****

**Universidade Federal do Rio Grande Do Norte**

**Campus Central (Natal)**

**Relatório Técnico**

**Ênfase em Mecatrônica**

**Aluno: Mateus de Assis Silva.**

**Professor: Samaherni Morais Dias**

**Disciplina: Circuitos Digitais - Laboratório**

**Implementação de Contador Limitado em *VHDL***

**Introdução:**

Um dos elementos utilizados para adicionar memória nos nossos circuitos se chama *flip-flop*. Trata-se de uma combinação de portas lógicas capaz de manter informação binária enquanto estiver energizado e mudar seu estado quando sua entrada vai para nível lógico alto junto com um sinal de sincronia (*clock*). Para os nossos desenvolvimentos futuros, iremos considerá-los como foi definido [1]. O *flip flop* aqui utilizado terá três entradas: *toggling* (que inverte o estado da saída), *preseting* (que força a saída pra um) e *clearing* (que força a saída para zero).

Outro conceito importante é a de máquina de estados finitos. Essa é uma entidade abstrata capaz de alterar sua informação de saída dentre um conjunto de saídas possíveis quando uma dada condição de entrada é atendida. Assim, podemos adicionar um elemento temporal às nossas conhecidas tabelas-verdade [2].

Usando tais definições, podemos construir elementos que envolvam manter um sinal por tempo até serem solicitados que mudem. Assim sendo, iremos desenvolver um contador *up-down* com limite definido.

O contador a ser desenvolvido é um componente capaz de contar incrementalmente partindo de um limite inferior até um certo limite superior, com ambos definidos pelo o usuário e, posteriormente, decrementar o seu valor quando esse limite superior é atingido. De forma a tornar esse sistema mais complexo, permitirá-se o anulamento do sinal (*clear*) e sua pausa (*disenabling*). Vale saber que, logo após o momento em que o sinal de saída é anulado, a contagem deve ser feita no sentido incremental.

**Desenvolvimento:**

De forma a poder construir tal sistema, dividiu-se em módulos menores, seguindo as instruções contidas na atividade disponibilizada pelo professor. O sistema completo pode ser visualizado na Figura 1, logo abaixo.

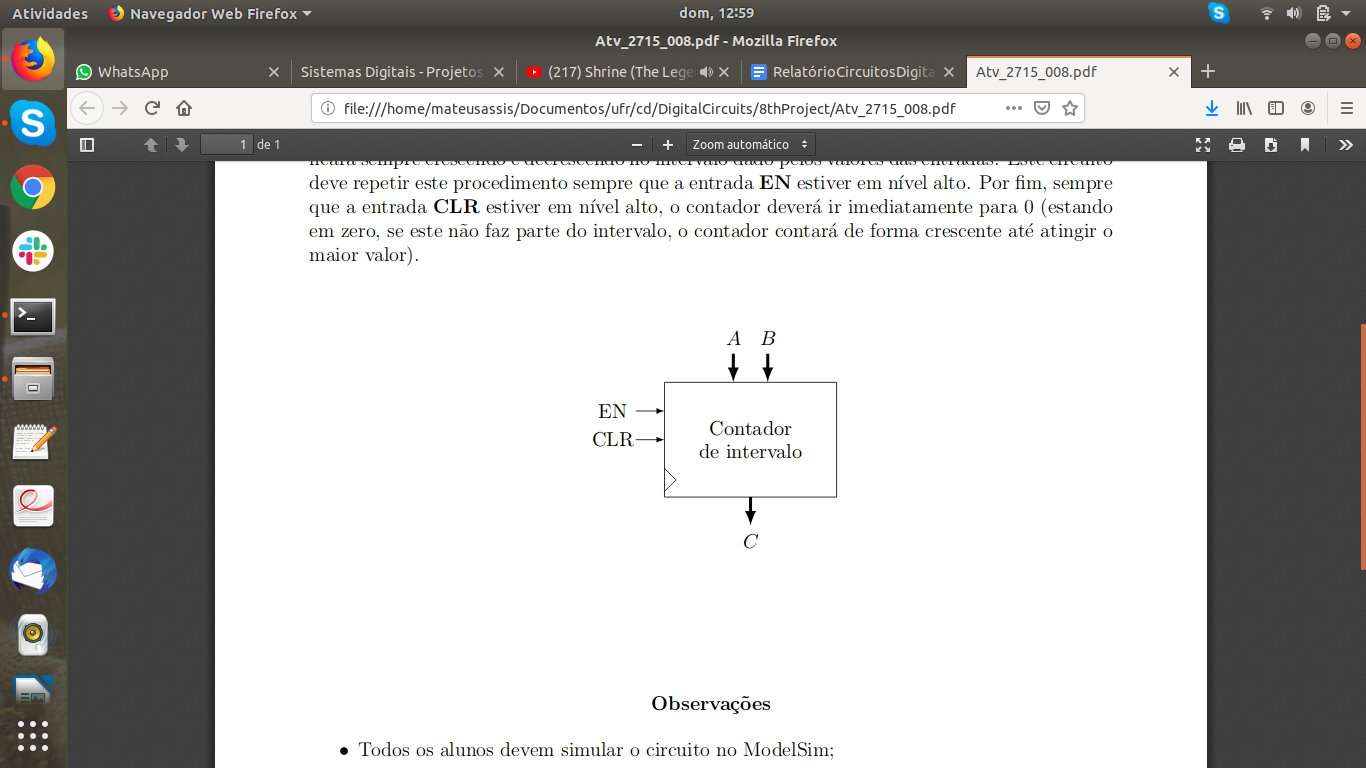


Figura 1: Sistema de contagem.

Para podermos desenvolver o sistema apresentado na Figura 1, precisamos definir os estados que serão possíveis, e os que não serão. Para tanto, considere a máquina de estados (FSM) apresentada na Figura 2. Para tal, consideramos que os estados variam no sentido horário (*up counting*, como pode ser notado) quando uma variável que orienta o sentido da contagem está com valor nulo (M = 0). Caso essa variável seja unitária (M = 1), a contagem é decrescente. Tal sistema de contagem é o mesmo desenvolvido na implementação anterior.

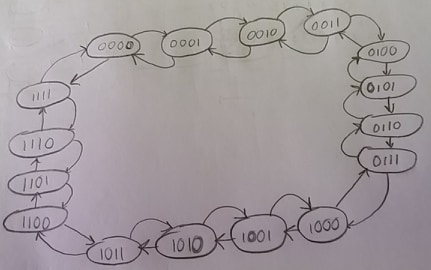


Figura 2: Máquina de Estados para o Sistema.

De forma a implementarmos a FSM, iremos utilizar *flip flops (FF)* conectados entre si através de circuitos combinacionais. De forma a deduzir uma lógica de alteração de estado para cada *FF*, iremos analisar como os estados variam, usando tabelas verdade. Além disso, iremos entender como esses estados mudam a partir da influência das entradas dos *FF*.

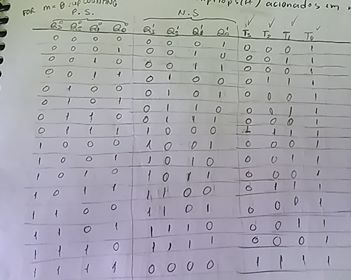


Figura 3: Tabela Verdade *up counting*.

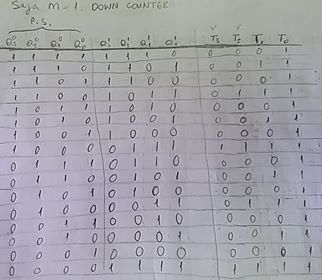


Figura 4: Tabela Verdade *down counting*.

A partir das tabelas, podemos simplificá-las usando mapas K. Note que a tabela apresentada na Figura 3 é válida pra M = 0, e aquela da Figura 4 pra M = 1. Logo, os resultados devem ser multiplicados por M e M’, respectivamente.

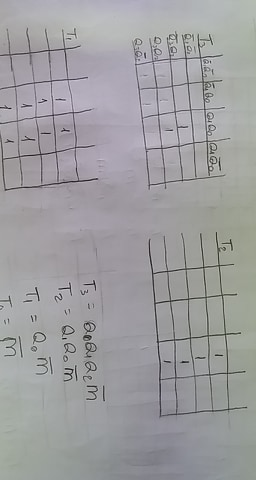


Figura 5: Simplificação da Lógica, *up counting*.

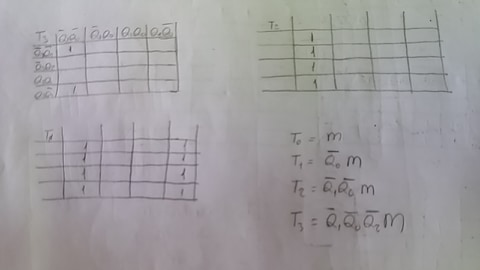


Figura 6: Simplificação da Lógica, *down counting*.

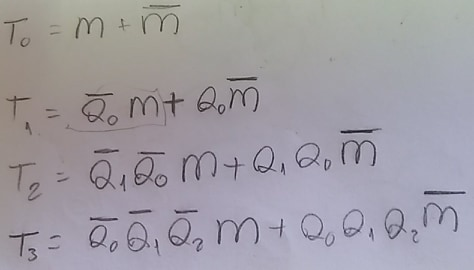


Figura 7: Lógica de Estados.

Vale notar que a contagem só deve ocorrer enquanto o contador estiver habilitado (*Enabled*). Para tanto, cada entrada *toggling* Ti estará em “lógica E” com uma entrada En (*Enable*).

Até então, o desenvolvimento segue o mesmo raciocínio da implementação da atividade 07. A primeira diferença do sistema ocorre no uso dos limites para inversão da sequência de contagem.

Agora que temos o contador pronto, precisamos adicionar uma forma de limitar a contagem. Isso é, dado a necessidade de impor limites, precisamos alterar o sentido da contagem. Ora, para implementarmos tal comportamento, usaremos comparadores. Um deles compara o valor atual da contagem com o limite superior, e força M para 1. O outro comparador compara o valor atual com o limite inferior, e força M para 0, quando a contagem atinge o limitante mínimo.

Por fim, a única mudança adicional necessária é a adição da carga paralela, de forma a permitir que a contagem se inicie no limite mínimo. Para isso, considere a seguinte circuitaria uma “customização“ do flip flop já conhecido.

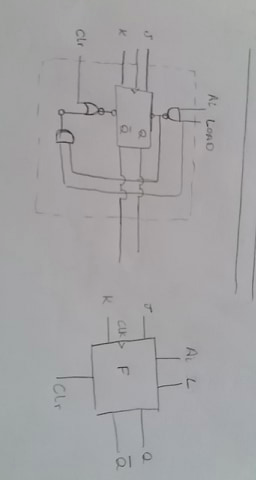


Figura 8: Flip Flop Customizado.

Definindo assim, podemos considerar nosso circuito completo como se segue:

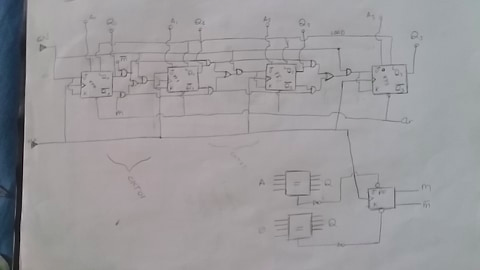


Figura 9: Sistema implementado.

Ora, definidos esse elementos, passou-se a implementação do sistema em linguagem *VHDL*. O primeiro dispositivo a ser incorporado foi o comparador de 2 entradas (cada uma com 4 bits), a partir de comparadores simples de duas entradas binárias.

Por fim, dado que o *flip flop* JK já estava definido em arquivos anteriores, bastou se utilizá-lo para concluir o sistema. Alguns testes foram realizados com limites 5 e 13 (valores nas primeiras faixas do diagrama de tempo), e seus resultados podem ser visto abaixo. Considere que a terceira faixa é o *Clear* e a quarta faixa é o *clock*. Por fim, a faixa seguinte o ‘*Load*’, seguida do *Enable*. Os códigos implementados encontram-se em anexo.

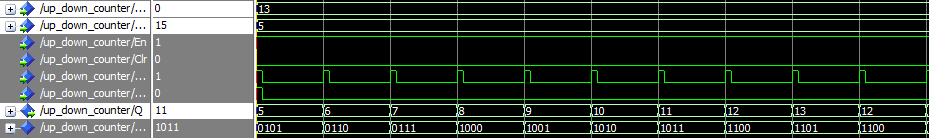


Figura 10: Contagem incremental.

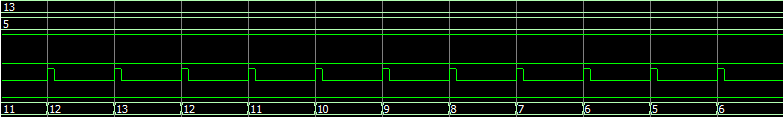


Figura 11: Contagem decremental.

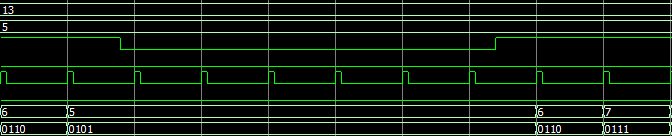


Figura 12: Pausa na contagem.

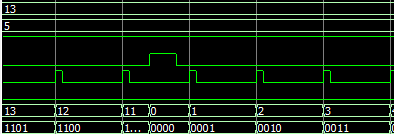


Figura 13: *Cleasing* seguido de contagem incremental.

**Conclusão:**

O projeto conseguiu demonstrar o funcionamento da carga paralela, aliada à contagem bidimensional. Também se experimentou a versatilidade das simulações, ante as implementações em *hardware*, devido à facilidade de alteração de componentes em *software*.

Por fim, encontrou-se os resultados esperados, ao fim das simulações. Esse são condizentes com o esperado do teórico.

**Referências:**

[1] Vahid, F. (2009). *Sistemas Digitais*. Bookman Editora. P.112 - 123.

[2] Vahid, F. (2009). *Sistemas Digitais*. Bookman Editora. P.129 - 151.

**Anexo A: Códigos**

-- Comparer for 2 bit equality

-- if two 1bit inputs are equal, the output goes to 1.

-- Coded by Mateus de Assis. For more info, go to github.com/mtxslv/DigitalCircuits

entity equal2input is

port( inputA,inputB: in bit;

output: out bit);

end equal2input;

architecture equal2input\_ckt of equal2input is

begin

output <= (inputA AND inputB)OR(NOT(inputA) AND NOT(inputB));

end equal2input\_ckt;

-- Comparer for 4-bit equality

-- For more info, go to github.com/mtxslv/DigitalCircuits

-- If the vector inputs are equal, the output goes to 1.

-- In order to work properly, make sure equal2input.vhd file is in the folder.

-- Coded by Mateus de Assis.

entity equal2input4bit is

port( vectorA,vectorB: in bit\_vector(3 downto 0);

equal: out bit);

end equal2input4bit;

architecture equal2input4bit\_ckt of equal2input4bit is

component equal2input is

port( inputA,inputB: in bit;

output: out bit);

end component;

signal aux : bit\_vector(3 downto 0);

begin

lbl\_3: equal2input port map(inputA =>vectorA(3),

inputB =>vectorB(3),

output => aux(3));

lbl\_2: equal2input port map(inputA =>vectorA(2),

inputB =>vectorB(2),

output => aux(2));

lbl\_1: equal2input port map(inputA =>vectorA(1),

inputB =>vectorB(1),

output => aux(1));

lbl\_0: equal2input port map(inputA =>vectorA(0),

inputB => vectorB(0),

output => aux(0));

equal <= aux(3) AND aux(2) AND aux(1) AND aux(0);

end equal2input4bit\_ckt;

-- up down counter

-- for more info go to github.com/mtxslv

-- in order to work properly, please

-- make sure equal2input4bit.vhd and ffjk.vhd files are

-- in the folder.

-- coded by Mateus de Assis Silva

entity up\_down\_counter is

port(upper\_lim, lower\_lim: in bit\_vector(3 downto 0); -- B = upper\_lim , A = lower\_lim

En, Clr, Clk, load: in bit;

Q: out bit\_vector(3 downto 0));

end up\_down\_counter;

architecture up\_down\_counter\_ckt of up\_down\_counter is

component equal2input4bit is

port( vectorA,vectorB: in bit\_vector(3 downto 0);

equal: out bit);

end component;

component ffjk IS

port(clk, J,K,P,C: IN BIT;

q: OUT BIT);

END component;

signal q\_aux : bit\_vector(3 downto 0);

signal preset\_aux, clear\_aux, not\_clear\_aux, not\_preset\_aux, M: bit;

signal ckt01, ckt12, ckt23: bit;

-- in order to make a useful use of lower limit, we'll need to use

-- parallel load

signal load\_lower\_lim, load\_clear: bit\_vector(3 downto 0);

-- the usage of parallel load demmands a circuit for clear

begin

-- parallel load

load\_lower\_lim(3) <= NOT(lower\_lim(3) AND load);

load\_lower\_lim(2) <= NOT(lower\_lim(2) AND load);

load\_lower\_lim(1) <= NOT(lower\_lim(1) AND load);

load\_lower\_lim(0) <= NOT(lower\_lim(0) AND load);

--parallel clear

-- watch out! In order to clear the output of the flipflops, Clr <- 1

load\_clear(3) <= NOT(Clr OR (load\_lower\_lim(3) AND load));

load\_clear(2) <= NOT(Clr OR (load\_lower\_lim(2) AND load));

load\_clear(1) <= NOT(Clr OR (load\_lower\_lim(1) AND load));

load\_clear(0) <= NOT(Clr OR (load\_lower\_lim(0) AND load));

COMP\_UPPER: equal2input4bit port map(vectorA(3) => upper\_lim(3),

vectorA(2) => upper\_lim(2),

vectorA(1) => upper\_lim(1),

vectorA(0) => upper\_lim(0),

vectorB(3) => q\_aux(3),

vectorB(2) => q\_aux(2),

vectorB(1) => q\_aux(1),

vectorB(0) => q\_aux(0),

equal => preset\_aux); --comparator of upper limit

COMP\_LOWER: equal2input4bit port map(vectorA(3) => lower\_lim(3),

vectorA(2) => lower\_lim(2),

vectorA(1) => lower\_lim(1),

vectorA(0) => lower\_lim(0),

vectorB(3) => q\_aux(3),

vectorB(2) => q\_aux(2),

vectorB(1) => q\_aux(1),

vectorB(0) => q\_aux(0),

equal => clear\_aux); --comparator of lower limit

not\_clear\_aux <= NOT(clear\_aux OR Clr); -- we need to define it this way so we can have a globally static expression

not\_preset\_aux <= NOT(preset\_aux); -- we need to define it this way so we can have a globally static expression

COMP\_FF: ffjk port map(clk => Clk,

J => '0',

K => '0',

P => not\_preset\_aux,

C => not\_clear\_aux,

q => M); -- the outputs of the comparators go to the jk flip flop

-- FF00 is the flip flop responsible for Q(0) bit

FF00: ffjk port map(clk => Clk,

J => En,

K => En,

P => load\_lower\_lim(0),

C => load\_clear(0),

q => q\_aux(0)); -- first jk flipflop

ckt01 <= ((q\_aux(0) AND NOT(M)) OR (NOT(q\_aux(0)) AND M)) AND En; --circuit between FF0 and FF1

-- FF01 is the flip flop responsible for Q(1) bit

FF01: ffjk port map(clk => Clk,

J => ckt01,

K => ckt01,

P => load\_lower\_lim(1),

C => load\_clear(1),

q => q\_aux(1)); -- second jk flipflop

ckt12 <= ((q\_aux(1) AND (q\_aux(0) AND NOT(M))) OR (NOT(q\_aux(1)) AND (NOT(q\_aux(0)) AND M))) AND En; --circuit between FF0 and FF1

-- FF02 is the flip flop responsible for Q(2) bit

FF02: ffjk port map(clk => Clk,

J => ckt12,

K => ckt12,

P => load\_lower\_lim(2),

C => load\_clear(2),

q => q\_aux(2)); -- third jk flipflop

ckt23 <= ((q\_aux(2) AND (q\_aux(1) AND (q\_aux(0) AND NOT(M)))) OR (NOT(q\_aux(2)) AND NOT(q\_aux(1)) AND (NOT(q\_aux(0)) AND M))) AND En; --circuit between FF2 and FF3

-- FF03 is the flip flop responsible for Q(3) bit

FF03: ffjk port map(clk => Clk,

J => ckt23,

K => ckt23,

P => load\_lower\_lim(3),

C => load\_clear(3),

q => q\_aux(3)); -- third jk flipflop

Q(3) <= q\_aux(3);

Q(2) <= q\_aux(2);

Q(1) <= q\_aux(1);

Q(0) <= q\_aux(0);

end up\_down\_counter\_ckt;

vsim up\_down\_counter

add wave \*

force upper\_lim 1101 0, 0000 750

force load 1 0, 0 1

force lower\_lim 0101 0, 1111 750

force En 1 0, 0 328, 1 384

force Clr 0 0, 1 484, 0 488

force Clk 1 0, 0 1 -repeat 10

run 750