O módulo LCD Parallel é constituído por dois blocos principais:

i) O registo RegLow;

ii) o registo RegHigh. Para enviar data (8bits) para o LCD, o módulo LCD Parallel implementa um modo de escrita High to Low, onde escreve os 4bits de maior peso no registo Low antes de os passar para o registo High e receber os 4bits de menor peso.

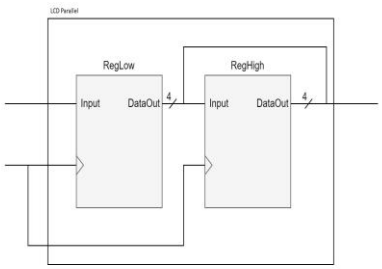

Figura 1 – Diagrama de blocos do módulo *LCD Parallel*

RegLow

O bloco RegLow recebe os valores de 4bits fornecidos pelo USBPort e vai passá-los para um sinal interno que liga tanto à entrada do bloco RegHigh como aos 4bits de menor peso do output.

RegHigh

O bloco RegHigh recebe os valores que lhe são disponibilizados pelo bloco RegLow e vai passá-los para os 4 bits de maior peso do output.

Interface com o *Control*

Implementou-se o módulo LCD em *software*, recorrendo a linguagem Kotlin.
A informação é passada para o hardware através do USBPort.
A função que escreve para o LCD em paralelo estabelece o rs em primeiro lugar (para definir se a informação é data ou comando) e de seguida envia os valores de menor peso através de uma máscara (0x0F), após essa operação, a informação sofre um shift right de 4 bits e é enviada novamente com a mesma máscara.

A classe *HAL* desenvolvida é descrita na secção 3.1, e o código fonte desenvolvido no Anexo **Error! Reference source not found.**.

Classe *HAL*

A classe HAL usa 3 funções para envio de informação, seguintes:

1 – writeBits, onde recebe uma máscara e um valor e só faz update dos valores definidos na máscara como 1, copiando os respetivos valores binários do valor dado.

2 – setBits, onde recebe uma máscara e faz update de todos os valores definidos na máscara para 1.

3 – clrBits, onde recebe uma máscara e faz update de todos os valores definidos na máscara para 0.

1 Conclusões

O módulo LCD Parallel tem como função estabelecer o envio de informação do software para o LCD hardware em paralelo.

A limitação da transferência da data é controlada pelo clock.

A. Descrição VHDL do bloco *Registo*

```
library ieee;
use ieee.std_logic_1164.all;

entity Registo is
port
	(
	A: in std_logic_vector(3 downto 0);
	Clk: in std_logic;
	Reset: in std_logic;
	E: in std_logic;
	S: out std_logic_vector(3 downto 0)
	);
end Registo;

architecture structure of Registo is
component FFD is
port
	(
	CLK: in std_logic;
	RESET: in STD_LOGIC;
	SET: in std_logic;
	D: in STD_LOGIC;
	EN: in STD_LOGIC;
	Q: out std_logic
	);
end component;

begin

	FFD1: FFD port map (CLK => Clk, RESET => Reset, SET => '0', D => A(0), EN => E, Q => S(0));
	FFD2: FFD port map (CLK => Clk, RESET => Reset, SET => '0', D => A(1), EN => E, Q => S(1));
	FFD3: FFD port map (CLK => Clk, RESET => Reset, SET => '0', D => A(2), EN => E, Q => S(2));
	FFD4: FFD port map (CLK => Clk, RESET => Reset, SET => '0', D => A(3), EN => E, Q => S(3));

end structure;
```

B. Descrição VHDL do bloco LCD Parallel

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity LCDPArallel is
port(
	DataIn: in std_logic_vector(7 downto 0);
	rst: in std_logic;
	clk: in std_logic;
	DataOut: out std_logic_vector(7 downto 0);
	RS: out std_logic;
	E: out std_logic

);
end LCDParallel;

architecture structural of LCDParallel is
component UsbPort IS
	PORT
	(
		inputPort:  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
		outputPort :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)
	);
END component;

component Registo is
port
	(
	A: in std_logic_vector(3 downto 0);
	Clk: in std_logic;
	Reset: in std_logic;
	E: in std_logic;
	S: out std_logic_vector(3 downto 0)
	);
end component;

signal enableReg: std_logic;
signal Sclk: std_logic;
signal SRegH: std_logic_vector(3 downto 0);
signal SRegL: std_logic_vector(3 downto 0);

begin


usb: UsbPort port map (
	inputPort => DataIn,
	outputPort(6) => RS,
	outputPort(5) => E,
	outputPort(4) => Sclk,
	outputPort(0) => SRegH(0),
	outputPort(1) => SRegH(1),
	outputPort(2) => SRegH(2),
	outputPort(3) => SRegH(3)
);
```

```
RegHigh: Registo port map(
        A => SRegH,
        Reset => rst,
        Clk => clk,
        E => Sclk,
        S => SRegL
);

RegLow: Registo port map(
        A => SRegL,
        Reset => rst,
        Clk => clk,
        E => Sclk,
        S(0) => DataOut(0),
        S(1) => DataOut(1),
        S(2) => DataOut(2),
        S(3) => DataOut(3)
);


        DataOut(4) <= SRegL(0);
        DataOut(5) <= SRegL(1);
        DataOut(6) <= SRegL(2);
        DataOut(7) <= SRegL(3);

end structural;
```

C. Atribuição de pinos do módulo LCD Parallel

```
set_global_assignment -name BOARD "MAX 10 DE10 - Lite"
set_global_assignment -name DEVICE 10M50DAF484C6GES
set_global_assignment -name FAMILY "MAX 10"



set_location_assignment PIN_C10 -to rst
set_location_assignment PIN_C11 -to kack
set_location_assignment PIN_P11 -to clk



set_location_assignment PIN_A10 -to kval
set_location_assignment PIN_E14 -to K[0]
set_location_assignment PIN_D14 -to K[1]
set_location_assignment PIN_A11 -to K[2]
set_location_assignment PIN_B11 -to K[3]

set_location_assignment PIN_W5 -to I[0]
set_location_assignment PIN_AA14 -to I[1]
set_location_assignment PIN_W12 -to I[2]
set_location_assignment PIN_AB12 -to I[3]
set_location_assignment PIN_AB11 -to O[0]
set_location_assignment PIN_AB10 -to O[1]
set_location_assignment PIN_AA9 -to O[2]
set_location_assignment PIN_AA8 -to O[3]
```

D. Código Kotlin – LCD

```
import HAL

object LCD { // Escreve no LCD usando a interface a 4 bits.
    // Dimensão do display.
    private const val LINES = 2
    private const val COLS = 16
    private val maskDlow = 0x0F
    private val maskDhigh = 0xF0
    private val maskRS = 0x40
    private val maskE = 0x20
    private val maskClk = 0x10
    private val parallelmask = 0x1F
    private class pos(val line: Int, val column: Int)

    // Escreve um byte de comando/dados no LCD em paralelo
    private fun writeByteParallel(rs: Boolean, data: Int) {
        if (rs) {
            HAL.setBits(maskRS)
        } else {
            HAL.clrBits(maskRS)
        }
        Thread.sleep(1)
        HAL.clrBits(maskClk)
        Thread.sleep(1)
        //write high
        var byte = data shr 4
        HAL.writeBits(0x0F, byte)

        HAL.setBits(maskClk)
        Thread.sleep(1)
        HAL.clrBits(maskClk)
        Thread.sleep(1)

        //write low
        byte = data.and(maskDlow)
        HAL.writeBits(0x0F, byte)

        HAL.setBits(maskClk)
        Thread.sleep(1)
        HAL.clrBits(maskClk)

        Thread.sleep(1)
        HAL.setBits(maskE)
        Thread.sleep(1)
        HAL.clrBits(maskE)
        Thread.sleep(1)

    }

    // Escreve um byte de comando/dados no LCD em série
    private fun writeByteSerial(rs: Boolean, data: Int) {
        TODO()
    }
```

```
    // Escreve um byte de comando/dados no LCD
    private fun writeByte(rs: Boolean, data: Int) {
        writeByteParallel(rs, data)
    }

    // Escreve um comando no LCD
     fun writeCMD(data: Int) {
        writeByte(false, data)
    }

    // Escreve um dado no LCD
    private fun writeDATA(data: Int) {
        writeByte(true, data)
    }

    // Envia a sequência de iniciação para comunicação a 4 bits.
    fun init() {
        Thread.sleep(16)
        writeCMD(0x30)
        Thread.sleep(5)
        writeCMD(0x30)
        Thread.sleep(2)

        writeCMD(0x30)
        Thread.sleep(1)
        writeCMD(0x38)
        writeCMD(0x08)
        writeCMD(0x01)
        writeCMD(0x06)
        writeCMD(0x0F)

    }

    // Escreve um caráter na posição corrente.
    fun write(c: Char) {
        writeDATA(c.code)
    }

    // Escreve uma string na posição corrente.
    fun write(text: String) {
        for(i in text)
            write(i)
    }

    // Envia comando para posicionar cursor (‘line’:0..LINES-1 , ‘column’:0..COLS-1)
    fun cursor(line: Int, column: Int) {
        val l = line * 64
        val pos = (l + column).or(0x80) //cmd DDRAM
        writeCMD( pos )
    }

    // Envia comando para limpar o ecrã e posicionar o cursor em (0,0)
    fun clear() {
        writeCMD(0x01)
    }
}
```

```
fun main() {
    HAL.init()
    LCD.init()
    KBD.init()
    /*

    LCD.write("Hello")
    LCD.cursor(1, 3)
    LCD.write("LIC")
    LCD.clear()
     */
  testKBD_LCD()

}

fun testKBD_LCD(){

    var c = 0
    while (true){

            val key = KBD.waitKey(1000)
            if (key != 0.toChar()){
                LCD.write(key)
                c++
            }
            if (c == 15){
                LCD.clear()
                c = 0
            }
    }

}

import HAL

object LCD { // Escreve no LCD usando a interface a 4 bits.
    // Dimensão do display.
    private const val LINES = 2
    private const val COLS = 16
    private val maskDlow = 0x0F
    private val maskDhigh = 0xF0
    private val maskRS = 0x40
    private val maskE = 0x20
    private val maskClk = 0x10
    private val parallelmask = 0x1F
    private class pos(val line: Int, val column: Int)

    // Escreve um byte de comando/dados no LCD em paralelo
    private fun writeByteParallel(rs: Boolean, data: Int) {
        if (rs) {
            HAL.setBits(maskRS)
        } else {
            HAL.clrBits(maskRS)
        }

        Thread.sleep(1)
        HAL.clrBits(maskClk)
```

```
    Thread.sleep(1)
    //write high
    var byte = data shr 4
    HAL.writeBits(0x0F, byte)

    HAL.setBits(maskClk)
    Thread.sleep(1)
    HAL.clrBits(maskClk)
    Thread.sleep(1)

    //write low
    byte = data.and(maskDlow)
    HAL.writeBits(0x0F, byte)

    HAL.setBits(maskClk)
    Thread.sleep(1)
    HAL.clrBits(maskClk)

    Thread.sleep(1)
    HAL.setBits(maskE)
    Thread.sleep(1)
    HAL.clrBits(maskE)
    Thread.sleep(1)

}

// Escreve um byte de comando/dados no LCD em série
private fun writeByteSerial(rs: Boolean, data: Int) {
    TODO()
}

// Escreve um byte de comando/dados no LCD
private fun writeByte(rs: Boolean, data: Int) {
    writeByteParallel(rs, data)
}

// Escreve um comando no LCD
 fun writeCMD(data: Int) {
    writeByte(false, data)
}

// Escreve um dado no LCD
private fun writeDATA(data: Int) {
    writeByte(true, data)
}

// Envia a sequência de iniciação para comunicação a 4 bits.
fun init() {
    Thread.sleep(16)
    writeCMD(0x30)
    Thread.sleep(5)
    writeCMD(0x30)
    Thread.sleep(2)

    writeCMD(0x30)
    Thread.sleep(1)
    writeCMD(0x38)
```

```
        writeCMD(0x08)
        writeCMD(0x01)
        writeCMD(0x06)
        writeCMD(0x0F)
     }

    // Escreve um caráter na posição corrente.
    fun write(c: Char) {
        writeDATA(c.code)
    }

    // Escreve uma string na posição corrente.
    fun write(text: String) {
        for(i in text)
            write(i)
    }

    // Envia comando para posicionar cursor (‘line’:0..LINES-1 , ‘column’:0..COLS-1)
    fun cursor(line: Int, column: Int) {
        val l = line * 64
        val pos = (l + column).or(0x80) //cmd DDRAM
        writeCMD( pos )
    }

    // Envia comando para limpar o ecrã e posicionar o cursor em (0,0)
    fun clear() {
        writeCMD(0x01)
    }
}
fun main() {
    HAL.init()
    LCD.init()
    KBD.init()
    /*

    LCD.write("Hello")
    LCD.cursor(1, 3)
    LCD.write("LIC")
    LCD.clear()
     */
  testKBD_LCD()
 }

fun testKBD_LCD(){
    var c = 0
    while (true){
             val key = KBD.waitKey(1000)
            if (key != 0.toChar()){
                LCD.write(key)
                c++
            }
            if (c == 15){
                LCD.clear()
                c = 0
            }
    }
 }
```