Universidade de Brasília - UnB

Faculdade do Gama - FGA

Engenharia Eletrônica

Implementação Eficiente de um Módulo de Detecção de Movimento em Vídeo para SoC FPGA

Autor: Leonardo Brandão Borges de Freitas

Orientador: Prof. Dr. Daniel Mauricio Muñoz Arboleda (FGA/UnB)

Brasília, DF

2022

Leonardo Brandão Borges de Freitas

Implementação Eficiente de um Módulo de Detecção de Movimento em Vídeo para SoC FPGA

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade do Gama - FGA

Orientador: Prof. Dr. Daniel Mauricio Muñoz Arboleda (FGA/UnB)

Brasília, DF

2022

Leonardo Brandão Borges de Freitas
Implementação Eficiente de um Módulo de Detecção de Movimento em Vídeo para SoC FPGA/ Leonardo Brandão Borges de Freitas. – Brasília, DF, 2022-
72 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Daniel Mauricio Muñoz Arboleda (FGA/UnB)

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade do Gama - FGA , 2022.

1. Detecção de Movimento em Vídeo. 2. FPGA. I. Prof. Dr. Daniel Mauricio Muñoz Arboleda (FGA/UnB). II. Universidade de Brasília. III. Faculdade do Gama. IV. Implementação Eficiente de um Módulo de Detecção de Movimento em Vídeo para SoC FPGA

CDU 02:141:005.6

Leonardo Brandão Borges de Freitas

Implementação Eficiente de um Módulo de Detecção de Movimento em Vídeo para SoC FPGA

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 11 de maio de 2022:

Prof. Dr. Daniel Mauricio Muñoz Arboleda (FGA/UnB)
Orientador

Prof. Dr. Gilmar Silva Beserra (FGA/UnB)
Convidado 1

Prof. Dr. Jones Yudi Mori Alves da Silva (ENM/FT/UNB)
Convidado 2

Brasília, DF
2022

Agradecimentos

Agradeço a minha família por proporcionar um ambiente de criação com muito amor, respeito e liberdade o que ajudou a me formar como ser humano. Agradeço aos meus professores e colegas pela participação ativa em minha formação escolar o que me ajudou a dar valor na educação e no conhecimento científico. Agradeço especialmente ao meu orientador Prof. Dr. Daniel por me ensinar e motivar na área de eletrônica digital e *design* de *hardware*, inclusive por me auxiliar na concepção deste trabalho. Agradeço também a toda a equipe da Associação GigaCandanga, principalmente ao meu amigo e tutor Dr. Paulo Ângelo que me incentiva todos os dias a investir em minha carreira profissional.

Resumo

Este trabalho propõe uma implementação eficiente para detecção de movimento em vídeo a partir de otimizações na técnica de diferenciação temporal e fatiamento das imagens em regiões de *pixel*, com objetivo de evitar falsos positivos na detecção de movimento causado pela projeção de sombras no cenário monitorado. O algoritmo desenvolvido foi descrito como uma máquina de estados finita, ou da sigla em inglês FSM, e validado primeiramente em *software* através de uma aplicação Python. Em seguida, o algoritmo foi codificado para linguagem de descrição de *hardware*, VHDL, com o auxilio da plataforma de desenvolvimento Vivado. O resultado da simulação comportamental indicou que o algoritmo proposto compara um par de *frames*, com dimensões de 640x480p, em menos de 13 ms, se garantido um novo *pixels* a cada 10ns, ou seja, conseguindo operar a quase 160fps. Em termos de implementação em *hardware* o algoritmo embarcado apresentou consumo energético de aproximadamente 0,02W e, em termos de consumo de recursos, utilizou apenas 12% das LUTs e 11% dos Flip-Flops disponíveis no SoC FPGA modelo PYNQ-Z2, utilizada no protótipo do projeto. Todavia, devido a utilização de um barramento AXI-Lite entre os ambientes de *software*, que captura e pre-processa as imagens, e *hardware*, que aplica o algoritmo de detecção de movimento, a velocidade de envio dos *pixels* de um ambiente para o outro se mostrou insuficiente para alcançar a velocidade da simulação comportamental. Porém, desconsiderando a velocidade de processamento, os resultados se mostraram semelhantes tanto na FSM em *software* via aplicação Python, quanto na simulação comportamental via Vivado e até mesmo quando embarcado na plataforma PYNQ-Z2, o que valida o funcionamento do algoritmo proposto.

Palavras-chave: Detecção de movimento em vídeo, Diferenciação temporal, FPGA, PYNQ.

Abstract

This project offers an efficient implementation for movement detection on video, based on the optimization of the temporal differencing technique and slicing of pixel area of images, aiming to avoid false positives on movement detection resultant of shadow projections in the monitored scenario. The algorithm developed was described as a finite state machine, or FSM, and validated first in software through a Python application. Then, the algorithm was coded in hardware description language, VHDL, utilizing the development platform Vivado. The result of the behavior simulation shows that the offered algorithm compares a pair of frames, with dimensions of 640x480p, in less than 13ms, if guaranteed a new pixel every 10ns, making it able to work at almost 160fps. In hardware implementation terms, the embedded algorithm shows energy consumption around 0,02W and, in resource consumption terms, it utilized only 12% of the LUTs and 11% of the Flip-Flops available in the SoC FPGA model PYNQ-Z2, used in the project prototype. Although, due to the use of a data bus AXI-Lite between software environments, that capture and pre-process images, and hardware, that applies the movement detection algorithm, the pixel's sending speed from an environment to another was proven insufficient to reach the almost 160fps of the behavior simulation. But, disregarding the processing speed, the results were similar, both in the software in FSM through Python application, in the behavior simulation through Vivado and even when embedded in the PYNQ-Z2 platform, validating that the algorithm works.

Key-words: video motion detection, temporal differencing, FPGA, PYNQ.

Lista de ilustrações

Lista de abreviaturas e siglas

SoC	*System on Chip*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
CI	Circuito Integrado
GPIO	*General Purpose Input/Output*
IEEE	*Institute of Electrical and Electronic Engineers*
CLB	*Configurable Logic Block*
LUT	*Look-up Table*
ASIC	*Application Specific Integrated Circuits*
HDL	*Hardware Description Language*
FPS	*Frames Per Second*
dpi	*dots per inch*
RTL	*Register Transfer Level*
FIFO	*First In First Out*
RTSP	*Real Time Streaming Protocol*
MVP	*Minimum Viable Product*
IoT	*Internet of Things*

Sumário

1 Introdução

1.1 Contextualização e Justificativa

A elevada densidade de habitantes em áreas urbanas tem intensificado problemas relacionados à segurança, mobilidade, saúde, meio ambiente, abastecimento, descarte de lixo, educação, desigualdade social, dentre outros. Por outro lado, a recente ascensão de tecnologias como *IoT* (*Internet of Things*) e visão computacional associadas ao processamento do grande fluxo de dados proveniente das metrópoles (*big data*) permitem a implementação de abordagens mais "inteligentes" e automatizadas àqueles problemas citados.

Não existe um consenso sobre o que de fato é uma cidade inteligente. Contudo, alguns pesquisadores associam o conceito ao investimento em tecnologias da informação e comunicação (TIC) (WEISS; BERNARDES; CONSONI, 2017). Além disso, outros pesquisadores afirmam que uma cidade inteligente é aquela capaz de impulsionar o crescimento econômico sustentável e incrementar a qualidade de vida de seus habitantes, com uma gestão inteligente dos recursos naturais e por meio de governança participativa (LAZZARETTI; SEHNEM; BENCKE, 2019).

Neste contexto, o projeto Campus Inteligente iniciado em 2018 pela Associação GigaCandanga visa de forma geral, e em primeiro momento, implementar tecnologias para coleta, processamento e armazenamento dos dados provenientes dos parâmetros ambientais e de usuários do campus Darcy Ribeiro da Universidade de Brasília, visando, principalmente, o incremento na qualidade de vida da população universitária. Não obstante, posteriormente, pretende-se expandir as soluções validadas na UnB para o Distrito Federal como um todo.

Indiferentemente dos conceitos adotados para definir o que é uma cidade inteligente, a infraestrutura de comunicação de dados, bem como as tecnologias utilizadas são fundamentais para implementar o conceito em questão. Para tanto, o projeto Campus Inteligente tem como objetivos:

1. Implementar conceitos de cidades inteligentes no espaço do campus Darcy Ribeiro da Universidade de Brasília a fim de testar as tecnologias e ferramentas existentes e efetivamente obter segurança, economia, acessibilidade e conforto nas dependências físicas do campus;

2. Contribuir com o engajamento de instituições e pesquisadores de diversas área a fim de consolidar um '*hub*' de desenvolvimento tecnológico para, inicialmente, instalar

soluções em *IoT*, visão computacional e *big data* no maior campus universitário de Brasília.

Tendo em vista os três pilares que compõem o escopo do projeto Campus inteligente: *IoT*, visão computacional e *big data*, tomou-se como ponto de partida o conceito de visão computacional, o que se concretizou no GigaView: Um sistema de monitoramento de vídeo inteligente, altamente escalável, que ao receber imagens de câmeras de vigilância, utiliza de algoritmos de *Machine Learning* para processar as imagens capturadas, resultando em produtos como, reconhecimento facial, reconhecimento de placas de automóveis, detecção de máscaras de proteção e detecção de aglomerações.

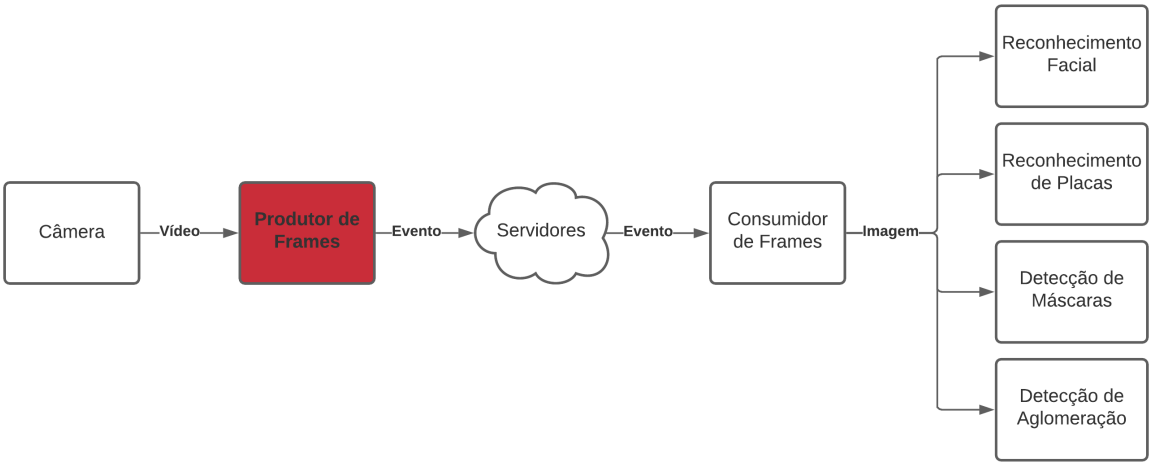


Figura 1 – GigaView: Fluxograma Geral do *Core*

Neste contexto, a figura 1 sintetiza o fluxo operacional das principais entidades e micro serviços do núcleo de visão computacional do GigaView. Logo após a câmera, encontra-se o micro serviço chamado **Produtor de Frames**. Sendo a primeira entidade do núcleo do sistema, o Produtor recebe o *streaming* de vídeo em tempo real e aplica um algoritmo de detecção de movimento sobre pares de *frames*. Se houver movimento, o evento em vídeo é registrado nos servidores em nuvem. Uma vez armazenado, este evento é disponibilizado aos **Consumidores de Frames**, os quais separam as imagens dos vídeos e enviam quadro a quadro aos serviços de detecção e reconhecimento de objetos.

Em primeiro momento, visando a concepção de um produto mínimo viável (MVP) para validar as funcionalidades do GigaView, tanto o Produtor, quanto o Consumidor de Frames foram escritos em Python. Contudo, buscando a escalabilidade do projeto em grandes proporções, otimizações são necessárias para atingir um melhor desempenho e um menor consumo computacional por parte destes micro serviços que o compõem o *core*, ou núcleo do sistema. Para tanto, este trabalho propõe uma implementação eficiente dos serviços do **Produtor de Frames**, mais especificamente propõe descrever em *hardware* o algoritmo de detecção de movimento que se mostrou muito custoso em *software* para máquinas de propósito geral. Em que, embarcado em uma máquina virtual Linux Ubuntu 20.04, com 4 VCPUs de um processador core I7 da Intel e 8 Gb de memória RAM DDR4, uma instância do **Produtor de Frames** consome aproximadamente 12,5% do processador e 2% da memória RAM.

Ou seja, em termos de CPU, uma máquina com as características descritas acima suportaria apenas 8 **Produtores de Frames** simultâneos. Porém, no contexto de cidades, centenas ou até mesmo milhares de câmeras devem ser monitoradas ao mesmo tempo, o que exigiria um elevado investimento em recursos computacionais apenas para capturar eventos em vídeo derivados de movimento em cena.

1.2 Produtor em alto nível

Em *software*, o **Produtor de Frames** orquestra duas *threads* computacionais: uma para capturar e detectar movimento e outra para formatar e enviar os eventos em vídeo para os servidores. A figura 2 apresenta as 5 etapas deste micro serviço.

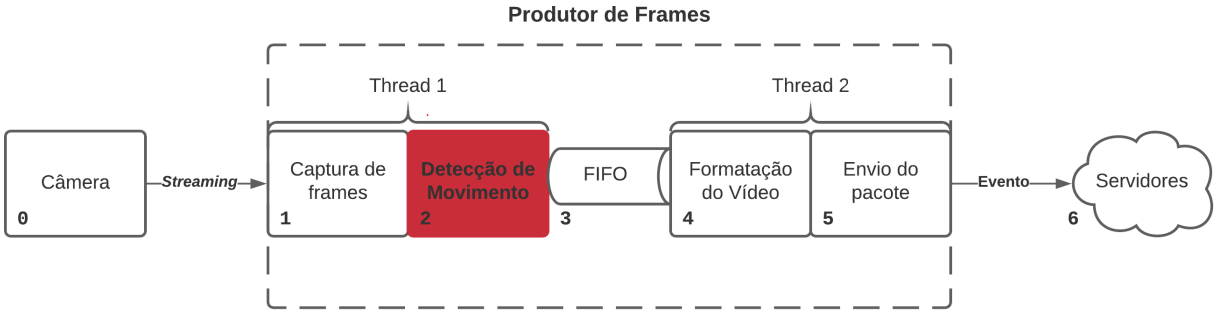


Figura 2 – Diagrama em alto nível do *Produtor de Frames*

Ainda da figura 2, cada Produtor atende a apenas uma câmera, em 0, que disponibiliza seu *streaming* de vídeo através de um servidor RTSP (*Real Time Streaming Protocol*). A primeira *thread*, em 1, captura e armazena em memória um par de imagens por vez. A cada captura, os dois quadros são comparados, em 2, através da diferenciação temporal, que está melhor detalhada nas seções 2.1.5 e 3, determinando se houve ou não movimento a partir de um limiar pre definido. Em caso afirmativo, um intervalo de imagens direto do *streaming* da câmera é enfileirado em uma FIFO (*First In First Out*), em 3. Vale ressaltar que este intervalo de imagens enfileiradas é configurado em segundos de acordo com a demanda do usuário. Não obstante, ao receber os quadros na saída da fila, a segunda *thread* formata um arquivo de vídeo (MP4) com o intervalo de imagens, em 4, e envia o pacote aos servidores em 5. Este pacote contem não apenas o *payload* como vídeo, mas também o nome do arquivo como aviso para as demais entidades, bem como o registro temporal do evento para o banco de metadados.

As etapas 3, 4 e 5 acontecem apenas quando um movimento for identificado. Em outro caso permanecem em estado inativo e por isso exigem da CPU (*Central Process Unit*) de forma extraordinária. No entanto, as etapas 1 e 2 trabalham ininterruptamente, de forma dispendiosa para o processador. Uma das formas de reduzir a carga sobre o processador é diminuindo a velocidade de captura, em 1, espaçando no tempo o armazenamento em memória dos pares de imagens capturadas. De qualquer forma, há um consumo de memória associado ao armazenamento dos quadros, bem como um consumo associado ao tempo de processamento para o algoritmo de detecção de movimento, o que, como já citado anteriormente, exige uma capacidade computacional inviável para a escalabilidade do GigaView com um número elevado câmeras.

O uso de arquiteturas reconfiguráveis tais como FPGAs (*Field Programmable Gate Arrays*) permitem explorar o paralelismo intrínseco dos algoritmos de detecção de movimento em vídeo, acelerando a sua execução diretamente em *hardware*. Adicionalmente, dispositivos reconfiguráveis permitem a adaptabilidade das soluções de *hardware* em função das condições experimentais, de esforço computacional ou de consumo de energia. Uma característica importante dos dispositivos SoC (*System on Chip*) FPGA é que aproveitam a flexibilidade do *software* e a robustez computacional do *hardware* para obter soluções eficientes. Desta forma, é possível obter uma implementação eficiente do algoritmo de detecção de movimento, otimizando o **Produtor de Frames** em busca de baratear a escalabilidade da solução.

1.3 Objetivo geral

Otimizar e descrever em *hardware* o algoritmo de detecção de movimento em vídeo do Produtor de Frames e embarca-lo em um *SoC FPGA* modelo PYNQ-Z2 com o intuito reduzir o consumo de recursos computacionais e o tempo de processamento da aplicação.

1.3.1 Objetivos específicos

1. Otimizar o algoritmo de detecção de movimento do **Produtor de Frames**, modelando-o em uma FSM (*Finite State Machine*), buscando validar seu funcionamento primeiramente em uma aplicação Python em *software*.

2. Programar em VHDL o algoritmo previamente otimizado e testado em *software*, validando seu funcionamento por meio de simulações comportamentais no Vivado.

3. Implementar o algoritmo descrito em VHDL e embarca-lo em um *SoC FPGA* modelo PYNQ-Z2.

4. Programar uma aplicação em *software* para o ARM do *SoC FPGA* que se comunique com o algoritmo de detecção de movimento em *hardware* e opere suas entradas e saídas em busca de validar seu funcionamento.

1.4 Contribuições do Trabalho

Este trabalho propõe uma otimização na técnica de diferenciação temporal para detecção de movimento em vídeo. Uma arquitetura sistêmica foi elaborada a partir do algoritmo descrito em uma aplicação Python em *software* com o objetivo de atingir uma descrição comportamental em *hardware* derivada de um projeto RTL descrito em VHDL. Para tanto, experimentos e testes foram realizados sob os critérios de otimalidade de um projeto digital. Contribuindo com mais um modelo de detecção de movimento em vídeo para um *SoC FPGA*.

1.5 Organização do Documento

Este documento propõem as etapas de desenvolvimento de um trabalho de conclusão de curso em engenharia eletrônica e descreve em 5 capítulos: a justificativa e os objetivos do projeto na introdução. A fundamentação teórica, no segundo capítulo, trata das técnicas utilizadas em processamento digital de imagens e também sobre *hardware* reconfigurável mais especificamente sobre a SoC FPGA da série Zynq-7000. O capítulo três trata sobre a metodologia aplicada na pesquisa. E os capítulos finais discutem sobre os resultados alcançados, bem como sobre a conclusão dos trabalhos realizados.

2 Fundamentação Teórica

Neste capítulo serão apresentados conceitos sobre a representação, a aquisição e o processamento digital de imagens, bem como sobre os métodos mais usuais para identificar movimento em vídeo. Além disso, será detalhada a estrutura de um *SoC FPGA* e, ao final do capítulo estão citados os trabalhos do estado da arte relacionados ao tema de pesquisa.

2.1 Processamento Digital de Imagens

2.1.1 Imagens Digitais

Considerando imagens em preto e branco, os autores em (GONZALES; WOODS, 2008) definem uma imagem como uma função bidimensional $f(x, y)$, cuja amplitude, para qualquer par de coordenadas espaciais x e y, determina a **intensidade** ou a **escala de cinza** daquela imagem, naquele ponto. Não obstante, uma imagem é considerada digital quando, tanto o par de coordenadas (x, y), quanto suas respectivas intensidade possuem valores finitos e discretos. Para tanto, uma imagem digital é constituída por uma matriz de elementos finitos, os quais possuem um valor de intensidade e uma posição em particular dentro desta matriz. Estes elementos, chamados de *picture elements* ou de *pixels* são as partículas fundamentais de uma imagem digital.

2.1.2 Aquisição de imagem

As imagens digitais provêm de um cenário real, cujos objetos em cena, ao serem iluminado por um fonte, refletem ou absorvem diferentes comprimentos de luz. Estes fótons, ou a ausência deles, podem ser percebidos por um material fotossensível que traduz a partícula eletromagnética em um formato de onda em tensão elétrica. Para tanto, através de um computador, este sinal em tensão pode ser quantizado e, desta forma, compor um *pixel* do cenário. Ao organizar vários destes *pixels* em uma matriz, uma imagem digital é formada, assim como exemplificado no esquemático da figura 3.

Diferentes sistemas de aquisição de imagens, tais como as câmeras digitais, possuem diferentes parâmetros de **resolução espacial** e de **intensidade luminosa**. Ainda segundo (GONZALES; WOODS, 2008), a **resolução espacial** é definida como a menor medida discernível em uma imagem. O que, quantitativamente, pode ser expresso em quantidade de pontos por unidade de distância, do inglês: *dots per inch*, ou dpi. Além disso, os autores ainda define que a **resolução de intensidade** se dá pela menor variação discernível sobre os níveis de intensidade dos *pixels* que compõem a imagem.

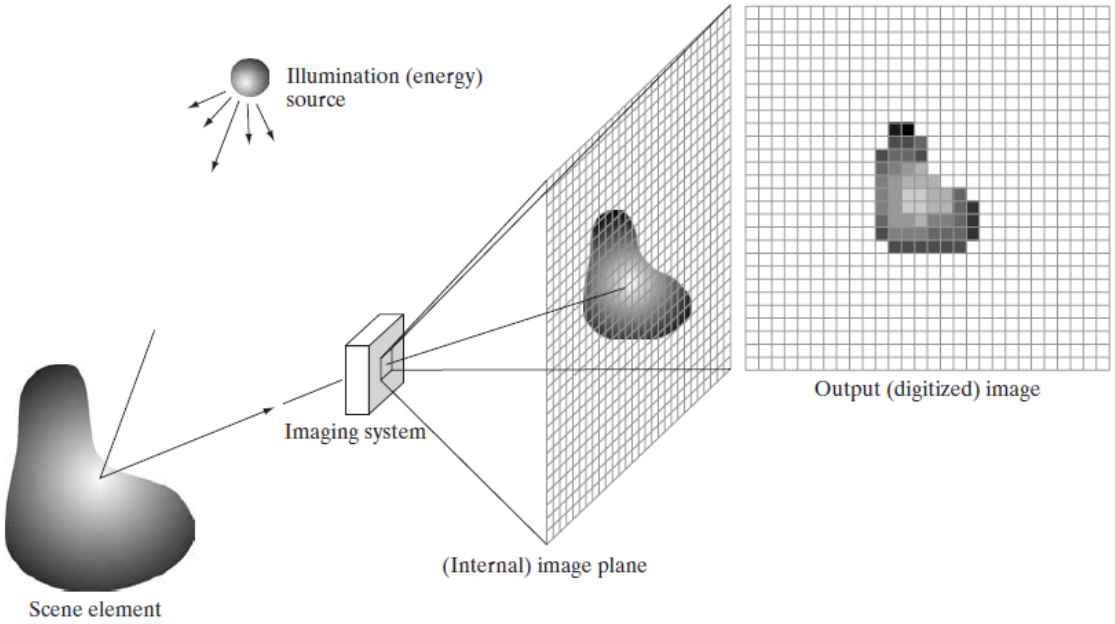


Figura 3 – Um exemplo do processo de aquisição de uma imagem digital. (Figura retirada de (GONZALES; WOODS, 2008), pág. 51, Figura 2.15)

2.1.3 Processamento Espacial

No domínio do espaço, processar uma imagem $f(x, y)$ significa manipular diretamente seus *pixels* através de um operador T, gerando outra imagem $g(x, y)$. Podendo ser expresso como:

$$g(x, y) = T(f(x, y)) \tag{2.1}$$

Existem duas categorias básicas em termos de processamento espacial de uma imagem, sendo elas: a transformação de intensidade e a filtragem espacial. O primeiro caso aplica um operador T sobre a intensidade individual 'r' de cada *pixel*, modificando-o para uma nova intensidade 's'. Ou seja, neste caso um elemento da imagem não interfere no outro, sendo a transformação de intensidade comumente utilizada na variação de contraste ou limiarização da imagem.

Observa-se do gráfico à esquerda na figura 4 que é possível gerar uma imagem com maior contraste, escurecendo os *pixels* com intensidade menor que um limiar k e também clareando os de intensidade maior que k. Todavia, do gráfico a direita da mesma figura, percebe-se que a imagem será binarizada pelo operador T. Ou seja, os *pixels* abaixo de k recebem a intensidade mínima, já os acima, recebem a intensidade máxima.

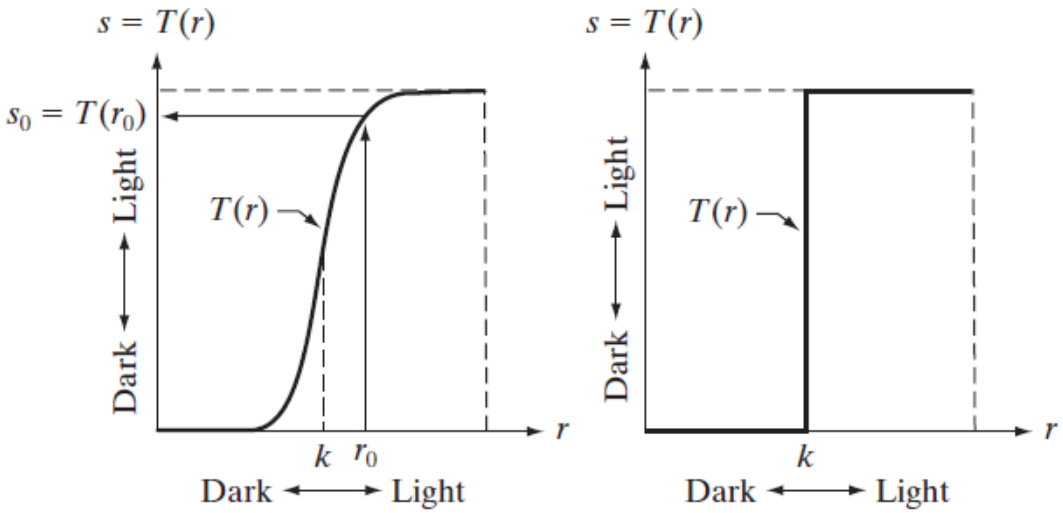


Figura 4 – Alargamento de Contraste e Limiarização. (Figura retirada de (GONZALES; WOODS, 2008), pág. 106, Figura 3.2)

A filtragem espacial, exemplificada pela figura 5, leva em consideração os valores de intensidade de uma janela de vizinhos próximos ao *pixel* que será operado por T. Esta janela, também conhecida como *kernel*, pode variar em tamanho de acordo com o filtro empregado, bem como nos pesos de contribuição dos vizinhos. Esta técnica de filtragem é comumente utilizada para suavizar ou destacar os contornos dos objetos na imagem, ou seja as áreas de alta frequência.

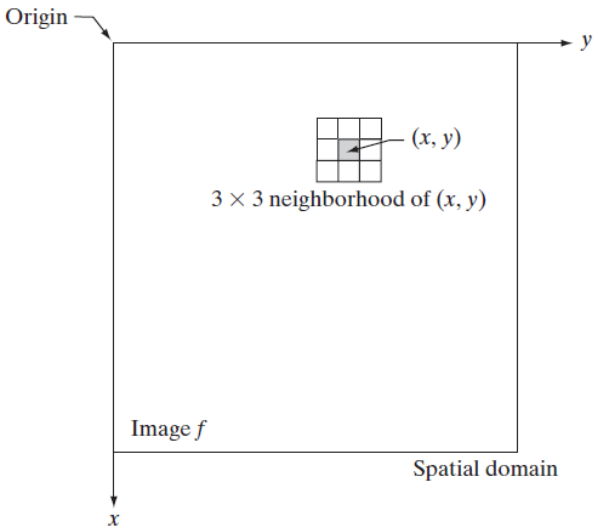


Figura 5 – Filtragem espacial e Vizinhos próximos. (Figura retirada de (GONZALES; WOODS, 2008), pág. 105, Figura 3.1)

Vale ressaltar que, ainda segundo (GONZALES; WOODS, 2008, sessão 3.1.1, pág. 105), o processamento espacial de imagens é geralmente mais eficiente e menos custoso computacionalmente quando comparado aos processamentos no domínio da frequência.

2.1.4 Fluxo de Imagens ou Vídeo

Um vídeo nada mais é do que imagens capturadas e dispostas de forma sequencial com o intuito de imprimir movimento. Seja por uma câmera com fundo estático, onde objetos se deslocam, ou até mesmo pela movimentação da própria câmera, com fundo dinâmico. De qualquer forma, o vídeo herda os parâmetros de resolução das imagens que o compõe e, além disso, em termos da cadência no fluxo de imagem, é comum definir uma velocidade de captura, ou seja, a velocidade com que o vídeo se apresenta em quadros por segundo, da sigla em inglês: FPS (*frames per second*).

2.1.5 Detecção de Movimento em vídeo

Seja para fugir de predadores ou até mesmo para identificar presas, desde que os primeiros aparatos ópticos se formaram na natureza é uma necessidade comum perceber o movimento através da variação na luz ambiental. Não obstante, com o advento das câmeras e da automação nos serviços de vigilância por vídeo, surge a necessidade de emular a detecção de movimento natural. Para tanto, existem diversas técnicas empregadas para responder às variações temporais e espaciais em vídeo causadas pelo movimento. De modo geral, as técnicas mais utilizadas baseiam-se em três princípios: subtração de fundo (*background subtraction*), diferenciação temporal (*temporal differencing*) e fluxo óptico (*optical flow*).

A subtração de fundo é a técnica mais usual empregada sobre câmeras estáticas. O método se baseia em um modelo do fundo do cenário, comparando-o com as imagens capturadas. Desta forma, a subtração de fundo consegue detectar e isolar os objetos inseridos no quadro. Contudo, essa técnica não se adapta bem em ambientes dinâmicos com fundos muito variáveis. Em (SEHAIRI; CHOUIREB; MEUNIER, 2017) são citadas diversas técnicas com o intuito de corrigir os problemas derivados de um fundo dinâmico.

A diferenciação temporal é um das técnicas matematicamente mais simples e eficientes para detectar mudanças temporais na iluminação ao comparar quadros consecutivos no vídeo a partir da subtração da intensidade de seus *pixels*. Esta técnica se adapta muito bem a ambientes dinâmicos, porém não é possível, através da diferenciação temporal, isolar os objetos que se movem. Resultando apenas em uma identificação binária: se há ou se não há movimento em um intervalo de vídeo. Esse método é empregado no módulo de detecção de movimento de (MISHRA et al., 2011). Os autores em (SEHAIRI; CHOUIREB; MEUNIER, 2017) detalham uma adaptação desta técnica que compara 3 *frames* consecutivos, conferindo robustez a ruídos e uma boa detecção para objetos que se movem lentamente.

O fluxo óptico, por sua vez, possui o algoritmo mais complexo e custoso dentre as três técnicas citadas. Este método, assim como a diferenciação temporal, utiliza da dife-

rença entre quadros consecutivos, porém esse se preocupa em traçar um vetor indicando a direção e o sentido de movimento do objeto, tal como exemplificado pela figura 6, retirada de (OPENCV, 2013), onde o autor detalha o algoritmo Lucas-Kanade para aplicar o fluxo óptico como classificador de movimento em vídeo.

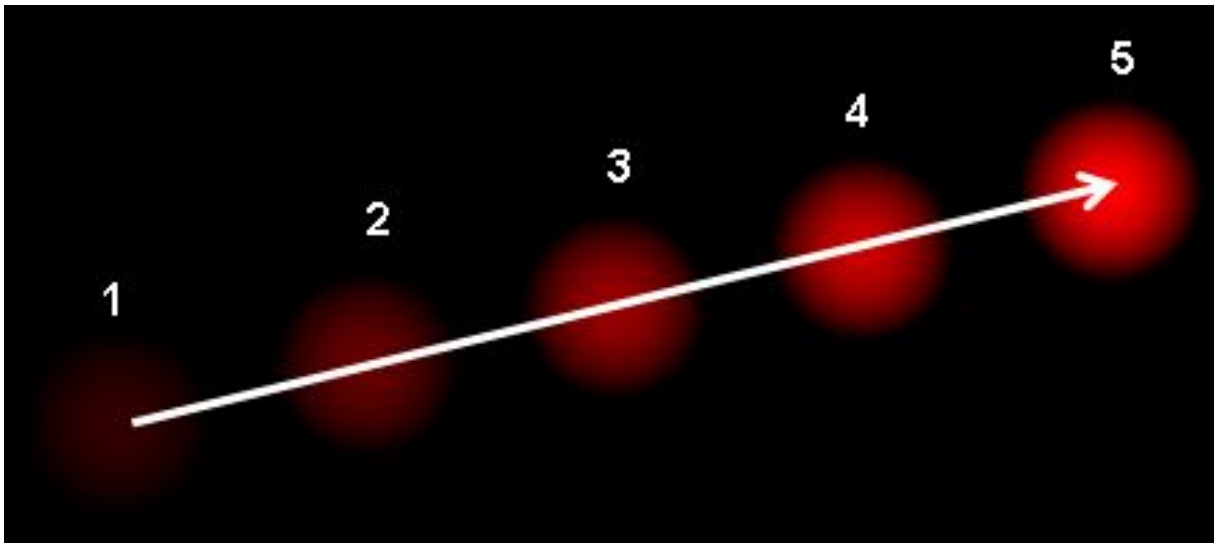


Figura 6 – Exemplo do fluxo óptico (Figura retirada de (OPENCV, 2013))

Para tanto, considerando que o engatilhador de captura do **Produtor de Frames** não se preocupa com o formato dos objetos em cena, nem mesmo com a direção do movimento, este trabalho descreve uma variação na técnica de **diferenciação temporal**, detalhada no capítulo 3, em busca de simplicidade matemática, para reduzir o tempo de processamento, e também em busca de eficiência no armazenamento das imagens capturadas, para reduzir o consumo de memória.

2.2 Hardware Reconfigurável

Os FPGAs, sigla em inglês para *Field Programmable Gate Array*, surgiram em meio aos circuitos integrados de aplicação específica (ASIC) para facilitar a descrição de hardware (HDL), trazendo o conceito de circuitos reconfiguráveis. Estes dispositivos desvincularam os blocos pre-programáveis do fabricante de ASICs, dando aos projetistas a possibilidade de criar blocos funcionais dedicados especificamente aos seus projetos.

2.2.1 Estrutura do FPGA

A estrutura básica de um FPGA consiste em dispor um grande número de portas lógicas em um arranjo matricial chamado de **Bloco Lógico Configurável**, ou da sigla em inglês CLB. Estes CLBs são rodeados por **Trilhas de Roteamento** multiplexadas

que os comunicam entre si e também permitem a a entrada ou saída de sinais através dos **Módulos I/O** (vide figura 7).

Um CLB é como uma célula tronco para FPGA. Se especializa a partir de uma tabela verdade, ou também conhecida como *Look-up table* (LUT) que relaciona os valores de entrada a um possível resultado na saída, armazenando-o em registradores Flip-Flops. Estas células são extremamente versáteis, tornando flexível o particionamento e o mapeamento da lógica digital pela FPGA, além de serem consultados de forma concorrente, o que os torna potencialmente úteis para processamento paralelo. A figura 8 a seguir exemplifica um CLB composto por uma LUT de 6 entradas e duas saída que podem ser roteadas diretamente ao próximo CLB, através da rota 'O' e 'MUX', ou multiplexadas para as unidades de memória (Flip-Flops ou Latch) com saídas em 'Q1' e 'Q2'.

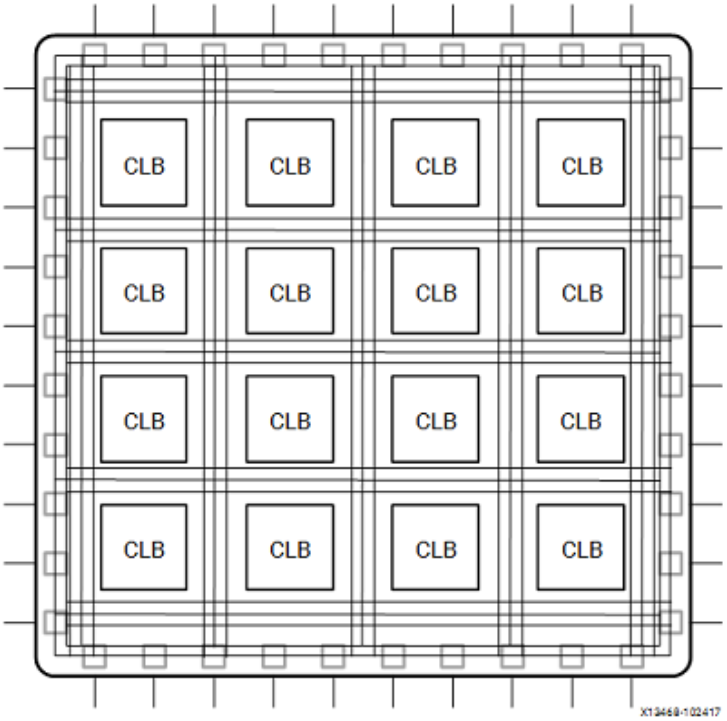


Figura 7 – Arquitetura Genérica de uma FPGA (Figura retirada de (XILINX, 2018))

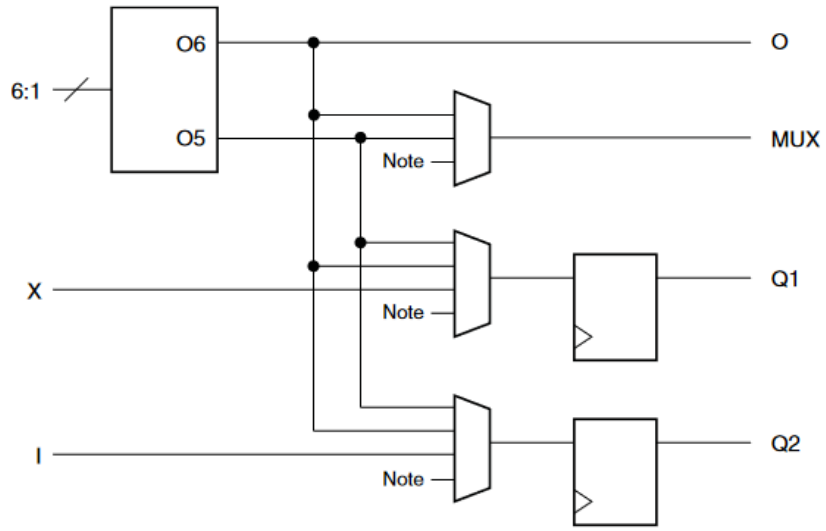


Figura 8 – Exemplo de uma LUT e elementos de Memória em um CLB (Figura retirada de (XILINX, 2017))

Os modelos mais atuais de FPGAs possuem variados ASIC integrados, tais como: blocos de RAM, FIFOs, blocos DSPs (*Digital Signal Processing*), conversores de dados ADCs, PLLs, Clock Managment, processadores ARM e até mesmo os *softprocessor*, uBlaze e PicoBlaze que são processadores construídos a partir dos próprios CLBs, BRAMs e DSPs do FPGA. A figura 9 exemplifica um modelo de FPGA atual.

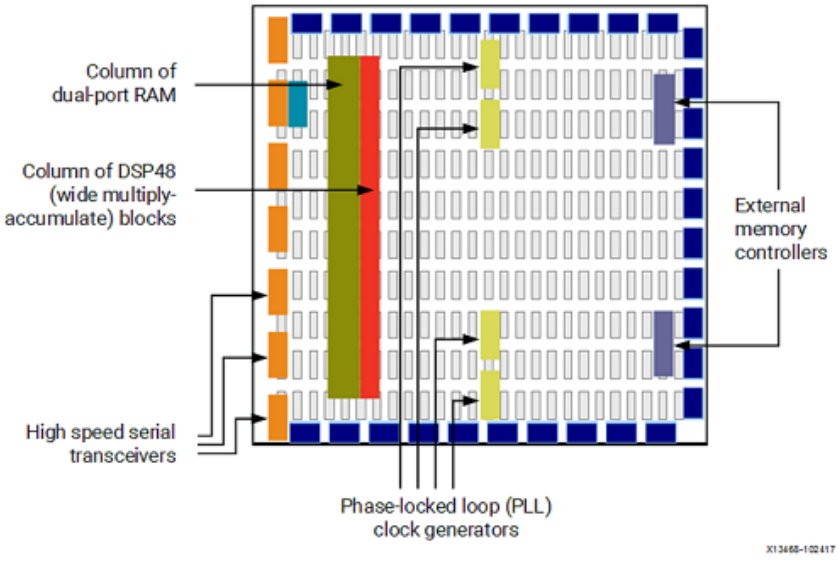


Figura 9 – Arquitetura contemporânea de uma FPGA (Figura retirada de (XILINX, 2018))

2.2.2 Estrutura de um SoC FPGA

O conceito por trás de um SoC FPGA é unir a velocidade de processamento e robustez de um FPGA com a flexibilidade de um CPU e seus periféricos de aplicação específica. Um exemplo é o SoC Zynq-7000 da Xilinx, que possui um processador ARM Cortex A9 *Dual Core* que dispõe de memória Cache com 512kb, dois barramentos de comunicação para cada protocolo *Machine to Machine*: UART, I2C e SPI, interfaces GPIO e USB, e também espaços em memória DRAM. Todas essa funcionalidades no mesmo *chip*, bem como a possibilidade de embarcar aplicações em software, facilitam a programação e os testes dos projetos descritos em *hardware*. A figura 10 exemplifica um SoC FPGA do modelo Zynq-7000.

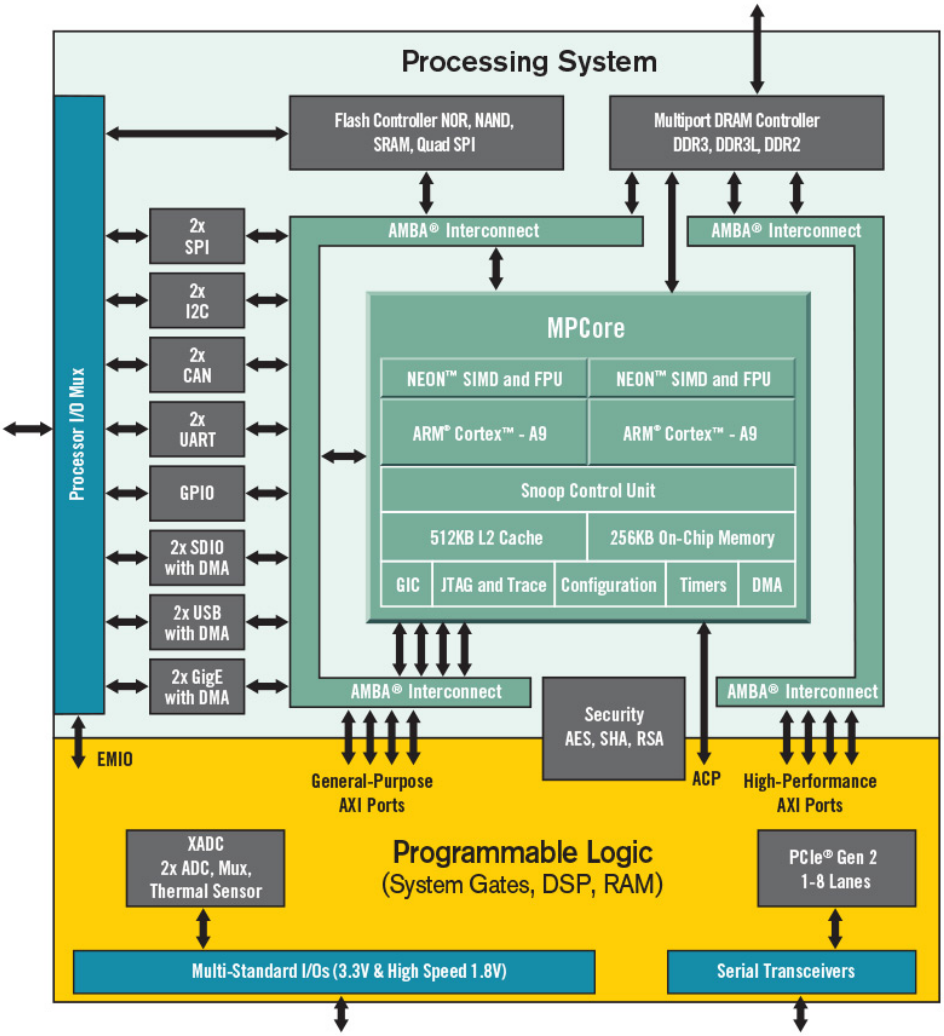


Figura 10 – Exemplo arquitetural de um SoC FPGA (Figura retirada de (XILINX, 2021b))

2.2.3 Abstrações em Projetos de Circuitos Digitais

O modelo de Gajski-Kuhn descreve um projeto de circuito integrado em três eixos principais: Comportamental, Estrutural e Físico. O primeiro eixo ou domínio descreve o projeto em termos de blocos comportamentais, mas não se preocupa em como atingir suas funcionalidades internas. O segundo domínio se preocupa com as ligações e estruturas básicas que compõem os blocos funcionais. Não obstante, o domínio físico descreve o projeto enquanto disposição geométrica de seus módulos ou componentes, voltada para fabricação do circuito.

Assim como mostrado na figura 11, os domínios são dispostos em 3 eixos que formatam um 'Y'. Ao adentrar para o centro do gráfico, o nível de abstração diminui e a complexidade do projeto aumenta.

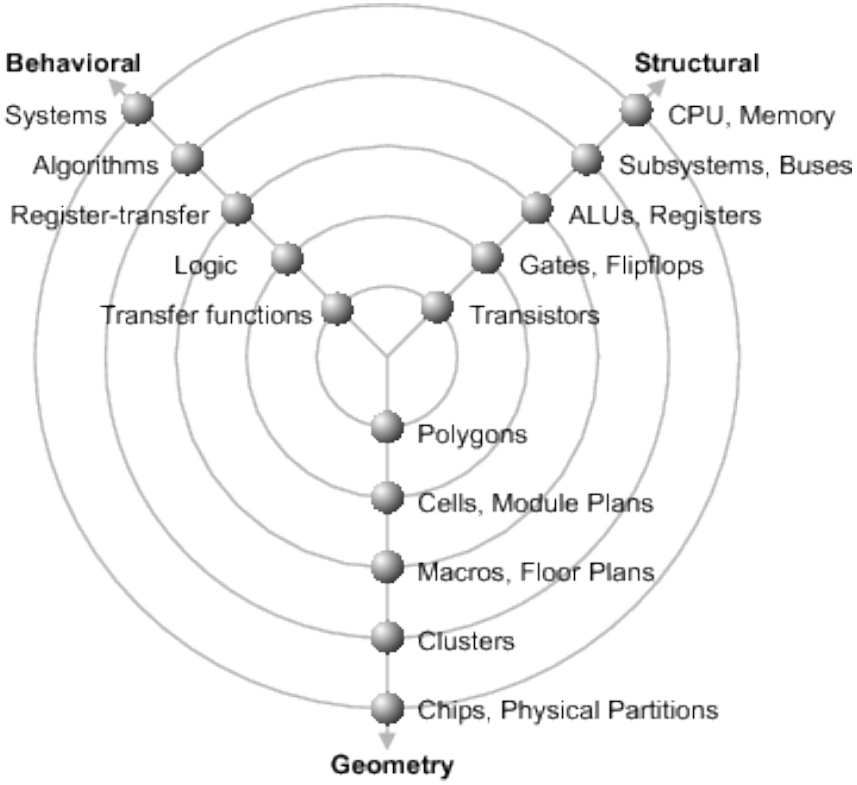


Figura 11 – Diagrama Y proposto por Gajski e Kuhn (Figura retirada de (SCHLOSSER, 2001))

Os 5 níveis de abstração estão representados pelos círculos ao redor dos domínios em 'Y'. Levando o projeto de uma abstração **Sistêmica** de altíssimo nível, geralmente lançando mão sobre projetos integrados entre *software* e *hardware* (HW/SW) para determinar se o sistema é viável.

Passando para abstração **Arquitetural** que simula o algoritmo de alto nível em alguma linguagem de descrição de hardware como HDL.

Fazendo possíveis otimizações na camada de abstração **Lógica**, onde as especificações funcionais do nível anterior são traduzidas em portas lógicas com o intuito de reduzir a quantidade de elementos biestáveis e acrescentar desempenho no projeto.

Para finalmente alcançar a camada **Física** que descreve o conjunto completo de informações necessárias à construção do circuito digital. Nesta última camada o projetista pode aplicar uma mascara de CI, traduzindo as funções lógicas em geométricas.

2.2.4 Fluxo de desenvolvimento do projeto

Em projeto digitais é comum orientar o desenvolvimento do trabalho a partir de um referencial *top-down*: partindo da abstração sistêmica e comportamental do projeto até alcançar as otimizações na camada física do circuito. Tratando-se de um projeto para FPGA a figura 12 apresenta uma visão geral sobre as etapas sugeridas pela Xilinx para construção de um projeto digital em hardware.

Após desenhar o projeto de forma sistêmica, o projetista escreve e testa o algoritmo em software para validar o projeto a nível comportamental. Em seguida ocorre a síntese em alto nível que descreve o comportamento do algoritmo em termos de fluxo de sinais ou transferência de dados entre registradores e blocos de operação lógica, chamado de RTL (*Register Transfer Level*) que é escrito em alguma linguagem de descrição de *hardware* (HDL). Neste ponto o projeto RTL é testado através de simulações comportamentais para possíveis ajuste e otimizações. Então, os módulos validados no projeto RTL passam pelo processo de síntese lógica que os traduzem para o nível de portas lógicas. Com as portas lógicas definidas ocorre o processo chamado de *Place and Route* que organiza de forma estrutural o posicionamento e as ligações entre os módulos lógicos e os registradores. Neste ponto, o projetista pode reposicionar os componentes para facilitar a comunicação entre os módulos mais críticos e também otimizar o tamanho do circuito lançando mão sobre ferramentas de análise de temporização do circuito antes de exporta-lo. Finalmente, na última etapa o *bitstream* do circuito é gerado e os testes físicos podem ser realizados. No caso de desenvolvimento *hardware-software*, um novo fluxo de projeto relacionado com o desenvolvimento da aplicação de *software* é iniciado, por exemplo, usando o SDK.

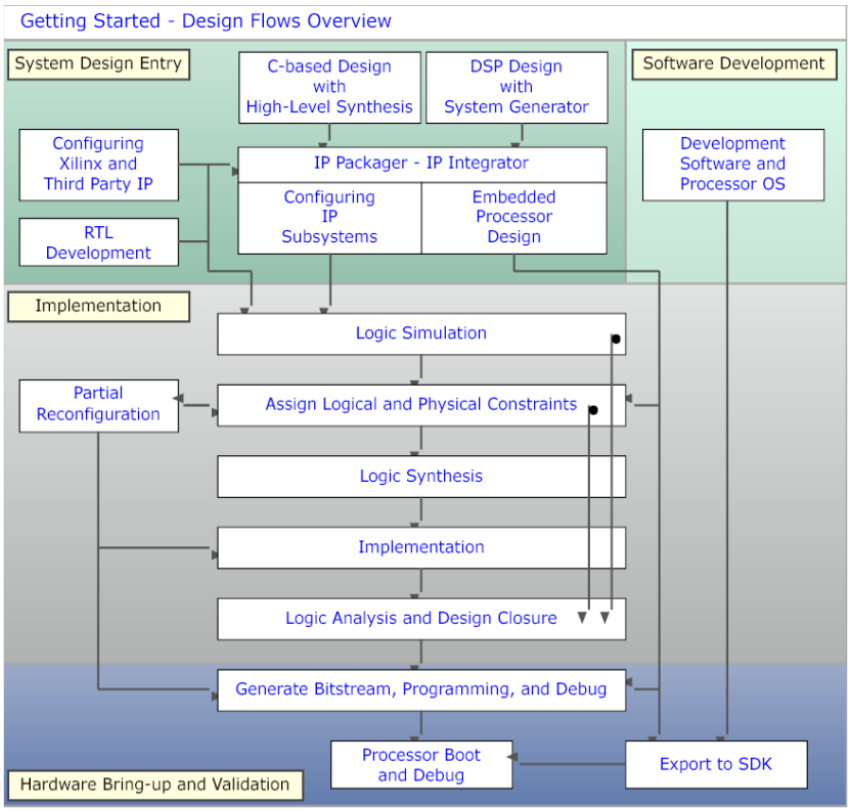


Figura 12 – Visão geral sobre o fluxo de desenvolvimento de um projeto digital (Figura retirada de (XILINX, 2016))

2.3 Revisão do Estado da Arte

Nesta seção será disposto um panorama geral sobre os trabalhos relacionados a detecção de movimento em vídeo com base em FPGA. Independentemente da aplicação fim, os trabalhos encontrados acerca deste tema seguem etapas semelhantes para alcançar os resultados, sendo as principais: Aquisição e tratamento das imagens, aplicação do algoritmo de detecção, aplicação dos resultados e análise de temporização e consumo energético do projeto. Vale ressaltar que dentre os trabalhos aqui citados é comum a utilização de desenvolvimento entre a validação dos algoritmos em *software* e a sua aceleração em *hardware* (*co-design HW/SW*).

Os autores em (COTHEREAU; DELAITE; GOURDIN, 2008) iniciam os trabalhos com um estudo bem detalhado sobre os paradigmas de desenvolvimento de *hardware* utilizados na pesquisa, bem como comparando modelos de FPGA entre as fabricantes Altera e Xilinx. No entanto, o foco principal da pesquisa é verificar a possibilidade de criar um sistema de vigilância de vídeo sobre uma FPGA. Neste contexto, eles implementaram a aquisição de imagens de uma câmera IP através de entidades escritas em Verilog para uma FPGA modelo DE2 da Altera e além disso as imagens capturadas passam por um algoritmo de subtração de fundo escrito em C e embarcado no processador *softcore* Nios II disponível na placa de desenvolvimento escolhida. Contudo, eles relatam muitos

problemas de concorrência entre as entidades descritas em hardware e o algoritmo em software ao acessar as imagens em memória.

Em (JAGDALE; VAIDYA, 2012) os autores utilizam de uma câmera do modelo "TRDB-D5M"com resolução de 2592x1944p e que representa as imagens no modelo Bayer. O sensor é conectado diretamente nos GPIOs do Soc FPGA modelo DE-II 70 da Altera e configurado via protocolo I2C. Devido a representação padrão da câmera o algoritmo proposto nesta pesquisa primeiro converte de Bayer para RGB e depois para escala de cinza. Assim como em (COTHEREAU; DELAITE; GOURDIN, 2008) a detecção de movimento se dá pela técnica de subtração de fundo escrita em C e embarcada no *softcore* Nios II. Contudo neste projeto uma FIFO foi colocada entre o bloco de captura e o bloco de detecção de movimento para enfileirar as imagens capturadas e evitar a concorrência no acesso à memória observado na pesquisa anterior.

Os trabalhos realizados em (SINGH; SHEKHAR; VOHRA, 2016) alcançaram um protótipo para um sistema de vigilância automático baseado na detecção de movimento em vídeo e implementado na Virtex-5 da Xilinx modelo FX130T. Os autores utilizam uma variação da técnica de diferenciação temporal sobre 4 quadros consecutivos do vídeo. Este quadros são separados em blocos de *pixels* para o cálculo da média da região delimitada pelo bloco, o que torna o algoritmo menos sensível a cenários pseudo estacionários, ou seja aqueles cenários com muita variação luminosa causada por projeção de sombras.

O modelo para detecção de movimento em vídeo proposta em (JUNIOR et al., 2017) é baseada em uma técnica de fluxo ótico que capta o cenário a partir de dois sensores A e B. As imagens de apenas 320x240p formadas pelos dois sensores passam por um módulo chamado de EMD (*Elementary Motion Detector*), vide (JUNIOR et al., 2017, fig. 2), que filtra as entradas através de um passa baixas e calcula a diferença entre a correlação de A e de B. Desta maneira o EMD consegue determinar se um objeto se move para a direita ou para a esquerda. Aqui os autores simulam o algoritmo através dos *software* MatLab e Simulink, mas convertem os blocos simulados para VHDL e embarcam o algoritmo em uma Spartan-6 da Xilinx. Como resultados os autores conseguem identificar carros em movimento em uma rodovia, inclusive conseguindo separar as sombras projetadas sobre a pista dos veículos que as geram. Vale ressaltar que neste trabalho os autores não queriam apenas detectar o movimento em sí, mas determinar e rastrear no vídeo os objetos que se movem, vide (JUNIOR et al., 2017, fig. 5)

De forma semelhante à pesquisa descrita no último parágrafo, os autores em (SLEDEVIC; SERACKIS; PLONIS, 2018) propõem uma estrutura de hardware para rastreamento de objetos em vídeo. Todavia, no trabalho em questão as imagens são capturadas a 60 fps e com resolução de 640x480p. Além disso, o algoritmo proposto em software é capaz de interagir com o usuário que escolhe um objeto em cena para ser demarcado e localizado ao longo do vídeo. Para isso os autores utilizam das técnicas HOG (*Histogram*

of Oriented Gradient) e LBP (*Local Binary Pattern*) para identificar a forma e a textura do objeto rastreado. Não obstante, ao embarcar o algoritmo em um FPGA Virtex 4 da Xilinx em VHDL é a detecção de movimento que seleciona o objeto a ser rastreado em cena. A detecção é semelhante ao proposto em (SINGH; SHEKHAR; VOHRA, 2016) em que as imagens são separadas em blocos de *pixels* para o cálculo da média daquela região e a diferenciação temporal é aplicada entre as regiões de imagens consecutivas. Os teste mostraram a robustez do algoritmo proposto utilizando um *clock* de apenas 25Mhz no FPGA.

3 Metodologia

Neste capítulo, serão tratados os métodos e os modelos utilizados para a implementação e teste do projeto. Tendo em vista a descrição sistêmica do Produtor de Frames em 1.2, a seção 3.1 detalha a detecção de movimento em vídeo utilizada por essa entidade. Em seguida, a seção 3.2 descreve uma proposta de implementação eficiente para o algoritmo de detecção de movimento em vídeo do Produtor de Frames, através de um diagrama de estados, ou FSM (*Finite State Machine*), e seu diagrama RTL equivalente, propondo otimizações para reduzir o consumo de recursos de *hardware*, bem como, acelerar o algoritmo. A seção 3.3 descreve uma aplicação Python destinada a validar o funcionamento da FSM em *software*. Na sequência, a seção 3.4, detalha a descrição de *hardware* do algoritmo proposto, em VHDL, através da plataforma Vivado. A seção 3.5, por sua vez, apresenta a metodologia empregada na simulação comportamental da FSM em VHDL. Finalmente, a seção 3.6 descreve os procedimentos para implementação física da FSM através da plataforma de desenvolvimento PYNQ.

3.1 A Detecção de Movimento no Produtor

Dentro do **Produtor de Frames** existe uma *thread* que executa o método chamado *CaptureNDetect*. Este método aplica a diferenciação temporal entre dois *frames* através de 4 funções principais (vide figura 13): a primeira captura um par de imagens, a segunda fatia as imagens em regiões de *pixels*, a terceira calcula a diferença absoluta entre estas regiões, e a última verifica um limiar para determinar em quantas delas houve movimento. Se houver em pelo menos 25% das regiões, a *flag* de movimento é levantada. Caso contrário a *flag* de não movimento que é levantada.

3.1.1 Captura e Filtragem do Par de Imagens

Para capturar um par de imagens, a aplicação executa um objeto capturador de vídeo da OpenCV (OPENCV, 2021a) que através de um *socket* RTSP (*Real Time Streaming Protocol*) se comunica com apenas uma câmera em rede TCP/IP. Após a captura, as imagens em memória sofrem uma transformação de intensidade para escala de cinza o que reduz o vetor de intensidade RGB dos *pixels* para um único valor de intensidade representado em 8bits, ou seja, alcançando valores inteiros de 0 até 255. Além disso, os contornos nas imagens são suavizados sob a aplicação de um filtro espacial de borramento com um *kernel* quadrado de lado 5 e peso 1/25 (vide figura 14). A convolução entre este *kernel* e a matriz que compõe os quadros reduz o contraste entre os *pixels* da imagem,

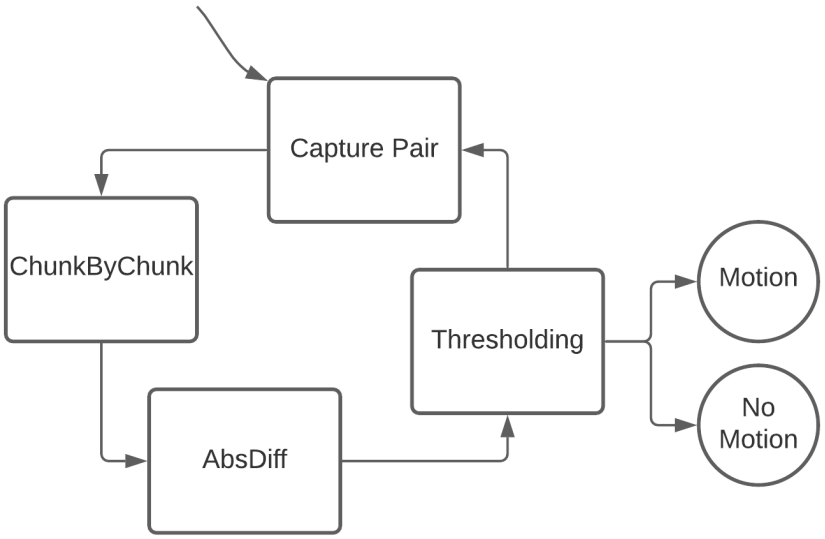


Figura 13 – Diagrama de operação da *thread* CaptureNDetect.

tornando-a mais homogênea em termos de intensidade, o que diminui os falsos positivos provenientes da diferenciação temporal.

$$\mathbf{K} = \frac{1}{\texttt{ksize.width*ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & & & & & \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

Figura 14 – Kernel K da função cv.blur() (Figura retirada de (OPENCV, 2021b))

Vale mencionar que em (OPENCV, 2021c) são detalhadas algumas funções para suavização de imagens através do OpenCV.

3.1.2 Fatiamento em regiões de *pixels*

Na detecção de movimento através da diferenciação temporal é calculado o módulo da subtração entre o *pixel* na posição (x, y) da imagem anterior I_{t-1} e o *pixel* na mesma posição da imagem atual I_t, de acordo com a equação 3.1 a seguir:

$$\Delta P = |I_{t-1}(x, y) - I_t(x, y)| \tag{3.1}$$

Desta maneira, para afirmar se houve movimento é levado em consideração a diferença absoluta de cada um dos elementos entre as imagens. Contudo, em um cenário

outdoor sob a projeção de sombras que podem ser geradas por árvores ao vento ou por nuvens que bloqueiam o sol de tempos em tempos, foi verificado que a análise isolada de cada um dos *pixels* resulta em variações muito discrepantes entre os quadro comparados. Isto torna a detecção de movimento muito sensível à sombras e engatilha falsos positivos.

Neste contexto, o fatiamento das imagens foi uma estratégia adotada para lidar com as áreas de demasiada oscilação na intensidade dos *pixels* afetados por sombras. Após definido um número de linhas e de colunas que dividem as imagens em regiões, ou *chunks*, calcula-se a média de intensidade dos *pixels* destas regiões. Para tal procedimento, primeiramente são calculados dois coeficientes que relacionam, respectivamente, a quantidade de linhas e colunas do fatiamento com o comprimento e a largura das imagens, de acordo com as equações a seguir.

$$L = \frac{comprimento}{Linhas} \tag{3.2}$$

$$C = \frac{largura}{Colunas} \tag{3.3}$$

De acordo com o diagrama da figura 15, após calcular os coeficientes através das equações 3.2 e 3.3, o programa executa dois laços de repetição aninhados para percorrer os vetores bidimensionais $I_t[comprimento, largura]$ e $I_{t-1}[comprimento, largura]$ que armazenam as imagens em memória. Para cada região é calculada a média das intensidades dos *pixels* através do método 'np.mean()' (COMMUNITY, 2021) da biblioteca Numpy, alocando os resultados em outras duas matrizes: $A[Linhas, Colunas]$ para imagem atual e $B[Linhas, Colunas]$ para imagem anterior.

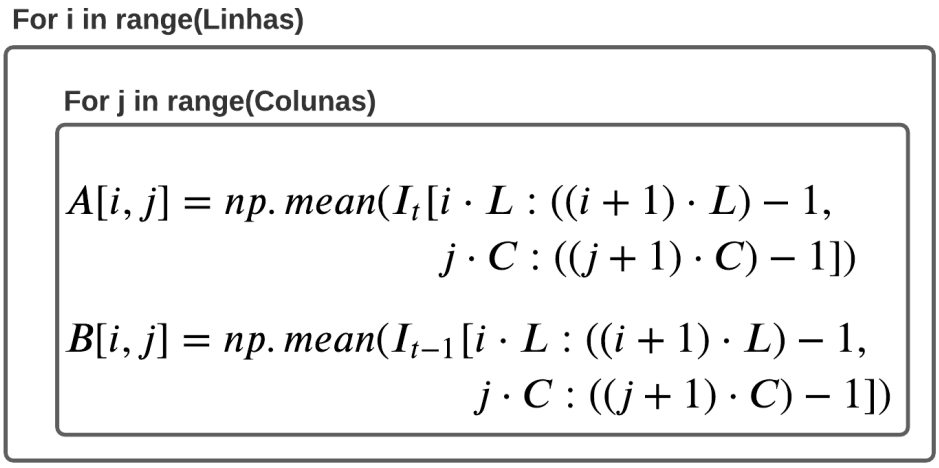


Figura 15 – Laço do cálculo das médias das regiões de *pixels*

Originalmente, o programa captura imagens com 1280 *pixels* de comprimento e 720 *pixels* de largura divididas em 4 linhas e 4 colunas. Para tanto, os coeficientes provenientes das equações 3.2 e 3.3 são dados por $L = 320$ e $C = 180$. Nesta configuração, a primeira iteração do laço aninhado aloca na primeira posição do vetor $A[0,0]$ a média da região $I_t[0:319, 0:179]$ da imagem atual. Em seguida, ocorre o mesmo procedimento para a mesma região na imagem anterior $I_{t-1}[0:319, 0:179]$, porém armazenando o resultado na primeira posição do vetor $B[0,0]$. A figura 16 imprime as duas matrizes A e B preenchidas com as médias das 16 regiões de *pixels* do par capturado.

$A[4,4]$

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]
A[3,0]	A[3,1]	A[3,2]	A[3,3]

$B[4,4]$

B[0,0]	B[0,1]	B[0,2]	B[0,3]
B[1,0]	B[1,1]	B[1,2]	B[1,3]
B[2,0]	B[2,1]	B[2,2]	B[2,3]
B[3,0]	B[3,1]	B[3,2]	B[3,3]

Figura 16 – *chunks* de médias

3.1.3 Diferença Absoluta e Limiarização

Após determinar as médias da região o programa aplica a diferença absoluta entre estes valores através da função $abs(B[i,j] - A[i,j])$ (FOUNDATION, 2021), determinando o módulo da variação média de intensidades daquela região entre os quadros capturados. Em seguida este módulo é comparado a um limiar pre-determinado pelo usuário que

indica a sensibilidade da detecção de movimento (vide figura 17). Ou seja, se o módulo calculado for maior ou igual ao limiar, determina-se que houve movimento naquela região incrementando em uma unidade uma variável que conta a quantidade de *chunks* com movimento.

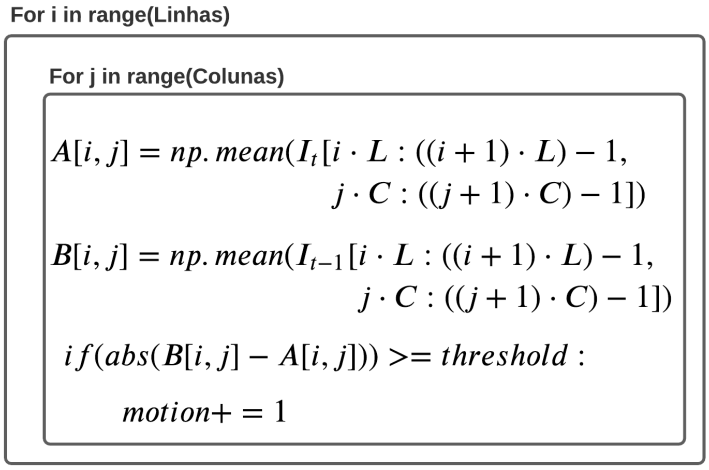


Figura 17 – Laço de comparação do par de *frames*

Portanto, ao finalizar o laço aninhado da figura 17 o contador *motion* pode armazenar um valor entre 0 a 16, indicando no primeiro caso que não houve movimento em nenhuma região das imagens comparadas ou, para o segundo caso, que houve movimento em todas as regiões comparadas.

3.1.4 Determinação de movimento

Por padrão o **Produtor de Frames** espera que ao menos um quarto dos *chunks* indiquem movimento para levantar a *flag* de movimento. Então, após a limiarização dos módulos calculados, se o contador *motion* for maior ou igual a 4, um trecho do *streaming* da câmera, com intervalo de tempo determinado pelo usuário na execução do programa, é armazenado em um arquivo de vídeo e enviado ao servidores. Caso contrário, o algoritmo volta a capturar um par de imagens, reiniciando a operação da *CaptureNDetect* descrita no diagrama da figura 13.

3.2 Proposta de Implementação Eficiente

Este trabalho propõe, em primeiro momento, otimizar e testar em *software* o algoritmo da *CaptureNDetect* do **Produtor de Frames**. Contudo, como o objetivo principal da pesquisa é desenvolver um módulo de detecção de movimento para um *SoC FPGA*, uma FSM foi descrita em alto nível visando a implementação em *hardware*, bem como minimizar a quantidade de operações e o consumo de memória provenientes da técnica de

diferenciação temporal descrito na seção anterior. Para tanto, nesta seção serão apresentadas as etapas propostas para uma implementação eficiente da detecção de movimento em vídeo através da diferenciação temporal destinada a um *SoC FPGA*, mas que será validada em primeiro momento através de um programa base em *software* e, em seguida, o algoritmo validado, será descrito em HDL destinado a um *SoC FPGA* modelo PYNQ. Os resultados das duas etapas de desenvolvimento estão apresentados no capítulo 4.

3.2.1 Pre-Definições

Para construção do algoritmo foi definido que as imagens teriam dimensões de 640x480p. Além disso, os quadros capturados serão fatiados em 40 colunas e 30 linhas totalizando 1200 *chunks* com dimensão de 16x16 *pixels*, assim como no diagrama resumo da figura 18.

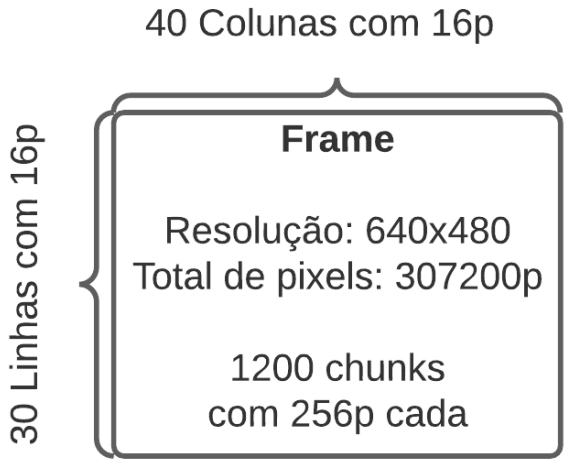


Figura 18 – Parâmetros do Fatiamento dos *Frames*

3.2.2 Captura e Preparação

Com o intuito de priorizar a detecção de movimento em si, a etapa de captura e tratamento das imagens foi abstraída da máquina de estado proposta. Todavia, o programa base em *software* executa uma *thread* que captura e trata o par de imagens de forma semelhante à descrita na seção 3.1.1, mudando apenas a fonte de captura de uma câmera IP para a *webcam* embutida no notebook que executa o programa. Na PYNQ, a captura e o tratamento das imagens é de responsabilidade do processador ARM que, embarcado com um sistema operacional Linux, também executa a *thread* descrita na seção 3.1.1, mas envia os *pixels* das imagens capturadas para o FPGA através de um barramento AXI, descrito em 3.6.1.

Com a captura dos quadros resolvida em *software* e considerando que a cada ciclo completo da máquina de estados um novo par de imagens é capturado e transformado

para escala de cinza, o algoritmo, em seu primeiro estado, aloca espaço em memória e atribui zero às variáveis de controle e aos registradores da FSM, assim como apresentado na figura 19.

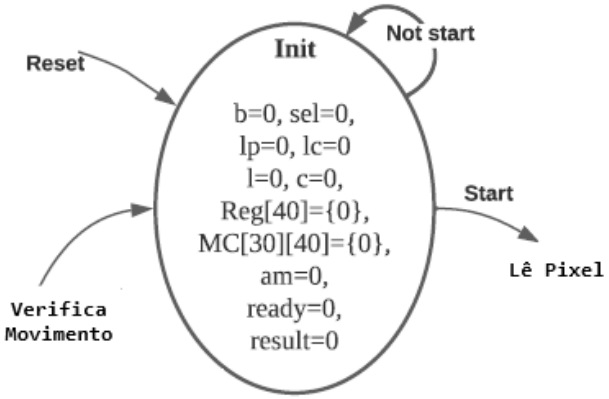


Figura 19 – Estados: Init

Ainda da figura 19, observa-se que a FSM possui dois sinais para controlar os procedimentos de inicialização (*start*) e reinicialização (*reset*) do algoritmo. O sinal *reset* é uma interrupção que, independente do estado atual, envia a FSM para o estado Init. Já o sinal *start* funciona apenas no estado Init, enviando a FSM para o estado Lê pixel.

3.2.3 Acumulo das intensidades por quadrante

Existem 3 variáveis fundamentais para orquestrar o funcionamento desta etapa da FSM, sendo elas:

- Contador de *bytes* ($0 \leq b < 16$)
- Seletora de registradores ($0 \leq Sel < 40$)
- Contador de linhas de *pixels* ($0 \leq l_p < 16$)

O contador de *bytes* b é iterado em uma unidade a cada valor de *pixel*, V_p, lido. A seletora de registradores Sel é iterada em uma unidade a cada 16 *bytes* lidos, ou seja a cada linha de um *chunk* lida. Já o contador de linhas de *pixels* l_p é iterado em uma unidade a cada 640 *bytes*, ou seja, a cada linha de *pixels* completa da imagem. Além disso, a máquina de estados utiliza um vetor com 40 posições para armazenar o somatório do valor dos *pixels* em uma determinada região, definida pela variável Sel. Neste contexto, a FSM lança mão sobre 4 estados que trabalham em um laço de leitura e acumulo dos *pixels*, como observado na figura 20.

O estado **Lê Pixel**, levanta a *flag* $nextPixel$ indicando à *thread* de captura que a FSM está pronta para receber o valor do próximo *pixel*. Em seguida, no estado **Acumulador**, o valor do *pixel* lido (V_p) é acumulado na posição Sel no vetor $Reg[Sel]$. A

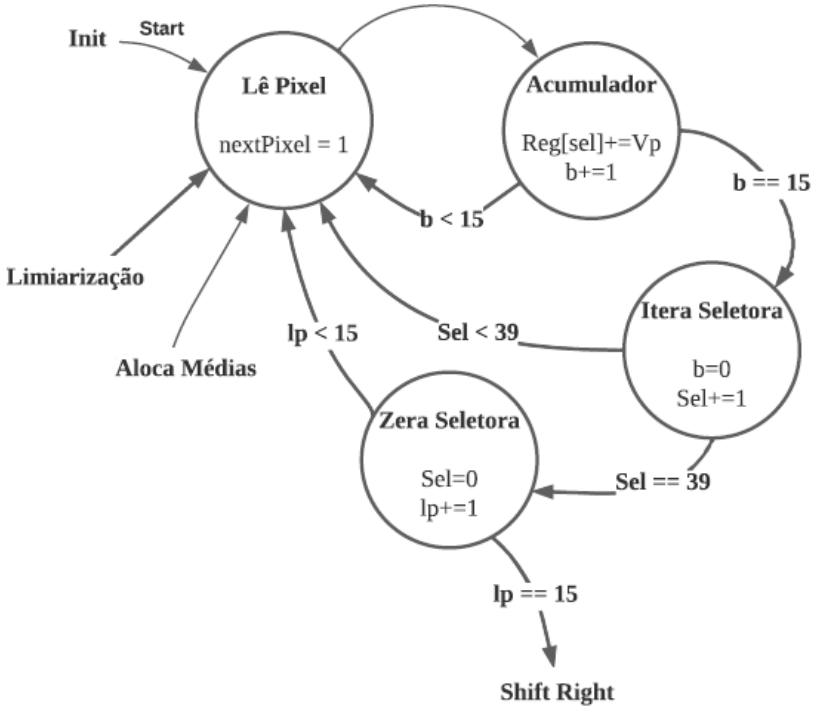


Figura 20 – Estados: Lê Pixel, Acumulador, Itera Seletora e Zera Seletora

cada V_p acumulado, o contador de *bytes* b é incrementando em uma unidade. Até 16 *pixels* serem somados ao registrador $Reg[Sel]$, referente ao *chunk* atual, a FSM oscila entre os estados **Lê Pixel** e **Acumulador**. Contudo, ao completar a linha de 16 *pixels* do *chunk* atual, a seletora Sel é iterada em uma unidade no estado **Itera Seletora** que, além disso, também zera o contador de *bytes* b. A seguir a FSM volta a oscilar entre **Lê Pixel** e **Acumulador**, mas agora apontando para o registrador do próximo *chunk*.

Após acumular 640 *pixels* a seletora Sel soma um total de 39 unidades. Para tanto, no estado **Zera Seletora**, atribui-se valor nulo para variável Sel que retorna a posição do registrador $Reg[Sel]$ ao primeiro *chunk* para começar o somatório da próxima linha. Além disso, ainda neste estado, o contador de linhas de *pixels* l_p é iterado em uma unidade. Ou seja, ao finalizar uma linha completa da imagem o algoritmo acumulou 16 *pixels* de comprimento para cada um dos 40 *chunks* no vetor $Reg[Sel]$. Desta forma, a FSM executa este laço de leitura e acumulo até alcançar 16 linhas de *pixels* completas (l_p) e mudar para o estado **Shift Right**, descrito adiante.

Em termos de um projeto RTL, esta etapa de leitura e acumulo de *pixels* pode ser representado pelo diagrama da figura 21. Em que um multiplexador controlado pela variável *sel* distribui os valores dos *pixels* que serão acumulados dentre as 40 posições do vetor de registradores $Reg[40]$.

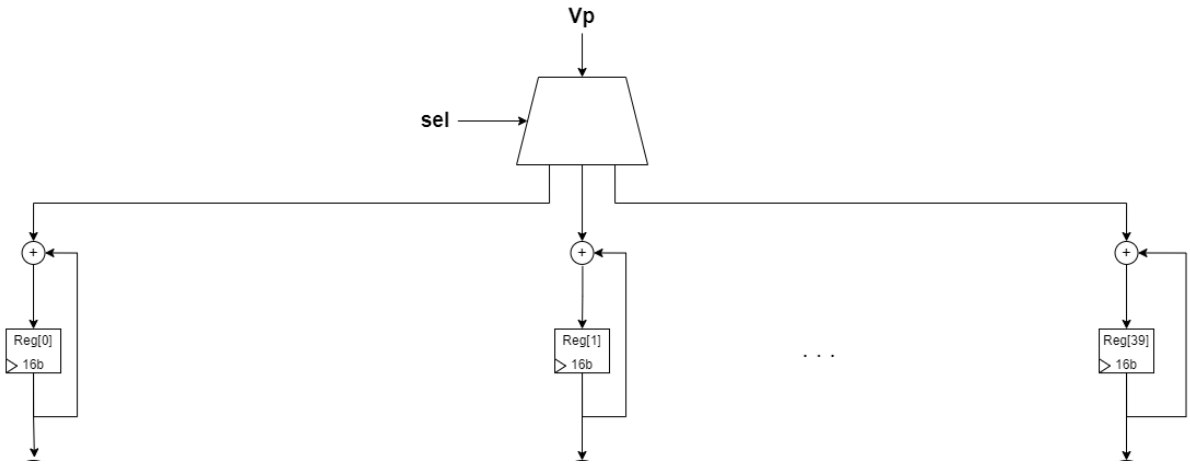


Figura 21 – Diagrama RTL: Leitura e Acumulo de *pixels*

3.2.4 Valor médio do quadrante

Ao acumular os valores de intensidade de 16 linhas completas, cada uma das 40 posições do vetor $Reg[40]$ guarda um montante do somatório de 256 *pixels*. Porém, como o interesse é sobre o valor médio das intensidades em cada região, o algoritmo executa, no estado **Shift Right** da figura 22, 8 deslocamentos a direita sobre estes montantes, o que equivale a dividir o valor acumulado por 256. A quantidade de *pixels* por quadrantes foi cuidadosamente escolhida visando simplificar as operações aritméticas envolvidas, por exemplo, substituindo as divisões por deslocamentos a direita o que acelera o cálculo das médias dos quadrantes. Além disso, em *hardware*, está operação é feita sobre os 40 registradores de forma paralela, em contraponto ao programa base em *software* que calcula a média das regiões de forma sequencial.

Ainda no estado **Shift Right**, o contador de linhas de *pixels* l_p é zerado e uma nova variável passa a ser considerada, o contador de linhas de *chunks* l_c. Se esta última for menor que 30, significa que a FSM ainda está processando o primeiro quadro e, para tanto, os valores médios dos *chunks* são armazenados em uma matriz chamada de $MC[30][40]$ que possui 30 linhas e 40 colunas para suportar os 1200 *chunks* da primeira imagem.

No estado **Aloca Médias**, a variável l_c determina em qual linha da matriz $MC[30][40]$ o valor médio dos 40 *chunks*, em $Reg[40]$, serão armazenado. E, assim como no estado anterior, quando em *hardware*, este processamento é feito de forma paralela. Para tanto, a cada linha de *chunks* armazenadas em $MC[30][40]$, a FSM incrementa l_c em uma unidade. Se l_c for igual a 30, conclui-se que a média de todas as regiões da primeira imagem já estão armazenada em $MC[30][40]$ e que a segunda imagem começou a ser processada. Desta maneira, o algoritmo segue para o estado **Médias Diff** explicado na seção 3.2.5.

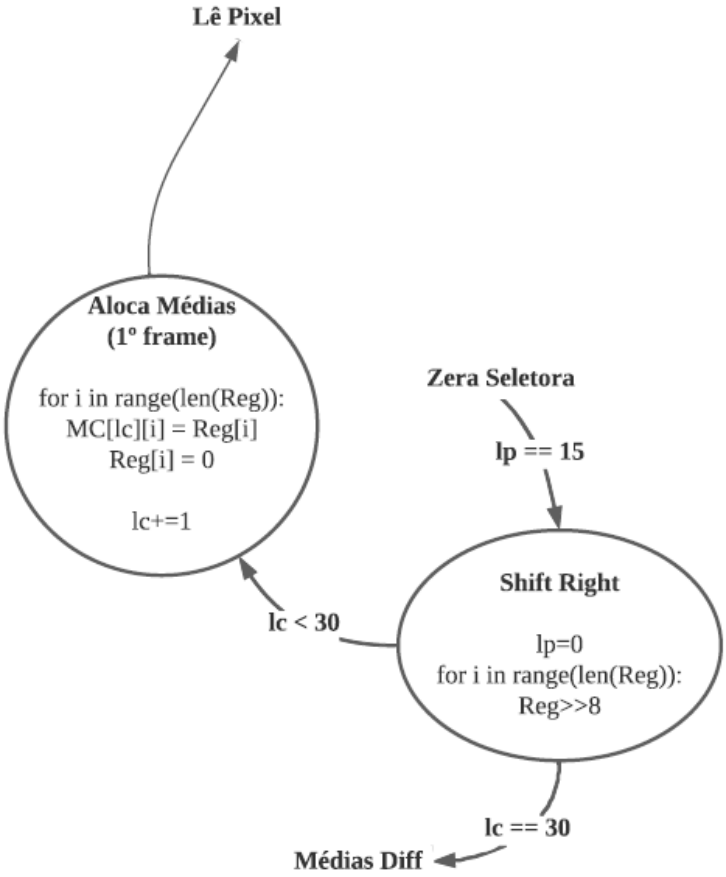


Figura 22 – Estados: Shift Right e Aloca Médias

Em termos de um projeto RTL, esta etapa de cálculo e armazenamento das médias dos quadrantes da primeira imagem pode ser representada pelo diagrama da figura 23. Em que o multiplexador controlado pela variável $sel2$, equivalente a l_C, determina em qual linha da matriz $MC[30][40]$ as médias das intensidades por quadrante serão armazenadas. Além disso, na etapa em questão, o multiplexador controlado pela variável $sel3$ e o demultiplexador controlado pela variável $sel4$ estão configurados de tal forma a deixar o sinal da média recém calculada ser armazenada diretamente em $MC[30][40]$.

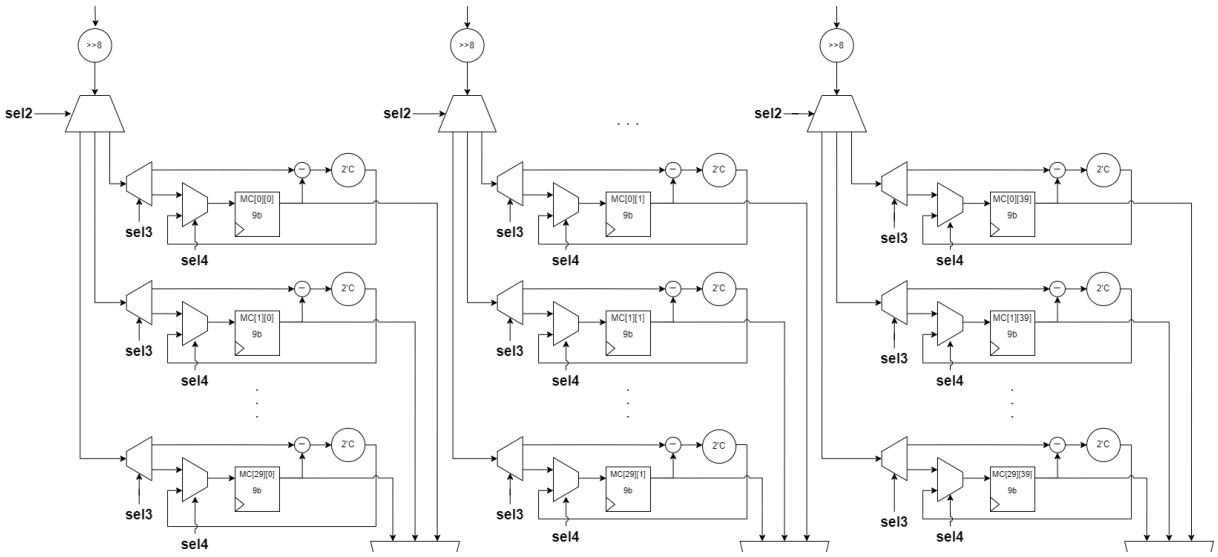


Figura 23 – Diagrama RTL: cálculo e armazenamento das médias dos quadrantes

3.2.5 Comparação e Limiarização

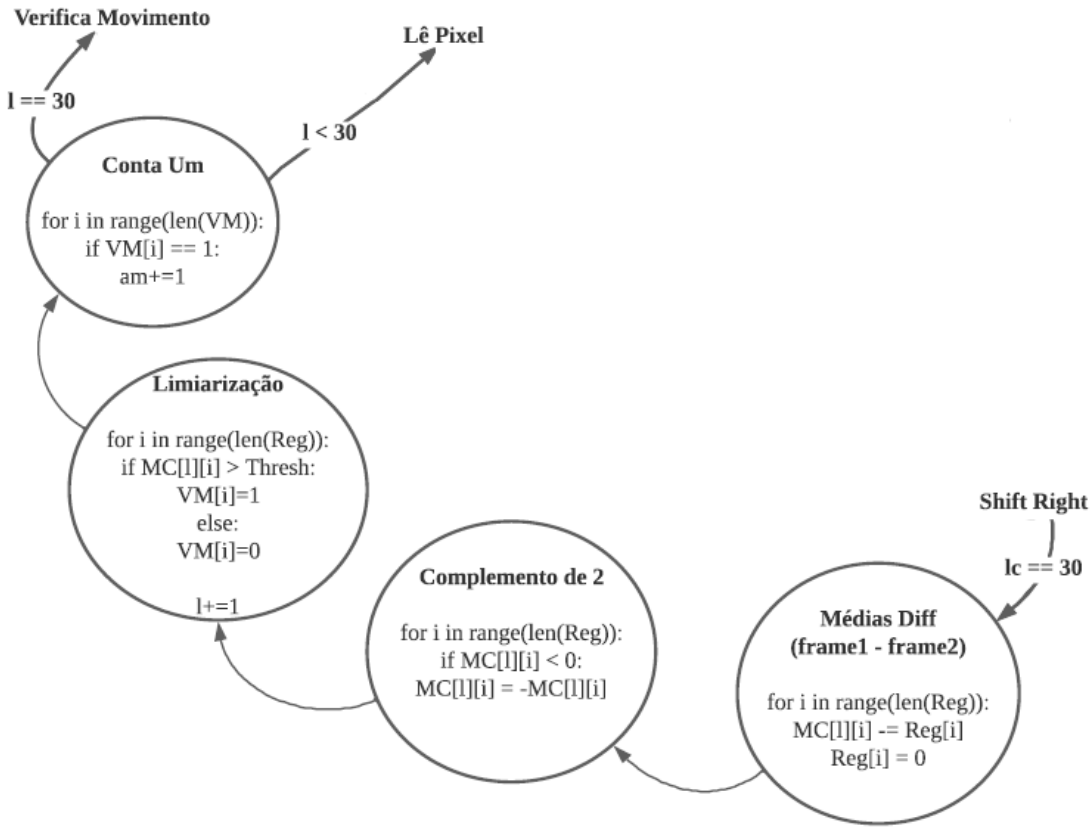


Figura 24 – Estados: Médias Diff, Complemento de 2 e Limiarização

Ao atingir $l_c = 30$, a matriz $MC[30][40]$ armazena a intensidade média dos 1200 quadrantes que dividem a primeira imagem. A partir daí, a FSM passa a calcular a média

das regiões da segunda imagem, comparando-as com aquelas da primeira. Neste contexto, em **Médias Diff** o algoritmo subtrai os valores contidos em $MC[30][40]$, referente ao primeiro *frame*, com os valores do vetor $Reg[40]$ que neste momento armazena as médias de 40 regiões do segundo *frame*. Assim, a variação de intensidade média das regiões de *pixels* é calculada e substituída na matriz $MC[30][40]$. Como o resultado desta subtração pode ser negativo, o estado **Complemento de 2** garante que todos os valores sejam positivos, mantendo apenas o módulo dos resultados.

No programa base em *software*, o módulo é calculado de forma tradicional, elevando a diferença ao quadrado e depois encontrando a raiz quadrada do valor. Contudo, visando reduzir o número de operações aritméticas para acelerar o sistema, na descrição de *hardware*, optou-se por representar os números em complemento de dois, modificando aqueles que indicassem 1 em seu bit mais significativo. Ou seja, mantendo os valores positivos e positivando os valores negativos.

Voltando para máquina de estados, logo que o módulo da variação dos *chunks* entre as imagens é calculado e positivado, o estado **Limiarização** compara este módulo a um limiar de sensibilidade atribuído pelo usuário. Se a variação for maior que o limiar, conclui-se que houve movimento naquele quadrante e, para tanto, a posição, referente ao *chunk* atual, do vetor de movimento $VM[40]$ recebe 1. Caso contrário, recebe 0.

Nesta etapa da FSM existe um contador de linhas l que funciona de forma semelhante à variável l_c no estado **Aloca Média**, determinando em qual linha da matriz $MC[30][40]$ a subtração entre o primeiro e o segundo *frame* deve ser armazenada. Vale ressaltar que em *hardware*, assim como nos estados **Shift Right** e **Aloca Médias**, as operações realizadas entre as 40 posições dos registradores $Reg[40]$ e das 40 colunas de $MC[30][40]$ em **Médias Diff**, **Complemento de 2** e **Limiarização** ocorrem de forma paralela. Além disso, a cada **Limiarização** o contador de linhas l é incrementado em uma unidade. Por outro lado, no estado **Conta Um**, o algoritmo incrementa o acumulador de movimento am em uma unidade para cada posição de $VM[40]$ que for igual a um, de forma sequencial. Ainda no estado **Conta Um**, enquanto l não atingir um valor total de 29 unidades, a FSM volta para o estado **Lê Pixel**, indicando que o segundo quadro ainda está sendo processado. Caso contrário, o próximo estado será **Verifica Movimento**, explicado na seção 3.2.6.

Em termos de um projeto RTL, esta etapa de comparação e limiarização entre as regiões do primeiro e do segundo *frame* pode ser representada pelo diagrama da figura 25. Em que o multiplexador controlado pela variável $sel2$, agora equivalente a l, determina qual linha da matriz $MC[30][40]$ será processada. Além disso, na etapa em questão, o multiplexador controlado pela variável $sel3$ está configurado para encaminhar o valor médio das regiões do segundo *frame* até a subtração com o valor médio das regiões do primeiro *frame*, armazenados em $MC[30][40]$, bem como, o demultiplexador controlado

pela variável *sel*4 esta configurado de tal forma a substituir o resultado da subtração já positivado em $MC[30][40]$. A seguir o demultiplexador controlado pela variável *sel*5, também equivalente a l, determina qual das linhas da matriz $MC[30][40]$, agora com a variação média das intensidades entre o par de imagens, que será comparada ao limiar de detecção atribuído pelo usuário. O valor binário resultante de cada uma das colunas comparadas é armazenado no registrador $VM[40]$ e esses valores são somados e acumulados em am.

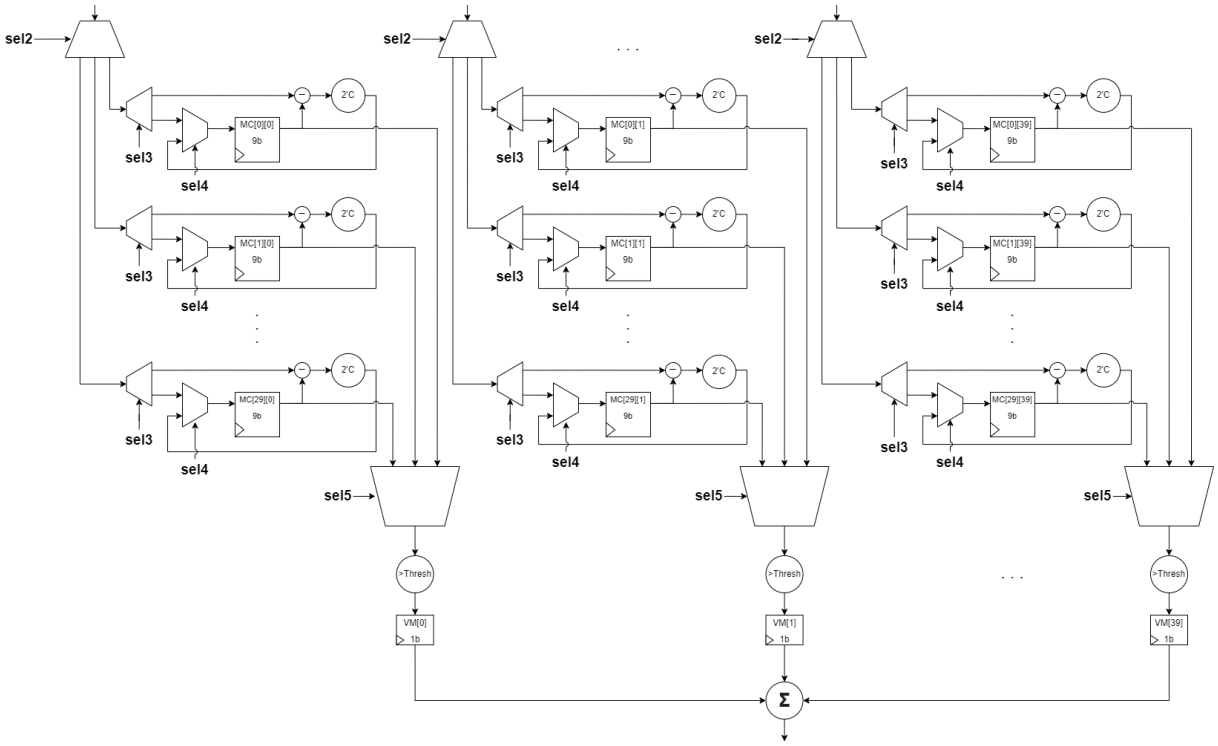


Figura 25 – Diagrama RTL: Comparação e Limiarização

3.2.6 Verificador de Movimento

Após a comparação entre todos os *chunks* das duas imagens, o acumulador de movimento am armazena um inteiro que pode ir de 0 a 1200, indicando algo entre: total ausência de movimento ou movimento em todos as regiões. Seguindo a métrica implementada no **Produtor de Frames**, o estado **Verifica Movimento** busca ao menos 25% de *chunks* com movimento para levantar a *flag* $result = 1$ que engatilha a captura de um trecho de vídeo proveniente do fluxo de imagem da câmera, chamado aqui de evento. No entanto, se am indicar um valor menor que 300 o estado **Verifica Movimento** abaixa a *flag* $result = 0$ e nada acontece. Porém independente do valor de $result$, o estado **Verifica Movimento** levanta a *flag* $ready = 1$, indicando que o algoritmo de detecção de movimento terminou de comparar um par de imagens. Finalmente, a FSM volta para o

estado **Init**, assim como observado na figura 26, reiniciando as variáveis e esperando pelo próximo *start*.

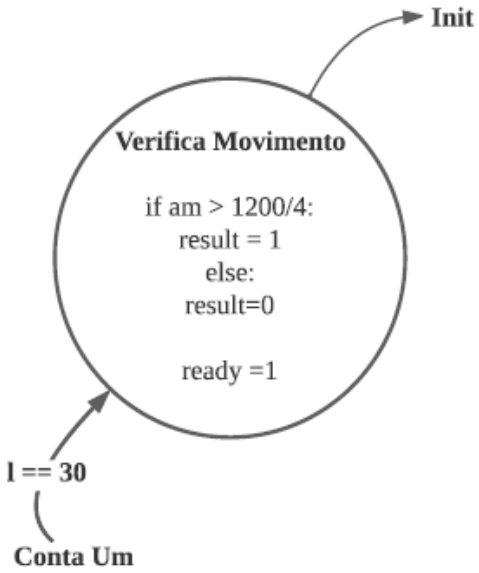


Figura 26 – Estado: Verifica Movimento

Em termos de um projeto RTL, esta etapa de Verificação de movimento pode ser representada pelo diagrama da figura 27.

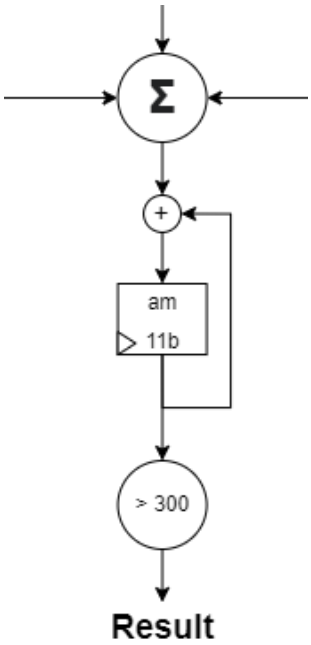


Figura 27 – Diagrama RTL: Verificação de movimento

3.2.7 Máquina de Estados e RTL Completos

As figuras 28 e 29 apresentam, respectivamente, a máquina de estados e o diagrama RTL, analisados nas últimas seções, em sua completude.

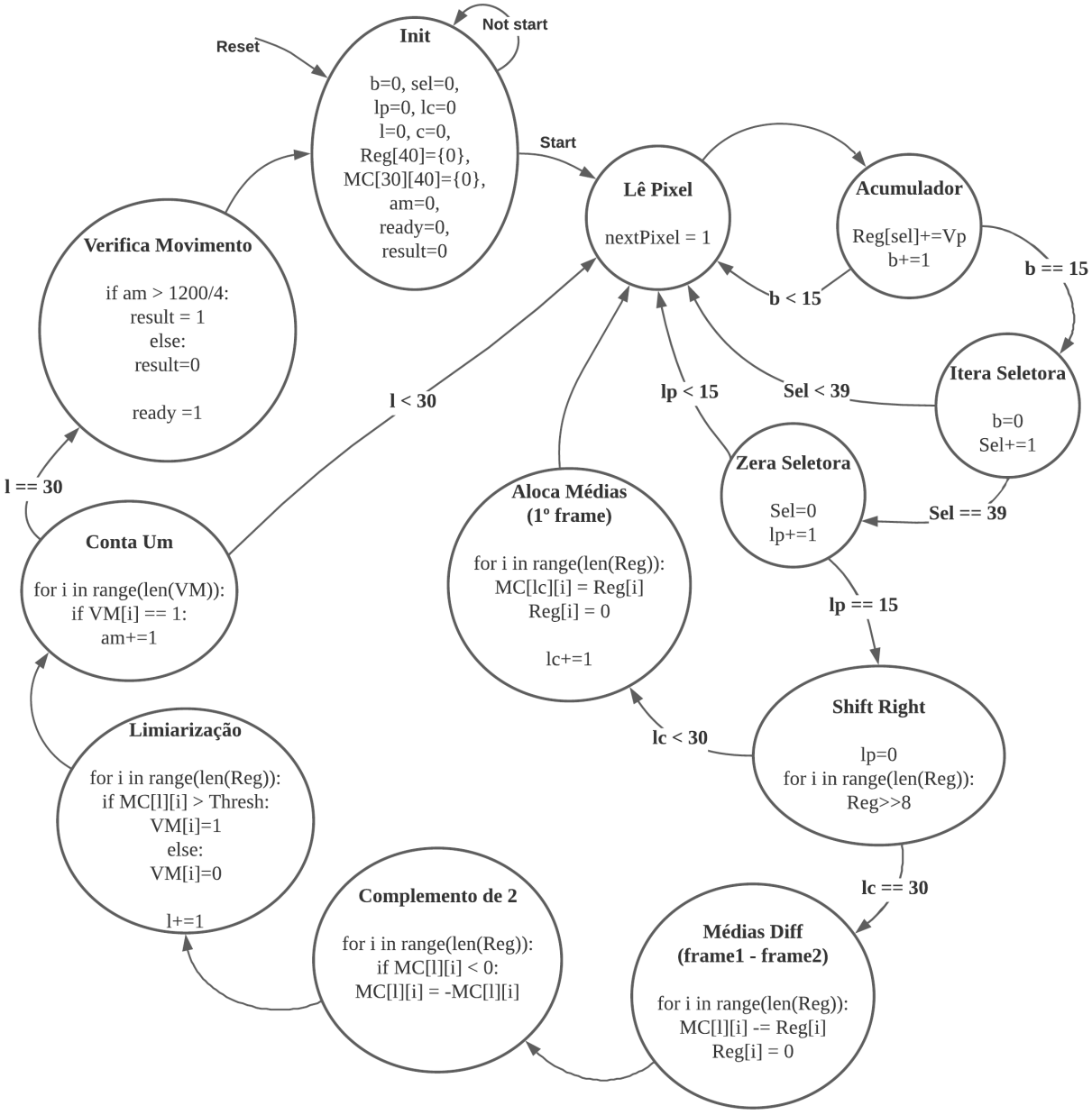


Figura 28 – FSM Completa

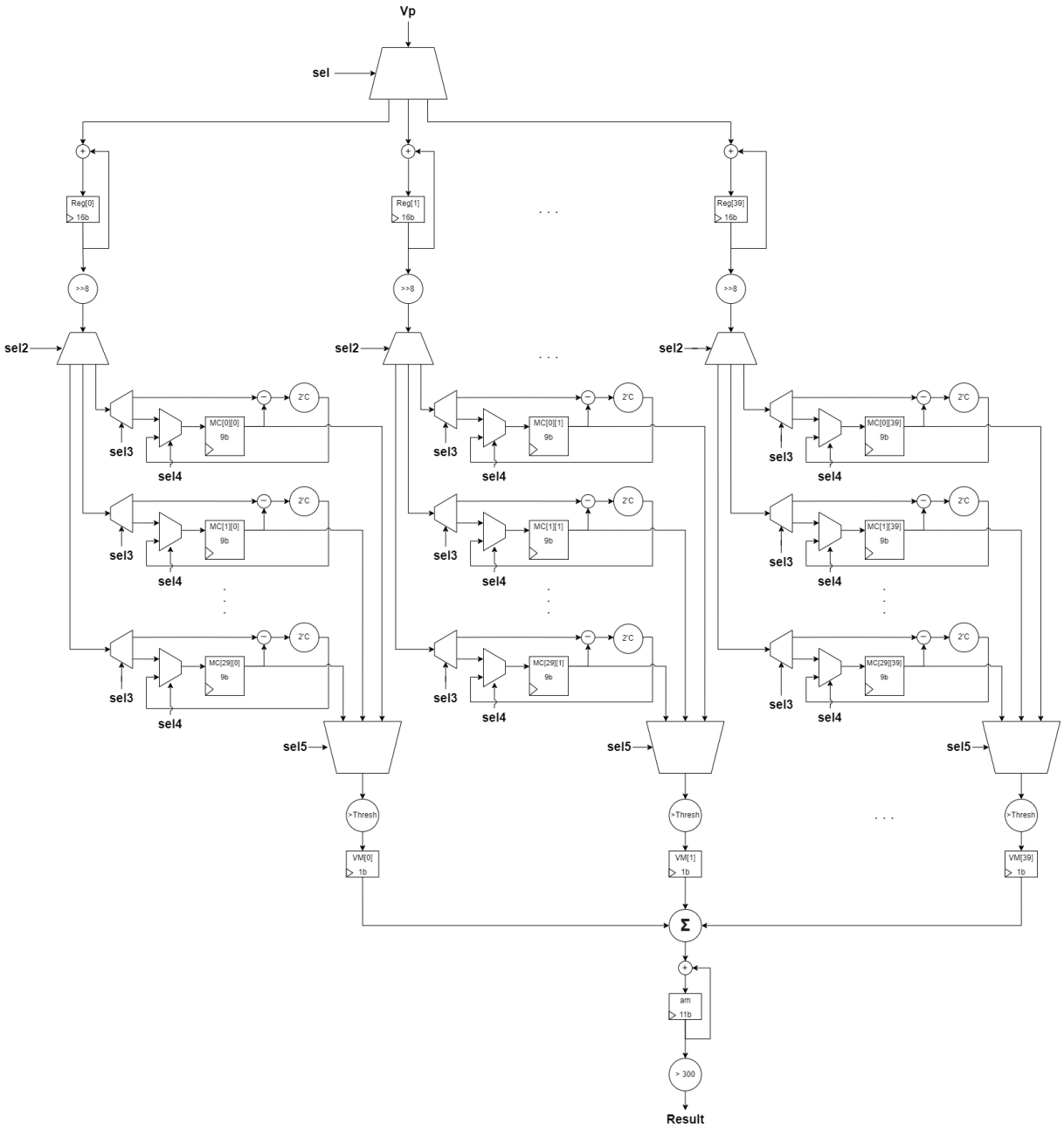


Figura 29 – RTL Completo

3.3 Validação da FSM em Python

Tendo em vista a máquina de estados descrita na seção 3.2, em um primeiro momento, descreveu-se uma aplicação Python, disponível em (FREITAS, 2021a), para validação em *software* do algoritmo proposto. Esta aplicação Python possui duas classes chamadas de *FrameCapture* e *MotionDetector* que, quando instanciadas como objetos, inciam duas *threads* computacionais. A primeira é responsável pela recepção, exibição e tratamento dos *frames* disponibilizados por uma fonte de captura, tal qual um câmera ou um arquivo de vídeo em mp4. A segunda *thread* opera a máquina de estados proposta

através do objeto *MotionDetector*, em que cada um de seus métodos representa um estado da FSM. Além disso, a aplicação em Python coloca um FIFO entre os objetos para enfileirar as imagens recebidas pelo *FrameCapture* que posteriormente serão processadas pelo *MotionDetector*.

Em primeiro momento, a FSM em Python foi validada em tempo real utilizando uma *webcam* com resolução de 0.3 Megapixel e capturando imagens com dimensões de 640x480p. Os resultados dessa validação estão descritos na seção 4.1. Em um segundo momento, a aplicação utilizou um vídeo de teste, disponível em (FREITAS, 2022g), como fonte de captura para termos de comparação com a simulação comportamental do algoritmo em VHDL (vide seção 3.5). Os resultados dessa comparação estão descritos na seção 4.3.

Para os dois testes descritos no parágrafo anterior, o programa foi executado em um processador Intel i5-10210U Quad Core com 8Gb de memória RAM, rodando um sistema operacional Ubuntu 20.04.

3.4 Descrição de Hardware da FSM

Tendo a máquina de estados validada em *software*, descreveu-se o algoritmo em VHDL através do *framework* Vivado que possui um conjunto de ferramentas para síntese e análise de projetos de linguagem de descrição de *hardware* oferecida pela Xilinx (XILINX, 2022). O código de referencia da FSM em VHDL está disponível em (FREITAS, 2022e).

Baseado nos estados, variáveis e sinais descritos na seção 3.2, a FSM em VHDL foi estruturada dentro de uma entidade que possui 5 entradas, dentre as quais, um bit para o sinal de *clock*, um bit para o sinal de *start*, um bit para o sinal *reset*, um bit para o sinal indicador de *pixel* disponível, chamado de $pixelAvailable$, e um barramento com um byte de comprimento para receber o valor dos *pixels*, chamado de $pixelValue$. Além disso, esta entidade possui 3 saídas, um bit para o sinal que pede pelo próximo *pixel*, chamado de $nextPixel$, um bit para o sinal que indica quando a detecção de movimento terminou, chamado de *ready* e um bit para o sinal que indica o resultado da detecção de movimento, chamado de *result*.

O comportamento desta entidade, foi descrito através de três processos independentes, sendo dois síncronos, ou seja orquestrados pelo sinal de *clock*, e um assíncrono. O primeiro processo síncrono é sensível apenas ao *clock* e é responsável pela mudança de estados, em que o estado atual recebe o próximo estado, se o sinal *reset* for zero, caso contrário o estado atual aponta para o estado $Init$ (vide seção 3.2.2). Já o segundo processo síncrono é sensível aos 5 sinais de entrada, bem como, ao estado atual e tem como responsabilidade lidar com as variáveis de controle e com os registradores da FSM, ou seja, a cada ciclo de *clock* e, também, a depender do estado atual e dos sinais *start*,

$reset$, $pixelAvailable$ e $nextPixel$, este processo atualiza as variáveis de controle b, Sel, l_P, l_C, l e am, bem como, atualiza os registradores $Reg[40]$, $MC[30][40]$ e $VM[40]$, melhor descritos na seção 3.2. O processo assíncrono, por sua vez, é sensível tanto ao estado atual, quanto aos sinais de entrada e também às variáveis de controle. Seu objetivo é determinar qual será o próximo estado baseado na relação das variáveis da sua lista de sensibilidade.

3.5 Simulação Comportamental da FSM

O Vivado disponibiliza uma ferramenta de simulação comportamental para testar a entidade descrita na seção anterior. Para tal procedimento, escreve-se um outro código também em VHDL, disponível em (FREITAS, 2022d), conhecido como *Testbench* (TB), que instancia a entidade como um componente, opera seus sinais de entrada e verifica seus sinais de saída ao longo do tempo para validar o algoritmo proposto.

3.5.1 Arquivo de *Pixels*

Para a simulação é necessário um vídeo de teste que alimentará o algoritmo com os valores de seus *pixels*. Para tanto, um vídeo com 3 segundos de duração foi gravado a 6 fps, acumulando um total de 18 *frames* em escala de cinza, disponível em (FREITAS, 2022g). Considerando que cada *frame* possui dimensão de 640x480p, o vídeo é composto por um total de 5.529.600 *pixels*. Com o intuito de disponibilizar este vídeo à ferramenta de simulação do Vivado, foi necessário criar um arquivo de texto, em que cada uma de suas linhas registra um byte referente a um *pixels* do video de teste. Este arquivo de texto foi gerado com auxilio de um programa em Python, disponível em (FREITAS, 2022f), que utiliza do capturador de video da OpenCV, vide (OPENCV, 2021a), e as funções de manipulação de arquivos, nativas do Python, para operar a escrita dos *pixels* no arquivo de texto, que está disponível em (FREITAS, 2022c).

3.5.2 Testbench

Além de instanciar a entidade como componente, no TB disponível em (FREITAS, 2022d), um processo síncrono foi declarado para abrir e ler o arquivo de pixels e, a cada ciclo de *clock*, inserir um novo byte no barramento $pixelValue$ do componente que opera a FSM. Não obstante, fora do processo de leitura do arquivo de *pixels*, os sinais de *start* e *reset* são acionados para garantir a inicialização e a reinicialização da FSM. Vale ressaltar que o *clock* foi declarado no TB como um sinal auxiliar com período de 10 ns. Além disso, pelo fato do processo de leitura do arquivo de *pixels* estar sincronizado em 100 Mhz, assim como o componente que opera a FSM, o sinal $pixelAvailable$ está sempre acionado como verdadeiro. Os resultados da simulação comportamental da FSM estão apresentados na seção 4.2.

3.6 Protótipo na plataforma PYNQ-Z2

Após validar o funcionamento da FSM a nível comportamental, o próximo passo é a concepção de um protótipo físico que utilize do algoritmo de detecção de movimento em vídeo através da placa de desenvolvimento PYNQ-Z2 que é o acrônimo em inglês para *Python Productivity for Zynq* modelo Z2 (TUL, 2021). A PYNQ é um projeto de fonte aberta da Xilinx que combina a linguagem e as bibliotecas Python com a série ZYNQ de *SoC FPGA* da Xilinx (XILINX, 2021a). A figura 30 a seguir apresenta a placa em questão.

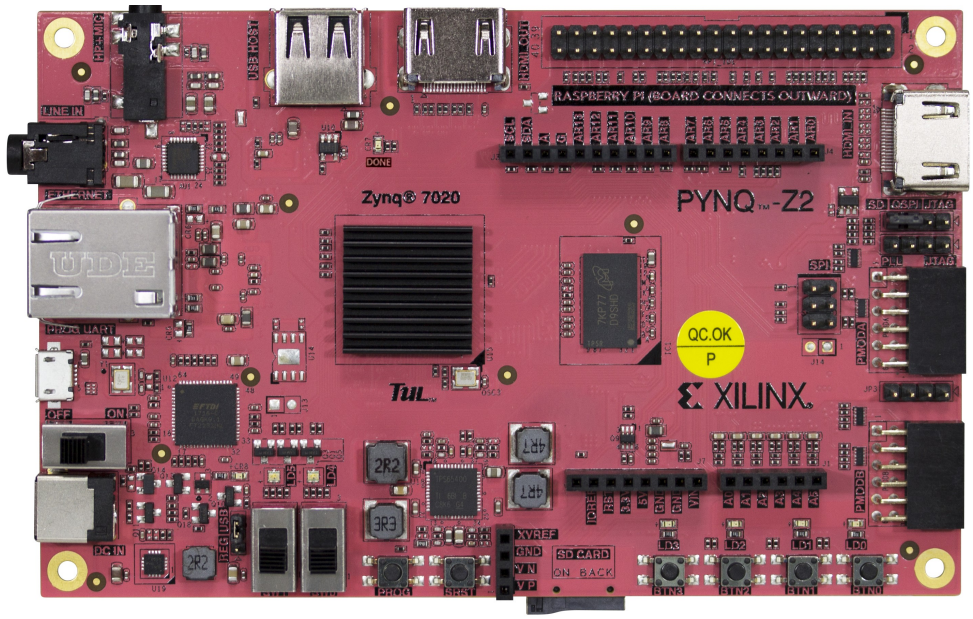


Figura 30 – SoC FPGA modelo PYNQ Z2 (Figura retirada de (TUL, 2021))

Neste contexto, a PYNQ possui um ambiente em *software*, conhecido como *Processing System* (PS), que opera um *framework* chamado Jupyter (JUPYTER, 2022) capaz de executar aplicações Python em seu processador dual-core ARM Cortex-A9. Além disso, os responsáveis pelo projeto da PYNQ desenvolveram um pacote Python (PROJECT, 2021) capaz de manipular o FPGA, permitindo que o algoritmo proposto neste trabalho fosse embarcado no ambiente de *hardware* da placa de desenvolvimento, ou também conhecido como *Programmable Logic* (PL), a partir de uma aplicação Python. Para tanto, o protótipo construído sobre a PYNQ tem com objetivo implementar fisicamente o algoritmo de detecção de movimento proposto no FPGA (PL) e testa-lo com auxilio do processador ARM (PS). As seções 3.6.1 e 3.6.2 descrevem, respectivamente, o funcionamento do PL e do PS para a concepção do protótipo que visa testar o conceito da solução.

3.6.1 Motion Detector e AXI Lite (PL)

Com o intuito de comunicar e sincronizar os ambientes de *software* (PS) e de *hardware* (PL) do projeto, um barramento AXI Lite foi anexado à entidade em VHDL que opera a FSM. O código referente a essa integração está disponível em (FREITAS, 2022b).

O protocolo AXI (DEVELOPER, 2022), acrônimo para *Advanced eXtensible Interface*, foi desenvolvido pela ARM (ARM, 2022) e é utilizado como barramentos de comunicação *on-chip*. Este protocolo facilita a comunicação entre blocos de projetos digitais, tais como, processadores, controladores de memória RAM e FPGAs, garantindo uma conexão direta entre as entidades chamadas de gerentes e subordinados. Neste contexto, e de forma geral, o gerente pode iniciar operações de leitura e escrita diretamente sobre os endereços de memória de seus subordinados. A figura 31 apresenta um diagrama que exemplifica um barramento AXI.

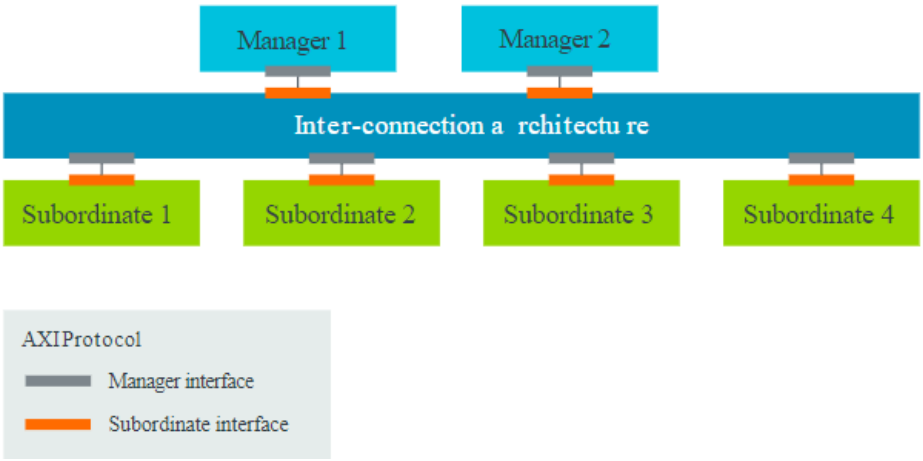


Figura 31 – Exemplificação do barramento AXI. (figura retirada de (DEVELOPER, 2022))

No Vivado, a Xilinx disponibiliza um catálogo de blocos IPs, ou *Intellectual Property*, com diversos modelos para barramentos AXI. Neste projeto, o barramento mais indicado seria o AXI Stream (DEVELOPER, 2010), pois, a partir de apenas um *handshake process*, que iniciaria a comunicação entre PS e PL, a cada ciclo de *clock* um novo *pixel* seria enviado à FSM de detecção de movimento, garantindo máximo desempenho e velocidade do algoritmo. Contudo, devido ao grau de complexidade, pouca expertise no assunto e ao tempo reduzido para implementação de tal barramento, optou-se por utilizar o AXI Lite, uma versão mais simples, que utiliza menos recursos e possui maior facilidade em sua implementação. Porém, o AXI Lite é recomendado apenas para controle de registradores e, para cada operação de leitura ou escrita, necessita de um novo *handshake process*. Ou seja, através do AXI Lite, a transferência de um *pixel* do PS para o PL demanda até 4

ciclos de *clock*, bem como, para o acionamento das *flags nextPixel* e *pixelAvailable*, melhor explicadas em 3.4, reduzindo o desempenho do algoritmo proposto, mas ainda assim validando seu funcionamento.

De qualquer forma, através da ferramenta de *block design* do Vivado foi possivel integrar o protocolo AXI entre o PS e o PL do projeto, proporcionando a comunicação entre o programa Python suportado pelo o ARM, explicado na seção a seguir, e a máquina de estados em VHDL embarcada no FPGA, apresentada na figura 28. A figura 32 apresenta a estrutura de blocos do projeto implementado.

Vale a pena mencionar que a interface AXI Lite realiza um mapeamento em memória dos periféricos ou co-processadores conectados e suporta até 256 registradores de 32 ou 64 bits de comprimento cada. Neste projeto, utilizou-se 9 registradores de 32 bits destinados a manipular as entradas e saídas da entidade detectora de movimento, descrita em 3.4, dentre as quais, os sinais de entrada, $start$, $pixelAvailable$, $pixelValue$ e $nextPixel$ e as saídas $ready$ e $result$. Os 3 registradores restantes foram destinados a sinais de saída complementares que foram adicionados à entidade em VHDL com o intuito de depurar o sincronismo entre os ambientes de *software* e *hardware* do projeto. Estes sinais complementares foram chamados de $pixelPosition$, $pixelAccumulator$ e $motionAccumulator$ e recebem, respectivamente, o contador de bytes b (vide seção 3.2.3), a primeira posição do registrador $Reg[40]$ (vide seção 3.2.3), e o acumulador de movimento am (vide seção 3.2.5).

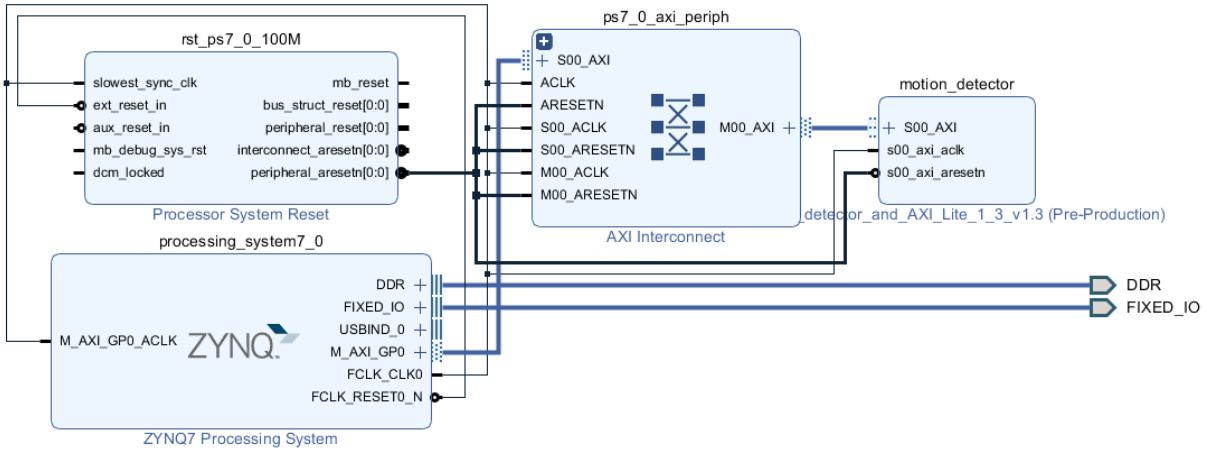


Figura 32 – Block Design: PS, PL e periféricos

Não obstante, percebe-se da figura 32 que além dos blocos principais *processing_system7_0*, referente ao ARM, e o *motion_detector*, referente a FSM, os demais blocos, *rst_ps7_0_100M* e *ps7_0_axi_periph* são periféricos responsáveis, respectivamente, pelo processo de reinicialização do projeto, controlado pelo PS, e pela interconexão, através do barramento AXI, entre PS e PL.

3.6.2 Programa do *Processing System*

Como já mencionado a PYNQ suporta um ambiente em *software* embarcado em seu processador ARM que roda uma distribuição Linux Ubuntu em que uma de suas aplicações é a plataforma Jupyter NoteBook. Nesta Plataforma foi desenvolvido um código em Python, disponível em (FREITAS, 2022a), que carrega o *bitstream* do *block design*, explicado na seção anterior, através do pacote conhecido como Overlays. Uma vez carregado, o Overlay opera a interface do AXI Lite através dos métodos *.read(addr)* e *.write(addr, data)* que recebem como argumento o endereço de memória em que se deseja ler ou escrever e, para o caso do método de escrita, recebe também o valor que será escrito. Em (PROJECT, 2018) o projeto PYNQ detalha o funcionamento do Overlay e de seus métodos. Esta ferramenta possibilita que o programa Python do PS leia e escreva das saídas e nas entradas da entidade detectora de movimento embarcada no PL. Além disso, como o intuito deste protótipo é validar a FSM quando implementada em *hardware* e testar o conceito de comunicação entre PL e PS, o mesmo arquivo de *pixel* utilizado na simulação comportamental do Vivado, descrita na seção 3.5, foi importado para o disco da PYNQ e disponibilizada para aplicação do PS através do pacote de manipulação de arquivos nativo do Python.

Neste contexto, na primeira célula do Jupyter, após indicar o caminho até o arquivo de *bitstream*, o programa verifica a integridade do Overlay apresentando um dicionário com os parâmetros do *block design* carregado. Nas duas células seguintes, o programa apresenta a frequência de operação dos *clocks* disponíveis na placa de desenvolvimento, bem como, define os endereços de memória para cada um dos registradores do AXI Lite, ou seja, para cada uma das entradas e saídas da entidade detectora de movimento carregada no PL. A última célula desta aplicação recarrega o Overlay através do método *ol.download()*, enviando novamente o .bit para o PL, e define um objeto chamado *md* que aponta para o endereço base do IP *motion_detector*, vide figura 32. Na sequência, a aplicação do PS abre o arquivo de *pixel* através da diretiva $fd = open('Movendo.txt')$ e inicia a FSM chamando o método *md.write(start_addr, 1)* que aciona a *flag start*. Em seguida, a aplicação entra em um laço de repetição que, primeiramente, lê todos os sinais de saída da FSM, através do método *md.read()*, e os apresenta em tela monitora. Para tanto, se a *flag nextPixel* for igual a 1, a aplicação lê uma linha do arquivo de *pixels*, insere o valor lido no barramento $pixelValue$, através do método *md.write(pixelValue_addr, pixel)*, e levanta a *flag pixelAvailable* através do método *md.write(pixelAvailable_addr, 1)*. A aplicação do PS fica nesse laço até que a *flag ready* seja levantada pela FSM. Neste momento, a aplicação lê o valor de $result$ e o apresenta na tela monitora. Os resultados da simulação deste protótipo estão descritos na seção 4.4.

4 Resultados Obtidos

4.1 Validação da Máquina de Estados em *Software*

Como mencionado no capítulo 3, a primeira parte do trabalho aqui descrito teve foco na otimização do algoritmo do *Produtor de Frames*, ainda em *software*, mas propondo uma implementação eficiente para detecção de movimento em vídeo com dimensões de 640 por 480 *pixels*. Para tanto, na função principal do programa de teste, disponível em (FREITAS, 2021a), existem duas classes instanciadas como objetos: o *FrameCapture* para capturar as imagens da *webcam* e o *MotionDetector* que opera a FSM proposta (vide figura 28) para detectar movimento através da diferenciação temporal. Entre esses dois objetos existe uma *FIFO* que enfileira as imagens capturados pelo *FrameCapture* e as disponibiliza para o *MotionDetector*. No vídeo de demonstração disponível em (FREITAS, 2021c) percebe-se que existe um pequeno atraso no consumo das imagens da fila devido a velocidade da detecção de movimento que é menor que a velocidade de captura das imagens. Vale ressaltar que como o objetivo principal da pesquisa é implementar o algoritmo em *hardware*, o código de teste foi escrito como um laço infinito que opera uma estrutura de decisão para determinar qual é o estado atual e qual será o próximo estado da FSM. Contudo, este modelo não é a melhor forma de executar a detecção de movimento proposta em um código sequencial em *software*, porém facilita a descrição em *hardware* do algoritmo, pois a lógica do código em Python já está estruturada de forma semelhante ao que se fará em VHDL.

De qualquer forma, independentemente dos atrasos relacionados a maneira com que o código em *software* foi estruturado, a execução do programa de teste mostra na figura 33 o estado sem movimento onde o acumulador de movimento é inferior a 300 unidades. Já na figura 34 o algoritmo indica movimento quando o objeto em cena se move e o acumulador passa da marca de 300 *chunks* moveis. Finalmente, na figura 35 o acumulador de movimento volta ao patamar inicial indicando ausência de movimento em cena. Este resultados validam o funcionamento em *software* do modelo de referência da máquina de estados proposta que foi implementada em *hardware*, e tem seus resultados descritos nas próximas seções.

É importante comentar que os resultados apresentado em vídeo (FREITAS, 2021c) e através das figuras 33, 34 e 35 foram alcançados sem a utilização do filtro espacial de borramento. Como comentando anteriormente, este filtro suaviza as áreas de alta frequência das imagens o que torna a detecção de movimento menos sensível as variações bruscas de intensidade entre os *pixels* vizinhos. Obviamente, este primeiro teste de validação foi executado em um ambiente fechado com baixa variação luminosa. Talvez em ambientes

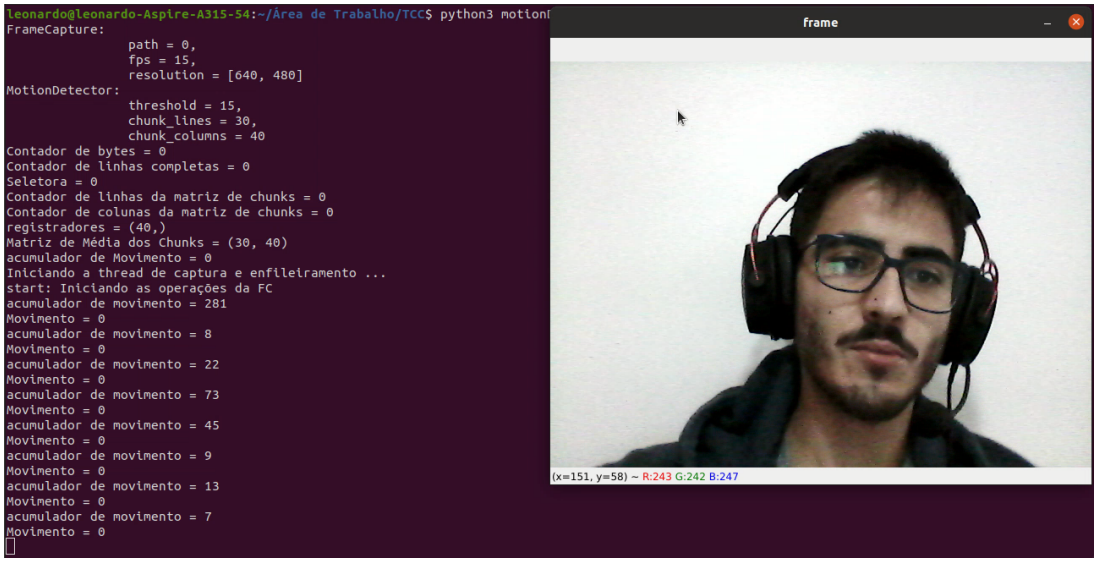


Figura 33 – Patamar inicial sem movimento ($AM < 300$).

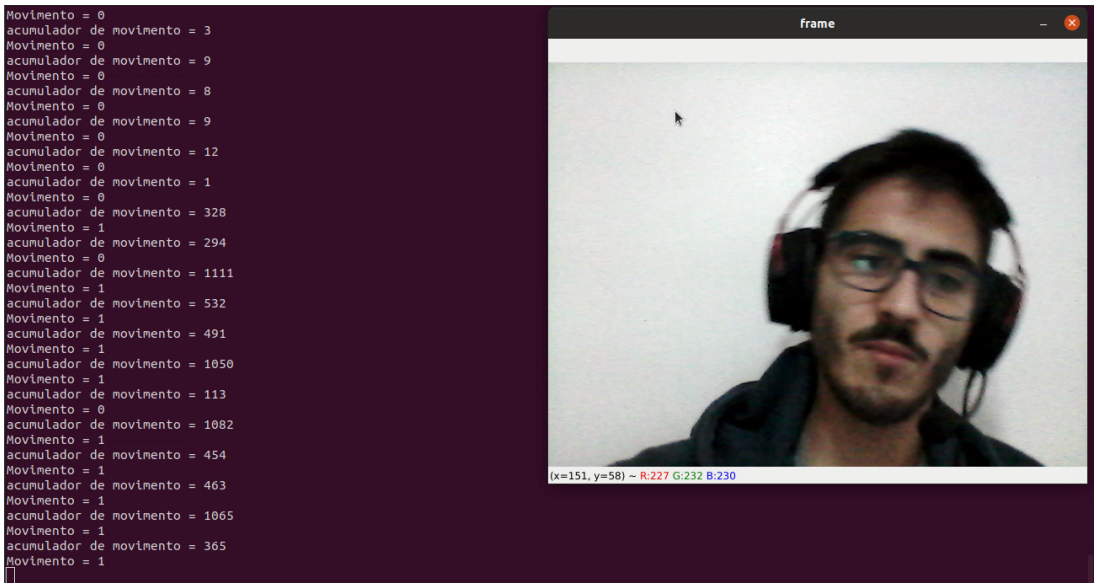


Figura 34 – Cena com movimento ($AM \geq 300$).

externos sobre o efeito de sombras seja necessário a aplicação do filtro de borramento para evitar falsos positivos na detecção de movimento.

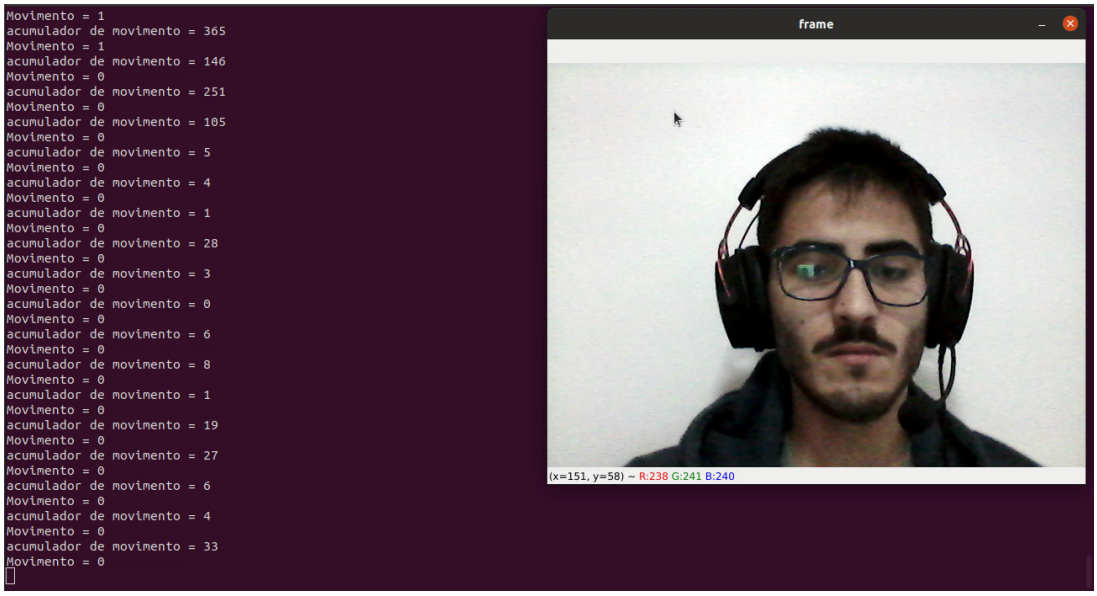


Figura 35 – Cena sem movimento ($AM < 300$).

4.2 Resultados da Simulação Comportamental

Como resultado da simulação comportamental via Vivado, obtém-se um linha do tempo onde são projetadas as variáveis provenientes da FSM e do TB. A figura 36 apresenta o incio da simulação e evidencia o instante, em 15 ns, onde o primeiro sinal de *reset* é levantado, zerando todas as variáveis de controle e registradores da FSM e também colocando seu estado atual como **init**.

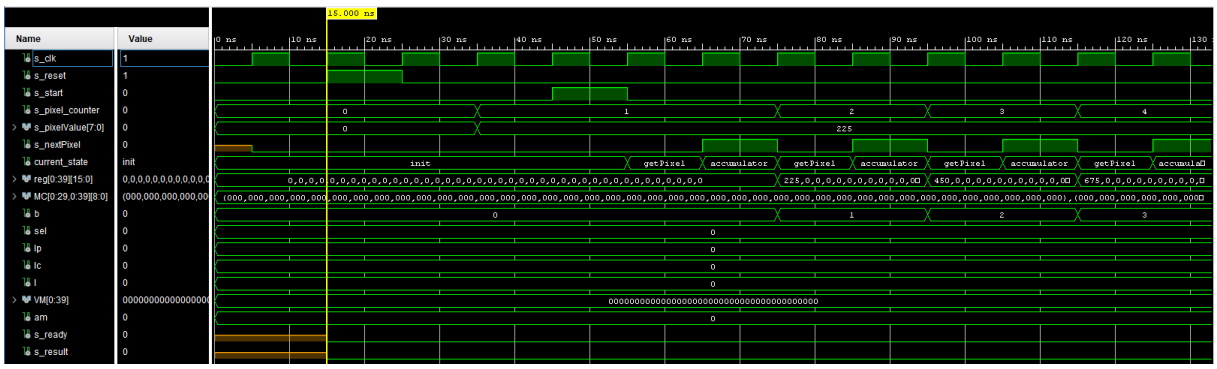


Figura 36 – Inicio da Simulação e reset acionado a 15 ns

Na figura 37, o instante de 45 ns evidencia o primeiro acionamento do sinal *start* o que, pelo fato do estado atual ser o **init**, no próximo ciclo de *clock* induz a FSM a avançar para o estado **getPixel**, equivalente ao estado Lê Pixel descrito na seção 3.2.3.

Na figura 38, com o inicio do ciclo leitura e acumulo de *pixels*, vide seção 3.2.3, percebe-se que ao final do estado **getPixel**, o sinal *nextpixel* está levantado, indicando que a FSM está pronta para receber e acumular o valor do *pixel*, na primeira posição

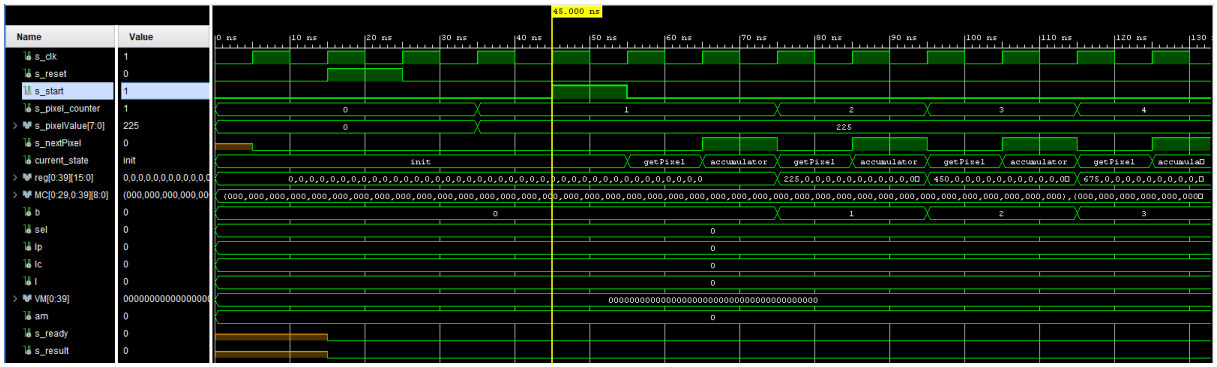


Figura 37 – *Start* acionado em 45 ns, inciando o ciclo de leitura e acumulo de *pixels*

do registrado $reg[0:39]$, quando atingir o estado **accumulator**, equivalente ao estado Acumulador descrito na seção 3.2.3.

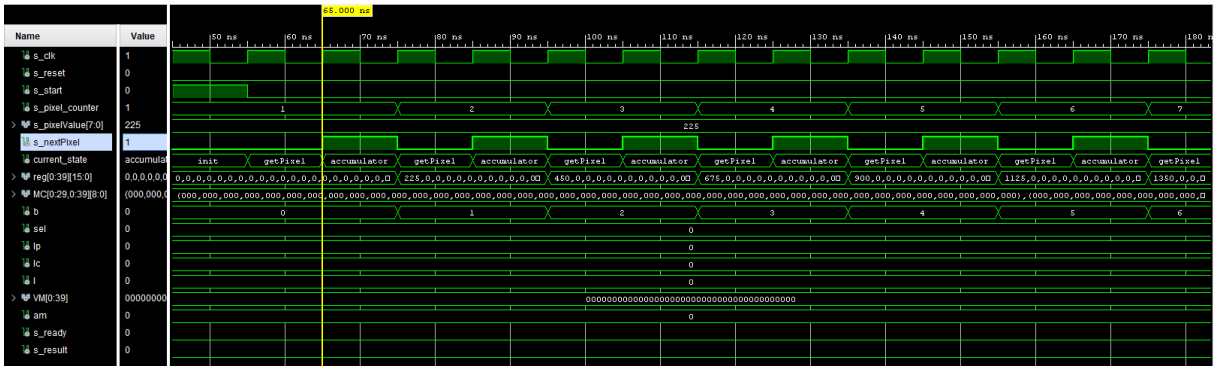


Figura 38 – *nextPixel* levantado em 65 ns. FSM preparada para acumular o valor do *pixel*

Gradualmente, o algoritmo alterna entre **getPixel** e **accumulator** até o contador de bytes b atingir 15 unidades e, como apresentado pela figura 39, alcançar o estado **selIterator**, equivalente ao Itera Seletora descrito na seção 3.2.3.

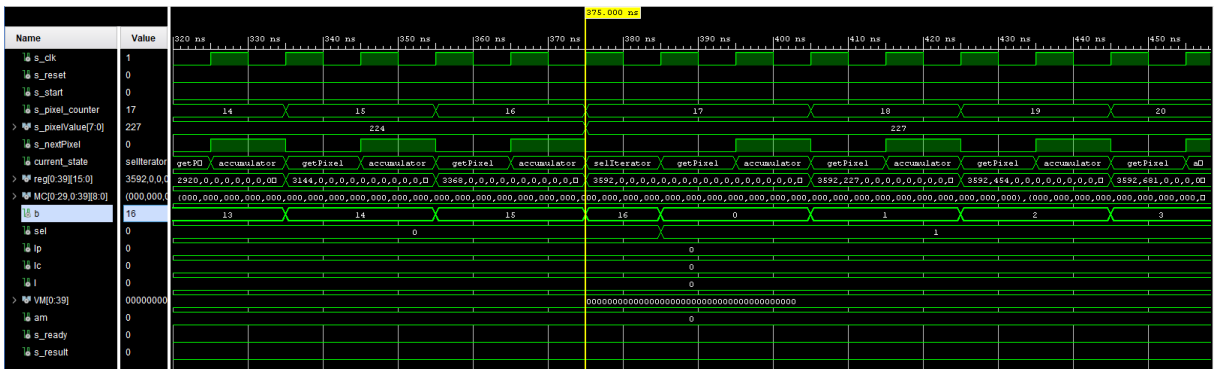


Figura 39 – Contador de bytes $b = 15$ em 375 ns. FSM alcança o estado Itera Seletora

Percebe-se na figura 40 que no estado **selIterator** a variável *sel* é incrementada em uma unidade, o que indica que o valor dos *pixels* lidos agora serão acumulados na segunda posição do registrador $reg[0:39]$. Além disso o contador de bytes b é zerado e mais um ciclo de leitura e acumulo de *pixel* se repete.

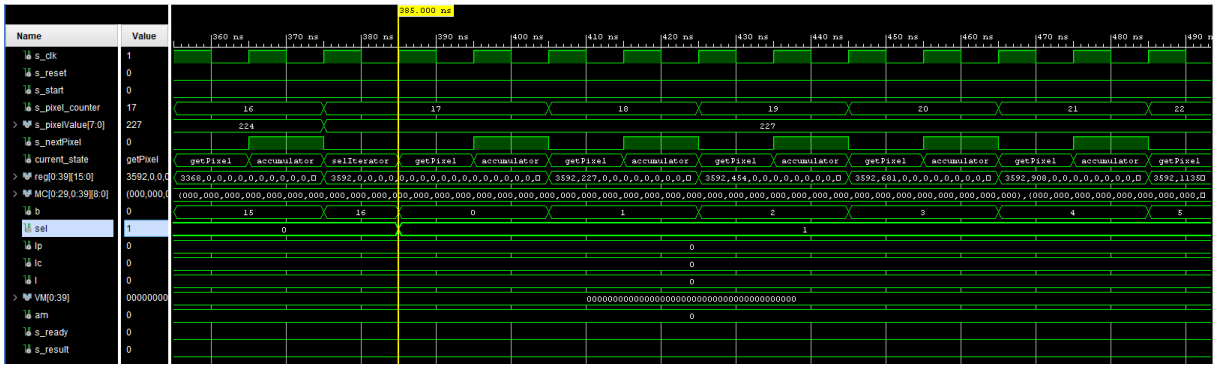


Figura 40 – *sel = 1* em 385 ns. O *pixels* passam a ser acumulados na segunda posição de *reg[0:39]*

Ainda na figura 40 é possível perceber que, até o estado **selIterator**, apenas a primeira posição do registrador $reg[0:39]$ armazena um valor maior que zero, no caso 3592. Contudo, com dois ciclos de *clock* a frente, a segunda posição do registrador $reg[0:39]$ armazena o valor de 227 equivalente àquele inserido em $pixelValue$.

Para tanto, o algoritmo segue em seu ciclo de leitura e acumulo de *pixels*, alternando entre as 40 posições do registrador $reg[0:39]$, até atingir $sel = 39$. Neste momento, assim como apresentado na figura 41, a FSM alcança pela primeira vez o estado **selZero**, equivalente ao estado Zera Seletora descrito na seção 3.2.3. No estado em questão, a variável *sel* recebe 0, bem como, o contador de linhas de *pixels* l_P é incrementado em uma unidade, indicando que uma linha completa da primeira imagem foi acumulada nos registradores. Fato este que também se reflete na variável *s_pixel_counter* que armazena um valor de 641. Esta variável é um sinal auxiliar do TB que soma uma unidade a cada linha lida do arquivo de *pixels*. Ou seja, o valor de 641 significa que o processo de leitura do TB já inseriu o primeiro byte da segunda linha de *pixels* da primeira imagem em $pixelValue$.

Adiante, o algoritmo acumulará mais outras 15 linhas de *pixels* para completar os 256 *pixels* de cada um dos 40 *chunks*. Então, como pode ser observado na figura 42, com $l_P = 15$, a FSM alcança o estado **shiftRight**, melhor explicado em 3.2.4, que calcula a média dos valores armazenados em cada uma das posições de $reg[0:39]$. Na sequência, como $l_C < 30$, a FSM passa para o estado **AverageAlloc**, equivalente ao Aloca Médias descrito em 3.2.4, que armazena as médias recém calculadas na matriz $MC[0:29][0:39]$. Além disso, ainda no estado **AverageAlloc**, todas as posições de $reg[0:39]$ são zeradas, bem como a variável l_C é incrementada em uma unidade, indicando que o algoritmo passará a acumular a próxima linha de *chunks*, ainda da primeira imagem.

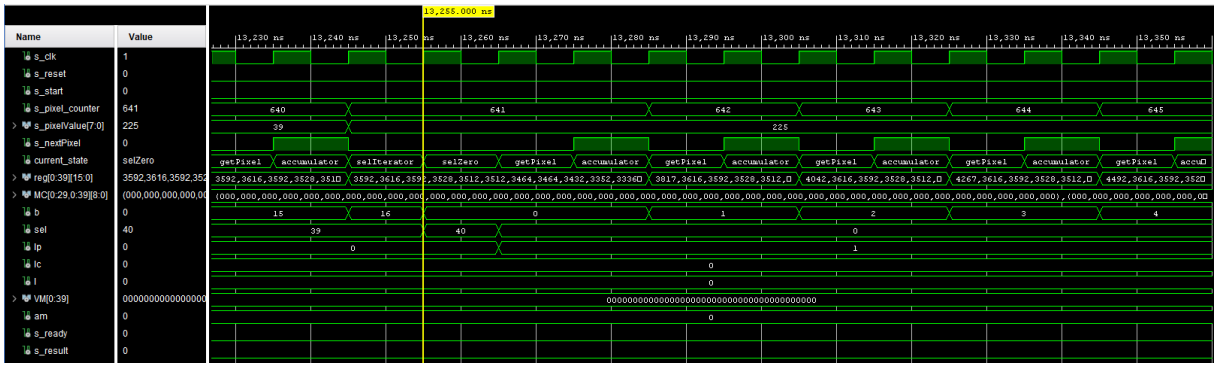


Figura 41 – *sel* = *39* em 13.255 ns. Uma linha de *pixels* completa foi acumulada

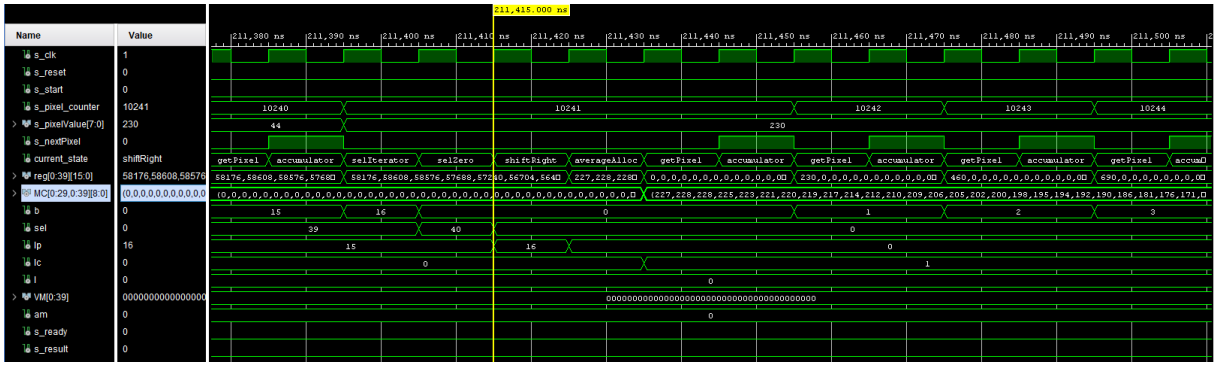


Figura 42 – $lp = 15$, $l_C < 30$ em 211.415 ns. Médias armazenadas na primeira posição de $MC[0:29][0:39]$

Assim como apresentado pela figura 43, Quando l_C atinge 30 unidades, significa que o algoritmo consumiu todas as linhas de *chunks* do primeiro *frame*, ou seja, atingiu a marca de $640 \cdot 480 = 307200$ *pixels* lidos. Para tanto, a partir dai, a FSM passa a acumular os *pixels* da segunda imagem, o que se reflete em $s\_pixel\_counter = 307201$.

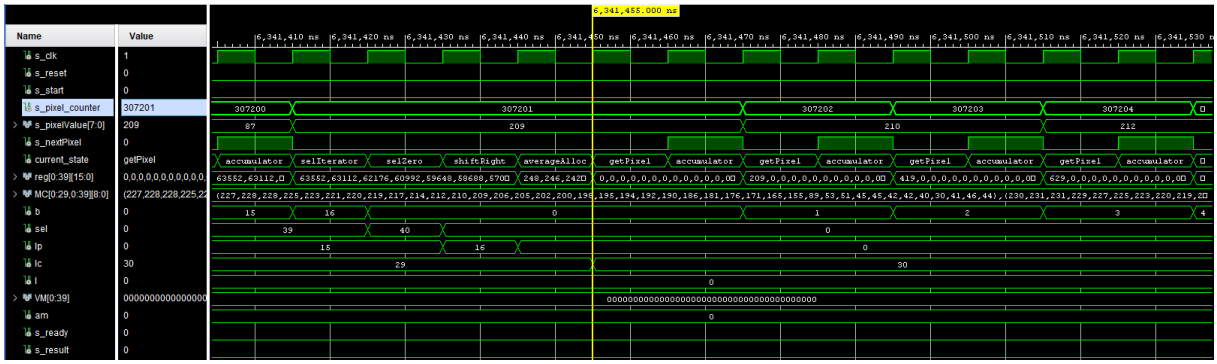


Figura 43 – l_C atinge 30 em 6.341.455 ns. Todos os 307200 *pixels* lidos da primeira imagem

Segundo a figura 44, como $l_C = 30$, na próxima vez que a FSM finalizar o estado **shiftRight**, calculando a intensidade médias dos primeiros 40 *chunks* do segundo frame, o próximo estado será **averageDiff**, equivalente ao estado Médias Diff detalhado em 3.2.5. Neste momento, o algoritmo subtrai os valores médios dos *chunks* do segundo *frame*, armazenados em $reg[0:39]$, dos valores da primeira linha da matriz $MC[0:29][0:39]$, que ainda guarda os valores médios dos *chunks* do primeiro *frame*. Contudo, após a subtração, o resultado é substituído na primeira linha da matriz $MC[0:29][0:39]$. Ou seja, a cada **averageDiff** uma linha da matriz $MC[0:29][0:39]$, definida pela variável l, tem seus valores substituídos pela subtração entre 40 *chunks* do segundo e o do primeiro *frame*. Além disso, a cada **averageDiff** a variável l é incrementada em uma unidade.

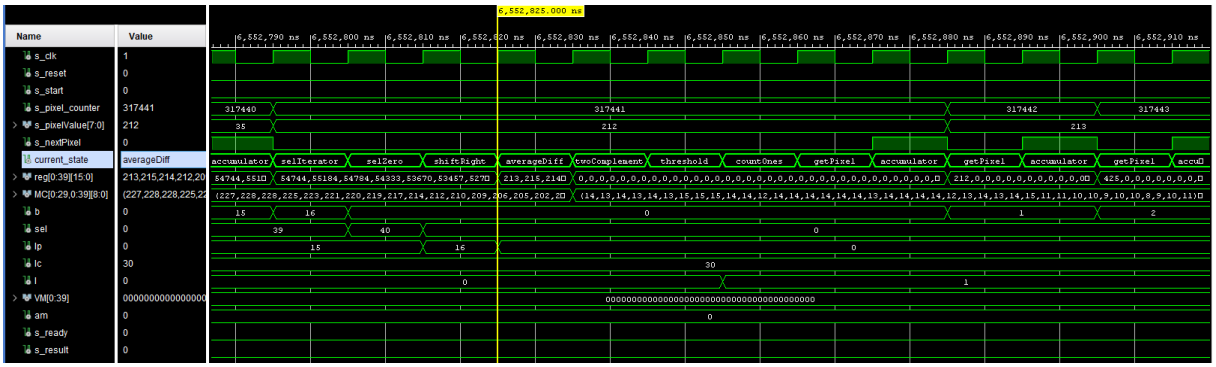


Figura 44 – A primeira linha de *chunks* é subtraída entre a primeira e a segunda imagem

Após a subtração em **averageDiff**, os valores armazenados em $MC[0:29][0:39]$ podem ser negativos, mas logo serão positivados em **twoComplement**, equivalente ao estado Complemento de 2 detalhado em 3.2.5. Em seguida, garantindo que a variação entre as regiões de *pixels* no par de imagens seja positiva, no estado **threshold**, este valores são comparados a um limiar que, no caso desta simulação, foi configurado como 15. Ou seja, aqueles valores da linha l de $MC[0:29][0:39]$ que forem maiores que 15 identificam uma região com movimento. Para tanto, a posição correspondente ao *chunk* é registrada no vetor $VM[0:39]$ como 1 para movimento, ou como 0 para ausência de movimento. Todavia, percebe-se da figura 44 que especificamente nessa primeira rotina de comparação e limiarização nenhum dos *chunks* foi superior a 15. Ou seja, mantendo todas as posições de $VM[0:39]$ nulas. Além disso, ao final da limiarização, a FSM atinge o estado **countOnes** que conta a quantidade de 1s no vetor $VM[0:39]$ e acumula esta contagem na variável *am* que será utilizada mais a frente para determinar se houve ou não movimento entre as imagens comparadas. Vale ressaltar que tanto o estado **threshold**, equivalente ao Limiarização, quanto o **countOnes**, equivalente ao Conta Um estão melhor explicados na seção 3.2.5.

Para exemplificar o acumulo de movimento, a figura 45 apresenta a próxima rotina de comparação e limiarização, onde $l = 1$ e a segunda linha de $MC[0:29][0:39]$ possui apenas um valor maior que 15. Desta forma, colocando 1 na décima quarta posição do vetor $VM[0:39]$ o que resulta, após o estado **countOnes**, em um acumulo de uma unidade na variável *am*.

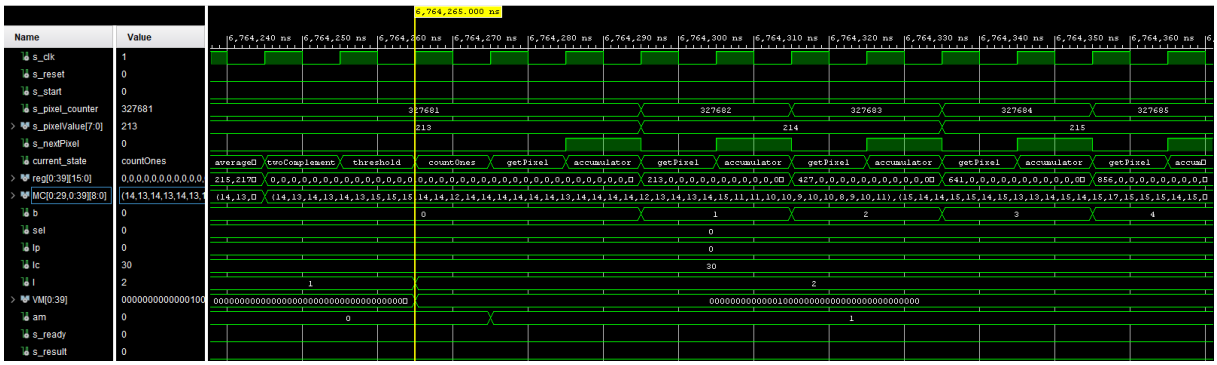


Figura 45 – $l = 1$ em 6.764.265 ns. $am = 1$ após **countOnes**

Após acumular um total de 614.400 *pixels* e ter comparado as 30 linhas de *chunks* entre os *frames* capturados, o algoritmo atinge sua etapa final, onde *am* acumula a quantidade de regiões em que houveram movimento, no caso deste primeiro par de imagens, $am = 294$, assim como apresentado na figura 46. Neste ponto, o estado atual, **motionCheck**, verifica se pelo menos um quarto dos 1200 chunks acusaram movimento. Como não é o caso, pois $am < 300$, a FSM levanta a flag *ready* como um, mas mantém *result* em zero. Vale ressaltar que o estado **motionCheck** equivale ao estado Verifica Movimento discutido em 3.2.6.

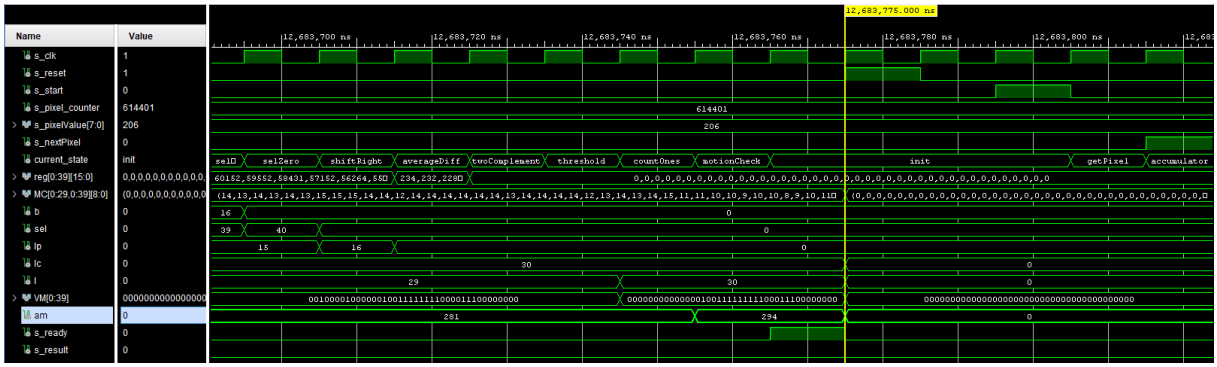


Figura 46 – com 614.400 *pixels* lidos, como $am = 294$, $result = 0$

Ainda da figura 46, percebe-se que, pelo menos em termos de simulação comportamental, se for garantido um *pixel* a cada ciclo de *clock*, ou seja a cada 10 ns, o algoritmo compara um par de frames, com resolução espacial de 640x480p, em apenas 12,68 ms.

Para exemplificar outros resultados da detecção de movimento, as figuras 47 e 48 apresentam a comparação dos próximos dois pares de *frames* lidos do arquivo de pixel (vide seção 3.5.1).

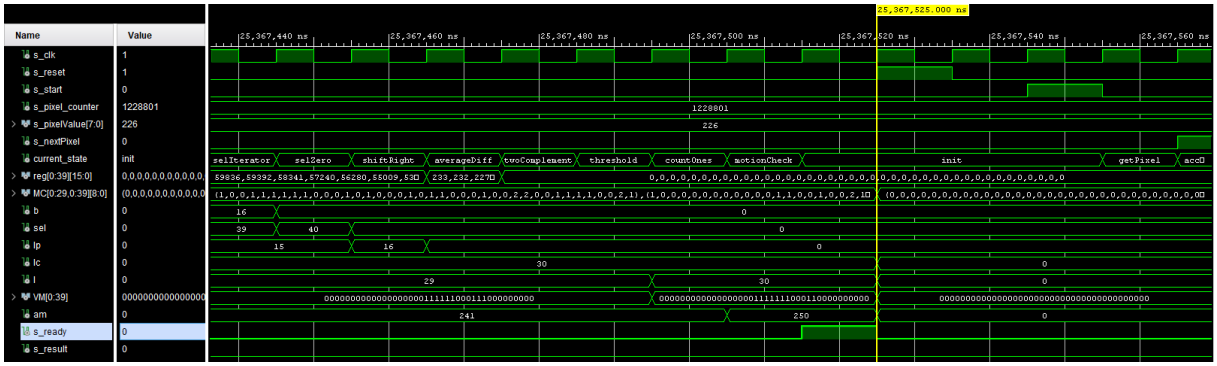


Figura 47 – Com 1.228.800 *pixels* lidos, como $am = 250$, $result = 0$

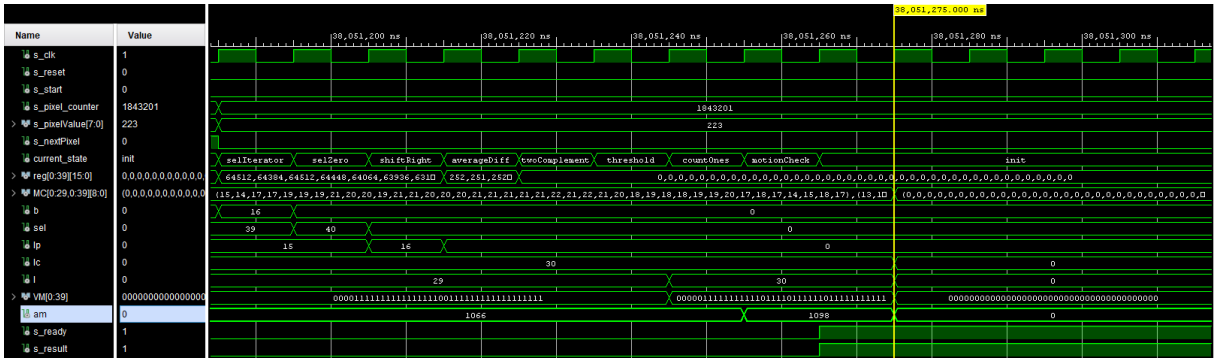


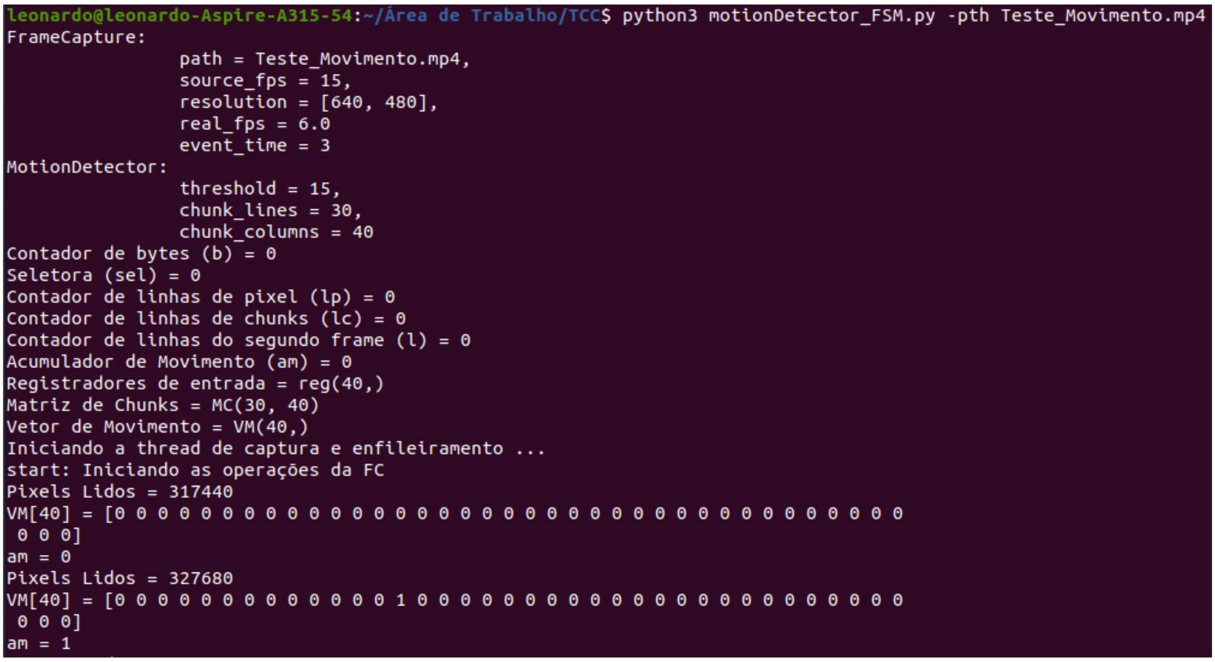
Figura 48 – Com 1.843.200 *pixels* lidos, como $am = 1098$, $result = 1$

4.3 Comparação entre a simulação comportamental em HW e SW

Para validar se o algoritmo proposto funciona de forma equivalente tanto em VHDL, quanto em Python, o mesmo trecho de vídeo, disponível em (FREITAS, 2022g), foi processado pelo programa de teste escrito em Python, disponível em (FREITAS, 2021a).

Observa-se na figura 49 a equivalência nos resultados apresentados nas figuras 44, 45 onde no primeiro ciclo de comparação e limiarização, aos 317.440 *pixels* lidos, o vetor de movimento $VM[0:29]$ está completamente nulo, o que resulta em $am = 0$. Contudo, aos 327.680 *pixels* lidos, o vetor de movimento $VM[0:29]$ indica que apenas na 14ª região da segunda linha de *chunks* houve movimento, o que resulta em $am = 1$.

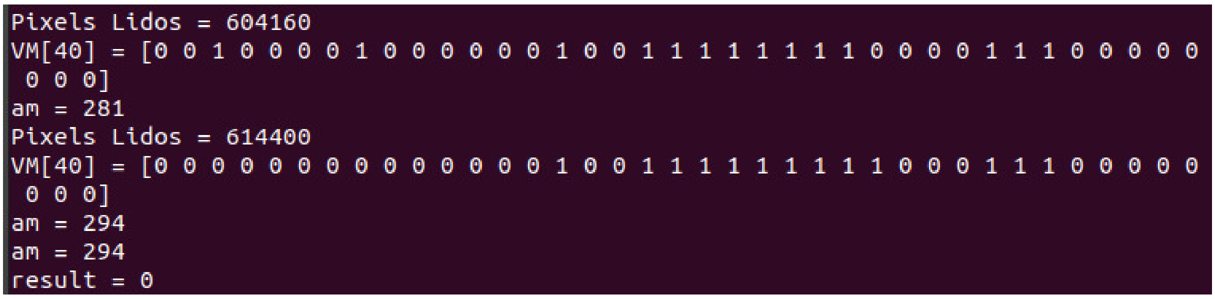
Além disso, as figuras 50, 51 e 52 mostram os resultados da detecção de movimento para o primeiro, segundo e terceiro par de *frames*, que estão de acordo com os resultados



```
leonardo@leonardo-Aspire-A315-54:~/Área de Trabalho/TCC$ python3 motionDetector_FSM.py -pth Teste_Movimento.mp4
FrameCapture:
                path = Teste_Movimento.mp4,
                source_fps = 15,
                resolution = [640, 480],
                real_fps = 6.0
                event_time = 3
MotionDetector:
                threshold = 15,
                chunk_lines = 30,
                chunk_columns = 40
Contador de bytes (b) = 0
Seletora (sel) = 0
Contador de linhas de pixel (lp) = 0
Contador de linhas de chunks (lc) = 0
Contador de linhas do segundo frame (l) = 0
Acumulador de Movimento (am) = 0
Registradores de entrada = reg(40,)
Matriz de Chunks = MC(30, 40)
Vetor de Movimento = VM(40,)
Iniciando a thread de captura e enfileiramento ...
start: Iniciando as operações da FC
Pixels Lidos = 317440
VM[40] = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0]
am = 0
Pixels Lidos = 327680
VM[40] = [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0]
am = 1
```

Figura 49 – Primeiro e segundo ciclo de comparação e limiarização do primeiro par de *frames*

apresentados nas figuras 46, 47, 48. Provando mais uma vez a equivalência entre a FSM escrita em Python (vide seção 3.3) e a arquitetura de hardware em VHDL (vide seção 3.4).



```
Pixels Lidos = 604160
VM[40] = [0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0
 0 0 0]
am = 281
Pixels Lidos = 614400
VM[40] = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0
 0 0 0]
am = 294
am = 294
result = 0
```

Figura 50 – Aos 614.400 *pixels* lidos, com $am = 294$ e $result = 0$

```
Pixels Lidos = 1218560
VM[40] = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0
 0 0 0]
am = 241
Pixels Lidos = 1228800
VM[40] = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0
 0 0 0]
am = 250
am = 250
result = 0
```

Figura 51 – Aos 1.228.800 *pixels* lidos, com $am = 250$ e $result = 0$

```
Pixels Lidos = 1832960
VM[40] = [0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1]
am = 1066
Pixels Lidos = 1843200
VM[40] = [0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
 1 1 1]
am = 1098
am = 1098
result = 1
```

Figura 52 – Aos 1.843.200 *pixels* lidos, com $am = 1098$ e $result = 1$

4.4 Resultados da Implementação em *Hardware*

Em relação a implementação do *block design*, descrito na seção 3.6.1, as figuras 53 e 54 apresentam o consumo de recursos de *hardware* do FPGA, tais como LUTs, Flip-Flops e Buffers.

Resource	Utilization	Available	Utilization %
LUT	6408	53200	12.05
LUTRAM	44	17400	0.25
FF	12091	106400	11.36
BUFG	1	32	3.13

Figura 53 – Tabela com a utilização de recursos de HW do FPGA

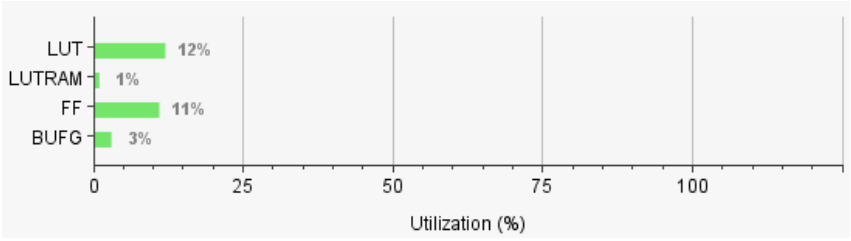


Figura 54 – Gráfico com a porcentagem da utilização de recursos de HW do FPGA

Ainda sobre a implementação, deriva-se o seguinte sumário de temporização do circuito, apresentado pela figura 55.

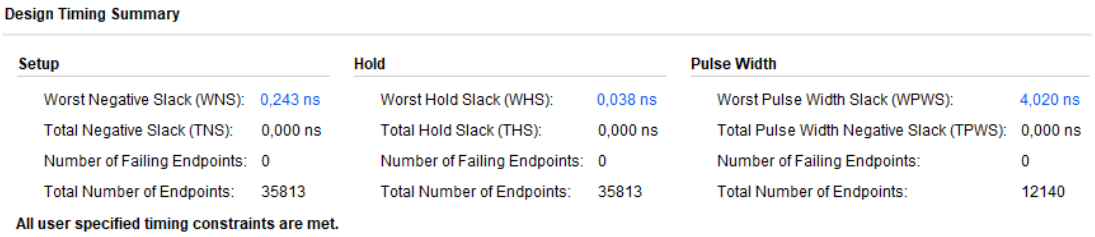


Figura 55 – Sumário de temporização do circuito implementado

A figura 56 apresenta a análise de consumo energético da implementação do projeto destinado a PYNQ-Z2 que demanda um total de 1,414W de potência. Sendo que 10% deste total deriva-se do consumo estático padrão da placa de desenvolvimento e dos 1,279W restante, 96% da potência consumida é demandada pelo PS, ou seja, pelo processador ARM. Neste contexto, a lógica combinacional, os recursos de processamento e armazenamento dos sinais, bem como os *clocks* necessários para o funcionamento do algoritmo embarcado no PL consomem menos de 4% da potência total demandada.

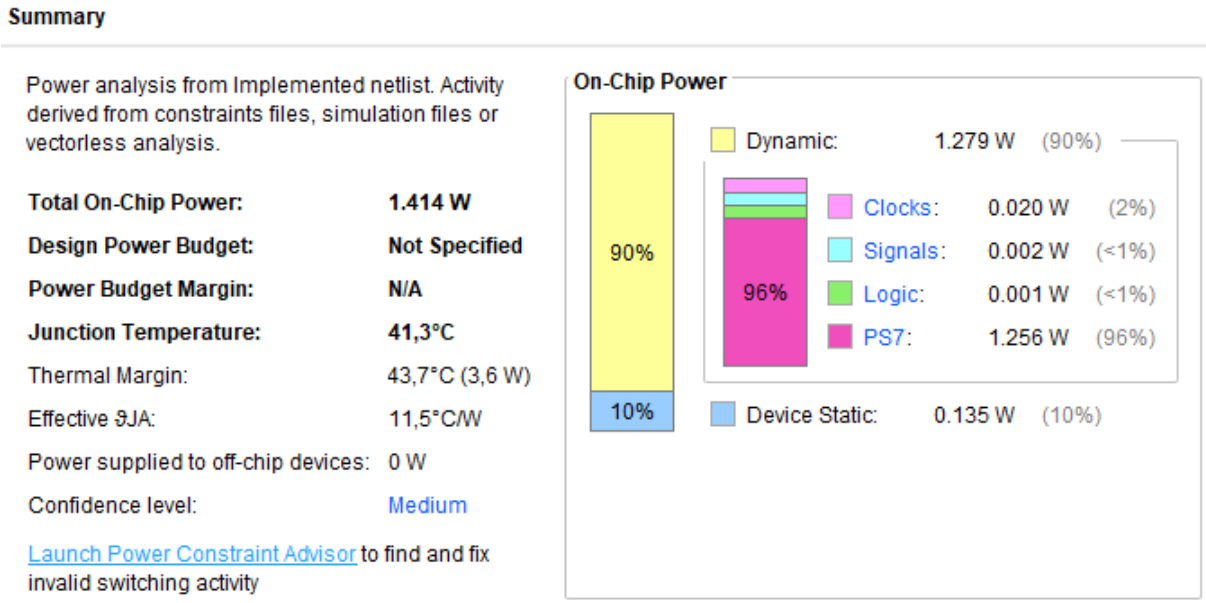


Figura 56 – Sumário de consumo energético do projeto

Nos primeiros testes de integração entre PS e PL, percebeu-se uma dessincronização entre a FSM e o programa do Jupyter. Em que após o primeiro *pixelAvailable* recebido a máquina de estado avançava indefinidamente sobre o ciclo de leitura e acumulo de *pixel*, acumulando o mesmo valor diversas vezes. Acredita-se que esse descompasso entre PS e PL ocorreu devido a diferença de velocidades entre as operações de escrita e

leitura do AXI Lite e as operações da FSM, pois a cada ciclo de relógio, a FSM espera por um novo *pixel* em seu barramento $pixelValue$ para que, no próximo ciclo de relógio e a depender da *flag* $pixelAvailable$, acumule o valor recebido no estado Acumulador (vide seção 3.2.3). Ou seja, as operações de leitura e acumulo de *pixel* ocorrem em 2 ciclos de *clock*. Porém o AXI Lite demora até 4 ciclos de *clock* para cada uma de suas operações. Portanto, sem considerar a leitura das saídas da FSM e também o tempo para ler uma nova linha do arquivo de *pixel*, a aplicação do PS demoraria 8 ciclos de relógio apenas para inserir um novo valor em $pixelValue$ e em seguida levantar a *flag* $pixelAvailable$. Neste contexto, para solucionar o problema, criou-se um novo estado na FSM para aplicar uma primitiva de sincronização, ou também conhecido como *lock* ou *mutex* (*mutual exclusion*).

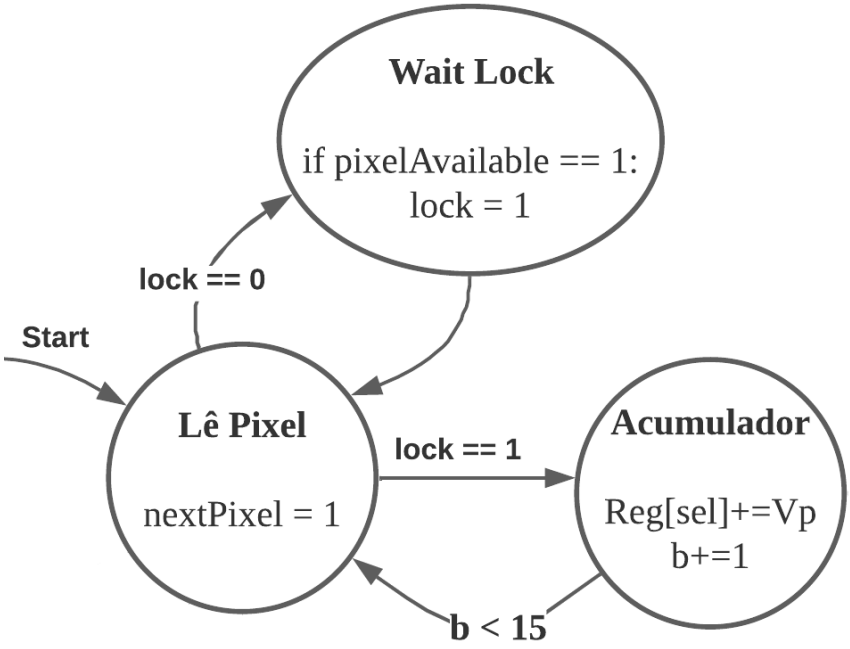


Figura 57 – Estado para sincronização entre PS e PL

A figura 57 apresenta o novo estado Wait Lock que se alterna com o estado Lê Pixel até que o sinal de entrada $pixelAvailable$ garanta que a aplicação do PS tenha inserido um novo byte em $pixelValue$. Desta maneira, o algoritmo perde em performance, pois com o acréscimo de mais um estado em sua lógica a velocidade de processamento das imagens diminui. Contudo, o *lock* controla o avanço indefinido do ciclo de leitura e acumulo de *pixel*, possibilitando o sincronismo entre o ARM e o FPGA.

Todavia, mesmo com a velocidade reduzida o algoritmo proposto se mostrou funcional quando apresentou os mesmos resultados da simulação comportamental, detalhada na seção 4.2, bem como os mesmos resultados do programa de teste em *software*, vide seção 4.3. Neste contexto, as figuras 58 e 59 apresentam resultados semelhantes ao visto nas figuras 44, 45 e 49, em que ao alcançar 317440 *pixels* lidos a FSM subtraiu a primeira linha de *chunks* entre a primeira e a segunda imagem, porém não foi identificado movimento

em nenhuma das regiões, atribuindo valor nulo para o acumulador de movimento *am*. Por outro lado, quando atinge 327680 *pixels* lidos a FSM subtraiu a segunda linha de *chunks*, em que apenas uma região foi identificada com movimento, resultando em $am = 1$.

```
pixel_counter = 307200     PL/PS
                pixelPosition: 15/0
                nextPixel: 1
                pixelValue = 87, reg[0] = 63552, am = 0

pixel_counter = 307200     PL/PS
                pixelPosition: 0/0
                nextPixel: 1
                pixelValue = 87, reg[0] = 0, am = 0

pixel_counter = 317440     PL/PS
                pixelPosition: 0/0
                nextPixel: 1
                pixelValue = 35, reg[0] = 0, am = 0
```

Figura 58 – Aos 317440 *pixels* lidos ocorre a comparação da primeira linha de *chunks* (am = 0)

```
pixel_counter = 317440     PL/PS
                pixelPosition: 0/0
                nextPixel: 1
                pixelValue = 35, reg[0] = 0, am = 0

pixel_counter = 327680     PL/PS
                pixelPosition: 15/0
                nextPixel: 1
                pixelValue = 38, reg[0] = 55232, am = 0

pixel_counter = 327680     PL/PS
                pixelPosition: 0/0
                nextPixel: 1
                pixelValue = 38, reg[0] = 0, am = 1
```

Figura 59 – Aos 327680 *pixels* lidos ocorre a comparação da segunda linha de *chunks* (am = 1)

Além disso, a figura 60 apresenta o momento em que todos os *chunks* dos dois *frames* foram comparados, ou seja, ao atingir 614400 *pixels* lidos. Neste momento, o acumulador de movimento *am* armazena um valor de 294 regiões com movimento, levantando a *flag ready*, mas não indica movimento na *flag result*, assim como apresentado nas figuras 46 e 50. Infelizmente, não é possível observar na figura 60, $am = 294$, pois, devido a baixa velocidade de leitura do AXI Lite, quando o acumulador de movimento é lido pelo PS, a FSM já se encontra no estado Init, onde o valor dos registradores são zerados. Contudo, observa-se aos 604160 *pixels* lidos que $am = 281$, assim como nas figuras 46 e 50, o que mais uma vez comprova o funcionamento do algoritmo proposto quando implementado em *hardware*.

```
pixel_counter = 604160     PL/PS
                pixelPosition: 0/0
                nextPixel: 1
                pixelValue = 80, reg[0] = 0, am = 281

pixel_counter = 614400     PL/PS
                pixelPosition: 0/0
                nextPixel: 0
                pixelValue = 73, reg[0] = 0, am = 0

pixel_counter = 614400     PL/PS
                pixelPosition: 0/0
                pixelValue = 73, reg[0] = 0, am = 0
                ready = 1, result = 0
```

Figura 60 – Aos 604160 *pixels* lidos ocorre a comparação da penúltima linha de *chunks* (am = 281)

Vale mencionar que o vídeo em (FREITAS, 2021b) demonstra o funcionamento do protótipo embarcado na plataforma PYNQ-Z2 e também compara seus resultados com o que foi observado na simulação comportamental através do Vivado.

5 Conclusões

A diferenciação temporal é uma técnica empregada na detecção de movimento em vídeo, em que compara-se os *pixels* de mesma posição entre quadros consecutivos do vídeo, em busca de um resultado binário, ou seja, se houve ou se não houve movimento. Contudo, foi verificado que a análise isolada de cada um dos *pixels* resulta em variações muito discrepantes entre as imagens comparadas, o que engatilha falsos positivos. Para tanto, optou-se por setorizar os quadros capturados em regiões *pixels* com dimensões pre determinadas. Desta forma, o valor médio dos *pixels* vizinhos de cada uma das regiões é calculado e utilizado na diferenciação temporal, o que tornou a detecção de movimento menos sensível aos falsos positivos.

Não obstante, por se tratar de comparação consecutivas a 30 fps, o algoritmo de diferenciação temporal implementado pelo Produtor de *Frames*, micro serviço escrito em Python para a plataforma GigaView, demanda cerca de 12,5% de tempo de processamento e 2% de memória RAM, operando em uma máquina virtual Linux Ubuntu 20.04, com 4 vCPUs de um processador I7 de 10ª geração da Intel e 8Gb de mémória RAM DDR4. Este consumo, inviabiliza a escalabilidade da plataforma para dezenas ou milhares de câmeras necessárias no monitoramento de uma metrópole.

Neste contexto, decidiu-se acelerar o algoritmo de detecção de movimento do Produtor de Frames em *hardware*, utilizando o SoC FPGA modelo PYNQ Z2, com objetivo de facilitar a validação do projeto sobre a ótica do *co-design HW/SW*. Para tanto, otimizações foram implementadas sobre o algoritmo proposto a partir de uma máquina de estados finita escrita em VHDL com o auxilio da plataforma Vivado. Está máquina de estados, descrita no capítulo 3, opera o algoritmo de diferenciação temporal quando embarcado no FPGA da placa de desenvolvimento PYNQ Z2. Por outro lado, o processador ARM da PYNQ opera uma aplicação Python responsável por lidar com as entradas e saídas da entidade detectora de movimento, em busca de validar seu funcionamento. Além disso, para comunicação entre os dois ambientes, *hardware* e *software*, do projeto, um barramento AXI Lite foi implementado.

Os resultados descritos no capítulo 4 mostram que a proposta de implementação eficiente para a detecção de movimento em vídeo detalhada no capítulo 3 é viável tanto em *software* quanto em *hardware*. Além disso, a simulação comportamental realizada via Vivado mostrou que o algoritmo proposto verifica movimento em um par de *frames* consecutivos, com resolução espacial de 640x480p cada, em apenas 12,68 ms, se garantido um novo *pixels* a cada 10 ns. Ou seja, o algoritmo é capaz de processar quase 80 pares de imagens por segundo, o que possibilita operar câmeras a aproximadamente 160 fps,

em contra ponto aos 30 fps das câmeras do GigaView. Além disso, em termos de consumo de recursos a implementação em *hardware* no SoC FPGA ZYNQ-7000 da PYNQ Z2 ocupou apenas 6408 das 53200 *Look-up Tables* disponíveis, apenas 12091 dos 106400 *Flip-Flops* disponíveis e apenas 1 dos 32 *buffers* disponíveis. Ainda da implementação em hardware, a potência total demanda pelo protótipo fica por volta de 1,414W, mas considerando que 96% dessa energia é consumida pelo processador ARM e desconsiderando o consumo estático dos recursos da placa de desenvolvimento, apenas 0,023W de potência são demandados pelo algoritmo de detecção de movimento.

Contudo, o protótipo apresentou lentidão no processamento das imagens devido ao barramento AXI Lite que demora até 4 ciclos de relógio para transferir um *pixel* entre os ambientes de *software* e *hardware* do projeto. De qualquer forma, os resultados da implementação se mostraram idênticos aos resultados derivados do algoritmo puramente em *software* e também ao que se observou da simulação comportamental.

Em termos de trabalhos futuros, planeja-se substituir o AXI Lite pelo AXI Stream ou até mesmo pelo AXI FIFO, possibilitando uma integração mais otimizada em termos de velocidade entre o ARM e o FPGA da PYNQ-Z2.

Referências

ARM. *ARM Company*. 2022. Disponível em: <https://www.arm.com>. Citado na página 50.

COMMUNITY, T. S. *API reference numpy.mean*. 2021. Disponível em: <https://numpy.org/doc/stable/reference/generated/numpy.mean.html>. Citado na página 33.

COTHEREAU, N.; DELAITE, G.; GOURDIN, E. Intelligent ip camera an fpga motion detection implementation. Aalborg University, Department of Computer Science, 2008. Citado 2 vezes nas páginas 28 e 29.

DEVELOPER, A. *About the AXI4-Stream protocol*. 2010. Disponível em: <https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol>. Citado na página 50.

DEVELOPER, A. *AXI protocol overview*. 2022. Disponível em: <https://developer.arm.com/documentation/102202/0200/AXI-protocol-overview>. Citado 2 vezes nas páginas 8 e 50.

FOUNDATION, P. S. *Funções embutidas: abs()*. 2021. Disponível em: <https://docs.python.org/pt-br/3/library/functions.html#absl>. Citado na página 34.

FREITAS, L. B. B. de. *Motion Detector FSM: código em Python*. 2021. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/Python/motionDetector_FSM.py>. Citado 3 vezes nas páginas 46, 53 e 61.

FREITAS, L. B. B. de. *Motion Detector FSM: resultados em vídeo da implementação do algoritmo na PYNQ-Z2*. 2021. Disponível em: <https://youtu.be/zJBpCF_KWUI>. Citado na página 67.

FREITAS, L. B. B. de. *Motion Detector FSM: resultados em vídeo para o TCC1*. 2021. Disponível em: <https://www.youtube.com/watch?v=2-G5Qk-LYPg>. Citado na página 53.

FREITAS, L. B. B. de. *Controlador do Motion Detector: código em Python para JupyterNotebook*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/Python/montio_detector_py.ipynb>. Citado na página 52.

FREITAS, L. B. B. de. *Motion Detector e AXI Lite: código em VHDL*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/VHDL/motion_detector_and_AXI_Lite_1_3_v1_3_S00_AXI.vhd>. Citado na página 50.

FREITAS, L. B. B. de. *Motion Detector FSM: arquivo de pixel*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/AuxFiles/Movendo.txt>. Citado na página 48.

FREITAS, L. B. B. de. *Motion Detector FSM: código do Test Bench em VHDL*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/VHDL/tb_motion_detector.vhd>. Citado na página 48.

FREITAS, L. B. B. de. *Motion Detector FSM: código em VHDL*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/VHDL/fsm.vhd>. Citado na página 47.

FREITAS, L. B. B. de. *Motion Detector FSM: put video in file*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/Python/putVideoinFile.py>. Citado na página 48.

FREITAS, L. B. B. de. *Motion Detector FSM: video de teste*. 2022. Disponível em: <https://github.com/leobbf/MotionDetector/blob/main/AuxFiles/Teste_Movimento.mp4>. Citado 3 vezes nas páginas 47, 48 e 61.

GONZALES, R. C.; WOODS, R. E. *Digital Image Processing*. [S.l.]: Pearson Education, Inc., 2008. Citado 4 vezes nas páginas 7, 18, 19 e 20.

JAGDALE, V. B.; VAIDYA, R. J. High definition surveillance system using motion detection method based on fpga de-ii 70 board. International Journal of Engineering and Advanced Technology (IJEAT), ISSN: 2249 – 8958, Volume-2, Issue-2, December-2012, 2012. Citado na página 29.

JUNIOR, E. I. et al. Fpga-based emd assist block for motion detection in critical environments. IEEE LATIN AMERICA TRANSACTIONS, VOL. 15, NO. 10, OCTOBER 2017, 2017. Citado na página 29.

JUPYTER. *Project Jupyter*. 2022. Disponível em: <https://jupyter.org>. Citado na página 49.

LAZZARETTI, K.; SEHNEM, S.; BENCKE, F. F. Cidades inteligentes: insights e contribuições das pesquisas brasileiras. https://periodicos.pucpr.br/Urbe/article/view/25670, 2019. Citado na página 13.

MISHRA, S. et al. A novel comprehensive method for real time video motion detection surveillance. Amity University, India, 2011. Citado na página 21.

OPENCV. Optical flow. 2013. Disponível em: <https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html>. Citado 2 vezes nas páginas 7 e 22.

OPENCV. cv::videocapture class reference. 2021. Disponível em: <https://docs.opencv.org/3.4/d8/dfe/classcv_1_1VideoCapture.html>. Citado 2 vezes nas páginas 31 e 48.

OPENCV. Image filtering, blur(). 2021. Disponível em: <https://docs.opencv.org/master/d4/d86/group___imgproc___filter.html#ga8c45db9afe636703801b0b2e440fce37>. Citado 2 vezes nas páginas 7 e 32.

OPENCV. Smoothing images. 2021. Disponível em: <https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html>. Citado na página 32.

PROJECT, P. *Overlay Tutorial*. 2018. Disponível em: <https://pynq.readthedocs.io/en/v2.6.1/overlay_design_methodology/overlay_tutorial.html>. Citado na página 52.

PROJECT, P. *PYNQ Overlays*. 2021. Disponível em: <https://pynq.readthedocs.io/en/v2.0/pynq_overlays.html>. Citado na página 49.

SCHLOSSER, J. Development and verification of fast c/c++ models for the star12 micro controller. 2001. Disponível em: <https://www.schlosser.info/wp-content/uploads/diplomathesis/thesis.html>. Citado 2 vezes nas páginas 7 e 26.

SEHAIRI, K.; CHOUIREB, F.; MEUNIER, J. Comparative study of motion detection methods for video surveillance systems. J. Electron. Imaging 26(2), 023025 (2017), doi: 10.1117/1.JEI.26.2.023025, 2017. Citado na página 21.

SINGH, S.; SHEKHAR, C.; VOHRA, A. Fpga-based real-time motion detection for automated video surveillance systems. CSIR—Central Electronics Engineering Research Institute and Electronic Science Department of Kurukshetra University, 2016. Citado 2 vezes nas páginas 29 e 30.

SLEDEVIC, T.; SERACKIS, A.; PLONIS, D. Fpga-based selected object tracking using lbp, hog and motion detection. Department of Electronic Systems, Vilnius Gediminas Technical University, 2018. Citado na página 29.

TUL. Tul pynq-z2 board. 2021. Disponível em: <https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.htmll>. Citado 2 vezes nas páginas 8 e 49.

WEISS, M. C.; BERNARDES, R. C.; CONSONI, F. L. Cidades inteligentes: casos e perspectivas para as cidades brasileiras. http://ric.cps.sp.gov.br/handle/123456789/516, 2017. Citado na página 13.

XILINX. Vivado design suite, user guide. 2016. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug892-vivado-design-flows-overview.pdf>. Citado 2 vezes nas páginas 7 e 28.

XILINX. Ultrascale architecture configurable logic block, user guide. 2017. Disponível em: <https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf>. Citado 2 vezes nas páginas 7 e 24.

XILINX. Understanding fpga architecture. 2018. Disponível em: <https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/odz1504034293215.html>. Citado 3 vezes nas páginas 7, 23 e 24.

XILINX. Pynq: Python productivity. 2021. Disponível em: <http://www.pynq.io/home.html>. Citado na página 49.

XILINX. Zynq-7000 soc product advantages. 2021. Disponível em: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>. Citado 2 vezes nas páginas 7 e 25.

XILINX. Vivado overview. 2022. Disponível em: <https://www.xilinx.com/products/design-tools/vivado.html>. Citado na página 47.