

CIRCUITOS SEQUENCIAIS

Máquina de estados finita (FSM)

Planejamento para FSMs

AULA 1 (teórica)

- Conceito
- Exemplo da sinaleira
- Detector de padrão

AULA 2 (prática)

- Conceito Bloco de Dados e Bloco de Controle
- Multiplicador serial
- Divisão Serial
- Árbitro de Barramento

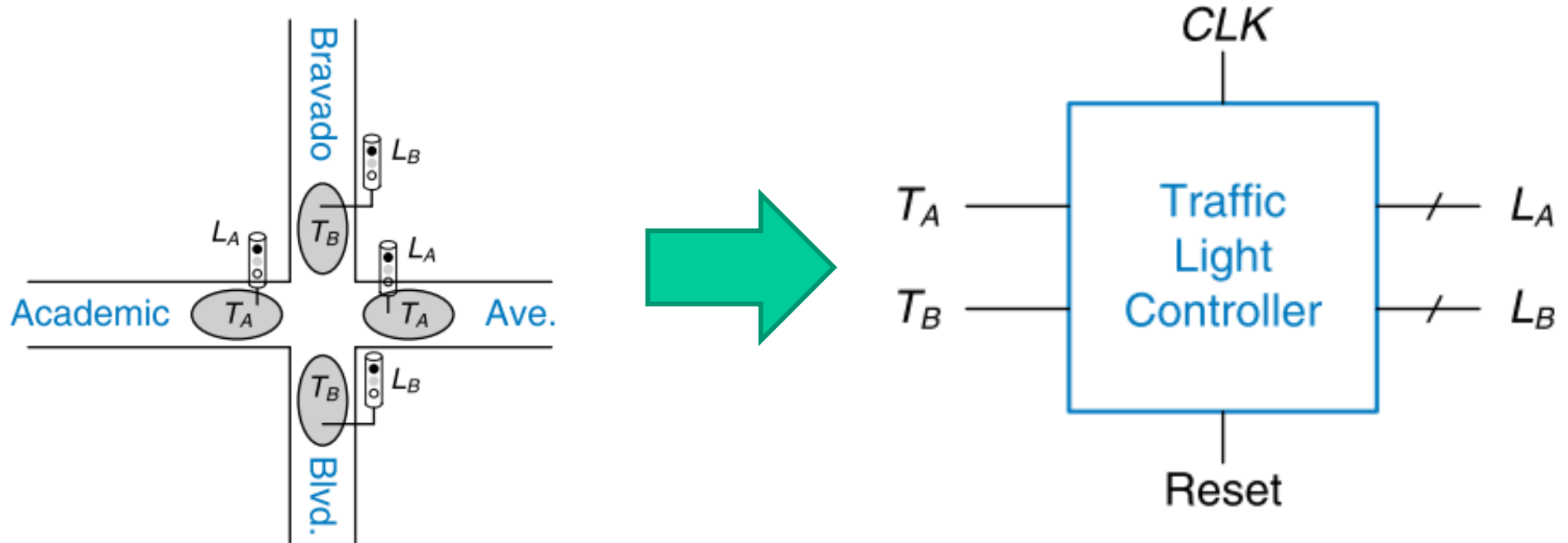
AULA 3 (prática)

- Transmissor serial
- Exercícios FSM

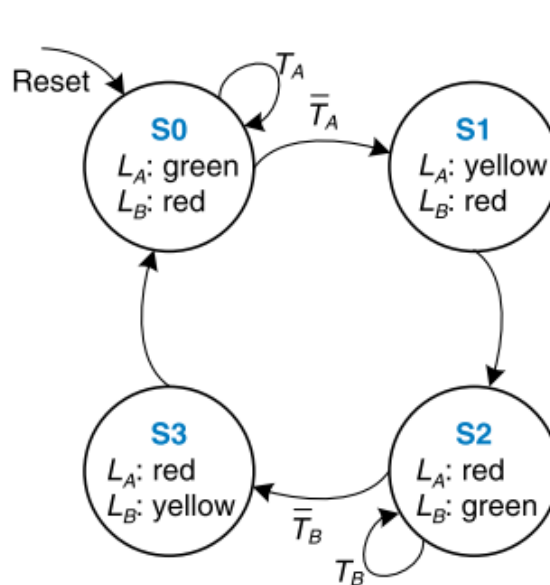
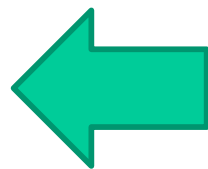
Máquina de estados finita (FSM)

- Uma máquina de estados finita (FSM - do inglês *Finite State Machine*) ou autômato finito é um modelo matemático usado para **representar** programas de computadores ou circuitos lógicos.
- O conceito é concebido como uma máquina abstrata que deve estar em um de um **número finito de estados**.
- A máquina está em apenas **um estado por vez**, este estado é chamado de **estado atual**. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada em um estado, no início do sistema, até o momento presente.
- Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento.

Problema → FSM → HDL

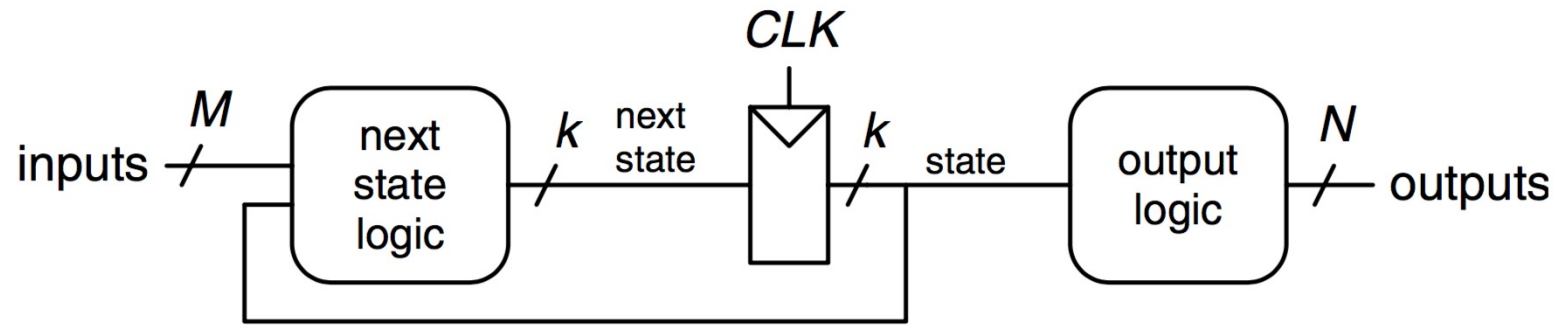


HDL

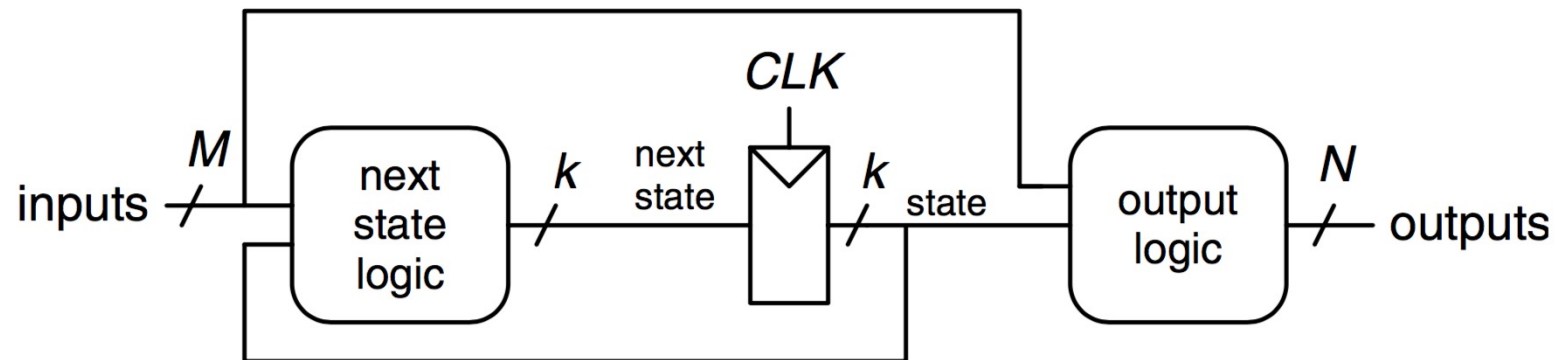


Solução completa em “Digital Design and Computer Architecture”. Sec 3.4.15

Tipos de FSM



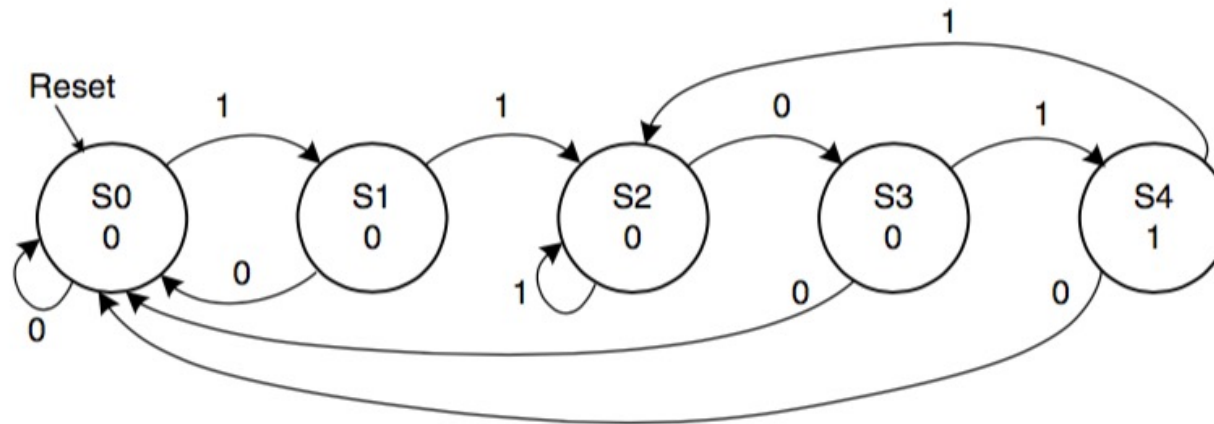
(a) Moore: valor de saída depende somente do estado atual



(b) Mealy: valor de saída depende do estado atual e das entradas

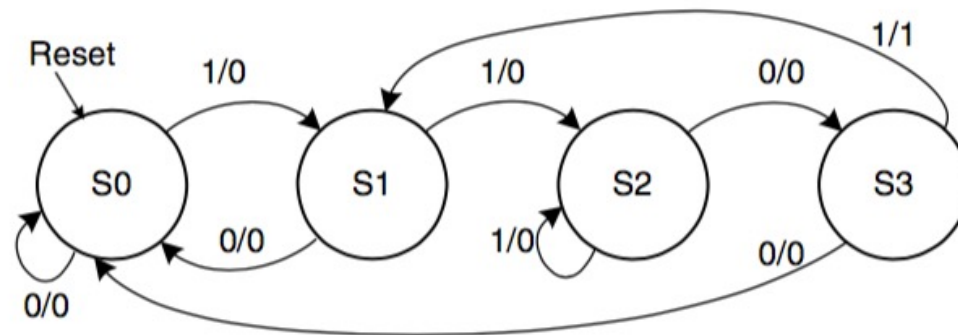
Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

Tipos de FSM



Moore: saída apresentada no estado, e nos arcos de transição apenas as entradas

(a)



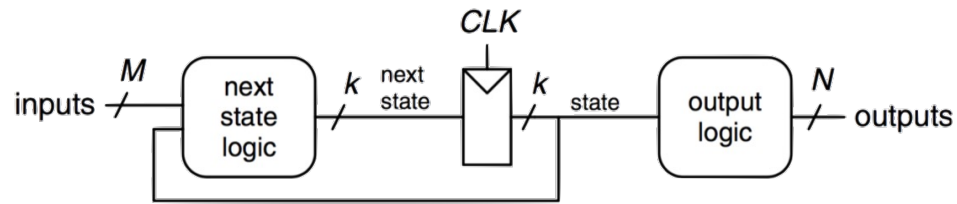
Mealy: entrada e saída apresentada nos arcos de transição

(b)

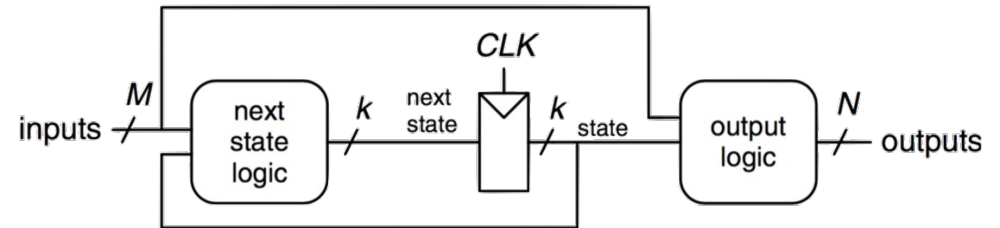
Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

Tipos de FSM

Moore:

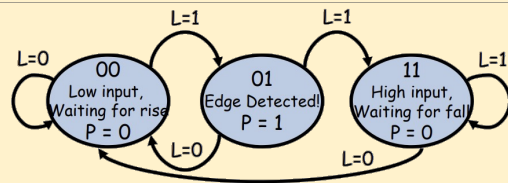


Mealy:

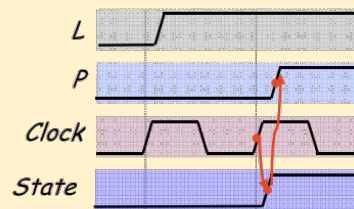


Qual a diferença?

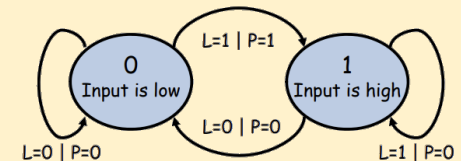
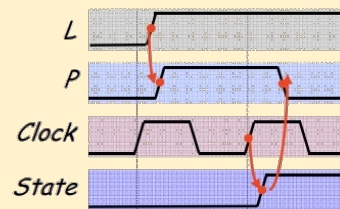
- **Moore:** saídas = $f(\text{estado})$ apenas
- **Mealy:** saídas = $f(\text{estado} + \text{entrada})$
- As saídas de Mealy geralmente ocorrem um ciclo antes do que um Moore:



Moore: delayed assertion of P



Mealy: immediate assertion of P



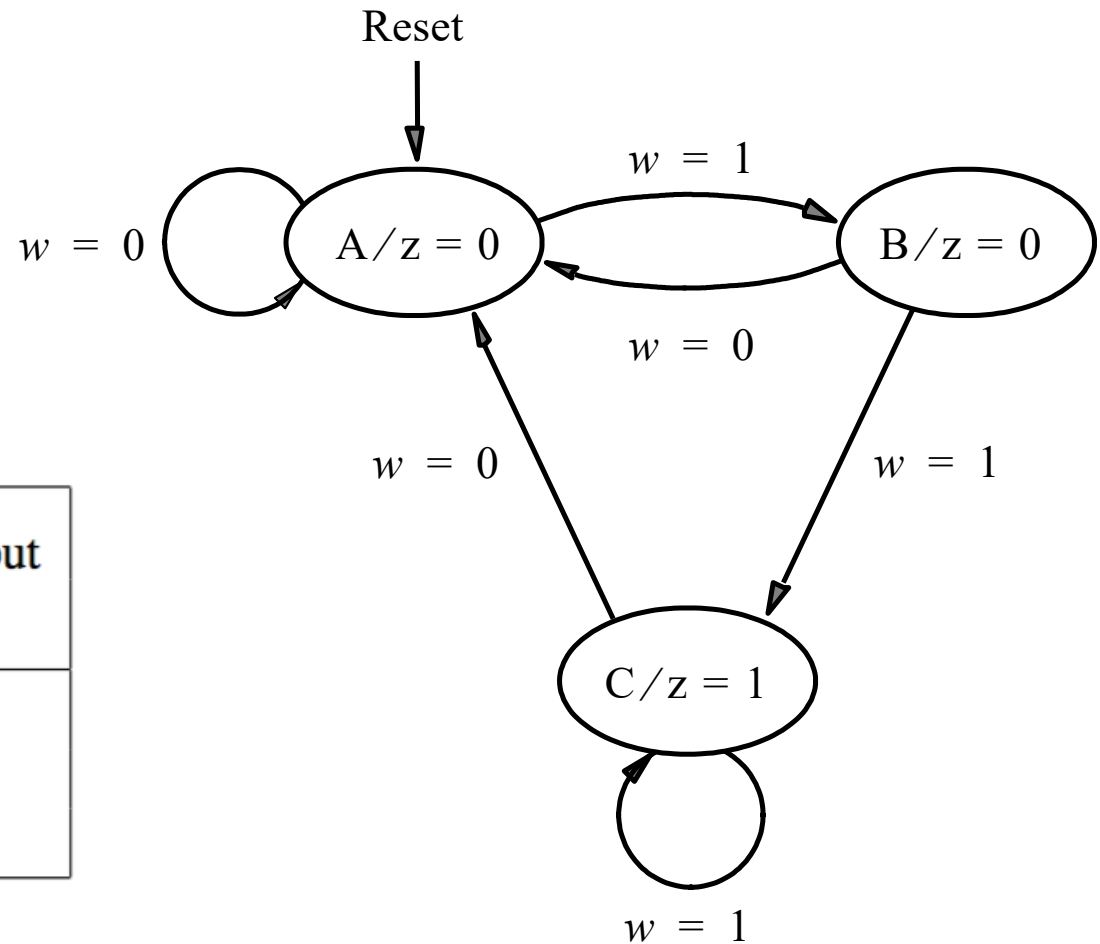
Em comparação com uma FSM Moore, uma FSM Mealy pode...

- Ser mais difícil de conceituar e projetar
- Ter menos estados

Que tipo de máquina de estados é esta?

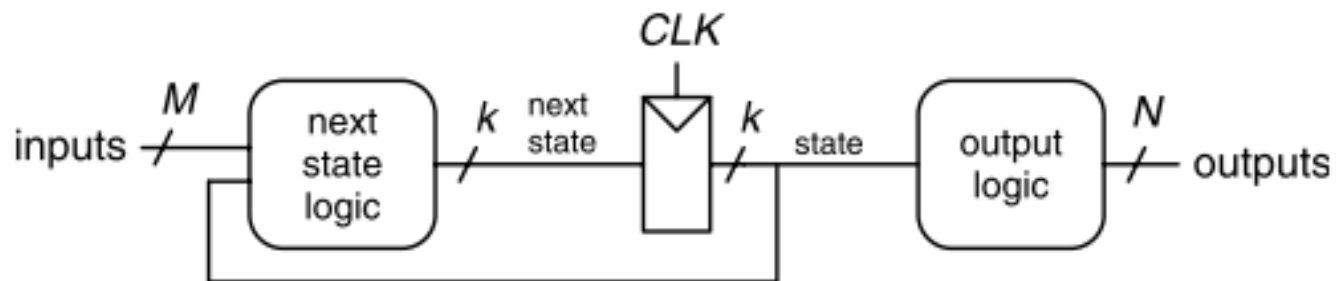
O que faz esta FSM ?

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1



FSM Moore

```
module moore (  
    input  logic clock, reset,  
    input  logic w,  
    output logic z  
);
```



```

module moore (
    input logic clock, reset,
    input logic w,
    output logic z
);

```

Ferramenta de síntese automaticamente determina o número de flip-flops necessários.

Cria um novo tipo de sinal.

```

// Definição dos estados
typedef enum logic [1:0] { A, B, C } state;

```

EA estado atual e PE próximo estado

```

state EA, PE;

```

Registrador do EA

```

always_ff @(posedge clock or posedge reset) begin
    if (reset)
        EA <= A;
    else
        EA <= PE;
end

```

O case descreve o comportamento da FSM
Cada cláusula representa um estado da FSM

```

// Lógica de transição de estados
always_comb begin
    unique case (EA)
        A: PE = (w == 1'b0) ? A : B;
        B: PE = (w == 1'b0) ? A : C;
        C: PE = (w == 1'b0) ? A : C;
        default: PE = A;
    endcase
end

```

Unique: gera **erro** se todos os casos **não** são cobertos

```

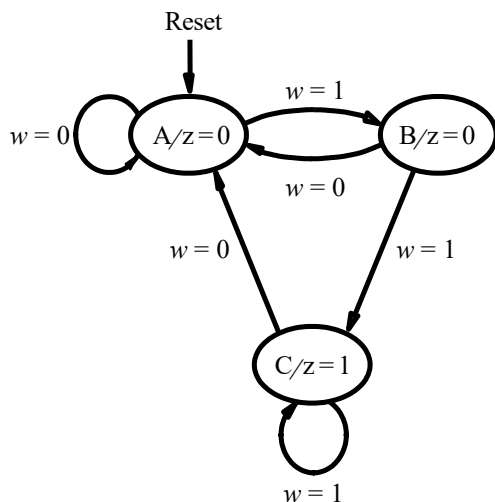
// Saída da máquina (Moore: depende apenas do estado)
assign z = (EA == C);

```

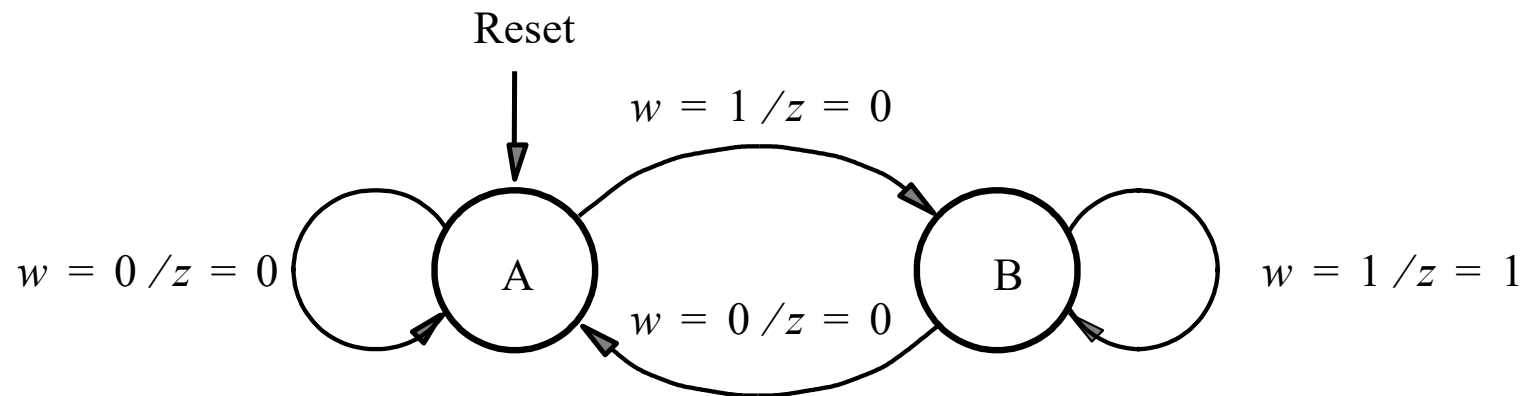
```

endmodule

```



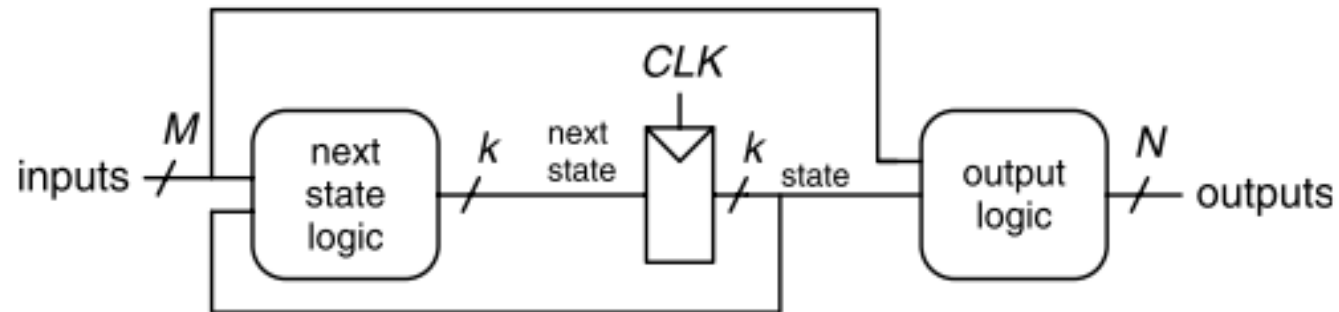
Que tipo de máquina de estados é esta?
O que faz esta FSM ?



Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

FSM Mealy

```
module mealy (  
    input  logic clock, reset,  
    input  logic w,  
    output logic z  
);
```



```

module mealy (
    input logic clock, reset,
    input logic w,
    output logic z
);

```

Create a user-defined
signal type.

```

// Definição dos estados
typedef enum logic [1:0] { A, B, C } state;

```

```

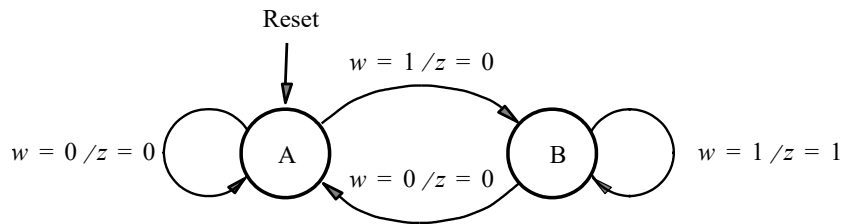
state EA, PE;

```

```

always_ff @(posedge clock or posedge reset) begin
    if (reset)
        EA <= A;
    else
        EA <= PE;
end

```



```

always_comb begin
    // Default
    PE = A;
    z = 1'b0;

```

```

    unique case (EA)
        A: begin

```

```

            PE = (w == 1'b0) ? A : B;
            z = 1'b0;
        end

```

```

        B: begin

```

```

            PE = (w == 1'b0) ? A : B;
            z = w;
        end

```

```

    end

```

```

endcase

```

```

end

```

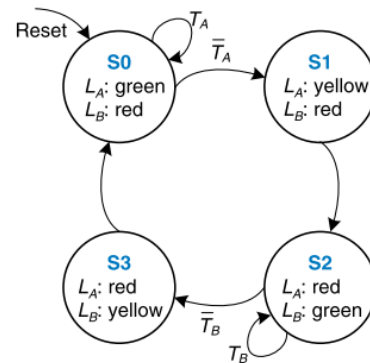
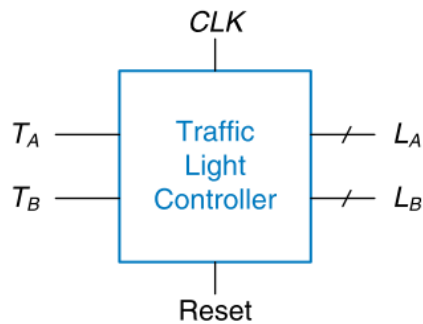
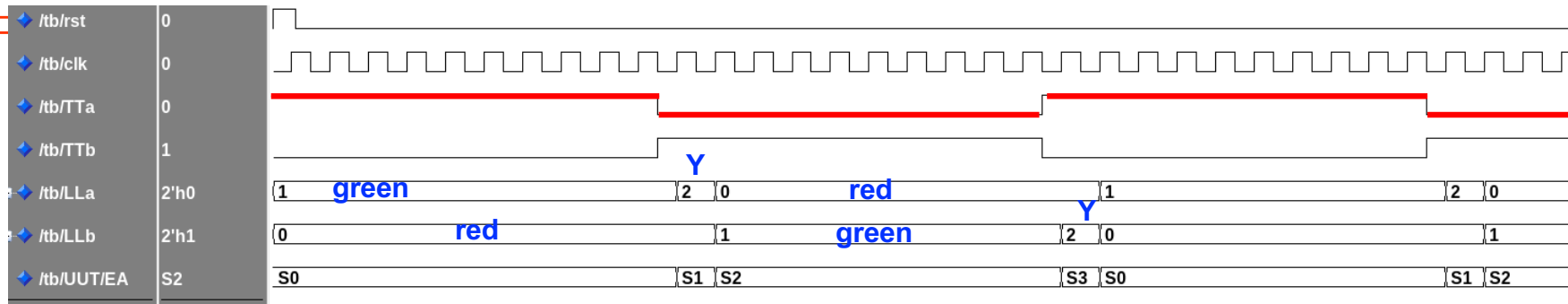
```

endmodule

```

Saídas definidas em função do
EA e das entradas

Voltando ao exemplo da sinaleira



```

module fsm1 (
    input logic clk, reset, Ta, Tb,
    output logic [1:0] La,
    output logic [1:0] Lb
);
    typedef enum logic [1:0] { S0, S1, S2, S3 } state;
    state EA, PE;

    // Estado atual
    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            EA <= S0;
        else
            EA <= PE;
    end

```

```

// Transições de estado
always_comb begin
    unique case (EA)
        S0: PE = (Ta == 1'b0) ? S1 : S0;
        S1: PE = S2;
        S2: PE = (Tb == 1'b0) ? S3 : S2;
        S3: PE = S0;
        default: PE = S0;
    endcase
end

```

```

// Saída La depende do estado atual
assign La = (EA == S0) ? 2'b01 :
            (EA == S1) ? 2'b10 :
            2'b00;

```

```

// Saída Lb depende do estado atual
assign Lb = (EA == S2) ? 2'b01 :
            (EA == S3) ? 2'b10 :
            2'b00;

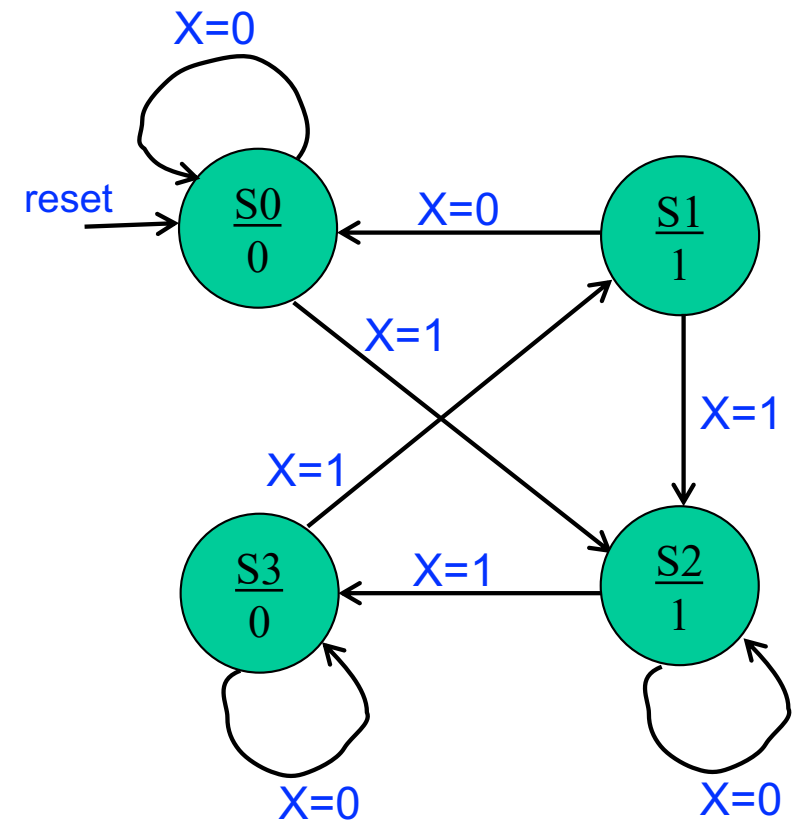
```

endmodule

FSM 1 - EXERCÍCIO - MÁQUINA DE ESTADOS

- Desenhe a FSM equivalente e diga de que tipo é esta FSM

```
module FSM (  
  input logic reset, clock, X,  
  output logic Z);  
  
  typedef enum logic [1:0] {S0, S1, S2, S3} state;  
  state EA, PE;  
  
  always_ff @(posedge clock or posedge reset) begin  
    if (reset)  
      EA <= S0;  
    else  
      EA <= PE;  
    end  
  
  always_comb begin  
    unique case (EA)  
      S0: PE = (X == 1'b0) ? S0 : S2;  
      S1: PE = (X == 1'b0) ? S0 : S2;  
      S2: PE = (X == 1'b0) ? S2 : S3;  
      S3: PE = (X == 1'b0) ? S3 : S1;  
      default: PE = S0;  
    endcase  
  end  
  
  assign Z = (EA == S1 || EA == S2); //saída  
  
endmodule
```



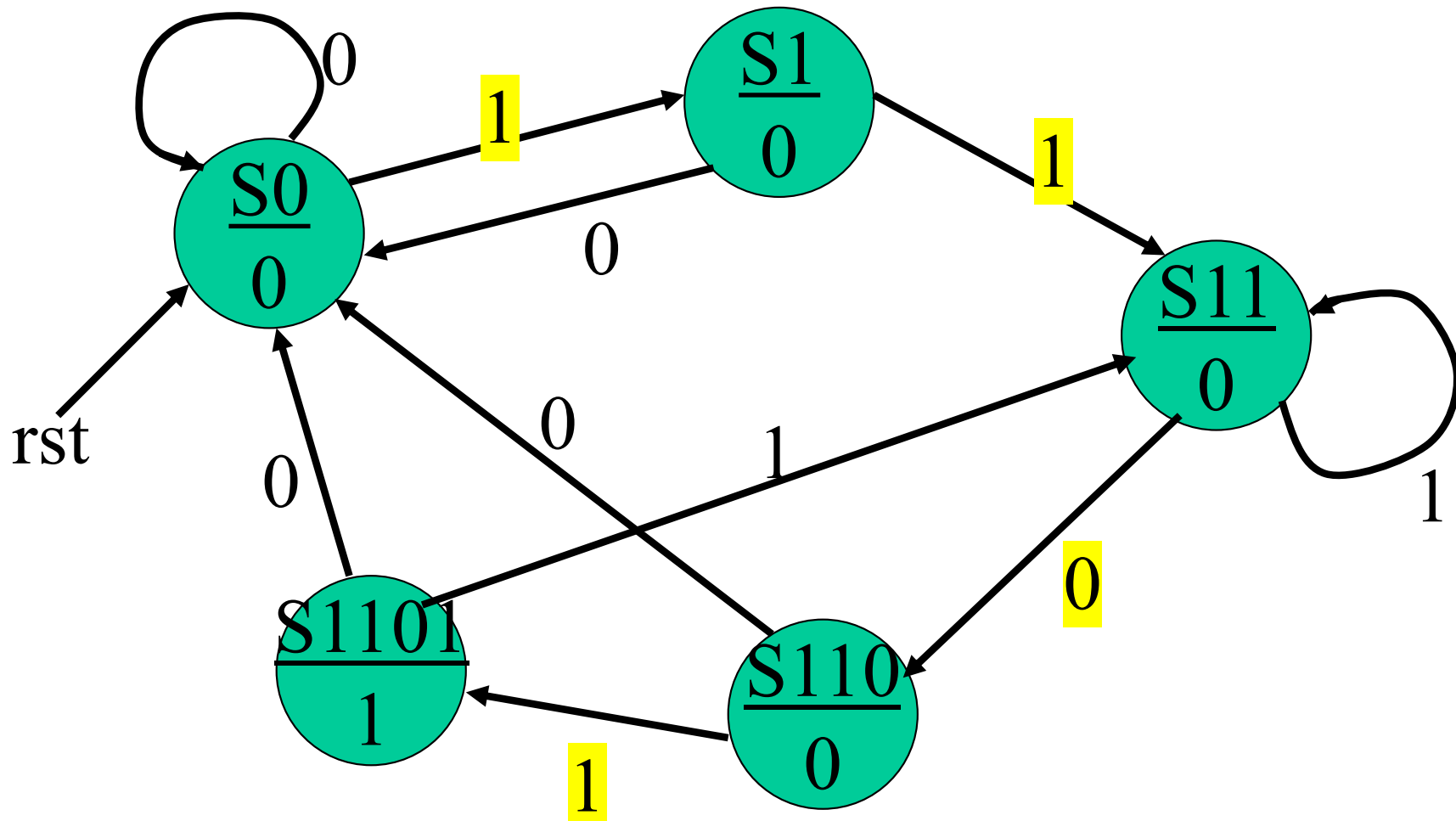
FSM 2 - Detector da sequência 1101



din	1	0	1	1	0	1	1	0	1	0	0	1	1	0	1	0	
dout	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0

Detect input sequence 1101

din 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0
dout 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0



```

module fsm (
    input logic clk, rst,
    input logic din,
    output logic dout
);

typedef enum logic [2:0] { S0, S1, S11, S110, S1101 } state;
state EA, PE;

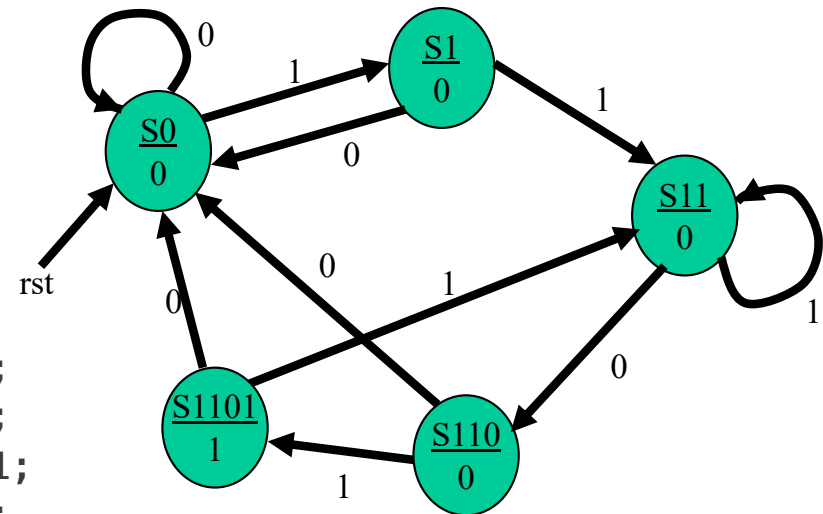
always_ff @(posedge clk or posedge rst) begin
    if (rst)
        EA <= S0;
    else
        EA <= PE;
end

always_comb begin
    unique case (EA)
        S0:      PE = (din == 1'b1) ? S1 : S0;
        S1:      PE = (din == 1'b1) ? S11 : S0;
        S11:     PE = (din == 1'b0) ? S110 : S11;
        S110:    PE = (din == 1'b1) ? S1101 : S0;
        S1101:   PE = (din == 1'b1) ? S11 : S0;
        default: PE = S0;
    endcase
end

assign dout = (EA == S1101);

endmodule

```



Exemplo de test bench

```
module tb;

    logic clk = 0;
    logic rst;
    logic din;
    logic dout;

    fsm dut ( .clk(clk), .rst(rst), .din(din), .dout(dout));

    always #5 clk = ~clk;

    logic [15:0] padrao_de_teste = 16'b1011011010011010;

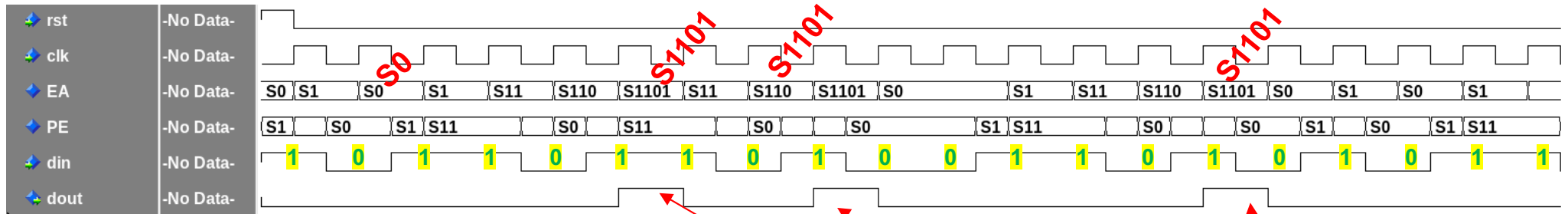
    initial begin
        rst = 1;
        #5 rst = 0;
    end

    initial begin
        for (int i = 15; i >= 0; i--) begin
            din = padrao_de_teste[i];
            #10;
        end

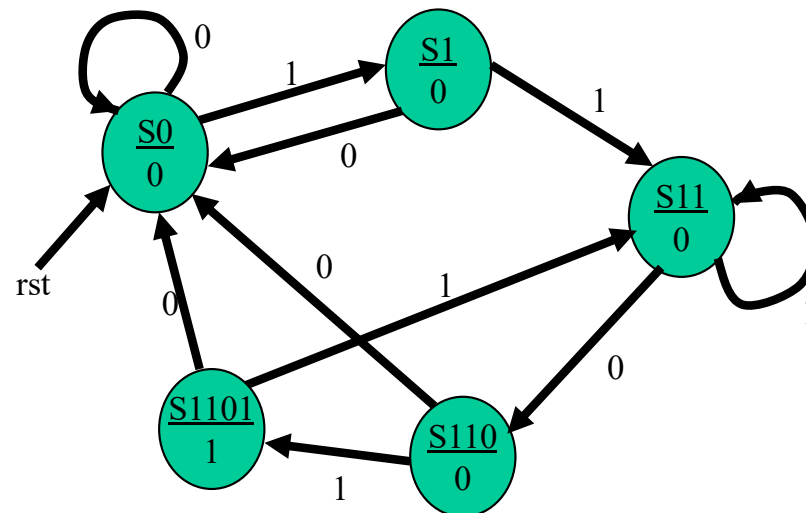
        $finish;
    end

endmodule
```

Detect input sequence 1101



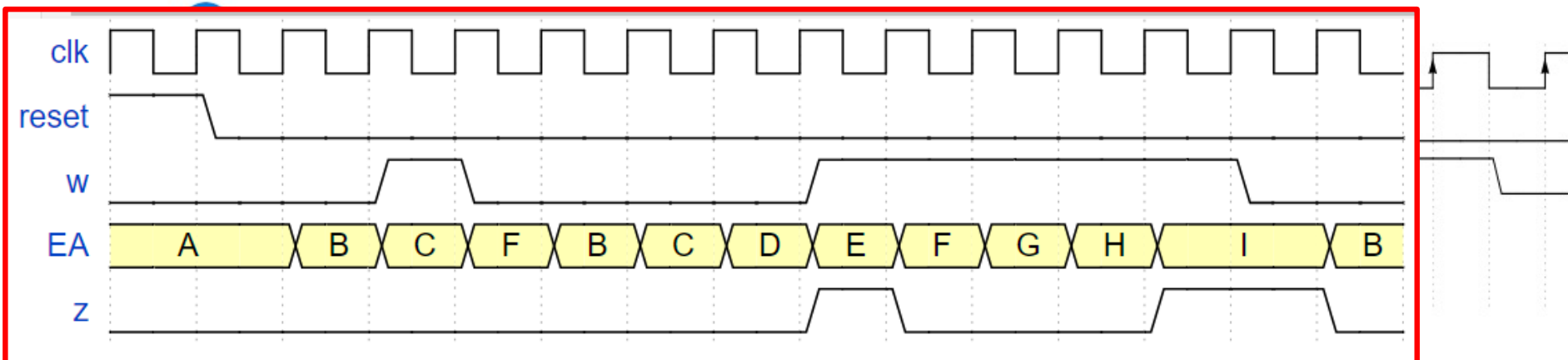
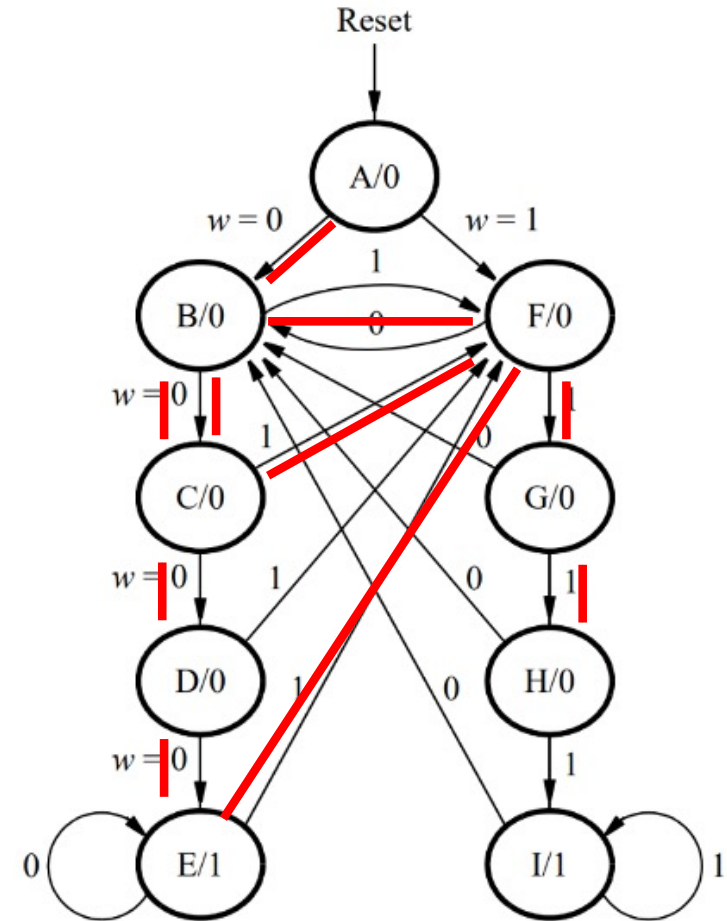
```
logic [15:0] padrao_de_teste = 16'b10111011010011010;
```



Exercício

13. **Máquina de estados (FSM).** Considere a máquina de estados finita (FSM) de Moore al lado, que possui 9 estados válidos (A, B, C, D, E, F, G, H, I), uma entrada (w) e uma saída (z). Lembre-se que por se tratar de uma máquina de Moore, as saídas dependem unicamente do estado atual (EA) da máquina.

Considerando esta FSM e as entradas de clk , $reset$ e w descritas na forma de onda abaixo, determine o comportamento dos sinais EA e z . Considere o valor de w apenas na borda de subida do sinal clk .



MATERIAL DE APOIO

Baixar o arquivo de apoio, o qual contém os exercícios que desenvolveremos com máquinas de estado.

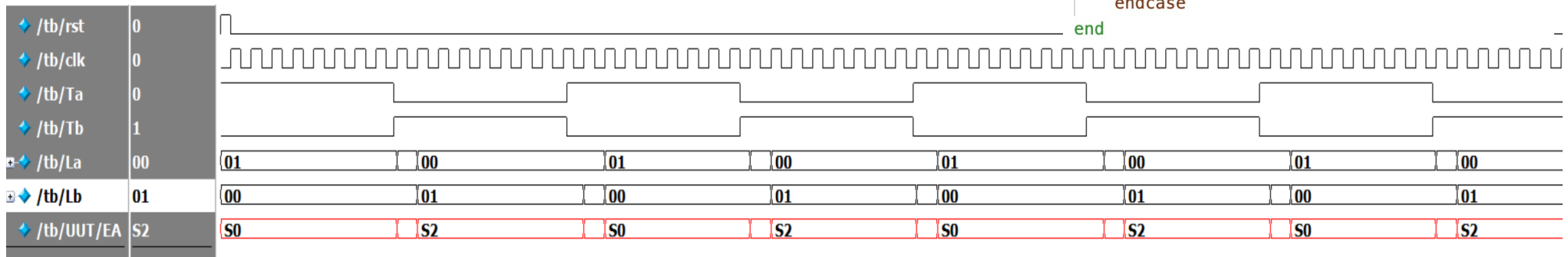
FSM_sv_alunos

- 1_sinaleira
 - fsm.sv
 - sim.do
 - tb.sv
 - wave.do
- 2_detecta_padrao_sv
 - detecta_padrao_tb.sv
 - detecta_padrao.sv
 - sim.do
 - wave.do
- 3a_multiplicador_serial_sv
 - multiplicador_serial_tb.sv
 - multiplicador_serial.sv
 - sim.do
 - wave.do
- 3b_divisor_serial_sv
 - divisor_serial_tb.sv
 - divisor_serial.sv
 - sim.do
 - wave.do

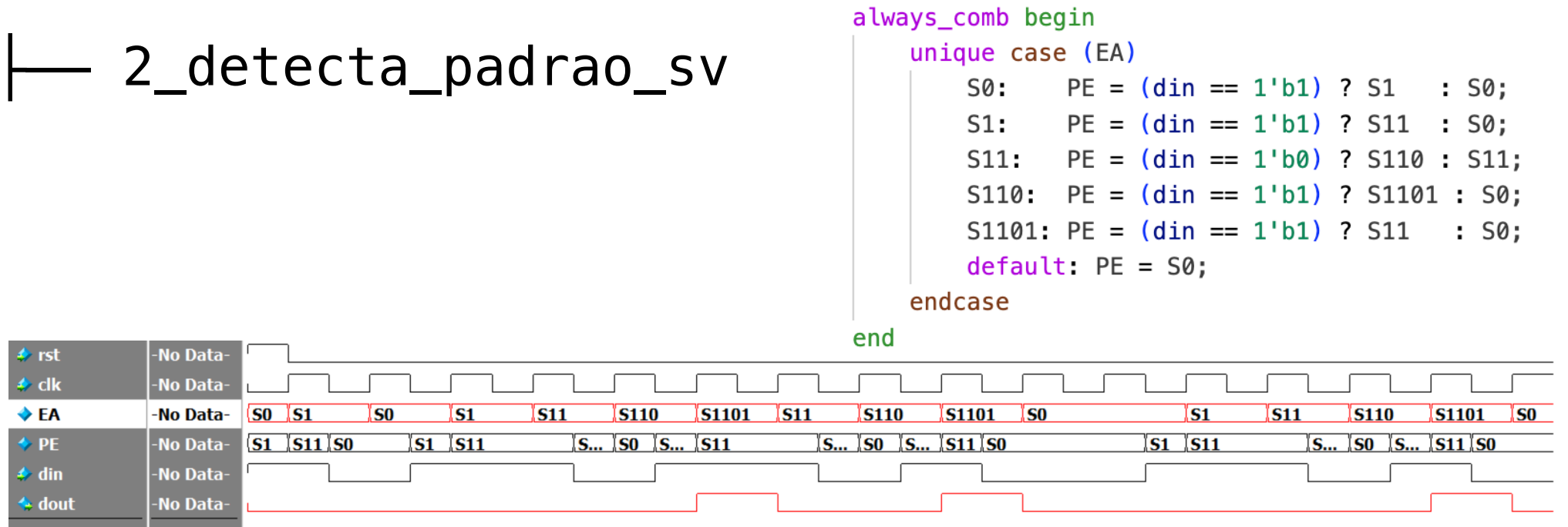
- 4_arbitro_sv
 - arbitro_tb.sv
 - arbitro.sv
 - sim.do
 - wave.do
- 5_tx_serial_sv
 - sim.do
 - tx_serial_tb.sv
 - tx_serial.sv
 - wave.do

Simular e compreender FSMs 1 e 2

1_sinaleira



2_detecta_padrao_sv



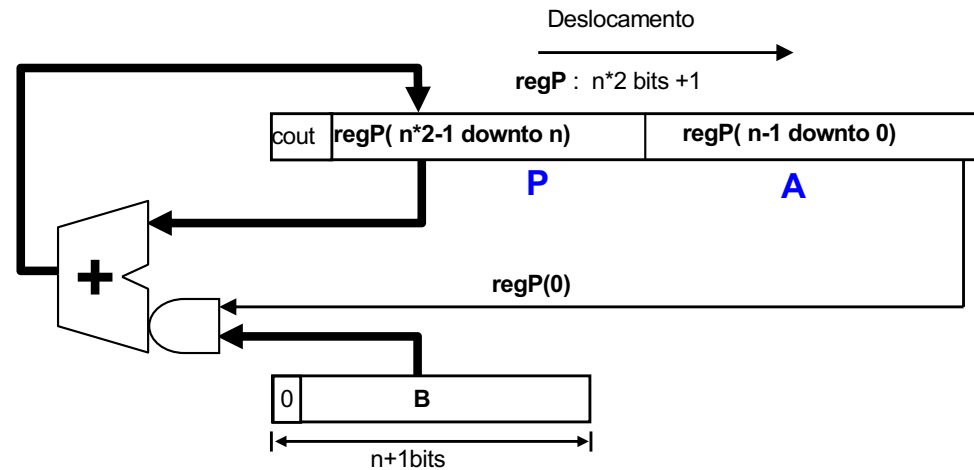
FSM 3 – Multiplicador Serial

$B * A$

$A = 11011 \quad (27)$

$B = 00101 \quad (5)$

$135 \rightarrow 100 \ 00111$



00101
* 11011

00101

1º produto parcial

00101
* 11011

00101
00101

001111

1º bit menos significativo definido (LSB)

00101
* 11011

00101
00101

00000
0001111

2º LSB

00101
* 11011

00101
00101

00000
00101

00110111

3º LSB

00101
* 10111

00101
00101

00000
00101

010000111

Produto Final

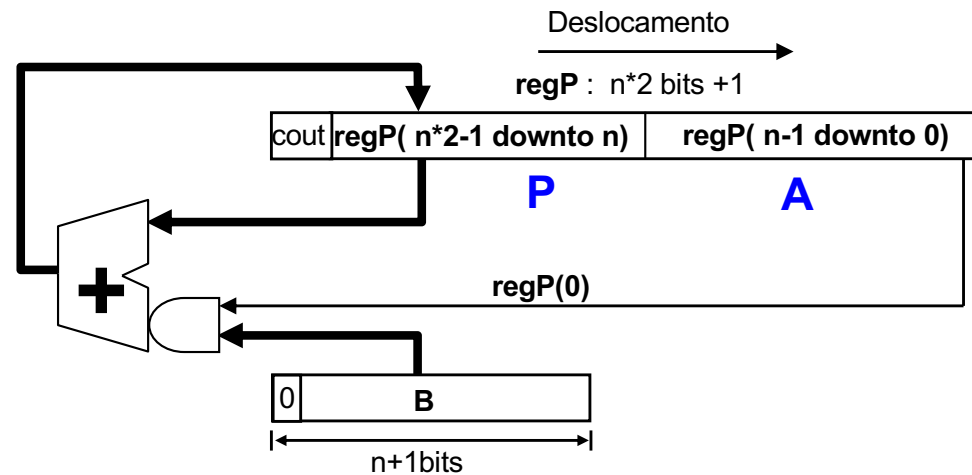
FSM 3 – Multiplicador Serial

$B * A$

$A = 11011$ (27)

$B = 00101$ (5)

$135 \rightarrow 100\ 00111$

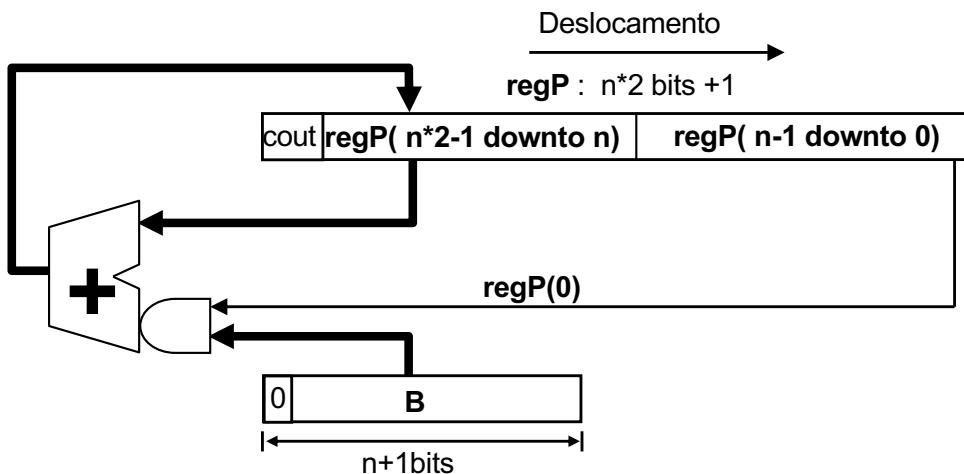
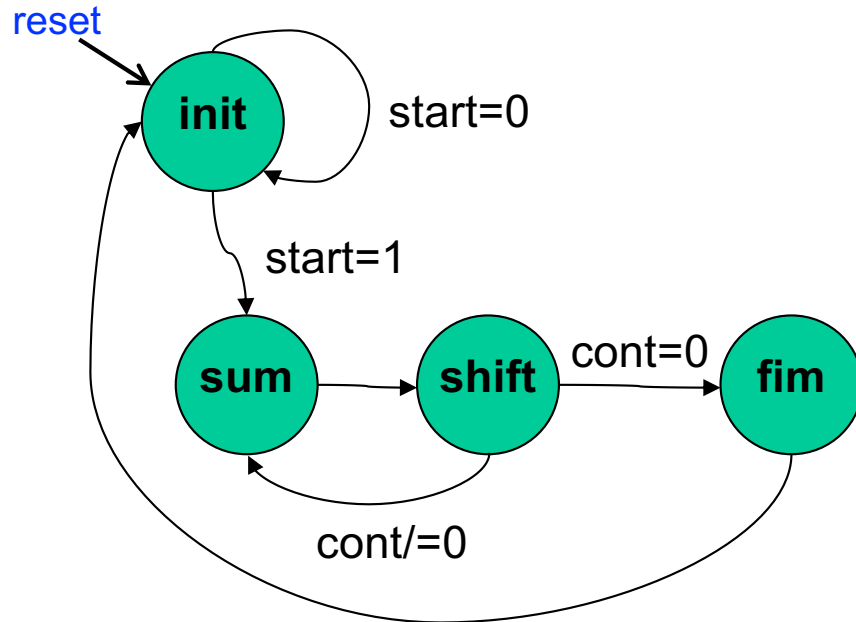


	cout	P					A				
passo	0	0	0	0	0	0	1	1	0	1	1
1	0	0	0	1	0	1	1	1	0	1	1
	0	0	0	0	1	0	1	1	1	0	1
2	0	0	0	1	1	1	1	1	1	0	1
	0	0	0	0	1	1	1	1	1	1	0
3	0	0	0	0	1	1	1	1	1	1	0
	0	0	0	0	0	1	1	1	1	1	1
4	0	0	0	1	1	0	1	1	1	1	1
	0	0	0	0	1	1	0	1	1	1	1
5	0	0	1	0	0	0	0	1	1	1	1
	0	0	0	1	0	0	0	0	1	1	1

FSM 3 – Multiplicador Serial

Bloco de Dados

FSM: controle



```

always_ff @(posedge clock or posedge reset) begin
    if (reset) begin
        regP <= '0;
        cont <= '0;
        produto <= '0;
        end_mul <= '0;
    end else begin
        case (EA)
            INIT: begin
                regP <= {33'd0, A};
                cont <= '0;
                end_mul <= '0;
            end
            SUM: begin
                cont <= cont - 1;
                if (regP[0] == 1'b1)
                    regP[64:32] <= regP[64:32] + {1'b0, B};
            end
            SHIFT: begin
                if (cont == 0)
                    end_mul <= 1'b1;
                regP <= {1'b0, regP[64:1]};
            end
            FIM: begin
                produto <= regP[63:0];
            end
        endcase
    end
end
  
```

```
module tb;
```

```
    logic clock = 0;
    logic reset, start, end_mul;
    logic [31:0] op1, op2;
    logic [63:0] produto;
```

```
    typedef struct packed {
        logic [31:0] A;
        logic [31:0] B;
    } test_vector_t;
```

```
    // Vetores de teste
```

```
    test_vector_t tests[4] = '{ {32'h12345678, 32'h33333333}, {32'hFFFFFFFF, 32'hFFFFFFFF},
                                {32'h00123045, 32'h33333333}, {32'hABCDEF89, 32'h98FEDCBA} };
```

```
    mult_serial dut ( .clock(clock), .reset(reset), .start(start), .A(op1), .B(op2),
                      .end_mul(end_mul), .produto(produto) );
```

```
    always #5 clock = ~clock;
```

```
    initial begin
        reset = 1;
        #3 reset = 0;
    end
```

```
    initial begin // Geração dos operandos
```

```
        for(int i=0; i<4; i++) begin
            op1 = tests[i].A;
            op2 = tests[i].B;
            start = 1;
            #10 start = 0;
            wait (end_mul == 1);
            $display("Produto: %h * %h = %h", op1, op2, produto);
            #40;
```

```
        end
```

```
        #100;
        $finish;
```

```
    end
```

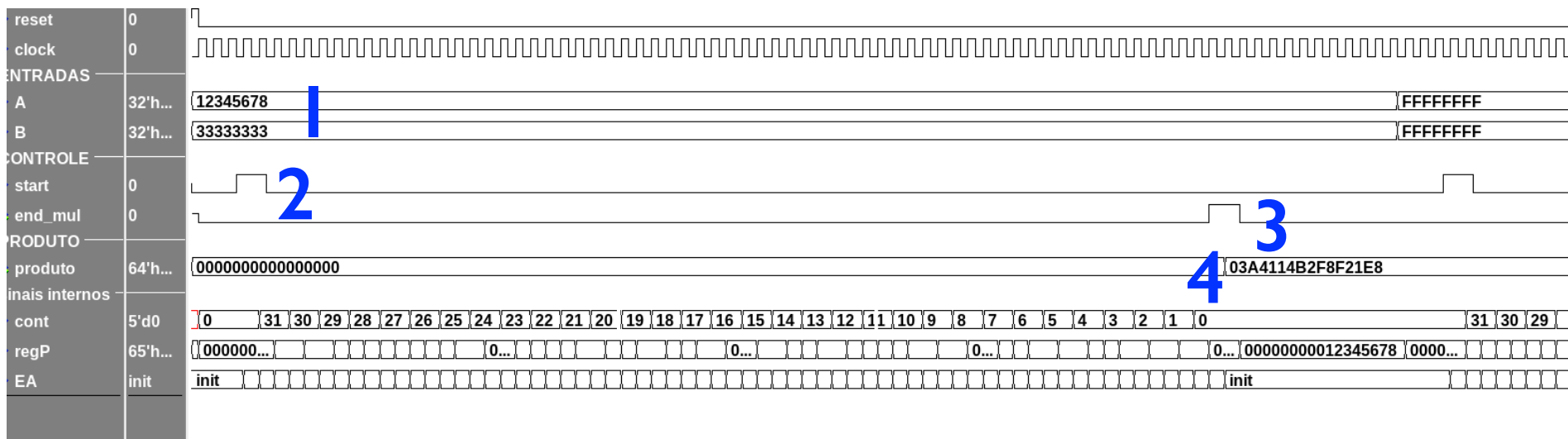
```
endmodule
```

FSM 3 – Multiplicador Serial

Implemente o
multiplicador
serial utilizando
este *test bench*

FSM 3 – Multiplicador Serial

```
test_vector_t tests[4] = { {32'h12345678, 32'h33333333}, {32'hFFFFFFFF, 32'hFFFFFFFF},  
                           {32'h00123045, 32'h33333333}, {32'hABCDEF89, 32'h98FEDCBA} };
```



PRODUTO																																	
produto		0000000...		000000000000000000										000000000000000000										03a4114b2f8f21e8									
sinais internos																																	
cont		27		0		31		30		29																							
EA		SHIFT		IN...		SUM		SHIFT		SUM		SHIFT		SUM		SHIFT																	

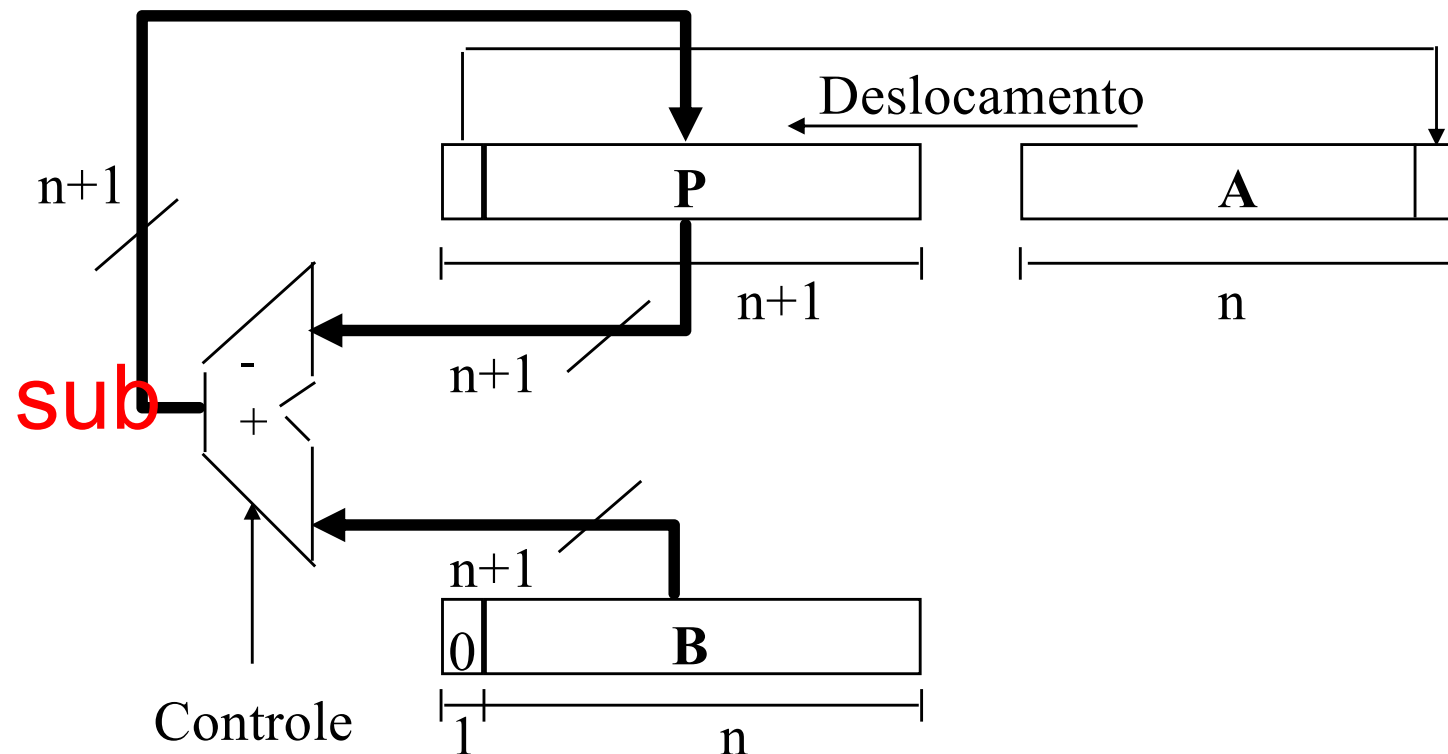
.....

2		1		0											
SHIFT		SUM		SHIFT		FIM		INIT							

1. x"12345678" * x"33333333"
2. Início da multiplicação
3. Final da multiplicação
4. Resultado: x" 0x3A4114B2F8F21E8"

Divisor Serial

- ➔ Solução para a/b : subtrações sucessivas, n passos



- ➔ Algoritmo:
- 1) desloca $P \& A$ p/ esq 1 bit; $sub \leftarrow P - B$;
 - 2) if ($sub < 0$), $A_0 = 0$ else { $A_0 = 1$; $P \leftarrow sub$ }

Divisão A/B

A = 11011 (27)

B = 00101 (5)

- 1) desloca P&A p/ esq 1 bit; $\text{sub} \leftarrow P-B$;
- 2) if ($\text{sub} < 0$), $A0=0$ else { $A0=1$; $P \leftarrow \text{sub}$ }

		P (conterá o resto)					A (conterá a divisão)				
passo		0	0	0	0	0	1	1	0	1	1
1		0	0	0	0	0	1	0	1	1	0
		0	0	0	0	1	1	0	1	1	0
2											
3											
4											
5											

Divisão A/B

A = 11011 (27)
B = 00101 (5)

- 1) desloca P&A p/ esq 1 bit; $\text{sub} \leftarrow P-B$;
- 2) if ($\text{sub} < 0$), $A0=0$ else { $A0=1$; $P \leftarrow \text{sub}$ }

passo		P (conterá o resto)					A (conterá a divisão)				
		0	0	0	0	0	1	1	0	1	1
1	0	0	0	0	0	1	1	0	1	1	0
	0	0	0	0	0	1	1	0	1	1	0
2	0	0	0	0	1	1	0	1	1	0	0
	0	0	0	0	1	1	0	1	1	0	0
3											
4											
5											

Divisão A/B

A = 11011 (27)
B = 00101 (5)

- 1) desloca P&A p/ esq 1 bit; $\text{sub} \leftarrow P-B$;
- 2) if ($\text{sub} < 0$), $A0=0$ else { $A0=1$; $P \leftarrow \text{sub}$ }

		P (conterá o resto)					A (conterá a divisão)				
passo		0	0	0	0	0	1	1	0	1	1
1		0	0	0	0	1	1	0	1	1	0
		0	0	0	0	1	1	0	1	1	0
2		0	0	0	1	1	0	1	1	0	0
		0	0	0	1	1	0	1	1	0	0
3		0	0	1	1	0	1	1	0	0	0
		0	0	0	0	1	1	1	0	0	1
4											
5											

00110 - 00101 = 001

Divisão A/B

A = 11011 (27)
B = 00101 (5)

- 1) desloca P&A p/ esq 1 bit; $sub \leftarrow P-B$;
- 2) if ($sub < 0$), $A0=0$ else { $A0 = 1$; $P \leftarrow sub$ }

		P (conterá o resto)					A (conterá a divisão)				
passo		0	0	0	0	0	1	1	0	1	1
1	0	0	0	0	0	1	1	0	1	1	0
	0	0	0	0	0	1	1	0	1	1	0
2	0	0	0	0	1	1	0	1	1	0	0
	0	0	0	0	1	1	0	1	1	0	0
3	0	0	0	1	1	0	1	1	0	0	0
	0	0	0	0	0	1	1	1	0	0	1
4	0	0	0	0	1	1	1	0	0	1	0
	0	0	0	0	1	1	1	0	0	1	0
5											

Divisão A/B

A = 11011 (27)
B = 00101 (5)

- 1) desloca P&A p/ esq 1 bit; $\text{sub} \leftarrow P-B$;
- 2) if ($\text{sub} < 0$), $A0=0$ else { $A0=1$; $P \leftarrow \text{sub}$ }

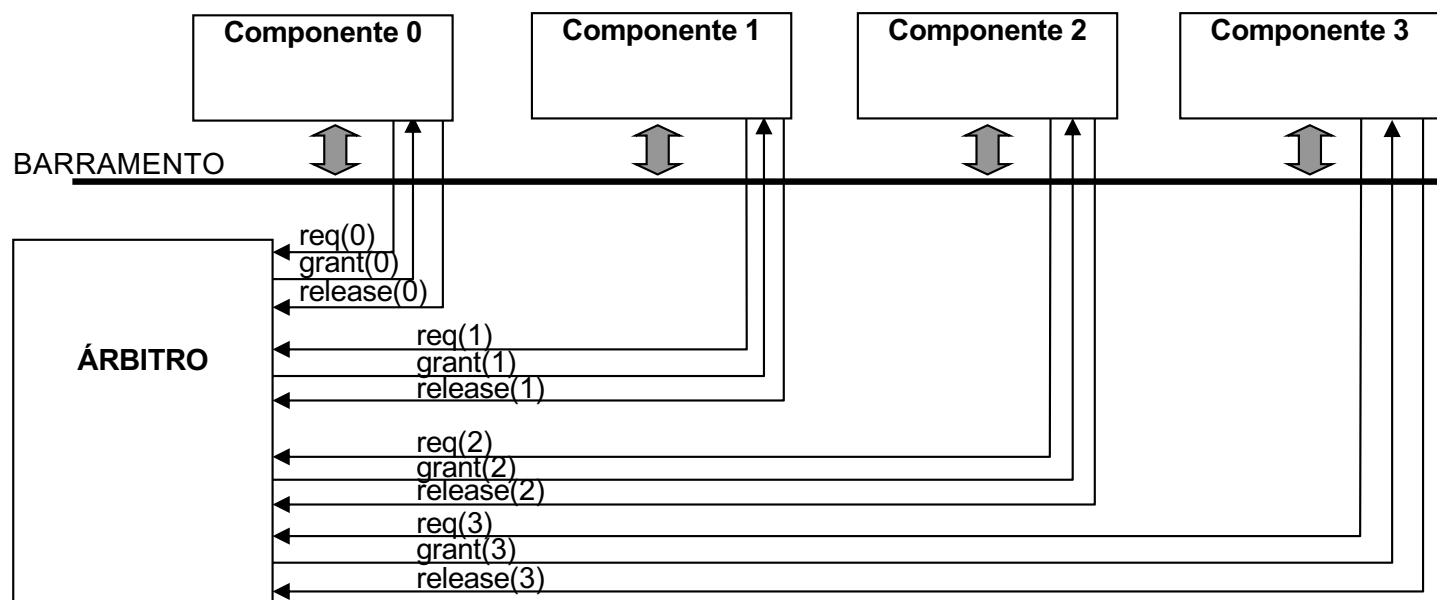
		P (conterá o resto)					A (conterá a divisão)				
passo		0	0	0	0	0	1	1	0	1	1
1	0	0	0	0	0	1	1	0	1	1	0
	0	0	0	0	0	1	1	0	1	1	0
2	0	0	0	0	1	1	0	1	1	0	0
	0	0	0	0	1	1	0	1	1	0	0
3	0	0	0	1	1	0	1	1	0	0	0
	0	0	0	0	0	1	1	1	0	0	1
4	0	0	0	0	1	1	1	0	0	1	0
	0	0	0	0	1	1	1	0	0	1	0
5	0	0	0	1	1	1	0	0	1	0	0
	0	0	0	0	1	0	0	0	1	0	1

Resto = 2

resultado=5

FSM 4 – Árbitro de Barramento

Um **árbitro** é um circuito que permite que um dado componente de um sistema digital (processador ou memória, por exemplo) se comunique com o **barramento**. Toda vez que um dado *componente* deseja comunicar-se com o barramento para enviar dados a outro componente, este componente deve ativar o sinal **req_i**. Se o barramento estiver livre, o árbitro ativa o sinal **grant_i**, por um ciclo de clock, avisando ao módulo que solicitou que pode colocar dados no barramento. Ao final da transmissão, o componente que detém o barramento ativa o sinal **release_i**, por um ciclo de clock informando ao árbitro que o barramento está livre.

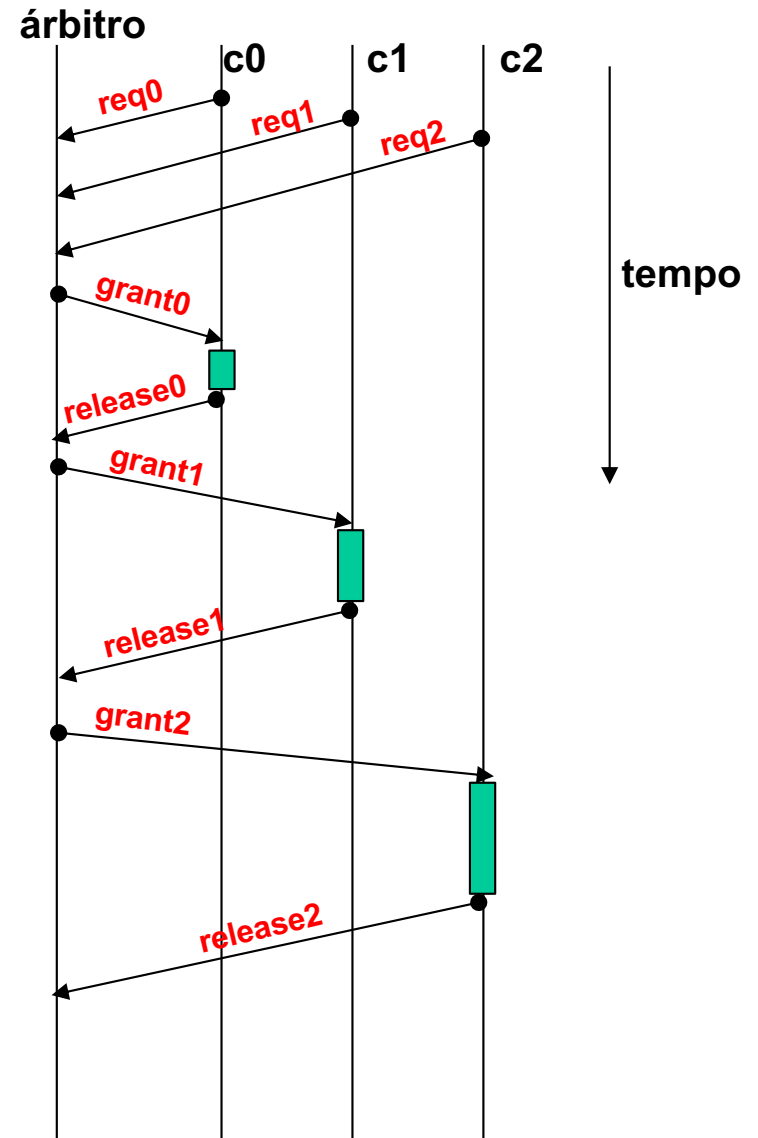
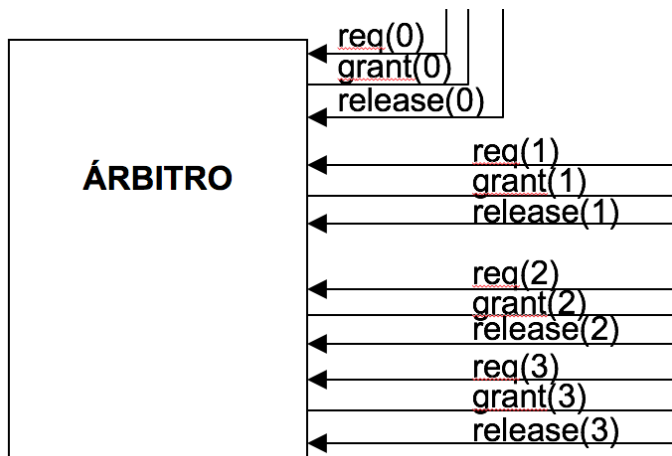


FSM 4 – Árbitro de Barramento

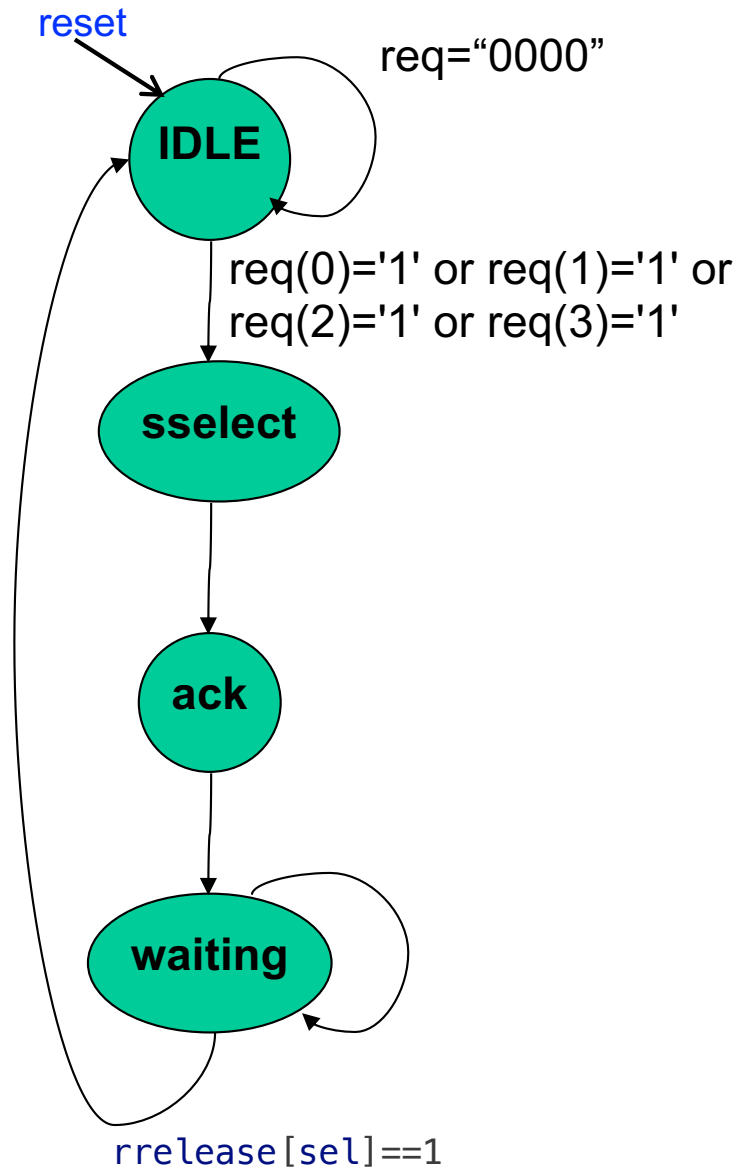
- Implementar o árbitro em SystemVerilo

Operação do árbitro.

- ficar sequencialmente lendo os sinais de solicitação (*req(0)* a *req(3)*).
- quando um dos sinais *req_i* estiver ativo, o sinal *grant_i* sobe por um ciclo de clock, aguardando-se que o sinal *release_i* suba.
- na subida do sinal *release_i* reinicia-se o **processo de leitura sequencial, a partir do sinal *req_{i+1}***. Isto é feito para evitar que um dado componente seja sempre atendido em detrimento dos outros.



Árbitro



Bloco de Dados

```

module arbitro (
    input logic clock, reset,
    input logic [3:0] req, rrelease,
    output logic [3:0] grant
);

typedef enum logic [1:0] {idle, sselect, ack,
                        waiting } state;
state EA, PE;

logic [1:0] sel;

// Bloco de controle
// máquina de estados

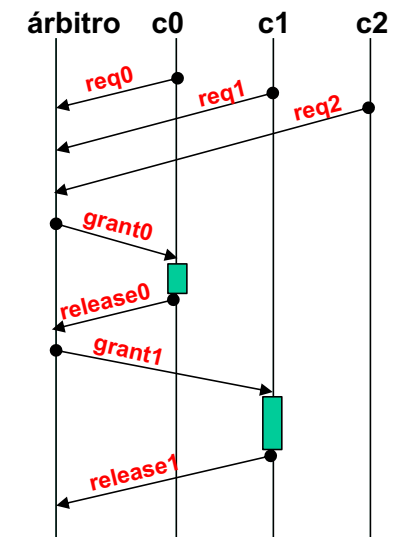
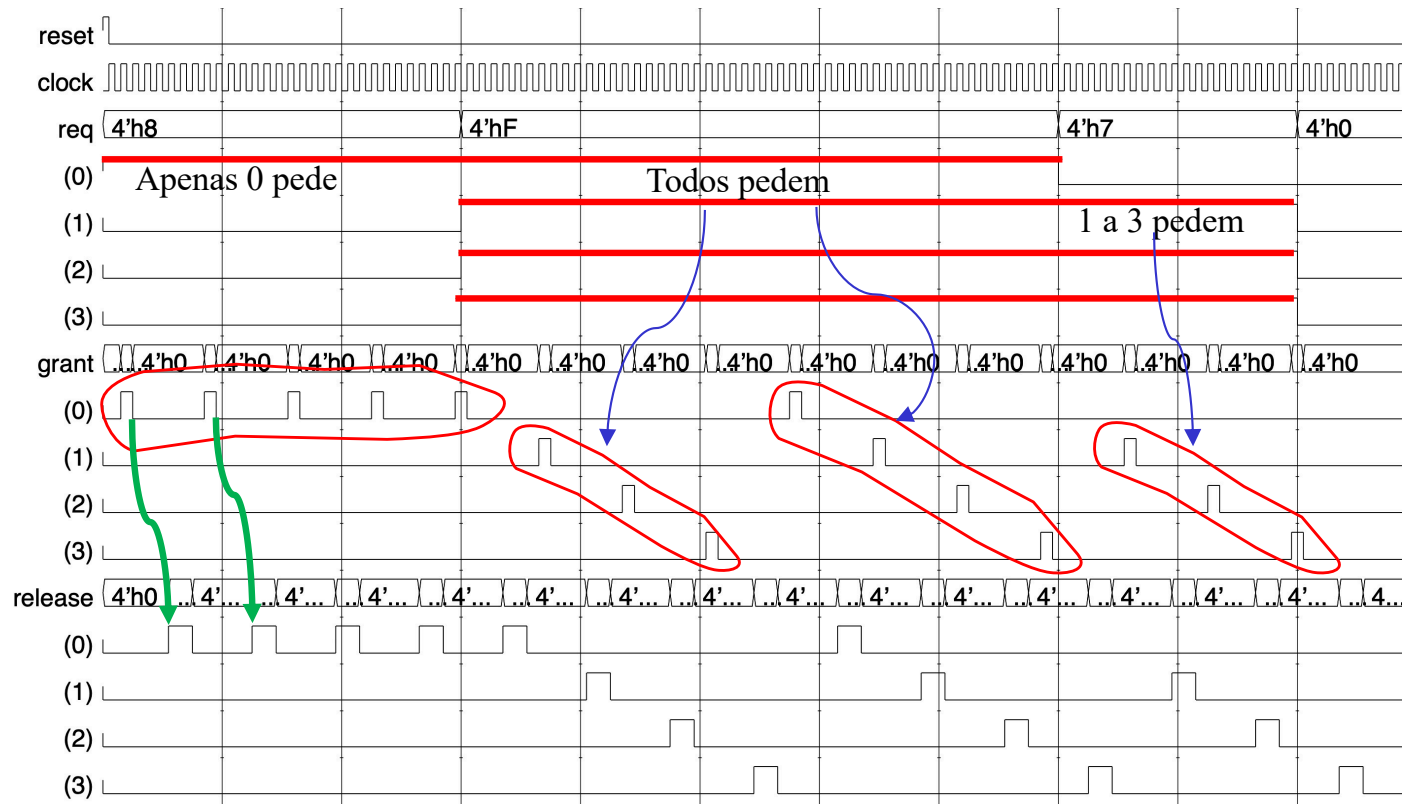
// Bloco de dados
always_ff @(posedge clock or posedge reset) begin
    if (reset) begin
        sel <= 2'b00;
    end else if (EA == sselect) begin
        if ( req[(sel + 1) & 2'b11])
            sel <= sel + 1;
        else if (req[(sel + 2) & 2'b11])
            sel <= sel + 2;
        else if (req[(sel + 3) & 2'b11])
            sel <= sel + 3;
        else
            sel <= sel;
    end
end

// Grant logic
assign grant = (EA == ack) ? (4'b0001 << sel) : 4'b0000;

endmodule
  
```

Registrador sel/ diz quem está sendo atendido

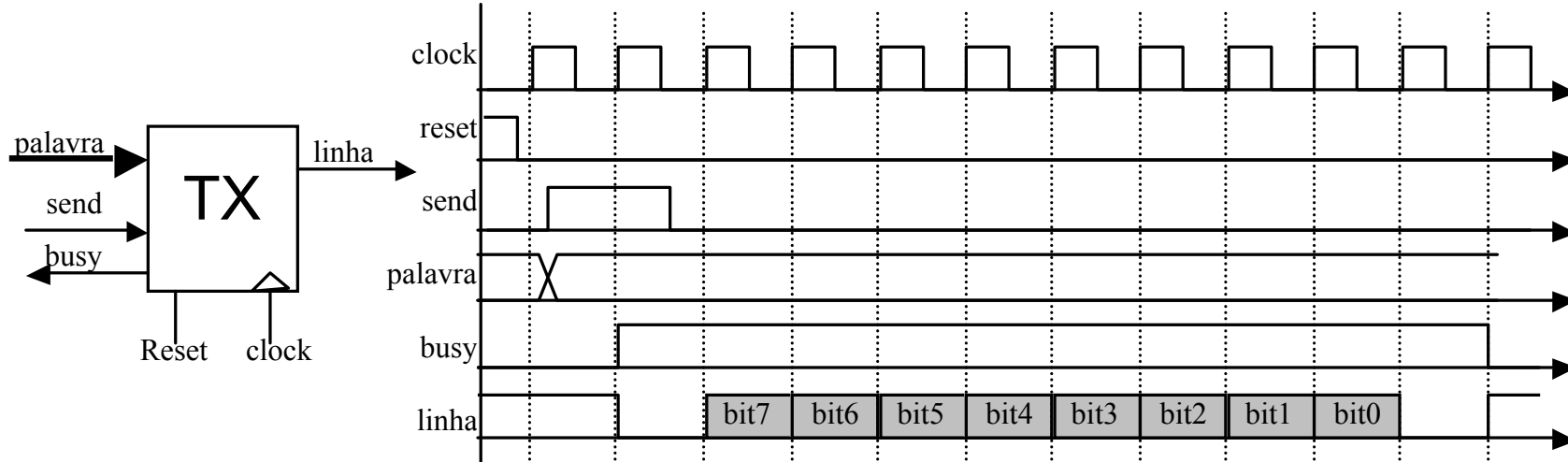
FSM 4 – Simulação do árbitro



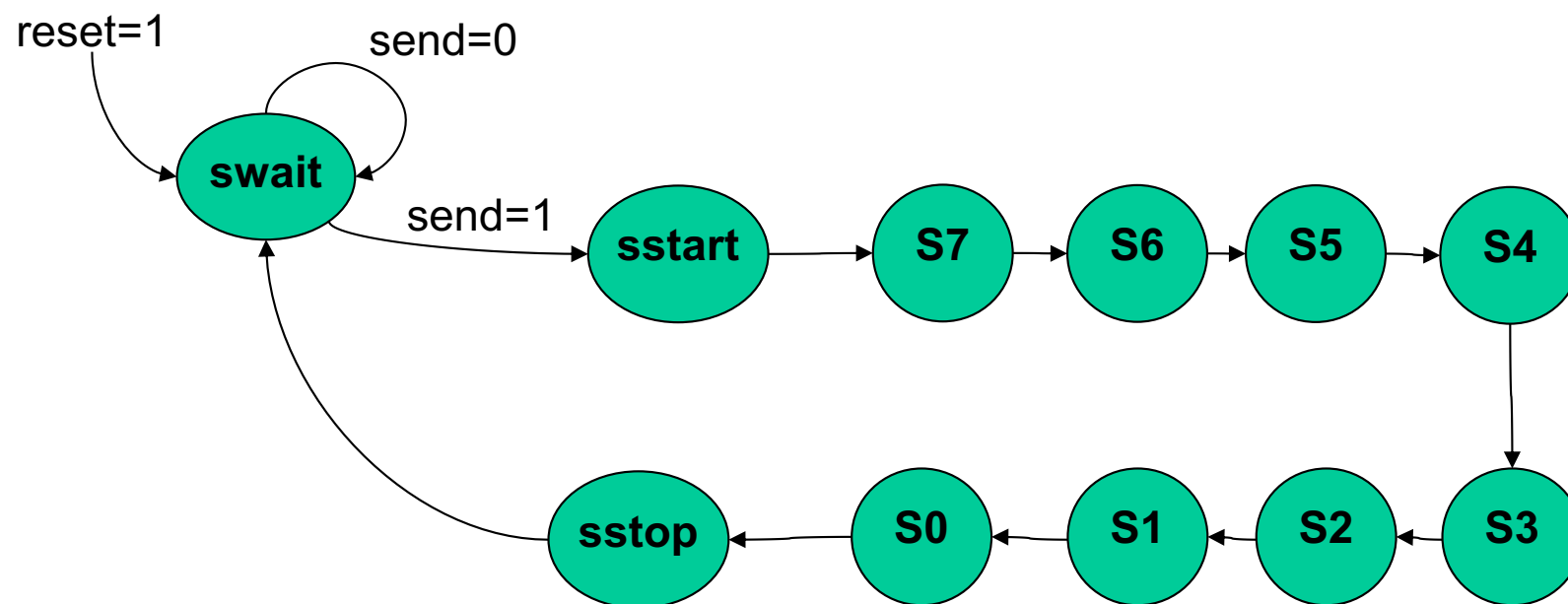
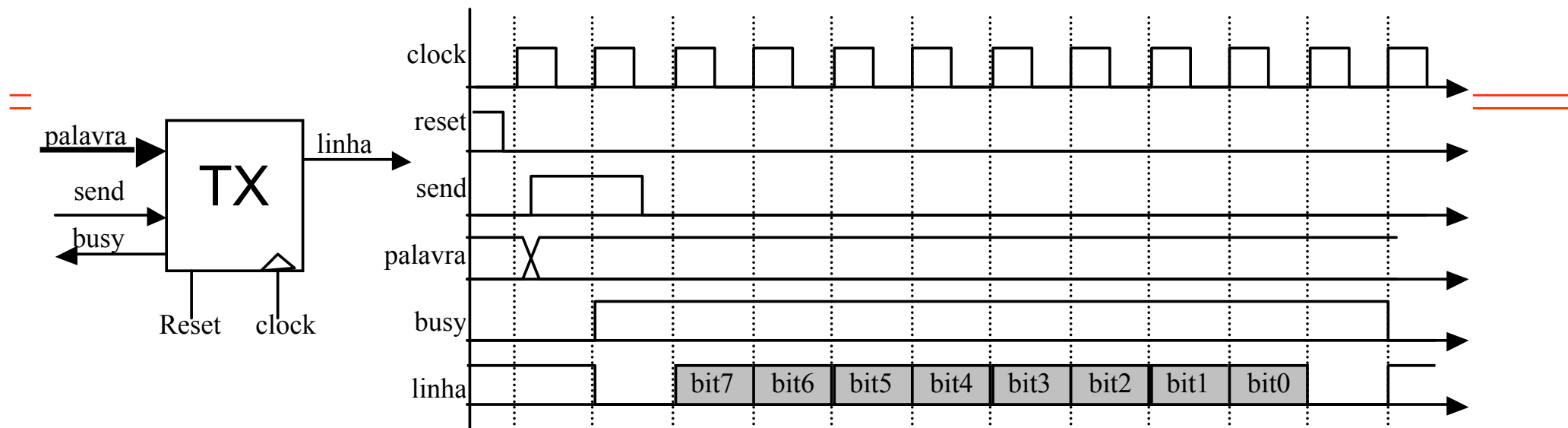
FSM 5 - Transmissão de Dados Serial

Considere o circuito abaixo – TX, o qual implementa uma transmissão serial de dados. A “linha” de dados (saída do circuito) está por default em ‘1’, indicando que não há transmissão de dados. O protocolo de transmissão é o seguinte:

1. O mundo externo ao TX (*test bench*) coloca um byte válido em “palavra”, e sobe o sinal “send”, indicando a TX que há dado a ser enviado para a “linha”.
2. No primeiro ciclo de *clock* após a subida de “send” o módulo TX sobe o sinal de “busy”, impedindo que o mundo externo solicite novos dados. Concorrentemente a esta ação a linha sai do repouso, indo a zero por um ciclo (bit denominado *start bit*).
3. Nos próximos 8 ciclos de *clock* o dado escrito em palavra é colocado bit a bit na “linha”
4. No décimo ciclo de *clock* após a detecção do *send* a linha vai a zero (stop bit) e o *busy* desce no final do ciclo.



DICA: a máquina de estados tem 11 estados. A controle da saída “linha” pode ficar dentro do processo combinacional da máquina de estados. O sinal de “busy” pode ser implementado como uma atribuição concorrente fora dos processos.



FSM 5 - Transmissão de Dados Serial

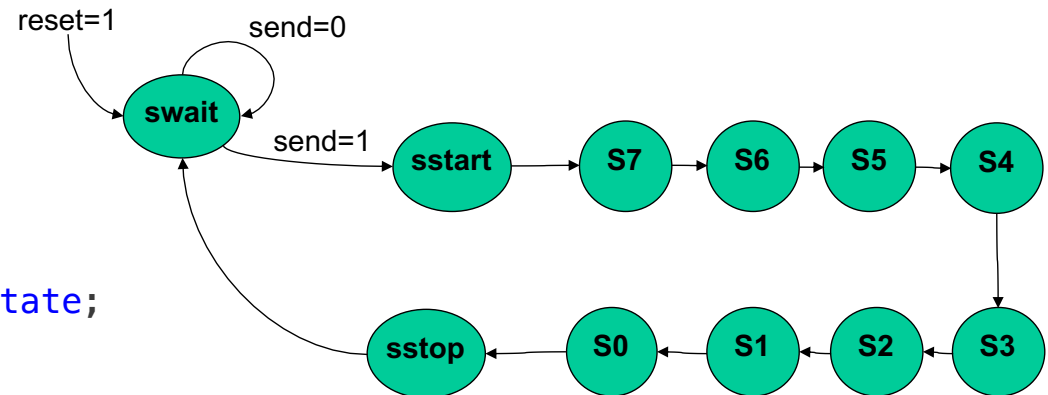
```
module transmissor (  
    input logic clock, reset, send,  
    input logic [7:0] palavra,  
    output logic busy,  
    output logic linha  
);  
typedef enum logic [3:0] { completar } state;  
state EA, PE;
```

descrever máquina de estados

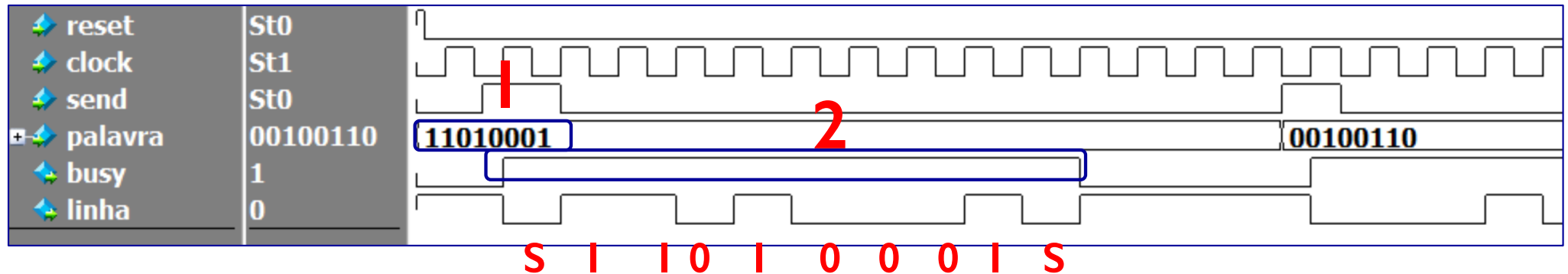
```
// Sinal de ocupado  
assign busy = completar;
```

```
always_comb begin  
    unique case (EA)  
        S7: linha = palavra[7];  
        ...  
        S0: linha = palavra[0];  
        SSART, SSTOP: linha = 1'b0;  
        default: linha = 1'b1;  
    endcase  
end
```

```
endmodule
```



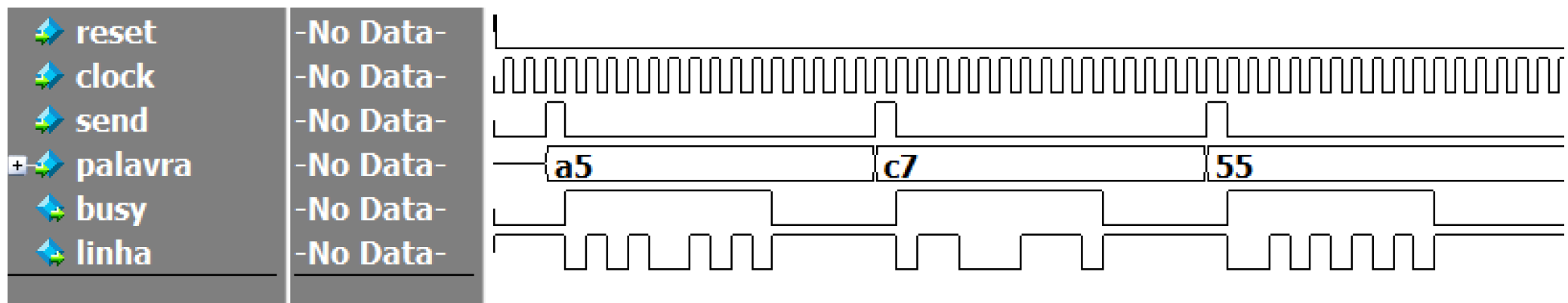
Transmissão de Dados Serial



1. Indicação de *send* com palavra 1101 0001
2. *busy* fica ativo durante toda a transmissão

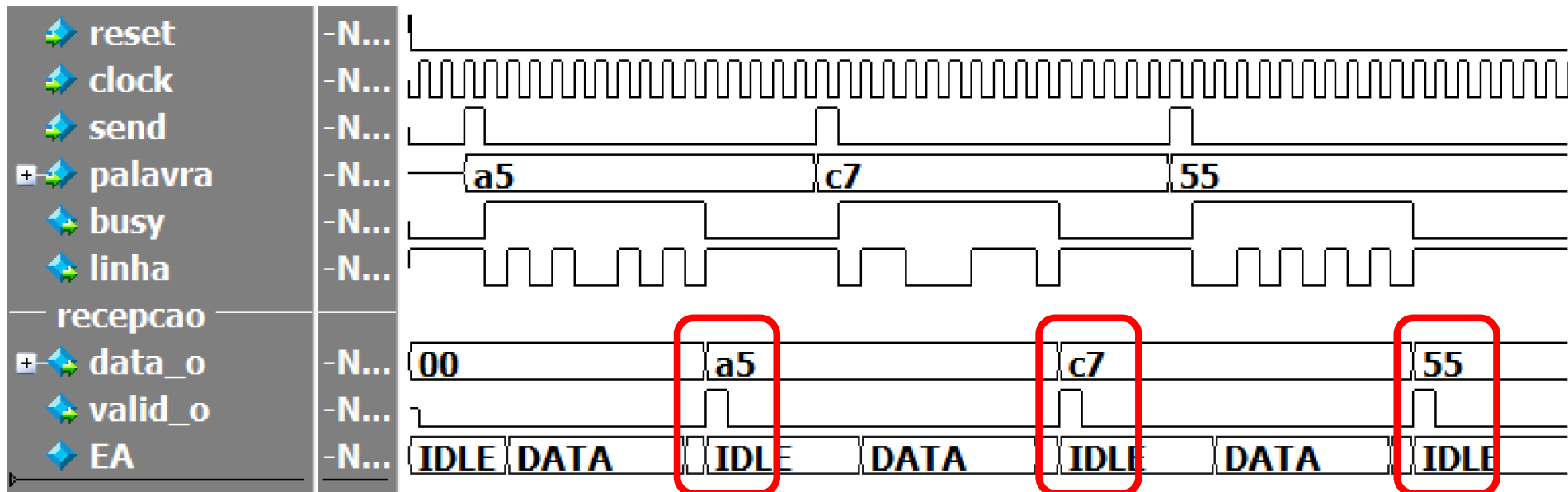
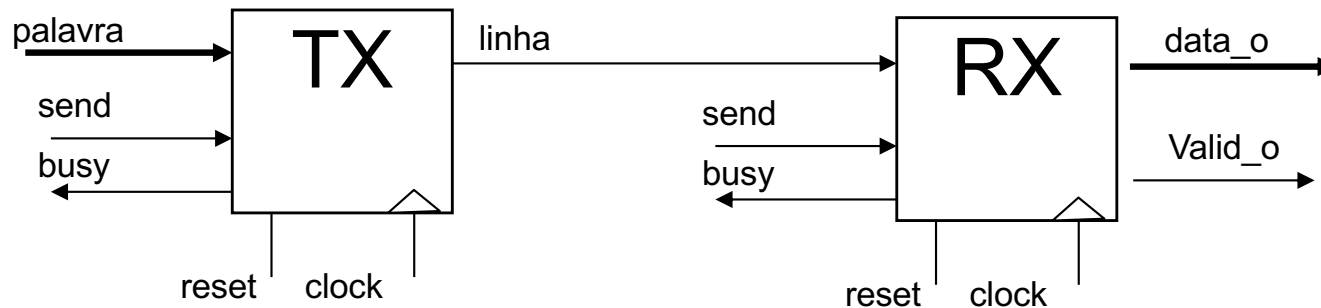
Na linha de dados temos {start bit, 1, 1, 0, 1, 0, 0, 0, 1, stop bit}

Resultado com o testbench fornecido



TXRX

O trabalho consiste em desenvolver o módulo RX. O RX detecta a linha indo para zero, armazena os 8 bits que estão chegando, e gera na saída a palavra `data_o` igual a palavra transmitida serialmente. **Ver o tb, o qual já instancia o RS (`rx_serial u_rx (.clock(clock),`).**



12. **Máquina de estados (FSM).** Distribuidor de Café. Este distribuidor vende café a R\$ 0,75, aceitando moedas de 25 e 50 centavos. Existe na máquina uma fenda para inserir moedas com um circuito capaz de reconhecer moedas de R\$ 0,25 e R\$ 0,50, e é capaz de devolver qualquer outro tipo de moeda ou objeto não reconhecido. Além disso, o usuário pode desistir da transação e apertar a tecla DEV que devolve as moedas inseridas até o momento. A devolução de excesso de moedas é automática sempre que o valor inserido antes de retirar um café ultrapassar R\$ 0,75. **Café só é ativado se o usuário pressionar ASK e a soma acumulada for R\$ 0,75.**



- **Entradas:** M25 (moeda de 25 centavos), M50 (moeda de 50 centavos), DEV (pedido de devolução do valor inserido), ASK (pedido de café), clock, reset. Apenas uma entrada M25/M50/DEV/ASK pode ser ativada por vez – não há sobreposição das entradas.
- **Saídas:** D25 (devolução de moeda de 25 centavos), D50 (devolução de moeda de 50 centavos), café (café fornecido).

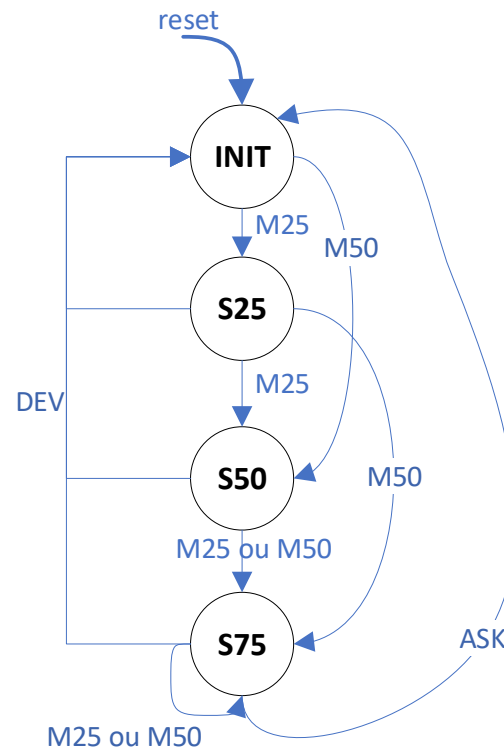
Modele o distribuidor de café como uma máquina de estados finita (FSM) – desenhar e explicar a FSM (não apresentar código SystemVerilog nesta questão). Apresentar as condições em que cada saída é ativada e apresente um texto sucinto das ações que ocorrem em cada estado.

12. **Máquina de estados (FSM).** Distribuidor de Café. Este distribuidor vende café a R\$ 0,75, aceitando moedas de 25 e 50 centavos. Existe na máquina uma fenda para inserir moedas com um circuito capaz de reconhecer moedas de R\$ 0,25 e R\$ 0,50, e é capaz de devolver qualquer outro tipo de moeda ou objeto não reconhecido. Além disso, o usuário pode desistir da transação e apertar a tecla DEV que devolve as moedas inseridas até o momento. A devolução de excesso de moedas é automática sempre que o valor inserido antes de retirar um café ultrapassar R\$ 0,75. Café só é ativado se o usuário pressionar ASK e a soma acumulada for R\$ 0,75.



- **Entradas:** M25 (moeda de 25 centavos), M50 (moeda de 50 centavos), DEV (pedido de devolução do valor inserido), ASK (pedido de café), clock, reset. Apenas uma entrada M25/M50/DEV/ASK pode ser ativada por vez – não há sobreposição das entradas.
- **Saídas:** D25 (devolução de moeda de 25 centavos), D50 (devolução de moeda de 50 centavos), café (café fornecido).

Modele o distribuidor de café como uma máquina de estados finita (FSM) – desenhar e explicar a FSM (não apresentar código SystemVerilog nesta questão). Apresentar as condições em que cada saída é ativada e apresente um texto sucinto das ações que ocorrem em cada estado.



Solução

```
module transmissor (  
    input logic clock, reset, send,  
    input logic [7:0] palavra,  
    output logic busy,  
    output logic linha  
);  
  
typedef enum logic [3:0] { SWAIT, SSART, S7, S6, S5, S4, S3, S2, S1, S0, SSTOP} state;  
state EA, PE;  
  
// Estado atual  
always_ff @(posedge clock or posedge reset) begin  
    if (reset)  
        EA <= SWAIT;  
    else  
        EA <= PE;  
    end  
  
// Próximo estado  
always_comb begin  
    case (EA)  
        SWAIT: PE = send ? SSART : SWAIT;  
        SSART: PE = S7;  
        S7: PE = S6;  
        S6: PE = S5;  
        S5: PE = S4;  
        S4: PE = S3;  
        S3: PE = S2;  
        S2: PE = S1;  
        S1: PE = S0;  
        S0: PE = SSTOP;  
        SSTOP: PE = SWAIT;  
        default: PE = SWAIT;  
    endcase  
end  
  
// Sinal de ocupado  
assign busy = (EA == SWAIT) ? 1'b0 : 1'b1;  
  
// Saída serial  
always_comb begin  
    unique case (EA)  
        S7: linha = palavra[7];  
        S6: linha = palavra[6];  
        S5: linha = palavra[5];  
        S4: linha = palavra[4];  
        S3: linha = palavra[3];  
        S2: linha = palavra[2];  
        S1: linha = palavra[1];  
        S0: linha = palavra[0];  
        SSART,  
        SSTOP: linha = 1'b0;  
        default: linha = 1'b1;  
    endcase  
end  
endmodule
```


Exemplo

Exemplo de FSM

- ✓ Construa uma fechadura eletrônica de combinação com um botão de reset, dois botões numéricos (0 e 1) e uma saída de destravamento.
- ✓ A combinação deve ser **01011**.

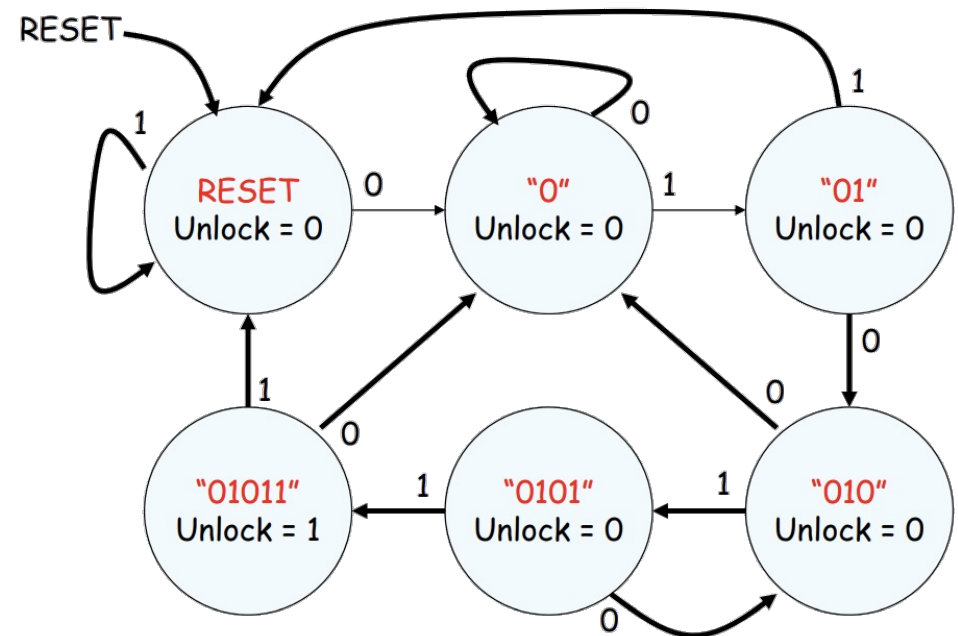
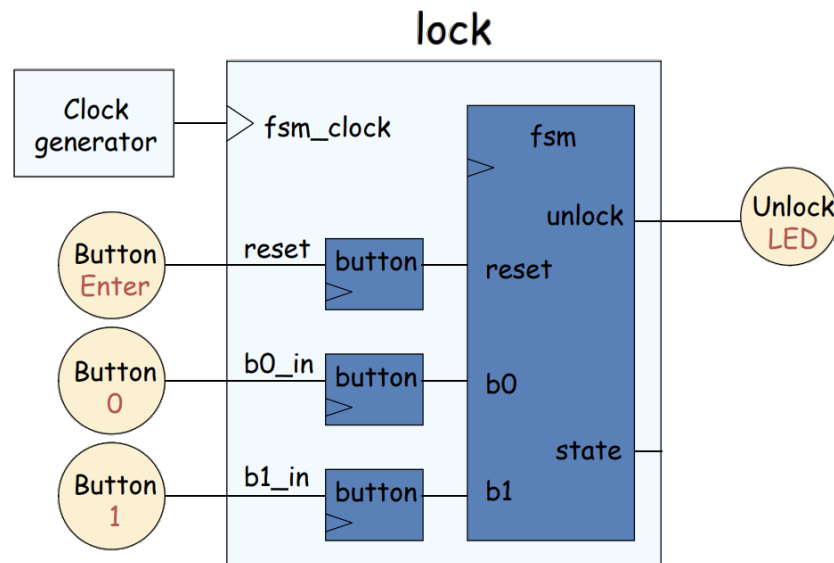


Projete a FSM de fechadura (diagrama de blocos, transições de estado)

DIAGRAMA DE BLOCOS & TRANSIÇÕES DE ESTADOS

Exemplo de FSM

- ✓ Construa uma fechadura eletrônica de combinação com um botão de reset, dois botões numéricos (0 e 1) e uma saída de destravamento.
- ✓ A combinação deve ser **01011**.



Fim

Parte FSM