CIRCUITOS SEQUENCIAIS

Máquina de estados finita (FSM)

Prof. Fernando G. Moraes

				digitais em RTL.
21	SEX	13/05/2022	CD	Máquinas de Estado Finita (FSMs) – máquinas de Moore e de Mealy; compreensão usando modelagem com HDL.
22	QUA	18/05/2022	CD	Modelagem e simulação de FSMs utilizado exemplos relacionados a blocos utilizados em processadores, como multiplicação, árbitro, e transmissão serial.
23	SEX	20/05/2022	CD	Modelagem e simulação de FSMs.
24	QUA	25/05/2022	CD	Modelagem e simulação de FSMs.
25	SEX	27/05/2022	CD	Instrumento de Avaliação 3 (30% G1) – TRABALHO - Relacionado à modelagem de sistemas digitais no nível RTL e FSMs - NÃO SUBSTITUÍVEL POR PS.

Máquina de estados finita (FSM)

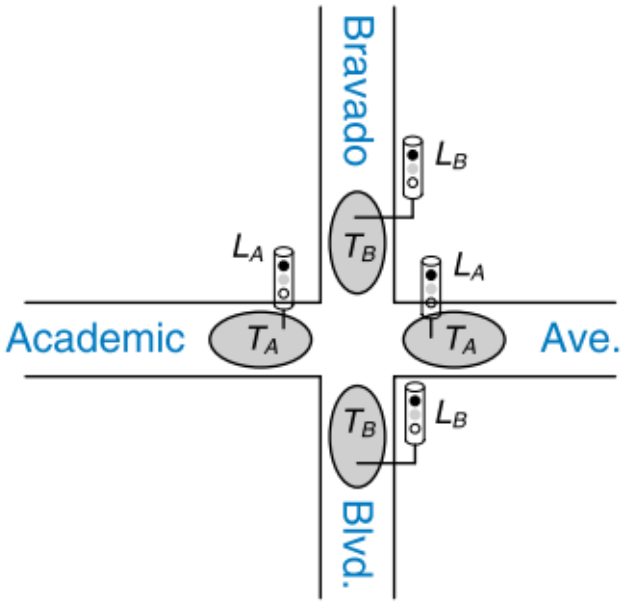
- Uma máquina de estados finita (FSM - do inglês *Finite State Machine*) ou autômato finito é um modelo matemático usado para **representar** programas de computadores ou circuitos lógicos.

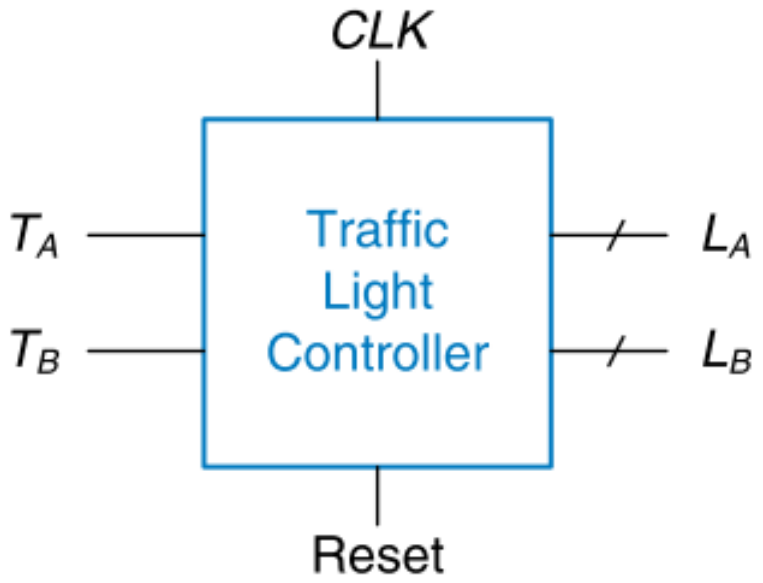
- O conceito é concebido como uma máquina abstrata que deve estar em um de um **número finito de estados**.

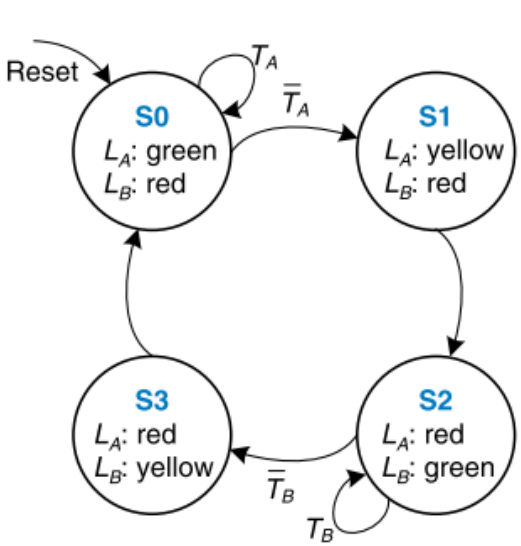
- A máquina está em apenas **um estado por vez**, este estado é chamado de **estado atual**. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada em um estado, no início do sistema, até o momento presente.

- 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 → VHDL



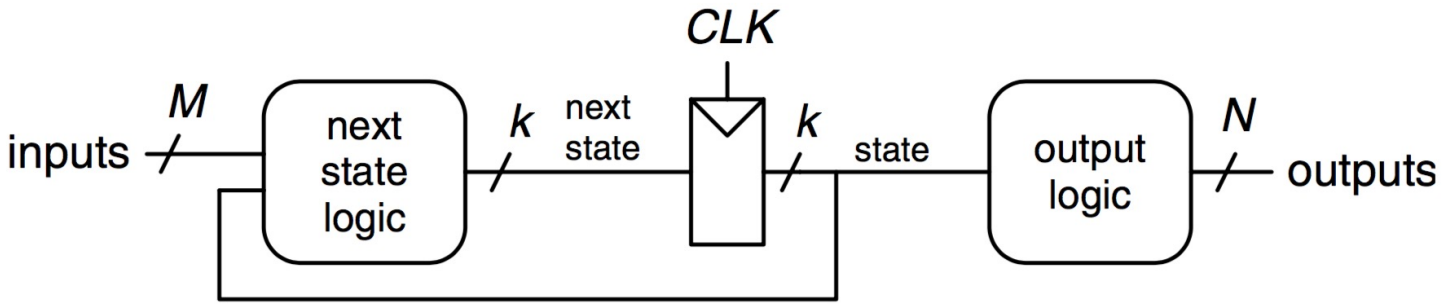




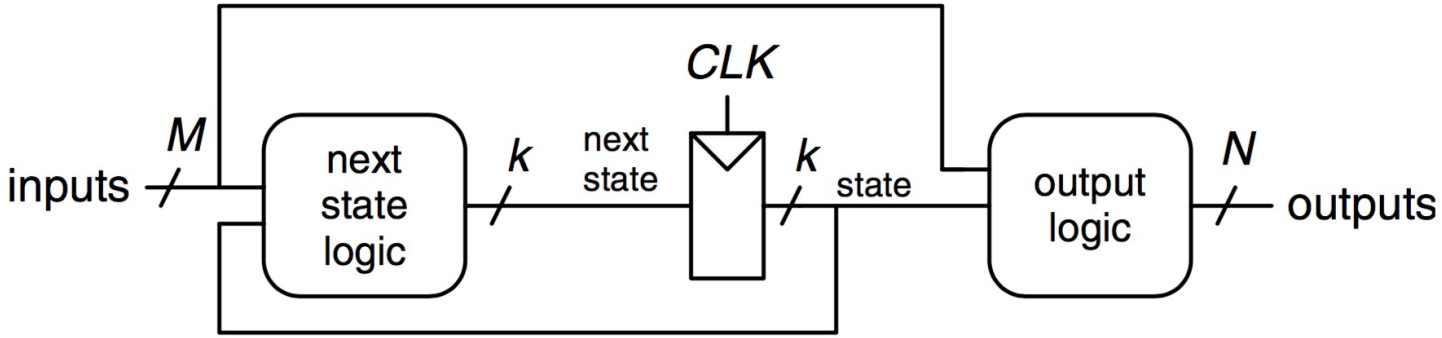
VHDL
Verilog

Solução completa em "Digital Design and Computer Architecture". Sec 3.4.15

Tipos de FSM



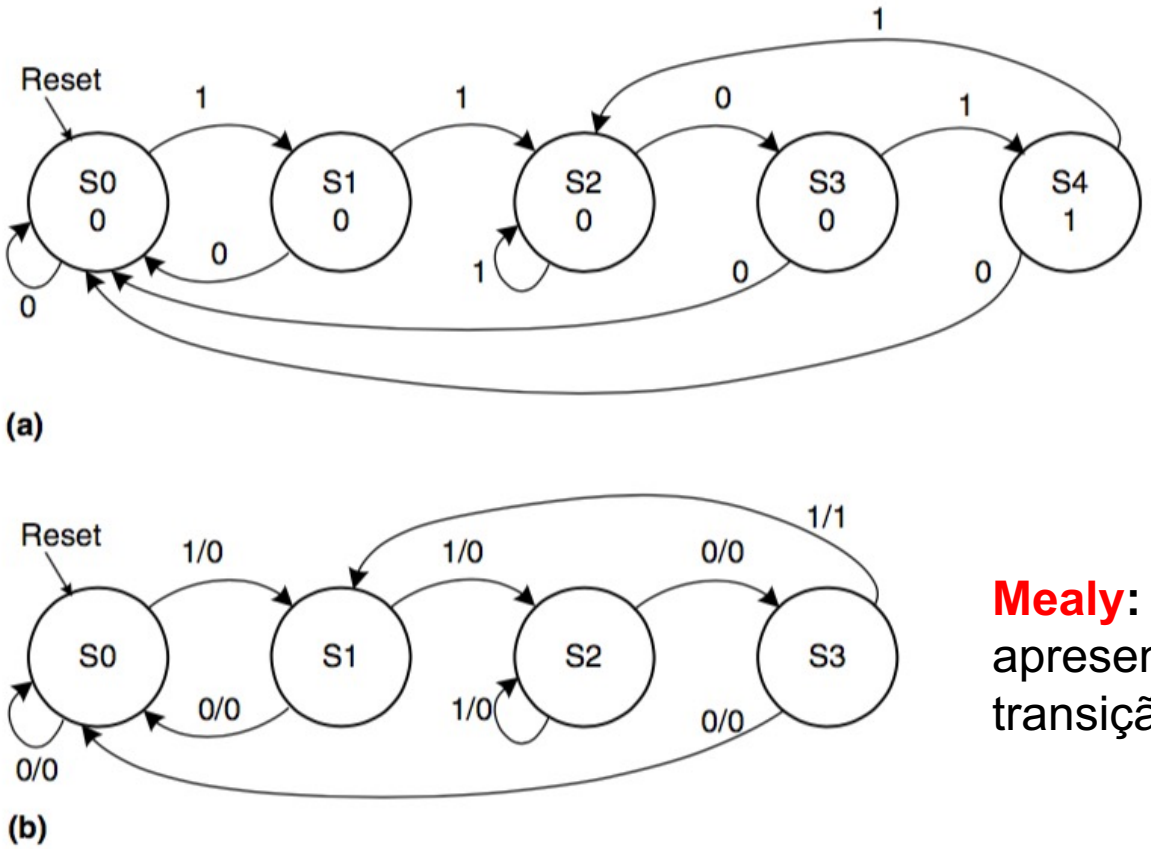
(a) Moore: valor de saída depende somente do estado atual



(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



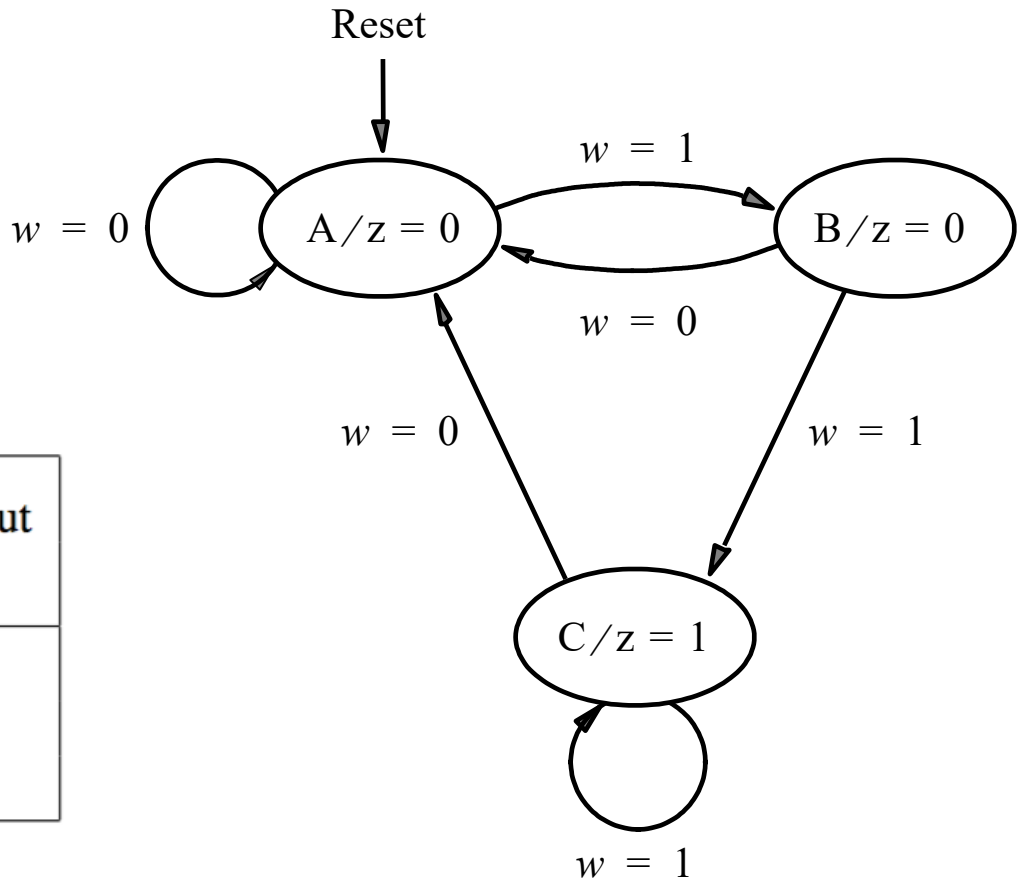
Moore: saída apresentada no estado, e nos arcos de transição apenas as entradas

Mealy: entrada e saída apresentada nos arcos de transição

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

Que tipo de máquina de estados é esta?

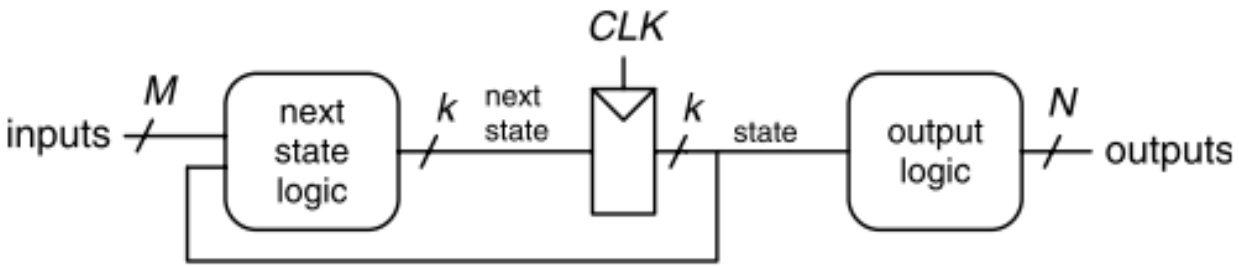
O que faz esta FSM ?



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

FSM Moore

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY moore IS
        PORT (  clock, reset, w     : IN     STD_LOGIC ;
                z                   : OUT    STD_LOGIC ) ;
END moore ;
```



Cria um novo tipo de sinal.

Compilador VHDL automaticamente determina o número de flip-flops necessários.

EA estado atual e PE próximo estado

```
7   ARCHITECTURE a1 OF moore IS
8       TYPE state IS (A, B, C) ;
9       SIGNAL EA, PE : state;
10  BEGIN
11      PROCESS ( Reset, Clock )
12      BEGIN
13        IF Reset = '1' THEN
14              EA <= A ;
15         ELSIF rising_edge(clock) THEN
16              EA <= PE;
17        END IF;
18      END PROCESS;
19      PROCESS(EA, w)
20        CASE EA IS
21        WHEN A =>
22              IF w = '0' THEN
23                    PE <= A ;
24              ELSE  PE <= B ;
25              END IF ;
26        WHEN B =>
27              IF w = '0' THEN
28                    PE <= A ;
29              ELSE  PE <= C ;
30              END IF ;
31        WHEN C =>
32              IF w = '0' THEN
33                    PE <= A ;
34              ELSE  PE <= C ;
35              END IF ;
36        WHEN OTHERS => PE <= A ;
37        END CASE ;
38      END PROCESS;
39
40      z <= '1' WHEN EA = C ELSE '0' ;
41  END a1 ;
```

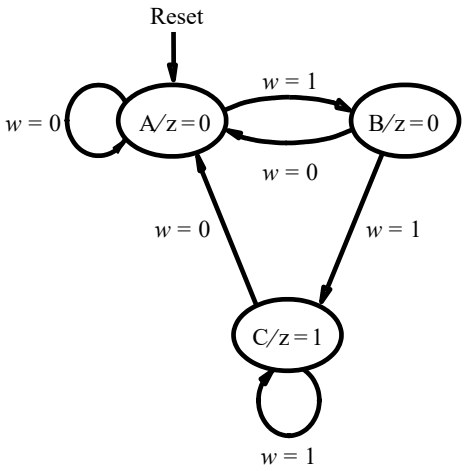
1 - Process describes the FSM as a sequential circuit.

Sensitivity list: entrada e estado atual

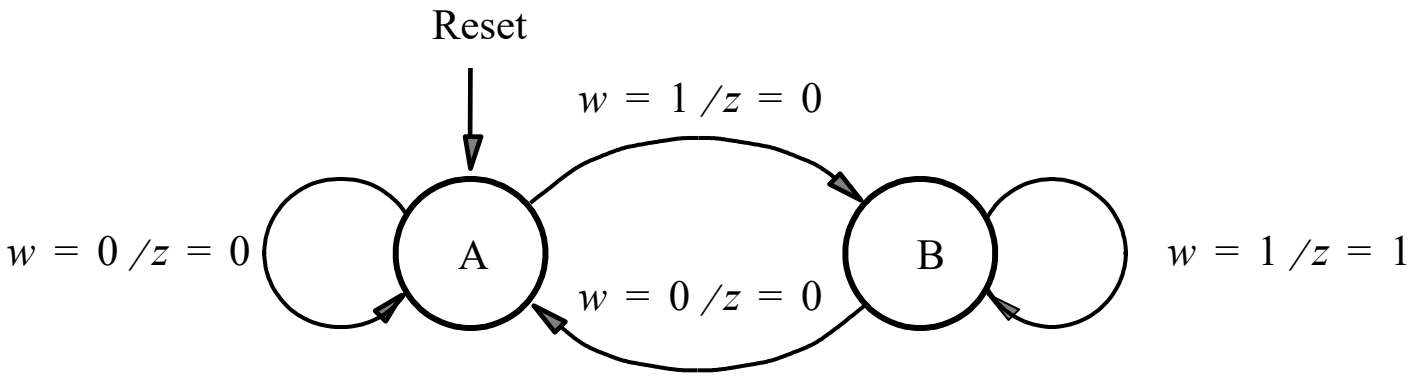
O Case descreve o comportamento da FSM

Cada cláusula When representa um estado da FSM

Saída



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

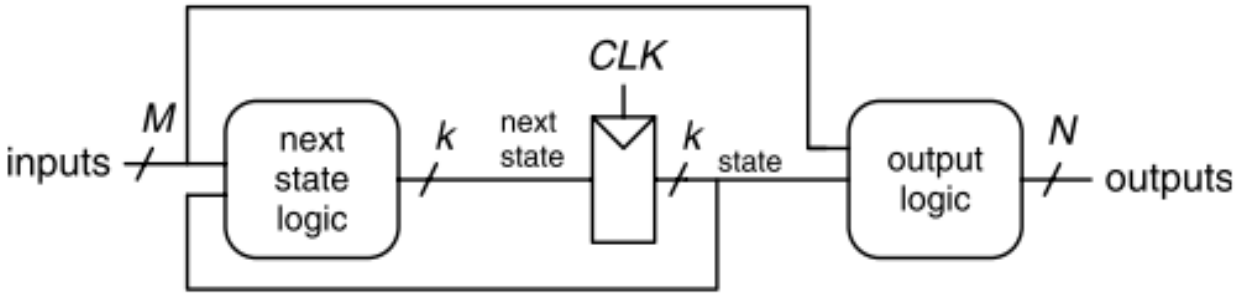


Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

* Fundamentals of Digital Logic with VHDL Design

FSM Mealy

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
        PORT ( clock, reset, w :   IN      STD_LOGIC ;
                    z                  : OUT   STD_LOGIC ) ;
END mealy ;
```

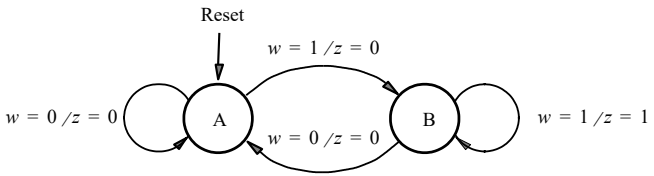


Create a user-defined signal type.

```
ARCHITECTURE a1 OF mealy IS
  TYPE state IS (A, B) ;
  SIGNAL EA, PE : state;
BEGIN
  PROCESS ( Reset, Clock )
  BEGIN
    IF Reset = '1' THEN
        EA <= A ;
    ELSIF rising_edge(clock) THEN
        EA <= PE;
    END IF;
  END PROCESS;

  PROCESS ( EA, w)
  BEGIN

    CASE EA IS
      WHEN A =>
        IF w = '0' THEN PE <= A ;
        ELSE PE <= B ;
        END IF ;
        z <= '0' ;
      WHEN B =>
        IF w = '0' THEN PE <= A ;
        ELSE PE <= B ;
        END IF ;
        z <= w;

      WHEN OTHERS =>
        PE <= A;
        z <= '0' ;
    END CASE ;
  END PROCESS ;

END a1 ;
```

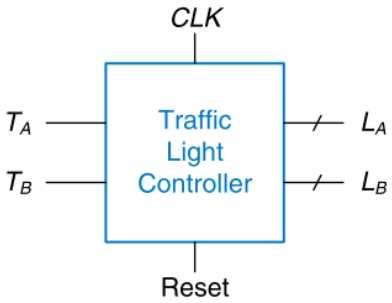
1 - Process describes the FSM as a sequential circuit.

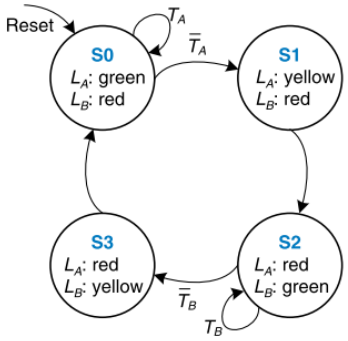
Sensitivity list



This Case describes the behavior of the FSM and the output(s)

Voltando ao exemplo da sinaleira






```
library ieee;
use ieee.std_logic_1164.all;

entity fsm1 is
        port (
                CLK, RST: in std_logic;
                Ta, Tb: in std_logic;
                La, Lb: out std_logic_vector(1 downto 0)
        );
end entity;

architecture a1 of fsm1 is
        type state is (S0, S1, S2, S3);
        signal EA, PE: state;
begin
```

```
begin

    seq: process(clk, rst)
    begin
      if rst= '1' then
          EA <= S0;
      elsif rising_edge(clk) then
          EA <= PE;
      end if;
    end process;

    comb: process(EA, Ta, Tb)
    begin
      case EA is

          when S0 =>  if Ta = '0' then   PE <= S1;
                      else   PE <= S0;
                end if;

          when S1 =>  PE <=S2;

          when S2 => if Tb = '0' then  PE <= S3;
                      else PE <= S2;
                end if;

          when S3 =>  PE <= S0;

          when others => null;
      end case;
    end process;

      La <= "01" when EA = S0 else
          "10" when EA = S1 else
          "00";

      Lb <= "01" when EA = S2 else
          "10" when EA = S3 else
          "00";

end architecture;
```

FSM

saídas

MATERIAL DE APOIO

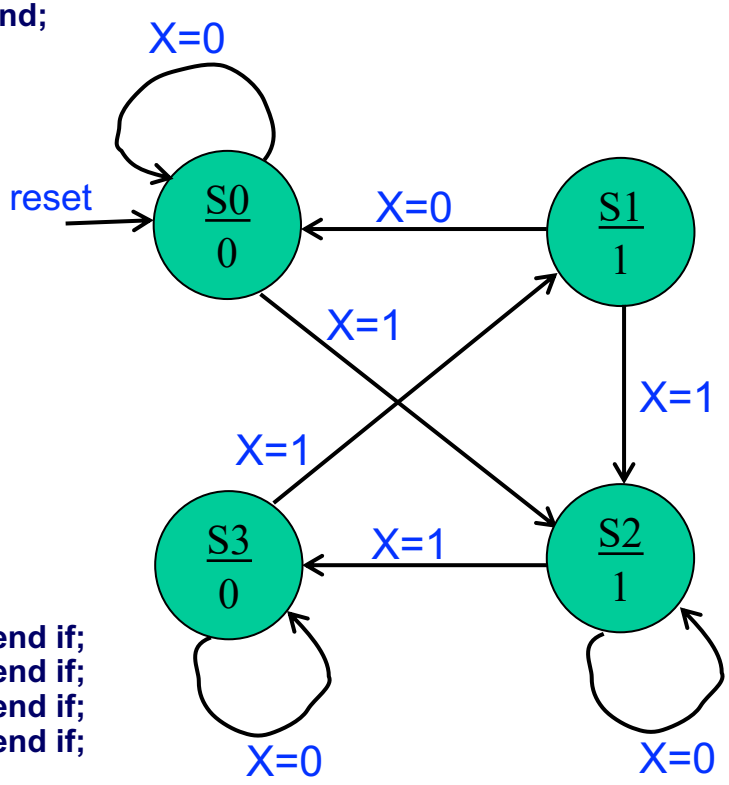
Baixar o arquivo de apoio, o qual contém os exercícios que desenvolveremos com máquinas de estado.

```
material_apoio_FSM
├── FSM0_sinaleira
│   ├── fsm.vhd
│   ├── sim.do
│   ├── solucao_wave_sinaleira.png
│   ├── tb.vhd
│   └── wave.do
├── FSM2_detecta_padrado
│   ├── detecta_padrao.vhd
│   ├── detecta_padrao_tb.vhd
│   ├── sim.do
│   ├── solucao_wave_detacta_padrao.png
│   └── wave.do
├── FSM3_multiplicador_serial
│   ├── multiplicador_serial.vhd
│   ├── multiplicador_serial_tb.vhd
│   ├── sim.do
│   ├── solucao_wave_divisor_multiplicador.png
│   └── wave.do
├── FSM3b_divisor_serial
│   ├── divisor_serial.vhd
│   ├── divisor_serial_tb.vhd
│   ├── sim.do
│   ├── solucao_wave_divisor_serial.png
│   └── wave.do
├── FSM4_arbitro
│   ├── arbitro.vhd
│   ├── arbitro_tb.vhd
│   ├── p_arb.vhd
│   ├── sim.do
│   ├── solucao_wave_arbitro.png
│   └── wave.do
├── FSM5_tx_serial
│   ├── sim.do
│   ├── solucao_wave_tx_serial.png
│   ├── tx_serial.vhd
│   ├── tx_serial_tb.vhd
│   └── wave.do
└── FSM6_exponenciacao
    ├── exponenciacao.vhd
    ├── exponenciacao_tb.vhd
    ├── multiplicador_serial.vhd
    ├── sim.do
    ├── solucao_wave_exponenciacao.png
    └── wave.do
```

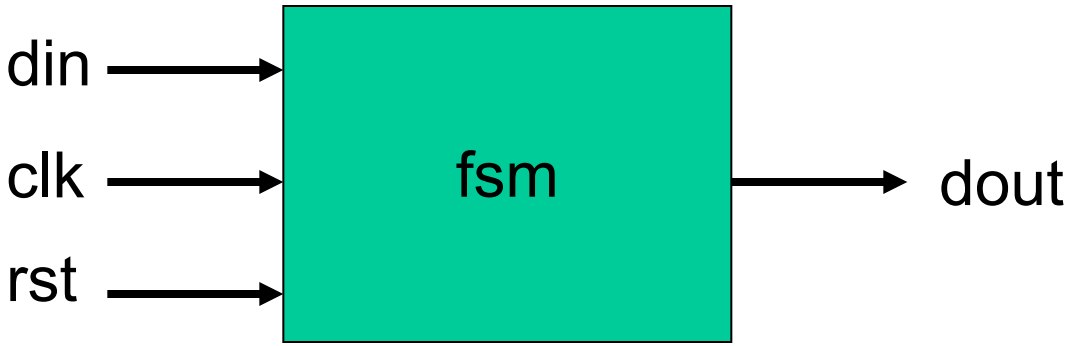
FSM 1 - EXERCÍCIO - MÁQUINA DE ESTADOS

- **Desenhe a FSM equivalente**
- **Diga de que tipo é esta FSM**

```
entity FSM is  port(X, reset, clock : in std_logic; Z: out std_logic);   end;

architecture A of FSM is
     type state is (S0, S1, S2, S3);                -- tipo enumerado
     signal EA, PE : state;
begin
     controle: process(clock, reset)
     begin
          if reset='1' then
               EA <= S0;
          elsif rising_edge(clock) then
               EA <= PE;
          end if;
     end process;

     combinacional: process(EA, X)
     begin
          case EA is
               when S0 =>     if X='0' then PE <=S0; else PE <= S2; end if;
               when S1 =>     if X='0' then PE <=S0; else PE <= S2; end if;
               when S2 =>     if X='0' then PE <=S2; else PE <= S3; end if;
               when S3 =>     if X='0' then PE <=S3; else PE <= S1; end if;
          end case;
     end process;

    Z <= '1' when EA=S2 or EA=S1 else '0';

end A;
```



FSM 2 - Detector da sequência 1101



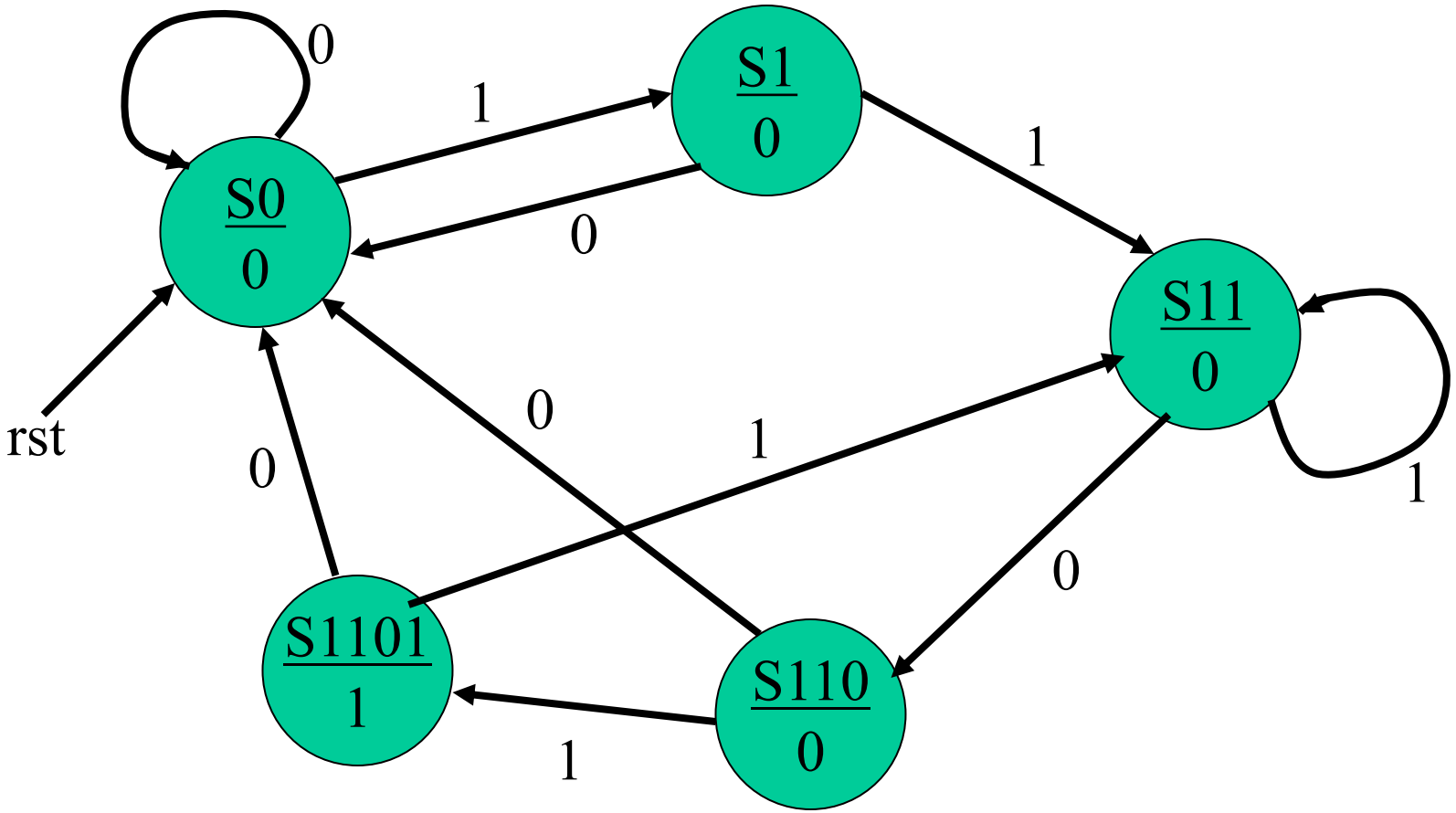
din 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0

dout 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0

Detect input sequence 1101

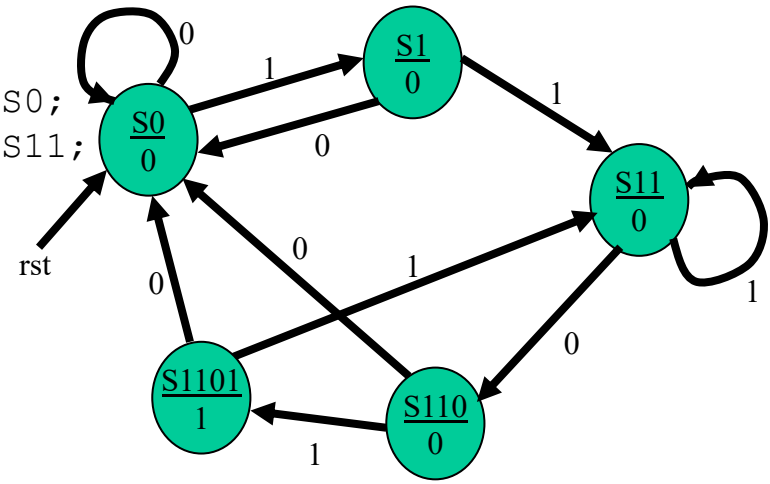
din 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0

dout 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0



```
entity fsm is
  port (clk: in STD_LOGIC;
        rst: in STD_LOGIC;
        din: in STD_LOGIC;
        dout: out STD_LOGIC);
end;

architecture a1 of fsm is

     type state is (S0, S1, S11, S110, S1101);
     signal EA, PE: state;
begin



seq: process(clk, rst)
     begin
       if rst= '1' then
         EA <= S0;
       elsif rising_edge(clk) then
         EA <= PE;
       end if;
 end process;
```

```
comb: process(EA, din)
    begin
       case EA is

          when S0 =>  if din = '1' then   PE <= S1;
                                   else   PE <= S0;
                      end if;

          when S1 =>  if din = '1' then  PE <= S11;
                                   else  PE <= S0;
                       end if;

          when S11 => if din = '0' then  PE <= S110;
                                   else  PE <= S11;
                      end if;

          when S110 => if din = '1' then PE <= S1101;
                                    else PE <= S0;
                       end if;

          when S1101 => if din = '0' then PE <= S0;
                                     else PE <= S11;
                        end if;

          when others => null;
       end case;
    end process;

    dout <= '1' when EA = S1101  else '0';

end a1;
```

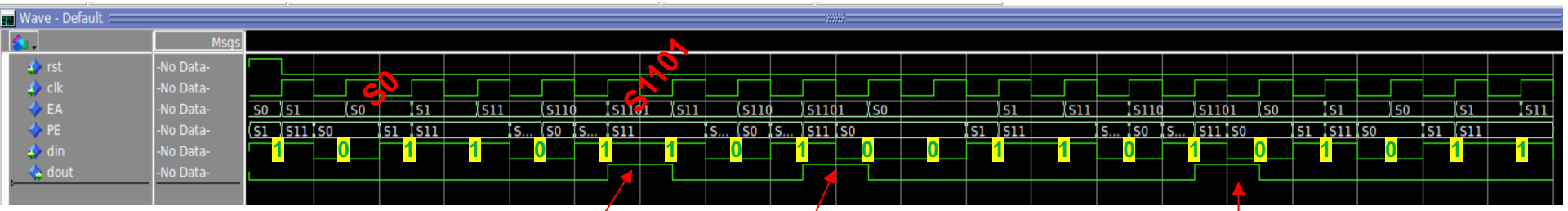


Exercício - simular

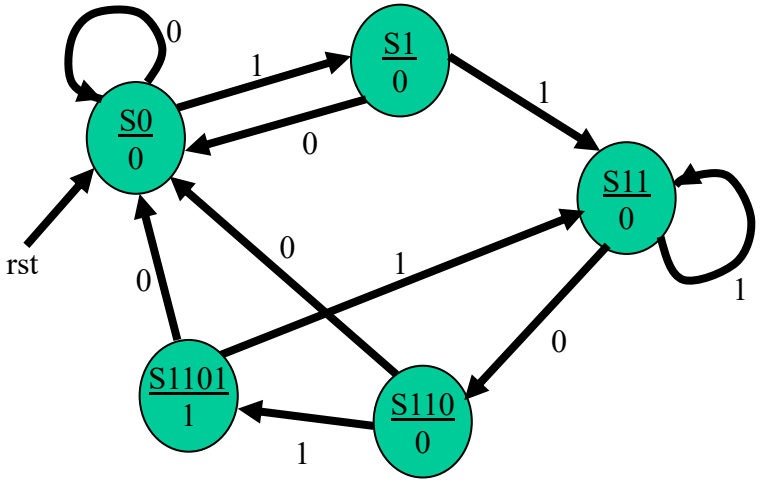
Moodle→material de apoio → VHDL do detector de sequencia

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tb is
end tb;

architecture arch of tb is
    signal reset: std_logic;
    signal clk : std_logic := '0' ;
    signal din, dout: STD_LOGIC;

    type dados is array(natural range <>) of std_logic;

    constant padrao_de_teste : dados := ('1', '0', '1', '1', '0', '1', '1', '0', '1', '0', '0', '1', '1', '0', '1', '0');

begin

    reset <= '1', '0' after 5 ns;

    test: process
    begin
      for i in 0 to padrao_de_teste'high loop
            din <= padrao_de_teste(i);
            wait for 10 ns;
       end loop;
    end process;

    clk <= not clk after 5 ns;

    seq: entity work.fsm  port map(clk => clk, rst => reset,  din => din, dout=>dout);

end arch;
```

Detect input sequence 1101



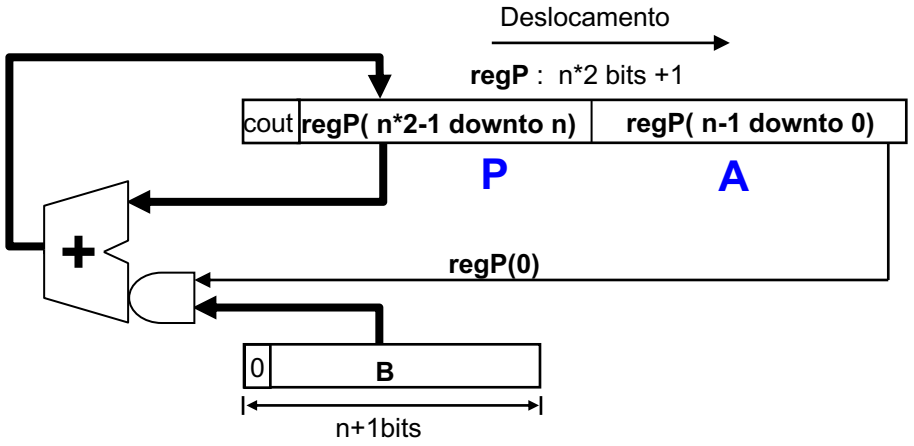
din := ('1', '0', '1', '1', '0', '1', '1', '0', '1', '0', '0', '1', '1', '0', '1', '0');



FSM 3 – Multiplicador Serial

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

135 ➔ 100 00111




```
  00101
* 11011
-------
  00101
```

1º produto parcial

```
  00101
* 11011
-------
  00101
 00101
-------
 001111
```

1º bit menos significativo definido (LSB)

```
  00101
* 11011
-------
  00101
 00101
00000
-------
0011111
```

2º LSB

```
  00101
* 11011
-------
  00101
 00101
00000
-------
00110111
```

3º LSB

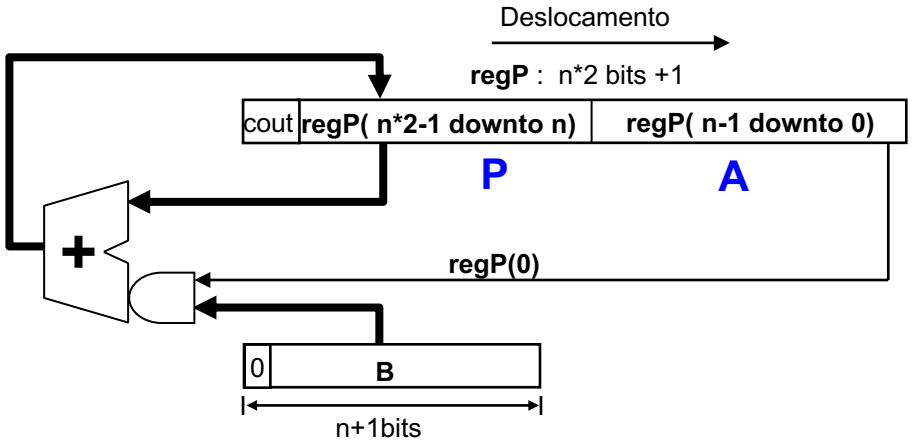
```
  00101
* 10111
-------
  00101
 00101
00000
-------
01000111
```

Produto Final

FSM 3 – Multiplicador Serial

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

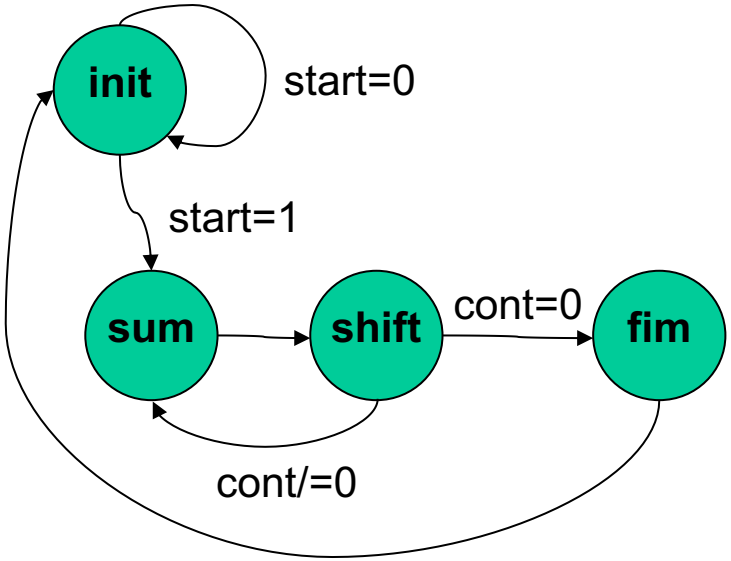
135 ➔ 100 00111



	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

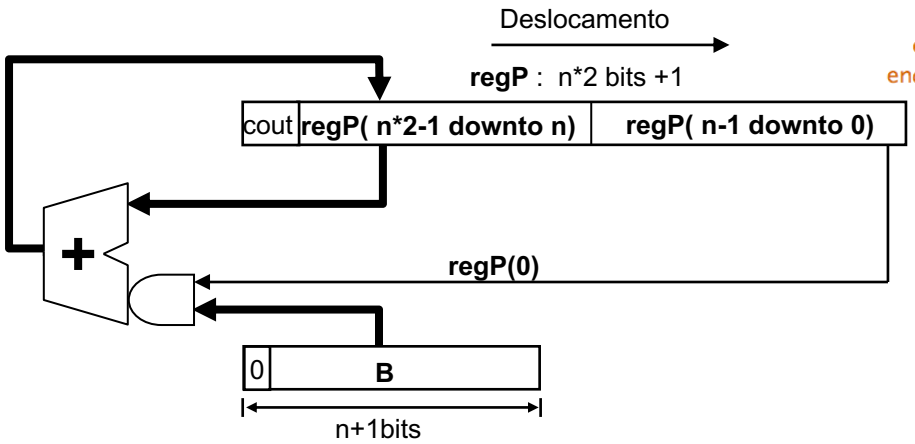
FSM 3 – Multiplicador Serial

FSM: controle



Bloco de Dados

```
-- bloco de dados
process (reset, clock)
begin
  if reset='1' then
     regP <= (others=>'0');
     produto <= (others=>'0');

  elsif rising_edge(clock) then
    case EA is
      when init =>  regP(64 downto 32) <= (others=>'0');
                    regP(31 downto 0)  <= A;
                    cont <= (others=>'0');
                    end_mul <='0';

      when sum =>   cont <= cont-1;
                    if regP(0)='1' then
                        regP(64 downto 32) <= regP(64 downto 32) + ('0' & B);
                    end if;

      when shift => if cont=0 then
                          end_mul <='1';
                    end if;
                    regP <= '0' & regP(64 downto 1);

      when fim =>   produto <= regP(63 downto 0);
     end case;
  end if;
end process;
```



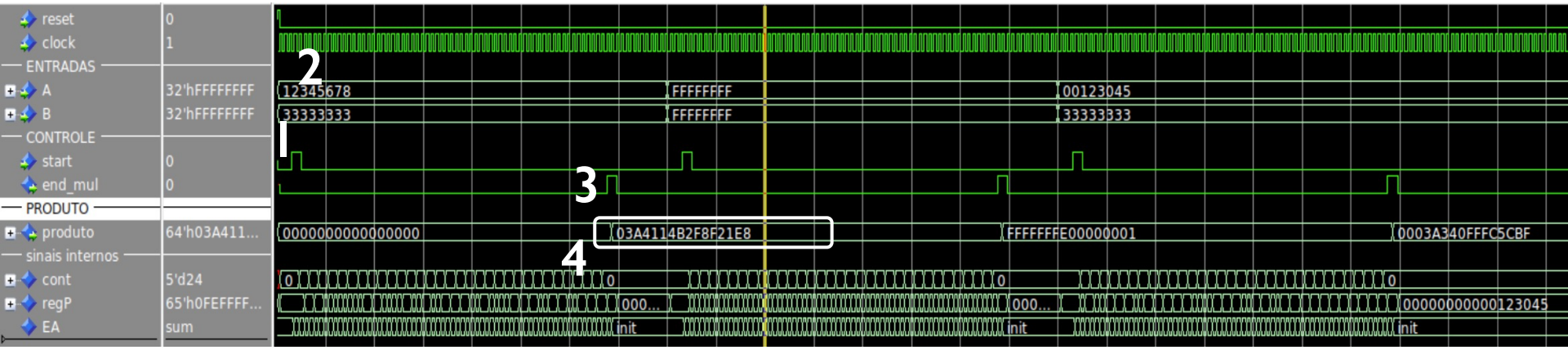
FSM 3 – Multiplicador Serial

Implemente o multiplicador serial utlizando este *test bench*

```
library ieee;
use ieee.std_logic_1164.all;

entity tb is
end tb;

architecture a1 of tb is

  signal op1, op2: std_logic_vector(31 downto 0);
  signal produto : std_logic_vector(63 downto 0);
  signal reset, start, end_mul  : std_logic;
  signal clock : std_logic := '0';

 begin

   mu: entity work.mult_serial
         port map( clock=> clock, reset=> reset, start=>start,
                  A => op1, B=> op2,  end_mul => end_mul,    produto=> produto, );

   reset <= '1', '0' after 5 ns;
   clock <= not clock after 5 ns;

   op1 <=  x"12345678",   x"FFFFFFFF" after 800 ns,  x"00123045" after 1600 ns;

   op2 <=  x"33333333",   x"FFFFFFFF" after 800 ns,  x"33333333" after 1600 ns;

   ---  respostas esperadas:  0x3A4114B2F8F21E8,  0xfffffffe00000001, 0x3A340FFFC5CBF

    start <= '0' , '1' after   30 ns, '0' after   50 ns,  '1' after  830 ns, '0' after  850 ns,
               '1' after 1630 ns, '0' after 1650 ns;

end architecture;
```

FSM 3 – Multiplicador Serial

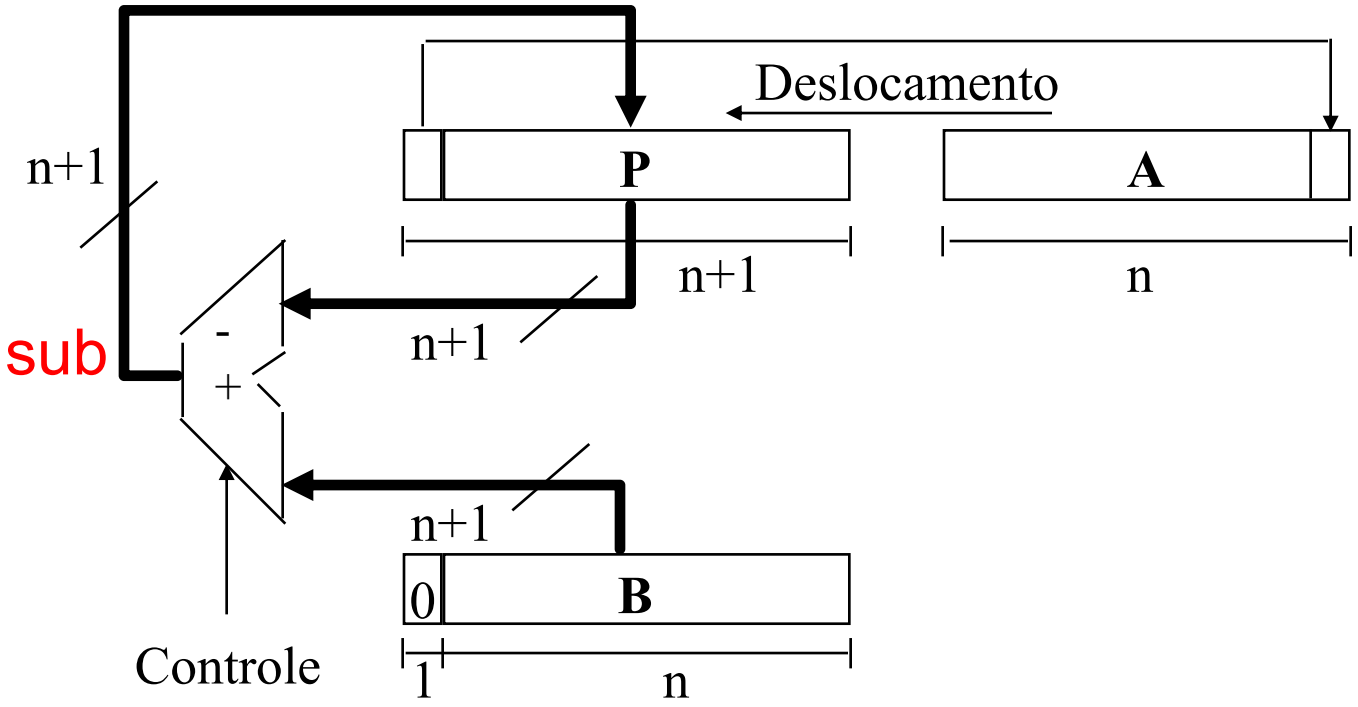
```
op1 <=  x"12345678",   x"FFFFFFFF" after 800 ns,  x"00123045" after 1600 ns;

op2 <=  x"33333333",   x"FFFFFFFF" after 800 ns,  x"33333333" after 1600 ns;

---  respostas esperadas:  0x3A4114B2F8F21E8,  0xfffffffe00000001, 0x3A340FFFC5CBF

start <= '0' , '1' after   30 ns, '0' after   50 ns,  '1' after  830 ns, '0' after  850 ns,
         '1' after 1630 ns, '0' after 1650 ns;

end architecture;
```



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 ← P-B;
  2) if (sub<0), A_0=0 else { A_0 =1; P ←sub}

Divisão A/B

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

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

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

Divisão A/B

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

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

	P (conterá o resto)						A (conterá a divisão)				
passo	0	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; sub ← P-B;
2) if (sub<0), A0=0 else { A0 =1; P ←sub}

	P (conterá o resto)						A (conterá a divisão)				
passo	0	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											
5											

00110 - 00101 = 001

Divisão A/B

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

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

		P (conterá o resto)					A (conterá a divisão)				
passo	0	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

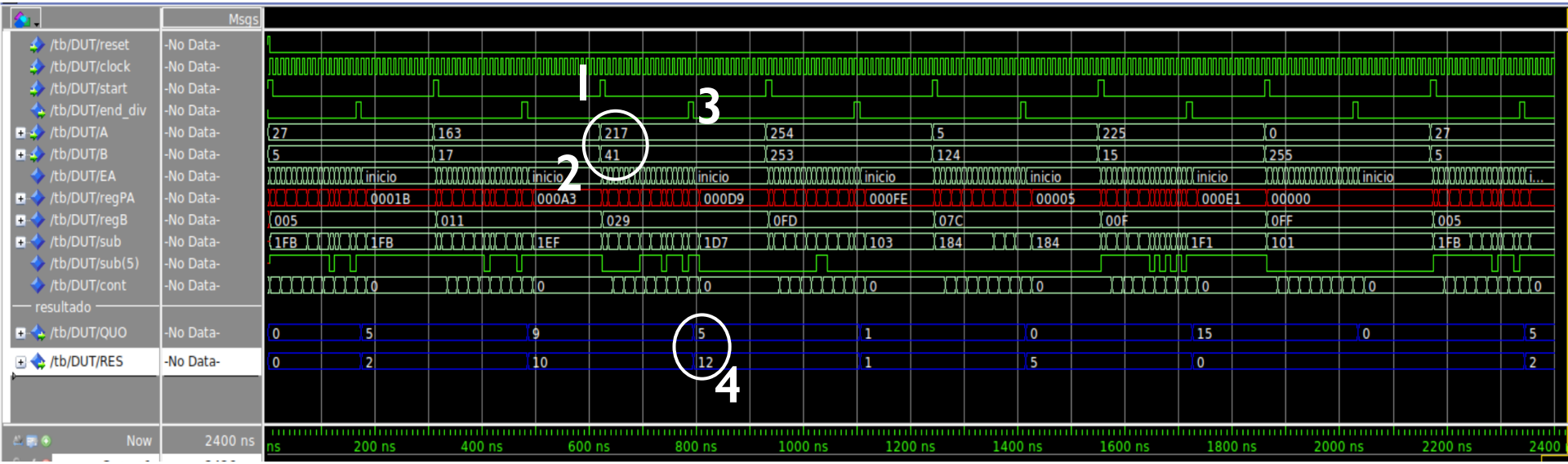
A = 11011 (27)
B = 00101 (5)

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

		P (conterá o resto)					A (conterá a divisão)				
passo	0	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**

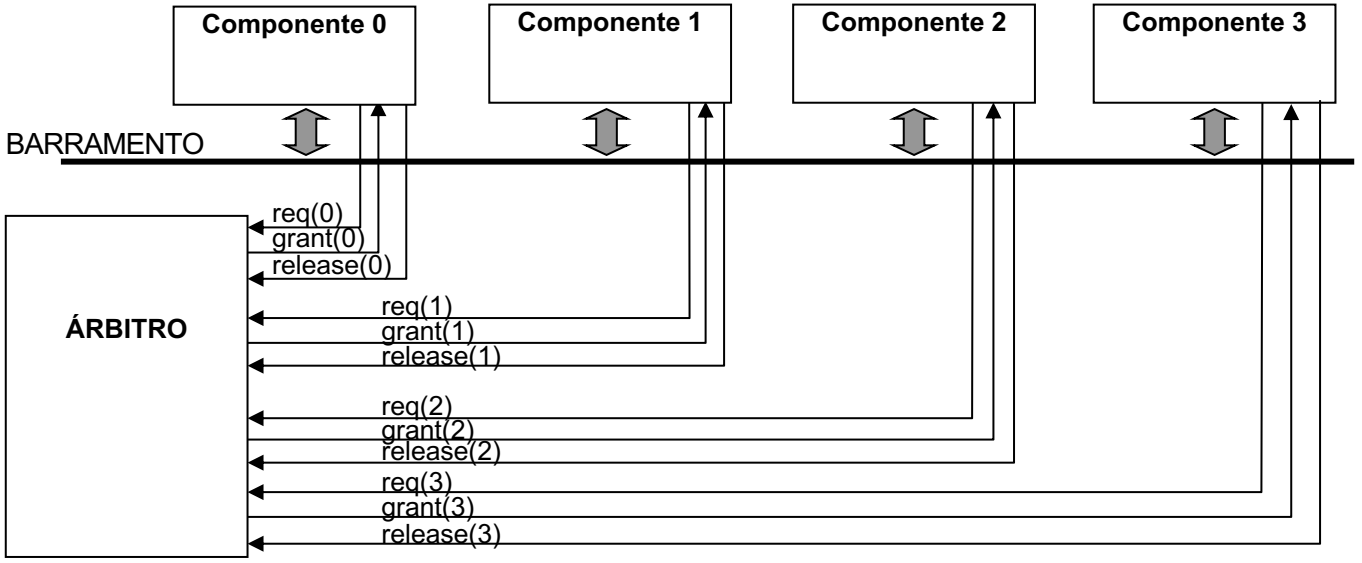
Divisão A/B



1. Inicio da divisão – *start=1*
2. Valores: 217 / 41
3. Final da divisão
4. Resultado: Quociente=**5** e resto=**12**

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 $\boldsymbol{req_i}$. Se o barramento estiver livre, o árbitro ativa o sinal $\boldsymbol{grant_i}$ 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 $\boldsymbol{release_i}$ por um ciclo de clock informando ao árbitro que o barramento está livre.



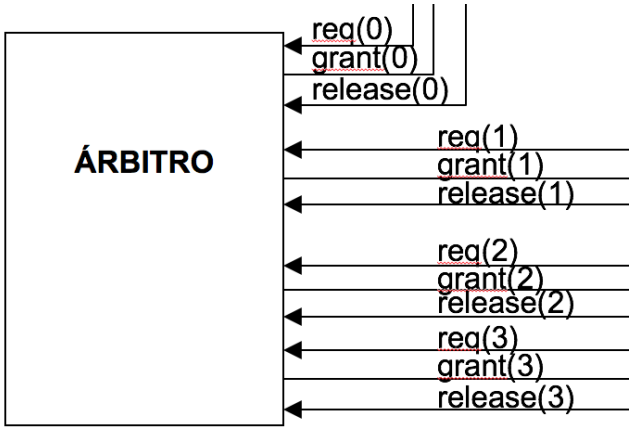
BARRAMENTO contém dados, endereço, clock, reset

FSM 4 – Árbitro de Barramento

- **Desenhar uma máquina de estados para o árbitro**
- **Implementar o árbitro em VHDL**

Operação do árbitro.

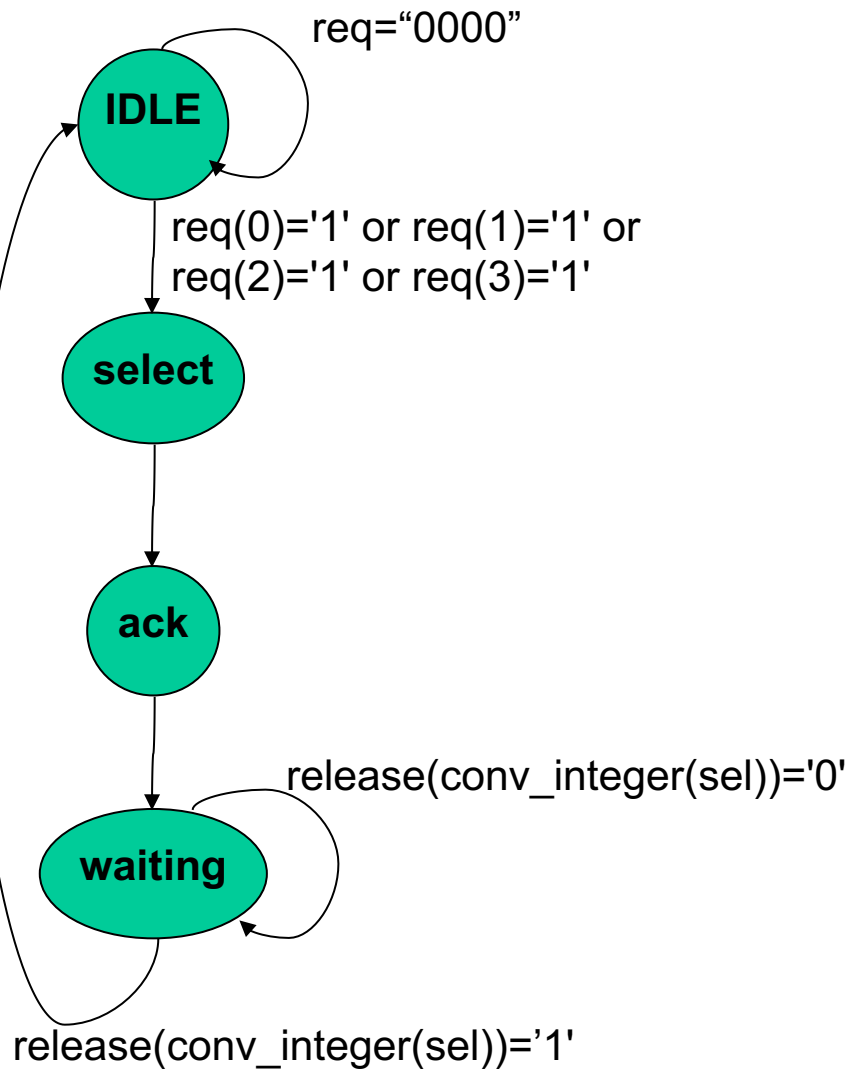
- ficar sequencialmente lendo os sinais de solicitação (***req(0)*** a ***req(3)***).
- quando um dos sinais req_i estiver ativo, o sinal $grant_i$ sobe por um ciclo de clock, aguardando-se que o sinal $release_i$ suba.
- na subida do sinal $release_i$ reinicia-se o **processo de leitura sequencial,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




```
library IEEE;
use IEEE.Std_Logic_1164.all;

package p_arb is
        type control is array(0 to 3) of std_logic;
end p_arb;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;
use work.p_arb.all;

entity arbitro is
        port(   clock, reset : in std_logic;
                req, release: in control;
                grant: out control
            );
end arbitro;

architecture ab of  arbitro is
```

Sugestão




```
architecture ab of  arbitro is
  type ESTADO is (IDLE, SSELECT, ACK, WAITING);
  signal EA, PE: ESTADO;
  signal sel: std_logic_vector(1 downto 0);
begin

  process(reset, clock)
  begin
    ...
      EA <= PE;
    ..
   end process;


  process(EA, req, release)
  begin
     case EA is
       when IDLE  => ....
       when SSELECT => ...
       ...
  end process;

  grant(0) <= '1' when  EA=____ and sel="00" else '0';
  grant(1) <= .....
  grant(2) <= .....
  grant(3) <= .....


  process(....)
   begin
    if reset='1' then
      sel <= "00";
    elsif rising_edge(.....) then

        if EA=SSELECT  then
          if req(CONV_INTEGER(sel+1))='1' then
             sel <= sel + 1;
           ......
        end if;
      end if;
   end process;
end ab;
```

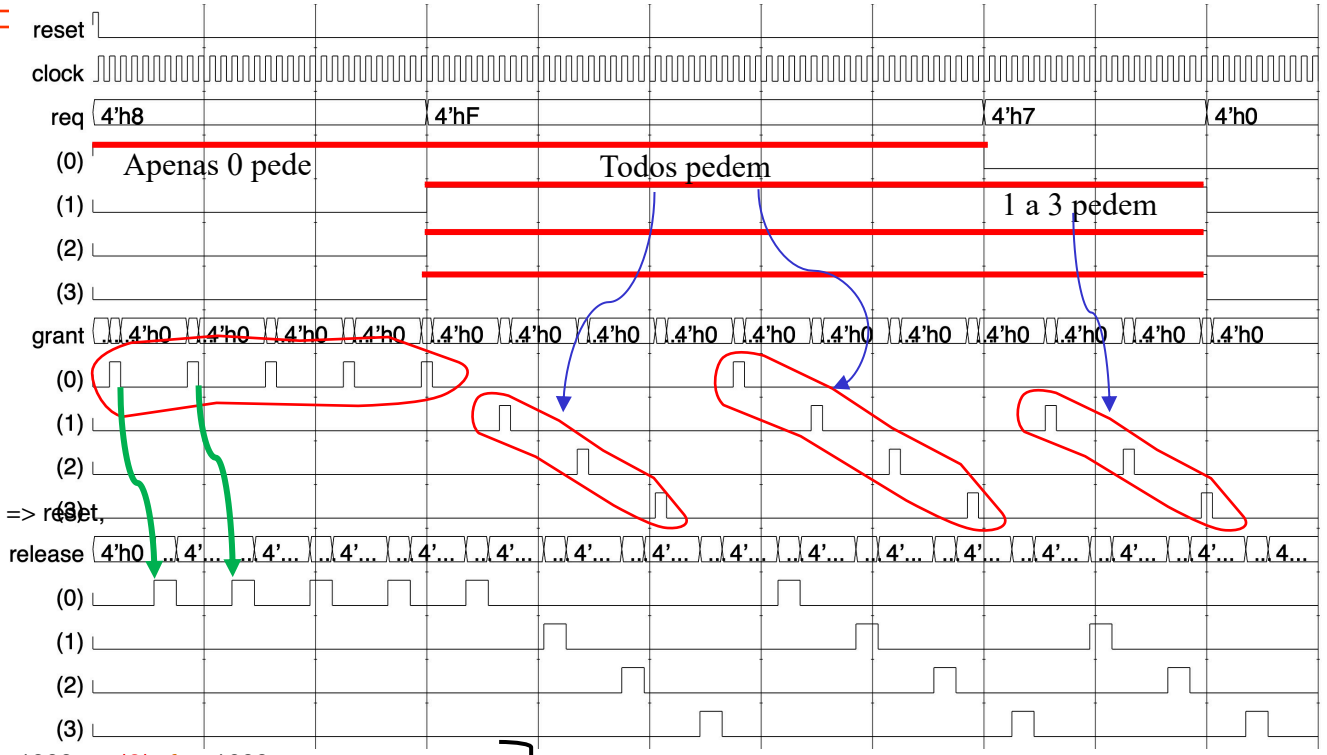
2 process para máquina de estados

Bloco de Dados

Registrador *sel* diz quem está sendo atendido

FSM 4 – Test bench para o árbitro

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.p_arb.all;
4
5  entity tb is
6  end tb;
7
8  architecture tb of tb is
9
10   signal clock : std_logic := '0';
11   signal reset : std_logic;
12
13   signal req, grant, release  : control;
14
15 begin
16      ab : entity work.arbitro
17         port map ( clock   => clock, reset   => reset,
18                    req     => req,
19                    release => release,
20                    grant   => grant
21                  );
22
23      reset <= '1', '0' after 5 ns;
24      clock <= not clock after 5 ns;
25
26      req(0) <= '1' ,  '0' after 800 ns, '1' after 1300 ns, '0' after 1600 ns;
27      req(1) <= '0' ,  '1' after 300 ns, '0' after 1000 ns;
28      req(2) <= '0' ,  '1' after 300 ns, '0' after 1000 ns, '1' after 1300 ns, '0' after 1600 ns;
29      req(3) <= '0' ,  '1' after 300 ns, '0' after 1000 ns;
30
31      modulos: for i in 0 to 3 generate
32           process
33           begin
34                release(i)<='0';
35                wait until grant(i)='1';
36                wait for 40 ns;
37                release(i) <='1';
38                wait for 20 ns;
39           end process;
40      end generate;
41
42 end tb;
```



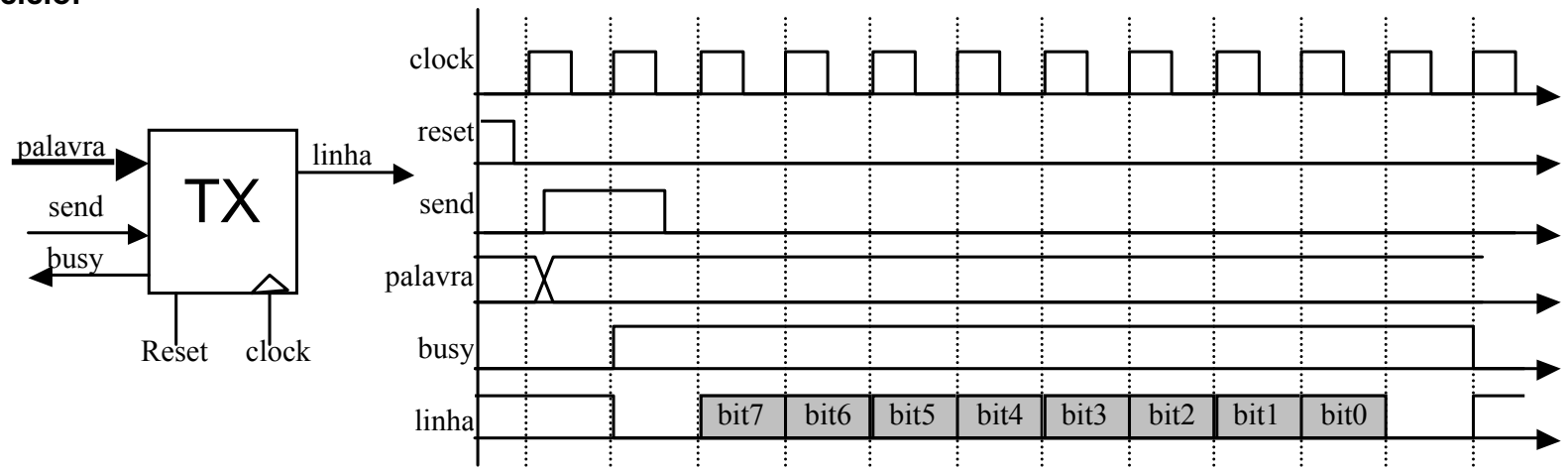
Geração dos *requests*

Gera o *release* 40 ns depois do *grant*

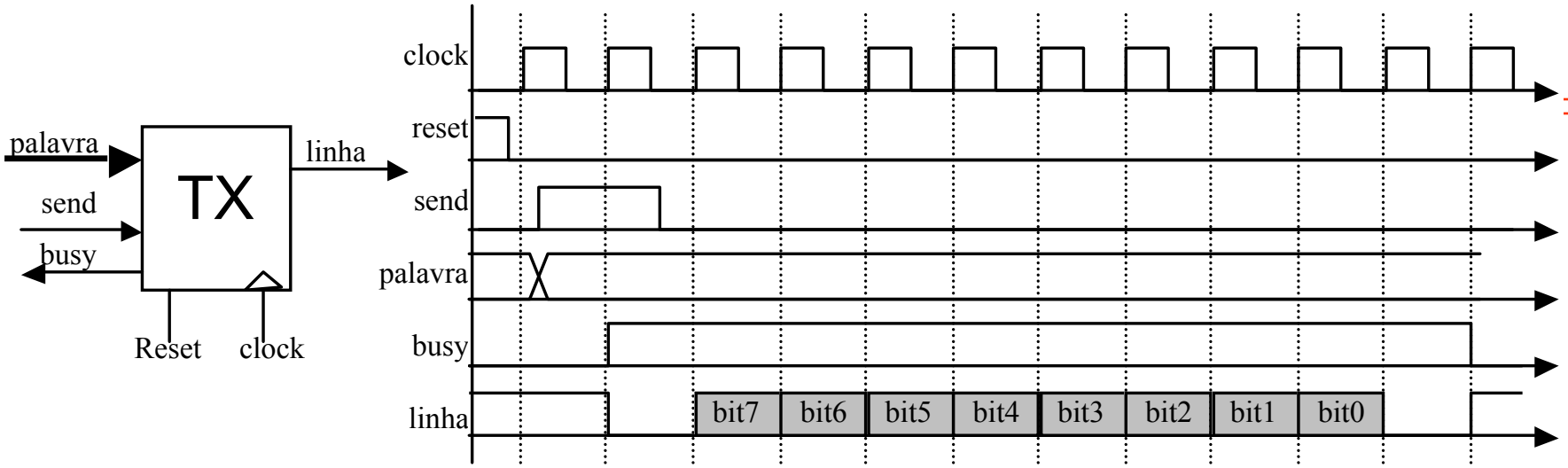
FSM 5 - Transmissão de Dados Serial

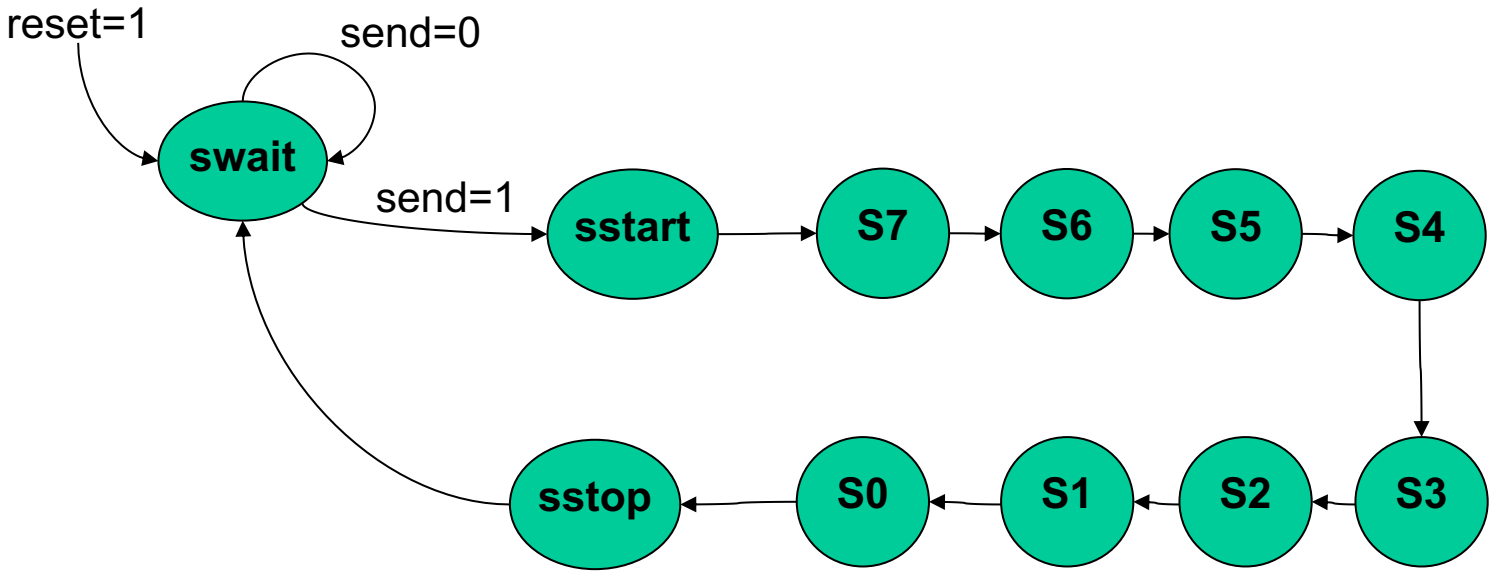
Considere o circuito abaixo – TX, o qual implementa uma transmissão serial de dados. A "linha" de dados (saída do circuito) está por default em '1', indicando que não há transmissão de dados. O protocolo de transmissão é o seguinte:

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.**

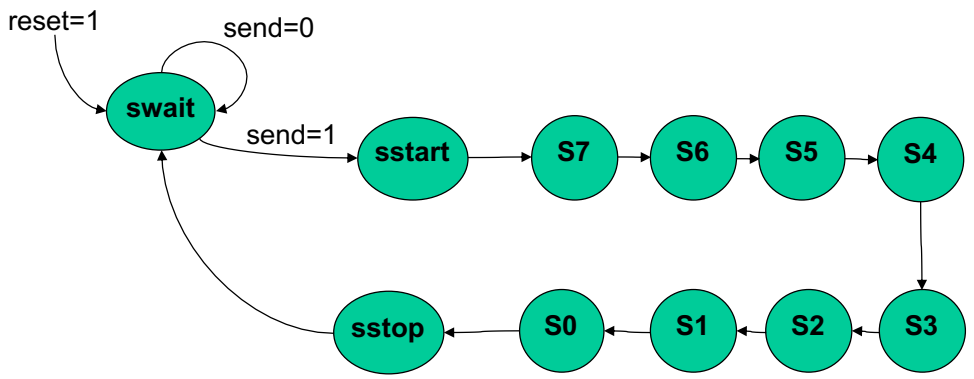


DICA: a máquina de estados tem 11 estados. A controle da saída "linha" pode ficar dentro do processo combinacional da máquina de estados. O sinal de "busy" pode ser implementado como uma atribuição concorrente fora dos processes.


palavra
send
busy
TX
linha
Reset
clock
clock
reset
send
palavra
busy
linha
bit7
bit6
bit5
bit4
bit3
bit2
bit1
bit0


reset=1
send=0
swait
send=1
sstart
S7
S6
S5
S4
S3
S2
S1
S0
sstop

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ex5 is
   port ( clock,reset,send: in _____;
          palavra: in _____;
          busy, linha: out _____
        );
end ex5;

architecture transmissor of ex5 is
   type STATES is (_____);
   signal EA, PE : STATES;
begin
  process(reset,clock)
       begin
            if reset='1' then
                EA <= _____;
            elsif rising_edge(clock) then
                EA <= PE;
            end  if;
       end process;

       process(EA, send)
       begin
            case EA is
                 -----
                 ------
                 -----

            end case;
       end process;

       busy <= _____;

       linha <=  _____;

end transmissor;
```

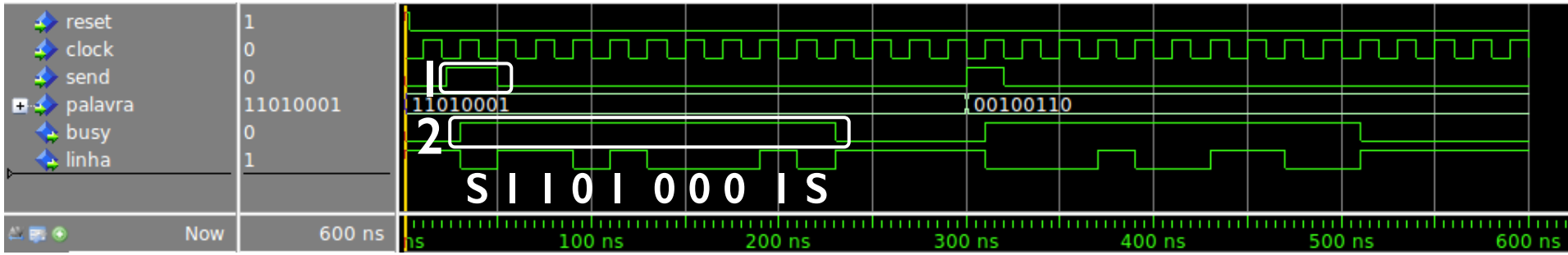


Transmissão de Dados Serial

Para a validação usar o test bench abaixo:

```
library ieee;
use ieee.std_logic_1164.all;

entity transmissor_tb is
end transmissor_tb;

architecture TB_ARCHITECTURE of transmissor_tb is
  signal busy, linha,reset, send: std_logic;
  signal palavra : std_logic_vector(7 downto 0);
  signal clock : std_logic := '0';

begin
          UUT : entity work.transmissor
                    port map (clock => clock, reset => reset, send => send,
                              palavra => palavra, busy => busy, linha => linha );
          clock <=  not clock after 10 ns;

          reset <= '1', '0' after 3 ns;

          send <= '0', '1' after 23 ns, '0' after 50 ns, '1' after 300 ns, '0' after 320 ns;

          palavra <= "11010001", "00100110" after 300ns;

end TB_ARCHITECTURE;
```

Transmissão de Dados Serial



1. Indicação de *send* com palara 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}

FSM 7 - Exponenciação

Você é responsável por desenvolver um módulo que calcula a exponenciação x^y. Assuma que exista um módulo de multiplicação que possui a seguinte interface:

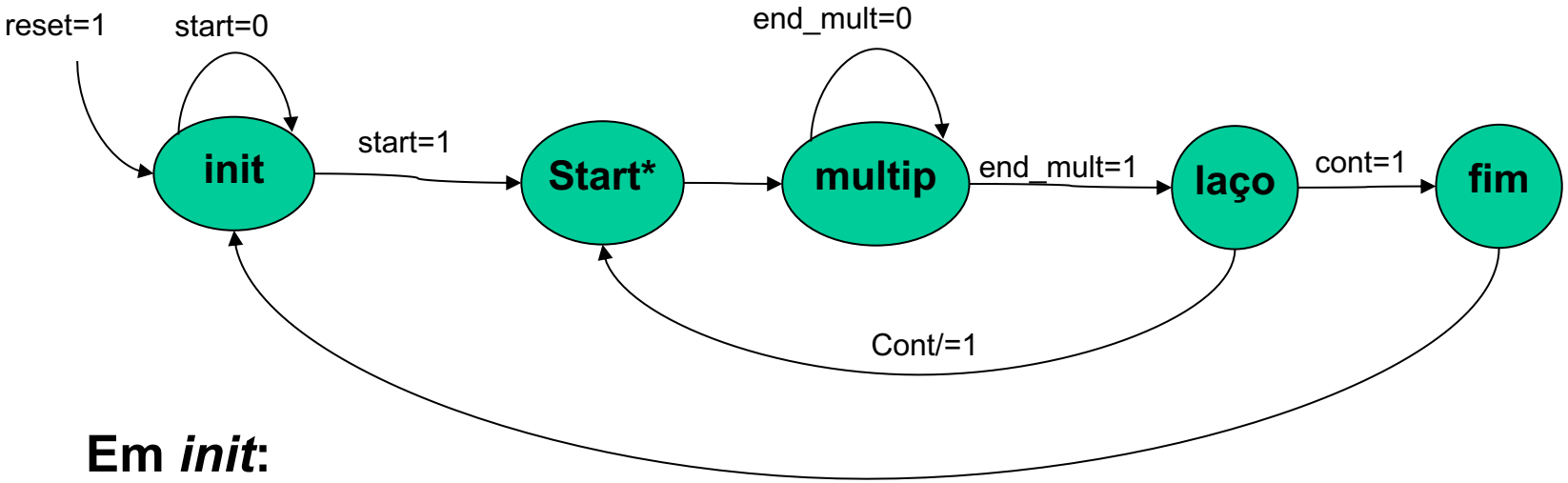
```
entity mult_serial is
        port( clock, reset : in std_logic;
                start: in std_logic;
                A: in std_logic_vector(31 downto 0);
                B: in std_logic_vector(31 downto 0);
               end : out std_logic;
               resultado: out std_logic_vector(63 downto 0)
             );
end mult_serial;
```

Este é o multiplicador serial visto anteriormente. O início da multiplicação é dado por *start=1* e o final da multiplicação é sinalizado por *end=1*.

O protocolo do módulo de exponenciação (*mod_exp*) é o mesmo do multiplicador. Quando o sinal de *start* for '1', os valores *X* (8 bits) e *Y* (4 bits) são capturados em registradores internos, e inicia-se o processo de exponenciação por multiplicações sucessivas. O processo de exponenciação termina quando *end=1*. Saída de 32 bits.

A. Defina em VHDL a entidade do ***mod_exp***.
B. Desenhe a máquina de estados (FSM) do ***mod_exp***, atribuindo valores aos sinais de controle e definindo as transições entre cada estado.
C. Descreva em VHDL apenas a máquina de estados, explicando os sinais utilizados para as transições entres os estados.

FSM 7 - Exponenciação



Em *init*:
cont ← Y
op1 ←X
op2 ←1

Em start*:
start_mult ← 1

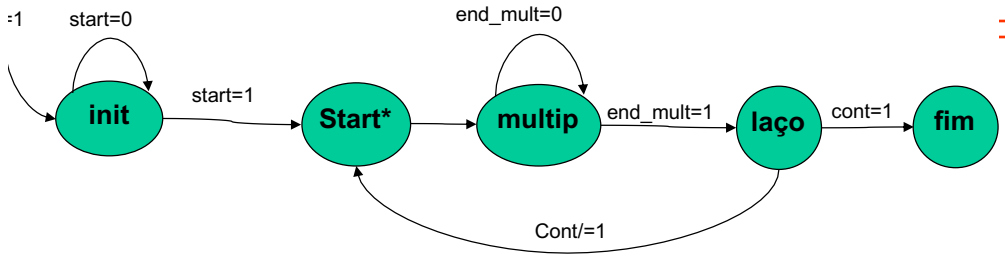
Em *laço*:
op2 ← produto
cont --

Em fim
end_exp← 1
resultado← produto

FSM 7 - Exponenciação

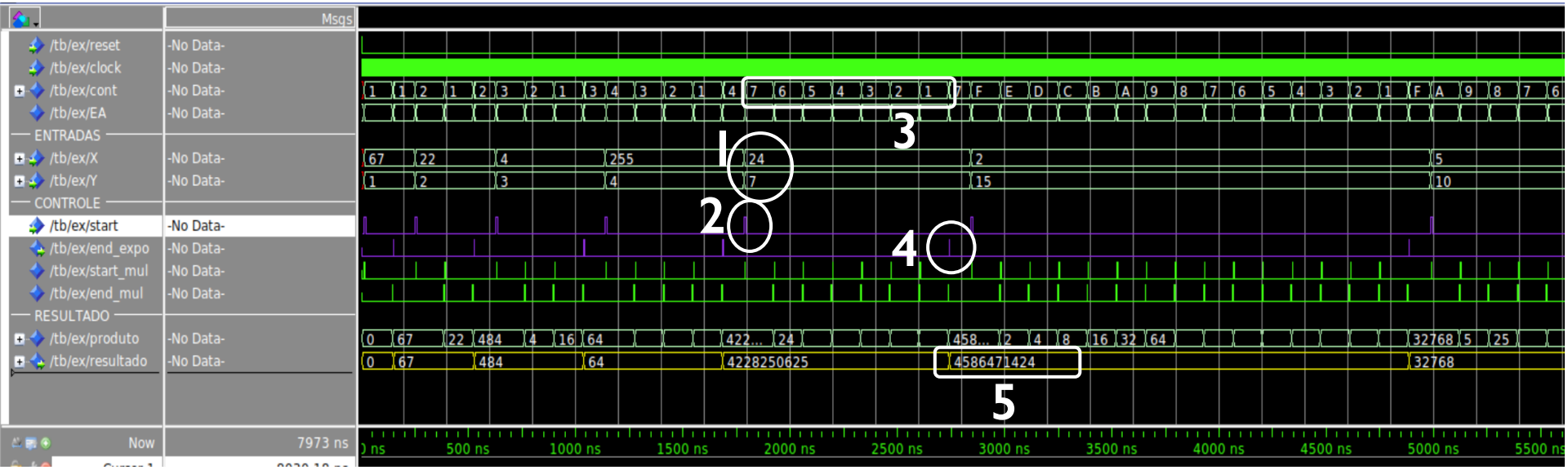
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity exponenciador is
        port( clock, reset : in std_logic;
              start : in std_logic;
              X : in std_logic_vector(7 downto 0);
              Y : in std_logic_vector(3 downto 0);
              end_expo : out std_logic;
              resultado : out std_logic_vector(63 downto 0)
            );
end exponenciador;

architecture exponenciador of exponenciador is
    type states is (init, start_multip, multip, laco, fim);
    signal EA, PE : states;
    signal end_mul, start_mul: std_logic;
    signal cont : std_logic_vector(3 downto 0);
    signal produto, op2 : std_logic_vector(63 downto 0);
    signal op1: std_logic_vector(31 downto 0);
begin
```

Completar aqui com a FSM




```
      mu: entity work.mult_serial
            port map( clock=> clock, reset=> reset, start=>start_mul,
                      A => op1, B=> op2(31 downto 0),
                      end_mul => end_mul,
                      produto=> produto
                    );

      -- bloco de dados
      process (reset, clock)
      begin
          if reset='1' then
              resultado <= (others=>'0');
          elsif rising_edge(clock) then

                start_mul <= '0';  -- default value
                end_expo <= '0';

                case EA is

                      when init =>   -- incializa os registradores
                                     cont <= Y;
                                     op1(7 downto 0)  <=  X;
                                     op1(31 downto 8)  <=  (others=>'0');
                                     op2 <=  (0=>'1', others=>'0');    --------- valor 1
                      when start_multip =>   start_mul <= '1';


                      when laco =>  -- armazena o resultado da multiplicacao e inicia nova iteracao
                                    op2 <= produto;
                                    cont <= cont -1;

                      when fim  =>  end_expo <= '1';
                                    resultado <= produto;

                      when others => null;

                end case;
          end if;
      end process;

end exponenciador;
```

FSM 7 - Exponenciação



1. 24 ^ 7
2. Controle de início da exponenciação
3. Contador que indica o número de multiplicações realizadas
4. Final da exponenciação
5. Resultado: 24 ^7 = 4.586.471.424