Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

DESENVOLVIMENTO E CARACTERIZAÇÃO DE UM MPSOC-NOC BASEADO EM PROCESSADORES RISC-V

Arthur Mendes Lima

PROJETO FINAL DE CURSO

ENGENHARIA ELÉTRICA

Brasília

2022

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

DESENVOLVIMENTO E CARACTERIZAÇÃO DE UM MPSOC-NOC BASEADO EM PROCESSADORES RISC-V

Arthur Mendes Lima

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro Eletricista

Orientador: Prof. Jones Yudi Mori Alves da Silva

Brasília
2022

Lima, Arthur Mendes.
L769d DESENVOLVIMENTO E CARACTERIZAÇÃO DE UM MPSOC-NOC BASEADO EM PROCESSADORES RISC-V / Arthur Mendes Lima; orientador Jones Yudi Mori Alves da Silva. -- Brasília, 2022.
59 p.

Projeto Final de Curso (Engenharia Elétrica) -- , 2022.

1. IP/CV. 2. MPSoC. 3. Computação Paralela. 4. Arquitetura de Computadores. I. Mori Alves da Silva, Jones Yudi, orient. II. Título

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

DESENVOLVIMENTO E CARACTERIZAÇÃO DE UM MPSOC-NOC BASEADO EM PROCESSADORES RISC-V

Arthur Mendes Lima

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro Eletricista

Trabalho aprovado. Brasília, 06 de Outubro de 2022:

Prof. Jones Yudi Mori Alves da Silva,
UnB/FT/ENM
Orientador

Prof. Daniel Mauricio Muñoz Arboleda,
UnB/FGA
Examinador

Prof. Prof. Renato Coral Sampaio,
UnB/FGA
Examinador

Brasília
2022

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Agradeço a minha família pelo apoio incondicional durante estes anos em Brasília. Sou muito grato aos meus tios e primos que tão bem me receberam aqui e fizeram que eu me sentisse em casa.

Meus agradecimentos também ao Prof. Dr. Jones Yudi pela orientação e parceria durante estes anos. Forte abraço nos meus colegas de curso e nos bichos de sete cabeças que conheci nesta Universidade.

“Do or do not.
There is no try”
(Master Yoda)

Resumo

Multi-processor System on Chip (MPSoCs) são sistemas de hardware compostos por múltiplos processadores interconectados. *Network-on-Chip* (NoCs) representam uma das mais recentes arquiteturas de comunicação intra-chip desenvolvida na indústria moderna. Este projeto propõe a continuação de uma série de trabalhos já realizados que desenvolveram uma arquitetura específica para *Image Processing / Computer Vision* (IP/CV) denominada de MCVision. Este trabalho utiliza da mesma arquitetura MCVision, com a mesma rede de comunicação intra-chip, adaptada para utilizar processadores RISC-V como elementos de processamento. Assim, foi feito o desenvolvimento da MCVision-RISCV utilizando FPGAs da *Xilinx* como base de prototipação. Os roteadores desenvolvidos foram integrados a processadores RISC-V disponíveis livremente (*open-hardware*). O resultado foi a implementação de uma rede 3x3 que realizou aplicações como *Sobel* e Rotação, foram utilizados 27K *look-up-table* e 35K *flip-flops*.

Palavras-chave: MPSoC, Computer Architecture, NoC

Abstract

Multi-processor System on Chip (MPSoCs) are hardware based systems composed of multiple tiles connected through a network known as *Network-on-Chip* (NoC). NoCs represent one of the state of the art solution for addressing the problems of inter-chip communications design. This project is a continuation of previous works developed in which was proposed an architecture specifically designed for *Image Processing / Computer Vision* (IP/CV) applications, The MCVision. Now, further exploration of MCVision architecture by integrating RISC-V processors (MCVision-RISCV), using *Xilinx* FPGAs and 3x3 mesh setup, the MCVision-RISC-V was implemented and deployed Rotation and Sobel applications developed in C/C++. The utilization report yields 7K *look-up-table* e 35K *flip-flops* consumed.

Keywords: IP/CV. MPSoC. Parallel Computing. Computer Architecture.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A difusão em massa de computadores na sociedade do século atual alterou drasticamente o cotidiano do ser humano. O entretenimento como filmes e videogames, antes exclusivos dos cinemas e fliperamas, hoje são a realidade de ordinários aparelhos domésticos como os celulares, notebooks, tablets e desktops. Para além do entretenimento, sistemas industriais, de segurança, hospitalares, de infraestrutura, de produção de alimentos, educacionais: computadores e processadores já são imprescindíveis para qualquer atividade na vida de grande parte da humanidade.

Inicialmente, computadores não foram construídos para um propósito tão abrangente, sendo os primeiros desenvolvidos para executar operações lógicas e aritméticas apenas. Sua evolução é uma consequência natural do desenvolvimento científico e tecnológico da indústria microeletrônica em dois elementos principais: a característica física (materiais que possuem um custo menor, mais robustez, menor consumo de energia, etc) e a característica organizacional (arquitetural) dos sistemas.

Observando as características dos primeiros circuitos integrados comercialmente disponíveis, Gordon Moore fez uma previsão de que a cada 2 anos a quantidade de transistores em um circuito de mesma área dobraria [1] . Após algum tempo, constatou-se que a indústria mantinha essa taxa de atualização a cada 18 meses, em média, o que durou por mais de duas décadas. Dificuldades tecnológicas, aliadas a fenômenos físicos outrora irrelevantes, começaram a atrasar a evolução, e a famosa Lei de Moore deixou de acompanhar a realidade da indústria. Um dos principais fenômenos é densidade de energia constante nos materiais semicondutores utilizados pela indústria microeletrônica, conhecido como *Dennard Scaling* [2].

Uma das soluções adotadas pela indústria para manter a evolução do poder de processamento dos computadores, foi o desenvolvimento de processadores com mais de um núcleo de processamento. Isso permite a execução em paralelo de diversas tarefas, que antes eram executadas de forma sequencial, potencialmente reduzindo o tempo total de processamento.

É evidente que a metodologia de desenvolvimento de microprocessadores para sistemas sistema embarcados que são projetados para desempenhar um conjunto de tarefas específicas não poderia ser o mesmo que para os processadores genéricos. Dentro deste contexto, soluções integrando exploração de paralelismo associadas à especialização focada na aplicação são de grande interesse.

[1] https://www.intel.com.br/content/www/br/pt/newsroom/opinion/moore-law-now-and-in-the-future.html

[2] https://en.wikipedia.org/wiki/Dennard_scaling

1.1 Motivação

Para muitas ocasiões, as soluções microprocessadas de uso geral atendem os requisitos das indústrias para aplicações simples de CV. Todavia, é comum que neste tipo de solução seja atribuído um hardware comum de uso geral para que o produto possa ser o mais flexível possível e isso negligencia a necessidade de especialização de uma arquitetura que seja ao mesmo tempo flexível para desempenhar tarefas diversas, seja facilmente programável e também possua uma arquitetura pautada para otimizar o processamento de imagens. Soluções gerais para CV não se valem de otimizações que podem ser realizadas por paralelização do processamento que é intuitivo num algoritmo de processamento de imagem.

Uma solução proposta por [1] e implementada fisicamente em [4] é a criação de uma arquitetura multiprocessada baseada em redes intra-chip capaz de explorar o paralelismo intrínseco dos algoritmos de processamento de imagens. Essa arquitetura, MCVision (*Many-Core Vision*), opera como mostrado na Figura 1. A imagem a ser processada e dividida em quadrantes e cada quadrante é atribuído a um *Tile*, responsável pelo processamento em si.

Porém esta implementação anteriormente feita possui limitações que impedem o desenvolvimento de aplicações mais sofisticadas demandadas no contexto corrente: o antigo elemento de processamento (PE) era essencialmente uma ULA (Unidade Lógica Aritmética) com apenas 16 instruções somente possível de ser programado diretamente por Assembly.

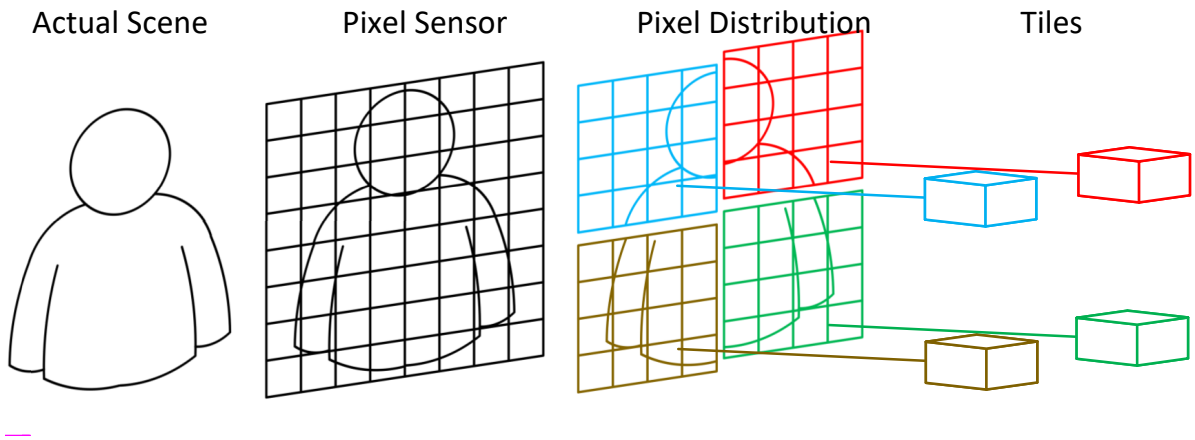
A despeito das limitações, os resultados dos trabalhos citados se mostraram promissores para aplicações em IP/CV, a grande motivação deste trabalho é exploração desta arquitetura de forma que seja possível expandir suas aplicações por meio da programabilidade dos PEs.

Assim, uma proposta mais interessante para explorar esta arquitetura com aplicações mais sofisticadas é utilizar um processador que possua um compilador para linguagem de alto nível como C/C++, e que seja possível customizar a seleção do seu conjunto de instrução ISA para atender diversas necessidade de *trade-off*.

Assim, é proposta a ideia de implementar a mesma arquitetura MCVision de [4] com um *Processing Element* (PE) mais adequado para aplicações de alto nível.

Neste trabalho, a escolha foi pelo processador RISC-V que é um processador de uso geral que possui escalabilidade para atender diversas aplicações por meio de um conjunto de instruções modulado, isso possibilita que o processador possa atender diversas aplicações com o mínimo de requisitos possíveis. Além disso, outro ponto importante de escolher um processador como o RISC-V, é o fato de ser *open-source*, há diversas implementações em *Hardware Descripton Languages* (HDL) disponíveis que democratizam o acesso ao RISC-V e ainda deixa possibilidade para alterá-lo como bem entender. Ainda mais, o RISC-V possui compiladores que permitem mais rapidamente o desenvolvimento de aplicações novas e mais complexas o que é fundamental para se trabalhar com algoritmos pesados de IP/CV.

Figura 1 – Particionamento da imagem em quadrantes atribuídos a diferentes Tiles. Fonte: [1]



A rede de comunicação utilizada é uma adaptação do trabalho de [4] que foi desenvolvida anteriormente pelo Autor. Foram feitas modificações pontuais para adequar à interface do RISC-V. Neste trabalho, foram implementadas adaptações à interface do RISC-V escolhido (descrição em seções seguintes) e a memória de pixel foi desenvolvida integralmente.

Ao longo do texto, quando for necessário especificar qual arquitetura estava em questão, MCVision é a implementação dos trabalhos posteriores e MCVision-RISCV é atribuída ao corrente trabalho. Ao longo do texto, as menções sobre a arquitetura MCVision também pode ser aplicada para MCVision-RISCV, o contrário não é verdade.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é proporcionar à arquitetura multiprocessada MCVision uma maior portabilidade de aplicações integrando o RISC-V como PE, resultando na MCVision-RISCV.

1.2.2 Objetivos Específicos

- Adaptar a NoC, desenvolvida em trabalhos anteriores, para o modelo RISC-V selecionado.
- Adaptar a PM, desenvolvida em trabalhos anteriores, para o modelo RISC-V selecionado.
- Selecionar um RISC-V em repositório aberto e adaptar a sua interface para integrar a NoC e a PM.
- Criar biblioteca em software para o RISC-V operar sobre pixels locais e remotos.

- Criar um guia de programação e desenvolvimento de aplicações para a arquitetura MCVision-RISCV.

1.3 Estrutura do Texto

O capítulo 2 seguirá com uma revisão teórica em que serão abordados os principais pontos do desenvolvimento do projeto: Microprocessadores, Linguagem de Programação, Compiladores, Assemblers, etc. O capítulo 3 é feita a descrição da proposta deste trabalho detalhando o desenvolvimento de cada componente do hardware, as plataformas de desenvolvimentos e a metodologia. O capítulo 4 são dispostos e discutidos os resultados do desenvolvimento do projeto. O capítulo 5 é dedicado às conclusões.

2 Revisão Teórica

Um processador é constituído por componentes físicos como registradores, *Arithmetic Logic Unit* (ALU), memórias de dados, memórias de instrução, barramento, multiplexadores, entre vários outros componentes. A forma como o processador aciona estes componentes é por meio de uma instrução. O conjunto de instruções que o processador pode realizar é chamado de *Instruction Set Architecture* (ISA). As instruções são colocadas dentro de uma parte específica da memória denominada memoria de instrução e o processador deve executar estas instruções sequencialmente.

2.1 RISC-V

Em primeiro lugar, *Reduced Instruction Set Computer* (RISC-V), proposto em [5], é um processador generalista, *open-source*, com ISA gerenciada por um comitê técnico constituído por membros do *RISC-V Foundation* com representação de grandes empresas da indústria de tecnologia.

Tradicionalmente, o desenvolvimento de novos processadores e com isso: conjuntos de instruções, têm financiamento das indústrias de SoC que por sua vez são impactadas pela decisões estratégicas do mercado, entre outros fatores que possivelmente influenciam no desenvolvimento tecnológico.

Esse tipo de interesse corporativista pode não ser saudável para o desenvolvimento tecnológico. Há diversos exemplos que interesses financeiros levaram as companhias desenvolver futuras versões de processadores com o mesmo ISA, porém com novas instruções, dessa forma, seria possível manter a compatibilidade de programas executados para versões anteriores do conjunto de instrução e melhorar, por meio da extensão do conjunto de instruções, o desempenho do que há por vir.

A perspectiva de desenvolvimento pode ser vantajosa do ponto de vista financeiro a curto prazo. Todavia, incrementar instruções e maneira inconsequente a longo prazo condena-se a arquitetura. O estudo [6] mostra que raramente instruções novas são utilizadas, e a melhoria de algo sempre será menor se comparado com o todo [7]. Portanto, implementa-se novas instruções para uma melhoria na eficiência muito discreta enquanto condena-se as novas arquiteturas a terem que utilizar este mesmo conjunto de instruções ampliado que não traz melhoria significante.

Em [8] discute o modo desenvolvimento de ISA por incremento ser análogo a um *buffet*, em que todo prato que alguém em algum momento chegou a pedir fará parte do *buffet*, e o consumidor será obrigado a pagar por todos os pratos, independente se irá ou não

consumir. O mesmo autor faz a analogia do RISC-V com um menu, em que o restaurante só será obrigado a produzir o que o consumidor pedir, e consequentemente, o consumidor só terá que pagar por aquilo que pedir.

Isso se dá por meio de um conjunto de instrução básico : **RV32I**, que constitui instruções obrigatórias para todas as implementações do RISC-V. Este conjunto de instrução será fixo e não se alterará com o tempo. Eventuais mudanças no conjunto de instruções serão deliberadas pelo comitê da *RISC-V Foundation* e as decisões serão tomadas apenas considerando os aspectos técnicos e serão, evidentemente, facultativas que são incluídas nas extensões do conjunto de instrução: **RV32M** adiciona instrução para multiplicação, **RV32F** para ponto flutuante.

Essa forma de atualizar instrução de um processador possibilita flexibilidade nas aplicações e em implementações físicas

2.2 Linguagem de Programação

A linguagem é uma abstração humana para que seja possível compreender duas partes distintas. Na língua portuguesa, essa abstração é muito elaborada, há formas objetivas e subjetivas de se comunicar com outro ser humano. No contexto de uma máquina que basicamente compreende somente lógica, a linguagem adotada deve ser muito mais rígida e sem possibilidade para dupla interpretação.

A intenção das linguagens de programação é fazer que o computador faça um determinado conjunto de instruções, o que é chamado de "programa"que podem ser desenvolvidos dos processos mais "baixo nível"que são aqueles que são constituídos por comandos do próprio processador, o conjunto ISA, até processos que são construídos em cima de várias instruções básicas e que são abstraídos ao humano por meio da linguagem "alto nível"como por exemplo o C/C++.

2.3 Compiladores

Compiladores são os responsáveis por abstrair trechos da linguagem de alto nível e transformar em um código Assembly que desempenha a tarefa de escrita em baixo nível.

IP/CV não é algo elementar de se desenvolver, as aplicações mais atuais são a consequência de vários anos de desenvolvimento de diferentes filtros de imagem e interpretação de dados que estão presente essencialmente em todas as aplicações de IP/CV por meio de uma camada de abstração mais baixo nível.

Não é razoável desenvolver este tipo de aplicação por meio de comandos do processador. O tempo de desenvolvimento seria inviável e a consequência são poucas aplicações.

É imprescindível ter um compilador para desenvolver aplicações mais sofisticadas que envolvem uma complexidade mais alta para estruturação de dados, com várias tomadas de decisões baseadas no legado das gerações passadas.

2.4 Assemblers

Assemblers são responsáveis por traduzir o código Assembly para a codificação da instrução de máquina que é desenvolvido junto à arquitetura do processador. Usualmente, para os processadores mais comuns como o próprio RISC-V, existem diversos tipos de instruções, que diferenciam no conteúdo das instruções e também na forma (posição em que cada informação está disposta na instrução).

Além disto, o Assembler é responsável por interpretar algo chamado de *linker script* em que é descrita a atribuição de cada posição da memória, por exemplo, trecho da memória dedicado a armazenar o conjunto de instruções a ser executado pelo processador, ou trecho da memória dedicado a salvar variáveis de programa alocadas durante a execução do algoritmo, os limites de memória que o SoC possui, entre outras finalidades importantes.

2.5 MPSoC

Multi Processor System-on-Chip (MPSoC) consiste em um sistema com mais de um processador em que cada um atua de forma independente do outro. É uma solução paradigmática para que seja possível desenvolver aplicações mais eficientes com a frequência de operação igual ou inferior a de processadores com problemas de limitação de energia.

Dessa forma, é comumente utilizado para solucionar problemas de eficiência no processamento por meio da distribuição de tarefas entre os diversos processadores, possibilitando a paralelização, de forma que, o sistema se torna mais eficiente quando são necessários menos ciclos de processamento para concluir uma determinada tarefa[9].

Estes sistemas também podem ser integrados naturalmente em aplicações específicas como no caso deste trabalho que é IP/CV em geral, explorando o que existe de paralelismo espacial no contexto de imagens, que são abstraídas pelos computadores como vetores com o índice escalonado pela posição do pixel em relação a linha ou coluna.

2.6 NoC

A comunicação intra-chip não é exatamente um assunto novo dentro dos tópicos de pesquisa. Já foram desenvolvidos para este propósito diversas formas de comunicação como: comunicação ponto a ponto, que pode ser algo viável para uma quantidade pequena de processadores na rede, todavia, é algo com a escalabilidade péssima que ocasiona gasto elevado

na produção do *Integrated Circuit* (IC), para que existam muitos canais de comunicação subutilizados.

Além de ponto a ponto existe a comunicação por meio de barramento, este tipo de comunicação possui um *Bus* compartilhado e conectado a todos elementos responsivos. Um exemplo é o protocolo desenvolvido pela *Phillips*, I2C, que funciona com apenas dois elementos utilizando o barramento ao mesmo tempo.

De forma geral, todas as formas de comunicação que se baseiam em rotear os fios entre um ponto a outro, no contexto de implementação para possibilitar maior eficiência, foram mal sucedidas devido a escalabilidade que ocasionaria um congestionamento de fios e levaria na existência de ruídos que inviabilizariam a solução [10]. A comunicação se tornou estado da arte quando foi proposto que fossem roteados pacotes de informação ao invés de fios condutores [11] constituindo a solução mais sofisticada para sistemas que pretendem atingir maior eficiência possível.

É evidente que quanto mais distribuído é um sistema, maior será o *overhead* de comunicação. De acordo com a lei de Ahmdal, menor será a melhora no desempenho, até que o tempo gasto na comunicação seja maior que o tempo gasto para o processamento não paralelizado, de forma que, o sistema ideal é fruto de um *trade-off* que pondera características como a eficiência, utilização de área do IC, consumo de energia, entre outros. Assim, diversas redes com tamanhos diferentes e com diferentes topologias são possíveis de serem adotadas.

A topologia não é a única variável que afeta significativamente a performance. Existem diversas formas de transmitir uma mensagem dentro da NoC que são significativas no desempenho na rede, por exemplo, explorando os recursos ociosos, outro exemplo é um bom mecanismo de arbitração que distribui de forma mais homogênea as mensagens ao longo da rede controlando o congestionamento de mensagens e ocasionando uma melhoria no desempenho da aplicação de forma geral.

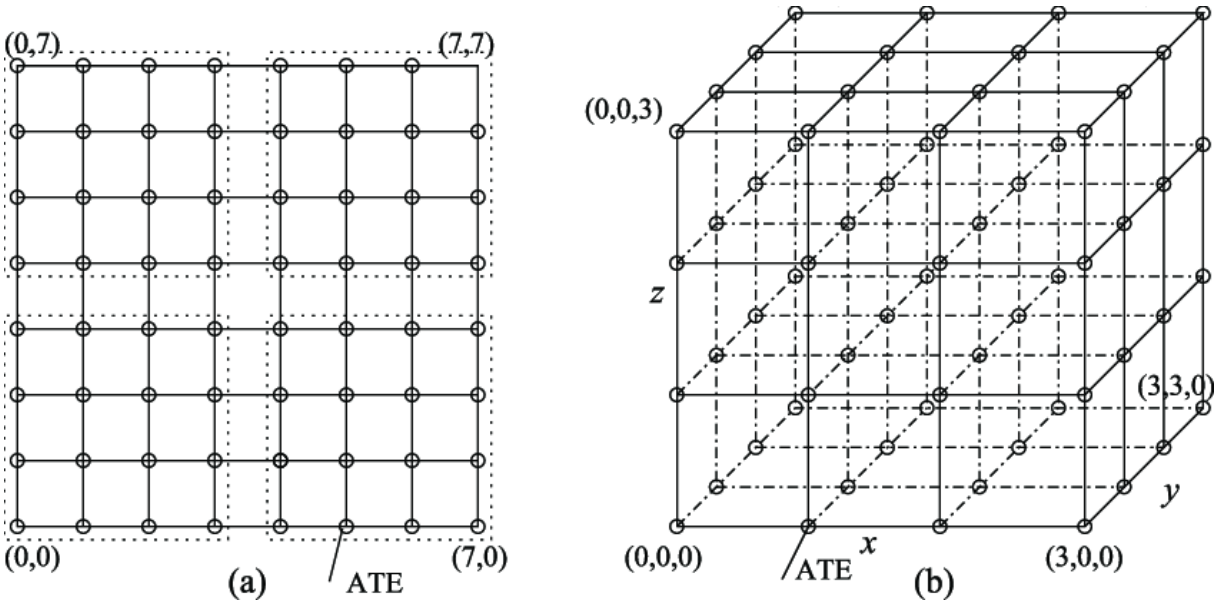
2.7 HDLs

Nas seções passadas foram discutidas linguagens de programação de forma geral e como estas linguagens podem ser traduzidas em instruções para o processador. Esta seção abordará um tipo diferente de linguagem para descrição de hardware.

Diferente das linguagens de software o que é feito com o código de uma implementação em HDL é uma síntese que gera uma sequência de bits, conhecida como *bitstream*, que configura fisicamente o hardware para aquilo que foi programado.

É possível utilizar HDLs para protótipos de duas formas: *Application-specific integrated circuit* (ASIC), que são circuitos integrados especialmente feitos para aquele propósito que não permitem reconfiguração. A Outra forma é por meio de *Field Programmable Gate*

Figura 2 – Diferentes topologias de redes intra-chip: a) topologia em 2D. b) topologia em 3D.
Fonte:[2]



Array (FPGA) que são estruturas programáveis que possibilitam que o programa em HDL seja fisicamente implementado por meio do *bitstream* sintetizado.

Existem duas principais HDLs, o Verilog, esta é uma linguagem menos tipada e mais amigável e com sintaxe semelhante a C/C++. E o VHDL que é fortemente tipada mais que não guarda semelhança com nenhuma outra linguagem. Neste projeto foram utilizadas ambas para o desenvolvimento.

3 Desenvolvimento

Este capítulo tem como objetivo realizar uma breve introdução sobre o ambiente de desenvolvimento, discriminando todos as ferramentas envolvidas e suas finalidades, relatar as etapas do desenvolvimento do trabalho e discutir acerca das decisões tomadas no desenvolvimento do projeto.

3.1 Ambiente de Desenvolvimento.

Neste projeto foi utilizado o Kit de desenvolvimento da *Xilinx ZCU104* que possui um SoC da família Zynq Ultrascale+[1] que possibilita o desenvolvimento em aplicações para processadores ARM e também possibilita o desenvolvimento em hardware com uma FPGA.

3.1.1 Parte Física

Este projeto pode ser dividido no contexto amplo em duas partes: No ARM-Host que é o processador embarcado na Zynq e a lógica programável na qual é implementado o MCVision-RISCV.

O desenvolvimento do MCVision é inteiramente realizado por implementação em FPGA utilizando linguagens de descrição de hardware como o VHDL e Verilog/SystemVerilog.

Outra característica muito importante de SoCs Zynq é a interface *Advanced eXtendable Interface* (AXI) que permite que registradores sintetizados em lógica programável sejam mapeados diretamente pelo ARM-Host possibilitando que o ARM-Host consiga expandir sua integração a mais periféricos do que serie possível utilizando apenas os GPIOs incluídos no sistema.

Para realizar a integração física destas partes, foi utilizado o software *Vivado 2019.1* que possui um ambiente de desenvolvimento capaz de realizar a síntese de HDLs, configuração do ARM-Host e também da interface AXI, permitindo customizações até o nível lógico.

O *Vivado* além disto possui uma biblioteca com IPs que contém utilidades como DMA, memórias, interfaces para protocolos como I2C, SPI, VGA e que podem ser utilizadas para auxiliar na implementação do projeto. Neste trabalho foram importantes as memórias, Interface AXI-lite e também a ILA, que é uma ferramenta para debug de sinais lógicos muito utilizada neste trabalho.

[1] Descrição do produto https://www.xilinx.com/products/boards-and-kits/zcu104.html

Como resultado, o *Vivado* gera um arquivo que descreve uma plataforma que compila uma série de informações como o *bitstream* para implementação física do sistema no FPGA, configuração do ARM-Host inclusive com o mapeamento de todos os endereços possíveis de serem acessados pelo ARM, isso incluindo os endereços para as entradas e saídas de elementos sintetizados em FPGA indicando quais são estes elementos.

3.1.2 Parte Lógica

Esta seção é dedicada a discutir acerca de como é feita a programação do ARM-Host, como o ARM-Host realiza a programação dos processadores RISC-V.

Para desenvolver uma aplicação em SoCs da família Zynq, a *Xilinx* oferece um ambiente de desenvolvimento *Xilinx SDK 2019.1* atualmente é chamado de *Vitis* para o desenvolvimento de aplicações programáveis pelo ARM-Host.

Este ambiente de desenvolvimento é integrado ao *Vivado*. A síntese da plataforma desenvolvida, feita pelo Vivado, é exportada para o *Vitis* que automaticamente gera APIs *Board Support Package* (BSP) que é uma camada de abstração feita em linguagem de alto nível C/C++, para o acesso fácil dos periféricos configurados na plataforma. Assim, o *Vitis* é utilizado para desenvolver a aplicação geral que controla o sistema de lógica programável.

Uma vez que a intenção é justamente ampliar as possibilidades de desenvolvimento desta arquitetura, o desenvolvimento das aplicações para o RISC-V é feita utilizando linguagem C.

Foi utilizado um compilador disponibilizado no sítio `https://godbolt.org/` que possui o compilador **rv32gc -gcc 12.1.0** que retorna o código do programa convertido para assembly.

Do assembly para código de máquina é utilizado o ambiente **RARS** `https://github.com/TheThirdOne/rars` que é um ambiente desenvolvido em Java e permite que o usuário configure um arquivo de configuração referente ao *linker script* e exporta o código em formato de texto.

Será visto posteriormente que o ARM-Host possui algumas interfaces com o MCVision-RISCV, uma delas é dedicada a escrever o código de máquina referente à programação da memória de programa do RISC-V pelo ARM-Host. A outra é utilizada para transferência de uma imagem de teste inicial. Há também interfaces para propósitos de *debug* e *profiling*.

3.2 Referencial teórico da arquitetura MCVision.

Já foi mencionado que este trabalho explora possíveis vantagens em embarcar um processador como o RISC-V especificamente para uma arquitetura já desenvolvida em

trabalhos anteriores. Para explicar a metodologia deste trabalho, é necessário explicar como é o funcionamento geral da arquitetura MCVision, este é o propósito desta seção.

No MCVision aspectos gerais da Arquitetura são descritos como parâmetros de forma a indicar ao sistema suas próprias características. Isso provê uma unificação de códigos de programa para desenvolvimento de aplicações, se tratando aqui de sistemas distribuídos.

Para tanto, é necessário realizar a sistematização de determinados parâmetros para que sejam atendidas necessidades abstraídas em requisitos. Observação: os parâmetros em si serão mais profundamente abordados nas seções 3.3.1 e 3.3.2 uma vez que a arquitetura MCVision-RISCV mantém todas as características destes parâmetros, e as alterações não afetam em nada a arquitetura MCVision.

Imagem

- Uma imagem precisa ser segmentada para que cada parte seja atribuída ao processamento de um *Tile*.

- Uma região designada a um determinado *Tile* só pode ser escrita por este mesmo Tile.

- Todos os *Tiles* devem ser capazes de demandar pixels de todos os *Tiles*, independente da localização.

NoC

- Todas as mensagens trafegadas na rede tem intenção de ler valores de pixel externo ao *Tile* que o requisitou.

- O *Tile* que envia a mensagem a outro *Tile* deve ser capaz de fornecer todas as informações necessárias para que a mensagem chegue ao *Tile* de destino e regresse ao *Tile* de origem com o valor de pixel desejado.

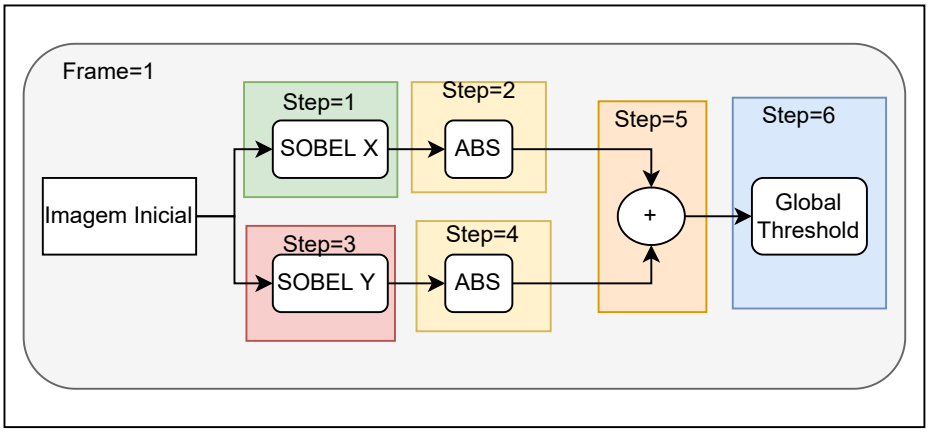
Memória de Pixel.

- Todos os *Tiles* devem ter uma memória que é independente do restante, capaz de ler informações da rede e retornar à sua origem com o valor de pixel requisitado, denominada de memória de pixel ou *Pixel Memory* (PM).

A topologia adotada no MCVision é bidimensional.

O MCVision tem a arquitetura capaz de fornecer as informações sempre que estiverem disponíveis para relacionar todas as etapas realizadas durante uma aplicação. É possível

Figura 3 – Ilustração do desenvolvimento de uma aplicação. Possível analisar a aplicação como a sequência de filtros que podem ser relacionados na memória de pixel por meio de coordenadas. **Fonte: Autor**



interpretar as etapas de aplicações IP/CV como filtros sendo executados e a aplicação seja algo que envolva uma sucessão destes filtros.

Com isso, é proposto a seguinte forma de organizar e estruturar as imagens

- São utilizadas 4 variáveis para o endereçamento de imagens, *(x,y,s,f)* em que S ou *step* indica qual é a etapa do processo, e *frame* indica de qual imagem.

$$\begin{aligned} Addr(x,y,s,f) =& (x - x\_init) * n\_frame * n\_step + \\ & image\_tile\_height * (y - y\_init) * n\_frame\ n\_step + \\ & f * s + f \end{aligned} \tag{3.1}$$

Estes requisitos aqui mencionados constituem a base para compreensão do funcionamento do sistema e a forma como isto é implementado será mostrado na seção seguinte.

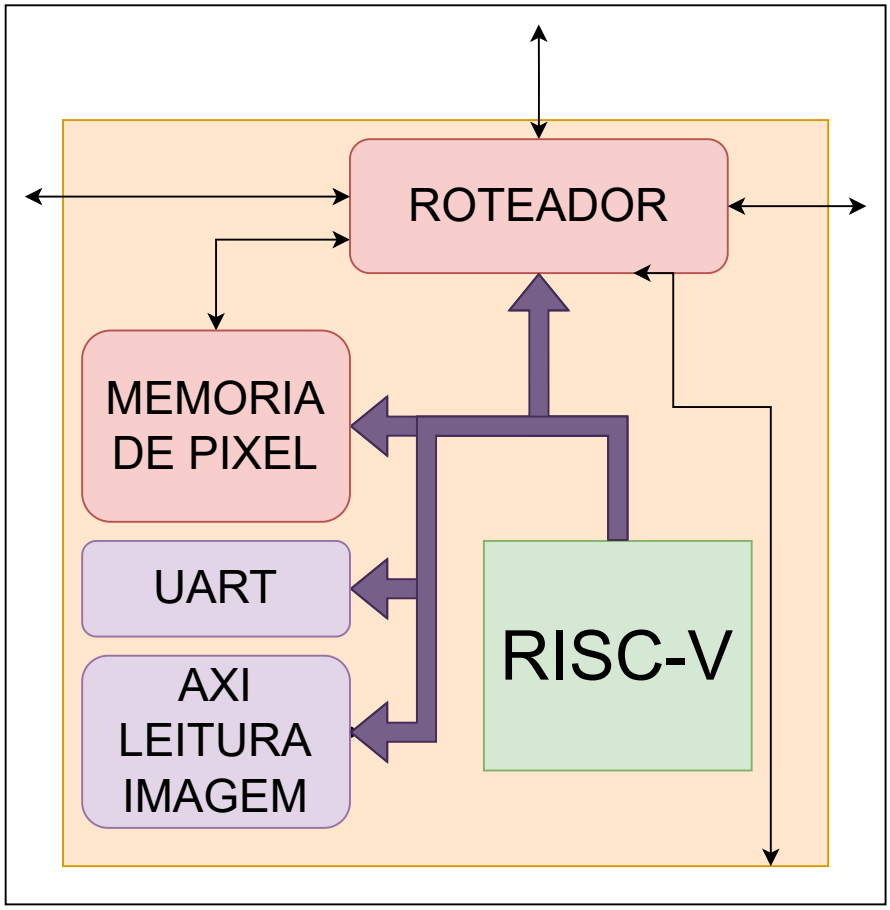
3.3 *Tile*

O *Tile* é essencialmente a junção e a interface de três outros elementos principais do MCVision, nomeadamente o elemento de processamento (RISC-V), memória de pixel (uma memória dedicada ao armazenamento de pixels) e o roteador.

3.3.1 Roteador

A NoC foi escolhida como estrutura de comunicação devido a sua escalabilidade. Em primeira análise, a comunicação poderia ter sido concebida muito bem por meio de conexões *point-to-point* (P2P) ou por barramento de dados quando se trata de um número limitado

Figura 4 – Esquema da constituição do *Tile*, o *Tile* pode ser visto como um SoC e cada periférico é ligado ao barramento do *Control Processing Unit* (CPU). **Fonte: Autor**



de *Tiles* presentes na rede. Isso é irrazoável para um MPSoC muito grande, a escalabilidade conferida pela NoC é importante e seus efeitos devem ser observados desde sua subutilização com *overhead* praticamente nulo até níveis de mais significativos de paralelização.

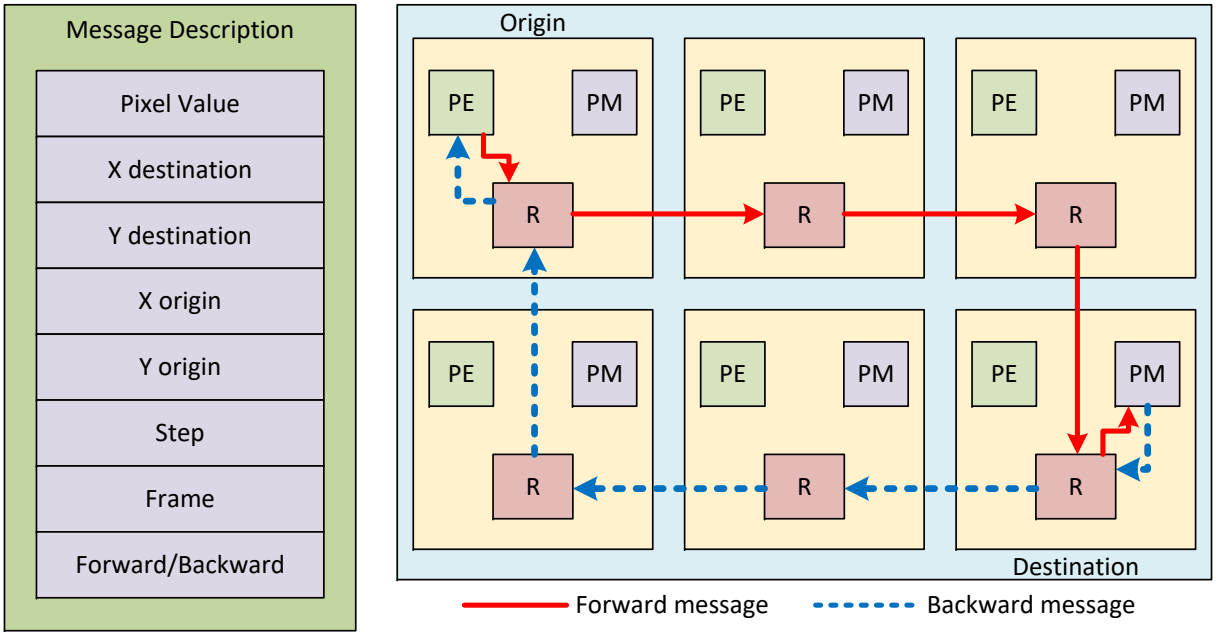
Uma NoC é essencialmente constituída por um roteador e o barramento para transmissão da mensagem (2 conexões internas para memória de pixel e para RISC-V e outras 4 conexões externas ao *Tiles*: Norte Sul Leste Oeste. A entidade do Roteador é responsável por identificar o destino a ser seguido e encaminhar para a saída adequada. Para que isso seja possível são necessárias uma série de informações a serem incluídas na mensagem.

Já foi dito que a rede prevista para o MCVision é bidimensional, e dessa forma, é possível utilizar parâmetros básicos como identificação do posicionamento por meio da indicação nos eixos das abcissas *(x)* e ordenadas *(y)*.

Portanto, uma mensagem deve conter a informação do seu destino utilizando as coordenadas do pixel requisitado além da indicação do *Tile* que requisitou a mensagem.

Uma vez que a mensagem chega ao *Tile* de destino e regressa à origem é necessário ter a informação que diz respeito se a mensagem está indo ao destino ou regressando a origem. Esta informação é feita por meio de um bit chamado de *Forward-Backward* (FB):

Figura 5 – Imagem descreve a dinâmica do fluxo da mensagens dentro da rede e as informações trazidas dentro da mensagem. Na figura temos um exemplo de uma requisição de pixel externo.**Fonte: [3]**



quando o nível lógico é 1, a mensagem deve estar regressando à origem, quando o nível lógico é 0 a mensagem está indo ao destino.

Também são necessárias informações de qual *step* e qual *frame* a mensagem se refere, essas informações foram discutidas ao final da seção 3.2.

A tabela abaixo descreve as informações contidas nas mensagens. O MCVision utiliza apenas o protocolo de transmissão 1 feito exclusivamente para leitura da memória de pixel.

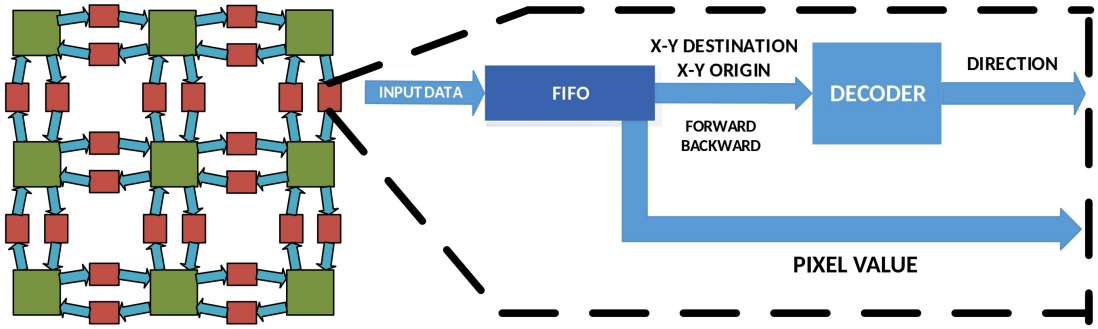
Tabela 1 – As mensagens que passam pelo roteador devem ter sempre o mesmo protocolo de transmissão, cada valor é designado a uma faixa de bits dentro desta palavra de 64 bits.

Pixel	x_dest	y_dest	step	frame	x_origin	y_origin	fb	handshake
63:49	48:41	40:32	31:27	26:19	18:11	10:3	[2:2]	[1:0]

O fluxo interno no roteador se inicia com o armazenamento da mensagem em um *Firt-In First-out* (FIFO) que pode ser parametrizado para sintetizar quantos buffers forem necessários. Esta é uma forma de amortecer os efeitos do congestionamento da rede que não precisará travar o seu funcionamento até que o roteador tenha a sua entrada disponível.

Em seguida a mensagem passa por um decodificar de região que possui um funcionamento baseado nas coordenadas *(x,y)* para direcionamento da mensagem. Primeiramente, é necessário saber se a mensagem está indo em direção à origem ou se está indo em direção ao destino. Como cada roteador compreende seu posicionamento dentro da rede é possível relacionar o destino da mensagem com o posicionamento local do roteador e designar a direção de saída apropriada.

Figura 6 – Esquema de acesso a memória de pixel. O acesso para registrar valor na memória é controlado exclusivamente pelo RISC-V e o processo de leitura é controlado exclusivamente pelo roteador.**Fonte:[3]**



A mensagem é designada a sua respectiva saída por meio de uma unidade responsável por controlar o fluxo de mensagens que tem prioridade para retirar mensagens da rede, fazendo que as mensagens com direções internas ao *Tile* (PM e RISC-V) sejam priorizadas. Esta característica pode não fazer efeito para aplicações que mandam as mensagens e esperam até que a mensagem regresse (quantidade de mensagens limitada ao dimensionamento da rede), mas para aplicações em que isso não ocorre, esta arbitração tende a ser um ponto positivo para redução do congestionamento na NoC.

3.3.2 Memória de Pixel

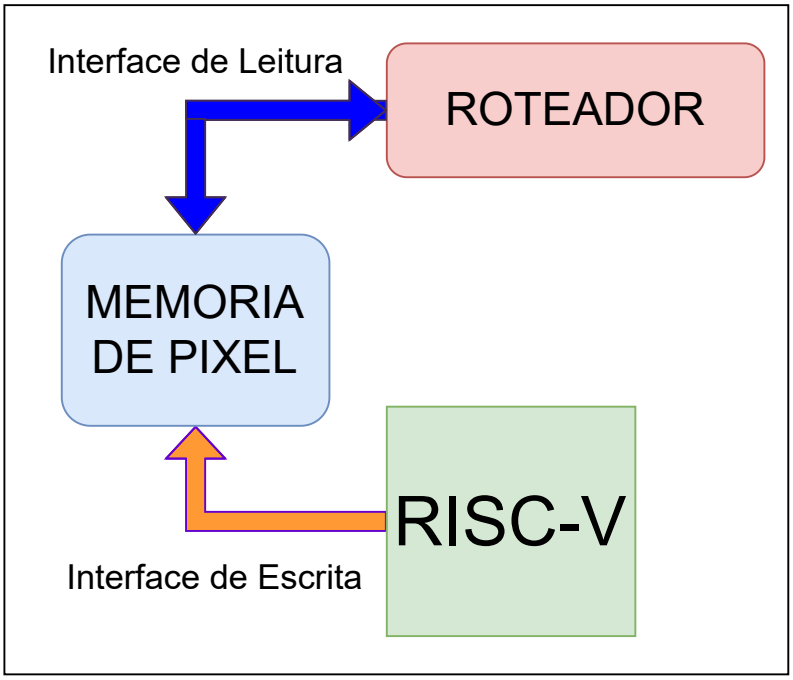
Cada *Tile* possui uma memória de pixel onde se encontram as imagens das regiões de processamento designadas ao respectivo *Tile*. A memória é implementada com os blocos de referência disponíveis no *Vivado*, na configuração *dual-port* em que somente uma entrada é dedicada a leitura e a outra dedicada a escrita.

Na arquitetura proposta, a porta de escrita é somente acessada pelo RISC-V e a porta da leitura é somente acessada via roteador.

As arquiteturas MCVision operam a partir do funcionamento independente do *Tile*, ademais, possibilitam que cada *Tile* possa requisitar informações de outros *Tiles*. Para que não seja necessário interromper o processamento, a memória de pixel é implementada separada do RISC-V e acessível pelo roteador. Dessa forma os *Tiles* com mais dados para processar não são sobrecarregados pelo fluxo de mensagens.

Todavia, sistemas paralelos podem ter algum problema na parte de sincronismo entre os *Tiles*, isto é esperado também para este projeto. Naturalmente, durante a execução de uma aplicação, um *Tile* ficará mais avançado que outro (geralmente *Tiles* centrais são os que precisam processar mais pixels e por isso demoram mais). Dessa forma, caso algum *Tile* receba um pedido de um determinado pixel que ainda não foi processado localmente, a memória de pixel retornará algum valor errado. Esse erro pode comprometer a qualidade da

Figura 7 – Ilustração do acesso para escrita e leitura a memória de pixel. **Fonte: Autor**



aplicação.

A solução aqui adotada para este problema é semelhante à adotada na arquitetura MCVision em que existe na comunicação, entre memória de pixel e o RISC-V, uma indicação de quais são as coordenadas *(x,y)* do *step (s)* referente a qual *frame (f)* que o RISC-V local está processando, caso a mensagem esteja requisitando um pixel que ainda não fora processado pelo *Tile* local, é feita uma retenção da mensagem até que o pixel esteja disponível para retornar.

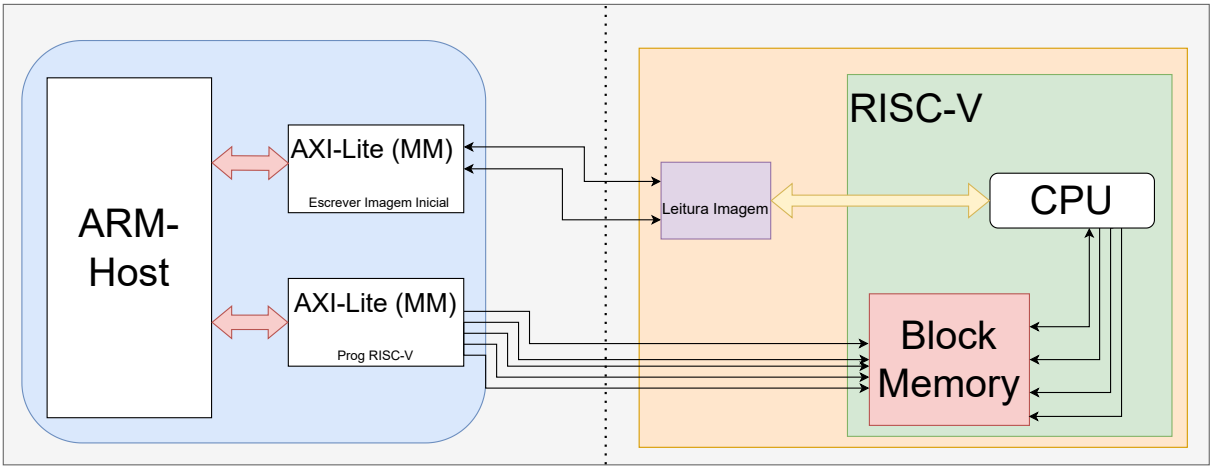
3.3.3 RISC-V, O Elemento de Processamento

Para o elemento de processamento, RISC-V utilizado foi `https://github.com/YosysHQ/picorv32` devido ao seu consumo de elementos lógicos reduzido. Ademais, possui diversas características configuráveis como as extensões RVIM, RVC, RCIMC, e também há disponível neste repositório um projeto de SoC básico que pode ser configurado para incluir uma interface *Universal Asynchronous Receiver Transmitter* (UART) e customizado para abranger mais entradas e saídas externas.

Tabela 2 – Esta tabela foi extraída do repositório `https://github.com/YosysHQ/picorv32` que mostra o impacto em lógica programável da escolha do conjunto de instrução no RISC-V.

Variante RISC-V	SLICE LUT	LUT as Memory	Slice Register
PicoRV32 (small)	761	48	442
PicoRV32 (regular)	917	48	583
PicoRV32 (large)	2019	8	1085

Figura 8 – Além das conexões feitas no *Tile*, esta figura descreve mais duas conexões. A primeira é a programação do RISC-V, que é feita exclusivamente pela interface AXI *Prog RISC-V*. A outra interface é utilizada para escrever na memória de pixel via acesso ao CPU. **Fonte: Autor**



Na figura 4 existe um barramento que sai do RISC-V e chega em todos os outros periféricos da placa. Esta ligação que contrasta com a ligação entre o roteador e outros *Tiles* de tal forma que é diferenciado o barramento por tratar de dados que são mapeados em memória dos que não são. Portanto todas as informações que o RISC-V lê ou escreve são mapeadas na sua memória.

Para ampliar o número de saídas do barramento de dados do RISC-V, foi necessário modificar o código do RISC-V em *Verilog* para atribuir endereços a cada uma das interfaces por meio de uma estrutura *elsif* que controla o acesso desse barramento. No Apêndice A está indicada a forma como foi implementada.

A mesma imagem 4 omite uma ligação interna do RISC-V que tem como objetivo efetuar a programação de sua memória interna (memória de programas e memória de dados) pelo ARM-Host via acesso à interface AXI-Lite com 4 registradores que assumem os valores de controle do bloco utilizado que é uma *Block Memory* (BRAM), *byte addressable*.

Quanto as saídas externas ao RISC-V, há uma interface dedicada a transferir a imagem inicial do ARM-Host para o MCVision. Isto é feito por meio de outra interface AXI-Lite que não passa pelo roteador e possui acesso exclusivo do RISC-V.

Tabela 3 – Barramento utilizado para acesso a configuração do conteúdo memória de pixel. Acesso exclusivo do RISC-V.

Pixel	X	Y	Handshake
[31:0]	[31:0]	[31:0]	[31:0]

Há outra saída não mencionada que é para propósito de desenvolvimento e não têm relevância para a explicação do projeto.

3.4 MCVision-RISCV

Esta parte é dedicada para avaliar os parâmetros do sistema de forma a discutir as tomadas de decisões no trabalho.

O protótipo implementado consiste em uma rede homogênea de microprocessadores com topologia 3x3. Com esta configuração é possível ter certeza da validação da imagem uma vez que o *Tile* central deve acionar todas as direções possíveis.

Os parâmetros descritos a seguir são inteiramente baseados na arquitetura MCVision. Todavia, para configuração do RISC-V são necessários parâmetros diferentes. Em 3.4.1, foram utilizados integralmente os parâmetros do MCVision.

3.4.1 Parâmetros da Imagem

Os primeiros parâmetros surgem de acordo com a imagem a ser processada, qual o limite de atuação de cada *Tile*, como é definido o início de atuação de cada *Tile*.

- Uma imagem é repartida de dentro do MCVision igualmente de acordo com o dimensionamento da rede.
- A largura da imagem repartida é dada pela quantidade de *Tiles* dispostos no eixo x, dividido pela largura total da imagem
- A altura da imagem repartida é dada pela quantidade de *Tiles* dispostos no eixo y, dividido pela altura total da imagem.
- Toda imagem tem inicio em X=0 e Y=0.

Na figura 9 é mostrada a relação da segmentação da imagem com a memória de pixel.

3.4.2 Parâmetros da Memória

O fator limitante para implementar MCVision-RISCV na ZCU104 é a quantidade de memória necessária demandada por uma aplicação pesada em RISC-V. Há duas memórias no *Tile*: a memória de pixel e a memória de programa.

A memória de pixel precisa atender aos requisitos de *(x,y,s,f)*. A memória de programa precisa do dimensionamento um tanto mais elaborado, uma vez que a alteração do seu tamanho desconfigura o sistema e o *linker script* deve ser atualizado.

Figura 9 – O bloco verde do diagrama está indicado uma memória de pixel, o jogo da velha dentro representa uma região da imagem e o seu endereçamento. As setas mostram a correspondência entre da imagem e o primeiro pixel do primeiro *Tile* e o primeiro pixel do *Tile* imediatamente abaixo. **Fonte: Autor**

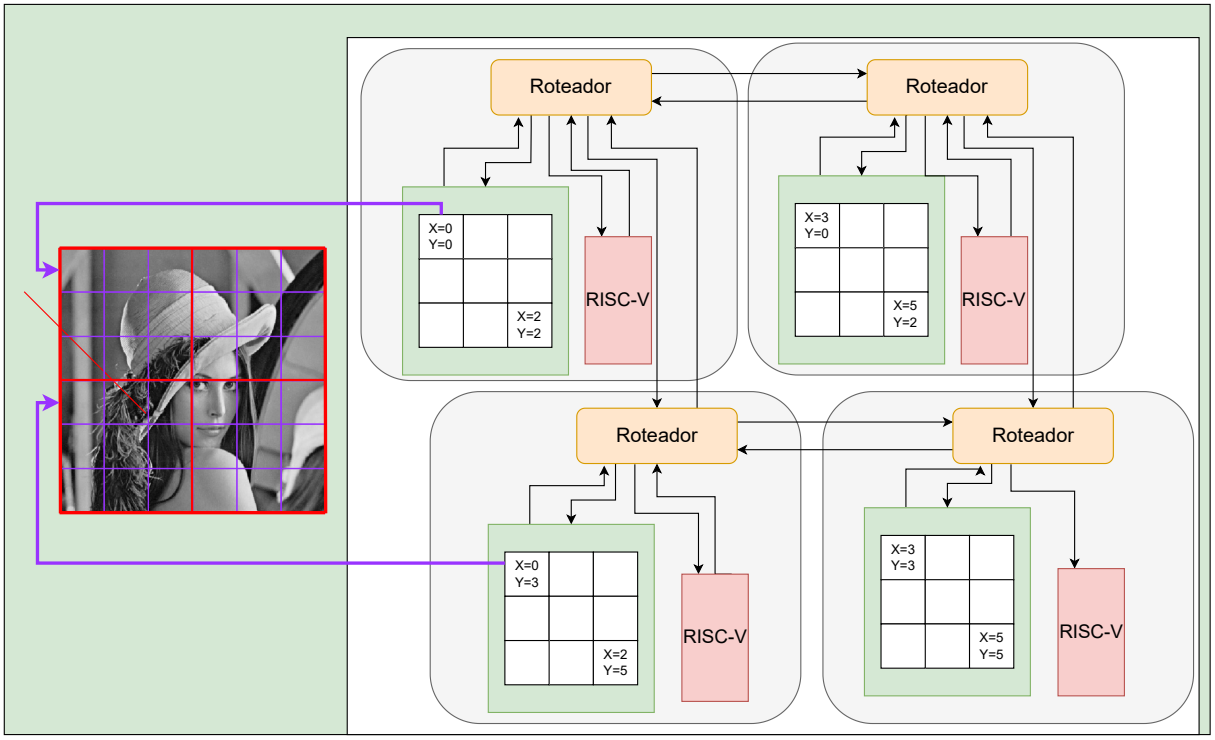


Tabela 4 – Esta é a configuração do *linker script* para a aplicação em RISC-V, o máximo de memória interna que o processador pode alcançar são 6144 palavras de 32 bits.

Seção	Endereço
stack base address	0x000059FF
stack limit address	0x00004800
heap base address	0x00004800
.data base address	0x00003500
.text limit address	0x000034FF
.text base address	0x00000000

Já a parte da memória de pixel, foi preciso atender com um vetor de um byte por elemento e com a quantidade de elementos dada por:

$$ADDR_{max} = x_{max}\, y_{max}\, s_{max}\, f_{max} \tag{3.2}$$

3.4.3 Programação

Um dos objetivos deste trabalho, possivelmente o principal, é abstrair a arquitetura do MCVision para ser utilizada como uma API em C/C++ com funções que incluem requisitar pixels *get_pixel(x,y,s,f)* e escrever pixels *set_pixel(pixel,x,y,s,f)*. Com isto, é possível

implementar em uma arquitetura especializada para aplicações de IP/CV, qualquer filtro de imagem para ser otimizado explorando o paralelismo espacial.

get_pixel(x,y,s,f) consiste em escrever na saída as coordenadas e esperar que alguma mensagem chegue no processador. Uma mensagem somente chegará no RISC-V caso este o tenha pedido, ou caso seja a etapa de transmissão de imagem.

Código 3.1 – Algumas funções que desempenham papeis importantes no MCVision-RISCV

```
1 void set_pixel(pixel,x,y,s,f) //configura o valor de pixel na PM
2 void print_image(s,f)  //Imprime a imagem inteira pela UART
3 void print_local_image(s,f)  //Imprime a  imagem inteira pela UART
4
5 void write_gpio(x_dest,x_dest,s,f,x_orig,y_orig,fb) // função de
     handshake e tratamento dos bits
6 void read_gpio(x_dest,x_dest,s,f,x_orig,y_orig,fb) // função de
     handshake e parsing de dados
```

Uma característica interessante é a existência de apenas um programa para todos os Tiles. Isso é devido a uma configuração interna que o *Tile* possui que permite registrar o valor dos parâmetros genéricos setados em HDL para que o software possa ler e configurar suas condições.

3.5 Transmissão e Aquisição de Imagens

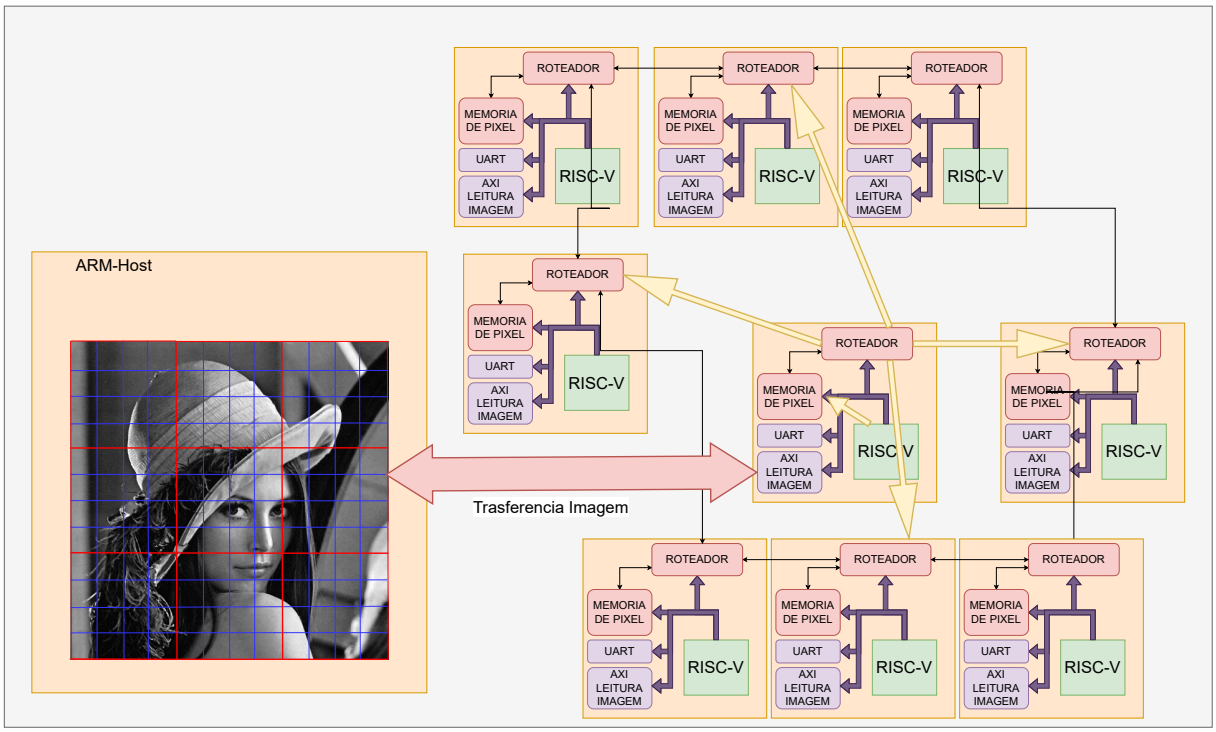
Para testar o MCVision-RISCV é necessário passar uma imagem para as memórias de cada *Tile*. Uma forma possível é multiplexar o canal de escrita para que possa funcionar ali também uma interface AXI comandada pelo ARM-Host. Esta é uma solução rápida que soluciona os problemas de testagem do sistema. Porém não é uma situação palpável, dificilmente será possível encontrar um sensor que possua a transmissão segmentada por áreas.

A partir de um *Tile* que seja geograficamente estratégico de forma a distribuir melhor os pixels, é proposto como forma de simulação de um sensor de imagem, utilizar a própria rede para distribuir uma imagem de teste, de forma que menos recursos da placa de desenvolvimento são gasto em simulação.

Na imagem 10 vemos uma rede 3x3 recebendo a imagem da Lena segmentada para cada *Tile*. O *Tile* central é o responsável por realizar a distribuição dos pixels para a rede que ocorre iniciando no canto superior esquerdo, avança até o final desta linha e segue para outra.

Essa transmissão é feita para simular os efeitos que podem causar problemas devido ao stream de pixels iniciais. Em várias ocasiões, as simulações são feitas já com a imagem completa nos buffers em que todos os pixels já estão disponíveis, neste caso, temos uma

Figura 10 – Transferência da Imagem **Fonte: Autor**



simulação mais fidedigna a realidade.

4 Resultados

A partir do estudo e caracterização dos parâmetros, as descrições dos padrões existentes em uma arquitetura multiprocessada, foi implementado o MCVision em uma topologia 3x3, com a memória de programa no tamanho de 6414 palavras de 32 bits e a memória de pixel estendendo até 4 *steps* para processamento.

Configuração da rede/característica da imagem

O dimensionamento da NoC e da imagem são duas variáveis que afetam uma a outra. Dependendo do dimensionamento da NoC, podemos ter diferentes valores de regiões para uma imagem igual, o mesmo é válido caso o contrário.

- Topologia adotada foi 3x3 (9 *Tiles*)
- Rede homogênea
- Imagem Utilizada é de 240x240 pixels, (cada *Tile* é designado a uma região com 80×80 pixels)

Configuração das memórias

- Memória de pixel permite região para o *Tile* de 80×80 pixels, para 4 *steps* de apenas 1 frame.
- Memória de programa é de 6414 palavras de 32 pixels.
- Tamanho do FIFO do roteador é de 3 mensagens.

Configuração do RISC-V

- RISC-V utilizado possui ativado a extensão RVIMC (a instrução de multiplicação é algo muito necessário uma vez que é necessário para diversas aplicações para IP/CV

Tabela 5 – Utilização dos recursos da ZCU104 Ultrascale+ para o MCVision 3x3

Recurso	Utilização	Disponíveis	Utilizados %
LUT	27521	230400	11.94
LUTRAM	502	101760	0.49
FF	35339	460800	7.67
BRAM	108	312	34.62
DSP	20	1728	1.16
IO	4	360	1.11
BUFG	11	544	2.02

Tabela 6 – Comparação entre os resultados de trabalhos anteriores.

	Este Trabalho	[12]
PL_max Frequency		126.04
LUT	27521	19543
LUTRAM	502	2604
FF	35339	16365
BRAM	108	167
URAM	0	9
DSP Blocks	20	36
Estimated Power	3.573	3.680

Sendo redundante, a tabela exibida em 5 é o mesmo MCVision com um CPU e barramento diferente e uma memória de pixel simplificada. A rede de comunicação se mantém a mesma que de outros.

Tabela 7 – Análise dos ciclos gastos para cada *Tile* receber uma imagem do ARM. As imagens são transmitidas pixel por pixel, do topo do canto esquerdo da imagem, para o topo do canto direito.

Tile	$\frac{t_{primeiro\_pixel\_trans}}{t_o}$	$\frac{t_{ultimo\_pixel\_trans}}{t_o}$	$\frac{t_{fim\_transmit}}{t_o}$
0	1.000171	1.733467	1.827779
1	1.003246	1.736564	1.839234
2	1.006343	1.73964	1.852894
3	1.739682	2.464832	2.627132
4	1.742756	2.46784	2.702161
5	1.745752	2.470916	2.716466
6	2.470959	3.204278	3.366618
7	2.474033	3.207376	3.365266
8	2.477131	3.210451	3.352751

- t_0 indica o tempo **(absoluto)** em que se inicia chamada da função de transferir imagem do ARM para MCVision
- $t_{primeiro\_pixel\_trans}$ é o tempo gasto **(absoluto)** para que o ARM transfira o primeiro pixel de um determinado *Tile*

- $t_{ultimo\_pixel\_trans}$ é o tempo gasto **(absoluto)** para que o ARM transfira o ultimo pixel de um determinado *Tile*
- $t_{fim\_transmit}$ tempo gasto **(absoluto)** para que o MCVision indique que o *Tile* já finalizou a escrita de todos os pixels enviados.

A tabela 7 indica o tempo gasto relativo para se transmitir a informação ao MVCvision. O parâmetro t_o indica quanto tempo/ciclos foram gastos até se chegar ao instante do começo do *profiling*.

Para este caso, em que se busca uma caracterização da arquitetura, não é interessante tratar de interpretar os resultados em dados absolutos, o interessante é notar nos dados relativos que existem degraus com um aumento significativo cada vez que a transmissão atinge os *Tiles* inferiores.

Portanto, quer dizer que o tempo gasto está aumentando de forma absoluta. Mas o congestionamento é maior na partir da segunda linha de *Tiles*.

Dessa forma, o interessante seria distribuir melhor a imagem para evitar que exista esse tipo de congestionamento, o que é claro, algo muito complicado de ser realizado tomando em conta esta estrutura, uma vez que não existe outro *Tile* sem ser o Central que desempenha este papel.

5 Conclusão

Neste trabalho é descrito o estudo, caracterização e implementação de um sistema multiprocessado especializado para aplicações que envolvem IP/CV denominado MCVision-RISCV. O trabalho foi baseado em [4] em que foi desenvolvida a mesma arquitetura MCVision, porém com um elemento de processamento reduzido a 16 instruções sem possibilidade de programação por linguagem de alto nível.

Este trabalho se propôs a desenvolver um sistema que seja mais democrático, que novas aplicações possam ser desenvolvidas utilizando linguagem C/C++ para a plataforma do MCVision, com isso expandindo as possibilidade de aplicações para diversas áreas que porventura possam se interessar pelo produto.

O MCVision-RISCV implementado constitui de uma rede de 9 *Tiles* 3x3 em que o elemento de processamento escolhido é o RISC-V de forma que possibilitar a integração de compiladores na framework do desenvolvimento. Estão incluídas interfaces para *debug* e transmissão de imagem inicial para o MCVision.

O resultado deste trabalho é a própria arquitetura capaz proporcionar a execução em paralelo de qualquer algoritmo implementável em C/C++.

Como este trabalho não envolve o desenvolvimento de aplicações para serem executadas no MCVision, não existem resultados de performance, futuros trabalhos podem ser feitos para explorar o estudo e caracterização do desenvolvimento de aplicações no MCVision.

Referências

[1] YUDI, J.; LLANOS, C. H.; HUEBNER, M. System-level design space identification for many-core vision processors. *Microprocessors and Microsystems*, Elsevier B.V., v. 52, p. 2–22, 7 2017. ISSN 01419331.

[2] XIANG, D.; SHEN, K. A new unicast-based multicast scheme for network-on-chip router and interconnect testing. *ACM Transactions on Design Automation of Electronic Systems*, v. 21, p. 1–23, 01 2016.

[3] SILVA, B. A. D.; LIMA, A. M.; YUDI, J. A manycore vision processor architecture for embedded applications. In: . [S.l.]: IEEE Computer Society, 2020. v. 2020-November. ISBN 9781728182865. ISSN 23247894.

[4] SILVA, B. A. da et al. A manycore vision processor for real-time smart cameras. *Sensors*, MDPI, v. 21, 11 2021. ISSN 14248220.

[5] WATERMAN, A. et al. The risc-v instruction set manual, volume i: User-level isa. *CS Division, EECE Department, University of California, Berkeley*, p. 28, 2014.

[6] KAMALELDIN, A.; HESHAM, S.; GOHRINGER, D. Towards a modular risc-v based many-core architecture for fpga accelerators. *IEEE Access*, v. 8, 2020. ISSN 21693536.

[7] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: . [S.l.: s.n.], 1967.

[8] DAVID, P. *In Praise of The RISC-V Reader*. 10 2017. Disponível em: <https://www.cs.sfu.ca/ ashriram/Courses/CS295/assets/books/rvbook.pdf>.

[9] SAPONARA, S.; FANUCCI, L.; PETRI, E. A multi-processor noc-based architecture for real-time image/video enhancement. *Journal of Real-Time Image Processing*, v. 8, 2013. ISSN 18618200.

[10] BENINI, L.; MICHELI, G. D. Networks on chips: A new soc paradigm. *Computer*, IEEE, v. 35, p. 70–78, 2002. ISSN 00189162.

[11] DALLY, W.; TOWLES, B. Route packets, not wires: on-chip interconnection networks. p. 684–689, 2002.

[12] SILVA, B. A. da; LIMA, A. M.; YUDI, J. A manycore vision processor architecture for embedded applications. In: *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*. [S.l.: s.n.], 2020. p. 1–8.

Apêndices

APÊNDICE A – Códigos

A.1 Programa para Inicialização da Plataforma

Código A.1 – Código de Inicialização do MCVision

```
#include <stdint.h>
#include <stdio.h>
//Este eh o unico codigo necessario programacao de todos os tiles do MCVision

// TILE_SETUP



// CONFIGURAçao da PM
#define PIXEL_SIZE 16
#define PM_LENGTH   9

#define reg_leds                    (*(volatile uint32_t*)0x03000000)

#define WIDTH  240
#define HEIGHT 240

//////////////////////////GENERICOS IRAO VIRAR REDE//////////////////////////
////////////////////////////////////////////////////////////////////////////
#define IMAGE_WIDTH                 (*(volatile uint32_t*)0x02310000)
#define IMAGE_HEIGHT                (*(volatile uint32_t*)0x02310004)
#define SUB_IMAG_WIDTH              (*(volatile uint32_t*)0x02310008)
#define SUB_IMAG_HEIGHT             (*(volatile uint32_t*)0x0231000C)
#define X_TILES                     (*(volatile uint32_t*)0x02310010)
#define Y_TILES                     (*(volatile uint32_t*)0x02310014)
#define X_LOCAL                     (*(volatile uint32_t*)0x02310018)
#define Y_LOCAL                     (*(volatile uint32_t*)0x0231001C)
#define X_INIT                      (*(volatile uint32_t*)0x02310020)
#define Y_INIT                      (*(volatile uint32_t*)0x02310024)
```

```
30
31 //////////////////INTERFACE DE TRANSMISSAO
      AXI//////////////////////////
32 ////////////////////////////////////////////////////////////////////////
33 #define AXI_IMAGE_PIXEL            (*(volatile
      uint32_t*)0x02300000)
34 #define AXI_IMAGE_X                (*(volatile
      uint32_t*)0x02300004)
35 #define AXI_IMAGE_Y                (*(volatile
      uint32_t*)0x02300008)
36 #define AXI_IMAGE_REQ              (*(volatile
      uint32_t*)0x0230000C)
37 #define AXI_IMAGE_ACK              (*(volatile
      uint32_t*)0x02300010)
38
39
40 ////////////////////////PROBES PARA
      PROFILING//////////////////////////
41 ////////////////////////////////////////////////////////////////////////
42 #define ENTRADA_INIT_PROG_FIM      (*(volatile
      uint32_t*)0x02300014)
43 #define SAIDA_INIT_PROG_FIM        (*(volatile
      uint32_t*)0x02300018)
44
45
46
47 ////////////////////////INTERFACE COM PM E
      ROUTER//////////////////////////
48 ////////////////////////////////////////////////////////////////////////
49 #define SET_PIXEL_BUFFER_TOP        (*(volatile
      uint32_t*)0x02230000)
50 #define SET_PIXEL_BUFFER            (*(volatile
      uint32_t*)0x02220000)
51 #define PE_WRITE_ADDRESS_TOP        (*(volatile
      uint32_t*)0x02210000)
52 #define PE_WRITE_ADDRESS            (*(volatile
      uint32_t*)0x02200000)
53 #define PE_READ_ADDRESS_TOP         (*(volatile
      uint32_t*)0x02110000)
54 #define PE_READ_ADDRESS             (*(volatile
      uint32_t*)0x02100000)
55
56
57 ////////////////////////CONFIGURACAO
      UART//////////////////////////////
58 ////////////////////////////////////////////////////////////////////////
59 #define reg_uart_clkdiv             (*(volatile
      uint32_t*)0x02000004)
60 #define reg_uart_data               (*(volatile
      uint32_t*)0x02000008)
61
62
```

```
63 // ASSIGNS DE PARAMETERS
64 /**/
65 #define X_TILESi                        (*(volatile
       uint32_t*)0x00006424)
66 #define Y_TILESi                        (*(volatile
       uint32_t*)0x00006420)
67 #define X_LOCALi                        (*(volatile
       uint32_t*)0x0000641C)
68 #define Y_LOCALi                        (*(volatile
       uint32_t*)0x00006418)
69 #define SUB_IMAG_WIDTHi                 (*(volatile
       uint32_t*)0x00006414)
70 #define SUB_IMAG_HEIGHTi                (*(volatile
       uint32_t*)0x00006410)
71 #define X_INITi                         (*(volatile
       uint32_t*)0x0000640C)
72 #define Y_INITi                         (*(volatile
       uint32_t*)0x00006408)
73 #define IMAGE_WIDTHi                    (*(volatile
       uint32_t*)0x00006404)
74 #define IMAGE_HEIGHTi                   (*(volatile
       uint32_t*)0x00006400)
75 #define write_buffer_top                (*(volatile
       uint32_t*)0x00006204)
76 #define write_buffer                    (*(volatile
       uint32_t*)0x00006200)
77
78 #define read_message_pixelValue         (*(volatile
       uint32_t*)0x00006100)
79 #define read_message_xDest              (*(volatile
       uint32_t*)0x00006104)
80 #define read_message_yDest              (*(volatile
       uint32_t*)0x00006108)
81 #define read_message_step               (*(volatile
       uint32_t*)0x0000610C)
82 #define read_message_frame              (*(volatile
       uint32_t*)0x00006110)
83 #define read_message_xOrig              (*(volatile
       uint32_t*)0x00006114)
84 #define read_message_yOrig              (*(volatile
       uint32_t*)0x00006118)
85 #define read_message_fb                 (*(volatile
       uint32_t*)0x0000611c)
86 #define read_message_req                (*(volatile
       uint32_t*)0x00006120)
87 #define read_message_ack                (*(volatile
       uint32_t*)0x00006124)
88
89
90
91
```

```
92  /////////MASCARAS PARA BIT HANDLING DE ESCRITA E
        LEITURA///////////////
93  ////////////////////////////////////////////////////////////////////////
94  #define SET_PM_PIXEL_MASK           0x0000FFFF
95  #define SET_PM_X_DEST_MASK          0x7F800000
96  #define SET_PM_Y_DEST_MASK          0x007F8000
97  #define SET_PM_STEP                 0x00007C00
98  #define SET_PM_FRAME                0x000003FC
99
100 #define SET_ADDRESS_MASK            0x000003FE
101 #define SET_REQ_MASK                0x00000001
102
103 #define PE_WRITE_ADDRESS_MASK_PIXEL 0xFFFF0000
104 #define PE_WRITE_ADDRESS_MASK_XDEST 255UL<<8
105 #define PE_WRITE_ADDRESS_MASK_YDEST 255UL<<0
106 #define PE_WRITE_ADDRESS_MASK_STEP  31UL<<27
107 #define PE_WRITE_ADDRESS_MASK_FRAME 255UL<<19
108 #define PE_WRITE_ADDRESS_MASK_XORIG 255UL<<11
109 #define PE_WRITE_ADDRESS_MASK_YORIG 255UL<<3
110 #define PE_WRITE_ADDRESS_MASK_FB    1UL<<2
111 #define PE_WRITE_ADDRESS_MASK_REQ   1UL<<1
112 #define PE_WRITE_ADDRESS_MASK_REQ   1UL<<1
113 #define PE_WRITE_ADDRESS_MASK_ACK   1UL
114
115
116 void setPixel(uint32_t pixel_Value);
117 void setXdest(uint32_t x_Dest);
118 void setYdest(uint32_t y_Dest);
119 void setStep(uint32_t step);
120 void setFrame(uint32_t frame);
121 void setXorig(uint32_t x_Orig);
122 void setYorig(uint32_t y_Orig);
123 void setFb(uint32_t fb);
124
125 void set_pm_pixel(uint32_t pixel);
126 void set_pm_x_dest(uint32_t x_dest);
127 void set_pm_y_dest(uint32_t y_dest);
128 void set_pm_step(uint32_t step);
129 void set_pm_frame(uint32_t frame);
130
131 void readPixel(void);
132 void readXdest(void);
133 void readYdest(void);
134 void readStep(void);
135 void readFrame(void);
136 void readXorig(void);
137 void readYorig(void);
138 void readFb(void);
139 void readReq(void);
140 void readAck(void);
141
142
```

```
143 void read_gpio(void);
144 void write_gpio(uint32_t pixel, uint32_t x_dest, uint32_t y_dest,
       uint32_t step, uint32_t frame, uint32_t x_orig, uint32_t
       y_orig, uint32_t fb);
145 void set_pixel(uint32_t pixel, uint32_t x_dest, uint32_t y_dest,
       uint32_t step, uint32_t frame);
146
147 void Read_message(void);
148
149
150 void putchart(char c);
151 void print(const char *p);
152 void print_dec(uint32_t v);
153
154
155 //utilidades
156 void delay(void);
157 void delay2(uint16_t k);
158
159
160
161 ////testes
162 void set_local_image(void);
163 void get_local_image(void);
164 void get_external_image(uint8_t tilex, uint8_t tiley);
165 void get_external_image_noprint(uint8_t tilex, uint8_t tiley);
166 void get_image(void);
167
168 void teste_local_image(void); //seta e le a pm interna
169 void teste_leitura_imagem(void);
170 void teste_com_entre_risc_mestre(void);
171 void teste_com_entre_risc_escravo(void);
172 void teste_parameters(void);
173
174 //func para a recepcao da imagem do ARM
175 void distribute_image_from_zynq(void);
176 int check_last_pixel(uint32_t x,uint32_t y);
177 int check_local_pixel(uint32_t x,uint32_t y);
178 void wait_for_checkmsg(void);
179
180 //func para a recepcao da imagem do ARM
181 void get_first_image_recv(void);
182 void send_transfer_finished_notice_recv(void);
183 void get_and_set_local_image_from_zynq_recv(void);
184 void get_and_set_local_pixel_from_zynq_recv(void);
185
186 void core_init(void);
187
188 int abs_paralelo(int numero);
189 uint32_t get_pixel(uint8_t x, uint8_t y, uint8_t s, uint8_t f);
190 void sobel_x(void);
191
```

```
192
193 int p_sobel_x[3][3]={{-1,0,1},{-2,0,2},{-1,0,1}};
194
195 void main()
196
197 {
198
199      reg_uart_clkdiv=868;
200
201      IMAGE_WIDTHi=IMAGE_WIDTH;
202      IMAGE_WIDTHi=IMAGE_WIDTH;
203      Y_INITi=Y_INIT;
204      X_INITi=X_INIT;
205      Y_LOCALi=Y_LOCAL;
206      X_LOCALi=X_LOCAL;
207      SUB_IMAG_HEIGHTi=SUB_IMAG_HEIGHT;
208      SUB_IMAG_WIDTHi=SUB_IMAG_WIDTH;
209
210         print("|");
211         print("|");
212         print(" ");
213         print("  ");
214         print("   ");
215         print("\r\n");
216         print("\r\n\n\r");
217           print_dec(221);
218         print("\n");
219
220         core_init();
221         while(1);
222
223
224
225 }
226
227 void core_init(void)
228 {
229         if(X_LOCALi==1){
230             if(Y_LOCALi==1){
231                 distribute_image_from_zynq();
232                 wait_for_checkmsg();
233                 delay();
234                 delay();
235                 delay();
236                 sobel_x();
237                 get_image();
238             }
239             else
240             {
241                 get_first_image_recv();
242             }
243         }
```

```
244         else
245         {
246             get_first_image_recv();
247 
248         }
249 
250 }
251 /*************************************************************************
252 ************************FUNCOES PARA ESCREVER NA PM LOCAL*****************************
253 *************************************************************************
254 void set_pixel(uint32_t pixel, uint32_t x_dest, uint32_t y_dest, uint32_t step, uint32_t frame)
255 {
256 // resetar o buff de envio
257 SET_PIXEL_BUFFER=0ULL;
258 SET_PIXEL_BUFFER_TOP=0;
259 
260 
261     set_pm_pixel(pixel);
262     set_pm_x_dest(x_dest);
263     set_pm_y_dest(y_dest);
264     set_pm_step(step);
265     set_pm_frame(frame);
266 
267 //setar o request
268 SET_PIXEL_BUFFER= SET_PIXEL_BUFFER | 0x2;
269 //esperar pelo acknowledge do recebimento
270 while((SET_PIXEL_BUFFER & 0x1)== 0);
271 
272 //reseta o pino de request
273 SET_PIXEL_BUFFER= SET_PIXEL_BUFFER & (~0x00000002) ;
274 //esperar pelo reset do acknowledge
275 while((SET_PIXEL_BUFFER & 0x1)== 1);
276 
277 SET_PIXEL_BUFFER=0ULL;
278 SET_PIXEL_BUFFER_TOP=0;
279 }
280 
281 void set_pm_pixel(uint32_t pixel)
282 {
283     SET_PIXEL_BUFFER_TOP  |=   (SET_PIXEL_BUFFER_TOP & ~SET_PM_PIXEL_MASK) | ((uint32_t)pixel);
284 }
285 
286 void set_pm_x_dest(uint32_t x_dest)
287 {
288   SET_PIXEL_BUFFER  |=   (SET_PIXEL_BUFFER & ~SET_PM_X_DEST_MASK) | ((uint32_t)x_dest<<23);
289 }
290 void set_pm_y_dest(uint32_t y_dest)
291 {
```

```
292   SET_PIXEL_BUFFER  |=   (SET_PIXEL_BUFFER & ~SET_PM_Y_DEST_MASK)
         | ((uint32_t)y_dest<<15);
293 }
294 void set_pm_step(uint32_t step)
295 {
296   SET_PIXEL_BUFFER  |=   (SET_PIXEL_BUFFER & ~SET_PM_STEP) |
         ((uint32_t)step<<10);
297 }
298 void set_pm_frame(uint32_t frame)
299 {
300   SET_PIXEL_BUFFER  |=   (SET_PIXEL_BUFFER & ~SET_PM_FRAME) |
         ((uint32_t)frame<<2);
301 }
302
303
304 /*************************************************************************
305 **************************FIM DAS FUNCOES PARA ESCREVER NA PM
       LOCAL*******************
306 *************************************************************************
307
308
309
310
311
312
313 /*************************************************************************
314 ************************************************FUNCOES
       PARA ESCREVER NA
       REDE*************************************************
315 *************************************************************************
316
317 void write_gpio(uint32_t pixel, uint32_t x_dest, uint32_t y_dest,
       uint32_t step, uint32_t frame, uint32_t x_orig, uint32_t
       y_orig, uint32_t fb)
318 {
319 // resetar o buff de envio
320 PE_WRITE_ADDRESS=0ULL;
321
322     PE_WRITE_ADDRESS=0;
323     PE_WRITE_ADDRESS_TOP=0;
324     write_buffer_top=0;
325     write_buffer=0;
326
327     setPixel(pixel);
328     setXdest(x_dest);
329     setYdest(y_dest);
330     setFrame(step);
331     setStep (frame);
332     setXorig(x_orig);
333     setYorig(y_orig);
334     setFb   (fb);
335     PE_WRITE_ADDRESS_TOP= write_buffer_top;
```

```
336     PE_WRITE_ADDRESS= write_buffer;
337
338
339 //setar o request
340 PE_WRITE_ADDRESS= PE_WRITE_ADDRESS | 0x2;
341
342 //esperar pelo acknowledge do recebimento
343 while((PE_WRITE_ADDRESS & 0x1)== 0);
344
345 //reseta o pino de request
346 PE_WRITE_ADDRESS= PE_WRITE_ADDRESS & (~0x00000002) ;
347
348 //esperar pelo reset do acknowledge
349 while((PE_WRITE_ADDRESS & 0x1)== 1);
350
351 }
352
353
354 void setPixel(uint32_t pixel_Value)
355 {
356     write_buffer_top |=   (write_buffer_top &
        ~PE_WRITE_ADDRESS_MASK_PIXEL) | ((uint32_t)pixel_Value<<16);
357 }
358 void setXdest(uint32_t x_Dest)
359 {
360     write_buffer_top |=   (write_buffer_top &
        ~PE_WRITE_ADDRESS_MASK_XDEST) | ((uint32_t)x_Dest<<8);
361 }
362 void setYdest(uint32_t y_Dest)
363 {
364     write_buffer_top |=   (write_buffer_top &
        ~PE_WRITE_ADDRESS_MASK_YDEST) | ((uint32_t)y_Dest);
365 }
366
367 void setStep(uint32_t step)
368 {
369       write_buffer |=   (write_buffer &
          ~PE_WRITE_ADDRESS_MASK_STEP) | ((uint32_t)step<<27);
370 }
371 void setFrame(uint32_t frame)
372 {
373       write_buffer |=   (write_buffer &
          ~PE_WRITE_ADDRESS_MASK_FRAME) | ((uint32_t)frame<<19);
374 }
375
376 void setXorig(uint32_t x_Orig)
377 {
378       write_buffer |=   (write_buffer &
          ~PE_WRITE_ADDRESS_MASK_XORIG) | ((uint32_t)x_Orig<<11);
379 }
380 void setYorig(uint32_t y_Orig)
381 {
```

```
382         write_buffer |=   (write_buffer &
               ~PE_WRITE_ADDRESS_MASK_YORIG) | ((uint32_t)y_Orig<<3);
383 }
384 void setFb(uint32_t fb)
385 {
386         write_buffer |=   (write_buffer & ~PE_WRITE_ADDRESS_MASK_FB)
               | ((uint32_t)fb<<2);
387 }
388
389
390
391 /*************************************************************************
392 **************************FIM DAS FUNCOES PARA ESCREVER NA
       REDE**************************
393 *************************************************************************
394
395
396
397
398
399
400
401
402
403
404
405 /*************************************************************************
406 **************************FUNCOES PARA LER NA
       REDE**************************
407 *************************************************************************
408 void read_gpio(void)
409 {
410
411     // espera algum sinal de request
412     while((PE_READ_ADDRESS & 0x2)==0);
413
414     // recebe a mensagem
415     readPixel();
416     readXdest();
417     readYdest();
418     readStep ();
419     readFrame();
420     readXorig();
421     readYorig();
422     readFb();    // seta o acknowledge
423     PE_READ_ADDRESS= PE_READ_ADDRESS | 0x1;
424
425     // espera o reset do request
426     while((PE_READ_ADDRESS & 0x2)==1);
427
428     // reseta o pino do acknowledge
429     PE_READ_ADDRESS= PE_READ_ADDRESS & (~0x1);
```

```
430 }
431
432 /*
433  "1 1111 1111"  "1111 1111"  "1111 1111"  "1111 1111"  "1111 1111"
       "1111 1111"  "1111 1111"   "1"    "1"  |  "1"
434     Pixel          Xdest        Ydest        Step         Frame
                Xorig        Yorig       fb    Req  |  Ack
435
436                              ESCRITA

                                    | LEITURA
437 */
438
439 void readPixel(void)
440 {
441
442     read_message_pixelValue= ((PE_READ_ADDRESS_TOP) &
          (PE_WRITE_ADDRESS_MASK_PIXEL))>>16;
443
444 }
445 void readXdest(void)
446 {
447     read_message_xDest= ((PE_READ_ADDRESS_TOP) &
          (PE_WRITE_ADDRESS_MASK_XDEST))>>8;
448 }
449 void readYdest(void)
450 {
451    read_message_yDest= ((PE_READ_ADDRESS_TOP) &
          (PE_WRITE_ADDRESS_MASK_YDEST));
452 }
453
454 void readStep(void)
455 {
456    read_message_step= ((PE_READ_ADDRESS) &
          (PE_WRITE_ADDRESS_MASK_STEP))>>27;
457 }
458 void readFrame(void)
459 {
460   read_message_frame=((PE_READ_ADDRESS) &
          (PE_WRITE_ADDRESS_MASK_FRAME))>>19;
461 }
462 void readXorig(void)
463 {
464    read_message_xOrig= ((PE_READ_ADDRESS) &
          (PE_WRITE_ADDRESS_MASK_XORIG))>>11;
465 }
466 void readYorig(void)
467 {
468   read_message_yOrig =((PE_READ_ADDRESS) &
          (PE_WRITE_ADDRESS_MASK_YORIG))>>3;
469 }
470 void readFb(void)
```

```
471 {
472   read_message_fb=((PE_READ_ADDRESS) &
        (PE_WRITE_ADDRESS_MASK_FB))>>2;
473 }
474 void readReq(void)
475 {
476   read_message_req=((PE_READ_ADDRESS) &
        (PE_WRITE_ADDRESS_MASK_REQ))>>1;
477 }
478 void readAck(void)
479 {
480   read_message_ack=((PE_READ_ADDRESS) &
        (PE_WRITE_ADDRESS_MASK_ACK))>>0;
481 }
482
483
484 /*****************************************************************
485 **************************FUNCOES PARA LER NA
      REDE**************************
486 *****************************************************************
487
488
489
490
491
492 /*****************************************************************
493 **************************FUNCOES PARA LER ARM SET IMAGEM
      INICIAL**********************
494 *****************************************************************
495
496 void putchart(char c)
497 {
498   if (c == '\n')
499     putchart('\r');
500   reg_uart_data = c;
501 }
502
503 void print(const char *p)
504 {
505   while (*p)
506     putchart(*(p++));
507 }
508
509 void print_dec(uint32_t v)
510 {
511
512   if       (v >= 900) { putchart('9'); v -= 900; }
513   else if (v >= 800) { putchart('8'); v -= 800; }
514   else if (v >= 700) { putchart('7'); v -= 700; }
515   else if (v >= 600) { putchart('6'); v -= 600; }
516   else if (v >= 500) { putchart('5'); v -= 500; }
517   else if (v >= 400) { putchart('4'); v -= 400; }
```

```
518   else if (v >= 300) { putchart('3'); v -= 300; }
519   else if (v >= 200) { putchart('2'); v -= 200; }
520   else if (v >= 100) { putchart('1'); v -= 100; }
521   else putchart('0');
522
523   if      (v >= 90) { putchart('9'); v -= 90; }
524   else if (v >= 80) { putchart('8'); v -= 80; }
525   else if (v >= 70) { putchart('7'); v -= 70; }
526   else if (v >= 60) { putchart('6'); v -= 60; }
527   else if (v >= 50) { putchart('5'); v -= 50; }
528   else if (v >= 40) { putchart('4'); v -= 40; }
529   else if (v >= 30) { putchart('3'); v -= 30; }
530   else if (v >= 20) { putchart('2'); v -= 20; }
531   else if (v >= 10) { putchart('1'); v -= 10; }
532   else putchart('0');
533
534   if      (v >= 9) { putchart('9'); v -= 9; }
535   else if (v >= 8) { putchart('8'); v -= 8; }
536   else if (v >= 7) { putchart('7'); v -= 7; }
537   else if (v >= 6) { putchart('6'); v -= 6; }
538   else if (v >= 5) { putchart('5'); v -= 5; }
539   else if (v >= 4) { putchart('4'); v -= 4; }
540   else if (v >= 3) { putchart('3'); v -= 3; }
541   else if (v >= 2) { putchart('2'); v -= 2; }
542   else if (v >= 1) { putchart('1'); v -= 1; }
543   else putchart('0');
544 }
545
546
547
548 ////////////////////////////////////////////////
549 //TESTES E APLICACOES//////////////////////////
550
551 void teste_com_entre_risc_mestre(void)
552 {
553 volatile uint32_t pixel;
554     for(volatile uint32_t i = 0; i < IMAGE_HEIGHT; i++)
555     {
556         for(volatile uint32_t j = 0; j < IMAGE_WIDTH; j++)
557         {
558         pixel=j+20*i;
559         write_gpio(pixel,0,10,0,0,X_INIT,Y_INIT,1);
560         read_gpio();
561         print_dec(read_message_pixelValue);
562         print("\n");
563
564
565         }
566
567     }
568 }
569
```

```
570 void teste_com_entre_risc_escravo(void)
571 {
572     for(volatile uint32_t i = Y_INIT; i < IMAGE_HEIGHT; i++)
573     {
574         for(volatile uint32_t j = X_INIT; j < IMAGE_WIDTH; j++)
575         {
576         read_gpio();
577         write_gpio(read_message_pixelValue,0,0,0,0,X_INIT,Y_INIT,1);
578         }
579     }
580 }
581
582
583
584 void set_local_image(void)
585 {
586  volatile int pixel;
587     for(volatile int i =Y_INITi ; i < SUB_IMAG_HEIGHTi+ Y_INITi;
        i++)
588     {
589         for(volatile int j = X_INITi; j < SUB_IMAG_WIDTHi+
            X_INITi; j++)
590         {
591         pixel=j+WIDTH*i;
592         set_pixel(pixel,j, i, 0, 0);
593         }
594     }
595 }
596
597 void get_local_image(void)
598 {
599  volatile int pixel;
600     for(volatile int i = Y_INITi; i < SUB_IMAG_HEIGHTi+ Y_INITi;
        i++)
601     {
602         for(int j = X_INITi; j < SUB_IMAG_WIDTH+ X_INITi; j++)
603         {
604         write_gpio(0,j,i,0,0,X_INIT,Y_INIT,0);
605         read_gpio();
606         print_dec(read_message_pixelValue);
607         }
608
609     }
610 }
611 void get_image(void)
612 {
613 volatile uint32_t pixel;
614     for(volatile uint32_t i = 0; i < 240; i++)
615     {
616         for(volatile uint32_t j = 0; j < 240; j++)
617         {
618         write_gpio(0,j,i,0,0,X_INIT,Y_INIT,0);
```

```
619         read_gpio();
620         print_dec(read_message_pixelValue);
621         print("\n");
622         }
623
624     }
625 }
626 void get_external_image(uint8_t tilex, uint8_t tiley)
627 {
628     for(volatile uint32_t i = (tiley*SUB_IMAG_HEIGHT); i <
        ((tiley+1)*SUB_IMAG_HEIGHT); i++)
629     {
630         for(volatile uint32_t j = (tilex*SUB_IMAG_WIDTH); j <
            ((tilex+1)*SUB_IMAG_WIDTH); j++)
631         {
632         write_gpio(0,j,i,0,0,X_INIT,Y_INIT,0);
633         read_gpio();
634         print_dec(read_message_pixelValue);
635         }
636
637     }
638 }
639
640 void get_external_image_noprint(uint8_t tilex, uint8_t tiley)
641 {
642     for(volatile uint32_t i = (tiley*SUB_IMAG_HEIGHT); i <
        ((tiley+1)*SUB_IMAG_HEIGHT); i++)
643     {
644         for(volatile uint32_t j = (tilex*SUB_IMAG_WIDTH); j <
            ((tilex+1)*SUB_IMAG_WIDTH); j++)
645         {
646         write_gpio(0,j,i,0,0,X_INIT,Y_INIT,0);
647         read_gpio();
648         }
649
650     }
651 }
652 void teste_local_image(void)
653 {
654     set_local_image();
655     get_local_image();
656 }
657
658
659 void teste_leitura_imagem(void)
660 {
661         for(volatile uint32_t i =0; i < X_TILES; i++)
662             for(volatile uint32_t j = 0; j<Y_TILES; j++)
663         {
664             {
665              get_external_image( i,j );
666             }
```

```
667         }
668
669 }
670
671 void delay(void)
672 {
673         for(volatile uint64_t ko = 0; ko < 20000; ko++);
674 }
675
676
677 void teste_parameters(void)
678 {
679
680 print_dec(IMAGE_WIDTH);
681         print("\r\n\n\r");
682 print_dec(IMAGE_HEIGHT);
683         print("\r\n\n\r");
684 print_dec(SUB_IMAG_WIDTH);
685         print("\r\n\n\r");
686 print_dec(SUB_IMAG_HEIGHT);
687         print("\r\n\n\r");
688 print_dec(X_TILES);
689         print("\r\n\n\r");
690 print_dec(Y_TILES);
691         print("\r\n\n\r");
692 print_dec(X_LOCAL);
693         print("\r\n\n\r");
694 print_dec(Y_LOCAL);
695         print("\r\n\n\r");
696 print_dec(X_INIT);
697         print("\r\n\n\r");
698 print_dec(Y_INIT);
699         print("\r\n\n\r");
700
701 }
702
703
704
705
706 void distribute_image_from_zynq(void)
707 {
708     volatile uint32_t count=0;
709
710     while(count!=WIDTH*HEIGHT)
711     {
712
713
714     // espera o req da imagem ser 1
715     while(AXI_IMAGE_REQ==0);
716
717     // verificar se o pixel é da memoria interna
718     if(check_local_pixel(AXI_IMAGE_X,AXI_IMAGE_Y))
```

```
719     {
720         //interface com a PM
721                 // espera o req da pm ser 0
722 
723         set_pixel(AXI_IMAGE_PIXEL,AXI_IMAGE_X,AXI_IMAGE_Y,0,0);
724 
725             if(check_last_pixel(AXI_IMAGE_X,AXI_IMAGE_Y))
726             {
727                SAIDA_INIT_PROG_FIM = 0x1;
728             }
729     }
730      else
731     {
732         //interface com o router
733        // delay2(30000);
734 
735 
736          write_gpio(AXI_IMAGE_PIXEL,AXI_IMAGE_X,AXI_IMAGE_Y,0,0,AXI_IMAGE_X,
737     }
738     AXI_IMAGE_ACK=1;
739     // esperar reset do req
740     while(AXI_IMAGE_REQ==1);
741     AXI_IMAGE_ACK=0;
742 
743     count=count+1;
744     }
745 }
746 int check_last_pixel(uint32_t x,uint32_t y)
747 {
748     if( y== (Y_INITi+SUB_IMAG_HEIGHTi-1)){
749         if( x== (X_INITi+SUB_IMAG_WIDTHi-1)){
750             return 1;
751         }
752     }
753     return 0;
754 }
755 int check_local_pixel(uint32_t x,uint32_t y)
756 {
757     if ((x >= X_INITi) & (x< X_INITi+SUB_IMAG_WIDTHi)){
758       if((y >= Y_INITi) & (y< (Y_INITi+SUB_IMAG_HEIGHTi))){
759         return 1;
760     }}
761         return 0;
762 }
763 
764 void wait_for_checkmsg(void)
765 {
766         while((ENTRADA_INIT_PROG_FIM & 0x1FF)!=0x1FF)
767         {
768             __asm("nop");
769         }
770 }
```

```
771
772
773
774 //funcs referentes aos tiles de fora
775 void get_first_image_recv(void)
776 {
777 get_and_set_local_image_from_zynq_recv();
778 send_transfer_finished_notice_recv();
779 }
780
781 void send_transfer_finished_notice_recv(void)
782 {
783     SAIDA_INIT_PROG_FIM = 0x1;
784 }
785
786
787 void get_and_set_local_image_from_zynq_recv(void)
788 {
789     for(volatile uint32_t i = 0; i < SUB_IMAG_HEIGHTi; i++)
790     {
791         for(volatile uint32_t j = 0; j < SUB_IMAG_WIDTHi; j++)
792         {
793         get_and_set_local_pixel_from_zynq_recv();
794         }
795     }
796 }
797
798 void get_and_set_local_pixel_from_zynq_recv(void)
799 {
800     read_gpio();
801     set_pixel(read_message_pixelValue,read_message_xDest,read_message_yDest,
802 }
803
804 void delay2(uint16_t k)
805 {
806     for (volatile uint16_t ko=0;ko<k;ko++){
807         __asm("nop");
808     }
809 }
810
811
812 uint32_t get_pixel(uint8_t x, uint8_t y, uint8_t s, uint8_t f)
813 {
814
815     write_gpio(0,x,y,s,f,X_INITi,Y_INITi,0);
816     read_gpio();
817
818     return  read_message_pixelValue;
819
820
821 }
822
```

```
823 int abs_paralelo(int numero)
824 {
825     if(numero<0)
826     {
827         numero=-1*numero;
828     }
829     return numero;
830 }
```

A.2 *Verilog* RISC-V Interface

Código A.2 – Interface Externa com Barramento RISC-V

```
1  //*****************************************************
2  //*RISC ESCREVE NO ROTEADOR*****************************
3  //*****************************************************
4
5  else if (iomem_valid && !iomem_ready && iomem_addr[31:0] == 32'h
       0220_0000) begin
6    iomem_ready <= 1;
7    iomem_rdata <= {in_router[31:1], ack};
8    if (iomem_wstrb[0]) in_router[ 7: 0] <= {iomem_wdata[ 7: 1],ack};
9    if (iomem_wstrb[1]) in_router[15: 8] <= iomem_wdata[15: 8];
10   if (iomem_wstrb[2]) in_router[23:16] <= iomem_wdata[23:16];
11   if (iomem_wstrb[3]) in_router[31:24] <= iomem_wdata[31:24];

12 end
13
14 else if (iomem_valid && !iomem_ready && iomem_addr[31:0] == 32'h
       0221_0000) begin // escreve mensagem para risc
15   iomem_ready <= 1;
16   iomem_rdata <= in_router[31:0];
17   if (iomem_wstrb[0]) in_router[39: 32] <= iomem_wdata[ 7: 0];
18   if (iomem_wstrb[1]) in_router[47: 40] <= iomem_wdata[15: 8];
19   if (iomem_wstrb[2]) in_router[55:48] <= iomem_wdata[23:16];
20   if (iomem_wstrb[3]) in_router[63:56] <= iomem_wdata[31:24];

21 end
```