Fernanda Silva Bucheri RA: 135529

Projeto de um contador crescente/decrescente utilizando uma máquina de estados finitos de Moore

São José dos Campos - Brasil

Fevereiro de 2021

Fernanda Silva Bucheri RA: 135529

Projeto de um contador crescente/decrescente utilizando uma máquina de estados finitos de Moore

Trabalho apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Circuitos Digitais.

Docente: Profª. Drª.: Fernanda Quelho Rossi

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Fevereiro de 2021

Resumo

Este projeto visa implementar, utilizando a linguagem de hardware *Verilog*, uma máquina de estados do tipo Moore comportando-se como um contador capaz de realizar uma contagem crescente ou decrescente da seguinte sequência numérica: 9 - 4 - 6 - 5 - 8 - 2 - 1 - 0 - 5. O código em *Verilog* foi elaborado na plataforma *Verilog IDE for DE2-115*. Posteriormente, foram realizadas simulações no software *Intel Quartus Prime*, onde foram geradas as *waveforms* e no Kit remoto de Desenvolvimento FPGA *(Field-programmable gate array)* DE2-115 da ALTERA, onde houve a implementação real do projeto: de acordo com o comando dado pelo usuário, será exibido em um display de 7 segmentos a contagem crescente, decrescente, irá manter um número no display ou apagá-lo.

Palavras-chaves: Circuitos Digitais, Máquina de Estados, Moore, Verilog.

Lista de ilustrações

Figura 1 – Palavras reservadas na linguagem Verilog. 14
Figura 2 – Diagrama em bloco das máquinas de estado. 18
Figura 3 – Placa DE2-115 19
Figura 4 – Displays de sete segmentos presentes na placa. 20
Figura 5 – Tabela-verdade para o processo de contagem. 21
Figura 6 – Código em Verilog do contador. 22
Figura 7 – Código em Verilog do divisor de frequência. 23
Figura 8 – Código em Verilog para o decodificador BCD para display de 7 segmentos. 25
Figura 9 – Diagrama de estados. 26
Figura 10 – Módulo *Moore*: Declaração das entradas, saídas e definição dos parâmetros. 27
Figura 11 – Módulo *Moore*: Declaração do bloco *always* e o caso do estado atual ser A, definindo o próximo estado de acordo com as entradas Up e Down. . 28
Figura 12 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja B. 28
Figura 13 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja C. 29
Figura 14 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja D. 29
Figura 15 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja E. 30
Figura 16 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja F. 30
Figura 17 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja G. 31
Figura 18 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja H. 31
Figura 19 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja I. 32
Figura 20 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja J e o encerramento do bloco *always*. 32
Figura 21 – Módulo *Moore*: Declaração dos dois blocos *always*: o bloco responsável por aplicar o clock (e o reset quando solicitado) e o bloco responsável por enviar para a saída do módulo *Moore* o valor do estado atual da máquina. 33

Figura 22 – Waveform da simulação que utiliza as entradas Up = 1, Down = 0 (ou seja, uma contagem crescente) e Reset ativado aos 300 ns e se mantendo por 300 ns. 41
Figura 23 – Waveform da simulação que utiliza as entradas Up = 0, Down = 1 (ou seja, uma contagem decrescente) e Reset ativado aos 300 ns e se mantendo por 300 ns. 42
Figura 24 – Waveform da simulação que utiliza as entradas Up = 1 e Down ativada aos 10 ns e se mantendo por 400 ns (o display fica apagado, após isso inicia uma contagem crescente). 42
Figura 25 – Waveform da simulação que utiliza as entradas Reset = 0, Up é ativada a cada 50 ns e se mantém ativa por 50 ns e Down é ativada a cada 100 ns e se mantém ativa por 100 ns. 43
Figura 26 – Waveform da simulação onde está ocorrendo uma contagem crescente, aplica-se o reset e, após isso, ocorre uma contagem decrescente. 44

Lista de tabelas

Tabela 1 – Portas lógicas em Verilog. 15
Tabela 2 – Sintaxe dos operadores. 16
Tabela 3 – Tabela-verdade do decodificador BCD para display do tipo ânodo comum. 24
Tabela 4 – Representação dos estados em número binário e a letra atribuída a cada um. 25
Tabela 5 – Tabela-verdade dos circuitos combinacionais de entrada e saída. 26

Sumário

1 INTRODUÇÃO 9

2 OBJETIVOS 11
2.1 Objetivos Gerais 11
2.2 Objetivos específicos 11

3 FUNDAMENTAÇÃO TEÓRICA 13
3.1 Linguagem de descrição hardware (HDL) 13
3.2 Verilog 13
3.2.1 Palavras reservadas e para que servem 13
3.2.2 Portas lógicas 15
3.2.3 Operadores 15
3.2.4 Sub-circuitos 16
3.2.5 Atribuições *Bloqueante* e *Não Bloqueante*. 17
3.3 Máquina de estados de Moore 17
3.4 Placa FPGA, Display de 7 segmentos e Conversor BCD 18

4 DESENVOLVIMENTO 21
4.1 Contador completo 21
4.2 Divisor de frequência 22
4.3 Decodificador BCD para Display de 7 segmentos 23
4.4 Diagrama de estados e tabela verdade 25
4.5 Código completo 34

5 RESULTADOS OBTIDOS 41
5.1 Simulação no software Quartus (waveforms) 41
5.2 Simulação no kit FPGA remoto 44

6 CONCLUSÃO 45

REFERÊNCIAS 47

1 Introdução

Circuitos digitais são essenciais na vida moderna. Possuem inúmeras aplicações que facilitam e proporcionam benefícios para a humanidade. Computadores, aparelhos de celular, Blu-ray, equipamentos hospitalares e muitos outros, são exemplos de aparelhos que baseiam parte do seu funcionamento em circuitos digitais. É possível desenvolver um projeto de circuito digital através de um esquemático ou utilizando uma linguagem de descrição de hardware *(Hardware Description Language - HDL)*, uma ferramenta extremamente útil, principalmente para desenvolvimento de circuitos mais complexos.

O presente trabalho visa abordar mais profundamente diferentes conceitos que englobam circuitos digitais com uma aplicação que consiste em elaborar uma máquina de estados finita *(Finite State Machine)*, mais especificamente, uma máquina de Moore, utilizando a linguagem de hardware *Verilog*.

2 Objetivos

2.1 Objetivos Gerais

Desenvolver uma máquina de estados de Moore que possui 10 estados com quatro funções diferentes, sendo elas: agir como um contador crescente ou decrescente da seguinte sequência: 9-4-6-5-8-2-1-0-5, manter seu estado atual ou não mostrar nada no display. O tempo de transição de um estado para o outro deve ser de 1 segundo, aproximadamente.

2.2 Objetivos específicos

I Criar um diagrama de estados com todos os estados que serão implementados na máquina de Moore;

II Elaborar as tabelas verdades com as funções de próximo estado e de saída;

III Utilizando a linguagem de descrição de hardware Verilog:

- i Desenvolver o circuito das funções de próximo estado e de saída;
- ii Desenvolver o circuito divisor de frequência para gerar um clock de 1 segundo a partir do clock interno de 50 MHz do kit FPGA;
- iii Desenvolver o circuito decodificador BCD para display de sete segmentos, com o propósito de exibir a contagem no display HEX[4] do kit FPGA;
- iv Implementar todo o circuito da máquina de Moore;

IV Por fim, implementar o circuito no kit FPGA;

3 Fundamentação teórica

A primeira descrição de um sistema numérico binário foi apresentada por volta do século III a.C., pelo matemático indiano Pingala. Já Gottfried Leibniz, publicou no século XVIII seu artigo "Explication de l'Arithmétique Binaire", apresentando o sistema numérico binário moderno, utilizando 0 e 1, tal como nos dias de hoje. George Boole, em 1854, publicou um artigo detalhando um sistema lógico, que ficou conhecido como Álgebra Booleana. Em 1937, foi produzida uma tese por Claude Shannon intitulada "A Symbolic Analysis of Relay and Switching Circuits", que implementava álgebra booleana e aritmética binária utilizando circuitos elétricos pela primeira vez, dando início assim ao projeto de circuitos digitais.

É possível implementar um circuito digital desenvolvendo um esquemático ou uma linguagem de descrição hardware.

3.1 Linguagem de descrição hardware (HDL)

Servem para modelar o comportamento e a estrutura de um hardware e possuem como vantagem proporcionar uma maior facilidade tanto para implementação das etapas do projeto, quanto para eventuais manutenções ou melhoramentos. Neste projeto será utilizada a linguagem *Verilog*.

3.2 Verilog

Desenvolvida entre 1983 e 1984, pela empresa *Gateway Design Automation*, verilog é uma linguagem amplamente utilizada para descrever sistemas digitais. Sua padronização atual é a IEEE 1364-2005. Possui uma sintaxe parecida com a linguagem de programação C, utilizando declarações de constantes, bibliotecas, módulos, entre outros.

3.2.1 Palavras reservadas e para que servem

A linguagem Verilog possui palavras reservadas (keywords) que indicam comandos. Estas estão indicadas na figura [1].

Figura 1 – Palavras reservadas na linguagem Verilog.

always	end	ifnone	or	rpmos	tranif1
and	endcase	initial	output	rtran	tri
assign	endmodule	inout	parameter	rtranif0	tri0
begin	endfunction	input	pmos	rtranif1	tri1
buf	endprimitive	integer	posedge	scalared	triand
bufif0	endspecify	join	primitive	small	trior
bufif1	endtable	large	pull0	specify	trireg
case	endtask	macromodule	pull1	specparam	vectored
casex	event	medium	pullup	strong0	wait
casez	for	module	pulldown	strong1	wand
cmos	force	nand	rcmos	supply0	weak0
deassign	forever	negedge	real	supply1	weak1
default	for	nmos	realtime	table	while
defparam	function	nor	reg	task	wire
disable	highz0	not	release	time	wor
edge	highz1	notif0	repeat	tran	xnor
else	if	notif1	rnmos	tranif0	xor

Fonte: CSIT Laboratory Web Site[1]

Abordaremos os principais utilizados neste projeto:

- *always*: utiliza-se este comando para realizar instruções quando houver mudança em alguma entrada ou saída pré-determinada;

- *assign*: sua função é atribuir um valor para uma entrada ou saída.

- *begin/end*: são utilizados para marcar o início e o fim de uma série de comandos.

- *case/endcase*: Assim como em outras linguagens (como C), esse comando verifica se um valor dado como parâmetro atende alguma das condições especificadas. Inicia-se o bloco com a palavra *case* e encerra-se o bloco com *endcase*.

[1] Disponível em: <http://www.csit-sun.pub.ro/courses/Masterat/Materiale_Suplimentare/Xilinx%20Synthesis%20Technology/toolbox.xilinx.com/docsan/xilinx4/data/docs/xst/verilog10.html>;. Acesso em Fev.2021.

- *default*: Utilizada junto com o case quando se deseja especificar o que deve ser feito se o parâmetro não atender nenhuma das condições declaradas.

- *module/endmodule*: Indica o início e o fim de um módulo, sendo um comando fundamental nesta linguagem, já que qualquer implementação em Verilog deve ser feita dentro de um módulo.

- *Input/output*: Utiliza-se para declarar as entradas e saídas, respectivamente.

- *reg*: utiliza-se em alguns casos onde, basicamente, é preciso armazenar um valor para ser utilizado posteriormente. Uma observação é que é necessário declarar a variável utilizada no bloco always como reg.

- *wire*: são como os "fios"que ligam dois pontos do circuito, ou seja, representam uma conexão ou nó.

3.2.2 Portas lógicas

Podemos representar uma porta lógica escrevendo seu nome ou símbolo:

Tabela 1 – Portas lógicas em Verilog.

Nome	**Símbolo**
AND	&
OR	|
NOT	∼
NAND	∼&
NOR	∼|
XOR	∧
XNOR	∼ ∧

Fonte: elaborado pela autora.

No entanto, para utilizar o símbolo é necessário instanciar o módulo utilizando *assign*.

3.2.3 Operadores

A tabela 2 contém todos os operadores utilizados em *Verilog*.

Tabela 2 – Sintaxe dos operadores.

Tipo	**Símbolo**
Aritmético	+ - * /
Bitwise	& | ^ ~
Lógico	! && ||
Relacional	<> == >=
Deslocamento	<< >>
Concatenação	{ }

Fonte: elaborado pela autora.

3.2.4 Sub-circuitos

Assim como no esquemático pode se utilizar uma ou mais *black-box*, na linguagem Verilog é possível utilizar sub-circuitos dentro do circuito principal, ou seja, utilizar um (ou mais) módulos dentro de um outro módulo.

Supondo que, de dentro de um módulo principal, se quer chamar um outro módulo, o qual possui uma entrada e uma saída. Seria utilizado o seguinte comando:

module_name instance_name(.port_name(expression), .port_name (expression));

onde:

module_name: é o nome do módulo que deseja-se chamar;

instance_name: atribui-se um nome qualquer;

port_name é o nome da porta descrita na declaração do segundo módulo, enquanto que *expression* é o nome da variável que está armazenando a informação que se deseja passar para aquela determinada porta do segundo módulo.

Exemplificando, através de um exemplo disponível no slide da Profª Drª Fernanda Quelho Rossi, temos dois módulos:

module primeiro (A,B,S);

input [2:0] A,B;

output [2:0] S;

wire [2:0] ligar;

segundo TX(.J(A),.K(B), .L(ligar));

endmodule

```
module segundo (J,K,L);
    input [2:0] J,K;
    output [2:0] L;
    //conteúdo do módulo
    endmodule
```

Onde, na linha destacada no módulo intitulado "primeiro", é possível observar a chamada do módulo intitulado "segundo".

3.2.5 Atribuições *Bloqueante* e *Não Bloqueante*.

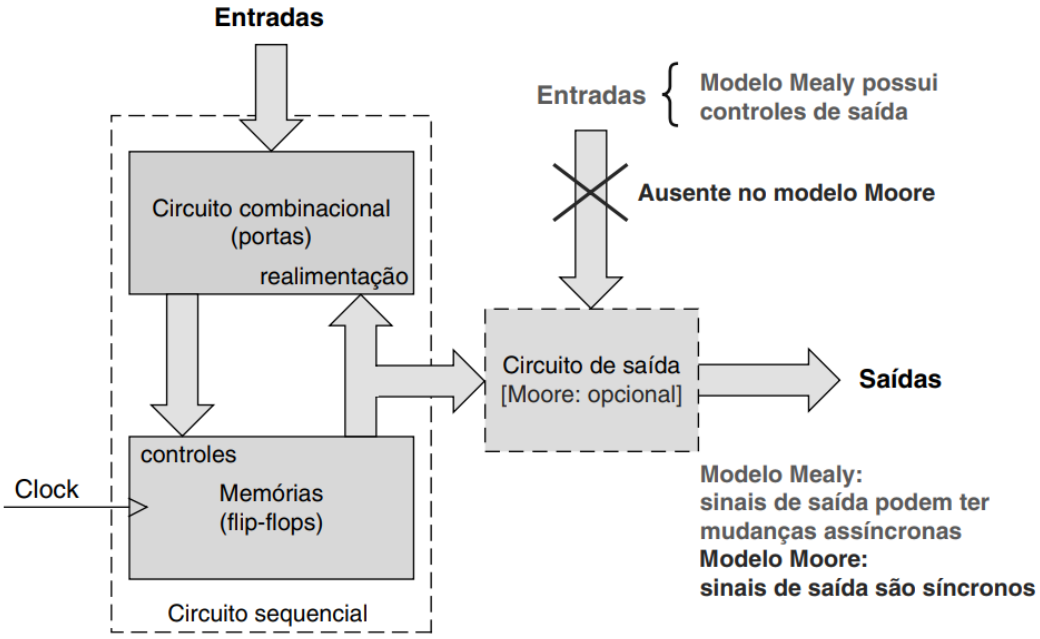
Em um procedimento bloqueante, a variável a recebe um valor através do símbolo de igual (=). Isso significa que a atribuição é executada em sequência no bloco.

Em um procedimento não-bloqueante, a variável a recebe um valor através dos símbolos menor e igual (<=). Isso significa que a atribuição é executada em paralelo no bloco.

3.3 Máquina de estados de Moore

Temos que uma Máquina de Estados Finitos é um circuito sequencial, que possui um número finito de estados (e estes são implementados através de flip flops) pré determinados, onde a máquina fica em apenas um estado por vez, sendo este estado chamado de estado atual. A transição de um estado para o outro ocorre de acordo com um clock. No modelo de Moore (nome em homenagem a Edward F. Moore, professor americano de matemática e ciência da computação responsável por publicar um artigo em 1956, chamado *"Gedanken-experiments on Sequential Machines*, onde apresentou o conceito de máquina de Moore), as saídas dependem apenas do estado atual, diferentemente da máquina de Mealy, onde a saída depende não somente do estado atual, mas também das entradas.

Figura 2 – Diagrama em bloco das máquinas de estado.

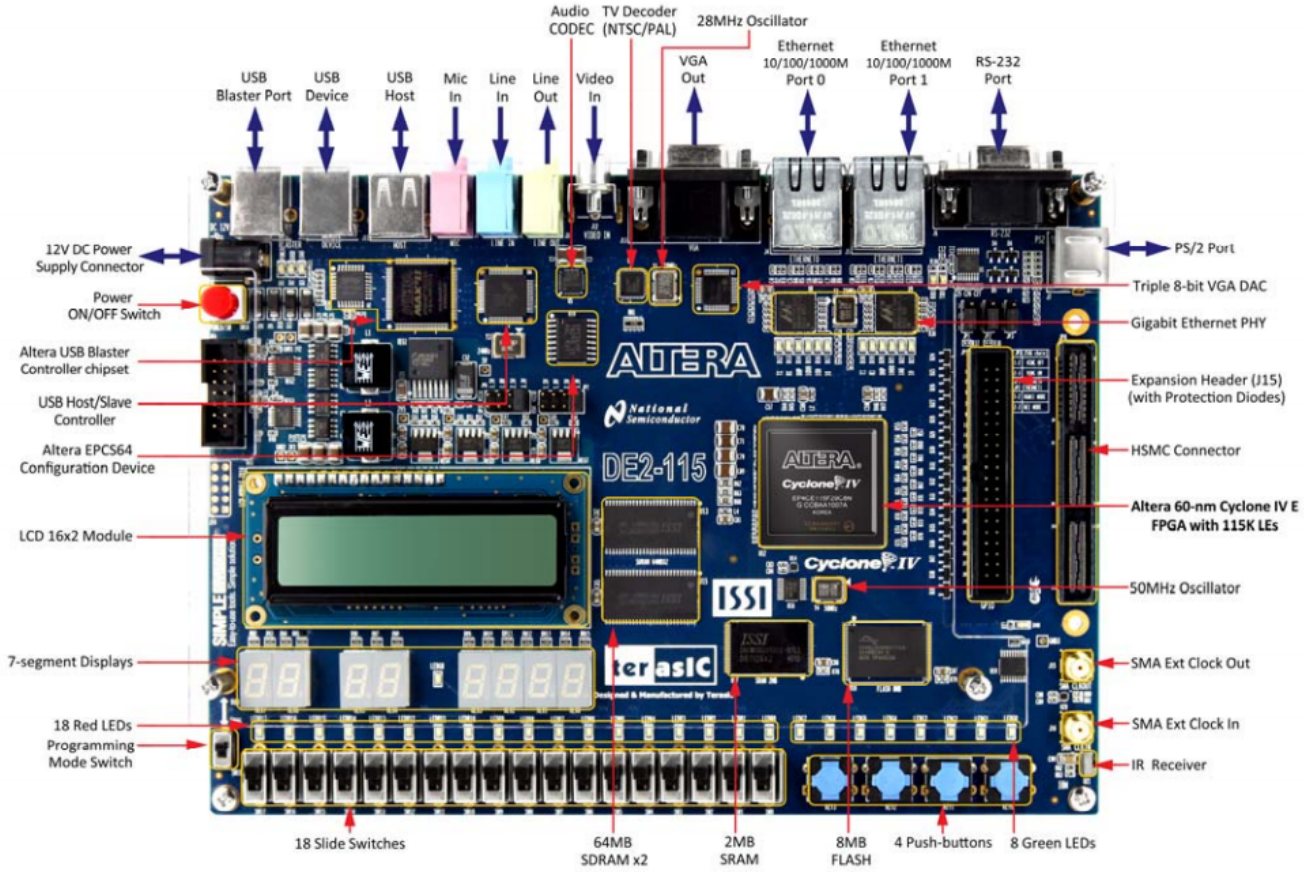


Fonte: Fonte: Tocci, Ronald J.; Widner, Neal S.; Moss, Gregory L. (2011, pág. 367)

3.4 Placa FPGA, Display de 7 segmentos e Conversor BCD

Neste projeto é utilizado o kit de Desenvolvimento FPGA *(Field Programmable Gate Arrays)* DE2-115 da Altera, a qual possui muitos recursos que permitem aos usuários implementar uma ampla gama de circuitos projetados, que vão desde circuitos simples a vários projetos de multimídia, como consta no manual do kit.

Figura 3 – Placa DE2-115

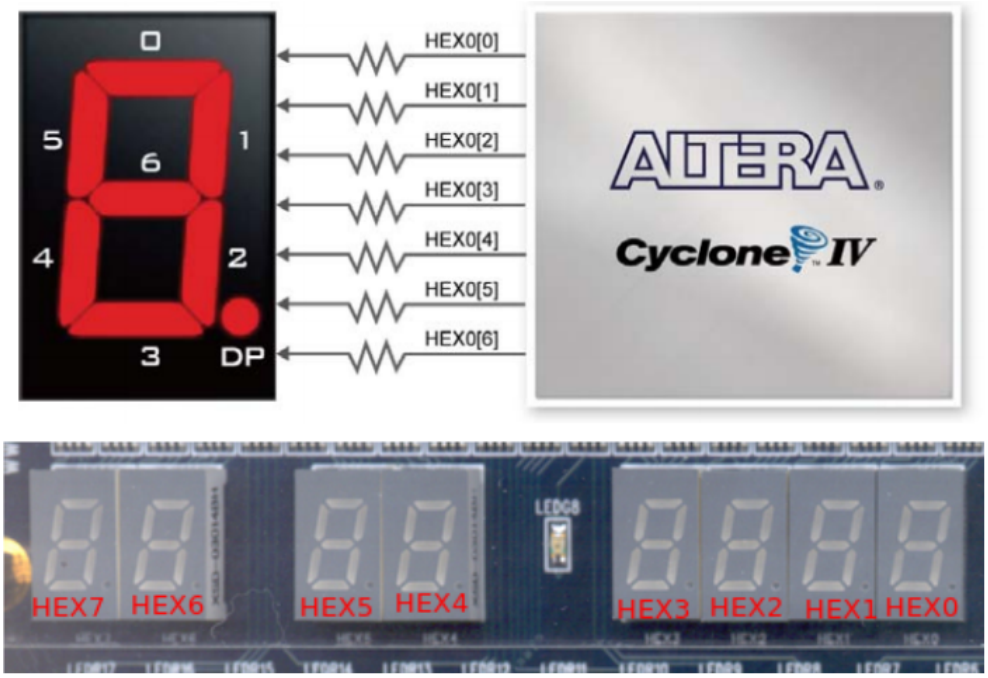


Fonte: Wiki IFSC do Campus São José[2]

Esta placa possui oito displays de sete segmentos. Estes displays são do tipo ânodo comum.

[2] Disponível em: <https://wiki.sj.ifsc.edu.br/index.php/Interfaces_de_entrada_e_sa%C3%ADda_da_DE2-115>. Acesso em Fev.2021.

Figura 4 – Displays de sete segmentos presentes na placa.



Fonte: Wiki IFSC do Campus São José[3]

Neste projeto utilizaremos um conversor BCD para transformar a saída de cada estado, ou seja, um número binário, em um número decimal e exibi-lo no display HEX[4].

[3] Disponível em: <https://wiki.sj.ifsc.edu.br/index.php/Interfaces_de_entrada_e_sa%C3%ADda_da_DE2-115>. Acesso em Fev.2021.

4 Desenvolvimento

4.1 Contador completo

O código em Verilog tem como módulo principal o módulo "Contador". Este possui duas entradas: As chaves do tipo SW e a entrada de Clock (este é o clock interno do Kit FPGA como já dito anteriormente) e a saída: o display de 7 segmentos HEX[4].

Das chaves do tipo SW, a SW[17] será responsável por acionar o RESET (assíncrono com o clock). Além disso, SW[1] representa a chave "UP"e SW[0] corresponde a chave "DOWN", que são as responsáveis pela designação da função que a máquina irá realizar, de acordo com a Figura 5.

Figura 5 – Tabela-verdade para o processo de contagem.

UP (SW1)	DOWN (SW0)	Contagem
0	0	Mantém
0	1	Decrescente
1	0	Crescente
1	1	Display Apagado

Fonte: Slide[1].

Além disso, declarou-se o *wire* de 1 bit intitulado "clk", responsável por receber o resultado do divisor de frequência e enviá-lo para a máquina de Moore e o *wire* "sm"de 4 bits, responsável por receber a saída da máquina de Moore e enviá-la para o conversor BCD.

Em seguida são chamados os blocos *DivFreq*, *Moore* e *BCD*, que serão detalhados mais a frente.

[1] ROSSI. Fernanda Quelho. Máquina de Estados para um Contador Crescente e Decrescente, 2021. 20 slides. Disponível em: <https://grad.sead.unifesp.br/pluginfile.php/238878/mod_resource/content/17/Aula%208-0%20-%20Maquina%20de%20Estados.pdf>. Acesso em: fev. 2021.

Figura 6 – Código em Verilog do contador.

```
module Contador(SW, CLOCK_50, HEX4);
input wire [17:0] SW;
input wire CLOCK_50;
output wire [0:6] HEX4;

wire clk;
wire [3:0] sm;

DivFreq df(.Clock(CLOCK_50), .Saida_DivFreq(clk));
Moore m(.Up(SW[1]), .Down(SW[0]), .Reset(SW[17]), .Clock(clk), .SaidaMoore(sm));
BCD b(.entrada(sm), .segmentos(HEX4));
endmodule
```

Fonte: Elaborado pela autora.

4.2 Divisor de frequência

O divisor de frequência é utilizado para obter um clock com frequência menor do que a frequência original. Neste projeto utiliza-se o clock nativo do kit FPGA. Este possui um clock de 50MHz. Entretanto, é solicitado que, ao exibir a contagem no display, o intervalo entre um número e outro durante a contagem seja de, aproximadamente, 1 segundo.

Sendo o período igual ao inverso da frequência:

$$T = \frac{1}{f}$$

Temos que o período para a frequência de 50MHz é:

$$T = \frac{1}{50 \cdot 10^6}$$

$$T = 20 \cdot 10^{-9}$$

Ou seja, um ciclo de clock demora 20ns, então, para completar 1 segundo, é necessário $50 \cdot 10^6$ ciclos.

A quantidade de bits necessária para uma contagem até $50 \cdot 10^6$ é dada por

$$2^n = 50 \cdot 10^6$$

$$n = \frac{ln(50000000)}{ln(2)}$$

Portanto,

$$n = 25,57542...$$

Logo, conclui-se que é necessário declarar um vetor com 26 posições (denominado "OUT"no código).

No bloco *always* realiza-se o seguinte:

Incrementa-se o valor do contador a cada pulso do clock. A saída do divisor de frequência (denominada *Saida_DivFreq* no código) se mantém igual a zero. Quando o contador atinge o valor de $50 \cdot 10^6$, a saída do divisor de frequência é igual a 1 e o contador é zerado e recomeça a contagem.

Figura 7 – Código em Verilog do divisor de frequência.

```
module DivFreq(Clock, Saida_DivFreq);
input Clock;
output reg Saida_DivFreq;
reg [25:0] OUT;
always @ (posedge Clock)
    if (OUT == 26'd50000000)
    begin
        OUT <= 26'd0;
        Saida_DivFreq <= 1;
    end
    else
    begin
        OUT<= OUT+1;
        Saida_DivFreq <= 0;
    end
endmodule
```

Fonte: Elaborado pela autora.

4.3 Decodificador BCD para Display de 7 segmentos

Para representar as saídas no display presente na placa FPGA precisamos de um circuito decodificador responsável por fazer com que os bits que compõem a saída acendam os leds corretos. Vale ressaltar que os displays da placa DE2-115 são do tipo ânodo comum, o que significa que aplicando um nível lógico baixo no pino correspondente fará com que o segmento se acenda, enquanto que a aplicação do nível lógico alto fará com que ele fique se apague. A tabela verdade do decodificador BCD segue abaixo

Tabela 3 – Tabela-verdade do decodificador BCD para display do tipo ânodo comum.

	Entradas				Saídas para o display						
Número (em decimal)	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	1	1	0	0
10	1	0	1	0	1	1	1	1	1	1	1
11	1	0	1	1	1	1	1	1	1	1	1
12	1	1	0	0	1	1	1	1	1	1	1
13	1	1	0	1	1	1	1	1	1	1	1
14	1	1	1	0	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	1	1

Fonte: elaborado pela autora.

A partir da tabela 3 foi elaborado o código em Verilog para o conversor BCD. Ele consiste basicamente em um bloco *always* com a implementação de case que compara o valor em *entrada*. Esse valor em *entrada* é o número em binário que se deseja exibir no display. Temos dez casos, que são as possibilidades do número estar entre 0 e 9 inclusive e o *default* que é o caso onde o número não está no intervalo descrito e o display fica apagado.

Figura 8 – Código em Verilog para o decodificador BCD para display de 7 segmentos.

```
module BCD (entrada, segmentos);
input wire [3:0] entrada;
output reg [0:6] segmentos;
always @ (*)
    case (entrada)
        4'b0000: segmentos=7'b0000001;
        4'b0001: segmentos=7'b1001111;
        4'b0010: segmentos=7'b0010010;
        4'b0011: segmentos=7'b0000110;
        4'b0100: segmentos=7'b1001100;
        4'b0101: segmentos=7'b0100100;
        4'b0110: segmentos=7'b0100000;
        4'b0111: segmentos=7'b0001111;
        4'b1000: segmentos=7'b0000000;
        4'b1001: segmentos=7'b0000100;
        default: segmentos = 7'b1111111;
    endcase
endmodule
```

Fonte: elaborado pela autora.

4.4 Diagrama de estados e tabela verdade

Para desenvolver a máquina de estados, primeiramente é necessário elaborar o diagrama de estados. Este mostra os possíveis estados de um objeto e as ações responsáveis pelas suas mudanças de estado. Abaixo na tabela 4 temos os estados em número binário e a letra que foi atribuída a cada estado.

Tabela 4 – Representação dos estados em número binário e a letra atribuída a cada um.

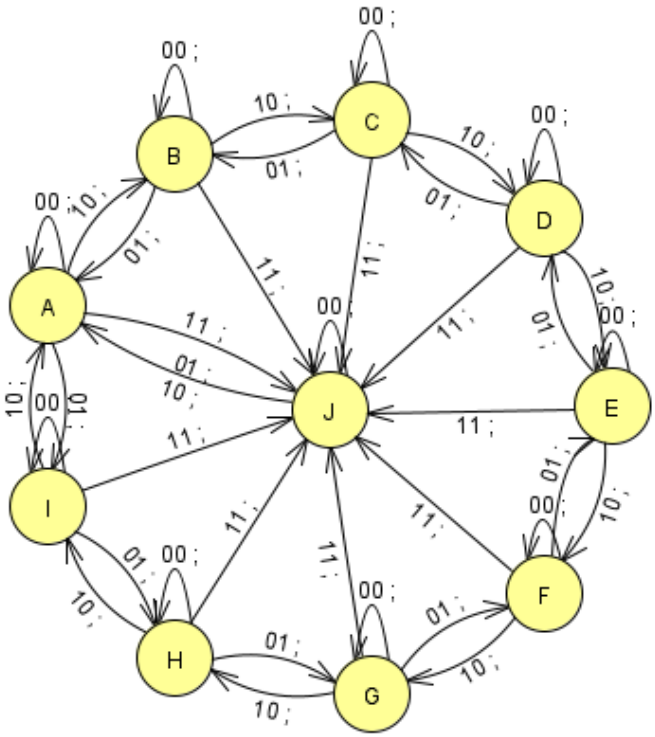
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J

Fonte: elaborado pela autora.

Utilizando a tabela acima, através do software JFLAP [2], cria-se o diagrama de estados.

[2] Disponível em: <http://www.jflap.org/>;. Acesso em Fev.2021.

Figura 9 – Diagrama de estados.



Fonte: elaborado pela autora.

A partir disto, desenvolve-se a tabela verdade [5] do circuito que possui os estados atuais, os próximos estados, as saídas e o modo (UP/DOWN).

Tabela 5 – Tabela-verdade dos circuitos combinacionais de entrada e saída.

Estado atual	Próximo Estado (E3', E2', E1', E0')				Saída
(E3, E2, E1, E0)	UD (00)	UD (01)	UD (10)	UD (11)	(S3, S2, S1, S0)
0000	0000	1000	0001	1001	1001
0001	0001	0000	0010	1001	0100
0010	0010	0001	0011	1001	0110
0011	0011	0010	0100	1001	0101
0100	0100	0011	0101	1001	1000
0101	0101	0100	0110	1001	0010
0110	0110	0101	0111	1001	0001
0111	0111	0110	1000	1001	0000
1000	1000	0111	0000	1001	0101
1001	1001	0000	0000	1001	1111

Fonte: elaborado pela autora.

Com base na Tabela 5, é possível desenvolver o código em Verilog da máquina de estados.

O módulo *Moore* contém as entradas *Up, Down, Reset* e *Clock* e a saída *SaidaMoore*. Além disso, utiliza-se dois vetores do tipo *reg* com 4 bits cada: *estadoAtual* e *proxEstado*.

Define-se, também, parâmetros *(parameter)*, onde cada letra corresponde a um estado, como descrito na tabela 4.

Figura 10 – Módulo *Moore*: Declaração das entradas, saídas e definição dos parâmetros.

```
module Moore(Up, Down, Reset, Clock, SaidaMoore);
input Up, Down, Reset, Clock;
output reg [3:0]SaidaMoore;

reg [3:0] estadoAtual;
reg [3:0] proxEstado;

parameter A= 4'b0000, B= 4'b0001, C= 4'b0010, D= 4'b0011, E= 4'b0100, F= 4'b0101, G= 4'b0110, H= 4'b0111, I= 4'b1000, J= 4'b1001;
```

Fonte: elaborado pela autora.

O bloco *always* é sensível as variáveis *estadoAtual*, *Up* e *Down* e, de acordo com o *estadoAtual* e as entradas *Up* e *Down*, escolhe-se qual será o próximo estado da máquina.

Se $\mathbf{Up = 0}$ e $\mathbf{Down = 0}$, o estado atual se mantém.

Se $\mathbf{Up = 0}$ e $\mathbf{Down = 1}$, o próximo estado será o estado anterior ao estado atual (ou seja, uma contagem decrescente).

Se $\mathbf{Up = 1}$ e $\mathbf{Down = 0}$, o próximo estado será o estado posterior ao estado atual (ou seja, uma contagem crescente).

Se $\mathbf{Up = 1}$ e $\mathbf{Down = 1}$, o próximo estado será o estado J, ou seja, o estado em que o *display* fica apagado.

Figura 11 – Módulo *Moore*: Declaração do bloco *always* e o caso do estado atual ser A, definindo o próximo estado de acordo com as entradas Up e Down.

```
always @(estadoAtual or Up or Down)
    begin
    case(estadoAtual)
        A:
            if(Up==0 & Down==0)
            begin
                proxEstado=A;
            end
            else if(Up==0 & Down==1)
            begin
                proxEstado=I;
            end
            else if(Up==1 & Down==0)
            begin
                proxEstado=B;
            end
            else if(Up==1 & Down==1)
            begin
                proxEstado=J;
            end
```

Fonte: elaborado pela autora.

Figura 12 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja B.

```
B:
    if(Up==0 & Down==0)
    begin
        proxEstado=B;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=A;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=C;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 13 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja C.

```
C:
    if(Up==0 & Down==0)
    begin
        proxEstado=C;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=B;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=D;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 14 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja D.

```
D:
    if(Up==0 & Down==0)
    begin
        proxEstado=D;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=C;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=E;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 15 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja E.

```
E:
    if(Up==0 & Down==0)
    begin
        proxEstado=E;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=D;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=F;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 16 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja F.

```
F:
    if(Up==0 & Down==0)
    begin
        proxEstado=F;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=E;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=G;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 17 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja G.

```
G:
    if(Up==0 & Down==0)
    begin
        proxEstado=G;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=F;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=H;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 18 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja H.

```
H:
    if(Up==0 & Down==0)
    begin
        proxEstado=H;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=G;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=I;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 19 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja I.

```
I:
    if(Up==0 & Down==0)
    begin
        proxEstado=I;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=H;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=A;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
```

Fonte: elaborado pela autora.

Figura 20 – Módulo *Moore*: definição do próximo estado de acordo com as entradas Up e Down caso o estado atual seja J e o encerramento do bloco *always*.

```
J:
    if(Up==0 & Down==0)
    begin
        proxEstado=J;
    end
    else if(Up==0 & Down==1)
    begin
        proxEstado=A;
    end
    else if(Up==1 & Down==0)
    begin
        proxEstado=A;
    end
    else if(Up==1 & Down==1)
    begin
        proxEstado=J;
    end
endcase
end
```

Fonte: elaborado pela autora.

Por fim, o módulo possui outros dois blocos *always*.

Neste projeto o *Reset* é assíncrono com o *Clock*. A maneira como o circuito é codificado tem um impacto direto nos sinais de controle síncronos ou assíncronos.

O primeiro bloco *always* é o responsável por aplicar o pulso de *Clock* e o *Reset*, sendo sensível a essas duas entradas. A keyword *posedge* está acompanhando ambas as entradas, o que significa que o *always* está sensível à borda de subida delas. Então, se a variável *Reset* muda, desconsidera-se o valor do sinal do *Clock* e executa-se o comando de *Reset*. Logo, esta implementação, em particular, é uma implementação assíncrona de um *Reset* que é independente do *Clock*.

Se **Reset = 1**, o estado atual assumirá o valor A. Caso contrário, o estado atual assumirá o valor do *proxEstado* (de acordo com o estipulado anteriormente).

O segundo bloco *always* faz com que a *saídaMoore* seja igual ao estado atual. É a variável *saídaMoore* que será enviada como entrada do conversor BCD para exibir no display de 7 segmentos o estado atual da máquina de Moore.

Figura 21 – Módulo *Moore*: Declaração dos dois blocos *always*: o bloco responsável por aplicar o clock (e o reset quando solicitado) e o bloco responsável por enviar para a saída do módulo *Moore* o valor do estado atual da máquina.

```
always @(posedge Clock or posedge Reset) // Reset
begin
    if(Reset == 1)
        estadoAtual <= A;
    else
        estadoAtual <= proxEstado;
end

always @(estadoAtual) // saida de acordo com o estado atual
begin
    case(estadoAtual)
        A: SaidaMoore = 4'b1001;
        B: SaidaMoore = 4'b0100;
        C: SaidaMoore = 4'b0110;
        D: SaidaMoore = 4'b0101;
        E: SaidaMoore = 4'b1000;
        F: SaidaMoore = 4'b0010;
        G: SaidaMoore = 4'b0001;
        H: SaidaMoore = 4'b0000;
        I: SaidaMoore = 4'b0101;
        J: SaidaMoore = 4'b1111;
    endcase
end
endmodule
```

Fonte: elaborado pela autora.

4.5 Código completo

```
module Contador(SW, CLOCK_50, HEX4);
input wire [17:0] SW;
input wire CLOCK_50;
output wire [0:6] HEX4;

wire clk;
wire [3:0] sm;

DivFreq df(.Clock(CLOCK_50), .Saida_DivFreq(clk));
Moore m(.Up(SW[1]),.Down(SW[0]),.Reset(SW[17]),.Clock(clk),.SaidaMoore(sm));
BCD b(.entrada(sm), .segmentos(HEX4));
endmodule


module DivFreq(Clock, Saida_DivFreq);
input Clock;
output reg Saida_DivFreq;
reg [25:0] OUT;
always @ (posedge Clock)
    if (OUT == 26'd50000000)
    begin
            OUT <= 26'd0;
            Saida_DivFreq <= 1;
    end
    else
    begin
            OUT<= OUT+1;
            Saida_DivFreq <= 0;
    end
endmodule


module Moore(Up, Down, Reset, Clock, SaidaMoore);
input Up, Down, Reset, Clock;
output reg [3:0]SaidaMoore;

reg [3:0] estadoAtual;
reg [3:0] proxEstado;

parameter A= 4'b0000, B= 4'b0001, C= 4'b0010, D= 4'b0011, E= 4'b0100,
```

```
F= 4'b0101, G= 4'b0110, H= 4'b0111, I= 4'b1000, J= 4'b1001;

always @(estadoAtual or Up or Down)
    begin
    case(estadoAtual)
        A:
                if(Up==0 & Down==0)
                        begin
                            proxEstado=A;
                    end
                    else if(Up==0 & Down==1)
                        begin
                        proxEstado=I;
                        end
                else if(Up==1 & Down==0)
                        begin
                        proxEstado=B;
                        end
                else if(Up==1 & Down==1)
                        begin
                        proxEstado=J;
                        end
        B:
                if(Up==0 & Down==0)
                        begin
                        proxEstado=B;
                        end
                else if(Up==0 & Down==1)
                        begin
                        proxEstado=A;
                        end
                else if(Up==1 & Down==0)
                        begin
                        proxEstado=C;
                        end
                else if(Up==1 & Down==1)
                        begin
                        proxEstado=J;
                        end
        C:
                if(Up==0 & Down==0)
                begin
```

```
                proxEstado=C;
            end
            else if(Up==0 & Down==1)
            begin
                proxEstado=B;
            end
            else if(Up==1 & Down==0)
            begin
                proxEstado=D;
            end
            else if(Up==1 & Down==1)
            begin
                proxEstado=J;
            end
        D:
            if(Up==0 & Down==0)
            begin
                proxEstado=D;
            end
            else if(Up==0 & Down==1)
            begin
                proxEstado=C;
            end
            else if(Up==1 & Down==0)
            begin
                proxEstado=E;
            end
            else if(Up==1 & Down==1)
            begin
                proxEstado=J;
            end
        E:
            if(Up==0 & Down==0)
            begin
                proxEstado=E;
            end
            else if(Up==0 & Down==1)
            begin
                proxEstado=D;
            end
            else if(Up==1 & Down==0)
            begin
```

```
                proxEstado=F;
        end
        else if(Up==1 & Down==1)
        begin
                proxEstado=J;
        end
F:
        if(Up==0 & Down==0)
        begin
                proxEstado=F;
        end
        else if(Up==0 & Down==1)
        begin
                proxEstado=E;
        end
        else if(Up==1 & Down==0)
        begin
                proxEstado=G;
        end
        else if(Up==1 & Down==1)
        begin
                proxEstado=J;
        end
G:
        if(Up==0 & Down==0)
        begin
                proxEstado=G;
        end
        else if(Up==0 & Down==1)
        begin
                proxEstado=F;
        end
        else if(Up==1 & Down==0)
        begin
                proxEstado=H;
        end
        else if(Up==1 & Down==1)
        begin
                proxEstado=J;
        end
H:
        if(Up==0 & Down==0)
```

```
                begin
                        proxEstado=H;
                end
                else if(Up==0 & Down==1)
                begin
                        proxEstado=G;
                end
                else if(Up==1 & Down==0)
                begin
                        proxEstado=I;
                end
                else if(Up==1 & Down==1)
                begin
                        proxEstado=J;
                end
        I:
                if(Up==0 & Down==0)
                begin
                        proxEstado=I;
                end
                else if(Up==0 & Down==1)
                begin
                        proxEstado=H;
                end
                else if(Up==1 & Down==0)
                begin
                        proxEstado=A;
                end
                else if(Up==1 & Down==1)
                begin
                        proxEstado=J;
                end
        J:
                if(Up==0 & Down==0)
                        begin
                        proxEstado=J;
                        end
                else if(Up==0 & Down==1)
                        begin
                        proxEstado=A;
                        end
                else if(Up==1 & Down==0)
```

```
                        begin
                        proxEstado=A;
                        end
                else if(Up==1 & Down==1)
                        begin
                        proxEstado=J;
                        end
        endcase
    end

always @(posedge Clock or posedge Reset) // Reset
begin
        if(Reset == 1)
                estadoAtual <= A;
        else
                estadoAtual <= proxEstado;
end

always @(estadoAtual) // saida de acordo com o estado atual
begin
        case(estadoAtual)
        A: SaidaMoore = 4'b1001;
        B: SaidaMoore = 4'b0100;
        C: SaidaMoore = 4'b0110;
        D: SaidaMoore = 4'b0101;
        E: SaidaMoore = 4'b1000;
        F: SaidaMoore = 4'b0010;
        G: SaidaMoore = 4'b0001;
        H: SaidaMoore = 4'b0000;
        I: SaidaMoore = 4'b0101;
        J: SaidaMoore = 4'b1111;
    endcase
end
endmodule


module BCD (entrada, segmentos);
input wire [3:0] entrada;
output reg [0:6] segmentos;
always @ (*)
    case (entrada)
                4'b0000: segmentos=7'b0000001;
```

```
                4'b0001: segmentos=7'b1001111;
                4'b0010: segmentos=7'b0010010;
                4'b0011: segmentos=7'b0000110;
                4'b0100: segmentos=7'b1001100;
                4'b0101: segmentos=7'b0100100;
                4'b0110: segmentos=7'b0100000;
                4'b0111: segmentos=7'b0001111;
                4'b1000: segmentos=7'b0000000;
                4'b1001: segmentos=7'b0000100;
                default: segmentos = 7'b1111111;
        endcase
endmodule
```

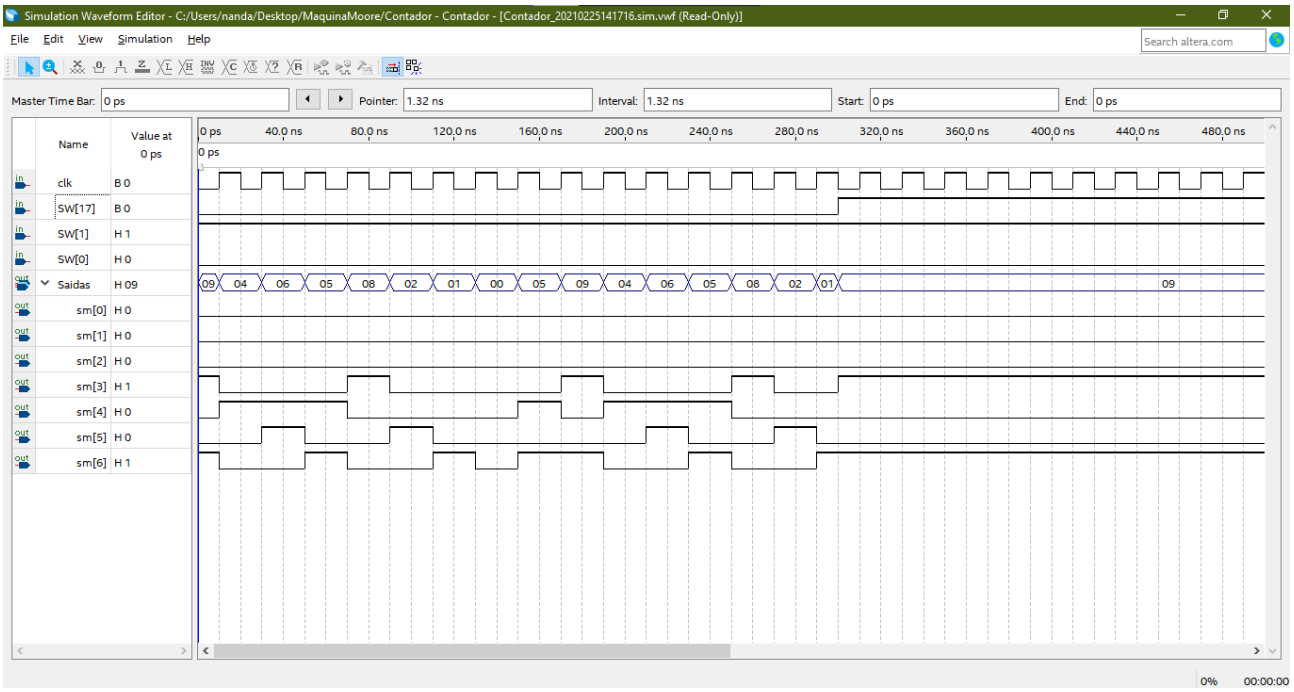
5 Resultados obtidos

5.1 Simulação no software Quartus (waveforms)

Abaixo seguem as imagens das *waveforms* geradas pelo software *Intel Quartus Prime*.

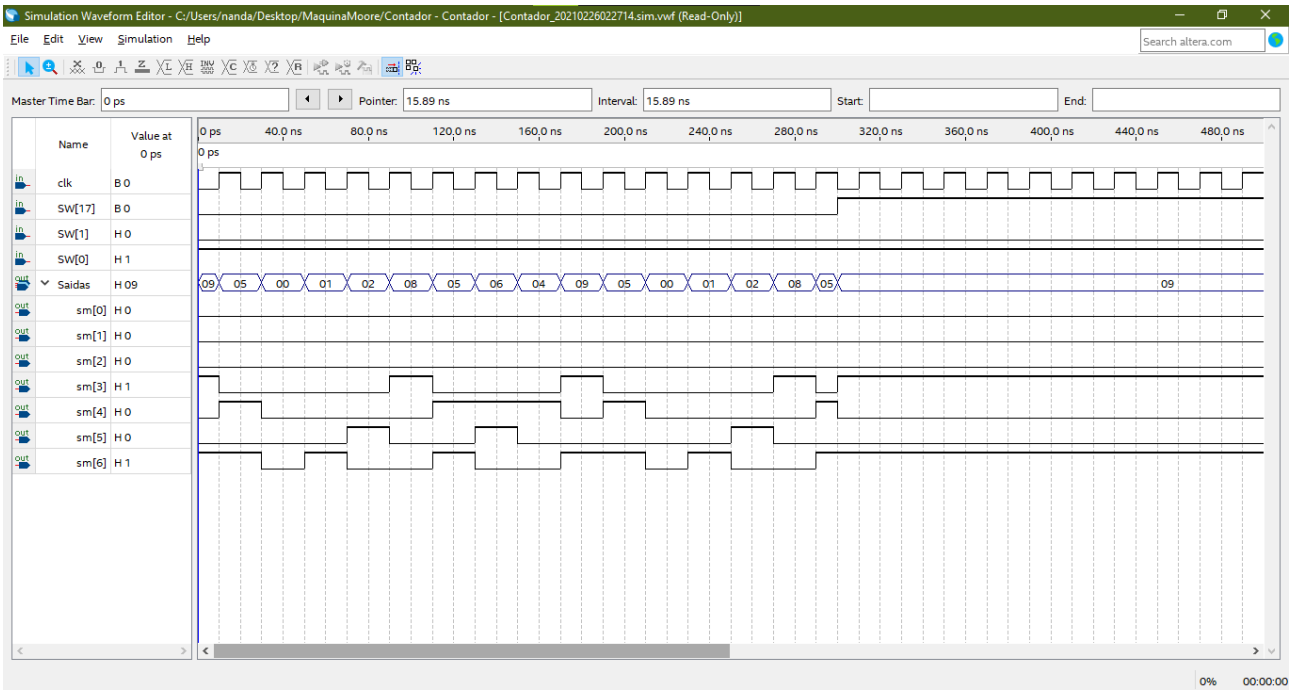
Temos Up = SW1, Down = SW0 e Reset = SW17. A saída está representada em hexadecimal.

Figura 22 – Waveform da simulação que utiliza as entradas Up = 1, Down = 0 (ou seja, uma contagem crescente) e Reset ativado aos 300 ns e se mantendo por 300 ns.



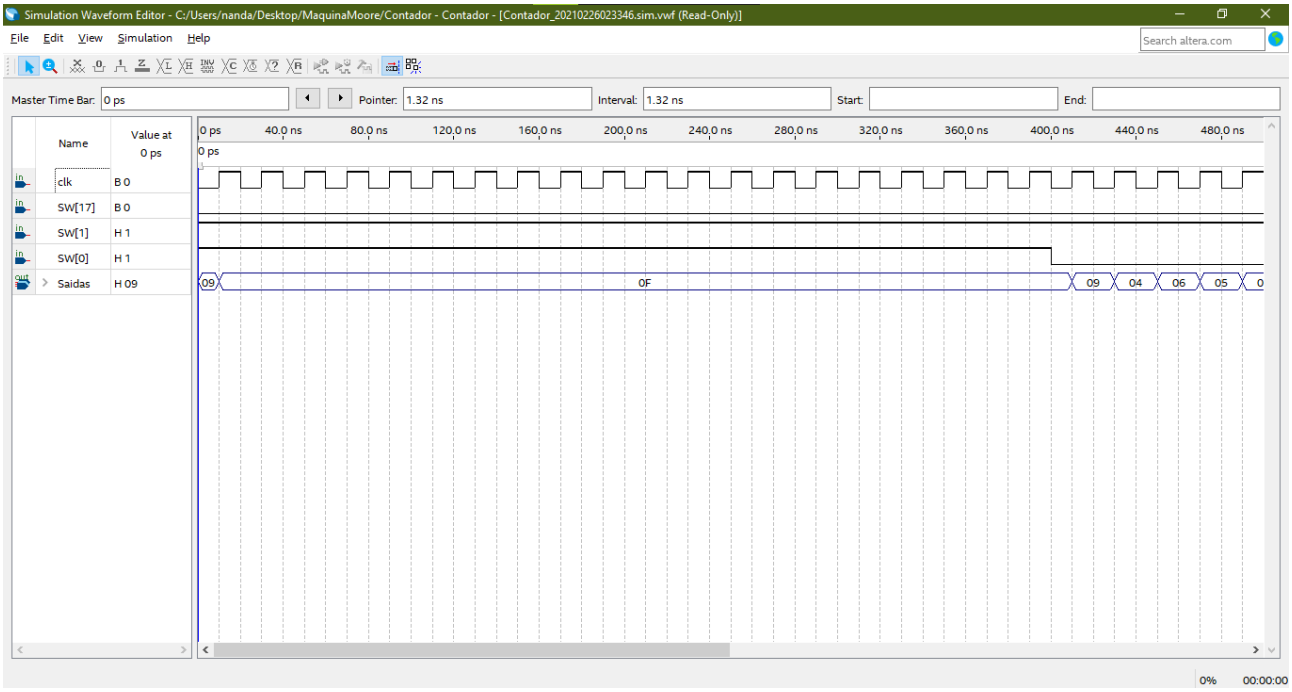
Fonte: elaborado pela autora.

Figura 23 – Waveform da simulação que utiliza as entradas Up = 0, Down = 1 (ou seja, uma contagem decrescente) e Reset ativado aos 300 ns e se mantendo por 300 ns.



Fonte: elaborado pela autora.

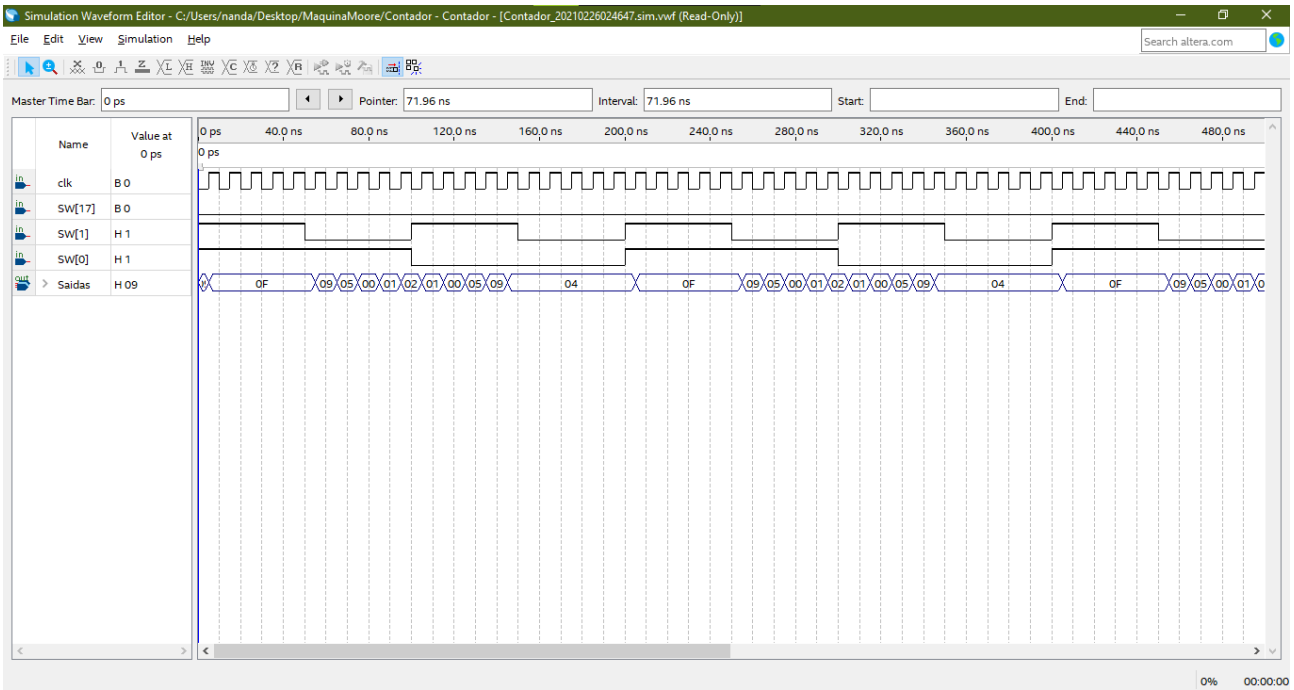
Figura 24 – Waveform da simulação que utiliza as entradas Up = 1 e Down ativada aos 10 ns e se mantendo por 400 ns (o display fica apagado, após isso inicia uma contagem crescente).



Fonte: elaborado pela autora.

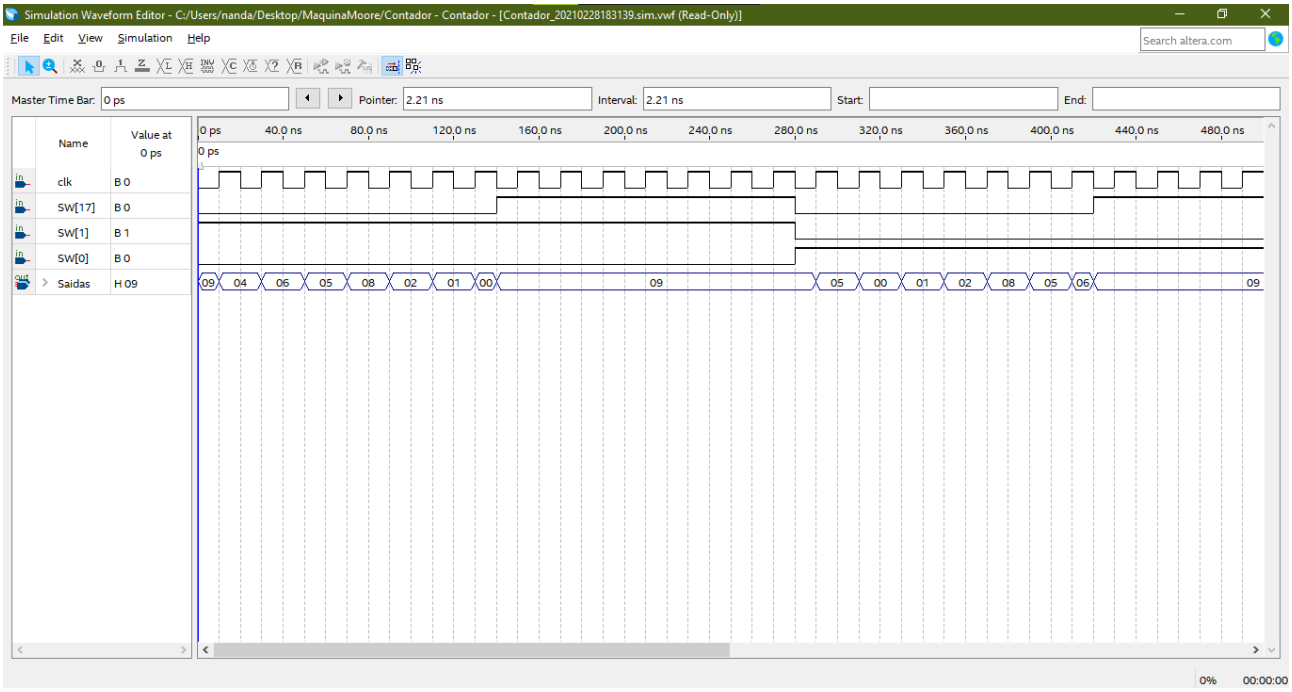
Na figura 25 temos uma simulação onde o display inicia apagado (0ns a 50ns - Up = 1 e Down = 1), realiza uma contagem decrescente(50ns a 100ns - Up = 0 e Down = 1), realiza uma contagem crescente (100ns a 150ns - Up = 1 e Down = 0) e se mantém no estado atual(150ns a 200ns - Up = 0 e Down = 0). Após isso, repete o mesmo procedimento.

Figura 25 – Waveform da simulação que utiliza as entradas Reset = 0, Up é ativada a cada 50 ns e se mantém ativa por 50 ns e Down é ativada a cada 100 ns e se mantém ativa por 100 ns.



Fonte: elaborado pela autora.

Figura 26 – Waveform da simulação onde está ocorrendo uma contagem crescente, aplica-se o reset e, após isso, ocorre uma contagem decrescente.



Fonte: elaborado pela autora.

5.2 Simulação no kit FPGA remoto

Foram realizadas, no kit FPGA, as seguintes simulações:

1) Apresentar todas as transições de estado crescente.

2) Apresentar todas as transições de estado decrescente.

3) Escolher um estado e manter por aproximadamente 3 segundos.

4) Na contagem crescente, entrar no estado apagado e manter neste estado por 3 segundos. Após isto, retornar à contagem decrescente.

5) Escolher um estado para aplicar o reset. Após isto, iniciar a contagem decrescente.

Em todas elas o circuito apresentou bom funcionamento, mostrando o resultado esperado e correto.

6 Conclusão

A máquina de estados de Moore não apresentou falhas ao representar os estados solicitados. Tanto as simulações em forma de onda realizadas no software *Intel Quartus Prime*, quanto as simulações realizadas no kit remoto foram um sucesso.

Referências

TOCCI, Ronald J.; WIDNER, Neal S.; MOSS, Gregory L. Sistemas Digitais: Princípios e Aplicações. 11º ed. – São Paulo : Pearson Prentice Hall, 2011.

VAHID, Frank. Sistemas Digitais: Projeto, Otimização e HDLs. 1ª ed. São Paulo: Bookman Companhia Editora Ltda, 2008.

COSTA, Cesar da. Projetos de Circuitos Digitais com FPGA. 3. ed. Editora Érica, 2013.

ROSSI. Fernanda Quelho. Introdução à Linguagem de Descrição de Hardware Verilog, 2021. 32 slides. Disponível em: <https://grad.sead.unifesp.br/pluginfile.php/240329/mod_resource/content/19/Aula%209-0%20-%20Introducao%20ao%20Verilog.pdf>. Acesso em: fev. 2021.

ROSSI. Fernanda Quelho. Introdução à Linguagem de Descrição de Hardware Verilog (Parte 2), 2021. 33 slides. Disponível em: <https://grad.sead.unifesp.br/pluginfile.php/241707/mod_resource/content/22/Aula%2010-0%20-%20Introducao%20ao%20Verilog.pdf>. Acesso em: fev. 2021.

ROSSI. Fernanda Quelho. Máquina de Estados para um Contador Crescente e Decrescente em Verilog, 2021. 10 slides. Disponível em: <https://grad.sead.unifesp.br/pluginfile.php/244083/mod_resource/content/25/Aula%2012-0%20-%20Maquina%20de%20Estados.pdf>. Acesso em: fev. 2021.

PEREIRA, Rodrigo. PROCESSADORES PROGRAMÁVEIS: como projetar um processador em VERILOG 14 abr. 2015. Disponível em: <https://www.embarcados.com.br/processador-em-verilog-3/>. Acesso em: fev. 2021.

SOUZA, Felipe de Assis. Tutorial Verilog, ed. 1, jan. 2011. Disponível em: <https://www.cin.ufpe.br/$\sim$eaa3/Arquivos/Verilog/Tutorial\T1\textbackslash{}20Verilog.pdf>. Acesso em: fev. 2021.

Verilog. In: WIKIPÉDIA: a enciclopédia livre. 11 mar. 2020. Disponível em: <https://pt.wikipedia.org/wiki/Verilog>. Acesso em: fev. 2021.

Edward F. Moore. In: WIKIPÉDIA: a enciclopédia livre. 30 mar. 2020. Disponível em: <https://pt.wikipedia.org/wiki/Edward_F._Moore>. Acesso em: fev. 2021.

Máquina de Moore. In: WIKIPÉDIA: a enciclopédia livre. 23 mar. 2019. Disponível em: <https://pt.wikipedia.org/wiki/M%C3%A1quina_de_Moore>. Acesso em: fev. 2021.

CURSO BÁSICO DE VERILOG. Disponível em: <http://paginapessoal.utfpr.edu.

br/chiesse/disciplinas/logica-reconfiguravel/verilog/Curso%20Basico%20de%20Verilog.pdf/at_download/file#:$\sim$:text=Verilog%20n%C3%A3o%20%C3%A9%20uma%20linguagem,em%20um%20conjunto%20de%20instru%C3%A7%C3%B5es.&text=A%20linguagem%20Verilog%20%C3%A9%20um,para%20simula%C3%A7%C3%A3o%20quanto%20para%20s%C3%ADntese>. Acesso em: 26 fev. 2021.