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 5 LATCHES E FLIP-FLOPS (REV. A)

O objetivo desta experiência é estudar os elementos construtivos básicos dos circuitos sequenciais – os *latches* e *flip-flops*. Como aplicação, estudaremos os contadores binários. Utilizaremos *flip-flops* também para verificar na prática o problema causado pela trepidação de chaves e botões eletromecânicos nos momentos de comutação (conhecido como *bounce*).

Para esta experiência, você deve fazer uma revisão das seções 7.1, 7.2 (*latches* e *flip-flops*) e 8.4 (contadores) do livro texto. Convém ter a mão durante a experiência o manual da placa Basys 3, que foi distribuído como anexo da apostila da experiência 4.

- 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

5.1 Contadores síncronos

Uma aplicação bastante simples porém muito importante dos *flip-flops* é a construção de contadores binários. A Figura 5.1 mostra como construir um contador de um bit, usando um flip-flop D com a saída realimentada na entrada de forma invertida. A cada borda de subida do sinal de *clock* C, a saída Q se inverte de 0 para 1 e de 1 para 0, sucessivamente.

Este circuito somente funciona porque há *atrasos* envolvidos. Como D é igual a Q', o valor de Q ficaria indefinido se todos os sinais mudassem instantaneamente, dado que teríamos Q = Q' no momento da transição. No entanto, como o sinal D leva alguns nanossegundos para se inverter, o *próximo valor* da saída Q após a borda de *clock* será o inverso do *valor atual* depois de um pequeno atraso, conforme mostra a carta de tempos da Figura 5.1. Ou seja, para um dado instante *k*, temos $Q^{k+1} = (Q^k)'$.

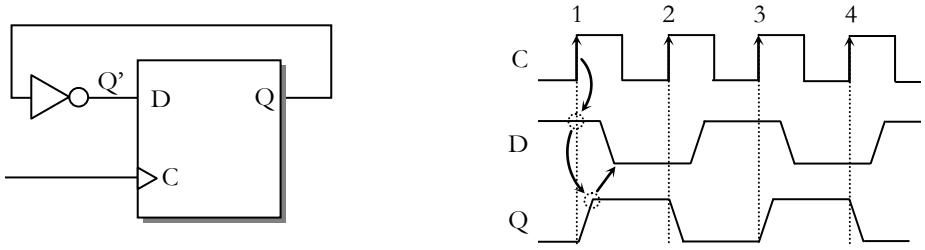
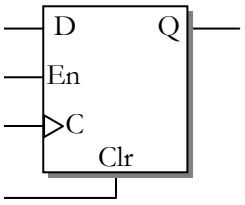


Figura 5.1 Contador de 1 bit

Repare que o sinal de *clock* C é uma onda quadrada ideal, com bordas verticais representando transições instantâneas. Já as mudanças nos sinas D e Q são representadas por bordas inclinadas para indicar que elas levam algum tempo para se consumarem. Note também que instantes antes da borda 1 do *clock*, temos Q = 0 e (devido ao inversor) D = 1. Após a borda de *clock*, a saída Q do flip-flop copia o valor de D, e depois de um atraso D se inverte.

5.1.1 Contador binário de *n* bits

Podemos cascatear vários contadores de 1 bit para formar um contador binário de *n* bits. Para isso, vamos usar um flip-flop D com *enable* mostrado na Figura 5.2. A entrada de *clear* (Clr) assíncrona, que zera a saída Q quando em 1, independentemente do sinal de *clock* C. Com Clr = 0, a saída Q é atualizada na borda de subida do *clock*, conforme o valor das entradas D e En.



Clr	C	En	D	Q^{k+1}	
1	X	X	X	0	(clear assíncrono)
0	↑	0	X	Q^k	(mantém)
0	↑	1	0	0	(copia D)
0	↑	1	1	1	(copia D)

Figura 5.2 Flip-flop D com *enable*

Como exemplo, vamos analisar a contagem com três bits, $Q_2Q_1Q_0$. Ciclicamente, a contagem fica:

$$Q_2Q_1Q_0 : 000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000 \rightarrow ...$$

Observe que, após cada incremento,

- O bit menos significativo (Q_0) sempre se inverte;
- O bit Q_1 se inverte quando $Q_0 = 1$ antes do incremento;
- e Q_2 se inverte quando $Q_1Q_0 = 11$ antes do incremento

O que ocorre é que o *i*-ésimo bit (Q_i) de um contador deve ser incrementado quando a contagem com i–1 bits atinge o máximo. Como estamos lidando com dígitos binários, incrementar um bit equivale a invertê-lo. Em resumo, deve-se habilitar a inversão de Q_i quando TODOS os bits menos significativos que ele forem iguais a 1, ou seja, se verificar a condição $Q_{i-1}.Q_{i-2}\ldots.Q_0 = 1$, com $i > 0$.

Usando a entrada de habilitação do flip-flop D da Figura 5.2 para controlar a inversão dos bits, podemos construir um contador de *n* bits como mostra a Figura 5.3. O sinal En foi incluído para se poder habilitar e desabilitar todo o contador.

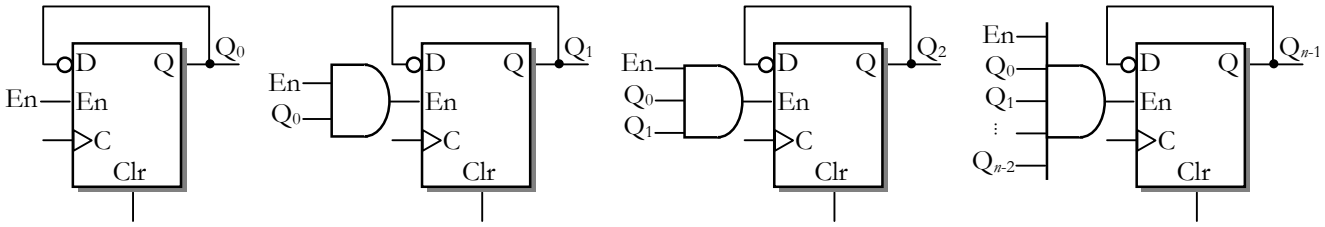
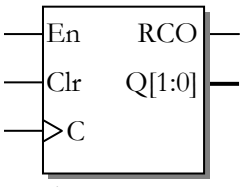


Figura 5.3 Construção de um contador binário de *n* bits usando flip-flops D com entrada de *enable*

5.2 Contador de dois bits CB2

Vamos definir e detalhar o contador binário de dois bits, denominado CB2, que usaremos nesta experiência. O símbolo e tabela característica desse contador se encontram na Figura 5.4.

A entrada En (*enable*) habilita a contagem quando em 1, fazendo as saídas Q_1Q_0 contarem ciclicamente de 00 a 11 a cada borda de subida do sinal de *clock* C. A entrada Clr zera o contador assincronamente, independentemente das entradas C e En. A saída RCO (*Ripple Carry Output*) vale 1 quando a contagem atinge o valor máximo (ou seja, 11) e o contador está habilitado (En = 1). Esta saída serve para fazer o cascateamento do contador, como veremos em seguida.



Clr	En	C	$Q[1:0]^{k+1}$	RCO	
1	X	X	0	0	(clear assíncrono)
0	0	X	$Q[1:0]^k$	0	(contagem desabilitada)
0	1	↑	$Q[1:0]^k + 1$	**Nota 1**	(incremento módulo 4)

Nota 1: RCO = $En.Q_1.Q_0$, independentemente de *clock* C

Figura 5.4 Contador síncrono de dois bits CB2, com *clear* assíncrono e *enable*

A Figura 5.5 mostra o diagrama lógico do contador. Por ser um circuito síncrono, o sinal de *clock* é o mesmo para os dois flip-flops. Estando habilitado com En = 1 e Clr = 0, o primeiro flip-flop (U0) inverte a saída Q_0 a cada borda de *clock*. O segundo flip-flop (U1) inverte a saída Q_1 sempre que a anterior for 1, ou seja, quando o bit menos significativo atinge seu final de contagem. A saída RCO é gerada seguindo a tabela da Figura 5.4.

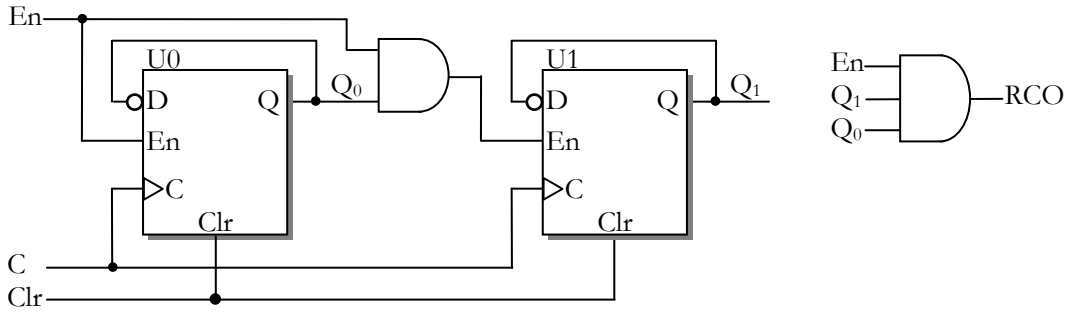


Figura 5.5 Contador CB2 construído com flip-flops D

A carta de tempos da Figura 5.6 ilustra o funcionamento do contador. Para deixá-la mais concisa, representamos as bordas de subida do sinal de *clock* por um trem de impulsos, e os bits Q_1 e Q_0 são representados pelo barramento condensado Q[1:0], no qual indicamos apenas as transições e o valor da contagem em binário.

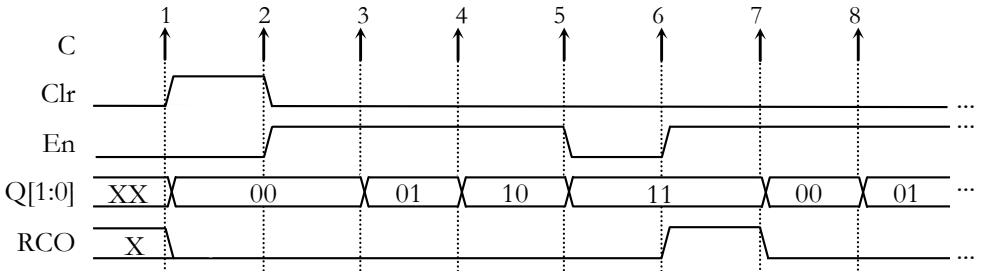


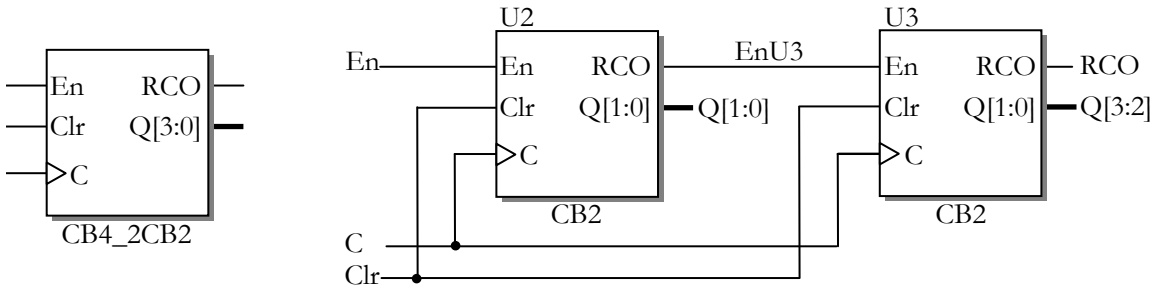
Figura 5.6 Carta de tempos do contador de dois bits CB2D

A carta retrata um fato comum em circuitos síncronos. Repare que as entradas Clr e En também variam sincronamente com a borda 2 do sinal de *clock*: nesse caso, deve-se considerar os níveis no momento **anterior** à borda (ou seja, Clr = 1 e En = 0), e por isso a contagem se mantém em zero. No instante 5, o contador ainda está habilitado e incrementa Q[1:0], pois o sinal En muda somente *após* a borda de *clock*. Analogamente, o contador ainda está desabilitado no instante 6, voltando a incrementar somente no instante 7. Concluindo: em circuitos síncronos, considere o valor presente nas entradas no momento *anterior* à borda de *clock*.

A contagem em Q_1Q_0 atinge 11 na borda 5 do *clock*, ao mesmo tempo em que o contador é desabilitado (En = 0), fazendo com que a saída RCO permaneça em zero. A saída RCO vai para 1 somente após a borda 6, em que o contador é novamente habilitado (En = 1).

5.3 Cascateamento de contadores síncronos

Para cascatear contadores, basta ligar a saída RCO do contador menos significativo à entrada En do contador seguinte, como mostra a Figura 5.7. Esse é o contador de quatro bits CB4_2CB2 que usaremos no laboratório. O sinal de *clock* deve ser o mesmo para todos os contadores para que funcionem sincronamente.



a) Símbolo b) Construção com dois contadores de dois bits em cascata

Figura 5.7 Contador de quatro bits CB4_2CB2

Fazendo a entrada En = 0, o primeiro contador (U2) é desabilitado e o sinal enU3 também vai a zero, garantindo que o segundo contador (U3) também seja desabilitado. Quando En estiver ativo e o primeiro contador atingir o final de contagem (Q_1Q_0 = 11), teremos EnU3 = 1 e o segundo contador será habilitado. Na borda seguinte do *clock*, as saídas Q_3Q_2 serão incrementadas juntamente com Q_1Q_0.

A carta de tempos da Figura 5.8 ilustra o mecanismo de cascateamento (usamos os mesmos sinais de entrada da Figura 5.6). Na borda 5 de *clock,* o primeiro contador é desabilitado, desabilitando também o segundo (EnU3 = 0), e a contagem de ambos se mantém a mesma na borda 6. No instante 7 ambos estão habilitados e a contagem nos bits Q[3:0] passa de 0011 para 0100.

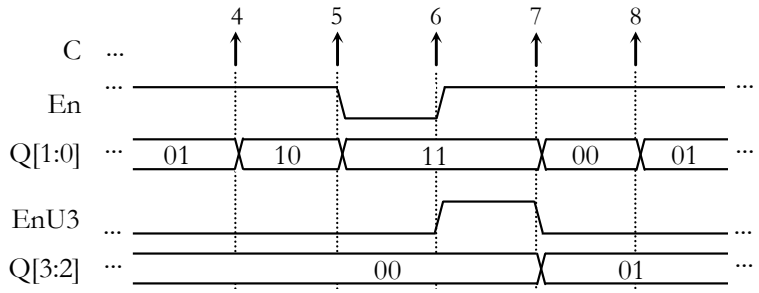


Figura 5.8 Carta de tempos resumida do contador cascateado da Figura 5.7

5.4 Circuitos sequenciais em Verilog

A descrição de circuitos sequenciais em Verilog envolve alguns detalhes que não são muito óbvios – coisas como entradas sensíveis a borda, *clear* assíncrono, atrasos, etc. Além disso, alguns cuidados devem ser tomados para simular corretamente esses circuitos

5.4.1 Entrada de *clock* sensível a borda.

O módulo FF_D mostrado na Figura 5.9 descreve em Verilog o flip-flop D básico mostrado na Figura 5.1 A declaração "*posedge C*"na lista de sensibilidade do bloco *always* faz com que este seja executado apenas a ocorrência de uma borda de subida (*positive edge*) do sinal de *clock* (neste caso, a entrada C).

```
module FF_D(
    input C, D,
    output Q );

    reg qr;

    always @ (posedge C) begin
        qr <= D;
    end // always

    assign Q = qr;

endmodule
```

Figura 5.9 Módulo FF_D: flip-flop tipo D básico

Lembre-se que em Verilog *outputs* são definidas por default como *nets* do tipo *wire*, que não podem ser alteradas dentro de blocos procedurais. Por isso, usa-se uma *net* auxiliar qr do tipo *reg* para copiar o valor do bit D dentro do bloco *always* nas bordas de clock, e atribui-se esse valor à saída Q fora do bloco (usando *assign*).

NOTA: para criar uma entrada sensível a borda de DESCIDA, usa-se a palavra "**negedge**" na lista de sensibilidade (ao invés de "posedge").

5.4.2 Simulação de sinais registrados: atribuições *blocking* e *nonblocking*

Na experiência anterior, usamos atribuições indicadas pelo habitual operador "=". Não comentamos naquela época, mas esse é um tipo de atribuição implementada pelo Verilog denominada *blocking*. Pois em Verilog existe ainda um segundo tipo de atribuição, chamada de *nonblocking*.

Na Figura 5.9, a linha "qr <= D" atualiza qr com o valor atual da entrada D através de uma atribuição do tipo *nonblocking*, indicada pelo "<=" (atenção: neste caso, NÃO É uma comparação do tipo "menor ou igual").

Atribuições do tipo *blocking* ("=") e *nonblocking* ("<=") se comportam de formas diferentes **apenas em simulações**. Na síntese de circuito, ambas darão o mesmo resultado.

A diferença entre elas é importante na simulação de circuitos combinacionais e sequenciais. Uma atribuição *blocking* faz com que o simulador atualize o valor do sinal imediatamente e já utilize o valor atualizado nas declarações seguintes.

Por exemplo, suponha o seguinte bloco procedural :

```
always @ (posedge C) begin
    qr0 = 0;
    qr1 = qr0;
end
```

Ao final da simulação desse bloco, tem-se que tanto qr0 como qr1 terminarão zerados e o valor anteriormente armazenado em qr0 será perdido!

Já uma atribuição *nonblocking* faz o simulador primeiro computar todas atribuições *nonblocking* habilitadas no corrente instante de tempo de simulação (o chamado *time slot*) usando os **valores atuais** das *nets* antes de atualizá-las. Assim, ao final da simulação de um bloco como

```
always @ (posedge C) begin
    qr0 <= 0;
    qr1 <= qr0;
end
```

o bit qr1 terá o valor que estava armazenado em qr0 antes deste ser zerado.

Regra geral: use atribuições "=" (*blocking*) ao descrever saídas de circuitos combinacionais, e atribuições "<=" (*nonblocking*) para saídas de flip-flops e outros registradores.

NOTA: as designações *blocking* e *nonblocking* parecem estar trocadas, mas é isso mesmo. A justificativa para esses nomes ficará clara mais a frente, quando tratarmos da simulação de sinais com atraso.

5.4.3 Entradas assíncronas em circuitos síncronos

Vejamos como implementar a entrada de *clear* assíncrono do flip-flop D da Figura 5.2. O módulo FF_DE mostrado na Figura 5.10 é uma possível forma de descrevê-lo em Verilog.

```
module FF_DE(
    input C, D, En, Clr,
    output Q );

    reg qr;

    always @ (posedge C or posedge Clr) begin
        if (Clr) qr <= 0;
        else if (En) qr <= D;
        else qr <= qr;
    end // always

    assign Q = qr;

endmodule
```

Figura 5.10 Módulo FF_DE: flip-flop D com *enable* e *clear* assíncrono

Olhando a lista de sensibilidade do bloco *always*, vê-se o circuito é sensível a borda de subida de DOIS sinais: C e Clr. No entanto, testa-se o *nível* do sinal Clr logo no início do bloco (com "if (Clr)") para garantir a precedência da condição Clr = 1, e assim zerar a saída Q independentemente do sinal de *clock*.

5.5 Contadores CB2 e CB4_2CB2

O contador binário de dois bits CB2 da Figura 5.5 pode ser descrito em Verilog de forma não-procedural sem muita dificuldade, usando-se o flip-flop D do módulo FF_DE. Veja a Figura 5.11. Os sinais D[1:0], En0 e En1 foram adicionados apenas para melhorar a documentação do módulo. Muito provavelmente, o compilador se encarregará de eliminá-las quando for sintetizar o circuito.

```
module CB2(
    input C, En, Clr,
    output [1:0] Q,
    output RCO );

    wire [1:0] D;
    wire EnU0, EnU1;

    assign D = ~Q;
    assign EnU0 = En;
    assign EnU1 = En & Q[0];
    assign RCO = En & Q[1] & Q[0];

    FF_DE U0 ( C, D[0], EnU0, Clr, Q[0] );
    FF_DE U1 ( C, D[1], EnU1, Clr, Q[1] );

endmodule
```

Figura 5.11 Módulo CB2: contador síncrono de dois bits

Seguindo o esquema de cascateamento mostrado na Figura 5.7, a construção do contador de quatro bits é imediata, como mostra a Figura 5.12.

```
module CB4_2CB2(
    input C, En, Clr,
    output [3:0] Q,
    output RCO );

    (* keep = "true" *) wire EnU3;

    CB2 U2 ( C, En, Clr, Q[1:0], EnU3 );
    CB2 U3 ( C, EnU3, Clr, Q[3:2], RCO );

endmodule
```

Figura 5.12 Módulo CB4_2CB2: cascateamento de dois contadores de dois bits

Cabe esclarecer um detalhe: a declaração "(* keep = "true" *)" instrui o compilador para preservar o sinal EnU3 durante a síntese do circuito, mesmo que ele possa ser descartado na etapa de otimização. Assim poderemos observá-lo durante a simulação do circuito resultante.

Nota importante (que talvez o desanime um pouco): na prática, a implementação de contadores em Verilog é muito mais simples, e não requer o uso de flip-flops ou o cascateamento de contadores (a menos que você queira ter controle total sobre sinais internos).

Contadores com qualquer número de bits podem ser descritos em Verilog de forma direta, por meio de declarações procedurais do tipo

```
qr <= qr + 1;
```

(simples, não?). Veja, por exemplo, a descrição do contador BCD (que conta de 0 a 9) ao final desta apostila.

5.5.1 Simulação do contador CB4_2CB2

A Figura 5.13 mostra o módulo que usaremos no laboratório para testar o contador CB4_2CB2 (o chamado *testbench*).

Na primeira linha, a diretiva "`*timescale* 100 us / 1us" define a simulação usará 100 µs como unidade de tempo, com precisão de 1 µs para cálculo de atrasos (usando a letra 'u' para representar a letra grega 'µ').

No começo do módulo, temos a declaração das variáveis e *nets* a serem usadas, com atribuição do valor inicial para algumas delas. Em particular, o *array* k servirá para contar os ciclos de *clock* e consequentemente para acumular o tempo transcorrido de simulação.

A simulação começa processando o bloco *initial*. Logo de início, o comando "#10;" faz a simulação avançar 1 ms (ou seja, 10 unidades de tempo definidas pela diretiva `*timescale*). Esse atraso inicial é necessário porque a FPGA contém um circuito que gera um sinal de *reset* global ao ser ligado (conhecido como *power-on reset*) para zerar todos os registradores e o contador não funcionaria antes disso. Esse tempo é de aproximadamente 100 ns. Usaremos 1 ms (bem mais do que o suficiente) para coincidir com o período do *clock* da simulação.

A linha seguinte, "Clr = 1;" zera o contador, fazendo o sinal de *clear* ir a 1. Note que isso ocorre no instante de tempo t = 1 ms da simulação.

A linha seguinte precisa ser explicada com calma:

```
Clr <= #10 0;
```

é uma atribuição ***nonblocking* com atraso**. Ela faz o simulador zerar o sinal Clr após um atraso de 1 ms, mas sem pausar a simulação por esse tempo – daí o nome *nonblocking* para a atribuição "<=". O simulador segue adiante, executando os comandos seguintes em paralelo com a temporização do atraso. Desse modo, o *loop* "for" começa no instante t = 1 ms e o sinal Clr vai a zero em t = 2 ms.

Por fim, observe que o *loop* "for" ao final do bloco *initial* gera o sinal de *clock* C com período de 1 ms e incrementa o inteiro k a cada período. Assim, k conta as bordas de subida do *clock*.

O bloco *always* monitora o inteiro k para mudar o sinal En (*Enable*) nos instantes desejados, também por meio de atribuições *nonblocking* com atraso. Uma observação importante (e pouco óbvia): esse bloco *always* roda EM PARALELO com o bloco *initial* e com a instanciação do contador CB4_2CB2!

```
`timescale 100us / 1us  // Unidade de tempo/precisão da simulação (us=micro-seg.)
module CB4_2CB2_sim( ); // Lista de entradas e saídas: vazia neste caso

    reg C = 0, Clr = 0, En = 0;
    wire [3:0] Q;
    wire RCO;
    integer k = 0;  // contador de bordas do clock C

    CB4_2CB2 UUT (C, En, Clr, Q, RCO); // instanciação do módulo a ser simulado

    initial begin
        $timeformat(-3, 1, "ms", 4);  // formato para imprimir $time
        #10;  // avança o tempo em 10x100 us = 1 ms
        Clr = 1;  // clear inicial
        Clr <= #10 0;  // Clr vai a 0 após atraso de 1 ms.
        for ( k = 1; k <= 22; k = k + 1 ) begin
            $write("Time %t: Clr=%b, En=%b => ", $time, Clr, En);
            $strobe("Q=%b RCO=%b EnU3=%b", Q, RCO, UUT.EnU3);
            C = 1; #5;  // clock C=1 por meio período (5x100 us = 0,5 ms)
            C = 0; #5;  // clock C=0 por meio período (0,5 ms)
        end; // for k
        $finish;
    end  // initial

    always @ (*) begin
        En <= 0;  // valor default de En
        if (k >= 2 && k != 09 && k != 18) En <= 1;  // Altera En em função de k
    end  // always

endmodule
```

Figura 5.13 Módulo CB4_2CB2_sim: *testbench* para simulação do contador CB4_2CB2.

A variável k conta as bordas de subida do sinal de *clock* C: começa com zero e é incrementada pela primeira vez em t = 1 s (confira). O *enable* En começa em zero e vai a um em k = 2, e volta a zero pela primeira vez quando k = 9. Portanto, temos novamente En =0 após a borda de *clock* em ... t = 9 ms! O segundo momento em que En = 0 fica por sua conta (mas não esqueça que a contagem não avança quando En = 0).

LEMBRE-SE: quando sinais de entrada mudam sincronamente com o *clock*, você deve considerar o valor deles imediatamente **antes** da borda. É o que acontece no instante em que k = 2, onde temos Clr =1 e En =0 antes do *clock*, e por isso a contagem permanece em zero. Nota: os sinais gerados nessa simulação NÃO reproduzem inteiramente a carta de tempos da Figura 5.6 (apenas os instantes iniciais).

Neste *testbench* usamos algumas *tasks* (comandos passados para o compilador) já vistas: ***$timeformat***, ***$time***, ***$write*** e ***$finish***. Reveja a definição delas na apostila da experiência anterior.

A *task* ***$strobe*** imprime mensagens e valores semelhantemente ao que ***$write*** faz, mas com algumas diferenças. Enquanto ***$write*** imprime os valores de *nets* **antes** de atribuições *non-blocking* ("<=") serem atualizadas, ***$strobe*** imprime os valores **após** a atualização e acrescenta também um fim de linha ("\n").

No *testbench* da Figura 5.13, usamos *$write* para imprimir os valores das entradas Clr e En válidos *antes* da borda de *clock*, e *$strobe* para imprimir os valores das saídas Q[3:0] resultantes *após* a borda de *clock*. O comando *$strobe* imprime também o valor da *net* **UUT.EnU3** que é interna ao módulo CB4_2CB2 (instanciação **UUT**).

5.6 Chaves, botões e o problema de *bouncing*

Chaves e botões são dispositivos muito utilizados como entradas em circuitos eletrônicos. No entanto, eles apresentam um problema de natureza eletromecânica que pode interferir no funcionamento do circuito.

Observe a Figura 5.14. Seria de se esperar que a saída Q invertesse cada vez que o botão BTN fosse pressionado. No entanto, devido a vibrações mecânicas e contatos elétricos imperfeitos, o sinal C apresenta rápidas "oscilações" que podem durar alguns milissegundos, conhecidas como *bounce*. Isso pode resultar em bordas espúrias tanto no momento em que o botão é apertado como quando é liberado.

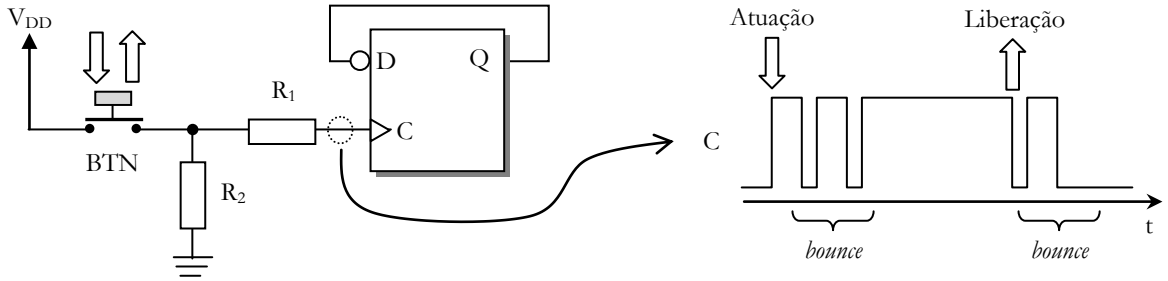


Figura 5.14 Circuito sujeito ao problema de bounce

O *bounce* acontece com todo tipo de dispositivo com contato eletromecânico, em maior ou menor grau. Dependendo da aplicação, podem ser utilizados circuitos eletrônicos para minimizar o problema – os chamados circuitos de *debounce*.

O mais simples é o filtro RC: acrescenta-se um capacitor entre a entrada C e 0 V, como mostra a Figura 5.15. Tendo-se R_1 bem menor que R_2, o capacitor C_1 se carrega rapidamente (e exponencialmente) através de R_1 quando o botão é apertado com constante de tempo $\tau = R_1C_1$, mas se descarrega lentamente através de R_1 e R_2 quando o botão é liberado com $\tau = (R_1 + R_2)C_1$.

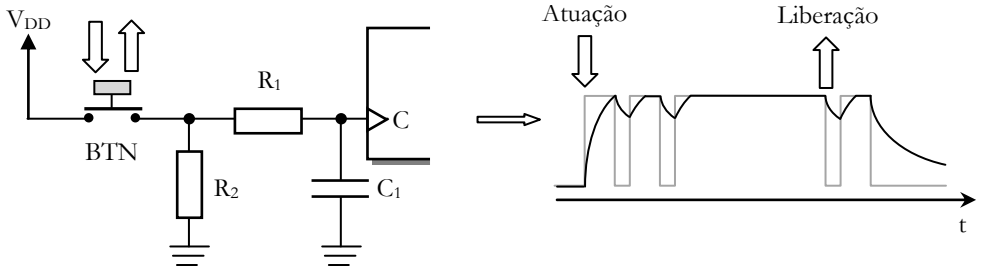


Figura 5.15 *Debounce* com filtro RC

O *debounce* também pode ser feito por um circuito conhecido como *timer* ou mono-estável, que gera um pulso de duração conhecida após ser disparado. Veremos como construir *timers* digitais usando contadores síncronos numa próxima experiência.

ATENÇÃO: entradas sensíveis a borda não devem ser alimentadas com sinais provenientes de saídas de circuitos combinacionais. Em circuitos desse tipo, atrasos relativos entre sinais também podem gerar pulsos espúrios – efeito conhecido como *hazard* estático, visto na experiência 2. Como regra geral, sinais de *clock* devem ser gerados por meio de circuitos sequenciais, como *latches* e flip-flops. É o caso do circuito que usaremos no laboratório.

A Figura 5.16 mostra o circuito que será implementado na placa Basys 3, e que se encontra no arquivo TestCB4_top.v . O módulo xxCB4 (U1) é um contador binário de 4 bits que você irá implementar

```
module TestCB4_top(
    input btnL, btnR, btnD,
    input [15:15] sw,
    output [15:0] ld );

    wire S, R, C, CN;
    wire En, Clr, RCO;
    wire [3:0] Q;

    assign S = btnL;
    assign R = btnR;
    assign C = ~(R | CN);  // latch SR positivo: saída Q
    assign CN = ~(S | C);  // latch SR positivo: saída QN
    assign ld[14] = C;
    assign ld[13] = CN;

    assign En = sw[15];
    assign Clr = btnD;
    assign ld[15] = En;

    xxCB4 U1 ( C, En, Clr, Q, RCO );
    assign ld[3:0] = Q;
    assign ld[4] = RCO;

endmodule
```

Figura 5.16 Módulo TestCB4_top: teste do contador com *clock* gerado por latch.

Repare que o *clock* C não está ligado diretamente a um botão, e sim à saída de um *latch* SR positivo) para garantir que não haja *bounce* na entrada de *clock*. O botão btnR zera o sinal de *clock*, enquanto que o botão btnL leva o *clock* a 1.

A entrada de *enable* En pode ser controlada pela chave sw[15] e observada no led ld[15]. O botão btnD permite zerar o contador. Os leds ld[14] e ld[13] permitem observar o estado do sinal de clock. As saídas Q[3:0] podem ser observados nos leds ld[3:0] e a saída RCO no led ld[4].

5.7 Declarações de *constraints* para circuitos sequenciais no Vivado

Na experiência anterior, vimos que o programa Vivado usa um arquivo texto para descrever como conectar sinais pinos físicos da FPGA, o arquivo de restrições (***constraints***), que tem extensão ".xdc".

Na verdade o arquivo de *constraints* tem finalidade mais ampla, que é definir regras de restrições do projeto (ou *DRC*, *Design Rule Constraints*) e a associação de sinais a pinos é uma dessas regras. E por *default*, o Vivado tem algumas regras de restrição habilitadas que impediriam a síntese de alguns dos módulos que usaremos na experiência.

5.7.1 Permitindo realimentação de saídas nas entradas

O módulo de teste do contador xxCB4 mostrado na Figura 5.16 é um deles. O problema é que o programa detecta a realimentação das saídas para as entradas do *latch* SR, e circuitos desse tipo não são mais comuns nos dias de hoje, uma vez que as linguagens HDL já incluem *latches* e flip-flops pré-definidos.

É preciso então fazer o compilador relaxar a DRC interna que impede a criação de *loops* temporais de um nível (LUTP-1). Para isso, você deverá incluir a seguinte linha no arquivo de *constraints*:

```
set_property SEVERITY {Warning} [get_drc_checks LUTLP-1]
```

5.7.2 Definição do sinal externo de *clock*

Na placa Basys 3 existe um componente dedicado para gerar um sinal de *clock* de 100 MHz de frequência que está conectado ao pino W5 da FPGA. Para usá-lo, é necessário incluir no arquivo de *constraints* o par de linhas de definição do pino, conforme explicado na apostila da experiência anterior. Além disso, deve-se acrescentar uma terceira linha para instruir o compilador para considerar esse sinal como *clock*. Chamando o sinal de *clock* de **C100M**, teríamos:

```
set_property PACKAGE_PIN W5 [get_ports C100M]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports C100M]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports C100M]
```

5.7.3 Uso de pino não apropriado para entrada de *clock* externo

Como sinais de *clock* são importantes em qualquer circuito sequencial, a FPGA que usamos no laboratório tem alguns pinos indicados para serem conectados a geradores de *clock* externo. O pino W5 é um deles.

No entanto, numa das atividades da experiência, vamos usar um dos botões da placa Basys 3 para gerar sinais de *clock*, mas nenhum deles está ligado a um pino da FPGA apropriado para entrada de *clock*. Teremos que desabilitar a DRC que impede que as vias internas de *clock* da FPGA sejam conectadas a um pino qualquer.

Para usar o botão btnL como gerador de *clock*, teremos que acrescentar a seguinte linha de *constraint*:

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets btnL_IBUF]
```

5.8 Aplicação: acionamento do mostrador de sete segmentos

Na placa Basys 3, temos quatro mostradores de sete segmentos. Como mostra a Figura 5.17, os segmentos são compostos por dispositivos semicondutores emissores de luz – os *leds*, que se acendem quando atravessados por uma corrente do terminal superior (chamado de *anodo*) para o inferior (o *catodo*). Cada mostrador tem um termina de ativação An_i (ativo em zero) chamado de *anodo comum*, que alimenta ao mesmo tempo todos os leds de um mostrador. Os oito catodos (CA a CG e DP) permitem acender os sete segmentos e o ponto decimal separadamente quando aterrados. Por exemplo, o segmento *a* (traço superior) se acende sempre que fazemos a entrada CA igual a 0.

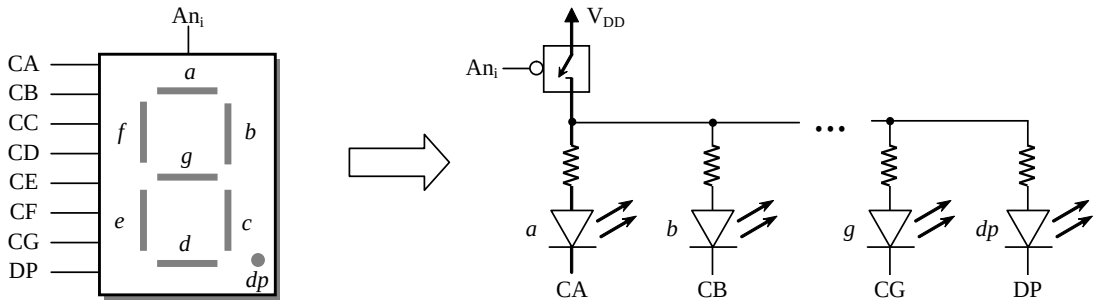


Figura 5.17 Mostrador de sete segmentos (esquerda) e esquema de ligação dos seus leds

A Figura 5.18 mostra como esse mostrador pode ser utilizado para apresentar (de forma um tanto precária) dígitos hexadecimais, de 0 a F. Dessa forma, para mostrar o dígito '0', por exemplo, devemos fazer os sinais de CA a CF em zero para acender os leds de *a* a *f*, e fazer CG e DP iguais a um para apagar os leds *g* e *dp*; para mostrar o dígito '2' fazemos apenas CB e CC iguais a zero, e assim por diante.

No entanto, os catodos são comuns aos quatro mostradores. Ou seja, ativando os quatro anodos (An[3:0] = 0000) ao mesmo tempo, os mostradores mostrarão o mesmo dígito imposto nos catodos CA a CG. Como faríamos, então, para mostrar um dígito diferente em cada um dos mostradores?

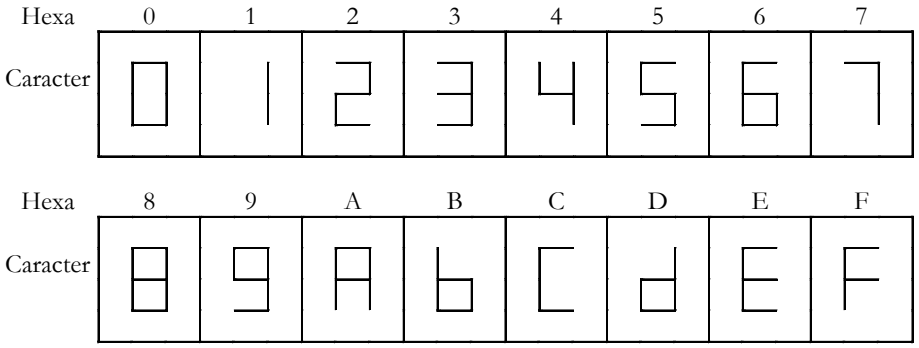


Figura 5.18 Padrão de apresentação de dígitos hexadecimais em 7 segmentos (sem o ponto decimal)

A solução é acionar um anodo de cada vez ciclicamente, e utilizar multiplexadores para colocar nos catodos o código de cada dígito de forma sincronizada com a ativação dos anodos. Ou seja, cada mostrador piscaria um dígito diferente, um de cada vez.

Se a taxa de varredura dos mostradores for alta (acima de trinta vezes por segundo, aproximadamente), ficaremos com a impressão que os mostradores estão sempre ligados, já que não conseguiremos mais perceber o efeito

pisca-pisca.

A Figura 5.19 mostra um circuito para apresentar quatro dígitos distintos (DA a DD) nos mostradores. Os dígitos entram em um multiplexador quádruplo de 4 bits (bloco H2), que a cada instante passa apenas um deles. O bloco H3 faz a conversão do número em binário para o padrão correspondente para apresentação no mostrador de sete seguimentos (o ponto decimal foi omitido no esquema). O bloco H1 representa um contador cíclico de 2 bits, que gera os bits de seleção do multiplexador e também do decodificador (bloco H3), garantindo a sincronização entre o dígito selecionado no mux e o mostrador a ser ligado.

O sinal de CLOCK deve ter uma frequência superior a 4 x 30 = 120 Hz. Essa é a taxa necessária para que cada mostrador seja ligado 30 vezes por segundo. O sinal ENABLE permite ligar e desligar todos os mostradores conjuntamente, desabilitando o decodificador e o contador.

A Figura 5.20 mostra o símbolo do módulo xxDsDrv. Para implementá-lo, você poderá usar os componentes mostrados na Figura 5.21. As listagem em Verilog destes módulos estão em anexo.

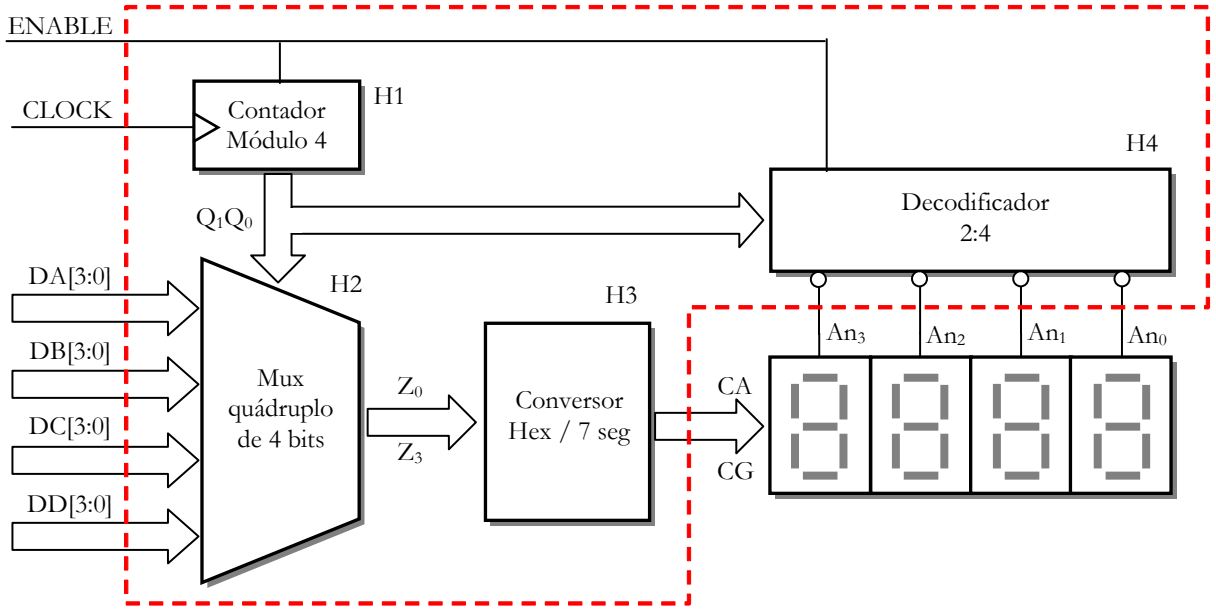


Figura 5.19 Circuito para acionamento dos mostradores de sete segmentos.

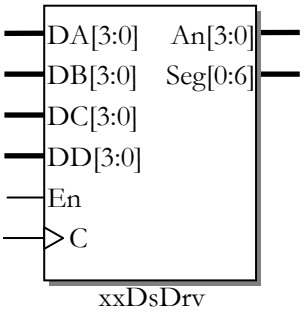


Figura 5.20 Símbolo do acionador de mostrador de sete segmentos

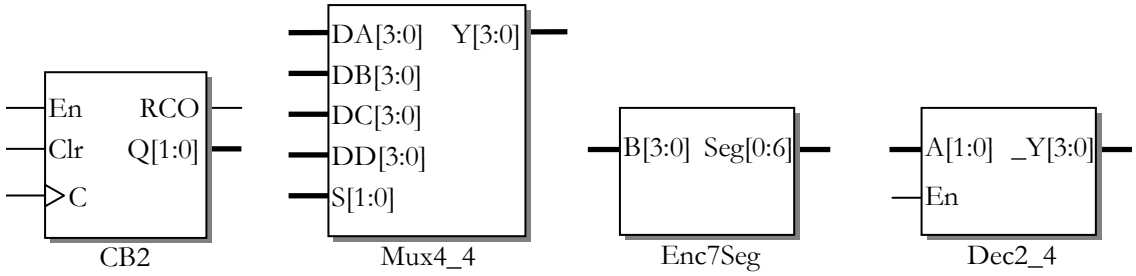


Figura 5.21 Componentes para o projeto do acionador de mostrador de sete segmentos

- **CB2**: contador de dois bits, com entradas *enable* (En) e *clear* assíncrono (Clr), já descrito anteriormente.
- **Mux4_4**: multiplexador quádruplo de quatro bits.
- **Enc7Seg**: conversor de código de binário para sete segmentos, com Seg[0] = CA ... Seg[6] = CG.
- **Dec2_4**: decodificador 2:4 com entrada de *enable* (En) e saídas ativas em zero.

5.9 Cronômetro

Vamos testar o acionador utilizando o circuito de um cronômetro, mostrado na Figura 5.22. O sinal de *clock* é gerado pelo módulo Clock400, cuja saída c400 fornece um sinal de 400 Hz. Esse sinal será utilizado para fazer a varredura do mostrador, e portanto cada dígito piscará a uma taxa de 100 vezes por segundo.

O contador CB2 divide a frequência do *clock* c400 por 4, resultando em uma base de tempo de 0,01 s. Os quatro contadores CD4 de quatro bits contam ciclicamente de 0 a 9 (são os chamados contadores de década ou contadores BCD). Portanto, cada um deles divide a frequência por dez. Como o primeiro contador é acionado a cada 0,01 s, temos no barramento Da a contagem em centésimos de segundo. Analogamente, em Db temos a contagem em décimos de segundo, e assim por diante.

Note como o cascateamento dos contadores é feito. O sinal de *clock* c400 de 400 Hz é conectado na entrada de *clock* de todos os contadores, mas apenas o contador CB2 (de módulo 4) está habilitado continuamente. O primeiro contador CD4 é habilitado pela saída RCO, que vai a 1 somente nos períodos em que a contagem passa pelo máximo (3, neste caso) e portanto o primeiro contador CD4 é incrementado apenas a cada 4 ciclos de *clock*. Pelo mesmo princípio, o segundo contador CD4 é incrementado a cada 40 ciclos, o terceiro a cada 400 ciclos, e o último a cada 4000 ciclos.

O botão btnD está ligado às entradas Clr (*Clear*) dos contadores. Dessa forma, o cronômetro é zerado sempre que pressionarmos esse botão.

```
module Cron_top(
    input clk, btnD,
    output [3:0] an,
    output [0:6] seg
    );

    wire c400, rst;
    wire [1:0] Q_U1;
    wire [3:0] Da, Db, Dc, Dd;
    wire EnU2, EnU3, EnU4, EnU5, RCO_U5;

    assign rst = btnD;

    Clock400 U0 ( clk, 0, c400 );

    CB2 U1 ( c400, 1, rst, Q_U1, EnU2 );
    CD4 U2 ( c400, EnU2, rst, Da, EnU3 );
    CD4 U3 ( c400, EnU3, rst, Db, EnU4 );
    CD4 U4 ( c400, EnU4, rst, Dc, EnU5 );
    CD4 U5 ( c400, EnU5, rst, Dd, RCO_U5 );

    xxDsDrv U6 ( c400, Da, Db, Dc, Dd, 1, an, seg );

endmodule
```

Figura 5.22 Módulo Cron_top: Circuito do cronômetro

5.10 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.

5.11 Outros módulos em Verilog usados nesta experiência

```
module CD4(
    input C, En, Clr,
    output [3:0] Q,
    output RCO
    );

    reg [3:0] qr;

    always @ (posedge C or posedge Clr) begin
        if (Clr) qr <= 0;
        else if (En)
            if (qr < 9) qr <= qr + 1;
            else qr <= 0;
        else
            qr <= qr;
    end // always

    assign Q = qr;
    assign RCO = En & ( Q == 4'b1001);

endmodule


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

    reg cr;
    reg [16:0] k;

    always @ (posedge C100M, posedge Clr) begin
        if (Clr) begin
            k <= 0;
            cr <= 0;
        end // if
        else if (k < 125000)
            k <= k + 1;
        else begin
            k <= 0;
            cr <= ~cr;
        end // else
    end // always

    assign C400 = cr;

endmodule
```

```
module Dec2_4(
    input [1:0] A,
    input En,
    output [3:0] _Y
    );

    reg [3:0] _yr;

    always @ (*) begin
        if (~En)
            _yr = 4'b1111;
        else case ( A )
            2'b00: _yr = 4'b1110;
            2'b01: _yr = 4'b1101;
            2'b10: _yr = 4'b1011;
            2'b11: _yr = 4'b0111;
        endcase
    end // always

    assign _Y = _yr;

endmodule
```

```
module Enc7Seg(
    input [3:0] B,
    output [0:6] Seg
    );

    reg [0:6] sr;

    always @ (*) begin
        case (B)
            4'h0 : sr = 7'b0000001;    // Hexadecimal 0
            4'h1 : sr = 7'b1001111;    // Hexadecimal 1
            4'h2 : sr = 7'b0010010;    // Hexadecimal 2
            4'h3 : sr = 7'b0000110;    // Hexadecimal 3
            4'h4 : sr = 7'b1001100;    // Hexadecimal 4
            4'h5 : sr = 7'b0100100;    // Hexadecimal 5
            4'h6 : sr = 7'b0100000;    // Hexadecimal 6
            4'h7 : sr = 7'b0001111;    // Hexadecimal 7
            4'h8 : sr = 7'b0000000;    // Hexadecimal 8
            4'h9 : sr = 7'b0000100;    // Hexadecimal 9
            4'hA : sr = 7'b0001000;    // Hexadecimal A
            4'hB : sr = 7'b1100000;    // Hexadecimal B
            4'hC : sr = 7'b0110001;    // Hexadecimal C
            4'hD : sr = 7'b1000010;    // Hexadecimal D
            4'hE : sr = 7'b0110000;    // Hexadecimal E
            4'hF : sr = 7'b0111000;    // Hexadecimal F
        endcase
    end

    assign Seg = sr;

endmodule
```

```
module Mux4_4(
    input [3:0] Da, Db, Dc, Dd,
    input [1:0] S,
    output [3:0] Y
    );

    reg [3:0] yr;

    always @ (*) begin
        case ( S )
            2'b00 :  yr = Da;
            2'b01 :  yr = Db;
            2'b10 :  yr = Dc;
            2'b11 :  yr = Dd;
        endcase
    end // always

    assign Y = yr;

endmodule
```