# 

# Sumário

[**Sumário 1**](#_oenv5sz9fy6z)

[**Alunas: 2**](#_t2htk52r2shx)

[**Introdução 3**](#_fczjbjm3jb2j)

[**Sobre VHDL 4**](#_sr31gw2vw74i)

[Entidades 4](#_8sbn6fv21yq6)

[Arquiteturas 4](#_mcjr6js1fq3v)

[PortMap 4](#_y4n1en1nv17t)

[Função Rising Edge 4](#_5ad29ty27cw8)

[Processos 4](#_71gflndkjpfd)

[**Desenvolvimento 5**](#_33bpd0vvaju3)

[Operações lógicas 5](#_8hk51414owm2)

[**AND 5**](#_m8bb2fyk6nvf)

[Código AND: 5](#_b2lirh5eoi81)

[**OR 6**](#_22r0prvj8pnu)

[Código OR: 6](#_7vo7w59ew9l5)

[**NOT 7**](#_lxgxpkwaibx)

[Código NOT: 7](#_m2ikq7hc41k1)

[Operações aritméticas 8](#_n00mkebvruvl)

[**SOMADOR COMPLETO 8**](#_n4c1h6ttyukv)

[Somador 1 bit 8](#_vigj059mcyfe)

[Somador 4 bits 8](#_ie06rirerfnr)

[Código ADDER: 9](#_afkd85vblgmx)

[Código FULL ADDER 4 BITS: 9](#_28pxngn59q8)

[**COMPLEMENTO DE 2 10**](#_hv69r3jrkdso)

[Código Complemento de 2: 10](#_j591j6ixycdz)

[**SUBTRATOR 10**](#_r303qxmeqfpt)

[Código Subtrator: 12](#_seghu83gahd1)

[**DESLOCADOR 13**](#_kskfh6t3ktxt)

[Código Deslocador: 13](#_xlppe471n6lo)

[**Main 16**](#_wtoh96vzo4y1)

[Código Main: 17](#_w1rk2hdsyrc6)

[**Interface 19**](#_459a264mrv5u)

[Controle do clock 19](#_d4hm4u6vfux0)

[Interface 20](#_5kafd5kopn3d)

[Código Interface: 22](#_dp4byeodcr8x)

[Pinagem 25](#_reg5obghpei7)

[Código Pinagem: 25](#_u6q29zbrtgl0)

[**Resultados - Simulações 27**](#_1qjxqlyyqw0f)

[Operações lógicas 27](#_lwt09p5pdn2h)

[AND 27](#_1j6glvj7luqp)

[OR 27](#_l12ukasqljfa)

[NOT 28](#_glsykjmexfp7)

[Operações aritméticas 28](#_wzivc24jbhs)

[SOMADOR COMPLETO 28](#_ii9b4w6n8moz)

[COMPLEMENTO DE 2 29](#_bnvjdemt4rdl)

[SUBTRATOR 29](#_4u2cce3vfye3)

[DESLOCADOR 29](#_44zqbsj5qo3)

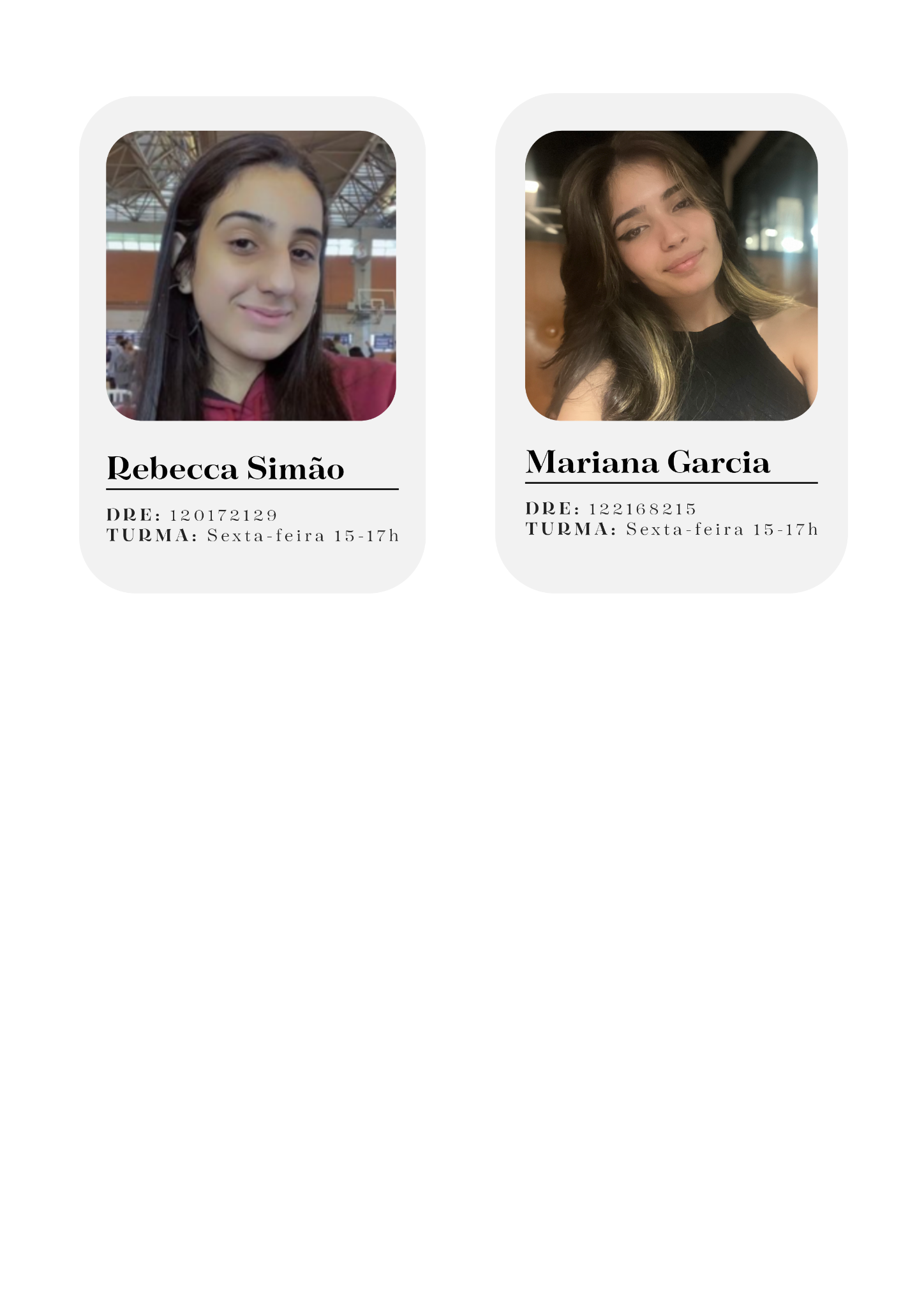
[MULTIPLICADOR 30](#_whlb6gwrnc5f)

[**Conclusão 31**](#_4d92dvcio82v)

[**Agradecimentos 31**](#_h4pvroyny0v9)

# 

# Alunas:



# 

# Introdução

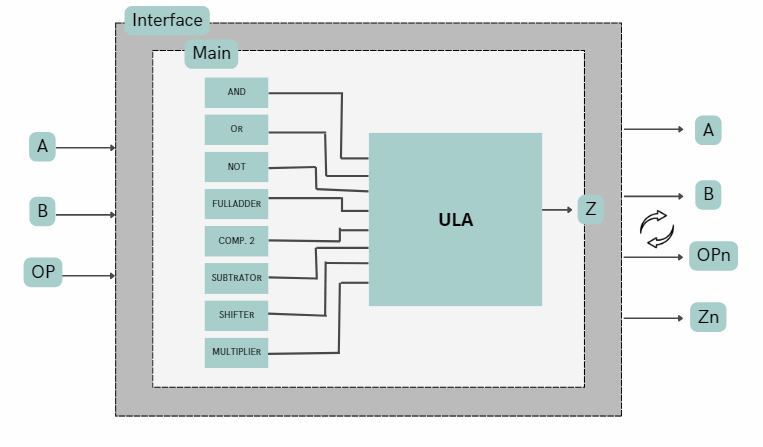
O relatório abrange o desafio da construção de uma ULA (Unidade lógica e aritmética) que receba o equivalente ao operando A (4 bits), ao operando B (4 bits) e ao bit de controle da ULA, que definirá a operação a ser realizada (3 bits). Posteriormente, espera-se na saída que se mostre os operandos escolhidos, a operação solicitada e seu resultado; entrando em um looping o qual sempre mostrará os operandos, a operação posterior da ULA e seu resultado, sucessivamente até que o sistema seja resetado. Sua visualização se dará por meio de um sistema de interface implementado que será também explicado.

Para sua realização, foram utilizadas

* Linguagem de programação: VHDL (VHSIC Hardware Description Language) - linguagem de descrição de hardware usada para descrever o comportamento e estrutura de um sistema digital;
* Placa: Spartan-3A/3AN;
* Programa: Oracle Virtual Box.

Assim, foi começado o projeto estruturando a ULA nas suas 8 operações escolhidas: AND, NOT e OR (operações lógicas), além da implementação de um Somador, Complemento de 2, Subtrator, Deslocador e Multiplicador (operações aritméticas). O módulo Main.vhd foi criado com a finalidade de unir os comandos internos da ULA e o módulo Interface.vhd, para permitir interação externa com os comandos internos do projeto.

Modelo esboçado:



# Sobre VHDL

VHDL (VHSIC Hardware Description Language) é uma linguagem de descrição de hardware amplamente utilizada para projetar e simular circuitos digitais. Aqui está uma explicação sobre os principais conceitos associados ao VHDL:

## **Entidades**

Em VHDL, uma entidade (`entity`) é uma descrição de uma interface do componente que define os sinais de entrada e saída. Ela atua como uma interface externa para o componente, especificando os sinais que o componente espera receber e os sinais que ele produz como saída. A entidade não define o comportamento interno do componente, apenas suas interfaces.

## **Arquiteturas**

Uma arquitetura (`architecture`) em VHDL descreve o comportamento interno de uma entidade. Em outras palavras, ela implementa a lógica do componente. Uma entidade pode ter várias arquiteturas, cada uma descrevendo uma implementação diferente do mesmo componente.

## **PortMap**

O port map (`port map`) é usado para conectar instâncias de entidades a uma arquitetura específica. Ele associa os sinais das instâncias das entidades aos sinais da arquitetura que estão conectados a essas entidades.

## **Função Rising Edge**

A função `rising\_edge` é uma função em VHDL usada para detectar a transição de um sinal de baixo (0) para um sinal de alto (1) em um sinal de clock. Ela é comumente usada dentro de processos sensíveis ao clock para disparar a execução de código em borda de subida do clock.

## **Processos**

Um processo (`process`) em VHDL é uma construção que descreve um bloco de código executado sequencialmente. Ele é usado principalmente para descrever o comportamento combinacional ou sequencial de um componente digital.

## 

# 

# Desenvolvimento

O desenvolvimento do projeto foi dividido em 4 partes:

* Desenvolvimento das operações lógicas (AND, NOT e OR);
* Desenvolvimento das operações aritméticas (Somador, Complemento de 2, Subtrator, Deslocador e Multiplicador);
* Desenvolvimento da Main para controle interno;
* Desenvolvimento da Interface para comunicação entre o Hardware e o código.

## **Operações lógicas**

### AND

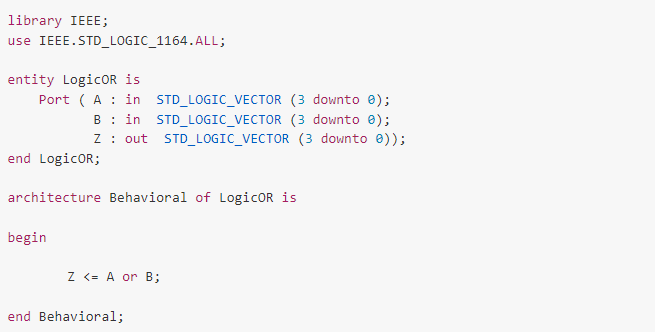
Para implementação dos módulos, é importante entender o funcionamento destes. Assim, o módulo lógico AND envolve a operação bit a bit do operando A com o operando B, tendo essa tabela-verdade:

| A | B | Z |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Z (saída) = AB**

#### 

#### **Código AND:**





### OR

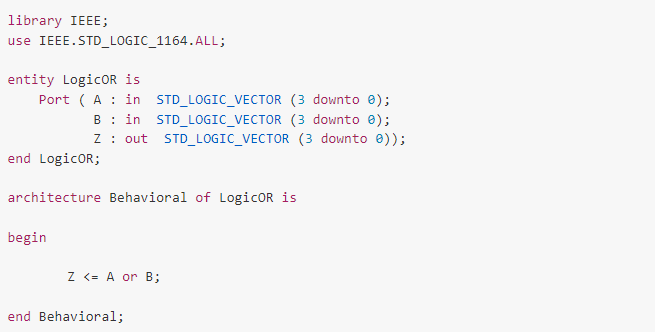
Por outro lado tem-se o módulo lógico OR, que envolve a operação bit a bit do operando A com o operando B, tendo essa tabela-verdade:

| A | B | Z |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Z (saída) = A + B**

#### 

#### **Código OR:**



### 

### NOT

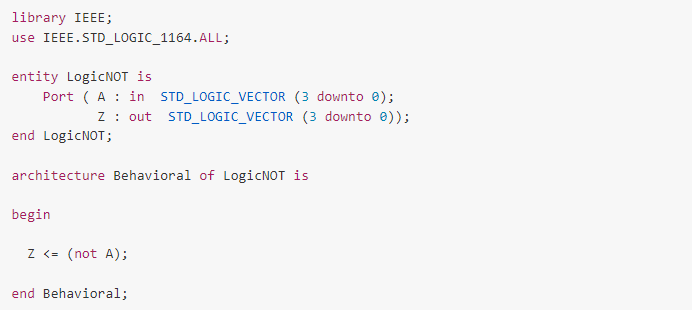
Para implementação dos módulos, é importante entender o funcionamento destes. Assim, o módulo lógico AND envolve a operação bit a bit do operando A com o operando B, tendo essa tabela-verdade:

| A | Z |
| --- | --- |
| 0 | 1 |
| 1 | 0 |

**Z (saída) = A’**

#### 

#### **Código NOT:**

****

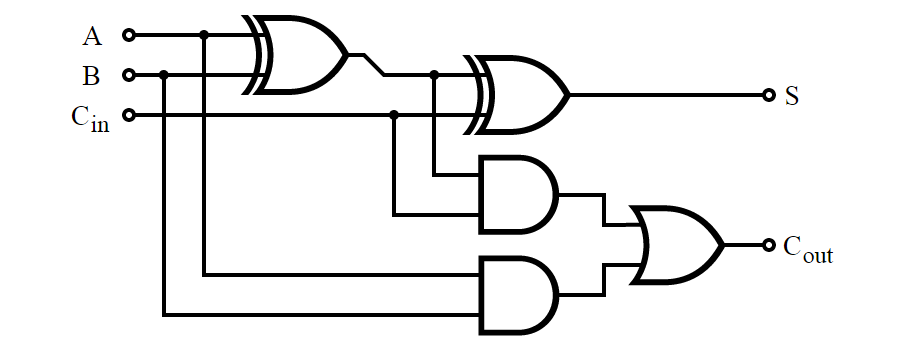
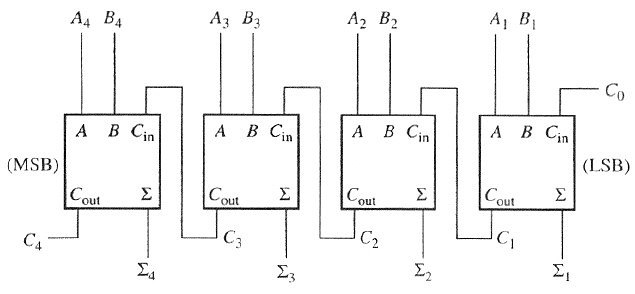
## 

## **Operações aritméticas**

Já as aritméticas envolvem uma complexidade maior de seu funcionamento para sua correta implementação.

### SOMADOR COMPLETO

Para implementar o somador completo de 4 bits, faz-se necessária a criação do módulo Somador de 1 bit. sendo o somador completo de 4 bits a união de 4 somadores de 1 bit.

Somador de 1 bit Somador de 4 bits

#### 

#### **Somador 1 bit**

Primeiro cria-se a entidade com:

* Entradas A, B, Cin;
* Saídas SUM e Cout.

Sendo em sua arquitetura:

* Soma = A xor B xor Cin
* E o carry de saída: Cout = (AB) + (Cin(A ⊕ B)

#### **Somador 4 bits**

Posteriormente cria-se outro módulo, este então será o Somador completo de 4 bits:

* Entradas da entidade: A e B de 4 bits; e um Carry inicial;
* Saídas: Z representando os 4 bits mais significativos da soma; e o Carry de saída.

E sua arquitetura chamando o somador supracitado.

A criação do sinal Carry de 4 bits se dá para o funcionamento do somador, que terá o Carry(0) assumindo valor de Cin e os demais, os carrys resultantes das operações de soma dos bits correspondentes.

Por fim, envia-se para o somador simples os bits correspondentes dos operandos e o carry, retornando o resultado da soma bit a bit para ser armazenado no vetor de saída Z.

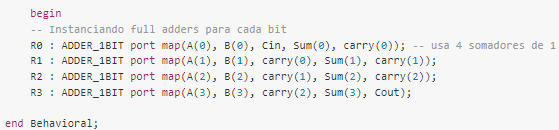
#### 

#### **Código ADDER 1 BIT:**



#### 

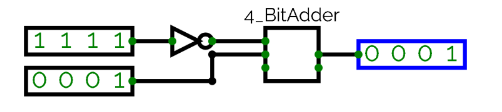
#### **Código FULL ADDER 4 BITS:**



### 

### COMPLEMENTO DE 2

Para o complemento de 2, deve-se inverter o Bit de entrada e somar 0001 a ele:



Módulo Complemento de 2 utilizando not e módulo somador.

Assim, em sua lógica tem-se A de entrada (4 bits) e Z de saída (4 bits).

Para que se some A invertido com 0001, faz-se pertinente a utilização do módulo somador 4 bits utilizado anteriormente, enviando para ele:

* O bit A invertido (A), 0001 (B), 0 (Cin) e recebendo a Soma dos bits com o Carry out da operação;

Seguindo o envio dos demais bits até que se consiga guardar em Z o resultado.

#### 

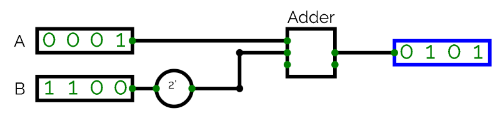
#### **Código Complemento de 2:**

### 

### 

### SUBTRATOR

Para o Módulo subtrator, deve-se somar A com (-B), para tal, utiliza-se o complemento de 2 em B para retornar (-B) e um somador para finalizar a operação:



Módulo Subtrator utilizando módulo somador e módulo complemento de 2.

Na entidade:

* Entradas: A e B de 4 bits;
* Saídas: Z representando os 4 bits mais significativos da soma; e o Carry de saída.

E sua arquitetura chamando o módulo complemento de 2 e o módulo somador completo de 4 bits, como supracitado.

* De início envia pelo recurso “port map” o B para função Complemento de 2, tendo como retorno B2 - equivalente a saída (-B);
* Por fim, soma-se ambos com auxílio do somador completo de 4 bits.

#### 

#### **Código Subtrator:**

### 

### 

### 

### DESLOCADOR

O módulo deslocador recebe um A (4 bits) e um Q (2 bits), devolvendo um Z (4 bits) correspondente.

O Q define a quantidade de deslocamentos que irão ocorrer, podendo ser 0, 1, 2 ou 3.

| Q | Z |
| --- | --- |
| 00 | A3 A2 A1 A0 |
| 01 | 0 A3 A2 A1 |
| 10 | 0 0 A3 A2 |
| 11 | 0 0 0 A3 |

Em sua arquitetura foi criado um processo para ligar a quantidade de deslocamentos e a entrada com a respectiva saída, usando um If para diferenciar as operações. Assim, quando precisa-se deslocar, o Z armazena a quantidade de 0 necessária e mescla com os bits restantes do operando original.

#### 

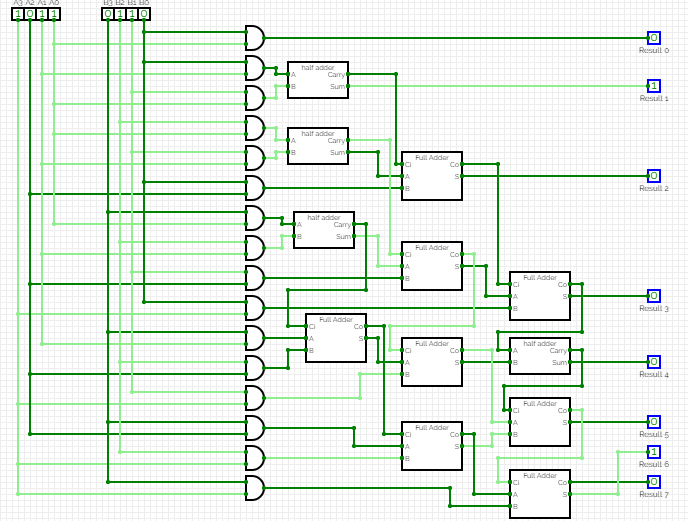
#### **Código Deslocador:**



**MULTIPLICADOR**

Por fim, o módulo multiplicador recebe A e B (4 bits) e retorna Z com os 4 bits mais significativos do resultado.

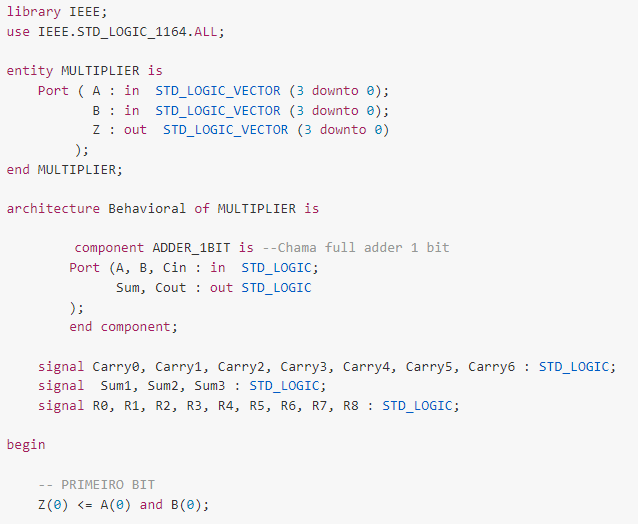
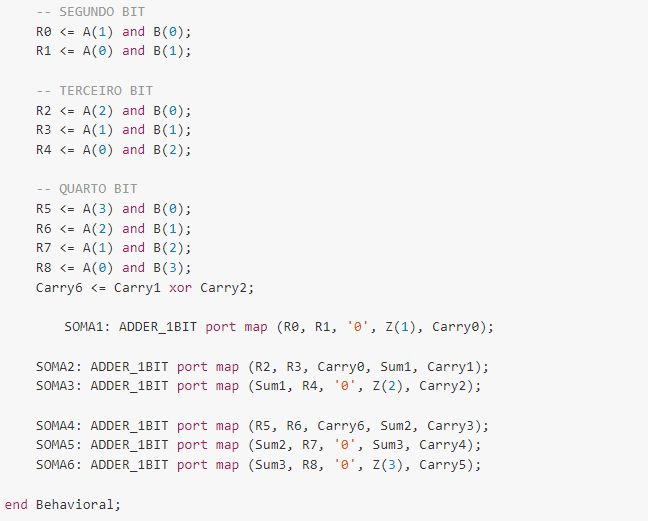
Circuito multiplicador para melhor entendimento do código:



Para tal, foi implementado um multiplicador tipo array de 4x4, utilizando ANDs e Somadores. Sua arquitetura inicia chamando o somador completo de 1 bit e definindo seus sinais auxiliares de Carry (CarryX) e soma (SumX), além de resultados (RX).

Feito isso, inicia as operações em si com os ANDs dos bits correspondentes, para que sejam aplicados posteriormente nos somadores e darem os resultados. Conforme a limitação do uso de Leds no desafio, nosso multiplicador não utilizará os 8 bits, mas sim 4.

**Código Multiplicador:**

****

## **Main**

Para implementação da Main precisou-se de um entendimento acerca das suas aplicações.

**Objetivo:** Receber A, B e operação do switch, encontrar operação correspondente, chamar função e devolver seu resultado. Uma maneira centralizada das operações, nossa ULA.

Assim, sua entidade pede A (4 bits), B (4 bits), OP (3 bits), e retorna Z (4 bits).

E sua arquitetura se dividiu em partes:

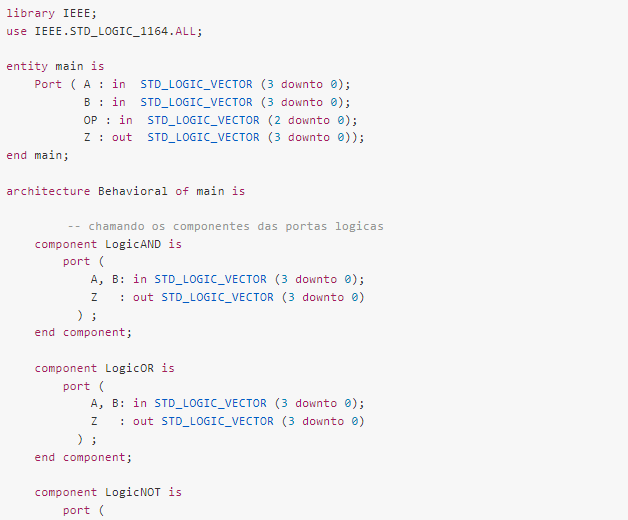
1. Apresentar as 8 operações;
2. Definir sinais auxiliares;

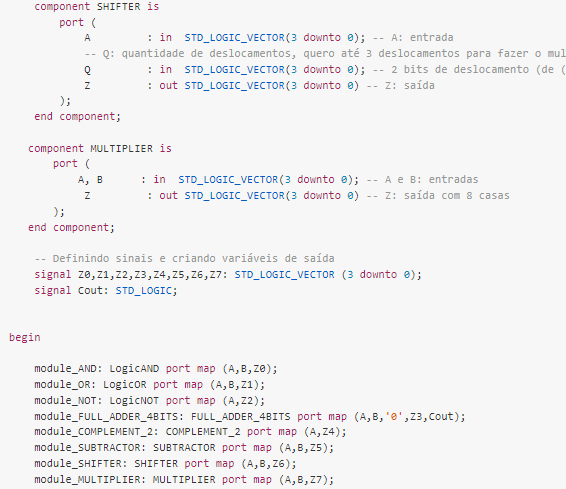
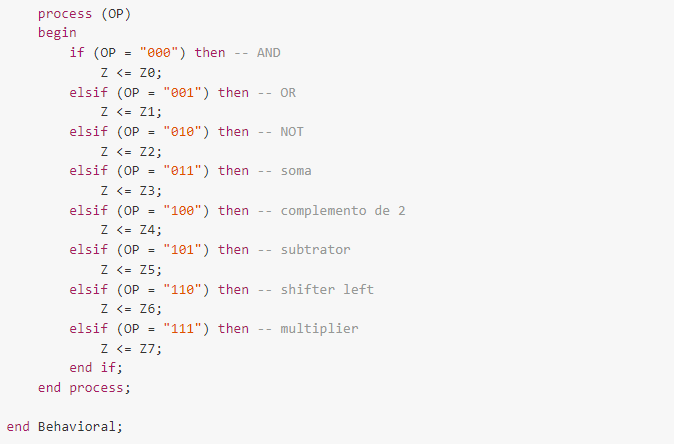
* Zx - saídas equivalentes a cada operação;
* Cout - Carry para ser saída do fulladder.

1. Pós begin, utiliza-se os Portmaps para reaproveitamento das entidades definidas na ULA, ele será o responsável por jogar nossos operadores na operação requisitada e devolver um resultado esperado;
2. Um processo é aberto com sensibilidade à mudança de OP, que, utilizando sistema de Ifs e Elsifs, fará a saída Z assumir o valor da saída da operação requisitada.

#### 

#### **Código Main:**

****

****

## **Interface**

Para implementação da interface do usuário, precisa-se de entendimento acerca de seu funcionamento, dividindo sua lógica em 2 partes:

Parte 1: Pegar bits

1. Com um switch, o usuário selecionará os bits de A e apertará um botão;
2. Com o mesmo switch, o usuário selecionará os bits de B e apertará um botão;
3. Da mesma forma ocorrerá para selecionar os bits de OP, e apertará o botão que iniciará a visualização do sistema.

Terão 8 Leds para visualização do sistema, sendo os mais à esquerda para mostrar o que está acontecendo e os mais à direita para revelar os Bits de A, B, operando ou Z.

Parte 2: Mostrar sequências

1. Vai mostrar A nos Leds com: 1000 (Informa ao usuário que vai mostrar A) e A3 A2 A1 A0 (mostra ao usuário, A). Após 2 segundos ->
2. Mostra B nos Leds com: 0100 (Informa ao usuário que vai mostrar B) e B3 B2 B1 B0 (mostra ao usuário, B). Após 2 segundos ->
3. Mostra OP nos Leds com: 0010 (Informa ao usuário que vai mostrar OP) e 0 OP2 OP1 OP0 (mostra ao usuário, OP). Após 2 segundos ->
4. Mostra Z nos Leds com: 0001 (Informa ao usuário que vai mostrar Z) e Z3 Z2 Z1 Z0
5. Os processos são repetidos alterando OP na sequência (OP requisitado -> Op requisitado + 1 -> Op requisitado + 2 -> …). Em looping até que um reset o faça retornar ao início para pegar outros bits de entrada.

Feito isso, precisa-se implementar uma máquina de estados para variar os estados do sistema, conseguindo bom funcionamento da visualização. Além de um controle de Clock para sincronizar as mudanças após 2 segundos.

### Controle do clock

Como explicado, precisou-se implementar um controle de clock para alterar o tempo de mudança de estado. O clock da placa utilizada tem frequência de 50M Hz, por tal, tem-se um processo sensível a mudança de clock da placa e:

1. Foi criada uma variável auxiliar que se inicia com valor 0 e que vai sendo incrementada a cada 100M Hz (dada utilização de um rising edge, espera-se transição de 0->1 e de 1->0 - 50M Hz \* 2 = 100M Hz);
2. Quando o auxiliar chegar a 100 milhões, significa ter passado 2 segundos.

Assim, conseguiu-se elevar o tempo para essa mudança até o escolhido.

### 

### Interface

De início, cria-se sua interface com:

* Entradas:
  + switches, como vetor de 0 a 3, para pegar operandos;
  + botão, reset, clock como sinais simples.
* Saídas:
  + Leds\_right, vetor de 0 a 3 para representar os LEDS de representação dos operandos e resultados;
  + Leds\_left, vetor de 0 a 3 para representar o que está sendo feito - exemplo: Mostrar A -> Leds\_left: 1000 e Leds\_right: A3 A2 A1 A0.

Para sua arquitetura,

1. Iniciei chamando a main e clock
2. Criei sinais e estados

* Estados:
  + Ai, Bi e OPi: estados iniciais para pegar A, B e OP
  + Zinicial, Za, Zb, Zc, Zd, Ze, Zf, Zg, Zh: Estados para mostrar resultados da operação referente;
  + Ax, Bx e OPx: estados para mostrar A, B e OP.

1. Assumiu-se leds auxiliares como leds right e left para guardar seus valores sem problema de reescrever os dados;
2. Foi utilizada um port map para chamar as funções dentro de main e ajustar o clock ao adequado;
3. Utilizando um processo sensível a mudança de botão, pega A, B e OP e utiliza OP para descobrir a operação que iniciará o sistema;
4. Por fim, outro processo é definido, sensível a clock, estado e , ele, caso reset não selecionado, assume a primeira operação reconhecida anteriormente pelo sistema e através dela vai entrar no ciclo de operações até que o reset seja selecionado.

**Código Controle de clock:**



### 

### 

### 

### 

#### **Código Interface:**

#### 

#### 

#### 

#### 

#### 

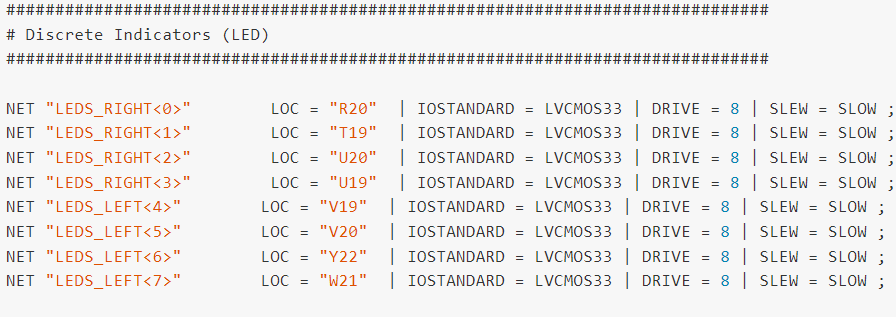
#### 

### Pinagem

### 

#### Código Pinagem:

Inicialmente, no arquivo de pinagem, estão definidas as localizações físicas (LOC) para os LEDs. Cada linha descreve a conexão entre um sinal (NET) e seu respectivo pino no FPGA, especificando o padrão de tensão (IOSTANDARD), a capacidade de acionamento (DRIVE), e a taxa de variação (SLEW) do sinal.



Na seção abaixo, são especificadas as localizações físicas para as chaves mecânicas (SW). Cada linha define a conexão entre um sinal (NET) e seu respectivo pino no FPGA, utilizando o padrão de tensão LVCMOS33.

As localizações físicas para os botões direcionais (BTN) também são definidas nesta seção. A linha NET "botao" CLOCK\_DEDICATED\_ROUTE = FALSE; desativa o roteamento dedicado de clock para o sinal "botao", permitindo um roteamento subótimo. No entanto, isso não é recomendado para designs de alta performance.

### 

Por fim, está definida a localização física para o clock principal do sistema. A linha NET "clock" LOC = "E12" | IOSTANDARD = LVCMOS33 | PERIOD = 20.000 ; OFFSET = IN 10.000 VALID 20.000 BEFORE "clock" ; OFFSET = OUT 20.000 AFTER "clock" ; especifica a conexão do sinal de clock com seu respectivo pino no FPGA, além de configurações de offset e período.

### 

# Resultados - Simulações

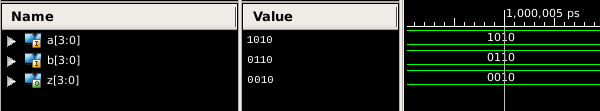
As simulações são sistemas que representam suas operações ao longo do tempo, sendo uma parte fundamental do fluxo de desenvolvimento de projetos FPGA. Elas permitem a verificação detalhada e precoce do design, economizando tempo e reduzindo os erros associados ao hardware. Através de simulações funcionais e temporais, os designers podem garantir que seus projetos não apenas atendem às especificações funcionais, mas também cumprem os requisitos temporais e de desempenho.

A seguir, estão os resultados das simulações dos códigos:

## **Operações lógicas**

### AND

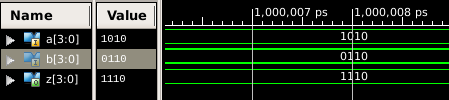
A função AND foi aplicada aos sinais vetoriais com números binários "1010" e "0110", resultando em uma saída esperada de "0010". A simulação confirmou que a saída da função AND foi, de fato, "0010", demonstrando que a saída é alta (1) apenas quando ambos os bits correspondentes nas entradas são 1. Isso confirma a correta implementação da lógica AND, onde cada bit da saída é 1 somente se os bits correspondentes das entradas forem ambos 1.

****

### 

### OR

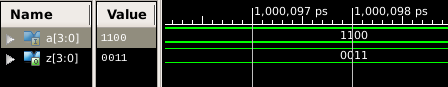
A função OR foi aplicada aos sinais vetoriais com números binários "1010" e "0110", resultando em uma saída esperada de "1110". A simulação confirmou que a saída da função OR foi, de fato, "1110", demonstrando que a saída é alta (1) quando pelo menos um dos bits correspondentes nas entradas é 1. Isso confirma a correta implementação da lógica OR, onde cada bit da saída é 1 se qualquer um dos bits correspondentes das entradas for 1.



### 

### NOT

A função NOT foi aplicada ao sinal vetorial com o número binário "1100", resultando em uma saída esperada de "0011". A simulação confirmou que a saída da função NOT foi, de fato, "0011", demonstrando que a saída é alta (1) onde os bits correspondentes da entrada são baixos (0) e vice-versa. Isso confirma a correta implementação da lógica NOT, onde cada bit da saída é o complemento do bit correspondente da entrada.



## 

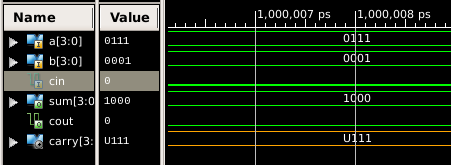
## 

## **Operações aritméticas**

### 

### SOMADOR COMPLETO

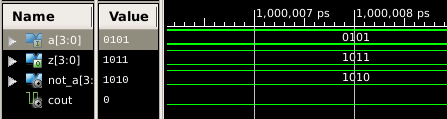
A simulação de um somador completo (full adder) de 4 bits foi realizada com os sinais de entrada "0111" e "0001". O resultado esperado para a soma desses dois números binários é "1000", sem carry out. A simulação confirmou que a saída do somador completo foi, de fato, "1000", indicando que a soma binária foi corretamente realizada. Este resultado demonstra a precisão do somador completo, que considera tanto as somas individuais dos bits correspondentes quanto os possíveis bits de carry gerados durante o processo, garantindo a adição correta dos números binários de múltiplos bits.



### 

### COMPLEMENTO DE 2

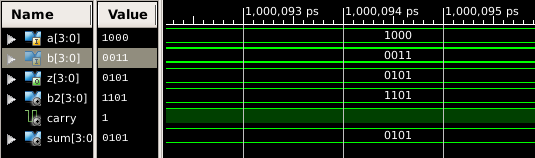
A função de complemento de 2 foi aplicada ao sinal de entrada com o número binário "0101", resultando em uma saída esperada de "1011". A simulação confirmou que a saída do complemento de 2 foi, de fato, "1011", demonstrando que o complemento de 2 é obtido invertendo todos os bits da entrada e somando 1 ao resultado. Isso confirma a correta implementação da lógica de complemento de 2, que é amplamente utilizada em representações binárias para operações aritméticas de números negativos.



### 

### SUBTRATOR

A simulação de um subtrator de 4 bits foi realizada com os sinais de entrada "1000" e "0011". O resultado esperado para a subtração desses dois números binários é "0101", sem borrow out. A simulação confirmou que a saída do subtrator foi, de fato, "0101", indicando que a subtração binária foi corretamente realizada. Este resultado demonstra a precisão do subtrator, que considera tanto a subtração dos bits correspondentes quanto os possíveis bits de borrow gerados durante o processo, garantindo a subtração correta dos números binários de múltiplos bits.

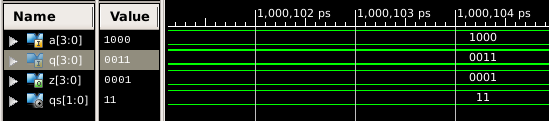


### 

### DESLOCADOR

A simulação de um deslocador foi realizada com o número binário inicial "1000" e o número de casas para deslocar sendo "0011". O resultado esperado é "0001".

Durante a simulação, a saída do deslocador foi confirmada como sendo "0001", indicando que o deslocamento para a esquerda foi realizado corretamente. Isso significa que os bits do número inicial foram deslocados para a esquerda por três posições, com zeros preenchendo as posições deslocadas. Essa operação é útil em várias aplicações, como multiplicação por potências de dois e na implementação de algoritmos de multiplicação e divisão em hardware digital.

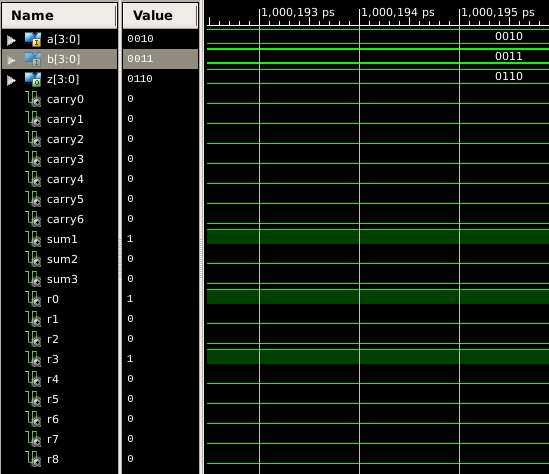


### 

### MULTIPLICADOR

A simulação de um multiplicador foi realizada com os números binários "0010" (2 em decimal) e "0011" (3 em decimal). O resultado esperado da multiplicação desses dois números é "0110" (6 em decimal).

Durante a simulação, a saída do multiplicador foi confirmada como sendo "0110", indicando que a multiplicação binária foi realizada corretamente. Isso demonstra a precisão do multiplicador, que executa a operação de multiplicação utilizando a técnica de multiplicação binária, onde cada bit do multiplicador é multiplicado pelo multiplicando, e os resultados são somados para obter o produto final.



# Conclusão

Neste trabalho, foi feito o desenvolvimento de uma ULA (Unidade Lógica-Aritmética) com o objetivo de projetar e implementar uma unidade capaz de realizar as principais operações lógicas e aritméticas em módulos vhdl, aplicando os conhecimentos teóricos adquiridos na disciplina.

Utilizando o software fornecido, modelamos e simulamos cada operador da ULA. Implementamos operações lógicas como AND, OR, NOT, e aritméticas como SOMA, SHIFTER, SUBTRAÇÃO, MULTIPLICAÇÃO e COMPLEMENTO DE 2. Também desenvolvemos a interface de usuário utilizando os botões e LEDs da placa. Para atender aos requisitos do projeto, implementamos uma máquina de estados.

Ao longo do processo, enfrentamos diversos desafios e tomamos decisões cruciais, especialmente em relação ao operador aritmético multiplicador e à máquina de estados. A máquina de estados, em particular, apresentou uma complexidade maior do que o esperado. Assim, criamos sinais auxiliares para colaborar com o seu desenvolvimento, o que contribuiu para a solução dos erros encontrados.

Concluímos que nossa ULA é capaz de executar as operações especificadas de maneira precisa e eficiente usando a implementação em FPGA. Entretanto, identificamos áreas que podem ser melhoradas, especialmente na otimização da máquina de estados.

# Agradecimentos

Por fim, gostaríamos de agradecer a todos que dedicaram seu tempo para ler este relatório. Apreciamos o interesse e a atenção dispensados ao nosso trabalho.

Queremos agradecer, também, ao professor João Batista por compartilhar seu vasto conhecimento e pela orientação durante todo o curso. As aulas foram fundamentais para a realização deste projeto, proporcionando a base prática necessária para o desenvolvimento da ULA. Sem seu suporte e ensinamentos, este trabalho não teria sido possível.