

Aceleração em Hardware de Visão Computacional utilizando FPGA

Relatório final 3^a Fase Embarcatech

Isaac Martins de Oliveira Braga e Sousa

Fevereiro 2026

1 Resumo

Este trabalho apresenta o desenvolvimento de uma arquitetura digital de aceleração em hardware para processamento de vídeo e reconhecimento de caracteres numéricos manuscritos, operando estritamente na borda (Edge Computing).

O problema central aborda a latência intrínseca e o alto custo computacional de processadores sequenciais e Redes Neurais Convolucionais (CNNs) quando aplicados à inferência de imagens em sistemas embarcados de baixo custo. Para solucionar este gargalo, implementou-se um pipeline de processamento puramente em portas lógicas (linguagem Verilog), sem a utilização de microprocessadores ou memórias externas

. O ambiente de hardware base consistiu em uma FPGA Lattice ECP5 (modelo LFE5U-25F), operando em conjunto com um sensor de imagem CMOS OV2640 para aquisição de dados via protocolo DVP, e um módulo adaptador VGA PS2 Board da Waveshare para a exibição gráfica.

A metodologia foi estruturada de forma modular, englobando o cruzamento de domínios de clock assíncronos, o desenvolvimento de um Color Tracker combinacional para binarização e extração da Região de Interesse (ROI), e a síntese de um classificador estatístico baseado em Correspondência de Molde (Template Matching).

Para mitigar as limitações geométricas deste modelo, implementou-se uma heurística de "Visão Periférica", conferindo tolerância espacial ao sistema. Os resultados práticos de bancada demonstraram a eficácia da arquitetura, que foi capaz de capturar vídeo, processar o rastreamento com sobreposição de interface (UI) gráfica no monitor VGA a 60Hz, e realizar a inferência do dataset MNIST instantaneamente, acionando um display de 7 segmentos sem interrupções por software.

Conclui-se que o uso de paralelismo em FPGA oferece uma alternativa de latência ultrabaixa para visão computacional, provando-se altamente eficiente perante severas restrições de memória interna e consumo lógico.

2 Introdução

2.1 Contextualização

Nos últimos anos, a inteligência artificial (IA) e a visão computacional deixaram de ser conceitos estritamente teóricos para se tornarem pilares estruturais da Quarta Revolução Industrial (Indústria 4.0). Atualmente, a capacidade das máquinas de ”verem” e interpretarem dados visuais é o motor por trás de tecnologias como veículos autônomos, diagnósticos médicos automatizados, robótica avançada e controle de qualidade em linhas de montagem. Segundo projeções da International Data Corporation (IDC), o volume global de dados gerados deve saltar para 175 zettabytes até 2025, sendo uma parcela massiva constituída por fluxos de vídeo e imagem contínuos gerados por sensores e dispositivos IoT (Internet of Things).

Historicamente, o processamento dessa montanha de dados visuais tem sido delegado a processadores de propósito geral (CPUs), placas gráficas (GPUs) ou enviado através de redes de alta velocidade para datacenters na nuvem (Cloud Computing). No entanto, essas abordagens esbarram em ”gargalos” físicos e lógicos severos. O processamento em nuvem introduz latências de rede e vulnerabilidades de conexão que são inaceitáveis em sistemas de tempo real crítico — como o acionamento de um freio de emergência automotivo baseado no reconhecimento de uma placa ou obstáculo. Por outro lado, o processamento local via CPUs tradicionais sofrem do problema clássico em que o processador gasta tempo excessivo trafegando dados sequencialmente entre a unidade lógica e a memória RAM, resultando em ineficiência e elevado consumo energético.

Nesse sentido, uma solução inerente à computação sequencial são os Arranjos de Portas Programáveis em Campo (FPGAs — Field-Programmable Gate Arrays). Diferente de um microprocessador que traduz linhas de código em instruções em fila, uma FPGA permite um molde as portas lógicas e os fios físicos do chip para que funcionem exatamente como o algoritmo exige. Isso possibilita o processamento massivamente paralelo e a criação de pipelines de dados, onde um pixel de imagem pode ser lido de uma câmera, filtrado matematicamente e ter - por exemplo - sua cor rastreada no mesmo ciclo de clock, garantindo uma latência determinística na casa dos nanosegundos e sem a interferência de um sistema operacional.

As perspectivas futuras para a tecnologia apontam para uma descentralização agressiva, migrando a inteligência da nuvem diretamente para o local onde os dados nascem — conceito conhecido como Edge AI (Inteligência Artificial na Borda). Neste cenário em rápida expansão, dispositivos autônomos precisarão classificar imagens localmente, em tempo real, usando baterias limitadas. Devido à sua capacidade única de aliar o altíssimo desempenho de um hardware físico dedicado com a flexibilidade de poder ser reprogramado em campo caso os algoritmos evoluam, a tecnologia FPGA se consolida não apenas como uma alternativa, mas como um caminho vital para o futuro da visão computacional de alta performance.

2.2 Definição do Problema

A implementação de sistemas de visão computacional diretamente em hardware reconfigurável (RTL/Verilog) é um desafio que transcende a simples escrita de algoritmos. Diferente do desenvolvimento em software, onde a infraestrutura de captura e exibição já é fornecida pelo sistema operacional, o projeto em FPGA exige a construção manual de todo o caminho de dados (datapath).

Historicamente, o reconhecimento de caracteres em imagens — como o famoso dataset numérico MNIST — é resolvido com eficiência no estado da arte através de Redes Neurais Convolucionais (CNNs). Contudo, a síntese de uma arquitetura convolucional completa, construída do zero em linguagem de descrição de hardware, esbarra em duas limitações críticas do nosso escopo: a escassez de blocos multiplicadores (DSPs) e memória na FPGA utilizada, e a severa limitação de tempo de desenvolvimento.

Diante dessa realidade, a questão central deste trabalho foi readequada: **Como dominar e construir um pipeline completo de processamento de vídeo em hardware puro, culminando em um modelo prático de classificação de dígitos, respeitando os prazos e os limites físicos da placa?**

Para solucionar essa questão principal, o problema precisou ser quebrado em uma escala evolutiva de complexidade técnica. A curva de aprendizado e desenvolvimento exigiu vencer os seguintes estágios:

1. **Geração de Sinal de Vídeo:** O primeiro desafio arquitetural foi dominar o protocolo VGA, implementando temporizadores horizontais e verticais para exibir cores sólidas em um monitor, provando o controle dos pinos de saída.
2. **Gerenciamento de Memória:** O passo seguinte foi compreender a leitura assíncrona/síncrona de blocos de memória, gravando uma imagem estática em uma memória ROM interna e varrendo seus endereços para exibi-la na tela VGA.
3. **Aquisição de Dados Externos:** A complexidade escalou drasticamente ao integrar o sensor de imagem OV2640. O problema passou a ser o cruzamento de múltiplos domínios de clock e a decodificação do protocolo DVP em tempo real, gravando o fluxo de pixels da câmera em um framebuffer na RAM.
4. **Processamento Intermediário (Color Tracker):** Com o fluxo da câmera estabelecido, o desafio seguinte foi atuar matematicamente sobre os pixels. Implementou-se um Color Tracker para binarizar imagens e identificar coordenadas limites (bounding boxes) de objetos coloridos, servindo como a prova de conceito de processamento em tempo real.
5. **A Classificação Final (MNIST):** O topo dessa escalada técnica consistiu em utilizar a área rastreada para alimentar um classificador de dígitos

sem usar um microprocessador. Devido à inviabilidade temporal e de recursos para uma CNN, optou-se por um modelo de Correspondência de Molde Euclidiana (Template Matching).

Essa abordagem iterativa apresentou grandes méritos educacionais e operacionais, resultando em um sistema extremamente rápido e sem consumo de CPU. Contudo, essa simplificação matemática carrega limitações intrínsecas: ao desconsiderar as convoluções, o sistema construído não é invariante a escala ou rotação. Pior do que isso, a dependência geométrica torna o classificador altamente vulnerável ao deslocamento humano — se o usuário desenha o dígito milímetros fora do centro esperado, o sistema falha ao comparar os pixels brutos.

Portanto, o problema prático que este trabalho se propõe a resolver é a construção de todo esse pipeline evolutivo do zero, mitigando em hardware as fragilidades de alinhamento espacial impostas pelo Template Matching rudimentar. A validação dessa arquitetura combinacional robusta pavimenta o caminho tecnológico necessário para que, em trabalhos futuros, com mais recursos lógicos e tempo, as portas lógicas do classificador estatístico possam ser substituídas por blocos operacionais de uma CNN completa.

2.3 Objetivos do trabalho

Este trabalho não tem como objetivo desenvolver uma Inteligência Artificial de aprendizado profundo (como Redes Convolucionais) capaz de reconhecer caracteres em qualquer escala, ângulo ou condição de iluminação extrema. Em vez disso, a solução proposta delimita-se à criação de uma prova de conceito focada na aceleração em nível de hardware (Edge Computing), processando o fluxo de dados bruto de uma câmera em tempo real para extrair características geométricas de dígitos centralizados, utilizando recursos lógicos estritamente limitados.

Neste contexto, os objetivos do projeto dividem-se em um objetivo geral, que encapsula a solução completa, e objetivos específicos, que representam os marcos arquiteturais necessários para a construção do pipeline de vídeo.

2.3.1 Objetivo Geral

Desenvolver uma arquitetura digital completa em FPGA para visão computacional embarcada, capaz de capturar vídeo em tempo real, isolar uma região de interesse e realizar a inferência de dígitos numéricos manuscritos utilizando aceleração de hardware puro baseada no método de Correspondência de Molde Euclidiana (Template Matching).

2.3.2 Objetivos Específicos

Para atingir o objetivo principal, a escalada técnica foi dividida em marcos que podem ser validados individualmente através de ensaios físicos na bancada:

1. **Geração Gráfica:** Implementar um controlador de vídeo VGA estrito aos temporizadores (Vsync/Hsync) para a exibição de quadros.

2. **Aquisição de Imagem:** Decodificar o protocolo DVP da câmera OV2640, sincronizando os sinais de pixel clock para armazenar quadros de forma contínua em um framebuffer.
3. **Filtragem em Tempo Real:** Desenvolver um módulo pré-processador (Color Tracker) para conversão de espectro de cor e binarização instantânea da imagem.
4. **Classificação em Hardware:** Projetar e instanciar um classificador estatístico de dígitos (0 a 9) alimentado por uma ROM interna, implementando técnicas de mitigação de erro de alinhamento espacial (Visão Periférica/Tolerância Espacial).

2.4 Proposta de Solução

Para solucionar o problema da alta latência e da dependência de processadores sequenciais na inferência de imagens, a solução proposta consiste em uma arquitetura de aceleração digital totalmente baseada em fluxo de dados (data streaming) em hardware reconfigurável (FPGA). A filosofia do projeto baseou-se em uma construção modular e evolutiva. Em vez de projetar o sistema complexo de uma só vez, a arquitetura foi desenvolvida, validada e expandida em quatro estágios de complexidade crescente, garantindo o funcionamento robusto de cada subsistema antes de integrá-lo ao próximo.

2.4.1 Estágio 1: Geração de Sinal e Leitura de Memória (Validação do VGA)

O primeiro passo consistiu em estabelecer a interface de saída. A FPGA foi programada para gerar os sinais de sincronismo (VGAHS e VGAVS) e varrer uma memória ROM interna contendo uma imagem estática predefinida, enviando os dados de cor (RGB) diretamente para o monitor.

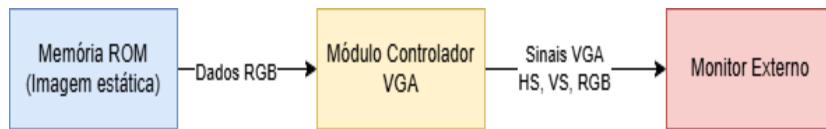


Figura 1: Diagrama de blocos do módulo VGA

2.4.2 Estágio 2: Aquisição de Vídeo em Tempo Real (Integração da Câmera OV2640)

Avançando na complexidade, a ROM estática foi substituída pelo fluxo dinâmico de uma câmera OV2640. O sistema passou a capturar os pixels via protocolo DVP, armazenando-os em um framebuffer (SRAM) na cadência do clock da câmera (dclk), enquanto o módulo VGA realizava a leitura dessa mesma

memória no domínio do clock da tela, garantindo a exibição do vídeo em tempo real.

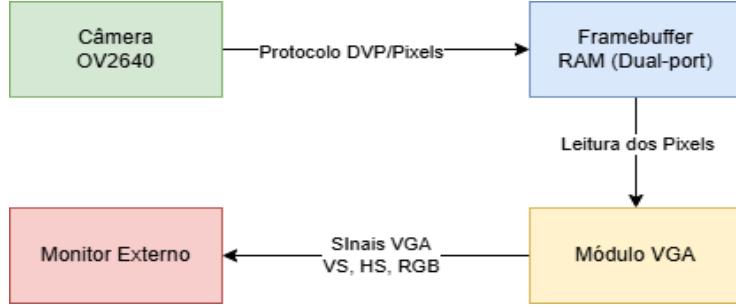


Figura 2: Diagrama de blocos da câmera OV2640

2.4.3 Estágio 3: Processamento Intermediário (Implementação do Color Tracker)

Com o fluxo de vídeo estabelecido, inseriu-se um estágio de processamento entre a câmera e a exibição. O Color Tracker foi acoplado ao fluxo de saída do módulo de captura, recebendo os pixels e binarizando-os com base em limiares (thresholds) de cor. Esse módulo computa simultaneamente as coordenadas limites (X e Y) dos objetos rastreados. Para fins de interface de usuário (UI), o módulo VGA foi alterado para receber as coordenadas da Região de Interesse (ROI) calculadas pelo tracker e desenhar, puramente em hardware, uma moldura verde sobreposta à imagem original no monitor.

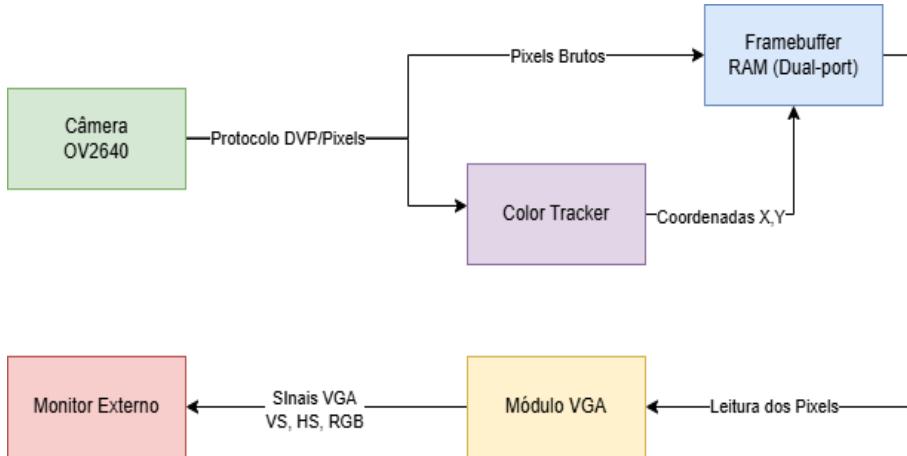


Figura 3: Diagrama de blocos do Color Tracker

2.4.4 Estágio 4: Aceleração de Inferência (A Arquitetura Final com Classificador MNIST)

A arquitetura final completa o pipeline integrando a inteligência artificial. O processamento do Color Tracker passa a alimentar não apenas o gerador gráfico, mas também o extrator de características. O sistema recorta os 28x28 pixels referentes à ROI e os armazena em uma SRAM dedicada. Ao final de cada quadro (frame), o Módulo Classificador (implementando o algoritmo de Template Matching Euclidiano com mitigação espacial por Visão Periférica) entra em ação. Ele compara o dado recém-capturado na RAM com os moldes ideais residentes na ROM de inferência. O resultado dessa classificação é então exportado para periféricos externos.

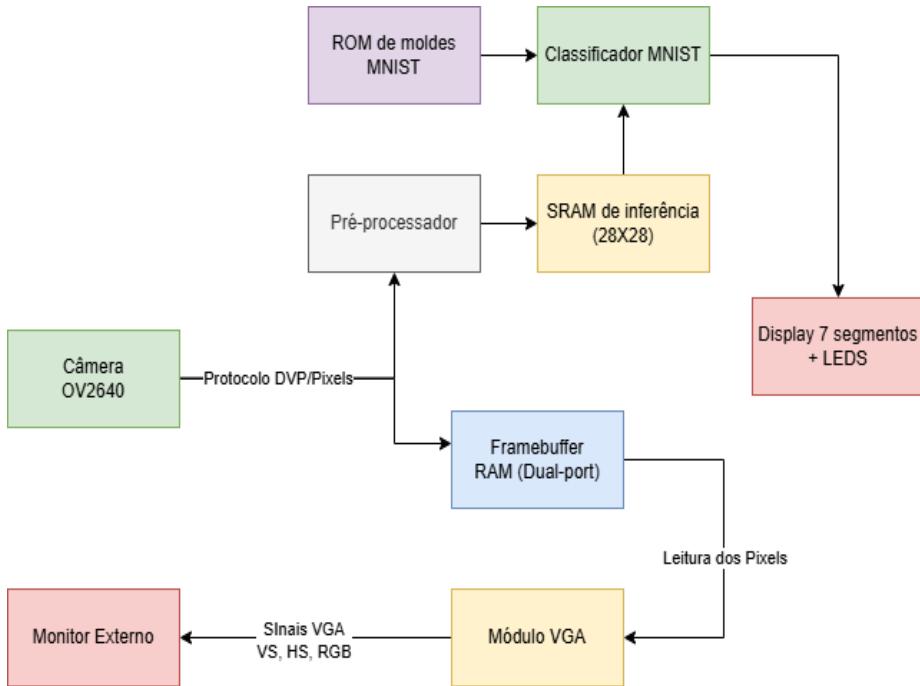


Figura 4: Diagrama de blocos do módulo MNIST

2.5 Detalhamento do Documento

Para apresentar o desenvolvimento, a fundamentação e as provas de conceito desta arquitetura, este documento está estruturado em seis partes, organizadas de forma a guiar o leitor desde os conceitos basilares até a validação física na bancada.

1. **Introdução:** Descrevemos a problemática da latência no processamento de imagens, delineamos os objetivos do projeto e apresentamos a arquitetura modular proposta para solucionar a inferência de dígitos sem o uso de microprocessadores.
2. **Revisão Bibliográfica:** abordará o referencial teórico que sustenta as escolhas de engenharia do projeto. Serão discutidos os paradigmas do processamento de vídeo em hardware (incluindo os protocolos VGA e DVP), os princípios de rastreamento de cor e binarização, e a fundamentação matemática que diferencia o classificador geométrico utilizado (Template Matching) das Redes Neurais Convolucionais (CNNs) convencionais.
3. **Metodologia:** detalhará o roteiro prático adotado para testar e validar cada módulo da solução. Este capítulo descreverá os métodos de aferição do sinal de vídeo, a estratégia de mitigação de erros espaciais através do conceito de "Visão Periférica" em hardware, e o uso de scripts em Python para realizar a engenharia reversa e o ajuste fino da matriz de memória ROM da inteligência artificial.
4. **Resultados:** apresentará as evidências físicas e visuais de que o sistema atingiu os requisitos propostos. Serão expostas as imagens da interface de rastreamento operando em tempo real no monitor, as plotagens da extração de características em memória e as fotografias do setup final, comprovando o acionamento preciso do Display de 7 Segmentos e LEDs em resposta aos dígitos manuscritos.
5. **Conclusões:** fará o fechamento do trabalho, confrontando os resultados práticos com os objetivos definidos inicialmente. Serão discutidos os pontos fortes da arquitetura paralela e, com total transparência técnica, as fragilidades e limitações inerentes à ausência de convoluções, finalizando com as perspectivas de aprimoramentos para trabalhos futuros.
6. **Apêndices:** todos os códigos-fonte descritos em Verilog, scripts de validação em Python e os arquivos de restrição física (LPF) desenvolvidos neste trabalho encontram-se integralmente disponíveis, documentados e versionados em repositório público no GitHub, cujo link de acesso é detalhado no Apêndice A deste documento.

3 Revisão Bibliográfica

Neste capítulo, são apresentados os fundamentos teóricos e as tecnologias que dão sustentação ao desenvolvimento da arquitetura de aceleração em hardware. Discute-se o processamento de vídeo em fluxo de dados, as técnicas clássicas de visão computacional para rastreamento de objetos e, por fim, o contraste entre os paradigmas modernos de classificação de imagens (Redes Convolucionais) e o método estatístico-geométrico adotado neste projeto.

3.1 Processamento de Vídeo em Hardware (FPGAs)

Diferente dos sistemas operacionais convencionais que processam imagens armazenando quadros completos na memória para posterior iteração por loops de software, as FPGAs permitem o processamento baseado em streaming (fluxo contínuo). Neste paradigma, cada pixel é processado no exato instante em que o sinal de clock o disponibiliza no barramento.

Para a interface de saída gráfica, o padrão VGA (Video Graphics Array) é amplamente utilizado por sua robustez e simplicidade de implementação em lógica digital. O protocolo não requer pacotes de dados complexos; ele baseia-se puramente em temporização estrita. Utilizando contadores sincronizados com o pixel clock, geram-se os sinais de sincronismo horizontal (Hsync) e vertical (Vsync), indicando ao monitor eletrônico o momento exato de iniciar uma nova linha ou um novo quadro, enquanto os canais de cor (RGB) são modulados simultaneamente .

Na extremidade da captura, sensores CMOS como o OV2640 transmitem dados através do protocolo DVP (Digital Video Port). O desafio em hardware reconfigurável consiste em cruzar o domínio de clock gerado pela câmera (geralmente assíncrono em relação à FPGA) para o domínio principal da placa, agrupando os bytes sequenciais recebidos em words de 16 bits (padrão RGB565) e armazenando-os em memórias RAM de porta dupla (Dual-Port RAM) para compor o framebuffer.

3.2 Visão Computacional de Baixo Nível: Binarização e Tracking

Em aplicações de visão computacional onde o processamento ocorre direto nas portas lógicas, extrair a informação essencial precocemente economiza recursos massivos de memória e lógica combinacional. Uma técnica clássica é a binarização por limiarização (Thresholding).

Em vez de operar sobre os 65.536 valores possíveis de um pixel RGB565, o sistema aplica comparadores lógicos que avaliam a intensidade dos canais vermelho, verde e azul. Se a assinatura do pixel corresponder à de uma tinta escura (ou outra cor de interesse configurada), o pixel recebe o valor lógico alto (1); caso seja o fundo da folha de papel, recebe o valor baixo (0). Essa técnica reduz a largura de banda necessária em 93% (de 16 bits para 1 bit por pixel).

A partir dessa binarização espacial, implementam-se os rastreadores de cor (Color Trackers). Ao detectar um pixel de interesse (nível 1), registradores atualizam continuamente as coordenadas máxima e mínima de X e Y encontradas durante a varredura do quadro. Ao final do sinal de Vsync, esses registradores delimitam perfeitamente uma caixa (Bounding Box), isolando a Região de Interesse (ROI) do restante da imagem, descartando fundo desnecessário antes da etapa de inferência.

3.3 Paradigmas de Classificação: Template Matching vs. Redes Convolucionais

O núcleo de inteligência artificial de sistemas de visão baseia-se em como os dados isolados são classificados. Na literatura atual, o dataset MNIST é dominado por Redes Neurais Convolucionais (CNNs). Uma CNN utiliza múltiplos filtros matemáticos (núcleos ou kernels) que deslizam sobre a imagem realizando multiplicações de matrizes para detectar bordas, curvas e texturas . Por aprenderem as "características" locais da imagem, as CNNs são altamente invariantes à translação espacial, escala e rotações leves. Contudo, implementar convoluções exige matrizes complexas de blocos multiplicadores de hardware (DSPs), algo muitas vezes indisponível em FPGAs de baixo custo.

A alternativa abordada neste trabalho, que se adequa razoavelmente bem as restrições severas de hardware, é a Correspondência de Molde Euclidiana (Template Matching ou Classificador de Centroide Mais Próximo). Neste paradigma, o sistema não procura por características isoladas, mas sim sobrepõe matematicamente toda a imagem capturada sobre matrizes ideais (moldes) pré-gravadas em memória ROM.

A métrica de similaridade é feita por uma contagem de pixels coincidentes. No entanto, sua maior fragilidade documentada é a alta sensibilidade à translação: um deslocamento de poucos pixels feito pelo usuário arruina o alinhamento com a máscara da ROM.

3.4 Trabalhos Relacionados e Justificativa da Abordagem

Embora o reconhecimento de caracteres numéricos (MNIST) seja um "**Hello Word!**", clássico no desenvolvimento de software via bibliotecas como TensorFlow e PyTorch, sua implementação pura em portas lógicas é uma área de pesquisa ativa para Edge Computing. A maioria dos trabalhos relacionados na academia atinge precisões superiores a 98% instanciando arquiteturas convolucionais como a LeNet-5 em placas de alto custo (família Xilinx Zynq ou Intel SoC), que contam com centenas de DSPs.

A abordagem proposta neste trabalho não busca superar a precisão absoluta do estado da arte em redes profundas. Pelo contrário, ela se justifica pela exploração radical do processamento na borda sob forte restrição. Ao adotar o Template Matching modificado com heurísticas de tolerância espacial local (a técnica implementada de "Visão Periférica" nos endereços de ROM), o projeto

demonstra que é possível extrair inteligência e reconhecimento de padrões utilizando frações dos recursos lógicos comuns, sem atrasos (delay), e abrindo mão de CPUs e multiplicadores custosos, adequando-se perfeitamente a ambientes de baixo consumo energético e prototipagem educacional.

4 Metodologia e Implementação

Este capítulo descreve o percurso metodológico e as etapas adotadas para a construção e validação da arquitetura de processamento de vídeo em hardware. O sistema foi desenvolvido e testado de forma modular e incremental. A validação de cada subsistema ocorreu através de ensaios físicos na bancada, garantindo que os domínios de clock, a temporização de sinais e a lógica combinacional estivessem maduros antes da integração da inteligência artificial.

4.1 Validação da Interface Gráfica (Módulo VGA)

A primeira etapa do projeto consistiu em garantir que a FPGA pudesse se comunicar com um monitor externo. Utilizou-se um adaptador físico (VGA PS2 Board, da Waveshare, **figura 5**) conectado aos pinos de I/O genéricos (GPIO) da placa. A metodologia de validação deste módulo foi dividida em três fases progressivas de teste de datapath:

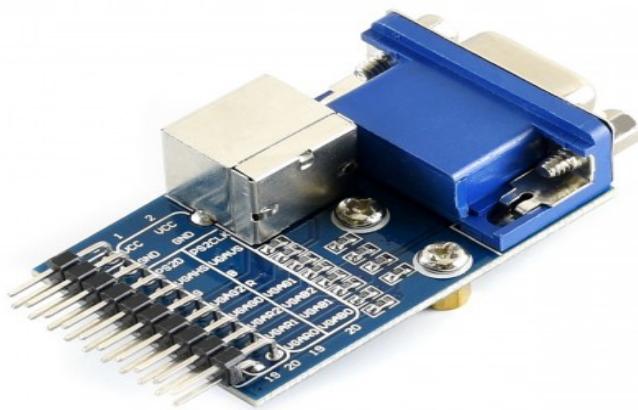


Figura 5: VGA PS2 Board, da Waveshare

4.1.1 Sincronismo e Tela Monocromática

Inicialmente, o objetivo foi validar exclusivamente os contadores de temporização horizontal (h_count) e vertical (v_count). Foi escrito um módulo Verilog para gerar os pulsos de VGAHS e VGAVS estritamente de acordo com a norma VGA (640x480 a 60Hz). O teste físico consistiu em forçar os pinos de cor para nível lógico alto em regiões específicas para pintar a tela de uma única cor, atestando que o monitor não perdia o sincronismo.



Figura 6: Tela monocromática azul

Aqui destacamos a logica sequencial principal do modulo VGA, responsavel por gerar os sinais de sincronismo (HS e VS) e as coordenadas de pixel com base nos temporizadores definidos:

```
1 // --- LOGICA PARA CONTADORES DAS POSICOES (X,Y) NA MATRIZ ---
2 always @ (posedge vga_clk) begin
3   if (!rst_n) begin
4     // (Inicializacao de variaveis e reset omitidos para
5     brevidade)
6   end else begin
7
8     // --- LOGICA CRUCIAL DE SINCRONISMO E HABILITACAO ---
9     // Sinal baixo durante a janela de sincronismo
10    VGAHS <= ((x_pixel >= INICIO_SINCRONISMO_HORIZONTAL) &&
11      x_pixel < FIM_SINCRONISMO_HORIZONTAL) ? 1'b0 : 1'b1;
12    VGAVS <= ((y_pixel >= INICIO_SINCRONISMO_VERTICAL) &&
13      y_pixel < FIM_SINCRONISMO_VERTICAL) ? 1'b0 : 1'b1;
14
15    // Sinal alto apenas quando esta na area valida do monitor
```

```

13      data_enable <= ((x_pixel <= END_AREA_ATIVA_HORIZONTAL) && (
14          y_pixel <= END_AREA_ATIVA_VERTICAL)) ? 1'b1: 1'b0;
15
16      // --- ATUALIZACAO DOS CONTADORES (Varredura da tela) ---
17      if (x_pixel == ULTIMO_PIXEL_HORIZONTAL) begin
18          x_pixel <= 0; // Volta para o inicio da linha
19
20          if (y_pixel == ULTIMO_PIXEL_VERTICAL) begin
21              y_pixel <= 0; // Volta para o topo da tela
22          end else begin
23              y_pixel <= y_pixel + 1; // Desce uma linha
24          end
25      end else begin
26          x_pixel <= x_pixel + 1; // Avanca um pixel para a direita
27      end
28
29  end

```

Listing 1: Logica central de sincronismo e controle do modulo VGA.

O módulo de topo (*top module*) atua como o maestro do sistema, conectando o controlador VGA aos pinos de saída da FPGA. Para validar a interface física com o monitor, implementamos uma lógica que pinta a tela inteira de azul.

```

1  // FIOS PARA POSICAO DO PIXEL E DATA ENABLE
2  wire [9:0] x, y;
3  wire data_enable;
4
5  // INSTANCIAMENTO DO CONTROLADOR VGA
6  vga_control VGA_CONTROL (
7      .vga_clk(clk),
8      .rst_n(rst_n),
9      .x_pixel(x),
10     .y_pixel(y),
11     .VGAHS(VGAHS),
12     .VGAVS(VGAVS),
13     .data_enable(data_enable)
14 );
15
16 // LOGICA COMBINACIONAL DE CORES (TESTE TELA AZUL)
17 always @(*) begin
18     if (data_enable) begin
19         // Dentro da area visivel: pinta de azul
20         VGA_R = 3'b000;
21         VGA_G = 3'b000;
22         VGA_B = 3'b111;
23     end else begin
24         // Fora da area visivel (Blanking/Sync): mantem os sinais
25         em zero
26         VGA_R = 3'b000;
27         VGA_G = 3'b000;
28         VGA_B = 3'b000;
29     end
end

```

Listing 2: Módulo principal instanciando o controlador VGA e gerando cor.

4.1.2 Padrão de Barras de Cor (Color Bars)

Para validar o barramento de dados RGB, a lógica foi expandida para gerar as clássicas barras de cor verticais. A tela foi dividida matematicamente utilizando o contador horizontal, atribuindo uma combinação diferente de bits RGB para cada faixa da tela. Este ensaio provou que todos os pinos físicos do adaptador VGA estavam soldados e roteados corretamente no arquivo de restrições (.lpf).



Figura 7: Tela com barras coloridas

Após validar o sincronismo com uma cor sólida, o módulo de topo foi expandido para gerar um padrão de teste clássico de vídeo: as barras de cores (Color Bars). Utilizando a coordenada horizontal (x) gerada pelo controlador, a largura total da tela (640 pixels) foi dividida em 8 segmentos iguais de 80 pixels.

O Código demonstra a lógica combinacional implementada. Dentro do período de `data_enable` ativo, uma estrutura condicional altera os sinais RGB dependendo da posição atual do feixe, desenhando as colunas sequencialmente:

```
1 // LOGICA COMBINACIONAL DE CORES (PADRAO BARRAS DE COR)
2 always @(*) begin
3     if (data_enable) begin
4         // --- 8 barras verticais de 80 pixels (640 / 8 = 80) ---
5
6         if      (x < 80) {VGA_R, VGA_G, VGA_B} = 9'b111_111_111;
7             // 1: Branco
8         else if (x < 160) {VGA_R, VGA_G, VGA_B} = 9'b111_111_000;
9             // 2: Amarelo
10            else if (x < 240) {VGA_R, VGA_G, VGA_B} = 9'b000_111_111;
11            // 3: Ciano
```

```

9      else if (x < 320) {VGA_R, VGA_G, VGA_B} = 9'b000_111_000;
10     // 4: Verde
11     else if (x < 400) {VGA_R, VGA_G, VGA_B} = 9'b111_000_111;
12     // 5: Magenta
13     else if (x < 480) {VGA_R, VGA_G, VGA_B} = 9'b111_000_000;
14     // 6: Vermelho
15     else if (x < 560) {VGA_R, VGA_G, VGA_B} = 9'b000_000_111;
16     // 7: Azul
17     else                 {VGA_R, VGA_G, VGA_B} = 9'b000_000_000;
18     // 8: Preto
19
end
end

```

Listing 3: Geração do padrão de teste Color Bars utilizando a coordenada X.

4.1.3 Leitura de Memória (Imagen Estática)

O último ensaio do módulo de vídeo foi a prova de conceito de leitura de memória. Uma imagem estática foi convertida em um arquivo .mif/.hex e instanciada em um bloco de memória ROM (Read-Only Memory) interna da FPGA. O controlador VGA foi então modificado para varrer os endereços da ROM sincronizados com o feixe de elétrons do monitor.



Figura 8: Tela com imagem gravada na ROM

Para exibir uma imagem estática na tela, foi necessário converter um arquivo de imagem comum (PNG/JPG) em um formato de memória inicializável lido pela FPGA (.hex). Devido às restrições de memória interna da arquitetura, implementou-se um script em Python para redimensionar a imagem para 320x240 pixels e aplicar uma compressão de cor de 24 bits (RGB888) para 8 bits (RGB332).

O Código 4 ilustra a lógica central do script. A função `to_rgb332` utiliza operações de deslocamento de bits (*bit-shifting*) para extrair os bits mais significativos de cada canal e empacotá-los em um único byte, que é então escrito sequencialmente no arquivo de saída:

```

1 from PIL import Image
2
3 # Função de conversão RGB (24 bits) para RGB332 (8 bits)
4 def to_rgb332(r, g, b):
5     r3 = (r >> 5) & 0x07 # Pega os 3 bits mais significativos do
6     Red
7     g3 = (g >> 5) & 0x07 # Pega os 3 bits mais significativos do
8     Green
9     b2 = (b >> 6) & 0x03 # Pega os 2 bits mais significativos do
10    Blue
11    return (r3 << 5) | (g3 << 2) | b2 # Empacota em 1 Byte
12
13 # Abre, converte para RGB e redimensiona a imagem
14 img = Image.open("minha_imagem.png").convert("RGB").resize((320,
15                         240))
16
17 # Gera o arquivo .hex varrendo pixel a pixel (Raster Scan)
18 with open("imagem_rgb332_320x240.hex", "w") as f:
19     for y in range(240):
20         for x in range(320):
21             r, g, b = img.getpixel((x, y))
22             val = to_rgb332(r, g, b)
23             f.write(f"{val:02X}\n") # Escreve no formato
24             Hexadecimal (00 a FF)
```

Listing 4: Script resumido para conversão de imagem em mapa de memória hexadecimal (RGB332).

A integração da imagem estática exigiu a leitura síncrona de uma memória ROM interna. Para exibir uma imagem de resolução 320x240 em um monitor 640x480, aplicou-se uma técnica de *upscale* baseada em deslocamento de bits (divisão por 2). Além disso, para evitar o uso de blocos multiplicadores de hardware (DSPs), o cálculo de endereço da matriz ($Y \times 320 + X$) foi otimizado utilizando soma de potências de base 2 ($Y \times 256 + Y \times 64$).

O Código 5 ilustra as otimizações matemáticas e a implementação de um *pipeline* de 1 ciclo de *clock*. Este atraso nos sinais de sincronismo é estritamente necessário para compensar a latência de leitura da ROM, garantindo que o dado de cor chegue ao monitor exatamente alinhado com o seu respectivo pixel na tela:

```

1 // 1. INICIALIZACAO DA ROM
2 reg [7:0] pixel_imagem [0:(320*240)-1];
3 initial $readmemh("imagem_rgb332_320x240.hex", pixel_imagem);
4
5 // 2. UPSCALE (640x480 -> 320x240) usando Shift Right (Divisao por
6 // 2)
7 wire [8:0] x_img = x_pixel >> 1;
8 wire [8:0] y_img = y_pixel >> 1;
9
10 // 3. CALCULO DE ENDERECO OTIMIZADO (Sem multiplicador de hardware)
11 // Y * 320 = Y * 256 + Y * 64 = (Y << 8) + (Y << 6)
12 wire [16:0] ender_rom = ({y_img, 8'd0}) + ({y_img, 6'd0}) + x_img;
13
14 // 4. LEITURA SINCRONA E COMPENSACAO DE LATENCIA (Pipeline)
15 always @(posedge vga_clk) begin
16   if (data_enable) begin
17     // Leitura do dado na ROM (Demora 1 ciclo de clock)
18     delay_1byte_pixel_rom <= pixel_imagem[ender_rom];
19   end
20
21   // Atraso de 1 ciclo nos sinais de controle para alinhar com a
22   // ROM
23   delay_data_enable <= data_enable;
24   delay_VGAHS <= VGAHS_in;
25   delay_VGAVS <= VGAVS_in;
26
27   // Sinais de saida atualizados com o delay
28   VGAHS <= delay_VGAHS;
29   VGAVS <= delay_VGAVS;
30
31   // Saída RGB mapeada apenas na área visível "atrasada"
32   if (delay_data_enable) begin
33     VGA_R <= delay_1byte_pixel_rom[7:5];
34     VGA_G <= delay_1byte_pixel_rom[4:2];
35     // (Logica do mapeamento do azul omitida para brevidade)
36   end else begin
37     {VGA_R, VGA_G, VGA_B} <= 9'b000_000_000;
38   end
39 end

```

Listing 5: Endereço upscale e pipelining para leitura da ROM.

4.2 Integração e Sincronismo da Câmera OV2640

Com a saída de vídeo validada, a metodologia avançou para a aquisição dinâmica de dados. A câmera OV2640 foi conectada à FPGA, introduzindo o desafio do cruzamento de domínios de clock (Clock Domain Crossing - CDC).



Figura 9: Câmera OV2640

A integração do sensor CMOS OV2640 representou o maior desafio de temporização e fluxo de dados do projeto. O sensor opera com seu próprio oscilador embutido e transmite dados de forma assíncrona ao clock principal da FPGA. Para solucionar esse cruzamento de domínios de clock (CDC) e garantir uma leitura de quadros sem corrupção, a "biblioteca da câmera" foi arquitetada hierarquicamente no módulo agrupador camera_top.

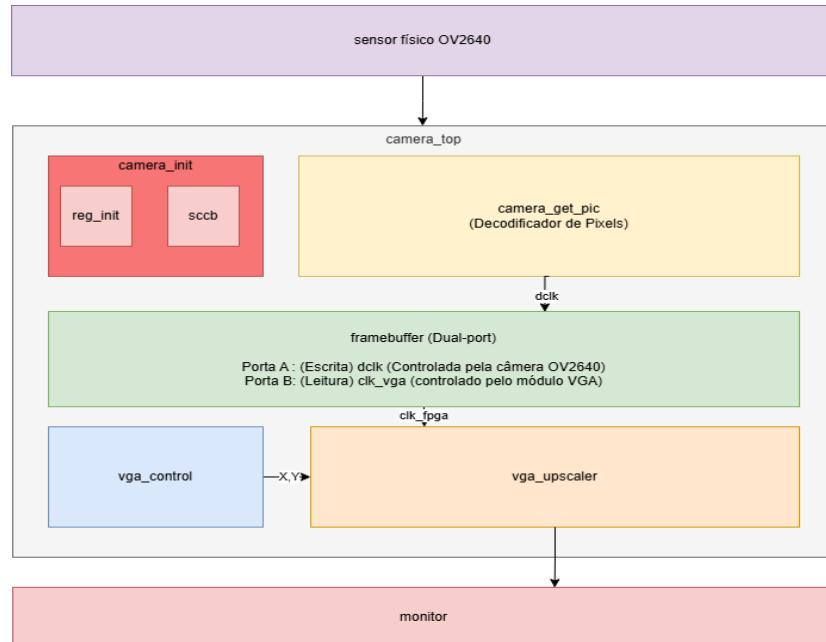


Figura 10: Diagrama de Blocos do Módulo da câmera

4.2.1 Configuração Inicial via SCCB (camera_init e reg_init)

Antes de enviar qualquer pixel válido, a OV2640 acorda em um estado inativo. Foi necessário implementar um controlador mestre compatível com o barramento I2C (chamado de SCCB pela Omnivision).

O módulo camera_init é responsável por gerar os pulsos seriais lógicos (scio_c e scio_d). Ele consome os dados do módulo reg_init, que funciona como uma imensa ROM contendo a tabela de inicialização do fabricante (definindo a resolução para QVGA/VGA, o formato RGB565, saturação e balanço de branco).

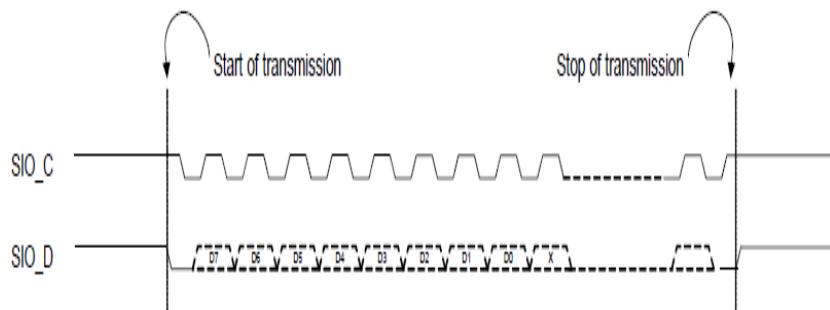


Figura 11: Procolo I2C/SCCB

4.2.2 O Decodificador de Pixels DVP (camera_get_pic)

O protocolo DVP (Digital Video Port) da câmera não envia o pixel inteiro de uma vez. Como o formato escolhido é RGB565 (16 bits), mas o barramento físico da câmera tem apenas 8 bits ($\text{data}[7:0]$), a câmera envia o pixel em dois ciclos do clock da câmera (dclk).

O módulo camera_get_pic atua como uma "máquina de costura digital". Ativado apenas quando a câmera sinaliza que o dado é válido na linha (href em nível alto), ele captura o primeiro byte na borda de subida do clock, armazena em um registrador temporário, captura o segundo byte no ciclo seguinte, concatena os dois para formar o pixel de 16 bits e gera um pulso de "escrita válida" (wr_en) juntamente com o endereço de memória calculado.

Para a aquisição de vídeo ao vivo, o módulo de interface com a câmera OV2640 precisa lidar com o protocolo de transmissão do sensor. Como a câmera envia dados em pacotes de 8 bits por ciclo de *clock* (`dclk`) e o formato de cor escolhido foi o RGB565 (16 bits), são necessários dois ciclos de *clock* válidos (com o sinal `href` em nível alto) para formar um único pixel.

O Código apresenta a solução implementada: um registrador de deslocamento atua como uma máquina de estados compacta (**status**), rastreando a chegada dos pares de *bytes*. Simultaneamente, os dados são concatenados e um sinal de habilitação de escrita (**wr_en**) é gerado a cada dois ciclos para salvar o pixel completo na memória.

```

1  always @ (posedge dclk) begin
2      // VSYNC em zero indica fim do quadro (reset dos contadores)
3      if (vsync == 1'b0) begin
4          next_addr <= 0;
5          status     <= 0;
6          wr_en     <= 0;
7      end else begin
8          // 1. CONCATENACAO DE BYTES
9          // O byte novo entra na parte baixa, empurrando o antigo
10         para a alta
11         rgb565 <= {rgb565[7:0], data_in};
12
13         // 2. MAQUINA DE ESTADOS (Shift Register)
14         // status[1] fica ALTO apenas quando o 2o byte do pixel e
15         // recebido
16         status <= {status[0], (href && !status[0])};
17
18         // 3. CONTROLE DE MEMORIA
19         // Habilita a escrita apenas quando o pixel de 16 bits esta
20         // completo
21         wr_en <= status[1];
22
23         // Incrementa o endereco da memoria para o proximo pixel (
24         // Limite 320x240)
25         if (status[1] == 1'b1) begin
26             if (next_addr < 76800 - 1)
27                 next_addr <= next_addr + 1;
28         end
29     end
30 end

```

Listing 6: Aquisição de dados da câmera OV2640 e montagem do pixel RGB565.

4.2.3 Resolução de Conflitos de Clock (framebuffer)

A sincronização entre o que a câmera "vê" e o que o monitor "mostra" ocorre neste módulo crítico. O framebuffer foi instanciado forçando a síntese de uma BRAM (Block RAM) verdadeira de Porta Dupla (Dual-Port).

Dessa forma, a porta de escrita (Porta A) é comandada exclusivamente pelos sinais de controle rápidos vindos do camera_get_pic (domínio do dclk). Simultaneamente e de forma assíncrona, a porta de leitura (Porta B) atende aos pedidos de endereço vindos do controlador de tela (domínio do clk da FPGA principal). Isso elimina problemas de metaestabilidade e garante que o fluxo do monitor nunca pare esperando a câmera processar uma linha.

Para sincronizar o fluxo de dados entre a câmera e o monitor, foi implementado um *Framebuffer* operando como uma memória Dual-Port RAM. Este módulo resolve o problema de travessia de domínio de *clock*, permitindo que a escrita ocorra na frequência da câmera (*wr_clk*) e a leitura ocorra na frequência de varredura do display (*rd_clk*).

O Código destaca o uso da diretiva de síntese `ram_style = "block"`, que instrui a ferramenta a mapear a matriz de pixels diretamente nos blocos de memória dedicados (BRAM) da FPGA. Além disso, a leitura foi projetada com

um estágio extra de registradores (*pipeline*) para garantir o fechamento de *timing* em altas frequências.

```

1 // Inferencia de Block RAM (Armazena 320x240 pixels em RGB565)
2 (* ram_style = "block" *)
3 reg [15:0] mem [0:76799];
4
5 // --- ESCRITA: DOMINIO DE CLOCK DA CAMERA (dclk) ---
6 always @(posedge wr_clk) begin
7     if (wr_en)
8         mem[wr_addr] <= wr_data;
9 end
10
11 // --- LEITURA: DOMINIO DE CLOCK DO VGA (25 MHz) ---
12 reg [15:0] mem_read_internal;
13
14 always @(posedge rd_clk) begin
15     if (rd_en) begin
16         // Leitura em 2 estagios (Pipeline) para aliviar o timing
17         mem_read_internal <= mem[rd_addr];
18         rd_data           <= mem_read_internal;
19     end
20 end

```

Listing 7: Framebuffer Dual-Port com inferência de Block RAM e domínios de clock independentes.

4.2.4 Adaptação de Resolução e Renderização (vga_upscaler)

Uma das restrições arquiteturais mais severas na síntese de processamento de vídeo “na borda” (*Edge Computing*) é a capacidade de armazenamento interno (*Block RAM*) da FPGA, uma vez que o projeto não utiliza chips de memória DDR externos. A norma VGA padrão exige um quadro de 640×480 pixels. Operando com o padrão de cor RGB565, cada pixel consome 16 bits (ou 2 bytes) de memória. O custo de *hardware* para armazenar um único quadro (*frame*) inteiro na resolução nativa do VGA seria:

$$Total_{pixels} = 640 \times 480 = 307.200 \text{ pixels}$$

$$Total_{bits} = 307.200 \times 16 \text{ bits/pixel} = 4.915.200 \text{ bits}$$

Isso equivale a aproximadamente 4,9 Megabits (ou ≈ 600 KB) de memória SRAM apenas para o *Framebuffer*. Contudo, as FPGAs de baixo custo utilizadas neste projeto possuem uma limitação física em torno de 1,0 Megabit de memória BRAM total. Ou seja, a exigência de um quadro VGA completo excede em quase 5 vezes a capacidade de silício da placa.

Para solucionar este gargalo físico, a câmera OV2640 foi configurada via protocolo I2C para transmitir em uma resolução subamostrada. Com isso, o consumo de memória despenca drasticamente. Tomando como base uma redução na captação, o novo custo passa a ser:

$$Total_{reduzido} = (160 \times 120) \times 16 \text{ bits/pixel} = 307.200 \text{ bits}$$

Esse valor consome apenas cerca de 30% da memória total da FPGA, deixando o restante livre para instanciar a ROM de Inteligência Artificial e a SRAM de inferência da rede neural. No entanto, para que a imagem não aparecesse como um pequeno “selo” no canto superior esquerdo do monitor VGA, o módulo `vga_upscaler` foi projetado para atuar como um interpolador lógico dinâmico (técnica de Vizinho Mais Próximo).

Este módulo combinacional recebe as coordenadas (X, Y) que o controlador VGA está desenhandando na tela de 640×480 e aplica divisões binárias (utilizando deslocamentos lógicos de *shift-right*, `>>`) para converter a coordenada ampla da tela em um endereço contraído da memória. Esse método comprovou a flexibilidade do *hardware* em redimensionar a imagem dinamicamente apenas mudando a lógica de endereçamento de leitura, sem alterar os dados gravados e mitigando a restrição de memória física do chip.

O Código 8 ilustra a implementação lógica do cálculo de endereço por deslocamento, associado ao *pipeline* de registradores necessário para compensar a latência de leitura da BRAM:

```

1 // 1. CALCULO DE ENDERECHO COM UPSCALE (Shift Right)
2 // O deslocamento dinamico dos bits (divisao) atua como
   interpolador de imagem
3 if (data_enable) begin
4     fb_rd_en    <= 1'b1;
5     fb_rd_addr <= x_pixel[9:1] + (y_pixel[9:1] * LARGURA_ORIGEM);
6     pipe_de[0]  <= 1'b1; // Sinaliza pedido de leitura no inicio do
                           pipeline
7 end else begin
8     fb_rd_en    <= 1'b0;
9     pipe_de[0]  <= 1'b0;
10 end
11
12 // 2. PIPELINE DE SINCRONIA (Atraso compensatorio)
13 // Propaga os sinais de sincronismo junto com a latencia da RAM
14 pipe_hs[0]  <= VGAHS_in;
15 pipe_vs[0]  <= VGAVS_in;
16
17 for(i=0; i<3; i=i+1) begin
18     pipe_hs[i+1] <= pipe_hs[i];
19     pipe_vs[i+1] <= pipe_vs[i];
20     pipe_de[i+1] <= pipe_de[i];
21 end
22
23 // 3. SAIDA SINCRONIZADA E MAPEAMENTO DE COR
24 delay_pixel <= fb_pixel; // Dado retornado da memoria interpolada
25 VGAHS        <= pipe_hs[3]; // Sincronismo da tela preservado
26 VGAVS        <= pipe_vs[3];
27
28 if (pipe_de[3]) begin
29     VGA_R <= delay_pixel[15:13];
30     VGA_G <= delay_pixel[10:8];
31     VGA_B <= delay_pixel[4:2];
32 end

```

Listing 8: Cálculo dinâmico de *upscale* (interpolação espacial) e *pipeline* de compensação de latência.

A perda de qualidade visual da imagem é o resultado de um duplo afunilamento na profundidade de cor ao longo do *hardware*. Enquanto os monitores e computadores atuais operam no padrão RGB888 (24 bits), exibindo mais de 16,7 milhões de cores, a câmera OV2640 captura e transmite o quadro em RGB565 (16 bits), reduzindo o espectro para 65.536 cores. O gargalo final e mais severo ocorre no adaptador VGA, que limita a saída física a apenas 1 byte por pixel (8 bits, no formato RGB332). Isso esmaga a paleta de cores para exatas 256 cores, justificando tecnicamente o forte efeito de posterização e a perda de degradês na tela como vemos nas imagens abaixo:

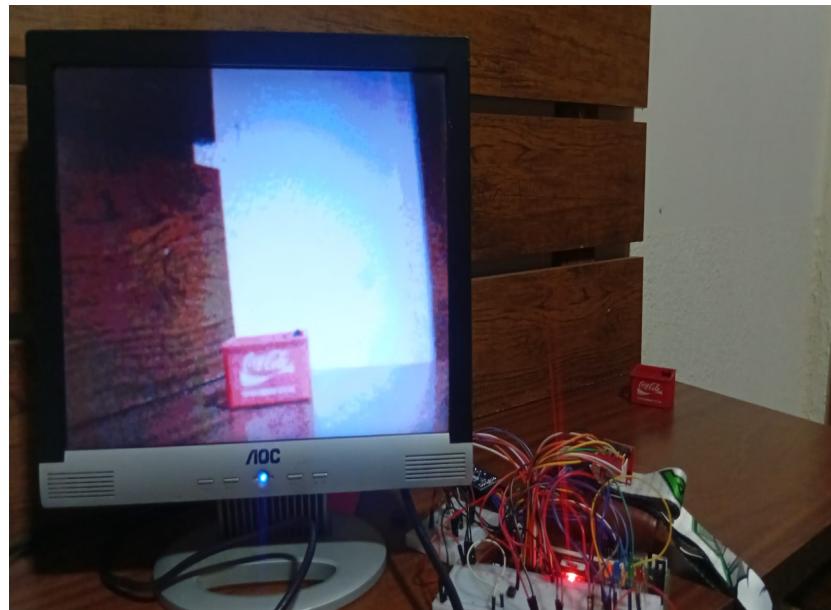


Figura 12: Embalagem decorativa de coca-cola

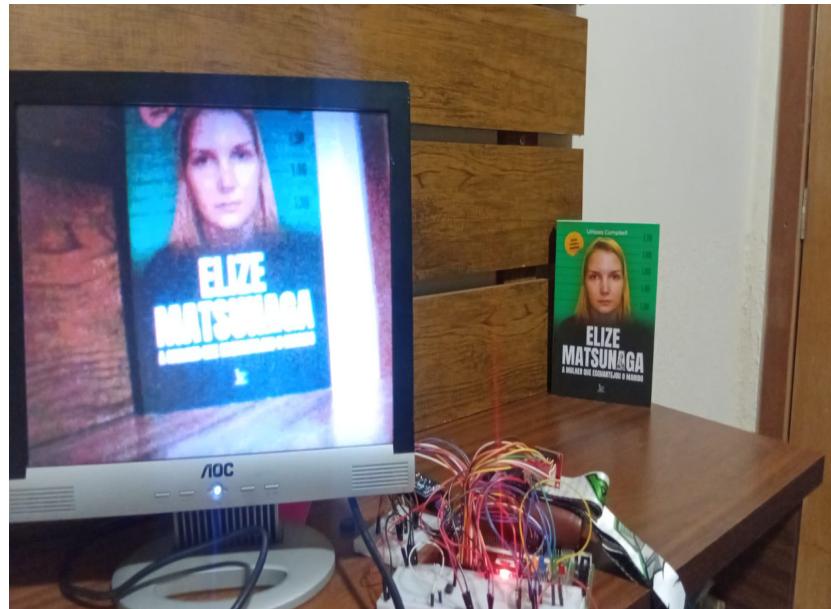


Figura 13: Livro com título grande

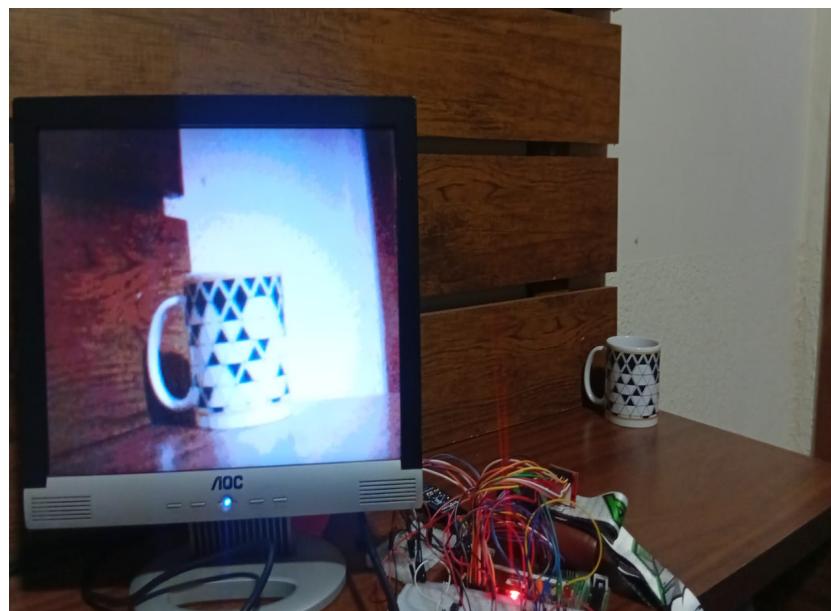


Figura 14: Xícara

4.3 Pré-Processamento e Rastreamento Dinâmico (Color Tracker)

Para que a arquitetura opere com eficiência lógica, é imprescindível reduzir a dimensionalidade e focar na informação útil antes de qualquer etapa posterior. Enviar um quadro completo de vídeo para processamento buscaria dados em um "oceano" de ruído. Portanto, desenvolveu-se o módulo color_tracker, responsável por interceptar o fluxo da câmera OV2640, filtrar as cores em tempo real e calcular a posição e o tamanho do objeto de interesse (tinta/desenho) através de uma caixa delimitadora (Bounding Box).

A metodologia de implementação deste estágio foi construída sobre quatro pilares sequenciais e combinacionais:

4.3.1 Rastreamento Espacial Sincronizado

Diferente de matrizes em linguagens de software como Python ou C, o fluxo de vídeo na FPGA não possui endereços fixos de X e Y atrelados aos pixels que chegam. O módulo precisou inferir a posição geométrica do dado observando os sinais de controle da câmera.

Utilizando registradores internos (curr_x e curr_y), o sistema zera as coordenadas sempre que o sinal de fim de quadro (vsync em nível baixo) é detectado. O eixo *X* é incrementado apenas quando o sinal de linha válida (href) e de pixel pronto (valid_pixel) estão em nível lógico alto. Detectando a borda de descida do href (o exato momento em que a linha acaba), o eixo *X* é resetado e o contador de linhas *Y* é incrementado.

Para que o módulo de visão computacional possa calcular o centroide e a caixa delimitadora (*Bounding Box*) de um objeto, é fundamental reconstruir a coordenada espacial (*X, Y*) de cada pixel recebido sequencialmente.

O Código demonstra a lógica de rastreamento de posição utilizando os sinais nativos da câmera. O contador horizontal (curr_x) é incrementado a cada pixel válido e resetado ao final de cada linha. Simultaneamente, o sinal de sincronismo vertical (vsync) garante que ambas as coordenadas sejam zeradas a cada novo quadro, mantendo o sistema perfeitamente alinhado com a varredura física do sensor.

```
1 always @ (posedge clk) begin
2     // Quadro nao valido zera a posicao (curr_x, curr_y) do pixel
3     if (!vsync) begin
4         curr_x <= 0;
5         curr_y <= 0;
6     end else begin
7         // Conta pixel quando HREF alto (Linha valida) e pixel
8         valido
9             if (href && valid_pixel)
10                 curr_x <= curr_x + 1;
11
12         // Reseta a posicao X a cada fim de linha
13         else if (!href)
14             curr_x <= 0;
```

```

15     // Incrementa o contador das linhas para o eixo Y
16     if (end_of_line)
17         curr_y <= curr_y + 1;
18     end
19 end

```

Listing 9: Rastreamento em tempo real das coordenadas X e Y do pixel em processamento.

4.3.2 Filtro de Cor Dinâmico e Rejeição de Ruído Lógico

O pixel recebido da câmera no formato RGB565 (16 bits) é imediatamente fatiado em seus canais constituintes através de wire slicing ($r = \text{pixel_data}[15:11]$, $g = \text{pixel_data}[10:5]$, $b = \text{pixel_data}[4:0]$). Uma condição combinacional simples (`is_color`) avalia se esses canais estão dentro de limiares parametrizados (`target_r_min`, `target_r_max`, etc.), identificando a cor desejada.

Entretanto, durante os ensaios de bancada, notou-se que sensores CMOS captam ruídos isolados (pixels corrompidos que assumem cores aleatórias por um milissegundo). Para evitar que um único pixel defeituoso estragasse a geometria da Bounding Box, implementou-se um filtro anti-ruído espacial. O registrador `streak_count` exige que o sistema detecte pelo menos 4 pixels válidos seguidos na mesma linha ($\text{streak_count} \geq 4$) para que a informação seja considerada um objeto real e comece a contabilizar para a caixa delimitadora. Caso um pixel inválido surja, a contagem de `streak` é zerada.

A identificação de objetos na imagem baseia-se em uma filtragem cromática parametrizável operando em tempo real. Para determinar se um pixel pertence ao objeto de interesse, o módulo extrai os canais RGB565 e os compara simultaneamente com limiares máximos e mínimos definidos via *hardware* (*Dynamic Color Filtering*). O Código 10 isola e detalha esta lógica:

```

1 // --- 1. FILTRO DE COR DINAMICO (Extrai e Compara) ---
2 wire [4:0] r = pixel_data[15:11];
3 wire [5:0] g = pixel_data[10:5];
4 wire [4:0] b = pixel_data[4:0];
5
6 // Compara o pixel atual com as margens de cor aceitaveis
7 wire is_color = (r >= target_r_min && r <= target_r_max) &&
8             (g >= target_g_min && g <= target_g_max) &&
9             (b >= target_b_min && b <= target_b_max);
10
11 // --- 2. REJEICAO DE RUIDO LOGICO (Spatial Filter) ---
12 always @ (posedge clk) begin
13     if (vsync && href && valid_pixel) begin
14         if (is_color) begin
15             // Incrementa a contagem de pixels consecutivos
16             if (streak_count < 15)
17                 streak_count <= streak_count + 1;
18
19             // CORTE DE RUIDO: Considera valido apenas apois 4
20             pixels seguidos
21             if (streak_count >= 4) begin

```

```

21         count <= count + 1; // Incrementa total de pixels
22         reais do objeto
23
24             // (Atualizacao dos limites da Bounding Box ocorre
25             aqui)
26             end
27         end else begin
28             // Quebrou a sequencia espacial: zera o contador de
29             ruidos
30             streak_count <= 0;
31         end
32     end else if (!href) begin
33         // Fim da linha quebra a continuidade fisica: zera o
34         contador
35         streak_count <= 0;
36     end
37 end

```

Listing 10: Filtro de cor dinâmico e rejeição de ruído espacial através de contador de sequência.

4.3.3 Extração da Bounding Box e Centroide

Uma vez superado o filtro de ruído, o sistema rastreia os limites do objeto. Os registradores `x_min`, `x_max`, `y_min` e `y_max` são constantemente comparados com a posição atual (`curr_x`, `curr_y`). Se o pixel atual extrapolar os valores armazenados, o limite da caixa "estica" para englobar o novo pixel.

O cálculo do centro geométrico do objeto só ocorre no final do quadro de vídeo. Para detectar este momento exato, criou-se um detector de borda de descida no sinal `vsync` (`end_of_frame`). Ao fim do quadro, o hardware soma os vértices e calcula a média. Na engenharia de hardware, divisões consomem muitos blocos lógicos. A solução adotada foi o uso do deslocamento de bits à direita (`right-shift`, `>> 1`), que executa matematicamente uma divisão por 2 instantânea.

A extração das métricas espaciais do objeto baseia-se na atualização contínua das coordenadas extremas (`x_min`, `x_max`, `y_min`, `y_max`). A cada pixel validado pelo filtro de cor e pelo mecanismo de rejeição de ruído, o sistema compara a posição atual com os limites registrados, expandindo a *Bounding Box* dinamicamente durante a varredura ativa do quadro.

O cálculo final do centroide geométrico e dos raios (largura e altura parciais) ocorre no momento em que o *frame* termina, sinalizado pela borda de descida do `vsync`. O Código 11 demonstra a eficiência computacional desta etapa: em vez de instanciar blocos divisores complexos em *hardware*, o sistema utiliza a operação de deslocamento lógico para a direita (`>> 1`) para realizar a divisão por 2 e encontrar o ponto médio das coordenadas.

```

1 // --- 1. ATUALIZACAO DINAMICA DOS VERTICES (Durante o Frame) ---
2 // Ocorre sempre que um pixel valido (e sem ruido) e detectado
3 if (curr_x < x_min) x_min <= curr_x;
4 if (curr_x > x_max) x_max <= curr_x;
5 if (curr_y < y_min) y_min <= curr_y;

```

```

6 if (curr_y > y_max) y_max <= curr_y;
7
8 // --- 2. CALCULO DO CENTROIDE E DIMENSOES (Fim do Frame) ---
9 // end_of_frame = borda de descida do VSYNC
10 if (end_of_frame) begin
11     // Filtro final: o objeto deve ter um volume minimo (ex: > 100
12     // pixels)
13     if (count > 100) begin
14         // Centroide (Media entre max e min): SOMA e DIVIDE POR 2
15         (>> 1)
16         obj_x <= (x_min + x_max) >> 1;
17         obj_y <= (y_min + y_max) >> 1;
18
19         // Calculo dos Raios (Metade da largura e altura + folga
20         // visual)
21         obj_half_w <= ((x_max - x_min) >> 1) + 4;
22         obj_half_h <= ((y_max - y_min) >> 1) + 4;
23
24         obj_detected <= 1;
25     end else begin
26         obj_detected <= 0;
27     end
28
29     // Prepara a memoria interna para a varredura do proximo quadro
30     count <= 0;
31     x_min <= 319; // Valor maximo inicial (para poder diminuir)
32     x_max <= 0; // Valor minimo inicial (para poder aumentar)
33     y_min <= 239;
34     y_max <= 0;
35
36 end

```

Listing 11: Atualização contínua dos vértices e cálculo final do centroide por deslocamento lógico.

4.3.4 Parametrização de Área

Para garantir que pequenos borrões na folha não acionassem o reconhecimento da inteligência artificial precocemente, estabeleceu-se um parâmetro de volume. O registrador count totaliza a massa do objeto; apenas se o objeto for maior que 100 pixels válidos ($count > 100$), a flag de saída `obj_detected` vai a nível lógico alto.

Essa mesma lógica foi atrelada fisicamente aos LEDs de debug da placa, onde o LED azul sinaliza a detecção estável de um volume contendo a cor alvo, permitindo ao usuário validar a iluminação do ambiente e o foco da câmera sem olhar para o monitor.

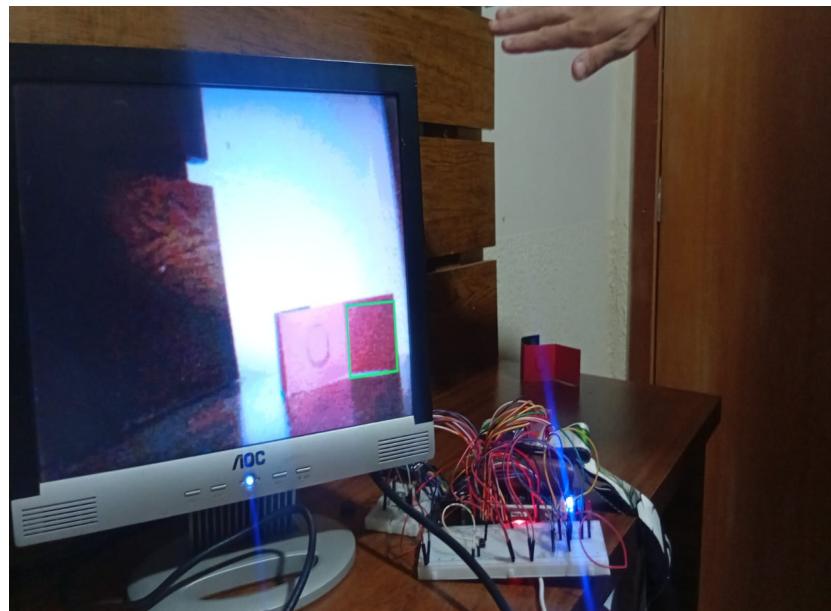


Figura 15: Objeto vermelho canto direito inferior

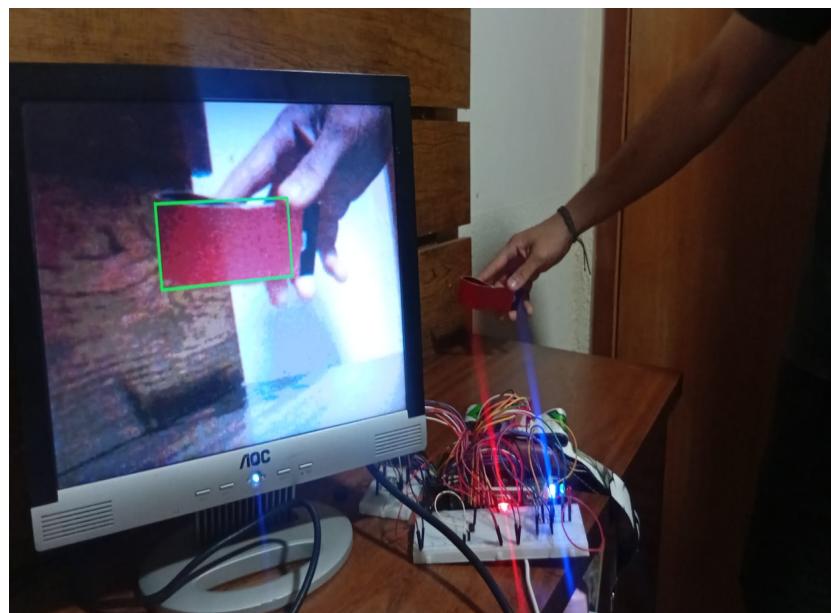


Figura 16: Objeto retangular horizontal vermelho no centro

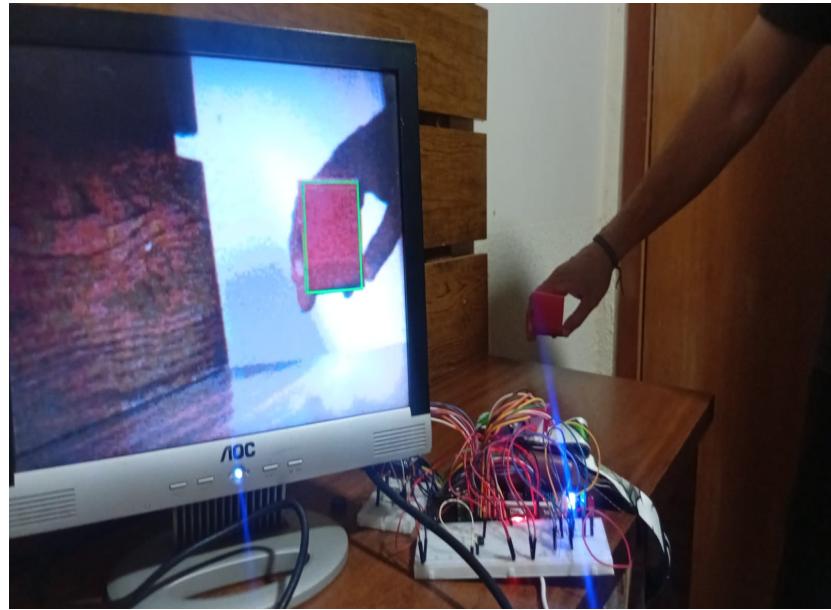


Figura 17: Objeto retangular vertical vermelho no centro

4.4 O Núcleo de Inferência em Hardware: Classificador MNIST (Template Matching)

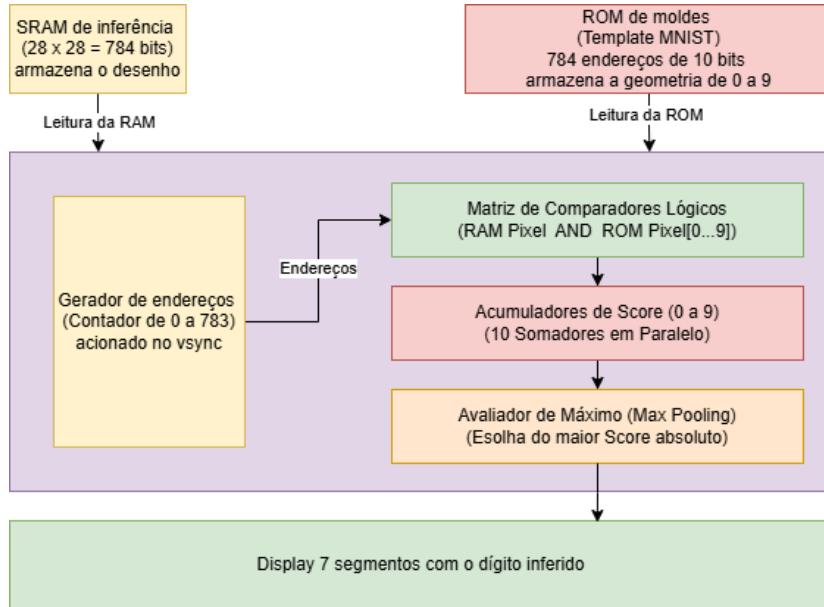


Figura 18: Objeto retangular vertical vermelho no centro

Com o objeto fisicamente isolado e enquadrado pelo Color Tracker, a arquitetura necessita extrair o significado semântico dos pixels. Diferente de implementações em software, onde matrizes são processadas por CPUs através de bibliotecas prontas, a inferência na FPGA exige um caminho de dados projetado a nível de registradores (RTL). Devido às restrições de blocos multiplicadores (DSPs) para a síntese de uma Rede Neural Convolucional (CNN), optou-se pela construção de um classificador geométrico estatístico baseado em Correspondência de Molde (Template Matching).

A metodologia de construção e validação do módulo classificador foi estruturada nas seguintes etapas:

4.4.1 Modelagem das Memórias de Inferência (RAM e ROM)

O sistema requer duas memórias distintas para operar. Primeiro, o fluxo contínuo de vídeo binarizado do Tracker é reduzido para uma resolução padrão de 28×28 pixels (tamanho canônico do dataset MNIST) e gravado sequencialmente em uma SRAM interna. Esta RAM atua como a "memória de curto prazo" do sistema, contendo o dígito que o usuário acabou de desenhar.

Em paralelo, instanciou-se uma memória ROM (Read-Only Memory) de 784 endereços (28×28) com largura de 10 bits. Cada bit desta palavra de

memória representa o pixel ideal de um dos 10 dígitos possíveis (0 a 9). Como a geometria codificada em Verilog ($10'b0001000100$, etc.) é visualmente opaca para o desenvolvedor durante a síntese, adotou-se uma metodologia de validação computacional auxiliar.

Foi desenvolvido um script na linguagem Python (via plataforma Google Colab) para realizar o parsing textual da ROM em Verilog, reconstruindo as matrizes de memória em imagens 2D. Esse procedimento de engenharia reversa garantiu que a FPGA estivesse sendo programada com os "moldes" geométricos corretos antes da gravação física na placa.

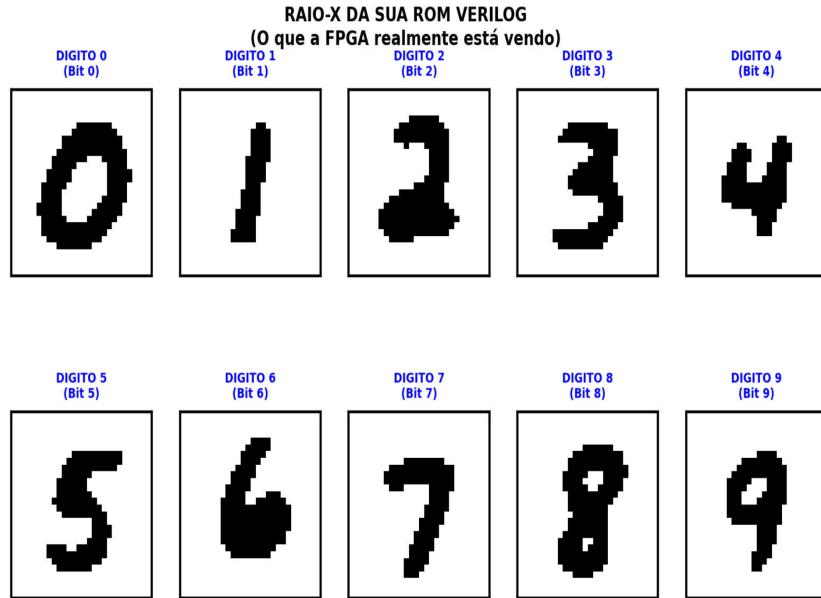


Figura 19: Dígitos MNIST

A implementação da rede de inferência para a classificação de dígitos (baseada no *dataset* MNIST) exigiu o armazenamento dos "moldes" de cada número (0 a 9) diretamente no *hardware*. Em vez de utilizar múltiplos blocos de memória RAM ou instanciar decodificadores complexos, optou-se por inferir uma memória ROM combinacional nativa.

Isso foi realizado através de uma função (`get_rom_data`) contendo uma estrutura `case` que mapeia os 784 endereços espaciais, correspondentes à matriz 28×28 da imagem de entrada. O grande diferencial de engenharia desta arquitetura é o uso de uma única palavra de 10 *bits* por endereço. Cada *bit* do barramento de saída representa a presença (1) ou ausência (0) de "tinta" para uma classe específica naquela coordenada exata. O *bit* 0 representa o molde do dígito 0, o *bit* 1 o molde do dígito 1, operando de forma paralela para as 10 classes.

O Código exibe um trecho estrutural desta matriz, evidenciando como os

múltiplos padrões foram comprimidos para otimizar a leitura lógica:

```

1 // Função atuando como ROM combinacional (784 endereços = Matriz 28
   x28)
2 function [9:0] get_rom_data;
3   input [9:0] addr;
4   begin
5     case (addr)
6       // [...] (Endereços iniciais omitidos para brevidade)
7
8       // Cada bit [9:0] indica se a classe respectiva (9 a 0)
9       // possui pixel ativo nesta coordenada específica.
10      10'd153 : rom_data = 10'b0101001101;
11      10'd154 : rom_data = 10'b0101001101;
12      10'd155 : rom_data = 10'b0101001111;
13      10'd156 : rom_data = 10'b0101001111;
14
15      // [...] (Mapeamento contínuo até o endereço 783)
16
17      default : rom_data = 10'b0000000000;
18    endcase
19  end
20 endfunction

```

Listing 12: Estrutura combinacional da ROM com moldes dos dígitos comprimidos em 10 bits.

4.4.2 O Motor Matemático de Comparação

A inferência ocorre instantaneamente após o término do quadro de vídeo (engatilhada pela borda do sinal vsync). Uma máquina de estados varre simultaneamente o endereço i da SRAM (desenho real) e o endereço i da ROM (molde ideal), varrendo de 0 a 783. Para cada dígito candidato $k \in \{0, 1, \dots, 9\}$, o sistema calcula uma pontuação (*Score*) baseada na coincidência lógica entre a tinta detectada na RAM e a tinta esperada na ROM. A equação lógica simplificada implementada em portas lógicas para cada bit k é:

$$Score_k = \sum_{i=0}^{783} (PixelRAM_i \text{ AND } PixelROM_{i,k}) \quad (1)$$

A implementação física da somatória da Equação ocorre dentro da máquina de estados do módulo classificador. O Código isola o bloco matemático responsável pela inferência.

Para otimizar o uso de silício, em vez de instanciar circuitos avaliadores individuais para cada um dos 10 dígitos, o *design* emprega um laço **for** gerador. A cada ciclo de *clock* em que uma coordenada contém “tinta” desenhada na tela (`ram_data == 1`), o sistema varre simultaneamente os 10 bits da palavra da ROM. Caso haja intersecção lógica entre o desenho e o molde (ou nas adjacências, devido à tolerância espacial), o *score* do dígito correspondente é bonificado (+10). Caso contrário, ele é penalizado (-20).

```

1 CALC: begin
2     // Condicao 1: So avalia se o usuario desenhou "tinta" no pixel
3     // atual da RAM
4     if (ram_data == 1) begin
5         total_ink <= total_ink + 1; // Contabiliza tinta total para
6         filtro de ruído
7
8         // Avalia paralelamente o "match" para os 10 digitos (0 a
9         9)
10        for (i=0; i<10; i=i+1) begin
11
12            // Condicao 2: Verifica se o molde (ROM) tambem tem
13            // tinta.
14            // A inclusao de rom_data_left e right cria uma
15            // tolerancia espacial (borrao)
16            if (rom_data[i] || rom_data_left[i] || rom_data_right[i])
17                begin
18                    score[i] <= score[i] + 10; // INTERSECAO
19                    VERDADEIRA: Acertou!
20                end else begin
21                    score[i] <= score[i] - 20; // FALSO POSITIVO:
22                    Penaliza fortemente
23                end
24
25            end
26        end
27
28        // (Logica de incremento de endereco omitida)
29    end

```

Listing 13: Motor matemático de comparação paralela e sistema de pontuação com tolerância espacial.

4.4.3 Decisão de Máxima Verossimilhança e Interface de Saída

No último ciclo da varredura (endereço 783), o sistema possui 10 pontuações consolidadas. Uma árvore de comparadores lógicos combinacionais ($>$ e $<$) determina instantaneamente qual dos registradores possui o maior valor absoluto, elegendo o dígito “vencedor”.

Após a varredura completa dos 784 pixels da matriz (estado **RESULT**), o classificador precisa determinar qual dígito obteve o maior índice de correlação espacial. Em arquiteturas de aprendizado de máquina, esta etapa de decisão final é conhecida como função *ArgMax*.

A implementação do *ArgMax* em *software* geralmente consome múltiplos ciclos de instrução para iterar sobre um *array*. Em contrapartida, a síntese em *hardware* permite realizar essa verificação em um único ciclo de *clock*, pagando-se o preço em área de silício. O Código 14 demonstra a topologia adotada: uma árvore de comparadores lógicos em cascata (*priority encoder*) que avalia as 10 pontuações simultaneamente e atribui o índice vencedor ao registrador **prediction**.

```

1 // ESTADO: RESULT
2 // Avalia qual classe (0 a 9) obteve o maior 'score' final

```

```

3
4 if (score[1] >= score[0] && score[1] >= score[2] && score[1] >=
   score[3] && score[1] >= score[4] && score[1] >= score[5] &&
   score[1] >= score[6] && score[1] >= score[7] && score[1] >=
   score[8] && score[1] >= score[9])
  prediction <= 1;
5 else if (score[2] >= score[0] && score[2] >= score[3] && score[2]
          >= score[4] && score[2] >= score[5] && score[2] >= score[6] &&
          score[2] >= score[7] && score[2] >= score[8] && score[2] >=
          score[9])
  prediction <= 2;
6 else if (score[3] >= score[0] && score[3] >= score[4] && score[3]
          >= score[5] && score[3] >= score[6] && score[3] >= score[7] &&
          score[3] >= score[8] && score[3] >= score[9])
  prediction <= 3;
7 else if (score[4] >= score[0] && score[4] >= score[5] && score[4]
          >= score[6] && score[4] >= score[7] && score[4] >= score[8] &&
          score[4] >= score[9])
  prediction <= 4;
8 else if (score[5] >= score[0] && score[5] >= score[6] && score[5]
          >= score[7] && score[5] >= score[8] && score[5] >= score[9])
  prediction <= 5;
9 else if (score[6] >= score[0] && score[6] >= score[7] && score[6]
          >= score[8] && score[6] >= score[9])
  prediction <= 6;
10 else if (score[7] >= score[0] && score[7] >= score[8] && score[7]
           >= score[9])
  prediction <= 7;
11 else if (score[8] >= score[0] && score[8] >= score[9])
  prediction <= 8;
12 else if (score[9] >= score[0])
  prediction <= 9;
13 else
  prediction <= 0; // Se nenhum for estritamente maior, assume 0

```

Listing 14: Implementação da função *ArgMax* em hardware através de árvore de comparadores paralelos.

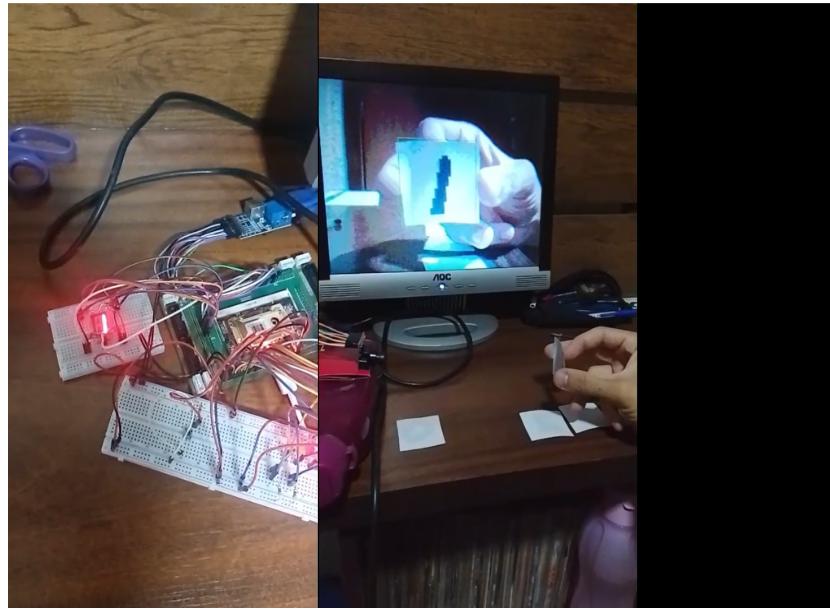


Figura 20: Dígito um

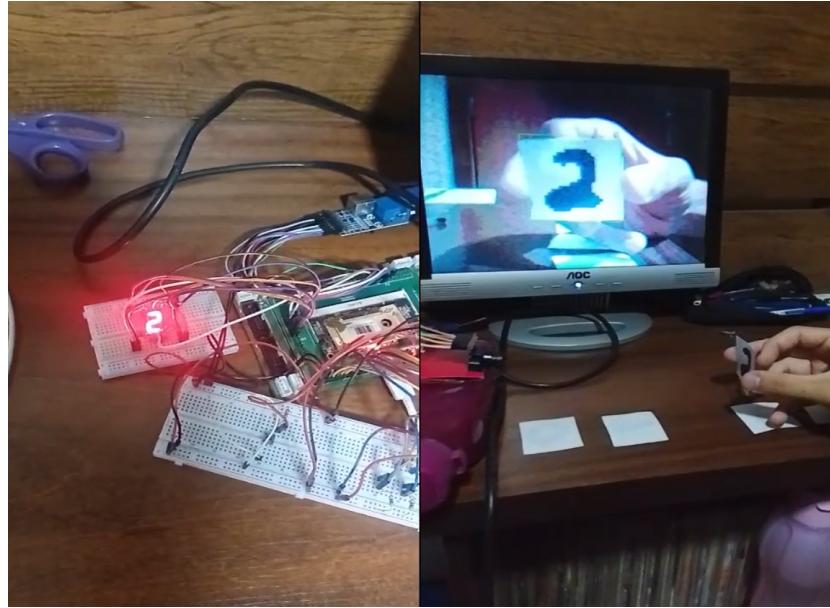


Figura 21: Dígito dois



Figura 22: Dígito três

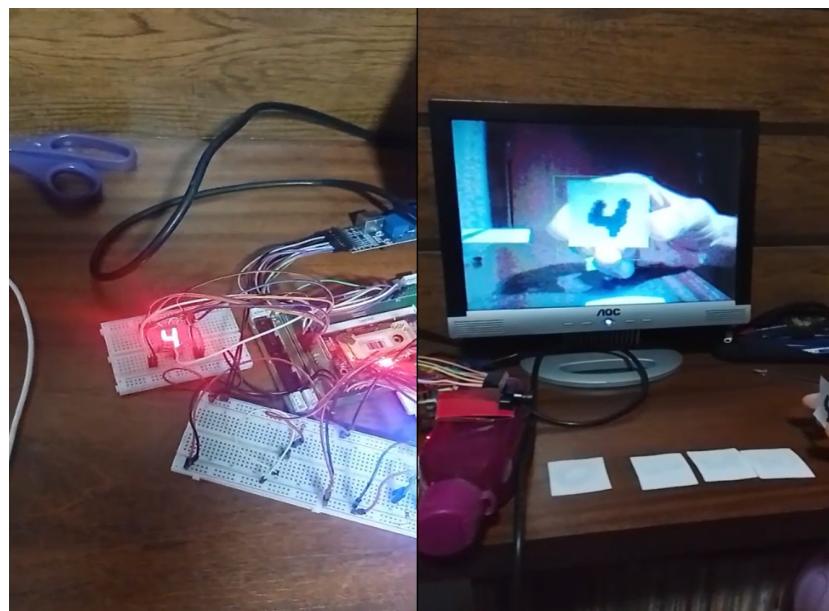


Figura 23: Dígito quatro

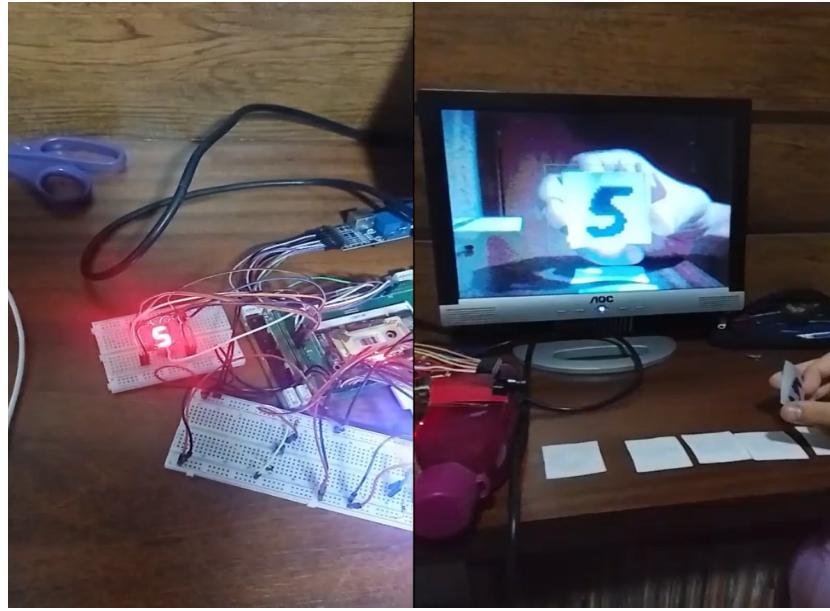


Figura 24: Dígito cinco



Figura 25: Dígito seis



Figura 26: Dígito sete

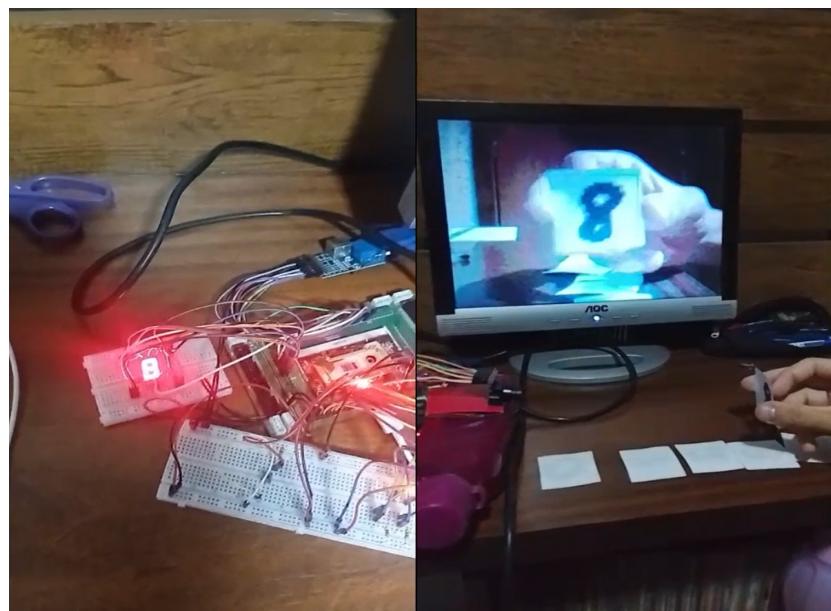


Figura 27: Dígito oito

5 Resultados e Discussões

Neste capítulo, são apresentados os resultados práticos obtidos após a síntese da arquitetura descrita na Metodologia. As validações ocorreram através de ensaios físicos na bancada, comprovando a eficácia do processamento de vídeo em hardware puro, a robustez do rastreamento de cor e a precisão do classificador de dígitos acoplado aos periféricos de saída.

5.1 Sincronismo de Vídeo e Aquisição de Imagem

O primeiro resultado crítico para o sucesso da arquitetura foi a comprovação de que o cruzamento de domínios de clock (Câmera → RAM → VGA) operava de maneira fluida.



Figura 28: Câmera, módulo VGA e framebuffer sincronizados

O monitor foi capaz de exibir o fluxo da câmera OV2640 com estabilidade a 60 Hz, sem artefatos visuais de corrupção de memória (tearing) ou perda de sincronismo (flickering). Isso atesta que o módulo framebuffer em RAM de porta dupla (Dual-Port) atuou perfeitamente como amortecedor lógico entre a escrita assíncrona do sensor CMOS e a varredura rígida demandada pelo padrão VGA. Adicionalmente, o módulo interpolador (vga_upscaler) comprovou sua eficácia matemática, redimensionando a imagem capturada em baixa resolução para ocupar a área útil do display sem distorções severas, levando conta nossa discussão da perda da qualidade pelos diferentes filtros na quantidade de bits reservados para cada pixel.

5.2 Rastreamento e Binarização em Tempo Real

Para avaliar o desempenho do Color Tracker, a câmera foi submetida a diferentes angulações e rotações do objeto vermelho em questão. O quadrado delimitador verde (Bounding Box), injetado diretamente no fluxo de vídeo por multiplexação combinacional, circunda exatamente os extremos geométricos do caractere.

A limiarização de cor (Thresholding) e o filtro espacial anti-ruído (que exigia um mínimo de pixels consecutivos) foram suficientes para ignorar pequenas imperfeições e sombras na folha.

A latência de cálculo da Região de Interesse (ROI) é imperceptível, uma vez que o overlay verde acompanha o movimento do papel em tempo real na cadência máxima do monitor.

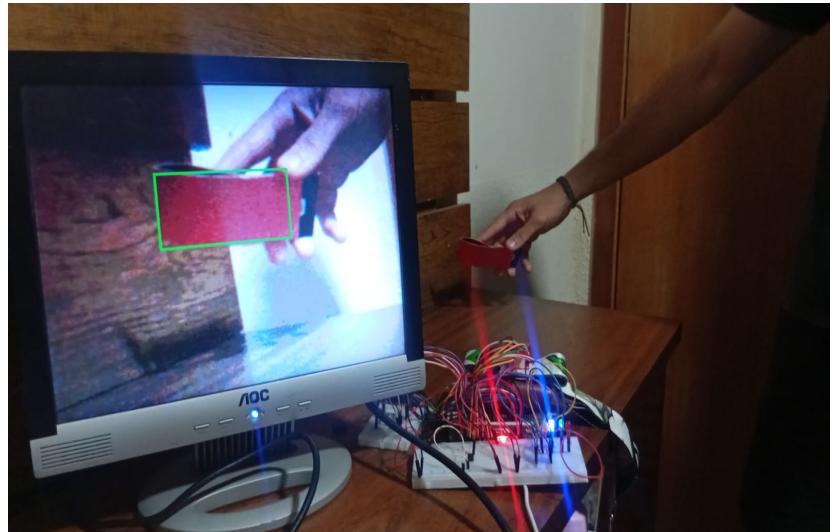


Figura 29: Color tracker e Bounding Box

5.3 Acurácia da Inferência

O teste definitivo da arquitetura consolidou o pipeline de captura com o módulo MNIST. Durante os testes, a implementação nativa do comparador lógico mostrou-se extremamente sensível à tremulação humana. Contudo, após a integração da heurística de "Visão Periférica" (onde a leitura da ROM utiliza portas lógicas OR abrangendo os endereços adjacentes laterais, rom_data_left e rom_data_right), observou-se um ganho drástico na tolerância espacial. O usuário adquiriu a liberdade de realizar leves deslocamentos horizontais com o papel sem que a pontuação (score) máxima fosse perdida.



Figura 30: Dígito 9

6 Conclusões e Trabalhos Futuros

6.1 Conclusões

O presente trabalho cumpriu com êxito o objetivo de desenvolver uma arquitetura digital completa para processamento de vídeo e inferência estatística embarcada em FPGA. Ao longo do desenvolvimento, comprovou-se na prática a tese de que a transição de algoritmos baseados em software sequencial para hardware de fluxo contínuo (data streaming) elimina o gargalo clássico de memória (von Neumann), resultando em uma latência de processamento praticamente nula.

A escalada metodológica modular provou-se fundamental para o sucesso da integração. O domínio sobre a geração de sinais VGA estritos e a decodificação do protocolo DVP em domínios de clock assíncronos permitiu a construção de um pipeline de captura robusto. Destaca-se como um dos maiores êxitos de projeto a implementação do Módulo Pré-Processador (Color Tracker). A capacidade de realizar a limiarização, o isolamento geométrico do alvo (Bounding Box) e a geração de uma interface visual com o usuário (o overlay do quadrado verde) puramente por multiplexação lógica combinacional demonstrou uma altíssima eficiência no uso de recursos, poupando a escassa memória interna da FPGA.

No tocante à inteligência do sistema, o classificador baseado em Correspondência de Molde Euclidiana (Template Matching) atingiu os requisitos propostos para a prova de conceito. A adoção da heurística de "Visão Periférica", relaxando a exigência de intersecção exata através de operadores lógicos espaciais (OR), mitigou de forma satisfatória a maior fraqueza teórica deste modelo:

a extrema sensibilidade à translação humana durante o desenho.

Contudo, para fins de transparência acadêmica, conclui-se que o modelo implementado possui limitações intrínsecas ao paradigma não-convolucional. O sistema exige que o dígito seja desenhado com espessura de traço e tamanho condizentes com os moldes pré-gravados na ROM. Pequenas Variações de escala, rotações no papel ou deformações topológicas resultam em falhas de classificação, evidenciando o limite da técnica estatística crua frente a redes de aprendizado profundo (Deep Learning). Ainda assim, o projeto atesta que, para ambientes de altíssima restrição de hardware e consumo energético, matrizes combinacionais oferecem uma alternativa formidável e instantânea aos microprocessadores tradicionais.

6.2 Trabalhos Futuros

A arquitetura consolidada neste trabalho pavimenta um terreno vasto para futuras pesquisas e otimizações na área de Edge Computing e microeletrônica. Como desdobramentos naturais deste projeto, propõem-se as seguintes frentes de avanço técnico:

1. **Aceleração de Redes Convolucionais (CNNs):** O passo evolutivo lógico para solucionar a vulnerabilidade de rotação e escala é a substituição do bloco de Template Matching por uma arquitetura convolucional completa (como a LeNet-5 e YoLo). Trabalhos futuros podem focar na alocação otimizada de blocos DSPs e na quantização de pesos (reduzindo redes de ponto flutuante para inteiros de 8 ou 4 bits) para caberem nas restrições da placa.
2. **Controladores de Memória Externa (SDRAM/DDR):** A atual limitação de instanciar apenas um framebuffer de baixa resolução devido à escassez de Block RAM interna pode ser superada desenvolvendo um controlador de memória de alta velocidade, permitindo o armazenamento e processamento do quadro VGA completo (640×480).
3. **Integração com Arquitetura RISC-V:** Uma abordagem híbrida (Hardware/Software Co-design) apresenta-se como um horizonte altamente promissor. A instanciação de um processador soft-core de arquitetura aberta, como o RISC-V, permitiria que a FPGA delegasse as tarefas sequenciais complexas de controle geral e interface de rede para o código em C, enquanto o processamento bruto e paralelo dos pixels da câmera permaneceria acelerado no hardware customizado desenvolvido neste projeto.

Referências

- [1] BROWN, S.; VRANESIC, Z. *Fundamentals of Digital Logic with Verilog Design*. 3. ed. Nova York: McGraw-Hill, 2013.

- [2] GONZALEZ, R. C.; WOODS, R. E. *Processamento Digital de Imagens*. 3. ed. São Paulo: Pearson Prentice Hall, 2010.
- [3] LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278-2324, nov. 1998.
- [4] OMNIVISION TECHNOLOGIES. *OV2640 Color CMOS UXGA (2.0 MegaPixel) CameraChip™ with OmniPixel2™ Technology: Datasheet*. Santa Clara, CA: OmniVision, 2006. Disponível em: <https://www.uctronics.com/download/cam_module/OV2640DS.pdf>. Acesso em: 22 fev. 2026.
- [5] SHI, F. *et al.* An FPGA-based hardware accelerator for template matching. In: *2017 IEEE 11th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. IEEE, 2017. p. 43-47.

Apêndices

A arquitetura desenvolvida neste projeto é composta por múltiplos módulos de hardware instanciados hierarquicamente, totalizando milhares de linhas de descrição em linguagem Verilog (RTL). Para facilitar a leitura deste documento, otimizar o espaço e, principalmente, permitir a replicação exata e o escrutínio técnico do hardware proposto, optou-se por não transcrever os códigos em sua totalidade nestas páginas impressas.

Todo o projeto — abrangendo os arquivos de design, os scripts de validação lógica e os arquivos de restrição física — encontra-se versionado, devidamente comentado e disponível publicamente na plataforma GitHub.

<https://github.com/Zazamartins/Visao-Computacional-com-FPGA>