Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

Relatório do Experimento 7

Autor: Henrique Morcelles Salum

Matrícula: 232003008

ENE0040 - Laboratório de Sistemas Digitais - Turma 07

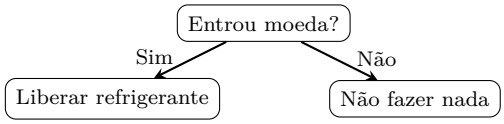
Sumário

1 Introdução

Este experimento é composto de apenas uma tarefa: implementar um moedeiro por meio de uma máquina de estados finita. Esse moedeiro é utilizado no contexto de uma máquina de refrigerantes simplificada, que contém apenas um refrigerante (ou apenas refrigerantes de mesmo preço, que são escolhidos por uma lógica exterior à máquina).

A função do moedeiro é simples: informar ao resto do sistema quando deve ser liberado o refrigerante e/ou devolvidas as moedas (como troco ou em virtude de desistência do cliente). A saída R representa que o refrigerante deve ser liberado e as saídas T_1 e T_0 representam, respectivamente, que deve ser devolvido R\$0,50 e R\$0,25.

Note que, caso o preço de um refrigerante fosse R\$1,00 e apenas houvesse moedas de R\$1,00, esse poderia ser um circuito combinacional com a seguinte lógica:



Em casos um pouco mais complexos, em que houvesse, por exemplo, apenas moedas de valor maior do que R\$1,00, ainda seria possível formular um sistema combinacional, que simplesmente devolveria o troco e liberaria o refrigerante. No nosso caso, porém, as moedas de valor menor do que R\$1,00 (R\$0,25 ou R\$0,50) impõem uma restrição: é necessária memória! É preciso armazenar o dinheiro que já foi inserido na máquina e, mais do que isso, a lógica do sistema muda em função desse valor acumulado.

Um diagrama de estados simplificados que apresenta o comportamento do moedeiro está exibido a seguir. Para melhor apresentação, foram omitidas as arestas que levam dos estados em que há saídas não nulas a estados diferentes do inicial.

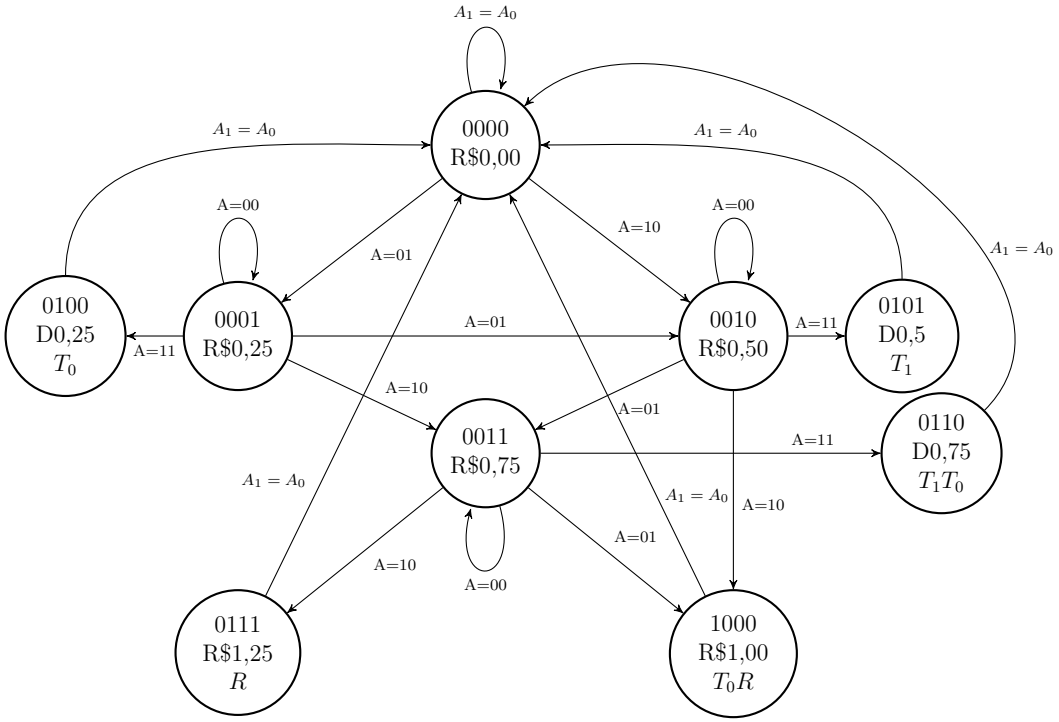


Figura 1: Diagrama de Estados do Moedeiro

A partir do diagrama de estados acima, pode-se montar a tabela a seguir. Ela condensa as tabelas de transição de estados e de saídas.

Estado Atual				**Próximo Estado (Q^*)**				**Saídas**		
Q_3	Q_2	Q_1	Q_0	$\overline{A_1}\,\overline{A_0}$	$\overline{A_1}A_0$	$A_1\overline{A_0}$	A_1A_0	R	T_1	T_0
0	0	0	0	0000	0001	0010	0000	0	0	0
0	0	0	1	0001	0010	0011	0100	0	0	0
0	0	1	0	0010	0011	0111	0100	0	0	0
0	0	1	1	0011	0111	1000	0110	0	0	0
0	1	0	0	0000	0001	0010	0000	0	0	1
0	1	0	1	0000	0001	0010	0000	0	1	0
0	1	1	0	0000	0001	0010	0000	0	1	1
0	1	1	1	0000	0001	0010	0000	1	0	0
1	0	0	0	0000	0001	0010	0000	1	0	1

Tabela 1: Tabela de transição de estado e saídas do moedeiro

Partindo dessa tabela é fácil formular um código em VHDL que implemente as lógicas de transição de estados e de saída, especialmente fazendo-se bom uso das estruturas de alto nível de abstração como *when-else*.

Note que muitos estados compartilham a lógica de próximo estado (as células correspondentes ao próximo estado são iguais em uma linha). Isso indica que uma máquina do tipo *Mealy* seria mais eficiente no sentido de economizar estados (e bits para representação de estados); os estados que compartilham a lógica de próximo estado poderiam ser unificados e a saída seria determinada a partir da entrada. O roteiro do experimento, impõe que máquina seja do tipo Moore.

2 Códigos

Para melhor organização, o moedeiro foi implementado como um *top-module*. Ele instancia três subsistemas: um registrador de deslocamento de 4 bits (feito no último experimento), um subcircuito que implementa a lógica de próximo estado e outro que implementa a lógica de saída.

Além disso, foi feito um *testbench* que alcança diversos casos de interesse. Como é um circuito sequencial, não bastaria gerar todas as combinações das entradas para um teste completo, por isso, a decisão foi fazer um *testbench* que só alcança alguns casos que ilustram o funcionamento.

2.1 Lógica de Saída

A lógica de saída é, basicamente, uma tradução direta da Tabela 1.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity LogicaSaidas is
5      port (
6          estado: in  std_logic_vector(3 downto 0);
7          T:      out std_logic_vector(1 downto 0);
8          R:      out std_logic
9      );
10 end entity LogicaSaidas;
11
12 architecture behavioral of LogicaSaidas is
13 begin
14     R <= '1' when estado = "0111" or estado = "1000" else '0';
15
16     T <= "01" when estado = "0100" or estado = "1000" else
17          "10" when estado = "0101" else
18          "00";
19 end architecture behavioral;
```

Código 1: Lógica de Saída

2.2 Lógica do Próximo Estado

A lógica de próximo estado é, também, quase uma tradução direta da Tabela 1. Isso ficaria mais claro se não fossem usados *shifts*, apenas *loads*, mas o uso de *shifts* é feito para melhor aproveitar o registrador de deslocamento instanciado.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity LogicaProxEstado is
5      port (
6          estado_atual:   in  std_logic_vector(3 downto 0);
7          A:              in  std_logic_vector(1 downto 0);
8          load:           out std_logic;
```

```
        data:           out std_logic_vector(3 downto 0);
        direction:      out std_logic;
        left:           out std_logic;
        right:          out std_logic
    );
end entity LogicaProxEstado;

architecture behavioral of LogicaProxEstado is
begin
    process(estado_atual, A)
    begin
        case estado_atual is
            when "0000" =>  -- S0
                case A is
                    when "01" =>
                        load <= '0';
                        left <= '1';
                        direction <= '0';   -- S1 (0001)
                    when "10" =>
                        load <= '1';
                        data <= "0010";     -- S2
                    when others =>
                        load <= '1';
                        data <= "0000";     -- S0
                end case;
            when "0001" =>  -- S1
                case A is
                    when "01" =>
                        load <= '0';
                        direction <= '0';
                        left <= '0';        -- S2 (0010)
                    when "10" =>
                        load <= '0';
                        direction <= '0';
                        left <= '1';        -- S3
                    when "11" =>
                        load <= '1';
                        data <= "0100";     -- S6
                    when others =>
                        load <= '1';
                        data <= "0001";     -- S1
                end case;
            when "0010" =>  -- S2
                case A is
                    when "01" =>
                        load <= '1';
                        data <= "0011";      -- S3
                    when "10" =>
                        load <= '1';
                        data <= "0111";      -- S4
                    when "11" =>
                        load <= '0';
                        direction <= '0';
                        left <= '1';        -- S7 (0101)
                    when others =>
                        load <= '1';
                        data <= "0010";    -- S2
                end case;
            when "0011" =>  -- S3
                case A is
```

```
                    when "01" =>
                        load <= '0';
                        direction <= '0';
                        left <= '1';    -- S4 (0111)
                    when "10" =>
                        load <= '1';
                        data <= "1000"; -- S5
                    when "11" =>
                        load <= '0';
                        direction <= '0';
                        left <= '0'; -- S8 (0110)
                    when others =>
                        load <= '1';
                        data <= "0011"; -- S3
                end case;
            when others =>  -- S4, S5, S6, S7, S8
                case A is
                    when "01" =>
                        load <= '1';
                        data <= "0001";      -- S1
                    when "10" =>
                        load <= '1';
                        data <= "0010";      -- S2
                    when others =>
                        load <= '1';
                        data <= "0000";      -- S0
                end case;
        end case;
    end process;
end architecture behavioral;
```

Código 2: Lógica de Próximo Estado

2.3 Máquina de Refrigerantes

A máquina de refrigerantes (moedeiro) é, como já explicado, uma estrutura *top-module*. Nela, instanciamos a lógica de saída, lógica de próximo estado e um registrador de deslocamento e conectamos tudo. Um detalhe é que há um processo extra para o *reset*. Ele foi necessário para que o estado não fosse 'U' (undefined) até a primeira batida do clock: no início da execução, *reset* é '1' para que o estado inicial seja definido assincronamente como $Q =$ "0000".

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MaquinaRefrigerante is
    port (
        CLK:    in  std_logic;
        A:      in  std_logic_vector(1 downto 0);
        T:      out std_logic_vector(1 downto 0);
        R:      out std_logic
    );
end entity MaquinaRefrigerante;

architecture structural of MaquinaRefrigerante is
```

```
    component Registrador4Bits is
        port (
            clock:      in std_logic;
            reset:      in std_logic;
            load:       in std_logic;
            data:       in std_logic_vector(3 downto 0);
            direction:  in std_logic;
            left:       in std_logic;
            right:      in std_logic;
            Q:          out std_logic_vector(3 downto 0)
        );
    end component;

    component LogicaProxEstado is
        port (
            estado_atual:   in  std_logic_vector(3 downto 0);
            A:              in  std_logic_vector(1 downto 0);
            load:           out std_logic;
            data:           out std_logic_vector(3 downto 0);
            direction:      out std_logic;
            left:           out std_logic;
            right:          out std_logic
        );
    end component LogicaProxEstado;

    component LogicaSaidas is
        port (
            estado: in  std_logic_vector(3 downto 0);
            T:      out std_logic_vector(1 downto 0);
            R:      out std_logic
        );
    end component;

    signal estado_atual: std_logic_vector(3 downto 0) := (others => '0');
    signal data: std_logic_vector(3 downto 0);
    signal reset: std_logic := '1';
    signal load, direction, left, right: std_logic;
begin
    registrador: Registrador4Bits
        port map (
            clock       => CLK,
            reset       => reset,
            load        => load,
            data        => data,
            direction   => direction,
            left        => left,
            right       => right,
            Q           => estado_atual
        );

    proximo_estado: LogicaProxEstado
        port map (
            estado_atual    => estado_atual,
            A               => A,
            load            => load,
            data            => data,
            direction       => direction,
            left            => left,
            right           => right
        );
```

```
74
75      logica_saidas: LogicaSaidas
76          port map (
77              estado      => estado_atual,
78              T           => T,
79              R           => R
80          );
81
82      process
83      begin
84          wait for 1 ns;
85          reset <= '0';
86          wait;
87      end process;
88  end architecture structural;
```

Código 3: Máquina de Refrigerantes

2.4 *Testbench*

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity tb_MaquinaRefrigerante is
5   end entity tb_MaquinaRefrigerante;
6
7   architecture testbench of tb_MaquinaRefrigerante is
8       component MaquinaRefrigerante is
9           port(
10              CLK : in  std_logic;
11              A   : in  std_logic_vector(1 downto 0);
12              T   : out std_logic_vector(1 downto 0);
13              R   : out std_logic
14          );
15      end component;
16      signal CLK : std_logic := '0';
17      signal A   : std_logic_vector(1 downto 0) := "00";
18      signal T   : std_logic_vector(1 downto 0);
19      signal R   : std_logic;
20
21      constant CLK_PERIOD : time := 20 ns;
22  begin
23      clk_proc: process
24      begin
25          CLK <= '0'; wait for CLK_PERIOD/2;
26          CLK <= '1'; wait for CLK_PERIOD/2;
27      end process clk_proc;
28
29      DUT: MaquinaRefrigerante
30          port map(
31              CLK => CLK,
32              A   => A,
33              T   => T,
34              R   => R
35          );
36
37      stim_proc: process
```

```
    begin
        wait for CLK_PERIOD;
        report "01) Inserção até R$1,00";
        -- 0.25 + 0.25 + 0.50 = 1.00
        A <= "01"; wait for CLK_PERIOD;
        A <= "01"; wait for CLK_PERIOD;
        A <= "10"; wait for CLK_PERIOD;
        A <= "00"; wait for CLK_PERIOD * 2;

        report "02) Inserção ultrapassando para R$1,25";
        -- 0.50 + 0.50 = libera, depois +0.25 gera troco
        A <= "10"; wait for CLK_PERIOD;
        A <= "10"; wait for CLK_PERIOD;
        A <= "01"; wait for CLK_PERIOD;
        A <= "00"; wait for CLK_PERIOD * 2;

        report "03) Cancelamento em cada valor armazenado";
        -- Cancelar em S1 (0.25)
        A <= "01"; wait for CLK_PERIOD;
        A <= "11"; wait for CLK_PERIOD;
        wait for CLK_PERIOD;
        -- Cancelar em S2 (0.50)
        A <= "10"; wait for CLK_PERIOD;
        A <= "11"; wait for CLK_PERIOD;
        wait for CLK_PERIOD;
        -- Cancelar em S3 (0.75): 25+50
        A <= "01"; wait for CLK_PERIOD;
        A <= "10"; wait for CLK_PERIOD;
        A <= "11"; wait for CLK_PERIOD;
        wait for CLK_PERIOD;

        report "04) Inserção após liberação ou cancelamento";
        -- Após troco (exemplo: máquina idle volta S0)
        A <= "01"; wait for CLK_PERIOD;
        A <= "01"; wait for CLK_PERIOD;
        A <= "10"; wait for CLK_PERIOD;
        wait for CLK_PERIOD;
        -- Novo ciclo completo até liberação
        A <= "10"; wait for CLK_PERIOD;
        A <= "10"; wait for CLK_PERIOD;
        A <= "10"; wait for CLK_PERIOD;
        wait for CLK_PERIOD * 2;

        report "Fim da simulação";
        wait;
    end process stim_proc;
end architecture testbench;
```

Código 4: Testbench da máquina de refrigerantes

3 Compilação

Após escrever os códigos, é necessário compilá-los pelo ModelSim para que se possa simular os sistemas digitais discutidos. Caso a compilação tenha sucesso, sabemos que não houve erros nos códigos apresentados, mas ainda não podemos afirmar que a lógica para implementar os circuitos está correta; isso será analisado nas próximas seções.

A seguir, está a mensagem de compilação dos códigos apresentados acima. Sem nenhum erro, como pode ser visto no terminal no canto inferior da figura. Os códigos do flip-flop e do multiplexador 4 para 1 são necessários para que o do registrador compile corretamente.

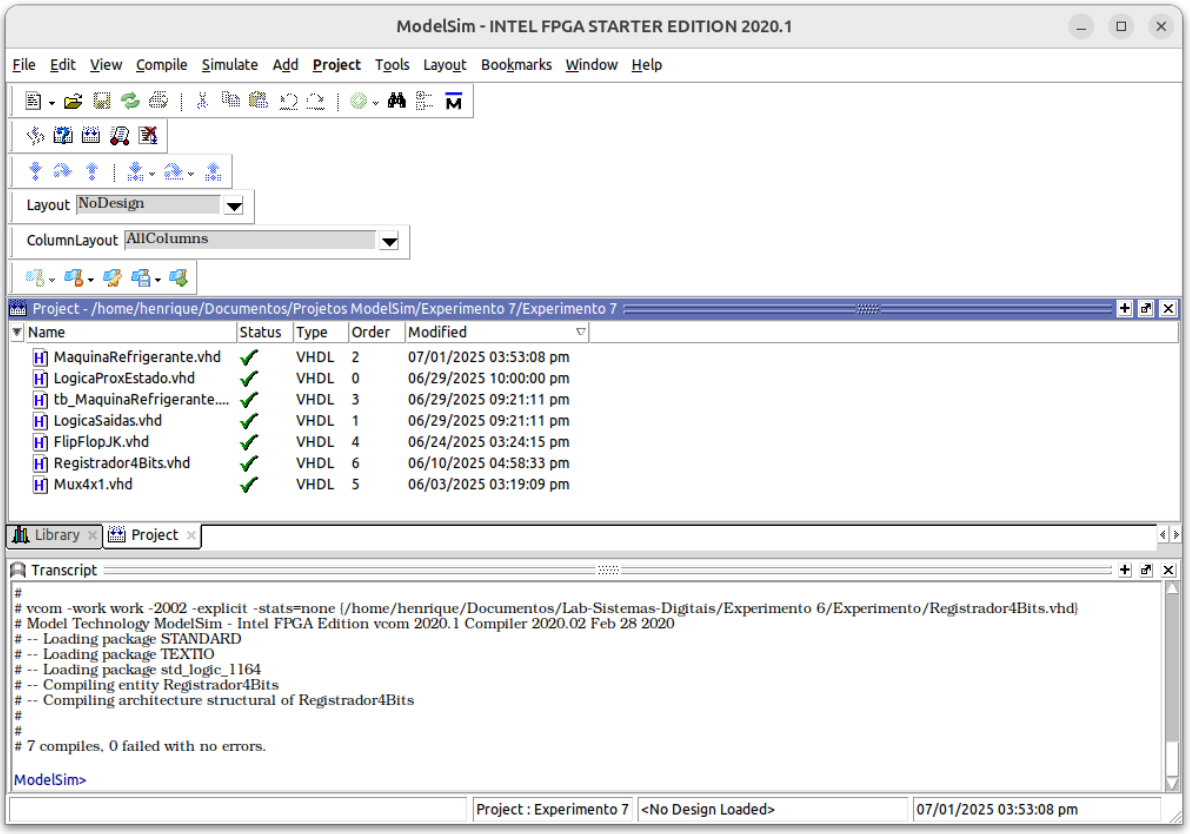


Figura 2: Compilação de todos os códigos apresentados

4 Simulação

O gráfico de forma de onda gerado pelo ModelSim ao simular o *top-module* está exibido abaixo. Foram marcados com cursores os momento de interesse para a análise.

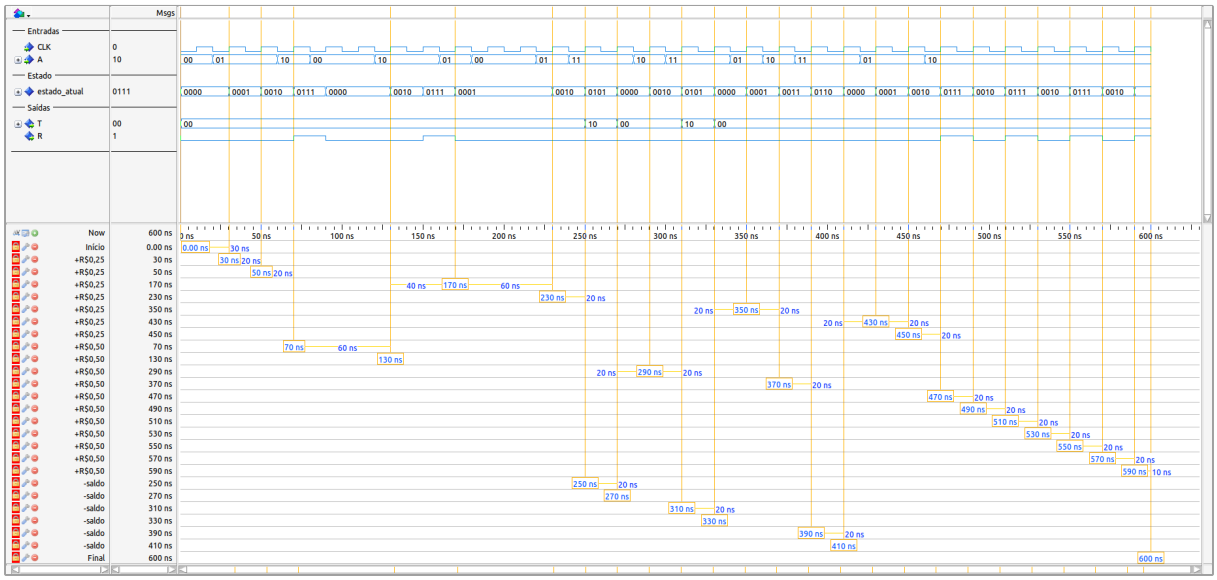


Figura 3: Simulação em forma de onda binária do moedeiro

5 Análise

Basta olhar o valor das saídas em Figura 3 que percebe-se o correto funcionamento do código apresentado. A saída R, de fato, é 1 apenas quando a máquina atinge ou ultrapassa R\$1,00 e as saídas T_1 e T_0 são ligadas apenas se o valor acumulado atinge R\$1,25 ou o cliente pede o dinheiro de volta (A = “11”).

6 Conclusão

Nesse experimento, pela primeira vez, implementamos uma máquina de estados. Máquinas de estados são essenciais para a eletrônica digital; computadores nada mais são do que enormes associações de máquinas de estado de alta complexidade. A importância desse experimento reside nisso.