# Trabalho 2: Jogo da Forca

**Anna Luisa de Sá dos Santos**(DRE: 121050671),
**Larissa Rocha dos Santos**(DRE: 121072380 ),
**Reginaldo da Silva Cardoso Santos**(DRE: 121039544).

17 de julho de 2024

# Sumário

# Capítulo 1

# Introdução

Ao longo deste relatório, temos o intuito de documentar o segundo trabalho desenvolvido para a disciplina de Sistemas Digitais da UFRJ.

O objetivo deste projeto foi programar um jogo da forca em uma FPGA, utilizando o display LCD da placa e um teclado externo de PS-2.

**Funcionamento**
Cada vez que o usuário acerta uma letra, esta aparece na tela, preenchendo os espaços correspondentes na palavra a ser adivinhada. Caso a letra escolhida não faça parte da palavra, o sistema decrementa o número de vidas do jogador. Esse é a mecânica de número de erros escolhida para o projeto, e este é exibido no display LCD, permitindo ao jogador acompanhar seu progresso e saber quantas tentativas incorretas ainda tem disponíveis antes de perder o jogo.

Este relatório detalha todas as etapas do desenvolvimento do projeto, desde a concepção inicial até a implementação final, incluindo a descrição dos componentes utilizados, os desafios enfrentados e as soluções encontradas. Através deste trabalho, buscamos aplicar os conhecimentos adquiridos ao longo do curso e demonstrar nossa capacidade de desenvolver soluções práticas e eficientes em sistemas digitais.

# Capítulo 2

# Implementação

Nesta seção discorreremos sobre o código em VHDL utilizado para a implementação do trabalho.

## 2.1 Teclado

Nessa seção vamos discorrer sobre os códigos referentes ao funcionamento do teclado PS-2. O arquivo principal é o kb code, mas para entendê-lo, precisamos ver antes os arquivos que ele importa.

### 2.1.1 ps2-rx

Este arquivo é responsável por estabelecer comunicação com o teclado conectado à FPGA e por entregar os dados enviados por ele. Em essência, ele desempenha o papel de receptor PS-2, capturando os sinais de dados, verificando a paridade e gerando um sinal que indica quando os dados estão prontos para serem lidos.
**Entidade:**
Assim, pegamos essas informações recebidas em um sinal de saída chamado 'dout', que é um vetor de 8 bits. Podemos ver, na definição da entidade, os sinais necessários para a operação deste módulo:

Sinais de entrada:

- clk: sinal de relógio, clock
- reset: sinal de reiniciar o sistema
- ps2d: dados do teclado

```
--================================================
-- fsmd to extract the 8-bit data
--================================================
-- registers
process (clk, reset)
begin
   if reset='1' then
      state_reg <= idle;
      n_reg  <= (others=>'0');
      b_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      n_reg <= n_next;
      b_reg <= b_next;
   end if;
end process;
```

Figura 2.1: Entidade do módulo ps2-rx

- ps2c: clock do teclado
- rx-en: sinal para habilitar a recepção, enable

Sinais de saída:

- rx-done-tick: sinal que indica quando a recepção está concluída
- dout: saída de 8 bits para os dados recebidos

**Arquitetura:**

- type statetype is (idle, dps, load): Define um tipo enumerado statetype com três estados: idle, dps, e load.
- signal state-reg, state-next: statetype: Sinais para o estado atual e o próximo estado.
- signal filter-reg, filter-next: std-logic-vector(7 downto 0): Sinais para o registro do filtro e seu próximo valor.
- signal f-ps2c-reg, f-ps2c-next: std-logic;: Sinais para o registro do filtro do clock PS/2 e seu próximo valor.
- signal b-reg, b-next: std-logic-vector(10 downto 0);: Sinais para o registro do buffer de dados e seu próximo valor.

```
port (
   clk, reset: in  std_logic;
   ps2d, ps2c: in  std_logic;  -- key data, key clock
   rx_en: in std_logic;
   rx_done_tick: out  std_logic;
   dout: out std_logic_vector(7 downto 0)
);
```

Figura 2.2: Arquitetura

- signal n-reg, n-next: unsigned(3 downto 0);: Sinais para o registro do contador e seu próximo valor.
- signal fall-edge: std-logic;: Sinal para detectar borda de descida.

**Processo de Filtragem e Detecção de Borda de Descida:**

- if reset='1' then ... elsif (clk'event and clk='1') then ... end if;: Reinicializa os registros filter-reg e f-ps2c-reg no reset, ou atualiza-os em cada ciclo de clock.
- filter-next ¡= ps2c & filter-reg(7 downto 1);: Atualiza filter-next deslocando filter-reg para a direita e adicionando ps2c na posição mais significativa.
- f-ps2c-next ¡= '1' when filter-reg="11111111"else '0' when filter-reg="00000000"else f-ps2c-reg: Atualiza f-ps2c-next para '1' se filter-reg for 11111111 e '0' se for 00000000.
- fall-edge ¡= f-ps2c-reg and (not f-ps2c-next);: Detecta uma borda de descida no sinal ps2c.

```
--================================================
-- filter and falling edge tick generation for ps2c
--================================================
process (clk, reset)
begin
   if reset='1' then
      filter_reg <= (others=>'0');
      f_ps2c_reg <= '0';
   elsif (clk'event and clk='1') then
      filter_reg <= filter_next;
      f_ps2c_reg <= f_ps2c_next;
   end if;
end process;

filter_next <= ps2c & filter_reg(7 downto 1);
f_ps2c_next <= '1' when filter_reg="11111111" else
               '0' when filter_reg="00000000" else
               f_ps2c_reg;
fall_edge <= f_ps2c_reg and (not f_ps2c_next);
```

Figura 2.3: Processo de Filtragem e Detecção de Borda de Descida

**Máquina de Estados Finita para Extrair os Dados:**

- if reset='1' then ... elsif (clk'event and clk='1') then ... end if;: Reinicializa os registros state-reg, n-reg e b-reg no reset, ou atualiza-os em cada ciclo de clock.

```
--================================================
-- fsmd to extract the 8-bit data
--================================================
-- registers
process (clk, reset)
begin
   if reset='1' then
      state_reg <= idle;
      n_reg  <= (others=>'0');
      b_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      n_reg <= n_next;
      b_reg <= b_next;
   end if;
end process;
```

Figura 2.4: FSM, process(clk,reset)

E sobre o processo de próximo estado, temos que:

- process(state-reg, n-reg, b-reg, fall-edge, rx-en, ps2d): Define um processo sensível a mudanças nos sinais state-reg, n-reg, b-reg, fall-edge, rx-en e ps2d.

- case state-reg is ... end case: Define a lógica de transição de estados: idle: Estado ocioso; dps: Estado de recepção de dados; load: Estado de carregamento final.

```
-- next-state logic
process(state_reg, n_reg, b_reg, fall_edge, rx_en, ps2d)
begin
   rx_done_tick <= '0';
   state_next <= state_reg;
   n_next <= n_reg;
   b_next <= b_reg;
   case state_reg is
      when idle =>
         if fall_edge='1' and rx_en='1' then
            -- shift in start bit
            b_next <= ps2d & b_reg(10 downto 1);
            n_next <= "1001";
            state_next <= dps;
         end if;
      when dps =>  -- 8 data + 1 parity + 1 stop
         if fall_edge='1' then
            b_next <= ps2d & b_reg(10 downto 1);
            if n_reg = 0 then
                state_next <= load;
            else
                n_next <= n_reg - 1;
            end if;
         end if;
      when load =>
         -- 1 extra clock to complete the last shift
         state_next <= idle;
         rx_done_tick <= '1';
   end case;
end process;
```

Figura 2.5: FSM, process(state-reg, n-reg, b-reg, fall-edge, rx-en, ps2d)

### 2.1.2 fifo

Este implementa uma fifo (first in first out) para armazenar dados temporariamente em uma fila. Ela guarda $2^W$ dados com B bits, sendo W e B dois inteiros definidos na entidade como 4 e 8 respectivamente.
Em detalhes, temos que:

- rd: bit de entrada que indica que desejo ler um dado da fifo
- wr: bit de entrada que indica que desejo escrever um dado na fifo
- r-data: vetor de saída com dado lido, caso rd seja ativado
- w-data: vetor de entrada com dado que desejamos escrever, caso wr seja ativado
- empty: bit de saída que indica se a fifo esta vazia
- full: bit de saída que indica se a fifo esta cheia

```
entity fifo is
   generic(
      B: natural:=8; -- number of bits
      W: natural:=4 -- number of address bits
   );
   port(
      clk, reset: in std_logic;
      rd, wr: in std_logic;
      w_data: in std_logic_vector (B-1 downto 0);
      empty, full: out std_logic;
      r_data: out std_logic_vector (B-1 downto 0)
   );
end fifo;
```

Figura 2.6: Entidade

**Arquitetura**

- type reg-file-type: Define uma matriz para armazenar os dados do FIFO.
- signal: Declara vários sinais para controlar ponteiros de leitura/escrita, estados de cheio/vazio e habilitação de escrita.

```
architecture arch of fifo is
   type reg_file_type is array (2**W-1 downto 0) of
        std_logic_vector(B-1 downto 0);
   signal array_reg: reg_file_type;
   signal w_ptr_reg, w_ptr_next, w_ptr_succ:
      std_logic_vector(W-1 downto 0);
   signal r_ptr_reg, r_ptr_next, r_ptr_succ:
      std_logic_vector(W-1 downto 0);
   signal full_reg, empty_reg, full_next, empty_next:
          std_logic;
   signal wr_op: std_logic_vector(1 downto 0);
   signal wr_en: std_logic;
begin
```

Figura 2.7: Arquitetura

**Registro de Dados**

- O processo de registro funciona de duas maneiras, dependendo do estado do sinal de reset.

- Reset Ativo: Quando o reset está ativo, todos os registros são zerados, limpando qualquer dado previamente armazenado. Se o reset não estiver ativo, a operação de escrita ocorre no flanco de subida do clock (clk).

- Flanco de Subida do Clock (clk): Se o reset não estiver ativo, no flanco de subida do clk, ocorrem as seguintes ações:
  - O dado contido em `w_data` é escrito na posição indicada por `w_ptr_reg`, desde que o sinal `wr_en` esteja ativo.

**Porta de Leitura e Habilitação de Escrita**

- Porta de Leitura: O valor lido pela porta de leitura é o dado localizado no endereço especificado pelo registrador `r_ptr_reg`. Este registrador aponta para a posição de memória correta, garantindo que a leitura seja precisa e eficiente.

- Habilitação de Escrita: O sinal de habilitação de escrita, `wr_en`, é ativado sob duas condições. Primeiramente, o sinal de escrita (wr) deve estar ativo, indicando que uma operação de escrita é desejada. Em segundo lugar, a FIFO (First In, First Out) não deve estar cheia. Somente quando ambas as condições são atendidas, o sinal `wr_en` permitirá a escrita do dado na posição de memória designada, assegurando a integridade e o correto funcionamento do sistema de armazenamento.

**Lógica de Controle da FIFO:** A lógica de controle da FIFO assegura que os ponteiros de escrita e leitura sejam atualizados corretamente e que os estados de cheio e vazio da FIFO sejam monitorados. O uso de um processo sensível ao flanco de subida do clock e ao sinal de reset garante que as operações de escrita e leitura sejam executadas de forma síncrona e que a FIFO funcione corretamente dentro do sistema em que está inserida.

- **Processo de Controle:**
  - **Reset:** Quando o sinal de reset (`reset='1'`) está ativo, todos os registradores são zerados. Isso inclui os ponteiros de escrita (`w_ptr_reg`) e leitura (`r_ptr_reg`), além dos sinais que indicam se a FIFO está cheia (`full_reg`) ou vazia (`empty_reg`).
  - **Flanco de Subida do Clock:**
    - Se o reset não estiver ativo, a atualização dos ponteiros e dos sinais de estado ocorre no flanco de subida do clock (`clk`).
    - Os ponteiros de escrita e leitura (`w_ptr_reg` e `r_ptr_reg`) recebem os valores dos próximos ponteiros calculados (`w_ptr_next` e `r_ptr_next`).
    - Os sinais `full_reg` e `empty_reg` são atualizados com os próximos valores (`full_next` e `empty_next`), que indicam se a FIFO estará cheia ou vazia após a próxima operação.
- **Ponteiros Sucessores:** Os ponteiros sucessores (`w_ptr_succ` e `r_ptr_succ` são calculados incrementando os valores atuais dos ponteiros de escrita e leitura (`w_ptr_reg`) e (`r_ptr_reg`). Esses cálculos são feitos utilizando a conversão para `std_logic_vector` dos valores incrementados.

**Lógica de Próximo Estado**

- Combinação wr e rd: Combina os sinais de escrita e leitura em um único sinal `wr_op`.

- Processo de Próximo Estado: Determina os próximos estados dos ponteiros e flags de cheio/vazio com base na operação (`wr_op`):

  – "00": Nenhuma operação.
  – "01": Leitura, avança o ponteiro de leitura se não estiver vazia.
  – "10": Escrita, avança o ponteiro de escrita se não estiver cheia.
  – Outros: Avança ambos os ponteiros para leitura/escrita simultâneas.

**Saídas:** Este código implementa uma FIFO síncrona que controla a escrita e leitura de dados de forma eficiente e ordenada.
As saídas full e empty são atualizadas com os valores dos registros internos `full_reg e empty_reg`, respectivamente. Esses registros indicam o estado atual da FIFO, ou seja, se está cheia ou vazia.

### 2.1.3 kbl-code

Módulo que recebe códigos de varredura (scan codes) do teclado PS-2 e os armazena na FIFO para serem lidos posteriormente.
**Entidade**

```
entity kb_code is
   generic(W_SIZE: integer := 2); -- 2^W_SIZE palavras na FIFO
   port (
      clk, reset: in std_logic;
      ps2d, ps2c: in std_logic;
      rd_key_code: in std_logic;
      key_code: out std_logic_vector(7 downto 0);
      kb_buf_empty: out std_logic
   );
end kb_code;
```

Figura 2.8: Entidade

- "W_SIZE": parâmetro genérico que define o tamanho do FIFO ($2^{W\_SIZE}$ palavras).
- clk, reset: sinais de relógio e reset.
- ps2d, ps2c: sinais de dados e clock do PS/2.

- rd_key_code: sinal para ler um código do FIFO.
- key_code: vetor de 8 bits que contém o código da tecla lida.
- kb_buf_empty: sinal que indica se o FIFO está vazio.

**Arquitetura**

```
architecture arch of kb_code is
   constant BRK: std_logic_vector(7 downto 0) := "11110000"; -- código de quebra (F0)
   type statetype is (wait_brk, get_code);
   signal state_reg, state_next: statetype;
   signal scan_out, w_data: std_logic_vector(7 downto 0);
   signal scan_done_tick, got_code_tick: std_logic;
```

Figura 2.9: Arquitetura

- "BRK": constante que representa o código de interrupção.
- "statetype": usado no estado da FSM (Máquina de Estados Finita).
- "state_reg, state_next": são sinais utilizados para o estado atual e o próximo da FSM.
- scan_out, w_data: são sinais de 8 bits para os dados de varredura.
- "scan_done_tick, got_code_tick": são sinais de controle usados na indicação da varredura quando a mesma está completa e no momento em que um código foi recebido.

**Máquina de Estados Finita(FSM)**
**Registro de Estado** Atualiza o estado atual ('state_reg') com o próximo

```
process(clk, reset)
begin
   if reset = '1' then
      state_reg <= wait_brk;
   elsif (clk'event and clk = '1') then
      state_reg <= state_next;
   end if;
end process;
```

Figura 2.10: FSM - Registro de Estado

estado ('state_next') na subida do 'clk' ou redefine para 'wait_brk' se 'reset' estiver ativo.

**Lógica do Próximo Estado**

Define a transição de estados e os sinais de controle baseados no estado atual e nas entradas:

```
process(state_reg, scan_done_tick, scan_out)
begin
   got_code_tick <= '0';
   state_next <= state_reg;
   case state_reg is
      when wait_brk => -- espera pelo código de quebra (F0)
         if scan_done_tick = '1' and scan_out = BRK then
            state_next <= get_code;
         end if;
      when get_code => -- pega o próximo código de varredura
         if scan_done_tick = '1' then
            got_code_tick <= '1';
            state_next <= wait_brk;
         end if;
   end case;
end process;
```

Figura 2.11: FSM - Registro de Estado

- "wait_brk": Espera pelo código de quebra (F0). Se "scan_done_tick"estiver ativo e scan_out for igual a BRK, transita para get_code.
- "get_code": Captura o próximo código de varredura. Se scan_done_tick estiver ativo, ativa got_code_tick e retorna para wait_brk.

Portanto, este módulo kb_code é projetado para capturar os códigos de teclas enviados e armazená-los na FIFO. A FSM garante que o módulo só armazena códigos de teclas que seguem um código de quebra (F0), indicando que uma tecla foi liberada. A FIFO gerencia os dados armazenados e facilita a leitura assíncrona desses códigos.

### 2.1.4 key2ascii

Este código VHDL serve como uma ponte entre linguagens, traduzindo códigos de teclas do teclado PS2 para o formato ASCII, permitindo a interação do teclado PS2 com o software. O componente mapeia os códigos de teclas recebidos do teclado PS2 para os respectivos códigos ASCII.

- **key_code** é um sinal de entrada que representa o código da tecla pressionada, sendo um vetor lógico de 8 bits (std_logic_vector).
- **ascii_code** é um sinal de saída que representa o código ASCII correspondente à tecla pressionada, também sendo um vetor lógico de 8 bits.

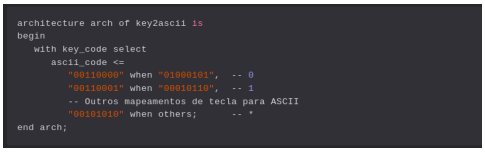


Figura 2.12: Enter Caption

- O mapeamento entre **key_code** e **ascii_code** é realizado utilizando a estrutura `with ... select`. Essa estrutura seleciona um valor baseado no valor de **key_code** e atribui o valor correspondente a **ascii_code**.

- As linhas **library ieee;**, **use ieee.std_logic_1164.all;**, e **use ieee.numeric_std.all;** especificam que o código utiliza as bibliotecas padrão do IEEE para lógica digital e aritmética numérica.

- **entity key2ascii is** define uma entidade chamada `key2ascii`, que contém os portos de entrada e saída do módulo:

  - **key_code: in std_logic_vector(7 downto 0);** declara um porto de entrada chamado **key_code** como um vetor lógico de 8 bits (7 downto 0), representando o código da tecla PS2.
  - **ascii_code: out std_logic_vector(7 downto 0);** declara um porto de saída chamado **ascii_code** como um vetor lógico de 8 bits (7 downto 0), representando o código ASCII correspondente à tecla pressionada.

  **architecture arch of key2ascii is** define a arquitetura denominada `arch` para a entidade `key2ascii`. O bloco de código começa com **begin**.

- A instrução **with key_code select** realiza a seleção baseada no valor do sinal **key_code**. **ascii_code ¡=** atribui um valor ao sinal **ascii_code** de acordo com a entrada **key_code**.

- Por exemplo, **"00110000"when "01000101"** indica que, se **key_code** for igual a "01000101", o valor "00110000"(código ASCII para o número 0) é atribuído a **ascii_code**.

- Se nenhuma das condições anteriores for satisfeita, a cláusula **others** atribui "00101010"(código ASCII para '*') a **ascii_code**.

## 2.2 Display LCD

### 2.2.1 Máquina de estados para o controle do LCD

O LCD é uma máquina de estados que através de cada um dos seus estados ele percorre um ponteiro definido como um sinal através de cada uma das posições do display, de modo que cada um dos seus estados são responsáveis por ações específicas.

Antes de qualquer coisa, ao definirmos as variáveis de estado utilizando, em VHDL, `type ___ is`, teremos os estados onde nosso código deve percorrer, a sua ordem será definida a seguir, pois em cada processo teremos a liberdade de apontar qual o próximo estado em que nosso código deve seguir, utilizando blocos condicionais como IF, case, ou até mesmo sem condicional, apenas indicando qual o próximo estado.

É importante também que seja definido um sinal à nossa variável de estado definida anteriormente, pois queremos percorrer pelos estados, então um sinal é essencial para que haja esta atualização para o próximo estado.

Primeiramente, temos um processo que inicializa nossa máquina de estados, indicando o que acontece quando o resetamos (reset assíncrono) e qual o primeiro estado em que ele deve se direcionar.

Segundamente é importante citar que possuimos mais um `type`, sendo este definido como `LCD_CMDS_T` que é um array, onde através do mesmo, que é atribuido ao `LCD_CMDS`, escreveremos utilizando o sinal `lcd_cmd_ptr`, portanto é importante definimos até onde queremos escrever. Figura : 2.13

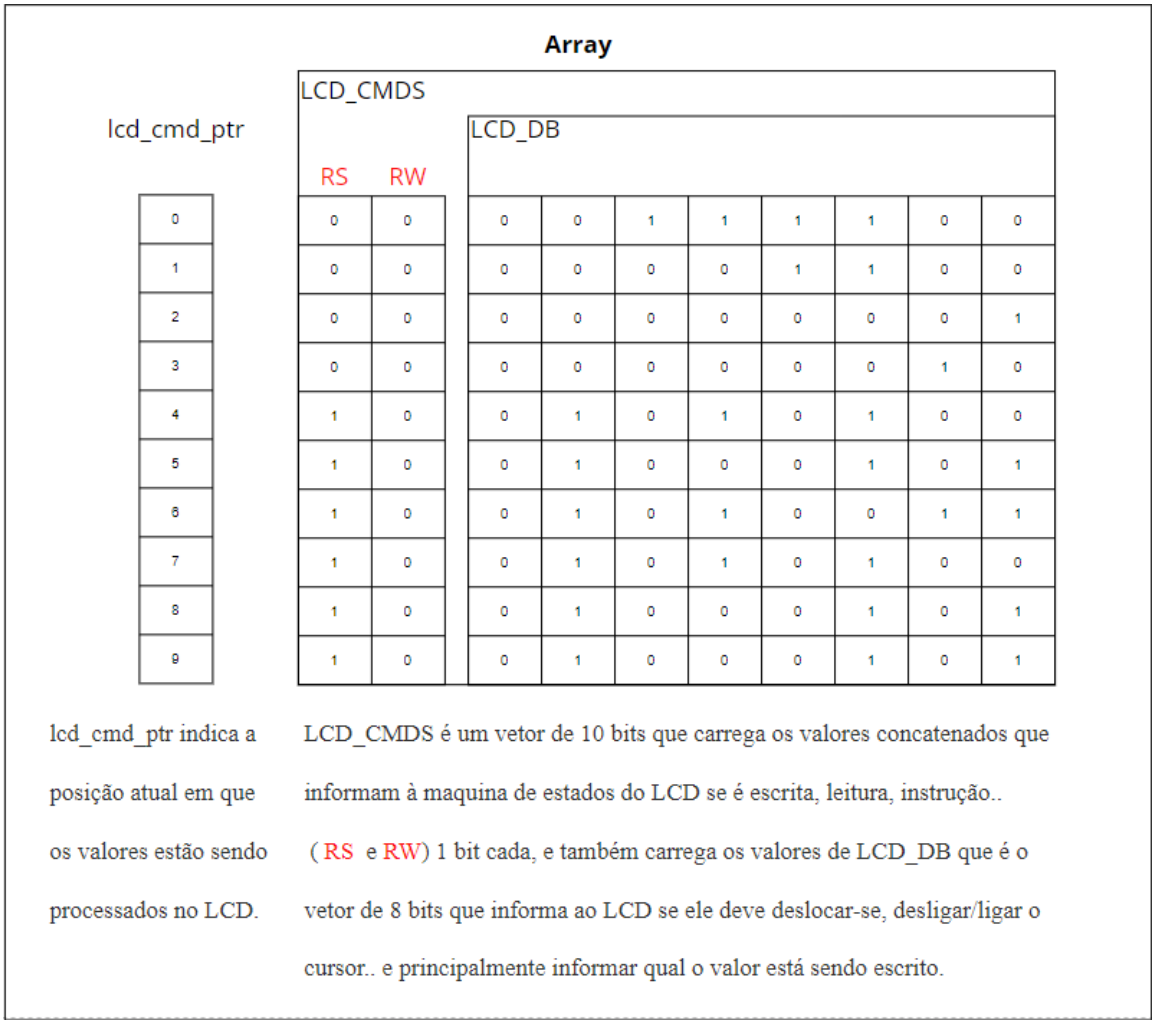


Figura 2.13: Diagrama do Array : LCD

Nosso LCD tem 80 posições físicamente, entretanto, só são visíveis 32 posições. É claro que podemos acessá-las deslocando as posições, que por sinal é uma das funções que podemos definir no `LCD_CMDS`, e já é válido citar que temos apenas alguns caractéres disponível, que estão no manual da placa, que são os caracteres ASCII e alguns katakana japoneses.
Nosso sinal inteiro `lcd_cmd_ptr` pode percorrer livremente o LCD somando posições até alcancar a posição mais significativa definida, então ele recebe que a escrita foi finalizada. Nesse caso, definimos (para o jogo da forca) no código do LCD que o ponteiro alcance a posição máxima 12, então retorne

para 3 que corresponde a função `Return Home` que redefine a posição do cursor e desloca os valores para posição inicial em caso de deslocamento. Figura: 2.14

```
process (lcd_cmd_ptr, oneUSClk)
      begin
      if (oneUSClk = '1' and oneUSClk'event) then
         if ((stNext = stInitDne or stNext = stDisplayCtrlSet or stNext = stDisplayClear) and writeDone = '0') then
            lcd_cmd_ptr <= lcd_cmd_ptr + 1;
         elsif stCur = stPowerOn_Delay or stNext = stPowerOn_Delay then
            lcd_cmd_ptr <= 0;
         elsif lcd_cmd_ptr = 12 then -- Se o ponteiro alcançar a posição 12, ele retorna para a posição 3 (return_home).
          lcd_cmd_ptr <= 3;
         else
            lcd_cmd_ptr <= lcd_cmd_ptr;
         end if;
      end if;
   end process;
```

Figura 2.14: Reposição do Ponteiro de índice do LCD

Os estados do LCD em essência são para inicializar e configurar seus valores de maneira padrão, seja a posição do cursor, o tamanho e a fonte da escrita, suas duas linhas, se há ou não quebra de linha. Todos estes estão definidos por seus estados:

`stFunctionSet`;

`stDisplayCtrlSet`;

`stDisplayClear`;

Cada um destes possui seu atraso respectivo, pois o display LCD em essência é muito lento, portanto é de extrema importância sempre que o alterarmos, também respeitarmos este atraso de inicialização e também o seu atraso de escrita.

Estes abaixo, indicam que o LCD está inicializado, e então nossa máquina de estados fica em uma espécie de "loop"entre estes quando recebe os devidos comandos para exibir no LCD de acordo com seu cursor. Também com seu devido atraso para mostrar um valor no LCD. Figura : 2.15

`stInitDne`;

`stActWr`;

`stChar_delay`

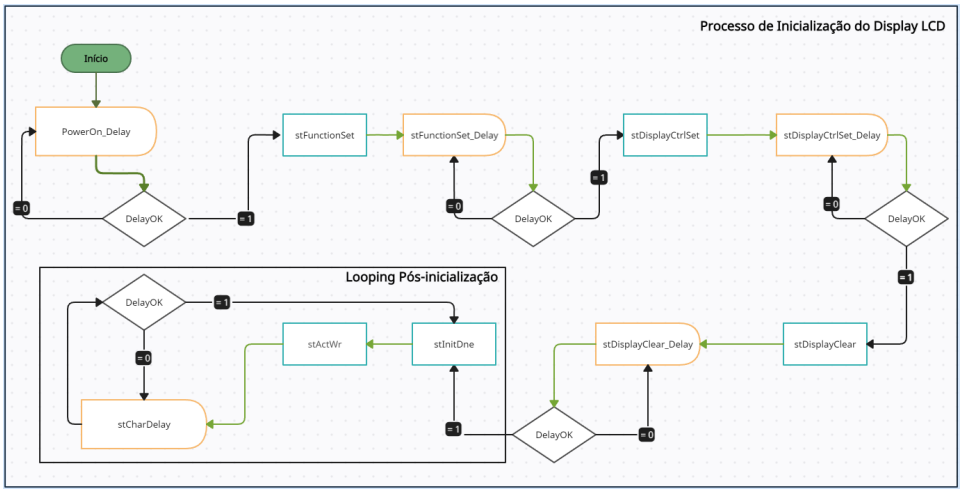


Figura 2.15: Máquina de estados de inicialização do LCD

### 2.2.2 Máquina de estados para o controle de escrita no LCD

No exato momento em que a máquina de estados responsável pela inicialização e alterações no display LCD, temos um sinal chamado `Activatew` que nos diz se é necessário ou não que rodemos a nossa segunda máquina de estados do LCD, responsável pela escrita no mesmo, sendo assim teremos o seguinte diagrama. Figura : 2.16

`stRw`

`stEnable`

`stIdle`

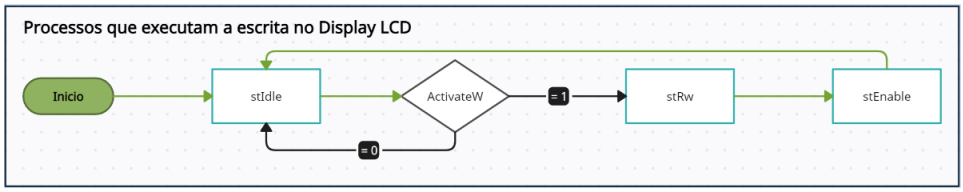


Figura 2.16: Máquina de estados de inicialização do LCD

## 2.3 Jogo da Forca

### 2.3.1 Máquina de estados do jogo da forca

Para projetar o jogo da forca, primeiro construimos um diagrama para entendermos como devemos operar sobre cada informação. Se devemos enviar, receber, e quais atrasos serão necessários para que não haja um problema de sincronia entre os mesmos. Decidimos por criar duas máquinas de estados.

Uma responsável pela interface, onde atualizará a tela do jogo a medida que o jogador progride, sendo estes os traços, as letras corretas e a vida. Figura : 2.18

E o outra máquina de estados estará responsável por alterar as variáveis a medida que recebe informação do teclado.2.17

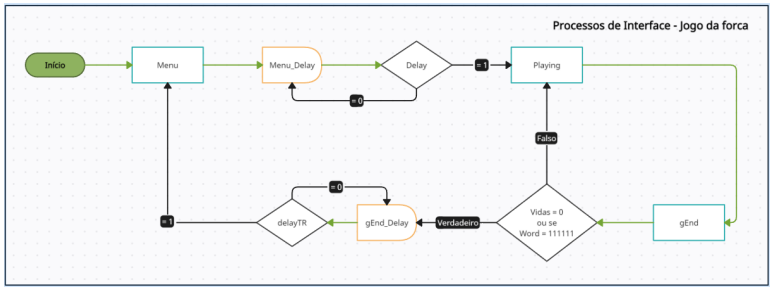


Figura 2.17: Máquina de estados de interface: Jogo da forca

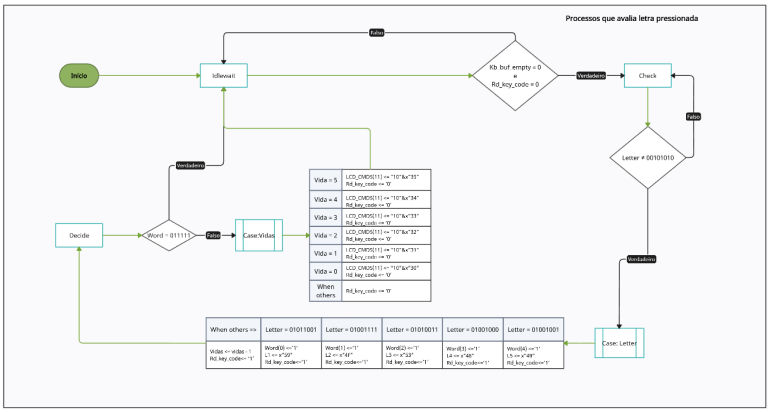


Figura 2.18: Máquina de estados de escrita: Jogo da forca

A comunicação entre os arquivos segue o seguinte diagrama. Figura : 2.34

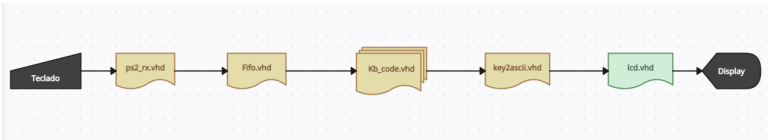


Figura 2.19: Diagrama de conexão entre os arquivos .vhd

### 2.3.2 Interface do jogo da forca

A interface é alterada coerentemente com a máquina de estados `gstate` que possui os seguintes estados :

```
type gstate is (
    menu,           -- Define a tela inicial.
    menu_Delay,     -- Espera que o ponteiro do Display LCD escreva em todas as posições e retorne antes de ir ao prox. estado.
    playing,        -- Define e redefine o display de acordo com a progressão do jogador pressionando o teclado.
    gEnd,           -- Se o jogador vencer o jogo (a palavra for descoberta) ou perder (vidas = 0) este estado mensageia o jogador.
    gEnd_delay      -- Este estado é responsável por fornecer o devido delay ao Display, para que a mensagem seja escrita.
);
```

Figura 2.20: Variaveis de estado de gstate

Para executarmos os estados, primeiramente criamos um processo onde teremos um `case` para cada estado, onde os definiremos utilizando:

`when <estado> =>`

No estado Menu, teremos a predefinição das variáveis funciona como um menu pois é nele que definiremos nosso sinais inicialmente.

OBS : A ideia seria utilizar o menu como um estado que garantisse que nossas variaveis iniciais foram escritas, então nosso próximo estado acaba sendo bem parecido com o próximo `playing` entretanto utilizar o menu como intermédio nos dá a possibilidade de escrever outras mensagens ao jogador sem interferir no desempenho do jogo ( pensando como uma melhoria futura).

```
process(crState,L1,L2,L3,L4,L5,word,vidas,delay)
begin


case crState is
-- Menu define as variáveis iniciais escrevendo da posição 4 até a posição 12
-- Com excessão da posição 11 que será definido apenas na outra maq de estados
when menu =>
LCD_CMDS(4) <= "10"&L1;          --_
LCD_CMDS(5) <= "10"&L2;          --_
LCD_CMDS(6) <= "10"&L3;          --_
LCD_CMDS(7) <= "10"&L4;          --_
LCD_CMDS(8) <= "10"&L5;          --_
LCD_CMDS(9) <= "10"&x"20";          --space
LCD_CMDS(10) <= "10"&x"20";         --space

nxState <= menu_Delay;
```

Figura 2.21: Menu

Logo em seguida temos o `Menu_delay`, este estado vai garantir que tenhamos o tempo necessário para que o ponteiro alcance a posição 12, utilizando o sinal delay que corresponde à um atraso condicionalmente habilitado quando o ponteiro `lcd_cmd_ptr` alcança 12.

```
process (lcd_cmd_ptr, oneUSClk)
      begin
      if (oneUSClk = '1' and oneUSClk'event) then
         if ((stNext = stInitDne or stNext = stDisplayCtrlSet or stNext = stDisplayClear) and writeDone = '0') then
            lcd_cmd_ptr <= lcd_cmd_ptr + 1;
         elsif stCur = stPowerOn_Delay or stNext = stPowerOn_Delay then
            lcd_cmd_ptr <= 0;
         elsif lcd_cmd_ptr = 12 then -- Se o ponteiro alcançar a posição 12, ele retorna para a posição 3 (return_home).
          lcd_cmd_ptr <= 3;
         else
            lcd_cmd_ptr <= lcd_cmd_ptr;
         end if;
      end if;
   end process;
```

Figura 2.22: Intervalo do Ponteiro

```
-- Menu_delay é responsável por aguardar o ponteiro escrever todas as variáveis do menu
-- Para só então liberar o próximo estado.
when menu_Delay =>
if delay = '1' then
nxState <= playing;
else
nxState <= menu_Delay;
end if;
```

Figura 2.23: Menu delay

O próximo estado é o estado `playing` onde nosso jogador visualizará

o progresso do seu jogo, letras acertadas com a troca dos traços pelas letras, erros com a dimunuição das vidas. É importante citar que definimos sinais para efetuar esta troca, para as letras, temos os seguintes sinais, L1,L2,L3,L4,L5, cada um corresponde à uma letra da palavra secreta, foram atribuidos às posições respectivas do LCD, como mostra a imagem a seguir :

```
-- No estado playing temos nosso estado "estático" onde nosso jogador visualizará os traços
-- e sua vida. E a medida que haja erros ou acertos, aparecerá as letras correspondentes aos acertos
-- e a vida diminuirá de acordo com os erros.
when playing =>
-- O estado playing espera que uma letra seja pressionada.
LCD_CMDS(4) <= "10"&L1;          --_
LCD_CMDS(5) <= "10"&L2;          --_
LCD_CMDS(6) <= "10"&L3;          --_
LCD_CMDS(7) <= "10"&L4;          --_
LCD_CMDS(8) <= "10"&L5;          --_
LCD_CMDS(9) <= "10"&x"20";         --space
LCD_CMDS(10) <= "10"&x"20";        --space

LCD_CMDS(12) <= "10"&x"20";        --space

nxState <= gEnd;
```

Figura 2.24: Estado playing

E é valido citar que apesar de nos valores padrão do sinal `LCD CMDS` definimos 25 posições iniciais, entretanto como utilizamos apenas aquelas posições declaradas, não precisamos que nosso ponteiro itere sobre todas as 25 posições, somente as 12 primeiras são suficientes, o que otimizará o tempo de resposta do nosso jogo já que o ponteiro tem menos posições para iterar sobre, permitindo que nosso jogo seja atualizado mais rapidamente.

O próximo estado `gEnd` verifica se o jogador venceu ou perdeu o jogo, que é definido por condição de duas variáveis do jogo, `word` e `vidas`, que ao satisfeitas escreve uma mensagem de derrota/vitoria, enquanto não as satisfaz o jogador sempre volta ao estado anterior e continua jogando.

```
-- gEnd é o estado responsável por verificar se o jogador venceu ou perdeu, temos duas verificações
-- Se a palavra foi completamente descoberta, ou se a variavel vidas é igual a 0.
-- Caso contrário sempre voltamos para o estado playing.
when gEnd =>
 if word = "111111" then

    LCD_CMDS(4) <= "10"&x"47";
    LCD_CMDS(5) <= "10"&x"41";
    LCD_CMDS(6) <= "10"&x"4E";
    LCD_CMDS(7) <= "10"&x"48";
    LCD_CMDS(8) <= "10"&x"4f";
    LCD_CMDS(9) <= "10"&x"55";

    nxState <= gEnd_Delay;

 elsif vidas = 0 then
    LCD_CMDS(4) <= "10"&x"50";
    LCD_CMDS(5) <= "10"&x"45";
    LCD_CMDS(6) <= "10"&x"52";
    LCD_CMDS(7) <= "10"&x"44";
    LCD_CMDS(8) <= "10"&x"45";
    LCD_CMDS(9) <= "10"&x"55";

    nxState <= gEnd_Delay;
 else
   nxState<= playing;
 end if;
```

Figura 2.25: Estado gEnd

O próximo estado `gEnd_delay`. É importante que tenhamos tempo suficiente para que a mensagem final seja escrita, portanto definimos mais um sinal de delay ( distinto dos outros ) para que possamos ter este tempo necessário.

```
-- gEnd_delay é um estado que possui o delay necessário para que a mensagem de vitória ou derrota apareça na tela
-- antes que o jogador seja redirecionado ao menu.
when gEnd_Delay =>
if delayTR= '1' then
   nxState <= menu;
else
nxState <= gEnd_Delay;
end if;
```

Figura 2.26: Estado de atraso do gEnd

**Importante:** A tela de vitória que implementamos não apareceu devidamente, ao jogador finalizar o jogo o cursor percorre apenas 3 posições escrevendo o último número de vidas registrado pelo jogador, o mesmo vale para a tela de derrota. O resultado pode ser visto na imagem abaixo.


Figura 2.27: Erro na implementação da tela de vitória/derrota

### 2.3.3 Processamento das teclas do jogo da forca

Para a escrita, definimos estas variaveis de estado a seguir :

```
type gletter is (
   idlewait,     -- Espera que o jogador pressione uma tecla.
   check,        -- Verifica a letra pressionada pelo jogador.
   decide     -- Verfica se o jogador venceu o jogo ou perdeu o jogo, e atualiza o valor atual da vida.
   );
```

Figura 2.28: Variáveis de estado de gletter

No decorrer destes estados, receberemos uma tecla pressionada do teclado através do sinal `Key_code` que através do `kb_code.vhd` e do `key2ascii.vhd` que faz a conversão para ASCII enviando a letra convertida para o sinal `Letter`.

A partir deste sinal, e do `Kb_buf_empty`, `rd_key_code` passaremos por cada um dos estados, esperando uma letra, verificando se pertence ou não à palavra e no final computando os acertos, e reduzindo a vida com os erros.

Logo inicialmente temos o nosso primeiro estado `Idlewait` que é responsável por aguardar que uma letra no teclado seja pressionada, `kb_buf_empty = 0`, e também `rd_key_code = 0` garantindo que estaremos lendo uma letra diferente pois este limpa o que tem em `key_code`.

```
-- Máquina de estados responsável por receber a tecla pressionada no teclado
-- e avaliar se ela pertence ou não à palavra, de modo a alterar os sinais
-- correspondentes à inicialmente os traços e também a diminuir a vida, a medida
-- que haja error.
process(crLstate,letter,word,vidas,L1,L2,L3,L4,L5,word,rd_key_code,kb_buf_empty)
begin

 case crLstate is
   -- Idlewait espera que uma tecla seja pressionada para ir ao proximo estado.
    when idlewait =>

      if kb_buf_empty = '0' and rd_key_code = '0'  then
         nxLstate<=check;
      else
         nxLstate<=idlewait;
      end if;
```

Figura 2.29: Processo gletter e Idlewait

O próximo estado `Check` é o estado mais importante deste processo, pois é o que vai definir se a letra pressionada pertence a palavra ou não. E também além desta verificação, reduzirá o número de vidas e alterará os sinais L1,L2,L3,L4,L5 de traços para letras quando houver acertos.

```
-- É importante que o rd_key_code receba '1', pois precisamos receber uma nova letra.
if letter /= "00101010" then
     case letter is
        when "01011001" =>
              word(0) <= '1';
              L1 <= x"59";
              nxLstate <= decide;
              rd_key_code <= '1';
        when "01001111" =>
              word(1) <= '1';
              L2 <= x"4f";
              nxLstate <= decide;
              rd_key_code <= '1';
        when "01010011" =>
              word(2) <= '1';
              L3 <= x"53";
              nxLstate <= decide;
              rd_key_code <= '1';
        when "01001000" =>
              word(3) <= '1';
              L4 <= x"48";
              nxLstate <= decide;
              rd_key_code <= '1';
        when "01001001" =>
              word(4) <= '1';
              L5 <= x"49";
              nxLstate <= decide;
              rd_key_code <= '1';
```

Figura 2.30: Estado Check

As condições para que a letra pressionada seja checada é que ela esteja limitada às letras disponíveis pelo conversor para ASCII, `letter` $\neq$ 00101010. E então o vetor `Word` tem seus bits alterados para o nível alto de acordo com que uma letra é correta, alterando também o valor do sinal respectivo para a letra correta.

Caso a letra não é a correta, o sinal permanece inalterado, e a vida é reduzida em 1, perceba que precisamos alterar nosso `rd_key_code` para 1, pois queremos que nosso `kb_buf_empty` volte para o nível alto, permitindo que uma outra letra seja pressionada :

```
            when others =>
               nxLstate <= decide;
               vidas <= vidas - 1;
               rd_key_code <= '1';
            end case;
      else
         nxLstate <= check;
      end if;
```

Figura 2.31: Erros e decremento da vida

**Importante:** Há um erro na lógica implementada para as vidas onde elas incrementam e decrementam aleatóriamente entre 0 e 5, mesmo constando devidamente caso a caso em nossa tabela à seguir, no próximo estado, onde definiremos os casos em que o número de vidas entra como condicional para escrevermos no display LCD.

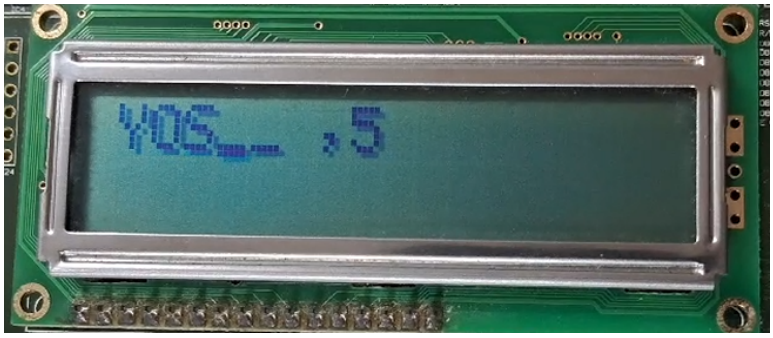

Figura 2.32: Inconsistência na implementação do decremento de Vidas

Por fim, nosso último estado é o estado `decide` onde realiza uma verificação similar ao processo de interface, entretanto neste processo seremos responsáveis por enviar o sinal complementando o valor do vetor `Word` declarando vitória ou reduzindo a vida até 0, neste estado também atualizamos o valor da vida, escrevendo no LCD quando a vida é igual a 5,4,3.. até alcançar 0.

```
-- decide é o estado responsável por finalizar o jogo,
-- Ele verifica se a palavra foi descoberta, então vai para o estado inicial,
-- caso o contrário ele atualiza a vida e passa para o próximo estado.
when decide =>
   if (word = "011111") then
         word(5) <= '1';
         nxLstate <= idlewait;
         rd_key_code <= '0';
   else
      case vidas is
         when "101" =>
            LCD_CMDS(11) <= "10"&x"35";
            nxLstate <= idlewait;
            rd_key_code <= '0';
         when "100" =>
            LCD_CMDS(11) <= "10"&x"34";
            nxLstate <= idlewait;
            rd_key_code <= '0';
         when "011" =>
            LCD_CMDS(11) <= "10"&x"33";
            nxLstate <= idlewait;
            rd_key_code <= '0';
```

Figura 2.33: Estado Decide

```
                when "010" =>
                   LCD_CMDS(11) <= "10"&x"32";
                   nxLstate <= idlewait;
                   rd_key_code <= '0';
                when "001" =>
                   LCD_CMDS(11) <= "10"&x"31";
                   nxLstate <= idlewait;
                   rd_key_code <= '0';
                when "000" =>
                   LCD_CMDS(11) <= "10"&x"30";
                   nxLstate <= idlewait;
                   rd_key_code <= '0';
                when others =>
                   nxLstate <= idlewait;
                   rd_key_code <= '0';
             end case;
          end if;

      end case;
      end process;
end Behavioral;
```

Figura 2.34: Estado Decide Pt.2

# Capítulo 3

# Melhorias


Figura 3.1: Apresentação do trabalho

O jogo da forca funcionou como esperado, preenchendo a palavra secreta quando a letra era acertada e reduzindo a vida quando erros eram cometidos. Entretanto houveram imprevistos, como no sistema de decréscimo de vidas, que não funcionou corretamente: o número de vidas aparecia aleatóriamente no lcd, obdecendo o intervalo de 0 à 5, ou seja, caso aparecesse 0(independente de ser o quinto erro ou não) o jogo terminava, como se o usuário já tivesse gastado suas 5 tentativas.

# Capítulo 4

# Conclusões

Inicialmente tivemos muita dificuldade em compreender como realizar a "conexão"entre os arquivos, entretanto com o decorrer compreendemos muito melhor como funciona a instanciação e a possibilidade de trabalhar com várias máquinas de estado operando sobre um propósito específico, no nosso caso, o jogo da forca.

Foi compreendido como implementar uma máquina de estados e como condicionar a transição de cada um dos estados de modo à realizar alterações necessárias.

Um dos desafios foi compreender individualmente cada um dos arquivos principais do teclado e do LCD, um dos motivos foi pela escassez de conteúdos disponíveis na internet, logo a utilização do manual da placa e do livro e a leitura ponta a ponta do código foram essenciais para a sua compreensão.

# Capítulo 5

# Referênciais

**Diagramas:**

https://creately.com/pt/plans/?ref=home

https://online.visual-paradigm.com/pt/diagrams/features/state-machine-diagram-softwar

**Máquina de estados:**

https://www.youtube.com/watch?v=eUMbCU8QxWQ

https://www.youtube.com/watch?v=EDodM1aPJdU

Pong P. Chu, "FPGA Prototyping by Examples (Xilinx Spartan-3 Version)",
Wiley-Interscience, 1st Ed.

**FPGA:**

Spartan-3AN User Guide