UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
BACHARELADO EM ENGENHARIA ELETRÔNICA

CEZAR BERTELLI

IMPLEMENTAÇÃO DE PONTE HPS-FPGA PARA O PROCESSAMENTO DIGITAL DE IMAGENS EM PLATAFORMA SOC

TRABALHO DE CONCLUSÃO DE CURSO

CAMPO MOURÃO
2020

CEZAR BERTELLI

IMPLEMENTAÇÃO DE PONTE HPS-FPGA PARA O PROCESSAMENTO DIGITAL DE IMAGENS EM PLATAFORMA SOC

Implementation of the HPS-FPGA Bridge for Digital Image processing on the SOC plataform

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Márcio Rodrigues da Cunha

CAMPO MOURÃO

2020

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Campo Mourão
Coordenação de Engenharia Eletrônica

TERMO DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO INTITULADO

IMPLEMENTAÇÃO DE PONTE HPS-FPGA PARA O PROCESSAMENTO DIGITAL DE IMAGENS EM PLATAFORMA SOC

DO(A) DISCENTE

CEZAR BERTELLI

Trabalho de Conclusão de Curso apresentado no dia **04 de dezembro de 2020** ao Curso Superior de Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná, Campus Campo Mourão. O(A) discente foi arguido(a) pela Comissão Examinadora composta pelos professores abaixo assinados. Após deliberação, a comissão considerou o trabalho **aprovado com alterações**.

Prof. Dr. Eduardo Giometti Bertogna
Avaliador(a) 1
UTFPR

Prof. Me. Lucas Ricken Garcia
Avaliador(a) 2
UTFPR

Prof. Dr. Marcio Rodrigues da Cunha
Orientador(a)
UTFPR

RESUMO

Com o crescente aumento do mercado relacionado às tecnologias autônomas, a área de processamento de imagens destaca-se como uma ferramenta no fomento dessas tecnologias, trazendo opções de automação em tempo real e análises de ambiente, sendo, portanto, um dos catalizadores da evolução de diversas aplicações nessa área. O projeto em questão, tem como finalidade demonstrar a implementação do barramento de comunicação Lightweight HPS-FPGA na placa de desenvolvimento DE10Nano e, a partir do mesmo, criar um processamendo de imagens simplificado com os dados recebidos e exibi-los pela saída HDMI.

Palavra-chave: De10Nano. Processamento de Imagens. Lightweight HPS-FPGA. HDMI.

ABSTRACT

With the growth of the market related to autonomous technologies, an area of image processing stands out as a tool for promoting technologies, bringing options for real-time automation and environmental analysis, thus being one of the catalysts for the evolution of several applications in this area . field. area. The project in question has to demonstrate the implementation of the Lightweight HPS-FPGA communication bus and, from there, create a simplified image processing with the received data and display it via HDMI output.

Keywords: De10Nano. Image Processing. Lightweight HPS-FPGA. HDMI.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

ASICs	*Circuitos Integrados de aplicação específica*
CPLDs	*Dispositivo Lógico Programável Complexo*
HPS	*Sistema de Processador Rígido*
FPGA	*Arranjo de Portas programáveis em campo*
IBM	*Máquina de negócios internacionais*
PnP	*Conectar e Funcionar*
CISC	*Conjunto de instruções complexos*
RISC	*Conjunto de instruções reduzidos*
RGB	*Vermelho Verde Azul*
SBT	*Ferramenta de construção de software*
SD	*Segurança Digital*
SPI	*Interface Periférica Serial*
SPLDs	*Dispositivo Lógico Programável Simples*
TTL	Lógica *Transistor-Transistor*
HDMI	*Interface Multimídia de Alta Definição*

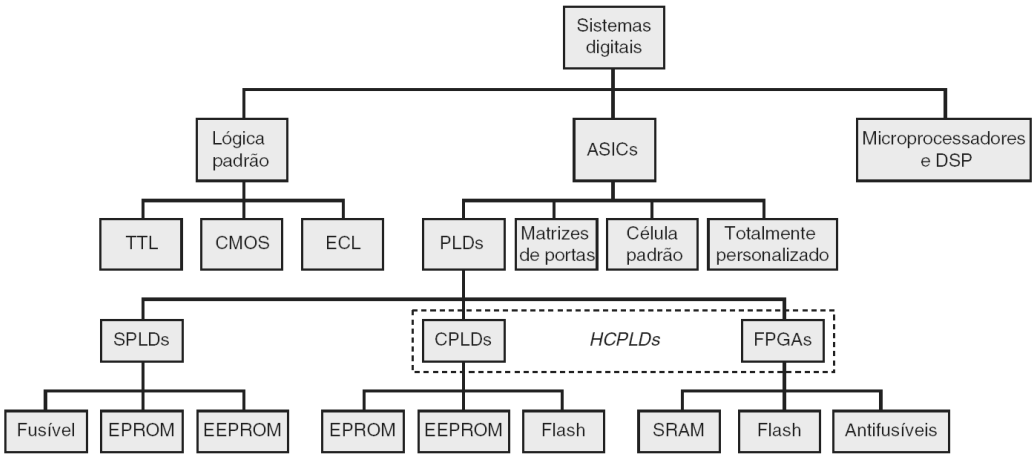
SUMÁRIO

1 INTRODUÇÃO

Desde os primórdios da eletrônica, sempre houve uma divisão bem definida entre duas grandes áreas, a digital e a analógica, diferenciando-se basicamente pelo modo de representação das informações, que em analógica são representadas como uma faixa contínua de dados e em digital esses dados são discretos.

A Eletrônica Digital refere-se a circuitos ou sistemas que processam informações binárias, ou seja, suas informações são representadas teoricamente como “0” (nível lógico baixo) e “1” (nível lógico alto).

Figura 1 - Diferentes caminhos para lógicas digitais.



Fonte: TOCCI et al (2011).

Uma análise real desse tipo de sistema tem sua dependência em cima do potencial de trabalho. Por exemplo, no caso da família TTL, o seu tratamento é feito de forma que seu nível lógico alto é 5 V e o nível lógico baixo é 0 V (TOCCI *et al*, 2011). No entanto, vale ressaltar que os dispositivos lógicos não respondem somente para 0 e 5 V. A resposta é baseada em faixas de potenciais, que para nível lógico baixo vai de 0 a 0,8 V, enquanto para nível lógico alto vai de 3,7 a 5 V (MORAIS, 2005).

Na Figura 1 é possível visualizar a família de Sistemas Digitais, e consequentemente, seus subgrupos. A Eletrônica Digital, possui três grandes ramificações, a lógica padrão, onde se encontram a maioria dos circuitos integrados utilizados em pequenos projetos; os ASICs (Application Specific Integrated Circuits), um conjunto de circuitos integrados que englobam desde circuitos totalmente personalizados de fábrica até circuitos semidedicados; e os microprocessadores que são mais utilizados em grandes projetos (TOCCI *et al*, 2011).

Dentro dos ASICs, duas diferentes formas de implementação de projetos se destacam entre as demais, sendo as lógicas reconfiguráveis e as específicas.

As lógicas específicas têm como foco principal a criação de um circuito dedicado. Dessa forma, quando projetado, suas especificações estarão voltadas diretamente à uma aplicação, conseguindo assim chegar a níveis altos de desempenho com menos recursos. Em contrapartida, projetos com essa implementação são custosos, sendo viável somente para grande escala de fabricação (CHAGAS, 2017).

As lógicas reconfiguráveis são as plataformas de propósito geral, ou seja, diferentemente da lógica específica, esses circuitos são fabricados de modo que, não se tem um problema concreto para se resolver, sua aplicação dependerá das funcionalidades implementadas, assim, um mesmo circuito pode ser utilizado para diversos propósitos (CHAGAS, 2017).

Adentrando nos conceitos de lógicas reconfiguráveis, duas de suas características primordiais são que, o seu hardware pode sofrer modificações constantemente, ou seja, conforme necessário, a estruturação do hardware poderá sofrer mudanças via descrição de hardware, não tendo que alterar o circuito de forma física. A segunda característica é que, como o circuito é efetuado diretamente no hardware, aplicações efetuadas no mesmo conseguem ter uma velocidade próxima de circuitos específicos (BONDALAPATI, K.; PRASANA, V.K, 2002).

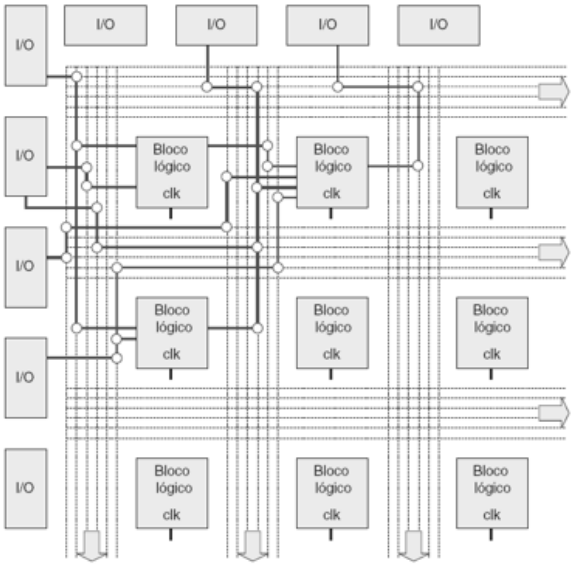
Em função desses fatos supracitados, entende-se que a utilização de lógica reconfigurável é um importante caminho para se realizar aplicações que exigem versatilidade, modularidade e arquiteturas não-convencionais.

Neste contexto, existem algumas categorias de circuitos lógicos reconfiguráveis, sendo as principais os SPLDs (*Simple Programmable Logic Device*), os CPLDs (*Complex Programmable Logic Device*) e os FPGAs (*Field Programmable Gate Array*) e são caracterizados por sua capacidade de sintetizar circuitos lógicos (TOCCI *at al*, 2011).

Em particular, o FPGA, introduzido em 1985 pela empresa *Xilinx Inc*., é um dispositivo de lógica reconfigurável que possui um suporte relativamente satisfatório para aplicações robustas. Sua composição contem um grande arranjo de portas reconfiguráveis, que podem ser transformadas em quaisquer tipos de funções lógicas a partir de uma linguagem de descrição de *hardware (National Instruments*, 2017).

Essas funções lógicas são criadas por interconexões de centenas de milhares de portas pré-fabricadas. Assim, é possível inserir uma máscara em seu contorno e comunicar somente as funções necessárias. A Figura 2 demonstra um exemplo de como essa máscara é inserida, pois o mesmo, quando gerado em fábrica possui todas as trilhas mas os pontos a serem conectados ficam a critério do programador. Quando gerada essas ligações, seu funcionamento se fundamenta em ligações normais de circuitos (*NATIONAL INSTRUMENTS*, 2017).

Figura 2 - Esquemático FPGA.



Fonte: TOCCI *at al* (2011).

Existem várias versões dessa plataforma, diferenciando-se em tamanho, aplicabilidade, suporte para comunicações externas (VGA, USB, GPIO) e o aprimoramento da ferramenta, disponibilizando cada vez mais recursos para aplicações em geral.

Uma área de pesquisa bastante explorada a partir de FPGAs é o processamento de imagem. Nesta área específica, por definição geral, podem ser considerados quaisquer tipos de processamentos de dados visto que sua entrada ou saída é definida como algum tipo de imagem, tais como figuras, fotografias ou até mesmo vídeos, ao qual particionado, é considerado uma sequência de quadros (MARQUES FILHO; VIEIRA NETO, 1999).

A área de processamento de imagens vem sendo objeto de crescente interesse por permitir viabilizar grande número de aplicações em duas categorias bem distintas: (1) o aprimoramento de informações pictóricas para interpretação humana; e (2) a

análise automática por computador de informações extraídas de uma cena (MARQUES FILHO; VIEIRA NETO, 1999).

Dentro dessas categorias, algumas aplicações interessantes são a análise de imagens antigas, aplicativos para correção de imagens, automação em linha de produção para análise de erros, inteligência artificial, análise de ambiente para veículos não tripulados, entre outros.

Para o trabalho em questão serão utilizadas as características gerais de processamento de imagens, mas, seu principal foco será na facilitação dos formatos existentes, gerando assim uma infraestrutura completa para determinados formatos de entrada de imagens.

1.1 OBJETIVO GERAL

O objetivo deste trabalho é a implementação de uma infraestrutura ao qual, fazendo o uso da placa DE10Nano desenvolvida pela Intel e utilizando um barramento já existente, realizará a troca de dados entre os dois chips principais (HPS e FPGA) disponibilizados na placa em questão. Os dados compartilhados serão processados com base na arquitetura reconfigurável presentes na FPGA, de forma a utilizar o paralelismo e o seu alto desempenho. Os dados após processados serão exibidos de forma visual em uma tela com tecnologia HDMI.

1.2 OBJETIVOS ESPECÍFICOS

Para melhorar o entendimento do objetivo geral deste trabalho, é possível fazer uma divisão em quatro metas específicas:

1. **Análise de periféricos e protocolos de comunicação**: realizar um levantamento de dados e informações sobre os periféricos e seus protocolos de comunicação, relacionados com a partição ARM e FPGA da placa de desenvolvimento e a saída HDMI (*High Definition Multimedia Interface*).
2. **Implementação da comunicação de dados**: implementar os protocolos de informação para fazer o recebimento de dados. Esse protocolo se resume principalmente na comunicação interna entre o setor ARM e o setor FPGA.

3. **Processamento da Imagem**: executar o processamento da imagem recebida, modificando os pixels recebidos de forma simples para exemplificar o escopo geral do trabalho.
4. **Disponibilização da imagem**: implementar os protocolos de comunicação entre a placa de desenvolvimento e um monitor com interface HDMI.

1.3 JUSTIFICATIVA

Com o crescente aumento do mercado relacionado às tecnologias autônomas, a área de processamento de imagens destaca-se como uma ferramenta no fomento dessas tecnologias, trazendo opções de automação em tempo real e análises de ambiente, sendo, portanto, um dos catalizadores da evolução de diversas aplicações nessa área.

O trabalho em questão, busca gerar a infraestrutura para que, em outras oportunidades, facilite o processo de utilização da placa. Contribuindo para que em um tempo menor, possa ser realizado melhoramentos tanto no processamento de imagens como também na utilização de outras pontes de comunicação disponíveis.

O processamento de imagens em si, pode ser realizado praticamente por qualquer tipo de sistema de controle, ou seja, quando se tem uma sequência de dados e existe a necessidade de efetuar alguma modificação, até microcontroladores simples conseguem efetuar esse tipo de alteração.

A necessidade do uso de FPGAs para realizar o processamento de imagens se da não pela execução em si, mas sim pela carência de uma resposta cada vez mais rápida nas aplicações em questão.

Com base nas características principais das FPGAs, seu processamento de forma paralelo é capaz de efetuar diversos tipos de procedimentos ao mesmo tempo, resultando em tempos de execução tão baixos que possibilitam aplicações que necessitam de resultados em tempo real, fatos que são inalcalcançaveis por microcontroladores de uso geral.

2 FUNDAMENTAÇÃO TEÓRICA

O presente trabalho possui três grandes etapas, a comunicação inicial entre FPGA e o ARM, o processamento de imagem e a disponibilização da informação via HDMI. Desta forma, nas seções seguintes será apresentada a fundamentação teórica relacionada com este projeto.

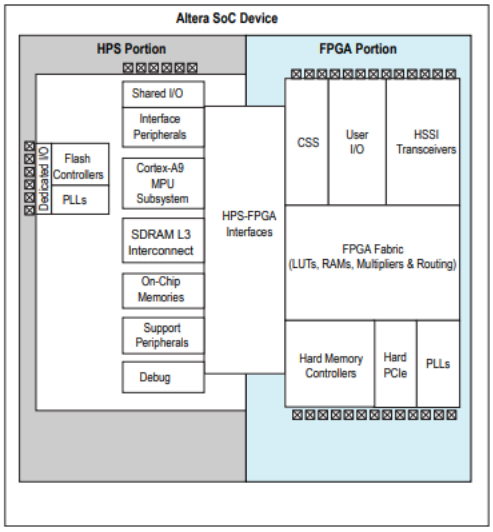
2.1 PLATAFORMA DE DESENVOLVIMENTO

Na concepção deste projeto propôs-se explorar as potencialidades da placa de desenvolvimento DE10-Nano, mediante a análise das necessidades deste trabalho, aliada a alta eficiência desta placa e sua disponibilidade na universidade.

2.2 DE10-NANO

A DE10-Nano é uma plataforma de desenvolvimento de hardware robusta e completa, em seu escopo principal possui dois núcleos com um alto grau de importância, o Cortex-A9 dual core e o Ciclone 5. A junção destes dois núcleos permite ao programador a realização de projetos utilizando o poder da reconfigurabilidade emparelhada com um sistema de processador de alto desempenho e baixo consumo de energia (TERASIC, 2020).

Figura 3 Diagrama de Blocos DE10Nano.



Fonte: INTEL (2014).

Como já mencionado e também mostrado na Figura 3, a placa de desenvolvimento DE10Nano, é dividida em duas partes gerais, sendo elas o lado HPS (Sistema de Processador Rígido) e FPGA (Arranjo de Portas programáveis em campo).

O lado HPS pode ser definido como a integração de todos os módulos que consiste o sistema do processador, unindo desde o ARM, as memórias, interfaces periféricas entre outros. Quando todas as partes se interagem é possível instanciar um Sistema operacional ao qual gerencia todas as aplicações e comunicações referente ao lado não reconfigurável.

O lado FPGA, similar ao lado HPS já comentado, também pode ser definido pela união de todos os módulos referentes ao funcionamento geral da tecnologia FPGA, reunindo assim os controles de memória, entradas e saídas, entre outros.

A união entre as duas partes, no geral, pode ser efetuada de três formas diferentes, sendo uma ponte HPS, ao qual o lado HPS tem grande acesso as informações referentes ao lado FPGA; uma ponte FPGA, ao qual a mesma tem grande acesso as informações do lado HPS; e por fim a ponte *Lightweight HPS-FPGA* que irá ser comentada nos próximos tópicos, mas no geral seu funcionamento é baseado em compartilhamento de memória entre os dois lados mencionados. As especificações desse SOC são apresentadas a ssguir:

1. Especificações FPGA:
    - *Intel Cyclone® V SE 5CSEBA6U23I7NDK device (110K LEs)*
    - *Serial configuration device – EPCS64 (revision B2 or later)*
    - *USB-Blaster II onboard for programming; JTAG Mode*
    - *HDMI TX, compatible with DVI 1.0 and HDCP v1.4*
    - *2 push-buttons*
    - *slide switches*
    - *8 green user LEDs*
    - *Three 50MHz clock sources from the clock generator*
    - *Two 40-pin expansion headers*
    - *One Arduino expansion header (Uno R3 compatibility), can be connected with Arduino Shields*
    - *One 10-pin Analog input expansion header (shared with Arduino Analog input)*

- *A/D converter, 4-pin SPI interface with FPGA*

2. Especificações ARM (HPS):
    - *800MHz Dual-core ARM Cortex-A9 processor*
    - *1GB DDR3 SDRAM (32-bit data bus)*
    - *1 Gigabit Ethernet PHY with RJ45 connector*
    - *USB OTG Port, USB Micro-AB conector*
    - *Micro SD card socket*
    - *Accelerometer (I2C interface + interrupt)*
    - *UART to USB, USB Mini-B conector*
    - *Warm reset button and cold reset button*
    - *One user button and one user LED*
    - *LTC 2x7 expansion header*

2.3 ARQUITETURA ARM

A descrição de *hardware* com FPGAs, muitas vezes, se deparam com implementações fora dos padrões especificados e trabalhados normalmente. Assim para suportar essas novas implementações foi criado, com as próprias expecificações da plataforma, processadores para trabalhar em conjunto com a implementação reconfigurável.

A arquitetura ARM, hoje amplamente conhecida e utilizada, teve seus primeiros passos nas mãos da empresa britânica Acorn Computers Ltd no início da década de 80, tal feito se deu pela falta de microprocessadores disponíveis na época quando os mesmos procuravam um substituto para o antigo 6502 dos BCC Micros (CAROL ATACK e ALEX VAN SOMEREN, 1993).

A evolução e aceitação da arquitetura chamou atenção de diversas empresas como Apple e VLSI Technology que no qual auxiliaram no crescimento e expansão da tecnologia. Atualmente a arquitetura ARM se encontra em um alto patamar e já é responsável por 95% do *Market share* de *smartphones* e corresponde a mais de 75% dos microprocessadores embarcados de 32 bits (TIMOTHY PRICKETT MORGAN, 2011).

Auxiliando na difusão da tecnologia, está em um dos principais motivos de sua grande expansão é a sua ênfase energética quando, diferente dos modelos CISC

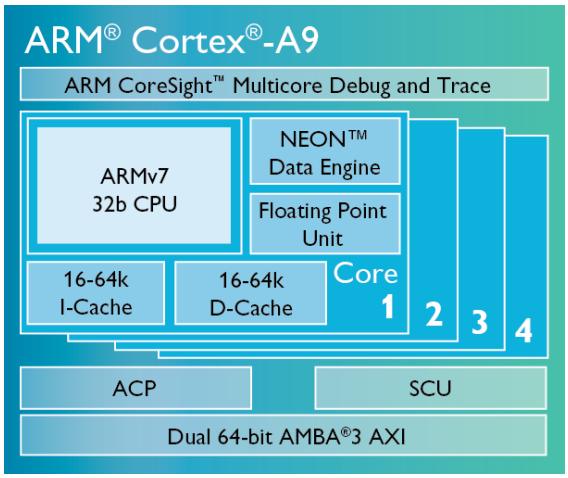
(complex instructions set computer), utiliza um conjunto reduzido de instruções RISC (reduced Instructions set computer), diminuindo assim o número de transistores e facilitando desenvolvimento e testes (ARM LIMITED, 2011)

Quando um processador é classificado como RISC, ideologicamente significa que um conjunto simples de instruções resultará em uma unidade de controle simples, barata e rápida, além de gerar o benefício da previsibilidade (VASCONCELOS DE BRITO, 2014).

Para ser mantida a formatação RISC, em comparação a outras arquiteturas, faz-se necessário delimitar os tipos de dados, diminuir os modos de endereçamento e reduzir o número de estágios de *pipeline*. Assim, como as instruções variam pouco, a unidade de controle pode prever quantos ciclos de tempo serão necessários, sabendo então exatamente quando começar uma nova tarefa (VASCONCELOS DE BRITO, 2014).

O modo como é disponibilizado este tipo de arquitetura, também possui grande aceitação no mercado, pois o mesmo deixa livre para que outros fabricantes parceiros usem a tecnologia e implementem periféricos da melhor forma no contexto de sua aplicação. Assim, de forma geral, é disponibilizado somente o projeto do *chip* ARM, deixando que os periféricos em geral como memória, gpu, rede de internet entre outras seja que total escolha da empresa requisitante.

Figura 4 Arquitetura do processador Cortex-A9.



Fonte: ARM Limited (2020).

O processador utilizado na placa de desenvolvimento escolhida que está representado na Figura 4, é um Cortex-A9, uma macrocélula ARM de alto desempenho e baixo consumo de energia com um subsistema de cache L1 que oferece recursos completos de memória virtual. O processador Cortex-A9 implementa a arquitetura ARMv7-A e executa instruções ARM de 32 bits (ARM LIMITED, 2020).

2.4 PROTOCOLOS DE COMUNICAÇÃO

A comunicação ARM/FPGA é um dos principais propósitos deste projeto. Essa comunicação é realizada por pontes. Neste trabalho será explorada a potencialidade da ponte de menor complexidade, *Lightweight HPS-to-FPGA Bridge*, aplicando ao processamento de imagem.

2.4.1 *Lightweight HPS-to-FPGA Bridge*

A comunicação ARM/FPGA utilizada é a mais simples entre as disponíveis no kit de desenvolvimento, fornecendo somente 2Mb de espaço de endereço e acesso à lógica, periféricos e memória implementados na FPGA. Normalmente essa ponte é utilizada para trocas de dados entre os dois lados, como para acessar registros e acessos a memoria direta (ALTERA, 2018).

Sua implementação menos complexa em relação as outras comunicações citadas, gera limitações para o sistema, uma de suas características é sua largura de barramento que consiste em somente 32 bits que se torna pouco quando comparada com as outras pontes que chegam a 128 bits. (ALTERA, 2018).

O funcionamento dessa comunicação é fundamentada nas definições de compartilhamento de memória, local no qual os dois lados da ponte de dados tem acesso direto, assim quando um dos lados insere qualquer informação nos bits citados, o outro lado consegue diretamente fazer o acesso, extrair os dados, processar e os devolver da mesma forma (ALTERA, 2018).

2.4.2 HDMI

A comunicação HDMI (*High-Definition Multimedia Interface*) é uma interface condutiva de áudio e vídeo digital, diferenciando-se de todas as outras comunicações onde a base geralmente são dados analógicos. Por ter essa referência totalmente digital, consegue-se fazer uma transferência de dados sem a necessidade de comprimir as informações.

O fato de não ser usado a compressão de dados, tem total relação com os padrões de tecnologias utilizadas, ou seja, diferente dos dados analógicos que possuem a necessidade de passarem por conversores AD/DA para serem processados e exibidos, essa tecnologia é processada diretamente, ganhando em velocidade e também não perdendo resolução como no caso dos conversores, conseguindo incluir padrões de 480i/p até 4k.

Como as informações de áudio e vídeo são transmitidas no mesmo barramento, faz-se necessário uma codificação diferenciada, abordando até 8 canais codificados em TMDS (Sinalização diferencial minimizada por transição), de acordo com a norma do Controle de Eletrônicos de Consumo para transmissão digital não comprimida. (HDMI-CEC, 2009)

Com a quantidade de aparelhos utilizando HDMI, criou-se uma necessidade interessante de evolução neste tipo de comunicação, assim existem diversas versões que foram implementadas até hoje e algumas de suas diferenças mais importantes são o aumento da resolução máxima que chegou a 10k a 120Hz, HDR *(*Dinâmica de Alto Alcance) dinâmico para especificar dados quadro-a-quadro entre outros.

A placa DE10Nano que foi escolhida já possui saída HDMI, assim toda as informações geradas como saída antes, durante e o pós processamento de imagens serão informado dessa forma.

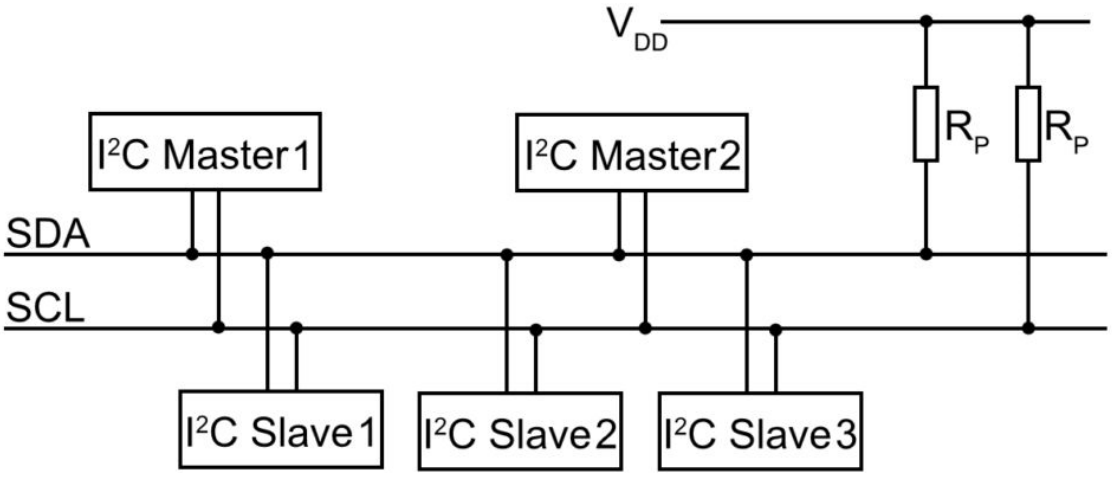
2.4.3 I²C

I²C (Circuito-Inter-Integrado) é um barramento multimestre desenvolvido pela Philips ao qual são utilizados para fazer comunicação de baixa velocidade entre componentes próximos.

Seu funcionamento mostrado na Figura 5, basea-se em duas linhas bidirecionais ao qual são, a linha de dados (SDA) e a linha de *clock (SCL),* ambas resistores de *pull-up.*

O funcionamento real do sistema, parte de um mestre que, informa o clock pelo barramento responsável e emite os dados pelo outro barramento, os escravos do mesmo recebem as informações e respondem quando forem solicitados respeitando o clock original (STACKOVERFLOW, 2018).

Figura 5 Funcionamento I²C.



Fonte: STACKOVERFLOW (2018).

2.5 PROCESSAMENTO DE IMAGENS

Processamento de Imagens é uma área bastante interessante e procurada atualmente, pois a sua utilização viabiliza um grande número de aplicações. Os primórdios desse tipo de processamento iniciaram no começo do século XX, buscando aprimorar a comunicação do sistema Bartlane, que fazia a comunicação entre Nova Iorque e Londres através de cabos marítimos (MARQUES FILHO; VIEIRA NETO, 1999).

No entanto, os principais avanços na área, se dão em meados do mesmo século, época em que houve a explosão das tecnologias em relação aos programas espaciais e também com o surgimento dos computadores de grande porte (MARQUES FILHO; VIEIRA NETO, 1999).

Nos dias de hoje, o processamento de imagens está presente em praticamente todos os ramos de atividades humanos, sendo de grande ajuda para diversos tipos de atividades domésticas, comerciais e industriais. Os exemplos são inúmeros, tais como os avanços feitos na medicina, que foram de excelente auxílio para a criação de novos equipamentos médicos.

O Processamento de Imagens pode ser definido como sendo quaisquer manipulações computacionais ao qual são efetuadas técnicas de análise, melhoramento ou aprimoramento de dados pictóricos, sendo que sua utilização está crescendo cada vez mais nos campos da eletrônica e ciências em geral (MARQUES FILHO; VIEIRA NETO, 1999).

Em seu contexto geral, costuma ser uma atividade bastante complexa, pois a mesma necessita de uma ordem cronológica para ser efetuada, ou seja, caso o estudo em cima de uma imagem não seguir a sequência de passos da Figura 6, o resultado esperado pode não ser alcançado (RANGEL DE QUEIROZ; MARTINS GOMES, 2001).

Como já comentado, as etapas presentes na Figura 6 são de extrema importância para o processamento de imagens, mas em contra partida, nem todos os seus processos deverão ser executados, ou seja, a partir do foco da aplicação que deseja realizar é escolhido as etapas a serem seguidas.

Para se iniciar o processo, necessita-se da captura dos dados pictóricos que serão trabalhados, utilizando sensores ou câmeras. Após a obtenção das informações, esses dados devem ser tratados de forma entendível para o processo computacional, inciando-se assim a etapa de pré-processamento (RANGEL DE QUEIROZ; MARTINS GOMES, 2001).

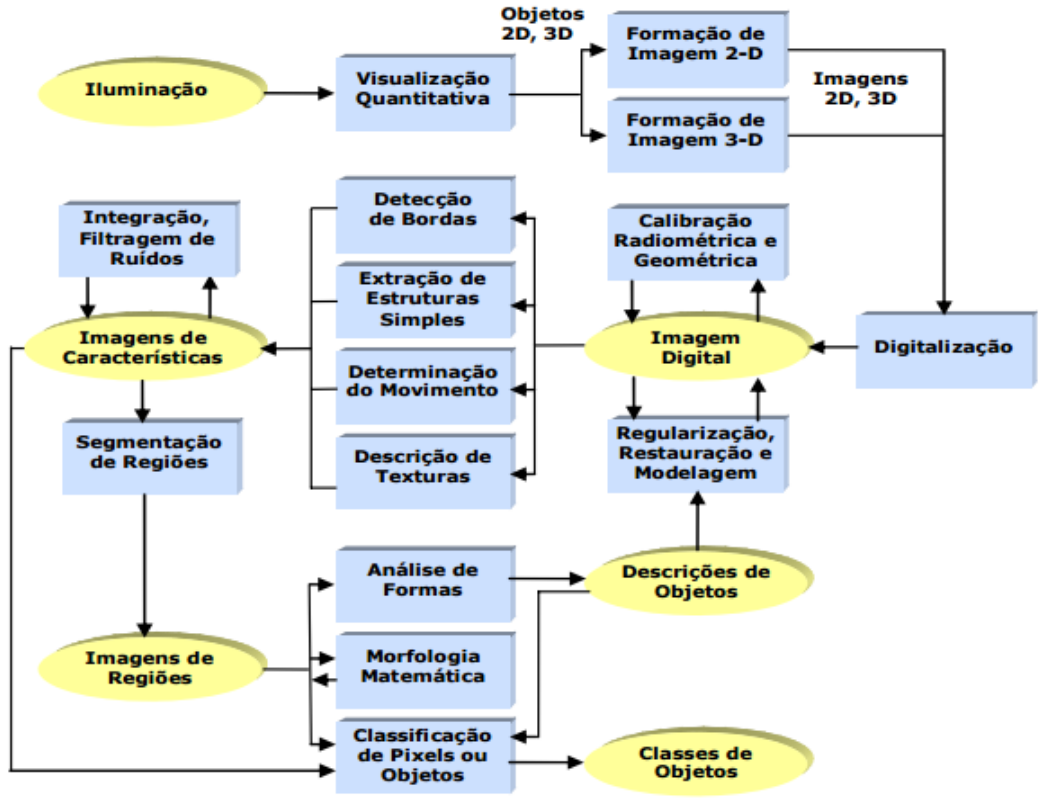
Esta etapa é bastante importante para a solução final, pois a mesma faz o preparo dos dados, de forma a ser executada uma filtragem de ruídos e a correção de distorções geométricas, trazendo consigo grandes melhorias para a imagem e consequentemente auxiliando a execução das outras tarefas do processamento (RANGEL DE QUEIROZ; MARTINS GOMES, 2001).

Em diversas etapas do processamento, se faz necessário o armazenamento das imagens para utilização posterior. Essas etapas, em muitos casos, são bastante complexas, pois as mesmas dependem de uma grande quantidade de bytes para acomodar as informações.

Existem em uma compilação geral, dois tipos de armazenamento, de curta duração, quando as imagens são salvas entre um processo, e o armazenamento de massa, utilizado como backup.

Quando realizado o processamento de imagens, um de seus focos é a identificação de objetos. Tal fato requer atributos avançados para o processo, então, para diminuir a dificuldade do mesmo, retira-se inicialmente as características e atributos da imagem, tais como bordas, textura e vizinhanças (RANGEL DE QUEIROZ; MARTINS GOMES, 2001).

Figura 6 Estrutura geral do processamento de imagens.



Fonte: RANGEL DE QUEIROZ; MARTINS GOMES (2001).

Continuando o processo, necessita-se também diferenciar o objeto em si para com o plano de fundo, para tal é utilizado processos de segmentação, características constantes e descontinuidade. O grau de dificuldade desta etapa é relativo, pois a mesma dependerá exclusivamente dos traços da imagem, ou seja, quanto mais o objeto a ser identificado se diferenciar do plano de fundo, mais fácil será a análise.

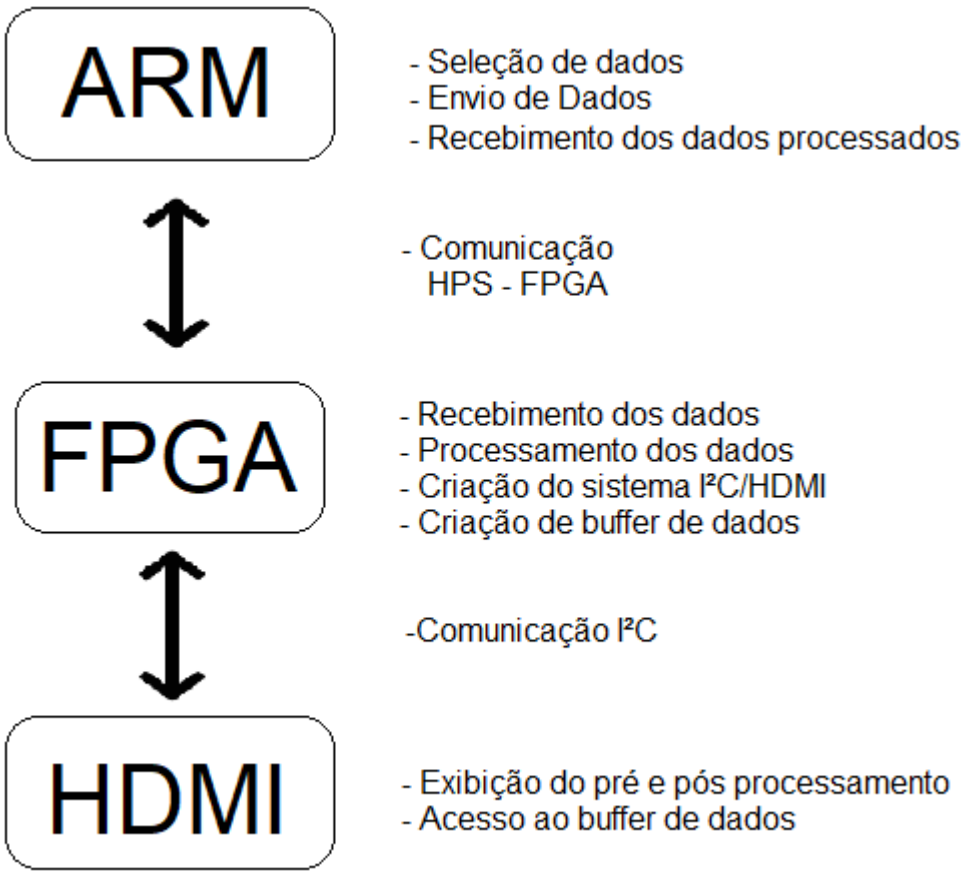
Partindo de que o objeto já foi diferenciado, passa-se ao trabalho de identificar e classificar o mesmo, ou seja, reconhecer, verificar ou inferir uma identidade ao objeto em função das características retiradas anteriormente.

Com a imagem processada, resta somente disponibilizar a mesma, para isso, utiliza-se os protocolos já vistos no decorrer deste trabalho, precisamente quando mencionado as características HDMI.

3 METODOLOGIA

Como o escopo do trabalho é implementar uma infraestrutura para o processamento de imagens utilizando FPGA, tem-se como principais objetivos fazer a integração entre ARM, FPGA e monitor HDMI e assim se basear no fluxograma presente na Figura 7.

Figura 7 Infraestrutura do projeto



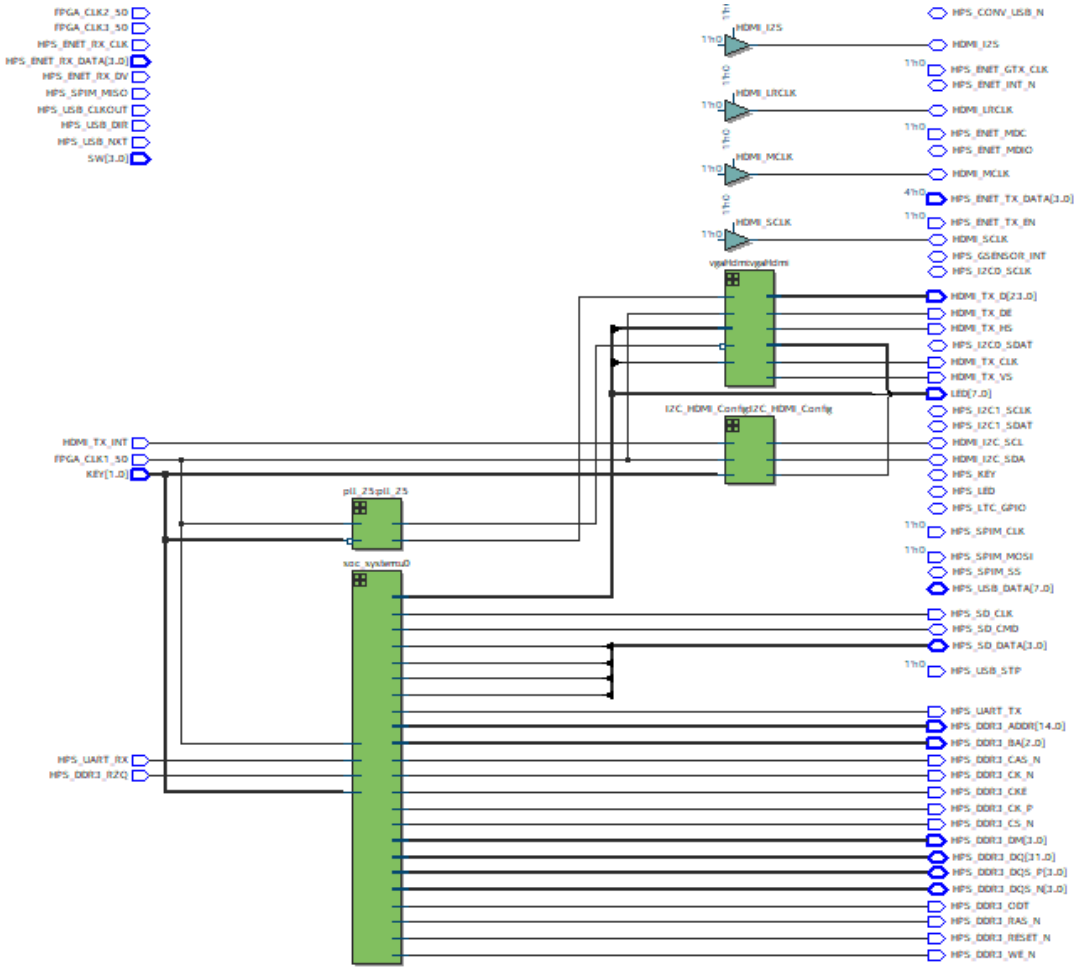
Fonte: Autoria própria (2020).

Primeiramente, utilizando arquivos gerais disponibilizados pela Altera/Intel, conseguiu-se dar os primeiros passos para iniciar as atividades deste projeto. Os arquivos utilizados, assim como as imagens do cartão SD utilizados estão disponíveis em: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=1046&PartNo=4

3.1 PROJETO QUARTUS

O processo de construção de todas as etapas no Quartus segue uma ordem cronológica, ao qual se inicia delimitando as necessidades principais das atividades do mesmo, a seguir passa a execução e por fim os testes de funcionamento. Com todas essas etapas realizadas, consegue-se chegar ao escopo geral presente na Figura 8.

Figura 8 Esquemático Quartus



Fonte: Autoria própria (2020).

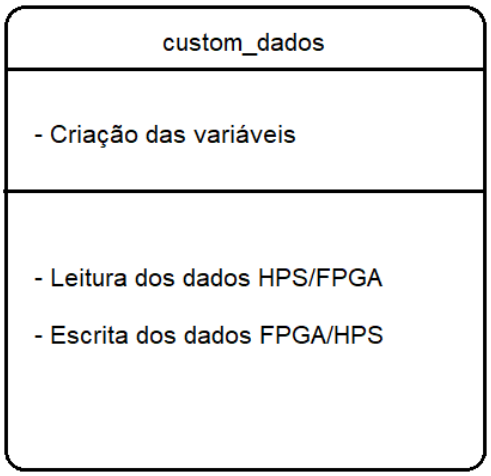
3.1.1 QSYS

O QSYS é um software de simples utilização que permite ao usuário construir desde pequenos sistemas até os mais complexos. Em seu escopo, estão presentes diversos módulos prontos e também possibilita a criação de módulos extras. Tais módulos auxiliam na integração de controladores de memória, processadores entre

outros, todos voltados a facilitação do projeto geral criado na FPGA e até mesmo para fazer a união com os módulos da parte HPS.

Com base em documentos disponibilizados pela Intel, verificou-se a necessidade da criação de um módulo principal para fazer a conexão entre as duas grandes partes desse projeto, o ARM e a FPGA, tal módulo tem a função de definir as rotas de comunicação, fixar os tempos de clock das mensagens a serem usados e o mesmo foi definido com o nome de “custom_dados” como na Figura 9.

Figura 9 Custom_dados.



Fonte: Autoria própria (2020).

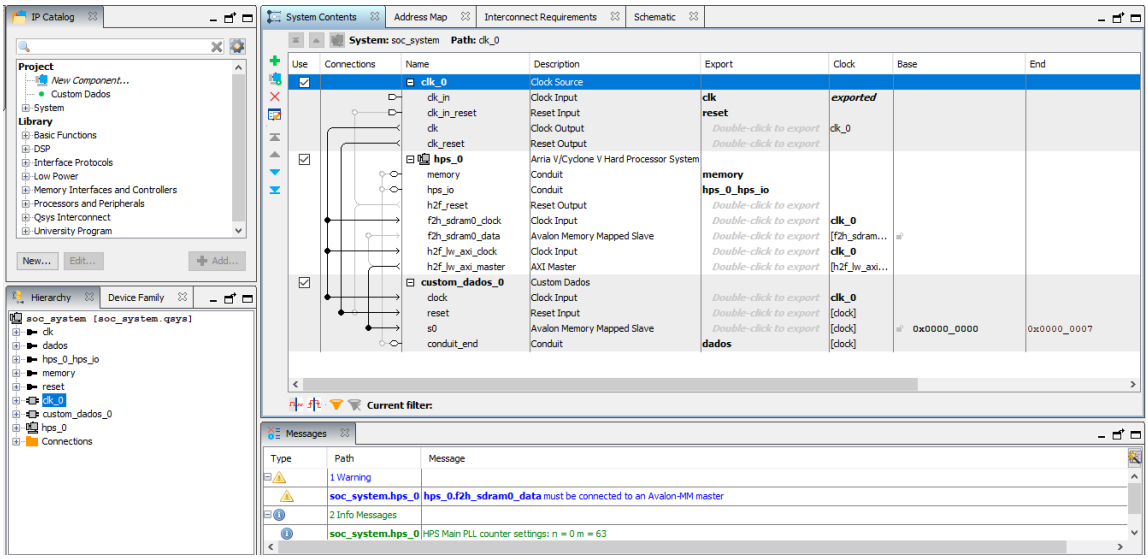
Com o módulo criado, o próximo passo a ser executado foi a instancialização da parte do *QSYS* do projeto, no qual é definido alguns componentes padrões do NIOS (clock do sistema e módulo HPS) e a ligação dos mesmos com o módulo criado anteriormente. O escopo do módulo criado pode ser visto e reproduzido utilizando o código presente no Apêndice I.

Os componentes básicos são responsáveis pela integração entre os pinos de diversos periféricos, ou seja, define qual bit de cada componente é ligado ao outro para assim compartilhar mesmo *clock*, mesmos *resets,* pinos de comunicação entre outros. O escopo final do QSYS utilizado foi o visto na Figura 10.

Todos os periféricos criados tem grande importância na criação do escopo do projeto, ou seja, a criação do *clock,* definição dos bits de HPS e junção com os pinos do custom_dados são necessários para a comunicação de mais baixo nível existente, pois como já definido, a ponte de informações que será utilizada é baseada em um

buffer, assim, tanto os pinos que irão acessar como também os bits de acesso devem ser definidos.

Figura 10 Criação QSYS.

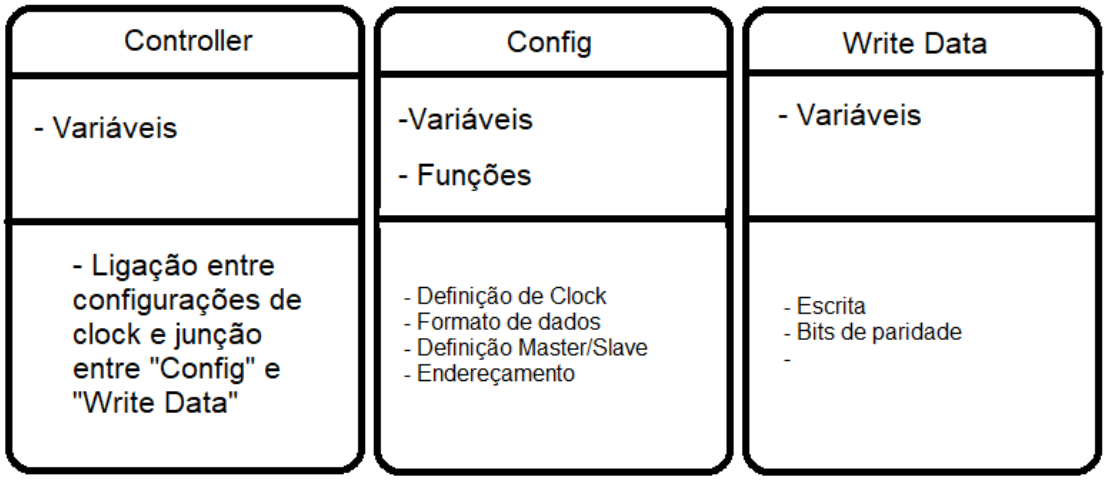


Fonte: Autoria própria (2020).

3.1.2 I²C

Visto que o sistema em questão necessita das funcionalidades e trocas de informações baseados no barramento I²C, foi necessário o estudo e a implementação do mesmo para tal funcionalidade, assim, de forma geral, foram criados três arquivos exibidos na Figura 11, para executar o controle de *clock,* definir as variáveis de acesso e por fim o controle de dados que respeitariam o tempo e sincronização para que fossem apresentados os dados no barramento HDMI.

Figura 11 Custom_I2C.



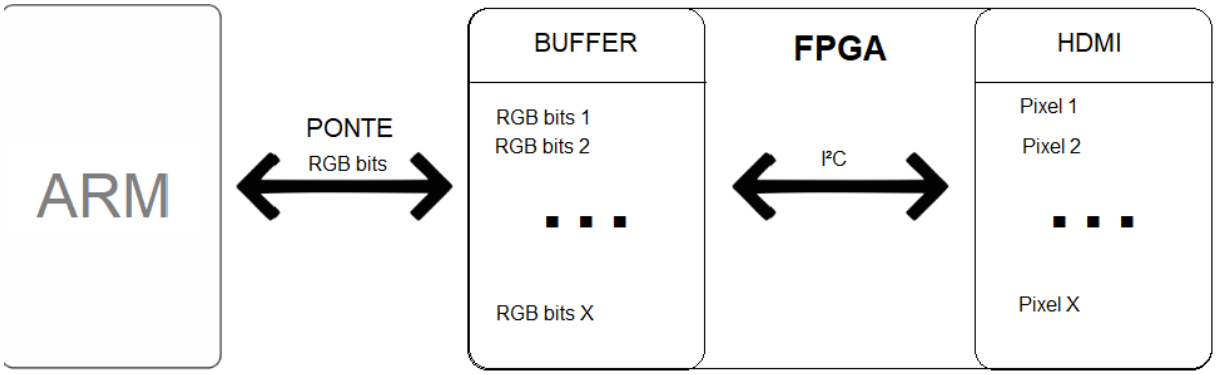
Fonte: Autoria própria (2020).

De forma geral são arquivos simples mas que dependem de um bom estudo para tal funcionamento, pois sua sincronização e junção de dados possuem um grau de compreensão dificultados. As informações utilizadas para criação do mesmo estão presentes nos Apêndices III, IV e V respectivamente.

3.1.3 Arquivo de controle e processamento de dados

Em si, no projeto, esse arquivo é o local ao qual é encontrado o coração do processo, ou seja, está presente nesse arquivo as entradas das informações referentes aos dados enviados pelo lado ARM, a criação do buffer de dados para sincronização e por fim a aplicação de um processamento de imagem simples, como mostrado na Figura 12.

Figura 12 Escopo Geral.

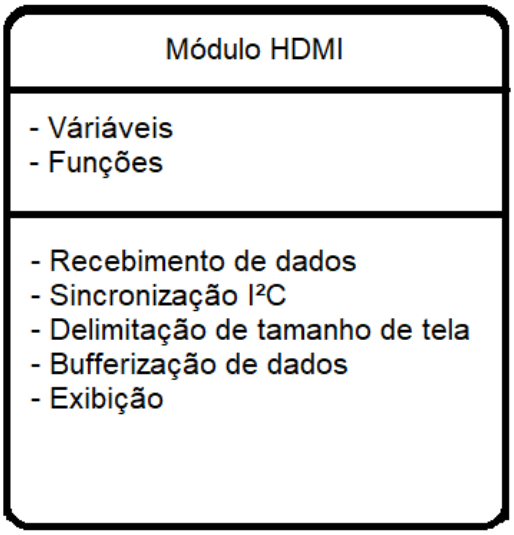


Fonte: Autoria própria (2020).

Antes de juntar as partes do sistema de comunicação pela ponte *Lightweight* HPS-FPGA, utilizou-se imagens comuns geradas pela FPGA para os testes de funcionamentos gerais. Quando as mesmas deram resultados positivos, foi integrado juntamente com os dados vindo do lado ARM.

Primeiramente é feito a criação das variáveis de entrada e saída, ao qual instanciando os arquivos anteriores, consegue-se ter o acesso direto as informações enviadas pelo ARM.

Figura 13 Modulo HDMI.



Fonte: Autoria própria (2020).

Como a sincronização de velocidade tanto do ARM quanto da FPGA não se ajustaram perfeitamente, se fez necessário criar um buffer de dados para que seja guardado as informações que estão chegando, e assim, serem exibidas no tempo certo pelo HDMI.

No mesmo momento que é feito o recebimento dos dados, já existe uma aplicação que faz um processamento simples nos dados recebidos. Como é uma implementação de teste, para esse processamento foi considerado somente a troca de cor recebida (vermelha), para uma cor secundaria (verde) e assim, cria-se um segundo buffer para guardar os dados processados.

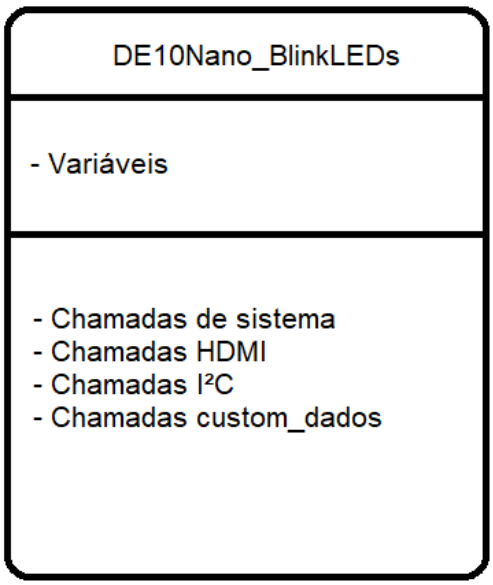
Por fim, utilizando o controle e informações de tempo no formato I²C é possível emitir as imagens diretamente para um monitor HDMI.

Para a reprodução do esquemático apresentado na Figura 13, pode-se usar os códigos inseridos no Apêndice VI.

3.1.4 Arquivo geral

Seguindo uma ordem cronológica dos passos realizados, dentro do sistema *Quartus,* baseado-se em documentos disponibilizados pela fabricante, criou-se um documento ao qual contem todos os referenciamentos de componentes que iriam ser utilizados, ligando assim a parte de *software* diretamente com a parte de *hardware.* Nesse arquivo também pode ser encontrado as chamadas de todas as funções necessárias para o bom funcionamento do projeto.

Figura 14 Arquivo geral.



Fonte: Autoria própria (2020).

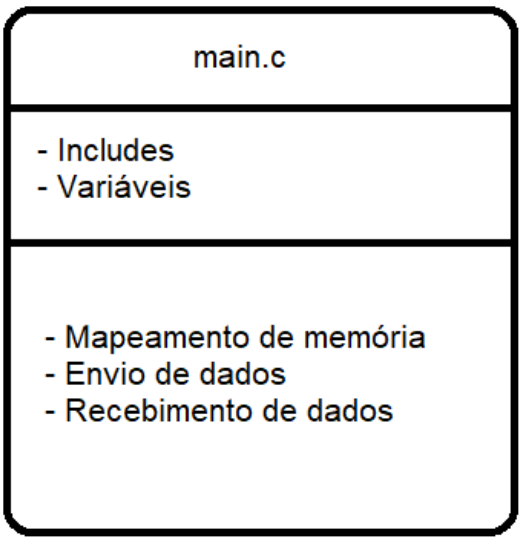
Essa etapa foi atualizada diversas vezes, pois é um arquivo que deve estar criado desde o inicio do projeto, assim, cada vez que é adicionado uma funcionalidade nova, deve fazer suas chamadas e definições de escopo nesse arquivo.

Quando finalizado, chegou-se ao resultado visto na Figura 14 em que seu código total está mencionado no Apêndice II.

3.2 PROJETO ARM

O processo de execução no lado ARM da placa em questão possui um escopo bem mais simples do que o mostrado no lado FPGA. Como todas as informações são executadas, processadas e exibidas pelas funções criadas no Quartus, restou para o ARM somente a criação de um arquivo de execução na linguagem C, como visto na Figura 15.

Figura 15 Arquivo ARM.



Fonte: Autoria própria (2020).

Tal arquivo mencionado tem como suas principais funcionalidades definir corretamente o local ao qual está enviando essas informações, ou seja, fazer o acesso de memória nos mesmos bits em que a FPGA consegue acessar e também enviar os dados de forma ordenada, que no qual foi definido o formato RGB.

Para ser feitos os testes intermediários de funcionamento dessa comunicação de dados, criou-se um módulo simplificado que fazia uma comunicação de somente 8 bits e que tinha o propósito de acender *LEDs* diretamente pelo ARM acessando o lado FPGA. Quando funcionando, aumentou-se a escrita para o máximo de 32 bits suportados pela ponte, tal módulo pode ser reproduzindo utilizando o código presente no Apêndice VII.

3.3 FUNCIONAMENTO

Após todas as etapas executadas, conectou-se todos os cabos necessários, compilou-se os códigos mencionados e foram gravados tanto na FPGA como também foi incluído no *boot* de inicialização do sistema, dessa forma, hora que executado o arquivo C no lado ARM, as informações de processamento são exibidas na tela HDMI.

4 RESULTADOS

O foco principal deste projeto, como já mencionado, era a criação de uma infraestrutura que integrasse uma comunicação direta entre ARM e FPGA além de fornecer suporte para realização de processamentos de dados em geral, assim, todos os resultados aqui apresentados estarão ligados diretamente ao funcionamento de todas as partes do desenvolvimento.

Seguindo uma ordem cronológica, de forma em que as execuções das etapas testadas foram apresentando evoluções, considerou-se que as mesmas eram de extrema importância para o projeto em si, e assim, serão demonstradas a seguir.

Partindo de que a disponibilização para um meio externo da placa DE10-Nano utiliza a comunicação HDMI, começaram-se as implementações e testes para que a mesma pudesse ser utilizada. Estudando a fundo o modo de execução da mesma, como já mencionado, precisou-se utilizar o barramento I²C para manter uma conexão estável.

A dificuldade primordial na implementação do barramento HDMI foi a população da tela, pois, quando enviado uma certa quantidade de bits, a velocidade em que a demonstração trabalha, fazia com que as imagens e cores se deslocavam e não respeitavam a sequência de exibição.

O momento em que se conseguiu definir os pixels nos lugares certos sem se sobreporem, pode ser considerado o primeiro teste positivo do projeto, como visto na Figura 16. Essas cores eram acionadas pelas chaves de acionamento dispostas pela própria placa DE10Nano.

Figura 16 Testes de cores para monitor HDMI.

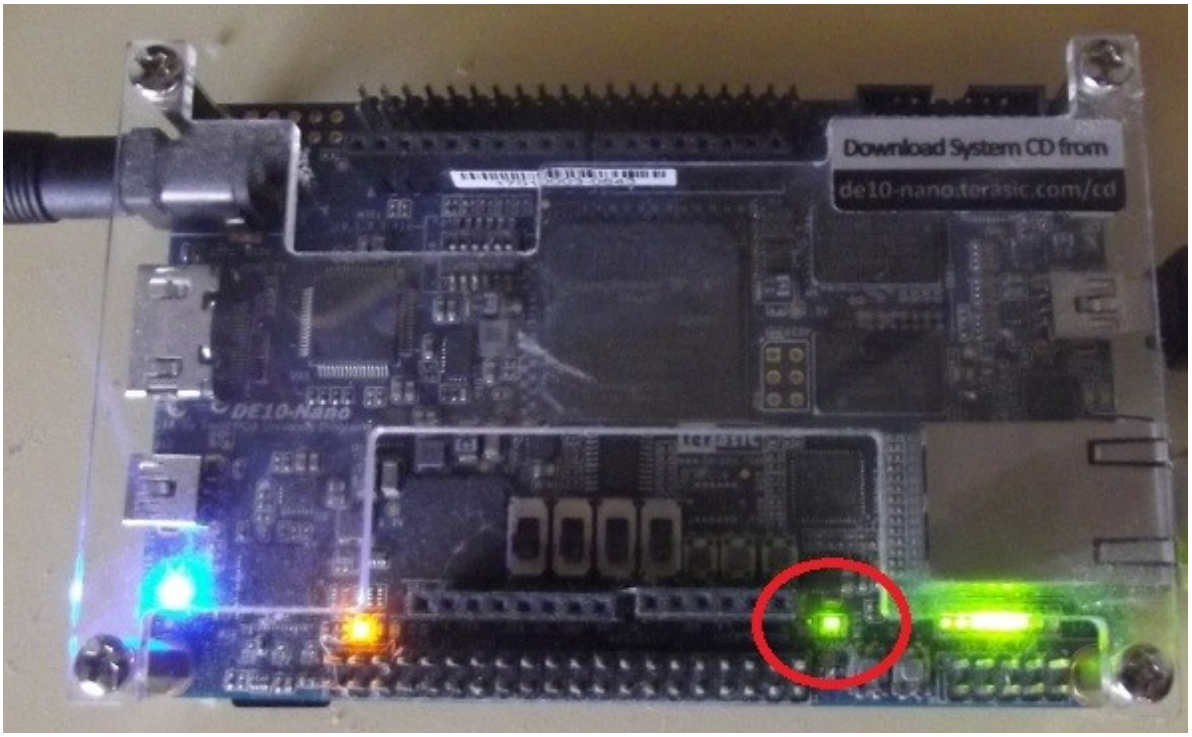
Fonte: Autoria própria (2020).

Após definir e funcionar a parte de HDMI e consequentemente os módulos I²C, o próximo passo era realizar o funcionamento da comunicação entre HPS e FPGA. Inicialmente, partindo sempre de uma realização mais simplificada, procurou-se efetuar somente a troca de um bit de comunicação entre os dois escopos principais.

De modo geral, o lado HPS enviava um bit no barramento, sendo o mesmo representado como alto ou baixo, dessa forma, o lado FPGA recebia esse bit e correspondendo ao nível recebido, acendia ou apagava um LED da placa.

O funcionamento dessa etapa pode ser vista na Figura 17 e o mesma pode ser considerada uma das etapas de maior impacto no processo geral, pois a partir da mesma, aumentando o barramento e fazendo os ajustes necessários conseguiria iniciar realmente a metodologia consistente do projeto.

Figura 17 Testes de barramento com leds.



Fonte: Autoria própria (2020).

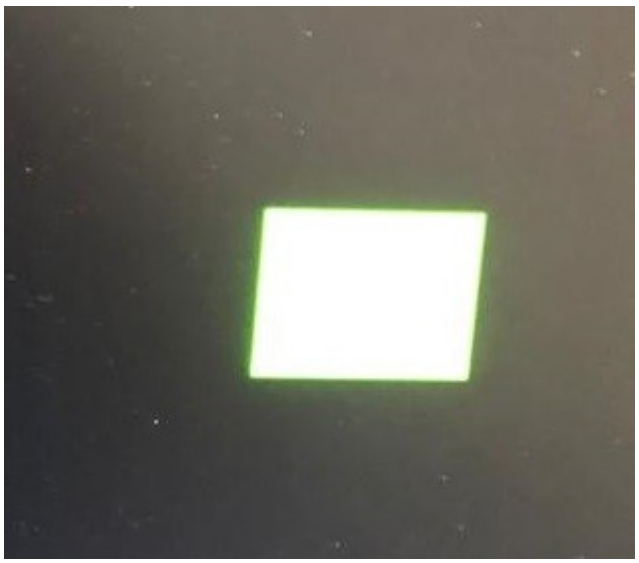
Com o funcionamento da ponte Lightweight HPS-FPGA, realizou-se as modificações no tamanho do barramento e também no envio de dados pelo HPS, chegando assim em um barramento de 32 bits de dados e com o envio no formato RGB.

Da mesma forma em que os bits quando recebidos acendiam e apagavam o LED correspondente, quando foi aumentado o barramento, a FPGA passou a ter acesso aos 32 bits de dados, diferenciando somente que, em vez de alocar o bit de forma a modificar o estado do LED os bits eram recebidos pelo bloco de processamento da FPGA, passaram a se transformar em pixels RGB e são enviados no barramento HDMI.

A temporização de comunicação entre a ponte Lightweight HPS-FPGA e o envio de dados da FPGA para o barramento HDMI não se encaixam perfeitamente, assim, se fez necessário a criação de um buffer de dados ao qual era atualizado quando os dados eram recebidos na ponte e, enquanto isso, a tela era atualizada pelos valores já recebidos anteriormente presentes no buffer.

Na Figura 18 é possível ver de forma real como ficava as informações recebidas pelo barramento e apresentadas pela HDMI após a bafferização dos dados.

Figura 18 Testes de barramento com HDMI.

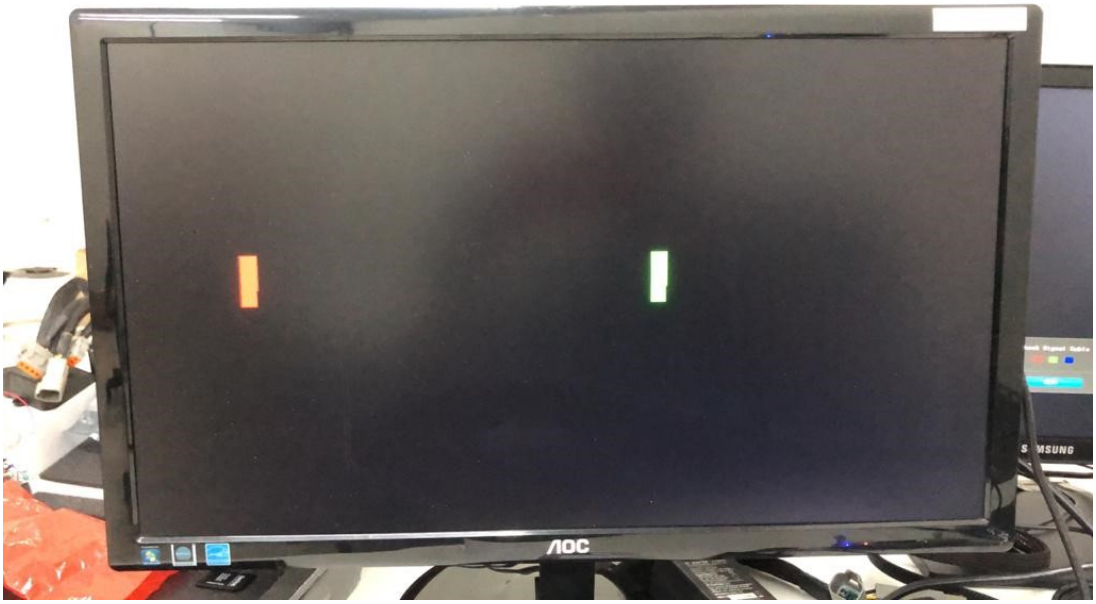


Fonte: Autoria própria (2020).

Finalmente com toda a estrutura montada, escolheu-se como um processamento de imagem simples a inversão dos bits RGB, ou seja, fazendo um deslocamento de bits. Dessa forma, quando recebido os bits correspondentes a cor vermelha, os mesmos passam a corresponder a cor verde, os relacionados a cor verde se transformam na cor azul e assim por diante como visto na Figura 19, ao qual, no lado esquerdo, está sendo criado o quadrado com os bits originais recebidos no

barramento e, do lado direito, mostra-se os bits após serem processados pelo sistema de deslocamento de bits.

Figura 19 Resultados finais da infraestrutura.



Fonte: Autoria própria (2020).

5 CONCLUSÃO

Desde os primeiros avanços que se conseguiu com a placa DE10Nano, a mesma se mostrou com um potencial bastante interessante tanto de processamento quanto para ser um dispositivo para fins gerais. A possibilidade de usar a mesma placa integrando as vantagens de um ARM e de uma FPGA são de grande valia acadêmica e possivelmente industrial.

Outra informação que se pôde chegar é que a melhor forma de se trabalhar com tecnologia reconfigurável seria com a divisão de processos, utilizando a complexidade da FPGA somente no momento em que se executa os processamentos, assim, para outros serviços, como comunicação homem-máquina, USB entre outros poderiam ser executados na parte HPS ao qual sua execução é menos complexa.

Com o término deste trabalho, pode-se concluir que a complexidade tanto da criação dos módulos como também intercomunicação entre todo o projeto se mostrou dificultosa, mas, em contra-partida, após a união e execução deste projeto em si, outros projetos poderão usar o mesmo como base e consequentemente terão a oportunidade de trabalhar mais não só com as pontes de comunicação HPS-FPGA como também com processamentos de imagens mais eficientes.

REFERÊNCIAS

TOCCI, Ronald J et al. **Sistemas digitais: princípios e aplicações**. São Paulo: Pearson Prentice Hall, 2011.

CHAGAS, Elnatan Ferreira. **Circuitos lógicos programáveis**. Disponível em: http://www.demic.fee.unicamp.br/~elnatan/ee610/24a%20Aula.pdf. Acesso em: 14 abr. 2017.

MARQUES FILHO, Ogê; VIEIRA NETO, Hugo. **Processamento digital de imagens**, Rio de Janeiro: Brasport, 1999

NATIONAL INSTRUMENTS. **Fundamentos da tecnologia FPGA**. Disponível em: http://www.ni.com/white-paper/6983/pt/. Acesso em: 17 abr. 2017.

FERNANDEZ BUENO, M. A.; SOARES BRASIL, C. R.; MARQUES, E. **Sistema operacional embarcado eCos com suporte a SMP para o processador Nios II**. *In*: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 27, 2007, Rio de Janeiro. Anais do XXVII Congresso da SBC, Rio de Janeiro, 2007. p. 764-776.

VASCONCELOS DE BRITO, Alisson. **Introdução a arquiteturas de computadores**. Disponível em: http://producao.virtual.ufpb.br/books/edusantana/introducao-a-arquitetura-de-computadores-livro/livro/livro.chunked/ch04s04.html. Acesso em: 25 abr. 2017 (2014).

MENDES DA SILVA, Raphael Philipe. **Processador softcore Altera NIOS II**. Disponível em: https://www.embarcados.com.br/altera-nios-ii/. Acesso em: 25 abr. 2017 (2014).

ALTERA, **Nios II processor reference handbook.** Altera Corporation, 2005.

BONDALAPATI, K.; PRASANA, V.K. **Reconfigurable computing systems**. Proceedings of the IEEE, v. 90, n. 7, p. 1201-1217, 2002.

SACCO, Francesco. **Comunicação SPI – parte 1**. Disponível em: https://www.embarcados.com.br/spi-parte-1/. Acesso em: 27 abr. 2017.

MACEDO, Diego. **Arquitetura: Von Neumman vs Harvard**. Disponível em: http://www.diegomacedo.com.br/arquitetura-von-neumann-vs-harvard/. Acesso em: 13 abr. 2017.

SPRING – DPI/INPE. **Teoria: processamento de imagens**. Disponível em: http://www.dpi.inpe.br/spring/teoria/realce/realce.htm. Acesso em: 30 abr. 2017.

SANTOS, M. M. D. **Redes de comunicação automotiva – características, tecnologias e aplicações**. São Paulo, 2010. 220p.

MORIMOTO, Carlos E. **Hardware – o guia definitivo.** GDH Press e Sul Editores. 2007.

PIXININE, Juliana. **USB: saiba o que significa e conheça a hitória desse importante padrão**. Disponível em: http://www.techtudo.com.br/noticias/noticia/2015/07/usb-saiba-o-que-significa-e-conheca-historia-deste-importante-padrao.html. Acesso em: 17 abr. 2017.(2015).

CLUBE DO *WARDWARE*. **USB – estrutura elétrica**. Disponível em: http://www.clubedohardware.com.br/artigos/placas-mae/usb-estrutura-el%C3%A9trica-r34052/?nbcpage=3. Acesso em: 17 abr. 2017.(1998).

FRANCIOLLON, A.; CASTELLUCCIA, C. **Code injection attacks on harvard-architeture devices**. *In*: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 15, 2008, Virgínia. CCS '08: Proceedings of the 15th ACM conference on Computer and communications security. Virgínia, 2008. p. 15-26.

RANGEL DE QUEIROZ, José Eustáquio; MARTINS GOMES, Hernan. **Introdução ao processamento digital de imagens.** Disponível em: http://www.dsc.ufcg.edu.br/~hmg/disciplinas/graduacao/vc-2014.1/Rita-Tutorial-PDI.pdf. Acesso em: 26 abr. 2017. (2001).

MORAES, Elisabete Moraes. **Níveis de entrada e saída das “v” e “i” dos cis digitais**. Disponível em: http://www.daelt.ct.utfpr.edu.br/elisanm/Digital/Rot3_3NiveisInOut_2oSem15.pdf. Acesso em: 27 maio 2017. (2015).

GTA UFRJ. **Funcionamento – portas e conectores**. Disponível em: https://www.gta.ufrj.br/grad/08_1/uwb/Complementos/Conteudo_USB_funcionamento.htm. Acesso em: 25 maio 2017. (2005).

ATACK Carol; VAN SOMEREN Alex. **The ARM RISC chip: a programmer's guide.** [s.n], 1993. Disponível em: http://www.ot1.com/arm/armchap1.html. Acesso em: 26 maio 2017. (1993).

ARM LIMITED. **ARM architecture reference manual**. 2011.

MORGAN Timothy Prickett. **ARM holdings eager for PC and server expansion**. [s.n.], 2011. Disponível em: https://www.theregister.com/2011/02/01/arm_holdings_q4_2010_numbers/. Acesso em: 06 jul. 2020.

ARM LIMITED. **Cortex-A9 technical reference manual.** [s.n], 2020. Disponível em: https://developer.arm.com/documentation/ddi0388/i/introduction/about-the-cortex-a9-processor?lang=en. Acesso em: 06 jul. 2020.

TERASIC. **DE10-Nano kit**. [s.n], 2020. Disponível em: https://www.terasic.com.tw/cgi-

bin/page/archive.pl?Language=English&CategoryNo=205&No=1046&PartNo=1. Acesso em: 06 jul. 2020

ALTERA. **Cyclone V hard processor system technical reference manual**. [s.n], 2018. Disponível em: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_54001.pdf. Acesso em: 10 ago. 2020

STACKOVERFLOW. **I²C,** 2018. Disponível em: https://stackoverflow.com/tags/i2c/info. Acesso em: 12 ago. 2020.

INTEL. **Introduction to Arria 10 hard processor system**. Disponível em: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_54001.pdf. Acesso em: 13 nov. 2020.

APÊNDICE I – MÓDULO DE DADOS

Este código consiste na criação de um módulo do QSYS.

```
module custom_dados
(
   input  logic        clk,               // clock.clk
   input  logic        reset,              // reset.reset

   // Memory mapped read/write slave interface
   input  logic        avs_s0_address,     // avs_s0.address
   input  logic        avs_s0_read,        // avs_s0.read
   input  logic        avs_s0_write,       // avs_s0.write
   output logic [31:0] avs_s0_readdata,    // avs_s0.readdata
   input  logic [31:0] avs_s0_writedata,   // avs_s0.writedata

   output logic [31:0]  dados
);

// Read operations performed on the Avalon-MM Slave interface
always_comb begin
   if (avs_s0_read) begin
      case (avs_s0_address)
         1'b0    : avs_s0_readdata = dados;
         default : avs_s0_readdata = 'x;
      endcase
   end else begin
      avs_s0_readdata = 'x;
   end
end

// Write operations performed on the Avalon-MM Slave interface
always_ff @ (posedge clk) begin
   if (reset) begin
      dados <= '0;
   end else if (avs_s0_write) begin
      case (avs_s0_address)
         1'b0    : dados <= avs_s0_writedata;
         default : dados <= dados;
      endcase
   end
end

endmodule // custom_dados
```

APÊNDICE II – MÓDULO DE CONFIGURAÇÃO GERAL

Esse código foi gerado pela Terasic System Builder

```
`include "vgaHdmi.v"
`include "i2c/I2C_HDMI_Config.v"

module DE10Nano_BlinkLEDs(

    ////////// CLOCK //////////
    input             FPGA_CLK1_50,
    input             FPGA_CLK2_50,
    input             FPGA_CLK3_50,

    ////////// HDMI //////////
    inout             HDMI_I2C_SCL,
    inout             HDMI_I2C_SDA,
    inout             HDMI_I2S,
    inout             HDMI_LRCLK,
    inout             HDMI_MCLK,
    inout             HDMI_SCLK,
    output            HDMI_TX_CLK,
    output   [23: 0]  HDMI_TX_D,
    output            HDMI_TX_DE,
    output            HDMI_TX_HS,
    input             HDMI_TX_INT,
    output            HDMI_TX_VS,

    ////////// HPS //////////
    inout             HPS_CONV_USB_N,
    output   [14: 0]  HPS_DDR3_ADDR,
    output   [ 2: 0]  HPS_DDR3_BA,
    output            HPS_DDR3_CAS_N,
    output            HPS_DDR3_CK_N,
    output            HPS_DDR3_CK_P,
    output            HPS_DDR3_CKE,
    output            HPS_DDR3_CS_N,
    output   [ 3: 0]  HPS_DDR3_DM,
    inout    [31: 0]  HPS_DDR3_DQ,
    inout    [ 3: 0]  HPS_DDR3_DQS_N,
    inout    [ 3: 0]  HPS_DDR3_DQS_P,
    output            HPS_DDR3_ODT,
    output            HPS_DDR3_RAS_N,
    output            HPS_DDR3_RESET_N,
    input             HPS_DDR3_RZQ,
    output            HPS_DDR3_WE_N,
    output            HPS_ENET_GTX_CLK,
    inout             HPS_ENET_INT_N,
    output            HPS_ENET_MDC,
    inout             HPS_ENET_MDIO,
    input             HPS_ENET_RX_CLK,
    input    [ 3: 0]  HPS_ENET_RX_DATA,
    input             HPS_ENET_RX_DV,
    output   [ 3: 0]  HPS_ENET_TX_DATA,
    output            HPS_ENET_TX_EN,
    inout            HPS_GSENSOR_INT,
    inout            HPS_I2C0_SCLK,
    inout            HPS_I2C0_SDAT,
    inout            HPS_I2C1_SCLK,
    inout            HPS_I2C1_SDAT,
    inout            HPS_KEY,
    inout            HPS_LED,
```

```
    inout             HPS_LTC_GPIO,
    output            HPS_SD_CLK,
    inout             HPS_SD_CMD,
    inout    [ 3: 0]  HPS_SD_DATA,
    output            HPS_SPIM_CLK,
    input             HPS_SPIM_MISO,
    output            HPS_SPIM_MOSI,
    inout             HPS_SPIM_SS,
    input             HPS_UART_RX,
    output            HPS_UART_TX,
    input             HPS_USB_CLKOUT,
    inout    [ 7: 0]  HPS_USB_DATA,
    input             HPS_USB_DIR,
    input             HPS_USB_NXT,
    output            HPS_USB_STP,

    ////////////// KEY ///////////
    input    [ 1: 0]    KEY,

    ////////////// LED ///////////
    output   [ 7: 0]    LED,

    ////////////// SW ///////////
    input    [ 3: 0]    SW
);

//=======================================================
//  REG/WIRE declarations
//=======================================================
wire uart0_rx;
wire uart0_tx;

assign uart0_tx = 1'b1;

wire clock25, locked;
wire reset;
reg[31:0] in;
assign reset = ~KEY[0];

// audio em alta impedancia, não será usado
assign HDMI_I2S  = 1'b z;
assign HDMI_MCLK  = 1'b z;
assign HDMI_LRCLK = 1'b z;
assign HDMI_SCLK  = 1'b z;

//=======================================================
//  Structural coding
//=======================================================

// **VGA CLOCK**
pll_25 pll_25(
  .refclk(FPGA_CLK1_50),
  .rst(reset),

  .outclk_0(clock25),
  .locked(locked)
  );

  // **VGA MAIN CONTROLLER**
vgaHdmi vgaHdmi (
```

```
  // input
  .clock      (clock25),
  .clock50    (FPGA_CLK1_50),
  .reset      (~locked),
  .start                                                    (in[24]),
  .entradaRGB (in[23:0]),
  .hsync      (HDMI_TX_HS),
  .vsync      (HDMI_TX_VS),

  // output
  .dataEnable (HDMI_TX_DE),
  .vgaClock   (HDMI_TX_CLK),
  .RGBchannel (HDMI_TX_D),
  .ledTeste                                                 (LED[6:1])
);

// **I2C Interface**
I2C_HDMI_Config #(
  .CLK_Freq (50000000), // 50MHz
  .I2C_Freq (20000)    // 20kHz for i2c clock
  )

I2C_HDMI_Config (
  .iCLK       (FPGA_CLK1_50),
  .iRST_N     (KEY[0]),
  .I2C_SCLK    (HDMI_I2C_SCL),
  .I2C_SDAT    (HDMI_I2C_SDA),
  .HDMI_TX_INT (HDMI_TX_INT),
  .READY       (LED[0])
);


soc_system u0(
            // Clock & Reset
            .clk_clk(FPGA_CLK1_50),

                                                          .reset_reset_n(KEY[0]),
            // HPS ddr3
            .memory_mem_a(HPS_DDR3_ADDR),
            .memory_mem_ba(HPS_DDR3_BA),
            .memory_mem_ck(HPS_DDR3_CK_P),
            .memory_mem_ck_n(HPS_DDR3_CK_N),
            .memory_mem_cke(HPS_DDR3_CKE),
            .memory_mem_cs_n(HPS_DDR3_CS_N),
            .memory_mem_ras_n(HPS_DDR3_RAS_N),
            .memory_mem_cas_n(HPS_DDR3_CAS_N),
            .memory_mem_we_n(HPS_DDR3_WE_N),
            .memory_mem_reset_n(HPS_DDR3_RESET_N),
            .memory_mem_dq(HPS_DDR3_DQ),
            .memory_mem_dqs(HPS_DDR3_DQS_P),
            .memory_mem_dqs_n(HPS_DDR3_DQS_N),
            .memory_mem_odt(HPS_DDR3_ODT),
            .memory_mem_dm(HPS_DDR3_DM),
            .memory_oct_rzqin(HPS_DDR3_RZQ),
            // HPS SD card

            .hps_0_hps_io_hps_io_sdio_inst_CMD(HPS_SD_CMD),

            .hps_0_hps_io_hps_io_sdio_inst_D0(HPS_SD_DATA[0]),
```

```
            .hps_0_hps_io_hps_io_sdio_inst_D1(HPS_SD_DATA[1]),
            .hps_0_hps_io_hps_io_sdio_inst_CLK(HPS_SD_CLK),
            .hps_0_hps_io_hps_io_sdio_inst_D2(HPS_SD_DATA[2]),
            .hps_0_hps_io_hps_io_sdio_inst_D3(HPS_SD_DATA[3]),
            // HPS UART
            .hps_0_hps_io_hps_io_uart0_inst_RX(HPS_UART_RX),
            .hps_0_hps_io_hps_io_uart0_inst_TX(HPS_UART_TX),
            .dados_readdata(in[31:0])
        );
             assign LED[7] = in[24];
endmodule
```

APÊNDICE III – MÓDULO DE CONFIGURAÇÃO I2C

Este módulo foi criado para fazer a configuração da comunicação I²C

```
`include "I2C_WRITE_WDATA.v"

module I2C_Controller (
        input  CLOCK,
        input  [23:0]I2C_DATA,
        input  GO,
        input  RESET,
        input  W_R,
        inout  I2C_SDAT,
        output I2C_SCLK,
        output END,
        output ACK
);

wire SDAO ;

assign I2C_SDAT = SDAO?1'bz : 1'b0  ;

I2C_WRITE_WDATA  wrd(
        .RESET_N  ( RESET),
        .PT_CK    ( CLOCK),
        .GO       ( GO   ),
        .END_OK   ( END  ),
        .ACK_OK   ( ACK  ),
        .BYTE_NUM ( 2    ),  //2byte
        .SDAI     ( I2C_SDAT ),//IN
        .SDAO     ( SDAO     ),//OUT
        .SCLO     ( I2C_SCLK ),
        .SLAVE_ADDRESS( I2C_DATA[23:16] ),
        .REG_DATA     ( I2C_DATA[15:0]  )
);



endmodule
```

APÊNDICE IV - MÓDULO DE CONTROLE I2C

Este código foi escrito para fazer o controle da comunicação I²C

```
`include "I2C_Controller.v"

module I2C_HDMI_Config ( // Host Side
    iCLK,
    iRST_N,
    // I2C Side
    I2C_SCLK,
    I2C_SDAT,
    HDMI_TX_INT,
    READY
     );
// Host Side
input    iCLK;
input    iRST_N;
// I2C Side
output   I2C_SCLK;
inout    I2C_SDAT;
input    HDMI_TX_INT;
output READY ;

// Internal Registers/Wires
reg [15:0] mI2C_CLK_DIV;
reg [23:0] mI2C_DATA;
reg    mI2C_CTRL_CLK;
reg    mI2C_GO;
wire    mI2C_END;
wire    mI2C_ACK;
reg [15:0] LUT_DATA;
reg [5:0]  LUT_INDEX;
reg [3:0]  mSetup_ST;
reg READY ;

// Clock Setting
parameter CLK_Freq = 50000000; // 50 MHz
parameter I2C_Freq = 20000;  // 20 KHz
// LUT Data Number
parameter LUT_SIZE = 31;

////////////////////// I2C Control Clock ////////////////////////
always@(posedge iCLK or negedge iRST_N)
begin
 if(!iRST_N)
 begin
  mI2C_CTRL_CLK <= 0;
  mI2C_CLK_DIV <= 0;
 end
 else
 begin
  if( mI2C_CLK_DIV < (CLK_Freq/I2C_Freq) )
   mI2C_CLK_DIV <= mI2C_CLK_DIV+1'b1;
  else
  begin
   mI2C_CLK_DIV <= 0;
   mI2C_CTRL_CLK <= ~mI2C_CTRL_CLK;
  end
 end
end
```

```
////////////////////////////////////////////////////////////////////
I2C_Controller  u0 ( .CLOCK(mI2C_CTRL_CLK), // Controller Work Clock
      .I2C_SCLK(I2C_SCLK),    // I2C CLOCK
           .I2C_SDAT(I2C_SDAT),    // I2C DATA
      .I2C_DATA(mI2C_DATA),   // DATA:[SLAVE_ADDR,SUB_ADDR,DATA]
      .GO(mI2C_GO),      // GO transfor
      .END(mI2C_END),     // END transfor
      .ACK(mI2C_ACK),     // ACK
      .RESET(iRST_N) );
////////////////////////////////////////////////////////////////////
////////////////////// Config Control ////////////////////////////
always@(posedge mI2C_CTRL_CLK or negedge iRST_N)
begin
 if(!iRST_N)
 begin
 READY<=0;
  LUT_INDEX <= 0;
  mSetup_ST <= 0;
  mI2C_GO  <= 0;
 end
 else
 begin
  if(LUT_INDEX<LUT_SIZE)
  begin
  READY<=0;
   case(mSetup_ST)
   0: begin
     mI2C_DATA <= {8'h72,LUT_DATA};
     mI2C_GO  <= 1;
     mSetup_ST <= 1;
    end
   1: begin
     if(mI2C_END)
     begin
      if(!mI2C_ACK)
      mSetup_ST <= 2;
      else
      mSetup_ST <= 0;
      mI2C_GO  <= 0;
     end
    end
   2: begin
     LUT_INDEX <= LUT_INDEX+1'b1;
     mSetup_ST <= 0;
    end
   endcase
  end
  else
  begin
    READY<=1;
    if(!HDMI_TX_INT)
    begin
      LUT_INDEX <= 0;
    end
    else
      LUT_INDEX <= LUT_INDEX;
  end
 end
end
////////////////////////////////////////////////////////////////////
```

```
////////////////////// Config Data LUT   //////////////////////////
always
begin
 case(LUT_INDEX)

 // Video Config Data
 0 : LUT_DATA <= 16'h9803;  //Must be set to 0x03 for proper operation
 1 : LUT_DATA <= 16'h0100;  //Set 'N' value at 6144
 2 : LUT_DATA <= 16'h0218;  //Set 'N' value at 6144
 3 : LUT_DATA <= 16'h0300;  //Set 'N' value at 6144
 4 : LUT_DATA <= 16'h1470;  // Set Ch count in the channel status to 8.
 5 : LUT_DATA <= 16'h1520;  //Input 444 (RGB or YCrCb) with Separate Syncs, 48kHz fs
 6 : LUT_DATA <= 16'h1630;  //Output format 444, 24-bit input
 7 : LUT_DATA <= 16'h1846;  //Disable CSC
 8 : LUT_DATA <= 16'h4080;  //General control packet enable
 9 : LUT_DATA <= 16'h4110;  //Power down control
 10 : LUT_DATA <= 16'h49A8;  //Set dither mode - 12-to-10 bit
 11 : LUT_DATA <= 16'h5510;  //Set RGB in AVI infoframe
 12 : LUT_DATA <= 16'h5608;  //Set active format aspect
 13 : LUT_DATA <= 16'h96F6;  //Set interrup
 14 : LUT_DATA <= 16'h7307;  //Info frame Ch count to 8
 15 : LUT_DATA <= 16'h761f;  //Set speaker allocation for 8 channels
 16 : LUT_DATA <= 16'h9803;  //Must be set to 0x03 for proper operation
 17 : LUT_DATA <= 16'h9902;  //Must be set to Default Value
 18 : LUT_DATA <= 16'h9ae0;  //Must be set to 0b1110000
 19 : LUT_DATA <= 16'h9c30;  //PLL filter R1 value
 20 : LUT_DATA <= 16'h9d61;  //Set clock divide
 21 : LUT_DATA <= 16'ha2a4;  //Must be set to 0xA4 for proper operation
 22 : LUT_DATA <= 16'ha3a4;  //Must be set to 0xA4 for proper operation
 23 : LUT_DATA <= 16'ha504;  //Must be set to Default Value
 24 : LUT_DATA <= 16'hab40;  //Must be set to Default Value
 25 : LUT_DATA <= 16'haf16;  //Select HDMI mode
 26 : LUT_DATA <= 16'hba60;  //No clock delay
 27 : LUT_DATA <= 16'hd1ff;  //Must be set to Default Value
 28 : LUT_DATA <= 16'hde10;  //Must be set to Default for proper operation
 29 : LUT_DATA <= 16'he460;  //Must be set to Default Value
 30 : LUT_DATA <= 16'hfa7d;  //Nbr of times to look for good phase

 default:  LUT_DATA <= 16'h9803;
 endcase
end
//////////////////////////////////////////////////////////////////
endmodule
```

APÊNDICE V – MÓDULO DE ESCRITA I²C

Este código foi escrito para fazer a escrita da comunicação I²C

```
module I2C_WRITE_WDATA  (
  input      RESET_N ,
 input          PT_CK,
 input          GO,
 input      [15:0] REG_DATA,
 input      [7:0]  SLAVE_ADDRESS,
 input            SDAI,
 output reg       SDAO,
 output reg       SCLO,
 output reg       END_OK,

 //--for test
 output reg [7:0]  ST ,
 output reg [7:0]  CNT,
 output reg [7:0]  BYTE,
 output reg       ACK_OK,
  input      [7:0]  BYTE_NUM  // 4 : 4 byte
);

//===reg/wire
reg   [8:0]A ;
reg   [7:0]DELY ;

always @( negedge RESET_N or posedge  PT_CK )begin
if (!RESET_N  ) ST <=0;
else
  case (ST)
    0: begin  //start
      SDAO   <=1;
       SCLO   <=1;
       ACK_OK <=0;
       CNT    <=0;
       END_OK <=1;
       BYTE   <=0;
       if (GO) ST  <=30 ; // inital
     end
    1: begin  //start
      ST <=2 ;
```

```
 { SDAO,  SCLO } <= 2'b01;
A <= {SLAVE_ADDRESS ,1'b1 };//WRITE COMMAND
 end
 2: begin  //start
   ST <=3 ;
 { SDAO,  SCLO } <= 2'b00;
 end

 3: begin
   ST <=4 ;
 { SDAO, A } <= { A ,1'b0 };
 end
 4: begin
   ST <=5 ;
 SCLO <= 1'b1 ;
CNT <= CNT +1'b1 ;
 end

 5: begin
 SCLO <= 1'b0 ;
 if (CNT==9) begin
   if ( BYTE == BYTE_NUM )  ST <= 6 ;
  else  begin
        CNT <=0 ;
        ST <= 2 ;
            if ( BYTE ==0 ) begin BYTE <=1  ; A <= {REG_DATA[15:8] ,1'b1 }; end
        else if ( BYTE ==1 ) begin BYTE <=2  ; A <= {REG_DATA[7:0] ,1'b1 }; end
   end
  if (SDAI ) ACK_OK <=1 ;
end
else ST <= 2;
 end

 6: begin         //stop
   ST <=7 ;
 { SDAO,  SCLO } <= 2'b00;
   end

 7: begin         //stop
   ST <=8 ;
```

```
    { SDAO,  SCLO } <= 2'b01;
      end
   8: begin          //stop
     ST <=9 ;
    { SDAO,  SCLO } <= 2'b11;

      end
 9: begin
     ST     <= 30;
  SDAO   <=1;
      SCLO   <=1;
      CNT    <=0;
      END_OK <=1;
      BYTE   <=0;
    end
 //--- END ---
   30: begin
        if (!GO) ST  <=31;
      end
   31: begin  //
     END_OK<=0;
  ACK_OK<=0;
  ST    <=1;
 end
 endcase
end

endmodule
```

APÊNDICE VI – MÓDULO HDMI

Este código foi escrito para fazer o controle da comunicação HDMI além de todo o processamento de imagens utilizado.

```
module vgaHdmi(
  // **input**
  input clock, clock50, reset, start,
  input [23:0] entradaRGB,

  // **output**
  output reg hsync, vsync,
  output reg dataEnable,
  output reg vgaClock,
  output reg [23:0] RGBchannel,
  output reg [5:0] ledTeste
);

reg [9:0]pixelH, pixelV; // pixel horizontal e vertical

reg [23:0] img [49:0][49:0];

integer x,y,a,b;

initial begin
  hsync     = 1;
  vsync     = 1;
  pixelH    = 0;
  pixelV    = 0;
  dataEnable = 0;
  vgaClock   = 0;
  x                    = 0;
  y                              = 0;
  a                    = 0;
  b                              = 0;

end

always @(posedge clock or posedge reset) begin
  if(reset) begin
    hsync  <= 1;
    vsync  <= 1;
```

```
    pixelH <= 0;
    pixelV <= 0;
  end
  else begin
    // Display Horizontal
    if(pixelH==0 && pixelV!=524) begin
      pixelH<=pixelH+1'b1;
      pixelV<=pixelV+1'b1;
    end
    else if(pixelH==0 && pixelV==524) begin
      pixelH <= pixelH + 1'b1;
      pixelV <= 0; // pixel 525
    end
    else if(pixelH<=640) pixelH <= pixelH + 1'b1;
    // Front Porch
    else if(pixelH<=656) pixelH <= pixelH + 1'b1;
    // Sync Pulse
    else if(pixelH<=752) begin
      pixelH <= pixelH + 1'b1;
      hsync  <= 0;
    end
    // Back Porch
    else if(pixelH<799) begin
      pixelH <= pixelH+1'b1;
      hsync  <= 1;
    end
    else pixelH<=0; // pixel 800

    // Sync Pulse
    if(pixelV == 491 || pixelV == 492)
      vsync <= 0;
    else
      vsync <= 1;
  end
end

// dataEnable signal
always @(posedge clock or posedge reset) begin
  if(reset) dataEnable<= 0;
```

```
  else begin
    if(pixelH >= 0 && pixelH <640 && pixelV >= 0 && pixelV < 480)
      dataEnable <= 1;
    else
      dataEnable <= 0;
  end
end

// VGA pixeClock signal
initial vgaClock = 0;

always @(posedge clock50 or posedge reset) begin
  if(reset) vgaClock <= 0;
  else      vgaClock <= ~vgaClock;
end
// ************************************************************
// Screen colors using de10nano switches for test
always @(posedge clock50 or posedge reset) begin
  if(reset) begin
              RGBchannel[23:16] = 8'd0;
              RGBchannel [15:8] = 8'd0;
              RGBchannel  [7:0] = 8'd0;
  end
  else if(pixelH > 150 && pixelH <= 200 && pixelV > 200 && pixelV <= 250) begin
              RGBchannel[23:16] = img[pixelH - 150][pixelV - 200][23:16];
              RGBchannel [15:8] = img[pixelH - 150][pixelV - 200] [15:8];
              RGBchannel  [7:0] = img[pixelH - 150][pixelV - 200]  [7:0];

  end
  else if(pixelH > 450 && pixelH <= 500 && pixelV > 200 && pixelV <= 250) begin
              RGBchannel[23:16] = img[pixelH - 450][pixelV - 200] [15:8];
              RGBchannel [15:8] = img[pixelH - 450][pixelV - 200][23:16];
              RGBchannel  [7:0] = img[pixelH - 450][pixelV - 200]  [7:0];
  end
  else begin
              RGBchannel[23:16] = 8'd0;
              RGBchannel [15:8] = 8'd0;
              RGBchannel  [7:0] = 8'd0;
  end
end
```

```
always @(posedge start) begin
  img[a][b]  [23:16] = entradaRGB[23:16];
  img[a][b]   [15:8] = entradaRGB[15:8];
  img[a][b]    [7:0] = entradaRGB[7:0];
// b <= b + 1;
// if(b >= 10) begin
//              a <= a + 1;
//              b <= 0;
// end
// if(a >= 10) begin
//                          a <= 0;
// end
  if(b < 49) begin
              b <= b + 1;
  end
  else if(b == 49 && a < 49) begin
              b <= 0;
              a <= a + 1;
  end
// else if(b == 49 && a == 49) begin
//              b <= 49;
//              a <= 49;
// end
  //ledTeste[5:0] = a[5:0];
end
/*
entradaImagens entradaImagens (
  // input
  .clk         (clock),
  .rst                     (reset),
  // output
  .Rchannel            (RGBchannel[23:16]),
  .Gchannel            (RGBchannel [15:8]),
  .Bchannel (RGBchannel  [7:0]),
);
*/
Endmodule
```

APÊNDICE VII – MÓDULO DA PONTE LIGHTWEIGHT (LADO HPS)

Este código foi escrito para fazer o acesso e envio de dados pelo lado HPS.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"
#include "socal/alt_gpio.h"
#include "hps_0.h"

#define REGS_BASE (ALT_LWFPGASLVS_OFST)
#define REGS_SPAN (0x00200000)

int main()
{
  int fd;
  void *virtual_base;
  int loop_count = 0;
  void *leds_addr;
  int counter = 0;

  printf("ALT_STM_OFST Value: 0x%08x\n", ALT_STM_OFST);
  printf("ALT_LWFPGASLVS_OFST Value: 0x%08x\n", ALT_LWFPGASLVS_OFST);

  // map the address space for the LED registers into user space so we can interact with them.
  // we'll actually map in the entire CSR span of the HPS since we want to access various registers within that span
  if ((fd = open( "/dev/mem", (O_RDWR | O_SYNC))) == -1)
  {
        printf("ERROR: could not open \"/dev/mem\"...\n");
        return -1;
  }

  virtual_base = mmap(NULL, REGS_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED, fd, REGS_BASE);

  if (virtual_base == MAP_FAILED)
  {
```

```
		printf("ERROR: mmap() failed...\n");
		close(fd);
		return -1;
	}

	leds_addr = virtual_base + CUSTOM_DADOS_0_BASE;

	// Toggle the LEDs in a counting pattern
	while (loop_count < 100000)
	{
		if(counter % 2 == 0){
			// Set LEDs to counter value
			*(uint32_t *)leds_addr = 16711680;
		} else {
			// Set LEDs to counter value
			*(uint32_t *)leds_addr = 33488896;
		}
		printf("%d\n", loop_count);

		// Wait 1s
		usleep(1000 * 50);

		// Increment loop counter
		++loop_count;
	}

	// Clean up our memory mapping and exit
	if (munmap(virtual_base, REGS_SPAN) != 0)
	{
		printf("ERROR: munmap() failed...\n");
		close(fd);
		return -1;
	}

	close(fd);
	return 0;
}
```