ENG1148 - Computação Digital - 2019.2

Relatório - Trabalho Final

Professor: Alberto Elias

Data: 25/09/2019

Grupo:

- **Alexandre de Mello: 1413183**
- **Rodrigo Pumar: 1221007**

1)Introdução

Neste trabalho, desenvolvido em 5 módulos, CPU2, ALU, RAM2, LCD e MapChar, implementamos um computador que executa até 32 instruções. A CPU controla o computador, a ALU executa operações lógicas, a RAM2 armazena as instruções a serem realizadas, ou o dados a serem lidos. O módulo LCD mostra no display do FPGA a instrução escrita com caracteres alfanumericos, e o MapChar é um módulo complementar ao módulo LCD, que mapeia os 11 caracteres que serão escritos no LCD a cada mudança de instrução.

2)Entidades

```
entity CPU2 is
   generic(
       CLOCK_COUNT_BUFFER_SIZE : integer := 25
   );
   port(CLK : in std_logic; -- clock do programa
       LCD_E : out std_logic; -- Enable do LCD
       LCD_RW : out std_logic; -- read or write
       LCD_RS : out std_logic; -- data manipulation or addressing
         DISABLE_STRATA_FLASH: out std_logic; -- desabilita a StrataFlash
por conflito com LCD
       SF_D: out std_logic_vector(3 downto 0); -- led que indica zero
       ZERO: out std_logic; -- led que indica zero
       NEGATIVE: out std_logic; -- led que indica negativo
       RESET: in std_logic; -- reset
         LEDS: out std_logic_vector(4 downto 0) -- leds data is mirror of
data in ram position 30
   );
```

```
end CPU2;
```

```
CLOCK_COUNT_BUFFER_SIZE : integer := 25
```

Essa variável é usada para que possamos definir na simulação clock equivalente ao clock do programa, para melhorar visibilidade da simulação.

```
entity ALU is
   generic(
       opN: integer := 5
   );
   port(
       reset: in std_logic;
       reg_A: in std_logic_vector(opN-1 downto 0); -- registrador A
       reg_B: in std_logic_vector(opN-1 downto 0); -- registrador B
       opcode: in std_logic_vector(4 downto 0); -- codigo da instrução
       zero: out std_logic; -- flag de que operação resultou em zero
           negative: out std_logic;-- flag de que operação resultou em
negativo
           result: out std_logic_vector(opN-1 downto 0) -- resultado da
operação da ALU
   );
end ALU;
```

```
entity LCD is
   generic(    N: integer :=20 );
   port(
       CLK : in std_logic; -- clk for display
       LCD_E : out std_logic; -- Enable do LCD
       LCD_RW : out std_logic; -- read or write
       LCD_RS : out std_logic; -- data manipulation or addressing
         DISABLE_STRATA_FLASH: out std_logic; -- Disable StrataFlash for
display
       SF_D: out std_logic_vector(3 downto 0); -- Data sent to display
        INSTRUCTION: in integer range 0 to 31 --Current Instruction to be
displayed
   );
end LCD;
```

```
entity RAM2 is
 port (
   clk      : in  std_logic;-- clk from CPU
   reset    : in  std_logic;-- reset
   we       : in  std_logic;  -- write enable
   address  : in  integer range 0 to 31; -- adress for current instruction
   datain   : in  std_logic_vector(4 downto 0);-- data to be written
   dataout  : out std_logic_vector(4 downto 0); --data read from ram
    dataAt30 : out std_logic_vector(4 downto 0) -- data at position 30 of
ram
 );
end entity RAM2;
```

```
entity MapChar is
   port(
       CLK: in std_logic; -- clock
       INSTRUCTION: in integer range 0 to 31; -- current instruction
       CHAR_AT: in unsigned(4 downto 0); -- position of char to be written
       OUTPUT_BUFFER: out std_logic_vector(7 downto 0) -- data sent to LCD
   );

end MapChar;
```

Constraints file:

```
NET "DISABLE_STRATA_FLASH" LOC = "D16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;

NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_RS" LOC = " L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;

NET "SF_D<0>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_D<1>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_D<2>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_D<3>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;

# Clock
NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;

# LEDs
NET "NEGATIVE" LOC = "E9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "ZERO" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LEDS<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LEDS<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
```

```
NET "LEDS<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LEDS<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LEDS<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
```

3)Explicação do código

a) CPU

1-Clock mais devagar de aproximadamente 1 segundo, para que possamos ver as instruções serem iteradas:

```
slowClockUpdate: process (CLK)
begin
    if rising_edge(CLK) then
        slow_clk_counter <= slow_clk_counter + 1;
    end if;
end process slowClockUpdate;
slow_clk <= slow_clk_counter(CLOCK_COUNT_BUFFER_SIZE - 1);
```

2-Os estados da CPU. Em fetch definimos a instrução a ser executada e atualizamos o contador do programa e endereço para a próxima instrução. Em decode, se necessário para aquela instrução, pegamos o endereço contido na memória, que será usado no estado Execute. Caso a instrução requeira o endereço contido na posição seguinte da ram, precisamos fazer duas atualizações do contador do programa e endereço na memória, assim fazemos isso nas instruções necessárias no estado execute.

```
type cpuState is (start, fetch, decode, execute);
```

```
update: process (slow_clk)
begin
    if rising_edge(slow_clk) then
        case state_reg is
            when start =>
                we <= '0';
                address <= 0;
                state_reg <= fetch;
            when fetch =>
                reg_IR <= dataOut;
                programCounter <= programCounter + 1;
                address <= address + 1;
                state_reg <= decode;
                INSTRUCTION <= to_integer(unsigned(dataOut));
            when decode =>
```

```
                    case reg_IR is
                        when mov_a_from_end =>
                            reg_end <= to_integer(unsigned(dataOut));
                            address <= to_integer(unsigned(dataOut));

                        when mov_end_from_a =>
                            we <= '1';
                            address <= to_integer(unsigned(dataOut));
                            dataIn <= reg_A;

                        when mov_a_from_b =>
                            NULL;

                        when mov_b_from_a =>
                            NULL;

                          when add_a_to_b | sub_a_to_b | and_a_b | or_a_b |
xor_a_b | not_a | nand_a_b | inc_a | inc_b | dec_a | dec_b =>
                            opcode <= reg_IR;

                        when jz_end =>
                            reg_end <= to_integer(unsigned(dataOut));

                        when jn_end =>
                            reg_end <= to_integer(unsigned(dataOut));

                        when jmp_end =>
                            reg_end <= to_integer(unsigned(dataOut));

                        when halt =>
                            NULL;

                        when others =>
                            NULL;
                    end case;

                    state_reg <= execute;
```

```
            when execute =>
                case reg_IR is
                    when mov_a_from_end =>
                        reg_A <= dataOut;
                        address <= programCounter + 1;
                        programCounter <= programCounter + 1;

                    when mov_end_from_a =>
                        we <= '0';
                        address <= programCounter + 1;
                        programCounter <= programCounter + 1;

                    when mov_a_from_b =>
                        reg_A <= reg_B;

                    when mov_b_from_a =>
                        reg_B <= reg_A;

                      when add_a_to_b | sub_a_to_b | and_a_b | or_a_b |
xor_a_b | not_a | nand_a_b | inc_a | dec_a =>
                        reg_A <= result;

                    when inc_b | dec_b =>
                        reg_B <= result;

                    when jz_end =>
                        if ZERO_READ = '1' then
                            address <= reg_end;
                            programCounter <= reg_end;
                        else
                            address <= programCounter + 1;
                            programCounter <= programCounter + 1;
                        end if;

                    when jn_end =>
                        if NEGATIVE_READ = '1' then
```

```
                            address <= reg_end;
                            programCounter <= reg_end;
                        else
                            address <= programCounter + 1;
                            programCounter <= programCounter + 1;
                        end if;

                    when jmp_end =>
                        address <= reg_end;
                        programCounter <= reg_end;

                    when halt =>
                        NULL;

                    when others =>
                        NULL;
                end case;

                if reg_IR /= halt then
                    state_reg <= fetch;
                else
                    state_reg <= execute;
                end if;
            end case;
        end if;
    end process update;
```

```
                if reg_IR /= halt then
                    state_reg <= fetch;
                else
                    state_reg <= execute;
```

Neste código acima, verificamos se estamos no estado halt pois caso estejamos, não sairemos do estado execute:

```
        when halt =>
                        NULL;
```

Caso contrario voltamos ao estado fetch.

b) ALU

Ela executa assincronamente todas as instruções lógicas dependendo se o código da operação corrente, colocando o resultado no output result.

```
--Retorna resultado de operação dependendo do código desta
   with opcode select inResult <=
       std_logic_vector(signed(reg_A) + signed(reg_B)) when "00101",
       std_logic_vector(signed(reg_A) - signed(reg_B)) when "00110",
       std_logic_vector(signed(reg_A) + 1) when "10000",
       std_logic_vector(signed(reg_A) - 1) when "10010",
       std_logic_vector(signed(reg_B) + 1) when "10001",
       std_logic_vector(signed(reg_B) - 1) when "10011",
       reg_A and reg_B when "00111",
       reg_A or reg_B when "01000",
       reg_A nand reg_B when "01011",
       reg_A xor reg_B when "01001",
       not reg_A when "01010",
       "00000" when others;

   -- Zero
   zero <=
       '0' when reset = '1' else
       '1' when inResult = std_logic_vector(to_unsigned(0,
inResult'length))
       else '0';

   -- Negativo
   negative <=
       '0' when reset = '1'
       else inResult(inResult'high);

   result <= inResult;
```

c) LCD

Imprime a instrução corrente

```
type state is (
       -- Initial State
       initial,
```

```
        -- Write
        write_1, write_2, write_3, write_4,

        -- Wait
        wait_1, wait_2, wait_3, wait_4,

        -- Function Set
        function_set_1, function_set_2, function_set_3, function_set_4,
            function_wait_long, wait_function_set_operation_time,

        -- Entry Set
        entry_set_1, entry_set_2, entry_set_3, entry_set_4,
            entry_wait_long, wait_entry_set_operation_time,

        -- Display On/Off
        display_on_off_1, display_on_off_2, display_on_off_3,
display_on_off_4,
            display_on_off_wait_long, wait_display_on_off_operation_time,

        -- Clear Display
        clear_display_1, clear_display_2, clear_display_3, clear_display_4,
            clear_display_wait_long, wait_clear_display_operation_time,

        -- Set DD Ram
        set_dd_ram_1_1, set_dd_ram_1_2, set_dd_ram_1_3, set_dd_ram_1_4,
            set_dd_ram_1_wait_long, wait_set_dd_ram_1_operation_time,

        -- Write Character
        writingChar_1, writingChar_2, writingChar_3, writingChar_4,
writingChar_5, writingChar_6,

        -- Idle State
        idle);
```

Flag para indicar que mudou a instrução, limpando o display caso seja o caso para que ele possa ser novamente escrito.

```
    onInstructionChange: process (INSTRUCTION)
    begin
```

```
        stateChanged <= '1';
    end process onInstructionChange;
```

Escrevemos 11 caracteres antes de ir para o estado idle, saindo dele apenas quando mudar a instrução corrente.
Nesse codigo fazemos a todos os estados para conversa com o protocolo do LCD, usando o count tick para sincronização.
Nos estados writting_char 1 até 6, escrevemos o caractere.

```
    combinatorial: process (CLK)
    begin
        if (falling_edge(CLK)) then
            LCD_RW <= '0';
            LCD_RS <= '0';
            case state_reg is
                -- Initialization steps
                when initial =>
                    if (count_tick = '1') then
                        state_next <= write_1;
                    end if;
                when write_1 =>
                    LCD_E <= '1';
                    data_buffer <= "0011";
                    count <= to_unsigned(12,N);
                    if (count_tick = '1') then
                        state_next <= wait_1;
                        LCD_E <= '0';
                    end if;
                when wait_1 =>
                    count <= to_unsigned(205000,N);
                    if (count_tick = '1') then
                        state_next <= write_2;
                    end if;
                when write_2 =>
                    LCD_E <= '1';
                    data_buffer <= "0011";
                    count <= to_unsigned(12,N);
                    if (count_tick = '1') then
                        state_next <= wait_3;
```

```
                LCD_E <= '0';
            end if;
        when wait_2 =>
            count <= to_unsigned(5000,N);
            if (count_tick = '1') then
                state_next <= write_3;
            end if;
        when write_3 =>
            LCD_E <= '1';
            data_buffer <= "0011";
            count <= to_unsigned(12,N);
            if (count_tick = '1') then
                state_next <= wait_3;
                LCD_E <= '0';
            end if;
        when wait_3 =>
            count <= to_unsigned(2000,N);
            if (count_tick = '1') then
                state_next <= write_4;
            end if;
        when write_4 =>
            LCD_E <= '1';
            data_buffer <= "0010";
            count <= to_unsigned(12,N);
            if (count_tick = '1') then
                state_next <= wait_4;
                LCD_E <= '0';
            end if;
        when wait_4 =>
            count <= to_unsigned(2000,N);
            if (count_tick = '1') then
                state_next <= function_set_1;
            end if;

        -- Function Set Steps
        when function_set_1 =>
            data_buffer <= "0010";
```

```
                    count <= to_unsigned(10,N); -- wait for signal
stabilization
                    if (count_tick = '1') then
                        state_next <= function_set_2;
                    end if;
                when function_set_2 =>
                    LCD_E <= '1';
                    count <= to_unsigned(120,N);
                    if (count_tick = '1') then
                        state_next <= function_wait_long;
                        LCD_E <= '0';
                    end if;
                when function_wait_long =>
                    count <= to_unsigned(50,N);
                    if (count_tick = '1') then
                        state_next <= function_set_3;
                    end if;
                when function_set_3 =>
                    data_buffer <= "1000";
                    count <= to_unsigned(10,N);
                    if (count_tick = '1') then
                        state_next <= function_set_4;
                    end if;
                when function_set_4 =>
                    LCD_E <= '1';
                    count <= to_unsigned(120,N);
                    if (count_tick = '1') then
                        state_next <= wait_function_set_operation_time;
                        LCD_E <= '0';
                    end if;
                when wait_function_set_operation_time =>
                    count <= to_unsigned(2500,N);
                        if (count_tick = '1') then
                            state_next <= entry_set_1;
                        end if;

            -- Entry Set Steps
```

```
            when entry_set_1 =>
                data_buffer <= "0000";
                count <= to_unsigned(10,N);
                if (count_tick = '1') then
                    state_next <= entry_set_2;
                end if;
            when entry_set_2 =>
                LCD_E <= '1';
                count <= to_unsigned(120,N);
                if (count_tick = '1') then
                    state_next <= entry_wait_long;
                    LCD_E <= '0';
                end if;
            when entry_wait_long =>
                count <= to_unsigned(50,N);
                if (count_tick = '1') then
                    state_next <= entry_set_3;
                end if;
            when entry_set_3 =>
                data_buffer <= "0110";
                count <= to_unsigned(10,N);
                if (count_tick = '1') then
                    state_next <= entry_set_4;
                end if;
            when entry_set_4 =>
                LCD_E <= '1';
                count <= to_unsigned(120,N);
                if (count_tick = '1') then
                    state_next <= wait_entry_set_operation_time;
                    LCD_E <= '0';
                end if;
            when wait_entry_set_operation_time =>
                count <= to_unsigned(2500,N);
                    if (count_tick = '1') then
                        state_next <= display_on_off_1;
                    end if;
```

```
            -- Display On/Off Steps
                when display_on_off_1 =>
                    data_buffer <= "0000";
                    count <= to_unsigned(10,N);
                    if (count_tick = '1') then
                        state_next <= display_on_off_2;
                    end if;
                when display_on_off_2 =>
                    LCD_E <= '1';
                    count <= to_unsigned(120,N);
                    if (count_tick = '1') then
                        state_next <= display_on_off_wait_long;
                        LCD_E <= '0';
                    end if;
                when display_on_off_wait_long =>
                    count <= to_unsigned(50,N);
                    if (count_tick = '1') then
                        state_next <= display_on_off_3;
                    end if;
                when display_on_off_3 =>
                    data_buffer <= "1111";
                    count <= to_unsigned(10,N);
                    if (count_tick = '1') then
                        state_next <= display_on_off_4;
                    end if;
                when display_on_off_4 =>
                    LCD_E <= '1';
                    count <= to_unsigned(120,N);
                    if (count_tick = '1') then
                        state_next <= wait_display_on_off_operation_time;
                        LCD_E <= '0';
                    end if;
                when wait_display_on_off_operation_time =>
                    count <= to_unsigned(2500,N);
                        if (count_tick = '1') then
                            state_next <= clear_display_1;
                        end if;
```

```
                -- Clear Display
                when clear_display_1 =>
                    data_buffer <= "0000";
                    count <= to_unsigned(10,N);
                    if (count_tick = '1') then
                        state_next <= clear_display_2;
                    end if;
                when clear_display_2 =>
                    LCD_E <= '1';
                    count <= to_unsigned(120,N);
                    if (count_tick = '1') then
                        state_next <= clear_display_wait_long;
                        LCD_E <= '0';
                    end if;
                when clear_display_wait_long =>
                    count <= to_unsigned(50,N);
                    if (count_tick = '1') then
                        state_next <= clear_display_3;
                    end if;
                when clear_display_3 =>
                    data_buffer <= "0001";
                    count <= to_unsigned(10,N);
                    if (count_tick = '1') then
                        state_next <= clear_display_4;
                    end if;
                when clear_display_4 =>
                    LCD_E <= '1';
                    count <= to_unsigned(120,N);
                    if (count_tick = '1') then
                        state_next <= wait_clear_display_operation_time;
                        LCD_E <= '0';
                    end if;
                when wait_clear_display_operation_time =>
                    count <= to_unsigned(85000,N);
                        if (count_tick = '1') then
                            state_next <= set_dd_ram_1_1;
                        end if;
```

```
            -- Set DD RAM #
            when set_dd_ram_1_1 =>
                data_buffer <= "1000";
                count <= to_unsigned(10,N);
                if (count_tick = '1') then
                    state_next <= set_dd_ram_1_2;
                end if;
            when set_dd_ram_1_2 =>
                LCD_E <= '1';
                count <= to_unsigned(120,N);
                if (count_tick = '1') then
                    state_next <= set_dd_ram_1_wait_long;
                    LCD_E <= '0';
                end if;
            when set_dd_ram_1_wait_long =>
                count <= to_unsigned(50,N);
                if (count_tick = '1') then
                    state_next <= set_dd_ram_1_3;
                end if;
            when set_dd_ram_1_3 =>
                data_buffer <= "0000";
                count <= to_unsigned(10,N);
                if (count_tick = '1') then
                    state_next <= set_dd_ram_1_4;
                end if;
            when set_dd_ram_1_4 =>
                LCD_E <= '1';
                count <= to_unsigned(120,N);
                if (count_tick = '1') then
                    state_next <= wait_set_dd_ram_1_operation_time;
                    LCD_E <= '0';
                end if;
            when wait_set_dd_ram_1_operation_time =>
                count <= to_unsigned(85000,N);
                    if (count_tick = '1') then
                        state_next <= writingChar_1;
```

```
                end if;
            -- Write Character
            when writingChar_1 =>
                LCD_RS <= '1';
                data_buffer <= OUTPUT_BUFFER(7 downto 4);
                count <= to_unsigned(10,N);
                if (count_tick = '1') then
                    state_next <= writingChar_2;
                end if;
            when writingChar_2 =>
                LCD_RS <= '1';
                LCD_E <= '1';
                count <= to_unsigned(120,N);
                if (count_tick = '1') then
                    state_next <= writingChar_3;
                    LCD_E <= '0';
                end if;
            when writingChar_3 =>
                LCD_RS <= '1';
                count <= to_unsigned(50,N);
                if (count_tick = '1') then
                    state_next <= writingChar_4;
                end if;
            when writingChar_4 =>
                LCD_RS <= '1';
                data_buffer <= OUTPUT_BUFFER(3 downto 0);
                count <= to_unsigned(10,N);
                if (count_tick = '1') then
                    state_next <= writingChar_5;
                end if;
            when writingChar_5 =>
                LCD_RS <= '1';
                LCD_E <= '1';
                count <= to_unsigned(120,N);
                if (count_tick = '1') then
                    state_next <= writingChar_6;
                    LCD_E <= '0';
```

```
                end if;
            when writingChar_6 =>
                count <= to_unsigned(85000,N);
                    if (count_tick = '1') then
                        if CHAR_AT = 11 then
                            CHAR_AT <= to_unsigned(0, CHAR_AT'length);
                            state_next <= idle;
                        else
                            CHAR_AT <= CHAR_AT + 1;
                            state_next <= writingChar_1;

                        end if;
                    end if;
            when idle =>
                if (stateChanged = '1') then
                    stateChanged <= '0';
                    state_next <= clear_display_1;
                end if;
        end case;
    end if;
end process combinatorial;
```

d) MapChar

Modulo responsável por mapear codigo de LCD de cada caractere que será usado no modulo LCD.

```
-- Characters
constant alpha_a_uc : std_logic_vector(7 downto 0) := "01000001"; -- A
constant alpha_b_uc : std_logic_vector(7 downto 0) := "01000010"; -- B
constant alpha_c_uc : std_logic_vector(7 downto 0) := "01000011"; -- C
constant alpha_d_uc : std_logic_vector(7 downto 0) := "01000100"; -- D
constant alpha_d_lc : std_logic_vector(7 downto 0) := "01100100"; -- d
constant alpha_e_uc : std_logic_vector(7 downto 0) := "01000101"; -- E
constant alpha_e_lc : std_logic_vector(7 downto 0) := "01100101"; -- e
constant alpha_h_uc : std_logic_vector(7 downto 0) := "01001000"; -- H
constant alpha_i_uc : std_logic_vector(7 downto 0) := "01001001"; -- I
constant alpha_j_uc : std_logic_vector(7 downto 0) := "01001010"; -- J
constant alpha_l_uc : std_logic_vector(7 downto 0) := "01001100"; -- L
constant alpha_m_uc : std_logic_vector(7 downto 0) := "01001101"; -- M
```

```
   constant alpha_n_uc : std_logic_vector(7 downto 0) := "01001110"; -- N
   constant alpha_n_lc : std_logic_vector(7 downto 0) := "01101110"; -- n
   constant alpha_o_uc : std_logic_vector(7 downto 0) := "01001111"; -- O
   constant alpha_p_uc : std_logic_vector(7 downto 0) := "01010000"; -- P
   constant alpha_r_uc : std_logic_vector(7 downto 0) := "01010010"; -- R
   constant alpha_s_uc : std_logic_vector(7 downto 0) := "01010011"; -- S
   constant alpha_t_uc : std_logic_vector(7 downto 0) := "01010100"; -- T
   constant alpha_u_uc : std_logic_vector(7 downto 0) := "01010101"; -- U
   constant alpha_v_uc : std_logic_vector(7 downto 0) := "01010110"; -- V
   constant alpha_x_uc : std_logic_vector(7 downto 0) := "01011000"; -- X
   constant alpha_z_uc : std_logic_vector(7 downto 0) := "01011010"; -- Z

   -- Symbols
   constant symbol_comma           :std_logic_vector(7 downto 0) :=
"00101100"; -- ,
   constant symbol_bracket_open    :std_logic_vector(7 downto 0) :=
"01011011"; -- [
   constant symbol_bracket_close   :std_logic_vector(7 downto 0) :=
"01011101"; -- ]
   constant symbol_space           :std_logic_vector(7 downto 0) :=
"00100000"; -- (space)
```

Abaixo o exemplo do que será escrito em uma instrução exemplo, os 11 caracteres mapeados.

```
   constant mov_a_from_end : string := (
       0 => alpha_m_uc(7 downto 0),
       1 => alpha_o_uc(7 downto 0),
       2 => alpha_v_uc(7 downto 0),
       3 => symbol_space(7 downto 0),
       4 => alpha_a_uc(7 downto 0),
       5 => symbol_comma(7 downto 0),
       6 => symbol_space(7 downto 0),
       7 => symbol_bracket_open(7 downto 0),
       8 => alpha_e_lc(7 downto 0),
       9 => alpha_n_lc(7 downto 0),
       10 => alpha_d_lc(7 downto 0),
       11 => symbol_bracket_close(7 downto 0)
   );
```

Abaixo o exemplo do que será escrito em uma instrução exemplo, os 11 caracteres mapeados.

e) RAM

Abaixo o código da ram usado para teste

```
  type ram_type is array (0 to 31) of std_logic_vector(datain'range);
  signal ram : ram_type := (
       0 =>    "00001", -- MOV A, [END]
       1 =>        "10011", -- 19
       2 =>        "00100", -- MOV B, A
       3 =>        "00001", -- MOV A, [END]
       4 =>        "10010", -- 18
       5 =>        "00110", -- SUB A, B
       6 =>        "00010", -- MOV end, A
       7 =>        "10010", -- 18
       8 =>        "01101", -- JN
       9 =>        "10001", -- 17
       10 =>   "00001", -- MOV A, [END]
       11 =>   "11110", -- 30
       12 =>   "10000",    --  INC A
       13 =>   "00010", -- MOV end, A
       14 =>   "11110", -- 30
       15 =>   "01111", -- JMP
       16 =>   "00011", -- 3
       17 =>   "01110", -- HALT
       18 =>   "01010",    -- 10
       19 =>   "00010", -- 2
       others=> (others => '0')
   );
  signal read_address : integer range 0 to 31 := 0;

begin

 RamProc: process(clk) is
 begin
   if falling_edge(clk) then
     if we = '1' then
       ram(address) <= datain;
```

```
    end if;
    read_address <= address;
  end if;
end process RamProc;

dataout <= ram(read_address);
dataAt30 <= ram(30);
```

No processo acima, verificamos se está habilitado escrita, usando o dataIn para tal, caso contrário, sabemos que estamos num estado de leitura, e enviamos o dado para dataOut, assim como enviamos para os leds, o dado da posição 30 com a porta dataAt30.