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 02
(MEMORIA - ROM)

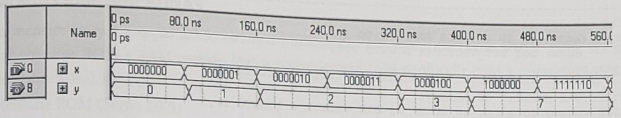


FIGURA 20.7. Resultados de simulação do codificador de prioridade da Figura 11.21(b) para $N = 7$.

20.5 MEMÓRIA ROM

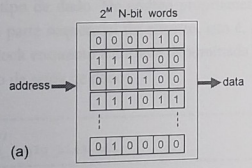
A construção de memórias, do ponto de vista tecnológico, foi discutida minuciosamente nos Capítulos 16 e 17. Nesta seção e na próxima, ilustraremos como VHDL pode ser usado para implementar circuitos ROM e RAM usando circuitos lógicos convencionais (não confundir com a instanciação de blocos de memória pré-fabricados). Os seguintes quatro casos serão examinados:

- Memória ROM (Seção 20.5)
- Memória RAM síncrona com barramentos de I/O separados (Seção 20.6)
- Memória RAM síncrona com barramento de I/O único (Seção 20.6)
- Memória RAM síncrona com barramentos de endereços R/W e de I/O separados (Seção 20.6)

MEMÓRIA ROM

Uma memória ROM é normalmente implementada usando células lógicas regulares em CPLDs (uma exceção é o CPLD MAX II — Seção 18.3) ou *lookup tables* (LUTs) em FPGAs (e em alguns CPLDs, como o MAX II). Em ambos os casos, o modelo LUT pode ser empregado, como na Figura 20.8(a), tendo *address* como a única entrada e *data* (isto é, o conteúdo armazenado no endereço fornecido) como a única saída.

A utilização desse tipo de circuito é exemplificada na Figura 20.8(b), que mostra uma tabela de conversão de BCD para SSD (Exemplo 11.4 e Seção 20.2). O primeiro entra pelo barramento de *endereço*, fazendo com que o segundo seja fornecido no barramento de *dados*.



Address	Stored word
0000 (0)	1111110 (126)
0001 (1)	0110000 (48)
0010 (2)	1101101 (109)
0011 (3)	1111001 (121)
0100 (4)	0110011 (51)
0101 (5)	1011011 (91)
0110 (6)	1011111 (95)
0111 (7)	1110000 (112)
1000 (8)	1111111 (127)
1001 (9)	1111011 (123)

(b)

FIGURA 20.8. (a) Arquitetura ROM; (b) Exemplo de ROM, a qual converte BCD para SSD.

CÓDIGO VHDL

Um código VHDL para essa ROM é mostrado a seguir. Na linha 8 foi especificado um novo tipo de dado, denominado *memory*, que permite a criação de um arranjo 1D × 1D com um total de 10 × 7 bits. Em seguida, uma CONSTANT, denominada *rom*, foi declarada na linha 9 em conformidade com o novo tipo de dado, com os dez valores de sete bits cada especificados nas linhas 10–19. Finalmente, no código propriamente dito (linha 21), ocorre a operação de leitura da memória.

ATIVIDADE 02
(ROM)


```
1  ------------------------------------------------------
2  ENTITY memory1 IS
3     PORT (address: IN INTEGER RANGE 0 TO 9;
4           data: OUT BIT_VECTOR(6 DOWNTO 0));
5  END memory1;
6  ------------------------------------------------------
7  ARCHITECTURE memory1 OF memory1 IS
8     TYPE memory IS ARRAY (0 TO 9) OF BIT_VECTOR(6 DOWNTO 0);
9     CONSTANT rom: memory := (
10       "1111110",
11       "0110000",
12       "1101101",
13       "1111001",
14       "0110011",
15       "1011011",
16       "1011111",
17       "1110000",
18       "1111111",
19       "1111011");
20 BEGIN
21    data <= rom(address);
22 END memory1;
23 ------------------------------------------------------
```

SOLUÇÃO COM `FUNCTION`

Uma outra abordagem é mostrada no código a seguir. Desta vez, a memória de conversão (*rom*) de BCD para SSD foi criada usando uma `FUNCTION` (Seção 19.15). Tal função, denominada *bcd_to_ssd*, foi construída em um `PACKAGE`, com o respectivo `PACKAGE BODY` (como na Seção 20.2), ambos denominados *my_package*. No código principal, basta uma chamada à função (linha 11) para produzir o circuito desejado. Observe na linha 2 do código principal a inclusão de uma declaração para tornar o pacote *my_package* visível ao compilador.

```
1  -----Package:-------------------------------------------------
2  PACKAGE my_package IS
3     FUNCTION bcd_to_ssd (SIGNAL bcd: INTEGER) RETURN BIT_VECTOR;
4  END my_package;
5  -------------------------------------------------------------
6  PACKAGE BODY my_package IS
7     FUNCTION bcd_to_ssd (SIGNAL bcd: INTEGER) RETURN BIT_VECTOR IS
8        TYPE memory IS ARRAY (0 TO 9) OF BIT_VECTOR(6 DOWNTO 0);
9        CONSTANT rom: memory := (
10          "1111110",
11          "0110000",
12          "1101101",
13          "1111001",
14          "0110011",
15          "1011011",
16          "1011111",
17          "1110000",
18          "1111111",
19          "1111011");
20    BEGIN
21       RETURN rom(bcd);
22    END bcd_to_ssd;
23 END my_package;
24 -------------------------------------------------------------
```

```
1  -----Main code:---------------------------
2  USE work.my_package.all;
3  ------------------------------------------
4  ENTITY ssd_driver IS
5     PORT (bcd: IN INTEGER RANGE 0 TO 9;
6           ssd: OUT BIT_VECTOR(6 DOWNTO 0));
7  END ssd_driver;
8  ------------------------------------------
9  ARCHITECTURE ssd_driver OF ssd_driver IS
10 BEGIN
11    ssd <= bcd_to_ssd(bcd);
12 END ssd_driver;
13 ------------------------------------------
```

ATIVIDADE 02
(RAM - 01)

20.6 MEMÓRIAS RAM SÍNCRONAS

Como mencionamos na Seção 20.5, estamos interessados em examinar como VHDL pode ser usado para implementar circuitos ROM e RAM usando lógica convencional (não confundir com a instanciação de blocos de memória pré-fabricados). Os seguintes quatro casos são examinados:

- Memória ROM (Seção 20.5)
- Memória RAM síncrona com barramentos de I/O separados (Seção 20.6)
- Memória RAM síncrona com barramento de I/O único (Seção 20.6)
- Memória RAM síncrona com barramentos de endereços R/W e de I/O separados (Seção 20.6)

RAM SÍNCRONA COM BARRAMENTOS DE I/O SEPARADOS

A Figura 20.9 mostra uma memória RAM síncrona com barramentos para entrada (*data_in*) e saída (*data_out*) de dados separados. Para todas as memórias, consideraremos que o número de bits de endereço é *M* e de bits de dados é *N*, de forma que a memória contém 2^M palavras de *N* bits. Observe que, ao contrário da ROM vista, o circuito na Figura 20.9 é *síncrono* (clocado, implementado com flip-flops).

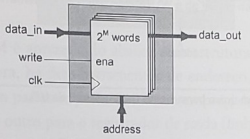


FIGURA 20.9. Memória RAM síncrona com barramentos de I/O separados.

Um código VHDL para essa RAM é apresentado a seguir. Observe que nenhuma das portas (linhas 8–11) é bidirecional. Como antes, um novo tipo de dado foi definido (denominado *memory*, linha 15) para permitir a criação de um arranjo 1D × 1D com um total de $2^M \times N$ bits. Na linha 16, um sinal denominado *ram* foi declarado como pertencente a esse tipo de dado. No código propriamente dito (ARCHITECTURE) foi usado um PROCESS (linhas 18–25) para criar a parte sequencial do circuito, isto é, os flip-flops que armazenam *data_in* quando ocorre uma borda positiva de clock enquanto a entrada denominada *write* (write enable) está alta. Finalmente, na linha 26, foi criado o barramento de saída de dados.

```
1   ----------------------------------------------------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ----------------------------------------------------------------------
5   ENTITY memory2 IS
6      GENERIC (N: INTEGER := 8; --Width of data bus
7               M: INTEGER := 4); --Width of address bus
8      PORT (clk, write: IN STD_LOGIC;
9            address: IN INTEGER RANGE 0 TO 2**M-1;
10           data_in: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
11           data_out: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
12  END memory2;
13  ----------------------------------------------------------------------
14  ARCHITECTURE memory2 OF memory2 IS
15     TYPE memory IS ARRAY (0 TO 2**M-1) OF STD_LOGIC_VECTOR(N-1 DOWNTO 0);
16     SIGNAL ram: memory;
17  BEGIN
18     PROCESS (clk)
19     BEGIN
20        IF (clk'EVENT AND clk = '1') THEN
```

ATIVIDADE 02
(RAM – 01 E 02)


```
21            IF (write = '1') THEN
22               ram(address) <= data_in;
23            END IF;
24         END IF;
25      END PROCESS;
26      data_out <= ram(address);
27   END memory2;
28   ------------------------------------------------------------
```

RAM SÍNCRONA COM BARRAMENTO DE I/O ÚNICO

Uma RAM síncrona com um único barramento para entrada e saída de dados é representada na Figura 20.10. A diferença fundamental entre ela e a RAM anterior é que agora o barramento de dados é único, portanto ele necessita ser bidirecional. Para tal, é necessário um buffer de três estados a fim de desligar a saída quando dados têm de ser escritos na RAM.

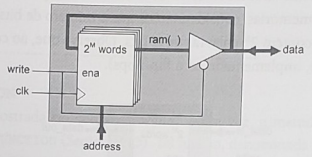


FIGURA 20.10. Memória RAM síncrona com um único barramento (bidirecional) de I/O.

Um código VHDL para esse circuito é apresentado a seguir. Como antes, foram empregados dois parâmetros genéricos (*N* e *M*, linhas 6–7) para especificar as larguras dos barramentos de dados e endereço. Observe também que, agora, uma das portas é bidirecional (*data*, linha 10). Para construir a parte sequencial do circuito (isto é, o banco de flip-flops), novamente foi empregado um PROCESS (linhas 18–25), enquanto que a parte combinacional (buffer de três estados) foi construída com a instrução concorrente WHEN (linha 26).

```
1    ------------------------------------------------------
2    LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    ------------------------------------------------------
5    ENTITY memory3 IS
6       GENERIC (N: INTEGER := 8; --Width of data bus
7                M: INTEGER := 4); --Width of address bus
8       PORT (clk, write: IN STD_LOGIC;
9             address: IN INTEGER RANGE 0 TO 2**M-1;
10            data: INOUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
11   END memory3;
12   ------------------------------------------------------
13   ARCHITECTURE memory3 OF memory3 IS
14      TYPE memory IS ARRAY (0 TO 2**M-1) OF
15         STD_LOGIC_VECTOR(N-1 DOWNTO 0);
16      SIGNAL ram: memory;
17   BEGIN
18      PROCESS (clk)
19      BEGIN
20         IF (clk'EVENT AND clk = '1') THEN
21            IF (write = '1') THEN
22               ram(address) <= data;
23            END IF;
24         END IF;
25      END PROCESS;
26      data <= ram(address) WHEN write='0' ELSE (OTHERS=>'Z');
27   END memory3;
28   ------------------------------------------------------
```

ATIVIDADE 02
(RAM - 02)

RAM SÍNCRONA COM BARRAMENTOS DE ENDEREÇOS R/W E DE I/O SEPARADOS

A última memória a ser discutida nesta seção é ilustrada na Figura 20.11. Trata-se de uma RAM com barramentos separados para dados e tambem barramentos separados para endereços de leitura e escrita (R/W). Desta forma, as operações de leitura e escrita podem ser executadas independentemente, sob o controle de dois clocks (*clk*1 para escrita, *clk*2 para leitura). O número total de flip-flops agora é $(2^M+1)\cdot N$ em vez de $2^M N$ porque a palavra selecionada por *rd_address* é armazenada na saída *data_out*.

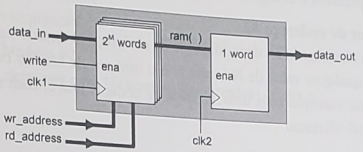


FIGURA 20.11. Memória RAM síncrona com barramentos de endereço de R/W e de I/O de dados separados.

Um código VHDL para essa RAM é apresentado abaixo. Sua estrutura global é semelhante à do circuito da Figura 20.9, com a diferença que, agora, há dois barramentos de endereço, dois clocks e um conjunto adicional de flip-flops. Como esse circuito só tem partes sequenciais, foram empregados dois processos, um para o banco de registradores global (linhas 20–27) e o outro para o registrador de saída (linhas 28–33).

```
1   ---------------------------------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ---------------------------------------------------
5   ENTITY memory4 IS
6      GENERIC (N: INTEGER := 8; --Width of data bus
7               M: INTEGER := 4); --Width of address bus
8      PORT (clk1, clk2, write: IN STD_LOGIC;
9            rd_address: IN INTEGER RANGE 0 TO 2**M-1;
10           wr_address: IN INTEGER RANGE 0 TO 2**M-1;
11           data_in: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
12           data_out: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
13  END memory4;
14  ---------------------------------------------------
15  ARCHITECTURE memory4 OF memory4 IS
16     TYPE memory IS ARRAY (0 TO 2**M-1) OF
17        STD_LOGIC_VECTOR(N-1 DOWNTO 0);
18     SIGNAL ram: memory;
19  BEGIN
20     PROCESS (clk1)
21     BEGIN
22        IF (clk1'EVENT AND clk1='1') THEN
23           IF (write = '1') THEN
24              ram(wr_address) <= data_in;
25           END IF;
26        END IF;
27     END PROCESS;
28     PROCESS (clk2)
29     BEGIN
30        IF (clk2'EVENT AND clk2 = '1') THEN
31           data_out <= ram(rd_address);
32        END IF;
33     END PROCESS;
34  END memory4;
35  ---------------------------------------------------
```

OBRIGADO