

Projeto 4 - Simulação de ULA em VHDL

Yan Tavares - 202014323

26 de novembro de 2023

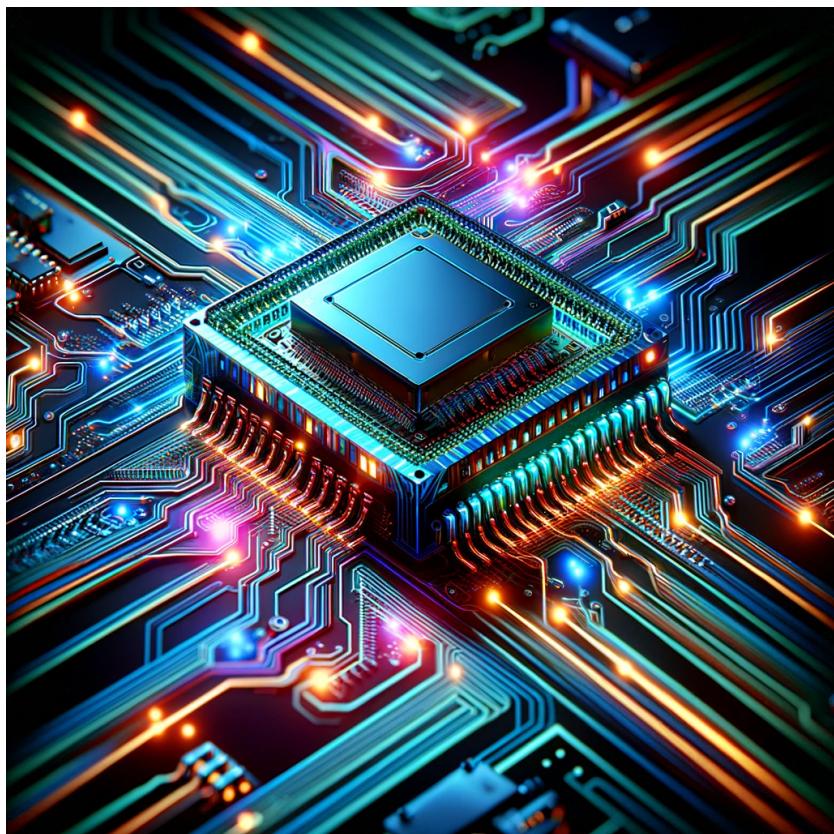


Figura 1: Representação digital de uma ULA (DALL-E 3)

Conteúdo

1	Introdução	3
2	Características da ULA	3
3	Interface e Arquitetura	3
4	Simulação e Verificação	3
5	Testes e Verificação	4
6	Descrição Detalhada	4
6.1	Comparação com e sem sinal	4
6.2	Capturas de Tela e Código	5
6.2.1	Ondas geradas (GTKWave)	6
6.2.2	Arquivo AluRV32.vhdl	6
6.2.3	Arquivo Testbench_AluRV32.vhdl	7
6.3	Overflow nas operações ADD e SUB	7
7	Conclusão	8
7.1	Desafios	8

1 Introdução

Este relatório apresenta o projeto e a simulação de uma Unidade Lógica Aritmética (ULA) de 32 bits para a arquitetura RISC-V. O objetivo é demonstrar a implementação e verificação das operações lógicas e aritméticas em VHDL.

2 Características da ULA

A ULA desenvolvida possui as seguintes características:

- Duas entradas de dados: A e B.
- Uma saída de dados: Z.
- Sinal Zero para detecção de valor zero na saída.
- Operações suportadas: ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU, SGE, SGEU, SEQ, SNE.

3 Interface e Arquitetura

A entidade ‘ulaRV’ foi definida com parâmetros genéricos, entradas e saídas. A arquitetura da ULA utiliza comandos ‘case-when’ para determinar a operação a ser realizada de acordo com o opcode recebido.

4 Simulação e Verificação

A simulação presente neste relatório foi realizada usando o software GHDL e GTKWave para a análise de ondas. Instruções de instalação de ambos os softwares podem ser encontradas no README do projeto. É importante ressaltar que o código também foi testado no EDA Playground e funciona como esperado. Cada operação da ULA foi testada individualmente para verificar a acurácia dos resultados e a geração do sinal zero.

5 Testes e Verificação

Foram realizados testes para cada operação da ULA, garantindo o correto funcionamento para entradas positivas, negativas e zero. Para isso, foram adicionados asserções no arquivo de testbench para garantir que o resultado é o esperado, além da análise das ondas.



```
● ● ●
1 BEGIN
2     -- ADD
3     opcode <= "0000"; -- ADD
4     A <= std_logic_vector(to_unsigned(5, 32));
5     B <= std_logic_vector(to_unsigned(3, 32));
6     WAIT FOR 10 ns;
7     ASSERT Z = std_logic_vector(to_unsigned(8, 32)) REPORT "ADD failed - Z" SEVERITY FAILURE;
8     ASSERT zero = '0' REPORT "ADD failed - zero" SEVERITY FAILURE;
9     REPORT "ADD OK";
10
11    -- ADD (zero)
12    opcode <= "0000"; -- ADD
13    A <= std_logic_vector(to_unsigned(0, 32));
14    B <= std_logic_vector(to_unsigned(0, 32));
15    WAIT FOR 10 ns;
16    ASSERT Z = std_logic_vector(to_unsigned(0, 32)) REPORT "ADD (Zero) failed - Z" SEVERITY FAILURE;
17    ASSERT zero = '1' REPORT "ADD (Zero) failed - zero" SEVERITY FAILURE;
18    REPORT "ADD (Zero) OK";
19
20    -- SUB
21    opcode <= "0001"; -- SUB
22    A <= std_logic_vector(to_unsigned(10, 32));
23    B <= std_logic_vector(to_unsigned(4, 32));
24    WAIT FOR 10 ns;
25    ASSERT Z = std_logic_vector(to_unsigned(6, 32)) REPORT "SUB failed - Z" SEVERITY FAILURE;
26    ASSERT zero = '0' REPORT "SUB failed - zero" SEVERITY FAILURE;
27    REPORT "SUB OK";
```

Figura 2: Exemplo de testes realizados para cada operação

Aqui está o report completo do arquivo testbench

6 Descrição Detalhada

6.1 Comparação com e sem sinal

A principal diferença entre comparações com e sem sinal em uma arquitetura como o RISC-V reside na forma como os valores são interpretados. Nas comparações com sinal, os números são tratados como valores inteiros com sinal, onde o bit mais significativo (MSB) é o bit de sinal. Assim, um número é

```
●▶ ghdl -r --std=08 Testbench_AluRV32
Testbench_AluRV32.vhdl:37:13:@10ns:(report note): ADD OK
Testbench_AluRV32.vhdl:46:13:@20ns:(report note): ADD (Zero) OK
Testbench_AluRV32.vhdl:55:13:@30ns:(report note): SUB OK
Testbench_AluRV32.vhdl:64:13:@40ns:(report note): AND OK
Testbench_AluRV32.vhdl:73:13:@50ns:(report note): OR OK
Testbench_AluRV32.vhdl:82:13:@60ns:(report note): XOR OK
Testbench_AluRV32.vhdl:91:13:@70ns:(report note): SLL OK
Testbench_AluRV32.vhdl:100:13:@80ns:(report note): SRL OK
Testbench_AluRV32.vhdl:109:13:@90ns:(report note): SRA OK
Testbench_AluRV32.vhdl:118:13:@100ns:(report note): SLT OK
Testbench_AluRV32.vhdl:127:13:@110ns:(report note): SLTU OK
Testbench_AluRV32.vhdl:136:13:@120ns:(report note): SGE OK
Testbench_AluRV32.vhdl:145:13:@130ns:(report note): SGEU OK
Testbench_AluRV32.vhdl:154:13:@140ns:(report note): SEQ OK
Testbench_AluRV32.vhdl:163:13:@150ns:(report note): SEQ (Zero) OK
Testbench_AluRV32.vhdl:172:13:@160ns:(report note): SNE OK
```

Figura 3: Resultado dos testes no console

interpretado como negativo se o MSB for 1. Já nas comparações sem sinal, todos os bits são considerados como parte do valor do número, tratando todos os números como positivos. Isso impacta diretamente a maneira como as operações de comparação são realizadas. Por exemplo, um número que é interpretado como grande positivo em uma comparação sem sinal pode ser considerado negativo em uma comparação com sinal. Por isso, é conveniente comparar números do mesmo tipo (com ou sem sinal) para evitar inconsistências.

6.2 Capturas de Tela e Código

É importante ressaltar que todos os arquivos estão presentes no arquivo .zip incluso no envio do projeto, as capturas de tela são apenas para referência.

6.2.1 Ondas geradas (GTKWave)

Podemos notar os resultados esperados ao visualizar as ondas. Foi utilizado um período de clock de 10 ns para a simulação

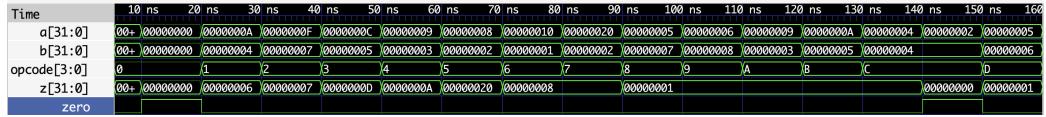


Figura 4: Ondas geradas pelo arquivo de testbench

6.2.2 Arquivo AluRV32.vhdl

Foram utilizadas extensões '.vhdl' ao invés de '.vhd' para diferenciar arquivos VHDL de Visual Hard Drive (VHD).

```

1 LIBRARY ieee;
2 USE ieee.std_logic.all;
3 USE ieee.numeric_std.all;
4
5 ENTITY AluRV32 IS
6 PORT (
7   opcode : IN std_logic_vector(3 DOWNTO 0);
8   A, B : IN std_logic_vector(31 DOWNTO 0);
9   Z : OUT std_logic_vector(31 DOWNTO 0);
10  zero : OUT std_logic
11 );
12 END AluRV32;
13
14 ARCHITECTURE behavior OF AluRV32 IS
15 BEGIN
16   PROCESS (opcode, A, B)
17   BEGIN
18     CASE opcode IS
19       WHEN "0000" => A0;
20         Z <- std_logic_vector(signed(A) + signed(B));
21       WHEN "0001" => A1;
22         Z <- std_logic_vector(signed(A) - signed(B));
23       WHEN "0010" => A2;
24         Z <- A AND B;
25       WHEN "0011" => A3;
26         Z <- A OR B;
27       WHEN "0100" => A4;
28         Z <- A XOR B;
29       WHEN "0101" => A5;
30         Z <- std_logic_vector(shift_left(unsigned(A), to_integer(unsigned(B)))); 
31       WHEN "0110" => A6;
32         Z <- std_logic_vector(shift_right(unsigned(A), to_integer(unsigned(B)))); 
33       WHEN "0111" => A7;
34         Z <- std_logic_vector(shift_right(signed(A), to_integer(unsigned(B)))); 
35       WHEN "1000" => A8;
36         Z <- (others => '0');
37       WHEN "1001" => A9;
38         Z <- (others => signed(A) < signed(B) ? '1' : '0');
39       WHEN "1010" => A10;
40         Z <- (others => '0');
41       WHEN "1011" => A11;
42         Z <- (others => '0');
43       WHEN "1100" => A12;
44         Z <- (others => '0');
45       WHEN "1101" => A13;
46         Z <- (others => '0');
47       WHEN "1110" => A14;
48         Z <- (others => '0');
49       WHEN "1111" => A15;
50         Z <- (others => '0');
51       WHEN OTHERS => NULL;
52     END CASE;
53   END PROCESS;
54
55   -- Necessary to make zero update in the same cycle as Z
56   PROCESS (Z)
57   BEGIN
58     zero <= '1' WHEN Z > std_logic_vector(to_unsigned(0, 32)) ELSE '0';
59   END PROCESS;
60
61 END behavior;

```

Figura 5: Captura de tela do código principal

6.2.3 Arquivo Testbench_AluRV32.vhdl

Figura 6: Captura de tela do código de testbench do projeto

6.3 Overflow nas operações ADD e SUB

O overflow ocorre quando o resultado de uma operação excede a capacidade do registrador de armazenamento. Na adição, há overflow se dois números

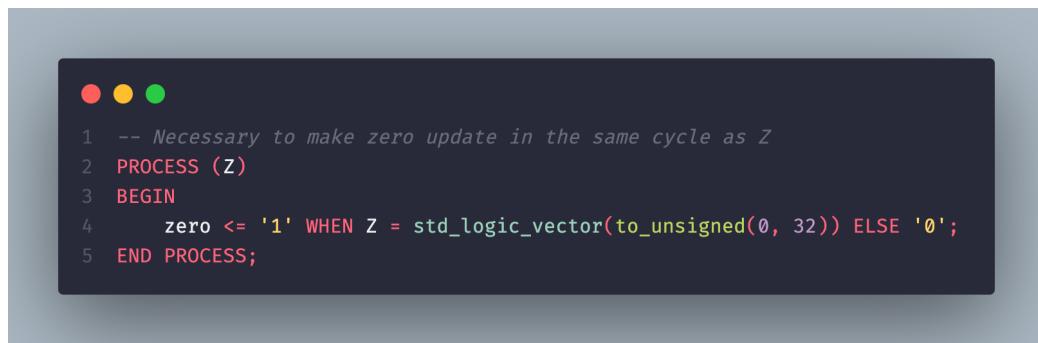
positivos geram um resultado negativo, ou se dois números negativos geram um resultado positivo. Na subtração, ocorre overflow se subtrairmos um número negativo de um número positivo e obtivermos um resultado negativo, ou se subtrairmos um número positivo de um número negativo e obtivermos um resultado positivo. A detecção de overflow pode ser realizada analisando os bits de sinal dos operandos e do resultado. Se os operandos têm o mesmo sinal mas o resultado tem um sinal diferente, então ocorreu overflow.

7 Conclusão

Em conclusão, todas as operações descritas na definição do trabalho foram corretamente implementadas em VHDL. Com a ULA e o Gerador de Immediatos prontos, estamos cada vez mais perto de simular o funcionamento de um processador que execute instruções RiscV.

7.1 Desafios

O maior desafio no projeto foi fazer com que o valor da variável *zero* atualizasse de forma síncrona com a saída *Z*. A solução foi criar um segundo process responsável apenas por atualizar o valor de *zero* de acordo com a saída *Z*.



```
● ● ●
1 -- Necessary to make zero update in the same cycle as Z
2 PROCESS (Z)
3 BEGIN
4     zero <= '1' WHEN Z = std_logic_vector(to_unsigned(0, 32)) ELSE '0';
5 END PROCESS;
```

Figura 7: Segundo process necessário