

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Prática em Sistemas Digitais (SSC-0108)

Relatório de Construção das Máquinas de Estado Finito

Arthur Ernesto de Carvalho 14559479

Luiz Felipe Diniz Costa 13782032

Thiago Zero Araujo - 11814183

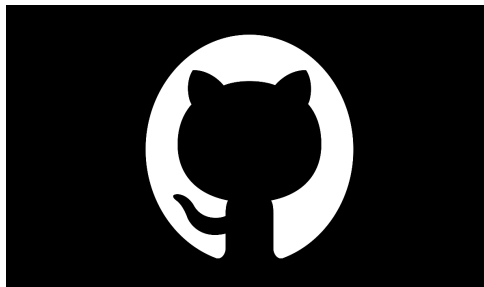
**São Carlos
2023**

Índice

Projeto Completo no Github	7
Introdução	7
Recursos Utilizados	8
Elevador	9
Diagrama de Estados	9
Tabela de Transição de Estados	10
Observações complementares	11
Implementação	12
Código Fonte	12
Importação das Bibliotecas	16
Declaração da Entidade	16
Portas de Entrada	17
Portas de Saída	17
Arquitetura e Declaração dos Componentes	17
Implementação da Lógica do Elevador	18
Processo Principal	19
Controle	19
Saída e Movimento	19
Atualização da Saída Current	19
Sinal de Movimento	19
Resumo	20
Máquina de Refrigerante	21
Diagrama de Estados	21
Tabela de Transição de Estados	22
Observações Complementares	23
Implementação	24
Código Fonte	24
Importação das Bibliotecas	31
Declaração da Entidade	31

Portas de Entrada	32
Portas de Saída	32
Arquitetura e Declaração de Componentes	33
Implementação da Lógica do Sistema	34
Processo Principal	34
Estados da Máquina	34
Função de Valor da Moeda	35
Saídas e Transições de Estado	35
Resumo	36
Conclusões Finais	36

Projeto Completo no Github



Finite States Machines

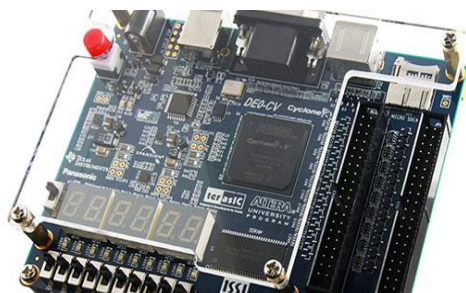
Desenvolvimento das máquinas de estados finitos: elevador e máquina de refrigerante

Introdução

Este segundo projeto envolve a construção de dois subprojetos distintos: um elevador e uma máquina de refrigerante. Diferentemente do projeto anterior, empregamos diretamente a linguagem VHDL e utilizando da lógica de máquinas de estado.

Recursos Utilizados

FPGA Cyclone V DE0-CV => Código da placa Cyclone 5: 5CEBA4F23C7 Confira o [Manual da FPGA](#)



FPGA Cyclone V DE0-CV

Código da placa Cyclone 5: 5CEBA4F23C7

Confira o Manual da FPGA



Quartus Prime

O Quartus Prime é uma plataforma de design digital versátil para criação e otimização de circuitos integrados

Elevador

Diagrama de Estados

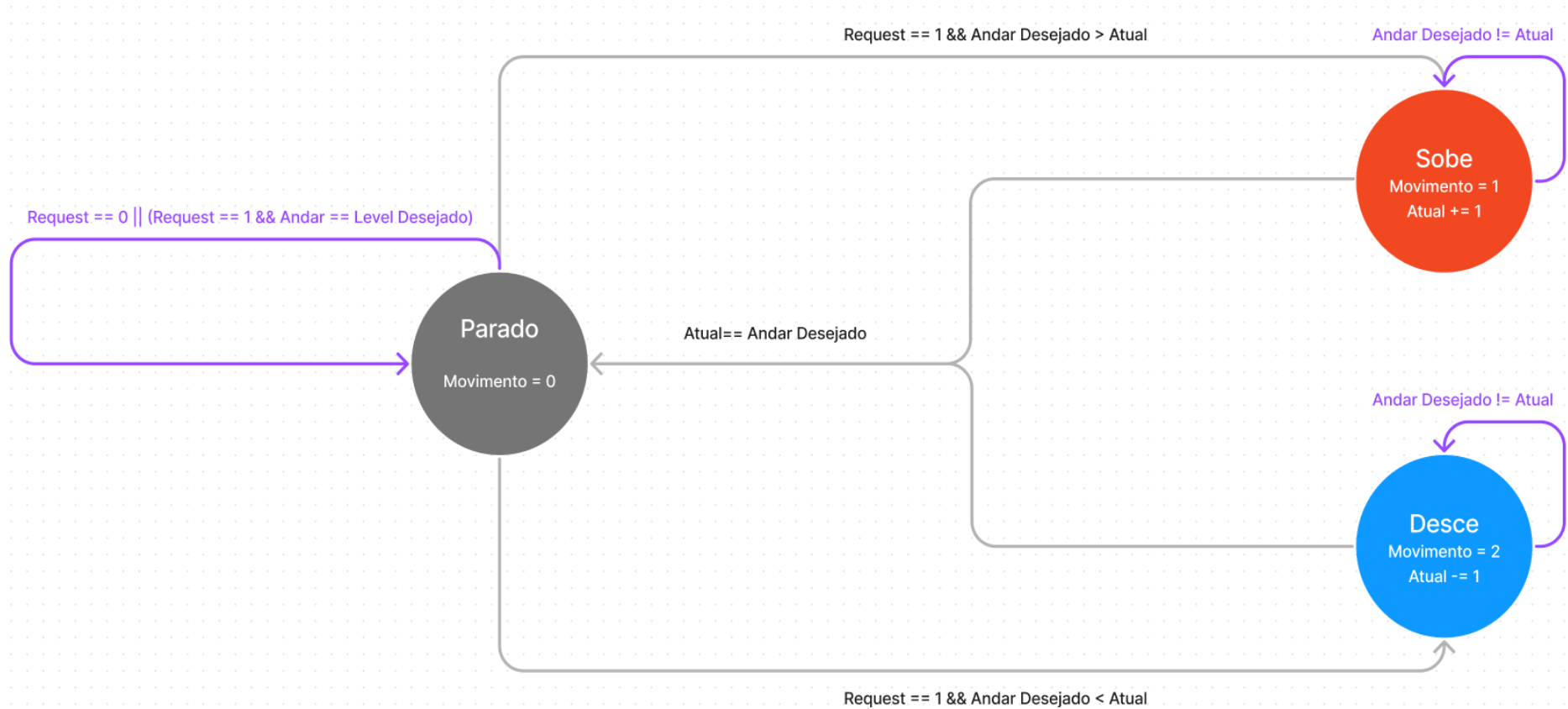


Tabela de Transição de Estados

Comportamento	(Andar atual - Andar Desejado)	Req	Not_Moving _{ant}	Rising _{an}	Descending _{ant}	Not_Moving	Rising	Descending
Reset = 1	Andares <= 0000	x	x	x	x	1	0	0
tava descendo e chegou	0	0	0	0	1	1	0	0
tava subindo e chegou	0	0	0	1	0	1	0	0
tá parado e n req	0	0	1	0	0	1	0	0
req pro msm andar	0	1	1	0	0	1	0	0
impossível	< 0	0	0	0	1	x	x	x
subindo	< 0	0	0	1	0	0	1	0
quer subir mas n req	< 0	0	1	0	0	1	0	0
quer subir e fez req	< 0	1	1	0	0	0	1	0
descendo	> 0	0	0	0	1	0	0	1
impossível	> 0	0	0	1	0	x	x	x
quer descer mas n req	> 0	0	1	0	0	1	0	0
quer descer e fez req	> 0	1	1	0	0	0	0	1

Outras configurações são impossíveis nas especificações dadas no enunciado.								

Estado Atual	Condição de Entrada	Próximo Estado	Observações
not_moving	req = '1' e desired_floor > current_floor	rising	Se há um pedido e o andar desejado é acima do atual
not_moving	req = '1' e desired_floor < current_floor	descending	Se há um pedido e o andar desejado é abaixo do atual
not_moving	req = '0' ou desired_floor = current_floor	not_moving	Sem atividade ou andar desejado já alcançado
rising	objective_floor = current_floor + 1	not_moving	Andar desejado alcançado ao subir
descending	objective_floor = current_floor - 1	not_moving	Andar desejado alcançado ao descer

Observações complementares

- Estado de Reset

Quando o sinal de **'reset'** é adicionado, o sistema retorna ao estado **'not_moving'**, independentemente do estado atual.

- Incremento/Decremento do Andar

No estado **'rising'**, **'current_floor'** é incrementado, já no estado **'descending'**, é decrementado.

- Atualização do Andar Atual

A atualização do andar atual, por meio do sinal **‘current’**, é feita nos estados **‘rising’** e **‘descending’**

Implementação

O código faz a implementação de um sistema de controle de elevador, vamos percorrê-lo e explicar os blocos importantes para o funcionamento

Código Fonte

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

entity elevator is
    port (
        clk, clk_placa: std_logic;
        reset: in bit;
        req: in bit;
        desired_floor: in unsigned(3 downto 0) := (others => '0');
        current: out unsigned(3 downto 0) := (others => '0');
        movement: out bit_vector(1 downto 0) := (others => '0'));

end elevator;
```

```

architecture behaviour of elevator is
    type states is (not_moving, rising, descending);
    signal state: states;
    signal current_floor: unsigned(3 downto 0) := (others => '0');
    signal objective_floor: unsigned(3 downto 0) := (others => '0');
    signal fixed_clock: std_logic;

    component Debouncing_Button_VHDL is
        port(
            button: in std_logic;
            clk: in std_logic;
            debounced_button: out std_logic
        );
    end component;

begin
    instance_debouncer: Debouncing_Button_VHDL
        port map (
            button => clk, clk => clk_placa,
            debounced_button => fixed_clock
        );

    process (fixed_clock)
    begin

```

```

if (reset = '1') then
    state <= not_moving;
    current_floor <= "0000";
    current <= "0000";
    objective_floor <= "0000";
    movement <= "00";

elsif (fixed_clock'event) and (fixed_clock = '1') then
    case state is

        when not_moving =>
            movement <= "00";
            if (req = '1' and desired_floor /= current_floor) then
                objective_floor <= desired_floor;
                if (desired_floor > current_floor) then
                    state <= rising;
                else
                    state <= descending;
                end if;
            end if;

        when rising =>
            movement <= "01";
    end case;
end if;

```

```

current_floor <= current_floor + 1;
current <= current_floor +1;

if (objective_floor = current_floor+1) then
    state <= not_moving;
end if;

when descending =>
    movement <= "10";

current_floor <= current_floor - 1;
current <= current_floor -1;

if (objective_floor = current_floor-1) then
    state <= not_moving;
end if;

end case;
end if;
end process;
end behaviour;

```

Importação das Bibliotecas

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

A biblioteca **'IEEE'** é necessária para importar alguns módulos, dentro dela **'std_logic_1164'** serve para fornecer os tipos de dados para a lógica digital, enquanto a **'numeric_std'** fornece tipos de dados para números, como o **'unsigned'** por exemplo.

Declaração da Entidade

```
entity elevator is  
  port (  
    clk, clk_placa: std_logic;  
    reset: in bit;  
    req: in bit;  
    desired_floor: in unsigned(3 downto 0) := (others => '0');  
    current: out unsigned(3 downto 0) := (others => '0');  
    movement: out bit_vector(1 downto 0) := (others => '0');  
  end elevator;
```

'elevator': define a interface do sistema de elevador.

Portas de Entrada

- **'clk'** e **'clk_placa'**: são o botão do clock e o clock interno à placa.
- **'reset'**: sinal para reiniciar o sistema.
- **'req'**: sinal de solicitação para mover o elevador.
- **'desired_floor'**: número do andar desejado (4 bits).

Portas de Saída

- **'current'**: andar atual em que o elevador se encontra (4 bits).
- **'movement'**: indica o estado do movimento do elevador (2 bits).

Arquitetura e Declaração dos Componentes

```
architecture behaviour of elevator is
    ...
    component Debouncing_Button_VHDL is ...
    end component;
    ...
begin
    ...
end behaviour;
```

A arquitetura behaviour do Elevador define o comportamento interno do sistema de elevador. Já o componente **'Debouncing_Button_VHDL'** é importado de outro arquivo e serve para para debouncing, que exerce a função de eliminar ruídos de um botão. Útil para estabilizar o sinal do clock.

Implementação da Lógica do Elevador

```
instance_debouncer: Debouncing_Button_VHDL
  port map (
    button => clk, clk => clk_placa,
    debounced_button => fixed_clock
  );

process (fixed_clock)
begin
  if (reset = '1') then
    ...
  elsif (fixed_clock'event) and (fixed_clock = '1') then
    ...
  end if;
end process;
```

Como comentado anteriormente, a instância do debouncer serve para integrá-lo dentro do circuito

Processo Principal

Executa a cada evento de subida de borda no 'fixed_clock'

- **Reset:** Se o sinal de reset estiver ativado, o estado do elevador e as variáveis de andar são reinicializadas.

Controle

Dentro deste processo, a lógica de controle do elevador é implementada:

- **'not_moving':** Se nenhum movimento é necessário, o elevador permanece neste estado. Se uma solicitação é recebida e o andar desejado não é o atual, o sistema se prepara para mover-se para o rising ou descending.
- **'rising':** O elevador está subindo. O andar atual é incrementado. Se o andar objetivo for alcançado, retorna ao estado not_moving.
- **'descending':** O elevador está descendo. O andar atual é decrementado. Se o andar objetivo for alcançado, retorna ao estado not_moving.

Saída e Movimento

Atualização da Saída Current

Em ambos os estados **'rising'** e descending, a saída **'current'** é atualizada para refletir o andar atual do elevador.

Sinal de Movimento

'movement' indica se o elevador está parado, subindo ou descendo.

Resumo

Nosso código implementa um elevador que funciona por meio de uma máquina de estados para gerenciar os movimentos do elevador com base nas solicitações dos andares. As transições de estado são sincronizadas com o `'fixed_clock'` para garantir uma operação estável.

Máquina de Refrigerante

Diagrama de Estados



Tabela de Transição de Estados

Comportamento	Req	Valor Inserido	\$ _{ant}	Guarda_ Dinheiro _{ant}	Sem_ Dinheiro _{ant}	\$	Guarda_ Dinheiro	Sem_ Dinheiro	Solta Troco	Solta Refri
Colocando dinheiro quando n tem nada	0	a	0	0	1	a	1	0	0	0
Está guardando dinheiro e \$ < 1.00	0	a	< 1.00	1	0	\$ _{ant} + a	1	0	0	0
Dinheiro ultrapassou o \$1.00	x	a	> 1.00	1	0	0	0	1	1	0
Tem 1.00, mas o infeliz botou mais \$ antes de req	0	a	1.00	1	0	\$ _{ant} + a	1	0	0	0
Tem 1.00 e fez req	1	x	1.00	1	0	0	0	1	0	1
Fez req mas n tinha 1.00 ainda	1	x	< 1.00	1	0	0	0	1	1	0
Fez req mas tinha exatamente 0 reais	1	x	0	0	1	0	0	1	0	0
Outras configurações são impossíveis nas especificações dadas no enunciado.										
'a' é um valor qualquer de moeda, onde a é 0.1\$ ou 0.25\$ ou 0.5\$ ou 1.00\$										

Estado Atual	Condição de Entrada	Próximo Estado	Ação/Resultado
no_money	coin -/"000"	money	Inicia a acumulação do valor
money	current_money + value(coin) = 20	no_money	Entrega Pepsi sem troco
money	current_money + value(coin) > 20	no_money	Entrega Pepsi com troco
money	current_money + value(coin) < 20	money	Continua acumulado valor
(Qualquer)	reset = '1'	no_money	Reinicia o sistema

Observações Complementares

- Condição de Reset: Independentemente do estado atual, se o sinal de reset é acionado, o sistema retorna ao estado **'no_money'** e todas as saídas e contadores são reiniciados.
- Entrada de Moeda: A inserção de uma moeda (**coin ≠ "000"**) resulta na transição do estado **'no_money'** para **'money'**, iniciando a acumulação do valor.
- Avaliação do Valor Acumulado: Se o valor acumulado for igual a 20 (o preço da **Pepsi**) e o botão for pressionado, o sistema entrega uma Pepsi e retorna ao estado **'no_money'** sem dar troco.

- Se o valor acumulado exceder 20, o sistema retorna ao estado **'no_money'** com sinal de troco - **'exchange'**.
- Se o valor ainda não atingiu 20 e o botão for pressionado, o sistema retorna ao estado **'no_money'** com sinal de troco - **'exchange'**.
- Se o valor ainda não atingiu 20, o sistema permanece no estado **'money'**, continuando a acumular valor.

Implementação

Código Fonte

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pepsi is
    port (
        clk, clk_placa: std_logic;
        reset: in bit;
        req: in bit;
        coin: in bit_vector(2 downto 0);

        current: out unsigned(4 downto 0) := (others => '0');
        pepsiCola: out bit := '0';
        exchange: out bit := '0';
        state_out: out bit := '0'
    );
end pepsi;
```

```
architecture behaviour of pepsi is
    type states is (money, no_money);
    signal state: states;
    signal current_money: unsigned(4 downto 0) := (others => '0');
    signal fixed_clock: std_logic;
```

```

component Debouncing_Button_VHDL is

    port(

        button: in std_logic;

        clk: in std_logic;

        debounced_button: out std_logic

    );

end component;

function value ( coin_in : in bit_vector(2 downto 0))

    return unsigned is

        variable value_out : unsigned(4 downto 0) := (others => '0');

begin

    case coin_in is

        when "001" => value_out := value_out + 2;

        when "010" => value_out := value_out + 5;

        when "011" => value_out := value_out + 10;

        when "100" => value_out := value_out + 20;

        when others => value_out := value_out;

    end case;

```

```

        return value_out;

    end function value;

begin

    instance_debouncer: Debouncing_Button_VHDL

        port map (

            button => clk, clk => clk_placa,

            debounced_button => fixed_clock

        );

    process (fixed_clock)

    begin

        if (reset = '1') then

            state <= no_money;

            pepsi_cola <= '0';

            exchange <= '0';

            state_out <= '0';

            current <= "00000";

            current_money <= "00000";

        end if;

    end process;

end architecture;

```



```

elsif (fixed_clock'event) and (fixed_clock = '1') then

    case state is

        when no_money =>

            if (coin /= "000") then

                state <= money;

                current <= current_money + value(coin);

                current_money <= current_money + value(coin);

                state_out <= '1';

            else

                pepsi_cola <= '0';

                exchange <= '0';

            end if;

        when money =>

```

```
if (current_money + value(coin) = 20 and req = '1') then

    state <= no_money;

    state_out <= '0';

    pepsi_cola <= '1';

    current <= "00000";

    current_money <= "00000";


elsif (current_money + value(coin) < 20 and req = '1') then

    state <= no_money;

    state_out <= '0';

    exchange <= '1';

    current <= "00000";

    current_money <= "00000";
```

```

elsif (current_money + value(coin) > 20) then

    state <= no_money;

    state_out <= '0';

    exchange <= '1';

    current <= "00000";

    current_money <= "00000";

else

    current <= current_money + value(coin);

    current_money <= current_money + value(coin);

end if;

end case;

end if;

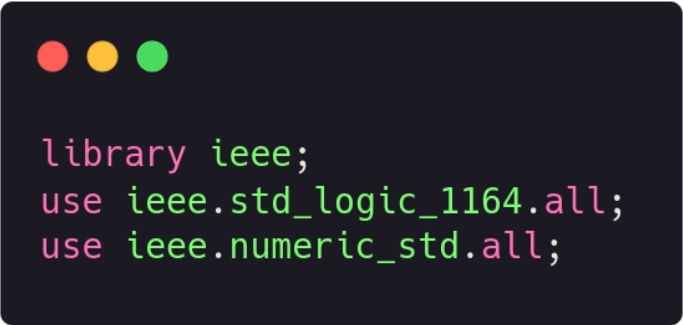
end process;

end behaviour;

```

Importação das Bibliotecas


Agora veremos a implementação do nosso sistema de venda automática que aceita moeda e entrega uma lata de refrigerante Pepsi quando o valor esperado é inserido.



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

A biblioteca **'IEEE'** é importante para importar módulos, dentro dela **'std_logic_1164'** serve para fornecer os tipos de dados para a lógica digital, enquanto a **'numeric_std'** fornece tipos de dados para números, como o **'unsigned'** por exemplo.

Declaração da Entidade



```
entity pepsi is
  port (
    ...
  );
end pepsi;
```

‘**pepsi**’: define a interface da máquina de refrigerante.

Portas de Entrada

- ‘**clk**’ e ‘**clk_placa**’: são o botão do clock e o clock interno à placa.
- ‘**reset**’: sinal para reiniciar o sistema.
- ‘**req**’: botão de solicitação para entregar o refrigerante.
- ‘**coin**’: vetor de bits representando o valor da moeda inserida.

Portas de Saída

- ‘**current**’: valor acumulado (5 bits).

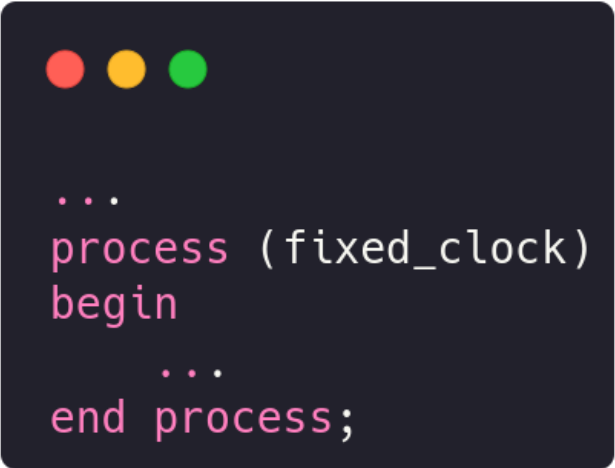
- **'pepsi_cola'**: sinal para entregar o refrigerante.
- **'exchange'**: sinal para o troco.
- **'state_out'**: indica o estado atual do sistema (dinheiro inserido ou não).

Arquitetura e Declaração de Componentes

```
architecture behaviour of pepsi is
    ...
    component Debouncing_Button_VHDL is ...
    end component;
    ...
begin
    ...
end behaviour;
```

A arquitetura behaviour do Elevador define o comportamento interno do sistema de elevador. Já o componente **'Debouncing_Button_VHDL'** é importado de outro arquivo e serve para para debouncing, que exerce a função de eliminar ruídos de um botão. Útil para estabilizar o sinal do clock.

Implementação da Lógica do Sistema



```
...  
process (fixed_clock)  
begin  
    ...  
end process;
```

Como comentado anteriormente, a instância do debouncer serve para integrá-lo dentro do circuito

Processo Principal


Controla a lógica de operação da máquina de refrigerante

- Reset: quando ativado, reinicia o estado da máquina para **'no_money'** e zera todas as saídas e contadores.

Estados da Máquina

- **'no_money'**: espera pela inserção de moeda.
- **'money'**: processa a inserção de moeda e verifica se o valor inserido é o suficiente para liberar o refrigerante e o botão for pressionado, sendo **'pepsi_cola'** <= '1', ou se o valor excede o necessário / botão pressionado antes do valor correto, **'exchange'** <= '1', retornando para o estado **'no_money'** após a operação.

Função de Valor da Moeda



```
function value (coin_in : in bit_vector(2 downto 0)) ...  
end function value;
```

Esta função calcula o valor da moeda inserida com base no vetor de bits **'coin'**.

Saídas e Transições de Estado

O sistema atualiza suas saídas - **'pepsi_cola'**, **'exchange'**, **'current'**, e **'state_out'** - dependendo do estado atual em que se encontra e do valor das moedas inseridas. Por exemplo, quando uma moeda é inserida, essas saídas refletem o novo total acumulado e o estado atual da máquina, seja aguardando mais moedas, pronto para dispensar uma Pepsi ou para entregar troco. Além disso, as transições de estado dentro da máquina são essenciais para o seu funcionamento. A máquina alterna entre os estados **'no_money'** e **'money'**, baseando-se na inserção de moedas e na solicitação de entrega de uma Pepsi. Quando uma moeda é inserida no estado **'no_money'**, a máquina transita para o

estado **'money'**, indicando que começou a acumular valor. Da mesma forma, após a entrega da Pepsi, ou quando o valor inserido excede o necessário, a máquina retorna ao estado **'no_money'**, indicando que está pronta para iniciar uma nova transação. Essas transições garantem que a máquina opere de maneira ordenada e eficiente, processando as solicitações de compra e gerenciando o fluxo de operações de venda.

Resumo

Nossa máquina de venda de refrigerante aceita moedas de diferentes valores, acumula o total inserido, e libera uma lata de Pepsi quando o valor necessário é alcançado ou excedido. A máquina de estados com **'no_money'** e **'money'** gerencia o fluxo do processo, enquanto o sistema de **'debouncing'** garante a estabilidade do sinal de **'clock'**.

Conclusões Finais

No desenvolvimento dos sistemas do elevador e máquina de refrigerante, usamos a linguagem VHDL com o Quartus pela primeira vez. Percebemos que programar via código facilitou a criação dos projetos, tornando-os mais claros e fáceis de escrever. No elevador, implementamos na prática a lógica sequencial e o gerenciamento de estados. Além disso, na máquina de refrigerante, foi possível perceber a diferença entre máquinas de Moore e Mealy.