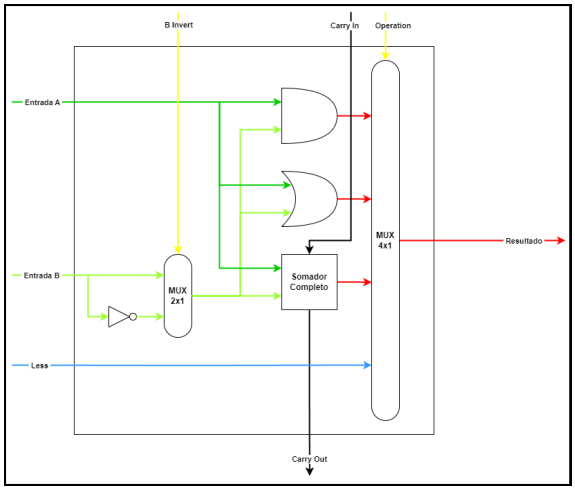

Figura 3. Modelo da ULA de 1 bit

Nessa arquitetura, apenas a última ULA se difere das demais, pois a mesma conta com um overflow detector, responsável pela saída de overflow da ULA, e também é a ULA com a saída ”Set”que serve como entrada para a entrada ”Less”da primeira ULA da sequência. A Figura 4 apresenta o modelo da ULA de 1 bit final com detector de Overflow.

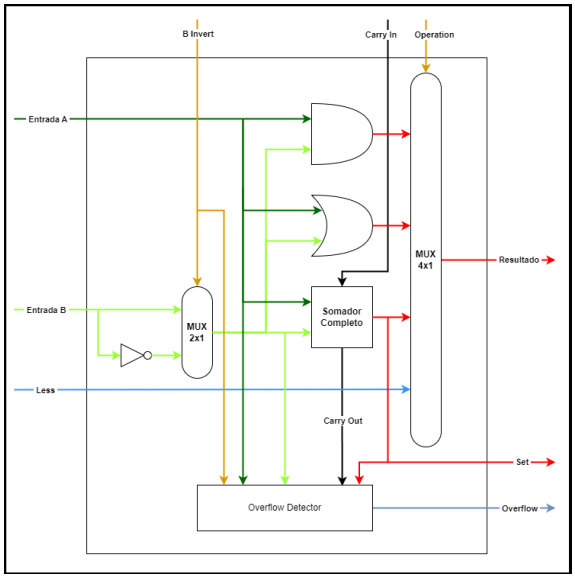

Figura 4. Modelo da última ULA de 1 bit

Nessa arquitetura, apenas a última ULA difere-se das demais, pois a mesma conta com um overflow detector, responsável pela saída de overflow da ULA, e também é a ULA com a saída ”Set”que serve como entrada para a entrada ”Less”da primeira ULA da sequência. A Figura 4 apresenta o modelo da ULA de 1 bit final com detector de Overflow.

```
entity ULAcontainer is
    port (
            in1 : in std_logic;
            in2 : in std_logic;
            in3 : in std_logic;
            a : in std_logic_vector (31 downto 0);
            b : in std_logic_vector (31 downto 0);

            outZero : out std_logic;
            overflow : out std_logic;
            finalResult : out std_logic_vector (31 downto 0)
    );
end entity;

architecture arch_ULAcontainer of ULAcontainer is

    component ULA32Bits is
    port (
            bInvert : in std_logic;
            Operation : in std_logic_vector(1 downto 0);
            a : in std_logic_vector (31 downto 0);
            b : in std_logic_vector (31 downto 0);

            result : out std_logic_vector (31 downto 0);
            overflowOut : out std_logic
    );
    end component;

    component decodificador is
        port(a, b, c : in std_logic;
            s1, s2, s3 : out std_logic);
    end component;

    signal s1 : std_logic;
    signal s2 : std_logic;
    signal s3 : std_logic;
    signal OperationDecode : std_logic_vector(1 downto 0);
    signal resultado : std_logic_vector(31 downto 0);
```

Figura 5. Entity da ULA

Para a realização dos testes, criaremos um ”testbench”em VHDL com diferentes operações que serão realizadas pela ULA. Para cada operação, criaremos 3 casos de teste, para desta forma confirmar o funcionamento da operação testada. Os casos de teste de cada operação lógica-aritmética estão na subseção a seguir.

4. Testes e Resultados

Nesta seção, apresentaremos os testes realizados e os resultados obtidos para todas as operações executadas pela ULA (Unidade Lógica e Aritmética). Essas operações incluem soma, subtração, AND, OR, SLT (Set on Less Than) e BEQ (Branch if Equal). Cada operação foi testada e analisada no ambiente de desenvolvimento EDA Playground, destacando o desempenho e a precisão do processador RISC projetado em VHDL. Para cada operação, foram colocados 3 testes com valores diferentes.

Os testes realizados para analizar o funcionamento da porta AND foram os apresentados na Figura 6. Para os testes mostrados na Figura 6, obtiveram-se os resultados apresentados na Figura 7 para a operação AND. Como é possível visualizar, apenas onde se encontra 1 AND 1 nas entradas foi possível encontrar o resultado 1, o que comprova o funcionamento da porta AND.

```
tb : process

begin

--and
in1 <= '0';
in2 <= '0';
in3 <= '0';
a <= "00000000000000000000000000001001";
b <= "00000000000000000000000000001100";

wait for 1 ns;

--and
in1 <= '0';
in2 <= '0';
in3 <= '0';
a <= "00000000000000000000000000000001";
b <= "11111111111111111111111111111111";

wait for 1 ns;

--and
in1 <= '0';
in2 <= '0';
in3 <= '0';
a <= "11111111111111111111111111111111";
b <= "01010101010101010101010101010101";

wait for 1 ns;

wait;

end process;
```

Figura 6. Testbench para AND

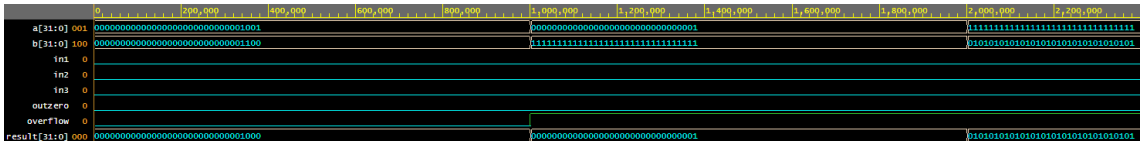

Figura 7. Resultados da AND

Os testes realizados para analisar o funcionamento da porta OR foram os apresentados na Figura 8. Para os testes mostrados na Figura 8, obtiveram-se os resultados apresentados na Figura 9 para a operação OR. Como é possível visualizar, o resultado é 1 na posição em que qualquer uma das entradas tenha o valor 1, o que comprova o funcionamento da porta OR.

```
tb : process

begin

--or
in1 <= '0';
in2 <= '0';
in3 <= '1';
a <= "00000000000000000000000000000000";
b <= "10101010101010101010101010101010";

wait for 1 ns;

--or
in1 <= '0';
in2 <= '0';
in3 <= '1';
a <= "00111000000111000000111000000111";
b <= "11000000111000000111000000111000";

wait for 1 ns;

--or
in1 <= '0';
in2 <= '0';
in3 <= '1';
a <= "00010010001101000101011001111000";
b <= "00101011111011010100010111101101";

wait for 1 ns;


wait;

end process;
```

Figura 8. Testbench para OR

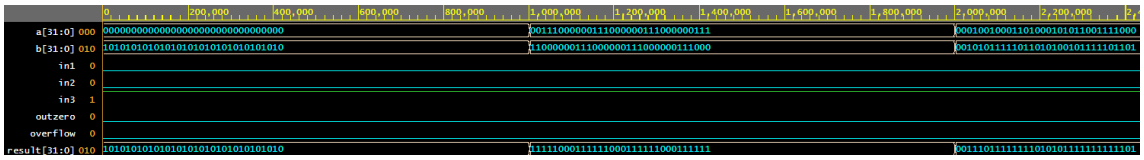

Figura 9. Resultados da OR

Os testes realizados para analisar o funcionamento da operação soma foram os apresentados na Figura 10. Para os testes mostrados na Figura 10, obtiveram-se os resultados apresentados na Figura 11 para a operação de soma. Como é possível visualizar, o resultado apresentado é a exata soma entre os valores de entrada, o que comprova o funcionamento da operação de soma.

```
tb : process

begin

--add
in1 <= '0';
in2 <= '1';
in3 <= '0';
a <= "00000000000000000000000000001010";
b <= "00000000000000000000000000000010";

wait for 1 ns;

--add
in1 <= '0';
in2 <= '1';
in3 <= '0';
a <= "00000000000000000000000010010111";
b <= "00000000000000000001110000000011";

wait for 1 ns;

--add
in1 <= '0';
in2 <= '1';
in3 <= '0';
a <= "10111010101110101011101010111010";
b <= "00101100101110010101100110111100";

wait for 1 ns;


wait;

end process;
```

Figura 10. Testbench para soma

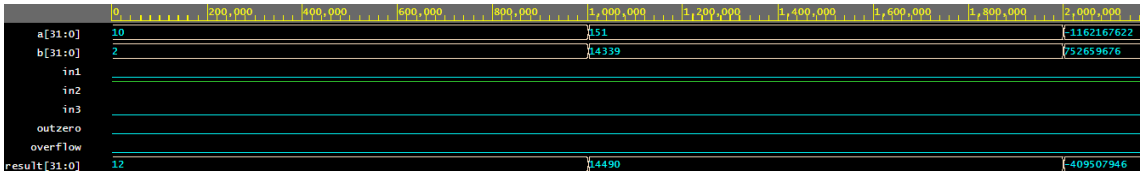

Figura 11. Resultados da soma

Os testes realizados para analisar o funcionamento da operação subtração foram os apresentados na Figura 12. Para os testes mostrados na Figura 12, obtiveram-se os resultados apresentados na Figura 13 para a operação de subtração. Como é possível visualizar, o resultado apresentado é a exata subtração entre os valores de entrada, o que comprova o funcionamento da operação de subtração.

```
begin

--sub
in1 <= '1';
in2 <= '1';
in3 <= '0';
a <= "00000000000000000000000000001010";
b <= "00000000000000000000000000000011";

wait for 1 ns;

--sub
in1 <= '1';
in2 <= '1';
in3 <= '0';
a <= "00000000000000011010000101010011";
b <= "00000000000000000000001111110011";

wait for 1 ns;

--sub
in1 <= '1';
in2 <= '1';
in3 <= '0';
a <= "10111111111110001100100001100010";
b <= "00001000100000111110000000000011";

wait for 1 ns;

wait;

end process;
```

Figura 12. Testbench para subtração

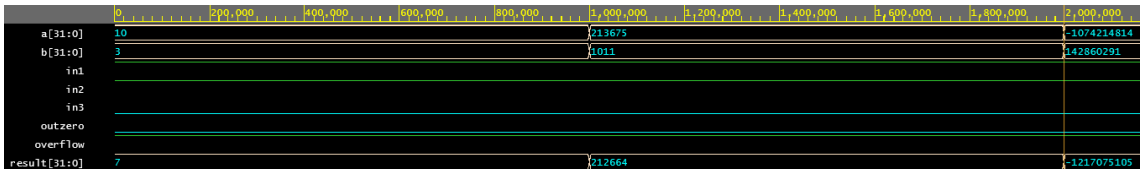

Figura 13. Resultados da subtração

Os testes realizados para analisar o funcionamento da operação SLT foram os apresentados na Figura 14. Para os testes mostrados na Figura 14, obtiveram-se os resultados apresentados na Figura 15 para a operação de set less than. Como é possível visualizar, o resultado apresentado é 1 quando a entrada a for menor que a entrada b e 0 quando a for maior que b, o que comprova o funcionamento da operação SLT.

```
begin

--slt
in1 <= '1';
in2 <= '1';
in3 <= '1';
a <= "10000000000000000000000000001010";
b <= "00000000000000000000000000000011";

wait for 1 ns;

--slt
in1 <= '1';
in2 <= '1';
in3 <= '1';
a <= "00000000000000000000000000000011";
b <= "00000000000000000000000000001010";

wait for 1 ns;

--slt
in1 <= '1';
in2 <= '1';
in3 <= '1';
a <= "00000000000000000000000000000011";
b <= "00000000000000000000000000000010";

wait for 1 ns;

wait;

end process;
```

Figura 14. Testbench para SLT

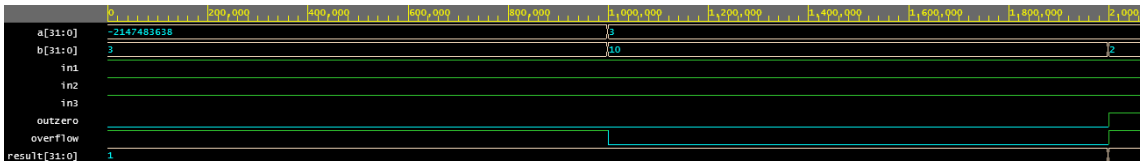

Figura 15. Resultados da SLT

Os testes realizados para analisar o funcionamento da operação BEQ foram os apresentados na Figura 16. Para os testes mostrados na Figura 16, obtiveram-se os resultados apresentados na Figura 17 para a operação de branch if equal. Como é possível vizualizar, o resultado apresentado é 1 na saída outZero quando as entradas forem iguais, e 0 na outZero quando as entradas são diferentes, o que comprova o funcionamento da operação BEQ.

```
begin

--beq
in1 <= '1';
in2 <= '1';
in3 <= '0';
a <= "00000000000000000000000000001111";
b <= "00000000000000000000000000001111";

wait for 1 ns;

--beq
in1 <= '1';
in2 <= '1';
in3 <= '0';
a <= "00000000000000000000000000011111";
b <= "00000000000000000000000000001111";

wait for 1 ns;

--beq
in1 <= '1';
in2 <= '1';
in3 <= '0';
a <= "00000000000000000000000000000111";
b <= "00000000000000000000000000000111";

wait for 1 ns;

wait;

end process;
```

Figura 16. Testbench para BEQ

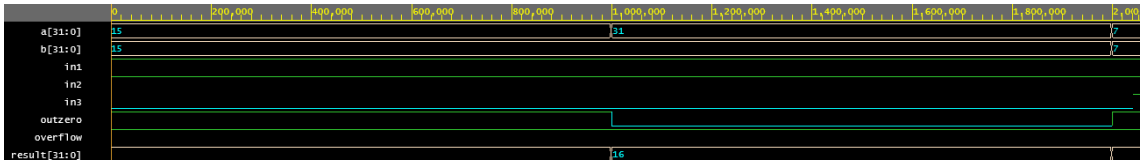

Figura 17. Resultados da BEQ

Após a avaliação dos resultados obtidos, é possível observar que a ULA proposta está funcionando corretamente na realização de cada uma das operações testadas.

5. Conclusões

Neste trabalho, foi desenvolvida uma Unidade Lógica Aritmética na linguagem de descrição de hardware VHDL. O projeto tem uma ULA que trabalha com entradas de até 32 bits e apenas números inteiros, fazendo uso de portas lógicas para apresentar seus resultados, sendo eles derivando-se de uma soma, subtração, AND, OR, Set Less Than e Branch On Equal, tendo para manusear essas operações 3 bits de controle. Sobre esta arquitetura proposta, ao ser implementada, ela nos mostrou resultados agradáveis ao ser submetida a testes, tendendo a 100% de eficácia. Para projetos futuros, pretende-se fazer com que a ULA consiga fazer mais operações como multiplicação e divisão, também adicionar um método para termos a opção de trabalhar com pontos flutuantes.

Referências

Inamdar, A., Ravi, J., Miller, S., Meher, S. S., Çelik, M. E., and Gupta, D. (2021). Design of 64-bit arithmetic logic unit using improved timing characterization methodology for rsfq cell library. *IEEE Transactions on Applied Superconductivity*, 31(5):1–7.

Perry, D. L. (2002). *VHDL: programming by example*, volume 4. McGraw-Hill New York.

Purnima, S., Mukesh, T., Jaikaran, S., and Sanjay, R. (2012). Vhdl environment for floating point arithmetic logic unit-alu design and simulation. *Research Journal of Engineering Sciences* _____ *ISSN*, 2278:9472.

Reaz, M., Islam, M., and Sulaiman, M. (2002). Pipeline floating point alu design using vhdl. In *ICONIP '02. Proceedings of the 9th International Conference on Neural Information Processing. Computational Intelligence for the E-Age (IEEE Cat. No.02EX575)*, pages 204–208.

Ritpurkar, S. P., Thakare, M. N., and Korde, G. D. (2015). Design and simulation of 32-bit risc architecture based on mips using vhdl. In *2015 International Conference on Advanced Computing and Communication Systems*, pages 1–6.

Verma, G. (2020). Design and analysis of alu for low power iot centric processor architectures. In *2020 Global Conference on Wireless and Optical Technologies (GCWOT)*, pages 1–5.