Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

Curso de Engenharia de Computação

# Relatório de Atividades Práticas

*Network-on-Chip com Topologia Mesh e Exibição de Imagem Processada via VGA*

**Alunos:**
Lucas Gabriel Valenti
Jonas Gabriel Fagundes
Lucas Mantovani

**Disciplina:** Arquiteturas de Alto Desempenho
**Professor:** Prof. Dr. Emerson Carlos Pedrino
**Data:** 30 de Junho de 2025

*"Architecture starts when you carefully put two bricks together. There it begins."*
*– Ludwig Mies van der Rohe*

# Sumário

# 1 Introdução

O aumento contínuo na demanda por desempenho computacional impulsionou o desenvolvimento de arquiteturas paralelas e sistemas com múltiplos núcleos de processamento. Nesse contexto, as redes de interconexão tornaram-se componentes fundamentais para garantir comunicação eficiente entre os diversos elementos de processamento, principalmente em arquiteturas baseadas em *chip multiprocessors* (CMPs) e *systems-on-chip* (SoCs). Entre as abordagens mais promissoras está a utilização de *Networks-on-Chip* (NoCs), que oferecem escalabilidade, modularidade e eficiência energética para integração de múltiplos núcleos em uma mesma pastilha.

Este projeto tem como objetivo o desenvolvimento de uma arquitetura baseada em uma topologia em malha bidimensional (*mesh*), na qual cada roteador da rede está acoplado a um processador simples, formando uma malha regular de comunicação. A arquitetura permite o envio e o recebimento de dados entre quaisquer nós da rede, utilizando uma lógica de roteamento do tipo XY e mecanismos internos de arbitragem e chaveamento.

Para validar a funcionalidade da arquitetura, foi implementado um sistema de geração de imagem no qual múltiplos processadores processam dados que, por meio da rede, convergem para um módulo de exibição VGA. Dessa forma, é possível observar o comportamento coordenado dos roteadores, a efetividade da rede de interconexão e a correta operação do sistema como um todo.

# 2 Desenvolvimento Teórico

Nesta seção são apresentados os fundamentos teóricos que embasam o projeto de arquitetura em malha com roteadores acoplados a processadores. Inicialmente, são discutidas diferentes arquiteturas de interconexão, com ênfase na topologia em malha (mesh), escolhida por sua escalabilidade e simplicidade de roteamento. Em seguida, são abordados os conceitos de *Network-on-Chip* (NoC), suas vantagens e a dinâmica de comunicação entre roteadores. Por fim, detalham-se os elementos internos do roteador, os algoritmos de roteamento utilizados e o acoplamento com os processadores, aspectos essenciais para garantir a correta operação do sistema.

## 2.1 Arquiteturas de Interconexão

Em sistemas paralelos compostos por múltiplos processadores, a maneira como esses elementos se comunicam influencia diretamente no desempenho, escalabilidade e consumo de energia da aplicação. As arquiteturas de interconexão definem como os nós do sistema (processadores, memórias, dispositivos de E/S) estão fisicamente e logicamente conectados, impactando o tempo de latência das mensagens, a largura de banda disponível e a tolerância a falhas.

A escolha da topologia de interconexão deve considerar o custo de implementação, a regularidade da estrutura, a eficiência do roteamento e a capacidade de escalonamento da rede para sistemas maiores. A seguir, são apresentadas a topologia Mesh, utilizada neste projeto, e outras topologias comuns em arquiteturas paralelas.

### 2.1.1 Topologia Mesh

A topologia Mesh, ou malha bidimensional, é uma das mais utilizadas em arquiteturas de interconexão devido à sua simplicidade, regularidade e boa escalabilidade. Nessa topologia, os nós (roteadores) estão organizados em uma grade 2D, com cada nó conectado aos seus vizinhos nas direções norte, sul, leste e oeste, exceto nos limites da malha.

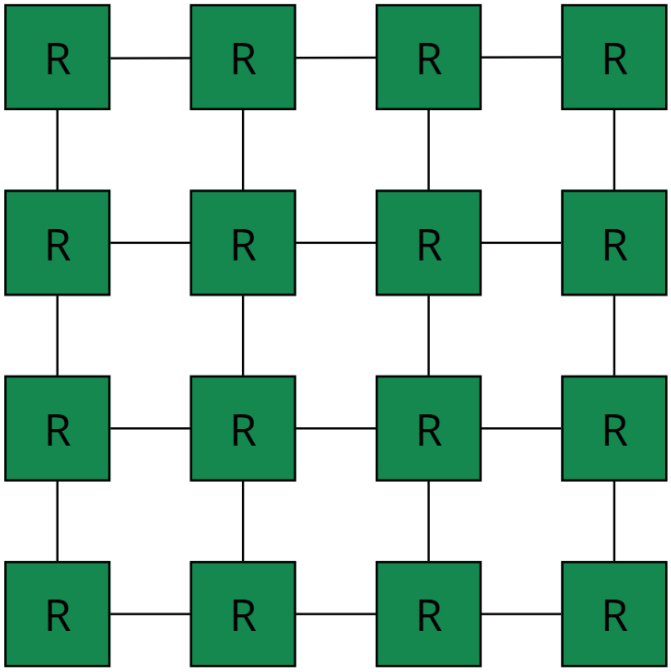


Figura 1: Topologia Mesh: Malha 2D

Cada roteador é responsável por encaminhar pacotes de dados entre os nós, baseando-se em algoritmos de roteamento determinísticos, como o algoritmo XY, que será melhor abordado posteriormente. A topologia Mesh oferece caminhos curtos e previsíveis entre os nós, permitindo comunicação paralela eficiente, especialmente em aplicações onde o tráfego é distribuído uniformemente.

Sua estrutura regular facilita a implementação em hardware e o mapeamento direto em chips, o que a torna uma escolha frequente em projetos de *Networks-on-Chip*.

### 2.1.2 Outras Topologias

Além da Mesh, diversas outras topologias são exploradas em arquiteturas paralelas, cada uma com características específicas:

- **Linha (Linear):** Os nós estão conectados em sequência, formando um caminho único. Apresenta baixa complexidade, mas também baixo desempenho e alta latência em sistemas maiores.

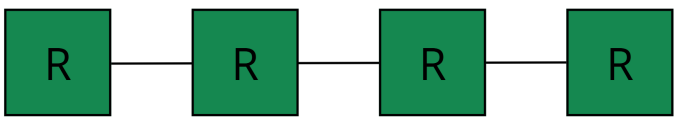


Figura 2: Topologia Linear

- **Anel (Ring):** Cada nó se conecta a dois vizinhos, formando um ciclo fechado. Melhora a regularidade em relação à linha, mas ainda possui limitações de largura de banda e latência.

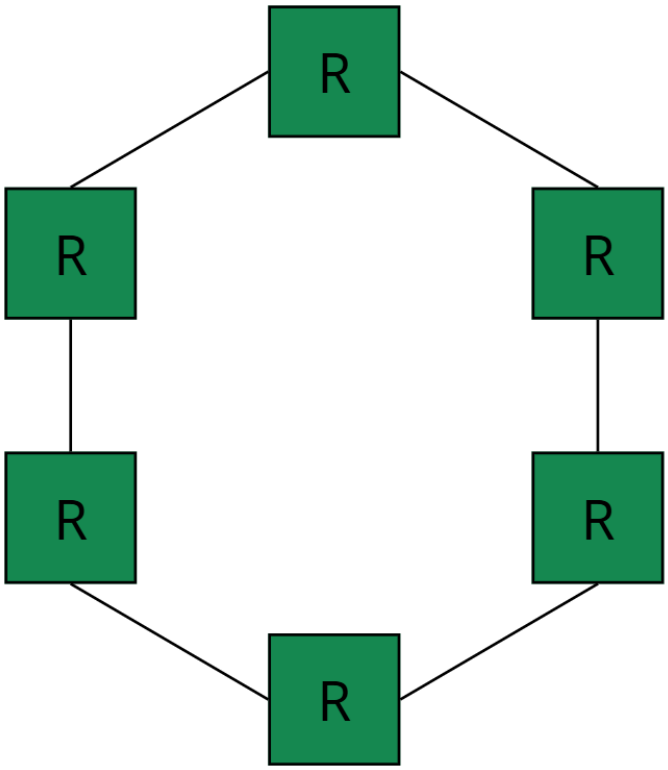


Figura 3: Topologia Anel

- **Árvore (Tree):** Estrutura hierárquica com raiz e ramificações. Apresenta baixa latência para acesso à raiz, mas sofre com gargalos e baixa tolerância a falhas.

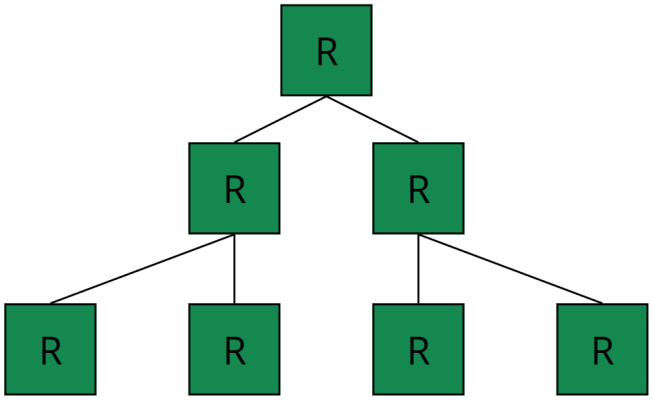


Figura 4: Topologia Árvore

Apesar das vantagens oferecidas por algumas dessas topologias, a escolha pela malha 2D neste projeto se deve à sua implementação direta, facilidade de roteamento e alinhamento com os objetivos didáticos e práticos da disciplina.

## 2.2 Network-on-Chip (NoC)

À medida que o número de núcleos em sistemas integrados aumenta, abordagens tradicionais de comunicação, como barramentos compartilhados e interconexões ponto-a-ponto, tornam-se ineficientes e limitadas. Nesse cenário, a *Network-on-Chip* (NoC) surge como uma solução escalável e

estruturada para permitir a comunicação entre os diversos elementos de um chip multicore. Inspirada em redes de computadores, a NoC adota conceitos como roteadores, topologias regulares e protocolos de comunicação padronizados para organizar o tráfego interno do sistema.

### 2.2.1 Conceitos e Benefícios

Uma NoC é composta por uma rede de roteadores interconectados, onde cada roteador está tipicamente acoplado a um núcleo de processamento, memória ou outro periférico. Essa rede forma uma infraestrutura de comunicação que permite a troca de dados entre diferentes partes do sistema, de forma paralela e coordenada.

Entre os principais benefícios da utilização de NoCs, destacam-se:

- **Escalabilidade:** A estrutura modular da NoC permite a expansão para dezenas ou centenas de núcleos, sem necessidade de reestruturações profundas.
- **Paralelismo de Comunicação:** Diversas mensagens podem trafegar simultaneamente em diferentes caminhos da rede, aumentando o throughput global.
- **Previsibilidade de Desempenho:** Com topologias regulares e algoritmos de roteamento determinísticos, é possível estimar a latência de comunicação.
- **Redução de Congestionamento:** A divisão do tráfego entre múltiplos roteadores e caminhos ajuda a evitar gargalos comuns em barramentos compartilhados.
- **Organização Física:** A estrutura regular da rede facilita o mapeamento físico em silício, otimizando o uso da área e da energia.

### 2.2.2 Comunicação entre Roteadores

A comunicação em uma NoC ocorre por meio do envio de pacotes de dados que trafegam entre os roteadores até atingirem seu destino. Cada pacote é composto, geralmente, por um cabeçalho com informações de roteamento (como coordenadas de destino) e por uma carga útil com os dados a serem transmitidos.

Os roteadores são responsáveis por receber os pacotes por uma ou mais portas de entrada, armazená-los temporariamente em filas (tipicamente FIFOs), tomar decisões baseadas no algoritmo de roteamento e encaminhá-los para as portas de saída adequadas. Esse processo pode envolver mecanismos de arbitragem, para resolver conflitos de múltiplos pacotes competindo por uma mesma saída.

A eficiência da comunicação entre os roteadores depende da topologia adotada, do algoritmo de roteamento utilizado e do controle de fluxo implementado. No projeto aqui descrito, os roteadores comunicam-se de forma síncrona, utilizando sinais de controle para coordenar o envio e o recebimento dos pacotes, garantindo integridade e ordenação dos dados na malha 2D.

## 2.3 Roteadores em NoC

Os roteadores são os elementos centrais de uma *Network-on-Chip*, responsáveis por intermediar o tráfego de dados entre os diversos nós da rede. Sua estrutura deve permitir o recebimento, armazenamento temporário, decisão de encaminhamento e transmissão de pacotes de forma eficiente e coordenada. Em redes regulares como a topologia Mesh, o roteador desempenha um papel crítico no desempenho e previsibilidade do sistema como um todo.

### 2.3.1 Estrutura Interna

A estrutura básica de um roteador em NoC inclui cinco componentes principais:

- **Portas de Entrada e Saída:** Interfaces físicas e lógicas para comunicação com os roteadores vizinhos (Norte, Sul, Leste, Oeste) e com o elemento local (processador).
- **Filas de Entrada (FIFOs):** Cada porta de entrada é conectada a uma fila que armazena temporariamente os pacotes até que possam ser roteados.

- **Módulo de Direção (Routing Logic):** Responsável por analisar o cabeçalho do pacote e determinar a direção de saída apropriada com base no destino.
- **Arbitrador:** Resolve conflitos quando múltiplos pacotes solicitam acesso à mesma porta de saída.
- **Switch (Crossbar):** Permite a comutação dos pacotes entre as portas de entrada e saída, conforme definido pela lógica de controle.
- **Controlador:** Módulo responsável por gerenciar todos esses componentes garantindo que a lógica do roteador funcione.

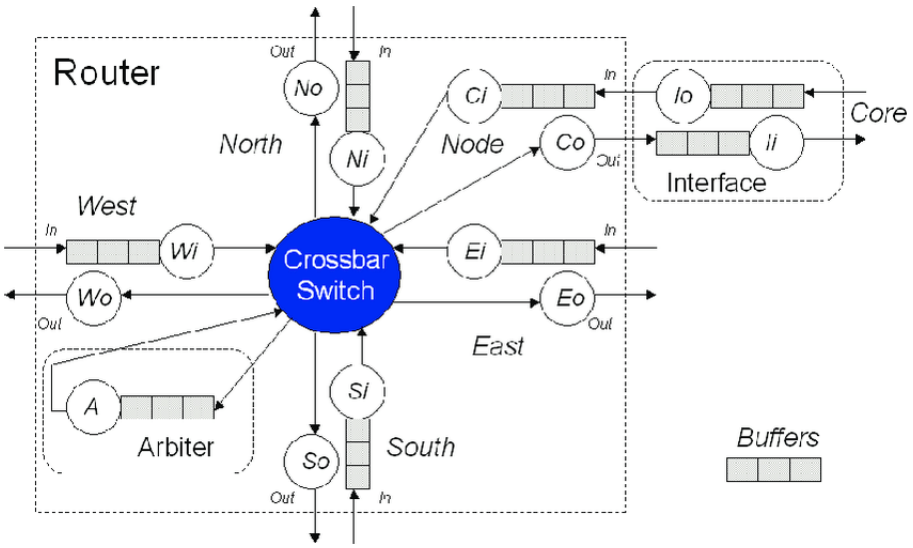


Figura 5: Estrutura do Roteador (Exemplo)

### 2.3.2 Algoritmo de Roteamento XY

Neste projeto, o algoritmo de roteamento adotado é o determinístico do tipo XY, amplamente utilizado em topologias em malha 2D por sua simplicidade e previsibilidade.

O algoritmo XY opera em duas fases:

1. Primeiramente, o pacote é deslocado no eixo X (horizontal), movendo-se para a esquerda ou direita até que a coordenada X do destino seja alcançada.
2. Em seguida, o deslocamento ocorre no eixo Y (vertical), subindo ou descendo até que a coordenada Y do destino seja atingida.

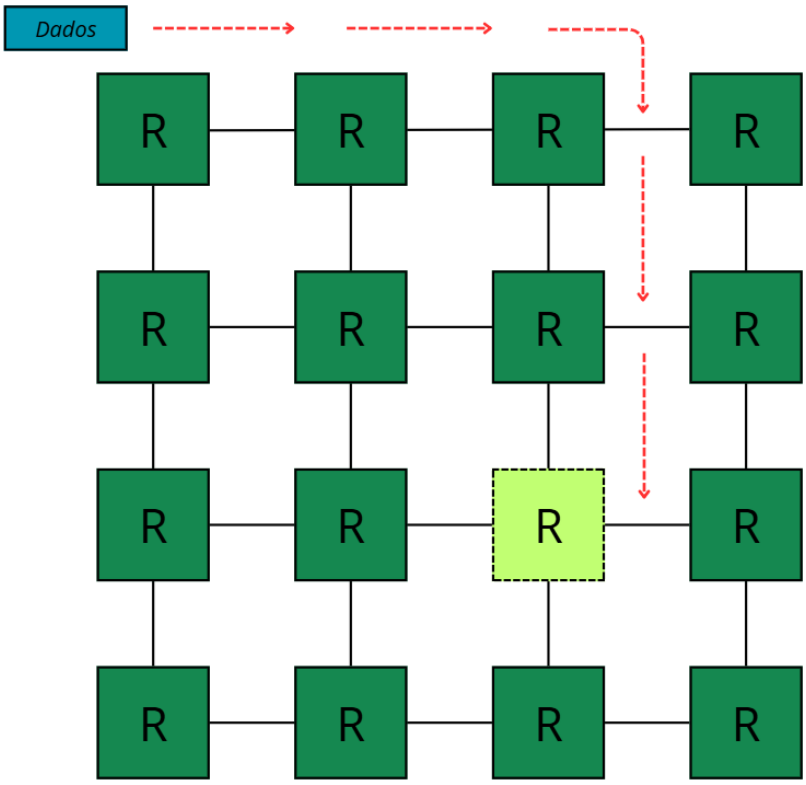


Figura 6: Exemplo: Roteamento XY

Essa estratégia evita ciclos e é livre de deadlocks, desde que implementada com controle de fluxo adequado. Além disso, seu comportamento determinístico permite prever o caminho que um pacote irá percorrer na malha, o que é útil para análise de desempenho e verificação formal.

### 2.3.3 Técnicas de Arbitragem

A arbitragem é necessária quando múltiplos pacotes competem por acesso simultâneo a uma mesma porta de saída. Isso pode ocorrer, por exemplo, quando dois pacotes provenientes de portas diferentes desejam ser encaminhados para o mesmo vizinho.

Neste projeto, foi adotada uma técnica de arbitragem por prioridade fixa, na qual cada porta de entrada possui uma ordem de precedência predeterminada. Quando múltiplos pacotes competem por uma mesma saída, vence aquele cuja entrada tem maior prioridade segundo essa ordem. Embora simples de implementar, essa abordagem pode levar à inanição de entradas com baixa prioridade em situações de tráfego intenso, o que não é o caso aqui.

Outras estratégias de arbitragem existentes na literatura incluem prioridade rotativa, prioridade com envelhecimento, e arbitragens baseadas em token ou análise de congestionamento. A escolha da técnica influencia diretamente a equidade, o atraso médio dos pacotes e a utilização dos recursos do roteador.

## 2.4 Processadores e Comunicação

Em arquiteturas baseadas em *Network-on-Chip*, os processadores representam os nós ativos que executam operações sobre os dados que trafegam pela rede. No projeto desenvolvido, cada processador está acoplado a um roteador da malha 2D e é responsável por realizar uma operação específica: a inversão do valor de um pixel binário recebido. Essa abordagem simples, mas eficaz, permite demonstrar o funcionamento do sistema de comunicação e processamento distribuído em rede.

### 2.4.1 Acoplamento Processador-Roteador

Cada processador se conecta diretamente ao roteador por meio de uma interface dedicada, utilizando sinais para coordenar a recepção e o envio de pacotes binários. O roteador trata o processador como uma porta local, permitindo que ele envie e receba pacotes de forma análoga às direções cardeais da malha.

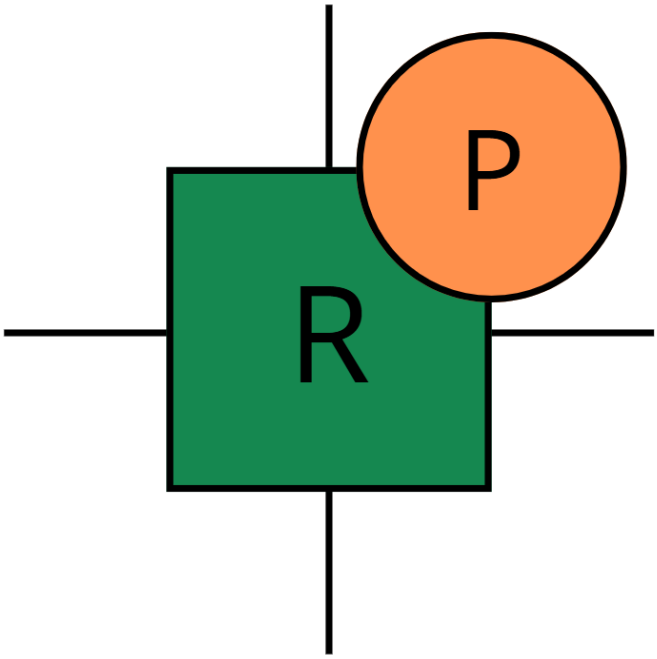


Figura 7: Roteador com Processador Acoplado

Essa interface é composta por sinais de controle e dados (no caso, o bit binário), e opera de forma síncrona com o clock do sistema. O roteador encaminha ao processador apenas os pacotes cujo campo de destino coincide com sua posição na malha.

## 2.5 Imagem Binária em VGA

A exibição da imagem binária processada foi realizada por meio de um monitor conectado via interface VGA (Video Graphics Array). Este método permite representar visualmente o conteúdo da memória processada em tempo real, o que é essencial para validar o pipeline morfológico e a etapa de detecção de bordas. A seguir, detalham-se os princípios fundamentais do funcionamento de um pixel e as características do cabo VGA.

### 2.5.1 Funcionamento de um Pixel

Em um monitor VGA, a imagem é formada por uma grade de pixels atualizada sequencialmente linha a linha (varredura raster). Cada pixel corresponde a uma coordenada única $(x, y)$ e é ativado de acordo com os sinais de sincronização horizontal (HSYNC) e vertical (VSYNC), além dos sinais de cor (vermelho, verde e azul).

No caso de uma **imagem binária**, cada pixel possui apenas dois estados possíveis: *ativo (branco)* ou *inativo (preto)*. No sistema implementado neste projeto, essa ativação é controlada por um bit da memória que representa o valor do pixel. Durante a varredura da tela:

- Se o valor do pixel for 1, os três canais RGB recebem sinal alto (cor branca);
- Se o valor for 0, os três canais permanecem baixos (cor preta).

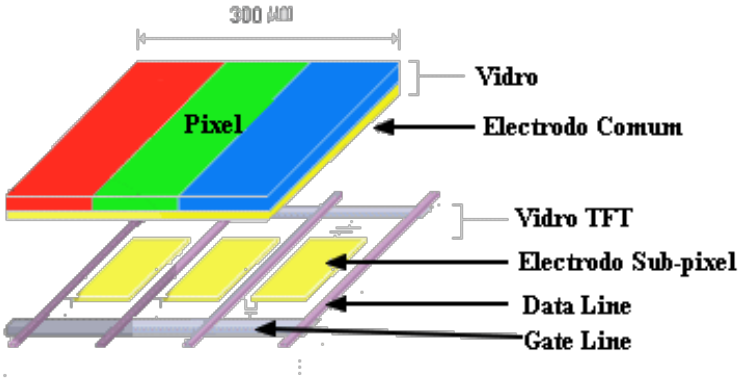


Figura 8: Partes de um Pixel

A lógica responsável por isso compara a posição atual de varredura com o endereço de memória correspondente, determinando a saída dos sinais RGB. Assim, a imagem binária é exibida em tempo real, sincronizada com os sinais VGA.

### 2.5.2 O Cabo VGA

O cabo VGA é um meio de transmissão analógico composto por **15 pinos**, entre os quais destacam-se os sinais de vídeo (RGB) e os sinais de sincronização horizontal e vertical. Os principais pinos utilizados no projeto foram:

- **HSYNC (pino 13):** sinal de sincronização horizontal;
- **VSYNC (pino 14):** sinal de sincronização vertical;
- **R, G, B (pinos 1, 2, 3):** canais de vídeo analógicos — controlam a cor exibida em cada pixel.

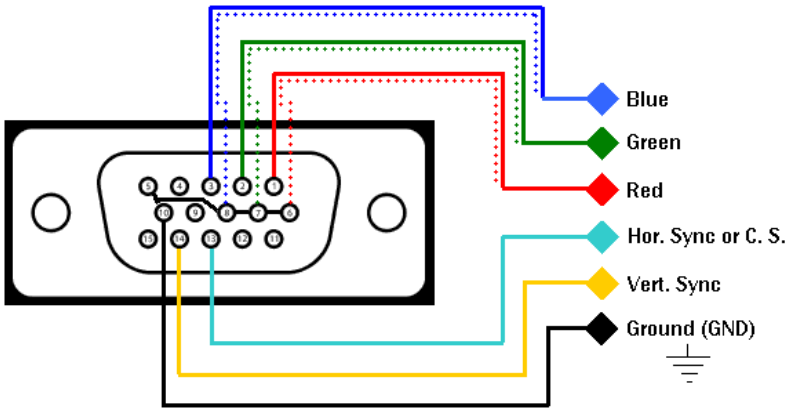


Figura 9: Pinos do Cabo VGA

Na implementação em FPGA, os sinais HSYNC e VSYNC são gerados por um contador de tempo configurado para atender à resolução desejada (por exemplo, 640×480 a 60 Hz). Os sinais de cor são controlados por meio de lógica combinacional, com base no valor do pixel na memória.

Essa estrutura permite exibir a imagem binária diretamente no monitor VGA, sem a necessidade de buffers intermediários ou processamento adicional. É uma abordagem eficiente, determinística e compatível com aplicações embarcadas de baixo custo.

# 3 Desenvolvimento Prático

Esta seção apresenta a implementação prática da arquitetura em malha proposta, detalhando os módulos que compõem o roteador, a organização sistêmica dos elementos na malha 2D e os periféricos utilizados para visualização do resultado. A implementação foi desenvolvida em linguagem Verilog e estruturada de forma modular, permitindo a reutilização e o teste independente de cada componente.

Inicialmente, são descritos os blocos internos que compõem o roteador, incluindo os mecanismos de controle de fluxo e comutação. Em seguida, são apresentados os módulos responsáveis pela montagem da rede como um todo, incluindo o mapeamento espacial dos roteadores, os processadores acoplados e os módulos de entrada e saída de dados. Por fim, são abordados os periféricos de exibição utilizados para visualizar os resultados do processamento distribuído.

## 3.1 Organização Modular do Roteador

O roteador foi implementado de forma modular, com componentes independentes que interagem por meio de sinais de controle bem definidos. Essa abordagem favorece a clareza do projeto, facilita a simulação individual dos blocos e permite escalabilidade para redes maiores. A seguir, são descritos os principais módulos internos que compõem o roteador.

### 3.1.1 Fila FIFO

Cada porta de entrada do roteador é associada a uma fila FIFO (First-In, First-Out), responsável por armazenar temporariamente os pacotes recebidos enquanto aguardam encaminhamento. Essas filas evitam a perda de dados quando múltiplos pacotes chegam simultaneamente ao roteador ou quando a porta de saída desejada está ocupada.

```
1  // módulo fila FIFO (first-in, first-out)
2  module fifo #(
3    parameter DATA_W = 14,          // largura dos dados
4    parameter DEPTH = 4)(           // número de posições da fila
5    input clk, rst,                // sinal de clock e reset
6    input wr_en, rd_en,             // sinal para habilitar escrita e
       habilitar leitura
7    input [DATA_W-1: 0] data_in,      // entrada de dados
8    output full, empty,             // sinal de fila cheia e fila vazia
9    output reg [DATA_W-1: 0] data_out); // saída dos dados
10
11   reg [DATA_W-1:0] mem [0:DEPTH-1];   // fila de fato / memória
12   reg [$clog2(DEPTH)-1:0] w_ptr, r_ptr; // ponteiro de escrita e leitura
13   reg [$clog2(DEPTH):0] count;        // contador
14   assign empty = (count == 0);        // define o sinal de fila vazia
15   assign full = (count == DEPTH);     // define o sinal de fila cheia
16
17   // sempre na borda positiva do clock ou reset
18   always @(posedge clk or posedge rst) begin
19     // reset está ativado
20     if (rst) begin
21       // zera ponteiros de escrita e leitura
22       w_ptr <= 0;
23       r_ptr <= 0;
24       // zera contador
25       count <= 0;
26       // zera saída
27       data_out <= 0;
28     end
29     // reset não está ativado
30     else begin
31       // se leitura foi solicitada e fila não está vazia
32       if (rd_en && !empty) begin
```

```
        // joga informação na saída
        data_out <= mem[r_ptr];
        // atualiza contador e ponteiro de leitura
        r_ptr <= r_ptr + 1;
        count <= count - 1;
      end
      // se escrita foi solicitada e fila não está cheia
      if (wr_en && !full) begin
        // grava novos dados na fila
        mem[w_ptr] <= data_in;
        // atualiza contador e ponteiro de escrita
        count <= count + 1;
        w_ptr <= w_ptr + 1;
      end
    end
  end

endmodule
```

Listing 1: Código Verilog Fila FIFO

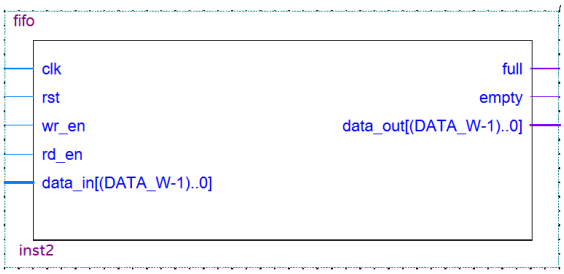


Figura 10: Módulo Fila FIFO

A FIFO foi implementada com um buffer circular e ponteiros de leitura e escrita, permitindo enfileirar e desenfileirar pacotes de forma síncrona. A lógica de controle garante que as operações de escrita e leitura respeitem os limites do buffer, evitando sobreposição ou leitura de dados inválidos.

### 3.1.2 Módulo de Direção

O módulo de direção é responsável por analisar o cabeçalho de cada pacote e determinar a direção de saída com base nas coordenadas de destino. No projeto, foi adotado o algoritmo de roteamento XY, que primeiro resolve o deslocamento no eixo horizontal (X) e depois no eixo vertical (Y).

```
// módulo de direção
module direcao_xy #(
  parameter CIMA = 3'b000,        // sinal que informa CIMA como destino
  parameter BAIXO = 3'b001,         // sinal que informa BAIXO como destino
  parameter ESQUERDA = 3'b010,      //sinal que informa ESQUERDA como
      destino
  parameter DIREITA = 3'b011,       // sinal que informa DIREITA como
      destino
  parameter LOCAL = 3'b100)(      // sinal que informa LOCAL como destino
  input [1:0] meu_x, meu_y,         // informa coordenada X desse roteador
  input [1:0] x_destino, y_destino, // informa coordenada Y desse roteador
  input pronto, empty,          // sinal de pronto (pacote já foi
      processado) e empty (fila vazia)
  output [2:0] direcao);          // direção selecionada

  // define a direção assumindo que pacote ainda não foi processado
  assign direcao_normal = (x_destino > meu_x) ? DIREITA :
```

```
                    (x_destino < meu_x) ? ESQUERDA :
                    (y_destino > meu_y) ? BAIXO :
                    (y_destino < meu_y) ? CIMA : LOCAL;

  // define a direção assumindo que pacote já foi processado
  assign direcao_final = (x_destino < 3) ? DIREITA :
                  (y_destino < 3) ? BAIXO : DIREITA;

  // escolhe a direção com base no bit de confirmação de processamento
  assign direcao_escolhida = pronto ? direcao_final : direcao_normal;

  // se a fila está vazia define joga sinal inválido na saída
  assign direcao = empty ? 3'b111 : direcao_escolhida;

endmodule
```

Listing 2: Código Verilog Módulo de Direção

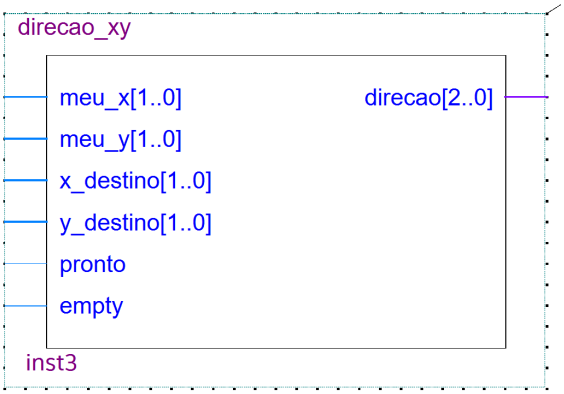


Figura 11: Módulo de Direção

Com base na posição atual do roteador e no destino do pacote, o módulo gera sinais de solicitação para a porta de saída correspondente (Cima, Baixo, Esquerda, Direita ou Local). A lógica é simples e determinística, favorecendo previsibilidade e facilidade de verificação.

Além disso, é importante destacar que, além da informação de destino do pacote, o módulo de direção verifica também extrai do pacote um bit de confirmação para informar se o pacote já foi processado ou não. Se o pacote já foi processado, os roteadores seguem a lógica de enviá-lo ao roteador final (inferior direito). Quando o pacote chega a esse roteador final, ele é enviado diretamente para a porta direita, onde um módulo coletor (que será abordado posteriormente) se encarrega de lidar com ele.

### 3.1.3 Módulo Arbitrador

Quando múltiplas filas FIFO solicitam acesso à mesma porta de saída, o módulo arbitrador é responsável por decidir qual entrada terá prioridade. A arbitragem implementada neste projeto adota uma política de prioridade fixa, com ordem predeterminada entre as portas, no caso Cima ¿ Baixo ¿ Esquerda ¿ Direita ¿ Local.

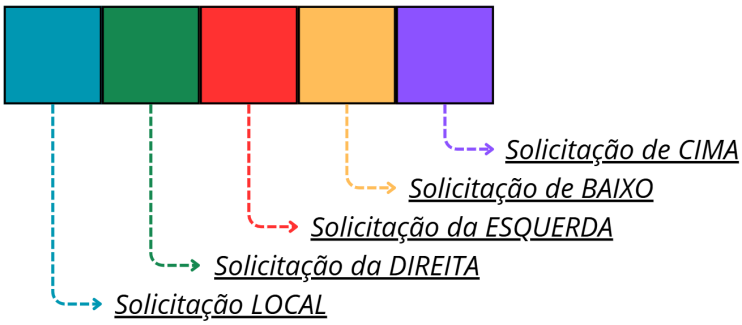


Figura 12: Vetor de Requisições

O arbitrador sabe quais filas querem acessar aquela saída com base em um vetor de requisições, sendo que cada bit do vetor representa a solicitação de uma das filas.

```
// módulo arbitrador
module arbitrador(
  input [4:0] req,    // entrada do vetor de requisições
  output [4:0] grant); // saída com vetor de garantia

  assign grant = req[4] ? 5'b10000 : // autorização para entrada de CIMA
           req[3] ? 5'b01000 : // autorização para entrada de BAIXO
           req[2] ? 5'b00100 : // autorização para entrada da ESQUERDA
           req[1] ? 5'b00010 : // autorização para entrada da DIREITA
           req[0] ? 5'b00001 : // autorização para entrada LOCAL
                 5'b00000;  // nenhuma autorização concedida

endmodule
```

Listing 3: Código Verilog Módulo Arbitrador

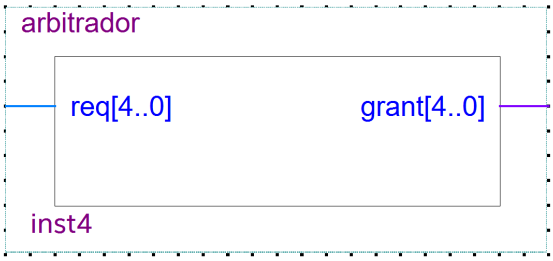


Figura 13: Módulo Arbitrador

A decisão do arbitrador é transmitida ao módulo controlador, que, com base nesse sinal, vai autorizar a comutação apenas da entrada selecionada. Essa abordagem reduz a complexidade da implementação, embora possa introduzir assimetrias no uso da largura de banda da rede em cenários específicos.

### 3.1.4 Switch Crossbar

O switch crossbar realiza a comutação dos pacotes entre as portas de entrada e saída do roteador. Ele funciona como uma matriz de conexões controlada, na qual cada entrada pode ser conectada dinamicamente a uma saída, desde que não haja conflito.

```
// módulo switch crossbar
module switch_crossbar #(
```

```
3   parameter DATA_W = 14)(                                    // tamanho
        dos pacotes
4   input [DATA_W-1:0] cima_in, baixo_in, esquerda_in, direita_in, core_in,
            // saída de cada uma das filas
5   input [2:0] sel_cima, sel_baixo, sel_esquerda, sel_direita, sel_core,
          // sinal seletor de cada saída do roteador
6   output [DATA_W-1:0] cima_out, baixo_out, esquerda_out, direita_out,
        core_out);  // saídas do roteador
7
8   // interpretação dos sinais seletores
9   localparam SEL_CIMA = 3'b000;
10  localparam SEL_BAIXO = 3'b001;
11  localparam SEL_ESQUERDA = 3'b010;
12  localparam SEL_DIREITA = 3'b011;
13  localparam SEL_CORE = 3'b100;
14
15  // comuta a porta da saída de cima
16  assign cima_out = (sel_cima == SEL_CIMA) ? cima_in :
17              (sel_baixo == SEL_CIMA) ? baixo_in :
18              (sel_esquerda == SEL_CIMA) ? esquerda_in :
19              (sel_direita == SEL_CIMA) ? direita_in :
20              (sel_core == SEL_CIMA) ? core_in : {DATA_W{1'b0}};
21
22  // comuta porta da saída de baixo
23  assign baixo_out = (sel_cima == SEL_BAIXO) ? cima_in :
24               (sel_baixo == SEL_BAIXO) ? baixo_in :
25               (sel_esquerda == SEL_BAIXO) ? esquerda_in :
26               (sel_direita == SEL_BAIXO) ? direita_in :
27               (sel_core == SEL_BAIXO) ? core_in : {DATA_W{1'b0}};
28
29  // comuta porta da saída da esquerda
30  assign esquerda_out = (sel_cima == SEL_ESQUERDA) ? cima_in :
31                (sel_baixo == SEL_ESQUERDA) ? baixo_in :
32                (sel_esquerda == SEL_ESQUERDA) ? esquerda_in :
33                (sel_direita == SEL_ESQUERDA) ? direita_in :
34                (sel_core == SEL_ESQUERDA) ? core_in : {DATA_W{1'b0}};
35
36  // comuta porta da saída da direita
37  assign direita_out = (sel_cima == SEL_DIREITA) ? cima_in :
38               (sel_baixo == SEL_DIREITA) ? baixo_in :
39               (sel_esquerda == SEL_DIREITA) ? esquerda_in :
40               (sel_direita == SEL_DIREITA) ? direita_in :
41               (sel_core == SEL_DIREITA) ? core_in : {DATA_W{1'b0}};
42
43  // comuta porta da saída local
44  assign core_out = (sel_cima == SEL_CORE) ? cima_in :
45              (sel_baixo == SEL_CORE) ? baixo_in :
46              (sel_esquerda == SEL_CORE) ? esquerda_in :
47              (sel_direita == SEL_CORE) ? direita_in :
48              (sel_core == SEL_CORE) ? core_in : {DATA_W{1'b0}};
49
50  endmodule
```

Listing 4: Código Verilog Switch Crossbar

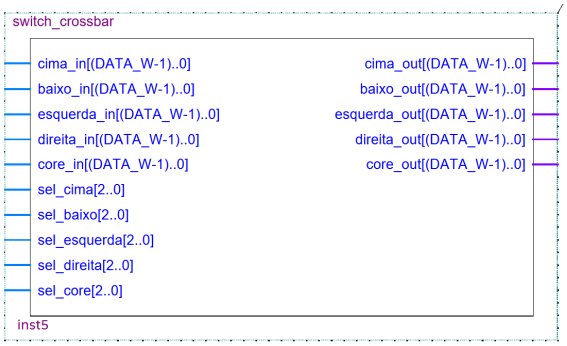


Figura 14: Módulo Switch Crossbar

O controle do crossbar é feito com base nas decisões do módulo de direção e do arbitrador. A estrutura foi implementada de forma combinacional, com multiplexadores controlados pelos sinais de seleção, garantindo baixo tempo de comutação e alta previsibilidade.

### 3.1.5 Módulo de Controle

O módulo de controle é responsável por coordenar o funcionamento dos demais blocos do roteador em ciclos de operação sincronizados. Ele monitora os sinais de solicitação das filas, os sinais de concessão do arbitrador e os estados do switch crossbar.

```
1  // módulo de controle
2  module controle #(
3    parameter DATA_W = 14)(                                          //
         tamanho do pacote
4    input clk, rst,                                                  // sinal
         de clock e reset
5    input [2:0] direcao_cima, direcao_baixo, direcao_esquerda,
         direcao_direita, direcao_core, // sinais recebidos dos módulos de
         direção de cada fila
6    input empty_cima, empty_baixo, empty_esquerda, empty_direita, empty_core,
                  // sinal de fila vazia de cada fila
7    input full_cima, full_baixo, full_esquerda, full_direita, full_core,
                   // sinal de fila cheia de cada fila
8    input [4:0] grant_cima, grant_baixo, grant_esquerda, grant_direita,
         grant_core,       // sinal de encaminhamento de cada fila
9    output reg rd_en_cima, rd_en_baixo, rd_en_esquerda, rd_en_direita,
         rd_en_core,        // sinal para habilitar leitura de pacotes de
         roteadores vizinhos e processador
10   output reg [4:0] req_cima, req_baixo, req_esquerda, req_direita, req_core
         ,             // sinal de requisição de cada fila
11   output reg [2:0] sel_cima, sel_baixo, sel_esquerda, sel_direita, sel_core
         );         // sinal seletor do switch crossbar
12
13   // interpretação dos sinais seletores
14   localparam CIMA = 3'b000;
15   localparam BAIXO = 3'b001;
16   localparam ESQUERDA = 3'b010;
17   localparam DIREITA = 3'b011;
18   localparam CORE = 3'b100;
19
20   // atribuição contínua
21   always @(*) begin
22     // zera todos os vetores de requisição
23     req_cima = 5'b0;
24     req_baixo = 5'b0;
25     req_esquerda = 5'b0;
```

```
    req_direita = 5'b0;
    req_core = 5'b0;

    // se a fila de cima não está vazia
    if (!empty_cima) begin
      // identifica a direção que o pacote quer seguir
      case(direcao_cima)
        // marca a requisição daquela saída se a fila do roteador vizinho n
            ão estiver cheia
        CIMA: if (!full_cima) req_cima[4] = 1;
        BAIXO: if (!full_baixo) req_baixo[4] = 1;
        ESQUERDA: if (!full_esquerda) req_esquerda[4] = 1;
        DIREITA: if (!full_direita) req_direita[4] = 1;
        CORE: if (!full_core) req_core[4] = 1;
      endcase
    end
    // se a fila de baixo não está vazia
    if (!empty_baixo) begin
      // identifica a direção que o pacote quer seguir
      case(direcao_baixo)
        // marca a requisição daquela saída se a fila do roteador vizinho n
            ão estiver cheia
        CIMA: if (!full_cima) req_cima[3] = 1;
        BAIXO: if (!full_baixo) req_baixo[3] = 1;
        ESQUERDA: if (!full_esquerda) req_esquerda[3] = 1;
        DIREITA: if (!full_direita) req_direita[3] = 1;
        CORE: if (!full_core) req_core[3] = 1;
      endcase
    end
    // se a fila da esquerda não está vazia
    if (!empty_esquerda) begin
      // identifica a direção que o pacote quer seguir
      case(direcao_esquerda)
        // marca a requisição daquela saída se a fila do roteador vizinho n
            ão estiver cheia
        CIMA: if (!full_cima) req_cima[2] = 1;
        BAIXO: if (!full_baixo) req_baixo[2] = 1;
        ESQUERDA: if (!full_esquerda) req_esquerda[2] = 1;
        DIREITA: if (!full_direita) req_direita[2] = 1;
        CORE: if (!full_core) req_core[2] = 1;
      endcase
    end
    // se a fila da direita não está vazia
    if (!empty_direita) begin
      // identifica a direção que o pacote quer seguir
      case(direcao_direita)
        // marca a requisição daquela saída se a fila do roteador vizinho n
            ão estiver cheia
        CIMA: if (!full_cima) req_cima[1] = 1;
        BAIXO: if (!full_baixo) req_baixo[1] = 1;
        ESQUERDA: if (!full_esquerda) req_esquerda[1] = 1;
        DIREITA: if (!full_direita) req_direita[1] = 1;
        CORE: if (!full_core) req_core[1] = 1;
      endcase
    end
    // se a fila do processador não está vazia
    if (!empty_core) begin
      // identifica a direção que o pacote quer seguir
      case(direcao_core)
        // marca a requisição daquela saída se a fila do roteador vizinho n
            ão estiver cheia
```

```
82          CIMA: if (!full_cima) req_cima[0] = 1;
83          BAIXO: if (!full_baixo) req_baixo[0] = 1;
84          ESQUERDA: if (!full_esquerda) req_esquerda[0] = 1;
85          DIREITA: if (!full_direita) req_direita[0] = 1;
86          CORE: if (!full_core) req_core[0] = 1;
87        endcase
88      end
89
90      // define o seletor de cima
91      sel_cima = grant_cima[4] ? CIMA :
92              grant_cima[3] ? BAIXO :
93              grant_cima[2] ? ESQUERDA :
94              grant_cima[1] ? DIREITA :
95              grant_cima[0] ? CORE : 3'b111;
96      // define o seletor de baixo
97      sel_baixo = grant_baixo[4] ? CIMA :
98              grant_baixo[3] ? BAIXO :
99              grant_baixo[2] ? ESQUERDA :
100             grant_baixo[1] ? DIREITA :
101             grant_baixo[0] ? CORE : 3'b111;
102     // define o seletor da esquerda
103     sel_esquerda = grant_esquerda[4] ? CIMA :
104               grant_esquerda[3] ? BAIXO :
105               grant_esquerda[2] ? ESQUERDA :
106               grant_esquerda[1] ? DIREITA :
107               grant_esquerda[0] ? CORE : 3'b111;
108     // define o seletor da direita
109     sel_direita = grant_direita[4] ? CIMA :
110               grant_direita[3] ? BAIXO :
111               grant_direita[2] ? ESQUERDA :
112               grant_direita[1] ? DIREITA :
113               grant_direita[0] ? CORE : 3'b111;
114     // define o seletor do processador
115     sel_core = grant_core[4] ? CIMA :
116             grant_core[3] ? BAIXO :
117             grant_core[2] ? ESQUERDA :
118             grant_core[1] ? DIREITA :
119             grant_core[0] ? CORE : 3'b111;
120
121     // define o sinal para ler de cada fila com base nos vetores de
            garantias/autorizações
122     rd_en_cima = grant_cima[4] || grant_baixo[4] || grant_esquerda[4] ||
            grant_direita[4] || grant_core[4];
123     rd_en_baixo = grant_cima[3] || grant_baixo[3] || grant_esquerda[3] ||
            grant_direita[3] || grant_core[3];
124     rd_en_esquerda = grant_cima[2] || grant_baixo[2] || grant_esquerda[2]
            || grant_direita[2] || grant_core[3];
125     rd_en_direita = grant_cima[1] || grant_baixo[1] || grant_esquerda[1] ||
             grant_direita[1] || grant_core[1];
126     rd_en_core = grant_cima[0] || grant_baixo[0] || grant_esquerda[0] ||
            grant_direita[0] || grant_core[0];
127   end
128
129 endmodule
```

Listing 5: Código Verilog Módulo de Controle

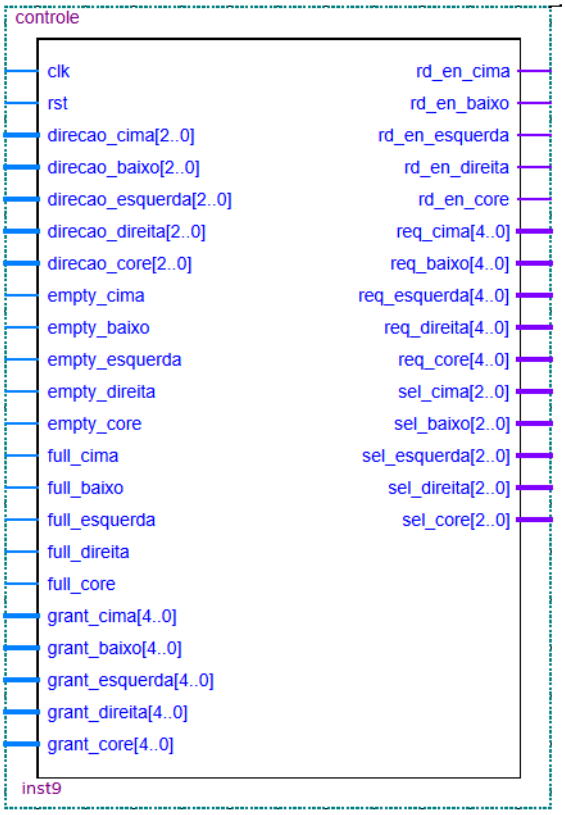


Figura 15: Enter Caption

A lógica implementada nesse módulo assegura que apenas uma transação ocorra por porta de saída a cada ciclo, evitando colisões e garantindo o avanço ordenado dos pacotes. O controle também gerencia os sinais de leitura das FIFOs e de ativação dos canais de saída.

### 3.1.6 Conexão dos Módulos

Todos os módulos do roteador estão interligados por meio de sinais de dados e controle padronizados. Cada porta de entrada possui sua própria FIFO, cujas saídas alimentam o módulo de direção. Este, por sua vez, gera solicitações que são arbitradas e repassadas ao switch crossbar. O módulo de controle orquestra toda a operação, atuando como um gerenciador central.

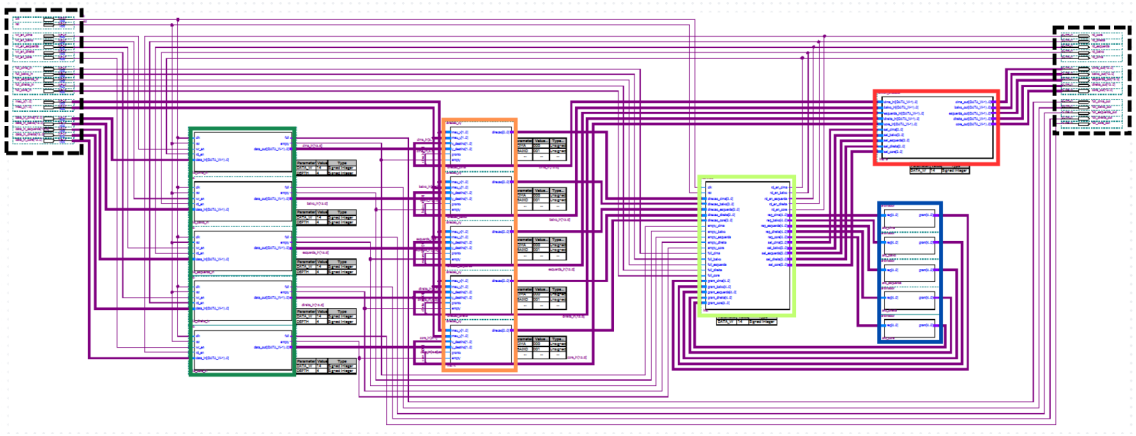

Figura 16: Diagrama Roteador

Na imagem podemos observar as cinco filas (uma para cada entrada) marcadas em verde escuro. Cada fila possui seu módulo de direção (marcados em laranja) e seu módulo arbitrador (marcados em azul). O módulo de controle (verde claro) e o switch crossbar (marcado em vermelho) são únicos.

Essa organização modular facilita a expansão do roteador para incluir novos recursos, como roteamento adaptativo ou suporte a múltiplas prioridades. Além disso, permite a reutilização dos blocos em diferentes topologias ou configurações de rede.

## 3.2 Organização Modular do Sistema

A arquitetura completa foi construída a partir da replicação e interconexão de múltiplos roteadores dispostos em uma malha bidimensional. A estrutura do sistema integra também os processadores acoplados, além de módulos periféricos responsáveis pela entrada (injeção) e saída (coleta) dos dados. O projeto foi modelado de forma parametrizável, permitindo variar a dimensão da malha sem necessidade de reescrita manual de cada conexão.

### 3.2.1 Roteadores: Mapeamento em Malha 2D

Os roteadores foram organizados logicamente em uma matriz de dimensão $4 \times 4$, formando uma topologia Mesh. Cada roteador conecta-se aos seus vizinhos adjacentes nas direções Norte (ou Cima), Sul (ou Baixo), Leste (ou Direita) e Oeste (ou Esquerda), exceto nos limites da matriz, onde as portas correspondentes permanecem desconectadas ou são explicitamente configuradas como inativas. Cada roteador está também conectado a um processador (Local).

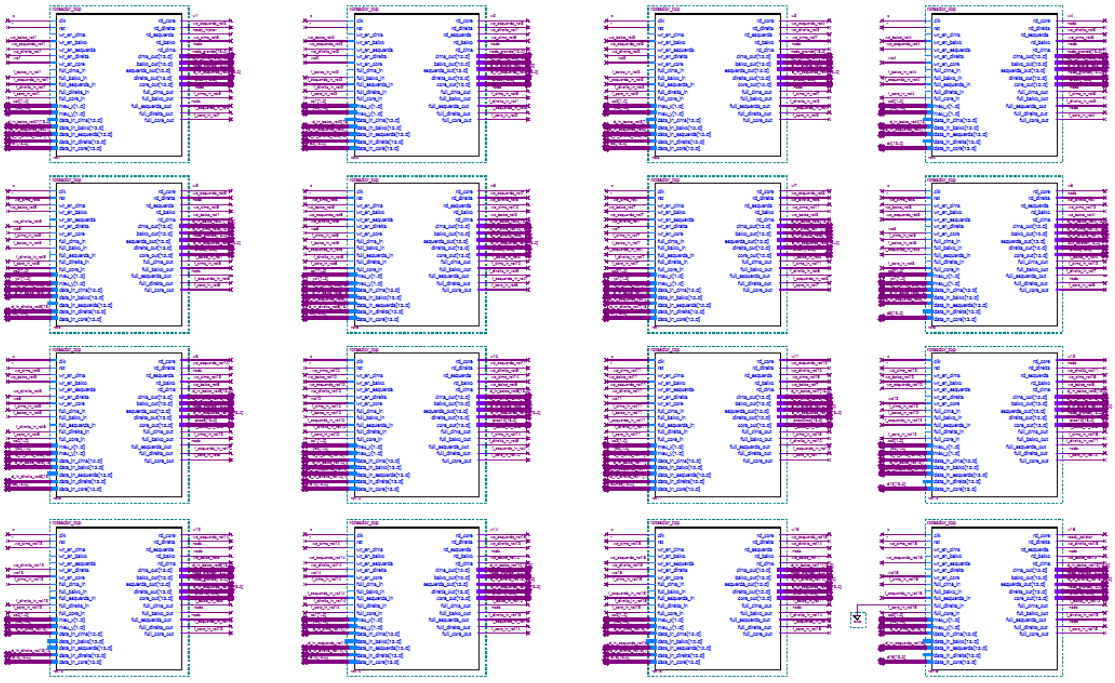

Figura 17: Malha 2D de Roteadores

A comunicação entre roteadores segue o protocolo interno previamente descrito, e os pacotes trafegam pela malha até atingir o roteador cujo endereço (coordenadas X e Y) corresponde ao destino indicado no cabeçalho do pacote.

### 3.2.2 Módulo Injetor

O módulo injetor é responsável por alimentar a rede com os pacotes iniciais que representam os dados a serem processados. Cada pacote inclui:

- Coordenada de destino (X, Y);
- Valor do pixel (0 ou 1).

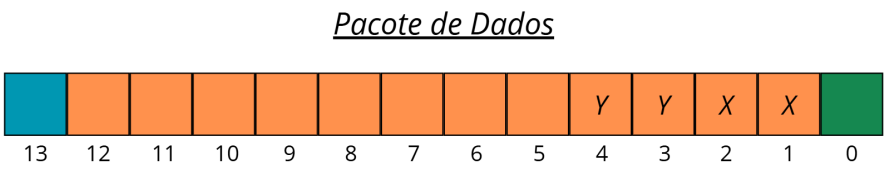


Figura 18: Pacote de Dados

Cada pacote possui 14 bits, sendo o bit 0 (verde) aquele que contém a informação de fato da imagem binária, os bits de 1 até 12 (laranjas) os responsáveis por armazenarem o endereço de memória ao qual aquele pixel de imagem pertence e o bit 13 (azul) aquele responsável por informar se o pacote já foi processado. Vale a pena destacar também que as coordenadas de destino extraídas pelos módulos de direção estão contidas nos bits 1 e 2 (coordenada X), e nos bits 3 e 4 (coordenada Y).

```
// módulo injetor
module injetor_imagem #(
  parameter DATA_W = 14)(          // tamanho do pacote
  input clk, rst,              // sinal de clock e reset
  input ready,                 // habilitador de leitura
  output reg wr_en,              // habilitador de escrita
  output reg [DATA_W-1:0] data_out);  // saída de dados

  reg [0:0] rom [0:4095]; // memória onde é armazenada a imagem

  // carrega a imagem na memória
  initial begin
    $readmemb("chessboard_64x64_clean_flat.txt", rom);
  end

  // registrador para endereço da memória
  reg [11:0] addr;

  // pixel é lido da memória
  wire pixel;
  assign pixel = rom[addr];

  // sempre na subida de clock
  always @(posedge clk) begin
    // habilita escrita
    wr_en <= 1;
    // reset está ativado
    if (rst) begin
      // zera saída
      data_out <= 0;
      // zera endereço da memória
      addr <= 0;
      // desabilita escrita
      wr_en <= 0;
    // reset não está ativado
    end else begin
      // habilitador de leitura está ativado e endereço é válido
      if (ready && (addr < 4096)) begin
        // monta o pacote e joga na saída
        data_out <= {1'b0, addr, pixel};
```

```
        // passa para o próximo endereço da memória
        addr <= addr + 1;
        // habilita escrita
        wr_en <= 1;
      // do contrário
      end else begin
        // zera saída
        data_out <= 0;
        // desabilita escrita
        wr_en <= 0;
      end
    end
  end

endmodule
```

Listing 6: Código Verilog Módulo Injetor

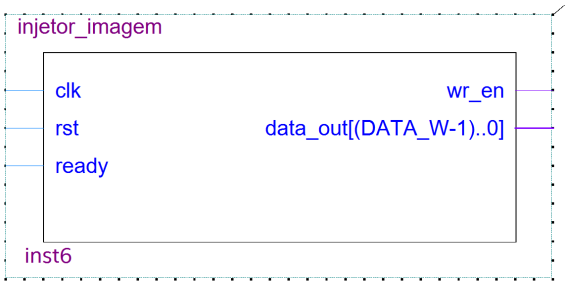


Figura 19: Módulo Injetor

O injetor passa pacote por pacote para o primeiro roteador da malha 2D, no canto superior esquerdo, e este fica responsável por iniciar o roteamento com os demais.

### 3.2.3 Módulo Coletor

O módulo coletor tem como função receber os pacotes de saída produzidos pelos processadores e roteadores, após a inversão do valor dos pixels. Ele está conectado ao roteador correspondente à posição final (mais inferior à direita).

```
// módulo coletor
module coletor_imagem #(
  parameter DATA_W = 14)(     // tamanho do pacote
  input clk, rst,          // sinais de clock e reset
  input wr_en,             // sinal para habilitar escrita
  input [9:0] row, column,    // recebe do VGA qual pixel ele está exibindo
  input [DATA_W-1:0] data_in, // entrada de pacotes
  output wire pixel);       // pixel correspondente na memória

  reg rom_padrao [0:4095]; // memória para a imagem original
  reg imagem [0:4095];     // memória para a exibição
  integer i;            // variável de iteração

  // grava a imagem original e copia na memória de exibição
  initial begin
    $readmemb("chessboard_64x64_clean_flat.txt", rom_padrao);
    for (i = 0; i < 4096; i = i + 1)
      imagem[i] = rom_padrao[i];
  end

```

```
  // registrador para endereço na memória
  wire [11:0] addr;
  assign addr = data_in[12:1]; // extrai endereço do pacote

  // a cada bora de subida de clock
  always @(posedge clk) begin
    // reset está ativado
    if (rst) begin
      // copia a imagem original na memória de exibição
      for (i = 0; i < 4096; i = i + 1)
            imagem[i] <= rom_padrao[i];
    end
    // reset não está ativado
    else begin
      // verifica se escrita está habilitada
      if (wr_en) begin
        // atualiza a memória de exibição com o pixel processado
        imagem[addr] <= data_in[0];
      end
    end
  end

  // saída é o pixel correspondente ao endereço na memória
  assign pixel = imagem[{row[5:0], column[5:0]}];

endmodule
```

Listing 7: Código Verilog Módulo Coletor

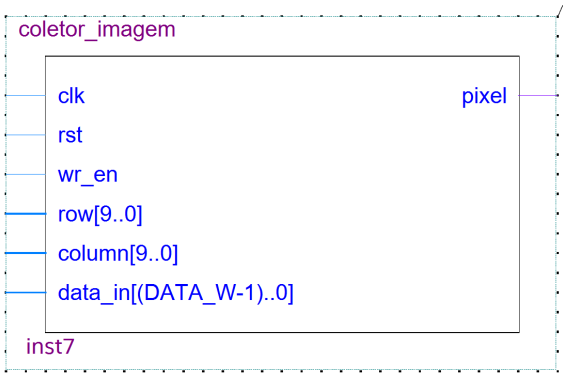


Figura 20: Módulo Coletor

O coletor possui uma memória com a imagem original carregada e, conforme os pacotes chegam até ele, ele atualiza os pixels processados nessa memória, pouco a pouco reconstituindo o conteúdo final da imagem após o processamento. Essa estrutura permite validar o funcionamento correto da rede e observar o resultado do tráfego completo de pacotes.

### 3.2.4 Processadores

Cada roteador da malha possui um processador simples acoplado, responsável por realizar a inversão lógica do bit de dado recebido. O processador opera de forma reativa: ao detectar um pacote destinado à sua coordenada, lê o valor, inverte o bit ($0 \rightarrow 1$ ou $1 \rightarrow 0$), e injeta novamente o pacote na rede. Como o pacote já foi processado, todos os roteadores entendem que a tarefa agora é enviá-lo para o roteador final.

```
// módulo processador
module processador #(
```

```
    parameter FLIT_W = 14 // tamanho do pacote
)(
    input  wire             clk, rst,  // sinal de clock e reset
    input  wire             wr_en_in,  // sinal para habilitar leiturta
    input  wire [FLIT_W-1:0] data_in, // entrada de pacotes
    output reg              wr_en_out, // sinal para habilitar escrita
    output reg [FLIT_W-1:0] data_out  // saída de pacotes
);

  // sempre na borda positiva de clock
  always @(posedge clk) begin
    // reset está ativado
    if (rst) begin
      // zera saída e desabilita escrita
      data_out  <= 0;
      wr_en_out <= 0;

    end
    // reset não está ativado
    else begin
      // leitura está habilitada
      if (wr_en_in) begin
        // gera pacote com bit invertido
        data_out  <= {1'b1, data_in[12:1], ~data_in[0]};
        // habilita escrita
        wr_en_out <= 1;
      end
      // leitura está desabilitada
      else begin
        // desabilita escrita
        wr_en_out <= 0;
      end
    end
  end

endmodule
```

Listing 8: Código Verilog Módulo Processador

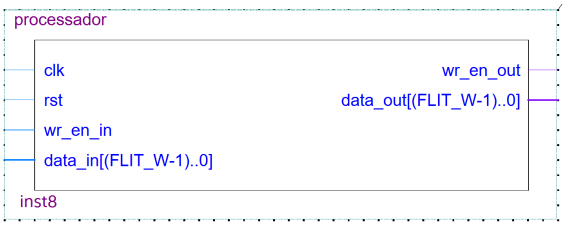


Figura 21: Módulo Processador

A simplicidade dessa operação permite verificar a integridade da comunicação e a sincronização entre os roteadores. Além disso, demonstra a viabilidade do acoplamento processador-roteador em uma malha regular, mantendo desempenho e previsibilidade.

### 3.2.5 Conexão dos Módulos

Os roteadores se conectam formando uma malha 2D de tamanho 4x4. No roteador superior esquerdo entram os pacotes provenientes do injetor, enquanto no roteador inferior direito os pacotes saem para o coletor. Cada roteador possui um processador acoplado a ele.

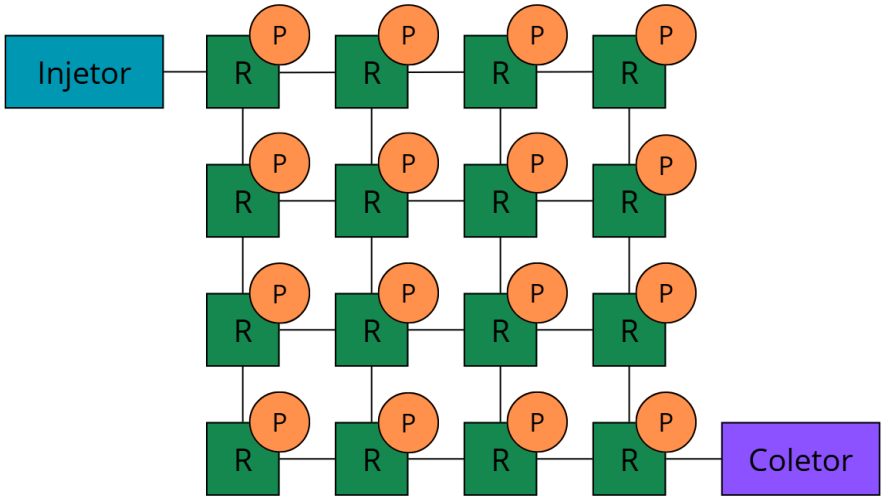


Figura 22: Conexão dos Módulos

O coletor é responsável por fornecer os pixels de imagem para o circuito gerador de imagem.

## 3.3 Geração de Imagem

Após o processamento distribuído dos pixels binários na malha, o resultado final é exibido por meio de um monitor conectado via interface VGA. Para isso, foram utilizados dois módulos principais: o *Módulo Janela*, responsável por identificar a região da tela onde a imagem deve ser renderizada, e o *Módulo VGA*, encarregado de gerar os sinais compatíveis com o padrão de vídeo. Essa etapa permite validar visualmente o correto funcionamento da malha de roteadores e a operação dos processadores acoplados.

### 3.3.1 Módulo de Janela

O módulo de janela tem como função verificar se o pixel atual está localizado dentro da área útil da imagem binária, a qual ocupa apenas uma região da tela VGA. Como a imagem a ser processada pode eventualmente ser configurada para possuir dimensões menores que as do monitor, é necessário restringir o processamento aos pixels válidos.

```
// módulo para definir se
// pixel pertence a janela
module janela(
  // entradas e saídas
  input [9:0]a,
  input [9:0]b,
  output s);

  // se pertencer a janela
  assign s = ((a < 64) & (b < 64));

endmodule
```

Listing 9: Código Verilog Módulo de Janela

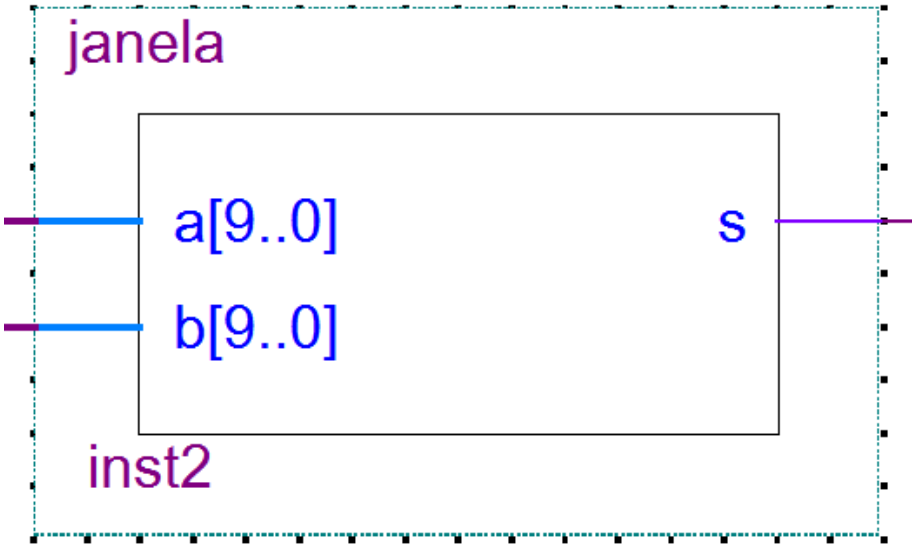


Figura 23: Módulo Janela

Para isso, o módulo recebe como entrada as coordenadas `pixel_column` e `pixel_row`, fornecidas pelo módulo VGA, e compara esses valores com os limites da janela predefinida de visualização. Quando o pixel se encontra dentro desses limites, o módulo emite um sinal de ativação indicando que o dado correspondente pode ser processado e exibido.

Essa verificação é essencial para garantir que os módulos de leitura de memória e exibição no VGA operem apenas sobre a região útil da imagem, evitando leituras inválidas e artefatos na borda da tela.

### 3.3.2 Módulo VGA

Este módulo é responsável por gerar os sinais de controle VGA, incluindo os pulsos de sincronização horizontal (**HSYNC**) e vertical (**VSYNC**), além da ativação dos sinais RGB. Ele também fornece as coordenadas de varredura **(x, y)**, representadas pelas saídas `pixel_row` e `pixel_column`, que determinam a posição atual na tela. Essas coordenadas são utilizadas por outros módulos para buscar o valor correspondente na memória de imagem.

```
1  library IEEE;
2  use  IEEE.STD_LOGIC_1164.all;
3  use  IEEE.STD_LOGIC_ARITH.all;
4  use  IEEE.STD_LOGIC_UNSIGNED.all;
5  -- Module Generates Video Sync Signals for Video Montor Interface
6  -- RGB and Sync outputs tie directly to monitor conector pins
7  ENTITY VGA_SYNC IS
8      PORT( clock_50Mhz, red, green, blue   : IN  STD_LOGIC;
9              red_out, green_out, blue_out, horiz_sync_out,
10             vert_sync_out, video_on, pixel_clock  : OUT STD_LOGIC;
11             pixel_row, pixel_column: OUT STD_LOGIC_VECTOR(9 DOWNTO 0));
12 END VGA_SYNC;
13 ARCHITECTURE a OF VGA_SYNC IS
14     SIGNAL horiz_sync, vert_sync, pixel_clock_int : STD_LOGIC;
15     SIGNAL video_on_int, video_on_v, video_on_h : STD_LOGIC;
16     SIGNAL h_count, v_count :STD_LOGIC_VECTOR(9 DOWNTO 0);
17 --
18 -- To select a different screen resolution, clock rate, and refresh rate
19 -- pick a set of new video timing constant values from table at end of code
       section
20 -- enter eight new sync timing constants below and
21 -- adjust PLL frequency output to pixel clock rate from table
```

```
-- using MegaWizard to edit video_PLL.vhd
-- Horizontal Timing Constants
    CONSTANT H_pixels_across:   Natural := 640;
    CONSTANT H_sync_low:    Natural := 664;
    CONSTANT H_sync_high:     Natural := 760;
    CONSTANT H_end_count:     Natural := 800;
-- Vertical Timing Constants
    CONSTANT V_pixels_down:   Natural := 480;
    CONSTANT V_sync_low:    Natural := 491;
    CONSTANT V_sync_high:     Natural := 493;
    CONSTANT V_end_count:     Natural := 525;
    COMPONENT video_PLL
    PORT
    (
        inclk0    : IN STD_LOGIC  := '0';
        c0      : OUT STD_LOGIC
    );
end component;

BEGIN

-- PLL below is used to generate the pixel clock frequency
-- Uses UP 3's 48Mhz USB clock for PLL's input clock
video_PLL_inst : video_PLL PORT MAP (
        inclk0   => Clock_50Mhz,
        c0   => pixel_clock_int
    );

-- video_on is high only when RGB pixel data is being displayed
-- used to blank color signals at screen edges during retrace
video_on_int <= video_on_H AND video_on_V;
-- output pixel clock and video on for external user logic
pixel_clock <= pixel_clock_int;
video_on <= video_on_int;

PROCESS
BEGIN
    WAIT UNTIL(pixel_clock_int'EVENT) AND (pixel_clock_int='1');

--Generate Horizontal and Vertical Timing Signals for Video Signal
-- H_count counts pixels (#pixels across + extra time for sync signals)
--
--  Horiz_sync  ------------------------------------_----------________
--  H_count     0                 #pixels            sync low      end
--
    IF (h_count = H_end_count) THEN
        h_count <= "0000000000";
    ELSE
        h_count <= h_count + 1;
    END IF;

--Generate Horizontal Sync Signal using H_count
    IF (h_count <= H_sync_high) AND (h_count >= H_sync_low) THEN
        horiz_sync <= '0';
    ELSE
        horiz_sync <= '1';
    END IF;

--V_count counts rows of pixels (#pixel rows down + extra time for V sync
    signal)
--
```

```
82  --  Vert_sync      -----------------------------------------------_______--
    ----------
83  --  V_count         0                          last pixel row      V sync low
          end
84  --
85      IF (v_count >= V_end_count) AND (h_count >= H_sync_low) THEN
86          v_count <= "0000000000";
87      ELSIF (h_count = H_sync_low) THEN
88          v_count <= v_count + 1;
89      END IF;
90
91  -- Generate Vertical Sync Signal using V_count
92      IF (v_count <= V_sync_high) AND (v_count >= V_sync_low) THEN
93          vert_sync <= '0';
94      ELSE
95          vert_sync <= '1';
96      END IF;
97
98  -- Generate Video on Screen Signals for Pixel Data
99  -- Video on = 1 indicates pixel are being displayed
100 -- Video on = 0 retrace - user logic can update pixel
101 -- memory without needing to read memory for display
102     IF (h_count < H_pixels_across) THEN
103         video_on_h <= '1';
104         pixel_column <= h_count;
105     ELSE
106         video_on_h <= '0';
107     END IF;
108
109     IF (v_count <= V_pixels_down) THEN
110         video_on_v <= '1';
111         pixel_row <= v_count;
112     ELSE
113         video_on_v <= '0';
114     END IF;
115
116 -- Put all video signals through DFFs to elminate any small timing delays
    that cause a blurry image
117         horiz_sync_out <= horiz_sync;
118         vert_sync_out <= vert_sync;
119 --    Also turn off RGB color signals at edge of screen during vertical and
     horizontal retrace
120         red_out <= red AND video_on_int;
121         green_out <= green AND video_on_int;
122         blue_out <= blue AND video_on_int;
123
124 END PROCESS;
125 END a;
```

Listing 10: Código do Módulo VGA Fornecido

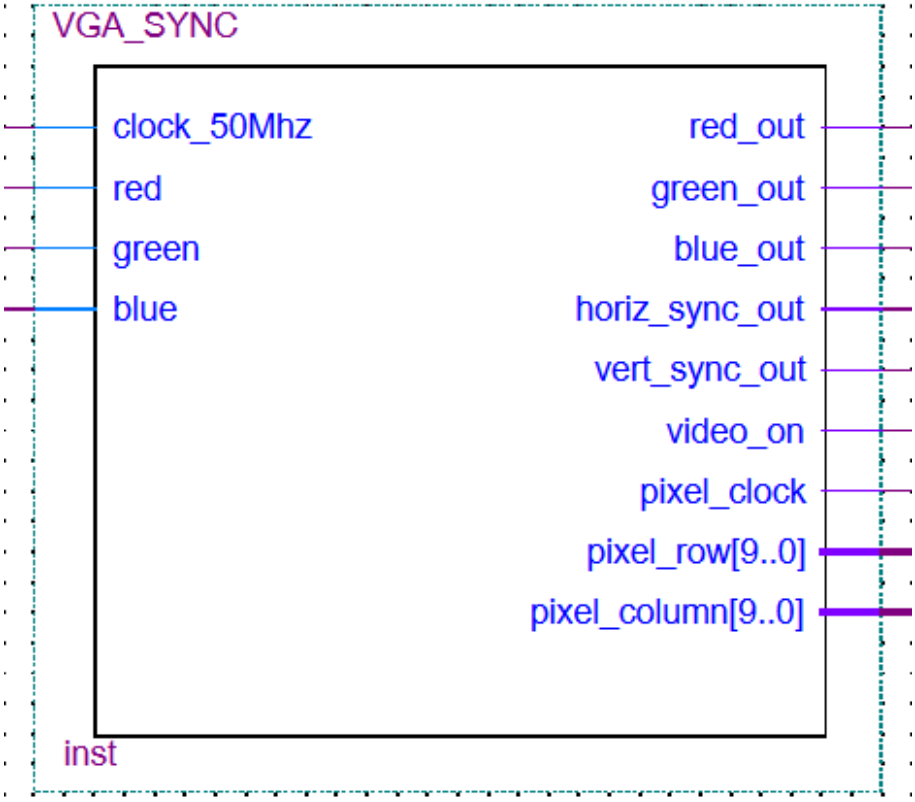


Figura 24: Módulo VGA

Além dos sinais básicos de sincronização e cor, o módulo também fornece:

- `pixel_clock`: sinal utilizado para marcar o instante exato de atualização do pixel, garantindo que as leituras de memória e mudanças nos sinais RGB ocorram de forma sincronizada com a taxa de varredura da tela;
- `video_on`: sinal lógico que indica se o pixel atual se encontra dentro da área visível da tela. Este sinal é essencial para garantir que apenas os dados válidos sejam enviados ao monitor, evitando artefatos fora da área ativa de exibição.

Os sinais de cor (`red`, `green`, `blue`) são ativados apenas quando `video_on` está em nível alto, assegurando uma imagem estável e consistente. O módulo VGA atua, portanto, como a base do sistema de exibição em tempo real do projeto.

# 4 Discussão dos Resultados

A implementação da malha de roteadores acoplados a processadores foi validada por meio da observação do comportamento funcional do sistema completo, com destaque para a visualização da imagem binária processada na interface VGA. A correta exibição da imagem final comprova o funcionamento integrado da rede, dos módulos de roteamento e dos processadores locais.

Durante os testes, foi possível verificar que os pacotes trafegaram adequadamente pela malha, sendo roteados até os nós de processamento e, posteriormente, reencaminhados até o módulo coletor. A inversão dos pixels binários realizada por cada processador pôde ser observada diretamente na imagem resultante, evidenciando que o fluxo de dados e a lógica de controle operaram conforme o esperado.

Do ponto de vista da comunicação, a ausência de travamentos, sobreposição de pacotes ou perdas de dados indica que os mecanismos de controle de fluxo, arbitragem e comutação do roteador

foram eficientes. A estrutura modular dos roteadores também facilitou a identificação e correção de eventuais falhas durante o desenvolvimento.

A latência de propagação dos pacotes — embora não medida com precisão em ciclos de clock — pôde ser inferida pela ordem com que os pixels da imagem eram atualizados no display, revelando que a malha permitiu o processamento distribuído de forma progressiva e coordenada.

Além disso, o uso do Módulo Janela foi essencial para delimitar a região de exibição da imagem no monitor, possibilitando uma renderização limpa e visualmente interpretável dos dados processados. A sincronia entre os módulos VGA, Janela e Coletor demonstrou o alinhamento correto entre os subsistemas gráficos e de comunicação.

Em termos gerais, os resultados obtidos demonstram que a arquitetura proposta é funcional, modular e reproduz com fidelidade os princípios de uma Network-on-Chip baseada em topologia Mesh. O projeto atendeu aos objetivos propostos e serviu como plataforma eficaz para o aprendizado prático de interconexão, roteamento e comunicação paralela em sistemas digitais.

# 5 Conclusões

O projeto desenvolvido ao longo da disciplina permitiu a construção e validação de uma arquitetura baseada em uma rede de interconexão do tipo malha 2D (topologia Mesh), composta por roteadores acoplados a processadores simples. A proposta demonstrou na prática os conceitos de *Network-on-Chip* (NoC), como roteamento determinístico, arbitragem, comutação e comunicação ponto a ponto entre múltiplos nós.

A implementação modular dos roteadores, aliada à replicação controlada para formação da malha, permitiu explorar os desafios típicos de projetos em larga escala, como interconexão, sincronismo e fluxo de dados. A lógica dos processadores locais, ainda que simples, foi suficiente para validar o processamento descentralizado dos dados e o tráfego funcional de pacotes pela rede.

A correta exibição da imagem invertida na interface VGA serviu como verificação empírica do sistema, evidenciando o alinhamento entre os subsistemas de comunicação e visualização. O uso do Módulo Janela, ao delimitar a região ativa da tela, contribuiu para uma renderização limpa e objetiva dos resultados.

Como possíveis melhorias para versões futuras do projeto, destacam-se:

- Implementação de algoritmos de roteamento adaptativos, para maior flexibilidade sob diferentes padrões de tráfego;
- Avaliação formal de desempenho (latência, throughput) em ciclos de clock.

Conclui-se, portanto, que o sistema projetado atendeu aos objetivos estabelecidos, servindo como uma plataforma eficaz para o estudo de arquiteturas paralelas e comunicação em redes intra-chip.

# Referências


[1] *O que é uma malha de dados?*. Amazon AWS. Disponível em: `https://aws.amazon.com/pt/what-is/data-mesh/`. Acesso em: junho de 2025.

[2] DATA SCIENCE ACADEMY. *Arquitetura Data Mesh*. YouTube, 2023. 8 minutos. Disponível em:`https://www.youtube.com/watch?v=rNmRcU9A0O0`. Acesso em: junho de 2025.

[3] CHARLEAUX, Lupa. SHIMABUKURO, Igor. *O que é um roteador? Veja para que serve e como funciona o dispositivo de rede*. Tecnoblog, 2025. Disponível em: `https://tecnoblog.net/responde/o-que-e-um-roteador-veja-para-que-serve-e-como-funciona-o-dispositivo-de-rede/`. Acesso em: junho de 2025.

[4] GOGONI, Ronaldo. *O que é VGA?*. Tecnoblog, 2020. Disponível em: `https://tecnoblog.net/responde/o-que-e-vga/`. Acesso em: junho de 2025.