2.2.3 Multiplexador

A forma mais básica de criação de códigos concorrentes é a utilização de operadores (AND, OR, +, -, *, sll, sra, dentre outros). Estes operadores podem ser utilizados para implementar diferentes combinações de circuitos. No entanto, como estes operadores mais tarde (fase de síntese) tornam-se aparentes, os circuitos complexos normalmente são mais facilmente descritos quando utilizam declarações seqüenciais, mesmo que o circuito não contenha lógica seqüencial. No exemplo que segue, um projeto utilizando apenas operadores lógicos é apresentado.

Um multiplexador de duas entradas para uma saída (`mux_2x1`) está apresentado na Figura 2.13. A saída `y` deve ser igual a uma das entradas (`a, b`) selecionadas pela entrada de seleção `sel`.

Entradas	Saídas
sel	y
0	a
1	b

$$y = \overline{sel} \cdot a + sel \cdot b$$

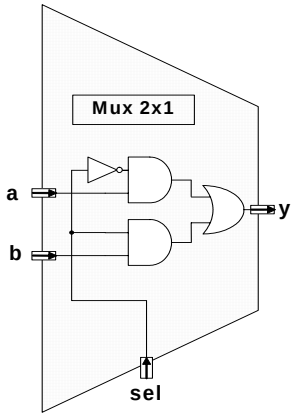


Figura 2.13 – Multiplexador 2x1, tabela verdade, equação booleana e bloco diagrama.

A implementação do multiplexador 2x1 utiliza apenas operadores lógicos, conforme código VHDL transcrito como segue:

```
-----------------------------------------------
--  Circuito: multiplexador 2x1:(mux_2x1.vhd)
--               sel Selecao da entrada
--                  a Entrada, sel = 0
--                  b Entrada, sel = 1
--                  y Saída y = nsel.a + sel.b
-----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mux2x1 is
    port (sel   : in STD_LOGIC;
          a,b   : in STD_LOGIC;
          y     : out STD_LOGIC);
end mux2x1;

architecture dataflow of mux2x1 is
begin
    y <= (a AND NOT sel) OR (b AND sel);
 end dataflow;
```

A simulação com *testbench*, para confirmar a funcionalidade do circuito do multiplexador 2x1, apresenta na saída do multiplexador dois sinais de relógio distintos, selecionados de forma assíncrona a título de exemplo e para introduzir a modelagem do sinal de *clock*.

```
-- ****************************************
-- Testbench para simulação Funcional do
--  ircuito: multiplexador 2x1:(mux_2x1.vhd)
--               sel Selecao da entrada
```

```
--                a Entrada, sel = 0
--                b Entrada, sel = 1
--                y Saída y = nsel.a + sel.b
-- ***************************************
ENTITY testbench5 IS END;
-----------------------------------------------
-- Testbench para mux_2x1.vhd
-- Validação sincrona (clk1 e clk2)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;

ARCHITECTURE tb_mux_2x1 OF testbench5 IS
---------------------------------
-- Declaração do componente mux2x1
---------------------------------
component mux2x1
    port (sel   : in STD_LOGIC;
          a,b   : in STD_LOGIC;
          y     : out STD_LOGIC);
end component;

constant T              : time := 5 ns; -- periodo para o clk_1
signal clk_1, clk_2     : std_logic;
signal tb_sel           : std_logic;

Begin

mux: mux2x1 PORT MAP (sel => tb_sel, a => clk_1, b => clk_2,
    y => open);

clk1: process                          -- clk_1 Generator
begin
    clk_1 <= '0', '1' after T/2, '0' after T;
    wait for T;
end process;

estimulo: PROCESS
   begin
      tb_sel <= '0'; clk_2 <= '0';
      wait for 22 ns; tb_sel <= '1';
      loop                             -- clk_2 Generator
          clk_2 <= '1';
          WAIT FOR 2 ns;
          clk_2 <= '1';
          WAIT FOR 8 ns;
       end loop;
    end PROCESS estimulo;
end tb_mux_2x1;
```

Implementação do processo de estímulo para o clk_1;

Implementação do processo de estímulo para o clk_2;

Para gerar estímulos síncronos em uma simulação podem ser utilizados diferentes métodos para descrever um sinal de relógio (c*lock*), pois o código VHDL permite a descrição de circuitos não sintetizáveis fisicamente.

Neste *testbench* são implementados dois métodos para a geração de *clock*. Na geração do sinal de relógio `clk_1` é utilizado um tipo constante (`constant T: time := 5 ns;`) para determinar o período do primeiro processo descrito no corpo da arquitetura do *testbench*. Esta forma simplifica a determinação do período do sinal de relógio `clk_1`, bastando para tal, escolher outro valor adequado de tempo para constante `T`. Neste método o sinal de relógio possui um ciclo de trabalho, do inglês *duty cycle*[11], padrão de 50%.

No segundo processo (`estimulo`) do *testbench* é utilizado um laço com a cláusula LOOP (terminado por END LOOP) com o objetivo de repetir a execução de linhas de código,

[11] *Duty Cycle* é utilizado para descrever a fração de tempo em que um sinal de relógio está em um nivel lógico "1" (estado ativo).

gerando um sinal de relógio. Nesta parte do código o gerador do segundo sinal de relógio `clk_2` é modelado, com a cláusula `WAIT FOR` (espera por), de forma a possuir um ciclo de trabalho ajustável (diferente de 50%).

O ciclo de trabalho, dado em termos percentuais, é a duração do sinal de relógio em nível lógico "1" durante um período de relógio. Pode ser calculado a partir da equação

$$\text{duty cycle} = \tau / T$$

onde,

τ é a duração do sinal de relógio é nível lógico "1";

T é o período do sinal de relógio.

O segundo sinal de relógio é modelado com um ciclo de trabalho definido pelas linhas 49 e 51 do código VHDL (Figura 2.15), gerando um sinal de relógio com período (T) de 10 ns, 2 ns em nivel lógico "1" e 8 ns em nivel "0", configurando um ciclo de trabalho de 20% (2/10).

Nesta simulação são apresentados somente os sinais do *port* do componente `mux` (multiplexador 2x1). Desta forma obtém-se uma janela "Wave – default" com a representação dos sinais de entrada e saída específicos do multiplexador, apresentando os sinais da coluna "Message" reduzidos, sem a indicação de todo o caminho (*path*). Esta configuração pode ser alternada de caminho completo para caminho curto simplesmente apontando e clicando no ícone na parte inferior à esquerda da coluna "Message" da janela "Wave – default", conforme ilustra a Figura 2.14.

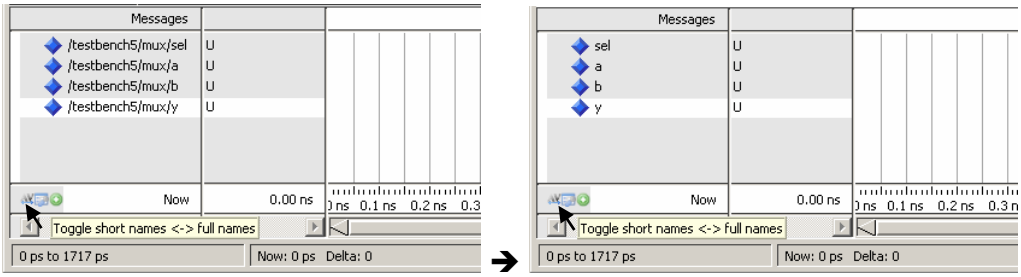


Figura 2.14 – Seleção para caminho curto da coluna "Message".

A Figura 2.15 ilustra os dois sinais de relógio gerados para estímulos das entradas (`a,b`) do multiplexador, bem como o sinal `sel` que comuta os sinais de relógio para saída `y` do multiplexador.

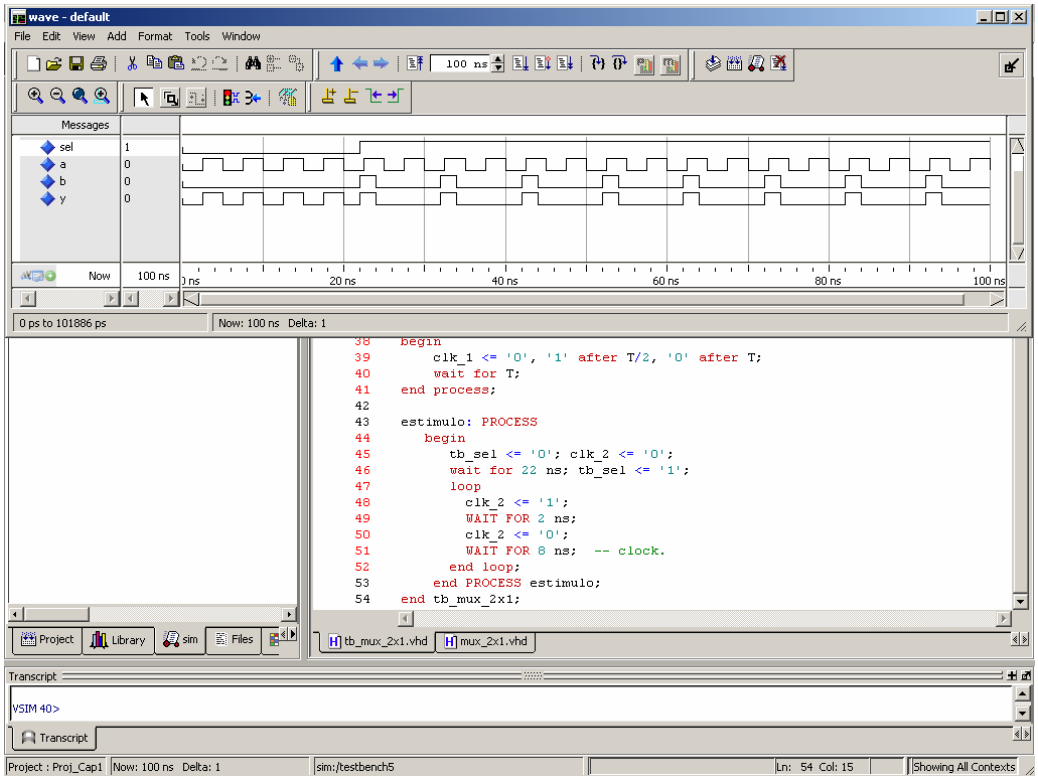


Figura 2.15 - Resultados da simulação do multiplexador 2x1.

Na Figura 2.16 está apresentado um detalhamento da simulação do multiplexador. Como pode ser observado, após 10 ns do início da simulação (cursor 1), a entrada `a` do multiplexador é transferida para a saída `y` do circuito, dado que `sel` foi inicializado com "0". Após 10 ns (cursor 2 em 15 ns) observa-se que o primeiro sinal de relógio (`clk_1`) do processo gerador possuir um período de 5 ns (5000 ps) e um ciclo de trabalho de 50%.

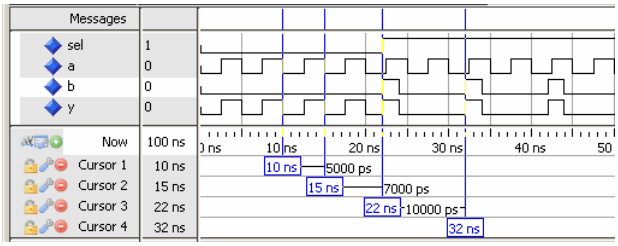


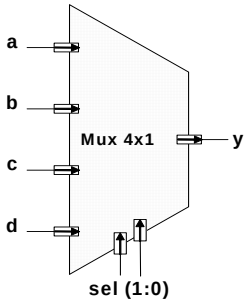
Figura 2.16 – Detalhamento dos resultados da simulação.

Em 22 ns de simulação (cursor 3) o sinal seletor `sel` foi levado a nível lógico "1", transferindo a entrada `b` do multiplexador, segundo sinal de relógio (`clk_2`), para a saída `y`.

O sinal de relógio `clk_2` está sendo gerado no laço LOOP/END LOOP (linhas 47 a 52 do código VHDL - Figura 2.15) com período de 10 ns (intervalo de 10000 ps entre os cursores 3 e 4) e um *duty cycle* de 20%.

Duas outras descrições são apresentadas, no Anexo B, com uso da declaração WHEN/ELSE. O primeiro exemplo apresenta um multiplexador 2x1 para barramentos de 8 bits, e o segundo exemplo de um *buffer tri-state* (alta impedância).

A partir do multiplexador anteriormente descrito, a Figura 2.17 ilustra o bloco diagrama e a tabela verdade da próxima estrutura a ser descrita em VHDL – a de um multiplexador 4x1. A estrutura do multiplexador em nível de portas lógicas está apresentada na Figura 2.18.



Entradas		Saídas
sel_1	sel_0	y
0	0	a
0	1	b
1	0	c
1	1	d

Figura 2.17 – Bloco diagrama e tabela verdade de um multiplexador 4x1.

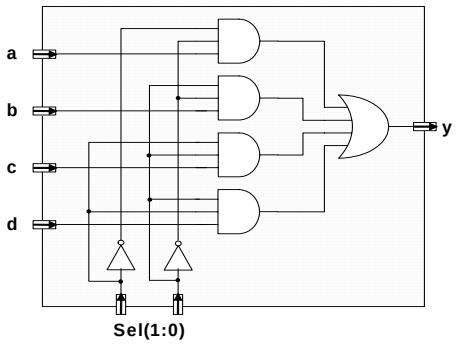


Figura 2.18 – Circuito do multiplexador 4x1 em nível de portas lógicas.

A implementação do multiplexador 4x1 exemplifica o uso das declarações WHEN/ELSE (*simple* WHEN) e de outras, tais como WITH/SELECT/WHEN (*selected* WHEN). Neste exemplo é apresentada uma solução com as declarações WHEN/ELSE conforme código VHDL transcrito como segue:

```
-----------------------------------------------
--  Circuito: multiplexador 4x1:(mux1_4x1.vhd)
--          sel (1:2) Selecao da entrada
--                     a Entrada, sel = 00
--                     b Entrada, sel = 01
--                     c Entrada, sel = 10
--                     d Entrada, sel = 11
--                     y Saída (WHEN/ELSE)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------

ENTITY mux4x1 IS
    PORT ( sel           : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
           a, b, c, d    : IN STD_LOGIC;
           y             : OUT STD_LOGIC);
    END mux4x1;
-----------------------------------------------
ARCHITECTURE mux1 OF mux4x1 IS
   BEGIN
     y <=   a WHEN sel="00" ELSE
            b WHEN sel="01" ELSE
            c WHEN sel="10" ELSE
            d;
   END mux1;
-----------------------------------------------
```

Uma proposta de *testbench* para validação do multiplexador 4x1 com as declarações WHEN/ELSE é apresentada conforme código VHDL transcrito como segue:

```
-- ****************************************
-- Testbench para simulação Funcional do
-- Circuito: multiplexador 4x1:(mux1_4x1.vhd)
--           sel (1:2) Selecao da entrada
--                   a Entrada, sel = 00
--                   b Entrada, sel = 01
--                   c Entrada, sel = 10
--                   d Entrada, sel = 11
--                   y Saída (WHEN/ELSE)
-- ****************************************
ENTITY testbench6 IS END;
-----------------------------------------------
-- Testbench para mux1_4x1.vhd
-- Validação sincrona (clk1 e clk2)
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE std.textio.ALL;

ARCHITECTURE tb_mux1_4x1 OF testbench6 IS
-----------------------------------
-- Declaração do componente mux2x1
-----------------------------------
component mux4x1
    PORT ( sel         : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
           a, b, c, d  : IN STD_LOGIC;
           y           : OUT STD_LOGIC);
end component;

constant T: time := 5 ns;       -- periodo para o clk_1
signal clk_1, clk_2   : std_logic;
signal tb_c, tb_d     : std_logic;
signal tb_sel         : STD_LOGIC_VECTOR (1 DOWNTO 0);

Begin

mux1: mux4x1 PORT MAP ( sel => tb_sel, a => clk_1, b => clk_2,
                        c => tb_c, d => tb_d, y => open);

clk1: process            -- gerador do clk_1
begin
    clk_1 <= '0', '1' after T/2, '0' after T;
    wait for T;
end process;

estímulo: PROCESS
   begin
      tb_sel <= "00"; clk_2 <= '0';
      tb_c<= '0'; tb_d <= '1';
      wait for 11 ns; tb_sel <= tb_sel + '1';
      loop                      -- gerador do clk_2
          clk_2 <= '1';
          WAIT FOR 2 ns;
          clk_2 <= '0';
          WAIT FOR 8 ns;
          tb_sel <= tb_sel + '1';
          if tb_sel = "01" then tb_c <= '1'; end if;
          if tb_sel = "10" then tb_d <= '0'; end if;
        end loop;
    end PROCESS estímulo;
end tb_mux1_4x1;
-----------------------------------------------
```

Nesta descrição um novo processo de estímulo é proposto baseado no *testbench* anterior (circuito do multiplexador 2x1), de forma a testar e validar o multiplexador 4x1. As modificações em relação ao anterior consistem em utilizar o LOOP gerador do segundo sinal de relógio (`clk_2`) para também gerar os estímulos das entradas seletoras (`sel`) do multiplexador 4x1. Este LOOP é também responsável pela geração dos sinais de estímulo para selecionar, bem como determinar os valores das entradas (`c, d`) do multiplexador.

A entrada `sel`, a medida que tem seu valor incrementado de "00" à "11", determina qual das entradas (`a, b, c` ou `d`) será direcionada para a saída `y` (Figura 2.19).

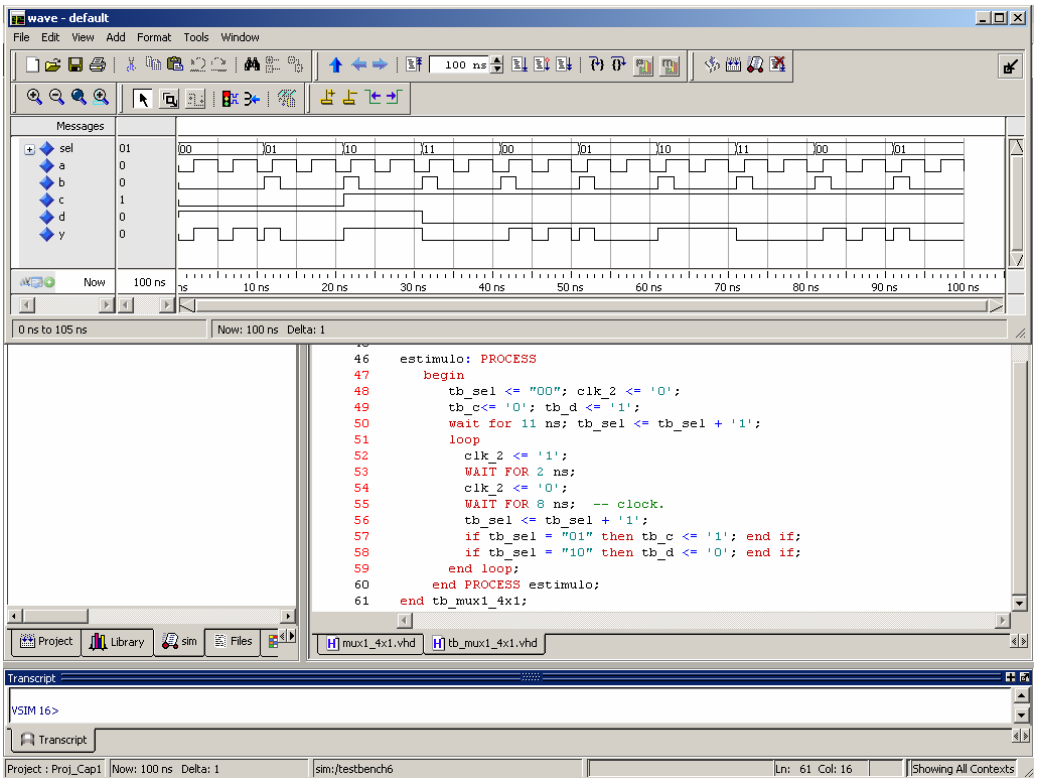


Figura 2.19 - Resultados da simulação do multiplexador 4x1.

A Figura 2.20 traz um detalhamento da simulação do multiplexador 4x1, onde pode ser observado que, após a inicialização de todos os estímulos de entrada (linhas 48 e 49 do código VHDL - Figura 2.19) é realizado o incremento de `tb_sel` (`tb_sel <= tb_sel + '1';`) (linha 50 - Figura 2.19), selecionando as diferentes entradas da componente `mux4x1`.

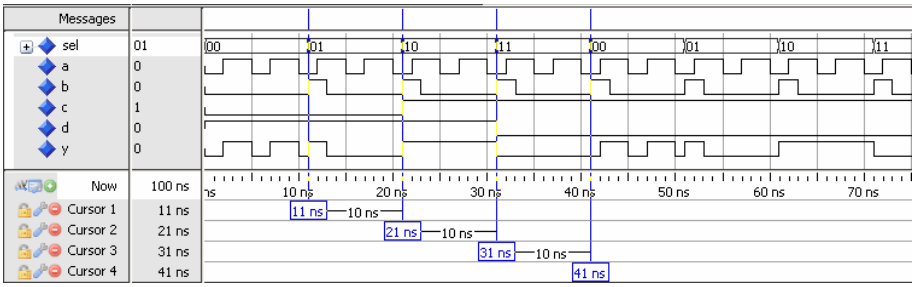


Figura 2.20 – Detalhamento dos resultados da simulação.

Passados 11 ns (cursor 1) do início da simulação o processo entra no LOOP que irá repetir-se enquanto a simulação estiver ativa. Este LOOP gerador do segundo sinal de relógio (`clk_2`) é também responsável pela geração dos sinais de estímulo das outras duas entradas (`c, d`) do multiplexador, que serão atualizadas com novos valores lógicos aos 21 e 31 ns (cursores 2 e 3) respectivamente (Figura 2.20).

Os estímulos para as entradas seletoras são gerados (linhas 50 e 56 do código VHDL - Figura 2.19) por meio de um contador de dois bits (`tb_sel`). Este contador testa de forma incremental todas as quatro entradas do multiplexador, conforme pode ser visto na seqüência identificada pelos cursores ilustrados na Figura 2.20. Aos 41 ns (cursor 4) o contador atinge o valor lógico "00" iniciando uma nova contagem na entrada seletora. Os valores lógicos as entradas do multiplexador (`c, d`) permanecem inalteradas com os valores lógicos "1" e "0", respectivamente.

Neste processo são utilizadas cláusulas que indicam decisão, cuja sintaxe básica é IF <condição> THEN <atribuição>, para simultaneamente selecionar em seqüência crescente todas as entradas (`a, b, c, d`), bem como modificar os valores iniciais dos sinais de estímulo para as entradas (`c, d`) do multiplexador.

Duas outras descrições possíveis são apresentadas no Anexo C, com uso da declaração WITH/SELECT/WHEN (*selected* WHEN). Os resultados simulados são evidentemente idênticos aos obtidos do componente multiplexador (mux4x1) anteriormente descrito em código concorrente.

Os estímulos para as entradas seletoras são gerados (linhas 50 e 56 do código VHDL - Figura 2.19) por meio de um contador de dois bits (`tb_sel`). Este contador testa, de forma incremental, todas as quatro entradas do multiplexador, conforme pode ser visto na seqüência identificada pelos cursores ilustrados na Figura 2.20. Aos 41 ns (cursor 4) o contador atinge o valor lógico "00" iniciando uma nova contagem nas entradas seletoras. Contudo os valores lógicos as entradas (`c, d`) do multiplexador permanecem indefinidamente com os valores lógicos "1" e "0", respectivamente.

Neste processo são utilizadas cláusulas que indicam decisão, cuja sintaxe básica é "IF condição THEN atribuição", para simultaneamente selecionar em seqüência crescente todas as entradas (`a, b, c, d`), bem como modificar os valores iniciais dos sinais de estímulo para as entradas (`c, d`) do multiplexador.

As declarações em VHDL da classe combinacional são WHEN e GENERATE. Além destas, atribuições usando apenas os operadores lógicos (AND, OR, +, -, *, sll, sra, etc.) também podem ser usadas para construir códigos concorrentes dentro de processos (process). No Anexo D, a cláusula CASE utilizada em seções de código seqüenciais é combinada com WHEN (classe combinacional) para descrever um multiplexdor 2x1.

A lógica combinacional, por definição, é aquela na qual a saída do circuito depende unicamente do nível atual dos sinais aplicados na entrada do circuito, como ilustra a Figura 21(a). Fica evidente que, a princípio, o sistema não requer memória e pode ser implementado usando portas lógica básicas.

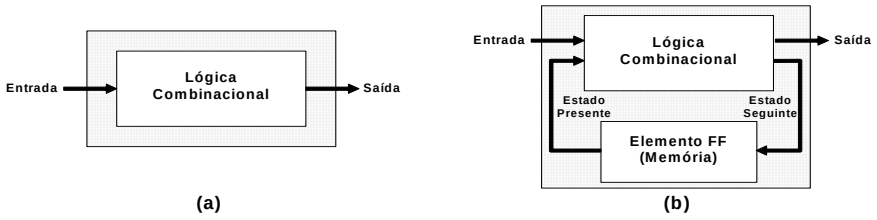


Figura 21 – Lógica combinacional (a) versus seqüencial (b).

Em contrapartida, a lógica seqüencial é definida como aquela em que a saída depende não só do estado atual das entradas, mas também do estado anterior da saída, conforme ilustrado na Figura 21(b). Portanto, elementos de armazenamento do tipo *flip-flop* (FF) são utilizados como memória do estado anterior, e são ligados ao bloco de lógica combinacional através de um laço de realimentação (*feedback loop*), de tal modo que também afetam a saída futura do circuito.

Um erro comum é pensar que qualquer circuito que possua elementos de armazenamento (*flip-flops*) é seqüencial. A Random Access Memory (RAM) é um exemplo. A memória RAM pode ser modelada como ilustra a Figura 22.

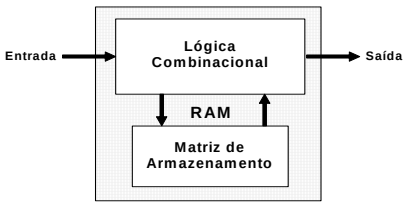


Figura 22 – Modelo de memória RAM.

Observe que os elementos que aparecem na matriz de armazenamento entre o caminho da entrada até a saída não utilizam um laço de realimentação (*feedback loop*). Por exemplo, a operação de leitura da memória depende apenas do vetor endereço, que é o estado presente aplicado à entrada da RAM, e os resultados obtidos na saída não são afetados por acessos anteriores à RAM.