# Relatório 1 - Sistemas Digitais

Trabalho Prático 1 - Unidade Lógica e Aritmética (ULA) de 4 bits e 8 operações

Link dos arquivos de código: https://github.com/Ana-Schenkel/VHDL_ULA

## Grupo

Ana Schenkel Braga de Mendonça - DRE: 123701383

Maria Eduarda Araújo - DRE: 119054059

Mariana Barbosa Ramos - DRE: 121156142

Thiago Mendonça Carneiro Morgado - DRE: 121055299

## Sumário

# Introdução

O projeto tem como objetivo o desenvolvimento de uma Unidade de Lógica Aritmética (ULA) de 4 bits, com o uso do kit Xilinx Spartan 3, capaz de executar 8 operações: AND, OR, XOR, Módulo de Complemento de 2, Comparação em Complemento de 2, Soma em Complemento de 2, Subtração em Complemento de 2 e Shift (tanto para direita, quanto para esquerda). Além disso, o projeto trata de identificar 4 flags: Zero, Negativo, Carry Out, Overflow. Para o funcionamento do projeto, foi necessária a implementação de uma lógica em VHDL que respeitasse a execução das operações da ULA, o roteiro de uso da placa e que também solucionasse o problema de *bouncing* para a recepção correta dos comandos.

Como foi designado, o projeto recebe entradas de até 4 bits que serão atribuídos às chaves (switches) da placa, onde a primeira entrada é referente a operação da ULA, a segunda é o vetor referente ao primeiro operando (A) e a terceira é o vetor do segundo operando (B). Cada uma dessas entradas é registrada separadamente, com a confirmação feita pelo botão do potenciômetro da placa, no qual o Debounce foi necessário. As saídas foram atribuídas a cada LED, formando um vetor de 8 bits como resultado final: os 4 LEDs mais à direita (os bits menos significativos) são os resultados das operações, já os 4 LEDs mais à esquerda (os bits mais significativos) representam as flags, onde cada um dos bits é um indicador de uma das 4 flags pré-determinadas (one-hot).

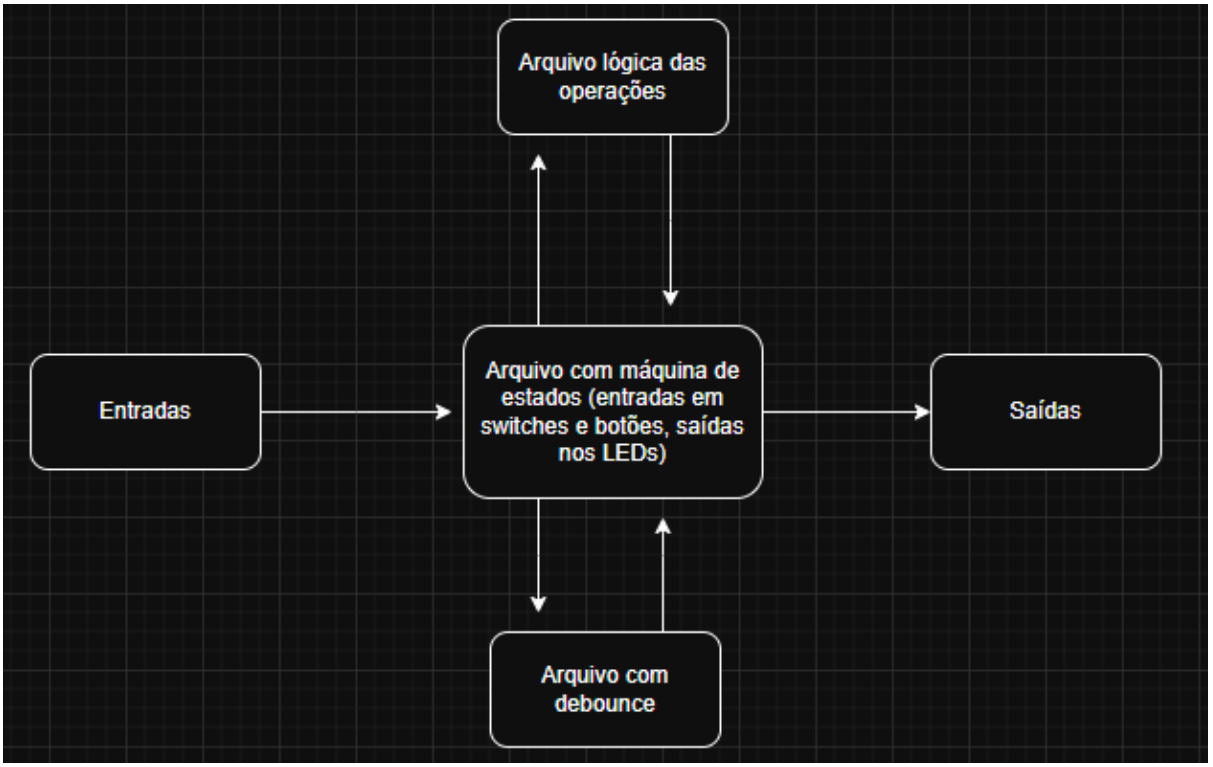
# Metodologia

Para a realização do projeto, dividimos a ULA em três módulos (arquivos em VHDL), como mostra o diagrama de blocos. O arquivo principal estabelece qual variável (no código, **signal**) deve ser registrada, para isso seria necessário que esse arquivo aplicasse a lógica de uma máquina de estados, em que:

- ➔ Estado 0: Registra o código da operação, configurado nas 4 chaves;
- ➔ Estado 1: Registra o primeiro operando, configurado nas 4 chaves;
- ➔ Estado 2: Registra o segundo operando, configurado nas 4 chaves;
- ➔ Estado 3: Exibe resultado e flags nos leds do kit.

A mudança desses estados seria vinculada ao botão, para isso, o arquivo de debounce deve tratar essa entrada, funcionando como um componente do arquivo principal, que retorna um pulso tratado do botão para a máquina de estados. Já o terceiro módulo está vinculado ao 4° estado, em que o arquivo retorna o resultado da operação conforme as entradas coletadas nos estados anteriores.

**Diagrama 1**: Diagrama de blocos da ULA.

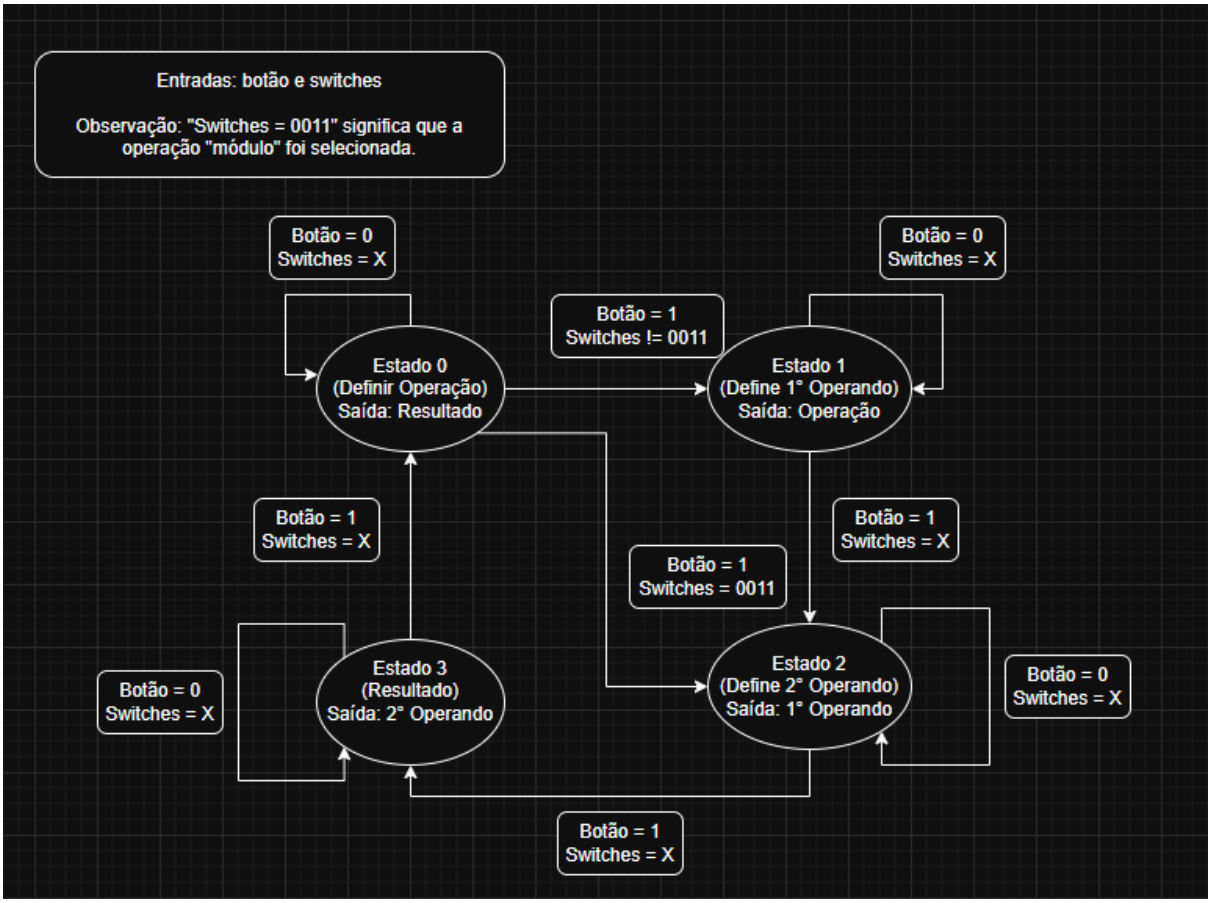



No código em VHDL do arquivo principal, os **components** são executados em paralelo (não sequencialmente) com o **process** da máquina de estados, para evitar erros, todos os arquivos foram sincronizados com um clock da placa Spartan 3.

# Máquina de estados

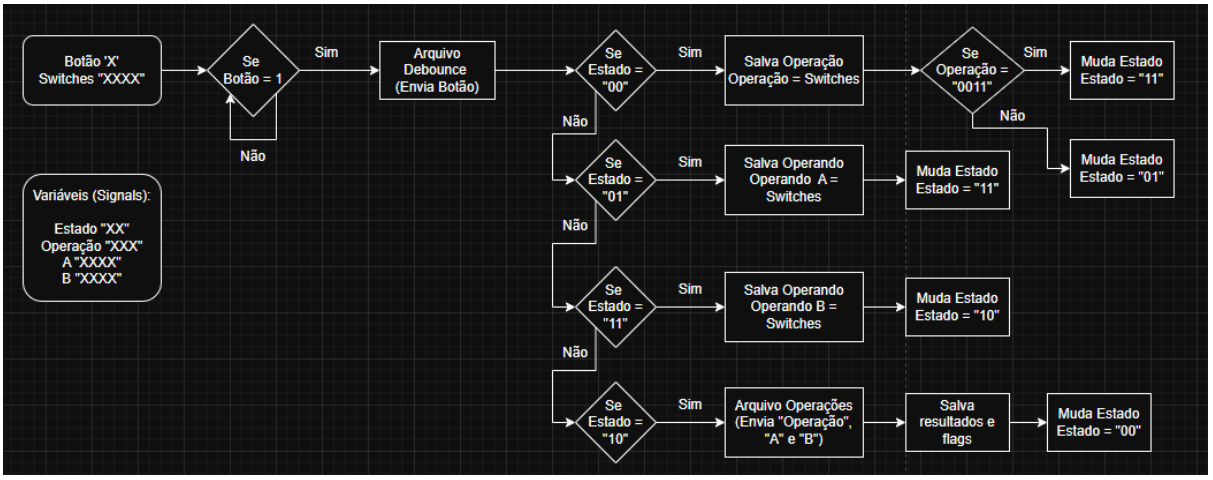
A máquina de estados idealizada foi uma máquina de Moore, em que, considerando a saída como o **vector_std_logic** vinculado aos LEDs, a saída só dependerá do estado atual e não da entrada. Além disso, a lógica da máquina de Moore precisou receber mais uma entrada além do botão (que rege a mudança de estado) devido à operação de módulo, que utiliza apenas um operando, ou seja, pula um estado. Dessa forma, o seguinte diagrama foi desenvolvido:

**Diagrama 2:** Diagrama de Estados do arquivo principal.




Com essa lógica da máquina de Moore, o usuário recebe nos LEDs o que ele selecionou no estado anterior, garantindo uma confirmação da operação que está sendo realizada. A partir do diagrama de estados e seguindo o código de Gray, foi feita a seguinte codificação: Estado 0 = “00”, Estado 1 = “01”, Estado 2 = “11”, Estado 3 = “10”. O que gerou o seguinte fluxograma para desenvolvimento do código em VHDL:

**Diagrama 3:** Fluxograma do arquivo principal.




O fluxograma foi aplicado dentro de um **process** (sequencial) no código, utilizando a função **case-when** para os estados e o **for in loop** para o registro das entradas das chaves e para mostrar as saídas.

**Figuras 1 e 2:** Lógica do **process** do arquivo principal.

```
begin
   if (clk'event and clk = '1') then
      if (debounce = '1') then

         case state is

            when "00" =>

               for i in 0 to 2 loop
               for i in 0 to 4 loop
               if (operation_var = "011") then
                  state <= "11";
               else
                  state <= "01";
               end if;

               opcode <= operation_var;
```

```
   when "01" =>

      a <= "0000";

      for i in 0 to 3 loop
      for i in 0 to 3 loop
      state <= "11";


   when "11" =>

      b <= "0000";

      for i in 0 to 3 loop
      for i in 0 to 3 loop
      state <= "10";

   when "10" =>
      leds <= result;
      state <= "00";
```

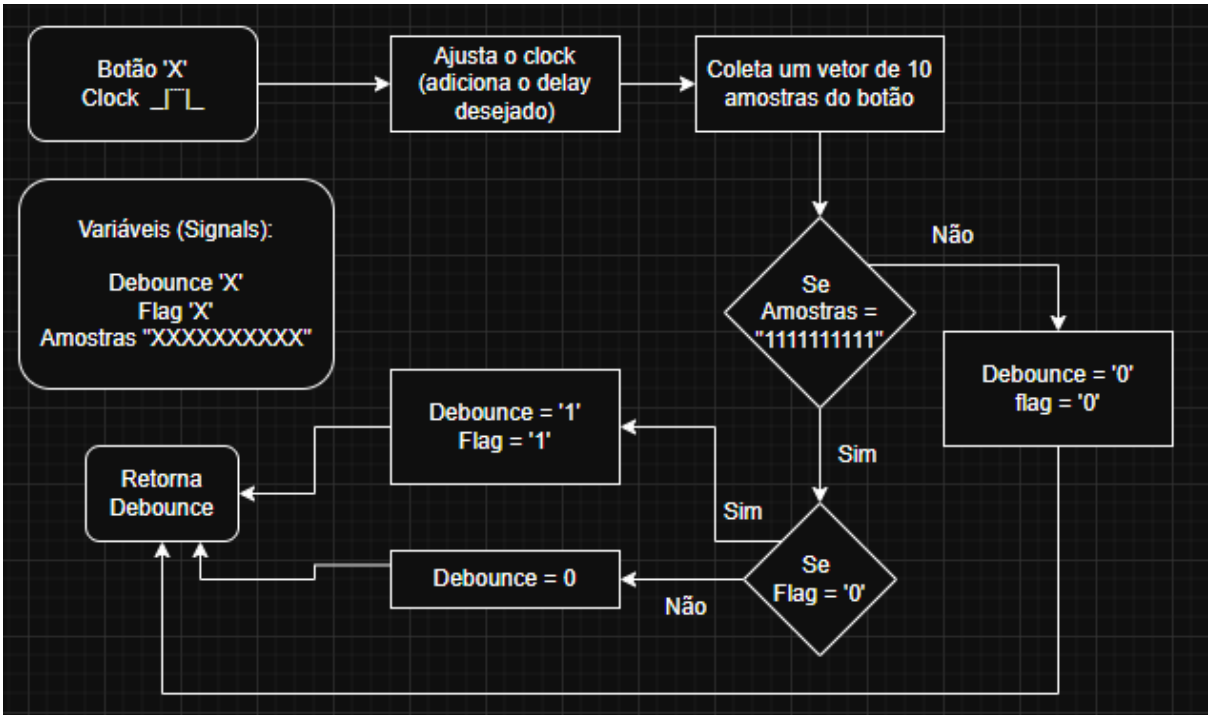
# Debounce

O Debounce é uma técnica necessária para captação correta das entradas, porque quando o botão é pressionado ou solto, é gerado um ruído devido ao movimento mecânico chamado de **bouncing**, ou seja, o sinal oscila rapidamente entre os estados de ligado e desligado, gerando múltiplos pulsos elétricos ao invés de um único pulso, como se o botão fosse pressionado repetidas vezes. Com o Debounce isso é tratado, gerando apenas um pulso cada vez que o botão é pressionado manualmente.

A lógica utilizada para desenvolver o debounce pode ser dividida em três módulos (no código, três **process** no arquivo de debounce): um divisor de clock, um registrador de amostras e um de comparação de amostras. No geral, é preciso considerar um pulso do botão como algo lento e estável, já que tentamos eliminar um ruído que ocorre em um espaço de tempo curto, para isso servem os primeiros dois módulos, um para retardar o clock da placa (que costuma ter frequências altas) e outro para coletar um vetor de amostras que definem a estabilidade da entrada. O terceiro módulo serve para comparar se o vetor de amostras está estabilizado em 1 (botão pressionado), em 0 (botão solto) ou oscilando (botão sendo pressionado ou sendo solto), além de gerar o pulso tratado, utilizando uma variável que indica se esse pulso já foi gerado ou não no ciclo de clock anterior.

Com essa lógica, considerando um vetor de 10 bits para o registro de amostras, chamando a variável de controle de “flag” e o pulso tratado de “Debounce”, foi desenvolvido o seguinte fluxograma:

**Diagrama 4:** Fluxograma do arquivo de Debounce




No código em VHDL o divisor de clock, que gera pulsos periódicos mais lentos para amostrar o estado do botão em intervalos controlados, é feito gerando pulsos em um **signal** (***sample_pulse***) a cada vez que o clock se repete n vezes. O número de repetição do clock ideal foi testado em laboratório com o clock de pino E12, e definido no **generic** "***delay***" como 50000, ou seja, a cada 50000 bordas de subida do clock da placa é gerado um pulso no ***sample_pulse***, diminuindo a frequência do clock. A imagem a seguir mostra a lógica do primeiro módulo:

**Figura 3:** Código do divisor de clock.

```
process(clk) --Clock Divider

variable count: integer := 0;

begin
   if (clk'event and clk = '1') then

      if (count < delay) then
         count := count + 1;
         sample_pulse <= '0';
      else
         count := 0;
         sample_pulse <= '1';
      end if;

   end if;
end process;
```

Para a amostragem, a cada pulso gerado (***sample_pulse***), o sistema desloca o valor atual do botão para um registrador de 10 bits (***sample***), que armazena o histórico das últimas

10 leituras do botão, que no teste em laboratório mostrou-se funcional. O registrador é inicialmente configurado com o valor “0001111000", apenas como valor de partida em que o pulso tratado com certeza não seria gerado. Dessa forma, o arquivo do segundo módulo é apenas uma lógica de shift com a entrada do botão, como mostra a imagem abaixo:

**Figura 4:** Código do registrador de amostras.

```
process(clk) --Sampling Process
begin
   if (clk'event and clk = '1') then

      if (sample_pulse = '1') then
         sample(9 downto 1) <= sample(8 downto 0); -- Left Shift
         sample(0) <= input;
      end if;

   end if;
end process;
```

Na comparação, quando todas as 10 amostras armazenadas forem iguais a 1, ou seja, ***sample*** = “1111111111”, indica que o botão foi pressionado por tempo o suficiente para ser considerado sem ruídos, gerando um pulso único de saída (***debounce_s***). Além disso, a ***flag*** impede que múltiplos pulsos sejam emitidos enquanto o botão ainda estiver pressionado, liberando uma nova leitura apenas após o botão ser solto. Dessa forma, a técnica garante que apenas um comando será enviado ao sistema, mesmo que o botão fisicamente gere múltiplos pulsos devido ao bouncing.

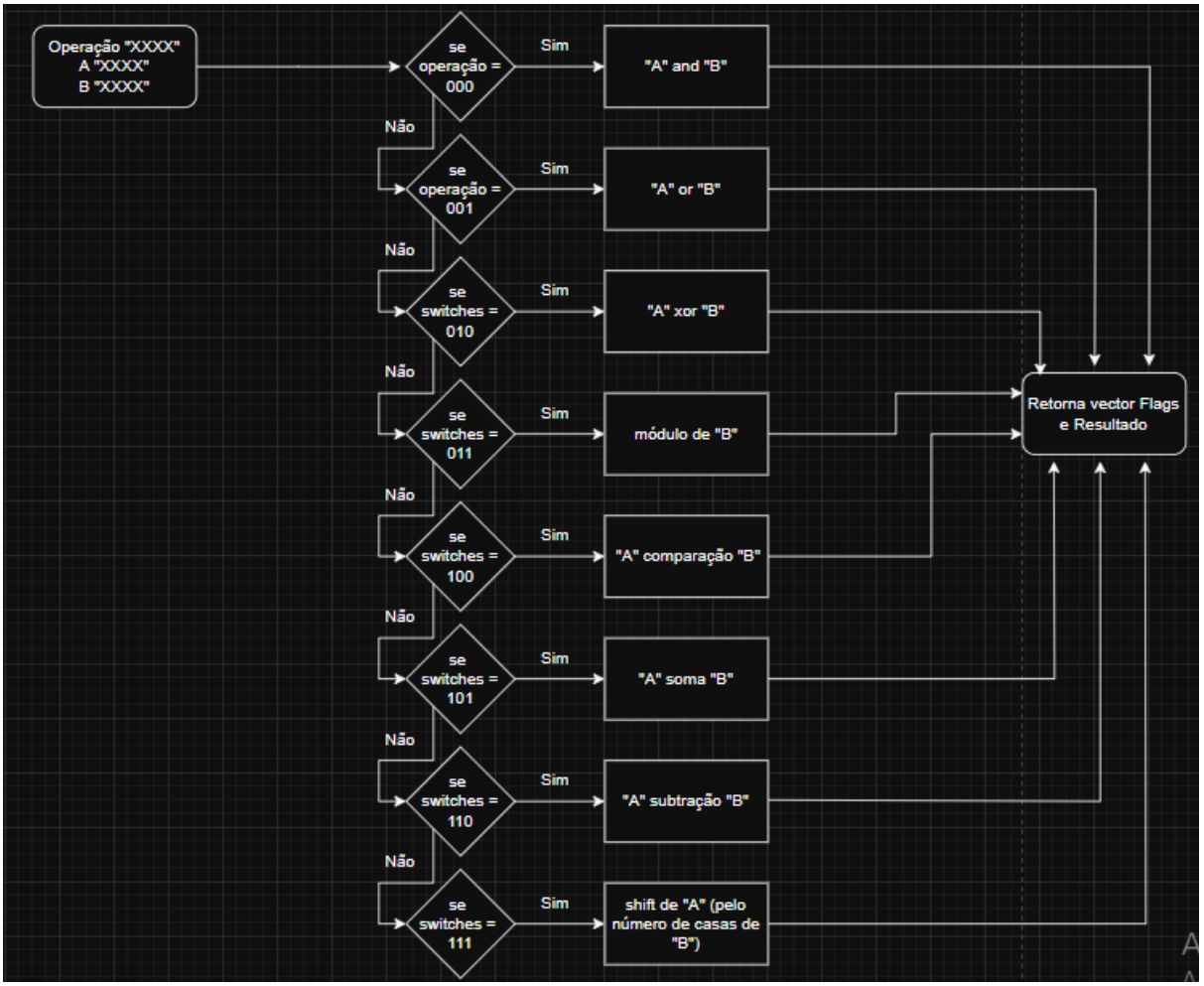
**Figura 5:** Código  do comparador de amostras e gerador de pulso tratado

```
process(clk) --Button Debouncing

variable flag: std_logic := '0';

begin
   if (clk'event and clk = '1') then

      if (sample = "1111111111") then --Active High Pulse Out

         if (flag = '0') then
            debounce_s <= '1';
            flag := '1';
         else
            debounce_s <= '0';
         end if;

      else
         debounce_s <= '0';
         flag := '0';
      end if;

   end if;
end process;
```

# Operações

O arquivo de operações foi estruturado com o objetivo de organizar de forma clara os processos a serem realizados pela ULA. Para isso, foram designados códigos de 3 bits, inseridos por meio das 4 chaves (switches) da placa, seguindo a lógica definida pela máquina de estados deste projeto. As operações foram designadas como AND = “X000”, OR = “X001”, XOR = “X010”, Módulo = “X011”, Comparação = “X100”, Soma = “X101”, Subtração = “X110” e Shift = “X111”. A partir desse arquivo, são produzidas uma saída correspondente ao resultado da operação selecionada e também flags específicas, que posteriormente são concatenadas para compor o resultado final, um vetor de 8 bits. A lógica estabelecida pode ser visualizada no fluxograma a seguir:

**Diagrama 5:** Fluxograma do arquivo de operações.



## Estrutura Lógica do Arquivo VHDL

Para detalhar as operações disponíveis, é importante compreender a estrutura interna do arquivo VHDL responsável por controlá-las. O código foi desenvolvido a partir de um bloco **process**, sensível à borda de subida do clock, onde ocorre a execução sequencial da lógica. Dentro desse processo, utilizam-se **signals** para a comunicação entre diferentes partes da arquitetura e para refletir alterações visíveis externamente, enquanto **variables** são empregadas para armazenar valores intermediários durante o ciclo de execução. Essa separação corrobora para a clareza no entendimento do código, como também em sua manutenção e identificação de erros.

A escolha entre **signals** e **variables** no desenvolvimento do arquivo foi feita com base no comportamento desejado dentro do processo. Como o bloco **process** é sensível às mudanças nos sinais de entrada (**A, B e opcode**), as variáveis foram utilizadas para armazenar dados intermediários e realizar operações que não precisam ser visíveis fora do processo. Por exemplo, as variáveis **A_signed** e **B_signed** armazenam os operandos em formato adequado para operações aritméticas, enquanto **sum** e **sub** são usadas com o tipo signed para possibilitar cálculos com sinal. A variável **shift_amount** define o número de posições a deslocar na operação de shift, e **op_result** guarda o resultado parcial da operação lógica ou aritmética executada. Já a variável **flags** armazena os indicadores como carry, overflow e negativo, que são posteriormente concatenados ao resultado final.

**Figura 6:** Parte do código de operações onde os signals são declarados

```
entity operations is
    Port(
            opcode : in std_logic_vector (2 downto 0); --the operation's code is inserted
            A : in std_logic_vector (3 downto 0); -- first number inserted
            B : in std_logic_vector (3 downto 0); -- second number inserted
            clk : in std_logic; -- process triggered on rising edge of clock
            result : out std_logic_vector (7 downto 0) -- output
);
end operations;
```

**Figura 7:** Parte do código de operações onde as variables são declaradas

```
process(clk)
    variable A_signed, B_signed : std_logic_vector (3 downto 0);
    variable shift_amount : integer;
    variable op_result : std_logic_vector (3 downto 0);
    variable sum, sub : signed(3 downto 0);
    variable flags : std_logic_vector (3 downto 0);
    variable c_in, c_out : std_logic;
```

AND

Essa operação realiza um AND bit a bit dos operandos, onde cada bit do resultado será igual a 1 somente se os bits correspondentes das entradas forem iguais a 1, caso contrário o bit do resultado será 0. Como descrito na tabela:

**Tabela 1:** Lógica do AND:

| A | B | S |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Código VHDL:

A lógica da operação AND bit a bit em VHDL foi implementada por meio de uma estrutura condicional que depende do valor do sinal opcode. Quando "***case opcode is when "000" =>"***, é executada a operação lógica AND entre os sinais de entrada A e B, ambos vetores de 4 bits. No código, essa operação é representada pela variável "***op_result := A and B;"***, utilizando o operador **and** para aplicar a operação bit a bit entre os operandos. Ao final do processo, que concentra as operações da ULA, o resultado dessa operação é combinado com a variável de controle ***flags***, formando o vetor ***result***, que possui 8 bits. Dessa forma, os 4 bits menos significativos correspondem ao resultado lógico, enquanto os bits mais significativos armazenam as informações relacionadas às flags da operação, que no caso desta operação será sempre preenchido por zeros (***flags := "0000")***, por não possuir flags.

**Figura 8:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

```
begin
    if clk'event and clk = '1' then
        case opcode is
```

**Figura 9**: Implementação da operação AND e definição das flags

```
when "000" => --AND
op_result := A and B;
flags := "0000";
```

OR

É feito nessa operação um OR bit a bit dos operandos, onde cada bit do resultado será igual a 1, se um dos bits correspondentes das entradas forem iguais a 1 e igual a 0, se os dois bits de entrada forem iguais a 0. A seguir é possível visualizar a sua lógica de saídas:

**Tabela 2**: Lógica do OR

| A | B | S |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Código VHDL:

A lógica da operação OR bit a bit em VHDL foi implementada por meio de uma estrutura condicional que depende do valor do sinal opcode. Quando "***case opcode is when "001" =>"***, é executada a operação lógica OR entre os sinais de entrada A e B, ambos vetores de 4 bits. No código, essa operação é representada pela variável "***op_result := A or B;"***, utilizando o operador **or** para aplicar a operação bit a bit entre os operandos. Ao final do processo, que concentra as operações da ULA, o resultado dessa operação é combinado com a variável de controle de flags, formando o vetor ***result***, que possui 8 bits. Dessa forma, os 4 bits menos significativos correspondem ao resultado lógico, enquanto os bits mais significativos armazenam as informações relacionadas às flags da operação, que neste caso também são preenchidas com zeros (***flags := "0000"***), uma vez que essa operação não gera as flags específicas.

**Figura 10:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

```
begin
    if clk'event and clk = '1' then
        case opcode is
```

**Figura 11:** Implementação da operação OR e definição das flags

```
when "001" => --OR
    op_result := A or B;
    flags := "0000";
```

## XOR

A operação XOR bit a bit é realizada aplicando o operador lógico exclusivo (XOR) entre os bits correspondentes dos operandos de entrada A e B. Ou seja, cada bit de A é comparado com o bit correspondente de B, onde para bits diferentes, a saída é 1 e para A=B, a saída é 0. Esse processo se repete para todos os quatro bits, formando o vetor de saída S com os resultados individuais de cada XOR.

**Tabela 3:** Lógica do XOR

| A | B | S |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Código VHDL:

No código VHDL, a operação XOR bit a bit é implementada por meio da instrução ***"op_result := A xor B;"***. Quando ***"case opcode is when "010" =>"***, a operação lógica exclusiva é executada entre os vetores de 4 bits, A e B, resultando em um novo vetor de 4 bits (**op_result**). Em seguida, esse valor é concatenado com o valor da variável flags, formando o vetor final **result** de 8 bits. Como a operação XOR não gera carry, overflow e negativo, a variável flags é definida como "0000" (***flags := "0000";)***.

**Figura 12:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

```
begin
    if clk'event and clk = '1' then
        case opcode is
```

**Figura 13:** Implementação da operação XOR e definição das flags

```
when "010" => -- XOR
    op_result := A xor B;
    flags := "0000";
```

## Módulo de Complemento de 2

Quando essa operação é selecionada, há apenas uma entrada de operando, que é convertido para seu módulo, invertendo todos os seus bits e somando 1 a este número.

Código VHDL:

Quando **“case opcode is when "011" =>”**, a operação de obtenção do módulo (valor absoluto) do vetor de 4 bits B é executada, considerando sua representação em complemento de dois.

Para isso, é verificado o bit mais significativo de B (**B(3)**), que indica o sinal do número (0 para positivo, 1 para negativo). Se B(3) = '1', ou seja, se B for um número negativo, é calculado seu valor absoluto por meio da instrução ***“op_result := std_logic_vector(unsigned(not B) + 1);”***, convertendo B para seu complemento de dois. Caso contrário, se B já for positivo, o vetor B é atribuído diretamente a **op_result**.

Como essa operação não envolve carry, overflow ou negativo, a variável **flags** é definida como "0000".

**Figura 14:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

```
begin
    if clk'event and clk = '1' then
        case opcode is
```

**Figura 15:** Implementação da operação de Módulo de Complemento de 2 e definição dos flags

```
when "011" => --2's Complement Module
    if B(3) = '1' then
        op_result := std_logic_vector(unsigned (not B)+1);
    else
        op_result := B;
    end if;

    flags := "0000";
```

## Comparação em Complemento de 2

Nesta operação é feita a verificação entre os valores dos operandos A e B, se A é maior que B, A igual a B ou A menor que B, considerando que estão sendo escritos em Complemento de 2. O resultado desta operação está em one-hot de um número de 4 bits, a relação entre resultado e seu número em one-hot está descrita na tabela abaixo:

**Tabela 4:** Relação entre código e resultado.

| Resultado | Código |
|---|---|
| A>B | 0001 |
| A<B | 0010 |
| A=B | 0100 |

Código VHDL:

Quando **“case opcode is when "100" =>”**, será executada a operação de comparação entre os vetores A e B, considerando suas representações em complemento de dois.

Inicialmente, é verificado se o bit mais significativo (bit 3) de A ou B é igual a 1, o que indicaria um número negativo na representação com sinal. Caso isso seja verdade, aplica-se a operação de complemento de dois (inversão com not e adição de 1), convertendo o valor para sua forma positiva com sinal. Em seguida, é realizada a comparação entre os operandos. Se ambos possuem o mesmo sinal, a comparação é feita diretamente entre os valores convertidos. Se **A_signed** for maior que **B_signed**, **op_result** recebe "0001"; se menor, "0010"; e em caso de igualdade, "0100". Além disso, se o número maior for negativo, o código inverte o resultado, o que garante que a saída reflita corretamente qual número é matematicamente maior. Se os sinais forem diferentes, basta comparar os bits mais significativos diretamente: o operando com bit 3 igual a 0 (positivo) será considerado maior. Para esta operação as flags carry out, overflow e negativo não fazem sentido para a lógica estabelecida.

**Figura 16:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

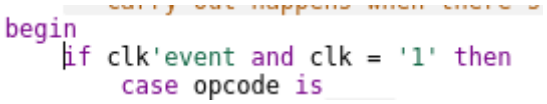


**Figura 17:** Conversão dos operandos para complemento de dois na operação de comparação

```
when "100" => --2's Complement Comparison
    if A(3) = '1' then
        A_signed := std_logic_vector(unsigned (not A)+1);
    else
        A_signed := A;
    end if;

    if B(3) = '1' then
        B_signed := std_logic_vector(unsigned (not B)+1);
    else
        B_signed := B;
    end if;
```

**Figura 18:** Lógica de comparação entre operandos com sinal

```
-- Comparison begins here

if A(3) = B(3) then
  if A_signed > B_signed then
      op_result := "0001";
      if A(3) = '1' then
      op_result := "0010";
      end if;

  elsif A_signed < B_signed then
      op_result := "0010";
      if A(3) = '1' then
      op_result := "0001";
      end if;

  else
      op_result := "0100";
  end if;

else
    if A(3) > B(3) then
    op_result := "0010";
    else
    op_result := "0001";
    end if;
end if;

flags := "0000"; -- No flags
```

## Soma em Complemento de 2

É feito nessa operação uma soma entre os operandos, lembrando de considerar o Complemento de 2 e as flags que serão explicadas.

Código VHDL:

Quando ***"case opcode is when "101" =>"***, a soma entre os vetores de 4 bits A e B é executada, considerando o transporte entre os bits para simular o funcionamento de um somador completo.

Inicialmente, a variável **c_out** é inicializada com '0'. Em seguida, um laço **for** é responsável por percorrer cada posição dos vetores de entrada, realizando a soma bit a bit. A cada iteração, o bit de transporte de entrada **c_in** recebe o valor do transporte anterior **c_out**. O bit de soma sum(i) é então calculado pela expressão (A(i) xor B(i)) xor **c_in**, enquanto o novo transporte de saída **c_out** é determinado com base na lógica de somador completo, pela expressão ((A(i) xor B(i)) and **c_in**) or (A(i) and B(i)).

Após o término do laço, o vetor **sum** é convertido para **std_logic_vector** e atribuído à variável **op_result**, armazenando os 4 bits resultantes da operação de soma.

Logo após, são realizadas verificações para definir o valor das flags que indicam condições especiais da operação. Se os bits mais significativos de A e B forem iguais, mas o bit mais significativo de **op_result** for diferente, então houve overflow. Nesse caso, se o resultado for negativo (***op_result(3) = '1'***), as flags são configuradas como "1100" (overflow + negativo); caso contrário, "0100" (apenas overflow). Se não houver overflow, mas o resultado for negativo, a flag "1000" é atribuída. Para os demais casos, a variável **flags** é configurada como "0000".

**Figura 19:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

```
begin
    if clk'event and clk = '1' then
        case opcode is
```

**Figura 20:** Lógica do somador  entre operandos com sinal e flags

```vhdl
when "101" => -- 2's Complement Adder

    c_out := '0';

    for i in 0 to 3 loop
        c_in := c_out;
        sum(i) := (A(i) xor B(i)) xor c_in;
        c_out := ((A(i) xor B(i)) and c_in) or (A(i) and B(i));
    end loop;

    op_result := std_logic_vector(sum); -- sum result in 4 bits

    if (A(3) = B(3)) and (op_result(3) /= A(3)) then -- Overflow
        if op_result(3) = '1' then
            flags := "1100"; -- Overflow + Negative
        else
            flags := "0100"; -- Overflow
        end if;

    elsif op_result(3) = '1' then
        flags := "1000"; -- Negative (sem overflow)
    else
        flags := "0000"; -- No flags
    end if;

    if c_out = '1' then
        flags := "0010" or flags; -- Carry Out flag
    end if;
```

## Subtração em Complemento de 2

Nesta operação, é realizada uma subtração entre os operandos, utilizando a representação em complemento de dois. Além disso, são consideradas as flags associadas à operação, que indicam condições específicas como overflow, zero, sinal ou carry, e que serão detalhadas posteriormente.

Código VHDL:

Quando ***"case opcode is when "110" =>"***, a subtração entre os vetores de 4 bits A e B é executada. Para isso, o vetor B é convertido para sua forma negativa usando complemento de dois, por meio da instrução ***"B_signed := std_logic_vector(unsigned(not B) + 1);"***. Essa conversão permite que a subtração A - B seja implementada como uma soma entre A e -B.

Em seguida, um laço **for** executa a soma bit a bit utilizando lógica de somador completo. A cada iteração, os bits correspondentes de **A** e **B_signed** são somados, junto ao bit de transporte **c_in**, gerando o bit resultante sub(i) e atualizando o transporte de saída

**c_out**. Após o término do laço, o vetor **op_result** armazena o resultado da subtração com a instrução ***"op_result := std_logic_vector(sub);"***.

Nesse contexto, é importante destacar que, ao realizar a subtração como uma soma com complemento de dois, o bit de transporte final (**c_out**) está diretamente relacionado à noção de "borrow out" (carry out = NOT(borrow out)). Se não há transporte (carry = 0), isso indica que houve um empréstimo na subtração, ou seja, o minuendo é menor que o subtraendo.

Logo após, são verificadas condições para definir as flags associadas à operação. Caso haja overflow, quando os sinais mais significativos dos operandos diferem e o resultado tem sinal oposto ao de A, a variável **flags** é configurada com "0100" ou "1100", dependendo do sinal do resultado. Se o resultado for negativo sem overflow, a flag é "1000". Caso contrário, a flag permanece "0000". Como as flags estabelecidas não envolviam a de borrow out, ele não foi incluído no código.

**Figura 21:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

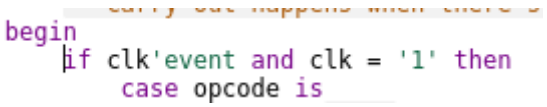



**Figura 22:** Lógica do subtrator entre operandos com sinal e flags

```
when "110" => -- 2's Complement Subtractor

    B_signed := std_logic_vector(unsigned (not B)+1);

    c_in := '0';

    for i in 0 to 3 loop
        sub(i) := (A(i) xor B_signed(i)) xor c_in;
        c_out := ((A(i) xor B_signed(i)) and c_in) or (A(i) and B_signed(i));
        c_in := c_out;
        end loop;

    op_result := std_logic_vector(sub); -- 4 bits subtractor result


    if (A(3) /= B(3)) and (op_result(3) /= A(3)) then -- Overflow
        if op_result(3) = '1' then
            flags := "1100"; -- Overflow + Negative
        else
            flags := "0100"; -- Overflow
        end if;

    elsif op_result(3) = '1' then
        flags := "1000"; -- Negative (no overflow)
    else
        flags := "0000"; -- No flags
    end if;

    if c_out = '1' then
        flags := "0010" or flags; --Carry Out flag
    end if;
```

## Shift

É feito nessa operação um Shift (deslocamento), o primeiro operando (A) é o número inicial e o segundo operando (B) indica como deverá ser o deslocamento, as duas casas menos significativas (B(1) e B(0)) indicam de quanto, em binário, será o deslocamento, podendo acontecer entre os valores de 0 e 3 casas, o terceiro bit de B (B(2)) indica se o deslocamento ocorrerá para a esquerda, se B(2) = ‘0’, ou para a direita, se B(2) = ‘1’. O bit mais significativo de B(B(3)) indica se as casas movidas devem ser preenchidas com 0, se B(3) = ‘0’, ou com 1 se B(3) = ‘1’.

Código VHDL:

Quando “***case opcode is when “111” =>”***, a operação é realizada sobre o vetor de 4 bits A, enquanto a quantidade de posições a ser deslocada é indicada por B.

Antes de realizar o deslocamento propriamente dito, o código verifica se a quantidade de deslocamentos (**shift_amount**) é maior ou igual a 4. Como os operandos possuem apenas 4 bits, qualquer deslocamento de 4 ou mais posições faz com que o conteúdo original se

perca completamente. Nesse caso, o resultado final é preenchido inteiramente com o valor contido no bit B(3), que define o bit de preenchimento (fill bit). Ou seja, se B(3) for igual a 0, o resultado será "0000"; se for igual a 1, o resultado será "1111".

Caso a quantidade de deslocamentos seja menor que 4, o código então analisa o bit B(2), que define a direção do deslocamento: se for 0, o deslocamento será para a esquerda; se for 1, para a direita. A operação de deslocamento é realizada utilizando as funções **shift_left** ou **shift_right**, aplicadas ao vetor A convertido para o tipo *unsigned*. Após esse deslocamento automático, o código entra em um laço for para sobrescrever os bits que foram "preenchidos" com o valor definido por B(3), garantindo que o bit de preenchimento seja coerente com a lógica pretendida pelo sistema. Nesta operação não existem flags, devido a isso foi definido "***flags := "0000";***".

**Figura 23:** Trecho do código onde o processo é ativado por clock e o **opcode** define a operação realizada

```
begin
    if clk'event and clk = '1' then
        case opcode is
```

**Figura 24:** Lógica do shift entre operandos sem sinal e flags

```
   when "111" => -- Shift
shift_amount := to_integer(unsigned(B(1 downto 0)));

if shift_amount >= 4 then
    if B(3) = '0' then
        op_result := (others => '0');
    else
        op_result := (others => '1');
    end if;

else
    if B(2) = '0' then -- Shift to the Left
    -- Left shift w/ manual filling
        op_result := std_logic_vector(shift_left(unsigned(A), shift_amount));
        for i in 0 to 3 loop
            if i <= (shift_amount - 1) then
                op_result(i) := B(3); -- will fill with 0's, if B(3) = 0, or 1's, if B(3) = 1
            end if;
        end loop;

    else -- B(2) = '1' => Shift to the Right
    -- Right shift w/ manual filling
        op_result := std_logic_vector(shift_right(unsigned(A), shift_amount));
        for i in 3 downto 0 loop
            if i <= (4 - shift_amount) then
                op_result(i) := B(3); -- will fill with 0's or 1's
            end if;
        end loop;
    end if;
end if;
flags := "0000"; -- shift doesn't alter flags
```

## Flags

As flags foram designadas em one-hot como Zero = “0001”, Negativo = “0010”, Carry Out = “0100”, Overflow = “1000”, permitindo a ativação de mais de uma flag simultaneamente.

- ➔ **Zero:** Indica quando todos os bits da saída são 0.
- ➔ **Negativo:** Indica quando a saída representa um número negativo, no Complemento de 2.
- ➔ **Carry out:** Indica se a operação aritmética ultrapassou o limite de representação de 4 bits.
- ➔ **Overflow:** Indica que o resultado de uma operação aritmética excede o intervalo de representação possível, ocorre quando o sinal do resultado está incorreto, pelo transbordamento do bit mais significativo (bit de sinal).

Código VHDL:

Para a implementação das flags no código VHDL, as condições referentes às flags carry out, overflow e negativo foram definidas dentro de cada operação, de acordo com a coerência de cada bloco, como foi explicado anteriormente. Após o bloco de case-when vinculado ao opcode, a flag Zero é adicionada às outras conforme o **op_result**, ou seja, “**flags := "0001" or flags;**” caso o resultado da operação seja 0.

**Figura 25:** Flag Zero

```
if op_result = "0000" then
   flags := "0001" or flags; -- Zero Flag
end if;
```

Por fim, para a obtenção do vetor de resultado final **result**, as variáveis **op_result** e **flags** são concatenadas, como a figura ilustra abaixo:

**Figura 26:** Concatenação das variables **flags** e **op_result** no signal de saída **result** (8 bits)

```
    result <= flags & op_result (3 downto 0); --8 bit output: flags (4bits) + op_result (4bits)
end process;
```

# Simulações

Durante o desenvolvimento, 2 simuladores foram usados para os testes dos 3 arquivos de forma separada e com diferentes testbenchs.

O simulador online EDA-playground foi útil por ser um programa leve e sem necessidade de instalação, permitindo o acesso de vários membros do grupo. No entanto, sua simulação é limitada para um único arquivo de código e para lógicas que não dependem do clock, pois não comporta um testbench mais de um **process**, além de não acusar todos os problemas de compilação acusados em laboratório

Para testes mais elaborados, foi utilizado o simulador Quartus e Modelsim-Altera indicado no poli moodle, que, por ser um programa pesado e que exige bastante espaço de memória, apenas um membro do grupo pôde acessar. Esse simulador foi essencial para o teste de arquivos que necessitam do clock, como o arquivo Debounce, e para os testes do projeto finalizado, em que todos os arquivos são sincronizados com o clock e o Debounce e Operações funcionam como componentes do arquivo principal.

Dessa forma, os testes foram realizados inicialmente de forma isolada, com testbenches diferentes e os estímulos correspondentes de entrada e saída de cada código, e depois de forma unificada, com um testbench similar ao funcionamento real do projeto.

## Simulação Debounce:

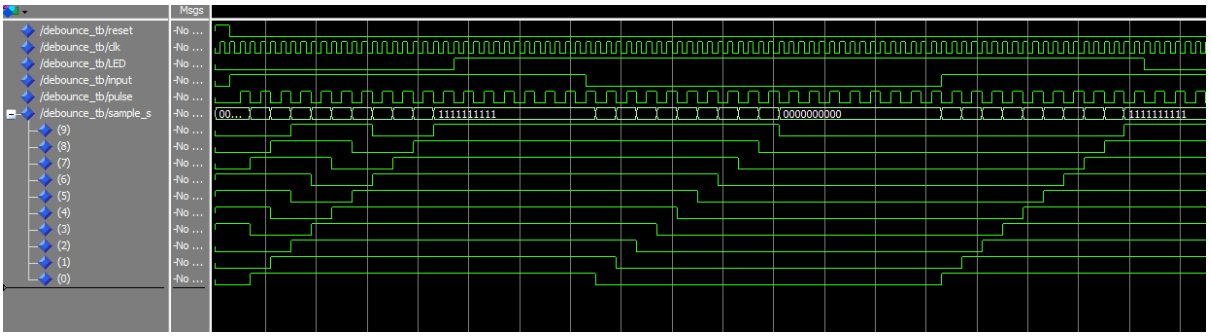
O arquivo de debounce foi desenvolvido principalmente com o simulador Quartus e Modelsim-Altera, no resultado da simulação isolada do arquivo de debounce pode-se observar uma lógica semelhante à descrita anteriormente. O arquivo simulado foi posteriormente testado em laboratório, e seu objetivo era de testar o debounce através do funcionamento de um LED, que deveria acender quando o botão for pressionado e apagar apenas quando o botão for pressionado novamente, incorporando o seguinte código no arquivo de debounce original:

**Figura 27:** Código de teste Debounce:

```
LED <= control;
pulse <= sample_pulse;
sample_s <= sample;


process(clk)
begin
   if (clk'event and clk = '1') then
      if(debounce_s = '1') then
         control <= not control;
      end if;
   end if;
end process;
```

Os estímulos no testbench desse arquivo são de clock, reset (retirado posteriormente) e input (nosso botão), e as saídas são o pulse, o LED e o sample_s. Na imagem, é possível visualizar o processo de shift da amostragem (sample_s), o clock com frequência mais baixa (pulse) e o próprio debounce através do LED, que só é ativado na borda de subida do clock quando o sample_s é "1111111111", ou seja, após 10 pulsos do clock retardado com o input ativo, e só é desativado na mesma condição.

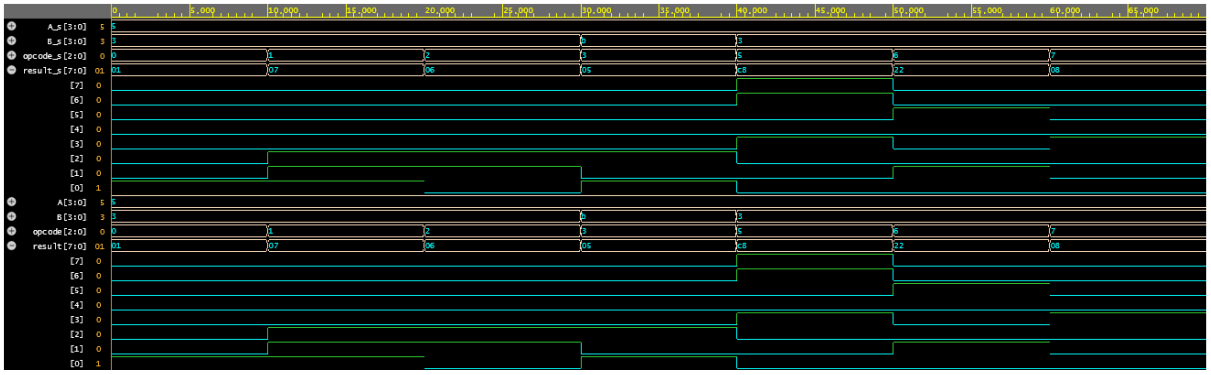
**Figura 28:** Simulação do Debounce com LED




## Simulação Operações:

O arquivo de operações foi desenvolvido principalmente no simulador online EDA-playground, pois a lógica desse arquivo não necessitava de clock. Não foi feito um arquivo de teste isolado no laboratório, mas foi elaborado um testbench, com os estímulos de **operation**, operando **A** e operando **B**, em que a saída é dada pelo **result** (8 bits). É possível conferir o resultado de todas as operações com os números A = "0101" e B = "0011" e apenas na operação de módulo B = "1011" na imagem abaixo:

**Figura 29:** Simulação do arquivo de operações.

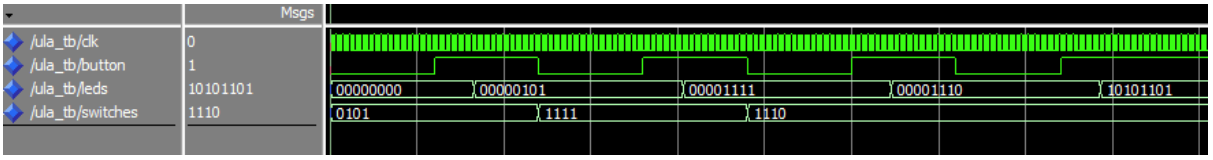


## Simulação ULA:

O arquivo de principal também foi testado de forma isolada, mas seu funcionamento também pode ser visualizado na simulação do projeto final com todos os arquivos integrados. O testbench desse arquivo simula o funcionamento real do projeto, em que os estímulos são o clock, o button e os switches, e sua saída são os LEDs.

Na imagem pode-se visualizar o funcionamento da máquina de estados, que registra e mostra os bits indicados no switch após alguns pulsos de clock com o button ativado, e após fazer isso 3 vezes (estados 0, 1 e 2), exibe o resultado da operação quando o botão é ativado pela quarta vez. O exemplo da imagem é uma soma entre os números “1111” e “1110”, que gera a flag de negativo e carry, com o resultado sendo “1101”:

**Figura 30:** Simulação da ULA completa.



# Desafios do projeto

Ao longo da construção e simulação, diversos desafios foram enfrentados, contribuindo para o aprofundamento do entendimento sobre os conceitos abordados na disciplina, além de proporcionar maior familiaridade com a linguagem VHDL, o funcionamento de sistemas digitais e os obstáculos práticos que surgem na aplicação desses conhecimentos no mundo real.

Um dos primeiros desafios encontrados foi a configuração incorreta do pino de clock na placa, o que impediu o funcionamento adequado do sistema. Esse problema nos levou a investigar mais profundamente o funcionamento da FPGA utilizada, permitindo descobrir que a placa possui múltiplos pinos de clock disponíveis. Assim, foi necessário identificar corretamente qual pino estava habilitado para uso no projeto e realizar o ajuste adequado, garantindo que a lógica e comunicação entre arquivos funcionasse corretamente.

Outro ponto de dificuldade foi o uso de lógica combinacional no processo do arquivo de operações. Inicialmente, esse arquivo foi desenvolvido e testado de forma isolada, sem considerar a interação com a máquina de estados e o uso do clock implementado no arquivo de debounce. Ao integrar os três arquivos — ULA, debounce e operações — o sistema passou a apresentar resultados incorretos: independentemente da operação selecionada, a flag zero era ativada. Isso ocorreu porque o processo dentro do arquivo de operações era sensível apenas a sinais combinacionais (***"process (A, B, opcode)"***), enquanto os outros dois módulos operavam com base na borda de subida do clock. Além disso, como a atribuição da flag zero estava fora da estrutura **case**, sua ativação acontecia de forma assíncrona e aleatória, interferindo no resultado final.

Durante a criação da operação do somador de 4 bits em complemento de dois, surgiram dúvidas importantes sobre os conceitos de carry out e overflow, especialmente sobre como diferenciá-los corretamente no contexto de aritmética binária com sinal. O carry out representa o transporte que sai do bit mais significativo (MSB) durante uma operação aritmética, enquanto o overflow ocorre quando o resultado da operação excede o intervalo de representação possível para números com sinal, como no caso de somar dois números positivos e obter um negativo, ou vice-versa. Também houve confusão em relação à flag zero, mais especificamente sobre o que ela deveria sinalizar: se representava unicamente a ocorrência do valor binário "0000" no resultado, ou se deveria indicar um "zero verdadeiro", no sentido lógico da operação executada. Nesse último caso, a flag representa o valor binário "0000".

Por fim, uma das dificuldades mais persistentes foi compreender e interpretar corretamente a flag de carry out em relação ao conceito de borrow out, especialmente porque optamos por reutilizar o laço **for** do somador para implementar a subtração, alterando o sinal da entrada B por meio do complemento de dois. Com isso, eventualmente a flag de carry out era acionada, o que gerou confusão na análise do funcionamento do subtrator. Através da busca na literatura e em vídeos no YouTube ficou claro que isso acontece porque o carry out do somador, que indica um transporte positivo na operação de adição, tem um significado invertido na subtração com complemento de dois, onde sua ausência pode indicar que houve um borrow.

# Conclusão

Durante o desenvolvimento do projeto da ULA em VHDL, foram criados três arquivos principais: um responsável pela lógica da Unidade Lógica e Aritmética (ULA), outro para o controle do debounce do botão e um terceiro contendo a implementação detalhada das operações.

Após etapas de pesquisa e aprendizagem dos softwares de simulação e da linguagem VHDL, o grupo desenvolveu os 3 arquivos de forma funcional, construindo um projeto de ULA de 8 operações para FPGA com aplicação prática. Dessa forma, o trabalho se constituiu como uma boa ferramenta de aprendizagem, incorporando desafios a serem solucionados em laboratório.

# Referências

Além dos materiais fornecidos no poli-moodle, foram acessados:


How to eliminate button bounces with digital logic. Disponível em: <https://zipcpu.com/blog/2017/08/04/debouncing.html>. Acesso em: 6 maio. 2025.

FPGA FOR BEGINNERS. Disponível em: <https://www.youtube.com/playlist?list=PLASHf7jviOmoArEq4TY1TlRfhttdddGWC>. Acesso em: 6 maio. 2025.

VHDL Button Debounce. Disponível em: <https://www.youtube.com/watch?v=eRawZX_R7Bg>. Acesso em: 6 maio. 2025.

GEEKSFORGEEKS. Full Subtractor in Digital Logic. Disponível em: <https://www.geeksforgeeks.org/full-subtractor-in-digital-logic/>. Acesso em: 6 maio. 2025.

GEEKSFORGEEKS. VHDL Very High Speed Integrated Circuit Hardware Description Language. Disponível em: <https://www.geeksforgeeks.org/vhdl-very-high-speed-integrated-circuit-hardware-description-language/>. Acesso em: 6 maio. 2025.


## Tutoriais simulador


Installing Quartus, ModelSim & MAX10 Drivers (March 2023). Disponível em: <https://www.youtube.com/watch?v=eOWBF4kNM8w>. Acesso em: 6 maio. 2025.

Intel Quartus Prime Lite edition | Behaviourial Simulation using VHDL Testbench code. Disponível em: <https://www.youtube.com/watch?v=76cNeZWTjc0>. Acesso em: 6 maio. 2025.