UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA - CT
CIRCUITOS DIGITAIS

PROJETO DE UMA MÁQUINA DE SNACKS

DCA0212.1 - Projeto 2

Igor Michael Araujo de Macedo
Isaac de Lyra Júnior
Pedro Henrique de Freitas Silva

Natal, 16 de fevereiro de 2022

1. INTRODUÇÃO

Os *snacks* são as melhores opções para os lanches rápidos durante uma reunião ou para aqueles dias de correria. Eles existem nas mais diversas opções e, na maioria dos casos, já estão prontos para o consumo ou podem ser facilmente preparados.

Para facilitar ainda mais a vida das pessoas, normalmente esses lanches são encontrados em máquinas de *snack*, onde empresas contratam um serviço que disponibiliza essas máquinas de alimentos como café expresso, chocolates, salgadinhos, latas de refrigerante, biscoitos, entre outros tipos de insumos. Elas têm como principal vantagem a facilidade e simplicidade para a compra do produto, precisando apenas de uma simples operação de seleção do produto, uma operação de pagamento e o alimento já está pronto para ser retirado por uma abertura. Toda a operação é feita de forma automatizada, sem necessidade de pessoas para auxiliar na operação.

Figura 1 - Máquina de *snacks*.

Fonte: Página de vendas Mercearia Pronta.

Sendo assim, conhecendo o que são as máquinas de *snacks* e a fim de colocar em prática os conceitos estudados durante a disciplina de Circuitos Digitais, este trabalho tem como objetivo projetar e implementar um sistema simplificado que simula o funcionamento de uma máquina desse tipo. A aplicação será desenvolvida utilizando o método de projeto RTL e, a nível de implementação, o sistema será descrito utilizando a linguagem de descrição de *hardware* VHDL. Por fim, serão feitas simulações utilizando o *software* ModelSim para verificar se o sistema está funcionando como o projetado.

2. DESENVOLVIMENTO

A ideia do projeto é desenvolver um circuito que simula o funcionamento do processador de uma máquina de *snacks*. Como o principal objetivo é colocar em prática os conceitos estudados na disciplina, o funcionamento será mais simplificado, mas pode ser expandido para maior número de produtos e tipos de valores aceitos, por exemplo, tal como ser melhorado futuramente.

O funcionamento da máquina se dará da seguinte maneira: o cliente irá adicionar dinheiro e inserir o código do produto que deseja; a máquina será capaz de verificar se o dinheiro é suficiente para o produto selecionado ou não; caso seja suficiente, o produto será liberado e será calculado o troco necessário; caso o dinheiro não seja suficiente, o cliente terá a opção de adicionar mais dinheiro ou cancelar a operação; cancelando a operação ele receberá seu dinheiro de volta.

Como foi dito anteriormente, o desenvolvimento se dará de maneira mais simplificada, sendo assim, a máquina receberá apenas valores inteiros de dinheiro, que variam de 1 a 15, e existirão apenas quatro diferentes tipos de produtos.

Dessa forma, tendo todo o funcionamento em mente, o sistema possuirá uma entrada de dados VALUE de 4 bits, que indicará o valor, em dinheiro, a ser adicionado; uma entrada INSERTING de um 1 bit que indica que o valor inserido deve ser adicionado na máquina; uma entrada COD_PROD de 2 bits que representa o código do produto; uma entrada SELECT de um 1 bit que informa à máquina que o cliente deseja pegar o produto de código informado; e, por fim, uma entrada CANCEL de 1 bit para cancelar o processo e receber o dinheiro de volta. Como saída a máquina possui uma indicação IS_RELEASE de 1 bit que indica que o produto selecionado foi liberado; e uma saída REMAINING_MONEY de 4 bits que representa o dinheiro restante/troco.

2.1 PROJETO RTL

A metodologia utilizada no projeto deste trabalho foi o método de projeto RTL, o qual é composto por 4 passos (VAHID, 2008):

1. Obtenção de uma máquina de estados finitos (FSM) de alto nível;
2. Criação de um bloco operacional;
3. Conexão entre o bloco operacional e um bloco de controle
4. Obtenção da máquina de estados de baixo nível do bloco de controle

2.1.1 Máquina de estados de alto nível

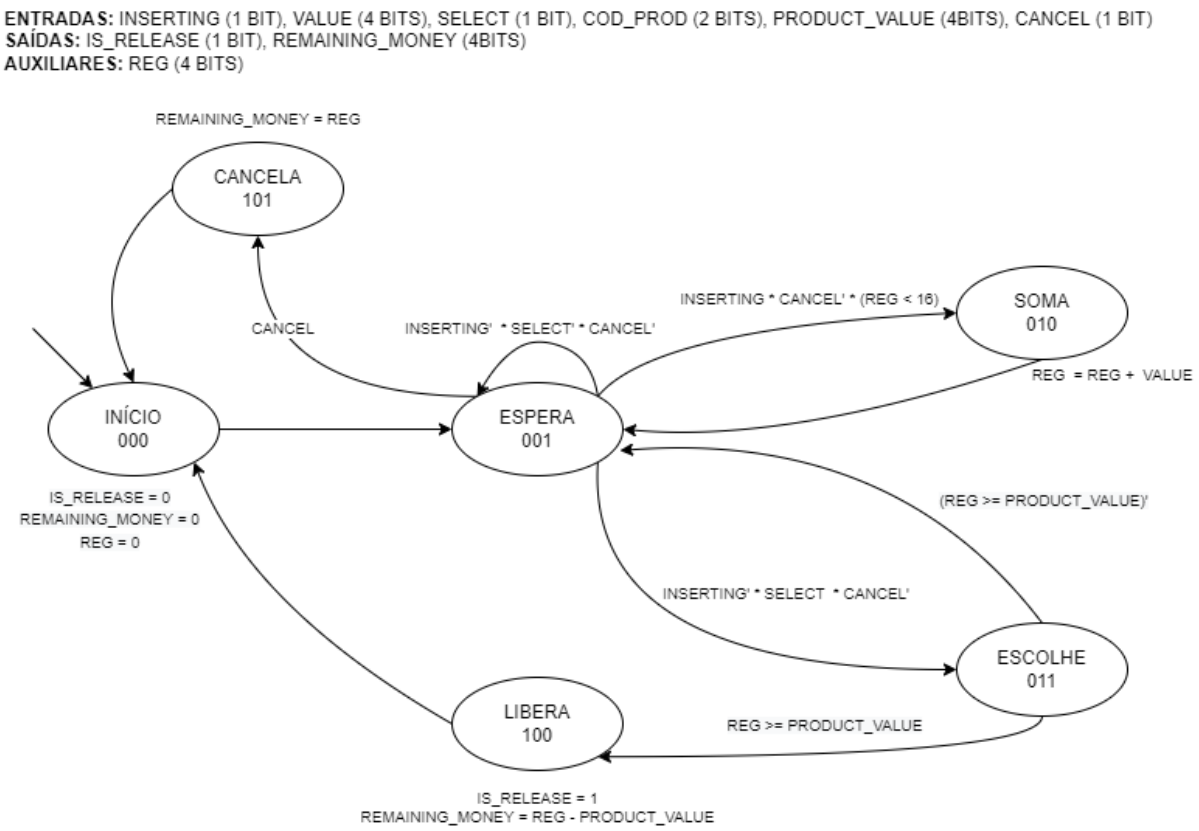
O primeiro passo consiste na captura da FSM de alto nível que deve descrever o comportamento desejado do bloco de controle. Ela é composta por 6 estados: INICIO, ESPERA, SOMA, ESCOLHE, LIBERA e CANCELA. O primeiro estado é o ponto de início, onde é responsável por zerar todos os registradores e saídas e, logo em seguida, ir para o próximo estado, onde é esperada qualquer ação do usuário, seja inserção de valor, seleção de produto ou cancelamento de operação.

A máquina está sempre recebendo os dados do código do produto e valor a ser inserido, o que irá efetivar a operação é o pressionamento dos botões INSERTING e SELECT. Ao pressionar INSERTING, a máquina vai para o estado de SOMA, acumula o dinheiro, e, no próximo ciclo de relógio, volta ao estado anterior; vale ressaltar que será acumulado dinheiro até um limite de 15, acima disso o dinheiro não é aceito.

Estando no estado de espera e pressionando o SELECT, a máquina vai para o estado ESCOLHE, onde será verificado se o valor inserido até o momento é suficiente para comprar o produto escolhido; caso seja, o próximo estado é o de LIBERA, onde é computado o valor do troco (valor acumulado na máquina menos valor do produto) e o produto é liberado (saída IS_RELEASE recebe nível lógico alto e a porta se abre). Caso o dinheiro não seja suficiente, a máquina volta para o estado de espera. Estando nessa situação, o cliente tem a opção de adicionar mais dinheiro ou cancelar toda a operação. Ao pressionar o CANCEL, a máquina vai para o estado de CANCELA, onde todo o dinheiro acumulado é devolvido por REMAINING_MONEY e, no próximo *clock*, volta para o estado inicial e o processo reinicia.

Toda a máquina de estados de alto nível pode ser vista na Figura 2, bem como entradas, saídas e registradores necessários.

Figura 2 - Máquina de estados de alto nível



Fonte: Autores

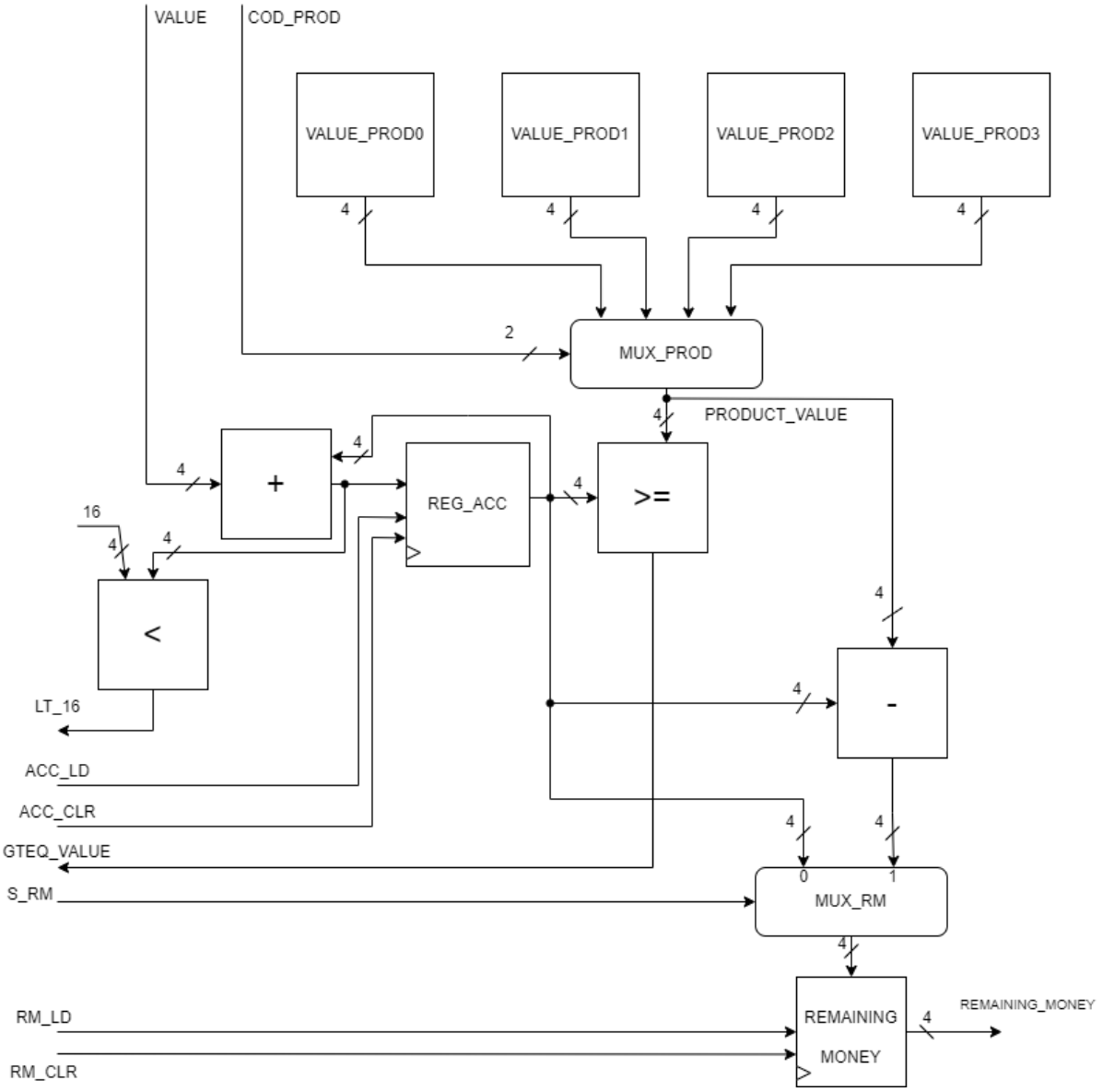
2.1.2 Bloco operacional

Partindo da máquina de estados de alto nível desenvolvido na seção 2.1, deve-se criar um bloco operacional responsável por lidar e realizar operações que envolvem dados. Ele pode ser visto na Figura 3 e é composto por:

- Quatro registradores internos que armazenam os valores dos quatro produtos da máquina;
- Um multiplexador 4x1, com barramentos de 4 bits, onde a chave seletora é a entrada COD_PROD (2 bits), inserida pelo usuário, e a saída é o valor do produto PRODUCT_VALUE;
- Um registrador de 4 bits onde será acumulado os valores inseridos pelo usuário;
- Um somador de 4 bits que será utilizado para o acumulador;
- Um comparador de magnitude que compara se o valor acumulado é menor que 16, uma vez que o valor limite é 15;
- Um comparador de magnitude que compara o valor acumulado com o valor do produto;

- Um subtrator de 4 bits que subtrai o valor acumulado pelo valor do produto, para calcular o troco;
- Por fim, o valor retornado ao usuário (REMAINING_MONEY) pode ser o troco ou o valor acumulado, sendo assim, é utilizado um registrador de 4 bits para isso e a entrada dele é a saída de um multiplexador 2x1, com barramento de 4 bits, para selecionar um desses dois possíveis retornos citados.

Figura 3 - Bloco operacional.

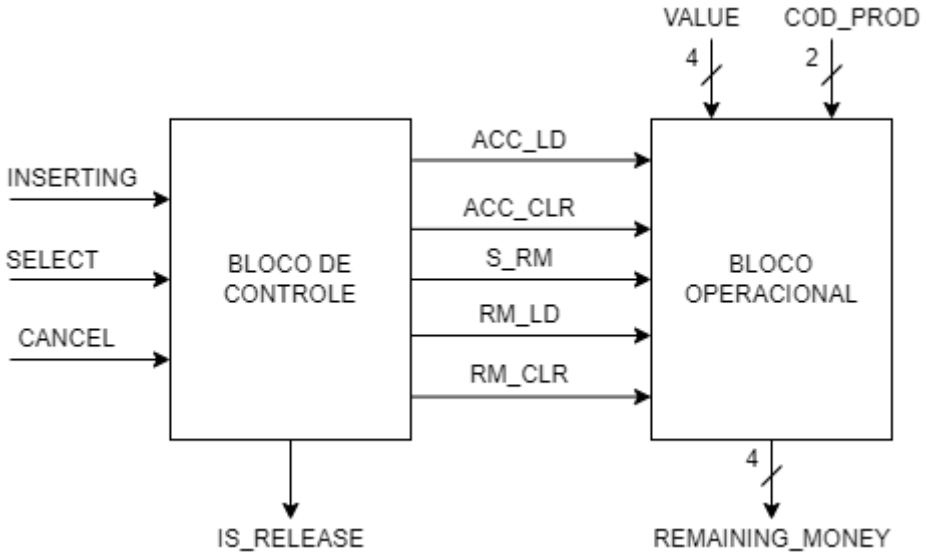


Fonte: Autores.

2.1.3 Bloco operacional e bloco de controle

Para o próximo passo deve-se conectar o bloco operacional a um bloco de controle. A conexão pode ser vista na Figura 4.

Figura 4 - Conexão bloco operacional e bloco de controle

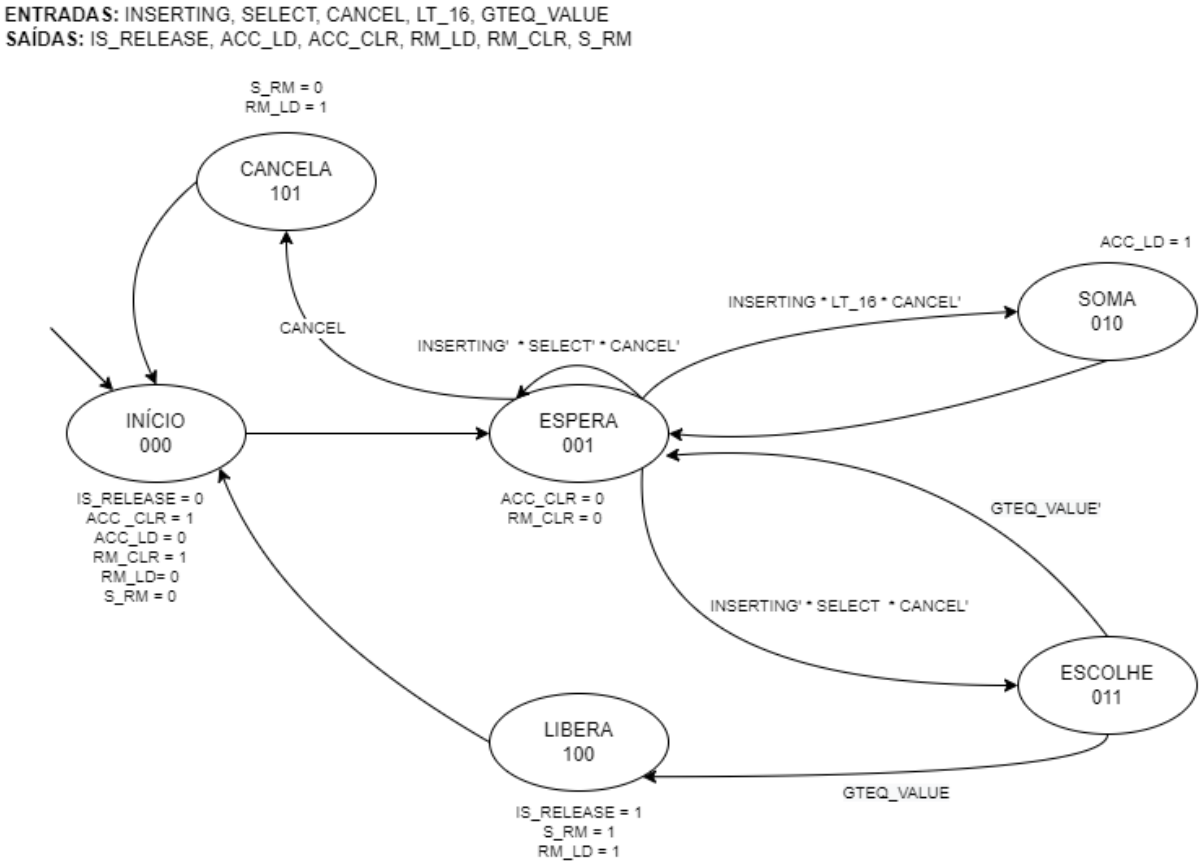


Fonte: Autores.

2.1.4 Máquina de estados do bloco de controle

Nesta etapa a máquina de estados de alto nível desenvolvida na seção 2.1 deve ser convertida em uma máquina de estados de baixo nível para o bloco de controle da seção 2.3. Para isso, todas as operações que envolvem dados devem ser substituídas por sinais de controle, que são ativados ou lidos pelo bloco de controle. O diagrama da Figura 4 já foi desenvolvido pensando neste fato. Na Figura 5 é possível ver como ficou a FSM.

Figura 5 - Máquina de estados do bloco de controle.



Fonte: Autores.

Tendo a FSM do bloco de controle em mãos, o passo seguinte é desenvolver a tabela verdade do sistema. Ele pode ser visto na Figura 6.

Figura 6 - Tabela verdade do sistema.

ESTADO ATUAIS			ENTRADAS					ESTADOS FUTUROS			SAÍDAS					
Q2	Q1	Q0	INSERTING	SELECT	LT_16	GTEQ_VALUE	CANCEL	D2	D1	D0	IS_RELEASE	ACC_CLR	ACC_LD	RM_CLR	RM_LD	S_RM
0	0	0	X	X	X	X	X	0	0	1	0	1	0	1	0	0
0	0	1	X	X	X	X	1	1	0	1	0	0	0	0	0	0
0	0	1	0	0	X	X	0	0	0	1	0	0	0	0	0	0
0	0	1	1	X	1	X	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	X	X	0	0	1	1	0	0	0	0	0	0
0	1	0	X	X	X	X	X	0	0	1	0	0	1	0	0	0
0	1	1	X	X	X	1	X	1	0	0	0	0	0	0	0	0
0	1	1	X	X	X	0	X	0	0	1	0	0	0	0	0	0
1	0	0	X	X	X	X	X	0	0	0	1	0	0	0	1	1
1	0	1	X	X	X	X	X	0	0	0	0	0	0	0	1	0
1	1	0	X	X	X	X	X	0	0	0	0	0	0	0	0	0
1	1	1	X	X	X	X	X	0	0	0	0	0	0	0	0	0

Fonte: Autores.

Tendo a tabela verdade em mãos, o próximo passo é retirar equações booleanas e elas podem ser vistas na Figura 7.

Figura 7 - Equações booleanas.

EQUAÇÕES BOOLEANAS
D2 = Q2'*Q1'*Q0*INSERTING'*SELECT'*LT_16'*GTEQ_VALUE'*CANCEL + Q2'*Q1*Q0*INSERTING'*SELECT'*LT_16'*GTEQ_VALUE*CANCEL'
D1 = Q2'*Q1'*Q0*INSERTING'*SELECT*CANCEL' + Q2'*Q1'*Q0*INSERTING*LT_16*CANCEL'
D0 = Q2'*INSERTING'*SELECT'*LT_16'*GTEQ_VALUE'*CANCEL' + Q2'*Q1'*Q0*INSERTING'*SELECT'*LT_16'*GTEQ_VALUE' + Q2'*Q1'*Q0*INSERTING'*LT_16'*GTEQ_VALUE'*CANCEL'
IS_RELEASE = Q2*Q1'*Q0'
ACC_CLR = Q2'*Q1'*Q0'
ACC_LD = Q2'*Q1*Q0'
RM_CLR = Q2'*Q1'*Q0'
RM_LD = Q2*Q1'*Q0' + Q2*Q1'*Q0
S_RM = Q2*Q1'*Q0'

Fonte: Autores.

2.2 IMPLEMENTAÇÃO

Com todas estas informações obtidas na seção 2.1, podemos começar a desenvolver os componentes do circuito proposto.

2.2.1 Registrador de 4 bits

O registrador de 4 bits foi implementado utilizando circuitos flip-flop. Para isso ser possível, foram utilizados 4 flip-flops para armazenar 1 bit cada um e 4 MUX 2x1, estes foram utilizados para selecionar se o registrador iria realimentar com seu bit de saída (assim permanecendo com o mesmo valor armazenado) ou se iria armazenar um novo dado, a depender da entrada load. O código VHDL do circuito de 4 bits pode ser visto na Figura 8.

Figura 8 - Código VHDL do Registrador de 4 bits

```
entity REG4 is
  port(A: in bit_vector(3 downto 0);
    ld, clk, clr: in bit;
    S: out bit_vector(3 downto 0));
end REG4;

architecture logic of REG4 is

component MUX21 is
  port(I0, I1, S: in bit;
          O: out bit
          );
end component;

component ffd is
  port ( clk ,D ,P , C : IN BIT;
    q: OUT BIT );
END component;

SIGNAL CLEAR: BIT;
  signal D, Q:bit_vector (3 downto 0);

  begin

    CLEAR <= NOT clr;

  MUX1: MUX21 port map (Q(0), A(0), ld, D(0));
  MUX2: MUX21 port map (Q(1), A(1), ld, D(1));
  MUX3: MUX21 port map (Q(2), A(2), ld, D(2));
  MUX4: MUX21 port map (Q(3), A(3), ld, D(3));

  FFD1: ffd port map (clk, D(0), '1', CLEAR, Q(0));
  FFD2: ffd port map (clk, D(1), '1', CLEAR, Q(1));
  FFD3: ffd port map (clk, D(2), '1', CLEAR, Q(2));
  FFD4: ffd port map (clk, D(3), '1', CLEAR, Q(3));

  S<=Q;

end logic;
```

Fonte: Autores.

2.2.2 Comparador de magnitude

A implementação do comparador de magnitude utiliza o conceito de circuito comportamental, onde este recebe 2 entradas (A e B) de 4 bits e devolve 3 saídas sendo elas LT, EQ e GT.

O comportamento das saídas se dá de forma que quando A é menor que B a saída LT recebe 1 e as demais 0. De maneira análoga, quando A é igual a B a saída EQ recebe 1 e as demais saídas tem nível lógico baixo. E por último, quando A é maior que B, a saída GT recebe nível lógico alto e as demais recebem nível lógico baixo.

O código VHDL do circuito comparador de magnitude de 4 bits pode ser observado na Figura 9.

Figura 9 - Código VHDL do Comparador de Magnitude

```
ENTITY COMP_4B IS
  PORT(
      A, B: IN bit_vector(3 downto 0);
      LT: OUT BIT;
      EQ: OUT BIT;
      GT: OUT BIT
  );
END COMP_4B;

ARCHITECTURE logic OF COMP_4B IS

  BEGIN
    GT <= '1' when (A > B)
    else '0';
    EQ <= '1' when (A = B)
    else '0';
    LT <= '1' when (A < B)
    else '0';

END logic;
```

Fonte: Autores.

2.2.3 Somador e Subtrator de 4 bits

A implementação dos circuitos somador e do subtrator de 4 bits do tipo *carry-ripple* foi feita através de portas lógicas, onde a entidade recebe 2 dados de 4 bits e fornece na saída a soma ou subtração desses dois dados, juntamente com o *carry out* da soma ou subtração.

O comportamento da saída é apenas o resultado da soma ou subtração das entradas providas no circuito. O código VHDL das entidades que constituem o circuito somador e o subtrator podem ser observadas nas Figuras 10, 11, 12 e 13.

Figura 10 - Código VHDL do Meio Somador

```
entity half_add is
    port (
        A, B: in bit;
        S, cout: out bit
    );
end half_add;

architecture logic of half_add is

begin
    S <= A xor B;
    cout <= A and B;
end logic;
```

Fonte: Autores.

Figura 11 - Código VHDL do Somador Completo

```
entity full_add is
    port (
        A, B, cin: in bit;
        S, cout: out bit
    );
end full_add;

architecture logic of full_add is

begin
    S <= A xor B xor cin;
    cout <= (B and cin) or (A and cin) or (A and B);
end logic;
```

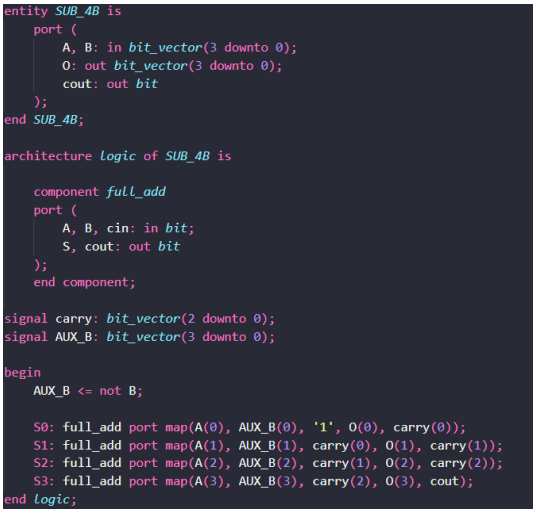
Fonte: Autores.

Figura 12 - Código VHDL do Somador de 4 bits

```
entity ADDER_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end ADDER_4B;

architecture logic of ADDER_4B is

    component half_add is
    port (
        A, B: in bit;
        S, cout: out bit
    );
    end component;

    component full_add
    port (
        A, B, cin: in bit;
        S, cout: out bit
    );
    end component;

signal carry: bit_vector(2 downto 0);

begin
    S0: half_add port map(A(0), B(0), O(0), carry(0));
    S1: full_add port map(A(1), B(1), carry(0), O(1), carry(1));
    S2: full_add port map(A(2), B(2), carry(1), O(2), carry(2));
    S3: full_add port map(A(3), B(3), carry(2), O(3), cout);
end logic;
```

Fonte: Autores.

Figura 13 - Código VHDL do Subtrator de 4 bits



```
entity SUB_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end SUB_4B;

architecture logic of SUB_4B is

    component full_add
    port (
        A, B, cin: in bit;
        S, cout: out bit
    );
    end component;

signal carry: bit_vector(2 downto 0);
signal AUX_B: bit_vector(3 downto 0);

begin
    AUX_B <= not B;

    S0: full_add port map(A(0), AUX_B(0), '1', O(0), carry(0));
    S1: full_add port map(A(1), AUX_B(1), carry(0), O(1), carry(1));
    S2: full_add port map(A(2), AUX_B(2), carry(1), O(2), carry(2));
    S3: full_add port map(A(3), AUX_B(3), carry(2), O(3), cout);
end logic;
```

Fonte: Autores.

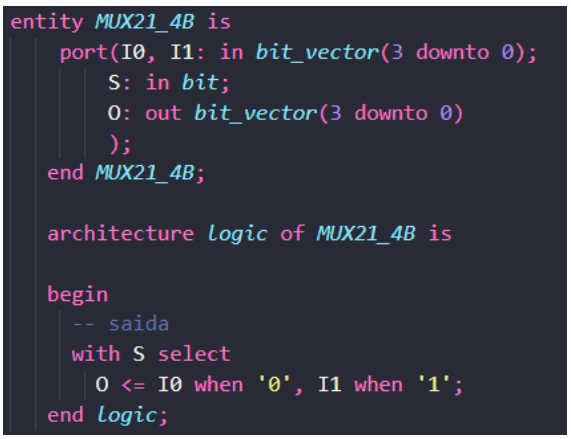
2.2.4 Multiplexador de 2 bits

A implementação do circuito multiplexador de 2 bits comportamental se desenvolve de forma onde recebe 2 entradas (I0 e I1) que serão os dados a serem escolhidos e uma entrada que serve de chave seletora (S) para estas outras entradas de dados.

O comportamento da saída segue conforme a entrada S, onde no momento que S for 0, a saída recebe I0 e se S tiver nível lógico alto, a saída recebe o dado de I1.

O código VHDL do circuito multiplexador de 2 bits comportamental pode ser observado na Figura 14.

Figura 14 - Código VHDL do Multiplexador 2x1 de 4 bits



```
entity MUX21_4B is
    port(I0, I1: in bit_vector(3 downto 0);
         S: in bit;
         O: out bit_vector(3 downto 0)
         );
end MUX21_4B;

architecture logic of MUX21_4B is

begin
    -- saida
    with S select
      O <= I0 when '0', I1 when '1';
end logic;
```

Fonte: Autores.

2.2.5 Multiplexador de 4 bits

A implementação do circuito multiplexador de 2 bits comportamental se desenvolve de maneira análoga ao circuito multiplexador de 2 bits, onde o circuito multiplexador de 4 bits recebe 4 entradas (I0, I1, I2, I3) ou simplesmente uma entrada de 4 bits (I) e outra entrada como chave seletora (S) de 2 bits que será utilizada para escolher o dado das outras entradas providas no circuito.

O comportamento da saída segue conforme a entrada S, onde no momento que S for 00, a saída recebe I(0), caso S seja 01 recebe a saída (I1) e assim por diante até S = 11, onde a saída será o dado I(3).

O código VHDL do circuito multiplexador de 4 bits comportamental pode ser observado na figura a seguir:

Figura 15 - Código VHDL do Multiplexador 4x1 de 4 bits

```
ENTITY MUX41_4B IS
    PORT(I0, I1, I2, I3: IN BIT_VECTOR(3 DOWNTO 0);
         S: IN BIT_VECTOR(1 DOWNTO 0);
         O: OUT BIT_VECTOR(3 DOWNTO 0));
END MUX41_4B;

ARCHITECTURE LOGIC OF MUX41_4B IS

BEGIN

    WITH S select
        O<= I0 WHEN "00",
            I1 WHEN "01",
            I2 WHEN "10",
            I3 WHEN "11";

END LOGIC;
```

Fonte: Autores.

2.2.6 Bloco Operacional

A implementação do bloco operacional se deu através da junção de todos os blocos desenvolvidos anteriormente para a construção de um único bloco responsável por lidar com todos os dados da máquina. Esta entidade vai ser responsável por receber o valor da moeda inserida, o código do produto e calcular o troco do cliente e também os sinais de *GTEQ_VALUE* e *LT_16* para o bloco de controle. A ideia do Bloco Operacional pode ser vista na Figura 3, e sua implementação em VHDL pode ser vista nas Figuras 16, 17 e 18.

Figura 16 - Código VHDL do início da Entidade do Bloco Operacional

```vhdl
ENTITY OP_BLOCK IS
    PORT (VALUE: IN BIT_VECTOR(3 DOWNTO 0);
        COD_PROD: IN BIT_VECTOR(1 DOWNTO 0);
        ACC_LD, ACC_CLR, S_RM, RM_LD, RM_CLR, CLK: IN BIT;
        REMAINING_MONEY: OUT BIT_VECTOR(3 DOWNTO 0);
        GTEQ_VALUE, LT_16: OUT BIT
    );
END OP_BLOCK;

ARCHITECTURE LOGIC OF OP_BLOCK IS

COMPONENT REG4 is
    port(A: in bit_vector(3 downto 0);
        ld, clk, clr: in bit;
        S: out bit_vector(3 downto 0));
end COMPONENT;

COMPONENT MUX41_4B IS
    PORT(I0, I1, I2, I3: IN BIT_VECTOR(3 DOWNTO 0);
        S: IN BIT_VECTOR(1 DOWNTO 0);
        O: OUT BIT_VECTOR(3 DOWNTO 0));
END COMPONENT;

COMPONENT MUX21_4B is
    port(I0, I1: in bit_vector(3 downto 0);
        S: in bit;
        O: out bit_vector(3 downto 0)
        );
end COMPONENT;

COMPONENT COMP_4B IS
  PORT(
      A, B: IN bit_vector(3 downto 0);
      LT: OUT BIT;
      EQ: OUT BIT;
      GT: OUT BIT
  );
END COMPONENT;
```

Fonte: Autores.

Figura 17 - Código VHDL do meio da Entidade do Bloco Operacional

```vhdl
COMPONENT ADDER_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end COMPONENT;

COMPONENT SUB_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end COMPONENT;

SIGNAL PROD_VALUE0, PROD_VALUE1, PROD_VALUE2, PROD_VALUE3, VALUE_PROD0, VALUE_PROD1, VALUE_PROD2,
        VALUE_PROD3, REG_OUT, ADD_RES, SUB_RES, PRODUCT_VALUE, RM_MUX: BIT_VECTOR(3 DOWNTO 0);
SIGNAL COMP_OUT: BIT_VECTOR(1 DOWNTO 0);
SIGNAL LIXO: BIT_VECTOR(4 DOWNTO 0);

BEGIN

-- VALORES DOS PRODUTOS E REGISTRADORES
PROD_VALUE0 <= "0101";
PROD_VALUE1 <= "1010";
PROD_VALUE2 <= "1000";
PROD_VALUE3 <= "0101";
REG_PROD0: REG4 PORT MAP(PROD_VALUE0, '1', CLK, '1', VALUE_PROD0);
REG_PROD1: REG4 PORT MAP(PROD_VALUE1, '1', CLK, '1', VALUE_PROD1);
REG_PROD2: REG4 PORT MAP(PROD_VALUE2, '1', CLK, '1', VALUE_PROD2);
REG_PROD3: REG4 PORT MAP(PROD_VALUE3, '1', CLK, '1', VALUE_PROD3);

-- MUX SELETOR DO PRODUTO
MUX_PROD: MUX41_4B PORT MAP(VALUE_PROD0, VALUE_PROD1, VALUE_PROD2, VALUE_PROD3, COD_PROD, PRODUCT_VALUE);
```

Fonte: Autores.

Figura 18 - Código VHDL do final da Entidade do Bloco Operacional

```
-- SOMADOR E REGISTRADOR
SOMA: ADDER_4B PORT MAP(VALUE, REG_OUT, ADD_RES, LIXO(0));
REG_ACC: REG4 PORT MAP(ADD_RES, ACC_LD, CLK, ACC_CLR, REG_OUT);

-- COMPARADOR DO ACUMULADOR X VALOR DO PRODUTO SELECIONADO
COMP_MONEY: COMP_4B PORT MAP(REG_OUT, PRODUCT_VALUE, LIXO(1), COMP_OUT(0), COMP_OUT(1));
GTEQ_VALUE <= COMP_OUT(0) OR COMP_OUT(1);

-- SUBTRAÇÃO
SUB_REM: SUB_4B PORT MAP(REG_OUT, PRODUCT_VALUE, SUB_RES, LIXO(2));

-- MUX SELETOR DO REMAINING MONEY
MUX_REM: MUX21_4B PORT MAP(REG_OUT, SUB_RES, S_RM, RM_MUX);

-- REGISTRADOR DE REMAINING MONEY
RESTO: REG4 PORT MAP(RM_MUX, RM_LD, CLK, RM_CLR, REMAINING_MONEY);

-- DEFINIÇÃO DE LT_16
COMP_LT: COMP_4B PORT MAP(ADD_RES, "1111", LT_16, LIXO(3), LIXO(4));

END LOGIC;
```

Fonte: Autores.

2.2.7 Bloco de Controle

O bloco de controle foi implementado de maneira comportamental, através de uma implementação de uma MDE do tipo Moore com dois processos, onde o primeiro processo é responsável por determinar qual o próximo estado a partir das entradas do sistema, e o segundo processo serve para atualizar o estado atual para o próximo estado definido no processo anterior. Ao final da entidade, as saídas são definidas de acordo com o estado atual da máquina. A implementação em VHDL da entidade do bloco de controle pode ser vista nas Figuras 19 e 20.

Figura 19 - Código VHDL do inicio da Entidade do Bloco de Controle e seu Primeiro Processo

```
library ieee ;
use ieee.std_logic_1164.all;

ENTITY MDE_B IS
    PORT(CLK, RST, INSERTING, SELECT_I, LT_16, GTEQ_VALUE, CANCEL: IN STD_LOGIC;
        IS_RELEASE, ACC_CLR, ACC_LD, RM_CLR, RM_LD, S_RM: OUT STD_LOGIC);
END MDE_B;

ARCHITECTURE LOGIC OF MDE_B IS
    TYPE STATE_TYPE is (INICIO, ESPERA, SOMA, ESCOLHE, LIBERA, CANCELA);
    SIGNAL Y_PRESENT, Y_NEXT: STATE_TYPE;
BEGIN
    PROCESS(INSERTING, SELECT_I, LT_16, GTEQ_VALUE, CANCEL, Y_PRESENT)
    BEGIN
        CASE Y_PRESENT IS
        WHEN INICIO =>
            Y_NEXT <= ESPERA;
        WHEN ESPERA =>
            IF (INSERTING = '0' AND SELECT_I = '0' AND CANCEL = '0') THEN Y_NEXT <= ESPERA;
            ELSIF (INSERTING = '1' AND LT_16 = '1' AND CANCEL = '0') THEN Y_NEXT <= SOMA;
            ELSIF (INSERTING = '0' AND SELECT_I = '1' AND CANCEL = '0') THEN Y_NEXT <= ESCOLHE;
            ELSIF (CANCEL = '1') THEN Y_NEXT <= CANCELA;
            END IF;
        WHEN SOMA =>
            Y_NEXT <= ESPERA;
        WHEN ESCOLHE =>
            IF (GTEQ_VALUE = '0') THEN Y_NEXT <= ESPERA;
            ELSIF (GTEQ_VALUE = '1') THEN Y_NEXT <= LIBERA;
            END IF;
        WHEN LIBERA =>
            Y_NEXT <= INICIO;
        WHEN CANCELA =>
            Y_NEXT <= INICIO;
        END CASE;
    END PROCESS;
```

Fonte: Autores.

Figura 20 - Código VHDL do final da Entidade do Bloco de Controle

```
    PROCESS (CLK, RST)
    BEGIN
        IF RST = '1' THEN
            Y_PRESENT <= INICIO;
        ELSIF (CLK'event AND CLK = '1') THEN
            Y_PRESENT <= Y_NEXT;
        END IF;
    END PROCESS;

    IS_RELEASE <= '1' WHEN Y_PRESENT = LIBERA ELSE '0';
    ACC_CLR <= '1' WHEN Y_PRESENT = INICIO ELSE '0';
    ACC_LD <= '1' WHEN Y_PRESENT = SOMA ELSE '0';
    RM_CLR <= '1' WHEN Y_PRESENT = INICIO ELSE '0';
    RM_LD <= '1' WHEN (Y_PRESENT = LIBERA OR Y_PRESENT = CANCELA) ELSE '0';
    S_RM <= '1' WHEN Y_PRESENT = LIBERA ELSE '0';

END LOGIC;
```

Fonte: Autores.

3. RESULTADOS

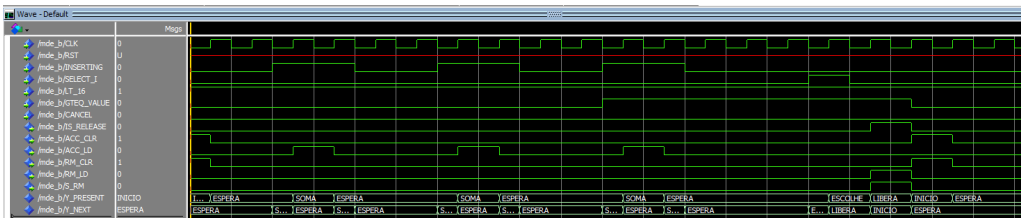
Para testar a eficácia da máquina desenvolvida foi simulado através do *software* ModelSim algumas situações para o controlador no intuito de verificar se o comportamento da máquina está de acordo com o esperado. Para isso, primeiro foi simulado a máquina em um processo normal, onde o cliente insere 3 vezes um determinado valor (a entrada *INSERTING* oscila 3 vezes de valor para indicar isto) e faz a seleção de um produto da máquina (a entrada *SELECT_I* muda para nível lógico alto), repare que para esta simulação consideramos que o cliente inseriu sempre valores válidos, onde a somatória dos valores é menor do que 16 (o que indica a entrada LT_16 em nível lógico alto) e ao final da inserção dos 3 valores a entrada *GTEQ_VALUE* foi forçada para nível lógico alto, para indicar que o valor inserido é igual ou superior ao produto escolhido, logo, o comportamento esperado é que a máquina libere o produto, ou seja, vá para o estado LIBERA e volte para o INICIO logo após. O código da simulação para este caso pode ser visto na Figura 21 e o resultado da simulação na Figura 22.

Figura 21 - Código de simulação do caso 1

```
1   vsim MDE_B
2
3   add wave *
4
5   force CLK 0 0, 1 1 -repeat 2
6   force INSERTING 0 0, 1 4, 0 8, 1 12, 0 16, 1 20, 0 24
7   force LT_16 1 0
8   force SELECT_I 0 0, 1 30, 0 32
9   force GTEQ_VALUE 0 0, 1 20, 0 35
10  force CANCEL 0 0
11  force LT_16 1 0
12
13  run 50
```

Fonte: Autores.

Figura 22 - Resultado da simulação do caso 1



Fonte: Autores.

No segundo caso vamos testar quase a mesma situação, porém com a diferença de dessa vez ao terminar de inserir os 3 valores, o *GTEQ_VALUE* ainda permanecerá desabilitado, neste caso, a máquina não deve ir do estado *ESCOLHE* para o estado *LIBERA* e

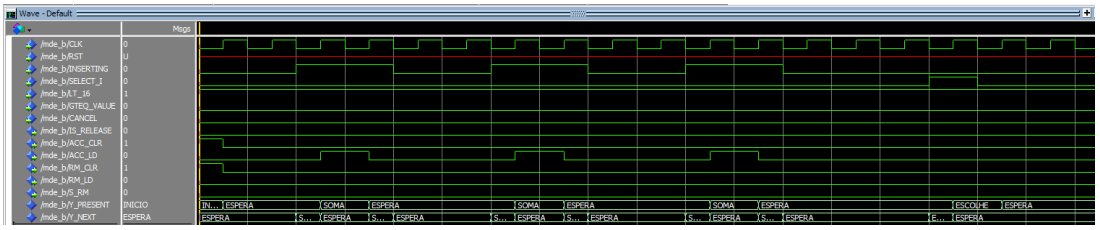
deve retornar para o estado *ESPERA*. O código da simulação e o resultado pode ser visto respectivamente nas Figuras 23 e 24.

Figura 23 - Código de simulação do caso 2

```
vsim MDE_B

add wave *

force CLK 0 0, 1 1 -repeat 2
force INSERTING 0 0, 1 4, 0 8, 1 12, 0 16, 1 20, 0 24
force LT_16 1 0
force SELECT_I 0 0, 1 30, 0 32
force GTEQ_VALUE 0 0
force CANCEL 0 0
force LT_16 1 0

run 50
```

Fonte: Autores.

Figura 24 - Resultado da simulação do caso 2



Fonte: Autores.

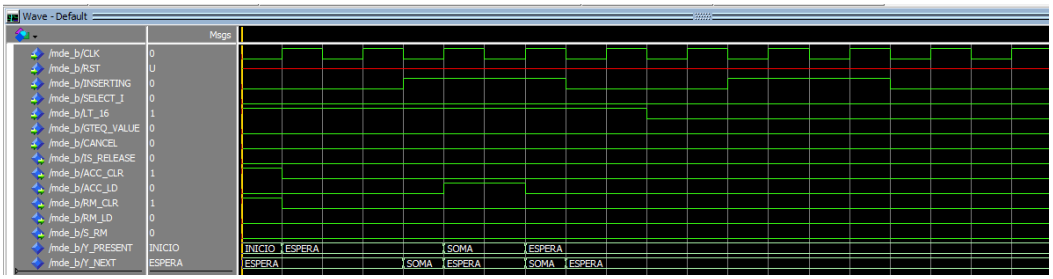
Em seguida, testamos o caso do cliente tentar inserir um valor que fosse superior ao limite de 15 reais da máquina, para isto, a entrada *LT_16* foi para 0 na segunda inserção de valor do cliente, logo, o comportamento esperado da máquina é que mesmo o *INSERTING* subindo para nível lógico alto, a máquina não saia do estado *ESPERA*. O código da simulação, bem como o resultado desta está exposto nas Figuras 25 e 26.

Figura 25 - Código de simulação do caso 3

```
1   vsim MDE_B
2
3   add wave *
4
5   force CLK 0 0, 1 1 -repeat 2
6   force INSERTING 0 0, 1 4, 0 8, 1 12, 0 16
7   force LT_16 1 0, 0 10
8   force SELECT_I 0 0
9   force GTEQ_VALUE 0 0
10  force CANCEL 0 0
11  force LT_16 1 0
12
13  run 50
14
```

Fonte: Autores.

Figura 26 - Resultado da simulação do caso 3



Fonte: Autores.

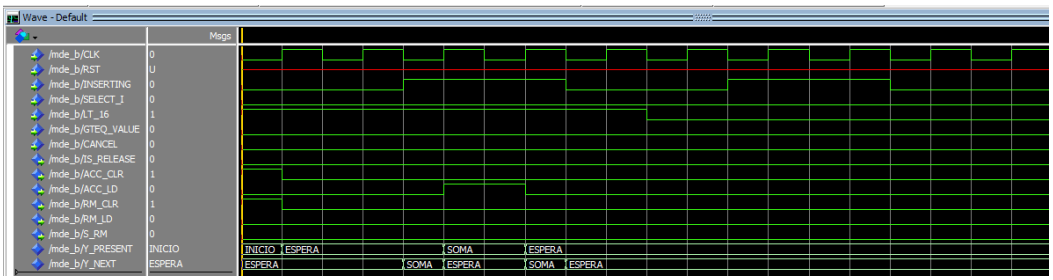
Por fim, foi testado a opção de cancelamento do cliente, o cliente deve ser capaz de cancelar a sua compra de produto, para testar esse caso, forçamos a entrada CANCEL para nível lógico alto logo após a inserção do primeiro valor do cliente, o comportamento esperado é que a máquina vá do estado ESPERA para o estado CANCELA e volte para o estado INICIO logo após. O código de simulação deste caso, bem como o resultado desta simulação pode ser visto nas Figuras 27 e 28.

Figura 27 - Código de simulação do caso 4

```
vsim MDE_B

add wave *

force CLK 0 0, 1 1 -repeat 2
force INSERTING 0 0, 1 4, 0 8, 1 12, 0 16, 1 20, 0 24
force LT_16 1 0
force SELECT_I 0 0, 1 30, 0 32
force GTEQ_VALUE 0 0
force CANCEL 0 0
force LT_16 1 0

run 50
```

Fonte: Autores.

Figura 28 - Resultado da simulação do caso 4



Fonte: Autores.

O código-fonte em VHDL na íntegra da implementação desenvolvida pode ser visualizado no ANEXO A.

4. CONCLUSÃO

Este trabalho teve como objetivo desenvolver e implementar um sistema digital que simula uma máquina de *snacks*. Esta recebe clientes os quais são capazes de adicionar dinheiro e inserir o código do produto que desejam comprar. A máquina será capaz de verificar se o dinheiro é suficiente para o produto selecionado ou não. Caso seja suficiente, o produto será liberado e será calculado o troco, se necessário. Caso o dinheiro não seja suficiente, o cliente terá a opção de adicionar mais dinheiro ou cancelar a operação. Cancelando a operação ele receberá seu dinheiro de volta. O desenvolvimento se deu a fim de pôr em prática os conceitos estudados na disciplina de Circuitos Digitais e, por isso, algumas funcionalidades da máquina foram simplificadas, principalmente grandezas dos dados, mas o mais importante foi atingido, que foi possível englobar quase todos os assuntos vistos durante o semestre, como circuitos combinacionais, sequenciais e modo de projetos RTL. Outro ponto bastante importante foi a utilização da linguagem de descrição de *hardware* para descrever todos os componentes do sistema, criar os blocos de controle e operacional, e conectar tudo. Por fim, com todas as simulações feitas, o comportamento esperado do sistema foi alcançado. A realização do trabalho foi muito proveitosa, a única dificuldade maior foi em relação ao tempo, principalmente pelo grupo estar com um integrante a menos, o que foi o principal fator para não dar tempo a entrega das simulações na apresentação da segunda (14/02), diante da falta de tempo dos membros, não foi possível a entrega da máquina totalmente conectada entre o bloco operacional e o bloco de controle, mas estes foram desenvolvidos.

REFERÊNCIAS

Mercearia Pronta. **Vending Machines**, 2022. Disponível em: <https://mercearia pronta.com.br/vending-machines/>. Acesso em: 11 de fevereiro de 2022.

VAHID, Frank. **Sistemas digitais: projeto, otimização e HDLS**. Rio Grande do Sul: Artmed Bookman, 2008. 558 p

ANEXO A - CÓDIGO-FONTE

```
ENTITY ffd IS
    port ( clk ,D ,P , C : IN BIT ;
        q : OUT BIT );
END ffd ;

ARCHITECTURE ckt OF ffd IS
    SIGNAL qS : BIT;
    BEGIN
PROCESS ( clk ,P ,C )
    BEGIN
    IF P = '0' THEN qS <= '1';
    ELSIF C = '0' THEN qS <= '0';
    ELSIF clk = '1' AND clk ' EVENT THEN
    qS <= D ;
    END IF;
END PROCESS ;
q <= qS ;
END ckt ;

entity MUX21 is
    port(I0, I1, S: in bit;
         O: out bit
         );
   end MUX21;

   architecture logic of MUX21 is

   begin
     -- saida
     with S select
       O <= I0 when '0', I1 when '1';
   end logic;

entity MUX21_4B is
    port(I0, I1: in bit_vector(3 downto 0);
        S: in bit;
        O: out bit_vector(3 downto 0)
        );
   end MUX21_4B;

   architecture logic of MUX21_4B is
```

```
   begin
     -- saida
     with S select
       O <= I0 when '0', I1 when '1';
   end logic;

ENTITY MUX41_4B IS
    PORT(I0, I1, I2, I3: IN BIT_VECTOR(3 DOWNTO 0);
        S: IN BIT_VECTOR(1 DOWNTO 0);
        O: OUT BIT_VECTOR(3 DOWNTO 0));
END MUX41_4B;

ARCHITECTURE LOGIC OF MUX41_4B IS

BEGIN

    WITH S select
        O<= I0 WHEN "00",
            I1 WHEN "01",
            I2 WHEN "10",
            I3 WHEN "11";

END LOGIC;

entity REG4 is
  port(A: in bit_vector(3 downto 0);
    ld, clk, clr: in bit;
    S: out bit_vector(3 downto 0));
end REG4;

architecture logic of REG4 is

component MUX21 is
  port(I0, I1, S: in bit;
         O: out bit
         );
end component;

component ffd is
  port ( clk ,D ,P , C : IN BIT;
    q: OUT BIT );
END component;
```

```
SIGNAL CLEAR: BIT;
  signal D, Q:bit_vector (3 downto 0);

  begin

    CLEAR <= NOT clr;

  MUX1: MUX21 port map (Q(0), A(0), ld, D(0));
  MUX2: MUX21 port map (Q(1), A(1), ld, D(1));
  MUX3: MUX21 port map (Q(2), A(2), ld, D(2));
  MUX4: MUX21 port map (Q(3), A(3), ld, D(3));

  FFD1: ffd port map (clk, D(0), '1', CLEAR, Q(0));
  FFD2: ffd port map (clk, D(1), '1', CLEAR, Q(1));
  FFD3: ffd port map (clk, D(2), '1', CLEAR, Q(2));
  FFD4: ffd port map (clk, D(3), '1', CLEAR, Q(3));

  S<=Q;

end logic;

entity half_add is
    port (
        A, B: in bit;
        S, cout: out bit
    );
end half_add;

architecture logic of half_add is

begin
    S <= A xor B;
    cout <= A and B;
end logic;

entity full_add is
    port (
        A, B, cin: in bit;
        S, cout: out bit
    );
end full_add;

architecture logic of full_add is
```

```
begin
    S <= A xor B xor cin;
    cout <= (B and cin) or (A and cin) or (A and B);
end logic;

entity ADDER_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end ADDER_4B;

architecture logic of ADDER_4B is

    component half_add is
    port (
        A, B: in bit;
        S, cout: out bit
    );
    end component;

    component full_add
    port (
        A, B, cin: in bit;
        S, cout: out bit
    );
    end component;

signal carry: bit_vector(2 downto 0);

begin
    S0: half_add port map(A(0), B(0), O(0), carry(0));
    S1: full_add port map(A(1), B(1), carry(0), O(1), carry(1));
    S2: full_add port map(A(2), B(2), carry(1), O(2), carry(2));
    S3: full_add port map(A(3), B(3), carry(2), O(3), cout);
end logic;

entity SUB_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
```

```
        cout: out bit
    );
end SUB_4B;

architecture logic of SUB_4B is

    component full_add
    port (
        A, B, cin: in bit;
        S, cout: out bit
    );
    end component;

signal carry: bit_vector(2 downto 0);
signal AUX_B: bit_vector(3 downto 0);

begin
    AUX_B <= not B;

    S0: full_add port map(A(0), AUX_B(0), '1', O(0), carry(0));
    S1: full_add port map(A(1), AUX_B(1), carry(0), O(1), carry(1));
    S2: full_add port map(A(2), AUX_B(2), carry(1), O(2), carry(2));
    S3: full_add port map(A(3), AUX_B(3), carry(2), O(3), cout);
end logic;

ENTITY OP_BLOCK IS
    PORT (VALUE: IN BIT_VECTOR(3 DOWNTO 0);
        COD_PROD: IN BIT_VECTOR(1 DOWNTO 0);
        ACC_LD, ACC_CLR, S_RM, RM_LD, RM_CLR, CLK: IN BIT;
        REMAINING_MONEY: OUT BIT_VECTOR(3 DOWNTO 0);
        GTEQ_VALUE, LT_16: OUT BIT
    );
END OP_BLOCK;

ARCHITECTURE LOGIC OF OP_BLOCK IS

COMPONENT REG4 is
    port(A: in bit_vector(3 downto 0);
        ld, clk, clr: in bit;
        S: out bit_vector(3 downto 0));
end COMPONENT;

COMPONENT MUX41_4B IS
```

```
    PORT(I0, I1, I2, I3: IN BIT_VECTOR(3 DOWNTO 0);
        S: IN BIT_VECTOR(1 DOWNTO 0);
        O: OUT BIT_VECTOR(3 DOWNTO 0));
END COMPONENT;

COMPONENT MUX21_4B is
    port(I0, I1: in bit_vector(3 downto 0);
        S: in bit;
        O: out bit_vector(3 downto 0)
        );
end COMPONENT;

COMPONENT COMP_4B IS
  PORT(
      A, B: IN bit_vector(3 downto 0);
      LT: OUT BIT;
      EQ: OUT BIT;
      GT: OUT BIT
  );
END COMPONENT;

COMPONENT ADDER_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end COMPONENT;

COMPONENT SUB_4B is
    port (
        A, B: in bit_vector(3 downto 0);
        O: out bit_vector(3 downto 0);
        cout: out bit
    );
end COMPONENT;

SIGNAL PROD_VALUE0, PROD_VALUE1, PROD_VALUE2, PROD_VALUE3, VALUE_PROD0,
VALUE_PROD1, VALUE_PROD2,
         VALUE_PROD3, REG_OUT, ADD_RES, SUB_RES, PRODUCT_VALUE, RM_MUX:
BIT_VECTOR(3 DOWNTO 0);
SIGNAL COMP_OUT: BIT_VECTOR(1 DOWNTO 0);
SIGNAL LIXO: BIT_VECTOR(4 DOWNTO 0);
```

```
BEGIN

-- VALORES DOS PRODUTOS E REGISTRADORES
PROD_VALUE0 <= "0101";
PROD_VALUE1 <= "1010";
PROD_VALUE2 <= "1000";
PROD_VALUE3 <= "0101";
REG_PROD0: REG4 PORT MAP(PROD_VALUE0, '1', CLK, '1', VALUE_PROD0);
REG_PROD1: REG4 PORT MAP(PROD_VALUE1, '1', CLK, '1', VALUE_PROD1);
REG_PROD2: REG4 PORT MAP(PROD_VALUE2, '1', CLK, '1', VALUE_PROD2);
REG_PROD3: REG4 PORT MAP(PROD_VALUE3, '1', CLK, '1', VALUE_PROD3);

-- MUX SELETOR DO PRODUTO
MUX_PROD: MUX41_4B PORT MAP(VALUE_PROD0, VALUE_PROD1, VALUE_PROD2,
VALUE_PROD3, COD_PROD, PRODUCT_VALUE);

-- SOMADOR E REGISTRADOR
SOMA: ADDER_4B PORT MAP(VALUE, REG_OUT, ADD_RES, LIXO(0));
REG_ACC: REG4 PORT MAP(ADD_RES, ACC_LD, CLK, ACC_CLR, REG_OUT);

-- COMPARADOR DO ACUMULADOR X VALOR DO PRODUTO SELECIONADO
COMP_MONEY: COMP_4B PORT MAP(REG_OUT, PRODUCT_VALUE, LIXO(1),
COMP_OUT(0), COMP_OUT(1));
GTEQ_VALUE <= COMP_OUT(0) OR COMP_OUT(1);

-- SUBTRAÇÃO
SUB_REM: SUB_4B PORT MAP(REG_OUT, PRODUCT_VALUE, SUB_RES, LIXO(2));

-- MUX SELETOR DO REMAINING MONEY
MUX_REM: MUX21_4B PORT MAP(REG_OUT, SUB_RES, S_RM, RM_MUX);

-- REGISTRADOR DE REMAINING MONEY
RESTO: REG4 PORT MAP(RM_MUX, RM_LD, CLK, RM_CLR, REMAINING_MONEY);

-- DEFINIÇÃO DE LT_16
COMP_LT: COMP_4B PORT MAP(ADD_RES, "1111", LT_16, LIXO(3), LIXO(4));

END LOGIC;

library ieee ;
use ieee.std_logic_1164.all;
```

```
ENTITY MDE_B IS
     PORT(CLK, RST, INSERTING, SELECT_I, LT_16, GTEQ_VALUE, CANCEL: IN
STD_LOGIC;
               IS_RELEASE,  ACC_CLR,  ACC_LD,  RM_CLR,  RM_LD,  S_RM:  OUT
STD_LOGIC);
END MDE_B;

ARCHITECTURE LOGIC OF MDE_B IS
       TYPE  STATE_TYPE  is  (INICIO,  ESPERA,  SOMA,  ESCOLHE,  LIBERA,
CANCELA);
    SIGNAL Y_PRESENT, Y_NEXT: STATE_TYPE;
BEGIN
    PROCESS(INSERTING, SELECT_I, LT_16, GTEQ_VALUE, CANCEL, Y_PRESENT)
    BEGIN
        CASE Y_PRESENT IS
        WHEN INICIO =>
            Y_NEXT <= ESPERA;
        WHEN ESPERA =>
              IF (INSERTING = '0' AND SELECT_I = '0' AND CANCEL = '0')
THEN Y_NEXT <= ESPERA;
              ELSIF (INSERTING = '1' AND LT_16 = '1' AND CANCEL = '0')
THEN Y_NEXT <= SOMA;
             ELSIF (INSERTING = '0' AND SELECT_I = '1' AND CANCEL = '0')
THEN Y_NEXT <= ESCOLHE;
            ELSIF (CANCEL = '1') THEN Y_NEXT <= CANCELA;
            END IF;
        WHEN SOMA =>
            Y_NEXT <= ESPERA;
        WHEN ESCOLHE =>
            IF (GTEQ_VALUE = '0') THEN Y_NEXT <= ESPERA;
            ELSIF (GTEQ_VALUE = '1') THEN Y_NEXT <= LIBERA;
            END IF;
        WHEN LIBERA =>
            Y_NEXT <= INICIO;
        WHEN CANCELA =>
            Y_NEXT <= INICIO;
        END CASE;
    END PROCESS;

    PROCESS (CLK, RST)
    BEGIN
        IF RST = '1' THEN
            Y_PRESENT <= INICIO;
```

```
        ELSIF (CLK'event AND CLK = '1') THEN
            Y_PRESENT <= Y_NEXT;
        END IF;
    END PROCESS;

    IS_RELEASE <= '1' WHEN Y_PRESENT = LIBERA ELSE '0';
    ACC_CLR <= '1' WHEN Y_PRESENT = INICIO ELSE '0';
    ACC_LD <= '1' WHEN Y_PRESENT = SOMA ELSE '0';
    RM_CLR <= '1' WHEN Y_PRESENT = INICIO ELSE '0';
     RM_LD <= '1' WHEN (Y_PRESENT = LIBERA OR Y_PRESENT = CANCELA) ELSE
'0';
    S_RM <= '1' WHEN Y_PRESENT = LIBERA ELSE '0';

END LOGIC;
```