Componentes em VHDL kit MERCURIO IV

Este documento apresenta uma descrição dos controladores implementados em VHDL para os componentes do kit MERCURIO IV. Os controladores implementados foram:

1. Matriz LEDs;
2. Display de 7 segmentos;
3. Display LCD;
4. Teclado;
5. RS-232;
6. Conversor AD;
7. Conversor DA.
8. Saída VGA ;
9. Sensor de Temperatura;
10. USB Device.

1) Matriz de LEDs

O controlador da matriz de LEDs implementa o controlador da matriz de LEDs multiplexada 5x8, sendo o valor das linhas comum a todas as colunas. Como entradas o componente apresenta um sinal de clock, um reset (rst), que impõem a todas as saídas o nível lógico baixo, e uma entrada de índice. O índice é utilizado para selecionar qual o caractere deve ser mostrado na matriz de LEDs. Todos os caracteres utilizados precisam ser configurados anteriormente no arquivo matriz_leds_pkg.vhd, em que os LEDs acesos são representados por nível lógico alto.

As saídas podem ser ligadas diretamente aos pinos ligados às linhas e colunas da matriz (out_col corresponde as colunas e out_row corresponde as linhas).

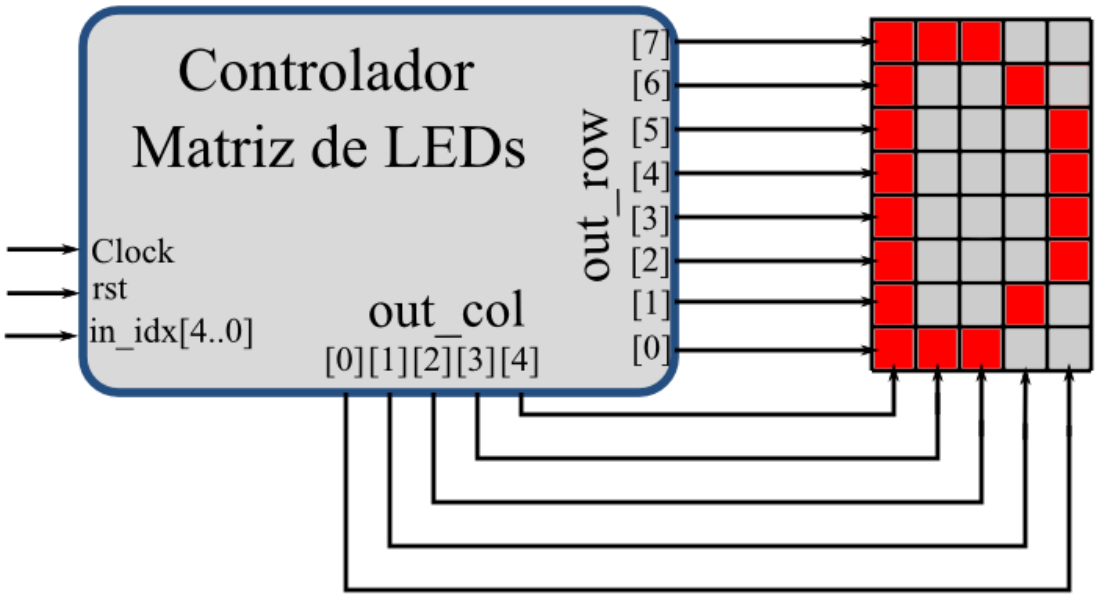

Figura 1: Diagrama do controlador e da matriz de LEDs

A frequência de multiplexação pode ser configurada na instanciação do componente através do parâmetro (*generic*) PREESCALER. O componente pode ser declarado e instanciado da seguinte forma:

Declaração do Componente em VHDL

```
component matriz_leds is
generic(
        PRESCALER: integer
);
port(
        clock   :   in std_logic;
        rst     :   in std_logic;
        in_idx  :   in std_logic_vector (3 downto 0);
        out_col :   out std_logic_vector (4 downto 0);
        out_row :   out std_logic_vector (7 downto 0)
);
end component;
```

Instanciação do Componente em VHDL

```
inst_matriz_de_leds: matriz_leds
port map (
        clock   => CLOCK_50Mhz,
        rst     => reset,
        in_idx  => in_idx,
        out_col => LEDM_C,
        out_row => LEDM_R
    );
```

1.1) Matriz de LEDs Package

O arquivo matriz_leds_pkg.vhd contém o mapeamento dos caracteres que serão mostrados na matriz e a declaração de uma ROM, que será utilizada para armazenar os caracteres que poderão ser escritos. A posição de um caractere na ROM determina qual o índice (in_idx) que é necessário passar para o controlador para que o caractere seja escrito. Como in_idx possui apenas 4 bits, pode-se armazenar até 16 caracteres.

Os caracteres armazenados podem ser alterados/adicionados alterando matriz_leds_pkg.vhd. Para adicionar um novo caractere, basta criar uma nova constante do tipo char (definido no pacote), sendo cada grupo de 8 bits corresponde a uma coluna e o bit mais significativo é atribuído ao LED mais acima, por exemplo, o caractere 2:

		Linhas							
		8	7	6	5	4	3	2	1
Colunas	1	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**
	2	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**
	3	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**1**
	4	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**
	5	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**

```
constant char_2: char := char'(
-- número 2
        "11000010",
        "10100001",
        "10010001",
        "10001001",
        "10000110"
    );
```

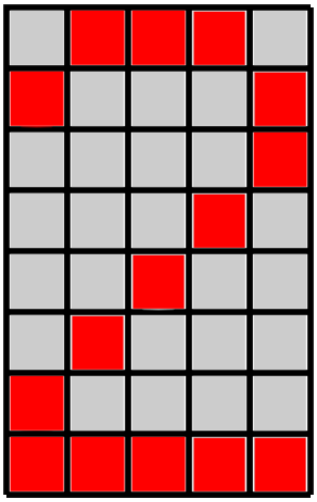

Figura 2: Montagem de um dado para inclusão na matriz_leds_pkg.vhd.

Para que o caractere fique acessível pelo controlador, é necessário aumentar o tamanho da ROM e acrescentá-lo a ela. A alteração do tamanho da ROM é feito no inicio do programa, através da constante NUM_CHAR. Para acrescentar o caractere a ROM, basta adicionar o nome da constante à declaração da ROM:

Declaração da ROM

```
type ROM is array (0 to NUM_CHAR-1) of char;
constant msg:  ROM := ROM'(
    char_0,
    char_1,
    char_2,
    char_3,
    char_4,
    char_5,
    char_6,
    char_7,
    char_8,
    char_9,
    char_null
);
```

O índice equivalente a cada caractere nada mais é do que a posição na memória em que ele está gravado, que vai de 0 até NUM_CHAR-1, seguindo a sequencia de declaração (char_0→in_idx = 0).

2) Display de 7 Segmentos

O controlador de display de 7 segmentos recebe um dado de quatro bits, através da interface entrada, e decodifica o dado, indicando quais os segmentos estarão acesos e quais estarão apagados de forma a apresentar o algarismo hexadecimal (0 a F) correspondente no display, pela interface saida.

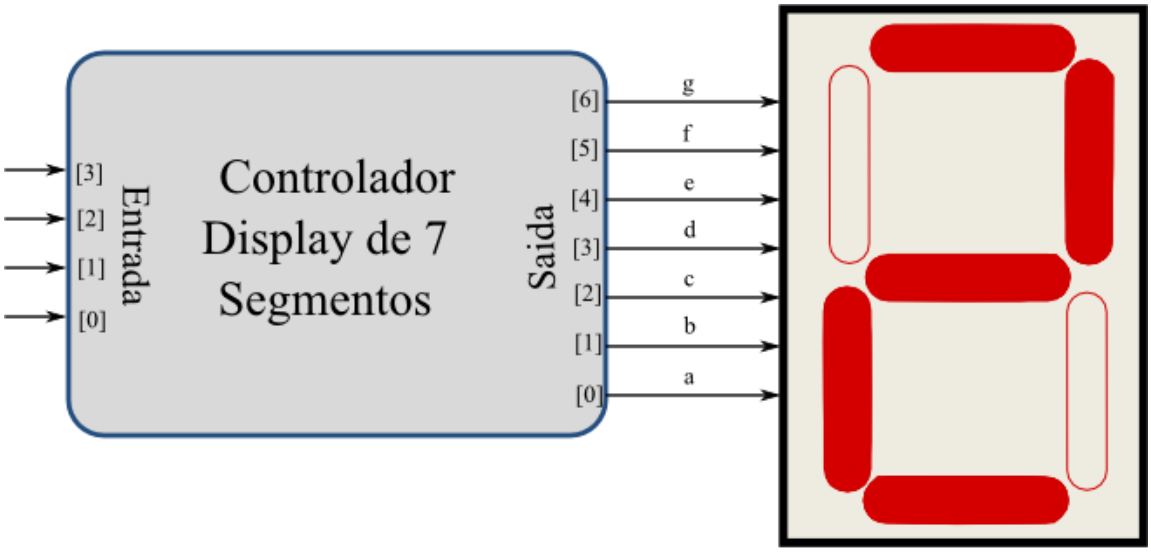

Figura 3: Diagrama do controlador de display de 7 segmentos conectado ao display.

A declaração e instanciação do componente podem ser feitas da seguinte forma:

Declaração do Componente em VHDL

```
component display_7seg is
    port(
        entrada :   in std_logic_vector(3 downto 0);
        saida   :   out std_logic_vector(6 downto 0)
    );
end component;
```

Instanciação do Componente em VHDL

```
Inst_display_7seg:
    display_7seg
    port map
    (
        entrada => entrada,
        saida   => DISP0_D(6 downto 0)
    );
```

3) Display LCD

O controlador de display LCD é destinado ao controle de um display 16x2. O componente possui diversas entradas e saídas:

Interface	Direção	Função
CLOCK_50	Entrada	Sinal de clock do componente. O clock padrão é 50MHz, a mudança no clock implica em mudança nos parâmetros (*generics*)
rst	Entrada	Sinal de reset do componente. Durante o reset não é escrito nada no LCD.
Start_i	Entrada	Sinal para o inicio da escrita no LCD (nível lógico alto).
Data_i	Entrada	Dado que será escrito no LCD em codificação ASCII.
Idx_o	Saída	Índice indicando qual o caractere que será escrito (0, 1, 2, ...).
Ready_o	Saída	Indica que a escrita do LCD foi terminada.
LCD_DATA	Bidirecional	Via pela qual o componente se comunica com o LCD. Deve ser conectado diretamente ao LCD.
LCD_RS	Saída	Quando em ‘0’ recebe comandos (clear screen, posição do cursor) e quando está em ‘1’ recebe dados para serem escritos. Deve ser conectado diretamente ao LCD.
LCD_RW	Saída	Sinal de leitura e escrita. Deve ser conectado diretamente ao LCD.
LCD_EN	Saída	Sinal de habilitação do LCD. Deve ser conectado diretamente ao LCD.
LCD_BLON	Saída	Sinal do Backlight. Deve ser conectado diretamente ao LCD.

Os sinais com o prefixo LCD, se referem a sinais destinados a comunicação direta com o display LCD. Os demais sinais devem ser fornecidos pelo sistema que utilizará o componente.

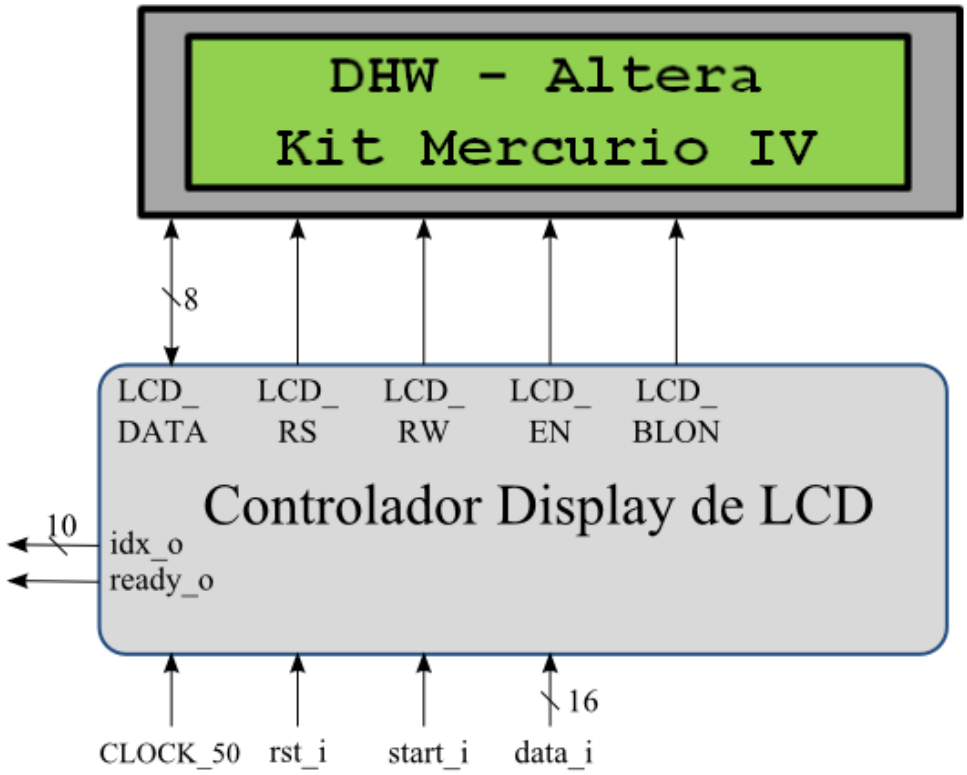

Figura 4: Diagrama de ligação entre o controlador e o display

Além dos sinais de entrada e saída, o controlador pode ser configurado para outros tipos de display ou clock. Os parâmetros presentes são:

Parâmetro	Função
TAM_DATA	Tamanho do dado de entrada.
NUM_DATA	Quantidade de caracteres a serem escritos.
DIV_NUM	Utilizada para a divisão do clock interno.
BOOT_TIME	Define o tempo de inicialização, com base na quantidade de ciclos.
WR_TIME	Define o tempo de escrita, com base na quantidade de ciclos.
CLR_TIME	Define o tempo de clear, com base na quantidade de ciclos.

A declaração e instanciação do componente podem ser feitas da seguinte forma:

Declaração do Componente em VHDL

```
COMPONENT lcd_top
    GENERIC (
        TAM_DATA  : INTEGER;
        NUM_DATA  : INTEGER;
        DIV_NUM   : INTEGER;
        BOOT_TIME : INTEGER;
        WR_TIME   : INTEGER;
        CLR_TIME  : INTEGER );
    PORT
    (
        CLOCK_50    :    IN STD_LOGIC;
        rst_i       :    IN STD_LOGIC;
        start_i     :    IN STD_LOGIC;
        data_i      :    IN STD_LOGIC_VECTOR(tam_data DOWNTO 1);
        idx_o       :    OUT INTEGER range 1 to NUM_DATA;
        ready_o     :    OUT STD_LOGIC;
        LCD_DATA    :    INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        LCD_RS      :    OUT STD_LOGIC;
        LCD_RW      :    OUT STD_LOGIC;
        LCD_EN      :    OUT STD_LOGIC;
        LCD_BLON    :    OUT STD_LOGIC
    );
END COMPONENT;
```

Instanciação do Componente em VHDL

```
   inst_lcd_top : lcd_top
   generic map(
        TAM_DATA    => TAM_DATA,
        NUM_DATA    => TAM_RAM,
        --LCD
        DIV_NUM     => DIV_NUM,
        BOOT_TIME   => BOOT_TIME,
        WR_TIME     => WR_TIME,
        CLR_TIME    => CLR_TIME
     )
    port map (
        CLOCK_50    => CLOCK_50,
        rst_i       => rst_i,
        start_i     => start_i,
        data_i      => data_i,
        idx_o       => idx_o,
        ready_o     => ready_lcd,
        -- LCD
        LCD_DATA    => LCD_D,
        LCD_RS      => LCD_RS,
        LCD_RW      => LCD_RW,
        LCD_EN      => LCD_EN,
        LCD_BLON    => LCD_BLON
    );
```

Os dados que devem ser escritos no LCD podem ser armazenados em uma memória RAM, preferencialmente com pelo menos 32 endereços (tamanho do LCD, 16x2) e inicializa-la com o valor hexadecimal 20 (caractere nulo em ASCII), e definir o parâmetro TAM_DATA = 32. Dessa forma, o sinal idx_o do controlador pode ser utilizado como endereço da memória, selecionando

qual dado deve ser enviado para o LCD. O valor da memória pode ser atualizado durante o funcionamento do sistema. Além disso, para que o LCD atualize o que está escrito, é necessário dar um pulso de nível lógico alto no sinal start_i para que o controlador reescreva os dados no LCD.

4) Teclado

O controlador do teclado recebe o valor das 12 teclas do teclado, através da interface push_buttons, e responde com o valor de 0 a 11 (em binário), na saída Key sendo o valor 10 (Ah) correspondente ao "*" e o 11 (Bh) ao "#". O teclado também da prioridade para teclas com menor valor e possui um sinal ready que indica quando existe um dado estável na saída, ou seja, depois que o tempo de debounce passar. O tempo de debounce pode ser configurado através de um generic DEBOUNCE presente no componente.

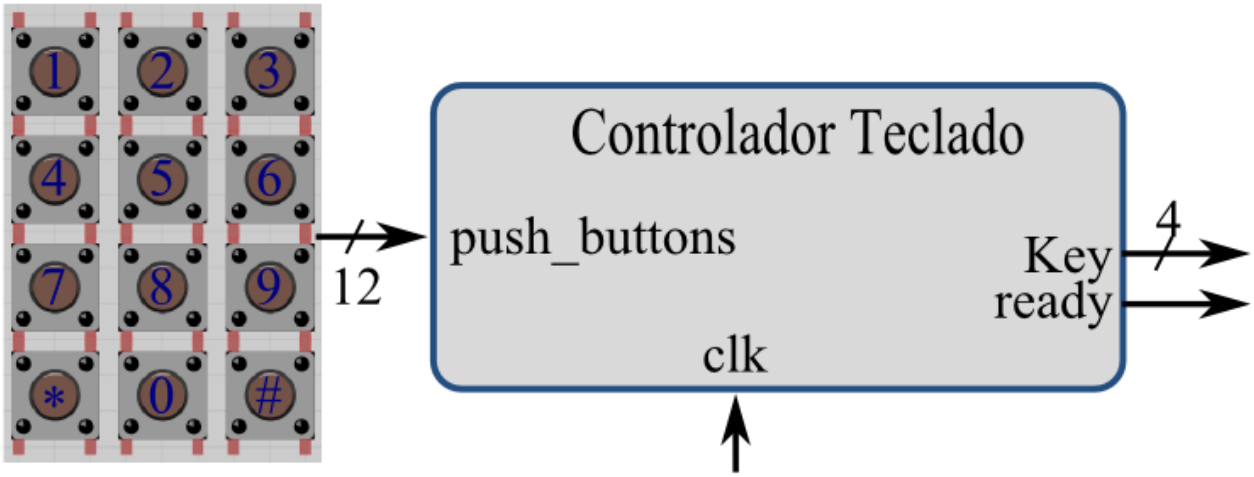

Figura 5: Diagrama da ligação entre o teclado e o controlador.

A declaração e instanciação do componente pode ser feita da seguinte forma:

Declaração do Componente em VHDL

```
component teclado_base is
generic(
    DEBOUNCE : integer
```

```
);
port(
        clk          :   in std_logic;
        push_button  :   in std_logic_vector (11 downto 0);
        key          :   out std_logic_vector (3 downto 0);
        ready            :   out std_logic
    );
end component;
```

Instanciação do Componente em VHDL

```
teclado:
    teclado_base
    generic map(
    DEBOUNCE => DEBOUNCE
    )
    port map
    (
        clk              => CLOCK_50MHz,
        push_button      => KEY,
        key              => key_data,
        ready            => ready_key
    );
```

5) RS-232 (UART)

O componente da interface UART implementa a comunicação no padrão RS-232. O componentes pode receber um dado de 8-bits para ser enviado, interface serial_write, sendo que a requisição de envio é feita por um pulso positivo no sinal start_tx. A recepção de dados é iniciada com um pulso em zero na interface serial_rx, seguindo o padrão da comunicação, e ao final da recepção de 8-bits, o componente apresenta o dado na interface serial_read. Os sinais busy_tx e busy_rx indicam que o componente está fazendo uma escrita e/ou leitura, respectivamente.

A interface valid_rx indica que o dado recebido é valido, caso o dado recebido apresente algum erro, a saída apresentará todos os bits em zero. As interfaces serial_rx e serial_tx são as interfaces por onde a comunicação serial será feita, sendo a serial_rx a interface de recepção de dados e serial_tx a interface de envio. O componente possui duas entradas de clock, clock_uart comanda a comunicação (envio e transmissão) e o clock_sample é utilizado para verificar se existe um novo dado chegando e pode ter uma frequência menor que a frequência do clock_uart.

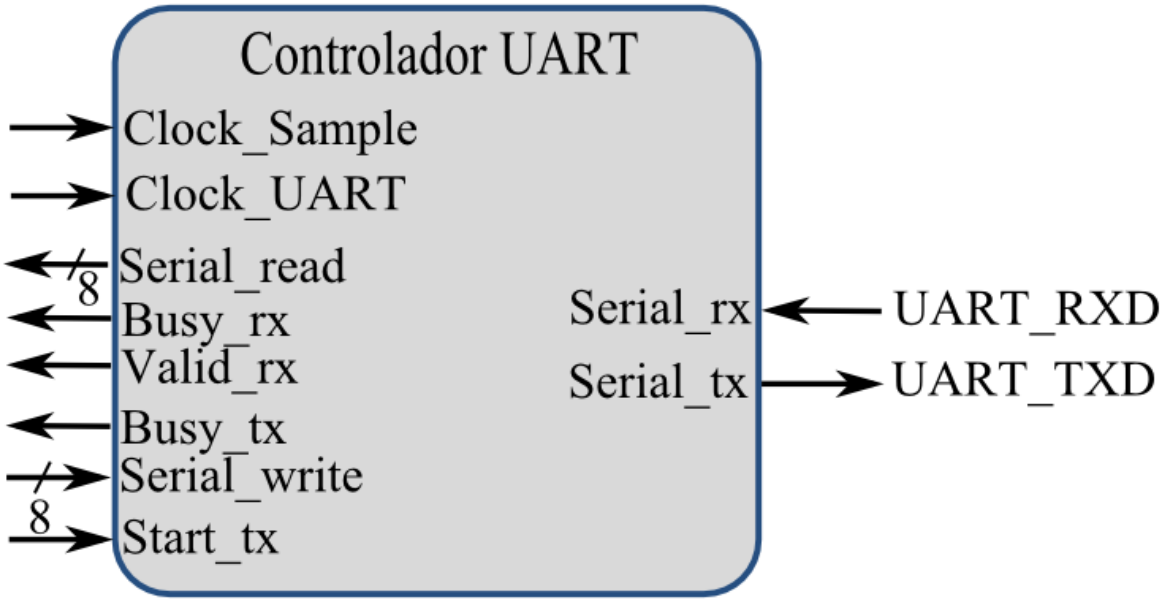

Figura 6: Componente controlador UART.

A declaração instanciação do componentes podem ser feitas da seguinte forma:

Declaração do Componente em VHDL

```
COMPONENT uart
    GENERIC (
    CLOCK_FREQUENCY : INTEGER := 50000000;  -- Frequência CLOCK_UART
    BAUDRATE_RX : INTEGER := 115200;     -- taxa de recepção
    BAUDRATE_TX : INTEGER := 115200 );  -- taxa de transmissão
    PORT
    (
        CLOCK_SAMPLE     :     IN STD_LOGIC;
        CLOCK_UART       :     IN STD_LOGIC;
        serial_rx        :     IN STD_LOGIC;
        start_tx         :     IN STD_LOGIC;
        serial_write     :     IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        busy_rx          :     OUT STD_LOGIC;
        busy_tx          :     OUT STD_LOGIC;
        valid_rx         :     OUT STD_LOGIC;
        serial_tx        :     OUT STD_LOGIC;
        serial_read      :     OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END COMPONENT;
```

Instanciação do Componente em VHDL

```
uart_inst: uart
    GENERIC MAP (
    CLOCK_FREQUENCY => CLOCK_FREQUENCY,
    BAUDRATE_RX => BAUDRATE_RX,
    BAUDRATE_TX => BAUDRATE_TX  )
    PORT MAP
    (
        CLOCK_SAMPLE    =>   CLOCK_SAMPLE,
        CLOCK_UART      =>   CLOCK_UART,
        serial_rx       =>   UART_RXD,
        start_tx        =>   send,
        serial_write    =>   send_data,
        busy_rx         =>   busy_rx,
        busy_tx         =>   busy_tx,
        valid_rx        =>   valid_rx,
        serial_tx       =>   UART_TXD,
        serial_read     =>   received_data
    );
```

6) Conversor AD

O controlador do conversor AD é um bloco que recebe alguns sinais de configuração e implementa a controle do conversor AD externo, através da comunicação SPI. O bloco possui quatro sinais de controle o csn_i, conv_i, sd_i, ub_i. O csn_i habilita o conversor quando em ‘0’, o conv_i recebe a requisição de conversão, ou seja quando em ‘1’ começa uma nova conversão caso não esteja ocupado, o sd_i indica se o conversor funcionará em modo single ou dual (1 para single e 0 para dual) e o ub_i indica se a conversão vai ser unipolar (entradas com referencia no terra) ou bipolar (entrada diferencial entre AIN1A-AIN0A e AIN1B-AINB) (0 para unipolar e 1 para bipolar). O sinal busy_o indica se o conversor está no meio de uma conversão ou está pronto para fazer uma nova conversão (1 para ocupado 0 para desocupado). A frequência máxima de conversão é 1,25MHz.

Os dados apresentados nas saídas data_ estão codificados em complemento de 2, sendo o valor 0 significa que a tensão na entrada do conversor é igual a tensão de referencia do conversor.

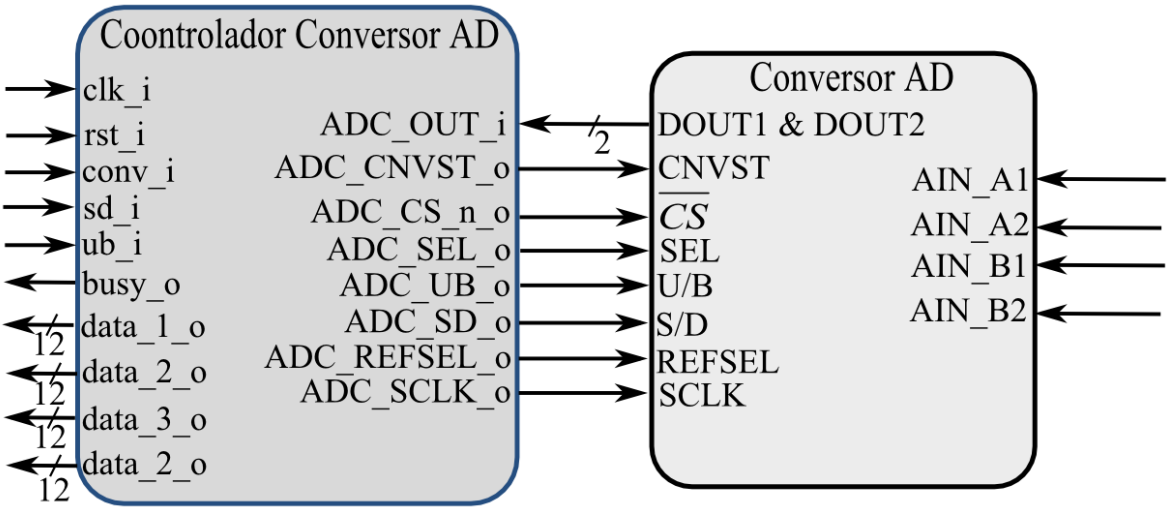

Figura 7: Diagrama do controlador de conversor AD conectado a um conversor AD.

Declaração do Componente em VHDL

```
COMPONENT conversor_AD
    GENERIC ( CLOCK_FREQUENCY : INTEGER := 50000000 );
    PORT
    (
        clk_i           :    IN STD_LOGIC;
        rst_i           :    IN STD_LOGIC;
        conv_i          :    IN STD_LOGIC;
        sd_i            :    IN STD_LOGIC;
        ub_i            :    IN STD_LOGIC;
        csn_i           :    IN STD_LOGIC;
        busy_o          :    OUT STD_LOGIC;
        data_1_o        :    OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
        data_2_o        :    OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
        data_3_o        :    OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
        data_4_o        :    OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
        ADC_DOUT_i      :    IN STD_LOGIC_VECTOR(2 DOWNTO 1);
        ADC_CNVST_o     :    OUT STD_LOGIC;
        ADC_CS_N_o      :    OUT STD_LOGIC;
        ADC_SEL_o       :    OUT STD_LOGIC;
        ADC_UB_o        :    OUT STD_LOGIC;
        ADC_SD_o        :    OUT STD_LOGIC;
        ADC_REFSEL_o    :    OUT STD_LOGIC;
        ADC_SCLK_o      :    OUT STD_LOGIC
    );
END COMPONENT;
```

Instanciação do Componente em VHDL

```
Inst_ADC: conversor_AD
    GENERIC MAP (CLOCK_FREQUENCY => CLOCK_FREQUENCY)
    PORT MAP
    (
        clk_i           =>  clk,
        rst_i           =>  rst,
        conv_i          =>  conv,
        sd_i            =>  sd,
        ub_i            =>  ub,
        csn_i           =>  csn,
        busy_o          =>  busy,
        data_1_o        =>  data1,
        data_2_o        =>  data2,
        data_3_o        =>  data3,
        data_4_o        =>  data4,
        ADC_DOUT_i      =>  ADC_OUT,
        ADC_CNVST_o     =>  ADC_CNVST,
        ADC_CS_N_o      =>  ADC_CS_N,
        ADC_SEL_o       =>  ADC_SEL,
        ADC_UB_o        =>  ADC_UB,
        ADC_SD_o        =>  ADC_SD,
        ADC_REFSEL_o    =>  ADC_REFSEL,
        ADC_SCLK_o      =>  ADC_SCLK
    );
```

7) Conversor DA

O controlador do conversor DA é um bloco que recebe alguns sinais de configuração e implementa a controle do conversor DA externo, através da comunicação SPI, utilizando o modo transparente do conversor. O bloco possui dois sinais de controle o csn_i, que seleciona o conversor quando em ‘0’, e o conv_i, que inicio o envio de um novo dado para o conversor DA quando em ‘1’ e o sinal de ocupado está em ‘0’. O sinal busy_o indica que o conversor já está no meio da comunicação com o conversor DA. As interface data_1_i e data_2_i recebem os valores que serão enviados ao conversor e convertidos em sinal analógico, sendo os valores de data_1_i apresentados no canal OUTA do DA e os valores de data_2_i no canal OUTB. A frequência máxima de conversão é de 1,25MHz.

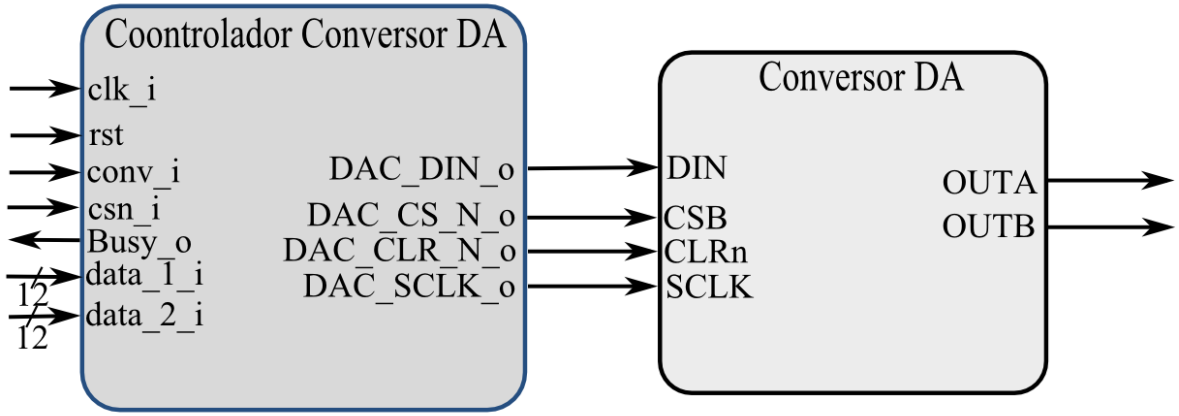

Figura 8: Diagrama do controlador de conversor DA conectado a um conversor DA.

Declaração do Componente em VHDL

```
COMPONENT conversor_DA
    GENERIC ( CLOCK_FREQUENCY : INTEGER := 50000000 );
    PORT
    (
        clk_i           :    IN STD_LOGIC;
        rst_i           :    IN STD_LOGIC;
        conv_i          :    IN STD_LOGIC;
        csn_i           :    IN STD_LOGIC;
        data_1_i        :    IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        data_2_i        :    IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        busy_o          :    OUT STD_LOGIC;
        DAC_CLR_N_o     :    OUT STD_LOGIC;
        DAC_CS_N_o      :    OUT STD_LOGIC;
        DAC_DIN_o       :    OUT STD_LOGIC;
        DAC_SCLK_o      :    OUT STD_LOGIC
    );
END COMPONENT;
```

Instanciação do Componente em VHDL

```
Inst_DAC: conversor_DA
    GENERIC MAP (CLOCK_FREQUENCY => CLOCK_FREQUENCY)
    PORT MAP
    (
        clk_i       => clk,
        rst_i       => rst,
        conv_i      => clock_sine,
        csn_i       => '0',
        data_1_i    => dado_da,
        data_2_i    => dado_da,
        busy_o      => open,
        DAC_CLR_N_o => DAC_CLR_N,
        DAC_CS_N_o  =>  DAC_CS_N,
        DAC_DIN_o   => DAC_DIN,
        DAC_SCLK_o  =>  DAC_SCLK
    );
```

8) VGA

O controlador VGA implementa os sinais sincronismo de um monitor VGA, utilizando a resolução 640x480. Como entradas ele apresenta o clock de 50MHz e o resetn (ativo em nível lógico baixo). As saídas que são destinadas ao VGA são VGA_HS (sincronismo horizontal) e VGA_VS (sincronismo vertical). A saída Video_on indica que o monitor está ligado, ou seja, está escrevendo algo na tela. Esse componente só responsável para a sincronização como monitor VGA, para a exibição de dados, é preciso que as saídas RGB da placa estejam com os valores desejados e como a interface VGA opera na frequência de 25,125MHz, é necessário mandar os valores de um novo pixel com um frequência de aproximadamente 25MHz. A saída Linha indica em qual linha o monitor VGA está (0 – 480) e a saída pixel indica qual o pixel daquela linha (0 – 680).

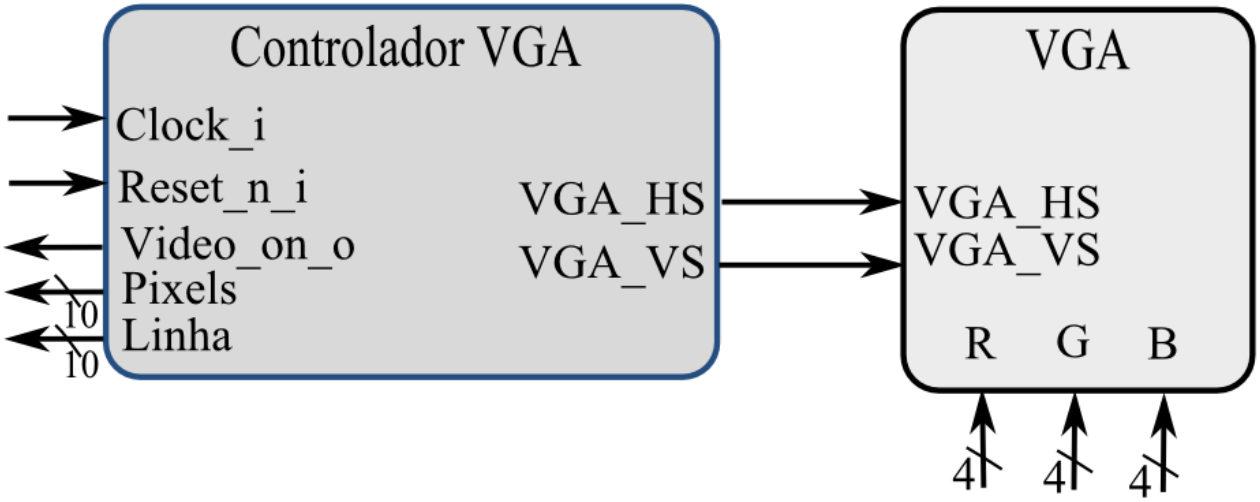

Figura 9: Controlador VGA

Declaração do Componente em VHDL

```
component VGA_controlador
    PORT
        (
                Clock_i, Reset_n_i  :   in std_logic;
                Video_on_o          :   out std_logic;
                Pixels_o,Linha_o    :   out std_logic_vector(10 downto 0);
                VGA_HS_o,VGA_VS_o   :   out std_logic
        );
end component VGA_controlador;
```

Instanciação do Componente em VHDL

```
VGA_controlador_inst : VGA_controlador port map (
```

```
            clock_i     => CLOCK_50MHz,
            Reset_n_i   => resetn,
            Video_on_o  => Video_on,
            Pixels_o    => Pixel,
            Linha_o     => Linha,
            VGA_HS_o    => VGA_HS,
            VGA_VS_o    => VGA_VS
    );
```

9) Sensor de Temperatura

O modulo sensor de temperatura implementa a comunicação no padrão I2C destinada a escrita e obtenção de dados do sensor de temperatura. As entradas existentes são o clock_i, que deve ser menor que 400kHz (frequência máxima do sensor); o start_write_i, que solicita uma ação do modulo; o i2c_address_i, que apresenta nos 7 bits mais significativos o endereço do sensor (o padrão de placa é 1001000) e o bit menos significativo indica se será feito uma leitura ou escrita (1 para leitura e 0 para escrita); o i2c_write_i recebe os dados que devem ser escritos em um dos registradores do sensor, sendo os 4 bits menos significativos reservados para o endereço do registrador a ser acessado. A porta bidirecional i2c_sda é a via de dados da comunicação e deve ser ligada diretamente na porta sda do sensor de temperatura. A saída i2c_scl determina o clock da comunicação, a i2c_busy indica se o modulo está ocupado ou não e a i2c_read apresenta os dados lidos do sensor.

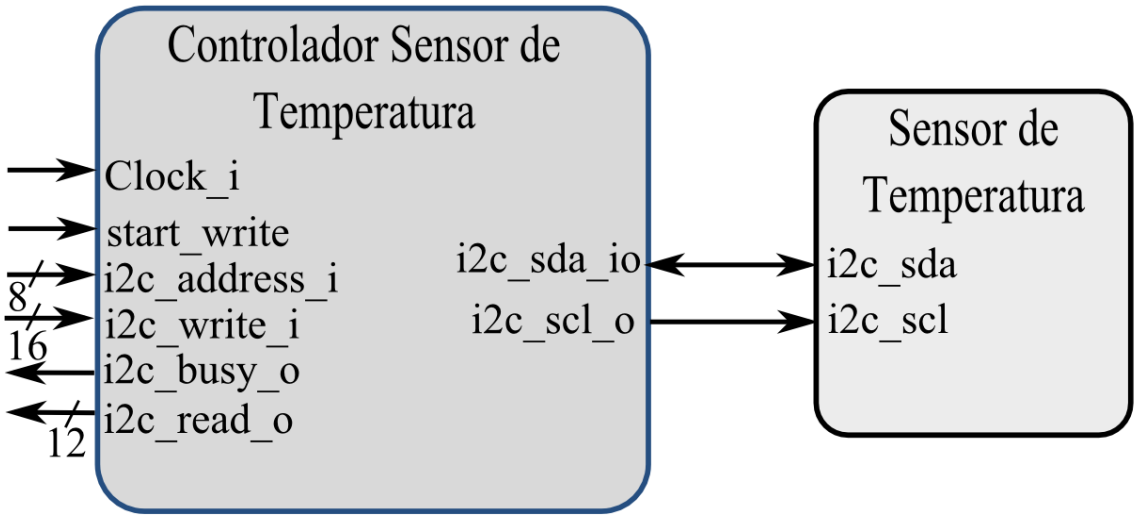

Figura 10: Componente controlador do sensor de temperatura.

Declaração do Componente em VHDL

```
COMPONENT i2c_temperatura
    PORT
    (
        clock_i          :    IN STD_LOGIC;
        start_write_i    :    IN STD_LOGIC;
        i2c_address_i    :    IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        i2c_write_i      :    IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        i2c_sda_io       :    INOUT STD_LOGIC;
        i2c_scl_o        :    OUT STD_LOGIC;
        i2c_busy_o       :    OUT STD_LOGIC;
        i2c_read_o       :    OUT STD_LOGIC_VECTOR(11 DOWNTO 0)
    );
END COMPONENT;
```

Instanciação do Componente em VHDL

```
    inst_sensor: i2c_temperatura
    PORT MAP
    (
        clock_i          => i2c_clk,
        start_write_i    => start_write,
        i2c_address_i    => i2c_address,
        i2c_write_i      => i2c_write,
        i2c_sda_io       => i2c_SDA,
        i2c_scl_o        => i2c_SCL,
        i2c_busy_o       => i2c_busy,
        i2c_read_o       => i2c_read
    );
```

10) USB Device

O módulo USB Device implementa a comunicação entre o FPGA e o FT245, que executa a comunicação no padrão USB. Por padrão o componente fica recebendo os dados do FT245 e apresenta na saída dado_rd_o o ultimo valor lido. Para identificar um novo dado, a saída rd_ready_o apresenta nível lógico durante a recepção, ou seja, cada novo dado é precedido de um pulso em nível lógico baixo. A requisição de escrita é feita impondo nível lógico alto na entrada wr_i, desde que wr_ready_o esteja em nível lógico alto e não existir nenhuma leitura a ser feita. Durante a escrita, a saída wr_ready_o apresenta nível lógico baixo. As interfaces com o prefixo USB destinam-se a conexão com o FT245.

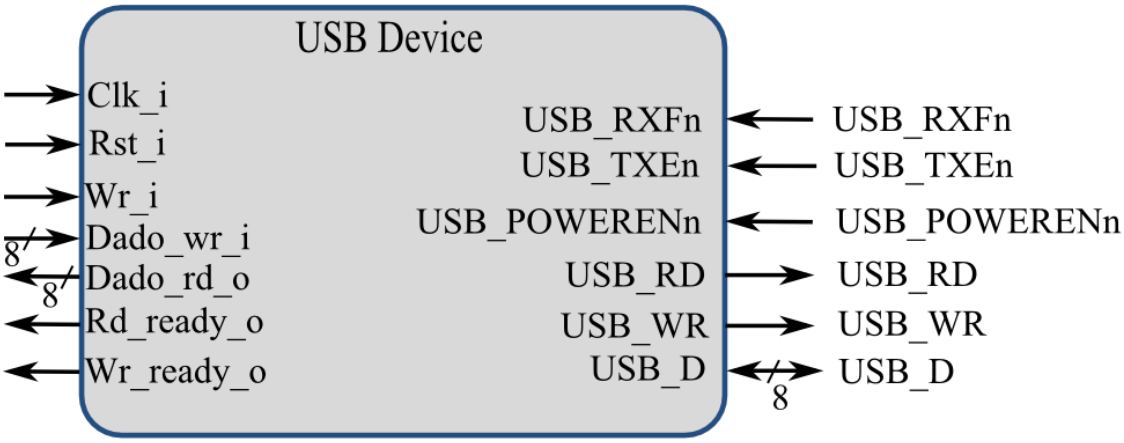

Figura 11: Diagrama do componente USB Device

Declaração do Componente em VHDL

```
COMPONENT USB_Device
    GENERIC ( CLOCK_PERIOD : INTEGER := 20 );
    PORT
    (
        clk_i           :    IN STD_LOGIC;
        rst_i           :    IN STD_LOGIC;
        wr_i            :    IN STD_LOGIC;
        dado_wr_i       :    IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        dado_rd_o       :    OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        rd_ready_o      :    OUT STD_LOGIC;
        wr_ready_o      :    OUT STD_LOGIC;
        USB_RXFn        :    IN STD_LOGIC;
        USB_TXEn        :    IN STD_LOGIC;
        USB_POWERENn    :    IN STD_LOGIC;
        USB_RD          :    OUT STD_LOGIC;
        USB_WR          :    OUT STD_LOGIC;
        USB_D           :    INOUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END COMPONENT;
```

Instanciação do Componente

```
inst_USB_Device: USB_Device
    GENERIC MAP ( CLOCK_PERIOD => CLOCK_PERIOD)
    PORT MAP
    (
        clk_i          => CLOCK_50MHz,
        rst_i          => rst,
        wr_i           => wr,
        dado_wr_i      => dado_wr,
        dado_rd_o      => dado_rd,
        rd_ready_o     => rd_ready,
        wr_ready_o     => wr_ready,
```

```
        USB_RXFn        => USB_RXFn,
        USB_TXEn        => USB_TXEn,
        USB_POWERENn    => USB_POWERENn,
        USB_RD          => USB_RD,
        USB_WR          => USB_WR,
        USB_D           => USB_D
    );
```