ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos

PMR3303 Eletrônica Digital para Mecatrônica 1º semestre de 2019

Nome: _____ Nº USP: _____

Experiência 6 MÁQUINA DE ESTADOS FINITOS (REV. A)

O objetivo desta experiência é projetar máquinas síncronas de estados finitos – também chamadas de máquinas algorítmicas, e implementá-las com dispositivos digitais. Para isso, você deve fazer uma revisão das seções 7.3 e seguintes do livro texto (Wakerly, "Digital Design – Principles and Practices").

- Estude a apostila **com antecedência**. Sua compreensão será avaliada na aula por **ARGUIÇÃO ORAL**.
- Faça os **EXERCÍCIOS** contidos na apostila e tire dúvidas com os professores **com antecedência**.
- Traga para a aula a apostila **IMPRESSA**. Os pontos importantes devem estar **destacados ou grifados**.

PARTE A TEORIA

6.1 Máquina Moore: Controle de Nível

Nesta experiência, vamos implementar uma máquina de estados para controlar o nível do reservatório mostrado na Figura 6.1. O reservatório é abastecido por meio de duas bombas: a bomba principal BP e a bomba auxiliar BA. A vazão de saída não é controlada e varia aleatoriamente.

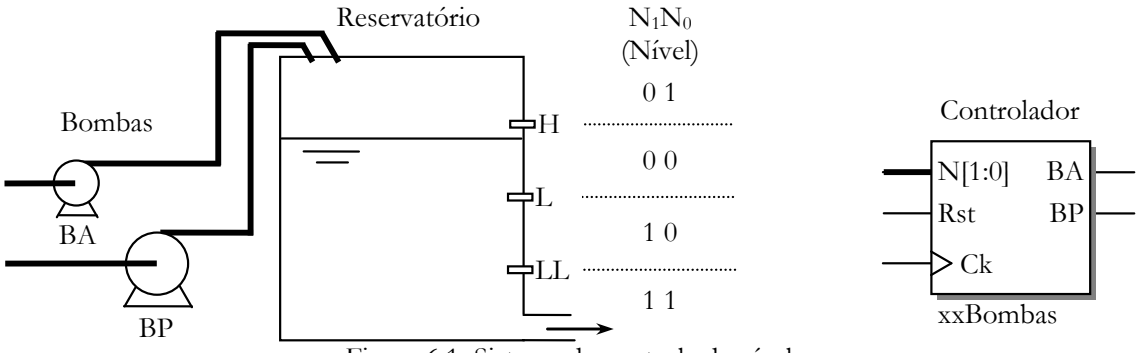


Figura 6.1 Sistema de controle de nível

O controlador das bombas (símbolo xxBombas) é um circuito sequencial, implementado na forma de uma máquina de estados finitos. Um sensor faz a leitura do nível em quatro faixas e as codificada em 2 bits, N[1:0], usando código Gray. As saídas BA e BP acionam as bombas. Ck é a entrada de *clock* do controlador e Rst é uma entrada de *reset* síncrono.

Sempre que o nível se encontrar abaixo de L ($N_1N_0 = 10$ ou $N_1N_0 = 11$), a bomba principal BP deve ser ligada, e assim deve permanecer até que o nível H seja ultrapassado ($N_1N_0 = 01$). A bomba auxiliar BA deve ser acionada sempre que o nível cair abaixo de LL ($N_1N_0 = 11$), devendo permanecer assim até que o nível volte a ficar acima de L ($N_1N_0 = 00$ ou $N_1N_0 = 01$).

6.1.1 Diagrama de estados

O controlador das bombas pode ser implementado usando-se os três estados mostrados na Tabela 6.1.

Tabela 6.1 Estados do controlador xxBombas

Estado	Descrição	BA	BP
SA	Aguardando o nível cair	desligada	desligada
SB	Enchimento normal	desligada	ligada
SC	Enchimento rápido	ligada	ligada

A Figura 6.2 mostra o diagrama de estados do controlador. O estado SA é o estado inicial. As condições das transições estão indicadas na forma N_1N_0, onde X representa um bit irrelevante (*don't care*). Nos estados, as saídas são dadas na forma BA BP – note que o controlador será uma máquina do tipo Moore, dado que as saídas são constantes em cada estado e não variam com as entradas.

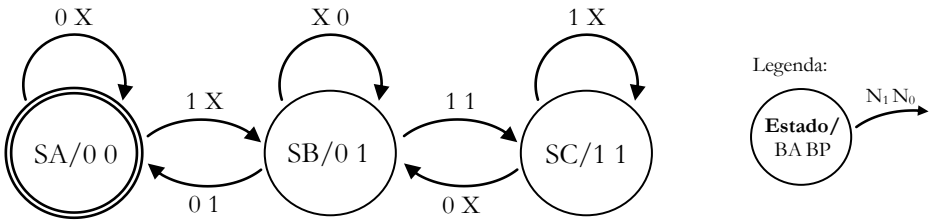


Figura 6.2 Diagrama de estados do controlador *xx*Bombas

Repare que esse controlador consegue lidar com variações nas leituras de nível que seriam improváveis a princípio. Por exemplo, suponha que ele se encontre no estado SA, em que normalmente o nível se encontra acima de L (N_1N_0 = 00 ou N_1N_0 = 01), e no instante seguinte os sensores indiquem que o nível caiu abaixo de LL (N_1N_0 = 11), pulando a faixa entre L e LL (N_1N_0 = 10).

Mesmo que isso aconteça, o controlador não se perderá: no instante em que ler N_1N_0 = 11, passará do estado SA para o estado SB, ligando apenas a bomba principal, e no *clock* seguinte cairá no estado SC onde acionará as duas bombas. Analogamente, o controlador percorrerá o caminho inverso caso leia N_1N_0 = 01 estando no estado SC, e passará primeiro para o estado SB e em seguida para SA, desligando uma bomba de cada vez. Por sua vez, o estado SB também é capaz de lidar com entradas consecutivas inesperadas:

- ao ler N_1N_0 igual a 10 seguido de 01, o controlador irá para o estado SA; e
- ao ler 00 seguido de 11, irá para o estado SC.

Alguma falha poderia levar a essas situações, mas há uma condição que ocorre mesmo em operação normal: quando o controlador é inicializado com o reservatório completamente vazio. Nessa situação, o controlador cairá no estado SA e neste estado a primeira leitura de nível será N_1N_0 = 11 (sutil, não?).

6.1.2 Codificação dos estados e tabela de transição

No laboratório, implementaremos o controlador xxBombas usando *flip-flops*, e para isso precisamos codificar os estados em binário. Para codificar os três estados, precisamos de no mínimo dois bits, Q_1 e Q_0.

Como podemos escolher a codificação livremente, vamos adotar a codificação mostrada na Tabela 6.2 que tem a vantagem de reproduzir a sequência das linhas do mapa de Karnaugh, o que facilitará o trabalho de gerar as equações de excitação dos *flip-flops* no passo seguinte. A Tabela 6.2 mostra também a transcrição do diagrama de estados para tabela de transições com os estados já codificados. Por segurança, caso a máquina caia no estado não utilizado (Q_1Q_0 = 10), deve-se desligar as saídas e voltar para o estado SA.

Tabela 6.2 Codificação dos estados e tabela de transição do controlador.

Codificação

Estado	$Q_1\,Q_0$
SA	0 0
SB	0 1
SC	1 1
---	1 0

Transições e Saídas

Estado Atual $(Q_1\,Q_0)^k$	Próximo Estado $(Q_1\,Q_0)^{k+1}$ $N_1\,N_0$ = 0 0	$N_1\,N_0$ = 0 1	$N_1\,N_0$ = 1 1	$N_1\,N_0$ = 1 0	Saídas BA BP
0 0	0 0	0 0	0 1	0 1	0 0
0 1	0 1	0 0	1 1	0 1	0 1
1 1	0 1	0 1	1 1	1 1	1 1
1 0	0 0	0 0	0 0	0 0	0 0

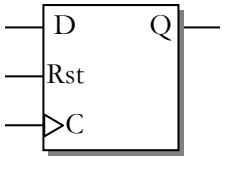
6.1.3 Equações das saídas

Inspecionando o valor das saídas BA e BP na Tabela 6.2, chegamos sem dificuldade às expressões lógicas de ativação das bombas em função dos bits de codificação dos estados Q_1 e Q_0:

$$BA = Q_1.Q_0 \quad e \quad BP = Q_0 \ . \tag{6.1}$$

6.1.4 Equações de excitação dos *flip-flops*

No circuito do controlador, usaremos o *flip-flop* D com entrada de reset síncrono mostrado na Figura 6.3.



Rst	C	D	Q^{k+1}	
1	↑	X	0	(*reset* síncrono)
0	↑	0	0	(copia D)
0	↑	1	1	(copia D)

Figura 6.3 Flip-flop D com *reset* síncrono

Nesse caso, a lógica de excitação (isto é, como levar o flip-flop de um estado para outro) se resume à

$$Q^{k+1} = D \ ,$$

ou seja, o estado do *flip-flop* será igual ao nível presente na entrada após a borda de *clock*.. Assim, os mapas de Karnaugh das entradas D_1 e D_0 dos flip-flops saem diretamente da tabela de transição, como mostra a Figura 6.4.

D_1

Q_1Q_0 \ N_1N_0	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	1	1
10	0	0	0	0

D_0

Q_1Q_0 \ N_1N_0	00	01	11	10
00	0	0	1	1
01	1	0	1	1
11	1	1	1	1
10	0	0	0	0

Figura 6.4 Mapas de Karnaugh das entradas dos flip-flops

Com isso, as expressões lógicas das entradas dos flip-flop ficam

$$D_1 = N_1.N_0Q_0 + N_1.Q_1Q_0 \ e \ \ D_0 = N_1.\overline{Q_1} + Q_1.Q_0 + \overline{N_0}.Q_0 \ . \qquad (6.2)$$

6.2 Controlador das bombas em Verilog

A Figura 6.5 mostra a descrição em Verilog do flip-flop D que usaremos.

```
module FF_DR (
    input Ck, Rst, D,
    output Q );

    reg qr; // net auxiliar

    always @ (posedge Ck)
        if (Rst) qr <= 0;    // Reset síncrono
        else qr <= D;        // Regsitra a entrada D

    assign Q = qr;  // Atualiza a saída Q
endmodule
```

Figura 6.5 Módulo FF_DR: flip-flop D com entrada de *reset* síncrono.

Além dele, precisaremos de circuitos lógicos para gerar os sinais BA e BP de ativação das bombas e os sinais D_1 e D_0 de excitação dos *flip-flops*. Esses circuitos podem ser implementados facilmente em Verilog com atribuições do tipo *assign*. A Figura 6.6 mostra o esqueleto do módulo xxBombas, que implementa o controlador.

```
module xxBombas (
    input Ck, Rst, [1:0] N,
    output BA, BP );

    wire D1, D0, Q1, Q0;    // Entradas e saídas dos flip-flops

    FF_DR U0 ( Ck, Rst, D0, Q0 );   // Instanciação dos flip-flops
    FF_DR U1 ( Ck, Rst, D1, Q1 );
    // Definição dos bits D0, D1 e saídas BA, BP
    // (por sua conta...)

endmodule
```

Figura 6.6 Módulo *xx*Bombas (incompleto): controlador das bombas BA e BP.

Como pode ver, o módulo *xx*Bombas está incompleto. Cabe a você completá-lo

6.2.1 Simulação do controlador

No laboratório, usaremos o módulo Bombas_sim mostrado na Figura 6.7 para simular o circuito do controlador de nível. **Em paralelo** com a simulação do módulo *xx*Bombas, rodam também dois blocos *initial* : o primeiro gera os pulsos de *clock* (Ck) e o segundo o sinal de reset (Rst) e a sequência de leituras de nível (N[1:0]).

```
`timescale 100us / 1us
module Bombas_sim(  );
    integer k=0;        // Contador de clocks
    reg Ck=0, Rst=0;    // Clock e reset
    reg [1:0] N=2'b0;   // Leitura de nível
    wire BA, BP;        // Bombas auxiliar e principal

    xxBombas UUT (Ck, Rst, N, BA, BP ); // Instanciação do controlador

    initial begin   // Gera pulsos de clock a cada 1 ms
        $timeformat( -3, 1, "ms", 4);  // formato para imprimir $time
        $monitor("Time %t: Ck=%b Rst=%b N=%b => BA=%b BP=%b",
                 $time, Ck, Rst, N, BA, BP);
        for ( k = 0; k <= 7; k = k+1 ) begin
            Ck = 1; #5;  // Ck=1 por meio período (0,5 ms)
            Ck = 0; #5;  // Ck=0 por meio período (0,5 ms)
        end  // for k
        $finish;
    end  // initial

    initial begin   // Vetor de teste
        Rst = 1;  #10;    // Ck 1: reset
        Rst = 0;  #5;     // Alinha o tempo com a descida do clock
        // Nível muda entre duas bordas de clock
        N = 'b01;  #10;  // Ck 2: nível acima de H
        N = 'b11;  #10;  // Ck 3: nível abaixo de LL
        N = 'b11;  #10;  // Ck 4
        N = 'b01;  #10;  // Ck 5: nível acima de H
        N = 'b01;  #10;  // Ck 6
    end  // initial
endmodule
```

Figura 6.7 Módulo Bombas_sim: *testbench* do controlador das bombas.

Relembrando: a diretiva "`*timescale* 100us / 1us" (no topo da listagem) fixa em 100 µs a unidade de tempo da simulação, e em 1 µs a precisão (a letra 'u' representa a letra grega 'µ').

No começo do segundo bloco *initial*, o comando "Rst = 1; #10;" leva o sinal de *reset* a um e avança a simulação 1 ms (10 unidades de tempo). Isso engloba o tempo necessário para zerar os registradores da FPGA após ser inicializada (*power-on reset*, de aproximadamente 100 ns para a FPGA que vamos usar).

O comando seguinte, "Rst = 0; #5;" zera o sinal de *reset* e avança o tempo em 0,5 ms, que corresponde à meio período de *clock*. Os avanços seguintes serão de um período de clock (1 ms) e com isso alinhamos as mudanças das entradas com as bordas de descida do *clock*, em t = 1,5 ms, 2,5 ms, etc. Simulamos assim sinais externos que não estão sincronizados com as bordas de subida do *clock*.

O vetor de teste começa com o nível acima de H no *clock* seguinte o faz cair abaixo de LL, permanece lá por 2 *clocks* e retorna em seguida para H (confira). É uma sequência improvável de leituras de nível, conforme comentamos no começo da apostila. Apesar disso, ela tem a vantagem de fazer o controlador passar por todos os estados (confira isso também!). No laboratório, ficará por sua conta fazer a simulação de uma sequência mais completa.

6.2.2 Circuito de teste do controlador das bombas

Para testar o controlador na placa Basys3, vamos usar o módulo de topo mostrado na Figura 6.8. O módulo ClockHz gera um sinal de *clock* de 1 Hz que pode ser monitorado pelo led LD15. Usamos um *clock* bem lento para podermos acompanhar as mudanças de estado do controlador com mais facilidade.

As chaves SW1 e SW0 fornecem os bits N_1 e N_0 de leitura de nível. Os sinais de ativação das bombas BA e BP acionam os leds LD14 e LD13. Por fim, o botão BTND está ligado à entrada de *reset* do controlador.

```
module BombasTest_top(
    input clk, btnD, [1:0] sw,
    output [15:13] ld );

    wire CkHz, Rst, BA, BP;
    wire [1:0] N;

    ClockHz U0 ( clk, 0, CkHz );  // Clock 1 Hz
    assign ld[15] = CkHz;

    assign N[1:0] = sw[1:0];
    assign Rst = btnD;
    xxBombas U1 ( CkHz, Rst, N, BA, BP );  // Instanciação do controlador
    assign ld[14] = BA;
    assign ld[13] = BP;
endmodule
```

Figura 6.8 Módulo BombasTest_top: circuito de teste do controlador das bombas

6.3 Diagrama de Estados em Verilog

A metodologia clássica de projeto de máquinas de estados finitos que usamos remonta aos duros tempos em que se podia contar apenas com flip-flops e portas lógicas discretas. Atualmente, linguagens como o Verilog oferecem muitos recursos para se implementar máquina de estados por meio de construções de alto nível.

```
module BombasDE (
    input Ck, Rst, [1:0] N,
    output BA, BP );

    parameter SA = 2'b00, SB = 2'b01, SC = 2'b11;   // Estados
    parameter SZERO = SA;                           // Estado inicial
    reg [1:0] Sreg=SZERO, Snext;  // Estado atual e próximo estado
    reg BAr, BPr;                 // nets auxiliares

    always @ (*) begin  // Circuito de próx. estado e saídas
        Snext = Sreg;   // Por default, mantém o estado atual
        case ( Sreg )
            SA : begin
                BAr = 0;  BPr = 0;
                if ( N[1] ) Snext = SB;
            end  // SA
            SB : begin
                BAr = 0;  BPr = 1;
                if ( N == 2'b11 ) Snext = SC;
                else if ( N == 2'b01 ) Snext = SA;
            end  // SB
            SC : begin
                BAr = 1;  BPr = 1;
                if ( ~N[1] ) Snext = SB;
            end  // SC
            default : begin // Estados não usados
                BAr = 0;  BPr = 0;
                Snext = SZERO;
            end  // default
        endcase
    end  // always

    always @ (posedge Ck) begin // Atualiza estado atual
        if ( Rst ) Sreg <= SZERO;
        else Sreg <= Snext;
    end  // always

    assign BA = BAr;
    assign BP = BPr;
endmodule
```

Figura 6.9 Módulo BombasDE: diagrama de estados do controlador das bombas em Verilog

Por exemplo, o módulo BombasDE da Figura 6.9 descreve o diagrama de estados do controlador das bombas diretamente em Verilog.

A variável Sreg armazena o estado atual do controlador, e Snext é usada para definir o próximo estado.

As declarações "parameter ..." servem para dar nomes às constantes que codificam os estados para tornar o código mais legível. Por exemplo, o estado 00 é representado pelo identificador "SA" – o compilador se encarregará de substituir todas as ocorrências de "SA" pelos bits 00. Além dos estados SA, SB e SC usados no diagrama de estados da Figura 6.2, define-se também o estado inicial SZERO, o qual leva ao estado SA.

O primeiro bloco *always* descreve os circuitos combinacionais que geram os sinais de saída e definem o próximo estado em função do estado atual armazenado em Sreg. Por exemplo, se Sreg for igual à constante SA (ou seja, 00), será executado o primeiro bloco da estrutura *case*, que zera BA e BP e carrrega o próximo estado na variável Snext dependendo do valor do bit N[1] (que codifica o nível).

A condição "*default* :" faz com que o controlador zere as saídas e retorne ao estado inicial caso venha a cair em um estado não previsto.

O segundo bloco *always* (ao fim do módulo) atualiza o estado atual sincronamente com a borda de subida do sinal de *clock* Ck, além de fazer o *reset* síncrono do controlador caso Rst = 1.

Repare que este modelo pode ser usado para máquinas Moore ou Mealy. Em máquinas Moore, as saídas são constantes (zero ou um) em cada estado, como é o caso do controlador das bombas. Para uma máquina Mealy, as saídas em cada estado serão expressões lógicas envolvendo sinais de entrada, como se verá mais a frente.

Nota: na verdade, a declaração "parameter" serve mesmo para parametrizar um módulo (definindo, por exemplo, o número N de bits de um contador), de tal forma que parâmetros podem ser passados para o módulo quando este é instanciado (veja um exemplo na página 309 do livro texto). Em Verilog, também é possível definir constantes na forma "`define SA 2'b00", semelhante a adotada em linguagem C ( com a diferença da crase inicial). Neste caso, preferimos usar "parameter" porque o escopo do identificador definido por ele se restringe ao módulo em que são declarados, ao contrário de "`define" que o compilador mantém ativo mesmo após a declaração "endmodule" (leia mais a respeito na página 305 do livro texto).

6.4 Fluxo de Dados e Unidade de Controle

O controlador de nível que acabamos de ver é um exemplo de circuito sequencial bem simples: a máquina de estados tem poucos estados e o circuito resultante por si só resolve o problema. Na prática, a maioria dos problemas envolve tantos estados e variáveis de entrada que inviabilizam o projeto baseado em diagramas de estados e flip-flops. Portanto, deve haver outro caminho mais prático...

De fato, aplicações reais devem ser atacadas de forma estruturada, dividindo-se o problema em dois grandes blocos: o Fluxo de Dados (FD, ou *datapath*) e a Unidade de Controle (UC). O FD armazena e processa informações, e a UC controla o funcionamento do FD.

Para tanto, a UC envia um conjunto de sinais de controle para o FD (a chamada *palavra de controle*), que por sua vez retorna para a UC um conjunto de sinais de status (*flags*). Entradas e saídas que interfaceiam o sistema com o mundo externo podem entrar e sair tanto da UC como do FD.

Cada um desses blocos pode ser projetado com técnicas apropriadas. A arquitetura do fluxo de dados geralmente envolve registradores, decodificadores, multiplexadores e barramentos. Já a unidade de controle é projetada na forma de uma máquina de estados finitos.
Vamos aplicar esses conceitos no nosso próximo projeto.

6.5 Temporizador Programável

A Figura 6.10 mostra o símbolo do temporizador TmrB4 que vamos projetar. Ao ser disparado, o temporizador carrega o número *n* em binário presente nas entradas D[3:0] e conta os pulsos do sinal de *clock* Ck.

A saída T vai a um no momento do disparo e se mantém durante a contagem, gerando um pulso de duração igual a *n* ciclos de *clock*. O valor de *n* vale de 0 a 15. Em particular, para *n* igual a zero (D[3:0] = 0000), T deve permanecer em um por 16 períodos de *clock*, e não zero *clocks* (só para complicar um pouco nossa especificação...).

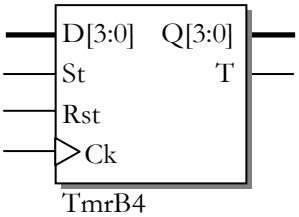


Figura 6.10 Temporizador programável de 4 bits

As saídas Q[3:0] contam os pulsos de *clock* da temporização regressivamente de *n* a 0, retornando a 1111 após o término de uma temporização. O disparo deve acontecer na primeira borda de *clock* em que a entrada St (*Start*) é igual a 1. A entrada Rst zera o temporizador sincronamente: as saídas Q[3:0] vão a 1111, T vai a zero e as demais entradas são ignoradas.

6.5.1 Fluxo de dados e unidade de controle do temporizador

Veja que seria bem trabalhoso projetar o temporizador TmrB4 inteiramente com uma máquina de estados finitos: teríamos no mínimo 16 estados para atender a maior temporização possível e 5 bits de entrada (St e D[3:0]).

Comecemos pelo fluxo de dados. A contagem de pulsos de *clock* pode ser feita por um circuito especializado nisso: um contador binário regressivo, que conta de 1111 a 0000. O contador deve ser capaz de carregar um valor inicial *n* para que se possa alterar o módulo da contagem (essa capacidade é conhecida como *carga em paralelo*). Vamos chamar esse contador de CBD4RL.

Com isso, resta projetar uma unidade de controle que faça o contador carregar o valor *n* no momento do disparo e depois aguarde o contador decrementar até 0000. A Figura 6.11 ilustra como ficará o circuito, e como o contador CBD4RL implementa sozinho o fluxo de dados do temporizador. Em resumo, a palavra de controle a ser gerada pelo bloco xxTmrUC deve conter os bits En (que habilita o contador) e Ld (que controla a carga inicial do contador). Como mostra a Tabela 6.3, o contador faz RC = 1 para sinalizar que a contagem regressiva chegou a zero. Esse sinal é o bit de status que deve ser realimentado para a UC.

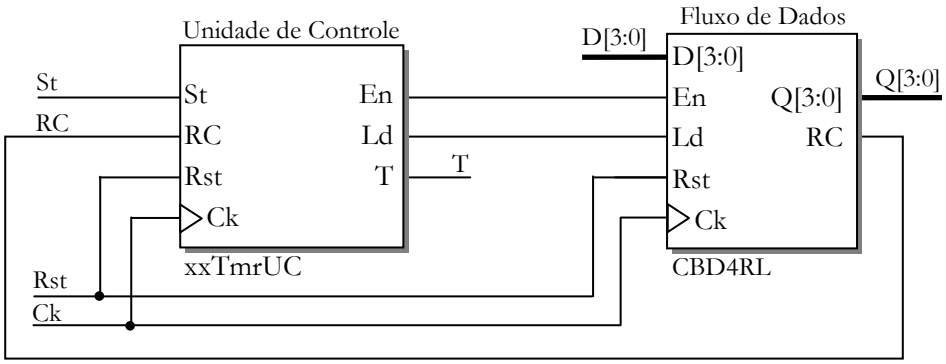


Figura 6.11 Diagrama do circuito do temporizador

Tabela 6.3 Operação do contador regressivo CBD4RL

Rst	En	Ld	Ck	$Q[3:0]^{k+1}$	RC	
1	X	X	↑	1111	0	(reset síncrono)
0	0	X	X	$Q[3:0]^{k}$	0	(contador desabilitado)
0	1	0	↑	$Q[3:0]^{k} - 1$	Q == 0000	(decremento módulo 16)
0	1	1	↑	D[3:0]	Q == 0000	(carga em paralelo)

6.5.2 Carta de tempos

A carta de tempos da Figura 6.12 ilustra o funcionamento do temporizador usando *n* = 3 como exemplo. Note que En e Ld são acionados tão logo St vai a 1, para habilitar a carga do contador na borda de *clock* 0. A saída T permanece em 1 por três períodos de *clock*, e retorna a 0 assim que RC vai a 1. O sinal En permanece em 1 por um período de *clock* a mais para garantir que a contagem retorne a 1111.

O sinal St é ignorado até que a temporização em curso termine – ou seja, o *timer* não é *redisparável*.

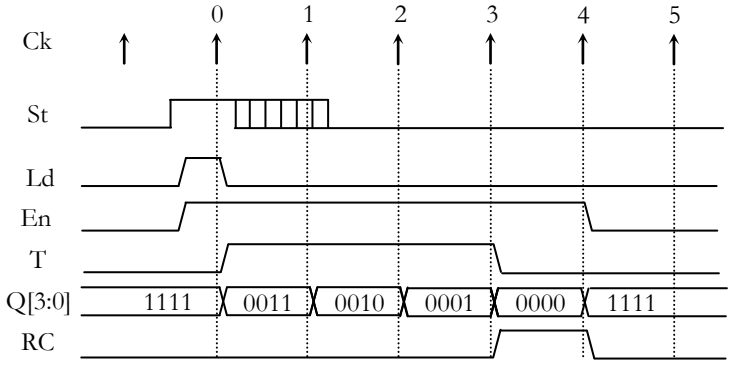


Figura 6.12 Carta de tempos do temporizador para *n* = 3 (D[3:0] = 0011)

6.6 Máquina Mealy: unidade de controle do temporizador

A unidade de controle deve ser implementada por uma máquina de estados finitos do tipo Mealy. Mas por que uma máquina Mealy, e não Moore?

É necessário usar o modelo Mealy neste caso porque os sinais Ld e En devem responder imediatamente a variações da entrada St, de modo a carregar o contador no primeiro *clock* em que tivermos St = 1. Uma máquina Moore teria que incluir um estado adicional para fazer En = 1 e Ld = 0, já que nela as saídas são constantes em cada estado, e o contador seria carregado apenas no *clock* seguinte (um *clock* seria perdido). O mesmo raciocínio se aplica ao sinal T, que deve permanecer em 1 por exatos *n* ciclos de *clock* e por isso deve voltar a 0 assim que RC vai a 1 (confira na carta de tempos).

Precisamos de pelo menos dois estados: WAIT – em que a unidade aguarda o pedido de início de temporização, e RUN – em que o circuito se encontra temporizando, aguardando o final da contagem.

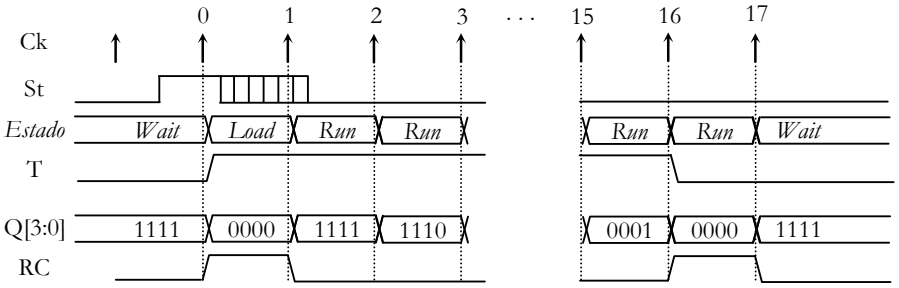


Figura 6.13 Caso particular em que *n* = 0 (D[3:0] = 0000)

No entanto, quando se tem *n* = 0, o sinal RC do contador vai a um após a carga em paralelo. Por isso, precisamos acrescentar um estado intermediário para iimpede que a contagem se encerre logo no primeiro *clock*. Esse estado, que chamaremos de LOAD, faz o temporizador ser decrementado uma vez antes de entrar no estado RUN. Portanto, o temporizador conta 16 ciclos de *clock* para *n* = 0, atendendo dessa forma a especificação dada (sutil, não?).

6.6.1 Diagrama de estados da unidade de controle

A Figura 6.14 mostra um possível diagrama de estados para a máquina de estados finitos xxTmrUC, capaz de gerar os sinais de controle do temporizador. As condições de entrada e o valor das saídas estão indicadas nas transições na forma "RC St / En Ld T", e X indica que o valor do bit é irrelevante (*don't care*).

No laboratório, vamos implementar esse diagrama de estados diretamente em Verilog.

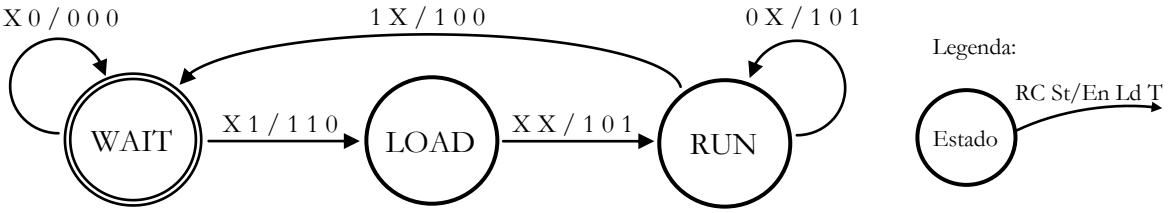


Figura 6.14 Diagrama de estados da unidade de controle do temporizador xxTmrUC.

6.6.2 Equações dos bits de saída

Para codificar em Verilog uma máquina Mealy, é preciso ter as equações dos bits de saída em função dos sinais de entrada para cada estado. Com poucas entradas e saídas, é possível extraí-las diretamente do diagrama de estado ou das cartas de tempo. Caso contrário, o jeito é separar as transições que tem a **mesma origem** para se determinar a relação entre os sinais entrada/saída.

Tomando como exemplo as duas transições que saem do estado WAIT na Figura 6.14, podemos montar a Tabela 6.4. Nela, fica fácil ver que En = St, Ld = St e T = 0.

Tabela 6.4 Entradas/saídas no estado WAIT

Entradas		Saídas		
RC	St	En	Ld	T
X	0	0	0	0
X	1	1	1	0

Basta então repetir esse procedimento para os estados LOAD e RUN. Resumindo, as equações de saída são:

- **Estado WAIT**: En = St, Ld = St T = 0.
- **Estado LOAD**: En = 1, Ld = 0 T = 1.
- **Estado RUN**: En = 1, Ld = 0 T = ~RC

6.7 Implementação do temporizador em Verilog

6.7.1 Circuito do temporizador TmrB4

Dividindo-se o circuito do temporizador entre o módulo da unidade de controle (xxTmrUC) e outro que implementa o fluxo de dados (contador CBD4RL), pode-se descrevê-lo como mostra a Figura 6.15.

```
module TmrB4 (
    input Ck, Rst, St, [3:0] D,
    output [3:0] Q, output T );

    wire En, Ld;   // Palavra de controle
    wire RC;       // Flags

    xxTmrUC U1 ( Ck, Rst, St, RC, En, Ld, T ); // Unidade de controle
    CBD4RL U2 ( Ck, Rst, Ld, En, D, Q, RC );   // Fluxo de dados
endmodule
```

Figura 6.15 Módulo TmrB4: temporizador programável

6.7.2 Simulação funcional do temporizador TmrB4

Para fazer a simulação funcional (sem considerar atrasos), usaremos o módulo *testbench* mostrado na Figura 6.16.

O período de Ck é dado pela constante TCK, definida na primeira linha, e vale 100 ns. Desta vez, declaramos constantes por "`defines" (ao invés de "parameters") para que possam valer em outros módulos que forem incluídos na simulação (e também para mostrar um exemplo de como usar "`defines" em Verilog). Dessa forma também normalizamos todos os intervalos de tempos usados na simulação pelo período do *clock*.

```
`define TCK 100  // Perído do clock (unidade definida por `timescale)
`timescale 1ns / 1ps  // Unidade de tempo / precisão da simulação

module TmrB4_sim (    );
    reg Ck=0, Rst=0, St=0;  // Clock, Reset e Start
    reg [3:0] D=0;          // Contagem inicial do temporizador
    wire [3:0] Q;           // Contagem de saída do temporizador
    wire T;                 // Pulso de saída do temporizador
    integer k=0;            // Contador de clocks

    TmrB4 UUT ( Ck, Rst, St, D, Q, T );  // Instanciaçao do timer

    initial begin  // Gera o sinal de clock
        $timeformat(-9, 0, "ns", 4); // Formato para imprimir $time
        for (k=0; k<=7; k=k+1) begin
            $write("Time %t: Rst=%b St=%b D=%b ", $time, Rst, St, D);
            $strobe("=> Q=%b T=%b", Q, T);
            Ck = 1;  #(`TCK/2);      // Clock com período TCK
            Ck = 0;  #(`TCK/2);
        end // for k
        $finish;
    end  // initial

    initial begin   // Gera o vetor de teste
        Rst = 1;  Rst <= #(`TCK) 0;     // Pulso de reset
        #(`TCK);    // Avanço de tempo: aguarda o power-on da FPGA
        St <= 1;  St <= #(`TCK*2.5) 0;  // Pulso de start
        D <= 3;     // Temporização com D = 3
    end  // initial
endmodule
```

Figura 6.16 Módulo TmrB4_sim: *testbench* do temporizador TmrB4

O primeiro bloco *initial* gera o sinal de *clock*, com período definido pela constante TCK. Os comandos "$timeformat", "$write" e "$strobe" se destinam a imprimir o valor dos sinais, como já visto na experiência 5.

O segundo bloco *initial* gera o vetor de teste para simular um disparo do temporizador com D[3:0] = 3. Começa com "Rst <= 1; Rst <= #(`TCK) 0;" que gera um pulso de *reset* de 100 ns. Lembrando que

```
Rst <= #100 0;
```

é uma atribuição com atraso do tipo *nonblocking* : faz o sinal Rst ir a zero com um atraso de 100 ns sem bloquear a simulação. Reveja os conceitos de atribuição *nonblocking* e *blocking* na apostila da experiência 5.

O comando seguinte "#(`TCK);" avança a simulação em 100 ns para aguardar o procedimento de *power-on reset* da FPGA (que dura mais ou menos isso).

As atribuições *nonblocking* "St <= 1; St <= #(`TCK*2.5) 0;" geram o pulso de *start* que vai de 100 a 350 ns, sem que a simulação pause. Essa forma de gerar os pulsos usando atribuições *nonblocking* com atraso permite variar a largura dos pulsos sem mexer nos demais tempos da simulação. Por isso, a atribuição "D <= 3;" acontece em t = 100 ns independentemente do tempo em que Rst ou St permanecem em um. Isso será particularmente útil nesta experiência, quando formos fazer simulações temporais.

O temporizador deve disparar na borda de *clock* em t = 200 ns. A duração do pulso de St, sendo de 2,5 vezes o período do *clock*, permite testar se o temporizador não redispara na borda de *clock* em t = 300 ns.

6.7.3 Circuito de teste do temporizador

O módulo TmrB4Test_top mostrada na Figura 6.17 servirá para testar o temporizador TmrB4 na placa Basys3. As entradas estão ligadas na placa assim: Rst no botão btnD, St ao botão btnU, e entradas D[3:0] às chaves sw[3:0].

```
module TmrB4Test_top (
    input clk, btnD, btnU, [3:0] sw,
    output [15:14] ld, [3:0] an, [0:6] seg );

    wire Ck400, CkHz, Rst, St, RC, T;
    wire [3:0] D, QT, QC, AnAux;
    wire rstCB4R;

    Clock400 U1 ( clk, 1'b0, Ck400 );  // Clock de 400 Hz
    ClockHz U2 ( clk, 1'b0, CkHz );    // Clock de 1 Hz
    assign ld[15] = CkHz;

    assign St = btnU;
    assign Rst = btnD;
    assign D = sw[3:0];
    TmrB4 U3 ( CkHz, Rst, St, D, QT, T );  // Instanciação do Timer
    assign ld[14] = T;

    assign rstCB4R = St & ~T;  // reset do contador
    CB4R U4 ( CkHz, rstCB4R, T, QC, RC );  // Contador binário de 4 bits

    xxDsDrv U5 ( Ck400, QC, 4'b0000, QT, D, 1'b1, AnAux, seg ); // Display driver
    assign an = { AnAux[3:2], 1'b1, AnAux[0] }; // an[1]=1: apaga o display 1
endmodule
```

Figura 6.17 Módulo TmrB4Test_top: circuito de teste do temporizador

A temporização habilita contador CB4R (U4), que permite verificar o número de *clocks* em que T permanece em 1. O módulo xxDsDrv (U5) é o acionador dos displays de sete segmentos que implementamos na experiência anterior. Com ele, podemos acompanhar alguns valores do teste (em hexadecimal) nos displays.

O display 1 mostra o valor colocado nas chaves e o display 2 mostra as saídas Q[3:0] do temporizador. A saída do contador CB4R é mostrada no display 4.

Ao pressionar btnU (sinal St), o segundo display deve mostrar a contagem regressiva e retornar a F no final. O último display deve contar os pulsos de *clock* até T voltar a zero, e deve ficar igual ao valor mostrado no primeiro display ao final da temporização.

Os módulos Clock400 e ClockHz geram sinais de *clock* de 400 Hz e 1 Hz respectivamente, a partir do *clock* de 100 MHz disponível na placa Basys3 (listagens incluídas no final da apostila).

6.8 Simulação temporal

Até agora, as simulações que fizemos foram funcionais. Ou seja, simula-se apenas o funcionamento lógico (teórico) dos circuitos, sem levar em conta atrasos de propagação e outras características temporais.

Nesta experiência, vamos também simular o circuito considerando os atrasos de propagação – a chamada *timing simulation*. Isso é importante para verificar se o circuito funcionará na velocidade real, levando em conta a frequência do *clock* e a frequência com que as entradas mudam.

Como exemplo, vamos fazer apenas a simulação *pós-síntese*, que considera os atrasos em blocos genéricos da FPGA. A simulação que seria mais próxima do circuito real seria a *pós-implementação*, que leva em consideração a forma com que o circuito será posteriormente alocado no CI (*placement*) e os atrasos que dependem dos pinos da FPGA que foram escolhidos para a entrada e saída de sinais.

Uma simulação temporal requer alguns cuidados a mais. No laboratório, veremos que é preciso respeitar os tempos de *setup* e *hold* dos *flip-flops* para que as entradas sejam registradas corretamente. Lembrando: *setup* é o tempo mínimo que a entrada deve permanecer estável **antes** da borda de clock, e *hold* é o equivalente **após** a borda.

Como o circuito do temporizador é bem simples e usaremos uma frequência de *clock* relativamente baixa (10 MHz) em comparação com a frequência máxima com que a FPGA pode operar, a princípio não temos que nos preocupar com o tempo de *setup*. E os atrasos internos do circuito dão conta do tempo de *hold*.

O problema são os sinais externos de excitação que são gerados artificialmente nas simulações. Temos costumado a fazer com que mudem sincronizadamente com o *clock*, resultando em *hold* igual a zero! Especificamente nesta experiência, o problema aparecerá nos sinais de Rst e St do temporizador.

Assim, ao fazer a simulação temporal do temporizador, teremos que gerar esses sinais de tal forma que mudem um pouco depois das bordas de *clock* .

6.9 Pré-relatório e Relatório

O formulário que se encontra na PARTE B da apostila constitui tanto o pré-relatório como o relatório desta experiência. Existem dois tipos de itens que você deverá responder:

- **Exercícios**: constituem o *pré-relatório* e **recomendamos fortemente** que sejam ser feitos com cuidado ***antes*** da aula. Se você tiver que fazê-los ou corrigi-los no laboratório, perderá tempo e poderá não conseguir concluir todas as atividades.
- **Anotações**: constituem o *relatório* e devem ser feitas individualmente *durante* a aula.

ATENÇÃO: leia as atividades da PARTE B e não apenas os enunciados dos exercícios do pré-relatório

Muitos detalhes necessários para fazer os exercícios estão descritos nas atividades em que se inserem. Além disso, você já terá uma noção do que deverá fazer e perderá menos tempo com a leitura durante a aula.

6.10 Outros módulos em Verilog usados nesta experiência

```
module CB4R (
    input Ck, Rst, En,
    output [3:0] Q, output RC );

    reg [3:0]  qr;   // net auxiliar

    always @ (posedge Ck) begin
        if (Rst) qr <= 0;   // Reset síncrono
        else if (En)
            qr <= qr + 1;   // Incremento
        else
            qr <= qr;       // Mantém
    end  // always

    assign Q = qr;
    assign RC = En & ( Q == 4'b1111);
endmodule
```

```
module CBD4RL(
    input Ck, Rst, Ld, En, [3:0] D,
    output [3:0] Q, output RC );

    reg [3:0] qr;   // Net auziliar

    always @ (posedge Ck) begin
        if (Rst) qr <= 4'b1111; // Reset síncrono
        else case ( {En, Ld} )
            2'b10: qr = qr - 1; // Decremento
            2'b11: qr <= D;     // Carga paralela
            default: qr <= qr;  // Mantém
        endcase
    end  // always

    assign Q = qr;
    assign RC = En & ( Q == 4'b0000);
endmodule
```

```
module Clock400 (
    input C100M, Clr,
    output C400 );

    reg [16:0] k;    // Contador do clock de 100 MHz
    reg cr;          // Net auxiliar

    always @ (posedge C100M, posedge Clr) begin
        if (Clr) begin  // Clear assíncrono
            k <= 0;
            cr <= 0;
        end  // if
        else if (k < 125000)
            k <= k + 1;
        else begin
            k <= 0;
            cr <= ~cr; // Divide a freq por 2x125 mil
        end // else
    end  // always

    assign C400 = cr;
endmodule
```

```
module ClockHz (
    input Ck100M, Clr,
    output CkHz );

    integer k;  // Contador do clock de 100 Mhz
    reg cr;     // net auziliar

    always @ (posedge Ck100M, posedge Clr) begin
        if ( Clr ) begin    // Clear assíncrono
            k <= 0;
            cr <= 0;
        end
        else if (k < 50000000)
            k <= k + 1;
        else begin
            k <= 0;
            cr <= ~cr; // Divide a freq por 2x50 milhoes
        end
    end  // always

    assign CkHz = cr;
endmodule
```