Universidade Federal do Rio de Janeiro

Universidade Federal
do Rio de Janeiro
Escola Politécnica

Unidade Lógico Aritmética

Relatório 1

Alunos	Gabriel Henrique Braga Lisboa
	Martina Marques Jardim
Professor	João Baptista de Oliveira e Souza Filho
Turma	EL1
Horário	Sex 13:00-15:00

Rio de Janeiro, 16 de junho de 2023

Conteúdo

1 Introdução **1**

2 Descrição da Implementação **1**
2.1 Módulo maq_estados 1
2.2 Módulo divisor_freq 5
2.3 Unidade Lógico Aritmética 6
2.3.1 Módulo MUX 6
2.3.2 AND 7
2.3.3 OR 7
2.3.4 NOT 7
2.3.5 XOR 7
2.3.6 Somador 7
2.3.7 Subtrator 8
2.3.8 Multiplicador 9

3 Resultados **10**
3.1 AND 11
3.2 OR 11
3.3 NOT 11
3.4 XOR 12
3.5 Soma com carry out 12
3.6 Subtração 12
3.7 Multiplicação 13
3.8 Teste do dispositivo na placa 14

4 Conclusões **15**

5 Apêndice **16**

1 Introdução

O presente relatório apresenta o desenvolvimento de uma Unidade Lógico-Aritmética em FPGA, em uma placa Xilinx Spartan. Foi utilizada a linguagem de programação VHDL, a qual é uma linguagem não sequencial criada especificamente para desenvolver circuitos integrados.

A unidade lógico-aritmética consiste em um circuito integrado o qual permite que o usuário, através das entradas disponíveis na placa, insira dois vetores binários diferentes e possa escolher uma operação lógica ou aritmética a ser feita com os vetores inserido, bem como ver os resultados obtidos. Os vetores inseridos pelo usuário e os resultados são representados visualmente por LEDs disponíveis na placa.

Em particular, a unidade lógico-aritmética implementada permite que o usuário insira, primeiramente, o primeiro número binário, seguido do segundo vetor binário, seguido da operação escolhida. A indicação de qual vetor deve ser inserido é dada pelos LEDs à esquerda da sequência de LEDs da placa. Após os vetores serem inseridos, os LEDs à direita exibem os vetores inseridos pelo usuário, seguidos dos resultados das operações feitas com os vetores, com o primeiro resultado mostrado sendo aquele referente à operação escolhida. Cada informação é exibida pelos LEDs durante dois segundos.

Nesse sentido, a implementação da unidade lógico-aritmética exige a presença de uma máquina de estados, a fim de exibir ciclicamente as informações. Além disso, também se fez necessário usar um divisor de frequência, para garantir a exibição dos dados durante dois segundos, e realizar diferentes operações lógicas e aritméticas com números binários de forma eficiente. O desenvolvimento das estruturas supracitadas é explicitado nas seções a seguir.

2 Descrição da Implementação

Todas as figuras referenciadas nesta seção se encontram no Apêndice deste relatório (seção 5 - página 16).

2.1 Módulo maq_estados

O módulo maq_estados é o vetor principal da implementação, contendo a máquina de estados, e conectando-a aos demais módulos, sendo esses os

relacionados as operações lógicas e aritméticas e ao divisor de frequência.

O módulo maq_estados recebe como entrada um clock de 50 MHz, um vetor de quatro posições (vetor), do tipo STD_LOGIC_VECTOR, três entradas de botões (botaoA, botaoB, botaoS) e um botão de reset (reset), do tipo STD_LOGIC. A entrada que aceita o clock será repassada para o divisor de frequência. A entrada vetor receberá os três vetores binários que o usuário irá inserir, sendo um desses o vetor que selecionará a operação a ser feita primeiramente, e os outros dois os vetores binários com os quais serão feitas as operações. Destaca-se que o vetor referente a operação possui quatro elementos, porém o bit mais significativo tem que ser obrigatoriamente igual a 0, ao passo que os outros vetores possuem quatro elementos livres em relação ao valor. A entrada botaoA registra o primeiro vetor inserido, da mesma forma que botaoB registra o segundo vetor inserido e botaoS registra a operação escolhida. O botão de reset é usado para fazer a máquina de estados retornar ao estado inicial.

Este módulo possui quatro saídas do tipo STD_LOGIC (led_A, led_B, led_S, Cout), usadas para acender os LEDs à esquerda, em que as três primeiras sinalizam qual é o vetor evidenciado pelos LEDs à direita, e a última serve como a flag de carry out/overflow para as operações aritméticas.
Por sua vez, o vetor evidenciado pelos LEDs à direita é representado pela saída display_leds, do tipo STD_LOGIC_VECTOR. Tais entradas e saídas estão representadas na imagem Fig. 1 no Apêndice. Destaca-se que a entrada Cin, apesar de presente, não foi utilizada.

Este módulo possui dois componentes, MUX e divisor_freq. Sucintamente, o módulo MUX é responsável por fazer as operações lógicas e aritméticas com os vetores de quatro bits inseridos. O módulo divisor_freq é um divisor de frequência, o qual recebe o clock de frequência 50 MHz e retorna um clock com período de dois segundos. O funcionamento particular de cada um desses módulos será explicitado nas seções adiante.

A máquina de estados possui 24 estados. Tal solução foi vista como a ideal pois, em razão de a linguagem VHDL ser paralela, torna-se inviável criar diferentes estados de forma sequencial, sendo necessário criar um estado para cada situação que deve ser observada nos LEDs.

Primeiramente, é definido um novo tipo de variável, o tipo estado, que será usado para definir os diferentes estados. Definir este novo tipo é necessário

pois os estados são sinais que não recebem valores numéricos, logo, não se adequam aos tipos previamente definidos pela linguagem VHDL. Existem 24 possibilidades de dados para um sinal deste novo tipo, logo, existem 24 estados possíveis.

Os sinais utilizados para os estados na máquina são estadoAtual e estadoAux, os quais são do tipo estado e tem o valor inicial de E0, o qual é considerado o estado inicial da máquina. Em seguida, é definido o sinal clk_2seg, do tipo STD_LOGIC, o qual atua como o clock da máquina de estados.

Além disso, são definidos os dois vetores que serão utilizados nas operações, bem como o vetor de seleção das operações como sendo do tipo STD_LOGIC_VECTOR. Por fim, são definidos oito vetores diferentes para o resultado de cada operação e um vetor de 7 posições para representar o carry-out de cada operação. Destaca-se que somente duas operações efetivamente possuem carry-out, mas escolheu-se essa estrutura vetorial para que se evitasse criar muitas variáveis

O primeiro procedimento realizado é ligar o divisor de frequência à máquina e obter um clock com o período de dois segundos. Isso é feito através de um port map com o componente divisor_freq, o qual recebe como entrada o clock de 50 MHz e retorna o clock com a frequência desejada, sendo esse aplicado em todo o restante da máquina. Logo depois, são definidos os resultados de todas as operações feitas pela ULA, através do componente MUX. Tais resultados serão consultados pela máquina de estados. Ver a imagem Fig.3 em anexo.

Após as definições acima, realiza-se a aplicação do módulo divisor_freq na máquina de estados. A cada pulso do clock resultante do divisor de frequência, o sinal estadoAtual recebe o valor armazenado pelo sinal estadoAux. Este processo realiza, em essência, uma mudança de estados. Ver a imagem Fig.3 em anexo.

O próximo processo define o que é feito em cada estado. Nesse sentido, ele recebe os vetores inseridos pelo usuário, as entradas de cada botão, as variáveis estadoAtual e estadoAux, os resultados de cada operação feita pela ULA, o vetor de carry-out e a seleção da operação.

O processo inicia-se verificando se é necessário voltar ao estado inicial E0, observando se a entrada reset está acionada. Em caso afirmativo, a variável estadoAux é definida como sendo E0 e os LEDs da direita da placa são apagados. Tais mudanças são implementadas no próximo pulso de clock. Em caso negativo, a máquina de estados inicia a coleta de dados.

Através da aplicação de uma estrutura case...when, verifica-se o valor da variável EstadoAtual, ou seja, observa-se em qual estado a máquina está. Os três primeiros estados coletam os dados necessários para realizar as operações. O primeiro estado (E0) coleta o primeiro vetor inserido e acende o primeiro LED à esquerda. Quando este LED estiver aceso, o vetor sendo coletado ou mostrado é o primeiro inserido. O vetor coletado é atribuído ao sinal A. As condicionais neste estado garantem que o segundo estado (E1) só é acessado após o primeiro botão ser apertado.

O segundo estado (E1) recolhe o segundo vetor de bits. Tem um funcionamento similar ao estado anterior (E0), acendendo o segundo LED à esquerda e passando ao estado seguinte somente se o segundo botão for apertado. O segundo vetor é atribuído ao sinal B. Essa dinâmica é vista em Fig. 4.

O terceiro estado (E2) recebe a operação escolhida pelo usuário. Neste estado, o terceiro LED à esquerda é aceso, indicando tratar-se do vetor seleção de operação. Este LED é aceso quando a operação selecionada será memorizada ou quando o resultado de uma operação é exibido. Após o botão referente ao input de operação ser apertado e a informação inserida ser armazenada na variável selecao, essa é analisada em uma série de condicionais, as quais verificam seu valor e assim, determinam qual é o próximo estado a ser acessado pela máquina.

Após o estadoAux assumir o valor do próximo estado a ser acessado, inicia-se uma dinâmica cíclica. O primeiro estado acessado nessa dinâmica simplesmente modifica os LEDs da placa. O primeiro vetor inserido é mostrado novamente nos quatro LEDs à direita e o primeiro LED à esquerda é aceso. O estadoAux passa a ser o estado seguinte. Após dois segundos, o estadoAtual assume o valor armazenado em estadoAux e o segundo vetor inserido é mostrado nos LEDs à direita e o segundo LED à esquerda é aceso. Novamente, estadoAux assume o valor do estado seguinte. Após dois segundos novamente, o estadoAtual assume o valor armazenado em estadoAux e o resultado da operação escolhida pelo usuário é evidenciado nos LEDs à direita e o terceiro LED à esquerda é aceso. Essa dinâmica é mostrada na Fig.6.

Após o resultado da operação escolhida ser mostrado, a máquina de estado mostra novamente o primeiro vetor inserido, em seguida o segundo vetor inserido e por fim o resultado da operação seguinte. Esta lógica será repetida infinitamente, de modo a mostrar os resultados de todas as operações da ULA, até o usuário apertar o botão de reset.

Destacam-se as operações de soma e subtração. Essas operações podem acender um LED além dos quatro usados para evidenciar os resultados. Esse LED, o qual fica a esquerda dos quatro usados normalmente, mostra se ocorre carry-out na operação de soma e overflow na operação de subtração.

A dinâmica cíclica entre os estados é evidenciada nas figuras Fig. 4 a Fig. 11 em Anexo.

2.2 Módulo divisor_freq

Este módulo tem como objetivo principal criar um clock com período de dois segundos a partir do clock padrão de frequência 50 MHz da placa utilizada. Conforme foi visto na seção anterior, este é o clock utilizado pela máquina de estados para realizar a mudança de estados e a exibição dos vetores. Seu código de implementação está na imagem Fig. 12 em Anexo.

Esse módulo recebe como entrada o clock de 50 MHz. O sinal de reset é utilizado apenas nas simulações e portanto está comentado no código. A única saída desse módulo é o novo clock, do tipo STD_LOGIC, uma vez que ele só pode assumir os valores de 0 ou 1. No início da arquitetura, são definidos dois sinais: temporal, que recolherá o valor do novo clock, e contador, que contará os pulsos do clock recebido como argumento. O sinal contador assume valores de 0 a 49999999, sendo o valor máximo igual a quantidade de pulsos (oscilações) em um segundo.

Após a definição dessas variáveis, um novo processo é iniciado. Inicialmente, verifica-se se o clock de 50 MHz está em rising edge, ou seja, se está prestes a passar do valor 0 ao valor 1. Na simulação, há antes uma estrutura condicional que é ativada caso o reset esteja em nível '1', para que as variáveis "*temporal*" e "*contador*" sejam zeradas.

Se o clock estiver na situação descrita anteriormente, realiza-se duas condicionais. Na primeira, se o contador estiver no valor igual a 49999999, o valor de temporal é invertido. Isso garante que o semi-período do novo clock

é um segundo, ou seja, que cada temporal fica um segundo em zero e um segundo em um. Além disso, nesta condição, o contador é redefinido para zero, a fim de reiniciar a contagem de um segundo. A segunda condicional é satisfeita se o contador tiver um valor diferente de 49999999. Isso significa que ainda não se passou um segundo. Assim, o contador é acrescido de um e temporal não é modificado.

Após essas verificações, o processo se encerra e o sinal de saída é redefinido para o valor armazenado em temporal.

2.3 Unidade Lógico Aritmética

2.3.1 Módulo MUX

Neste módulo, ocorre a seleção das possíveis operações lógicas ou aritméticas designadas pelo vetor de seleção ("*selecao*"), semelhante ao funcionamento de um multiplexador. O vetor de seleção possui 3 bits, portanto pode assumir 8 valores distintos, cada um correspondendo à uma operação diferente. Porém, como a ULA implementada possui apenas 7 operações, o valor "111" não será utilizado.

- 000: AND
- 001: OR
- 010: NOT
- 011: XOR
- 100: SOMA
- 101: SUBTRAÇÃO
- 110: MULTIPLICAÇÃO

Esse módulo recebe do Vetor_Fixo as entradas "*selecao*", A, B, Cin e tem como saídas o vetor "*saida*" e Cout (Figura 17)

O módulo MUX chama os módulos responsáveis por cada operação da ULA com os valores de A e B passados pelo módulo Vetor_Fixo e armazena as saídas nos sinais declarados (Figuras 19 e 20). Os sinais d1, d2, d3, d4, d5, d7 e d9 são vetores de 4 bits e armazenam os resultados das operações. Os sinais d6, d8 e d10 são do tipo STD_LOGIC e armazenam as flags de carry das operações aritméticas.

Todos esses sinais se encontram presentes na lista de sensibilidade do processo P1 (Figura 21) composto de uma estrutura condicional que irá selecionar a saída do módulo MUX de acordo com o valor do vetor "*selecao*". Para cada valor de "*selecao*", há um if/elsif que será acionado, relacionando os resultados e flags das operacoes com as saídas de MUX.

2.3.2 AND

Em razão de a linguagem VHDL ser própria para a descrição de hardware, ele possui uma sintaxe prórpia para operações lógicas. Nesse sentido, a implementação da operação lógica AND é bastante simples, consistindo apenas da execução do comando AND. Ver Fig. 13.

2.3.3 OR

De maneira análoga à operação AND, a implementação da operação OR também é feita utilizando-se da execução do comando OR em VHDL. Ver Fig. 14.

2.3.4 NOT

Novamente, esta implementação exige somente um comando próprio da linguagem VHDL, que inverte os vetores de entrada. Por ser uma operação feita somente com um vetor, escolheu-se que seria feito com o primeiro vetor inserido pelo usuário, o vetor A. Ver Fig. 15.

2.3.5 XOR

Também usa-se somente um comando de VHDL que realiza a operação XOR entre os vetores. Ver Fig. 16.

2.3.6 Somador

Os módulos BLOCO_SOMADOR e SOMA_ULA constituem o somador da ULA. Como a soma é feita entre dois vetores de 4 bits, o código implementado segue a lógica da concatenação de 4 somadores de 1 bit.

O módulo SOMA_ULA (Figura 22) funciona como um somador de 1 bit, com as entradas A, B e Cin. A e B são os bits a serem somados, e Cin atua como o carry in do somador. A saída "*soma*" corresponde ao resultado de $A + B$, e Cout ao flag de carry out do somador. Todas as entradas e saídas desse módulo são do tipo STD_LOGIC.

O módulo BLOCO_SOMADOR (figura 23) recebe como entrada dois vetores de 4 bits A e B e Cin (carry in). As saídas são o vetor de 4 bits "*soma*" e a flag de carry out, Cout. Na implementação desse módulo, SOMA_ULA é chamado 4 vezes, cada uma delas representado um somador de 1 bit concatenados entre si. Foi criado um sinal "*c*" do tipo STD_LOGIC_VECTOR com 4 bits. Cada bit desse vetor funciona ao mesmo tempo como carry out de uma instância da SOMA_ULA e carry in da próxima instância, propagando o carry por todo o bloco somador. O resultado de cada chamada do módulo SOMA_ULA é associado a um bit do resultado final da soma de 4 bits, armazenada pelo vetor "*soma*" do módulo BLOCO_SOMADOR.

2.3.7 Subtrator

O módulo SUBTRATOR é responsável pela operação de subtração da ULA. Neste caso, a operação realizada é $B - A$, e não $A - B$. Para realizar a subtração, o vetor A é convertido para o sistema complemento de 2, e depois é realizada uma soma entre B e o complemento de 2 de A.

Como mostrado na figura 24, esse módulo possui duas entradas A e B do tipo STD_LOGIC_VECTOR, com 4 bits cada. A saída "*resultado*" é um vetor de 4 bits que corresponde ao resultado da operação $B - A$ e a saída "$overflow$" corresponde ao flag que indica se houve overflow ou não, já que estamos utilizando o sistema de complemento de 2 para realizar essa operação. Foram criados 5 sinais para a implementação deste módulo:

- iA : do tipo STD_LOGIC_VECTOR, representa o inverso do vetor A.
- C2A: do tipo STD_LOGIC_VECTOR, representa o complemento de 2 do vetor A.
- saida: do tipo STD_LOGIC_VECTOR, representa o resultado final da operação $B - A$.
- carryOut1, carryOut2: do tipo STD_LOGIC, correspondem aos flags de carry out dos somadores.

Primeiramente, o SUBTRATOR converte o vetor A para complemento de 2. O módulo NOT_ULA é chamado para inverter A, que em seguida é somado com o vetor "0001" no BLOCO_SOMADOR. O vetor C2A, associado ao resultado do BLOCO_SOMADOR, corresponde ao complemento de 2 do vetor A. Em seguida, o módulo BLOCO_SOMADOR é chamado novamente, dessa vez para somar C2A com o vetor B. Como não há carry in neste caso,

ambos os valores de Cin das chamadas do BLOCO_SOMADOR estão em nível 0. Por último, para verificar a ocorrência de overflow, é iniciado um processo com uma estrutura condicional. Se o bit mais significativo de C2A e B forem iguais, e o resultado da soma entre eles for diferente desses dois bits, significa que houve overflow, então essa flag é colocado em nível 1. Caso contrário, overflow é igual a 0. Ao terminar o processo, o vetor "*saida*" é associado ao vetor "*resultado*" que irá indicar o resultado da subtração $B - A$.

2.3.8 Multiplicador

O multiplicador de vetores binários deste projeto apenas exibe os quatro bits menos significativos, tendo em vista que isso foi o exigido no roteiro do projeto. O multiplicador recebe os dois vetores inseridos pelo usuário e devolve somente os quatro bits menos significativos do resultado da multiplicação. A multiplicação deve ser feita de forma paralela, tendo em vista que VHDL não é uma linguagem sequencial. No início da arquitetura, são definidos quatro vetores de quatro bits, os quais representam os resultados intermediários da multiplicação (vetor1, vetor2, vetor3, vetor4). Além disso, também são definidos os vetores intermediários para a soma e o vetor do resultado final (soma1, soma2, resultado).

Neste módulo, são chamados dois componentes: BLOCO SOMADOR, o qual realiza a soma de vetores de 4 bits, e SOMA_ULA, o qual realiza somas bit a bit. Apenas o primeiro módulo é utilizado. Após isso, inicia-se um novo processo, onde são feitas diversas comparações e formações de novos vetores.

A primeira comparação é feita utilizando o segundo vetor inserido (chamado de B). Caso o bit menos significativo desse vetor seja igual a zero, o primeiro vetor intermediário, chamado vetor1, é um vetor nulo de quatro bits. Caso contrário, vetor1 será igual ao primeiro vetor de entrada. As próximas comparações definem os outros vetores intermediários da multiplicação de forma a obterem o deslocamento necessário para encontrar o resultado final correto.

Na comparação seguinte, observa-se segundo bit menos significativo do vetor B: se este for zero, vetor2 será nulo. Se for 1, vetor2 será construído a partir de uma concatenação entre zero e os três bits menos significativos de A, de forma que zero seja o bit menos significativo de vetor2. O zero na posição do bit menos significativo representa o deslocamento feito durante

a operação de multiplicação. Essa lógica é aplicada para a construção dos demais vetores, de forma que vetor3 tenha zeros como seus dois bits menos significativos e vetor4 tenha seus três últimos bits como sendo iguais a zero. Reforça-se que os vetores serão construídos assim se o bit de vetor B em análise seja igual a um.

Após a formação dos vetores intermediários da multiplicação, são feitas três somas vetoriais, utilizando o módulo BLOCO_SOMADOR. Na primeira soma, são somados vetor1 e vetor2 e na segunda soma, são somados vetor3 e vetor4. Por fim, a soma final é feita com os resultados das somas anteriores. O vetor "*resultado*" é equivalente ao resultado da soma final e é também o resultado da multiplicação, dado pelo vetor "$saidaMult$". As flags de carry out oriundas das chamadas do modulo BLOCO_SOMADOR não são importantes nesse caso, visto que estamos mostrando apenas os 4 bits menos significativos do resultado da multiplicação. Implementação visível nas imagens 26 a 28.

3 Resultados

Para exemplificação da prática, no final desta seção estão dispostas fotos da placa Xilinx Spartan 3AN durante a execução do projeto.

Para facilitar o funcionamento da VHDL Testbench, as seguintes simulações foram feitas utilizando apenas os módulos associados à ULA, sem a utilização da máquina de estados. O clock que aparece nessas simulações foi inserido apenas para possibilitar a simulação, não tendo nenhuma influência no resultado final, visto que a máquina de estados está excluída desta Testbench. Como decidimos não utilizar o carry in na versão final do projeto, para todas as simulações a entrada Cin estará em nível 0.

O vetor de 3 bits "*selecao*" indica qual a operacao que deve ser feita pela ULA. Como foram implementadas 7 operações, os valores utiliados para esse vetor variam de "000" até "110". O vetor de 4 bits "*saida*" indica o resultado da operação escolhida entre os operandos A e B.

Para todas as operações, os seguintes valores para os operandos A e B foram utilizados:

- $A = 1100$
- $B = 0101$

3.1 AND

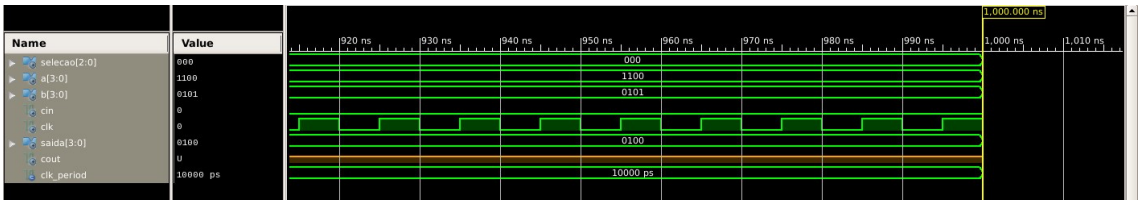


Figura 3.1: Diagrama de formas de onda representando a operação AND.

Para o vetor "*selecao*" em nível "000", a operação AND é realizada pela ULA. O resultado esperado para $A \cdot B$ com os valores utilizados é igual a "0100", que é compatível com o resultado encontrado para o vetor "*saida*" na simulação. Repare que a saída "*cout*", que corresponde ao flag de carry out da ULA está com valor indeterminado, pois não há flag de carry para nenhuma das operações lógicas.

3.2 OR

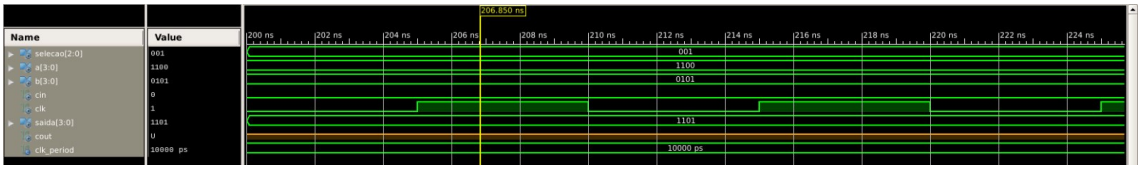


Figura 3.2: Diagrama de formas de onda representando a operação OR.

Modificando o valor do vetor "*selecao*" para "001", será realizada a operação OR. Como A or $B = 1101$, percebe-se que a simulação obteve o resultado esperado.

3.3 NOT

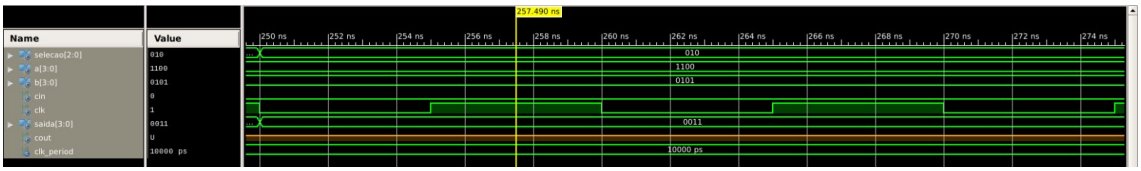


Figura 3.3: Diagrama de formas de onda representando a operação NOT.

O valor do vetor "*selecao*" agora é "010". Como a operação NOT necessita apenas de um operando, o vetor B é desconsiderado nesse caso. Para $notA$, esperamos o resultado "0011", que foi o valor obtido pela simulação acima.

3.4 XOR

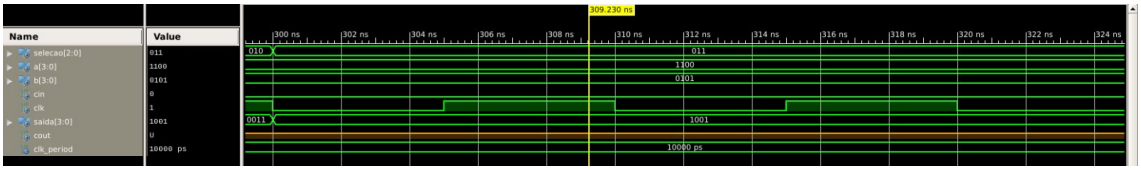


Figura 3.4: Diagrama de formas de onda representando a operação XOR.

Com o vetor "*selecao*" em nível "011", a ULA realizará a operação lógica XOR. Com esses valores para os operandos, $A \bigoplus B =$ "1001, valor compatível com o encontrado na simulação.

3.5 Soma com carry out

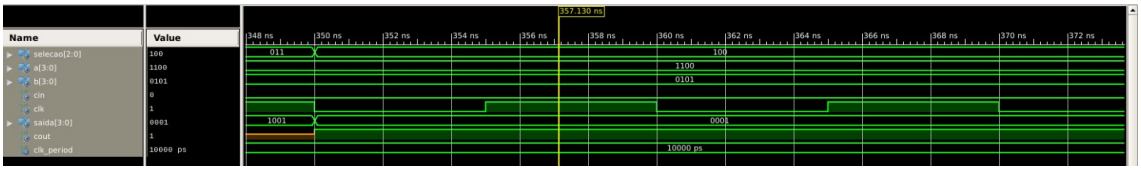


Figura 3.5: Diagrama de formas de onda representando a operação de soma.

Com o vetor "*selecao*" em nível "100", será realizada a primeira operação aritmética da ULA: a soma. O resultado esperado para $A+B$ é o mesmo que foi encontrado na simulação, 0001. Note que o flag de carry out, denominado "*cout*", passa de indeterminado para nível 1 no momento em que a operação de soma é feita pela ULA, pois agora a saída Cout é definida no módulo do somador.

3.6 Subtração

Fazemos a subtração modificando o valor do vetor "*selecao*" para "101". É importante ressaltar que nessa ULA foi implementada a operação de subtração $B - A$, e não $A - B$. Sendo assim, o vetor A é convertido para o

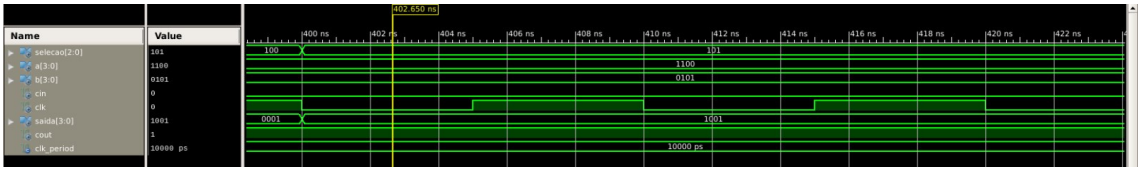


Figura 3.6: Diagrama de formas de onda representando a operação de subtração.

sistema complemento de 2, e uma soma entre esses B e C_2A é realizada, com resultado esperado de 1001, o mesmo resultado encontrado na simulação. No caso da subtração o flag de carry out indica se houve overflow na soma de complemento a 2 ou não.

3.7 Multiplicação

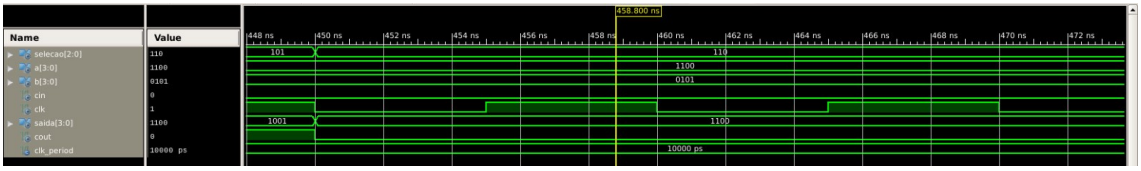


Figura 3.7: Diagrama de formas de onda representando a operação de multiplicação.

Por último, a multiplicação é realizada passando o vetor de "*selecao*" para nível "110". Como os operandos possuem 4 bits cada, o resultado da multiplicação entre esses dois vetores pode ter até 8 bits. Porém, para tornar possível a implementação desse módulo na placa FPGA com número limitado de LEDs, a ULA foi projetada para retornar apenas os 4 bits menos significativos no resultado desta operação. Portanto, o resultado esperado é igual ao encontrado ao simular a multiplicação na ULA, com o vetor "saida"sendo igual a "1100".

3.8 Teste do dispositivo na placa

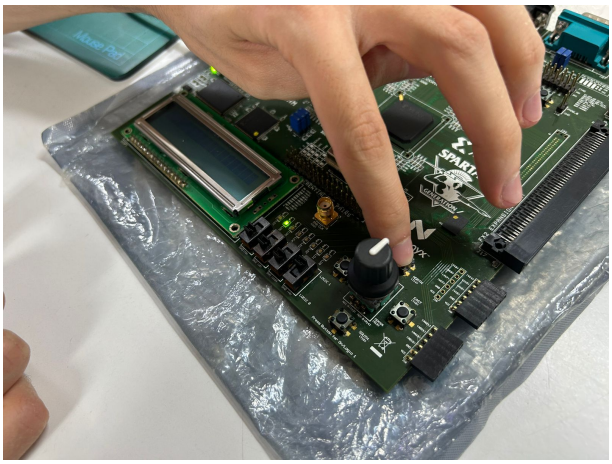


Figura 3.8: Momento em que o vetor B está sendo selecionado, estado indicado pelo led aceso, aguardando o usuário apertar o botão para que o valor de B seja armazenado.

Figura 3.9: Momento em que o vetor A está sendo mostrado pelos 4 leds mais à direita.

Figura 3.10: Momento em que o vetor B está sendo mostrado pelos 4 leds mais à direita.

Figura 3.11: Momento em que resultado de uma das operações entre A e B está sendo mostrada pelos 4 leds mais à direita.

4 Conclusões

O dispositivo foi testado em hardware e em simulação computacional, apresentando todos os resultados de acordo com as operações configuradas. A utilização de uma linguagem de descrição de máquina facilitou em grande parte a confecção dos módulos, permitindo que a Unidade Lógico Aritmética e a máquina de estados fossem projetadas, testadas e implementadas de forma eficiente. Além disso, permitiu que diferentes métodos de construção dos módulos fossem testados de forma a selecionar o que melhor se enquadrasse

nos requisitos do projeto e nas limitações de hardware.

5 Apêndice

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity maq_estados is
    Port ( clk_50MHz: in STD_LOGIC;
                        vetor : in  STD_LOGIC_VECTOR (3 downto 0);
                        botaoA: in STD_LOGIC;
                        botaoB: in STD_LOGIC;
                        botaoS: in STD_LOGIC;
                        reset: in STD_LOGIC;
                        Cin: in STD_LOGIC;
                        led_A: out STD_LOGIC;
                        led_B: out STD_LOGIC;
                        led_S: out STD_LOGIC;
                        display_leds : out  STD_LOGIC_VECTOR (3 downto 0);
                        Cout: out STD_LOGIC);
end maq_estados;
```

Figura 1: Entradas e saídas do módulo maq_estados

```
architecture Behavioral of maq_estados is

component MUX is

        Port ( selecao : in  STD_LOGIC_VECTOR (2 downto 0);
                           A : in  STD_LOGIC_VECTOR (3 downto 0);
                           B : in  STD_LOGIC_VECTOR (3 downto 0);
                           Cin: in STD_LOGIC;
                           saida : out  STD_LOGIC_VECTOR (3 downto 0);
                           Cout: out STD_LOGIC);

end component;

component divisor_freq is
        Port(
                clk_50MHz: in STD_LOGIC;
                clk_out: out STD_LOGIC
                 );

end component;
```

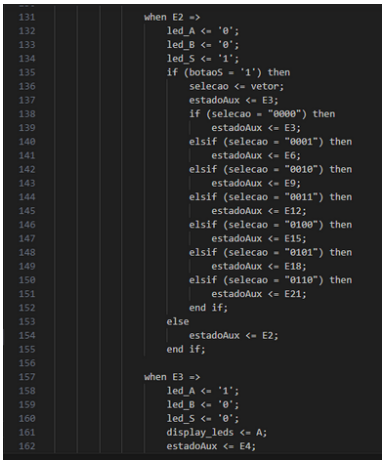
Figura 2: Componentes usados em maq_estados.

```
type estado is (E0, E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11, E12,
                                    E13, E14, E15, E16, E17, E18, E19, E20, E21, E22, E23);
signal estadoAtual, estadoAux: estado := E0;
signal clk_2seg: STD_LOGIC;
signal A, B, resultado, selecao: STD_LOGIC_VECTOR (3 downto 0);
signal Z0, Z1, Z2, Z3, Z4, Z5, Z6, Z7: STD_LOGIC_VECTOR (3 downto 0);
signal c: STD_LOGIC_VECTOR (6 downto 0);

begin

        dividir_clock: divisor_freq port map (clk_50MHz, clk_2seg);

        AND_ULA: MUX port map ("000", A, B, Cin, Z0, c(0));
        OR_ULA: MUX port map ("001", A, B, Cin, Z1, c(1));
        NOT_ULA: MUX port map ("010", A, B, Cin, Z2, c(2));
        XOR_ULA: MUX port map ("011", A, B, Cin, Z3, c(3));
        SOMA_ULA: MUX port map ("100", A, B, Cin, Z4, c(4));
        SUB_ULA: MUX port map ("101", A, B, Cin, Z5, c(5));
        MULT_ULA: MUX port map ("110", A, B, Cin, Z6, c(6));


        process (clk_2seg, estadoAux)
        begin
                if (rising_edge(clk_2seg)) then
                        estadoAtual <= estadoAux;
                end if;
        end process;
```

Figura 3: Sinais usados na máquina de estados e implementação do divisor de frequência.

```
98      process (vetor, botaoA, botaoB, botaoS, reset, estadoAtual, estadoAux, c,
99                  A, B, selecao, Z0, Z1, Z2, Z3, Z4, Z5, Z6)
100
101     begin
102
103         if (reset = '1') then
104             estadoAux <= E0;
105             display_leds <= "0000";
106
107         else
108             case estadoAtual is
109                 when E0 =>
110                     led_A <= '1';
111                     led_B <= '0';
112                     led_S <= '0';
113                     if (botaoA = '1') then
114                         A <= vetor;
115                         estadoAux <= E1;
116                     else
117                         estadoAux <= E0;
118                     end if;
119
120                 when E1 =>
121                     led_A <= '0';
122                     led_B <= '1';
123                     led_S <= '0';
124                     if (botaoB = '1') then
125                         B <= vetor;
126                         estadoAux <= E2;
127                     else
128                         estadoAux <= E1;
129                     end if;
```

Figura 4: Início da máquina de estados.



```
131             when E2 =>
132                 led_A <= '0';
133                 led_B <= '0';
134                 led_S <= '1';
135                 if (botaoS = '1') then
136                     selecao <= vetor;
137                     estadoAux <= E3;
138                     if (selecao = "0000") then
139                         estadoAux <= E3;
140                     elsif (selecao = "0001") then
141                         estadoAux <= E6;
142                     elsif (selecao = "0010") then
143                         estadoAux <= E9;
144                     elsif (selecao = "0011") then
145                         estadoAux <= E12;
146                     elsif (selecao = "0100") then
147                         estadoAux <= E15;
148                     elsif (selecao = "0101") then
149                         estadoAux <= E18;
150                     elsif (selecao = "0110") then
151                         estadoAux <= E21;
152                     end if;
153                 else
154                     estadoAux <= E2;
155                 end if;
156
157             when E3 =>
158                 led_A <= '1';
159                 led_B <= '0';
160                 led_S <= '0';
161                 display_leds <= A;
162                 estadoAux <= E4;
```

Figura 5: Terceiro e quarto estados.

```
when E3 =>
    led_A <= '1';
    led_B <= '0';
    led_S <= '0';
    display_leds <= A;
    estadoAux <= E4;


when E4 =>
    led_A <= '0';
    led_B <= '1';
    led_S <= '0';
    display_leds <= B;
    estadoAux <= E5;


when E5 =>
    led_A <= '0';
    led_B <= '0';
    led_S <= '1';
    display_leds <= Z0;
    Cout <= c(0);
    estadoAux <= E6;
```

Figura 6: Estados de exibição dos vetores de entrada (E3, E4) e exibição do resultado da operação AND (E5).

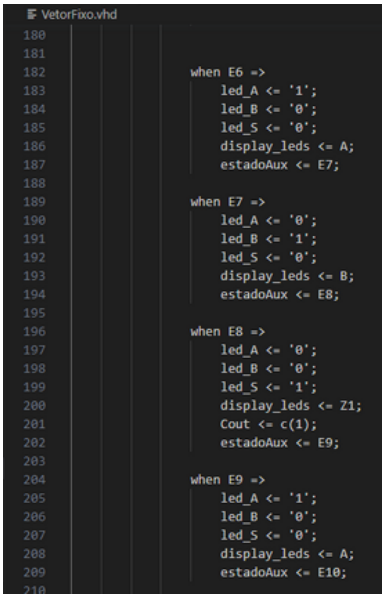


Figura 7: Estados de E6 a E9. E6 e E7 mostram os primeiro e segundo vetores inseridos, respectivamente. E8 mostra o resultado da operação OR. E9 mostra, novamente, o primeiro vetor inserido.

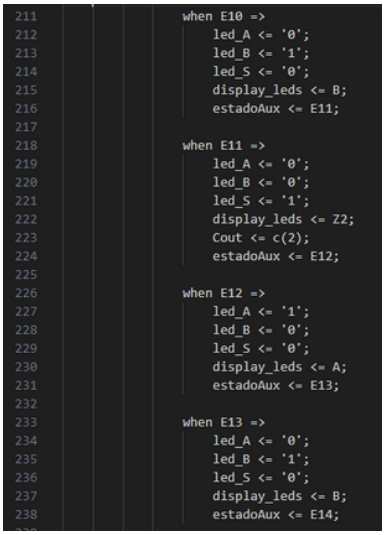
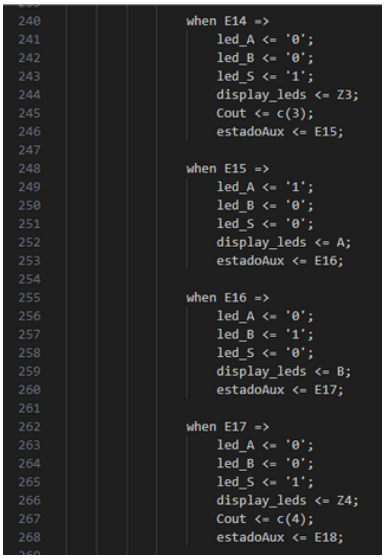
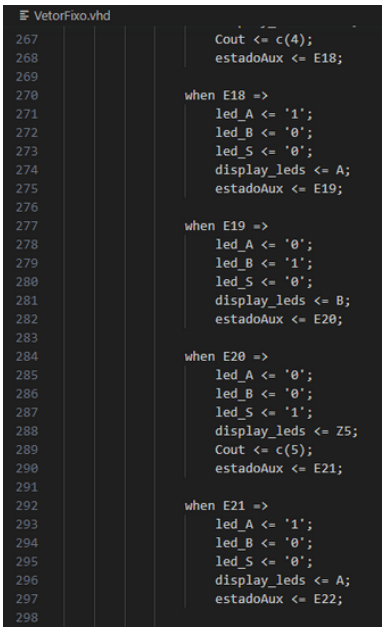


Figura 8: E10 e E13 mostram o segundo vetor inserido. E11 mostra o resultado da operação NOT, feita com o primeiro vetor inserido.



```
240                 when E14 =>
241                     led_A <= '0';
242                     led_B <= '0';
243                     led_S <= '1';
244                     display_leds <= Z3;
245                     Cout <= c(3);
246                     estadoAux <= E15;
247
248                 when E15 =>
249                     led_A <= '1';
250                     led_B <= '0';
251                     led_S <= '0';
252                     display_leds <= A;
253                     estadoAux <= E16;
254
255                 when E16 =>
256                     led_A <= '0';
257                     led_B <= '1';
258                     led_S <= '0';
259                     display_leds <= B;
260                     estadoAux <= E17;
261
262                 when E17 =>
263                     led_A <= '0';
264                     led_B <= '0';
265                     led_S <= '1';
266                     display_leds <= Z4;
267                     Cout <= c(4);
268                     estadoAux <= E18;
```

Figura 9: E14 mostra o resultado da operação XOR. E17 mostra o resultado da operação de soma. O estado E17 é um dos que permite a exibição de carry-out diferente de zero. E15 mostra o primeiro vetor inserido. E16 mostra o segundo vetor inserido.



```
VetorFixo.vhd
267                     Cout <= c(4);
268                     estadoAux <= E18;
269
270                 when E18 =>
271                     led_A <= '1';
272                     led_B <= '0';
273                     led_S <= '0';
274                     display_leds <= A;
275                     estadoAux <= E19;
276
277                 when E19 =>
278                     led_A <= '0';
279                     led_B <= '1';
280                     led_S <= '0';
281                     display_leds <= B;
282                     estadoAux <= E20;
283
284                 when E20 =>
285                     led_A <= '0';
286                     led_B <= '0';
287                     led_S <= '1';
288                     display_leds <= Z5;
289                     Cout <= c(5);
290                     estadoAux <= E21;
291
292                 when E21 =>
293                     led_A <= '1';
294                     led_B <= '0';
295                     led_S <= '0';
296                     display_leds <= A;
297                     estadoAux <= E22;
298
```

Figura 10: E18 e E21 mostram o primeiro vetor inserido. E19 mostra o segundo vetor inserido. E20 mostra o resultado da subtração.

```
299                     when E22 =>
300                         led_A <= '0';
301                         led_B <= '1';
302                         led_S <= '0';
303                         display_leds <= B;
304                         estadoAux <= E23;
305
306                     when E23 =>
307                         led_A <= '0';
308                         led_B <= '0';
309                         led_S <= '1';
310                         display_leds <= Z6;
311                         Cout <= c(6);
312                         estadoAux <= E3;
313
314                     when others =>
315                         null;
316
317                 end case;
318             end if;
319         end process;
320
321
322     end Behavioral;
323
324
```

Figura 11: E22 mostra o segundo vetor inserido. E23 mostra o resultado da multiplicação.

```
entity divisor_freq is
   port (
      clk_50MHz: in std_logic;
      --reset: in std_logic;
      clk_out: out std_logic
         );
end divisor_freq;

architecture Behavioral of divisor_freq is

   signal temporal: std_logic;
   signal contador: integer range 0 to 49999999 := 0;

begin
   dividir_clk: process (clk_50MHz) begin
      --if (reset = '1') then
         --temporal <= '0';
         --contador <= 0;
      if (rising_edge(clk_50MHz)) then
         if (contador = 49999999) then
            temporal <= not (temporal);
            contador <= 0;
         else
            contador <= contador + 1;
         end if;
      end if;
   end process;

   clk_out <= temporal;

end Behavioral;
```

Figura 12: Implementação e arquitetura do divisor de frequência.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity AND_ULA is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           C : out  STD_LOGIC_VECTOR (3 downto 0));
end AND_ULA;

architecture Behavioral of AND_ULA is

begin

C <= A and B;

end Behavioral;
```

Figura 13: Código fonte da operação AND.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity OR_ULA is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           D : out  STD_LOGIC_VECTOR (3 downto 0));
end OR_ULA;

architecture Behavioral of OR_ULA is

begin

D <= A or B;

end Behavioral;
```

Figura 14: Código fonte da operação OR.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity NOT_ULA is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           E : out  STD_LOGIC_VECTOR (3 downto 0));
end NOT_ULA;

Sarchitecture Behavioral of NOT_ULA is

begin

E <= not A;

end Behavioral;
```

Figura 15: Código fonte da operação NOT.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity XOR_ULA is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           F : out  STD_LOGIC_VECTOR (3 downto 0));
end XOR_ULA;

architecture Behavioral of XOR_ULA is

begin

F <= A xor B;

end Behavioral;
```

Figura 16: Código fonte da operação XOR.

```vhdl
entity MUX is

        Port ( selecao : in  STD_LOGIC_VECTOR (2 downto 0);
                         A : in  STD_LOGIC_VECTOR (3 downto 0);
                         B : in  STD_LOGIC_VECTOR (3 downto 0);
                         Cin: in STD_LOGIC;
                         saida : out  STD_LOGIC_VECTOR (3 downto 0);
                         Cout: out STD_LOGIC);

end MUX;
```

Figura 17: Entradas e saídas do módulo MUX.

```
architecture Behavioral of MUX is

component AND_ULA is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           C : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

component OR_ULA is
         Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           D : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

component NOT_ULA is
         Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           E : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

component XOR_ULA is
         Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           F : out  STD_LOGIC_VECTOR (3 downto 0));
end component;
```

Figura 18: Componentes usados em MUX.

```
component BLOCO_SOMADOR is
        Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
          B : in  STD_LOGIC_VECTOR (3 downto 0);
                     Cin : in  STD_LOGIC;
          soma : out  STD_LOGIC_VECTOR (3 downto 0);
          Cout : out  STD_LOGIC);
end component;

component SUBTRATOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           resultado : out  STD_LOGIC_VECTOR (3 downto 0);
           overflow : out  STD_LOGIC);
end component;

component MULTIPLICADOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           saidaMult : out  STD_LOGIC_VECTOR (3 downto 0);
                       Cout : out STD_LOGIC);
end component;


       signal d1, d2, d3, d4, d5, d7, d9: STD_LOGIC_VECTOR (3 downto 0);
       signal d6, d8, d10: STD_LOGIC;
```

Figura 19: Resto dos componentes usados em MUX e declaração dos sinais usados no módulo.

```
begin

   resposta1: AND_ULA port map (A, B, d1);

   resposta2: OR_ULA port map (A, B, d2);

   resposta3: NOT_ULA port map (A, d3);

   resposta4: XOR_ULA port map (A, B, d4);

   resposta5: BLOCO_SOMADOR port map (A, B, Cin, d5, d6);

   resposta6: SUBTRATOR port map (A, B, d7, d8);

   resposta7: MULTIPLICADOR port map (A, B, d9, d10);
```

Figura 20: O módulo MUX chama os módulos das operações da ULA e guarda os resultados das operações nos sinais declarados anteriormente.

```
  P1: process (selecao, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10)

  begin


     if (selecao = "000") then
        saida <= d1;

     elsif (selecao = "001") then
        saida <= d2;

     elsif (selecao = "010") then
        saida <= d3;

     elsif (selecao = "011") then
        saida <= d4;

     elsif (selecao = "100") then
        saida <= d5;
        Cout <= d6;

     elsif (selecao = "101") then
        saida <= d7;
        Cout <= d8;

     elsif (selecao = "110") then
        saida <= d9;
        Cout <= d10;

     end if;

  end process P1;

end Behavioral;
```

Figura 21: Processo criado no módulo MUX.

```
entity SOMA_ULA is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           soma : out  STD_LOGIC;
           Cout : out  STD_LOGIC);
end SOMA_ULA;

architecture Behavioral of SOMA_ULA is

begin

soma <= (A xor B) xor Cin;
Cout <= (A and B) or (Cin and (A xor B));

end Behavioral;
```

Figura 22: Código fonte do módulo SOMA_ULA, o somador de 1 bit.

```
entity BLOCO_SOMADOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Cin : in  STD_LOGIC;
           soma : out  STD_LOGIC_VECTOR (3 downto 0);
           Cout : out  STD_LOGIC);
end BLOCO_SOMADOR;

architecture Behavioral of BLOCO_SOMADOR is

signal c: STD_LOGIC_VECTOR (3 downto 0);

component SOMA_ULA
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           soma : out  STD_LOGIC;
           Cout : out  STD_LOGIC);
end component;

begin

S0: SOMA_ULA port map(A(0), B(0), Cin, soma(0), c(0));
S1: SOMA_ULA port map(A(1), B(1), c(0), soma(1), c(1));
S2: SOMA_ULA port map(A(2), B(2), c(1), soma(2), c(2));
S3: SOMA_ULA port map(A(3), B(3), c(2), soma(3), Cout);

end Behavioral;
```

Figura 23: Código fonte do módulo BLOCO_SOMADOR, o somador de 4 bits.

```
entity SUBTRATOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           resultado : out  STD_LOGIC_VECTOR (3 downto 0);
           overflow : out  STD_LOGIC);
end SUBTRATOR;

architecture Behavioral of SUBTRATOR is

component NOT_ULA is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           E : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

component BLOCO_SOMADOR is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           Cin : in  STD_LOGIC;
           soma : out  STD_LOGIC_VECTOR (3 downto 0);
           Cout : out  STD_LOGIC);
end component;

   signal iA, C2A, saida : STD_LOGIC_VECTOR (3 downto 0);
   signal carryOut1, carryOut2 : STD_LOGIC;
```

Figura 24: Declaração das entradas e saídas, componentes e sinais utilizados no módulo SUBTRATOR.

```
begin

   inverso_A: NOT_ULA port map (A, iA);

   complemento2_A: BLOCO_SOMADOR port map (iA, "0001", '0', C2A, carryOut1);

   resposta: BLOCO_SOMADOR port map (C2A, B, '0', saida, carryOut2);

   P2: process (C2A, B, saida)

   begin

   if (C2A(3) = B(3) and saida(3) /= C2A(3)) then
      overflow <= '1';

   else
      overflow <= '0';

   end if;

   end process;

   resultado <= saida;

end Behavioral;
```

Figura 25: Lógica do módulo SUBTRATOR.

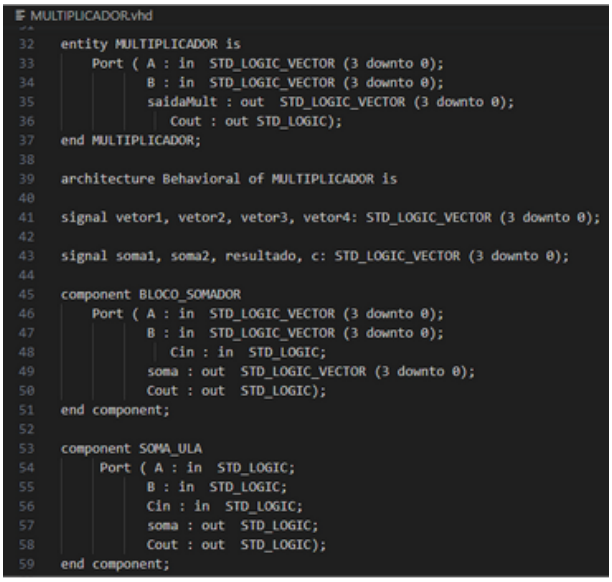


Figura 26: Declaração das entradas, saídas, componentes e sinais do Multiplicador.

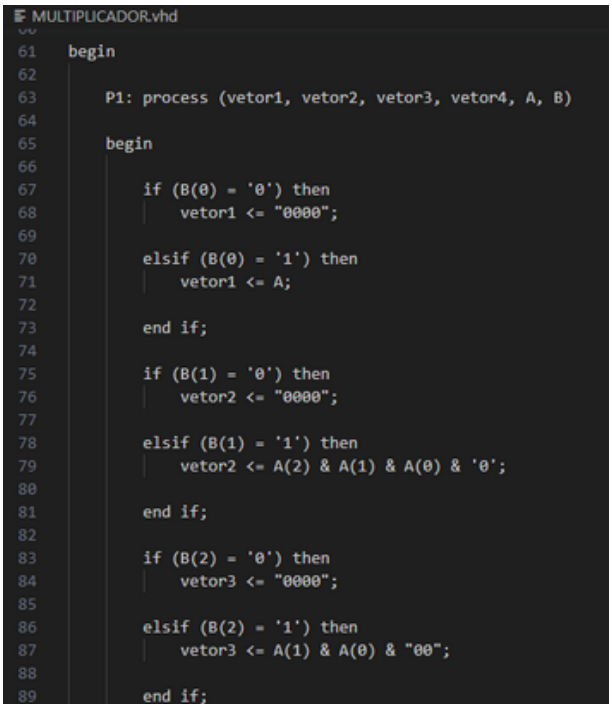
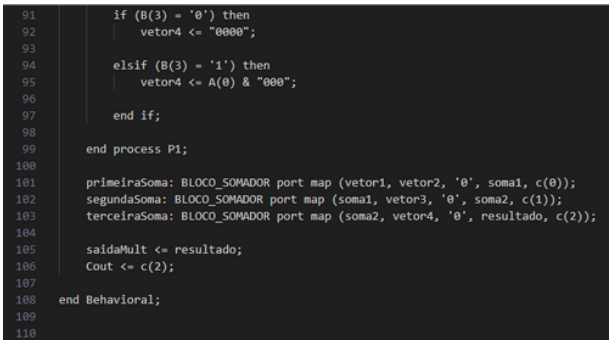


Figura 27: Processo do código do Multiplicador.



```
91          if (B(3) = '0') then
92              vetor4 <= "0000";
93
94          elsif (B(3) = '1') then
95              vetor4 <= A(0) & "000";
96
97          end if;
98
99      end process P1;
100
101     primeiraSoma: BLOCO_SOMADOR port map (vetor1, vetor2, '0', soma1, c(0));
102     segundaSoma: BLOCO_SOMADOR port map (soma1, vetor3, '0', soma2, c(1));
103     terceiraSoma: BLOCO_SOMADOR port map (soma2, vetor4, '0', resultado, c(2));
104
105     saidaMult <= resultado;
106     Cout <= c(2);
107
108 end Behavioral;
109
110
```

Figura 28: Final do código do Multiplicador.