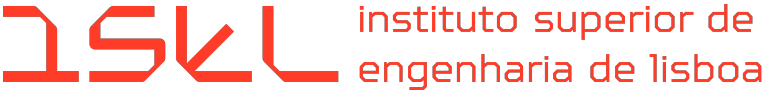

Keyboard Reader

3º Relatório do Projeto *Roulette Game*

Trabalho realizado por:
Gustavo Costa | Nº 52808
Ian Frunze | Nº 52867
Rafael Pereira | Nº 52880

Turma: **LEIC24D**
Docentes: **David Velez e Rui Duarte**
Licenciatura em Engenharia Informática e de Computadores
Laboratório de Informática e de Computadores (LIC)
2024 / 2025 - Semestre de Verão

Conteúdo

Lista de Figuras

Lista de Códigos

1 Introdução

O módulo **Keyboard Reader** representa um componente central no desenvolvimento do projeto **RouletteGame**, sendo responsável por estabelecer a interface entre o utilizador e o sistema através de um teclado matricial 4×4. A sua principal função consiste em captar as teclas premidas, processá-las de forma fiável e disponibilizar essa informação ao módulo de controlo, respeitando os requisitos de robustez, precisão e sincronização.

Integrado num sistema híbrido (hardware/software), este módulo assume um papel crucial na cadeia de funcionamento global, devendo garantir uma comunicação fluida entre os sinais gerados pela ação do utilizador e o processamento interno do jogo. Esta comunicação é assegurada por uma arquitetura modular composta por três blocos distintos: **Key Decode**, **Ring Buffer** e **Output Buffer**. Cada bloco desempenha uma função específica, contribuindo para a organização e clareza do sistema como um todo.

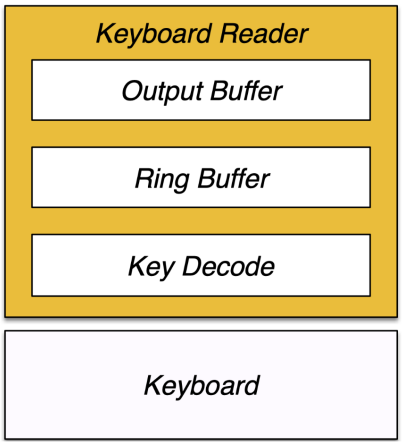

Figura 1: Arquitetura do módulo *Keyboard Reader*.

O *Key Decode* realiza o varrimento do teclado e deteta a tecla premida, convertendo-a num código digital. O *Ring Buffer* atua como uma fila FIFO que armazena temporariamente os dados, desacoplando a frequência de entrada de teclas do ritmo de leitura do sistema consumidor. Por fim, o *Output Buffer* assegura a entrega controlada e sem perdas dos dados ao consumidor, através de um protocolo de *handshake* bem definido.

Esta estrutura modular, suportada por sinais de controlo como `key_pressed`, `data_ready` e `ack`, garante um fluxo de dados ordenado, fiável e tolerante a condições adversas, como a pressão rápida ou simultânea de várias teclas. A implementação foi realizada em VHDL, respeitando boas práticas de design digital e utilizando máquinas de estados para controlo interno.

Nas secções seguintes, descreve-se em detalhe o funcionamento interno de cada bloco, a sua implementação, e os testes de simulação que validam o correto funcionamento do módulo.

1.1 Especificação de Requisitos

O desenvolvimento do módulo *Keyboard Reader* foi guiado por um conjunto de requisitos funcionais que garantem a sua integração eficaz no sistema global do *RouletteGame*. Estes requisitos visam assegurar o correto funcionamento do módulo, mesmo em condições adversas ou de elevada utilização:

- **Leitura fiável e inequívoca das teclas:** O sistema deve identificar corretamente qualquer uma das 16 teclas do teclado matricial 4×4, sem falhas ou ambiguidade na deteção.
- **Armazenamento temporário dos eventos de tecla:** É fundamental garantir que as teclas premidas são armazenadas, mesmo que o sistema consumidor esteja temporariamente indisponível para processá-las, evitando assim perdas de dados.
- **Desacoplamento temporal:** A arquitetura deve permitir ritmos de operação distintos entre a introdução de dados (premir de teclas) e o seu consumo, assegurando fluidez no funcionamento global.
- **Protocolo de comunicação robusto:** A interface com o módulo consumidor deve seguir um protocolo bem definido, baseado em sinais de controlo como `data_ready` e `ack`, garantindo a entrega ordenada e sem duplicações.
- **Resiliência a condições adversas:** O sistema deve funcionar corretamente mesmo em cenários de utilização intensiva, como a pressão rápida ou simultânea de várias teclas, aplicando mecanismos de *debouncing* e validação.

A arquitetura adotada — composta pelos blocos *Key Decode*, *Ring Buffer* e *Output Buffer* — visa satisfazer plenamente estes requisitos, oferecendo uma solução modular, fiável e facilmente validável, como será demonstrado nas secções seguintes.

2 Arquitetura do Keyboard Reader

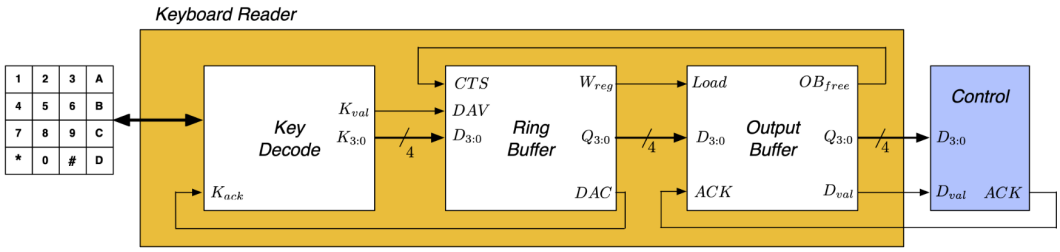

Figura 2: Arquitetura do módulo Keyboard Reader

A arquitetura do módulo Keyboard Reader é composta por três blocos funcionais principais: **Key Decode**, **Ring Buffer** e **Output Buffer**. Cada um destes blocos desempenha um papel específico na leitura, armazenamento e entrega dos códigos das teclas, garantindo um funcionamento eficiente e robusto do sistema.

2.1 Apresentação - Key Decode

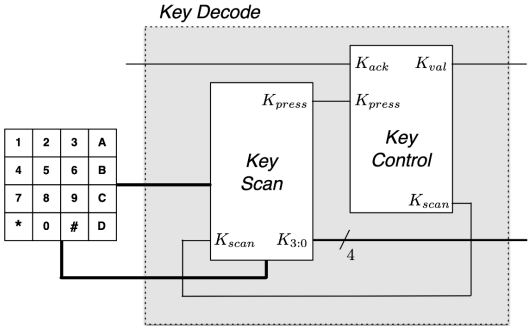

Figura 3: Construção interna do KeyDecode

O bloco **Key Decode** é responsável pelo varrimento do teclado matricial, detectando a tecla premida e convertendo-a num código digital correspondente. Este bloco implementa um algoritmo de varrimento eficiente, garantindo uma deteção rápida e precisa das teclas, mesmo em condições de utilização intensiva. A arquitetura do KeyDecode está referenciada na figura 3, onde se pode observar a interligação entre os diferentes componentes e os sinais de controlo que garantem um funcionamento eficiente.

2.2 Apresentação - Ring Buffer

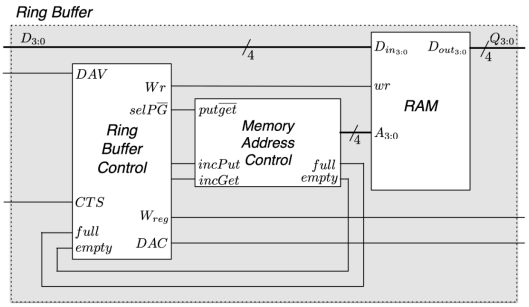

Figura 4: Construção interna do RingBuffer

O **Ring Buffer** constitui um elemento central da arquitetura, implementando uma fila FIFO (First In, First Out) que armazena temporariamente os códigos das teclas. Esta estrutura é fundamental para desacoplar o ritmo de entrada do teclado do ritmo de processamento pelo sistema. O Ring Buffer permite acumular múltiplos códigos de teclas, garantindo que não há perda de dados mesmo quando o sistema consumidor está temporariamente ocupado. A figura 4 ilustra a arquitetura do Ring Buffer, destacando os principais componentes e sinais de controlo.

2.3 Apresentação - Output Buffer

O **Output Buffer** serve como interface final com o módulo de controlo, implementando um protocolo de handshake que garante a entrega controlada dos códigos das teclas ao consumidor. Este bloco assegura que os dados são entregues de forma ordenada e sem perdas, mesmo em situações de utilização intensa. A figura 5 apresenta a arquitetura do Output Buffer, evidenciando os principais componentes e sinais de controlo envolvidos na comunicação com o módulo consumidor.

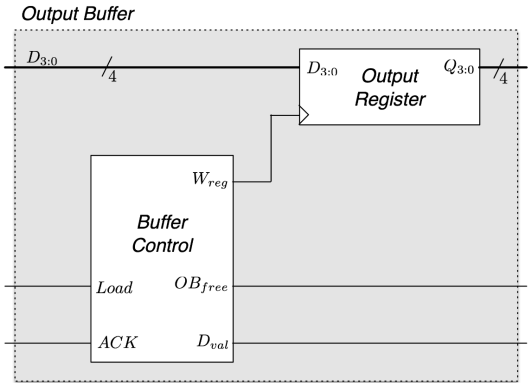

Figura 5: Construção interna do OutputBuffer

A interligação entre estes blocos é realizada através de sinais de controlo que implementam protocolos de handshake representados na figura 2. Estes sinais são fundamentais para garantir um fluxo de dados ordenado e sem perdas entre os diferentes componentes do sistema. A tabela 1 apresenta uma descrição detalhada dos principais sinais de controlo utilizados na arquitetura do Keyboard Reader.

Sinal	Descrição	Direção
`key_pressed`	Indica que uma tecla foi premida	Key Decode → Ring Buffer
`data_ready`	Indica que há dados disponíveis no Ring Buffer	Ring Buffer → Output Buffer
`ack`	Confirmação de recepção de dados pelo Output Buffer	Output Buffer → Ring Buffer

Tabela 1: Sinais de controlo utilizados na arquitetura do Keyboard Reader

Esta estrutura modular, com responsabilidades bem definidas e interfaces claras entre os blocos, contribui significativamente para a robustez e fiabilidade do sistema, facilitando também o processo de teste e validação. A implementação de protocolos de handshake entre os diferentes componentes assegura que o fluxo de dados é controlado e eficiente, minimizando o risco de perda de informações e garantindo uma resposta rápida do sistema a eventos de entrada do utilizador.

3 Bloco Key Decode

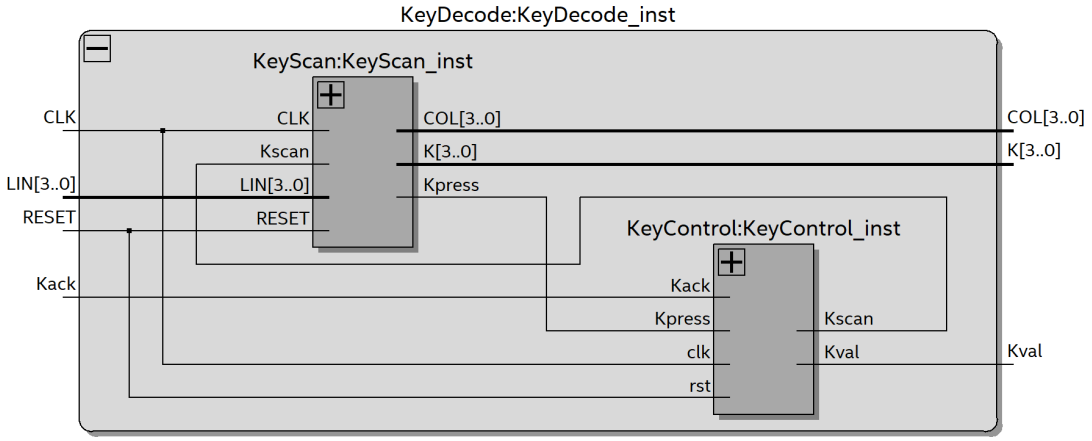

Figura 6: Diagrama do Key Decode

3.1 Descrição e Funcionamento

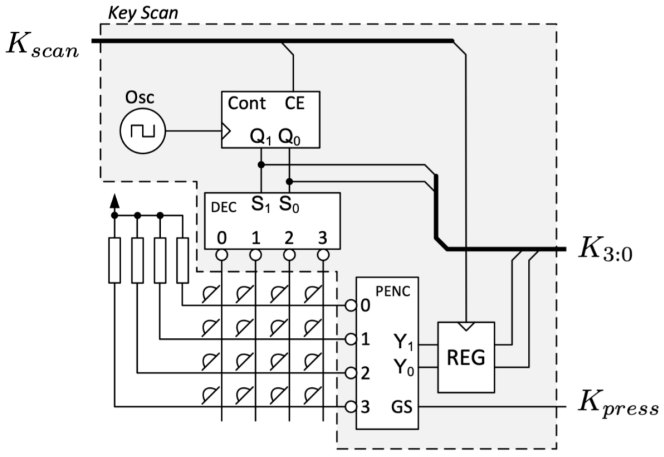

Figura 7: Diagramas do KeyScan

O bloco Key Decode constitui a primeira etapa do processamento, sendo responsável pela interface direta com o hardware do teclado matricial 4×4. A sua implementação engloba dois componentes principais:

- **KeyScan**: responsável pelo varrimento sequencial das colunas do teclado e pela deteção de teclas premidas.
- **KeyControl**: encarregado de gerir o fluxo de dados entre o KeyScan e o consumidor, assegurando a correta interpretação dos sinais gerados.

O teclado matricial 4×4 é composto por 16 teclas dispostas em 4 linhas e 4 colunas. Cada tecla, quando pressionada, estabelece uma conexão entre a linha e a coluna correspondente, permitindo a identificação da tecla através da sua posição na matriz. O KeyScan realiza o varrimento sequencial das colunas, aplicando um sinal a cada coluna e monitorizando o estado das linhas para detetar a pressão de teclas.

Quando uma tecla é premida, estabelece-se uma conexão elétrica entre a coluna ativa e uma das linhas, permitindo a identificação da tecla através da sua posição na matriz. Na figura 6 é apresentado o diagrama do Key Decode, onde se pode observar a interligação entre os componentes e as suas respectivas entradas e saídas.

O componente **KeyScan** realiza o varrimento sequencial das colunas do teclado, aplicando um sinal a cada coluna e monitorizando o estado das linhas para detetar a pressão de teclas. Quando uma tecla é premida, estabelece-se uma conexão elétrica entre a coluna ativa e uma das linhas, permitindo a identificação da tecla através da sua posição na matriz.

O KeyScan gera um sinal de pressão de tecla (*Kpress*) e um vetor que representa a tecla premida (K). O sinal *Kpress* é ativado quando uma tecla é detetada, enquanto o vetor K contém a informação da tecla correspondente. O sinal *Kscan* é utilizado para controlar o varrimento das colunas, alternando entre os estados de ativação e desativação

das colunas. Esta implementação utiliza apenas um contador de 2 bits, e como o clock da placa de10-lite é de 50Mhz, 20 ns para efetuar um ciclo, e o contador precisando de 4 ciclos para dar a volta, então temos uma varredura completa do teclado em 80 ns, essa velocidade gerava alguns erros indesejados ao capturar a tecla, e também como é improvável um ser humano premir 1 tecla a cada 80 ns, foi implementada um divisor de clock, para chegar a valor de 200 voltas por segundo, ou seja o contador faz 200 varredura ao teclado por segundo, para tal dividimos o clock por 62500, optamos por este valor, pois ainda é superior a capacidade humana, e também não verificamos mais nenhum erro ao ler a tecla.

O componente **KeyControl** gere o processo de varrimento e implementa o controlo de fluxo com o consumidor. Esta gestão é crucial para garantir que cada tecla é reconhecida apenas uma vez, evitando problemas como o "bouncing"ou leituras duplicadas. O KeyControl utiliza um estado de espera (*WAIT*) para aguardar a deteção de uma tecla, e um estado de confirmação (*CONFIRM*) para assegurar que a tecla foi realmente premida.

Após a confirmação, o KeyControl gera um sinal de reconhecimento (*Kack*) que é enviado ao consumidor, indicando que a tecla foi processada com sucesso. O diagrama de estados do KeyControl é apresentado na figura 8.

O estado *WAIT* aguarda a ativação do sinal *Kpress*, que indica que uma tecla foi premida. Quando o sinal *Kpress* é ativado, o sistema transita para o estado *CONFIRM*, onde verifica se a tecla permanece premida. Se a tecla continuar premida, o sistema gera um sinal de reconhecimento (*Kack*) e retorna ao estado *WAIT* para aguardar a próxima tecla. Caso contrário, o sistema permanece no estado *WAIT* até que uma nova tecla seja detetada.

O KeyControl também implementa um mecanismo de debouncing, que assegura que o sinal de pressão de tecla (*Kpress*) é mantido por um período mínimo antes de ser considerado válido. Este mecanismo é essencial para evitar leituras incorretas devido a oscilações elétricas que podem ocorrer quando uma tecla é premida ou solta.

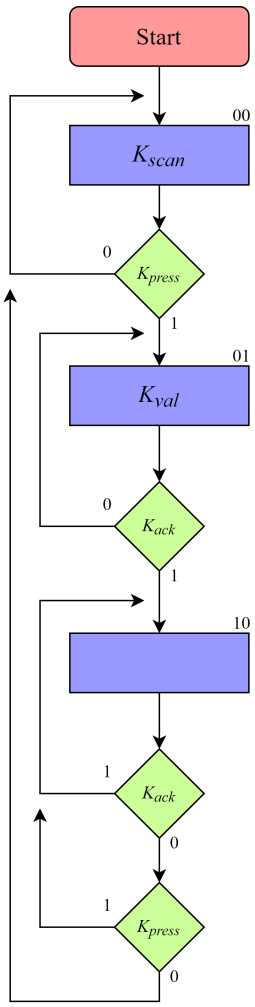

Figura 8: Diagrama do KeyControl

3.2 Implementação em VHDL

A implementação do Key Decode em VHDL reflete fielmente a arquitetura descrita. O código seguinte apresenta a estrutura principal do componente:

```
architecture arch_KeyDecode of KeyDecode is
component KeyScan is
port(
    CLK: in std_logic;
    RESET: in std_logic;
    Kscan: in std_logic;
    LIN: in std_logic_vector(3 downto 0);
    COL: out std_logic_vector(3 downto 0);
    Kpress: out std_logic;
    K: out std_logic_vector(3 downto 0)
);
end component;

component KeyControl is
port (
    clk: in std_logic;
    rst: in std_logic;
    Kack: in std_logic;
```

```
    Kpress: in std_logic;
    Kscan: out std_logic;
    Kval: out std_logic
);
end component;
```

```
signal temp_Kpress, temp_Kscan: std_logic;
```

Código 1: Estrutura do Key Decode em VHDL

A instanciação e interligação destes componentes é realizada conforme o seguinte excerto:

```
KeyScan_inst: KeyScan port map(
    CLK => CLK,
    RESET => RESET,
    Kscan => temp_Kscan,
    LIN => LIN,
    COL => COL,
    Kpress => temp_Kpress,
    K => K
);
```

```
KeyControl_inst: KeyControl port map(
    clk => CLK,
    rst => RESET,
    Kack => Kack,
    Kpress => temp_Kpress,
    Kscan => temp_Kscan,
    Kval => Kval
);
```

Código 2: Instanciação dos componentes do Key Decode

Esta implementação garante uma deteção fiável das teclas premidas, com um controlo de fluxo que evita leituras duplicadas e assegura que cada tecla é corretamente processada antes de iniciar um novo ciclo de varrimento.

4 Bloco Ring Buffer

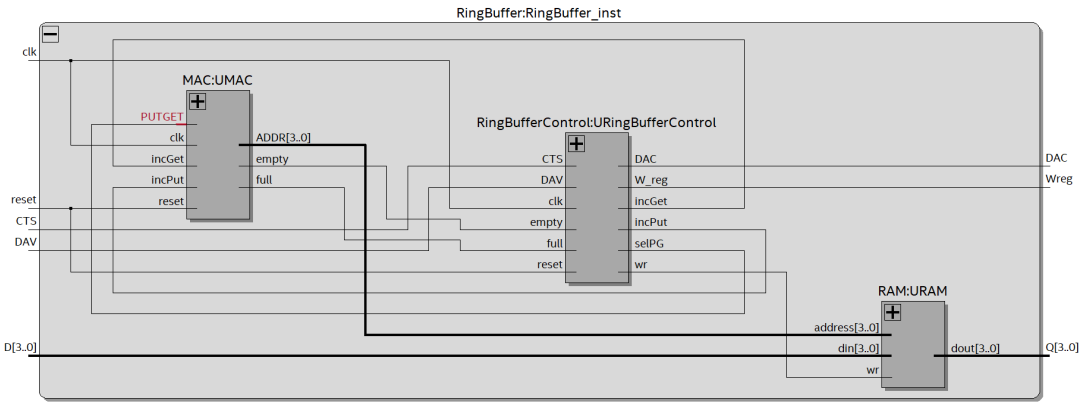

Figura 9: Diagrama de blocos do Ring Buffer

4.1 Descrição e Funcionamento

O Ring Buffer é uma estrutura de dados do tipo FIFO (First-In, First-Out), desenvolvida para armazenar até 16 palavras de 4 bits, funcionando como um intermediário entre o KeyDecode e o OutputBuffer. A sua principal função é garantir o armazenamento ordenado dos dados recebidos, respeitando a ordem de chegada, e permitir a sua leitura posterior por outra entidade, quando esta estiver pronta.

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

entity RingBuffer is
    port(
        clk: in std_logic;
        reset: in std_logic;
        CTS: in std_logic;
        DAV: in std_logic;
        D: in std_logic_vector(3 downto 0);
        Wreg: out std_logic;
        Q: out std_logic_vector(3 downto 0);
        DAC: out std_logic
    );
end RingBuffer;
```

Código 3: Entidade RingBuffer em VHDL

A operação de escrita inicia-se com a ativação do sinal DAV (Data Available), que indica a disponibilidade de um novo dado por parte do produtor. Se houver espaço livre na memória, o dado presente na entrada D é armazenado, e o sinal DAC (Data Accepted) é ativado para confirmar a receção. Este sinal é desativado apenas após DAV ser desligado, garantindo a sincronização entre os dois sistemas.

```
architecture ringBuffer_arch of RingBuffer is

component RingBufferControl is
    port(
        clk: in std_logic;
        reset: in std_logic;
        DAV: in std_logic;
        CTS: in std_logic;
        full: in std_logic;
        empty: in std_logic;
        DAC: out std_logic;
```

```
        selPG: out std_logic;
        wr: out std_logic;
        W_reg: out std_logic;
        incPut: out std_logic;
        incGet: out std_logic
    );
end component;
```

Código 4: Instanciamento do RingBufferControl em VHDL

A estrutura é baseada numa memória RAM, cuja leitura e escrita são controladas por endereços gerados pelo bloco Memory Address Control (MAC). Este bloco contém dois registos: PUT_REG4 (para escrita) e GET_REG4 (para leitura), permitindo as operações de incremento (incPut e incGet). O MAC também fornece os sinais full e empty, que indicam se o buffer está cheio ou vazio, respetivamente.

```
architecture ringBuffer_arch of RingBuffer is
component RAM is
    generic(
        ADDRESS_WIDTH : natural := 3;
        DATA_WIDTH : natural := 4
    );
    port(
        address : in std_logic_vector(
    ADDRESS_WIDTH - 1 downto 0);
        wr: in std_logic;
        din: in std_logic_vector(DATA_WIDTH - 1
    downto 0);
        dout: out std_logic_vector(DATA_WIDTH - 1
     downto 0)
    );
end component;
```

```
component MAC is
    port(
        clk: in std_logic;
        reset: in std_logic;
        incPut: in std_logic;
        incGet: in std_logic;
        PUTGET: in std_logic;
        ADDR: out std_logic_vector(3 downto 0);
        full: out std_logic;
        empty: out std_logic
    );
end component;
```

Código 5: RAM e MAC instanciado no RingBuffer em VHDL

A leitura de dados ocorre quando o sinal CTS (Clear To Send), vindo do OutputBuffer, está ativo, permitindo a entrega do dado armazenado. O Ring Buffer garante assim uma comunicação eficiente e sincronizada entre os módulos produtores e consumidores de dados, sendo essencial em sistemas digitais onde a transferência ordenada de informação é necessária.

```
signal full_temp, empty_temp, Wr_temp, selPG_temp
    , incPut_temp, incGet_temp: std_logic;
signal address_temp: std_logic_vector(3 downto 0)
    ;
begin
    URingBufferControl: RingBufferControl
        port map(
            clk => clk,
            reset => reset,
            DAV => DAV,
            CTS => CTS,
            full => full_temp,
            empty => empty_temp,
            DAC => DAC,
            selPG => selPG_temp,
            wr => Wr_temp,
            W_reg => Wreg,
            incPut => incPut_temp,
            incGet => incGet_temp
        );
    UMAC: MAC
        port map(
            clk => clk,
            reset => reset,
            incPut => incPut_temp,
            incGet => incGet_temp,
            PUTGET => selPG_temp,
            ADDR => address_temp,
            full => full_temp,
            empty => empty_temp
        );
    URAM: RAM
        generic map (
        ADDRESS_WIDTH => 4,
        DATA_WIDTH    => 4
        )
        port map(
            address => address_temp,
            wr => Wr_temp,
            din => D,
            dout => Q
        );
```

```
end ringBuffer_arch;
```

Código 6: RingBufferControl instanciado no RingBuffer em VHDL

4.2 RingBufferControl

A máquina de estados responsável por controlar o Ring Buffer inicia no estado "000". Neste estado inicial, todas as saídas são colocadas a '0', exceto o sinal SelPG, que mantém o valor da última operação realizada (leitura ou escrita). O sinal SelPG é alterado apenas durante as etapas de escrita (ficando a '1') ou de leitura (ficando a '0'), permitindo selecionar o endereço correto para a operação pretendida.

O diagrama ASM (Algorithmic State Machine), ilustrado na Figura 10, descreve o comportamento sequencial do RingBufferControl. A transição entre estados depende das combinações dos sinais de controlo CTS, DAV, full e empty, assegurando que apenas são realizadas operações válidas de leitura ou escrita conforme o estado do buffer.

Em seguida com o CTS valor lógico '1' e empty com valor lógico '0', ou seja (CTS AND (not empty)) a máquina irá para o estado de leitura:

- Estado "001": Deixa todos os sinais, incluindo o SelPG, com valor lógico '0' , e vai para o estado seguinte
- Estado "010": Ativa o sinal wreg, e mantem todos os outros sinais a '0', este sinal é ativado apenas neste estado e não no estado anterior, pois assim garante que o MAC tenha tempo de processar o sinal SelPG para que o adereço esteja estável na saída do MAC, em seguida vai para o estado "011"
- Estado "011": Deixa com valor lógico '1', e mantem todos os outros sinais a '0', e volta para o estado inicial "000"

Caso o resultado da combinação lógica (CTS AND (not empty)) seja '0', e (DAV AND (not Full)) tenha como resultado '1' á máquina vai para o processo de escrito, esatdo "100", caso contrário volta para o estado "000":

- Estado "100": Ativa o sinal selPG, e desativa os restantes, vai para o estado seguinte "101"
- Estado "101": Ativa o sinal wr e mantem o selPG com valor lógico '1', em seguida muda para o estado "110"
- Estado "110": Deixa o sinal DAC e mantem o selPG com valor lógico '1', caso a entrada DAV seja '1' mantem o estado atual "110", se esta entrada for '0' vai para o estado "000"e ativa o sinal incPut

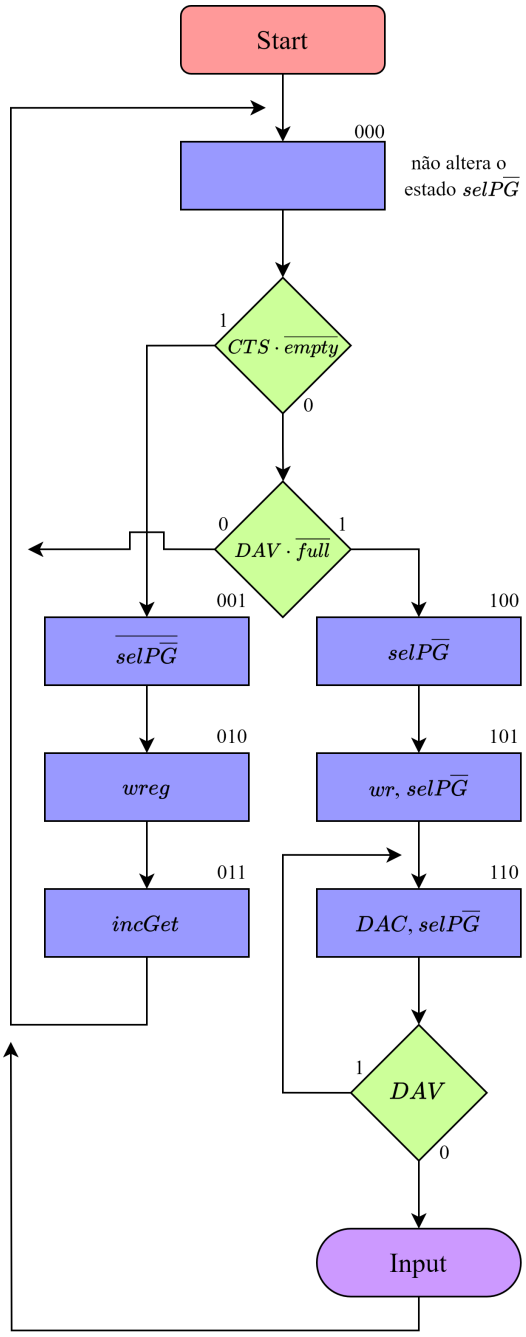

Figura 10: ASM do RingBufferControl

4.3 MAC

O bloco Memory Address Control (MAC) é responsá-
vel pela gestão dos endereços de escrita e leitura da memória no Ring Buffer, garantindo o funcionamento correto da estrutura FIFO. Este bloco controla os índices putIndex (para escrita) e getIndex (para leitura) guardando-os em dois registros respectivamente, PUT_REG4 e GET_REG4, ambos implementados como registos de 4 bits, uma vez que a capacidade do buffer é de 16 posições ($2^4 = 16$).

Cada índice pode ser incrementado através dos sinais incPut e incGet, respetivamente. O incremento é feito de forma circular (módulo 16), assegurando o comportamento de buffer circular.

O multiplexador controlado pelo sinal putGet seleciona qual dos dois índices (put ou get) será utilizado para endereçar a RAM, dependendo se a operação em curso é de escrita ou leitura. O funcionamento do nosso MAC necessita de que o sinal PutGet seja preservado pelo controlo, será visto mais afrente, ou seja quando há um acesso , por exemplo escrita, o que acontece é o seguinte:

O sinal PutGet através do MUX seleciona o registro PUT_REG4, assim saindo o index armazenado nesse mesmo registo, em seguida esse índice entra no bloco ADDER, em que é incrementado em 1 valor, assim o sinal do resultado é preenchido por esse novo índice.

No momento em que o sinal incPut é ativado a '1' o registo PUT_Reg4 guarda o resultado proveniente do ADDER.

A lógica dos sinais FULL e empty

As saídas dos registros PUT_Reg4 e GET_Reg4 são comparadas, caso estas sejam iguais é produzido um sinal com valor lógico '1', isto porque, sempre que a memoria está fazia ou cheio os índices de PUT e GET são iguais.

Como o nosso contol garante que o sinal PutGet corresponde constantemente á última operação efetua, é fácil diferenciar se a igualdade acima provem de um GET (RAM vazia) ou de um PUT(RAM cheia), assim basta efetua um simples AND:

Full = PutGet and equals empty = (not PutGet) and equals

5 Bloco Output Buffer

O bloco Output Buffer é responsável por interligar o Ring Buffer com o sistema consumidor, que neste caso é o módulo Control. Este bloco atua como um registo temporário de 4 bits que guarda os dados lidos do Ring Buffer até serem consumidos.

Inicialmente, o Output Buffer indica que está livre através do sinal OBfree. Quando o Ring Buffer tem dados prontos para entrega e OBfree está ativo, ele escreve os dados no Output Buffer ativando o sinal Wreg. Após receber os dados, o Output Buffer desativa OBfree e ativa o sinal Dval, informando o módulo Control que os dados estão válidos e prontos para serem lidos.

O sistema consumidor lê os dados e, quando terminar, ativa o sinal ACK. Ao detetar este sinal, o Output Buffer desativa Dval e volta a ativar OBfree, indicando que está novamente disponível para receber novos dados.

Este mecanismo garante uma transferência sincronizada e segura entre produtor e consumidor, evitando perdas ou sobreposição de dados

O Output Buffer representa a interface final do Keyboard Reader com o sistema consumidor. A sua implementação em VHDL é apresentada a seguir:

```
architecture arc_OutputBuffer of OutputBuffer is
component OutputRegister is
port (
    clk: in std_logic;
    reset: in std_logic;
    D: in std_logic_vector(3 downto 0);
    Q: out std_logic_vector(3 downto 0)
);
end component;

component BufferControl is
port (
    clk: in std_logic;
    reset: in std_logic;
    Load: in std_logic;
    ACK: in std_logic;
    Wreg: out std_logic;
    OBfree: out std_logic;
    Dval: out std_logic
);
end component;

signal Wreg_temp: std_logic;
```

Código 7: Estrutura do OutputBuffer em VHDL

A interligação destes componentes é realizada conforme o trecho seguinte:

```
UBufferControl: BufferControl
port map (
    clk => clk,
    reset => reset,
    Load => Load,
    ACK => ACK,
    Wreg => Wreg_temp,
    OBfree => OBfree,
    Dval => Dval
);

UOutputRegister: OutputRegister
port map (
    clk => Wreg_temp,
```

```
    reset => reset,
    D => D,
    Q => Q
);
```

Código 8: Instanciação dos componentes do OutputBuffer

O Output Buffer é composto por dois componentes principais. O BufferControl implementa a máquina de estados que gere o protocolo de handshake com o sistema consumidor, controlando os sinais OBfree, Dval e reagindo ao sinal ACK. O OutputRegister armazena temporariamente o código da tecla, garantindo a sua estabilidade durante todo o processo de transferência.

O funcionamento do Output Buffer segue um ciclo bem definido. Inicialmente, o buffer está no estado livre, indicado pelo sinal OBfree ativo. Quando o Ring Buffer tem um código disponível, ativa o sinal Load, que é recebido pelo BufferControl. Em resposta, o BufferControl ativa o sinal Wreg_temp, permitindo a escrita no OutputRegister, desativa o sinal OBfree e ativa o sinal Dval, indicando ao consumidor que um código válido está disponível. O sistema consumidor, ao verificar que Dval está ativo, lê o código e ativa o sinal ACK para confirmar a receção. Ao receber ACK, o BufferControl desativa Dval e reativa OBfree, preparando-se para um novo ciclo.

Esta implementação do Output Buffer garante uma interface robusta com o sistema consumidor, assegurando que cada código de tecla é entregue uma e apenas uma vez, sem perdas ou duplicações. A utilização de um registo dedicado (OutputRegister) para armazenar o código durante o processo de transferência evita problemas de metaestabilidade ou leituras parciais, contribuindo para a fiabilidade global do sistema.

6 Simulação

Para comprovar o funcionamento do circuito, foi realizada uma simulação utilizando o RTL Simulator. Para isso, foi criada uma testbench em VHDL que simula o funcionamento do módulo completo. A testbench gera os sinais de entrada necessários e verifica se as saídas correspondem aos valores esperados.

O circuito foi testado com diferentes entradas e os resultados foram comparados com os esperados. A figura 11 mostra a simulação do circuito completo, onde é possível observar o comportamento das entradas e saídas ao longo do tempo. Os sinais de entrada foram aplicados em diferentes momentos, e as saídas foram monitoradas para verificar se o circuito estava a funcionar corretamente.

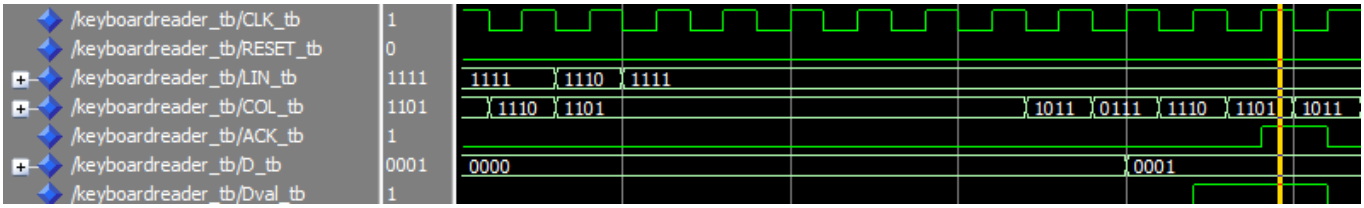

Figura 11: Simulação do circuito completo.

Ao obsercar a simulação, é possível notar que, neste caso, está a ser simulado o premir da tecla "2"visto que a linha ativada é a linha 1 e a coluna 2. Observamos isto analizando os valores de LIN e COL. A tabela de dupla entrada 2, mostra os valores de LIN e COL que representam cada tecla do teclado. A partir da tabela, podemos concluir que o circuito está a funcionar corretamente, uma vez que as saídas correspondem aos valores esperados.

	COL[0]	COL[1]	COL[2]	COL[3]
LIN[0]	1	2	3	A
LIN[1]	4	5	6	B
LIN[2]	7	8	9	C
LIN[3]	*	0	#	D

Tabela 2: Tabela de correspondência entre linhas (LIN) e colunas (COL) do teclado matricial 4x4.

Depois de o utilizador (simulado) premir a tecla "2", o circuito irà validar e apenas será validad não porque o barramento de saída (D) já esteja ativo mas sim porque o sinal de validação (Dval) está ativo. O circuito apenas irá aceitar novas teclas quando receber o acknowledge (ack) do módulo externo de controlo (software). O barramento de saída (D) apenas é alterado quando uma nova tecla é aceite. Assim o valor da última tecla premida é mantido até que uma nova tecla seja premida e validada.

Para facilitar a simulação, a testbench foi configurada para verificar automaticamente se as saídas correspondem aos valores esperados. Caso haja alguma discrepância, a testbench irá gerar um erro indicando que o circuito não está a funcionar corretamente. Isso permite uma verificação rápida e eficiente do funcionamento do circuito, garantindo que ele esteja a operar conforme o esperado.

7 Conclusão

O desenvolvimento do módulo Keyboard Reader permitiu a criação de uma solução robusta e eficiente para a leitura e gestão de entradas provenientes de um teclado matricial 4x4. A arquitetura modular, composta pelos blocos Key Decode, Ring Buffer e Output Buffer, demonstrou ser eficaz na separação de responsabilidades, facilitando a manutenção, o teste e a escalabilidade do sistema.

O bloco Key Decode mostrou-se fundamental para a deteção fiável das teclas premidas, implementando mecanismos de varrimento e debouncing que garantem a precisão na identificação das entradas do utilizador. O Ring Buffer, ao atuar como intermediário entre a leitura e o consumo dos dados, assegurou a integridade e a ordem das informações, mesmo em situações de elevada utilização. Por fim, o Output Buffer permitiu uma comunicação controlada com o módulo consumidor, evitando perdas de dados e promovendo a sincronização entre os diferentes blocos.

A implementação em VHDL de cada componente respeitou as melhores práticas de design digital, recorrendo a máquinas de estados e estruturas de memória adequadas para garantir o correto funcionamento do sistema. Os testes realizados confirmaram a fiabilidade da solução, validando o correto funcionamento dos protocolos de handshake e a ausência de perdas ou duplicações de dados.

Em suma, o módulo Keyboard Reader cumpre todos os requisitos propostos, apresentando uma arquitetura clara, modular e facilmente integrável em sistemas digitais mais complexos. O trabalho realizado contribuiu para o aprofundamento dos conhecimentos em design digital, VHDL e integração de sistemas, preparando os autores para desafios futuros na área da engenharia eletrotécnica e de computadores.