SERVIÇO PÚBLICO FEDERAL
UNIVERSIDADE FEDERAL DO SUL E SUDESTE DO PARÁ - UNIFESSPA
INSTITUTO DE GEOCIÊNCIAS E ENGENHARIAS - IGE
FACULDADE DE COMPUTAÇÃO E ENG. ELÉTRICA – FACEEL
CURSO ENGENHARIA DE COMPUTAÇÃO

Microeletrônica

T-2018

Prof. José Carlos Da Silva
jcdsilv@hotmail.com
jose-carlos.silva@unifesspa.edu.br
whatsApp: 94-981431852

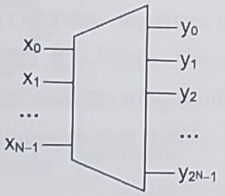
Maio/2022

ATIVIDADE 3
(DECODIFICADOR DE ENDEREÇO GENÉRICO)

20.1 DECODIFICADOR DE ENDEREÇO GENÉRICO

Decodificadores de endereço foram estudados na Seção 11.5. Ilustraremos agora o projeto do decodificador de endereço da Figura 20.1 (emprestado da Figura 11.7) nas duas situações seguintes:

- Com *N*=3 usando a instrução WHEN (a tabela-verdade para esse caso é mostrada na Figura 20.1).
- Ainda usando WHEN, mas para tamanho arbitrário (*N* genérico).



x	y
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

FIGURA 20.1. Decodificador de endereço. A tabela-verdade é para o caso de *N=3*.

CÓDIGO PARA *N*=3 COM `WHEN`

Um código VHDL para esse problema é mostrado abaixo. Como mencionamos na Seção 19.2, três seções de código são necessárias; contudo, a primeira (declarações de bibliotecas/pacotes) foi omitida porque neste exemplo são empregadas somente as bibliotecas padrões, as quais são visíveis automaticamente.

ATIVIDADE 3
(DECODIFICADOR DE ENDEREÇO GENÉRICO)

A segunda seção do código (ENTITY) é responsável por definir as portas (isto é, entradas e saídas) do circuito. Ela está nas linhas 2–5, sob o nome *address_decoder*. O sinal *x* é declarado como uma entrada de 3 bits, enquanto que *y* é uma saída de 8 bits, ambos do tipo BIT_VECTOR.

A terceira seção do código (ARCHITECTURE) é responsável pelo código propriamente dito (estrutura ou comportamento do circuito), e consta nas linhas 7–17, neste caso com o mesmo nome da entidade (poderia ser praticamente qualquer nome). A instrução concorrente WHEN, vista na Seção 19.10, foi empregada para implementar o circuito (linhas 9–16). Observe que essa solução não é muito prática porque o código cresce com *N*.

As linhas tracejadas (1, 6 e 18) foram empregadas somente para melhorar a organização e legibilidade do código. Observe também que *x* e *y* poderiam ter sido declarados como INTEGER, por exemplo, em vez de BIT_VECTOR.

```
1   -----Code for N=3 with WHEN:----------------------
2   ENTITY address_decoder IS
3      PORT (x: IN BIT_VECTOR(2 DOWNTO 0);
4            y: OUT BIT_VECTOR(7 DOWNTO 0));
5   END address_decoder;
6   -------------------------------------------------
7   ARCHITECTURE address_decoder OF address_decoder IS
8   BEGIN
9      y <= "00000001" WHEN x = "000" ELSE
10          "00000010" WHEN x = "001" ELSE
11          "00000100" WHEN x = "010" ELSE
12          "00001000" WHEN x = "011" ELSE
13          "00010000" WHEN x = "100" ELSE
14          "00100000" WHEN x = "101" ELSE
15          "01000000" WHEN x = "110" ELSE
16          "10000000";
17  END address_decoder;
18  -------------------------------------------------
```

Resultados de simulação do código acima são apresentados na Figura 20.2. O gráfico de cima mostra somente os valores agrupados de *x* e *y*, enquanto que o segundo gráfico mostra também os bits individuais de *x* e *y*. Em todas as outras simulações que virão em seguida, será exibido somente o primeiro tipo de apresentação (mais compacto). Como podemos observar, o circuito realmente funciona como esperado (*x* abrange a faixa de 0 to 7, enquanto que *y* tem somente um bit alto de cada vez – observe que os valores de *y* são todos potências de 2).

CÓDIGO PARA *N* ARBITRÁRIO COM WHEN E GENERATE

Observe que o tamanho do próximo código é fixo, independentemente do valor *N*, o qual foi especificado usando GENERIC (linha 3). Portanto, basta mudar o valor de *N* naquela linha para obter qualquer tamanho de decodificador de endereço.

Neste caso, *x* foi declarado como INTEGER (linha 4) e a instrução GENERATE (linhas 10–12) foi empregada em combinação com WHEN para criar 2^N instâncias de *y* (isto é, de *y*(0) a *y*(2^N–1)), cujos índices foram copiados de *x* com o atributo x'RANGE (linha 10). A etiqueta (obrigatória) escolhida para GENERATE foi *gen*. Para *N*=3, os resultados da simulação são obviamente os mesmos vistos na Figura 20.2. (*Nota:* veja exercício 20.2.)

```
1   -----Code for arbitrary N:-----------------------------
2   ENTITY address_decoder IS
3      GENERIC (N: INTEGER := 3); --pode ser qualquer valor
4      PORT (x: IN INTEGER RANGE 0 TO 2**N-1;
5            y: OUT BIT_VECTOR(2**N-1 DOWNTO 0));
6   END address_decoder;
7   ------------------------------------------------------
8   ARCHITECTURE address_decoder OF address_decoder IS
9   BEGIN
10     gen: FOR i IN x'RANGE GENERATE
11        y(i) <= '1' WHEN i = x ELSE '0';
12     END GENERATE;
13  END address_decoder;
14  ------------------------------------------------------
```

ATIVIDADE 3
(DECODIFICADOR DE ENDEREÇO GENÉRICO)

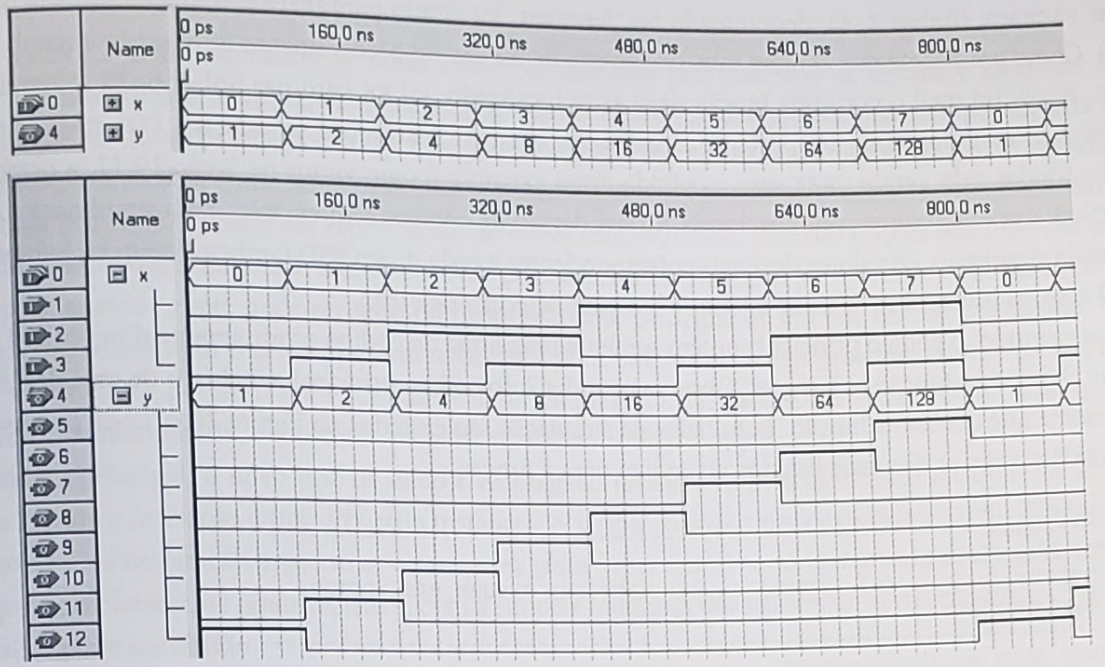


FIGURA 20.2. Resultados de simulação do decodificador de endereço da Figura 20.1, com $N = 3$. O gráfico de cima mostra somente os valores agrupados de *x* e *y*, enquanto que o segundo gráfico mostra também os bits individuais de *x* e *y* (somente o primeiro tipo de apresentação será exibido em exemplos futuros).

ATIVIDADE 3
(REGISTRADOR)

22.1 REGISTRADOR DE DESLOCAMENTO COM DATA-LOAD

A Figura 22.1 mostra um registrador de deslocamento (SR) de *N* bits e *M* estágios com capacidade para carregamento (*data load*) do valor inicial (esse circuito foi estudado na Seção 14.1). Quando *load*='1', o vetor *x* é carregado no SR na próxima borda ascendente do clock, enquanto que para *load*='0' o circuito opera como um SR regular. Nesta seção, ilustraremos o projeto deste SR sob as seguintes duas premissas: (i) *M* e *N* devem ser *genéricos*; (ii) a abordagem deve ser *estrutural* (isto é, usando `COMPONENT` para instanciar os multiplexadores e flip-flops).

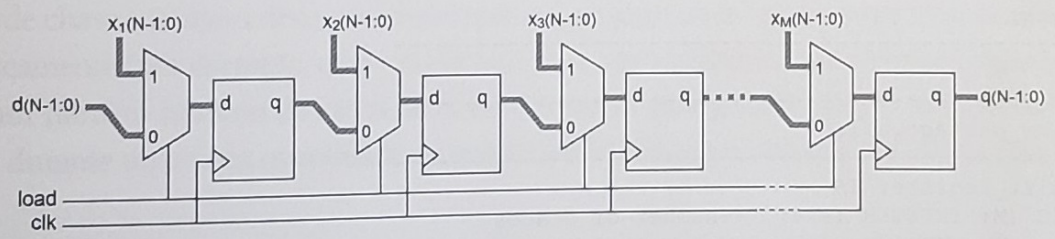


FIGURA 22.1. Registrador de deslocamento de *N* bits e *M* estágios com capacidade de carregamento (*data load*) do valor inicial.

Um código VHDL para esse circuito é mostrado a seguir. Como *M* e *N* devem ser valores arbitrários, é necessário um tipo de dado definido pelo usuário para *x* porque nenhum dos tipos predefinidos (Figura 19.6) satisfaz a necessidade presente. Como tal tipo é necessário no início do código principal (na `ENTITY` — veja

ATIVIDADE 3 (REGISTRADOR)

linha 8 do código principal), ele foi especificado em um PACKAGE (denominado *my_package*), que fica visível para o projeto por meio da linha 2 do código principal.

O multiplexador e o banco de flip-flops foram projetados separadamente, porque devem ser instanciados como *componentes* no código principal (como vimos na Seção 19.14, COMPONENT é simplesmente um código VHDL convencional que foi previamente projetado). Observe que esses dois códigos (*multiplexer* e *ff_bank*) também são genéricos, portanto seus parâmetros genéricos devem ser sobrescritos pelo código principal (veja GENERIC MAP nas linhas 34 e 36 do código principal). A instrução GENERATE (linhas 33–38) foi empregada para criar *M* instâncias dessas unidades. Observe que as atribuições adotadas em GENERIC MAP e PORT MAP neste exemplo são todas *posicionais* (Seção 19.14). Resultados de simulação deste código são apresentados na Figura 22.2.

```
1   -----Package:-------------------------------------------------------------
2   PACKAGE my_package IS
3      CONSTANT bits: POSITIVE := 8;
4      TYPE x_input IS ARRAY (NATURAL RANGE <>) OF BIT_VECTOR(bits-1 DOWNTO 0);
5   END my_package;
6   ---------------------------------------------------------------------------

1   -----Multiplexer (a component):----------------------
2   ENTITY multiplexer IS
3      GENERIC (bits: POSITIVE);
4      PORT (inp1, inp2: IN BIT_VECTOR(bits-1 DOWNTO 0);
5            sel: IN BIT;
6            outp: OUT BIT_VECTOR(bits-1 DOWNTO 0));
7   END multiplexer;
8   ------------------------------------------------------
9   ARCHITECTURE multiplexer OF multiplexer IS
10  BEGIN
11     outp <= inp1 WHEN sel='0' ELSE inp2;
12  END multiplexer;
13  ------------------------------------------------------

1   -----ff_bank (another component):----------------------
2   ENTITY ff_bank IS
3      GENERIC (bits: POSITIVE);
4      PORT (d: IN BIT_VECTOR(bits-1 DOWNTO 0);
5            clk: IN BIT;
6            q: OUT BIT_VECTOR(bits-1 DOWNTO 0));
7   END ff_bank;
8   ------------------------------------------------------
9   ARCHITECTURE ff_bank OF ff_bank IS
10  BEGIN
11     PROCESS (clk)
12     BEGIN
13        IF (clk'EVENT AND clk='1') THEN
14           q <= d;
15        END IF;
16     END PROCESS;
17  END ff_bank;
18  ------------------------------------------------------

1   -----Main code:----------------------------------------
2   USE work.my_package.all;
3   ------------------------------------------------------
4   ENTITY shift_register IS
5      GENERIC (M: INTEGER := 4;  --number of stages
6               N: INTEGER := 8); --number of bits
7      PORT (clk, load: IN BIT;
8            x: IN x_input(1 TO M);
9            d: IN BIT_VECTOR(N-1 DOWNTO 0);
10           q: OUT BIT_VECTOR(N-1 DOWNTO 0));
11  END shift_register;
12  ------------------------------------------------------
13  ARCHITECTURE structural OF shift_register IS
14     SIGNAL temp1: x_input(0 TO M);
15     SIGNAL temp2: x_input(1 TO M);
16     ----------------------------
```

ATIVIDADE 3 (REGISTRADOR)


```
17    COMPONENT multiplexer IS
18       GENERIC (bits: POSITIVE);
19       PORT (inp1, inp2: IN BIT_VECTOR(bits-1 DOWNTO 0);
20             sel: IN BIT;
21             outp: OUT BIT_VECTOR(bits-1 DOWNTO 0));
22    END COMPONENT;
23    ------------------------------
24    COMPONENT ff_bank IS
25       GENERIC (bits: POSITIVE);
26       PORT (d: IN BIT_VECTOR(bits-1 DOWNTO 0);
27             clk: IN BIT;
28             q: OUT BIT_VECTOR(bits-1 DOWNTO 0));
29    END COMPONENT;
30    ------------------------------
31 BEGIN
32    temp1(0) <= d;
33    g: FOR i IN 1 TO M GENERATE
34       mux: multiplexer GENERIC MAP (N)
35          PORT MAP (temp1(i-1), x(i), load, temp2(i));
36       ff: ff_bank GENERIC MAP (N)
37          PORT MAP (temp2(i), clk, temp1(i));
38    END GENERATE g;
39    q <= temp1(M);
40 END structural;
41 -------------------------------------------------------------
```

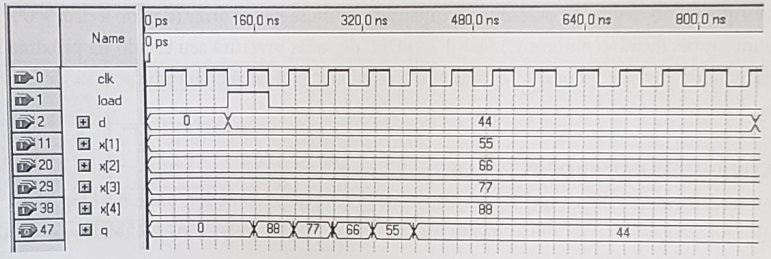


FIGURA 22.2. Resultados de simulação do código para o registrador de deslocamento da Figura 22.1.

OBRIGADO