Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Engenharia Elétrica

Uma Metodologia de Projeto e Validação de Sistemas de Detecção de Faces

Nelson Carlos de Sousa Campos

Dissertação de Mestrado submetida à Coordenadoria do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Antônio Marcus Nogueira Lima, Dr.
Orientador
Elmar Uwe Kurt Melcher, Dr.
Orientador

Campina Grande, Paraíba, Brasil

Uma Metodologia de Projeto e Validação de Sistemas de Detecção de Faces

Nelson Carlos de Sousa Campos

Dissertação de Mestrado apresentada em 29 de dezembro de 2017

Antônio Marcus Nogueira Lima, Dr.
Orientador
Elmar Uwe Kurt Melcher, Dr.
Orientador

Alisson Vasconcelos de Brito, Dr.
Componente da Banca

Ângelo Perkusich, Dr.
Componente da Banca

Marcos Ricardo Alcântara Morais, Dr.
Componente da Banca

Campina Grande, Paraíba, Brasil, 29 de dezembro de 2017

"When he had received the drink,
Jesus said: ***It is finished.***
With that, he bowed his head
and gave up his spirit."
—John 19:30

Dedicatória

Dedico este trabalho a todos aqueles que incessantemente lutam para atingir os seus objetivos e nunca desistem dos seus sonhos, indenpendentemente de suas origens e dos obstáculos que os acompanham durante a longa jornada da vida.

Agradecimentos

Agradeço primeiramente a Deus por cada dia que me é concedido. Agradeço também ao apoio da minha família que sempre me incentivou nos momentos difíceis.

Agradeço meus orientadores. Ao professor Antônio Marcus, que com seu nível de excelência sempre me incentivou a questionar os problemas com metodologia científica. Ao professor Elmar Melcher, pelo aprendizado técnico de alto nível e pela pontualidade que levarei pra vida profissional. Sem dúvidas, a soma das exigências desses professores contribuiram pra minha forma de resolver problemas de maneira criteriosa. Não poderia deixar de ser grato ao professor Heron Monteiro, sem o qual a conclusão deste trabalho não seria possível.

Agradeço aos meus amigos e colegas que de alguma forma contribuiram para o desenvolvimento deste trabalho, dentre os quais posso citar Elton Brasil, Felipe Gonçalves, Herbet, Plateny Ponchet e Thiago Cordeiro. Muitos deles não estão citados aqui, mas tiveram grande importância ao longo desta longa jornada.

Agradeço ao Programa de Pós-Graduação em Engenharia Elétrica (PPgEE - COPELE) da UFCG, e em particular à Ângela, pelo apoio e suporte durante esses longos dois anos em que estive no mestrado e à CAPES e ao CNPq pelo financiamento deste trabalho. Agradeço ao professor Gutemberg Júnior e a toda equipe do PEM pelas oportunidades que tive de trabalhar neste projeto.

Resumo

Uma metodologia para projeto em *hardware* e validação de sistemas de detecção de faces é discutida. O projeto começa em um alto nível de abstração usando uma biblioteca de *software* como o modelo de referência. O modelo de referência e um modelo de base (modelo em validação) são convertidos para uma descrição de nível de transação (TLM) ou uma descrição de nível de transferência de registradores (RTL). O grau de similaridade das faces detectadas é usado para determinar se o modelo em validação cumpre os requisitos de projeto prescritos. A metodologia foi concebida para aplicações de processamento de imagem de alta resolução em tempo real e é baseada na metodologia de verificação universal (UVM). Como estudo de caso, a metodologia foi aplicada para validar um modelo de base escrito em C++, utilizando funções da biblioteca OpenCV, considerada como o modelo de referência da especificação. Neste caso, ambos os modelos foram convertidos para TLM e, portanto, o tempo necessário para determinar se o modelo de linha de base é válido ou não é significativamente reduzido. Assim, pode-se decidir se o modelo TLM precisa ser refinado antes da conversão do modelo de base para um modelo RTL.

Palavras chave: Projeto de Hardware, Verificação Funcional, UVM, SystemC, UVM Connect, OpenCV, Viola Jones, Matching de Retângulos.

Abstract

A methodology for hardware design and validation of face detection systems is discussed. The design begins at a high-level of abstraction by using a software library as the golden reference model. The golden reference model and the baseline model (model under validation) are converted to a transaction level (TLM) description or a register transfer level (RTL) description. The degree of similarity of the detected faces is used to determine whether the model under validation fulfills the prescribed design requirements. The methodology is conceived for real time high resolution image processing applications and is based on the universal verification methodology (UVM). As a case study, the methodology was applied to validate a C++ baseline model written to implement the OpenCV library considered as the golden reference model. In this case both models have been converted to TLM and thus the time required for determining whether the baseline model is valid or not is significantly reduced. Thus, one can decide before proceeding with the conversion of the baseline model to RTL, if the TLM model needs improvement.

Keywords: Hardware Design, Verification, UVM, SystemC, UVM Connect, OpenCV, Viola Jones, Rectangles Matching.

Sumário

Lista de abreviaturas, acrônimos e siglas

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com a crescente violência na sociedade, como a onda de terrorismo que assola o mundo, o cenário de vigilância tem sido fortalecido com a proliferação de câmeras inteligentes nos mais diversos locais, como por exemplo em aeroportos. Essas câmeras contém sensores de imagem de resolução HD ou Full HD[1], e também podem conter módulos de processamento matemático que possibilitam a detecção e o reconhecimento de faces de um determinado indivíduo.

A detecção de faces é uma tarefa nos campos de pesquisa da visão computacional e processamento digital de imagens e é o primeiro passo para o reconhecimento facial, que possui aplicações em neurociência, psicologia, interação entre homem-máquina, vigilância de vídeo, dentre outras [2][3][4]. Várias técnicas foram propostas para detecção de faces nas últimas décadas. Entre elas, soluções com Redes Neurais [5], Máquinas de Vetores de Suporte (SVM, do inglês: *Support Vector Machine*) [6], Modelos Ocultos de Markov (HMM, do inglês: *Hidden Markov Models*) [7] e a estrututa proposta por Viola-Jones, que foi um dos primeiros métodos de detecção de faces concebida para processamento em tempo real [8].

1.1 Motivação

As aplicações de processamento de vídeo requerem a manipulação de um grande volume de dados de imagens com operações de alto esforço computacional.

[1] Em [1] é proposto um sitema de vigilância utilizando câmeras de alta resolução.

Frequentemente o projeto de sistemas de processamento de imagens deve ser conduzido com restrições de minimização de tamanho, peso e baixo consumo de energia, que podem ser satisfeitas com uma combinação de soluções de *hardware* e *software*, onde as partes críticas do aplicativo são implementadas em *hardware* ou em arquiteturas dedicadas para acelerar a computação do sistema.

O projeto de circuitos integrados mais complexos, viabilizado pelo avanço das tecnologias VLSI (do inglês: *Very Large Scale Integration*), é impulsionado por diversas demandas de consumidores dos sistemas que usam tais circuitos, sejam eles leigos, acadêmicos, militares ou da indústria. Os circuitos integrados que realizam operações de detecção e reconhecimento de faces são descritos por arquiteturas da ordem de milhões de transistores. A verificação funcional é um dos maiores gargalos no fluxo de projeto de arquiteturas dessa complexidade, consumindo cerca de 70% do recursos do projeto [9]. Ela fortalece o grau de confiança do correto funcionamento de circuitos integrados digitais antes mesmo de sua fabricação, evitando assim prejuízos da ordem de milhões de dólares. Portanto, considerando-se a verificação funcional como um requisito indispensável na execução desse fluxo, ela deve ser tomada como prioridade ainda nos estágios iniciais do projeto.

1.2 Definição do problema

O tempo para o mercado é o tempo total para a finalização de um projeto e a introdução do produto no mercado. Este tempo é um fator determinante no fluxo de projetos de sistemas digitais. A prototipação rápida do sistema pode ser feita em arquiteturas reprogramáveis (FPGA, do inglês: *Field Programmable Gate Array*), que são circuitos integrados que podem ser redescritos pelo projetista. Como já mencionado anteriormente, a verificação funcional é uma etapa que consome grande parte do tempo no fluxo de projeto. A implementação da arquitetura de sistemas complexos não é completa nos estágios iniciais do projeto, o que pode aumentar o tempo de mercado. Para resolver este problema, os ambientes de validação devem ser criados nos estágios iniciais do projeto, antes que a arquitetura do sistema seja definida, permitindo assim que a validação do sistema seja realizada de maneira incremental em todo o fluxo do projeto, além de

explorar o conceito de reuso avaliando se uma aplicação de domínio público (por exemplo, um modelo de detecção de faces escrito em C++) pode ser utilizado como modelo de referência para a descrição arquitetural de um circuito dedicado para detecção de faces.

1.3 Objetivo geral

O objetivo geral desta dissertação é desenvolver uma metodologia de projeto e validação de sistemas digitais com aplicações em processamento de imagens e visão computacional, tendo como aplicação particular sistemas de detecção de faces.

1.3.1 Objetivos específicos

Os objetivos específicos deste trabalho de dissertação estão listados a seguir:

- propor uma metodologia de projeto e validação de sistemas digitais com ênfase em procesamento digital de imagens para verificar funcionalmente algoritmos de visão computacional;
- adaptar a metodologia de validação do item anterior para validar sistemas digitais antes que o nível arquitetural esteja disponível nas fases iniciais do projeto, tendo como aplicação de estudo de caso a validação de um sistema de detecção de faces;
- projetar uma plataforma de prototipação em FPGA de sistemas digitais com ênfase em processamento de imagens e vídeo digital das aplicações propostas nos itens anteriores.

1.4 Metodologia

As etapas metodológicas que serão utilizadas neste trabalho de dissertação estão listadas a seguir:

- revisar a literatura com o intuito de analisar e comparar as soluções já existentes de projeto em *hardware* e validação funcional de sistemas com aplicações de processamento de imagens e visão computacional;

- implementar aplicações em *software* de processamento digital de imagens e visão computacional e avaliar estes modelos a partir de modelos já previamente validados;
- implementar aplicações em *hardware* de processamento digital de imagens e visão computacional e avaliar estes modelos a partir de modelos em *software* já previamente validados.

1.5 Organização da dissertação

A organização desse trabalho de dissertação está estruturada da seguinte forma:

- **Capítulo 1 - Introdução**
  A introdução (este capítulo) engloba o contexto e a motivação da pesquisa conduzida na dissertação.
- **Capítulo 2 - Trabalhos Relacionados**
  Este capítulo realiza uma revisão da literatura em trabalhos de verificação funcional de sistemas digitais e métricas de comparação de algoritmos de processamento de imagens.
- **Capítulo 3 - Fundamentação Teórica**
  A Fundamentação Teórica engloba com mais detalhes o contexto técnico que dá suporte para a pesquisa. De maneira geral, são discutidos tópicos em processamento digital de imagens, visão computacional, métodos de detecção de faces e o fluxo de projeto e validação de sistemas digitais.
- **Capítulo 4 - Estrutura Proposta**
  Este capítulo propõe uma estrutura de projeto e validação de sistemas de processamento de imagens, enfatizando a validação de um sistema de detecção de faces utilizando métricas de correspondência de objetos, com o intuito de avaliar um modelo em desenvolvimento em função de um modelo de referência.
- **Capítulo 5 - Resultados e discussões**
  Este capítulo realiza uma análise e discussão dos resultados obtidos no capítulo 4.

- **Capítulo 6 - Conclusões e trabalhos futuros**

  Este capítulo sumariza os capítulos 2, 3, 4 e 5 e enfatiza as contribuições do trabalho, além de enfatizar algumas sugestões de trabalhos futuros.

Capítulo 2

Trabalhos relacionados

Neste capítulo será apresentada uma revisão da literatura com relação aos tópicos de métricas de classificação de imagens e também verificação funcional de sistemas digitais, especialmente com foco em aplicações de processamento de imagens.

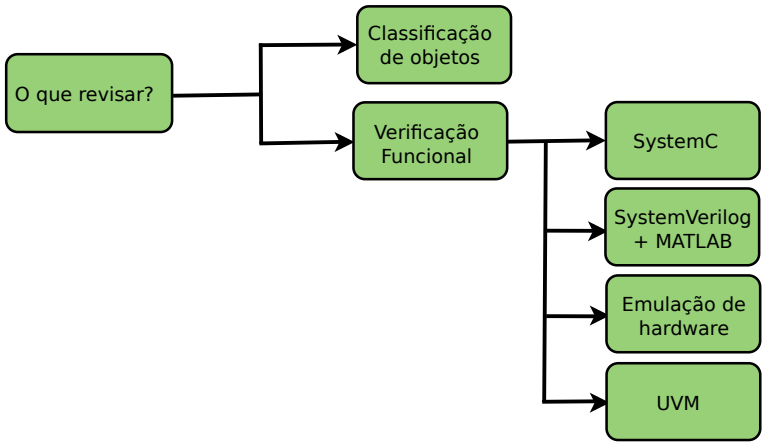
Em [10] é abordado o estudo de reuso de testes verticais (no campo de sistemas embarcados). Este artigo fornece um mapeamento sistemático sobre estudos prévios realizados em uma determinada área, levantando questões sobre quais trabalhos foram publicados na área de estudos verticais e classificando estes trabalhos por temas.

Neste presente trabalho de dissertação será abordada uma temática de classificação da literatura de acordo com a Figura 2.1. Os tópicos revisados estão divididos em classificação de objetos (imagens) e metodologias de verificação funcional de sistemas digitais.

2.1 Classificação de objetos

As métricas de classificação de objetos (como por exemplo, o coeficiente de *dice*), já discutidas na seção 3.3, tem aplicações em reconhecimento facial. Em [11] é proposta uma estrutura de projeto e validação de um sistema de reconhecimento da faces. A estrutura é escrita em C++ utilizando a biblioteca OpenCV, que fornece funções de processamento de imagens e visão computacional. O reconhecimento facial é dividido na seguintes etapas: a detecção dos objetos; o registro dos objetos detectados; a correção de luminosidade, que é uma etapa de processamento de imagens; a extração de características dos objetos

Figura 2.1: Organização da revisão bibliográfica



Fonte: Elaborada pelo autor

detectados e a comparação dos objetos detectados com os objetos de uma galeria de imagens armazenadas em uma base de dados.

O trabalho citado em [11] é um exemplo de uma plataforma de reconhecimento de faces com implementação em *software*. Embora estas implementações possam ser modelos de referência para sistemas descritos em *hardware*, a abordagem proposta em [11] é limitada para projetos deste tipo, que tem um fluxo de projeto mais complexo, além de requerer a validação com uso de metodologias de verificação funcional. Um fluxo de projeto adequado para esta situação será tratado no decorrer deste trabalho de dissertação.

2.2 Verificação funcional

Nesta seção são discutidos os trabalhos da literatura que abordavam os temas de verificação funcional utilizando. Dentre os métodos propostos nos trabalhos podem ser citados verificação funcional com SystemC, SystemVerilog e Matlab, verificação por emulação de *hardware* e a metodologia de verificação universal UVM.

2.2.1 Verificação funcional com SystemC

O trabalho [12] propõe a criação de uma nova metodologia, chamada de metodologia VeriSC, para verificação funcional de circuitos integrados digitais, que permite o

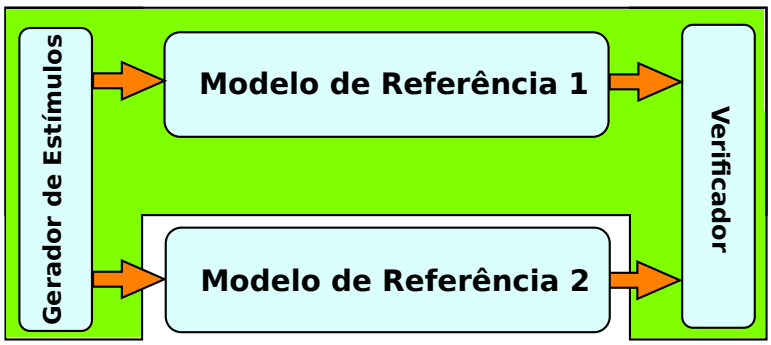
acompanhamento do fluxo de projeto, de forma que o *testbench* (ambiente de simulação para estimular e testar módulos descritos em *hardware*) seja gerado antes da implementação do dispositivo que está sendo verificado (DUT, do inglês: *Design Under Test*), tornando o processo de verificação funcional mais rápido e o *testbench* mais confiável, devido a ele ser verificado antes do início da verificação funcional do DUT [12].

A metodologia VeriSC é composta de um novo fluxo de verificação, que não se inicia pela implementação do DUT. Nesse fluxo, a implementação do *testbench* e do modelo de referência antecedem a implementação do DUT. Para permitir que o *testbench* seja implementado antes do DUT, a metodologia implementa um mecanismo para simular a presença do DUT com os próprios elementos do *testbench*, sem a necessidade da geração de código adicional que não será reutilizado depois.

A implementação do *testbench* é feita em pequenas etapas incrementais, cada passo resultando em uma simulação de trabalho que pode ser facilmente depurada. Essas etapas consistem também na reutilização de alguns módulos do *testbench* para substituir a funcionalidade do DUT.

Em um primeiro momento, como pode ser visto na Figura 2.2, o modelo de referência deve ser testado quanto à capacidade de interagir com o *testbench*, recebendo e produzindo dados de transações, que são gerados por um Gerador de Estímulos. As transações de entrada são processadas por dois modelos de referência, que posteriormente serão comparadas por um Verificador que tem por objetivo de validar a funcionalidade de um modelo de referência sob teste.

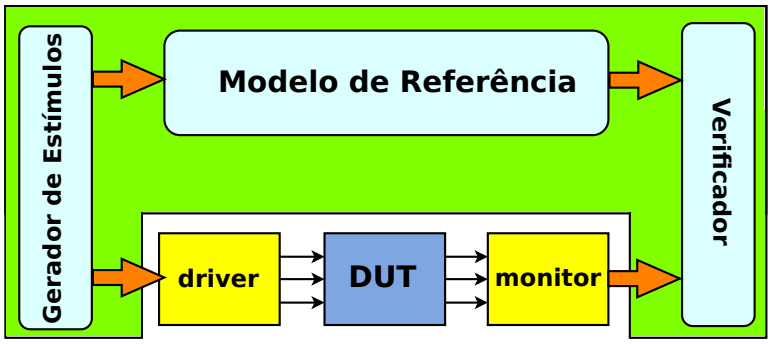
Figura 2.2: Testbench VesiSC: Validação do Modelo de Referência



Fonte: Adaptada de [12]

Numa etapa seguinte, como pode ser visto na Figura 2.3, o modelo de referência em teste é substituído pelo DUT, que tem na interface sinais digitais e pulsos lógicos. Para que a entrada do DUT compreenda os dados gerados pelo gerador de estímulos, um *driver* é adicionado na interface de entrada para converter transações em sinais. O *monitor* gera transações a partir dos sinais de saída do DUT para que o Verificador (que valida os estímulos) possa comparar as transações.

Figura 2.3: *Testbench* VesiSC: Validação do DUT



Fonte: Adaptada de [12]

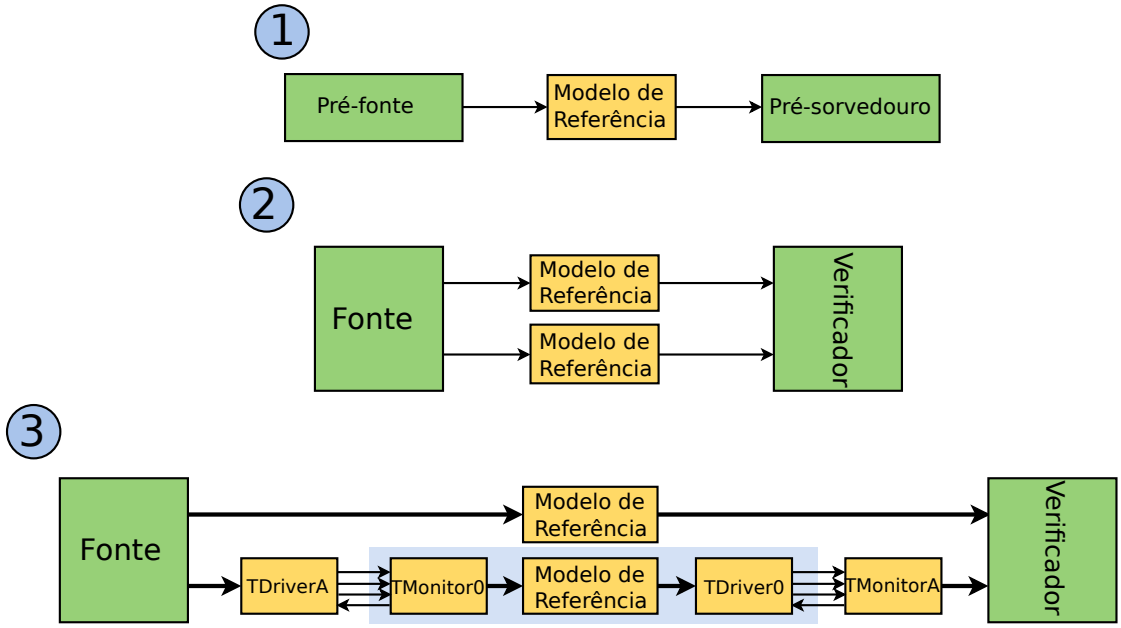
Em [13] é proposta a metodologia de verificação funcional VeriSC2, que utiliza a biblioteca de verificação funcional do SystemC (SCV, do inglês: *SystemC Verification*). Esta metodologia orienta a implementação de *testbenches* com decomposição hierárquica e permite o refinamento do projeto ainda antes que o projeto em nível de registradores esteja disponível.

Para implementar os *testbenches*, o primeiro passo é a geração do *testbench* para a validação do DUT completo. Este *testbench* é construído em três sub-etapas, conforme ilustrado na Figura 2.4. Primeiro o modelo de referência deve ser testado para avaliar sua interação com o ambiente de teste. Em seguida, a geração de estímulos deve ser processsada por duas instâncias do modelo de referência e os estímulos devem ser testados excitando-se a Fonte e o Verificador. Por fim, os blocos TDriver (que convertem estímulos em sinais de pinos) e TMonitor (que convertem sinais de pinos em estímulos) devem ser acoplados no DUT para excitá-lo e comparar seu comportamento com o modelo de referência.

Em [14] é proposta a refatoração de *testbenches* utilizando a metodologia VeriSC. A

refatoração proposta melhora a reutilização de componentes *testbench* que foram desenvolvidos durante a verificação de blocos autônomos e preserva as outras características desejáveis da VeriSC, como suporte a ferramentas para a construção *testbench*, reutilização de componentes do *testbench* durante a fase de decomposição e técnicas para sintetizar novos critérios de cobertura durante a fase de composição.

Figura 2.4: Geração de um Testbench VesiSC2: Validação de DUT



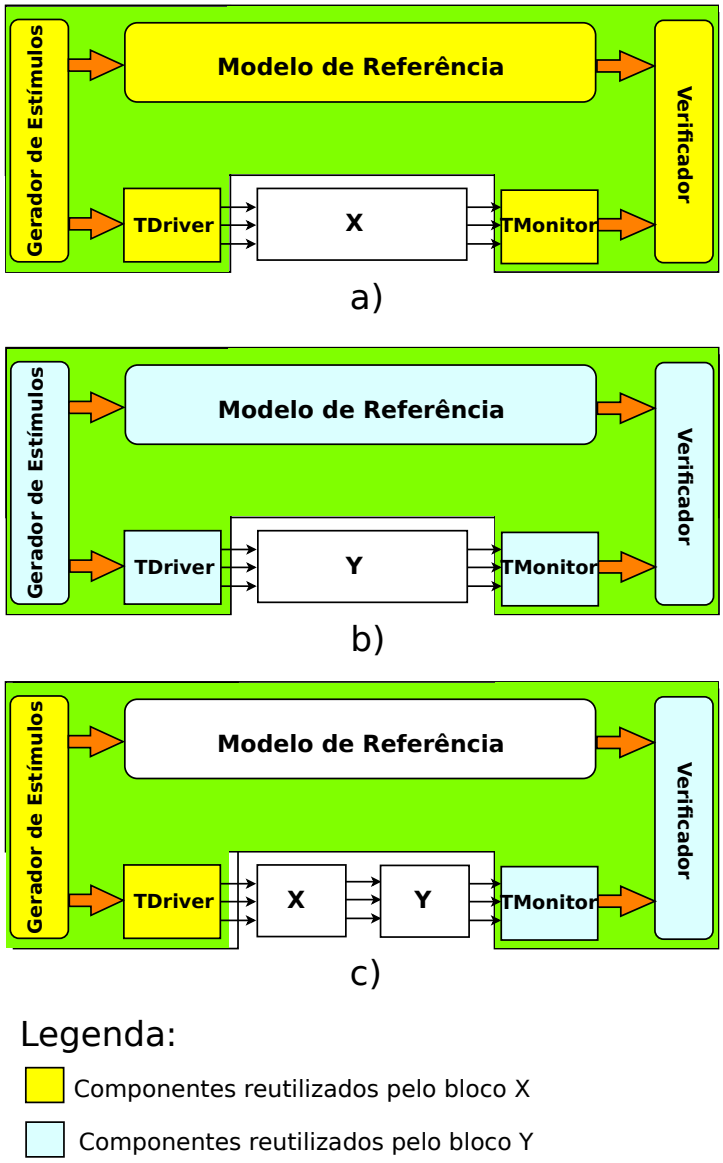
Fonte: Adaptada de [13]

Em um projeto hierárquico, a fase de composição é realizada combinando um par de blocos por tempo. O engenheiro de verificação pode combinar mais de dois blocos por hora, mas isso não é recomendado porque quando um erro é observado, o custo para localizá-lo é maior. Assim, é considerada a composição em que um par é combinado por tempo. A verificação funcional de dois blocos, de acordo com a metodologia VeriSC, requer um banco de teste para cada bloco (Figura 2.5, itens a e b). A composição de ambos os blocos produzirá um teste em que os componentes Fonte e TDriver relativos ao bloco X serão reutilizados (Figura 2, item c). Seguindo o mesmo princípio, os componentes TMonitor e Checker serão reutilizados a partir do bloco Y.

A metodologia VeriSC permite a construção de ambientes de validação antes que o circuito sob teste esteja disponível nas fases iniciais do fluxo de projeto, no entanto, a

indústria adotou recentemente como padrão de verificação a metodologia de verificação universal (UVM, do inglês: *Universal Verification Methodology*), que dispõe de mecanismos de automação de dados e geração automática de sequências de testes [15]. Por esta razão, a metodologia UVM será empregada como metodologia de verificação funcional nos ambientes de validação que serão propostos nos capítulos seguintes.

Figura 2.5: Refatoração de *testbenches* com a metodologia VeriSC



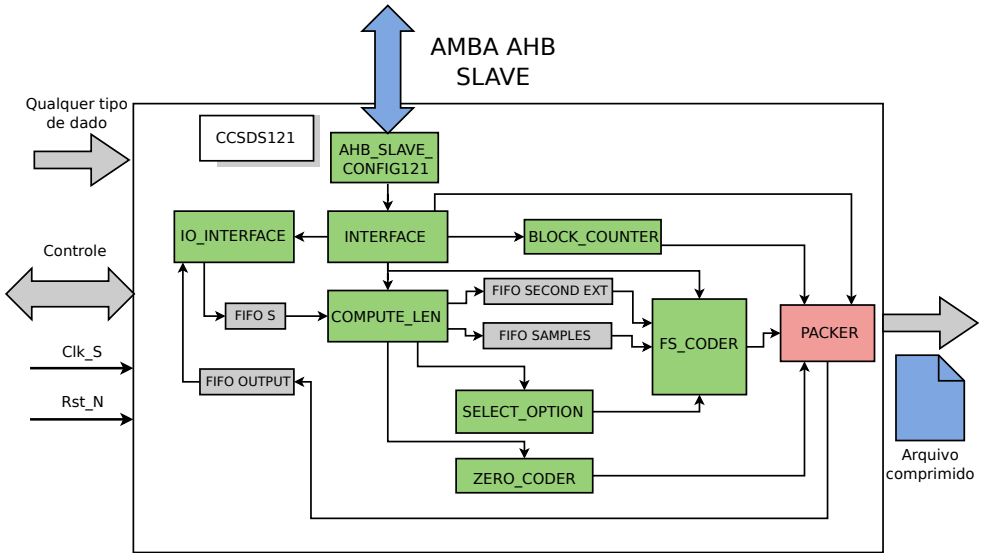
Fonte: Adaptada de [14]

Em [16] é realizada a modelagem em nível de sistema eletrônico e a verificação funcional

utilizando SystemC de dois algoritmos de compressão sem perdas com aplicações espaciais. Ambos os algoritmos foram especificamente projetados para operar a bordo de satélites e eles podem ser utilizados como compressores autônomos independentes, bem como em conjunto. Os modelos também são descritos utilizando modelagem em nível de transações com SystemC/TLM (como exemplo, ver Figura 2.6).

Embora SystemC ainda seja bem utilizado na verificação funcional de sistemas digitais, as modernas metodologias de verificação funcional tem seus ambientes de validação compostos por uma combinação de SystemC com a metodologia UVM, utilizando a biblioteca UVM Connect [17], que também será aplicada nos ambientes de validação propostos neste trabalho de dissertação.

Figura 2.6: Modelo SystemC dos módulos CCSDS-121



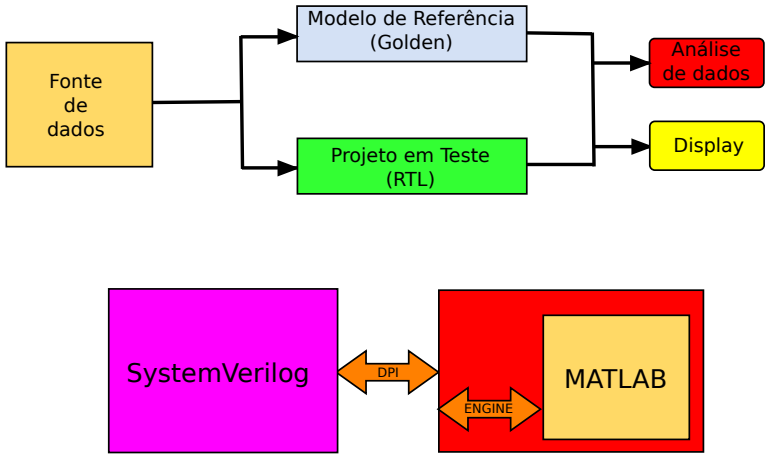
Fonte: Adaptada de [16]

2.2.2 Verificação funcional com SystemVerilog/MATLAB

Os métodos de verificação funcional que tem sido propostos na literatura objetivam atingir o máximo grau de confiabilidade em projeto de circuitos digitais. Os métodos tradicionais de verificação tem se mostrado insuficientes para a verificação de sistemas de processamento de imagens. Em [18] é proposta uma metodologia de verificação funcional que integra SystemVerilog com MATLAB para acelerar o processo de verificação. A integração é feita entre SystemVerilog e C via DPI, que é um mecanismo de SystemVerilog

para chamar funções nativas do C. O MATLAB utiliza um mecanismo para se integrar com o C, como pode ser visto na Figura 2.7. Esta integração permite utilizar a grande variedade de funções disponíveis no MATLAB para verificar funcionalmente módulos descritos em SystemVerilog.

Figura 2.7: Co-simulação entre SystemVerilog e MATLAB



Fonte: Adaptada de [18]

A integração entre o MATLAB com SystemVerilog permite incorporar o legado de funções matemáticas disponíveis em ambientes de verificação funcional. No entanto, a estrutura proposta em [18] não conta com os recursos de automação e randomização de dados, o que permitiria uma maior flexibidade ao gerar casos de testes durante o proceso de verificação funcional.

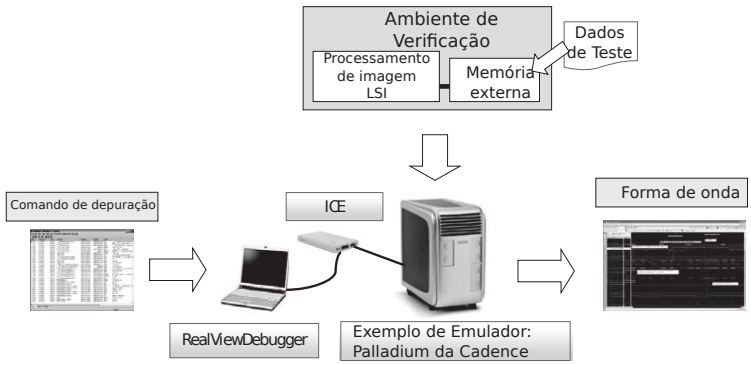
2.2.3 Verificação funcional com emulação em *hardware*

A Fujitsu Semiconductor tem utilizado em seus projetos de circuitos integrados com aplicações em processamento de imagens um emulador de *hardware* (um dispositivo capaz de mapear o circuito a ser verificado para *hardware* dedicado para executá-lo em alta velocidade) para estabelecer a tecnologia de verificação que garanta qualidade e melhora a eficiência de verificação ao mesmo tempo o aplicando para modelos de produtos. Em [19] é apresentada a co-verificação de *hardware*/*software*, verificação de desempenho e estimativa

de consumo de energia usando um emulador de *hardware* para verificar circuitos dedicados de processamento de imagem.

O ambiente de verificação proposto em [19] contém um emulador de *hardware* (ICE, do inglês: *In-circuit Emulation*) com um depurador conectado à interface JTAG, como pode ser visto na Figura 2.8.

Figura 2.8: Ambiente de verificação com emulador



Fonte: Adaptada de [19]

Todo o sistema foi integrado de forma eficiente em um ambiente de emulação fazendo uso das plataformas já validadas. Durante a verificação do sistema, pode ser obtida uma forma de onda detalhada da operação de *hardware* correspondente ao comportamento do *software*. Além disso, qualquer erro pode ser detectado inserindo um verificador de asserção. Além disso, usando um cabo dedicado para conectar a interface JTAG com o depurador ICE, um depurador de *software* semelhante à placa de avaliação real pode ser usado em um ambiente de verificação de emulador. Isso permitiu depurar o software em um ambiente semelhante ao *hardware* real mesmo antes do lançamento do dispositivo real.

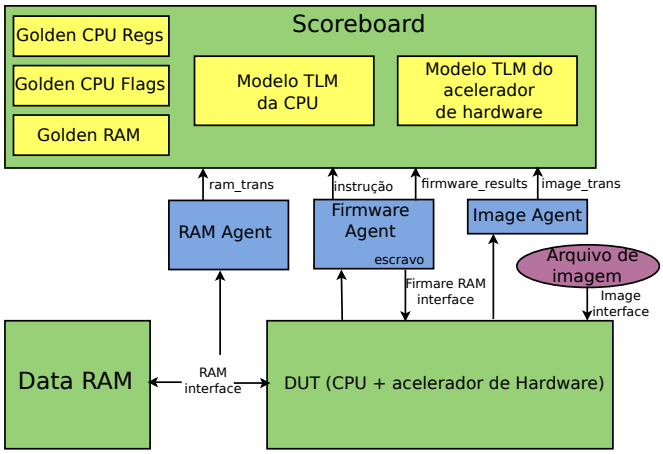
Embora os ambientes de validação utilizando emulação em *hardware* reduza o tempo de implementação dos testes, muitas vezes eles não são viáveis por terem custos elevados.

2.2.4 Verificação funcional com SystemVerilog/UVM

Em [20] é apresentado o projeto e a verificação de uma CPU de processamento de imagem com acelerador de *hardware* utilizando a metodologia UVM. A CPU e o acelerador tem por

objetivo final o projeto em ASIC. UVM foi escolhido como a metodologia de verificação porque é uma metodologia comprovada, com suporte para funções de randomização e testes de cobertura. O ambiente de verificação usado que atende aos requisitos acima é ilustrado na Figura 2.9. Além de verificar o acelerador em *hardware* da CPU, o *testbench* UVM em nível de blocos foi capaz de verificar a ferramenta de *assembler*.

Figura 2.9: Ambiente de verificação de uma CPU de processamento de imagens



Fonte: Adaptada de [20]

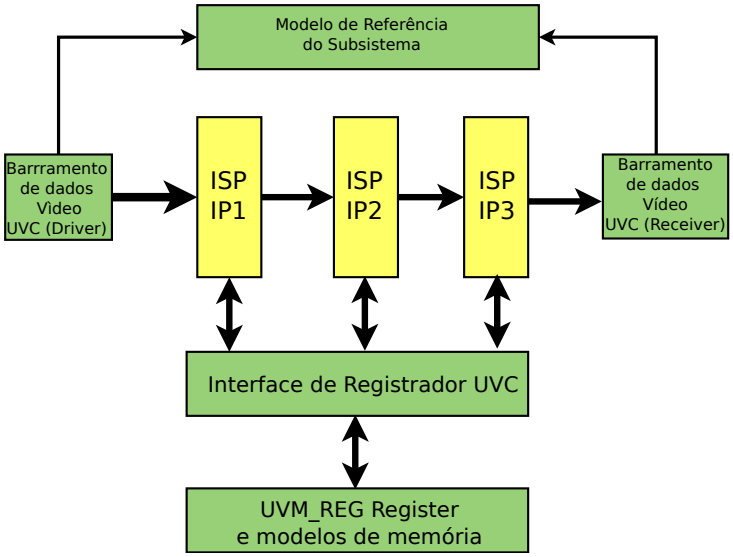
Em [21] é discutido a aceleração do desenvolvimento de componentes UVM a partir de componentes padrões utilizando SystemVerilog e seu uso em aplicações de processamento digital de imagens para aumentar o desempenho do tempo de execução do projeto. Os algoritmos de processamento de sinal de imagem são desenvolvidos e avaliados usando modelos descritos em Python antes da implementação do IP-Core[1] em nível RTL. Uma vez finalizado o algoritmo, os modelos em Python são usados como um modelo de referência para o desenvolvimento da descrição do *hardware*. Para maximizar a reutilização do esforço de projeto, os protocolos de barramento comuns são definidos para o registro interno e as transferências de dados. Uma combinação de tais módulos descritos em *hardware* de processamento de sinal de imagem configuráveis são integrados para satisfazer uma ampla gama de processos complexos de processamento de vídeo em sistemas de cirtuitos integrados.

No ambiente de verificação da Figura 2.10 existem alguns componentes UVM para os

[1]Um IP-Core é um módulo reutilizável em projetos de sistemas digitais descrito em qualquer nível de abstração, incluindo componentes de circuito.

barramentos de entrada e saída do bloco de processamento de vídeo, e modelos de registradores UVM_REG para a interface do DUT e modelos de memória. As fontes de vídeo que excitam as transações do DUT também excitam o modelo de referência descrito em Python. Esta estrutura permite a rápida verificação funcional de uma vasta gama de aplicações em processamento de imagens.

Figura 2.10: Blocos básicos do ambiente de verificação de IP utilizando SystemVerilog/UVM



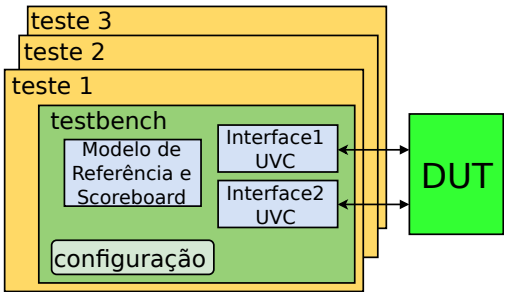
Fonte: Adaptada de [21]

Em [22] é proposto uma plataforma de verificação utilizando UVM para aplicações em física de alta energia. O alvo específico são os circuitos integrados de leitura de pixels da próxima geração para o Colisor de Hadrão de Alta Luminosidade no CERN. Os principais requisitos da plataforma são a geração flexível de estímulos de entrada (dados provenientes de simulações de detector completo externo ou simulação de sensores e gerados dentro da própria estrutura com distribuições aleatórias restritas determinadas que permitem aos projetistas testar casos extremos. Vários testes podem ser instanciados no *testbench* como mostrado na Figura 2.11.

A estrutura do *testbench* define interfaces genéricas para o dispositivo em teste (DUT) e fornece alta reutilização dos blocos de construção e configuração de cenários no teste, tendo UVM como metodologia padrão. A definição da organização da plataforma de verificação foi focada em alcançar a reutilização por vários projetistas de *hardware* e em diferentes

níveis de descrição do DUT.

Figura 2.11: Camadas hierárquicas de um banco de teste UVM: reutilização do mesmo *testbench* para diferentes testes



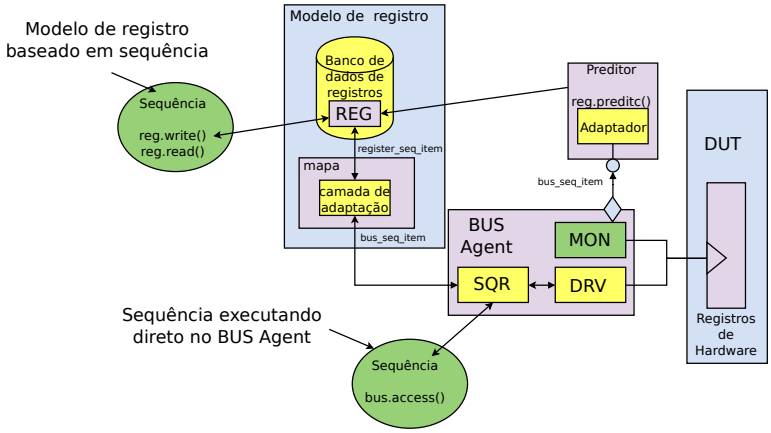
Fonte: Adaptada de [22]

A camada de abstração de registradores do UVM é uma grande ferramenta da metodologia. Essa camada tenta modelar no *testbench* os modelos de registradores que estão contidos no RTL. Sendo assim, ao estimular os modelos de registradores UVM_REG, os registradores do RTL serão excitados de acordo com esses estímulos. Embora úteis, esses modelos de registradores requerem experiência avançada em UVM por parte do engenheiro de verificação. Em [23] é proposta uma maneira de simplificar e automatizar o uso da biblioteca de registradores UVM_REG. No ambiente de verificação (ver Figira 2.12) proposto por [23], o registro de UVM_REG e o modelo de memória são usados para uma verificação eficiente do registro e da memória.

Os ambientes de verificação com o modelo de registro UVM_REG são utilizados para verificar uma variedade de dispositivos, por exemplo, aplicações em internet das coisas.

Em [24] são abordadas as opiniões dos especialistas em verificação funcional na indústria de semicondutores sobre as últimas técnicas de verificação. O artigo mostra as atividades de melhoria de verificação mais utilizadas que foram implementadas por empresas de semicondutores e seu impacto na qualidade, custo e tempo dos produtos. Experiências foram feitas para avaliar os novos métodos de verificação antes de usá-los em projetos reais. A referida pesquisa centra-se principalmente na identificação das lacunas entre a expectativa dos especialistas em verificação e suas experiências reais de técnicas de verificação mais recentes para o processo de verificação de produtos na indústria de semicondutores.

Figura 2.12: Ambiente de Verificação com Registros UVM Integrados



Fonte: Adaptada de [23]

De acordo com o painel de discussão de especialistas líderes da indústria (tanto usuários como vendedores) e da academia sobre o futuro da metodologia de verificação em 2014, apenas uma combinação das várias técnicas - asserções, verificadores, análise de cobertura e conexões inteligentes entre diferentes níveis de abstração permitirá que os usuários obtenham o nível de confiança suficientemente alto para fabricar seus produtos. Embora as abordagens sejam padronizadas, o maior problema é integrar as equipes de *software* e *hardware* e desenvolver o *hardware* com o *software* desde o início.

Parte do trabalho de pesquisa está envolvida em empresas que usaram ou planejavam usar os últimos métodos de verificação, como o ambiente de verificação baseado em UVM para verificação de seus produtos. O resultado ajuda a descobrir as lacunas entre suas expectativas e experiências atuais com esses métodos de verificação. Existem diferenças significativas nos processos e fatores de sucesso nos últimos fluxos de verificação avançados, em comparação com os fluxos de verificação tradicionais. As principais questões incluem conhecimentos técnicos, requisitos de infra-estrutura e métricas para medir o progresso da verificação. Estar ciente e planejado para resolver esses problemas ajudará os gerentes de engenharia a aproveitar ao máximo os ganhos de produtividade oferecidos pelas metodologias avançadas de verificação.

Na indústria de semicondutores, os algoritmos de processamento de imagem são desenvolvidos e avaliados usando modelos de *software* antes da implementação de RTL. Após a finalização do algoritmo, os modelos de *software* são usados como um modelo de

referência para o RTL do processador de sinal de imagem (ISP, do inglês: *Image Signal Processor*) e desenvolvimento de *firmware*. Em [25], é descrita uma estrutura de modelagem unificada e modular de algoritmos de processamento de imagem usados para diferentes aplicações, como desenvolvimento de algoritmos para ISP, implementação de modelos de referência para *hardware* (HW) e para a implementação do *firmware* (FW). Esta estrutura de modelagem e verificação funcional utilizando UVM é utilizada em aplicações de processamento de sinal de imagem em tempo real, incluindo celular, câmeras inteligentes e compressão de imagem. A principal motivação por trás desse trabalho é propor a melhor estrutura eficiente, reutilizável e automatizada para modelagem e verificação de projetos de processador de sinal de imagem (ISP). A estrutura proposta mostra melhores resultados e observa-se uma melhoria significativa no tempo de verificação do produto, no custo de verificação e na qualidade dos projetos.

O fluxo de desenvolvimento de processadores de sinais de imagens em *hardware* é ilustrado na Figura 2.13. Na etapa inicial, o modelo de referência é escrito em Python para a avaliação dos algoritmos de processamento de imagem. Nesta etapa não há um particionamento entre *hardware* e *software*. Depois que o algoritmo em Python é validado, o modelo de referência é utilizado pra validação do ISP RTL. Nessa etapa é realizado um particionamento entre *hardware* e *software*. A metodologia UVM é utilizada na etapa inicial da verificação do RTL. Depois de alguns testes direcionados, alguns cenários de simulação são executados com testes randômicos. A verificação do RTL é realizada por blocos e depois que cada bloco é verificado isoladamente, é realizada uma verificação do modelo topo, com os blocos integrados.

Conforme [26], com a crescente complexidade em projetos de sistemas digitais, a verificação mais rápida requer o desenvolvimento precoce do ambiente de verificação com vetores de teste mais amplos antes que o projeto de RTL esteja disponível. Em [26] é apresentada uma nova abordagem do desenvolvimento precoce do fluxo de verificação reutilizável, escrita em múltiplas linguagens de descrição de *hardware* e programação, abordando quatro atividades principais de verificação:

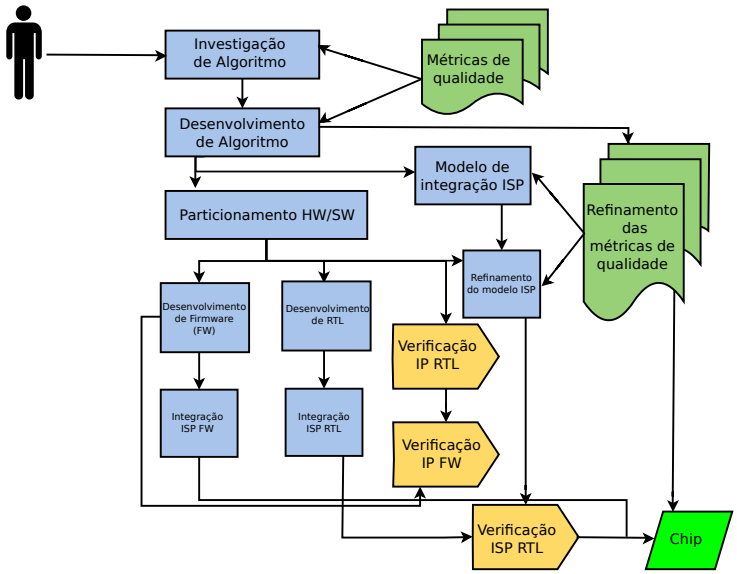
1. rápida concepção da aplicação de especificação

2. rápida concepção do ambiente de verificação

3. rápida concepção de vetores de teste

4. melhor e maior reutilização de blocos.

Figura 2.13: Metodologia de desenvolvimento: de ideia para produto



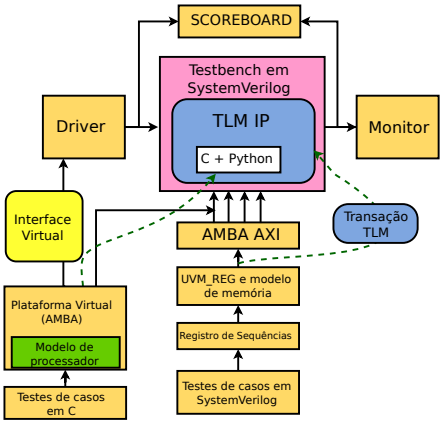
Fonte: Adaptada de [25]

O foco do trabalho proposto em [26] é o desenvolvimento inicial do ambiente de verificação de projetos, com aplicações de processamento de imagens, baseados em UVM utilizando modelos de referência em nível TLM que depois serão substituídos pelo modelo de RTL.

Geralmente, o desenvolvimento do ambiente de verificação para validação de projetos é iniciado após a disponibilidade do RTL, o que resulta em atraso no início e conclusão da verificação dos projetos. No ambiente proposto por [26] o modelo RTL pode ser substituído temporariamente por uma versão TLM como pode ser visto na Figura 2.14.

O uso do modelo de referência TLM como uma versão prévia do RTL antes de sua concepção nos estágios iniciais do projeto pode ser uma solução para a rápida concepção de ambientes de validação com aplicações em processamento de imagens. O modelo do ambiente proposto em [26] é integrado com Python, que fornece uma vasta biblioteca de

Figura 2.14: Desenvolvimento inicial do ambiente de verificação UVM com modelo TLM



Fonte: Adaptada de [26]

processamento de imagens. Um esforço adicional seria realizar a integração do ambiente com a biblioteca OpenCV, que fornece funções nativas para processamento de imagens e visão computacional.

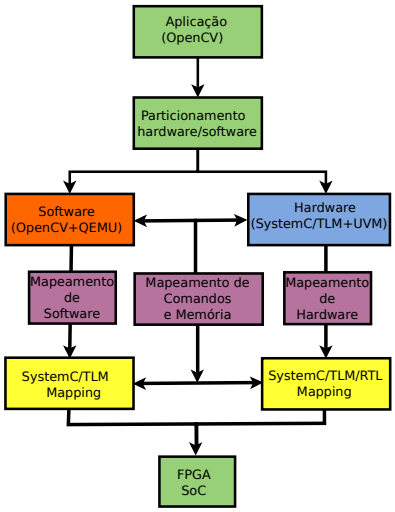
Em [27] é apresentada uma estrutura para prototipagem rápida de aplicativos de processamento de vídeo em sistemas embarcados. Com base em uma especificação de alto nível escrita em OpenCV, são aplicados refinamentos semi-automáticos na especificação em vários níveis (TLM e RTL), onde menor nível de abstração é o protótipo do sistema em uma plataforma FPGA. O refinamento aproveita a estrutura de aplicações de processamento de imagem da biblioteca OpenCV para mapear representações de alto nível em implementações de menor abstração. A estrututra proposta integra a biblioteca de visão por computador OpenCV para *software*, SystemC/TLM para representação de *hardware* de alto nível, UVM para verificação funcional e QEMU-OS para prototipagem virtual.

A Figura 2.15 ilustra o fluxo de projeto da plataforma proposta em [27]. O fluxo proposto começa com a especificação do sistema, onde a aplicação é descrita em C/C++ dentro de um ambiente OpenCV. Uma vez que a etapa de especificação é concluída, o SystemC é usado para modelar o sistema por meio de modelagem de nível de transação. O próximo passo é refinar as descrições abstratas do nível de sistema eletrônico em uma estrutura final que pode ser traduzida para um modelo de *hardware* sintetizável no nível de RTL.

Embora a simulação seja bem suportada no SystemC, a implementação de cobertura e

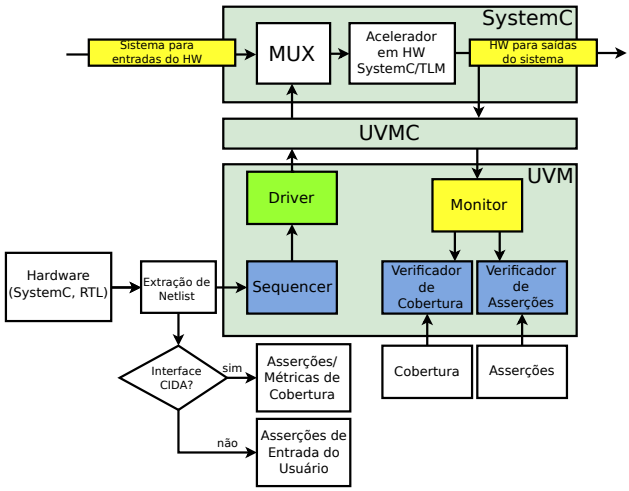
asserções no SystemC é um processo propenso a erros devido aos recursos de asserções limitadas da biblioteca de classes. Esta limitação pode ser superada usando uma extensão SystemC como SCV (do inglês: *SystemC Verification*) ou integrando um ambiente de verificação de função independente, como o UVM. Uma combinação de SystemC e UVM proporcionaria uma estrutura sólida para especificações de projeto eficientes com uma verificação perfeita e eficiente. A Figura 2.16 ilustra o ambiente de validação proposto por [27], onde a integração de SystemC com UVM é feita com a biblioteca UVM Connect.

Figura 2.15: Fluxo de projeto proposto em [27]



Fonte: Adaptada de [27]

Figura 2.16: Ambiente de validação proposto em [27]



Fonte: Adaptada de [27]

O ambiente de projeto e validação rápida proposto em [27] é flexível, importando funções do OpenCV nativas para aplicações de processamento de imagens e visão computacional. No entanto, métricas de avaliação de similaridades entre objetos devem ser implementadas na estruturas propostas por [27] e [26] para avaliar a comparação de diferentes algoritmos de processamento de imagens tendo como modelo verdadeiro uma aplicação de alto nível já validada.

A combinação dos trabalhos relacionados revisados neste capítulo foram essenciais como ponto de partida para o escopo desta dissertação, seja fornecendo os fundamentos de classificação de imagens ou os conceitos de verificação funcional. No entanto, nenhum trabalho fornecia em sua completude uma base para a construção de uma plataforma de projeto e validação para aplicações de processamento de imagens, em particular a detecção de faces.

2.3 Conclusões

Neste capítulo foi apresentada uma revisão da literatura com relação aos tópicos de métricas de classificação de imagens e também verificação funcional de sistemas digitais, especialmente com foco em aplicações de processamento de imagens. Esses tópicos serão aprofundados nos capítulos seguintes.

Capítulo 3

Fundamentação teórica

Este capítulo engloba um conjunto de tópicos com o intuito de consolidar a fundamentação da pesquisa com os seguintes temas: a seção 3.1 aborda os conceitos de processamento digital de imagens e a seção 3.2 trata sobre visão computacional enquanto que métricas de classificação de imagens serão discutidas na seção 3.3. Na seção 3.4 serão abordadas algumas técnicas de deteção de faces populares na literatura. Na seção 3.5 fornece uma base introdutória para o fluxo de projeto e verificação de sistemas digitais. No capítulo anterior, foi introduzido o contexto geral deste trabalho que será aprofundado neste e nos capítulos seguintes.

3.1 Processamento digital de imagens

Antes de ressaltar o processamento digital de imagens, alguns conceitos devem ser definidos. Uma imagem monocromática digital não codificada é uma captura instantânea de uma cena do mundo real, que matematicamente pode ser representada por uma matriz bidimensional, cujos elementos $f(x, y)$ representam as intensidades discretizadas da luminância, onde $0 \leq x \leq M$ e $0 \leq y \leq N$ são, respectivamente, as coordenadas espaciais da largura e altura do valor de f em qualquer posição da matriz. A intensidade de luz pode ser modelada de acordo com a Equação 3.1, onde $i(x, y)$ é a iluminação do ambiente $(0 \leq i(x, y) \leq \infty)$ e $r(x, y)$ é a refletância dos objetos $(0 \leq r(x, y) \leq 1)$. Uma imagem digital colorida é uma imagem que contém uma função de intensidade de luz em três

frequências, correspondentes às cores vermelho, verde e azul [28].

$$f(x,y) = i(x,y)r(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \dots & f(0,N) \\ f(1,0) & f(1,1) & \dots & f(1,N) \\ \vdots & \vdots & \vdots & \vdots \\ f(M,0) & f(M,1) & \dots & f(M,N) \end{bmatrix} \tag{3.1}$$

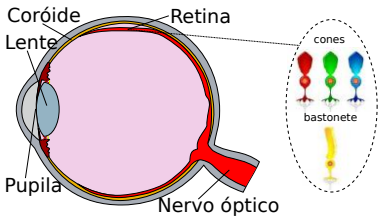
3.1.1 Espaços de cores RGB e YC_bC_r

Uma imagem colorida pode ser representada em diferentes espaços de cores, dentre os quais, serão destacados neste trabalho os modelos de espaços de cores digitais RGB e YC_bC_r.

O espaço RGB é baseado no sistema visual humano, sendo composto por três cores aditivas primárias: vermelho (R), verde (G) e azul (B), com comprimentos de ondas de 580 nm, 540 nm e 450 nm, respectivamente.

O sistema visual humano (ver Figura 3.1) percebe este espectro de frequência luminosa como uma variação suave de cores, como pode ser visto na Figura 3.2. O olho humano contém uma lente que focaliza uma imagem da natureza. A pupila, cujo diâmetro é variável, ajusta a quantidade de luz admitida. A luz é captada da natureza por células fotossensíveis constituídas de fibras nervosas que formam a retina.

Figura 3.1: Representação simplificada do olho humano

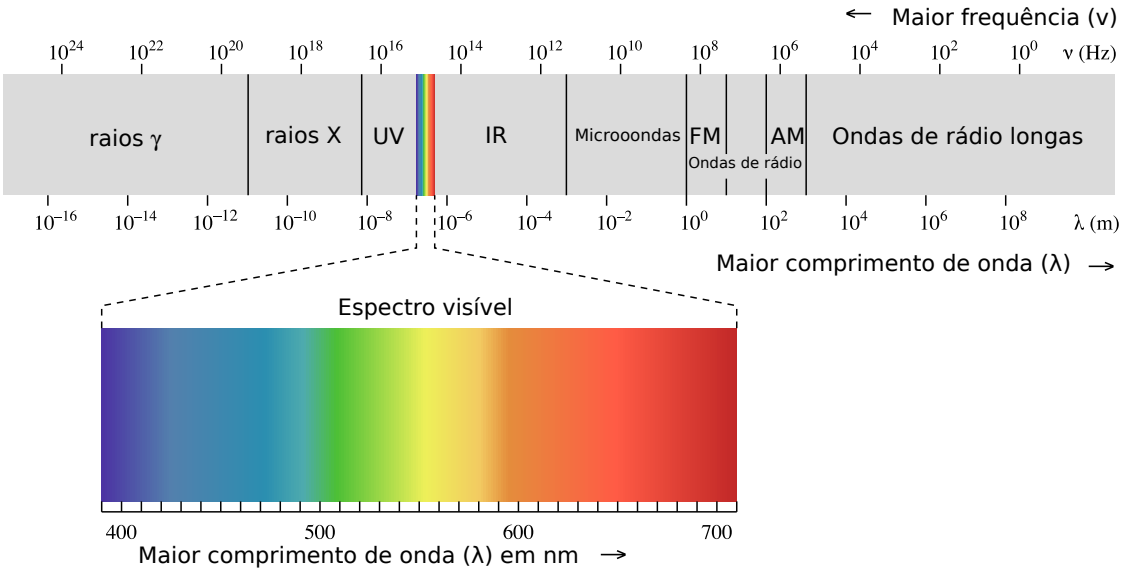


Fonte: Elaborada pelo autor

Existem dois tipos de células sensoras de luz na retina: os bastonetes (cerca de 120 milhões), que reagem apenas à variação de luminância; e os cones (cerca de 6 milhões), que são sensíveis não apenas à variação de luz, mas também às cores vermelha, verde e azul.

O espaço YC_bC_r tem aplicações em compressão de imagens, sendo composto pela luminância (Y) e pelas diferenças entre crominâncias azul (C_b) e vermelha (C_r).

Figura 3.2: Espectro visível



Fonte: Elaborada pelo autor

Como o número de bastonetes na retina é maior que o número de cones, o olho humano tem uma resolução espacial maior para variações de luz do que para variações de cor. Como consequência, a largura de banda para sinais de crominância é menor que a dos sinais de luminância, resultando em um menor número de amostras na resolução espacial das matrizes de cores em sistemas digitais [29].

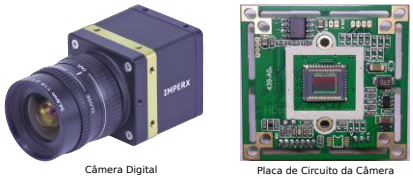
O espaço RGB codifica um píxel, que é o menor elemento de uma imagem digital, como uma palavra de 24-bits, composta por três componentes de 8-bits em cada canal. Para uma imagem monocromática, de 8-bits, os níveis de tonalidades da luminância variam de 0 (que representa a cor branca) à 255 (que define a cor preta). A conversão de um píxel do espaço RGB para o espaço YC_bC_r foi definida na norma da ITU-R BT.601 [30] de acordo com a Equação 3.2.

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \frac{1}{256} \begin{bmatrix} 65.738 & 129.057 & 25.064 \\ -37.945 & -74.494 & 112.439 \\ 112.439 & -94.154 & -18.285 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.2}$$

3.1.2 Aquisição e codificação de imagens

Antes de serem processadas, as imagens devem ser adquiridas e armazenadas em um computador. O processo de aquisição de uma imagem ocorre com a conversão de uma cena tridimensional em uma imagem bidimensional. A captura das imagens é realizada por uma matriz de sensores, que converte energia luminosa em energia elétrica. Um dispositivo amplamente utilizado nas câmeras digitais, como a que está ilustrada na Figura 3.3 é o sensor CCD (do inglês, *Charge Coupled Device*), mas o sensor mais utilizado hoje em dia é o sensor CMOS (do inglês, *Complementary Metal-Oxide-Semiconductor*) por ter um custo menor. Ambos são compostos por uma matriz de células semicondutoras fotossensíveis, produzindo um sinal elétrico proporcional à energia luminosa incidente.

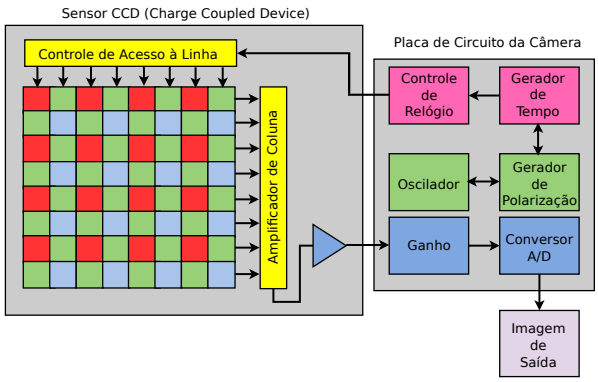
Figura 3.3: Câmera Digital com sensor CCD



Fonte: Elaborada pelo autor

Uma vez que a imagem é capturada pelo sensor, um circuito eletrônico acoplado ao dispositivo controla a posição espacial e temporal, como pode ser visto na Figura 3.4. Depois de amplificado, o sinal que já foi amostrado espacialmente nas células semicondutoras é amostrado no tempo e quantizado em amplitude.

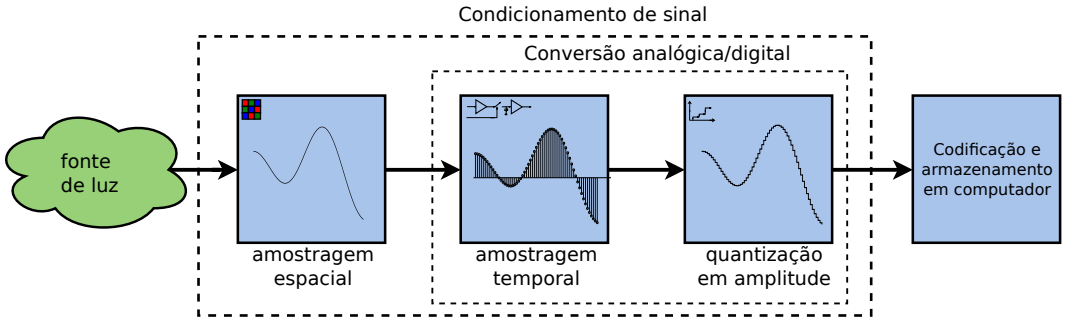
Figura 3.4: Um sensor CCD



Fonte: Elaborada pelo autor

Conforme a Equação 3.1, a resolução da imagem depende do número de elementos do sensor ($M \times N$). A quantização limita o valor da intensidade de cada píxel entre 0 e $2^q - 1$, onde 2^q é a profundidade de cor para uma imagem monocromática ou é o número de níveis de intensidade de cada componente RGB de uma imagem colorida. Depois de amostrada e quantizada, a imagem é digitalizada por um processo de conversão analógico digital. Uma vez digitalizada, a imagem pode ser codificada e armazenada em um computador. Os processos de digitalização de uma imagem podem ser vistos na Figura 3.5.

Figura 3.5: Processo de digitalização de uma imagem



Fonte: Elaborada pelo autor

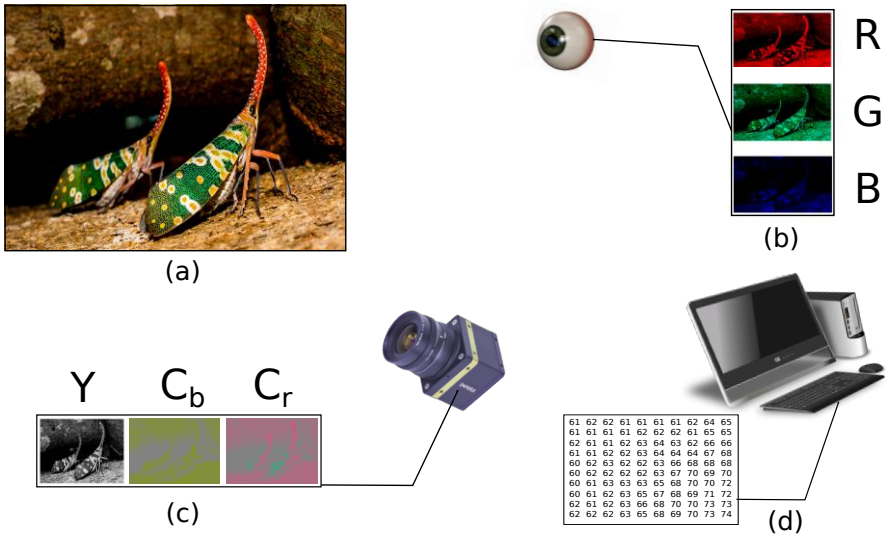
3.2 Visão computacional

A visão computacional[1] é a ciência cujo objetivo é emular a visão humana em máquinas. A imagem da Figura 3.6.(a) tem diferentes representações. O sistema visual humano enxerga esta imagem como a reflexão de ondas eletromagnéticas que são captadas pelas células fotossensíveis da retina. Conforme foi dito anteriormente, estas células são sensíveis ao espectro das cores vermelho, verde e azul. Assim, o olho humano decodifica a imagem em três componentes primárias, como pode ser visto na Figura 3.6.(b). Uma câmera fotográfica decodifica esta mesma imagem em três componentes YC_bC_r, como ilustrado na Figura 3.6.(c), pois neste espaço de cores a resolução espacial de crominância pode conter menos amostras, o que é favorável para a compressão e armazenamento dos dados. Por último,

[1]Para aprofundamento no assunto é recomendável a leitura de [31][32][33][34]. Para um aprendizado prático, consulte [35].

conforme a Figura 3.6, depois de amostrada e codificada, a imagem é vista pelo computador como uma grade de números que podem ser armazenados e processados.

Figura 3.6: Diferentes representações de uma imagem



Fonte: Elaborada pelo autor

3.2.1 Filtragem espacial

A grade de números da Figura 3.6.(d) é uma imagem digital codificada que pode ser manipulada e processada com operações matemáticas. Denotando esta grade como uma matriz $I(x, y)$, onde x e y são, respectivamente, os eixos horizontal e vertical da matriz, cujos elementos representam as intensidades das cores nos três canais da imagem, os contornos dessa imagem são os pontos onde a variação de intensidade dos pixéis é máxima. No domínio da frequência, estes contornos são obtidos filtrando-se as altas variações de luminosidade na distribuição espacial[2].

O módulo do gradiente da matriz $I(x, y)$ é dado por $|\nabla I(x, y)| = \sqrt{(\frac{\partial I}{\partial x})^2 + (\frac{\partial I}{\partial y})^2} \approx |\frac{\partial I}{\partial x}| + |\frac{\partial I}{\partial y}|$, que representa a taxa de variação da intensidade dos pixéis da matriz no espaço. Os termos $\frac{\partial I}{\partial x}$, e $\frac{\partial I}{\partial y}$ podem ser aproximados pelas expressões de diferença finita $\frac{\partial I}{\partial x} \approx I(x + 1, y) - I(x, y)$ e $\frac{\partial I}{\partial x} \approx I(x, y + 1) - I(x, y)$.

Outra aproximação pode ser baseada na diferença central dos pixéis, que pode ser vista

[2]Para aprofundamento no assunto é recomendável a leitura de [36].

como uma convolução da matriz $I(x, y)$ com dois núcleos de Sobel de acordo de acordo com as Equações 3.3 e 3.4, onde K_x e K_y são os núcleos do gradiente horizontal e vertical, respectivamente. Note que o núcleo do gradiente vertical pode ser obtido com uma rotação de 90^o da matriz K_x.

$$\frac{\partial I}{\partial x} \approx K_x * I(x,y) = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} K_x(i,j) I(x+i, y+j) \quad (3.3)$$

$$\frac{\partial I}{\partial y} \approx K_y * I(x,y) = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} K_y(i,j) I(x+i, y+j) \quad (3.4)$$

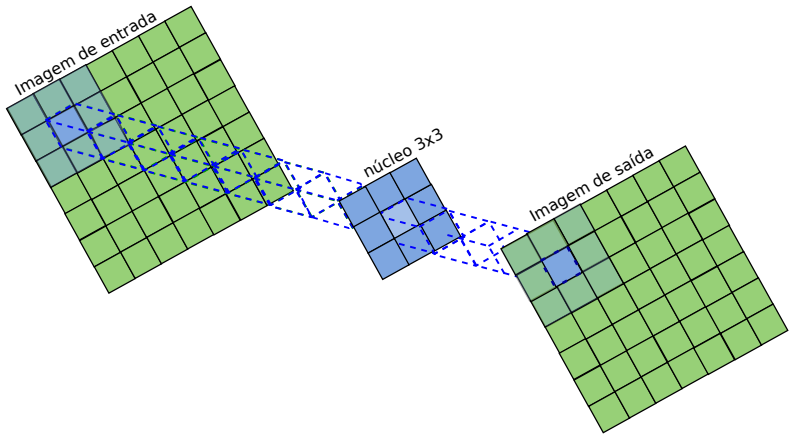
Algumas imagens contém picos nos contornos, e estas transições podem ser amenizadas com o laplaciano de $I(x, y)$, que pode ser aproximado por $\nabla^2 I(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \approx [I(x+1, y) - I(x, y)] - [I(x, y) - I(x-1, y)] + [I(x, y+1) - I(x, y)] - [I(x, y) - I(x, y-1)]$, que pode ser visto como a convolução de matrizes da Equação 3.5.

$$\nabla^2 I(x,y) \approx K * I(x,y) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} * I(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} K(i,j) I(x+i, y+j) \quad (3.5)$$

De maneira geral, a filtragem espacial de uma imagem é a convolução de um núcleo de ordem $n \times n$ com uma matriz de ordem $M \times N$, como poder ser visto na Figura 3.7.

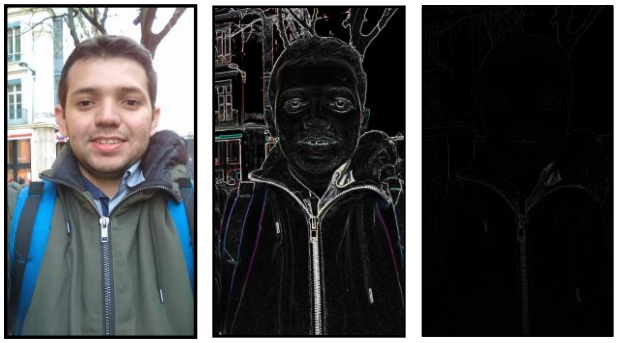
Para ilustrar o efeito dessas filtragens, foi aplicado um gradiente e um laplaciano na Figura 3.8.(a). Como pode ser visto na Figura 3.8.(b), o gradiente filtrou as altas frequências de variações de luminosidade, realçando apenas os contornos da imagem. Na Figura 3.8.(c) são destacados os picos de variações das bordas, que são detectados pelas derivadas de segunda ordem no plano espacial.

Figura 3.7: Convolução de um núcleo em uma imagem



Fonte: Elaborada pelo autor

Figura 3.8: Filtragem Espacial: gradiente e laplaciano



Fonte: Elaborada pelo autor

3.2.2 Integral da imagem

A *feature* (literalmente característica) é um conceito importante em inteligência artificial, processamento digital de imagens e visão computacional. No contexto de tratamento de imagens, uma *feature* pode conter informações sobre pontos, contornos ou áreas de regiões em uma imagem. Por exemplo, a área que delimita a região dos olhos é uma característica importante em aplicações de detecção e reconhecimento de objetos.

A integral da imagem é uma técnica que acelera o cálculo de características em imagens digitais. Esta técnica foi introduzida por [37] para acelerar o cálculo do mapeamento de texturas e ganhou notoriedade com a sua aplicação na estrutura de detecção de faces usando

o algoritmo de Viola Jones proposto por [8].

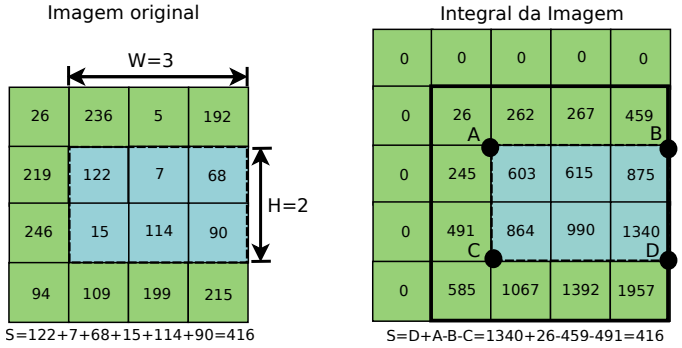
De acordo com a Equação 3.6, a integral da imagem $II(x, y)$ de uma imagem $I(x, y)$ no ponto (x, y) é a soma de todos os pixéis acima e à esquerda de (x, y), inclusive. A Equação 3.6 tem uma forma recursiva de acordo com a Equação 3.7.

$$II(x, y) = \sum_{x' \leq x, y' \leq y} I(x, y) \tag{3.6}$$

$$II(x, y) = I(x, y) + II(x - 1, y) + II(x, y - 1) - II(x - 1, y - 1) \tag{3.7}$$

Depois da imagem integrada, o cálculo da soma acumulada dos pixéis contidos numa região de área WH é computado com o acesso à quatro posições da região delimitada pelos pontos A, B, C e D, localizados na imagem integrada, conforme ilustra a Figura 3.9. Caso a soma fosse feita na imagem original, seriam efetuadas WH somas e a integração da imagem produz uma aceleração no cálculo da área com um fator de $\frac{WH}{4}$, para $WH \geq 4$, o que justifica sua aplicação em detecção de faces na estrutura proposta em [8].

Figura 3.9: Integral da Imagem



Fonte: Elaborada pelo autor

3.3 Métricas de classificação de imagens

De acordo com [38], o problema da comparação de algoritmos de processamento de imagens é muitas vezes subjetiva. No referido trabalho é descrito um sistema que fornece uma interface para geração de métricas para comparação de algoritmos de processamento de

imagens. A avaliação consiste na checagem de correspondência entre dois conjuntos: **alvos** e **candidatos**[3].

Para um dado par de elementos (um do conjunto alvo A e um do conjunto candidato C), a estrutura de avaliação proposta em [38] define algumas métricas de avaliação, dentre as quais:

- **coeficiente de sobreposição**: dados dois conjuntos A e C, o coeficiente de sobreposição OC é definido como $OC = \frac{|A \cap C|}{|A|}$ é uma medida assimétrica que garante apenas que um alvo é detectado.

- **coeficiente de *dice***: o coeficiente de *dice* DC é definido de acordo com a Equação 3.8 , é uma medida de similaridade entre os elementos dos conjuntos A e C sendo normalizado no intervalo $[0, 1]$.

$$DC = 2\frac{|A \cap C|}{|A| + |C|} \tag{3.8}$$

3.4 Métodos de detecção de faces

A face humana é um objeto dinâmico e tem um certo grau de variabilidade na sua aparência. Uma grande variedade de técnicas tem sido propostas para detecção de faces, desde algoritmos simples, baseados em bordas, até algoritmos que são compostos por avançadas técnicas de reconhecimentos de padrões [39].

Dentre as dificuldades em se detectar uma face numa imagem digital pode-se citar:

- **pose**: a posição da face pode variar em relação à câmera e isto pode obstruir alguns componentes do rosto, como por exemplo, olhos ou nariz

- **presença de elementos**: a presença de barba, óculos ou bigode pode atrapalhar a detecção do rosto, modificando sua cor, formato e estrutura

- **expressão facial**: as expressões faciais podem alterar a aparência do rosto

[3]Imagine a situação de comparar dois conjuntos de regiões que podem ou não conter faces. Os alvos são os objetos que definem regiões de faces na imagem. Os candidatos são objetos que podem ou não ser regiões que delimitam faces.

- **oclusão**: as faces podem estar parcialmente obstruídas por outros objetos
- **condições da imagem**: fatores como iluminação e qualidade da câmera afetam a aparência do rosto.

Serão abordados a seguir alguns métodos propostos na literatura que tratam sobre a detecção de faces.

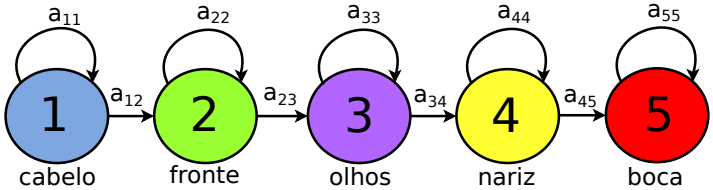
3.4.1 Modelos ocultos de Markov

Um modelo oculto de Markov (HMM, do inglês: *Hidden Markov Model*) é um modelo estatístico utilizado para caracterizar propriedades estatísticas de um sinal [40].

Um modelo HMM é constituído por um conjunto de estados $S = \{S_1, S_2, ..., S_N\}$. O estado do modelo no tempo t é dado por $q_t \in S$, para $1 \leq t \leq T$, onde T é o comprimento da sequência de observação de estados. De maneira compactar, um modelo HMM é definido como um vetor $\lambda = (A, B, \Pi)$, onde:

- $\Pi = \{\pi_i\}$ é a distribuição inicial do estado, onde $\pi_i = P[q_1 = S_i]$, $1 \leq i \leq N$
- $A = \{a_{ij}\}$ é a matriz de probabilidade de transição de estados, com $a_{ij} = P[q_t = S_j | q_{t-1} = S_i]$, $i \leq i$, $j \leq N$; com a restrição de $0 \leq a_{ij} \leq 1$ e $\sum_{j=1}^{N} a_{ij} = 1$, $1 \leq i \leq N$
- $B = \{b_j(O_t)\}$ é a matriz de probabilidade de estados obtidos por observação, com $b_i(O_t) = \sum_{k=1}^{M} c_{ik} N(O_t, \mu_{ik}, U_{ik})$, $1 \leq i \leq N$, onde c_{ik} é o coeficiente de mistura da *k-ésima* mistura no estado i, $N(O_t, \mu_{ik}, U_{ik})$ é assumido como uma função de densidade de probabilidade Gaussiana com média μ_{ik} e com matriz de covariância U_{ik}.

Figura 3.10: Detecção de Faces com HMM



Fonte: Elaborada pelo autor

Em [41], um modelo HMM é aplicado na detecção e reconhecimento de faces em imagens monocromáticas. O modelo é composto por 5 estados (como pode ser visto na Figura 3.10) que definem as probabilidades de uma região da imagem ser uma das regiões da face frontal de cima pra baixo: cabelo, fronte, olhos, nariz ou boca.

Os vetores de observação de estados são coeficientes KLT (*Karhunen-Loève Transform*) de uma imagem, que são obtidos com a transformada (linear e reversível) de *Karhunen-Loève*[4]. A transformada KLT remove a redundância da informação pela descorrelação dos dados [42]. O modelo proposto em [41] obteve uma taxa de detecção de 90% utilizando uma base de dados de teste com 48 imagens de 16 pessoas diferentes.

3.4.2 Máquina de vetores de suporte

Uma máquina de vetores de suporte (SVM, do inglês: *Support Vector Machine*) é um modelo de aprendizagem de máquina que executa o reconhecimento de padrões entre duas classes de vetores de suporte (treinamento), que são separados em um hiperplano ótimo. Os vetores de treinamento $\mathbf{x_i} \in R^n$, pertecem à uma de duas classes de vetores $y_i \in \{-1, 1\}$, onde $i = 1, 2, ..., N$.

Assumindo que os dados são linearmente separáveis, o objetivo é maximizar a separação das duas classes de vetores de suporte no hiperplano f, que tem a forma da Equação 3.9, onde os coeficientes α_i e b são as soluções de um problema de programação quadrática [43].

$$f(\mathbf{x}) = \sum_{i=1}^{l} \alpha_i y_i \mathbf{x_i}.\mathbf{x} + b \tag{3.9}$$

A classificação de um novo vetor $\mathbf{x}$ é realizada calculando o sinal do lado direito da Equação 3.9. A distância d de $\mathbf{x}$ ao hiperplano f é dada pela Equação 3.10.

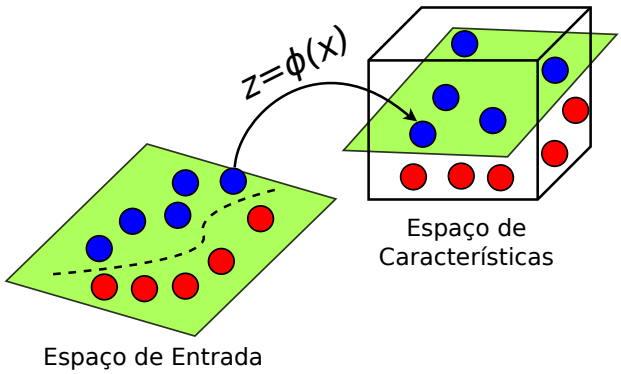
$$d(\mathbf{x}) = \frac{\sum_{i=1}^{l} \alpha_i y_i \mathbf{x_i}.\mathbf{x} + b}{\left\| \sum_{i=1}^{l} \alpha_i y_i \mathbf{x_i}.\mathbf{x} \right\|} \tag{3.10}$$

Cada ponto $\mathbf{x}$ no espaço de entrada é mapeado em um ponto $z = \phi(x)$ do espaço de características, como pode ser visto na Figura 3.11.

[4] A formalização desta transformada encontra-se no Apêndice D.

Em [44] são apresentados e comparados dois sistemas para detecção de faces frontais: um sistema de detecção de face inteira e um sistema de detecção baseado em componentes. Ambos os sistemas são treinados a partir de exemplos e usam modelos SVM como classificadores. O primeiro sistema detecta todo o padrão de rosto com um único modelo SVM. Em contraste, o sistema baseado em componentes executa a detecção por meio de uma hierarquia de dois níveis de classificadores. No primeiro nível, os classificadores de componentes detectaram de forma independente partes da face, como olhos, nariz e boca. No segundo nível, o classificador de configuração geométrica combina os resultados dos classificadores de componentes e executa o passo de detecção final.

Figura 3.11: Ilustração de uma máquina de vetores de suporte



Fonte: Elaborada pelo autor

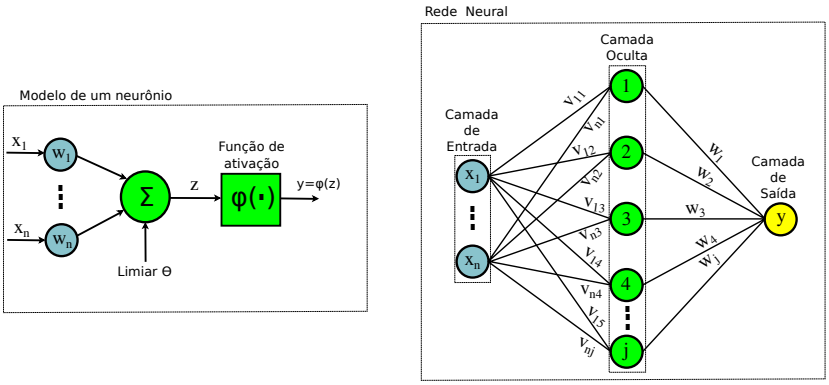
3.4.3 Redes neurais convolucionais

Uma rede neural artificial (RNA) é um modelo computacional inspirado no sistema nervoso humano que realiza uma determinada tarefa, por exemplo, aprendizado de máquina ou reconhecimento de padrões. A rede é composta por uma interligação de neurônios, que atribuem pesos numéricos w_i a um conjunto de entradas x_i, onde $1 \leq i \leq n$, para produzir uma saída y. No modelo clássico do neurônio proposto em [45] e ilustrado na Figura 3.12, a saída y é ativada com uma função de ativação $\varphi(z) = \frac{1}{1-e^{-z}}$, ilustrada na Figura 3.13, onde $z = \sum_{i=1}^{n} w_i x_i + \theta$ e θ é o limiar de ativação de φ.

As redes neurais são modelos de aprendizagem de máquina que tem como objetivo criar um algoritmo genérico para resolver diferentes aplicações. No entanto, estes modelos

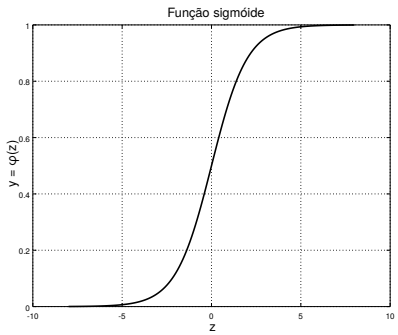
precisam ser treinados com um grande volume de dados. Considere o problema de detecção de faces, cujo modelo de face é uma imagem centrada e frontal, como pode ser visto na Figura 3.14.(a). Uma rede neural é treinada com uma grande amostra de faces e produz como saída a probabilidade de encontrar a região de uma nova face, que não pertence ao conjunto de treinamento.

Figura 3.12: Modelo de uma rede neural



Fonte: Elaborada pelo autor

Figura 3.13: Função de ativação de um neurônio

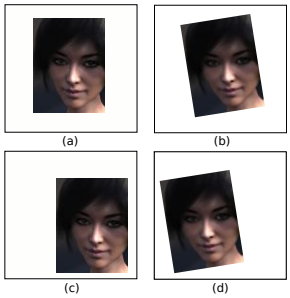


Fonte: Elaborada pelo autor

Se uma face sofre uma rotação, como pode ser visto na Figura 3.14.(b), ela não será detectada pela rede neural como uma face, pois este modelo de dado não foi inserido na entrada do sistema. Este problema pode ser resolvido aumentando a base de dados com faces com ângulos variados e aumentado o número de camadas ocultas na rede neural. Esta técnica de aprendizagem com redes neurais com profundas camadas ocultas é chamada de *deep learning*.

Outro problema que pode surgir é uma translação de faces, como as ilustradas nas Figuras 3.14.(c) e 3.14.(d), pois um algoritmo de aprendizagem trata uma mesma imagem situada em uma posição espacial diferente como outra classe de dados. Como consequência, as redes neurais tradicionais não distinguiriam as faces translacionadas. Uma nova classe de redes neurais surgiu para resolver este problema. As Redes Neurais Convolucionais[5] (CNN, do inglês: *Convolutional Neural Network*) tem sido amplamente aplicadas em reconhecimento de imagens, e embora concebidas na década de 60, só foram popularizadas recentemente, com o avanço das tecnologias de placas gráficas.

Figura 3.14: Faces em diferentes posições e ângulos



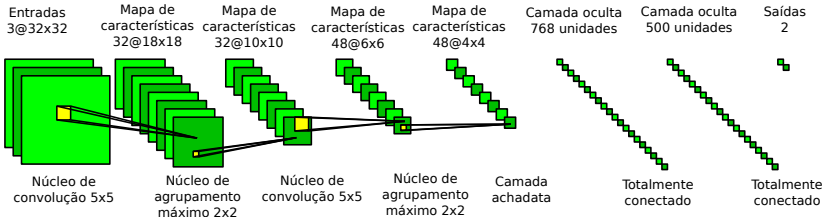
Fonte: Elaborada pelo autor

A arquitetura de uma rede neural convolucional[6] está ilustrada na Figura 3.15. Ao invés de alimentar a rede neural com a imagem da Figura 3.16.(a), a imagem é fragmentada em pequenos blocos que são sobrepostos de maneira semelhante a uma janela deslizante que percorre toda a imagem (ver Figura 3.16.(b)).

Uma rede neural recebe cada fragmento da imagem convoluída e descarta regiões que não são interessantes, armazenando as prováveis regiões que contenham uma face em um vetor. Este vetor que mapeia caracaterísticas de prováveis regiões com faces ainda contém um grande volume de dados. Um próximo passo do algoritmo seria subamostrar essas regiões com um núcleo de agrupamento máximo (o termo em inglês é o algoritmo de *max-pooling*), que coleta o fragmento que tem o maior número de características faciais, como pode ser visto na Figura 3.17. Outras etapas de processamento pode ser executadas em profundas camada da rede neural até que o algoritmo finalize a predição final.

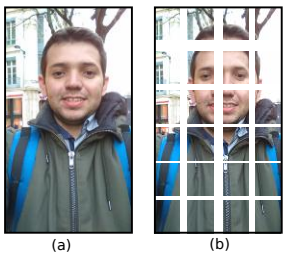
[5]Para maiores informações, consulte [46].
[6]Para maior aprofundamento, consulte [47].

Figura 3.15: Arquitetura de uma Rede Neural Convolucional



Fonte: Elaborada pelo autor

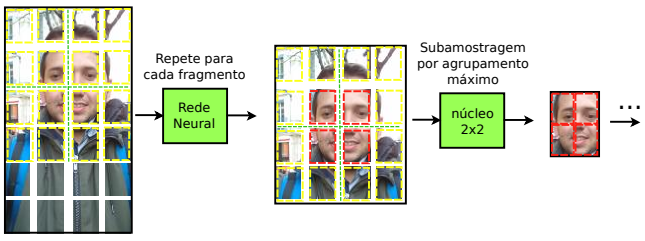
Figura 3.16: Imagem fragmentada em fragmentos sobrepostos



Fonte: Elaborada pelo autor

Em [48] é proposto um método de detecção de faces sem restrições, que não é influenciado por fatores como posição da face, expressão, postura, escala e condições de iluminação. Neste método, primeiramente é extraída a imagem de plano de fundo e depois é aplicada uma detecção de pele. Em seguida, é aplicada uma janela de deslizamento em várias escalas e a rede neural convolucional em cascata é aplicada para detectar faces. Nesse trabalho foram selecionadas 200 imagens aleatórias e a taxa de sensibilidade[7] foi de 92.34%.

Figura 3.17: Etapas da rede neural convolucional



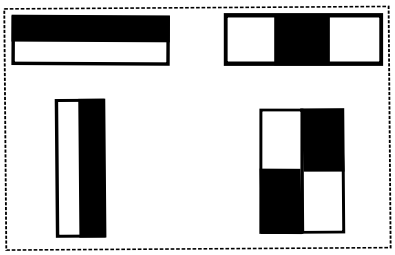
Fonte: Elaborada pelo autor

[7] A sensibilidade é calculada por $\frac{VP}{VP+FN}$, onde VP = n^o de verdadeiros positivos e FN = n^o de falsos negativos.

3.4.4 Estrutura de Viola-Jones

A estrutura de Viola-Jones é um algoritmo de detecção de faces que é a primeira técnica na literatura proposta para aplicações em tempo real [8]. O algoritmo classifica imagens com base no valor de características simples (ver Figura 3.18), como por exemplo, a característica de que a região dos olhos é mais escura que a região do nariz. Estas características são funções de Haar[8], utilizadas em [49] para detecção de objetos. O valor de uma característica é a diferença entre a soma dos pixéis em um retângulo preto e a soma dos pixéis em um retângulo branco. Estas áreas são calculadas de maneira eficiente com a integral da imagem, como já foi discutido na seção 3.2.2.

Figura 3.18: Algumas características simples



Fonte: Elaborada pelo autor

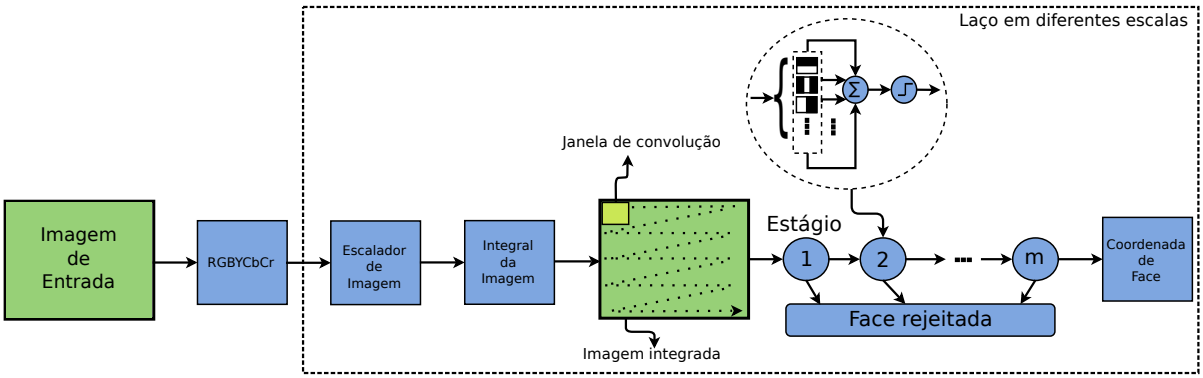
Dado que a resolução do algoritmo de Viola-Jones é de 24×24 pixéis, o número total de características que podem ser compostas em uma imagem desta resolução é de cerca de 160000. No entanto, a maior parte destas características é irrelevante, pois uma característica que define a região dos olhos não contém informação relevante na região da bochecha. Assim, o algoritmo de aprendizagem de *AdaBoost* é aplicado para reduzir o número destas características [50]. Estas características são classificadas por um classificador em cascata, que é uma combinação de classificadores fracos que descartam regiões que não são faces para concentrar o esforço computacional nas possíveis regiões que contenham uma face.

A técnica de Viola-Jones é baseada na exploração da imagem por meio de uma janela de busca de características. Esta janela é dimensionada para encontrar faces de diferentes tamanhos. A arquitetura da estrutura está ilustrada na Figura 3.19. Inicialmente, uma

[8] A definição da transformada de Haar encontra-se no Apêndice C.

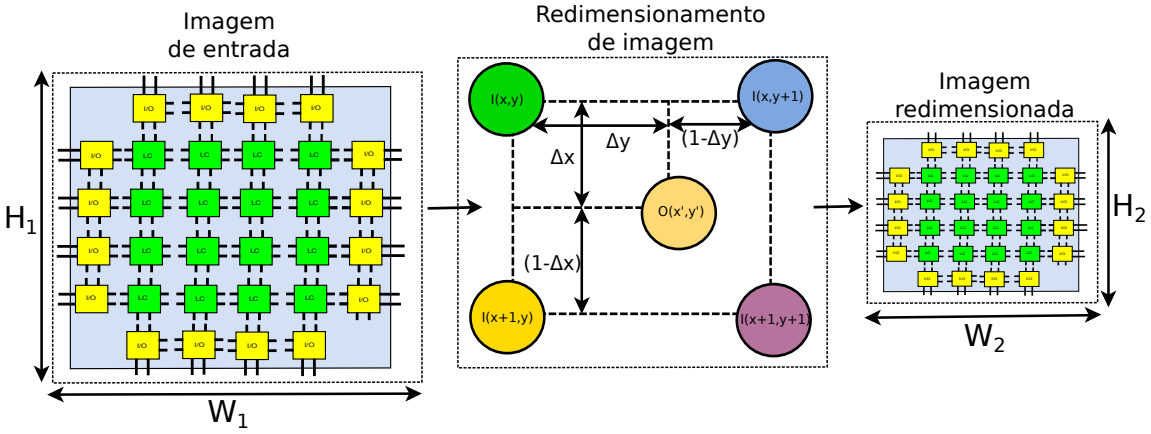
imagem de entrada sofre uma transformação de espaços de cores, sendo convertida em YC_bC_r. Em seguida, a luminância da imagem é redimensionada com uma interpolação bilinear[9] e depois integrada para rápido cálculo de características.

Figura 3.19: Arquitetura da Estrutura de Viola-Jones



Fonte: Elaborada pelo autor

Figura 3.20: Redimensionamento de Imagem por Interpolação bilinear



I(x,y): pixel de entrada na posição (x,y), 0≤x≤H1, 0≤y≤W1
O(x',y'): pixel de saída na posição (x',y'), 0≤x'≤H2, 0≤y'≤W2
$O(x',y') = I(x,y)(1-\Delta x)(1-\Delta y)+I(x,y+1)(1-\Delta x)\Delta y+I(x+1,y)\Delta x(1-\Delta y)+I(x+1,y+1)\Delta x\Delta y$

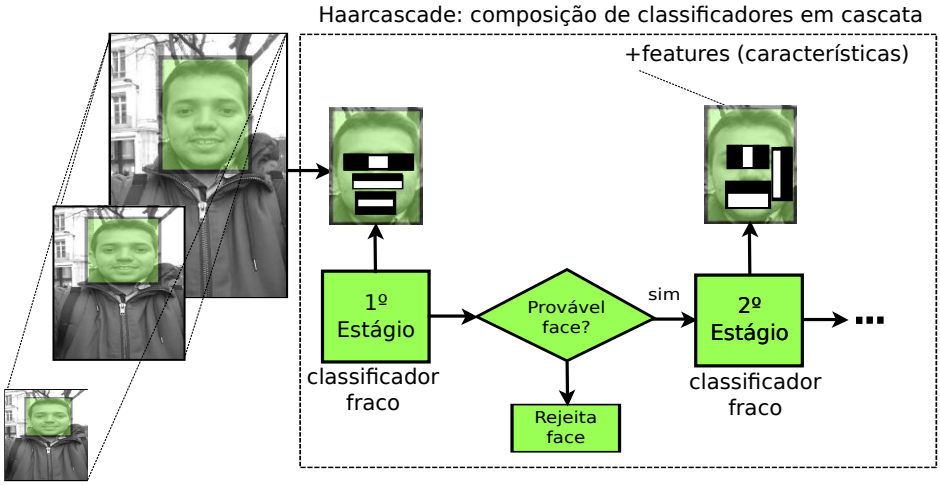
Fonte: Elaborada pelo autor

Os primeiros estágios do classificador consistem em classificadores fracos, que eliminam as regiões que não contém faces com pouco esforço computacional. Nos estágios sucessivos, a complexidade dos classificadores é aumentada para fazer uma análise mais detalhada das

[9] A Figura 3.20 ilustra como uma imagem pode ser redimensionada utilizando esta técnica.

características. Se um classificador falha em um determinado estágio, a região de busca é logo descartada. Uma face é detectada se ela passa em todos os classficadores da cascata, como pode ser visto na Figura 3.21.

Figura 3.21: Classificadores Haar em Cascata



Fonte: Elaborada pelo autor

O classificador seleciona uma característica f_i se o valor v_i desta característica for maior que um limiar L, de acordo com a Equação 3.11.

$$f_i = \begin{cases} +1 & v_i \geq L \\ -1 & v_i < L \end{cases} \tag{3.11}$$

O número de características que o classificador utiliza em cada classificador fraco pode ser definido na etapa de treinamento. Uma soma ponderada de classificadores fracos compõe um classificador forte, conforme a Equação 3.12 [51].

$$F = sign(w_1 f_1 + w_2 f_2 + ... + w_n f_n) \tag{3.12}$$

Na Figura 3.21 cada estágio da cascata contém um árvore de decisões treinada com características, obtidas a partir de uma base de dados que contém imagens de faces e de objetos que não são faces. Normalmente, os nós são organizados em ordem crescente de complexidade, de modo que características ruins são descartadas sem muito esforço, enfatizado o cálculo em características fortes, onde há maior probabilidade de se encontrar

uma face.

Durante o modo de execução do algoritmo de Viola Jones, uma janela de busca de diferentes tamanhos varre a imagem original. Na prática, de 70 a 80% de objetos que não são faces são rejeitados nos primeiros dois nós da cascata de rejeição [52]. Esta rapidez na rejeição de objetos indesejados acelera o processo de detecção de uma face.

O algoritmo de Viola-Jones da biblioteca OpenCV distribui 2135 características[10] (treinadas para classificar faces em posições frontais) em 22 estágios. A tabela 3.1 mostra o número de características de cada estágio . Estas características estão organizadas em um arquivo XML chamado haarcascade_frontalface_alt.xml. Outro arquivo que contém características de classificação de faces em posições frontais é o haarcascade_frontalface_default.xml, que distribui 2913 características em 25 estágios do classificador, como pode ser visto na tabela 3.2 [53].

Cada estágio é composto por um número variável de características e por um limiar de estágio, que é definido como sendo um número real fracionário. O sistema de detecção do OpenCV implementa a detecção de faces em dois modos:

- Modo 1: a detecção de faces é feita redimensionando a imagem de entrada. Neste modo, a imagem é reduzida até atingir uma dimensão mínima. A cada redimensionamento da imagem são realizados dois cálculos de integral da imagem (um é o somatório da intensidade dos pixéis e o outro é somatório do quadrado da intensidade dos pixéis) em cada etapa de redimensionamento, necessários para o cálculo da variância da imagem. A janela de busca tem dimensão constante em todo o processo.

- Modo 2: a deteção de faces é feita redimensionando os classificadores. Neste modo, os dois cálculos de integral da imagem necessários para o cômputo da variância é realizado uma única vez. A janela de busca (que contém as características) é redimensionada até atingir o tamanho original (24×24 pixéis).

Cada característica Haar do OpenCV é composta por:

- um limiar de características T, que é definido como sendo um número real fracionário

[10] Essas caractéristicas também conhecidas como *Haar-like features* são transformações dos pixéis de imagens digitais em características similares aos *wavelets* da Transformada de Haar.

- dois pesos de estágio P_1 e P_2, que são definidos como números reais fracionários
- duas ou três regiões de retângulos, que são ponderadas com números inteiros.

Tabela 3.1: haarcascade_frontalface_alt.xml: n^o de características por estágio

Estágio	n^o **de características**	**Estágio**	n^o **de características**
1	3	**12**	103
2	16	**13**	111
3	21	**14**	102
4	39	**15**	135
5	33	**16**	137
6	44	**17**	140
7	50	**18**	160
8	51	**19**	177
9	56	**20**	182
10	71	**21**	211
11	80	**22**	213

Tabela 3.2: haarcascade_frontalface_default.xml: n^o de características por estágio

Estágio	n^o **de características**	**Estágio**	n^o **de características**
1	9	**14**	135
2	16	**15**	136
3	27	**16**	137
4	32	**17**	159
5	52	**18**	155
6	53	**19**	169
7	62	**20**	196
8	72	**21**	197
9	83	**22**	181
10	91	**23**	199
11	99	**24**	211
12	115	**25**	200
13	127	–	–

Em ambos os modos de detecção (modo 1 e modo 2) os componentes das características Haar (retângulos, pesos e limiares) são redimensionados com as dimensões da janela de busca. Para uma janela de busca com dimensão WH o peso P de cada retângulo é escalado para um novo peso P' de acordo com a Equação 3.13.

$$P' = \frac{P}{WH} \tag{3.13}$$

A soma das características F^s_{Haar} na janela de busca é dada pela Equação 3.14, onde A_i é a soma de todos os píxeis contidos no *i-ésimo* retângulo.

$$F^s_{Haar} = \sum_{i=1}^{3} A_i P'_i \tag{3.14}$$

Para determinar o próximo valor do peso da soma dos estágios P^s_{est}, o valor de $\sum F^{Haar}$ é comparado com um limiar T^{norm}, normalizado pela variância σ_{VJ} ($T^{norm} = T\sigma_{VJ}$) da imagem de acordo com a Equação 3.15, onde $P^{s'}_{est}$ é o valor de P^s_{est} na iteração anterior do algoritmo.

$$P^s_{est} = \begin{cases} P^{s'}_{est} + P_{2j}, & \text{se} \sum_j F^{Haar}_j \geq T^{norm} \\ P^{s'}_{est} + P_{1j}, & \text{se} \sum_j F^{Haar}_j < T^{norm} \end{cases} \tag{3.15}$$

Na Equação 3.15, $j = 1, 2, ..., 2135$ caso os classificadores estejam contidos no arquivo haarcascade_frontalface_alt.xml ou $j = 1, 2, ..., 2913$ para os classificadores contidos no arquivo haarcascade_frontalface_default.xml. O valor da variância σ_{VJ} , onde $p_x \subset x$ representa os pixéis contidos na imagem x, é calculado de acordo com Equação 3.16.

$$\sigma_{VJ} = \sqrt{\frac{\sum\limits_{p_x \subset x} x^2}{WH} - \left(\frac{\sum\limits_{p_x \subset x} x}{WH}\right)^2} \tag{3.16}$$

Aplicando-se a Equação 3.13 na Equação 3.15, a expressão de comparação $\sum_j F^{Haar}_j < T^{norm}$ é expandida de acordo com a Inequação 3.17.

$$\frac{\sum\limits_{i=1}^{3} A_i P_i}{WH} < T\sqrt{\frac{\sum\limits_{p_x \subset x} x^2}{WH} - \left(\frac{\sum\limits_{p_x \subset x} x}{WH}\right)^2} \tag{3.17}$$

De acordo com [52], multiplicando-se ambos os lados da Inequação 3.17 por WH, a Inequação 3.18 é obtida.

O cálculo de decisão do classificador é acelerado pela variância ajustada σ'_{VJ}, definida na Equação 3.19, que elimina operações de divisão em cada etapa de redimensionamento

da imagem.

$$\sum_{i=1}^{3} A_i P_i < T\sigma'_{VJ} \tag{3.18}$$

$$\sigma'_{VJ} = \sqrt{WH \sum_{p_x \subset x} x^2 - (\sum_{p_x \subset x} x)^2} \tag{3.19}$$

A Inequação 3.18 é da forma $a < b\sqrt{c}$ e a raiz quadrada[11] desta expressão pode ser eliminada com a relação $a < b\sqrt{c} \Rightarrow a|a| < b|b|c$, onde a, b e c são definidos na Equação 3.20.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} A_i P_i \\ T_j \\ WH \sum_{p_x \subset x} x^2 - \left(\sum_{p_x \subset x} x \right)^2 \end{bmatrix} \tag{3.20}$$

O algoritmo de Viola-Jones implementado em [8] foi avaliado com uma base de dados de faces frontais do MIT+CMU, que consistem de 130 imagens com 507 faces frontais marcadas [55].

Tabela 3.3: Taxas de detecção para vários números de falsos positivos na base de dados MIT+CMU.

Detector	Falsos postivos							
	10	31	50	65	78	95	167	422
	Taxa de acerto							
Viola-Jones	76.1%	88.4%	91.4%	92.0%	92.1%	92.9%	93.9%	94.1%
Rowley-Baluja-Kanade	83.2%	86.0%	–	–	–	89.2%	90.1%	89.9%

Os experimentos foram realizados em função do número de falsos positivos de cada sistema. Por exemplo, para uma taxa de 10 falsos positivos o algoritmo de Viola Jones teve uma taxa de acerto[12] de $76,1\%$ (o que dá um número de $507 \times 0.761 \approx 385$ faces detectadas) enquanto que o sistema proposto por Rowley-Baluja-Kanade teve uma taxa de acerto de

[11] Um algoritmo com implementação em *hardware* de raiz quadrada pode ser visto em [54].

[12] A taxa de acerto T_a é calculada de acordo com a expressão $T_a = \frac{N_D}{N_T}$, onde N_D é o número de faces detectadas em um conjunto e N_T é o número de faces totais no conjunto.

83.2% (cerca de $507 \times 0.832 \approx 421$ faces detectadas) para a mesma taxa de falsos positivos. Os resultados da taxa de detecção e dos falsos postivos estão ilustrados na Tabela 3.3[13].

3.5 Projeto e verificação de sistemas digitais

O desenvolvimento de um circuito integrado digital passa por múltiplas transformações, que vão desde a especificação de projeto até a etapa de fabricação. Cada uma dessas transformações corresponde a uma diferente descrição do sistema com sua respectiva semântica. O nível de abstração de cada representação do sistema é gradualmente mais detalhado em cada etapa do fluxo de projeto.

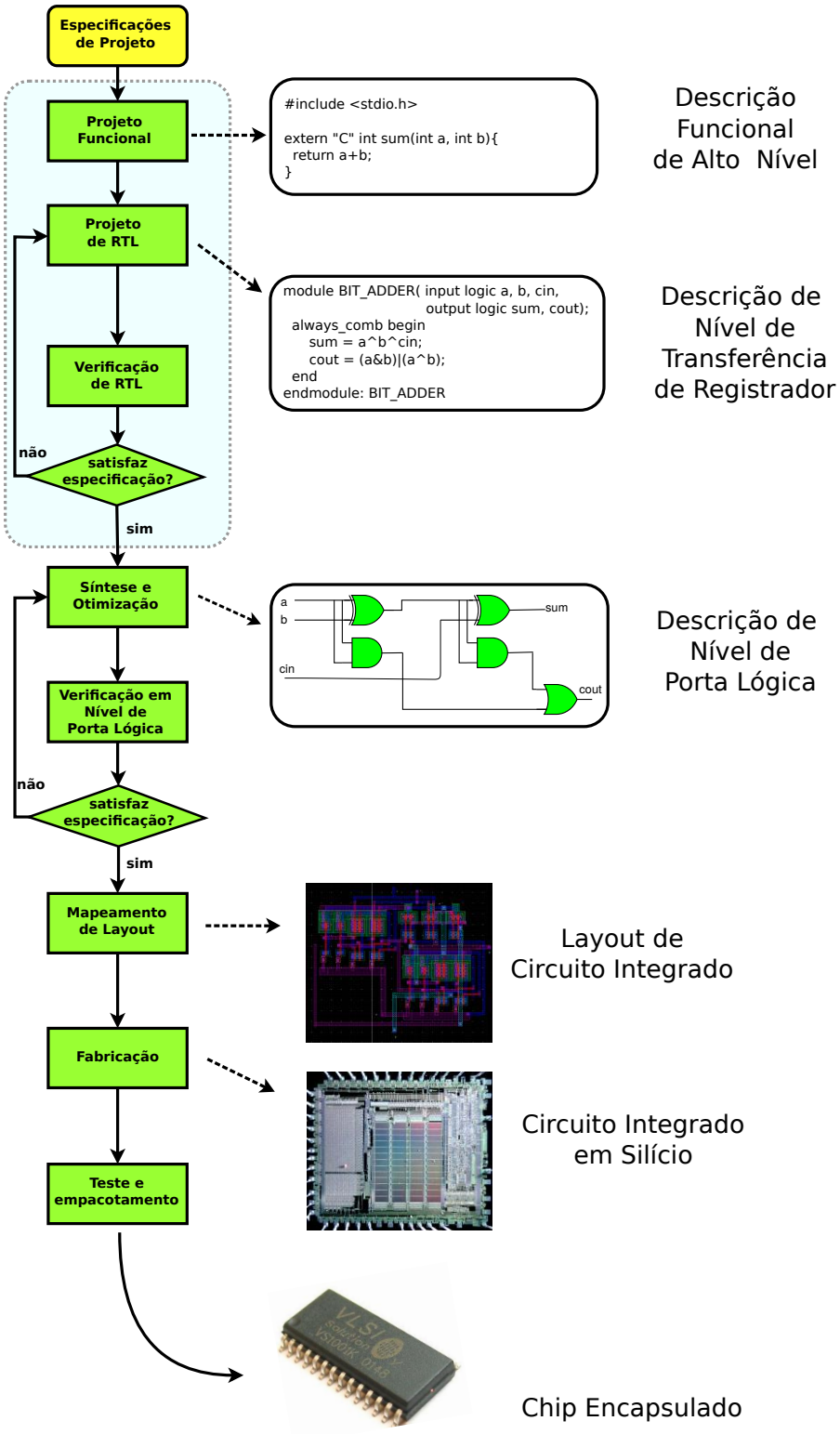
A Figura 3.22 ilustra o fluxo de projeto de sistemas digitais. As especificações de projeto geralmente são apresentadas como um documento que descreve um conjunto de funcionalidades e restrições que deverão ser satisfeitas pela solução final. Neste contexto, o projeto funcional é a modelagem do sistema a partir dessas especificações e requisitos de projeto.

Em sistemas de alta complexidade o desenvolvimento do projeto funcional geralmente é realizado sob uma abordagem hierárquica, de modo que o sistema é particionado em módulos que possuem interfaces para integração com outros componentes do projeto, o que possibilita que um único projetista da equipe de projeto possa se concentrar em uma porção do modelo em qualquer momento. O resultado desta fase de projeto é uma descrição funcional de alto nível, que geralmente é um modelo em *software* de alto nível que simula o comportamento do projeto com a precisão de um ciclo de relógio e reflete a partição do módulo. Este projeto funcional é utilizado para análise de desempenho algorítmico e também como modelo de referência para verificar o comportamento dos projetos mais detalhados que serão desenvolvidos nas próximas etapas.

A partir do modelo de projeto funcional, a equipe de projeto prossegue para a fase de projeto em Nível de Transferência de Registradores (RTL, do inglês: *Register Transfer Level*). Durante esta fase, a descrição arquitetural é refinada: elementos de memória e componentes funcionais de cada modelo são projetados utilizando Linguagens de Descrição de *Hardware* (HDL, do inglês: *Hardware Description Language*).

[13] A tabela completa está disponível em [8].

Figura 3.22: Fluxo de Projeto e Validação de Sistemas Digitais



Fonte: Elaborada pelo autor

Durante a fase de projeto de RTL, a etapa de projeto funcional é encerrada e o processo de verificação funcional se inicia. A verificação do RTL consiste em adquirir uma confiança razoável de que um circuito funcionará corretamente sob o pressuposto de que nenhuma falha de fabricação estará presente. A motivação subjacente é remover todos os possíveis erros de projeto antes da fabricação dos circuitos integrados digitais, que é um processo caro.

A equipe de verificação desenvolve várias técnicas e várias séries de testes para verificar se o comportamento do projeto corresponde às especificações. Cada vez que são encontrados erros funcionais e assumindo que o modelo funcional atende às especificações iniciais, o projeto de RTL precisa ser modificado e atualizado para satisfazer o comportamento definido na especificação de projeto.

Todo o desenvolvimento do projeto até a concepção do RTL verificado tem suporte mínimo de ferramentas de *software* de projeto assistido por computador (CAD, do inglês: *Computer Aided Design*) e são quase inteiramente realizadas manualmente pela equipe de projeto e verificação. A partir da fase da síntese e otimização, a maioria das atividades é semi-automática, com suporte de ferramentas de CAD.

O modelo sintetizado precisa ser verificado. O objetivo da verificação do RTL em nível de porta lógica é garantir que nenhum erro tenha sido introduzido durante a fase de síntese. Esta verificação é uma atividade automática que requer uma interação humana mínima, que compara a descrição do RTL antes e depois da síntese, a fim de garantir a equivalência funcional dos dois modelos.

Na próxima etapa é possível proceder ao mapeamento, posicionamento e roteamento do circuito que são feitos por uma ferramenta CAD. O resultado do processo é uma descrição do circuito em termos de um leiaute geométrico usado para o processo de fabricação. Por último, o projeto é enviado para fabricação e os microchips são testados e embalados.

O fluxo de projeto apresentado na Figura 3.22 é ideal. Em fluxos mais realistas, por exemplo, geralmente há muitas iterações na etapa de síntese, sejam devidos às mudanças na especificação ou à descoberta de falhas durante a verificação de RTL, além de serem consideradas restrições em baixo consumo de energia no fluxo de projeto[14]. No escopo desse

[14]A metodologia UPF (do inglês: *Unified Power Format*) vem sendo utilizada em projetos de circuitos integrados com baixo consumo de energia. Para maiores informações, consulte [56] e [57].

trabalho, serão abordadas as etapas de projeto funcional e projeto e verificação de RTL.

3.5.1 Projeto funcional

O desenvolvimento de sistemas em chip (SoC, do inglês: *System on Chip*) é composto por aplicações em *hardware* e *software*, que são co-desenvolvidas em equipes particionadas. Na etapa de projeto funcional o sistema é modelado com um alto nível de abstração. O nome predominante para esta abordagem de concepção de sistemas complexos é o projeto em nível de sistema eletrônico (ESL, do inglês: *Electronic System Level*) [58].

A modelagem do sistema ESL é realizada em nível de transações (TLM, do inglês: *Transaction Level Modeling*). A modelagem TLM descreve o comportamento do sistema como um componente com entradas e saídas, omitindo detalhes de comunicações entre componentes, que poderão ser adicionados numa etapa posterior do fluxo de projeto. A transação (no contexto da metodologia UVM a transação é uma classe orientada a objetos) é a menor unidade de informação, com um alto nível de abstração, que trafega entre componentes e a comunicação destes é realizada por meio de portas e canais [59]. Os membros de uma transação também podem ser controlados por eventos com abstração temporal, como por exemplo, atrasos ou ciclos de relógio.

A seguir são apresentadas os conceitos de alguns termos, que serão definidos no contexto de TLM.

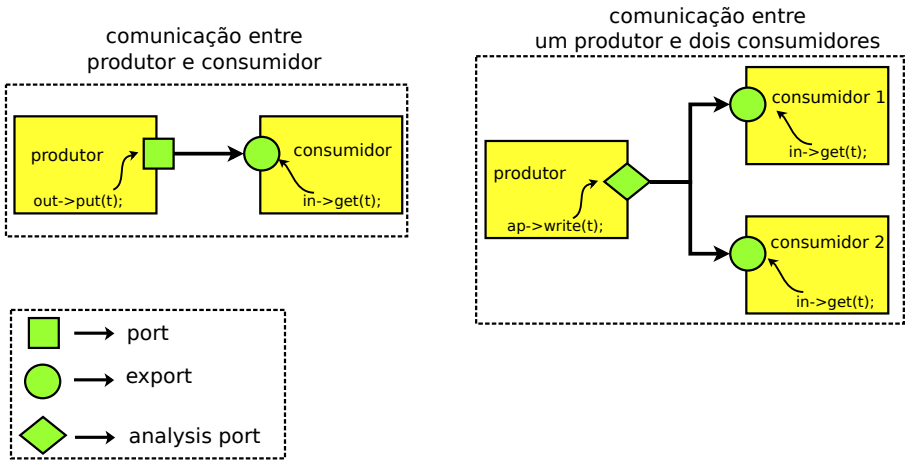
- **interface**: uma classe que define um ou mais protótipos de um método, mas não os implementa
- **port**: um objeto através do qual as chamadas de interface são feitas
- **export**: um objeto que exporta uma implementação de um método
- **produtor**: um componente que produz transações
- **consumidor**: um componente que consome ou executa transações.

A indústria vem utilizando SystemC na modelagem de projetos funcionais [60]. SystemC é um conjunto de classes em C++ que fornece estruturas para modelagem em nível ESL e

em nível arquitetural, incorporando alguns conceitos de linguagens de descrição de *hardware* como Verilog ou VHDL. A modelagem em TLM do SystemC envolve a comunicação entre processos SystemC usando chamadas de função.

A Figura 3.23 ilustra a comunicação entre componentes TLM. A comunicação entre um produtor e um consumidor é bem simples: o produtor produz a transação t e a TLM *port* chama o método *put()* para enviar a transação para o consumidor, que utiliza uma TLM *export* para exportar o método *get()* para receber a transação. Note que uma TLM *port* pode ser conectada a uma única TLM *export*. Quando existe a necessidade de comunicação entre um produtor e mais de um consumidor, uma TLM *analysis port*, que pode ser conectada à várias TLM *export*, pode ser aplicada na porta de saída do produtor.

Figura 3.23: Exemplos de comunicação TLM



Fonte: Elaborada pelo autor

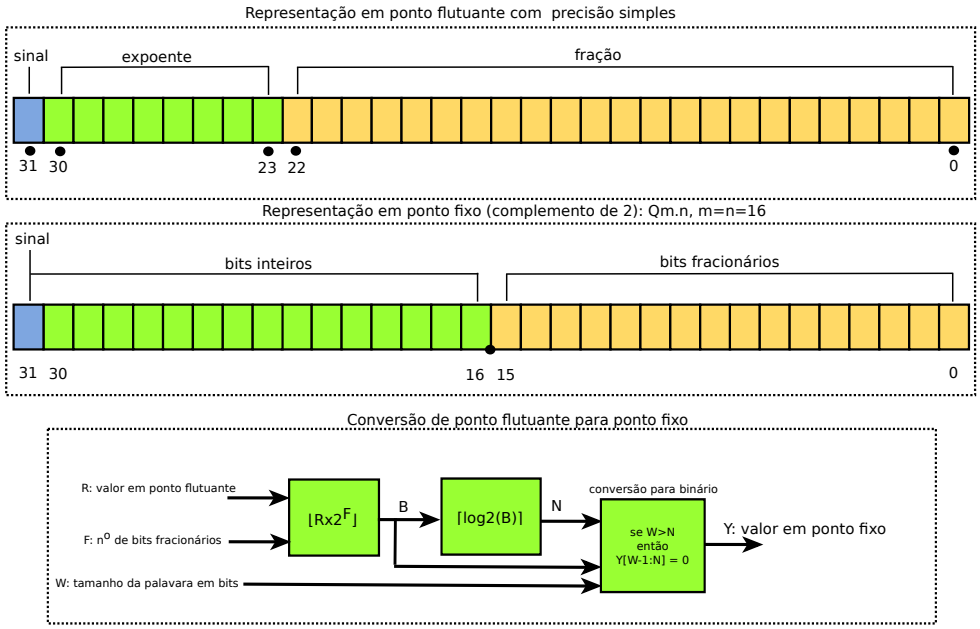
3.5.2 Projeto de RTL

As variáveis do modelo ESL na fase de projeto funcional frequentemente são representadas em ponto flutuante[15] que não tem suporte nativo em linguagens HDL. A

[15]O conjunto dos números reais é infinito. No entanto a representação dos números (por exemplo, em ponto flutuante) para processamento em computador é finita. Por esta razão, uma operação aritmética em ponto flutuante pode resultar em um número com expoente superior ao valor máximo permitido, ocorrendo o fenômeno denominado de *overflow*. Quando este expoente é menor que o valor mínimo, ocorre o *underflow*. Para evitar estes problemas, o cálculo da variância de uma imagem de dimensão WH e com valor máximo i_{max} da intensidade de píxel no domínio integral deve ser computada por duas variáveis (uma com pelo menos $\lceil log_2(WHi_{max}) \rceil$ bits para calcular a integral dos pixéis e outra com no mínimo $\lceil log_2(WHi^2_{max}) \rceil$ bits para o cálculo da integral do quadrado dos pixéis da imagem.

transição do projeto funcional para o RTL é acompanhada por uma transformação na representação dos dados, geralmente para a notação em ponto fixo[16], como pode ser visto na Figura 3.24.

Figura 3.24: Representações Numéricas em ponto fixo e ponto flutuante



Fonte: Elaborada pelo autor

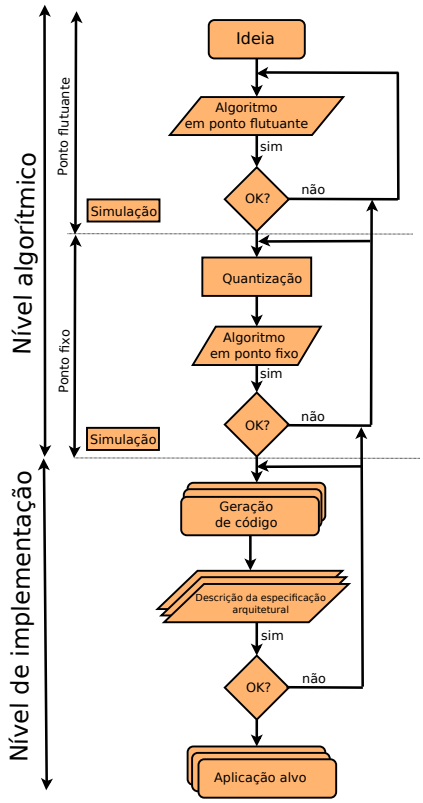
Uma das metodologias de conversão para ponto fixo[17] foi proposta por [61], transforma variáveis de ponto flutuante para ponto fixo em uma aplicação escrita em linguagem C. A primeira etapa desta técnica é chamada de anotações, onde o usuário define o formato em ponto fixo de algumas variáveis cuja especificação em ponto fixo é conhecida. Depois, uma etapa de interpolação determina a especificação da aplicação em ponto fixo. Os dados formatados em ponto fixo são obtidos a partir do fluxo de controle da Figura 3.25.

Durante a etapa de projeto de RTL a arquitetura do sistema é mais detalhada, sendo descrita pela transferência de dados entre registradores por meio de operações lógicas combinacionais e sequenciais. Neste nível de abstração, os circuitos são descritos por linguagens de descrição de *hardware* como Verilog, SystemVerilog ou VHDL, que descrevem a estrutura e o comportamento de circuitos digitais com seus conjuntos de

[16]Para maiores detalhes, consulte o Apêndice E para uma breve explanação sobre implementações de filtros digitais em ponto fixo.

[17]Para uma análise do erro de quantização devido ao efeito desssa conversão, consulte o Apêndice F.

Figura 3.25: Fluxo de conversão de ponto flutuante para ponto fixo



Fonte: Adaptada de [61]

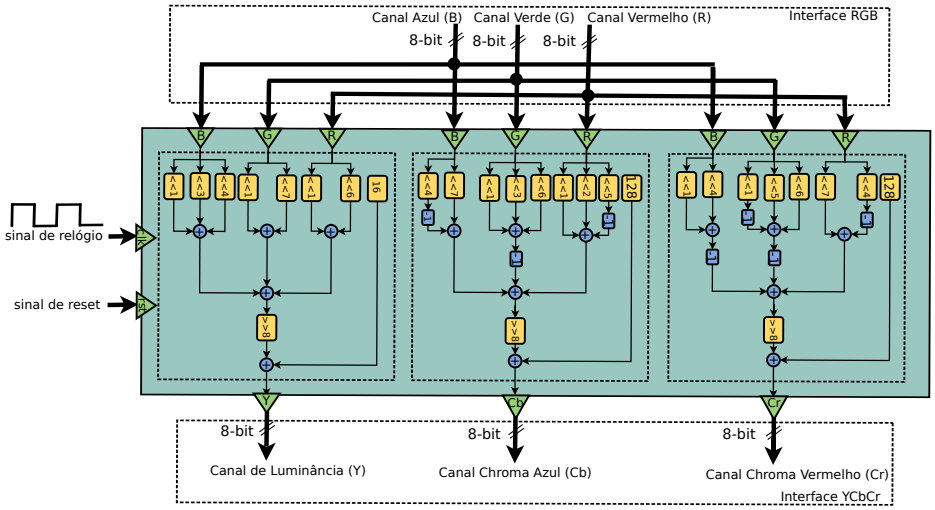
sintaxe e semântica. Como exemplo, o circuito da Figura 3.26 descreve um conversor de cores de RGB para YC_bC_r que pode ser representado por uma descrição em SystemVerilog no Código G.1.

3.5.3 Verificação de RTL

A metodologia de verificação universal (UVM, do inglês *Universal Verification Methodology*) tornou-se o padrão de verificação de projetos de circuitos integrados. UVM é uma biblioteca de classes em SystemVerilog com um alto nível de abstração que automatiza sequências de testes, tendo suporte das principais empresas de projeto de eletrônica automática (EDA, do inglês: *Electronic Design Automation*).

Uma das grandes características de UVM são as suas classes. Um bloco em UVM é um componente derivado de uma classe. As classes mais importantes são mostradas na árvore

Figura 3.26: Descrição de um Conversor de Cores RGB para YC_bC_r em nível RTL



Fonte: Elaborada pelo autor

da Figura 3.27.

A classe *uvm_void* é a classe base para todas as classes UVM. Ela é uma classe abstrata sem membros ou funções. Cada classe tem fases de simulações, que são etapas ordenadas de execução implementadas como métodos. As fases mais importantes são listadas (e estão ilustradas na Figura 3.28) a seguir:

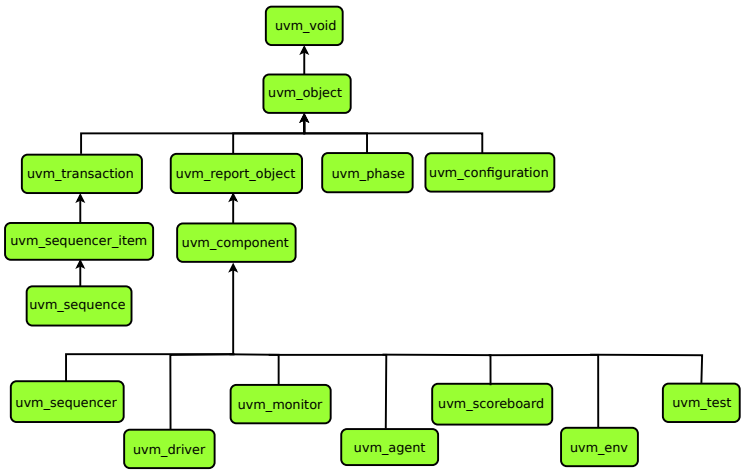
- a fase *build_phase* é responsável pela criação e configuração da estrutura do *testbench* e construção dos componentes da hierarquia
- a fase *connect_phase* é usada para conectar diferentes componentes em uma classe
- a fase *run_phase* é a principal fase da execução, onde a simulação será executada
- a fase *report_phase* pode ser utilizada para exibir resultados da simulação.

Para implementar alguns métodos importantes em classes e variáveis UVM oferece as UVM Macros. As macros mais comuns são:

- *'uvm_component_utils*: para registrar um novo tipo de classe quando a classe é derivada de uma classe *'uvm_component*
- *'uvm_object_utils*: semelhante à *'uvm_component_utils*, mas a classe é derivada da classe *'uvm_object*

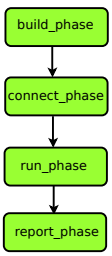
- *'uvm_field_int*: para registrar uma variável na *UVM factory*. Esta macro fornece funções como *copy()*, *compare()* e *print()*
- *'uvm_info*: para imprimir as mensagens durante o tempo de simulação no ambiente
- *'uvm_error*: esta *macro* envia mensagens com registros de erros.

Figura 3.27: Algumas classes de UVM



Fonte: Elaborada pelo autor

Figura 3.28: Algumas fases de UVM



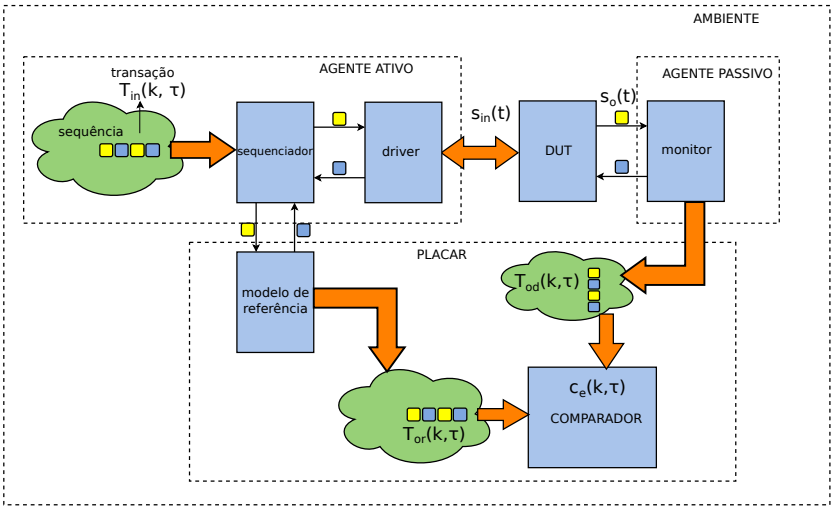
Fonte: Elaborada pelo autor

3.5.3.1 Estrutura de um ambiente UVM

A Figura 3.29 ilustra a estrutura geral de um ambiente UVM. Para estimular o circuito em teste (DUT, do inglês: *Design Under Test*) uma classe chamada *sequence* gera sequências de transações que serão transmitidas para outra classe chamada *sequencer*. Uma vez que o

DUT só reconhece os sinais provenientes da interface, uma classe *driver* recebe as transações do *sequencer* e converte os dados em sinais que estimulam o DUT.

Figura 3.29: Estrutura de um ambiente UVM



Fonte: Elaborada pelo autor

As transações que atravessam a interface devem ser captadas para uma posterior validação dos estímulos. Uma vez que o *driver* apenas converte transações para sinais, um outro bloco deve fazer a função inversa dele. O *monitor* é um componente que escuta a comunicação entre o *driver* e o DUT e recupera a transação. A classe *monitor* observa os sinais na interface e envia a transação para ser comparada com o modelo de referência.

Um *agent* é uma classe que normalmente contém três componentes: um *sequencer*, um *driver* e um *monitor*. Existem dois tipos de *agents*: *Active Agent*, que contém todos os três componentes e o *Passive Agent*, que contém apenas o *monitor* e o *driver*. O *agent* contém funções para a *build_phase*, para criar hierarquias, e para a *connect_phase*, para conectar os componentes.

O modelo de referência (refmod, do inglês: *Reference Model*) é um modelo concebido em uma fase inicial, antes da implementação do *testbench*, assumida como sendo o modelo ideal do RTL. Ele é um modelo do agent em um alto nível de abstração.

Um comparador (*comparator*) é uma classe que compara os dados entre o modelo de referência e o DUT. O modelo de referência e o comparador constituem uma classe chamada placar (*scoreboard*). O comparador verifica se as transações do *refmod* e a saída do DUT,

convertidas para transações pelo monitor de saída estão iguais.

Um ou mais *agents* e o *scoreboard* formam a classe *environment*. A classe *test* é responsável por fazer testes criando o ambiente (*environment*) e conectando a sequência (*sequence*) ao sequenciador (*sequencer*). O módulo *top* contém instâncias do DUT com suas interfaces e da classe *test*, além de definir sinais de relógio e *reset*.

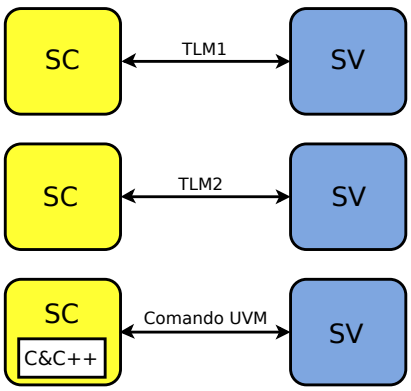
3.5.3.2 Direct Programming Interface (DPI)

A interface DPI (do inglês: *SystemVerilog Direct Programming Interface*) é um mecanismo de SystemVerilog para importar funções de linguagens externas como C, C++, SystemC ou até mesmo Octave [62]. A DPI é composta por duas camadas: a camada SystemVerilog e a camada da linguagem externa, que são isoladas entre si.

3.5.3.3 UVM Connect

Embora UVM forneça um alto nível de abstração para a construção de modelos de referência, SystemC tem um legado em modelos de alto nível para o projeto e validação de *hardware*. Com o intuito de permitir a reutilização de modelos de referência escritos em SystemC em *testbenches* UVM, a biblioteca UVMC (do inglês: *UVM Connect*) dispõe de conectividade TLM entre modelos e componentes escritos em SystemC e SystemVerilog, como pode ser visto na Figura 3.30 [17].

Figura 3.30: Visão geral da biblioteca UVMC



Fonte: Elaborada pelo autor

3.6 Conclusões

Este capítulo englobou um conjunto de tópicos com o intuito de consolidar a fundamentação da pesquisa com os seguintes temas: a seção 3.1 abordou os conceitos de processamento digital de imagens e a seção 3.2 tratou sobre visão computacional enquanto que métricas de classificação de imagens foram discutidas na seção 3.3.. Na seção 3.4 foram abordadas algumas técnicas de deteção de faces populares na literatura. A seção 3.5 forneceu uma base introdutória para o fluxo de projeto e verificação de sistemas digitais. As ferramentas introduzidas neste capítulo darão suporte aos assuntos que serão tratados no capítulo seguinte.

Capítulo 4

Metodologia proposta

Neste capítulo será descrita uma metodologia de projeto e validação de sistemas digitais com aplicações em processamento de imagens e visão computacional. Uma visão geral da metodologia é sumarizada na seção 4.1. A seção 4.2 descreve a aplicação dessa metodologia para realizar a verificação funcional de uma implementação em *hardware* de um conversor de espaço de cores de RGB para YC_bC_r utilizando a metodologia UVM. A metodologia é adaptada para realizar a validação de um modelo de referência em nível TLM de um sistema de detecção de faces utilizando o algoritmo de Viola Jones. Por fim, na seção 4.3 é descrita uma plataforma de prototipação em FPGA para auxiliar no processo de validação dos módulos de processamento de imagens.

4.1 Visão geral da metodologia

Em relação ao que foi exposto nos capítulos anteriores, a contribuição desta dissertação de mestrado é a proposição de uma metodologia de projeto e validação de sistemas com aplicação em visão computacional (como aplicação específica esta dissertação aborda sistemas de detecção de faces), tendo como diferencial a possibilidade de reuso de aplicações de domínio público (isto é, um modelo de detecção de faces descrito em *software* é utilizado como ponto de partida para o projeto de um circuito dedicado de detecção de faces). A metodologia proposta é composta pelos seguintes atributos:

- utiliza as ferramentas de verificação funcional da metodologia UVM;

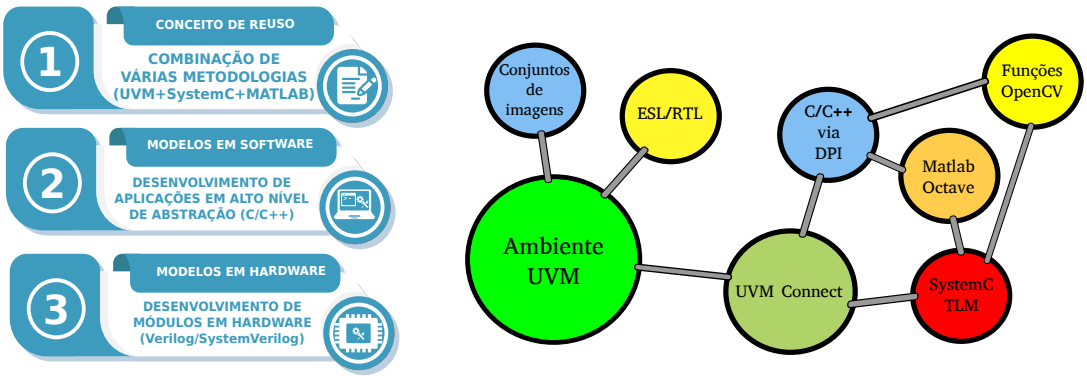
- integra o legado herdado de SystemC com ambientes de verificação em UVM por meio da biblioteca UVM Connect;
- exporta funções de processamento de imagem e visão computacional da biblioteca OpenCV.

Com essa combinação de atributos, é possível realizar a validação de aplicações de processamento de imagens de duas maneiras:

- **modo de operação 1**: uma verificação funcional em nível de sistemas, onde um modelo de referência avalia o funcionamento de um modelo em validação com alto nível de abstração;
- **modo de operação 2**: uma verificação funcional em nível de RTL, onde um modelo de referência avalia o funcionamento de um dispositivo em teste.

Uma visão geral da metodologia está ilustrada na Figura 4.1, onde cada conexão entre os elementos do diagrama representa uma integração dos componentes do diagrama. A metodologia é composta por um ambiente de verificação funcional UVM integrado com a biblioteca UVM Connect utilizado para validar modelos em nível ESL ou RTL. O ambiente pode importar funções de processamento de imagens e visão computacional utilizando a biblioteca OpenCV por meio de chamadas DPI ou dentro de módulos em SystemC. Da mesma forma, rotinas em MATLAB ou Octave também podem ser importadas para dentro do ambiente UVM.

Figura 4.1: Visão geral da metodologia de projeto e validação em *hardware*



Fonte: Elaborada pelo autor

Os dois modos de projeto e validação da metodologia requerem como entradas um banco de dados de imagens. Exemplos de resultados obtidos da validação podem ser visualizações de formas de ondas dos sinais das interfaces do RTL ou transações do ambiente UVM. Também podem ser gerados relatórios em modo texto com a análise do modelo de em teste (um módulo RTL ou modelo ESL) em função de um modelo de referência[1].

4.2 Validação funcional de aplicações de processamento de imagens

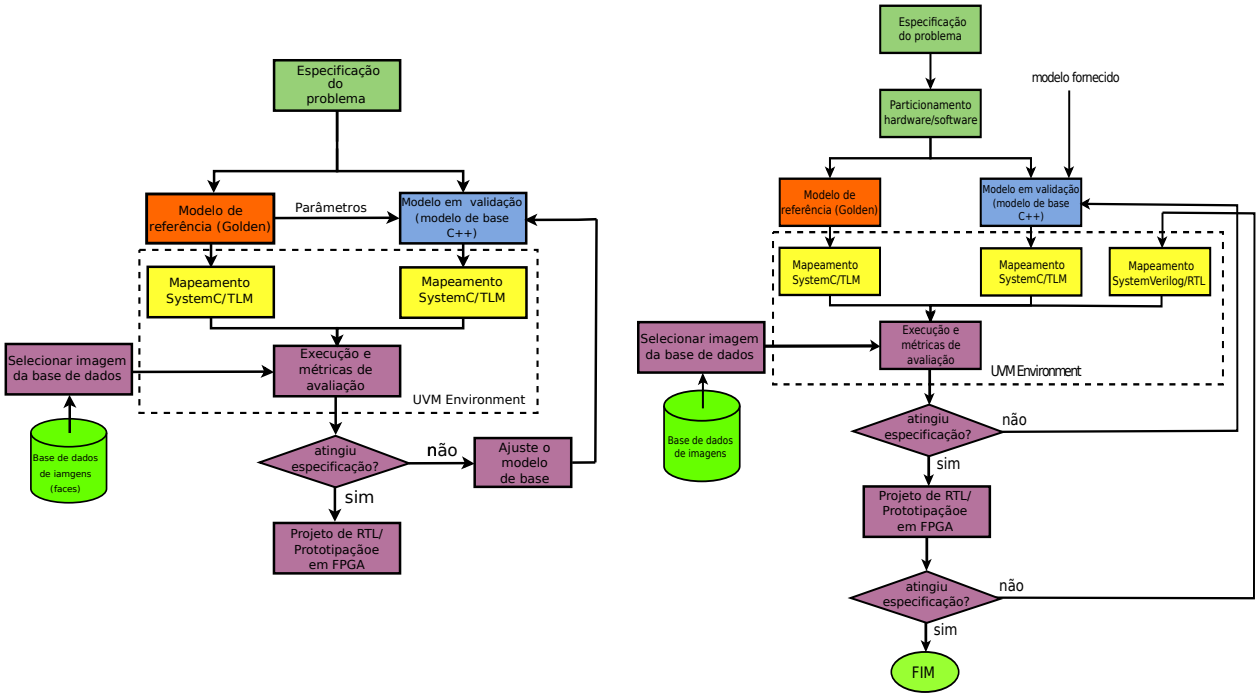
Existem diferentes maneiras de se executar um fluxo de projeto de sistemas digitais com ênfase em aplicações de processamento de imagem. A Figura 4.2 ilustra o fluxo de projeto e validação nos dois modos de operação da metodologia proposta neste trabalho. Dentre as ferramentas que são utilizadas (funcionando em um ambiente Linux) ao longo deste fluxo podem ser citadas as ferramentas de EDA (por exemplo, VCS da Synopsys), compiladores de C/C++ (gcc e g++) além do Octave (versão alternativa do MATLAB).

No modo de operação 1, o primeiro passo do fluxo é a especificação do problema, que é o nível mais alto de abstração e descreve a aplicação do projeto. Depois, uma partição manual separa o projeto em dois ramos: uma partição de projeto em *software* (é nesta partição onde serão descritos em C/C++/SystemC os modelos de alto nível de abstração, como por exemplo, o modelo de referência) e outra em *hardware* (nesta partição são descritos em Verilog/SystemVerilog os módulos em RTL e os componentes do ambiente de verificação em UVM). No primeiro ramo do fluxo, um modelo de referência escrito em *software* de alto nível é mapeado em nível de algoritmo, enquanto o segundo ramo que mapeia uma aplicação em nível de sistema em uma versão arquitetural[2]. Ambos os modelos (modelo de referência e modelo em validação) são mapeados em nível TLM, sendo integrado dentro de um ambiente UVM que automatiza a geração de testes, coletando transações de imagens a partir de uma

[1] Um exemplo de relatório seria a comparação do valor das intensidades dos píxeis contidos numa imagem produzidos por dois modelos de conversor de cores de RGB para YC_bC_r (um descrito em SystemVerilog e outro sendo importado da biblioteca OpenCV dentro de um módulo SystemC).

[2] Por decisão de projeto foi utilizada uma versão já disponível para o modelo em validação, sendo uma alternativa ao método de projeto *bit-exact*, onde modelo de referência e modelo em validação são implementados em paralelo.

Figura 4.2: Fluxo de projeto e validação da metodologia proposta nos dois modos de operação



(a) **modo de operação 1**: validação de ESL

(b) **modo de operação 2**: validação de RTL

Fonte: Elaborada pelo autor

base de dados que contém imagens com faces humanas[3]. A última etapa do fluxo de projeto é a métrica de avaliação dos sistemas, que é feita dentro de um ambiente de integração *hardware* e *software*. O fluxo encerra quando a aplicação da partição em *hardware* satisfaz as retrições estabelecidas na especificação do problema. O modelo matemático do ambiente UVM e das métricas de avaliação para este modo de operação serão detalhados na subseção 4.2.3.

Para o modo de operação 2, o fluxo de projeto proposto na Figura 4.2.(b) estimula o modelo de referência para avaliar o funcionamento de um modelo em nível RTL. Para este modo de operação será utilizado como exemplo de aplicação a validação de um conversor de espaço de cores de RGB para YC_bC_r e as métricas de avaliação neste exemplo consistem em determinar se um píxel no espaço YC_bC_r produzido por um modelo de referência em SystemC equivale ao píxel produzido pelo modelo RTL dado que ambos os modelos recebem como entrada o mesmo píxel no espaço RGB. Este exemplo de aplicação será detalhado na

[3]Uma publicação sobre este assunto pode ser vista em [63].

subseção 4.2.2.

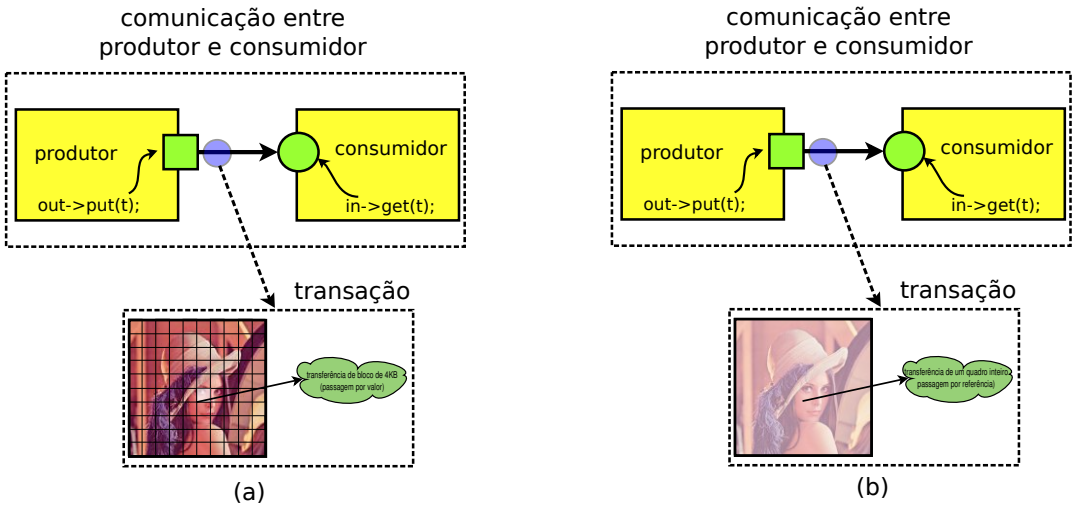
A metodologia de projeto e validação pode ser generalizada para qualquer aplicação de processamento de imagem e visão computacional, sendo necessário modificar as interfaces do RTL e os componentes *sequence*, *driver* e *monitor* do ambiente UVM. Uma plataforma de prototipação em FPGA auxilia no fluxo de projeto e validação por meio de simulação com ferramentas de EDA. Este protótipo permite visualizar operações de processamento de imagens em um monitor VGA e será detalhado na seção 4.3.

4.2.1 Mapeamento TLM

O mapeamento de uma aplicação de processamento de imagens em nível de transações TLM é a primeira etapa para a construção de um ambiente de validação com ênfase em aplicações de visão computacional utilizando a biblioteca UVM Connect. Neste estágio, a aplicação de alto nível, que é descrita com funções da biblioteca OpenCV, fornece funções nativas para processamento de imagens que são importadas para o ambiente UVM utilizando a interface DPI.

A Figura 4.3.(a) ilustra a comunicação entre um produtor e um consumidor. A transação produzida na fonte (produtor) é uma imagem (matriz de inteiros) que deve ser transferida para o destino (consumidor).

Figura 4.3: Comunicação TLM com transferência de grande volume de dados



Fonte: Elaborada pelo autor

Uma classe chamada *frame_sequence* ilustrada no Código 4.1 gera as transações do produtor. Para iniciar a transação, uma função externa chamada *readframe* exporta funções da biblioteca OpenCV implementas em C++ via DPI para um ambiente UVM/SystemVerilog, como pode ser visto no código 4.2.

Código 4.1: Classe *frame_sequence* em SystemVerilog

```
context function longint unsigned readframe(string filename);
class frame_seq extends uvm_sequence #(frame_tr);
  `uvm_object_utils(frame_seq)
  function new(string name="frame_seq");
    super.new(name);
  endfunction: new

  string filename = "img.jpg";

  task body;
    frame_tr tr = frame_tr::type_id::create("tr");
    start_item(tr);
    tr.a = readframe(filename);
    finish_item(tr);
  endtask:body
endclass: frame_seq
```

Código 4.2: Uma função externa do OpenCV para leitura de um quadro de imagem

```
extern "C" unsigned long long readframe(const char* filename)
{
  Mat image = imread(filename, 1);
  return (unsigned long long)image.data;
}
```

O problema em questão é que o tamanho máximo do pacote de dados que pode ser transferido em um ambiente UVM Connect é de $4KB$ por transação. Para lidar com este problema, uma técnica descrita em [64] divide a transação que tranfere os dados da imagem em transações de pequenos blocos da imagem. A definição das transações em um ambiente UVM Connect é composta por uma camada em SystemVerilog e outra em SystemC. Os Códigos 4.3 e 4.4 definem as transações *frame_transaction* definidas como blocos de $4KB$. Esta abordagem é funcional, mas não é eficiente quando um grande volume de dados deve ser transmitido por cada transação, resultando em um tempo de simulação longo.

Ao invés de enviar o conteúdo de dados, que é limitado em $4KB$ por pacote, a abordagem proposta neste trabalho envia o endereço de memória da transação (como pode ser visto na Figura 4.3.(b)), que consiste em um endereço de 64 bits de memória, que pode ser armazenado em uma variável **long unsigned integer** dentro de um módulo SystemVerilog e em uma variável **unsigned long long** em um módulo SystemC. Essa abordagem leva a

uma redução significativa no tempo de simulação em comparação com o método proposto em [64].

Código 4.3: Camada em SystemVerilog da transação *frame_transaction* definida como um bloco de 4KB de dados

```
`define SIZE_CHUNK 1024
class a_tr extends uvm_sequence_item;
  int a[`SIZE_CHUNK];
  `uvm_object_param_utils_begin(a_tr)
    `uvm_field_sarray_int(a, UVM_DEFAULT)
  `uvm_object_utils_end

  function new (string name = "a_tr");
    super.new(name);
  endfunction
endclass
```

Código 4.4: Camada SystemC da transação *frame_transaction* definida como um bloco de 4KB de dados

```
#define SIZE_CHUNK 1024
struct a_tr {
   int a[SIZE_CHUNK];
};
UVMC_UTILS_1(a_tr, a)
```

O esboço das transações definidas como endereço de memória pode ser visto nos códigos 4.5 e 4.6. Este esboço é o núcleo do ambiente de validação em UVM com ênfase em aplicações de processamento de imagens proposto neste trabalho.

Código 4.5: Camada SystemVerilog da transação *frame_transaction* definida como endereço de memória da imagem

```
class a_tr extends uvm_sequence_item;
  longint unsigned a;
  `uvm_object_param_utils_begin(a_tr)
    `uvm_field_int(a, UVM_DEFAULT)
  `uvm_object_utils_end

  function new (string name = "a_tr");
    super.new(name);
  endfunction
endclass
```

Código 4.6: Camada SystemC da transação *frame_transaction* definida como endereço de memória da imagem

```
struct a_tr {
  unsigned long long a;
};
UVMC_UTILS_1(a_tr, a)
```

4.2.2 Validação funcional de um conversor de cores

Conforme supracitado na subseção 3.5.3.1, a estrutura geral do ambiente de validação em UVM da Figura 3.29 é composta por um agente ativo, que estimula um dispositivo em teste (DUT), por um agente passivo que coleta dados para serem avaliados e por um *scoreboard* que avalia os dados (transações) vindo dos dois agentes.

No caso particular de um ambiente de validação com ênfase em processamento de imagens, para efeito de simplicidade será realizada a validação de um conversor de espaço de cores de RGB para YC_bC_r utilizando a estrutura de projeto e validação proposta neste trabalho.

O agente ativo cria uma sequência de transações aleatórias $T_{in}(k,\tau)$, de acordo com a Equação 4.1, onde $T_{in}(k,\tau)$ é um vetor de inteiros aleatórios $t_{in}(\cdot,\cdot) \sim \mathcal{U}(-2^{31}, 2^{31}-1)$ e $\mathcal{U}(-2^{31}, 2^{31}-1)$ denota uma distribuição discreta de probabilidade uniforme que estimula o modelo de referência (um modelo de alto nível escrito em SystemC e OpenCV do conversor de cores de RGB para YC_bC_r) e o *driver*. Para o caso em que as transações são imagens naturais adquiridas com uma câmera ou armazenadas em uma base de dados, na frequência ω, $T_{in}(k,\tau)$ tem densidade espectral de potência $S(|\omega|) = \frac{c}{|\omega|^{2-\chi}}$, onde $c > 0$ e $\chi < 1$ [65].

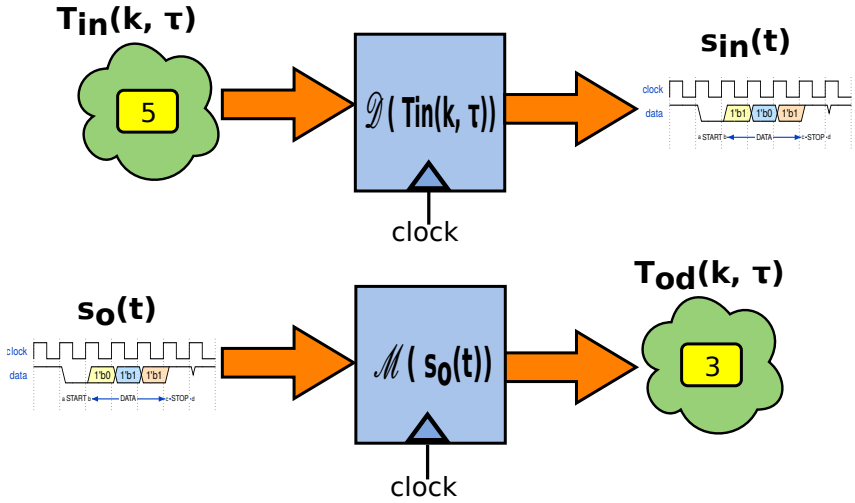
$$T_{in}(k,\tau) = \left[\begin{array}{cccc} t_{in}(0,\tau) & t_{in}(1,\tau) & \ldots & t_{in}(N-1,\tau) \end{array} \right] \tag{4.1}$$

O *driver* converte a transação $T_{in}(k,\tau)$ em sinais digitais $s_{in}(t) = \mathscr{D}(T_{in}(k,\tau))$, onde a função $\mathscr{D}(\cdot)$ converte transações em sinais que estimulam a interface do DUT. O DUT produz sinais de saída $s_o(t) = DUT(s_{in}(t))$ que são coletados pelo *monitor*. O *monitor* define a função $\mathscr{M}(\cdot)$ que converte os sinais $s_o(t)$ em transações de saída $T_{od}(k,\tau) = \mathscr{M}(s_o(t))$, onde $T_{od}(k,\tau)$ é definido na Equação 4.2. A operação das funções $\mathscr{D}(\cdot)$ e $\mathscr{M}(\cdot)$ está ilustrada na Figura 4.4. De maneira similar, as transações de saída do modelo de referência $T_{or}(k,\tau)$ tem a forma da Equação 4.3.

$$T_{od}(k,\tau) = \left[\begin{array}{cccc} t_{od}(0,\tau) & t_{od}(1,\tau) & \ldots & t_{od}(N-1,\tau) \end{array} \right] \tag{4.2}$$

$$T_{or}(k,\tau) = \left[\begin{array}{cccc} t_{or}(0,\tau) & t_{or}(1,\tau) & \ldots & t_{or}(N-1,\tau) \end{array} \right] \tag{4.3}$$

Figura 4.4: Operação das funções $\mathscr{D}(\cdot)$ and $\mathscr{M}(\cdot)$



Fonte: Elaborada pelo autor

O tempo τ da *k-ésima* transação é relacionado com o *t-ésimo* período de *clock* de acordo com a Equação 4.4, onde N é o tamanho de $T_{in}(k,\tau)$.

$$\begin{cases} k \equiv t(\text{mod } N) \\ \tau = \frac{t-k}{N} \end{cases} \tag{4.4}$$

Ambas as transações geradas pelo *monitor* e pela saída do modelo de referência devem ser comparadas pelo comparador do ambiente UVM, que computa um erro de comparação $c_e(k,\tau)$. Se a diferença entre $T_{or}(k,\tau)$ e $T_{od}(k,\tau)$ é menor que um limiar ϵ, então ocorre um *match* entre $T_{or}(k,\tau)$ e $T_{od}(k,\tau)$, caso contrário ocorre um *mismatch* entre eles de acordo com a Equação 4.5.

$$c_e(k,\tau) = \begin{cases} 1, & |T_{or}(k,\tau) - T_{od}(k,\tau)| \leq \epsilon \\ 0, & \text{caso contrário} \end{cases} \tag{4.5}$$

A modelagem matemática do ambiente de validação[4] com ênfase em processamento de

[4]Os códigos do ambiente de validação do conversor de cores encontram-se no apêndice G.3.

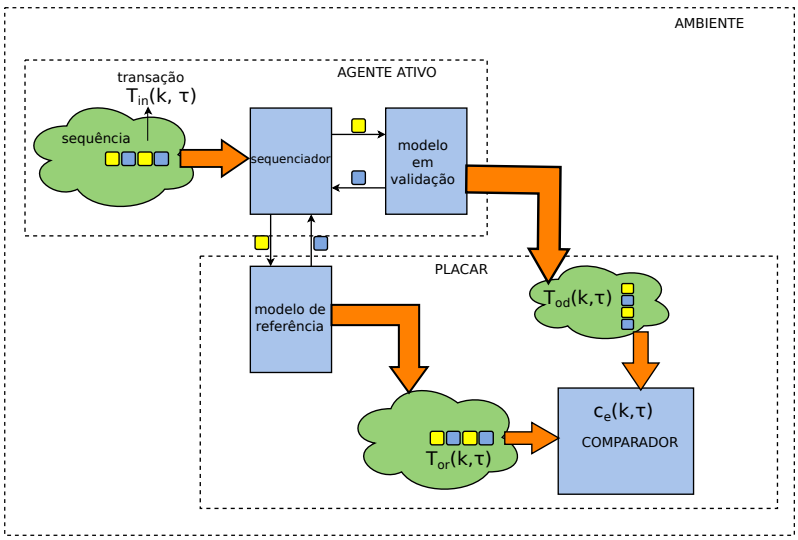
imagens é uma descrição formal que pode ser estendida para outras aplicações. No caso particular do conversor de cores a interface de entrada RGB coleta no instante t três pixéis vermelhos $R(t)$, $G(t)$ e $B(t)$ e o DUT produz três pixéis de saída $Y(t)$, $C_b(t)$ e $C_r(t)$ de acordo com a Equação 4.6.

$$\begin{bmatrix} Y(t) \\ C_b(t) \\ C_r(t) \end{bmatrix} = DUT(R(t), G(t), B(t)) = \begin{bmatrix} 16 + \frac{66R(t)}{256} + \frac{129G(t)}{256} + \frac{25B(t)}{256} \\ 128 - \frac{37R(t)}{256} - \frac{74G(t)}{256} + \frac{112B(t)}{256} \\ 128 + \frac{112R(t)}{256} - \frac{94G(t)}{256} - \frac{18B(t)}{256} \end{bmatrix} \tag{4.6}$$

4.2.3 Validação funcional de um detector de faces

Uma etapa de pré-validação é feita quando a implementação do RTL não está disponível nos estágios iniciais do fluxo de projeto de sistemas complexos. Nesses casos, o ambiente da Figura 3.29 é estruturado de acordo com a Figura 4.5. Nesta estrutura, o componente *driver* é removido do agente ativo e o agente passivo é retirado, uma vez que, não há interface de sinais nesta fase do fluxo de projeto. Em seguida, a validação é realizada em nível algorítmico e sistêmico e o comparador avalia as transações provenientes do modelo de referência e do modelo em validação, que é uma versão TLM da especificação que será substituída pelo modelo de RTL em uma etapa posterior.

Figura 4.5: Estrutura de um ambiente de pré-validação em UVM

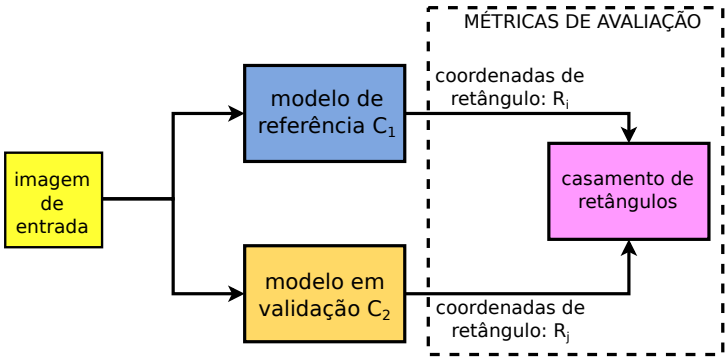


Fonte: Elaborada pelo autor

4.2.3.1 Métricas de avaliação

Nesta sub-subseção, o modelo de referência é definido como o modelo C_1 e o modelo em validação, que é uma implementação do algoritmo de Viola Jones em ponto fixo[5], como o modelo C_2, que são dois sistemas com a mesma imagem de entrada. A saída desses modelos são coordenadas regiões de retângulos que delimitam uma provável face, como mostrado na Figura 4.6. Esses modelos de detecção de faces são compostos por classificadores em cascata, conforme já foi mencionado na Figura 3.21. Cada estágio do classificador contém árvores de decisões para classificar características. No contexto do algoritmo de Viola Jones (ver Figura 4.7) as regras de busca de características Haar são definidas pela Equação 3.18. Caso a soma ponderada da integral da imagem das características (filtros retangulares de Haar) seja menor que o limiar definido para cada nó da árvore de decisões, o algoritmo executa a regra do nó esquerdo da árvore, caso contrário, o algoritmo executa a regra do nó direito [66].

Figura 4.6: Métricas de Avaliação do comparador do ambiente UVM da Figura 4.5



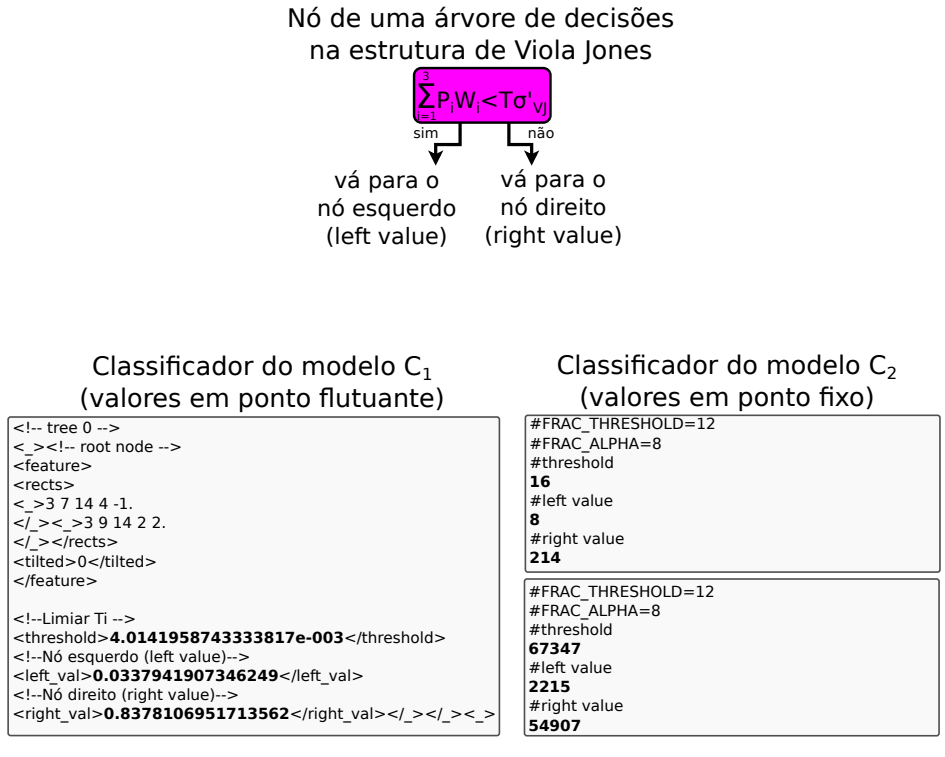
Fonte: Elaborada pelo autor

Conforme pode ser visto na Figura 4.7, o classificador em cascata do modelo C_1 (que é uma implementação em OpenCV do algoritmo de Viola Jones) define os valores das características com representação em ponto flutuante em um arquivo XML. Por outro

[5]Os valores dos limiares que definem se uma característica é uma face ou não, bem como os valores dos limiares de cada estágio do classificador em cascata e os valores atribuidos aos pesos dos retângulos que contém as características são convertidos para a representação em ponto fixo. O *script* de conversão do arquivo XML do OpenCV para um arquivo texto contendo as características de faces em ponto fixo encontra-se no Código G.4. A escolha da quantidade de bits para as variáveis do classificador é manual, sendo consequência da decisão de projeto em utilizar um modelo em validação já disponível ao invés de se escrever ambos os modelos de referência e de validação (solução *bit-exact*).

lado, o modelo C_2 define esses valores na representação de ponto fixo em um arquivo texto[6]. Assim, por exemplo, no primeiro nó do primeiro estágio do classificador os valores do limiar da característica e os valores do nós esquerdo e direitos no modelo C_1 estão definidos com os números $4.0141958743333817 \times 10^{-3}$, 0.0337941907346249 e 0.8378106951713562, respectivamente. Ao se utilizar $FRAC\_THRESHOLD = 12$ bits fracionários para a representação em ponto fixo do limiar e $FRAC\_ALPHA = 8$ bits[7] fracionários para a representação dos valores dos nós esquerdo e direitos, os valores destas variáveis no modelo C_2 se tornam 16 ($\lfloor 4.0141958743333817 \times 10^{-3} \times 2^{12} \rfloor$), 8 ($\lfloor 0.0337941907346249 \times 2^{8} \rfloor$) e 214 ($\lfloor 0.8378106951713562 \times 2^{8} \rfloor$), respectivamente. De modo similar, se $FRAC\_THRESHOLD = 24$ e $FRAC\_ALPHA = 16$, então os mesmos valores do limiar, nó esquerdo e nó direito são representados com os números 67347, 2215 e 54907, respectivamente.

Figura 4.7: Árvore de decisões do algoritmo de Viola Jones



Fonte: Elaborada pelo autor

[6]Isto seria uma decisão de projeto arquitetural conveniente para o caso dos valores desses classificadores serem armazenadas em uma memória ROM (do inglês: *Read-only Memory*) durante a etapa de descrição do RTL

[7]As variáveis $FRAC\_THRESHOLD$ e $FRAC\_ALPHA$ são parâmetros ajustáveis no modelo C_2.

O comparador do ambiente de pré-validação em UVM[8] executa uma análise nas transaçãoes de saída provenientes de C_1 e C_2. Para computar um *match*, a Equação 4.5 deve ser adaptada para calcular a similaridade entre retângulos. Uma métrica que pode ser aplicada para medir a similaridade entre dois alvos em uma imagem é o coeficiente de *dice* [38].

Os retângulos $R_i = (x_i, y_i, L_i)$ e $R_j = (x_j, y_j, L_j)$ tem áreas $S(R_i) = L_i^2$ e $S(R_j) = L_j^2$, respectivamente. O coeficiente de *dice* entre R_i e R_j é definido como $dice(R_i, R_j) = \frac{2S(R_i \cap R_j)}{S(R_i)+S(R_j)} = \frac{2S(R_i \cap R_j)}{L_i^2+L_j^2}$, onde $0 \leq dice(R_i, R_j) \leq 1$. Esta métrica fornece o percentual de similaridade entre dois retângulos e ela é utilizada no cálculo de comparação do comparador do ambiente UVM.

A área da interseção entre os retângulos R_i e R_j é calculada aplicando a Equação 4.7 na Equação 4.8. Note que o coeficiente de *dice* é inversamente proporcional à média aritmética da área dos dois retângulos R_i e R_j.

O comparador do ambiente de pré-validação também executa uma métrica normalizada entre 0 e 1 que é inversamente proporcional à média geométrica da área dos retângulos R_i e R_j de acordo com a Equação 4.9.

$$\begin{bmatrix} \delta & \zeta \\ \eta & \rho \end{bmatrix} = \begin{bmatrix} max(x_i, x_j) & min(x_i + L_i, x_j + L_j) \\ max(y_i, y_j) & min(y_i + L_i, y_j + L_j) \end{bmatrix} \tag{4.7}$$

$$S(R_i \cap R_j) = \begin{cases} (\zeta - \delta)(\rho - \eta), & \text{se } \zeta > \delta \text{ e } \rho > \eta \\ 0, & \text{caso contrário} \end{cases} \tag{4.8}$$

$$m(R_i, R_j) = \frac{S(R_i \cap R_j)}{\sqrt{S(R_i)S(R_j)}} = \frac{S(R_i \cap R_j)}{L_i L_j} \tag{4.9}$$

Se o comparador está avaliando as transações oriundas de C_1 e C_2 com o percentual de similaridade de retângulos normalizado pela média geométrica das áreas dos retângulos R_i e R_j, então a expressão de comparação é dada pela Equação 4.10.

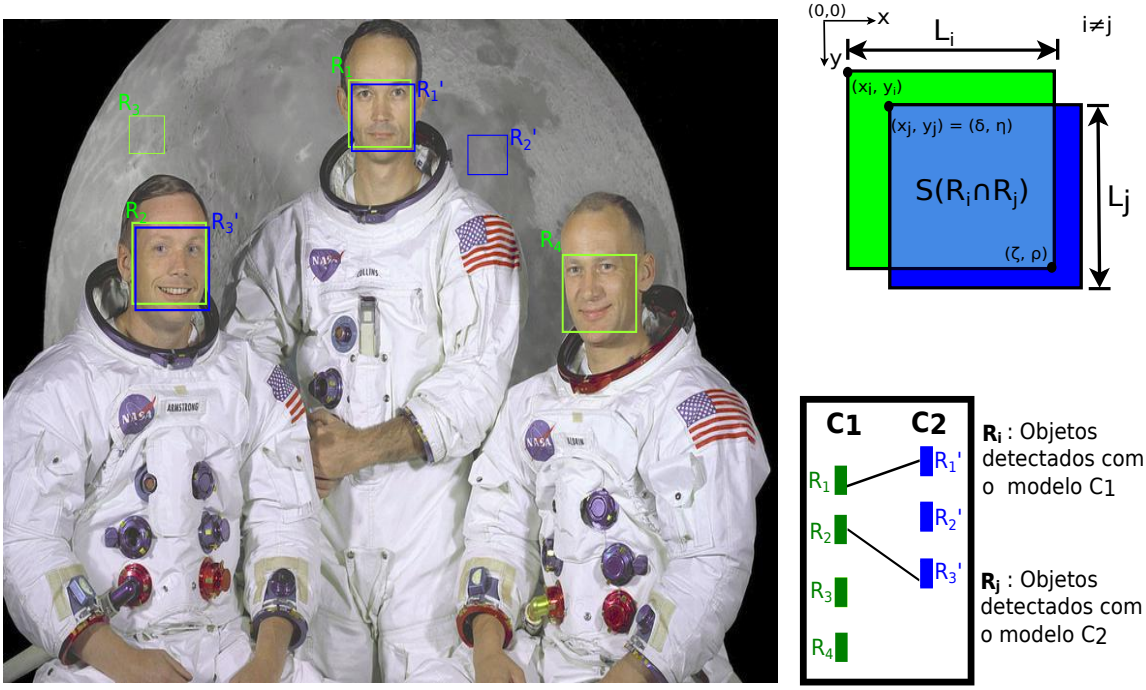
[8]Os códigos do ambiente de pré-validação do sistema de detectação de faces utilizando o algoritmo de Viola Jones encontram-se no apêndice G.5.

$$c_e(k,\tau) = \begin{cases} 1, & m(R_i(k,\tau), R_j(k,\tau)) > \epsilon \\ 0, & \text{caso contrário} \end{cases} \tag{4.10}$$

Como uma decisão de projeto, foi escolhido o limiar de comparação $\epsilon = 0.75$, semelhante a tolerância estabelecida no percentual de sobreposição da técnica de avaliação de performance de vídeo proposta em [38].

Para ilustrar o algoritmo de casamento de retângulos, na Figura 4.8, os quadrados verdes R_i, onde $i = 1, 2, 3, 4$ são os vetores de saída produzidos por C_1, enquanto que os quadrados azuis R_j, para $j = 1', 2', 3'$ são produzidos por C_2. De acordo com esses vetores, ocorreram dois *matches*, um entre R_1 e R'_1, e outro entre R_2 e R'_3. Mais ainda, o modelo C_1 produziu um falso positivo (detectou um objeto que não é uma face) na região definidas pelas coordenadas do retângulo R_3. O modelo C_2 teve um falso negativo (não detectou uma face na região onde C_1 detectou R_4), consequentemente, ocorreu um *mismatch* no retângulo R_4 do modelo C_1.

Figura 4.8: Casamento de Retângulos (fotografia: Nasa on The Commons)

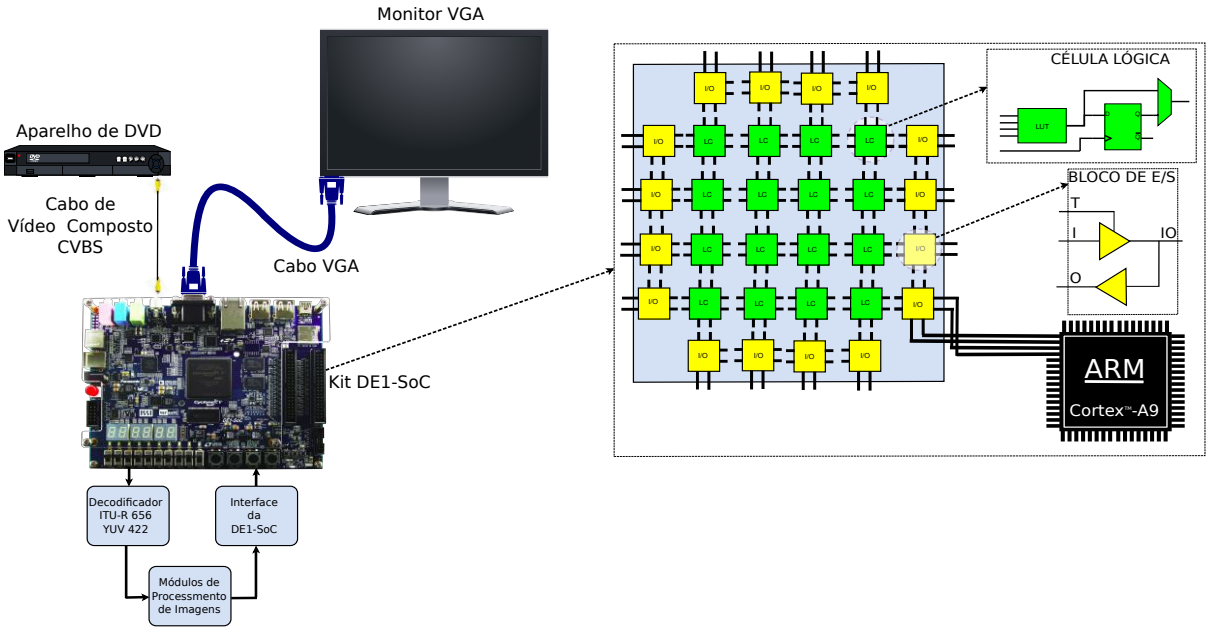


Fonte: Elaborada pelo autor

4.3 Plataforma para prototipação em FPGA

A plataforma de prototipação em FPGA[9] com ênfase em aplicações de visão computacional é uma estrutura para processamento de vídeo em tempo real ilustrada na Figura 4.9. A estrutura é composta por um *kit* de desenvolvimento em FPGA DE1-SoC da Terasic, que contém cerca de 85 mil elementos lógicos acoplado à um processador ARM *dual-core* de $800MHz$ [68]. A entrada de vídeo é conectada por um cabo de vídeo composto CVBS (do inglês: *Composite Video Blanking and Sync*) à uma entrada de vídeo na DE1-SoC. A fonte de vídeo (no caso da plataforma, um aparelho de DVD ou uma câmera) transmite sinal de vídeo composto analógico no formato YUV que é decodificado em formato YC_bC_r por um circuito decodificador de TV (ADV7180). O sinal em YC_bC_r é armazenado em um *buffer* na FPGA, que converte os pixéis para o formato RGB. Um controlador de vídeo VGA (do inglês: *Video Graphics Array*) converte os sinais RGB para sinais analógicos que reproduzem o vídeo em um monitor digital.

Figura 4.9: Plataforma para Prototipação em FPGA



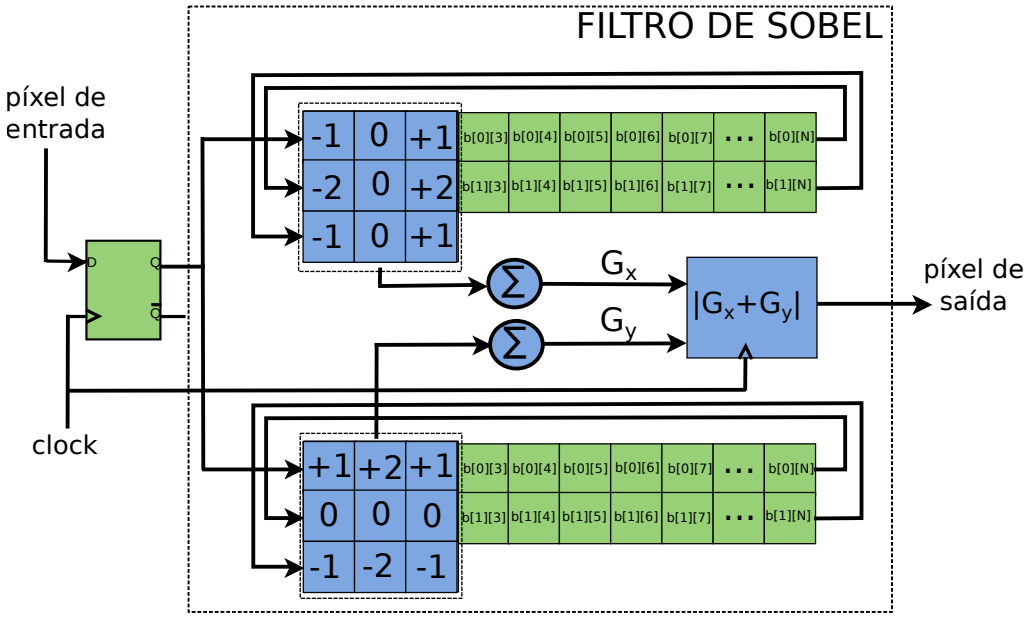
Fonte: Elaborada pelo autor

Os sinais de vídeo armazenados no *buffer* podem ser processados, por exemplo, com

[9]Uma versão dessa plataforma com vídeo em alta resolução pode ser vista em [67].

uma filtragem espacial já discutida na subseção 3.2.1. Diferentemente do que ocorre na representação da imagem em *software*, que é representada por uma matriz bidimensional, o cabo CVBS transmite um sinal de píxel a cada pulso de relógio e o fluxo de sinais de vídeo processados em FPGA armazena uma linha da imagem com sinal de sincronismo de linha. Desse modo, para realizar uma convolução de uma imagem com dois núcleos de Sobel descritos pelas matrizes das equações 3.3 e 3.4, uma imagem com resolução VGA de 800×600 pixéis é armazenada em um *buffer* que contém 2 linhas, onde cada linha armazena 800 pixéis da imagem. A Figura 4.10 ilustra a arquitetura de um filtro de Sobel em *hardware*, que processa os pixéis armazenados no *buffer* da FPGA e transmite os sinais processados para a saída VGA.

Figura 4.10: Filtro de Sobel em Hardware



Fonte: Elaborada pelo autor

A convolução de um fluxo de $B = 800$ pixéis $p_{in}(t)$ por linha com os núcleos K_x e K_y no instante t geram dois gradientes $p^x_{out}(t)$ e $p^y_{out}(t)$ de acordo com as equações 4.11 e 4.12, onde o pixel de saída é dado por $p_{out}(t) = |\sum p^x_{out}(t)| + |\sum p^y_{out}(t)|$.

A implementação em SystemVerilog da plataforma de prototipação de aplicações de processamento de vídeo na DE1-SoC encontra-se no apêndice G.2. Um módulo do filtro de Sobel está implementado na plataforma como exemplo, mas outros módulos de

processamento podem ser adicionados na plataforma, haja vista que sua arquitetura é modularizada.

$$p^{x}_{out}(t) = \begin{bmatrix} K_x(0,0) \times p_{in}(t) & K_x(0,1) \times p_{in}(t-B-3) & K_x(0,2) \times p_{in}(t-2\times B-6) \\ K_x(1,0) \times p_{in}(t-1) & K_x(0,0) \times p_{in}(t-B-4) & K_x(0,0) \times p_{in}(t-2\times B-7) \\ K_x(2,0) \times p_{in}(t-2) & K_x(0,0) \times p_{in}(t-B-5) & K_x(0,0) \times p_{in}(t-2\times B-8) \end{bmatrix} \tag{4.11}$$

$$p^{y}_{out}(t) = \begin{bmatrix} K_y(0,0) \times p_{in}(t) & K_y(0,1) \times p_{in}(t-B-3) & K_y(0,2) \times p_{in}(t-2\times B-6) \\ K_y(1,0) \times p_{in}(t-1) & K_y(0,0) \times p_{in}(t-B-4) & K_y(0,0) \times p_{in}(t-2\times B-7) \\ K_y(2,0) \times p_{in}(t-2) & K_y(0,0) \times p_{in}(t-B-5) & K_y(0,0) \times p_{in}(t-2\times B-8) \end{bmatrix} \tag{4.12}$$

4.4 Conclusões

Neste capítulo foi descrita metodologia de projeto e validação de sistemas digitais com aplicações em processamento de imagens e visão computacional. Uma visão geral da metodologia foi sumarizada na seção 4.1. A seção 4.2 descreve uma aplicação dessa metodologia para realizar a verificação funcional de uma implementação em *hardware* de um conversor de espaço de cores de RGB para YC_bC_r utilizando a metodologia UVM. A metodologia foi adaptada para realizar a validação de um modelo de referência em nível TLM de um sistema de detecção de faces utilizando o algoritmo de Viola Jones. Por fim, na seção 4.3 foi descrita uma plataforma de prototipação em FPGA para auxiliar no processo de validação dos módulos de processamento de imagens.

Capítulo 5

Resultados e discussões

Neste capítulo serão apresentados os resultados da plataforma proposta no capítulo anterior. Na seção 5.1 são ressaltados os resultados do desempenho obtido com a técnica de extensão do tamanho máximo do pacote de dados por transação utilizando UVM Connect. Na seção 5.2 são discutidos os resultados da da estrutura de validação de um conversor de cores de RGB para YC_bC_r. Na seção 5.3 são apresentados os resultados do projeto e validação de um sistema de detecção de faces utilizando o algoritmo de Viola Jones com as métricas de avaliação propostas no capítulo anterior. Em último momento, na seção 5.4 são discutidos os resultados da plataforma de prototipação em FPGA de aplicações de visão computacional.

5.1 Extensão da carga máxima por transação em um ambiente UVM Connect

Para comparar o tempo de execução para enviar transações de quadros de imagens de uma fonte (produtor) para um destino (consumidor), foram propostos dois métodos: o primeiro método consistem em fragmentar os dados de acordo com a técnica proposta em [64]. O segundo método é a abordagem proposta neste trabalho de dissertação que consiste em definir a transação como um endereço de memória que armazena os dados da imagem. A tabela 5.1 mostra os resultados do tempo de simulação para transferir imagens de $400KB$, $4000KB$, $20000KB$ e $40000KB$ utilizando os dois métodos.

Tabela 5.1: **Método 1**: transmissão por valor. **Método 2**: transmissão por referência.

Tamanho da Imagem	**Tempo de Simulação**		**Ganho**
	Método 1	**Método 2**	
400KB	1,134s	0,818s	1,39
4000KB	4,070s	0,846s	4,81
20000KB	16,891s	0,924s	18,28
40000KB	33,150s	0,915s	36,23

O tempo de simulação para enviar uma transação utilizando o método 1 em segundos é calculado em função do número de blocos ($N_C = n^o$ de blocos de 4KB) de acordo com a Equação 5.1, que é obtida por interpolação linear dos pontos da Tabela 5.1, onde $t_{ref} \pm \alpha \approx 0.82s$ é aproximadamente o tempo para enviar a transação com o método 2, tal que α é um número muito pequeno.

$$t(N_C) \approx t_{ref} \pm \alpha + 0,00325N_C \tag{5.1}$$

É importante notar que esta relação é baseada na configuração do computador utilizado nos experimentos e que a passagem por referência modifica o valor do endereço da memória que contém a transação, podendo em alguns casos ser uma fragilidade[1] deste método. A seguinte configuração foi utilizada para obter estes resultados: um computador Intel Pentium (R) CPU G630 com 3.6GB de memória RAM com um sistema operacional CentOS 7 de 64-bits.

5.2 Validação funcional de um conversor de cores

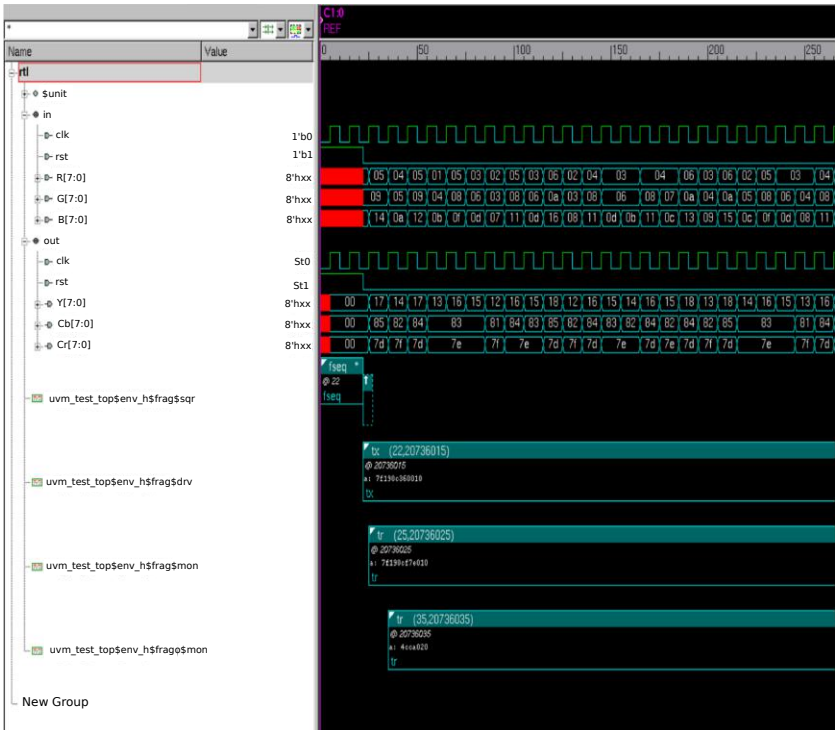
Uma das maneiras de se observar os sinais de um projeto em Verilog ou SystemVerilog é utilizando a declaração `$display` para observar os sinais em uma saída textual. Outra maneira de visualizar os sinais nos módulos é instruir o simulador de HDL a escrever cada transição dos sinais observados em um arquivo. O padrão para este tipo de arquivo é chamado de VCD (do inglês: *Value Change Dump*). Por este arquivo tender a ficar muito grande rapidamente, a Synopsys criou um formato proprietário que comprime um arquivo

[1]Em SystemVerilog uma variável declarada como **const ref** permite que o valor seja lido, mas não modificado.

VCD em um formato binário chamado de VPD (do inglês: *VCD Plus*), que pode visualizar os sinais com a ferramenta DVE [69].

Na Figura 5.1, os sinais das interfaces RGB e YC_bC_r pode ser vistos na ferramenta DVE. Note que os pixéis R, G e B estão codificados com 8 bits em hexadecimal. Veja ainda na Figura 5.1 que em aproximadamente 25 unidades de tempo simulados os valores de R, G e B são, respectivamente, 5, 9 e 20 em decimal. De acordo a Equação 4.6, estes sinais produzem três pixéis de saída Y, C_b e C_r. Como exemplo, a luminância Y é calculada como sendo aproximadamente $Y = 16 + (66 \times 5 + 129 \times 9 + 25 \times 20)/256 = 23,78 \approx 23$, que em hexadecimal é 14 de acordo com o gráfico da figura.

Figura 5.1: Exemplo de saída da plataforma de projeto e validação: visualização da variação temporal dos sinais das interfaces RGB e YC_bC_r e das transações de imagens no software DVE



Fonte: Elaborada pelo autor

5.3 Validação funcional de um detector de faces

Para avaliar a performance do modelo em validação, que é a base do algoritmo de Viola Jones implementado em [70], foram realizados alguns testes com a base de dados BIOID

[71], composta por 1521 imagens monocromáticas com resolução de $384 \times 286 \times 1$ pixéis[2]. O algoritmo de Vila Jones da biblioteca OpenCV foi utilizado como modelo de referência, apresentando 296 falsos positivos e 40 falsos negativos quando executado com a base de dados BIOID, como pode ser visto na Tabela 5.2. Em contraste, para a mesma base de dados, o modelo em validação cometeu 2 falsos positivos e 417 falsos negativos. Para avaliar o critério de *match*, o comparador do ambiente UVM computou duas métricas. A primeira métrica que é proposta neste trabalho, é normalizada com a média geométrica[3] das áreas dos retângulos que definem as regiões das faces enquanto que a outra métrica é o coeficiente de *dice*[4]. Para computar um *match*, o percentual de ambas as métricas deve ser de pelo menos 75%. De acordo com a Tabela 5.2, utilizando o coeficiente da métrica geométrica o comparador avaliou 38 *matches* com 83,87% de similaridade e 1064 *matches* com 93,74%. Utilizando o coeficiente de *dice* os resultados foram próximos. O número de *mismatches* usando o coeficiente da média geométrica foi de 419 com 4,93% de similaridade. Note que, este é o número total de falsos negativos em ambos os modelos de referência e em validação. No entanto, usando o coeficiente de *dice*, o comparador realizou 2 *mismatches* adicionais com porcentagem de 74%, uma vez, que o limiar estabelecido para um *match* é de 75%.

Uma investigação foi feita no modelo em validação para analisar a discrepância dos resultados em relação ao modelo de referência. Notou-se que o fator de escala do algoritmo de Viola Jones [8] do modelo em validação *model under validation* foi configurado em 20% enquanto que o modelo de referência tinha ajustado este valor para um fator de 10%. Este problema foi resolvido igualando-se o fator de escala do modelo de referência com o modelo de validação (ambos iguais a 10%) e o modelo em validação aumentou o número de falsos

[2]A validação dos resultados fica condicionada ao conjunto de imagens de cada base esecolhida. A escolha desta base se deu pelo fato de conter um conjunto expressivo de imagens (como especificação de projeto eram necessárias no mínimo mil amostras de faces), que era mais importante do que avaliar um conjunto menor de imagens coloridas de alta resolução.

[3]Dados dois sistemas que produzem duas regiões retangulares que contém uma face, o cálculo do percentual de similaridade entre as faces é obtido como sendo a razão entre a área de interseção dos dois retângulos que contém as faces e a média geométrica da área dos dois retângulos.

[4]Dados dois sistemas que produzem duas regiões retangulares que contém uma face, os coeficientes da média geométrica e de dice são obtidos aplicando-se a equações 4.9 (esta equação é da forma $\frac{2m}{l_1^2+l_2^2}$, onde m seria a área da interseção entre as duas regiões retangulares que contém as faces e l_1 e l_2 seriam as áreas dos quadrados que contém cada face) e 3.8 (de maneira análoga, esta equação é da forma $\frac{m}{l_1 l_2}$) nas coordenadas dos retângulos produzidos pelos dois modelos detectores de face. Enquanto que para o cômputo do coeficiente de *dice* são necessárias duas multiplicacões, o coeficiente da média geométrica é obtido com apenas uma, sendo portanto mais eficiente.

positivos para 5 e reduziu o número de falsos negativos para 304 como pode ser visto na Tabela 5.3. Depois do ajuste de escala, o comparador avaliou 36 *matches* com percentual de similaridade de 83,06% e 1179 *matches* com percentual de 92,84% usando o coeficiente da média geométrica. Usando o coeficiente de *dice* estes resultados foram próximos. O número de *mismatches* utilizando as duas métricas foi de 306 com percentual de similaridade de 4,89%.

Tabela 5.2: Taxas de detecção de faces para a base de dados BIOID [71] (1521 imagens com resolução de $384 \times 286 \times 1$ pixéis)

Modelo de Referência		**Modelo em Validação**	
Falsos Positivos		**Falsos Positivos**	
frequência	n^o de falsos	frequência	n^o de falsos
1225	0	1519	0
269	1	2	1
25	2	–	–
2	3	–	–
Falsos Negativos		**Falsos Negativos**	
frequência	n^o de falsos	frequência	n^o de falsos
1481	0	1104	0
40	1	417	1
Percentual de *Match*			
coeficiente da média geométrica		**coeficiente de *dice***	
n^o de *matches*	percentual	n^o de *matches*	percentual
38	83,87%	42	83,87%
1064	93,74 %	1058	93,73%
n^o de *mismatches*	percentual	n^o de *mismatches*	percentual
–	–	2	74,00%
419	4,93%	419	4,93%

Depois de outra extensa análise no modelo de validação, verificou-se que a expressão final de comparação do algoritmo de Viola Jones (vide expressão 3.18) não estava codificada corretamente no algoritmo, pois a expressão diz que o somatório dos pesos das características em cada estágio deve ser menor que um limiar normalizado pela variância da integral da imagem. No algoritmo do modelo em validação este limiar estava sendo multiplicado por

$0,4$ (para verificar este detalhe o modelo em validação foi inspecionado por módulos, tendo sido realizada isoladamente a verificação funcional de cada módulo do algoritmo de Viola Jones, tais como a verificação de *overflow* na integral da imagem ou o número de pixéis que a janela de convolução desloca a cada iteração na busca por características) de acordo com o Código 5.1. Por causa disso, as taxas de detecção de faces no modelo SystemC estavam tendo 304 falsos negativos, enquanto que o modelo do OpenCV tinha apenas 40 falsos. Ao ser feita a correção, removendo-se esse $0,4$ da multiplicação pelo limiar e tornando a expressão mais coerente com o seu modelo matemático, o número de falsos negativos reduziu para 37. Assim, a estrutura de projeto e validação mostrou sua funcionalidade executando uma avaliação automática em ambos os modelos.

Tabela 5.3: Taxas de detecção de faces para a base de dados BIOID [71] (1521 imagens com resolução de $384 \times 286 \times 1$ pixéis) depois de ajustar o fator de escala

Modelo de Referência		**Modelo em Validação**	
Falsos Positivos		**Falsos Positivos**	
frequência	n^o de falsos	frequência	n^o de falsos
1225	0	1516	0
269	1	5	1
25	2	–	–
2	3	–	–
Falsos Negativos		**Falsos Negativos**	
frequência	n^o de falsos	frequência	n^o de falsos
1481	0	1217	0
40	1	304	1
Percentual de *Match*			
coeficiente da média geométrica		**coeficiente de *dice***	
n^o de *matches*	percentual	n^o de *matches*	percentual
36	83,06%	43	83,06%
1179	92,84%	1172	92,84%
n^o de *mismatches*	percentual	n^o de *mismatches*	percentual
306	4,89%	306	4,89%

Código 5.1: Expressão de comparação final no algoritmo de Viola Jones

```
if( stage_sum < 0.4*stages_thresh_array[i] ){
      return -i;
} /* end of the per-stage thresholding */
```

5.4 Plataforma para prototipação em FPGA

Para ilustrar a funcionalidade da plataforma de prototipação em FPGA, o filtro de Sobel da Figura 4.10 foi implementado e sintetizado em uma placa de desenvolvimento DE1-SoC e um relatório de síntese foi gerado utilizando o software Quartus Prime da Altera[5] [72]. De acordo com a Tabela 5.4, o módulo de processamento de vídeo com resolução de 800×600 pixéis ocupou 2% dos elementos lógicos ALMs[6] da FPGA, além de ocupar 6 blocos de processamento digital de sinais (DSP, do inglês: *Digital Signal Processor*) para realização de operações de multiplicação e um elemento PLL (do inglês, *Phase-Locked Loop*[7]) para geração de sinais de relógio com frequências múltiplas.

Tabela 5.4: Resultado da Síntese em FPGA

Plataforma de Desenvolvimento	**Kit DE1-SoC**
Elementos Lógicos Adaptativos (em ALMs)	705/32070 (2%)
Total de Registradores	1414
Total de Pinos	241/457 (53%)
Total de bits de blocos de memória	61024/4065280 (2%)
Total de blocos de DSP	6/87 (7%)
Total de PLLs	1/6 (17%)

5.5 Conclusões

Neste capítulo foram apresentados os resultados da estrutura proposta no capítulo anterior. Na seção 5.1 foram ressaltados os resultados do desempenho obtido com a técnica de extensão do tamanho máximo do pacote de dados por transação utilizando

[5]O resultado da síntese é obtido a partir da compilação de um projeto no Quartus Prime. Este valores variam de acordo com o dispostivio de FPGA em uso. Neste caso foi utilizada uma matriz de células 5CSEMA5F31C6 da família Cyclone V. O relatório é gerado pelo sintetizador do Quartus a partir do mapeamento a descrição do *hardware* nas células da FPGA.

[6]Para maiores informações sobre os elementos lógicos adaptativos, consulte o Apêndice A.

[7]Uma descrição sobre um circuito PLL encontra-se no Apêndice B.

UVM Connect. Na seção 5.2 foram discutidos os resultados da da estrutura de validação de um conversor de cores de RGB para YC_bC_r. Na seção 5.3 foram apresentados os resultados do projeto e validação de um sistema de detecção de faces utilizando o algoritmo de Viola Jones com as métricas de avaliação propostas no capítulo anterior. Em último momento, na seção 5.4 foram discutidos os resultados da plataforma de prototipação em FPGA de aplicações de visão computacional.

Capítulo 6

Conclusões

Neste capítulo são sumarizados os objetivos desta dissertação e discutidos os resultados obtidos no capítulo anterior, além de serem enfatizadas algumas sugestões de trabalhos futuros. Em particular, na seção 6.1 são sumarizadas as contribuições da dissertação e, finalmente, na seção 6.2 é realizada a conclusão da dissertação com possíveis sugestões de trabalhos futuros.

6.1 Sumário de contribuições

Neste trabalho foi apresentada uma plataforma para projeto de *hardware* e validação de sistemas de detecção de faces. A estrutura é adequada para aplicações com ênfase em processamento de imagem. Como decisão de projeto, foi utilizada a biblioteca UVM Connect, que permitiu incorporar o legado das classes SystemC/TLM em ambientes UVM. No entanto, essa escolha apresentava a limitação de definir transações UVM com tamanho de até 4KB. As técnicas tradicionais de validação de sistemas de processamento de imagens são baseadas na fragmentação da imagem em pequenos pedaços. No entanto, isso aumenta o tempo de execução da simulação, dependendo do tamanho dos dados a serem transmitidos. A abordagem proposta neste trabalho apenas transmite os dados por referência a um endereço de memória. A abordagem proposta alcançou um tempo quase constante para simular transações de grandes volumes de dados e é independente do tamanho da imagem. Isto aumenta significativamente o desempenho da simulação quando

uma grande quantidade de dados está sendo transmitida. Os experimentos mostraram que a abordagem proposta realizou a transmissão de uma imagem dentro de um ambiente de verificação cerca de 33 vezes mais rápido para uma imagem de 40000KB em comparação com a abordagem anterior apresentada na literatura.

Outro problema no fluxo de projeto ocorria quando os modelos em nível RTL ainda não estavam disponíveis nas etapas iniciais do fluxo. Para tanto, a plataforma foi adaptada para realizar a validação de modelos em níveis de sistemas ESL. Além disso, usando o sistema de validação proposto para validar um sistema de detecção de faces, foram definidas métricas de casamento de retângulos para avaliar a implementação uma implementação do algoritmo de Viola Jones em SystemC/TLM em função de um modelo de referência escrito com funções da biblioteca OpenCV. Essa validação em nível de sistema reduziu significativamente o tempo necessário para decidir se o modelo de linha de base é válido ou não.

6.2 Perspectivas de trabalhos futuros

A validação do sistema de detecção de faces utilizando o algoritmo de Viola Jones foi realizada em nível TLM. Além do mais, para que uma verificação funcional seja completa, tanto em nível ESL quanto em RTL, todos os casos de testes devem ser cobertos. Em muitos casos práticos isso não é possível e a cobertura funcional é uma métrica ótima que atinge os casos importantes durante os planos de testes. Por fim, o projeto de circuitos integrados devem ser projetados com otimização de consumo de energia. Por estas razões, os seguintes tópicos são sugestões de trabalhos futuros:

- a validação de um sistema de detecção de faces em nível RTL, onde o fluxo de projeto seria adaptado de acordo com a Figura 4.2.(b) e transações geradas pelo modelo de referência do detector de faces seriam comparadas com sinais da interface de saída do modelo RTL do algoritmo de Viola Jones;

- uma técnica cobertura funcional (para cobrir casos de testes em parte direcionados e em parte aleatórios) com geração pseudo-aleatórias de faces (uma sugestão seria criar uma base de dados de faces com diferentes cenários utilizando inteligência artificial a partir do treinamento de bases de faces existentes) para a verificação funcional de

circuitos digitais de detecção de faces;

- adicionar ao ambiente de projeto e validação critérios de baixo consumo de energia (com a metodologia de projeto e validação proposta nesta dissertação as restrições de minimização de projeto alcançadas com a descrição em *hardware* são ganho em performance computacional, peso e tamanho) utilizando a metodologia UPF.

Referências bibliográficas

1 XU, Zhengya; WU, Hong Ren. Smart video surveillance system. In: IEEE. *Industrial Technology (ICIT), 2010 IEEE International Conference on*. Vina del Mar, Chile, 2010. p. 285–290.

2 VIOLA, Paul; JONES, Michael J; SNOW, Daniel. Detecting pedestrians using patterns of motion and appearance. *International Journal of Computer Vision*, Springer, v. 63, n. 2, p. 153–161, 2005.

3 WECHSLER, Harry; PHILLIPS, Jonathon P; BRUCE, Vicki; SOULIE, Francoise Fogelman; HUANG, Thomas S. *Face recognition: From theory to applications*. Stirling, Scotland, UK: Springer Science & Business Media, 2012. v. 163.

4 ZHAO, Wenyi; CHELLAPPA, Rama; PHILLIPS, P Jonathon; ROSENFELD, Azriel. Face recognition: A literature survey. *ACM computing surveys (CSUR)*, ACM, v. 35, n. 4, p. 399–458, 2003.

5 FEFAUD, R; BERNIER, OJ; VIALLET, JE; COLLOBERT, M. A fast and accurate face detection based on neural network. *IEEE Transactions on PAMI*, v. 23, n. 1, p. 42–53.

6 OSUNA, Edgar; FREUND, Robert; GIROSIT, Federico. Training support vector machines: an application to face detection. In: IEEE. *Computer vision and pattern recognition, 1997. Proceedings., 1997 IEEE computer society conference on*. San Juan, Puerto Rico, USA, 1997. p. 130–136.

7 CORCORAN, Peter M; IANCU, Claudia. Automatic face recognition system for hidden markov model techniques. In: *New Approaches to Characterization and Recognition of Faces*. Open Access: InTech, 2011.

8 VIOLA, Paul; JONES, Michael J. Robust Real-Time Face Detection. *International Journal of Computer Vision*, Springer, v. 57, n. 2, p. 137–154, 2004.

9 BERGERON, Janick. *Writing testbenches: functional verification of HDL models*. New York, NY, USA: Springer Science & Business Media, 2012.

10 FLEMSTRÖM, Daniel; SUNDMARK, Daniel; AFZAL, Wasif. Vertical test reuse for embedded systems: A systematic mapping study. In: IEEE. *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*. Funchal, Portugal, 2015. p. 317–324.

11 CAPELLE, F. *Design and validation of a face recognition framework*. Dissertação (Mestrado) — University of Twente, 2013.

12 SILVA, Karina RG. *Uma metodologia de Verificação Funcional para Circuitos Digitais.(2007), 119 p.* Tese (Doutorado) — Universidade Federal de Campina Grande, 2007.

13 SILVA, Karina RG Da; MELCHER, Elmar UK; MAIA, Isaac; CUNHA, Henrique do N. A methodology aimed at better integration of functional verification and rtl design. *Design Automation for Embedded Systems*, Springer, v. 10, n. 4, p. 285–298, 2005.

14 RODRIGUES, Cássio L; SILVAY, Karina RG da; MELCHERZ, Elmar; FIGUEIREDOZ, Jorge CA de; GUERREROZ, Dalton DS. Refactoring verisc testbenches to improve the functional verification during the integration phase. In: IEEE. *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society*. Melbourne, VIC, Australia, 2011. p. 2820–2825.

15 ACCELLERA, Universal Verification Methodology. *Universal Verification Methodology I. 0 User's Guide*. 2011.

16 SANTOS, Lucana; GÓMEZ, Ana; HERNÁNDEZ-FERNÁNDEZ, Pedro; SARMIENTO, Roberto. Systemc modelling of lossless compression ip cores for space applications. In: IEEE. *Design and Architectures for Signal and Image Processing (DASIP), 2016 Conference on*. Rennes, France, 2016. p. 65–72.

17 ERICKSON, Adam. Introducing uvm connect. *Mentor Graphics Verif. Horiz*, v. 8, n. 1, p. 6–12, 2012.

18 MODI, Dhaval; SITAPARA, Harsh; SHAH, Rahul; MEHUL, Ekata; ENGINEER, Pinal. Integrating matlab with verification hdls for functional verification of image and video processing asic. *International Journal of Computer Science & Emerging Technologies*, v. 2, n. 2, p. 258–265, 2011.

19 HAYASHI, Yoshihiko; IKUMA, Noriyuki; KAWAMURA, Taku; TOKUE, Takashi. Verification of system lsis for image processing. *FUJITSU Sci. Tech. J*, v. 49, n. 1, p. 124–130, 2013.

20 BECVAR, Milos; TUMBUSH, Greg. *Design and Verification of an Image Processing CPU using UVM*. San Jose, CA, USA: DVCon, 2013.

21 JAIN, Abhishek; GUPTA, Piyush Kumar; GUPTA, Dr; DHAR, Sachish et al. Accelerating systemverilog uvm based vip to improve methodology for verification of image signal processing designs using hw emulator. *arXiv preprint ar Xiv:1401.3554*, 2014.

22 MARCONI, Sara; CONTI, Elia; CHRISTIANSEN, Jorgen; PLACIDI, Pisana. Reusable systemverilog-uvm design framework with constrained stimuli modeling for high energy physics applications. In: IEEE. *Systems Engineering (ISSE), 2015 IEEE International Symposium on*. Rome, Italy, 2015. p. 391–397.

23 JAIN, Abhishek; GUPTA, Richa. Scaling the uvm_reg model towards automation and simplicity of use. In: IEEE. *VLSI Design (VLSID), 2015 28th International Conference on*. Bangalore, India, 2015. p. 164–169.

24 JAIN, Abhishek; GUPTA, Richa. Empirical study of identifying gaps between users expectation and experience in the verification methodologies of semiconductor industry. In: IEEE. *Signal Processing and Communication (ICSC), 2016 International Conference on*. Noida, India, 2016. p. 464–469.

25 JAIN, Abhishek; GUPTA, Richa. Unified and modular modeling and functional verification framework of real-time image signal processors. *VLSI Design*, Hindawi Publishing Corporation, v. 2016, 2016.

26 JAIN, Abhishek; GUPTA, Dr; JANA, Sandeep; KUMAR, Krishna et al. Early development of uvm based verification environment of image signal processing designs using tlm reference model of rtl. *arXiv preprint arXiv:1408.1150*, 2014.

27 MEFENZA, Michael; YONGA, Franck; SALDANHA, Luca B; BOBDA, Christophe; VELIPASSALAR, Senem. A framework for rapid prototyping of embedded vision applications. In: IEEE. *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*. Madrid, Spain, 2014. p. 1–8.

28 GONZALEZ, Rafael C; WOODS, Richard E. *Processamento Digital de Imagens*. São Paulo, Brasil: Pearson Prentice Hall, 2010.

29 PEREIRA, Fernando; FERREIRA, Aníbal; SALEMA, Carlos; TRANCOSO, Isabel; CORREIA, Paulo; ASSUNÇÃO, Pedro; FARIA, Sérgio. *Comunicações Audiovisuais: Tecnologias, Normas e Aplicações*. Lisboa, Portugal: IST Press, 2009.

30 NEELAMANI, Ramesh; QUEIROZ, Ricardo De; FAN, Zhigang; DASH, Sanjeeb; BARANIUK, Richard G. JPEG Compression History Estimation for Color Images. *IEEE Transactions on Image Processing*, IEEE, v. 15, n. 6, p. 1365–1378, 2006.

31 TRUCCO, Emanuele; VERRI, Alessandro. *Introductory Techniques for 3-D Computer Vision*. Upper Saddle River, NJ, USA: Prentice Hall, 1998. v. 201.

32 FORSYTH, David; PONCE, Jean. *Computer Vision: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall, 2011.

33 HARTLEY, Richard; ZISSERMAN, Andrew. *Multiple view geometry in computer vision*. Cambridge, UK: Cambridge university press, 2003.

34 FAUGERAS, Olivier. *Three-dimensional Computer Vision: A Geometric Viewpoint*. Cambridge, MA, USA: MIT press, 1993.

35 BRADSKI, Gary; KAEHLER, Adrian. *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol, CA, USA: "O'Reilly Media, Inc.", 2008.

36 JAIN, Ramesh; KASTURI, Rangachar; SCHUNCK, Brian G. *Machine Vision*. New York, NY, USA: McGraw-Hill New York, 1995. v. 5.

37 CROW, Franklin C. Summed-Area Tables for Texture Mapping. *ACM SIGGRAPH Computer Graphics*, ACM, v. 18, n. 3, p. 207–212, 1984.

38 DOERMANN, David; MIHALCIK, David. Tools and techniques for video performance evaluation. In: IEEE. *Pattern Recognition, 2000. Proceedings. 15th International Conference on*. Barcelona, Spain, 2000. v. 4, p. 167–170.

39 DATTA, Asit Kumar; DATTA, Madhura; BANERJEE, Pradipta Kumar. *Face Detection and Recognition: Theory and Practice*. Boca Raton, FL, USA: CRC Press, 2015.

40 RABINER, Lawrence R; JUANG, Biing-Hwang. Fundamentals of speech recognition. PTR Prentice Hall, 1993.

41 NEFIAN, Ara V; HAYES, Monson H. Face detection and recognition using hidden markov models. In: IEEE. *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*. Chicago, IL, USA, 1998. v. 1, p. 141–145.

42 DONY, R. The Transform and Data Compression Handbook. *Karhunen-Loève Transform. CRC Press LLC, Boca Raton*, 2001.

43 VAPNIK, Vladimir Naumovich; VAPNIK, Vlamimir. *Statistical learning theory*. New York, NY, USA: Wiley New York, 1998. v. 1.

44 HEISELE, Bernd; PONTIL, Massimilinao et al. *Face detection in still gray images*. Cambridge, MA, USA, 2000.

45 MCCULLOCH, Warren S; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

46 HUBEL, David H; WIESEL, Torsten N. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, Wiley Online Library, v. 195, n. 1, p. 215–243, 1968.

47 LU, Le; ZHENG, Yefeng; CARNEIRO, Gustavo; YANG, Lin. *Deep Learning and Convolutional Neural Networks for Medical Image Computing: Precision Medicine, High Performance and Large-Scale Datasets*. Cham, Switzerland: Springer, 2017.

48 LI, Junjie; KARMOSHI, Saleem; ZHU, Ming. Unconstrained Face Detection Based on Cascaded Convolutional Neural Networks in Surveillance Video. In: IEEE. *Image, Vision and Computing (ICIVC), 2017 2nd International Conference on*. Chengdu, China, 2017. p. 46–52.

49 PAPAGEORGIOU, Constantine P; OREN, Michael; POGGIO, Tomaso. A general framework for object detection. In: IEEE. *Computer vision, 1998. sixth international conference on*. Bombay, India, 1998. p. 555–562.

50 SCHAPIRE, Robert E; FREUND, Yoav; BARTLETT, Peter; LEE, Wee Sun et al. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, Institute of Mathematical Statistics, v. 26, n. 5, p. 1651–1686, 1998.

51 LIENHART, Rainer; MAYDT, Jochen. An extended set of haar-like features for rapid object detection. In: IEEE. *Image Processing. 2002. Proceedings. 2002 International Conference on*. Rochester, NY, USA, 2002. v. 1, p. I–900.

52 ACASANDREI, Laurentiu; BARRIGA, Angel. Accelerating viola-jones face detection for embedded and soc environments. In: INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. Ghent, Belgium, 2011.

53 OPENCV library. <www.opencv.org>. Acessado em 12/02/2016.

54 CAMPOS, Nelson; COSTA, Roberto; COSTA, Elton; JUNIOR, Gutemberg; MELCHER, Elmar. *A 4-MHz parameterized Logarithm-Square Root IP-Core*. Grenoble, France: Design and Reuse IPSOC, 2016.

55 ROWLEY, Henry A; BALUJA, Shumeet; KANADE, Takeo. Neural network-based face detection. *IEEE Transactions on pattern analysis and machine intelligence*, IEEE, v. 20, n. 1, p. 23–38, 1998.

56 FLYNN, David; AITKEN, Rob; GIBBONS, Alan; SHI, Kaijian. *Low power methodology manual: for system-on-chip design*. Boston, MA, USA: Springer Science & Business Media, 2007.

57 BEMBARON, Freddy; KAKKAR, Sachin; MUKHERJEE, Rudra; SRIVASTAVA, Amit. Low power verification methodology using upf. *Proc. DVCon*, p. 228–233, 2009.

58 BLACK, David C; DONOVAN, Jack; BUNTON, Bill; KEIST, Anna. *SystemC: From the ground up*. New York, NY, USA: Springer Science & Business Media, 2011. v. 71.

59 CAI, Lukai; GAJSKI, Daniel. Transaction level modeling: an overview. In: ACM. *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Newport Beach, CA, USA, 2003. p. 19–24.

60 GHENASSIA, Frank et al. *Transaction-level modeling with SystemC*. New York, NY, USA: Springer, 2005.

61 KEDING, Holger; WILLEMS, Markus; COORS, Martin; MEYR, Heinrich. Fridge: a fixed-point design and simulation environment. In: IEEE COMPUTER SOCIETY. *Proceedings of the conference on Design, automation and test in Europe*. Paris, France, 1998. p. 429–435.

62 SUTHERLAND, Stuart; MOORBY, P; DAVIDMANN, Simon; FLAKE, Peter. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. New York, NY, USA: Springer Science & Business Media, 2006.

63 CAMPOS, Nelson C. S; MONTEIRO, Heron A.; BRITO, Alisson V.; N., Lima Antonio M.; MELCHER, Elmar U. K.; MORAIS, Marcos R. A. *A Framework for Design and Validation of Face Detection Systems*. Pucón, Chile: CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), 2017.

64 FAST packer converters. <https://verificationacademy.com/verification-methodology-reference/uvmc-2.3/docs/html/files/examples/xlerate-connections/README-txt.html>. Acessado em 12/02/2016.

65 RUDERMAN, Daniel L. The statistics of natural images. *Network: computation in neural systems*, Taylor & Francis, v. 5, n. 4, p. 517–548, 1994.

66 KLETTE, Reinhard. *Concise computer vision*. Auckland, New Zealand: Springer, 2014.

67 MONTEIRO, Heron A.; CAMPOS, Nelson C. S.; OLIVEIRA, Jozias P.; LIMA, Antonio Marcus N.; BRITO, Alisson V.; MELCHER, Elmar U. K. *Energy Consumption Measurement of a FPGA Full-HD Video Processing Platform*. Fortaleza, Brasil: WCAS 2017 - Workshop on Circuits and System Design, 2017.

68 TERASIC DE1-SoC User Manual. <https://courses.cs.washington.edu/courses/cse467/15wi/docs/DE1_SoC_User_Manual.pdf>. Acessado em 28/07/2017.

69 SYNOPSYS VCS. <https://www.synopsys.com/verification/simulation/vcs.html>. Acessado em 23/08/2017.

70 F. Comaschi. <https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection>. Acessado em 28/05/2017.

71 BIOID Face Database - FaceDB. <https://www.bioid.com/About/BioID-Face-Database>. Acessado em 28/05/2017.

72 INTEL Quartus Prime Design Software Overview. <https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html>. Acessado em 28/05/2017.

73 STRATIX, II. Device handbook. *Altera corporation*, San Jose, CA, USA, v. 2610, p. 95134–2020, 2014.

74 SALOMON, David. *Data compression: the complete reference*. New York, NY, USA: Springer Science & Business Media, 2004.

75 MENARD, Daniel; CHILLET, Daniel; SENTIEYS, Olivier. Floating-to-fixed-point conversion for digital signal processors. *EURASIP journal on applied signal processing*, Hindawi Publishing Corp., v. 2006, p. 77–77, 2006.

76 CONSTANTINIDES, George A; CHEUNG, Peter YK; LUK, Wayne. Truncation noise in fixed-point sfgs [digital filters]. *Electronics Letters*, IET, v. 35, n. 23, p. 2012–2014, 1999.

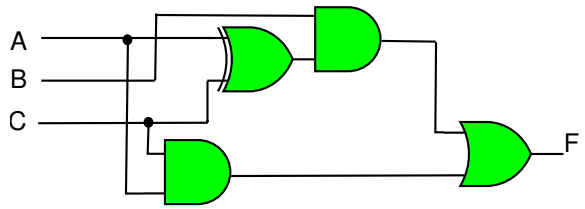
Apêndice A

Arquitetura de uma FPGA

A arquitetura de uma FPGA é composta células lógicas, matrizes de interconexões entre células lógicas e elementos de entrada e saída.

Essas células lógicas são unidades básicas que são compostas por tabelas LUT (do inglês: *Look-up Table*) que formam funções lógicas a partir de um conjunto de bits armazenados na entrada. Para ilustrar com um exemplo, o circuito lógico da Figura A.1 tem a tabela verdade descrita pela Tabela A.1.

Figura A.1: Circuito lógico combinacional: $F(A, B, C) = or(and(xor(A, C), B), and(A, C))$



Fonte: Elaborada pelo autor

Para implementar o mesmo circuito lógico da Figura A.1, a LUT da Figura A.2 pode ser utilizada. Para reproduzir a mesma lógica basta configurar os valores de entrada I_k, para $k = 7, 6, 5, ..., 0$, como sendo iguais ao valor F da Tabela A.1.

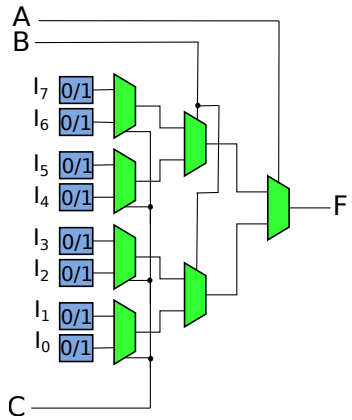
A Altera foi pioneira na introdução de LUTs com 8 entradas na família de dispostivos Stratrix II em 2004. A arquitetura das células lógicas das FPGAs da Altera são definidas como sendo elementos lógicos adaptativos (ALM, do inglês: *Adaptive Logic Element*). O diagrama de circuito de um ALM pode ser visto na Figura A.3. Com um bloco ALM é ser

possível implementar, por exemplo, funções com uma LUT de 6 entradas ou 2 LUTs de 4 entradas, além de implementar circuitos sequenciais com os registradores inclusos na célula ALM[1].

Tabela A.1: Tabela verdade para a função da Figura A.1

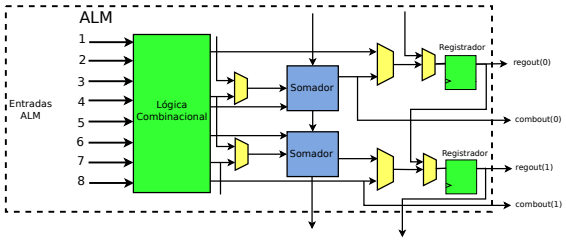
A	**B**	**C**	**F**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figura A.2: Uma LUT de três entradas



Fonte: Elaborada pelo autor

Figura A.3: Uma LUT de três entradas



Fonte: Adaptada de [73]

[1]Para maiores informações, consulte [73].

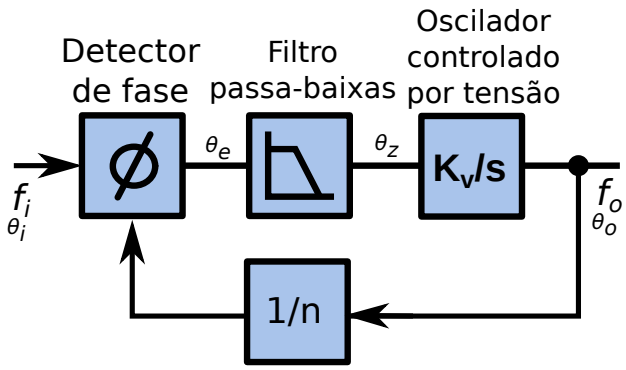
Apêndice B

Circuito PLL

Um circuito PLL (do inglês, *Phase-Locked Loop*) é um sistema de controle de frequência em malha fechada em que o sinal de realimentação é utilizado para sincronizar a frequência do sinal de saída com a frequência do sinal de entrada. Dentre as as aplicações deste circuito pode-se citar, por exemplo, sistemas de comunicações e microeletrônica, onde o módulo PLL pode ser utilizado para multiplicar as frequências de sinais de relógio por múltiplos inteiros.

O circuito é composto por um detector de fase, um filtro passa baixas, um oscilador controlado por tesnsão e um divisor, como pode ser visto na Figura B.1.

Figura B.1: Diagrama de blocos de um circuito PLL



Fonte: Elaborada pelo autor

O detector de fase compara o sinal de entrada (com frequência f_i) com o sinal realimentado (com frequência f_r), produzindo um erro de fase θ_e que é proporcional a diferença de fase entre entre esses dois sinais. O filtro passa baixas produz uma tensão

contínua proporcional a diferença de fase entre ϕ_i e ϕ_r, eliminando as componentes de alta frequência.

O erro de fase $e(s)$ no domínio da frequência, onde $s = j\omega$, é dado pela Equação B.1.

$$e(s) = f_i(s) - \frac{f_o(s)}{n} \tag{B.1}$$

O sistema em malha fechada faz com que o erro $e(s)$ seja nulo em regime permanente. Sendo assim, se $e(s) = 0$ então $f_o(s) = nf_i(s)$, ou seja, a frequência de saída é um múltiplo inteiro da frequência de entrada.

Outra maneira de obter este resultado é por meio da função de transferência em malha fechada do sistema. Sendo $Z(s)$ a função de transferência do filtro passa baixas e $\frac{K_v}{s}$ a função de transferência do oscilador controlado por tensão, então a relação entre a saída $f_o(s)$ e a entrada $f_i(s)$ do sistema no domínio da frequência é dada pela Equação B.2.

$$\left(f_i(s) - \frac{f_o(s)}{n}\right) Z(s)\frac{K_v}{s} = f_o(s) \Rightarrow \frac{f_o(s)}{f_i(s)} = \frac{\frac{Z(s)K_v}{s}}{\frac{Z(s)K_v}{sn} + 1} \tag{B.2}$$

Para um valor ganho alto K_v, a expressão da Equação B.2 se torna $\frac{f_o(s)}{f_i(s)} = n$, o que dá o mesmo resultado pela obtido pela análise em regime permanente do erro de fase no domínio da frequência.

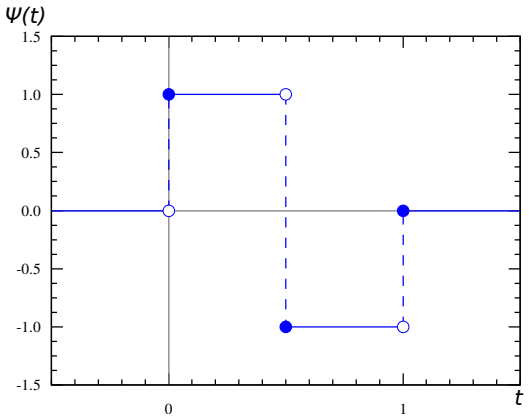
Apêndice C

Transformada de Haar

A Transformada de Haar é um transformada matemática discreta tendo como aplicações, por exemplo, processamento e análise de sinais, compressão de dados. Ela foi proposta em 1909 pelo matemático húngaro Alfred Haar. A transformada de Haar é um caso particular de transformada discreta de *wavelet*, onde o *wavelet* é um pulso quadrado definido pela Equação C.1, cujo gráfico está ilustrado na Figura C.1.

$$\Psi(t) = \begin{cases} 1, & 0 \leq t < 0,5 \\ -1, & 0,5 \leq t < 1 \\ 0, & \text{para outros valores de } t \end{cases} \qquad \text{(C.1)}$$

Figura C.1: Gráfico de uma onda Haar (*wavelet*)



Fonte: Elaborada pelo autor

A transformada de Haar[1] pode ser usada para representar uma função $f(t)$ de acordo com a Equação C.2, onde $\phi(t)$ é definido de acordo com a Equação C.3 e os parâmetros c_k e $d_{j,k}$ são constantes.

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} d_{j,k} \Psi(2^j t - k) \tag{C.2}$$

$$\phi(t) = \begin{cases} 1, & 0 \leq t \leq 1 \\ 0, & \text{para outros valores de } t \end{cases} \tag{C.3}$$

Como exemplo, a função definida pela Equação C.4 pode ser representada como $f(t) = 4\phi(t) + \Psi(t)$, onde $c_0 = 4$, $d_{0,0} = 1$ e $c_n = 0$ e $d_{n,m} = 0$ para $n \neq 0$ e $m \neq 0$.

$$(t) = \begin{cases} 5, & 0 \leq t < 0,5 \\ 3, & 0,5 \leq t < 1 \end{cases} \tag{C.4}$$

O algoritmo de Viola-Jones define um característica (*Haar-like feature*) como sendo a soma de uma região retângular definida por uma onda *wavelet*. Comparando as Figuras C.1 e 3.18, percebe-se que a região escura da característica corresponde aos valores positivos de $\Psi(t)$ e as regiões claras correspondem aos valores negativos de $\Psi(t)$. Como já foi mencionado na subseção 3.2.2, o cálculo dessas características é acelarado com a integral da imagem.

[1] Para se aprofundar no assunto, recomenda-se a leitura de [74].

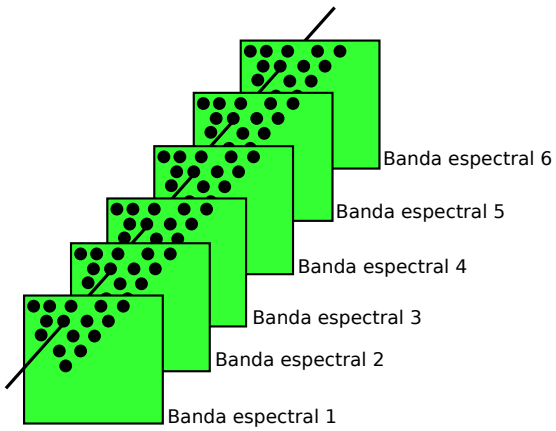
Apêndice D

Transformada de Karhunen-Loève

A transformada de Karhunen-Loève (ou transformada KLT) é uma transformada linear e reversível que remove a redundância da informação pela descorrelação dos dados [42]. Esta transformada tem aplicações em compressão de e codificação de vídeos e é definida a seguir.

Considere uma população de vetores x_i, onde $i = 0, 1, ..., n$. Cada vetor pode representar, por exemplo, a intensidade em escala de cinza de um pixel contido numa imagem i. Esta população pode ainda surgir a partir da combinação de pixéis contidos em diferentes imagens, como pode ser visto na Figura D.1.

Figura D.1: Formação de uma população de vetores a partir de pixéis contidos em diferentes imagens



Fonte: Elaborada pelo autor

A média m_x da população de vetores x_i é definida como de acordo com a Equação D.1, onde a operação $E[x_i]$ é o valor esperado de x_i.

$$m_x = E[x] = \begin{bmatrix} m_1 & m_2 & ... & m_n \end{bmatrix}^T = \begin{bmatrix} E[x_1] & E[x_2] & ... & E[x_n] \end{bmatrix}^T \tag{D.1}$$

A matriz de covariância C da população de vetores é definida de acordo com a Equação D.2.

$$C = E[(x - m_x)(x - m_x)^T] \tag{D.2}$$

Para uma população com M vetores, onde M é suficientemente grande, a média m_x pode ser aproximada de acordo com a Equação D.3.

$$m_x = \frac{1}{M} \sum_{k=1}^{M} x_k \tag{D.3}$$

Se a matriz C tem dimensão $n \times n$, então o escalar λ é dito um autovalor de C se existe um vetor não nulo e tal que $Ce = \lambda e$, onde o vetor e é o autovetor da matriz C que corresponde ao autovalor λ.

Seja A uma matriz cujas linhas sejam formadas pelos autovetores da matriz de covariância C da populĺação de vetores de x_i.

As linhas da matriz A estão ordenadas de tal forma que a primeira linha de A é o autovetor que corresponde ao maior autovalor e a última linha de A é o autovetor que corresponde ao menor autovalor.

A transformada de Karhunen-Loève é definida de acordo com a Equação D.4.

$$y = A(x - m_x) \tag{D.4}$$

A transformada KLT tem as seguintes propriedades:

- $E[y] = 0$
- $C_y = AC_xA^T = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \cdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$

Apêndice E

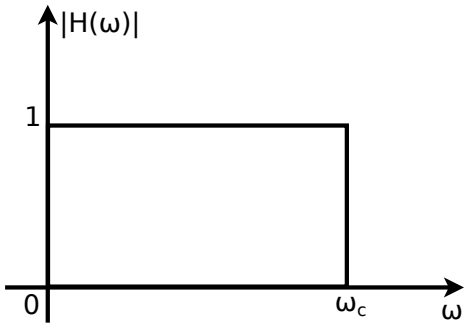
Projeto de filtros FIR em ponto fixo

As aplicações de processamento digital de sinais são especificadas com representação em ponto flutuante, mas estas operações são geralmente implementadas em processadores digitais de sinais que operam na representação em ponto fixo[1] [75].

Como exemplo de aplicação, o projeto de filtros digitais FIR (do inglês: *Finite Impulse Response*) em *hardware* consistem na determinação de coeficientes com representação em ponto flutuante. No entanto, uma implementação em *hardware* desses filtros digitais realizam a transformação desses coeficientes para números inteiros ou codificados com notação de ponto fixo.

Para ilustrar os conceitos mencionados anteriormente, considere o filtro passa baixas da Figura E.1.

Figura E.1: Módulo da função de transferência de um filtro passa baixas



Fonte: Elaborada pelo autor

[1] A Figura 3.24 ilustra as duas representações numéricas.

A função de transferência do filtro é dada pela Equação E.1, onde ω_c é a frequência de corte do filtro.

$$H(\omega) = \begin{cases} 1, & |\omega| < \omega_c \\ 0, & \text{caso contrário} \end{cases} \tag{E.1}$$

Aplicando-se a transformada inversa de tempo discreto de Fourier na Equação E.1, é obitda a resposta ao impulso em tempo discreto do filtro (de ordem N) $h_d(n)$, onde $n = kt$ para $k = 0, 1, 2, ..., N-1$.

$$h_d[n] = \frac{1}{2\pi} \int_{-\pi}^{+\pi} H(\omega) e^{j\omega n} d\omega \tag{E.2}$$

Aplicando-se a Equação E.1 na Equação E.2 a resposta ao impulso do filtro é dada por:

$$h_d[n] = \frac{1}{2\pi} \int_{-\omega_c}^{+\omega_c} e^{j\omega n} d\omega = \frac{sin(n\omega_c)}{n\pi} \tag{E.3}$$

A expressão geral de um filtro FIR de comprimento M (de ordem $N = M-1$) no domínio da Transformada Z é dada pela Equação E.4.

$$H(z) = \sum_{k=0}^{M-1} b_k z^{-k} \tag{E.4}$$

Para que o filtro seja causal e fisicamente realizável, é aplicado um deslocamento de $\frac{M-1}{2}$ amostras em $h_d[n]$, obtendo-se o filtro deslocado $h[n]$, onde $w[n]$ é uma janela retangular descrita pela Equação E.6.

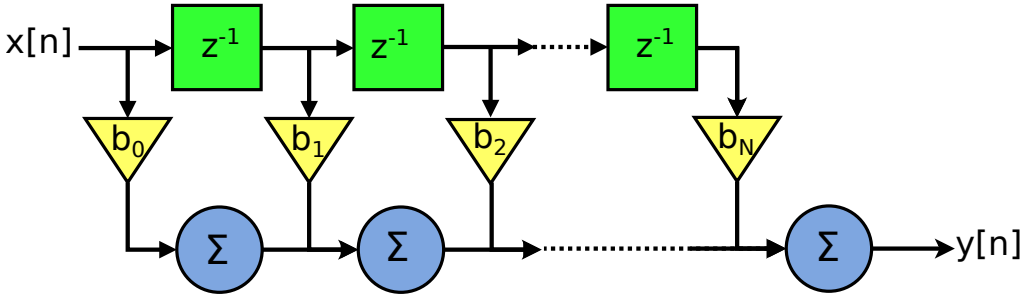
$$h[n] = h_d[n - \frac{M-1}{2}]w[n] \tag{E.5}$$

$$w[n] = \begin{cases} 1, & n = 0, 1, 2, ..., M-1 \\ 0, & \text{caso contrário} \end{cases} \tag{E.6}$$

A Figura E.2 ilustrar a arquitetura de um filtro FIR. Cada coeficiente $h[n]$ do filtro é

um número real representado em ponto flutuante. Para uma conversão desses coeficientes para ponto fixo com B bits de precisão, os coeficientes $h_q[n]$ são obtidos de acordo com a Equação E.7.

Figura E.2: Arquitetura de um filtro FIR



Fonte: Elaborada pelo autor

$$h_q[n] = \frac{\lfloor h[n]2^B \rfloor}{2^B} \tag{E.7}$$

Para efeitos práticos, o código G.6 implementado em C gera um filtro de ordem N em *hardware* (um módulo em Verilog).

Código E.1: Geração automática de filtros FIR em Verilog

```
#include <stdio.h>
#include <math.h>
#define pi acos(-1)

double sinc(double x){
  return (x == 0) ? 1 : sin(pi*x)/(pi*x);
}


int main(){
  FILE *file;
  file = fopen("fir.v", "w");

  int N, H , W, w_type;
  printf("Enter the order of the FIR Filter N: ");
  scanf("%d", &N);
  N = (N%2) ? N : N+1;
  printf("Enter the number of bits (H) of the FIR Filter coefficients h: ");
  scanf("%d", &H);
  printf("Enter the number of bits (W) of the input and output data: ");
  scanf("%d", &W);
  printf("Enter the window type"
         "\n0 for Rectangular Windowing"
         "\n1 for Hamming Windowing: ");
  scanf("%d", &w_type);
```

```
  fprintf(file, "parameter N = %d;\n"
                "parameter W = %d;\n"
                "parameter H = %d;\n"
                 "\nmodule fir_filter(input clock,"
                 "\n\t\t  input signed [W-1:0] Xin,"
                 "\n\t\t  output reg signed [W-1:0] Y);\n\n\n"
                 "\treg signed [H-1:0] h[N];"
                 "\n\treg signed [W+H-1:0] y[N];\n\n\n", N, W, H);
  int m = (N-1)/2;
  int h, h_rect;
  double hamming;
  for(int i = 0; i < N; i++){
    h_rect = pow(2,H)*0.5*sinc(0.5*(i-m));
    hamming = 0.54-0.46*cos(2*pi*i/(N-1));
    h = !w_type ? h_rect : h_rect*hamming;
    fprintf(file, "\tassign h[%d] = %d;\n", i, h);
  }
  fprintf(file, "\n\tassign Y = y[N-1]>>H;\n");
  fprintf(file, "\n\talways@ (posedge clock) begin\n"
                 "\t  y[0] <= h[0]*Xin;\n"
                 "\t  for(int i = 0; i < N; i++)\n"
                 "\t    y[i] <= y[i-1]+h[N-i-1]*Xin;"
                 "\n\tend");

  fprintf(file, "\n\nendmodule");
  fclose(file);
  return 0;
}
```

Apêndice F

Conversão de ponto flutuante para ponto fixo

A conversão de variáveis com representação infinita (ou na prática representações em ponto flutuante) para ponto fixo, que tem representação finita, gera erros de quantização que se propagam nas operações aritméticas efetuadas com as variáveis da nova representação numérica.

Em [76] é realizada uma análise desse erro de quantização considere uma variável com palavra definida por n_1 bits de comprimento e com p bits fracionários com representada pelo vetor (n_1, p). O erro de truncamento e de um sinal representado por (n_1, p) convertido para um sinal (n_2, p) na aritmética de complemento de 2 é dada pela Equação F.1.

$$-2^p(2^{-n_2} - 2^{-n_1}) \leq e \leq 0 \tag{F.1}$$

Partindo da premissa que o erro e tem distribuição uniforme de probabilidade. A média e a variância de e representadas em complemento de dois são dadas pelas equações F.2 e F.3, respectivamente.

$$m_q = -\frac{1}{2^{n_1-n_2}} \sum_{i=0}^{2^{n_1-n_2}-1} 2^{p-n_1} i = -2^{p-1}(2^{-n_2} - 2^{-n_1}) \tag{F.2}$$

$$\sigma_q^2 = \frac{1}{2^{n_1-n_2}} \sum_{i=0}^{2^{n_1-n_2}-1} (2^{p-n_1} i)^2 - m_q = \frac{1}{12} 2^{2p}(2^{-2n_2} - 2^{-2n_1}) \tag{F.3}$$

Note que para $n_1 >> n_2$ e para $p = 0$ a Equação F.3 é simplificada pela expressão da Equação F.4, que corresponde ao erro de variância no modelo contínuo.

$$\sigma_q^2 = \frac{1}{12} 2^{-2n_2} \tag{F.4}$$

Apêndice G

Códigos

G.1 Códigos da fundamentação teórica

Código G.1: Conversor de cores RGB para YC_bC_r em SystemVerilog

```
interface rgb_if(input logic clk, rst); //========= rgb_if.sv =========
    logic [7:0] R, G, B;
    modport in(input clk, rst, R, G, B);
endinterface: rgb_if

interface ycbcr_if(input clk, rst); //========= ycbcr_if.sv =========
    logic [7:0] Y, Cb, Cr;
    modport out(input clk, rst, output Y, Cb, Cr);
endinterface: ycbcr_if

module RGB2YCbCr(rgb_if.in rgb, ycbcr_if.out ycbcr); //========= RGB2YCbCr.sv =========
  always @(posedge rgb.clk)begin
    if(rgb.rst) begin
          ycbcr.Y <= 0;
          ycbcr.Cb <= 0;
          ycbcr.Cr <= 0;
       end
       else begin
          ycbcr.Y <= 16+(((rgb.R<<6)+(rgb.R<<1)
                         +(rgb.G<<7)+rgb.G
                         +(rgb.B<<4)+(rgb.B<<3)+rgb.B)>>8);

          ycbcr.Cb <= 128 + ((-((rgb.R<<5)+(rgb.R<<2)+(rgb.R<<1))
                             -((rgb.G<<6)+(rgb.G<<3)+(rgb.G<<1))
                             +(rgb.B<<7)-(rgb.B<<4))>>8);

          ycbcr.Cr <= 128 + (((rgb.R<<7)-(rgb.R<<4)
                             -((rgb.G<<6)+(rgb.G<<5)-(rgb.G<<1))
                             -((rgb.B<<4)+(rgb.B<<1)))>>8);
       end
    end
endmodule: RGB2YCbCr
```

G.2 Códigos da plataforma para prototipação em FPGA

Código G.2: Plataforma para Prototipação em FPGA para aplicações de Visão Computacional

```
//=======================  DE1_SoC_TV.sv  ========================
// ============================================================================
// Copyright (c) 2013 by Terasic Technologies Inc.
// ============================================================================
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera Development
//   Kits made by Terasic.  Other use of this code, including the selling
//   ,duplication, or modification of any portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design reference
//   which illustrates how these types of functions can be implemented.
//   It is the user's responsibility to verify their design for
//   consistency and functionality through the use of formal
//   verification methods.  Terasic provides no warranty regarding the use
//   or functionality of this code.
//
// ============================================================================
//
//  Terasic Technologies Inc
//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//                     web: http://www.terasic.com/
//                     email: support@terasic.com
//
// ============================================================================
//Date:  Mon Jun 17 20:35:29 2013
// ============================================================================

//`define ENABLE_HPS

// January 11 2017, Nelson Campos,
// Sliding window and Sobel Filter modules can be seen in www.sistenix.com/sobel.html

parameter WORD_SIZE = 10;
parameter ROW_SIZE = 800;
parameter BUFFER_SIZE = 3;

module sliding_window #(parameter WORD_SIZE=10, BUFFER_SIZE=3)
               (input  logic clock, reset,
                input logic [WORD_SIZE-1:0] inputPixel,
                output logic [BUFFER_SIZE-1:0][WORD_SIZE-1:0]sliding[BUFFER_SIZE-1:0]);

  logic [(BUFFER_SIZE-1)*WORD_SIZE-1:0] buffer[ROW_SIZE-1:0];
  logic [$clog2(ROW_SIZE)-1:0] ptr;

  always_ff @(posedge clock)
    if(reset) begin
      ptr <=0;
      sliding[0][0] <= 0;
      sliding[0][1] <= 0;
```

```
      sliding[0][2] <= 0;
      sliding[1][0] <= 0;
      sliding[1][1] <= 0;
      sliding[1][2] <= 0;
      sliding[2][0] <= 0;
      sliding[2][1] <= 0;
      sliding[2][2] <= 0;
    end
    else begin
      sliding[0][0] <= inputPixel;
      sliding[1][0] <= sliding[0][0];
      sliding[1][1] <= sliding[0][1];
      sliding[1][2] <= sliding[0][2];
      sliding[2][0] <= sliding[1][0];
      sliding[2][1] <= sliding[1][1];
      sliding[2][2] <= sliding[1][2];

      buffer[ptr] <= sliding[BUFFER_SIZE-1][BUFFER_SIZE-2:0];
      sliding[0][BUFFER_SIZE-1:1] <= buffer[ptr];
      if(ptr < ROW_SIZE-BUFFER_SIZE) ptr <= ptr + 1;
          else ptr <= 0;
    end
endmodule: sliding_window

module sobel #(parameter WORD_SIZE=10)
              (input logic clock,reset,
               input logic [WORD_SIZE-1:0] inputPixel,
               output logic [WORD_SIZE-1:0] outputPixel);

        localparam BUFFER_SIZE=3;

  logic [BUFFER_SIZE-1:0] [WORD_SIZE-1:0] sliding [BUFFER_SIZE-1:0];
  sliding_window #(WORD_SIZE,BUFFER_SIZE) my_window(.*);

  logic [WORD_SIZE+1:0] gx1, gx2, gy1, gy2;

   always_ff @(posedge clock)
     if (reset) begin
        gx1 <= 0;
        gx2 <= 0;
        gy1 <= 0;
        gy2 <= 0;
     end
     else begin
       gx1 <= sliding[0][0] + sliding[2][0] + (sliding[1][0]<<1);
       gx2 <= sliding[0][2] + sliding[2][2] + (sliding[1][2]<<1);
                 gy1 <= sliding[2][0] + sliding[2][2] + (sliding[2][1]<<1);
       gy2 <= sliding[0][0] + sliding[0][2] + (sliding[0][1]<<1);
     end


  logic [WORD_SIZE+1:0] gx, gy;
   always_comb begin
     if (gx1 > gx2) gx <= gx1-gx2;
      else gx <= gx2 - gx1;
     if (gy1 > gy2) gy <= gy1-gy2;
      else gy <= gy2-gy1;
   end

  logic [WORD_SIZE+2:0] g;

   always_comb g <= gy+gx;
   always_ff @(posedge clock)
```

```
    if (reset)
       outputPixel <= 0;
    else
      if (g[WORD_SIZE+2]) outputPixel <= {WORD_SIZE{1'b1}};
          else outputPixel <= g[WORD_SIZE+1:2];
endmodule

module DE1_SoC_TV(

    ///////// ADC /////////
    output              ADC_CONVST,
    output              ADC_DIN,
    input               ADC_DOUT,
    output              ADC_SCLK,

    ///////// AUD /////////
    input               AUD_ADCDAT,
    inout               AUD_ADCLRCK,
    inout               AUD_BCLK,
    output              AUD_DACDAT,
    inout               AUD_DACLRCK,
    output              AUD_XCK,

    ///////// CLOCK2 /////////
    input               CLOCK2_50,

    ///////// CLOCK3 /////////
    input               CLOCK3_50,

    ///////// CLOCK4 /////////
    input               CLOCK4_50,

    ///////// CLOCK /////////
    input               CLOCK_50,

    ///////// DRAM /////////
    output      [12:0]  DRAM_ADDR,
    output      [1:0]   DRAM_BA,
    output              DRAM_CAS_N,
    output              DRAM_CKE,
    output              DRAM_CLK,
    output              DRAM_CS_N,
    inout       [15:0]  DRAM_DQ,
    output              DRAM_LDQM,
    output              DRAM_RAS_N,
    output              DRAM_UDQM,
    output              DRAM_WE_N,

    ///////// FAN /////////
    output              FAN_CTRL,

    ///////// FPGA /////////
    output              FPGA_I2C_SCLK,
    inout               FPGA_I2C_SDAT,

    ///////// GPIO /////////
    inout       [35:0]          GPIO_0,
    inout       [35:0]          GPIO_1,


    ///////// HEX0 /////////
    output      [6:0]   HEX0,
```

```
      //////// HEX1 ////////
      output      [6:0]   HEX1,

      //////// HEX2 ////////
      output      [6:0]   HEX2,

      //////// HEX3 ////////
      output      [6:0]   HEX3,

      //////// HEX4 ////////
      output      [6:0]   HEX4,

      //////// HEX5 ////////
      output      [6:0]   HEX5,

`ifdef ENABLE_HPS
      //////// HPS ////////
      inout               HPS_CONV_USB_N,
      output      [14:0]  HPS_DDR3_ADDR,
      output      [2:0]   HPS_DDR3_BA,
      output              HPS_DDR3_CAS_N,
      output              HPS_DDR3_CKE,
      output              HPS_DDR3_CK_N,
      output              HPS_DDR3_CK_P,
      output              HPS_DDR3_CS_N,
      output      [3:0]   HPS_DDR3_DM,
      inout       [31:0]  HPS_DDR3_DQ,
      inout       [3:0]   HPS_DDR3_DQS_N,
      inout       [3:0]   HPS_DDR3_DQS_P,
      output              HPS_DDR3_ODT,
      output              HPS_DDR3_RAS_N,
      output              HPS_DDR3_RESET_N,
      input               HPS_DDR3_RZQ,
      output              HPS_DDR3_WE_N,
      output              HPS_ENET_GTX_CLK,
      inout               HPS_ENET_INT_N,
      output              HPS_ENET_MDC,
      inout               HPS_ENET_MDIO,
      input               HPS_ENET_RX_CLK,
      input       [3:0]   HPS_ENET_RX_DATA,
      input               HPS_ENET_RX_DV,
      output      [3:0]   HPS_ENET_TX_DATA,
      output              HPS_ENET_TX_EN,
      inout       [3:0]   HPS_FLASH_DATA,
      output              HPS_FLASH_DCLK,
      output              HPS_FLASH_NCSO,
      inout               HPS_GSENSOR_INT,
      inout               HPS_I2C1_SCLK,
      inout               HPS_I2C1_SDAT,
      inout               HPS_I2C2_SCLK,
      inout               HPS_I2C2_SDAT,
      inout               HPS_I2C_CONTROL,
      inout               HPS_KEY,
      inout               HPS_LED,
      inout               HPS_LTC_GPIO,
      output              HPS_SD_CLK,
      inout               HPS_SD_CMD,
      inout       [3:0]   HPS_SD_DATA,
      output              HPS_SPIM_CLK,
      input               HPS_SPIM_MISO,
      output              HPS_SPIM_MOSI,
      inout               HPS_SPIM_SS,
      input               HPS_UART_RX,
```

```
      output              HPS_UART_TX,
      input               HPS_USB_CLKOUT,
      inout       [7:0]   HPS_USB_DATA,
      input               HPS_USB_DIR,
      input               HPS_USB_NXT,
      output              HPS_USB_STP,
`endif /*ENABLE_HPS*/

      ///////// IRDA /////////
      input               IRDA_RXD,
      output              IRDA_TXD,

      ///////// KEY /////////
      input       [3:0]   KEY,

      ///////// LEDR /////////
      output      [9:0]   LEDR,

      ///////// PS2 /////////
      inout               PS2_CLK,
      inout               PS2_CLK2,
      inout               PS2_DAT,
      inout               PS2_DAT2,

      ///////// SW /////////
      input       [9:0]   SW,

      ///////// TD /////////
      input               TD_CLK27,
      input       [7:0]   TD_DATA,
      input               TD_HS,
      output              TD_RESET_N,
      input               TD_VS,

      ///////// VGA /////////
      output      [7:0]   VGA_B,
      output              VGA_BLANK_N,
      output              VGA_CLK,
      output      [7:0]   VGA_G,
      output              VGA_HS,
      output      [7:0]   VGA_R,
      output              VGA_SYNC_N,
      output              VGA_VS
);



//=======================================================
//  REG/WIRE declarations
//=======================================================

wire CLK_18_4;
wire CLK_25;

//For Audio CODEC
wire AUD_CTRL_CLK; //For Audio Controller

//For ITU-R 656 Decoder
wire [15:0] YCbCr;
wire [9:0] TV_X;
wire TV_DVAL;

//For VGA Controller
```

```
wire [9:0] mRed;
wire [9:0] mGreen;
wire [9:0] mBlue;
wire [10:0] VGA_X;
wire [10:0] VGA_Y;
wire VGA_Read; //VGA data request
wire m1VGA_Read; //Read odd field
wire m2VGA_Read; //Read even field

//For YUV 4:2:2 to YUV 4:4:4
wire [7:0] mY;
wire [7:0] mCb;
wire [7:0] mCr;

//For field select
wire [15:0]      mYCbCr;
wire [15:0]      mYCbCr_d;
wire [15:0]      m1YCbCr;
wire [15:0]      m2YCbCr;
wire [15:0]      m3YCbCr;

//For Delay Timer
wire TD_Stable;
wire DLY0;
wire DLY1;
wire DLY2;

//For Down Sample
wire [3:0] Remain;
wire [9:0] Quotient;

wire mDVAL;

wire [15:0]      m4YCbCr;
wire [15:0]      m5YCbCr;
wire [8:0] Tmp1,Tmp2;
wire [7:0] Tmp3, Tmp4;

wire NTSC;
wire PAL;
//=============================================================================
// Structural coding
//=============================================================================


//All inout port turn to tri-state

assign  AUD_ADCLRCK      = AUD_DACLRCK;
assign  GPIO_A  = 36'hzzzzzzzzz;
assign  GPIO_B  = 36'hzzzzzzzzz;

//       Turn On TV Decoder
assign  TD_RESET_N       = 1'b1;

assign  AUD_XCK = AUD_CTRL_CLK;

assign  LED      = VGA_Y;


assign  m1VGA_Read = VGA_Y[0] ? 1'b0 : VGA_Read;
assign  m2VGA_Read = VGA_Y[0] ? VGA_Read : 1'b0;
assign  mYCbCr_d = !VGA_Y[0] ? m1YCbCr : m2YCbCr;
assign  mYCbCr = m5YCbCr;
```

```
assign  Tmp1 = m4YCbCr[7:0]+mYCbCr_d[7:0];
assign  Tmp2 = m4YCbCr[15:8]+mYCbCr_d[15:8];
assign  Tmp3 = Tmp1[8:2]+m3YCbCr[7:1];
assign  Tmp4 = Tmp2[8:2]+m3YCbCr[15:9];
assign  m5YCbCr = {Tmp4,Tmp3};

//7 segment LUT
SEG7_LUT_6 u0(.oSEG0(HEX0),
              .oSEG1(HEX1),
              .oSEG2(HEX2),
              .oSEG3(HEX3),
              .oSEG4(HEX4),
              .oSEG5(HEX5),
              .iDIG(SW));

//TV Decoder Stable Check
TD_Detect u2 (.oTD_Stable(TD_Stable),
              .oNTSC(NTSC),
              .oPAL(PAL),
              .iTD_VS(TD_VS),
              .iTD_HS(TD_HS),
              .iRST_N(KEY[0]));

//Reset Delay Timer
Reset_Delay u3 (.iCLK(CLOCK_50),
                .iRST(TD_Stable),
                .oRST_0(DLY0),
                .oRST_1(DLY1),
                .oRST_2(DLY2));

//ITU-R 656 to YUV 4:2:2
ITU_656_Decoder u4 (//TV Decoder Input
                    .iTD_DATA(TD_DATA),
                    //Position Output
                    .oTV_X(TV_X),
                    //YUV 4:2:2 Output
                    .oYCbCr(YCbCr),
                    .oDVAL(TV_DVAL),
                    //Control Signals
                    .iSwap_CbCr(Quotient[0]),
                    .iSkip(Remain==4'h0),
                    .iRST_N(DLY1),
                    .iCLK_27(TD_CLK27));

//For Down Sample 720 to 640
DIV u5 (.aclr(!DLY0),
        .clock(TD_CLK27),
        .denom(4'h9),
        .numer(TV_X),
        .quotient(Quotient),
        .remain(Remain));

//SDRAM frame buffer
Sdram_Control_4Port u6 (//HOST Side
                        .REF_CLK(TD_CLK27),
                        .CLK_18(AUD_CTRL_CLK),
                        .RESET_N(DLY0),
                        //FIFO Write Side 1
                        .WR1_DATA(YCbCr),
                        .WR1(TV_DVAL),
                        .WR1_FULL(WR1_FULL),
                        .WR1_ADDR(0),
```

```
                        .WR1_MAX_ADDR(NTSC ? 640*507 : 640*576), //525-18
                        .WR1_LENGTH(9'h80),
                        .WR1_LOAD(!DLY0),
                        .WR1_CLK(TD_CLK27),
                        //FIFO Read Side 1
                        .RD1_DATA(m1YCbCr),
                        .RD1(m1VGA_Read),
                        .RD1_ADDR(NTSC ? 640*13 : 640*22),//Read odd field and bypess blanking
                        .RD1_MAX_ADDR(NTSC ? 640*253 : 640*262),
                        .RD1_LENGTH(9'h80),
                        .RD1_LOAD(!DLY0),
                        .RD1_CLK(TD_CLK27),
                        //FIFO Read Side 2
                        .RD2_DATA(m2YCbCr),
                        .RD2(m2VGA_Read),
                        .RD2_ADDR(NTSC?640*267:640*310), //Read even field and bypess blanking
                        .RD2_MAX_ADDR(NTSC ? 640*507 : 640*550),
                        .RD2_LENGTH(9'h80),
                        .RD2_LOAD(!DLY0),
                        .RD2_CLK(TD_CLK27),
                        //SDRAM Side
                        .SA(DRAM_ADDR),
                        .BA(DRAM_BA),
                        .CS_N(DRAM_CS_N),
                        .CKE(DRAM_CKE),
                        .RAS_N(DRAM_RAS_N),
                        .CAS_N(DRAM_CAS_N),
                        .WE_N(DRAM_WE_N),
                        .DQ(DRAM_DQ),
                        .DQM({DRAM_UDQM,DRAM_LDQM}),
                        .SDR_CLK(DRAM_CLK));

//YUV 4:2:2 to YUV 4:4:4
YUV422_to_444 u7 (//YUV 4:2:2 Input
                  .iYCbCr(mYCbCr),
                  //YUV 4:4:4 Output
                  .oY(mY),
                  .oCb(mCb),
                  .oCr(mCr),
                  //Control Signals
                  .iX(VGA_X-160),
                  .iCLK(TD_CLK27),
                  .iRST_N(DLY0));

//YCbCr 8-bit to RGB-10 bit
YCbCr2RGB u8 (//Output Side
              .Red(mRed),
              .Green(mGreen),
              .Blue(mBlue),
              .oDVAL(mDVAL),
              //Input Side
              .iY(mY),
              .iCb(mCb),
              .iCr(mCr),
              .iDVAL(VGA_Read),
              //Control Signal
              .iRESET(!DLY2),
              .iCLK(TD_CLK27));

//      VGA Controller
wire [9:0] vga_r10;
wire [9:0] vga_g10;
wire [9:0] vga_b10;
```

```
assign VGA_R = vga_r10[9:2];
assign VGA_G = vga_g10[9:2];
assign VGA_B = vga_b10[9:2];

logic [9:0] red_filter;
logic [9:0] green_filter;
logic [9:0] blue_filter;
logic [9:0] edge_filter;
logic [2:0] video_mode;

sobel filter(.clock(TD_CLK27), .reset(1'b0), .inputPixel(mRed),
             .outputPixel(edge_filter));

assign video_mode = SW[2:0];

always @(video_mode)
  case(video_mode)
    3'd0: begin
      red_filter = mRed;
                green_filter = mGreen;
                blue_filter = mBlue;
    end
         3'd1: begin
      red_filter = mRed;
                green_filter = 10'd0;
                blue_filter = 10'd0;
    end
         3'd2: begin
      red_filter = 10'd0;
                green_filter = mGreen;
                blue_filter = 10'd0;
    end
         3'd3: begin
      red_filter = 10'd0;
                green_filter = 10'd0;
                blue_filter = mBlue;
    end
         3'd4: begin
      red_filter = mY;
                green_filter = mCb;
                blue_filter = mCr;
    end
         3'd5: begin
      red_filter = mY;
                green_filter = mY;
                blue_filter = mY;
    end
         3'd6: begin
      red_filter = edge_filter;
                green_filter = 10'd0;
                blue_filter = 10'd0;
    end
         3'd7: begin
      red_filter = edge_filter;
                green_filter = edge_filter;
                blue_filter = edge_filter;
    end

endcase
VGA_Ctrl u9 (//Host Side
             .iRed(red_filter),
             .iGreen(green_filter),
             .iBlue(blue_filter),
```

```
                .oCurrent_X(VGA_X),
                .oCurrent_Y(VGA_Y),
                .oRequest(VGA_Read),
                //VGA Side
                .oVGA_R(vga_r10 ),
                .oVGA_G(vga_g10 ),
                .oVGA_B(vga_b10 ),
                .oVGA_HS(VGA_HS),
                .oVGA_VS(VGA_VS),
                .oVGA_SYNC(VGA_SYNC_N),
                .oVGA_BLANK(VGA_BLANK_N),
                .oVGA_CLOCK(VGA_CLK),
                //Control Signal
                .iCLK(TD_CLK27),
                .iRST_N(DLY2));

//Line buffer, delay one line
Line_Buffer u10 (.aclr(!DLY0),
                 .clken(VGA_Read),
                 .clock(TD_CLK27),
                 .shiftin(mYCbCr_d),
                 .shiftout(m3YCbCr));

Line_Buffer u11 (.aclr(!DLY0),
                 .clken(VGA_Read),
                 .clock(TD_CLK27),
                 .shiftin(m3YCbCr),
                 .shiftout(m4YCbCr));

AUDIO_DAC u12 (//Audio Side
               .oAUD_BCK(AUD_BCLK),
               .oAUD_DATA(AUD_DACDAT),
               .oAUD_LRCK(AUD_DACLRCK),
               //Control Signals
               .iSrc_Select(2'b01),
               .iCLK_18_4(AUD_CTRL_CLK),
               .iRST_N(DLY1));

//Audio CODEC and video decoder setting
I2C_AV_Config u1 (//Host Side
                  .iCLK(CLOCK_50),
                  .iRST_N(KEY[0]),
                  //I2C Side
                  .I2C_SCLK(FPGA_I2C_SCLK),
                  .I2C_SDAT(FPGA_I2C_SDAT));

endmodule
```

G.3 Códigos da validação funcional de um conversor de cores

Código G.3: Makefile

```
ifeq ("$(UVMC_HOME)","")
$(error ERROR: UVMC_HOME environment variable is not defined)
endif
```

```
# definitions needed for OpenCV:
CV_COPTS=`pkg-config --cflags opencv`
CV_LOPTS=`pkg-config --libs opencv`

# definitions only needed for VCS:
VCS_UVMC_SC_OPTS=-tlm2 -cflags "-g -I. -I$(VCS_HOME)/etc/systemc/tlm/include/tlm/tlm_utils -I$(UVMC_HOME)
    /src/connect/sc" $(UVMC_HOME)/src/connect/sc/uvmc.cpp $(UVM_HOME)/src/dpi/uvm_dpi.cc


VCS_UVMC_SV_OPTS=-q -sverilog -ntb_opts uvm -timescale=1ns/1ps +define+UVM_OBJECT_MUST_HAVE_CONSTRUCTOR +
    incdir+$(UVM_HOME)/src+$(UVM_HOME)/src/vcs+$(UVMC_HOME)/src/connect/sv +define+
    UVM_OBJECT_MUST_HAVE_CONSTRUCTOR $(UVM_HOME)/src/uvm_pkg.sv $(UVM_HOME)/src/vcs/*.sv $(UVMC_HOME)/src
    /connect/sv/uvmc_pkg.sv

VCS_UVMC_OPTS=-q -sysc=deltasync -lca -sysc -debug_pp -timescale=1ns/1ps uvm_custom_install_recording
    sc_main $(TOP)

VCS_SIMV= +UVM_NO_RELNOTES +UVM_TR_RECORD +UVM_TESTNAME=$(TEST)

# definitions for files
TOP = top
REFMOD = external/refmod
TEST= simple_test
vcs:    clean
        g++ -c $(CV_COPTS) external/call_opencv.cpp
        g++ -c $(CV_COPTS) external/cvFunction.cpp
        syscan $(VCS_UVMC_SC_OPTS) $(REFMOD).cpp
        vlogan $(VCS_UVMC_SV_OPTS) $(TOP).sv
        vcs    $(VCS_UVMC_OPTS) $(CV_LOPTS) $(TOP) $(CV_LOPTS) cvFunction.o call_opencv.o
        ./simv $(VCS_SIMV) +UVM_VERBOSITY=MEDIUM

vcs_debug:clean
        g++ -c $(CV_COPTS) external/call_opencv.cpp
        g++ -c $(CV_COPTS) external/cvFunction.cpp
        syscan $(VCS_UVMC_SC_OPTS) $(REFMOD).cpp
        vlogan $(VCS_UVMC_SV_OPTS) $(TOP).sv
        vcs    $(VCS_UVMC_OPTS) $(CV_LOPTS) $(TOP) $(CV_LOPTS) cvFunction.o call_opencv.o
        ./simv $(VCS_SIMV) +UVM_VERBOSITY=HIGH

# if Using Octave
vcs_oct:
        syscan $(VCS_UVMC_SC_OPTS) -cflags "$(OCT_COPTS) -DREFMOD_OCTAVE" $(REFMOD).cpp
        vlogan -q $(VCS_UVMC_SV_OPTS) $(TOP).sv
        vcs -q $(VCS_UVMC_OPTS) $(CV_LOPTS) $(TOP)
        ./simv +UVM_VERBOSITY=LOW

vcs_oct_debug:
        syscan $(VCS_UVMC_SC_OPTS) -cflags "$(OCT_COPTS) -DREFMOD_OCTAVE" $(REFMOD).cpp
        vlogan $(VCS_UVMC_SV_OPTS) $(TOP).sv
        vcs -q $(VCS_UVMC_OPTS) $(CV_LOPTS) $(TOP)
        ./simv +UVM_VERBOSITY=MEDIUM

clean:
        rm -rf a.out # simulation output file
        rm -rf INCA_libs irun.log ncsc.log # ius
        rm -rf work certe_dump.xml transcript .mgc_simple_ref .mgc_ref_nobuff .mgc_ref_oct # mgc
        rm -rf csrc simv simv.daidir ucli.key .vlogansetup.args .vlogansetup.env .vcs_lib_lock simv.vdb
            AN.DB vc_hdrs.h *.diag *.vpd *tar.gz # vcs

# Wave form
view_waves:
        dve -vpd vcdplus.vpd &
```

Código G.4: Códigos em UVM/SystemVerilog da Verificação Funcional de um Conversor de cores RGB para YC_bC_r

```
//==================== cvFunction.svh ====================
`ifndef CVFUNCTION
`define CVFUNCTION

`define CHANNELS 3
`define HEIGHT 1920
`define WIDTH 1080
`define SIZE CHANNELS*WIDTH*HEIGHT

import "DPI-C" context function longint unsigned setPixel (longint unsigned frame, int x, int y, int r,
    int g, int b);

import "DPI-C" context function longint unsigned allocateFrame();

import "DPI-C" context function longint unsigned readframe();

import "DPI-C" context function longint unsigned getChannel (longint unsigned frame, int x, int y, int c)
    ;

import "DPI-C" context function string frame_tr_convert2string(longint unsigned a_ptr, int rgb_ycbcr);

import "DPI-C" context function int frameCompare (longint unsigned pp1, longint unsigned pp2);

`endif
//=======================================

//==================== frame_tr.sv ====================
`include "cvFunction.svh"

// Basic frame transaction
class frame_tr extends uvm_sequence_item;
    longint unsigned a;

    `uvm_object_utils_begin(frame_tr)
        `uvm_field_int(a, UVM_ALL_ON|UVM_HEX)
    `uvm_object_utils_end

    function new(string name="frame_tr");
        super.new(name);
    endfunction: new

    function int compare(frame_tr tr);
       return frameCompare(a, tr.a);
    endfunction

endclass: frame_tr
//=======================================

//==================== frame_rgb_tr.sv ====================
`include "cvFunction.svh"
// pixels are RGB
class frame_rgb_tr extends frame_tr;
  `uvm_object_utils(frame_rgb_tr)

    function new(string name="frame_rgb_tr");
        super.new(name);
```

```
    endfunction: new

    function string convert2string();
       return $sformatf("a=%d %s", a, frame_tr_convert2string(a, 1));
    endfunction

endclass: frame_rgb_tr

//=====================================

//==================== frame_rgb_ycbcr.sv ====================
`include "cvFunction.svh"
// pixels are Y Cb Cr
class frame_ycbcr_tr extends frame_tr;
  `uvm_object_utils(frame_ycbcr_tr)

    function new(string name="frame_ycbcr_tr");
        super.new(name);
    endfunction: new

    function string convert2string();
       return frame_tr_convert2string(a, 0);
    endfunction

endclass: frame_ycbcr_tr
//=====================================

//==================== frame_seq.sv ====================
`include "cvFunction.svh"

//Input Sequence
class frame_seq extends uvm_sequence #(frame_rgb_tr);
    `uvm_object_utils(frame_seq)

    function new(string name="frame_seq");
        super.new(name);
    endfunction: new

    task body;
         frame_rgb_tr tx;
            tx = frame_rgb_tr::type_id::create("tx");
            start_item(tx);
            tx.a = readframe();
            finish_item(tx);
    endtask: body
endclass: frame_seq
//=====================================

//==================== frame_sqr.sv ====================
//Input Sequencer
class frame_sqr extends uvm_sequencer #(frame_rgb_tr);
    `uvm_component_utils(frame_sqr)

    function new (string name = "frame_sqr", uvm_component parent = null);
        super.new(name, parent);
    endfunction
endclass: frame_sqr
//=====================================

//==================== frame_driver.sv ====================
`include "cvFunction.svh"

//Input Driver
```

```
typedef virtual rgb_if rgb_vif;

class frame_driver extends uvm_driver #(frame_rgb_tr);
    `uvm_component_utils(frame_driver)
    rgb_vif vif;
    frame_rgb_tr tr;

    function new(string name = "frame_driver", uvm_component parent = null);
        super.new(name, parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        assert(uvm_config_db#(rgb_vif)::get(this, "", "vif", vif));
    endfunction

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        fork
            reset_signals();
            get_and_drive(phase);
        join
    endtask

    virtual protected task reset_signals();
        wait (vif.rst === 1);
        forever begin
            vif.R <= 'x;
            vif.G <= 'x;
            vif.B <= 'x;
            @(posedge vif.rst);
        end
    endtask

    int rows, cols;
    string message, message2;
    virtual protected task get_and_drive(uvm_phase phase);
        wait (vif.rst === 1);
        @(negedge vif.rst);

        //forever begin
        seq_item_port.get(tr);
        begin_tr(tr, "frame_driver");
        for(rows = 0; rows < `HEIGHT; rows++)begin
             for(cols = 0; cols < `WIDTH; cols++)begin
                 vif.R = getChannel(tr.a, cols, rows, 2);
                 vif.G = getChannel(tr.a, cols, rows, 1);
                 vif.B = getChannel(tr.a, cols, rows, 0);
                 `uvm_info("DRIVER",$sformatf("Sent pixel R=%d G=%d, B=%d", vif.R, vif.G, vif.B), UVM_LOW
                     )
                 @(posedge vif.clk);
             end
         end
         end_tr(tr);
         `uvm_info("DRIVER","ended", UVM_LOW)
        //end
    endtask

endclass
//=====================================

//=================== frame_monitor.sv ================
`include "cvFunction.svh"
```

```
class frame_monitor extends uvm_monitor;
    rgb_vif  vif;

    frame_rgb_tr tr;
    uvm_analysis_port #(frame_rgb_tr) item_collected_port;
    `uvm_component_utils(frame_monitor)

    function new(string name, uvm_component parent);
        super.new(name, parent);
        item_collected_port = new ("item_collected_port", this);
        `uvm_info("MONITOR CREATED","new",UVM_MEDIUM)
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        assert(uvm_config_db#(rgb_vif)::get(this, "", "vif", vif));
        tr = frame_rgb_tr::type_id::create("tr", this);
    endfunction

    virtual task run_phase(uvm_phase phase);
        `uvm_info("Monitor rgb","started run_phase",UVM_MEDIUM)

        wait (vif.rst === 1);
        @(negedge vif.rst);

        //forever begin
            tr.a = allocateFrame();
            @(posedge vif.clk);
            begin_tr(tr, "frame_monitor");
            for(int i = 0; i < `HEIGHT; i++)begin
                for(int j = 0; j < `WIDTH; j++)begin
                    setPixel(tr.a, i, j, vif.R, vif.G, vif.B);
                    `uvm_info("Monitor rgb",$sformatf("Received pixel i=%d j=%d R=%d G=%d, B=%d", i, j,
                        vif.R, vif.G, vif.B), UVM_LOW)
                    @(posedge vif.clk);
                end
            end
            `uvm_info("Monitor rgb ", "Received frame", UVM_LOW)
            item_collected_port.write(tr);
            `uvm_info("Monitor rgb ", "Wrote frame to port", UVM_LOW)
            end_tr(tr);
            `uvm_info("Monitor rgb ", "ended tr", UVM_LOW)
        //end
    endtask

endclass
//=====================================

//================== frame_agent.sv ==================
class frame_agent extends uvm_agent;
    frame_sqr sqr;
    frame_driver    drv;
    frame_monitor   mon;

    uvm_analysis_port #(frame_rgb_tr) item_collected_port;

    `uvm_component_utils(frame_agent)

    function new(string name = "agent", uvm_component parent = null);
        super.new(name, parent);
        item_collected_port = new("item_collected_port", this);
    endfunction
```

```
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        sqr = frame_sqr::type_id::create("sqr", this);
        drv = frame_driver::type_id::create("drv", this);
        mon = frame_monitor::type_id::create("mon", this);
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        mon.item_collected_port.connect(item_collected_port);
        drv.seq_item_port.connect(sqr.seq_item_export);
    endfunction
endclass: frame_agent
//=====================================

//=================== frame_driver_out.sv ====================
typedef virtual ycbcr_if ycbcr_vif;

class frame_driver_out extends uvm_driver #(frame_ycbcr_tr);
    `uvm_component_utils(frame_driver_out)
    ycbcr_vif vif;

    function new(string name = "frame_driver_out", uvm_component parent = null);
        super.new(name, parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        assert(uvm_config_db#(ycbcr_vif)::get(this, "", "vif", vif));
    endfunction

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        fork
            reset_signals();
            drive(phase);
        join
    endtask

    virtual protected task reset_signals();
        wait (vif.rst === 1);
        forever begin
            vif.Y <= 'x;
            vif.Cb <= 'x;
            vif.Cr <= 'x;
            @(posedge vif.rst);
        end
    endtask

    virtual protected task drive(uvm_phase phase);
        wait(vif.rst === 1);
        @(negedge vif.rst);
        @(posedge vif.clk);
    endtask
endclass
//=====================================

//=================== frame_monitor_out.sv ====================
`include "cvFunction.svh"

class frame_monitor_out extends uvm_monitor;
    ycbcr_vif  vif;
```

```
    frame_ycbcr_tr tr;
    uvm_analysis_port #(frame_ycbcr_tr) item_collected_port;
    `uvm_component_utils(frame_monitor_out)

    function new(string name, uvm_component parent);
        super.new(name, parent);
        item_collected_port = new ("item_collected_port", this);
        `uvm_info("MONITOR OUT CREATED","new",UVM_MEDIUM)
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        assert(uvm_config_db#(ycbcr_vif)::get(this, "", "vif", vif));
        tr = frame_ycbcr_tr::type_id::create("tr", this);
    endfunction

    virtual task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        `uvm_info("Monitor ycbcr ","started run_phase",UVM_MEDIUM)

        wait (vif.rst === 1);
        @(negedge vif.rst);
        @(posedge vif.clk);
        //forever begin
         tr.a = allocateFrame();
         @(posedge vif.clk);
         begin_tr(tr, "frame_monitor_out");
         for(int i = 0; i < `HEIGHT; i++)begin
             for(int j = 0; j < `WIDTH; j++)begin
                 setPixel(tr.a, i, j, vif.Y, vif.Cb, vif.Cr);
                 `uvm_info("Monitor ycbcr",$sformatf("Received pixel i=%d j=%d Y=%d Cb=%d, Cr=%d", i, j,
                      vif.Y, vif.Cb, vif.Cr), UVM_LOW)
                 `uvm_info("Monitor ycbcr",$sformatf("loop i=%d j=%d before clk", i, j), UVM_LOW)
                 @(posedge vif.clk);
                 `uvm_info("Monitor ycbcr",$sformatf("loop i=%d j=%d ended", i, j), UVM_LOW)
              end
         end
         `uvm_info("Monitor ycbcr ", "Received frame", UVM_LOW)
         item_collected_port.write(tr);
         `uvm_info("Monitor ycbcr ", "Wrote frame to port", UVM_LOW)
         end_tr(tr);

        //end
        @(posedge vif.clk);
       `uvm_info("Monitor ycrcb", "Stopping the test", UVM_LOW);
        phase.drop_objection(this);
    endtask

endclass
//=====================================

//==================== frame_agent_out.sv ====================
class frame_agent_out extends uvm_agent;
    frame_driver_out    drv;
    frame_monitor_out   mon;

    uvm_analysis_port #(frame_ycbcr_tr) item_collected_port;

    `uvm_component_utils(frame_agent_out)

    function new(string name = "frame_agent_out", uvm_component parent = null);
        super.new(name, parent);
```

```
        item_collected_port = new("item_collected_port", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        drv = frame_driver_out::type_id::create("drv", this);
        mon = frame_monitor_out::type_id::create("mon", this);
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        mon.item_collected_port.connect(item_collected_port);
    endfunction
endclass: frame_agent_out
//=========================================

//==================== bvm_comparator.sv ====================
class bvm_comparator #( type T = int ) extends uvm_scoreboard;

  typedef bvm_comparator #(T) this_type;
  `uvm_component_param_utils(this_type)

  const static string type_name =
    "bvm_comparator #(T)";

  uvm_put_imp #(T, this_type) from_refmod;
  uvm_analysis_imp #(T, this_type) from_dut;

  typedef uvm_built_in_converter #( T ) convert;

  int m_matches, m_mismatches;
  T exp;
  bit free;
  event compared;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    from_refmod = new("from_refmod", this);
    from_dut = new("from_dut", this);
    m_matches = 0;
    m_mismatches = 0;
    exp = new("exp");
    free = 1;
  endfunction

  virtual function string get_type_name();
    return type_name;
  endfunction

  virtual task put(T t);
    if(!free) @compared;
    exp.copy(t);
    free = 0;
    @compared;
    free = 1;
  endtask

  virtual function bit try_put(T t);
    if(free) begin
      exp.copy(t);
      free = 0;
      return 1;
    end
```

```
    else return 0;
  endfunction

  virtual function bit can_put();
    return free;
  endfunction

  virtual function void write(T rec);
    string s;

    if (free) begin
      $sformat(s, "No expect transaction to compare with %s", convert::convert2string(rec));
      uvm_report_fatal("Comparator no expect", s);
    end

    if(!exp.compare(rec)) begin
      $sformat(s, "%s differs from %s", convert::convert2string(rec),
                                        convert::convert2string(exp));
      uvm_report_warning("Comparator Mismatch", s);
      m_mismatches++;
    end
    else begin
      s = convert::convert2string(exp);
      uvm_report_info("Comparator Match", s);
      m_matches++;
    end

    -> compared;
  endfunction

endclass
//===================================

//=================== env.sv ====================
class env extends uvm_env;

    frame_agent frag;

    frame_agent_out frago;
    bvm_comparator #(frame_ycbcr_tr) comp;

    uvm_tlm_analysis_fifo #(frame_rgb_tr) to_refmod;
    `uvm_component_utils(env)

    function new(string name, uvm_component parent = null);
        super.new(name, parent);
        to_refmod = new("to_refmod", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        frag = frame_agent::type_id::create("frag", this);
        frago = frame_agent_out::type_id::create("frago", this);
        comp = bvm_comparator#(frame_ycbcr_tr)::type_id::create("comp", this);
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        // connect agent to refmod via fifo
        frag.item_collected_port.connect(to_refmod.analysis_export);
        uvmc_tlm1 #(frame_rgb_tr)::connect(to_refmod.get_export, "refmod_i.in");

        //Connect scoreboard
```

```
        uvmc_tlm1 #(frame_ycbcr_tr)::connect(comp.from_refmod,"refmod_i.out");
        frago.item_collected_port.connect(comp.from_dut);
    endfunction

    virtual function void end_of_elaboration_phase(uvm_phase phase);
        super.end_of_elaboration_phase(phase);
    endfunction

    virtual function void report_phase(uvm_phase phase);
        super.report_phase(phase);
        `uvm_info(get_type_name(), $sformatf("Reporting matched %0d", comp.m_matches), UVM_NONE)
        if (comp.m_mismatches) begin
            `uvm_error(get_type_name(), $sformatf("Saw %0d mismatched samples", comp.m_mismatches))
        end
    endfunction
endclass
//=====================================

//=================== simple_test.sv ==================
//Test
class simple_test extends uvm_test;
  env env_h;
  frame_seq fseq;

  `uvm_component_utils(simple_test)

  function new(string name, uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_h = env::type_id::create("env_h", this);
    fseq = frame_seq::type_id::create("fseq", this);
  endfunction

  task run_phase(uvm_phase phase);
    fork
        fseq.start(env_h.frag.sqr);
    join
  endtask: run_phase

endclass
//=====================================

//=================== top.sv ==================
import uvm_pkg::*;
import uvmc_pkg::*;
`include "uvm_macros.svh"

`include "./rgb_if.sv"
`include "./ycbcr_if.sv"
`include "RGB2YCbCr.sv"
`include "./frame_tr.sv"
`include "./frame_rgb_tr.sv"
`include "./frame_ycbcr_tr.sv"
`include "./frame_seq.sv"
`include "./frame_sqr.sv"
`include "./frame_driver.sv"
`include "./frame_monitor.sv"
`include "./frame_agent.sv"
`include "./frame_driver_out.sv"
`include "./frame_monitor_out.sv"
```

```
`include "./frame_agent_out.sv"

`include "./bvm_comparator.sv"
`include "./env.sv"
`include "./simple_test.sv"

//Top
module top;
  logic clk;
  logic rst;

  initial begin
    clk = 0;
    rst = 1;
    #22 rst = 0;

  end

  always #5 clk = !clk;

  rgb_if in(clk, rst);
  ycbcr_if out(clk, rst);

  RGB2YCbCr rtl(in, out);

  initial begin
    `ifdef INCA
       $recordvars();
    `endif
    `ifdef VCS
       $vcdpluson;
    `endif
    `ifdef QUESTA
       $wlfdumpvars();
       set_config_int("*", "recording_detail", 1);
    `endif

    uvm_config_db#(rgb_vif)::set(uvm_root::get(), "*.env_h.frag.*", "vif", in);
    uvm_config_db#(ycbcr_vif)::set(uvm_root::get(), "*.env_h.frago.*", "vif", out);
    run_test("simple_test");
  end
endmodule
```

Código G.5: Códigos em C/C++/SystemC da Verificação Funcional de um Conversor de cores RGB para YC_bC_r

```
//================== external/definitions.h ==================
#ifndef DEFINITIONS__H
#define DEFINITIONS__H

#define HEIGHT  1920
#define WIDTH   1080
#define CHANNELS 3

#endif
//============================================

//================== external/cvFunction.cpp ==================
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
```

```
#include <sstream>
#include "definitions.h"

#include "svdpi.h"
#include <math.h>
#include <stdio.h>
#include <cv.h>
#include <highgui.h>

#include <stdlib.h>
#include "vpi_user.h"

using namespace cv;
using namespace std;

extern "C" unsigned long long allocateFrame(){
  static Mat image(HEIGHT,WIDTH, CV_8UC3);
  void *frame_data = malloc(image.total()*image.elemSize());
  // verificar frame_data != NULL
  cout << "allocateFrame: " << image.total()*image.elemSize() << "bytes frame_data=" <<  (unsigned long
      long)frame_data << endl;
  return (unsigned long long)frame_data;
}

extern "C" unsigned long long readframe()
{
  const char* filename = "index.jpg";
  Mat image;
  image = imread(filename, 1);
  if(!image.data)
    cout <<"No image data" <<endl;

  cout << "readframe after imread: rows=" << image.rows <<" cols=" << image.cols << " data=" << (unsigned
       long long)image.data  << endl;

  Vec3b intensity = image.at<Vec3b>(0, 0);
  cout << "readframe: image at 0,0 b=" << (int)intensity.val[0] << " g=" << (int)intensity.val[1] << " r=
      " << (int)intensity.val[2] << endl;
  intensity = image.at<Vec3b>(0, 1);
  cout << "readframe: image at 0,1 b=" << (int)intensity.val[0] << " g=" << (int)intensity.val[1] << " r=
      " << (int)intensity.val[2] << endl;
  intensity = image.at<Vec3b>(1, 0);
  cout << "readframe: image at 1,0 b=" << (int)intensity.val[0] << " g=" << (int)intensity.val[1] << " r=
      " << (int)intensity.val[2] << endl;

  unsigned long long pp = allocateFrame();
  Mat frame(HEIGHT,WIDTH, CV_8UC3, (void *)pp);
  cout << endl << "readframe: frame.data=" << (unsigned long long)frame.data << endl;
  memcpy(frame.data, image.data, image.total()*image.elemSize());

  intensity = frame.at<Vec3b>(0, 0);
  cout << "readframe: frame at 0,0 b=" << (int)intensity.val[0] << " g=" << (int)intensity.val[1] << " r=
      " << (int)intensity.val[2] << endl;
  intensity = frame.at<Vec3b>(0, 1);
  cout << "readframe: frame at 0,1 b=" << (int)intensity.val[0] << " g=" << (int)intensity.val[1] << " r=
      " << (int)intensity.val[2] << endl;
  intensity = frame.at<Vec3b>(1, 0);
  cout << "readframe: frame at 1,0 b=" << (int)intensity.val[0] << " g=" << (int)intensity.val[1] << " r=
      " << (int)intensity.val[2] << endl;

  return (unsigned long long)frame.data;
}
```

```
extern "C" int frameCompare (unsigned long long pp1, unsigned long long pp2){
  bool flag = 1;
  cout << "frameCompare: inicio pp1=" << pp1 << " pp2=" << pp2 << endl;
  Mat frame1(HEIGHT,WIDTH, CV_8UC3, (void *)pp1);
  Mat frame2(HEIGHT,WIDTH, CV_8UC3, (void *)pp1);
  for(int i = 0; i < HEIGHT; i++)
    for(int j = 0; j < WIDTH; j++) {
      cout << "frameCompare: i=" << i << " j=" << j << endl;
      Vec3b intensity1 = frame1.at<Vec3b>(i, j);
      Vec3b intensity2 = frame2.at<Vec3b>(i, j);
      for(int c = 0; c < CHANNELS; c++) {
        if(abs(intensity1.val[c] - intensity2.val[c]) > 1) { // tolerance of 1
          flag = 0;
          cout << "MISMATCH!"<<" from refmod: " << (int)intensity1.val[c] <<" from dut: " << (int)
              intensity1.val[c];
        }
        else cout << "MATCH!" <<" from refmod: " << (int)intensity1.val[c] <<" from dut: " << (int)
            intensity1.val[c];
        cout << endl;
      }
  }
  return flag;
}

extern "C" int getChannel(unsigned long long pp, int i, int j, int c){
    cout << "getChannel: pp=" << pp << " i=" << i << " j=" << j << " c=" << c;
    Mat image(HEIGHT,WIDTH, CV_8UC3, (void *)pp);
    cout << endl << "getChannel: image.data=" << (unsigned long long)image.data << endl;
    Vec3b intensity = image.at<Vec3b>(i, j);
    cout << " val=" << (int)intensity.val[c] << endl;
    return intensity.val[c];
}

extern "C"void setPixel(unsigned long long pp, int i, int j, int r, int g, int b){
     cout << "setPixel: pp=" << pp << " i=" << i << " j=" << j << " r=" << r << " g=" << g << " b =" << b
         << endl;
     Mat image(HEIGHT,WIDTH, CV_8UC3, (void *)pp);
     cout << "setPixel: at=" << (unsigned long long)&(image.at<Vec3b>(i, j)) << endl;
     image.at<Vec3b>(i, j) = Vec3b(b, g, r);
}

ostream& os_pixel (ostream& os, const Vec3b& intensity, int rgb_ycbcr) {
  if(rgb_ycbcr)
      os << " R=" << (uint8_t)intensity.val[2] << " G=" << (uint8_t)intensity.val[1] << " B=" << (uint8_t
          )intensity.val[0] << endl;
  else
      os << " Y=" << (uint8_t)intensity.val[0] << " Cb=" << (int8_t)intensity.val[1] << " Cr=" << (int8_t
          )intensity.val[2] << endl;
  return os;
}

extern "C"  const char* frame_tr_convert2string (unsigned long long pp, int rgb_ycbcr){
     std::stringstream os;
     Mat image(HEIGHT,WIDTH, CV_8UC3, (void *)pp);
     Vec3b intensity = image.at<Vec3b>(0, 0);
     os << "first pixel " << os_pixel(os, intensity, rgb_ycbcr) << endl;
     intensity = image.at<Vec3b>(image.rows-1, image.cols-1);
     os << "last pixel"  << os_pixel(os, intensity, rgb_ycbcr) << endl;
     string str(os.str());
     return str.c_str();
}
//=====================================
```

```
//=================== external/call_opencv.cpp ===================
#include <stdio.h>
#include <cv.h>
#include <highgui.h>
#include "definitions.h"
using namespace cv;

void rgb2ycbcr(unsigned long long pp){
  Mat image(HEIGHT,WIDTH, CV_8UC3, (void *)pp);
  cvtColor(image, image, CV_BGR2YCrCb);
}
//================================================

//=================== external/refmod.cpp ===================
#include "systemc.h"
#include "tlm.h"
using namespace tlm;

void rgb2ycbcr(unsigned long long);

struct tr {
  unsigned long long a;
};

#include "uvmc.h"
using namespace uvmc;
UVMC_UTILS_1(tr, a)


SC_MODULE(refmod) {
  sc_port<tlm_get_peek_if<tr> > in;
  sc_port<tlm_put_if<tr> > out;

  void p() {
    tr tr;
    while (1) {

      tr = in->get();
      rgb2ycbcr(tr.a); // in place
      out->put(tr);

    }
  }
  SC_CTOR(refmod): in("in"), out("out") { SC_THREAD(p); }
};

int sc_main(int argc, char* argv[]) {

  refmod  refmod_i("refmod_i");

  uvmc_connect(refmod_i.in, "refmod_i.in");
  uvmc_connect(refmod_i.out, "refmod_i.out");

  sc_start();
  return(0);
}
```

G.4 Códigos do script de conversão das variáveis do arquivo XML para ponto fixo

Código G.6: Script de conversão das variáveis do arquivo XML para ponto fixo

```
printf ("Convertendo arquivo = haarcascade\n");
fn1='haarcascade_frontalface_default.xml';
fid=fopen(fn1, 'r');
stg=0;
nos=0;
max=0;
fn2='class25.txt'
fid2=fopen(fn2,"w");
fn3='info25.txt'
fid3=fopen(fn3,"w");
c=fgetl(fid);

% To read from haar.cpp
FILE = textread('haar.cpp', "%s");
B1 = FILE(215); %get FRAC_WEIGHT value
B1 = int32(cell2mat(B1)); %convert to string
B1 = B1-48; %convert to int32
B1c = 0;
for i=1:size(B1,2)
  B1c = 10*B1c+B1(i);
end

B2 = FILE(218); %get FRAC_THRESHOLD value
B2 = int32(cell2mat(B2)); %convert to string
B2 = B2-48; %convert to int32
B2c = 0;
for i=1:size(B2,2)
  B2c = 10*B2c+B2(i);
end


B3 = FILE(221); %get FRAC_ALPHA value
B3 = int32(cell2mat(B3)); %convert to string
B3 = B3-48; %convert to int32
B3c = 0;
for i=1:size(B3,2)
  B3c = 10*B3c+B3(i);
end


B4 = FILE(224); %get FRAC_STAGE_THRESHOLD value
B4 = int32(cell2mat(B4)); %convert to string
B4 = B4-48; %convert to int32
B4c = 0;
for i=1:size(B4,2)
  B4c = 10*B4c+B4(i);
end

FRAC_WEIGTH = 2^B1c;
FRAC_THRESHOLD = 2^B2c;
FRAC_ALPHA = 2^B3c;
FRAC_STAGE_THRESHOLD = 2^B4c;

while ~feof(fid)
```

```
if (strfind(c,'<!-- stage'))
 if (stg > 0)
  fdisp(fid3,num2str(nos));
 endif
 stg=stg+1
 nos=0;
endif;
if (strfind(c,'<!-- tree'))
 nos=nos+1;
 c=fgetl(fid); #<_>
 c=fgetl(fid); #<!-- root node -->
 c=fgetl(fid); #<feature>
 c=fgetl(fid); #<rects>
 c=fgetl(fid); #<_>x x x x x.</_>
 rect=0;
 while (strfind(c,'<_>'))
   c=strrep(c,'>','');
   c=strrep(c,'<','');
   c=strrep(c,'_','');
   c=strrep(c,'/','');
   c=strrep(c,'.','');
   c=strrep(c,"rects",'');
   x=strsplit(c);
   v1=cell2mat(x(1,2));
   v2=cell2mat(x(1,3));
   v3=cell2mat(x(1,4));
   v4=cell2mat(x(1,5));
   v5=num2str(FRAC_WEIGTH * str2double(cell2mat(x(1,6))));
   fdisp(fid2,v1);
   fdisp(fid2,v2);
   fdisp(fid2,v3);
   fdisp(fid2,v4);
   fdisp(fid2,v5);
   rect=rect+1;
   c=fgetl(fid); #tilted
 end
 if (rect==2)
   fdisp(fid2,0);
   fdisp(fid2,0);
   fdisp(fid2,0);
   fdisp(fid2,0);
   fdisp(fid2,0);
 endif
 c=fgetl(fid); #threshold
   c=strrep(c,'<','');
   c=strrep(c,'>','');
   c=strrep(c,'/','');
   c=strrep(c,' ','');
   c=strrep(c,"threshold",'');
   v4 = FRAC_THRESHOLD*str2double(c);
   if (v4>0)
     v5=num2str(uint32(v4));
   else
     v5=num2str(int32(v4));
   endif
   fdisp(fid2,v5);
 c=fgetl(fid); #alpha1
   c=strrep(c,'<','');
   c=strrep(c,'>','');
   c=strrep(c,'/','');
   c=strrep(c,' ','');
   c=strrep(c,"left_val",'');
   v4 = FRAC_ALPHA*str2double(c);
```

```
      if (v4>0)
        v5=num2str(uint32(v4));
      else
        v5=num2str(int32(v4));
      endif
      fdisp(fid2,v5);
    c=fgetl(fid); #alpha2
      c=strrep(c,'<','');
      c=strrep(c,'>','');
      c=strrep(c,'/','');
      c=strrep(c,' ','');
      c=strrep(c,'_','');
      c=strrep(c,"rightval",'');
      c=strrep(c,"trees",'');
      v4 = FRAC_ALPHA*str2double(c);
      if (v4>0)
        v5=num2str(uint32(v4));
      else
        v5=num2str(int32(v4));
      endif
      fdisp(fid2,v5);
    c=fgetl(fid);
    if (strfind(c,'<stage_threshold>'))
      c=strrep(c,'<','');
      c=strrep(c,'>','');
      c=strrep(c,'/','');
      c=strrep(c,' ','');
      c=strrep(c,"stage_threshold",'');
      v4 = FRAC_STAGE_THRESHOLD*str2double(c);
      if (v4>0)
        v5=num2str(uint32(v4));
      else
        v5=num2str(int32(v4));
      endif
      fdisp(fid2,v5);
    endif
  endif
  c=fgetl(fid);
end
fdisp(fid3,num2str(nos));
fdisp(fid3,num2str(stg));
fclose(fid);
fclose(fid2);
fclose(fid3);
disp(max);
```

G.5 Códigos da validação funcional de um detector de faces

Código G.7: Makefile

```
ifeq ("$(UVMC_HOME)","")
$(error ERROR: UVMC_HOME environment variable is not defined)
endif

# definitions only needed for OSCI:
```

```
SYSTEMC_HOME=/opt/rede/systemc-2.3.0
SYSC_COPTS=-I$(SYSTEMC_HOME)/include
SYSC_LOPTS=-L$(SYSTEMC_HOME)/lib-linux64 -Wl,-rpath -Wl,$(SYSTEMC_HOME)/lib-linux64 -lsystemc

# definitions needed for OpenCV:
CV_COPTS=`pkg-config --cflags opencv`
CV_LOPTS=`pkg-config --libs opencv`

# definitions only needed for VCS:
VCS_UVMC_SC_OPTS=-tlm2 -cflags "-g -I. -I$(VCS_HOME)/etc/systemc/tlm/include/tlm/tlm_utils -I$(UVMC_HOME)
    /src/connect/sc" $(UVMC_HOME)/src/connect/sc/uvmc.cpp $(UVM_HOME)/src/dpi/uvm_dpi.cc


VCS_UVMC_SV_OPTS=-q -sverilog -ntb_opts uvm -timescale=1ns/1ps +define+UVM_OBJECT_MUST_HAVE_CONSTRUCTOR +
    incdir+$(UVM_HOME)/src+$(UVM_HOME)/src/vcs+$(UVMC_HOME)/src/connect/sv +define+
    UVM_OBJECT_MUST_HAVE_CONSTRUCTOR $(UVM_HOME)/src/uvm_pkg.sv $(UVM_HOME)/src/vcs/*.sv $(UVMC_HOME)/src
    /connect/sv/uvmc_pkg.sv

VCS_UVMC_OPTS=-q -sysc=deltasync -lca -sysc -debug_pp -timescale=1ns/1ps uvm_custom_install_recording
    sc_main $(TOP)

VCS_SIMV= +UVM_NO_RELNOTES +UVM_TR_RECORD +UVM_TESTNAME=$(TEST)

# definitions for files
TOP = top
REFMOD = top
TEST= test
GCC ?= g++
HEADERS := image.h haar.h


vcs: call_opencv.o image.o haar.o rectangles.o
        g++ -c $(CV_COPTS) $(HEADERS) external.cpp
        syscan $(VCS_UVMC_SC_OPTS) $(REFMOD).cpp
        vlogan $(VCS_UVMC_SV_OPTS) $(TOP).sv
        vcs    $(VCS_UVMC_OPTS) $(CV_LOPTS) $(TOP) image.o haar.o rectangles.o external.o call_opencv.o
        ./simv $(VCS_SIMV) +UVM_VERBOSITY=MEDIUM

call_opencv.o: call_opencv.cpp
        g++ $(CV_COPTS) -c $^

image.o: image.c $(HEADERS)
        $(GCC) -o $@ -c $<

haar.o: haar.cpp $(HEADERS)
        $(GCC) -o $@ -c $<

rectangles.o: rectangles.cpp $(HEADERS)
        $(GCC) -o $@ -c $<

clean:
        rm -rf a.out *.o *.gch *.jpg report.csv # simulation output file
        rm -rf INCA_libs irun.log ncsc.log # ius
        rm -rf work certe_dump.xml transcript .mgc_simple_ref .mgc_ref_nobuff .mgc_ref_oct # mgc
        rm -rf csrc simv simv.daidir ucli.key .vlogansetup.args .vlogansetup.env .vcs_lib_lock simv.vdb
            AN.DB vc_hdrs.h *.diag *.vpd *tar.gz # vcs

# Wave form
view_waves:
        dve -vpd vcdplus.vpd &
```

Código G.8: Códigos em UVM/SystemVerilog da Verificação Funcional de um sistema de detecção de faces utilizando o algoritmo de Viola Jones

```
//==================== frame_tr.sv ====================
class frame_tr extends uvm_sequence_item;
    string message;
    longint unsigned a;

    `uvm_object_utils_begin(frame_tr)
        `uvm_field_string(message, UVM_DEFAULT|UVM_HEX)
        `uvm_field_int(a, UVM_DEFAULT)
    `uvm_object_utils_end

    function new(string name="frame_tr");
        super.new(name);
    endfunction: new
endclass: frame_tr
//=====================================

//==================== faceObjects_tr.sv ====================
class faceObjects_tr extends uvm_sequence_item;
    string message;
    int numFaces;
    longint unsigned x;
    longint unsigned y;
    longint unsigned l;

    `uvm_object_utils_begin(faceObjects_tr)
        `uvm_field_string(message, UVM_DEFAULT|UVM_HEX)
        `uvm_field_int(numFaces, UVM_DEFAULT|UVM_HEX)
        `uvm_field_int(x, UVM_DEFAULT|UVM_HEX)
        `uvm_field_int(y, UVM_DEFAULT|UVM_HEX)
        `uvm_field_int(l, UVM_DEFAULT|UVM_HEX)

    `uvm_object_utils_end

    function new(string name="faceObjects_tr");
        super.new(name);
    endfunction: new
endclass: faceObjects_tr
//=====================================

//==================== frame_sequence.sv ====================
import "DPI-C" context function string read_message(string msg);
import "DPI-C" context function longint unsigned readframe_dpi(string filename);

class frame_sequence extends uvm_sequence #(frame_tr);
    `uvm_object_utils(frame_sequence)

    function new(string name="frame_sequence");
        super.new(name);
    endfunction: new

    int data_file, scan_file;
    string filename;
    longint unsigned a;

    function void open_file();
      data_file = $fopen("database.txt", "r");
      if(data_file == 0)begin
         $display("file could not be open!!!");
      end
```

```
     endfunction: open_file

    task body();
        frame_tr tr;
        open_file();
        while(!$feof(data_file))begin
            scan_file = $fscanf(data_file, "%s\n", filename);
            tr = frame_tr::type_id::create("tr");
            start_item(tr);
            tr.message = filename;
            tr.a = readframe_dpi(filename);
            finish_item(tr);
        end
    endtask: body
endclass: frame_sequence
//=====================================

//==================== frame_sequencer.sv ====================
class frame_sequencer extends uvm_sequencer #(frame_tr);
    `uvm_component_utils(frame_sequencer)

    function new (string name = "frame_sequencer", uvm_component parent = null);
        super.new(name, parent);
    endfunction
endclass: frame_sequencer
//=====================================

//==================== frame_driver.sv ====================
class frame_driver extends uvm_driver #(frame_tr);
    frame_tr tr;

    `uvm_component_utils(frame_driver)
    uvm_analysis_port #(frame_tr) item_collected_port;

   function new(string name, uvm_component parent);
        super.new(name, parent);
        item_collected_port = new ("item_collected_port", this);
    endfunction

    virtual task run_phase(uvm_phase phase);
      forever begin
        seq_item_port.get_next_item(tr);
        begin_tr(tr, "frame_driver");
          item_collected_port.write(tr);
        end_tr(tr);
        seq_item_port.item_done();
      end
    endtask
endclass: frame_driver
//=====================================

//==================== comparator.sv ====================
typedef struct{
  string message;
  int numFaces;
  longint unsigned x;
  longint unsigned y;
  longint unsigned l;
}face_tr;

import "DPI-C" context function void compareFaces(face_tr faceSC, face_tr faceCV);

class comparator #(type T = faceObjects_tr) extends uvm_scoreboard;
```

```
   typedef comparator #(T) this_type;
   `uvm_component_utils(this_type)

    uvm_tlm_fifo #(T) from_refmod;
    uvm_tlm_fifo#(T) from_refmod_low;

     T tr1, tr2;
     face_tr faceSC, faceCV;
     int match, mismatch;

    function new(string name, uvm_component parent);
      super.new(name, parent);
      from_refmod = new("from_refmod", null, 1);
      from_refmod_low = new("from_refmod_low", null, 1);
      tr1 = new("tr1");
      tr2 = new("tr2");
    endfunction

    function void connect_phase(uvm_phase phase);
        uvmc_tlm1 #(T)::connect(from_refmod.put_export, "refmod_i.out");
        uvmc_tlm1#(T)::connect(from_refmod_low.put_export, "refmod_low_i.out");
    endfunction: connect_phase

    task run_phase(uvm_phase phase);
      forever begin
        from_refmod.get(tr1);
        faceCV.message = tr1.message;
        faceCV.numFaces = tr1.numFaces;
        faceCV.x = tr1.x;
        faceCV.y = tr1.y;
        faceCV.l = tr1.l;

        from_refmod_low.get(tr2);
        faceSC.message = tr2.message;
        faceSC.numFaces = tr2.numFaces;
        faceSC.x = tr2.x;
        faceSC.y = tr2.y;
        faceSC.l = tr2.l;
        compare();
      end
    endtask: run_phase

    virtual function void compare();
      compareFaces(faceSC, faceCV);
    endfunction: compare

endclass: comparator
//=====================================

//================== env.sv ==================
`include "comparator.sv"

class env extends uvm_env;
    frame_sequencer frame_sqr;
    frame_driver frame_drv;
    comparator #(faceObjects_tr) comp;

    uvm_tlm_analysis_fifo #(frame_tr) to_refmod;
    uvm_tlm_analysis_fifo #(frame_tr) to_refmod_low;

    `uvm_component_utils(env)

    function new(string name, uvm_component parent = null);
```

```
        super.new(name, parent);
        to_refmod = new("to_refmod", this);
        to_refmod_low = new("to_refmod_low", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        frame_sqr = frame_sequencer::type_id::create("frame_sqr", this);
        frame_drv = frame_driver::type_id::create("frame_drv", this);
        comp = comparator #(faceObjects_tr)::type_id::create("comp", this);
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        frame_drv.seq_item_port.connect(frame_sqr.seq_item_export);
        frame_drv.item_collected_port.connect(to_refmod.analysis_export);
        uvmc_tlm1 #(frame_tr)::connect(to_refmod.get_export, "refmod_i.in");

        frame_drv.item_collected_port.connect(to_refmod_low.analysis_export);
        uvmc_tlm1 #(frame_tr)::connect(to_refmod_low.get_export, "refmod_low_i.in");
    endfunction

    virtual function void end_of_elaboration_phase(uvm_phase phase);
        super.end_of_elaboration_phase(phase);
    endfunction

endclass
//=====================================

//================== test.sv ==================
class test extends uvm_test;
  env env_h;
  frame_sequence frame_seq;

  `uvm_component_utils(test)

  function new(string name, uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_h = env::type_id::create("env_h", this);
    frame_seq = frame_sequence::type_id::create("frame_seq", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    frame_seq.start(env_h.frame_sqr);
    phase.drop_objection(this);
  endtask: run_phase

endclass: test
//=====================================

//================== top.sv ==================
import uvm_pkg::*;
import uvmc_pkg::*;

`include "uvm_macros.svh"
`include "frame_tr.sv"
`include "faceObjects_tr.sv"
`include "frame_sequence.sv"
`include "frame_sequencer.sv"
```

```
`include "frame_driver.sv"
`include "env.sv"
`include "test.sv"

//Top
module top;

  initial begin
    `ifdef INCA
       $recordvars();
    `endif
    `ifdef VCS
       $vcdpluson;
    `endif
    `ifdef QUESTA
       $wlfdumpvars();
       set_config_int("*", "recording_detail", 1);
    `endif

    run_test("test");
  end
endmodule
```

Código G.9: Códigos em C/C++/SystemC da Verificação Funcional de um sistema de detecção de faces utilizando o algoritmo de Viola Jones

```
//=================== definitions.h ====================
#ifndef DEFINITIONS__H
#define DEFINITIONS__H

#define HEIGHT 1200//1082//1200//340
#define WIDTH 900//1600//900//394
#define CHANNELS 3
#endif
//======================================

//=================== external.cpp ====================
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>


using namespace std;



extern "C" const char* read_message(const char* message){
   string msg;
   ifstream myfile(message);
   if(myfile.is_open()){
      getline(myfile, msg);
      return msg.c_str();
   }
   else
     cout << "unable to open file";
}
//======================================

//=================== call_opencv.cpp ====================
#include <stdio.h>
```

```
#include <cv.h>
#include <highgui.h>

#include "image.h"
#include "haar.h"

using namespace cv;
using namespace std;

#include "definitions.h"

struct center{
       int x;
       int y;
       };

struct sv_faceObjects_tr{
  string message;
  int numFaces;
  unsigned long long x;
  unsigned long long y;
  unsigned long long l;

};

struct faceObjects_tr{
  string message;
  int numFaces;
  int* x;
  int* y;
  int* l;

  faceObjects_tr() = default;

  explicit faceObjects_tr(const sv_faceObjects_tr & t) :
    message(t.message),
    numFaces(t.numFaces),
    x(reinterpret_cast<int*>(t.x)),
    y(reinterpret_cast<int*>(t.y)),
    l(reinterpret_cast<int*>(t.l)) //((int*)(t.l))
  {}

  explicit operator sv_faceObjects_tr() const
  {
    return sv_faceObjects_tr{
        message,
        numFaces,
        reinterpret_cast<unsigned long long>(x),
        reinterpret_cast<unsigned long long>(y),
        reinterpret_cast<unsigned long long>(l)
        };
  }
};

struct img_size{
    int channels;
    int height;
    int width;
};

img_size allocate_image(const char* filename){
    Mat mat = imread(filename, CV_LOAD_IMAGE_COLOR);
    img_size size;
```

```
    size.channels = (int)mat.channels();
    size.height = (int)mat.rows;
    size.width = (int)mat.cols;
    return size;
}

faceObjects_tr allocate_faces(std::vector <MyRect> faces){

      faceObjects_tr faceObj;
      int *x, *y, *l;
      faceObj.numFaces = faces.size();
          x = (int*)malloc(sizeof(int)*faceObj.numFaces);
      y = (int*)malloc(sizeof(int)*faceObj.numFaces);
      l = (int*)malloc(sizeof(int)*faceObj.numFaces);

          for(int i = 0; i < faceObj.numFaces; i++){
             x[i] = faces[i].x;
             y[i] = faces[i].y;
             l[i] = faces[i].height;
          }
          faceObj.x = x;
          faceObj.y = y;
          faceObj.l = l;
      return faceObj;
}

faceObjects_tr faceDetect(const char* filename){
  Mat img = imread(filename);
  std::vector<Rect> faces;
  Mat frame_gray;

  String face_cascade_name = "haarcascade_frontalface_default.xml";
  CascadeClassifier face_cascade;
  if( !face_cascade.load( face_cascade_name ) ){ printf("--(!)Error loading\n"); }
  cvtColor( img, frame_gray, CV_BGR2GRAY );
  equalizeHist( frame_gray, frame_gray );
  face_cascade.detectMultiScale( frame_gray, faces, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE, Size(30, 30) );

  //printf("DETECT %d faces\n", faces.size());
  for( size_t i = 0; i < faces.size(); i++ )
  {
    rectangle(img, faces[i], CV_RGB(0, 255,0), 1);
    //printf("%d %d %d %d\n", faces[i].x, faces[i].y, faces[i].height, faces[i].width);
  }

  faceObjects_tr faceObj;
  int *x, *y, *l;
  faceObj.numFaces = faces.size();
  x = (int*)malloc(sizeof(int)*faces.size());
  y = (int*)malloc(sizeof(int)*faces.size());
  l = (int*)malloc(sizeof(int)*faces.size());

  //printf("%d\n", faceObj.numFaces);
  for( int i = 0; i < faceObj.numFaces; i++ ){
    x[i] = faces[i].x;
    y[i] = faces[i].y;
    l[i] = faces[i].height;
    //printf("%d %d %d\n", x[i], y[i], l[i]);
  }
  /*****/
  faceObj.x = x;
  faceObj.y = y;
  faceObj.l = l;
```

```
  //imwrite("out.png",img);
  return faceObj;
}

extern "C" void compareFaces(faceObjects_tr faceSC, faceObjects_tr faceCV){
  int i, j;
  int match;
  int a, b, c, d;
  float m_geometric_mean, m_arithmetic_mean;
  float m_geometric_mean_valid, m_arithmetic_mean_valid;
  int geometric_mean_valid, arithmetic_mean_valid;
  Mat mat = imread(faceSC.message.c_str(), 0); //read image in gray scale
  FILE *file = fopen("report.csv", "a");
  if(faceCV.numFaces && faceSC.numFaces){
    for(i = 0; i < faceCV.numFaces; i++){
      match = 0;
      geometric_mean_valid = 0;
      arithmetic_mean_valid = 0;
      for(j = 0; j < faceSC.numFaces; j++){
        a = max(faceCV.x[i],faceSC.x[j]);
        b = min(faceCV.x[i]+faceCV.l[i],faceSC.x[j]+faceSC.l[j]);
        c = max(faceCV.y[i],faceSC.y[j]);
        d = min(faceCV.y[i]+faceCV.l[i],faceSC.y[j]+faceSC.l[j]);
        m_geometric_mean = (b>a)&&(d>c) ? (float)(b-a)*(d-c)/(faceCV.l[i]*faceSC.l[j]) : 0;
        m_arithmetic_mean = (b>a)&&(d>c) ? (float) 2*(b-a)*(d-c)/(faceCV.l[i]*faceCV.l[i]+faceSC.l[j]*
            faceSC.l[j]) : 0;

        if(m_geometric_mean > 0.75){
         m_geometric_mean_valid = m_geometric_mean;
         geometric_mean_valid = 1;
        }
        if(m_arithmetic_mean > 0.75){
         m_arithmetic_mean_valid = m_arithmetic_mean;
         arithmetic_mean_valid = 1;
        }
        if(geometric_mean_valid || arithmetic_mean_valid)
          match = 1;
      }
    }
  }

  int fpCV, fnCV;
  fpCV = faceCV.numFaces > 1 ? faceCV.numFaces-1 : 0;
  fnCV = !faceCV.numFaces ? 1 : 0;
  printf("CV: false positives = %d false negatives = %d\n", fpCV, fnCV);


  int fpSC, fnSC;

  fpSC = faceSC.numFaces > 1 ? faceSC.numFaces-1 : 0;
  fnSC = !faceSC.numFaces ? 1 : 0;

  printf("SC: false positives = %d false negatives = %d\n", fpSC, fnSC);
  fprintf(file, "%d, %d, %d, %d, %f, %f\n", fpCV, fnCV, fpSC, fnSC, m_geometric_mean_valid,
      m_arithmetic_mean_valid);
  fclose(file);

  vector<Rect> faces1(faceCV.numFaces), faces2(faceSC.numFaces);
  for(i = 0; i < faceCV.numFaces; i++){
      faces1[i] = Rect(faceCV.x[i], faceCV.y[i], faceCV.l[i], faceCV.l[i]);
      rectangle(mat, faces1[i], CV_RGB(255, 255,255), 1);
  }
```

```
  for(i = 0; i < faceSC.numFaces; i++){
      faces2[i] = Rect(faceSC.x[i], faceSC.y[i], faceSC.l[i], faceSC.l[i]);
      rectangle(mat, faces2[i], CV_RGB(255, 255,255), 1);
  }
  string str = faceSC.message.c_str();
  //substr 13 for /home/nelson/
  //+6=19 for bioid.txt (/home/nelson/BIOID/)
  //4 for myfile.txt (img/filename)
  str = str.substr(19,str.length());
  string path;
  path = "./IMG/"+str;
  cout << path << endl;
  if(!fnCV && fnSC)
    imwrite(path, mat);
}

extern "C" unsigned long long readframe_dpi(const char* filename )
{
  Mat mat = imread(filename, CV_LOAD_IMAGE_COLOR);
  uint8_t *pp = (uint8_t *)malloc(sizeof(uint8_t)*mat.channels()*mat.rows*mat.cols);

  for(int i = 0; i < mat.rows; i++)
    for(int j = 0; j < mat.channels()*mat.cols; j++)
      pp[i*mat.channels()*mat.cols+j] =  mat.data[i*mat.step+j];

  Mat image(mat.rows, mat.cols, CV_8UC3, pp);
  return (unsigned long long)pp;
}

uint8_t* readframe(const char* filename)
{
  Mat mat = imread(filename, CV_LOAD_IMAGE_COLOR);
  Mat gray_image;
  cvtColor( mat, gray_image, CV_BGR2GRAY );
  //printf("readframe-> %d %d\n", mat.rows, mat.cols);
  uint8_t *pp = (uint8_t *)malloc(sizeof(uint8_t*)*mat.rows*mat.cols);
  for(int i = 0; i < mat.rows*mat.cols; i++)
    ((uint8_t*)pp)[i] = gray_image.data[i];

  return (uint8_t*)pp;
}


void writeframe(MyImage *image, const char* filename){
  Mat img(image->height, image->width, CV_8UC1, image->data);

  string str = filename;
  str = str.substr(4,str.length());
  string path;
  path = "./"+str;
  imwrite(path, img);
}
//=====================================

//================== refmod.cpp ==================
#include "systemc.h"
#include "tlm.h"
#include <string>

using namespace std;

using namespace tlm;
struct img_size{
```

```
    int channels;
    int height;
    int width;
};

typedef struct faceObjects_tr{
  string message;
  int numFaces;
  unsigned long long x;
  unsigned long long y;
  unsigned long long l;
}faceObjects_tr;


faceObjects_tr faceDetect(const char*);

struct tr {
  string message;
  unsigned long long a;
};

#include "uvmc.h"
using namespace uvmc;
UVMC_UTILS_2(tr, message, a)
UVMC_UTILS_5(faceObjects_tr, message, numFaces, x, y, l)

SC_MODULE(refmod) {
  sc_port<tlm_get_peek_if<tr> > in;
  sc_port<tlm_put_if<faceObjects_tr> > out;

  void p() {

    tr tr;
    faceObjects_tr faceObj;
    while(1){
      tr = in->get();

      faceObj = faceDetect((tr.message).c_str());
      faceObj.message = tr.message;
      //cout <<"packet_out " << faceObj.numFaces <<endl;
      //cout <<"refmod: " <<faceObj.message <<"\n";
      out->put(faceObj);
      //printf("refmod: numFaces = %d\n", faceObj.numFaces);
      //for(int i = 0; i < faceObj.numFaces; i++)
        //printf("refmod: x[%d] = %d y[%d] = %d l[%d] = %d\n", i, ((int*)faceObj.x)[i], i, ((int*)faceObj
            .y)[i], i, ((int*)faceObj.l)[i]);
    }
  }
  SC_CTOR(refmod): in("in"), out("out") { SC_THREAD(p); }
};
//=====================================

//==================== refmod_low.cpp ====================
#include "image.h"
#include "haar.h"
#include "definitions.h"

img_size allocate_image(const char* filename);
uint8_t* readframe(const char*);
void writeframe(MyImage *, const char*);
faceObjects_tr faceDetect(const char* filename);
faceObjects_tr allocate_faces(std::vector<MyRect> faces);
```

```
SC_MODULE(refmod_low){
  sc_port<tlm_get_peek_if<tr> > in;
  sc_port<tlm_put_if<faceObjects_tr> > out;

  void p() {

    tr tr;
    while(1){
      tr = in->get();

    const char* filename = (tr.message).c_str();
    int mode = 1;
        int i;
    /* detection parameters */
        float scaleFactor = 1.1;
        int minNeighbours = 1;

    //printf("-- entering main function --\r\n");

        //printf("-- loading image --\r\n");
        MyImage imageObj;
        MyImage *image = &imageObj;
        faceObjects_tr faceObj;
    img_size size;
    size = allocate_image(filename);
    image->height = size.height;
        image->width = size.width;
        int size_tr = size.channels*image->height*image->width;
        image->data = (uint8_t*)malloc(sizeof(uint8_t)*size_tr);
        tr.a = (unsigned long long)readframe(filename);
        for(i = 0; i < size_tr; i++)
          image->data[i] = ((uint8_t*)tr.a)[i];

        //printf("-- loading cascade classifier --\r\n");
    myCascade cascadeObj;
        myCascade *cascade = &cascadeObj;
        MySize minSize = {20, 20};
        MySize maxSize = {0, 0};

        /* classifier properties */
        cascade->n_stages=25;//22;//25;
        cascade->total_nodes=2913;//2135;//2913;
        cascade->orig_window_size.height = 24;
        cascade->orig_window_size.width = 24;

    readTextClassifier();
        vector<MyRect> faces;

        //printf("-- detecting faces --\r\n");

        faces = detectObjects(image, minSize, maxSize, cascade, scaleFactor, minNeighbours);
        //printf("RFM_LOW: after detect!!!\n\n\n");
        for(i = 0; i < faces.size(); i++ )
        {
                MyRect r = faces[i];
                drawRectangle(image, r);
                //printf("\nObjects %d %d %d\n", faces[i].x, faces[i].y, faces[i].height);
        }

          faceObj = allocate_faces(faces);
          faceObj.message = tr.message;
          //cout <<"refmod_low: " <<tr.message <<"\n";
      //printf("refmod_low: numFaces = %d\n", faceObj.numFaces);
```

```
          //for(int i = 0; i < faceObj.numFaces; i++)
            // printf("refmod_low: x[%d] = %d y[%d] = %d l[%d] = %d\n", i, ((int*)faceObj.x)[i], i, ((int
                *)faceObj.y)[i], i, ((int*)faceObj.l)[i]);

      out->put(faceObj);
      //writeframe(image, filename);
      releaseTextClassifier();
          free(image->data);
    }
  }
  SC_CTOR(refmod_low): in("in"), out("out") { SC_THREAD(p); }
};
//=====================================

//==================== top.cpp ====================
#include "refmod.cpp"
#include "refmod_low.cpp"

int sc_main(int argc, char* argv[]) {

  refmod  refmod_i("refmod_i");
  refmod_low refmod_low_i("refmod_low_i");

  uvmc_connect(refmod_i.in, "refmod_i.in");
  uvmc_connect(refmod_low_i.in, "refmod_low_i.in");
  uvmc_connect(refmod_i.out, "refmod_i.out");
  uvmc_connect(refmod_low_i.out, "refmod_low_i.out");

  sc_start();
  return(0);
}
```