EDMAR ANDRÉ BELLORINI

CLUPA: UM AMPLIADOR DIGITAL DE DOCUMENTOS IMPRESSOS SOBRE UMA PLATAFORMA *MULTICORE*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2015

B447c Bellorini, Edmar André
Clupa: um ampliador digital de documentos impressos sobre uma plataforma multicore / Edmar André Bellorini. – Curitiba, 2015.
101 f. : il. ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2015.

Orientador: Roberto André Hexsel,
Bibliografia: p. 76-82.

1. Sistemas embutidos de computador. 2. Lupa. 3. Deficientes visuais - Tecnologia. I. Hexsel, Roberto André. II. Universidade Federal do Paraná. III. Título.

CDD: 006.22

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Edmar André Bellorini, avaliamos o trabalho intitulado, *"Implementação de um Ampliador Digital de Documentos Impressos Sobre uma Plataforma Multicore""*, cuja defesa foi realizada no dia 13 de março de 2015, às 09:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:

(X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 13 de março de 2015.

Prof. Dr.Roberto André Hexsel
DINF/UFPR – Orientador

Prof. Dr. Marcio Seiji Oyamada
Unioeste - Membro Externo

Prof. Dr. Eduardo Todt
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Inicialmente, agradeço à Jéssica, minha excelentíssima namorada que, mesmo longe durante dois anos, me motivou para conclusão desta etapa Agradeço por sua compreensão, paciência, por aguentar tanto tempo de saudades e pelo carinho infinito que emana desta pequena grande pessoa para comigo. Também agradeço à sua família: Paulo, Odete e Nátaly, por ter me cedido refúgio nos momentos de fuga da pesquisa.

Agradeço a minha família, meu pai Elmo, irmão Elcimar, irmã Eluana e sobrinho João. Que foram compreensivos e pacientes com minha ausência, além de me apoiarem em todas as etapas de formação. Aproveito a oportunidade para dedicar este trabalho para minha mãe Emidia.

Agradeço ao meu orientador, Prof. Roberto André Hexsel, por acreditar e me aceitar no programa de Pós-Graduação, pela paciência e dedicação nas inúmeras horas de dúvidas, e pelas conversas que me mantinham motivado. Também agradeço aos demais professores e funcionários da Universidade Federal do Paraná que, de forma direta ou indireta, contribuiram com minha formação e finalização deste trabalho. Agradeço ao meu co-orientador Prof. Marcio Oyamada, que contribuiu com idéias e observações que enriqueceram este trabalho. Também agradeço aos professores e colegas da Universidade Estadual do Oeste do Paraná, que sempre proveram dicas de como ser um bom mestrando.

Um grande obrigado aos meus colegas de programa dos laboratórios de pesquisa LARSIS, VRI e FAES, em especial, Guilherme, Luiz, Maurício, Ailton, Diego(Ca), José, Luiza, Diego(Lu), Rita, Rogério, Rui e Josiney. Foram muitos momentos de filas no RU, "hora do café", seções da tarde e trabalhos nos finais de semana, com discussões de idéias, muitas vezes sobre as pesquisas, e momentos de distrações. Também agradeço ao André, colega da aventura de se afastar de amigos e parentes para aperfeiçoamento que também acabou dividindo teto/dívidas destes últimos dois anos.

Obrigado a todos!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta o projeto do cLUPA (cMIPS-LUPA). cLUPA é um sistema embarcado portátil para ampliar documentos impressos utilizando uma câmera de vídeo para captura e um monitor para exibição. O *hardware* foi projetado com dois núcleos cMIPS que se comunicam através de uma fila, e o *software* foi escrito baseado nas funcionalidades do xLupa Embarcado e para ser executado diretamente sobre o *hardware*, sem a necessidade um sistema operacional.

O projeto, modelado em VHDL, levou em consideração a plataforma disponível Mercurio IV para futura prototipação, o que limitou algumas decisões do projeto.

O cLUPA foi validado e seu desempenho avaliado com a realização de simulações com configurações de ampliação ativada ou desativada e contraste inativo, cinza, verde ou vermelho, totalizando oito configurações distintas. Os resultados mostram que o cLUPA processa de 2,3 a 9,3 quadros por segundo, e seu desempenho é limitado pelo relógio de 50MHz, que é o relógio da plataforma de prototipação.

Este trabalho também propõe e discute algumas alterações necessárias ao modelo do cLUPA para prototipação utilizando o conjunto de desenvolvimento Mercúrio IV.

ABSTRACT

This text presents the design cLUPA (cMIPS-LUPA). cLUPA is an embedded portable system that magnifies printed documents using a video camera to capture the image and a monitor for displaying the amplified version. The hardware comprises two cMIPS cores that communicate through a queue, and the software was written to support the functionalities of "xLupa Embarcado", and to run directly on top of the hardware, without an operating system.

cLUPA was designed for prototyping on the Mercurio IV platform and that imposed some constraints on some parts of the design.

The cLUPA was validated by simulations, and its performance assessed with magnification settings enabled and disabled, and contrast inactive, gray, green or red, into eight different configurations. The results indicate that cLUPA can process images at 2.3 to 9.3 frames per second, and its performance is limited by the 50MHz clock, which is a limitation of the development platform..

Some changes to the design are needed for prototyping cLUPA on the Mercurio IV development kit. These are presented and discussed.

CAPÍTULO 1

INTRODUÇÃO

As tecnologias assistivas (TAs) fornecem um caminho para que as pessoas com deficiência ampliem suas funcionalidades e obtenham melhor inclusão no meio social. As TAs são encontradas em diversas áreas tais como vida diária com talheres e abotoadores anatômicos, urbanismo com guias nas calçadas e avisos visuais e sonoros, acesso a computadores com os teclados e *mouses* adaptados, impressoras Braille, leitores de telas, e ampliadores digitais [7].

O IBGE informa que, em 2010, existiam aproximadamente 48 milhões de brasileiros que apresentam algum tipo de deficiência visual, auditiva ou mental. A deficiência visual predomina sobre as demais, alcançando 35,6 milhões de brasileiros, o equivalente a 18,8% da população do nosso país, dos quais 600 mil são cegos, sendo portanto, 35 milhões de brasileiros diagnosticados com visão subnormal [31], e destes, 6 milhões diagnosticados com visão subnormal alta.

Frequentemente pessoas com visão subnormal alta dependem de ampliadores analógicos ou de um outro indivíduo para ler documentos impressos, como manuais de equipamentos eletrônicos, menus ou folhetos informativos de hotéis [7, 8].

Em uma tentativa de prover maior independência para as pessoas com visão subnormal, os ampliadores de documentos impressos são sistemas embarcados que, através de uma câmera de vídeo, capturam documentos impressos e os exibem em um monitor ou televisão. O xLupa Embarcado [17, 25] é um projeto de ampliador de documentos impressos derivado do xLupa [8].

Novos e melhores produtos são possíveis devido ao avanço das tecnologias de circuitos integrados, possibilitando equipamentos menores, mais potentes computacionalmente e com um conjunto maior de funcionalidades, com custo reduzido de projeto e implementação. Os avanços incluem equipamentos que permitem rápida prototipação de cir-

cuitos integrados, tais como os *Field-Programmable Gate Arrays* (FPGAs) e organizações para os componentes, tais como os sistemas *multicore* e *System-on-Chip* (SoC) [19, 27, 37].

O objetivo deste trabalho é prover uma solução de ampliação de documentos impressos portátil com código livre para pessoas com visão subnormal. Assim, todo o *hardware* e *software* do projeto é disponibilizado sob a licença *GNU General Public License* (GPL) [23].

Este trabalho descreve o projeto de um ampliador de documentos impressos, chamado cLUPA (cMIPS-LUPA) que baseia-se nas funcionalidades de ampliação e contraste do xLupa Embarcado. O projeto é composto por três módulos: uma câmera que realiza a captura das imagens dos documentos impressos, um módulo de processamento e um dispositivo de vídeo que recebe as imagens tratadas de acordo com configurações definidas pelo usuário.

O módulo de processamento contém um processador *multicore* com dois núcleos cMIPS [29] que se comunicam através de uma fila. Cada processador tem sua memória principal e dispositivos independentes, sendo a fila de comunicação o único dispositivo compartilhado.

A plataforma de *hardware* do cLUPA foi modelada em VHDL [50] e o modelo compilado com GHDL [22]. A plataforma que foi considerada para futura implementação é o conjunto de desenvolvimento Mercurio IV [15], que contém recursos limitados em termos de dispositivos e seu relógio é de $50MHz$. Estas limitações condicionaram várias decisões de projeto, tais como executar o código do cLUPA sem um sistema operacional, bem como otimizações para melhorar o desempenho.

Este documento está organizado da seguinte forma: o Capítulo 2 apresenta os conceiros da arquitetura MIPS e descreve a implementação chamada cMIPS.

O Capítulo 3 apresenta o ampliador digital e leitor de tela xLupa que executa em ambiente Linux, seu projeto filho xLupa Embarcado, que compreende em um ampliador digital portátil de documentos impressos para exibição em monitores ou televisores, e apresenta alguns produtos semelhantes ao xLupa Embarcado.

O Capítulo 4 descreve a plataforma de *hardware* do cLUPA, adições realizadas no

cMIPS, dispositivo de comunicação entre núcleos e dispositivos de acesso direto à memória. Também descreve a *Hardware Platform Interface* (HPI), módulos de *software* que executam nos núcleos cMIPS e algumas adaptações realizadas no *software* do xLupa Embarcado.

O Capítulo 5 apresenta as simulações e as métricas que foram utilizadas para mensurar o desempenho das plataforma cLUPA e os resultados obtidos destas simulações. O Capítulo 6 apresenta as alterações necessárias na plataforma cLUPA para futura prototipação com o conjunto de desenvolvimento Mercurio IV. O Capítulo 7 discute os resultados obtidos das simulações.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

O rápido desenvolvimento das tecnologias computacionais nos permite adquirir computadores com mais poder de processamento, capacidade de armazenamento e melhor eficiência energética por 0,05% do valor de um computador da década de 70 [28]. Isso é possível devido às novas tecnologias de fabricação de componentes de circuito integrado em conjunto com as inovações em seus projetos. Em relação às inovações de projetos, estão disponíveis sistemas *multicore* e processadores especializados em executar tarefas específicas [18, 27, 19].

Este Capítulo apresenta conceitos usados para a proposta da plataforma do cLUPA. A arquitetura MIPS é apresentada na Seção 2.1 e é base para a implementação do núcleo cMIPS (Seção 2.1.1). Nas Seções 2.2, 2.3 e 2.4 são apresentados os conceitos de organização *multicore*, *System-on-chip* e FPGAs, e o conjunto de desenvolvimento Mercurio IV é detalhado na seção 2.5. O Capítulo finaliza apresentando alguns trabalhos que se utilizam de organização *multicore* e/ou SoC e que contribuíram para o desenvolvimento do cLUPA.

2.1 MIPS

MIPS (*Microprocessor without Interlocked Pipe Stages*) é a arquitetura de um processador de propósito geral [27, 56]. O principal objetivo de projeto da arquitetura é o alto desempenho na execução de código compilado, objetivo alcançado com um conjunto de instruções simples com execuções que se aproximam das micro-operações, e na sua versão original com a transferência da responsabilidade de gerenciar as dependências de dados e controle do *hardware* para o *software* [27, 38].

Existem diversas implementações da arquitetura MIPS. Os processadores MIPS R2000 e R3000 foram as primeiras implementações do MIPS com *pipeline* de cinco estágios. O

processador MIPS R4000 introduziu instruções de 64 bits sobre a arquitetura original, o processador R8000 aprimorou o R4000 adicionando uma unidade de ponto-flutuante que opera em paralelo à unidade de inteiros e o processador R10000 foi a primeira implementação superescalar [28, 56].

O modelo original do fluxo de execução das instruções da arquitetura MIPS é um *pipeline* de cinco estágios, mostrado na Figura 2.1, adaptada de [34], e pode conter multiplas instruções ativas simultaneamente [26]. Cada instrução foi definida para utilizar todos os estágios do *pipeline*, mesmo que em algum determinado estágio não resulte em modificações de seus valores [56]. Desta forma, é possível iniciar uma nova instrução a cada ciclo do relógio [28, 34].

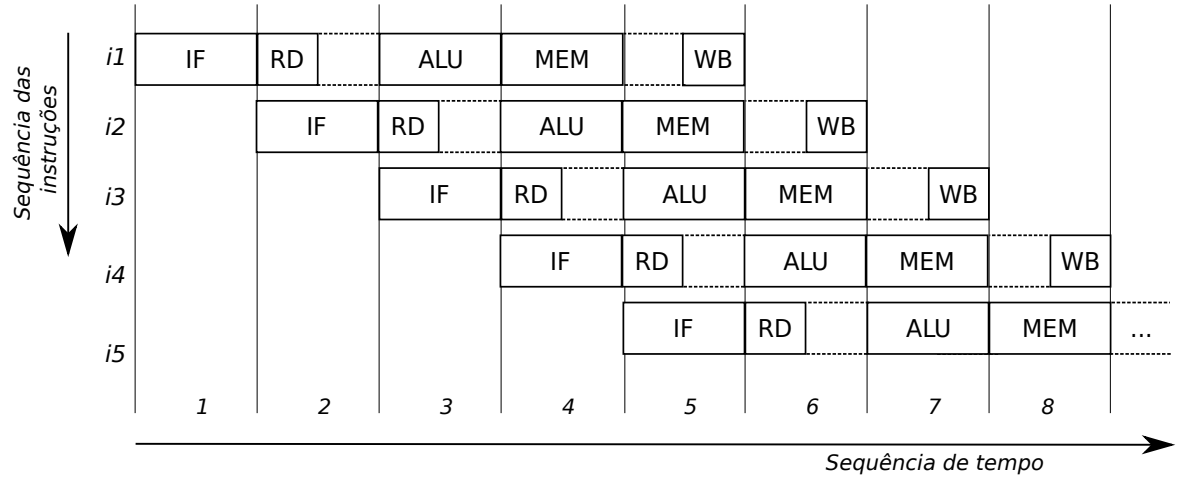


Figura 2.1: *Pipeline* de cinco estágios da implementação MIPS R3000

O *pipeline* mostrado na Figura 2.1 é dividido em cinco estágios: IF (*Instruction-Fetch*) busca a próxima instrução na memória *cache* de instruções, RD (*Read-Registers*) decodifica a instrução atual e busca os operandos no banco de registradores, ALU (*Arithmetic and Logic Unit*) executa operação aritmética ou lógica sobre os operandos, MEM (*Memory*) acessa operandos na memória *cache* de dados, e WB (*Write-Back*) que grava o resultado obtido da execução no banco de registradores.

A busca por instruções e operandos em memória são realizadas em estágios distindos do *pipeline*, e podem concorrer entre si no acesso à memória principal, como acontece nos casos das instruções i1 (WR) e i5 (IF) na Figura 2.1. Assim, a arquitetura MIPS utiliza duas portas de memória: uma para a memória *cache* de instruções, que mantém

somente as instruções de um programa, e a outra para a *cache* de dados. Esta separação foi pensada para permitir a busca de instruções sem afetar o acesso aos operandos de uma instrução que acessa a memória.

O banco de registradores é composto por 32 registradores (de 32 bits cada) para propósito geral [34, 56], numerados de 0 à 31. Emprega-se uma convenção de *software* que define nomes e funções específicas para estes registradores. Os registradores são:

Tabela 2.1: Registradores da arquitetura MIPS

Registradores	Nome	Descrição
0	$zero	sempre retorna o valor "zero"
1	$at	temporário reservado para uso do montador
2-3	$v0-v1	retorno de valores de funções e procedimentos
4-7	$a0-a3	parâmetros para chamada de funções e procedimentos
8-15	$t0-t7	temporários livres para uso em funções e procedimentos (valores não são preservados)
16-23	$s0-s7	registradores "salvados" (valores devem ser preservados pelas funções e procedimentos)
24-25	$t8-t9	mesmo que registradores 8-15
26-27	$k0-k1	reservados para gerenciamento de interrupções (sistema operacional)
28	$gp	ponteiro global (sistema operacional)
29	$sp	ponteiro para pilha de dados
30	$fp	ponteiro para registro de ativação de funções
31	$ra	registrador para retorno das chamadas de funções ou procedimentos

O acesso aos operandos de uma instrução na memória é definido pelo modo de endereçamento. A arquitetura MIPS utiliza apenas três: endereçamento por registrador, que define o local do operando em um registrador especificado, endereçamento com operando imediato que contém o operando diretamente na instrução, e endereçamento por deslocamento, que consiste em somar o conteúdo de um registrador base com uma constante imediata de 16 bits para computar o endereço efetivo do operando. Esse endereço efetivo é obrigatoriamente alinhado: a memória é um vetor de bytes que pode ser acessado através de endereços que sejam múltiplos dos tamanhos das palavras *half-word* (2 bytes) e *word* (4bytes) [34, 56].

O tamanho de uma instrução na arquitetura MIPS é de 32 bits e seus campos são definidos em três tipos [28], como mostra a Figura 2.2, adaptada de [28].

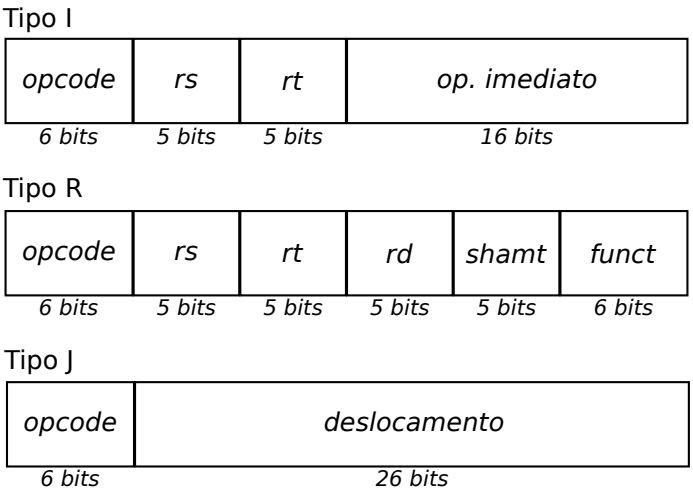


Figura 2.2: Formato das instruções da arquiteura MIPS

Todos os formatos de instruções, invariavelmente, apresentam um campo de 6 bits chamado opcode, que identifica a instrução a ser executada. As instruções que apresentam operandos em registradores os identificam com campos de 5 bits. O formato de instrução tipo I é utilizado para instruções de leitura e gravação na memória principal e instruções de saltos condicionais. O campo rs representa o registrador fonte da instrução e o campo rt representa o registrador de destino. O tipo R identifica as instruções de operação lógica e aritméticas entre registradores. O campo rs representa o registrador fonte-1 (primeiro operando), o campo rd é o registrador de destino, o campo rt é o registrador fonte-2 (segundo operando). A operação lógica ou aritmética é definida pelo campo de 6 bits funct e, para os casos de operações lógicas de deslocamento, o campo shamt (de 5 bits) define de quantos bits será o operando deslocado. O tipo J identifica as instruções de saltos incondicionais que, além do campo opcode, contém apenas um campo de 26 bits, chamado deslocamento, que determina o valor que será atribuído ao registrador *program counter*.

Uma implementação da arquitetura MIPS que segue o fluxo original de execução e contém as características apresentadas nesta Seção é o cMIPS [29], que é apresentado na Seção 2.1.1.

2.1.1 cMIPS

cMIPS é um modelo escrito em VHDL do *pipeline* de cinco estágios da arquitetura MIPS que se aproxima da descrição apresentada em [49]. Com o objetivo de auxiliar no ensino da disciplina de arquitetura de computadores [29] o modelo executa código em linguagem de programação C compilado pelo GCC.

Para simular o cMIPS emprega-se o arquivo para teste (*testbench*) com organização mostrada na Figura 2.3. O núcleo cMIPS é ligado à duas memórias *cache*. A memória *cache* de instruções é representado pelo módulo CACHE-INSTR e mantém cópias das instruções armazenadas na memória ROM. CACHE-INSTR envia dois sinais para a memória ROM: mem_i_sel, que indica ativação de leitura da memória, e os 32 bits de endereço mem_i_addr. Como resposta da memória ROM, a CACHE-INSTR recebe os 32 bits da instrução pelo sinal datrom. A comunicação da *cache* de instruções com o núcleo cMIPS contém os mesmos sinais da comunicação da *cache* com a memoria ROM nomeados inst_aVal, equivalente ao mem_i_sel, i_addr, que espelha mem_i_addr e cpu_inst, que é a instrução recebida pela *cache* pelo sinal datrom.

A memória *cache* de dados, módulo CACHE-DADOS, mantém cópias dos dados presentes na memória RAM. As memórias *cache* usam mapeamento direto, e a memória *cache* de dados utiliza a política *write-through*, que atualiza os dados escritos simultaneamente com a memória RAM. A memória *cache* CACHE-DADOS ao receber um sinal de ativação de leitura/escrita de dados na memória (data_aVal) repassa os sinais wr (escrita ou leitura), d_addr (endereço), cpu_data (dados para a operação de escrita) e cpu_xfer (determina bytes escritos/lidos da palavra de 32 bits) para a memória RAM com os sinais mem_wr, mem_addr, datram_out e mem_xfer, respectivamente, em conjunto com o sinal de ativação mem_d_sel. A memória RAM responde uma operação de leitura ao módulo CACHE-DADOS com os dados requeridos através do sinal datram_inp.

A memória ROM é inicializada por um arquivo chamado prog.bin que contém o código de máquina gerado após compilar o código de um programa escrito em linguagem C ou assembly. A ação de compilar código também gera o arquivo inicial da memória RAM, chamado data.bin. O arquivo de teste do cMIPS, durante a execução da simulação, pode

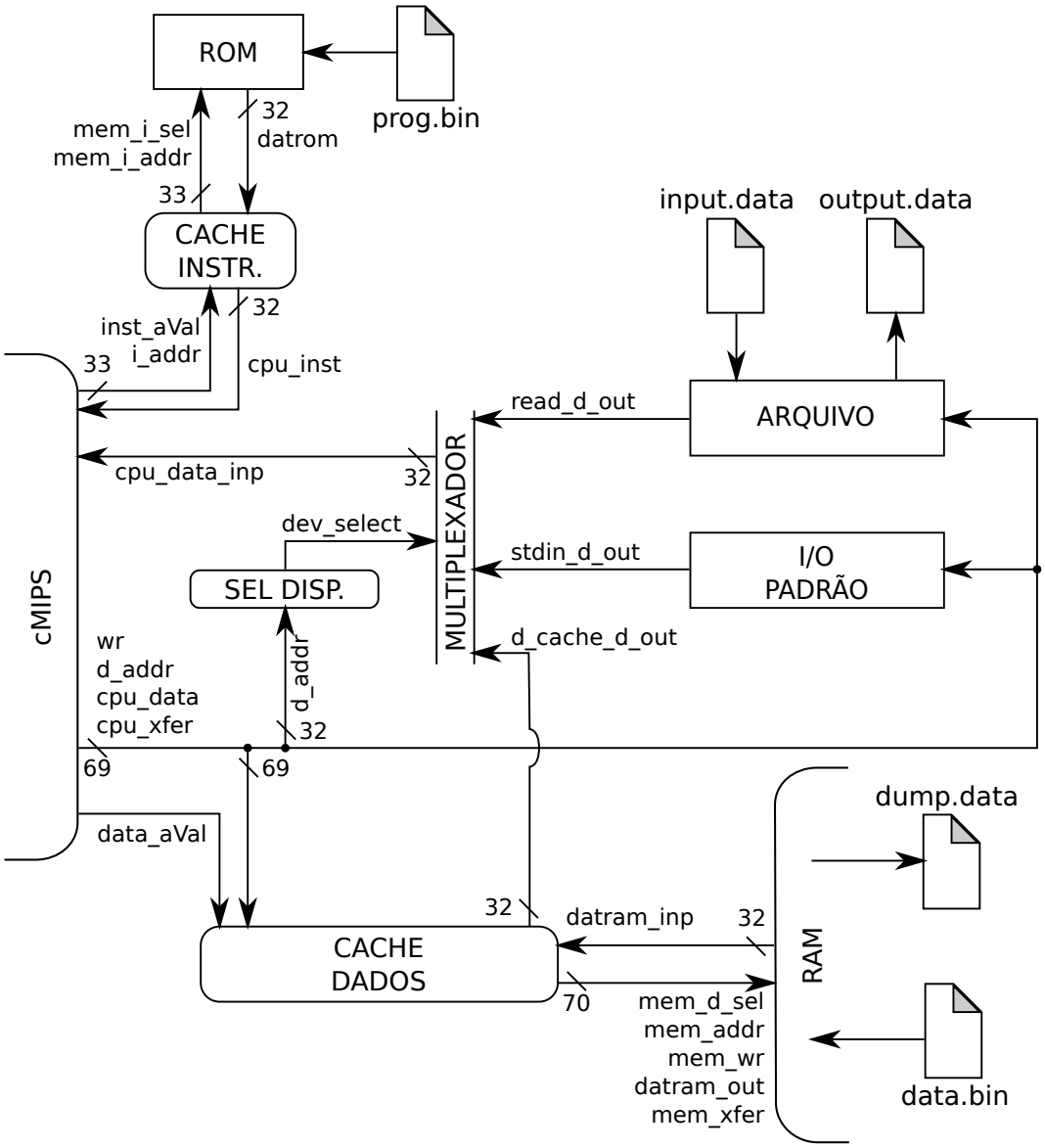


Figura 2.3: Diagrama de blocos do arquivo de testes do cMIPS

gerar um arquivo de *dump* (ou *snapshot*) chamado dump.data, que contém os valores de todos os bytes da memória RAM no momento da sua criação.

Todos os dispositivos, com excessão das memórias *cache* de instrução, ROM e RAM, são ligados ao cMIPS através de um barramento comum que contém os sinais wr, d_addr, cpu_data e cpu_xfer. Os sinais que representam os dados de retorno dos módulos são ligados através de um barramento multiplexado, chamado MULTIPLEXADOR, que é controlado por um seletor. O seletor SEL-DISP decodifica o endereço que está presente no barramento de endereço e seleciona o chaveamento adequado no MULTIPLEXADOR com o sinal dev_select, que entrega o dado requerido ao núcleo cMIPS através do sinal cpu_data_inp.

Em termos de dispositivos, o programa de teste do cMIPS contém um módulo para ler ou escrever em arquivos, chamado ARQUIVO na Figura 2.3. O módulo ARQUIVO é capaz de ler um inteiro (quatro bytes) do arquivo binário input.data e devolver ao núcleo através do sinal read_d_out ou escrever o inteiro presente no sinal cpu_data no arquivo output.data. O módulo I/O-PADRÃO é reponsável por escrever na saída padrão do simulador o inteiro presente no sinal cpu_data representado no formato hexadecimal de quatro bytes ou um caractere de um byte, também é possível ler um caractere da entrada padrão e entregá-lo ao núcleo cMIPS pelo sinal stdin_d_out.

Um programa escrito em linguagem de programação C precisa incluir o arquivo de cabeçalho cMIPS.h para ser compilado para o cMIPS. Este arquivo de cabeçalho contém o endereçamento dos dispositivos e também a HPI (*Hardware Platform Interface*), apresentada no Quadro 2.1, que são funções necessárias para utilizar os dispositivos do arquivo de teste do cMIPS.

```
1  // Sistema ==========================================================
2  extern void exit(int); // Finaliza a execução de um programa
3
4  // I/O Padrão =======================================================
5  extern void print(int);          // Escreve um inteiro na saída padrão
6  extern void to_stdout(char c);   // Escreve um caractere na saída padrão
7  extern int  from_stdin(char *); // Lê um caractere da entrada padrão
8
9  // Arquivo ==========================================================
10 extern int  readInt(int*);     // Lê um inteiro de input.data
11 extern void writeInt(int);     // Escreve um inteiro em output.data
12 extern void writeClose(void); // Fecha arquivo output.data
13
14 // RAM ==============================================================
15 extern void dumpRAM(void);  // Efetua o dump de memória para dump.data
```

Quadro 2.1: HPI do arquivo de testes do cMIPS

A plataforma de *hardware* do cLUPA proposta neste documento usa dois núcleos cMIPS. O arquivo de teste do cMIPS foi alterado de forma a adicionar o segundo núcleo,

os novos dispositivos (discutidos no Capítulo 4) e comunicar-se entre si. Alguns conceitos para processadores com vários núcleos são apresentados na Seção 2.2 e as alterações efetuadas narquivo de teste são discutidas na Seção 4.1.1.

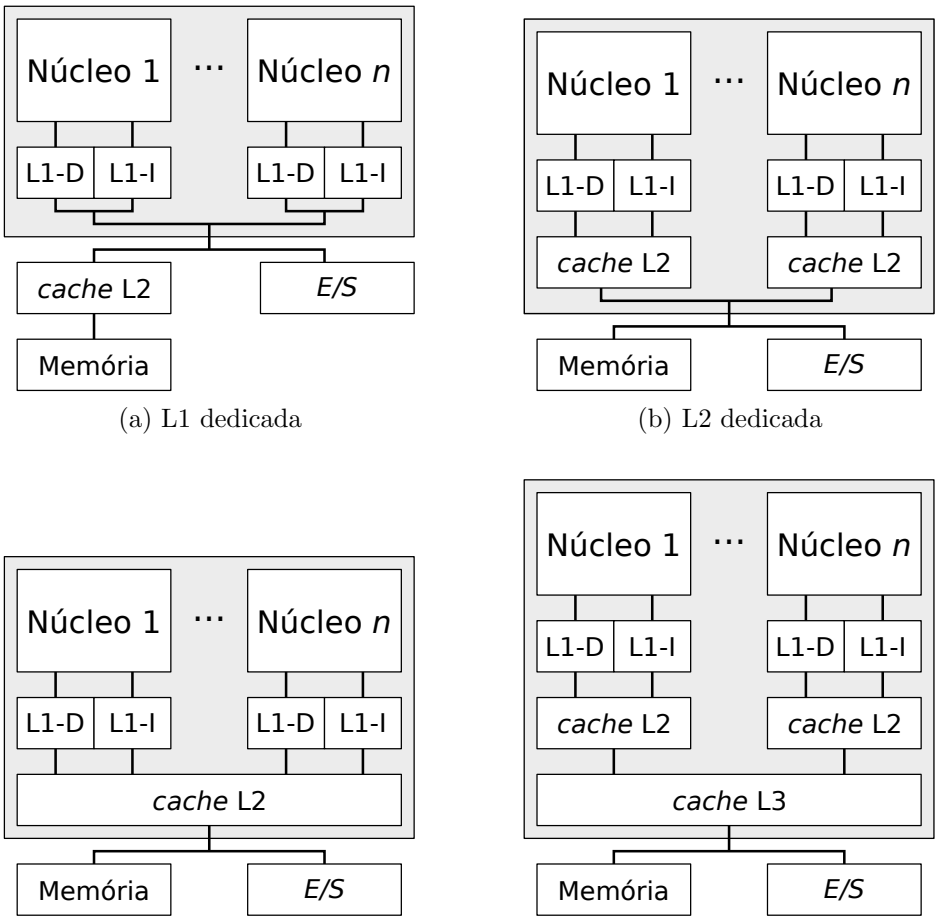
2.2 Multicores

Multicores são sistemas com dois ou mais núcleos (*cores*) em uma única pastilha de silício [19, 55]. Entende-se o termo "núcleo" como sendo um processador independente com memórias *cache* primárias. Assim, um sistema *multicore* é composto por vários núcleos organizados de modo a oferecer melhor desempenho do que processadores *single-core* (apenas um núcleo) [21].

A organização de processadores *multicore* pode ser definida, resumidamente, em níveis de memória *cache*, se essas memórias são internas ou externas ao circuito integrado, compatilhamento da memória *cache* entre os processadores, como será o acesso à memória principal, quantidade de processadores e a simetria [28, 55].

Memórias *cache* L1 geralmente são exclusivas a cada processador; memórias *cache* L2 e L3 podem ser compartilhadas entre os processadores. Conforme a Figura 2.4, adaptada de [46, 55], são quatro as organizações para memória: (a) memória *cache* L2 externa e compartilhada, usada em processadores *multicore* antigos e em alguns processadores embarcados, (b) memória *cache* L2 exclusiva, que provê melhor desempenho para aplicações com forte localidade, mas necessita de algum protocolo que mantenha os dados compartilhados coerentes, (c) memória *cache* L2 interna e compartilhada, elimina o problema de coerência, pois a interferência nos dados é construtiva e (d) memória *cache* L2 dedicada e L3 compartilhada.

O acesso à memória principal, por parte dos processadores *multicore*, pode ser projetado de duas formas: *Uniform Memory Access* (UMA) é a configuração na qual todos os processadores podem acessar, tanto para leitura quanto para escrita, toda a extensão da memória principal disponível, com o mesmo tempo de acesso; e *Non-Uniform Memory Access* (NUMA) é a configuração na qual todos os processadores acessam toda a extensão da memória, porém o tempo de acesso é diferente entre os processadores [28, 39, 55].



(a) L1 dedicada

(b) L2 dedicada

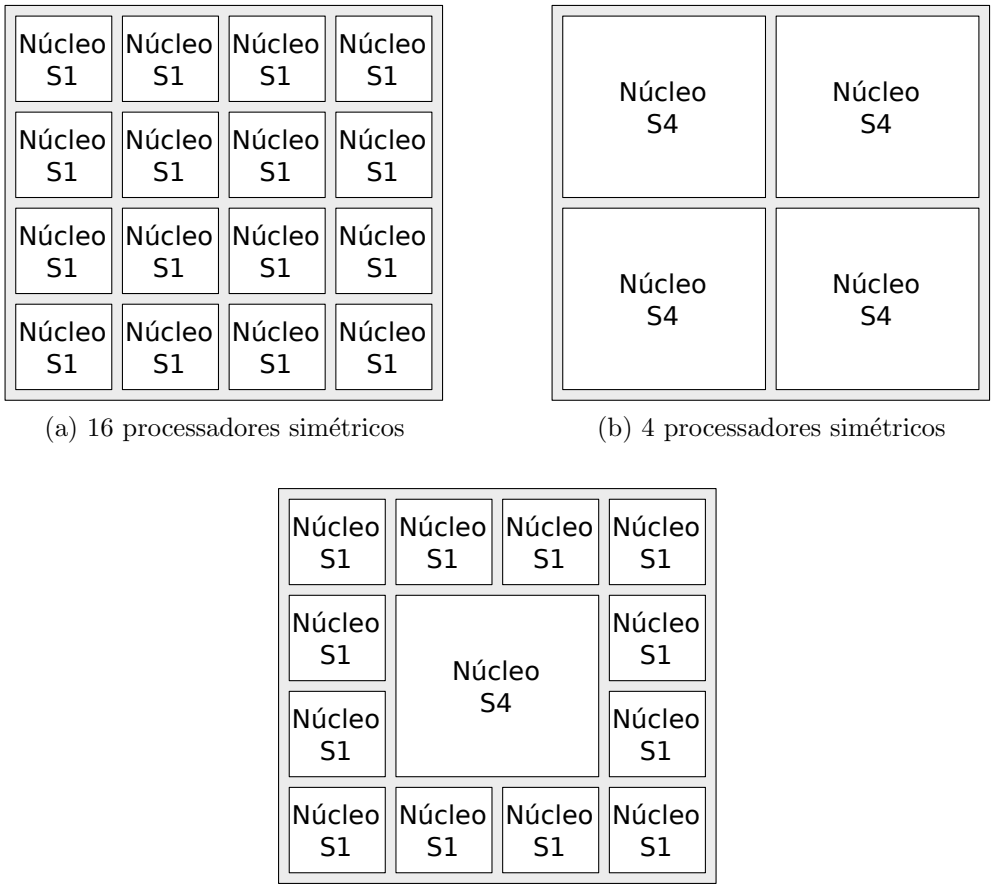
(c) L2 compartilhada

(d) L3 compartilhada

Figura 2.4: Organizações das memórias *cache*.

O número de processadores e sua simetria com relação a poder computacional geram três possíveis organizações. A Figura 2.5, adaptada de [30], apresenta uma abstração dessas organizações *multicore*, S1 e S4 são tamanhos relativos dos processadores (núcleos), em termos de área ocupada e número de transistores, e S4 é quatro vezes maior que S1. O tamanho relativo também representa a capacidade computacional, quanto maior o processador, maior é sua capacidade computacional.

O uso de processadores simétricos (2.5a e 2.5b) difere, em termos de organização, na área total do circuito integrado (CI) que será destinada para cada processador. Em termos de *software*, usar um conjunto grande de processadores relativamente pequenos favorece aplicações com granularidade fina, enquanto usar um conjunto pequeno de processadores relativamente poderosos favorece aplicações com granularidade grossa e a execução de múltiplas aplicações independentes que requerem grande carga computacional [4].



(a) 16 processadores simétricos

(b) 4 processadores simétricos

(c) 12 + 1 processadores assimétricos

Figura 2.5: Abstração das organizações *multicore*.

A Figura 2.5c apresenta a organização assimétrica (ou heterogênea) [36] que consiste em mesclar os tamanhos relativos dos processadores. Essa organização provê ao menos um processador mais poderoso e destinado a executar a região serial do código da aplicação e vários outros processadores menores que executam as as regiões paralelas [30].

2.3 System-on-Chip

Uma forma de organização para sistemas computacionais é utilizar um conjunto de processadores projetados para funcionalidades específicas, chamados de sistemas híbridos [44, 52] ou SoCs (*System-on-Chip*). Esta organização provê diversos processadores, sendo cada um deles destinados por executar parte da computação de uma aplicação. A Figura 2.6, adaptada de [52], apresenta uma abstração dessa organização. Cada processador, designado por núcleo *Ex*, tem a sua função específica. Por exemplo, no SoC Snapdragon 800,

que equipa telefones móveis (*smartphones*), o núcleo E1 é um processador de propósito geral, E2 é uma unidade de processamento gráfico, E3 é um processador de sinais DSP (*Digital Signal Processor*), E4 é um processador para conectividade (rede 4G, GSM, Wi-Fi, Bluetooth e USB), E5 é um processador multimídia (áudio, vídeo e gestos), E6 controla sensores, E7 controlador da(s) câmera(s) fotográfica, E8 é o controlador do display e, por fim, E9 é o processador de localização, que utiliza sensores de E6 em conjunto com as informações de conectividade de E4 para determinar localização do dispositivo [52].

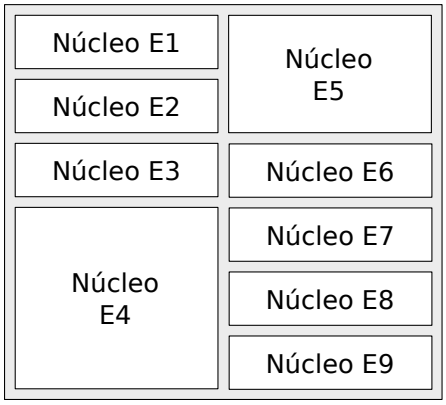


Figura 2.6: Processador *multicore* híbrido.

Com as características dos sistemas *multicore* e SoCs apresentadas, é possível definir a plataforma de *hardware* do cLUPA como sendo um SoC *multicore* de dois núcleos simétricos com memória dedicada. A Seção 2.4 apresenta alguns conceitos sobre FPGAs (*Field-Programmable Gate Array*) e logo em seguida descreve o conjunto Mercurio IV empregado para o desenvolvimento do cLUPA (Seção 2.5).

2.4 Field-Programmable Gate Arrays

Um *Field-Programmable Gate Array* (FPGA) é um CI que pode ser programado posteriormente à sua fabricação para implementar circuitos digitais arbitrários [37, 60]. A história dos FPGAs inicia em 1960 com o desenvolvimento do conceito de alterar a função lógica de um CI após a sua fabricação (*field-programmability*). Em 1970 surgiram os dispositivos programáveis baseados em memórias apenas de leitura (PROM - *Programmable Read-Only Memory*), que evoluiram para os PLAs (*Programmable Logic Arrays*) no final

da década de 70 [37].

Os primeiros PLAs, chamados atualmente de PLDs (*Programmable Logic Devices*), consistem em dois níveis de portas lógicas AND e OR que podem ser programadas para realizar funções lógicas diversas [14, 37]. A Figura 2.7, adaptada de [14], mostra a função lógica de comparação entre duas palavras de 4 bits e sua implementação equivalente em uma célula de um PLD.

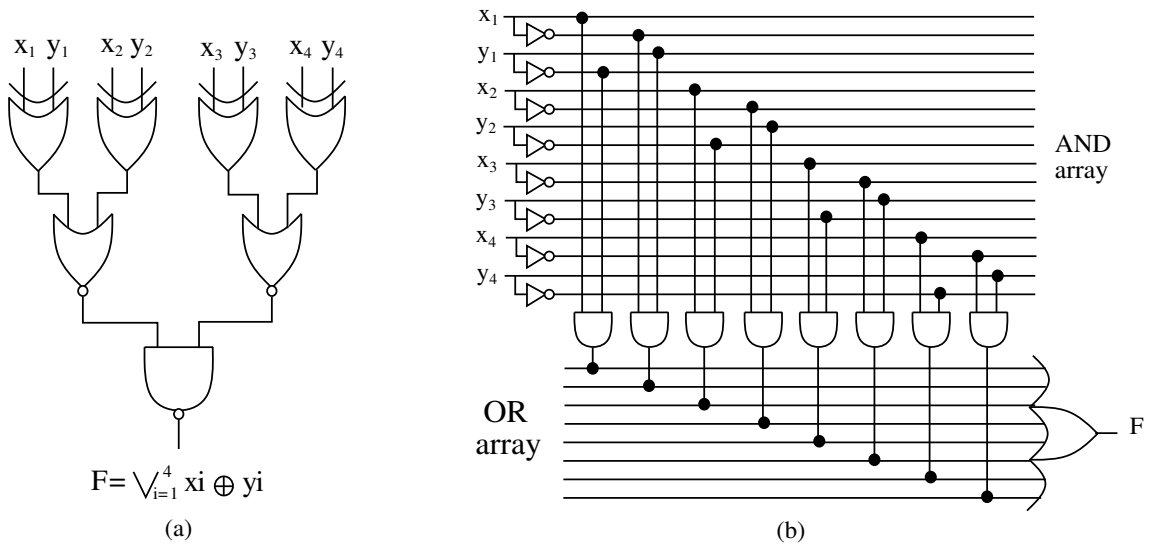


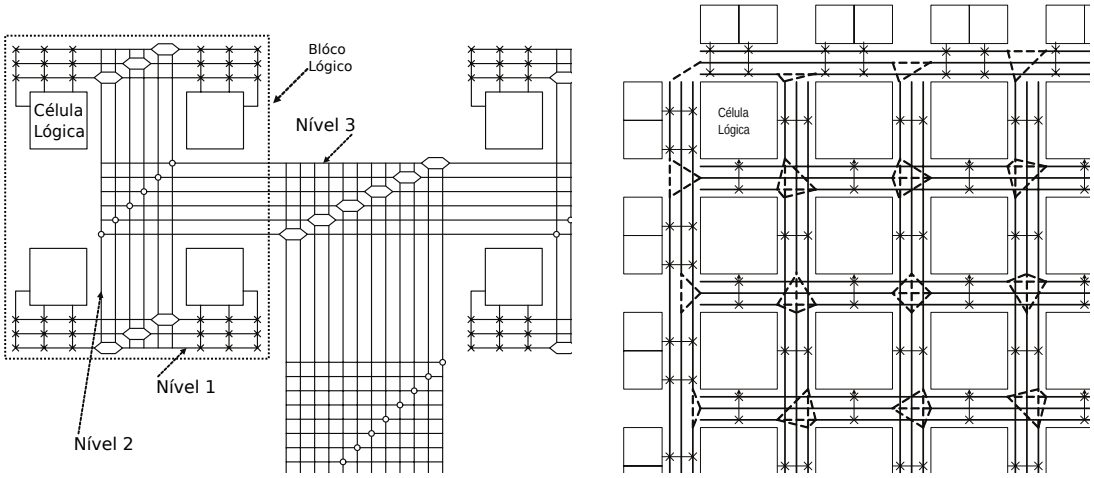
Figura 2.7: Lógica num PLD.

O FPGA é um tipo de dispositivo que contém células lógicas reconfiguráveis e interconectadas [37, 60]. As células lógicas são compostas por *Look-up Tables* (LUTs) e *Flip-Flops* (FFs). LUTs são as unidades básicas de um FPGA que permitem implementar qualquer função lógica de N variáveis booleanas e os FFs armazenam os valores computados pelas LUTs.

Os FPGAs contém circuitos especializados que ampliam suas capacidades. Blocos de Memória (BRAM - *Block Random Access Memory*) são memórias RAMs *dual-ported* que se comportam como RAMs, ROMs ou registradores de deslocamento. Os blocos DSP (*Digital Stream Processor*) executam operações lógicas e aritméticas [60].

As células lógicas de um FPGA devem ser interconectadas. A estrutura destas interconexões varia para permitir flexibilidade de ligação durante o processo de sintetização do circuito. Existem duas formas de interligar as células lógicas: a primeira é usar uma estrutura hierárquica que consiste em agrupar as células lógicas por funcionalidade e

conectá-las em um nível inferior. Esses grupos lógicos são interconectados por níveis superiores na hierarquia de conexões. A segunda forma é utilizar "ilhas", todas as células lógicas estão distribuídas em uma matriz de duas dimensões juntamente com recursos de roteamento [37]. A Figura 2.8, adaptada de [37], mostra as possíveis formas de interconexões entre as células lógicas em um FPGA.



(a) Conexão Hierárquica (b) Conexão Ilhas

Figura 2.8: Interconexão de blocos lógicos (a) hierárquica e (b) ilhas em FPGA.

As células de E/S (entrada e saída) compreendem a lógica para entrada e saída de dados do FPGA, permitindo a comunicação com o meio externo e suporte a periféricos. Normalmente estão posicionadas circundando as demais células lógicas e consomem uma parcela aproximada de 35% da área de um FPGA [37].

A disposição geral das células lógicas, células específicas e células de E/S segue o padrão mostrado na Figura 2.9, adaptada de [60]. Grupos de células lógicas são alocadas em colunas, entre essas colunas de células lógicas podem existir colunas de células específicas (colunas de BRAMs ou DSPs) e circundando as células lógicas e específicas encontram-se as células de E/S. Todas as células são interconectadas por meio de fios configuráveis por hierarquia ou roteamento [37, 60].

Tanto a implementação das funções lógicas quanto a configuração das interconexões das células lógicas são feitas através de um vetor de bits chamado *bit stream*. O *bit stream* contém as ligações que deverão ser ativadas nos LUTs e FFs para formar as funções lógicas desejadas e a configuração das ligações dos fios que compõem as interconexões.

O *bit stream* é carregado uma vez para uma memória estática de configuração do FPGA e toda vez que este é inicializado, os bits desse arquivo são carregados configurando as conexões de todos os componentes de modo a fornecer a configuração do circuito digital definido na síntese do projeto [37, 54].

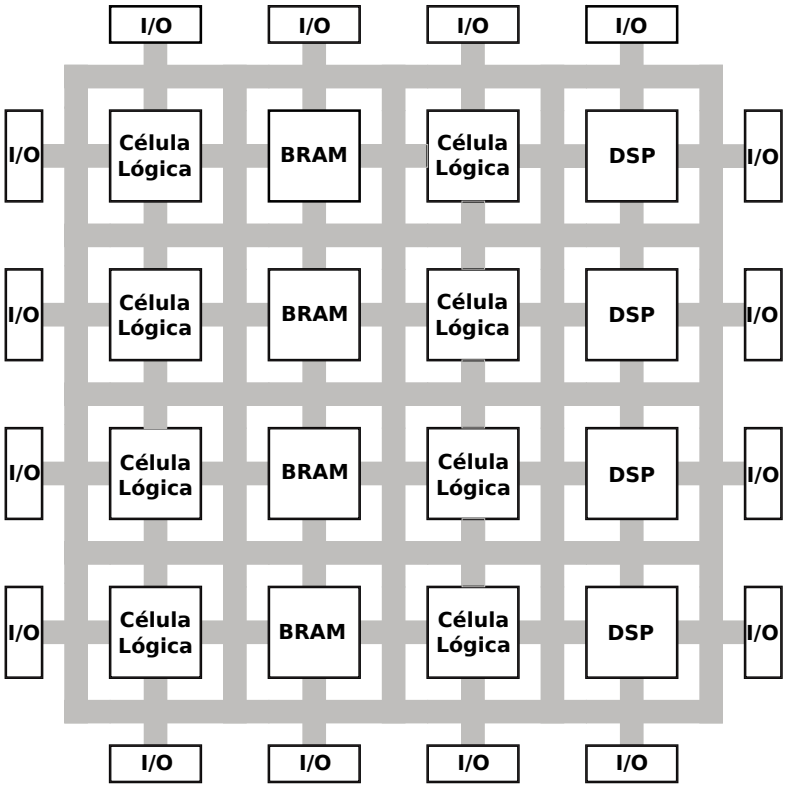


Figura 2.9: Arquitetura genérica de um FPGA

O fluxo de projeto com FPGAs é dividido em quatro etapas [54]. A primeira é criação de uma representação computacional da especificação do circuito. Esta representação é realizada com o uso de ferramentas como as linguagens de descrição de *hardware* (HDL - *Hardware Description Languages*). As duas HDL mais comuns são a Verilog e VHDL (*Very High Speed Integrated Circuit HDL*) [45].

A segunda etapa é síntese do circuito com uma ferramenta, geralmente disponibilizada pelo fabricante do próprio FPGA, que recebe como entrada o código HDL do projeto e o FPGA alvo. Como resultado, a ferramenta produz uma lista de componentes utilizados (células lógicas ou blocos lógicos especializados) e como estão interconectados. Esta lista é tradicionalmente chamada de *netlist*. Neste passo também é possível que existam otimizações de lógica e balanceamento de recursos do FPGA, que são executados pela

ferramenta com o objetivo de aumentar o desempenho do circuito projetado.

O próximo passo é executar o mapeamento e roteamento. Através da *netlist* a ferramenta aloca os componentes do projeto em células lógicas específicas do FPGA e também os conecta através das conexões hierárquicas ou conexões de roteamento de acordo a satisfazer os requisitos de tempo.

O último passo é gerar o arquivo *bit stream* para o FPGA alvo. O arquivo *bit stream* contém o resultado final do passo anterior (mapeamento e roteamento) e é carregado para a memória estática de configuração do FPGA.

2.5 Conjunto de desenvolvimento Mercurio IV

O conjunto de desenvolvimento Mercurio IV [15] é idealizado para o desenvolvimento de produtos e para o ensino e pesquisa de maneira simplificada e com baixo custo. Distribuído por Macnica DHW, é baseado no FPGA Cyclone IV EP4CE30F23 [1] da Altera e é composta por 22 componentes, como mostra a Figura 2.10, retirada de [15].

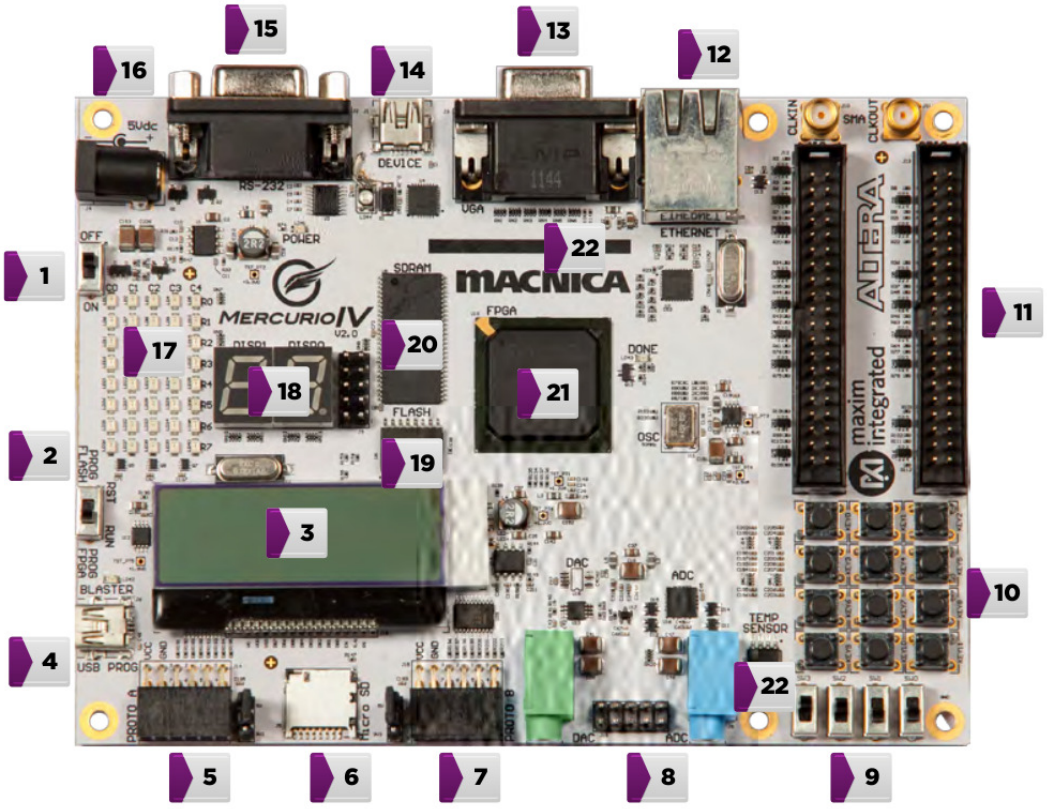


Figura 2.10: Componentes da placa Mercurio IV.

Os componentes numerados na Figura 2.10 são:

1. Botão de acionamento da placa,
2. Modos de operação JTAG ou *Active-Serial*
3. Display LCD de 2 linhas por 16 caracteres,
4. *USB_Blaster* usado para alimentação e configuração do FPGA Cyclone IV,
5. Interface para dispositivos PMOD,
6. Entrada para cartão microSD,
7. Interface PMOD para dispositivos adicionais,
8. Conversores Analógico-Digital e Digital-Analógigo,
9. Quatro chaves do tipo *slide-switch*,
10. Teclado numérico de 12 botões (0-9, * e #),
11. Dois Barramentos GPIO de 32 pinos para propósito geral,
12. Conexão ethernet,
13. Conexão VGA (conector D-SUB),
14. Conexão USB em configuração “escravo”,
15. Porta Serial no padrão RS-232,
16. LED RGB de alto brilho,
17. Matriz de LEDs de 5 colunas por 8 linhas,
18. Dois displays de sete segmentos independentes,
19. Memória Flash de 64 Mbits,
20. Memória SDRAM de 512 Mbits, interface de 16 bits e *clock* de 100 MHz,
21. FPGA Cyclone IV EP4CE30F23 com 30 mil células lógicas e *clock* de 50 MHz,
22. Sensor de temperatura.

Dos componentes da placa Mercurio IV listados, dois deles: 13-VGA e 14-USB são utilizados na plataforma desenvolvida neste trabalho. Tanto a saída VGA quanto a conexão USB são emuladas pelo cLUPA com o objetivo de simular o funcionamento dos núcleos cMIPS com a execução do *software* cLUPA.

O componente VGA é usado para enviar sinal analógico de vídeo para um monitor. Este componente é dividido em duas partes: o sinal analógico dos componentes verde,

vermelho e azul do *pixel* obtidos de um sinal de 12 bits digital (4 bits para cada componente) do FPGA. A segunda parte é um sinal digital de controle, que consiste nos sinais de sincronização vertical e horizontal.

O conjunto de desenvolvimento Mercurio IV provê um componente para o controle do VGA. O controlador-VGA, modelado em VHDL, gerencia os sinais de sincronismo horizontal e vertical, apresenta informações de atividades do componente VGA e contém dois sinais de entrada e cinco sinais de saída, como mostra a Figura 2.11. Os sinais de entrada são clock_i que recebe um sinal de *clock* com frequência de 50 MHz e reset_n_i que é um sinal de inicialização ativo em nível lógico "0" (zero). Os sinais de saída VGA_HS e VGA_VS são enviados ao componente VGA e determinam, respectivamente, o sincronismo horizontal e o sincronismo vertical. Os sinais de saída restantes são enviados ao módulo que utiliza o controlador-VGA contendo informações das atividades do componente VGA, o sinal video_on_o indica quando o componente VGA está enviando os *pixels* para exibição no monitor e os sinais linha e pixel indicam a linha e a coluna do *pixel* que está sendo exibido em derminado momento.

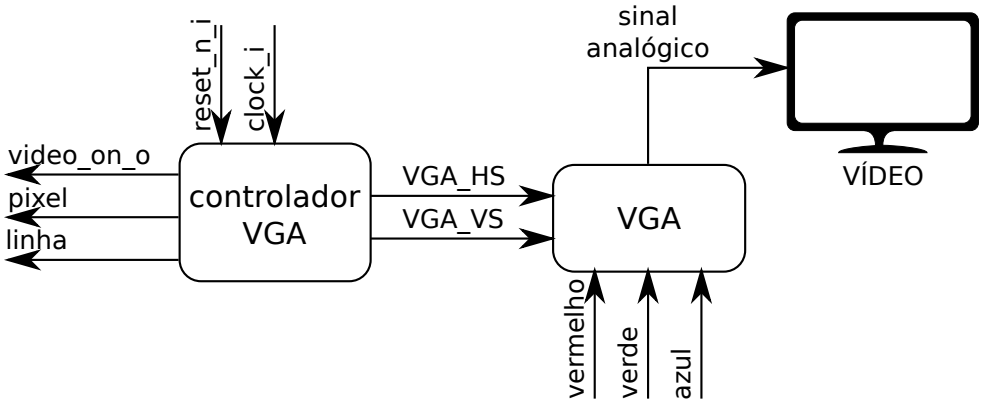


Figura 2.11: Componentes controlador-VGA e VGA disponíveis na placa Mercurio IV.

Desta forma, o módulo que envia imagens para a saída VGA da placa Mercurio IV deve verificar se o módulo VGA está enviando *pixels* ao monitor através do sinal pixel_on_o, e caso esteja ativo, deve ler os sinais linha e pixel para conhecer qual *pixel* deve ser enviado aos sinais vermelho, verde e azul.

A transferência de dados entre a placa do conjunto Mercurio IV e o ambiente externo é possível com a utilização da conexão USB. Diferente da conexão *USB_Blaster* que

apenas é utilizada para configuração do FPGA Cyclone IV e alimentação de energia, a conexão USB pode-se comunicar com qualquer dispositivo USB que opera no modo *Host*. Esta limitação existe devido ao componente FT245RQ [3, 40] ser configurado em modo de operação *Device* [15]. O componente FT245RQ é destinado a gerenciar o protocolo USB 2.0, tornando a utilização de componentes USB facilitada, pois qualquer módulo implementado para o conjunto de desenvolvimento Mercurio IV que deseja se comunicar com dispositivos USB devem apenas gerenciar o fluxo de dados.

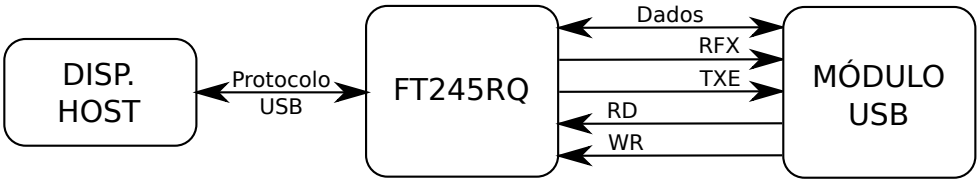


Figura 2.12: FTDI-FT245RQ disponível na placa Mercurio IV.

A Figura 2.12, adaptada de [40], mostra os sinais de comunicação entre o componente FT245RQ e um módulo USB implementado. Os sinais de informações Dados é o barramento destinado a transferir palavras de 8 bits entre o módulo USB e o FT245RQ nas direções: leitura de dados do FT245RQ por módulo USB e escrita de dados no FT245RQ por módulo USB.

Durante um processo de leitura o FT245RQ sinaliza para o módulo USB que existe uma palavra de 4 bytes pronta a ser lida através da ativação do sinal RFX. Ao detectar a ativação do sinal RFX, o módulo USB deve responder com a ativação do sinal RD. Assim, o componente FT245RQ disponibiliza a palavra no barramento de dados.

Com relação à gravação, o módulo USB deve aguardar até o componente FT245RQ sinalizar a possibilidade de gravação através da ativação do sinal TXE. Isso feito, o módulo USB coloca a palavra de 4 bytes no barramento de dados e ativa o sinal WR, indicando ao FT245RQ que os dados estão estabilizados no barramento e podem ser copiados.

2.6 Trabalhos relacionados

Esta Seção apresenta alguns trabalhos relacionados e que contribuíram com o desenvolvimento do *hardware* do cLUPA. Com relação ao *software* do cLUPA, os trabalhos

relacionados são apresentados na Seção 3.2.3.

Moch et al. [6, 44] apresentam uma arquitetura *multicore* híbrida para processamento de imagem e vídeo, que é composta por três núcleos: HIPAR-DSP otimizado para processamento de sinais digitais, MP (*Macroblock Processor*) para computação de propósito geral com uma unidade funcional para processamento vetorial; e o núcleo SP (*Stream Processor*) para computação de propósito geral com arquitetura MIPS. Os três núcleos se comunicam através de *buffers dual-ported* dispostos em anel circular entre eles, e cada núcleo pode escrever em seu *buffer* posterior e ler do seu *buffer* anterior. O barramento AMBA conecta os núcleos com as interfaces externas.

Feng, Chao e Dong [20] apresentam uma arquitetura para decodificação MPEG, composta por dois núcleos: O primeiro é um MIPS para propósito geral com extensão DSP que controla a aplicação e processamento de audio, e o segundo é um *Multimedia Engine* (MME) para processamento de vídeo. Escrito em VHDL, o protótipo foi implantando no FPGA Xilinx da família Virtex-4 e decodifica em tempo real um arquivo multimídia no formato MPEG-4 com o FPGA operando com frequência de 33MHz.

Heracles [35] é um projeto de um processador *multicore* baseado em MIPS completamente sintetizável e configurável para FPGAs escrito em Verilog. O *pipeline* do núcleo MIPS foi adaptado para conter sete estágios devido ao tempo de acesso às memórias do FPGA Xilinx da família Virtex-5 e não implementa previsor de desvios nem unidades funcionais para ponto-flutuante.

Tortato [57] descreve o MMCC (Multiprocessador Minimalista com Coerência de Cache). O MMCC é um sistema *multicore* baseado na arquitetura MIPS com pipeline de 5 estágios que se aproxima do MIPS R3000. Cada núcleo contém uma memória *cache* de nível 1 dedicada e utiliza-se dos protocolos de coerência entre memórias *cache* baseado em espionagem. O MMCC foi implementado e validado no FPGA Xilinx da família Virtex-4 e seu desempenho foi avaliado através de uma aplicação teste de multiplicação de matrizes configurando o MMCC para utilizar 1, 2, 4 e 8 núcleos.

Cardoso et al. [13] apresentam um sistema *multicore* com conjunto de instruções MIPS. A implementação do sistema *multicore*, modelado em Verilog, consiste em dois núcleos

com memória *cache* de dados dedicadas, com política *write-through*, e ligadas à memória principal por um barramento compartilhado. Para manter a coerência entre os dados mantidos pelas memórias *cache* foi utilizado o protocolo por espionagem *write-invalidade*, que a cada escrita na *cache* de dados realizada pelo núcleo invalida o dado compartilhado na outra *cache*.

A arquitetura de decodificação apresentada por Feng, Chao e Dong [20] foi pensada para realizar a decodificação MPEG-4 de forma a aproveitar o paralelismo com a divisão do processamento de audio e vídeo entre os núcleos. O processamento da aplicação do tratamento das imagens do cLUPA pode aproveitar este tipo de paralelismo. Inicialmente, foi avaliada a possibilidade de utilizar apenas um núcleo cMIPS. As simulações confirmaram a suspeita de que um núcleo, operando sozinho, não seria o suficiente para o procesamento das imagens. Assim, optou-se por utilizar dois núcleos cMIPS.

Com a utilização de dois núcleos surge o problema de coerência de dados entre suas memórias *cache*, que é possível de solucionar como apresentado nos trabalhos de Tortato [57] e Cardoso et al. [13]. Devido à limitação de tempo para a implementação do cLUPA, a funcionalidade de mapeamento da memória *cache* de dados não foi modelada, passando os sinais de requisição diretamente entre o núcleo cMIPS e a memória RAM.

O sistema híbrido de Moch et al. [44] permite a comunicação direta entre os núcleos através do uso de *buffers*. Desta forma, a comunicação entre núcleos não sobrecarrega o barramento AMBA, reduzindo a concorrência pelo acesso deste, que poderia degradar o desempenho geral do sistema. Esta é uma técnica simples e eficaz e portanto optou-se por implementar um *buffer* de comunicação entre os núcleos cMIPS para evitar a concorrência pelo acesso ao barramento entre os módulos e núcleos do do cLUPA.

O componente de *hardware* do cLUPA é um sistema *multicore* homogêneo com dois núcleos cMIPS que se comunicação através de uma memória *dual-ported*. O detalhamento do *hadware* do cLUPA é apresentado na Seção 4.1.

CAPÍTULO 3

XLUPA E XLUPA EMBARCADO

Segundo o IBGE [31], existem aproximadamente 45 milhões de brasileiros com algum tipo de deficiência visual, auditiva ou mental. A Tabela 3.1 mostra a relação entre os tipos e graus de deficiência com a totalidade da população brasileira. Observa-se que 18,76% são deficientes visuais, e destes, 18,49% são classificados como grau alto e baixo (aproximadamente 35 milhões de pessoas), que são agrupados em "baixa visão" ou "visão subnormal". Seguindo as definições da Organização Mundial de Saúde [59], são consideradas pessoas com deficiência visual extremas aquelas que são totalmente cegas ou legalmente cegas (que apresentam uma acuidade visual Snellen[1] entre 20/500 e 20/1000 ou um ângulo visual abaixo de 10 graus). Pessoas com visão subnormal baixa apresentam uma acuidade visual Snellen entre 20/70 e 20/160 e pessoas com visão subnormal alta apresentam acuidade visual Snellen entre 20/200 e 20/400 ou ângulo visual entre 10 e 20 graus.

Tabela 3.1: Pessoas com deficiência no Brasil

	Visual	Auditiva	Mental
Extrema	0,28%	0,18%	
Alta	3,18%	0,94%	1,37%
Baixa	15,31%	3,97%	
Total	18,76%	5,10%	1,37%

Levando em consideração o número de brasileiros com visão subnormal alta, aproximadamente 6 milhões, apenas recentemente nota-se o aumento no número de trabalhos voltados à soluções tecnológicas, chamadas também de "tecnologias assistivas", para auxílio deste público [8]. Uma tecnologia assistiva (TA) pode ser considerada um conjunto de recursos e serviços que contribuem para proporcionar ou ampliar habilidades funcionais de pessoas com deficiências, promovendo independência e inclusão [7, 8]. Entre as TAs

[1] Acuidade visual Snellen: técnica que determina a capacidade visual de uma pessoa comparando com a acuidade visual normal. Por exemplo, 20/100 significa que a pessoa avaliada consegue ler o texto do gráfico de Snellen, com lentes corretivas, à 20 pés (6m) o mesmo texto que uma pessoa com acuidade visual normal consegue ler à 100 pés (30m) [59]

encontram-se aquelas voltadas ao auxílio diário para pessoas com visão subnormal, tais como relógios falados e ampliadores [7].

Os ampliadores digitais são uma subclasse dos ampliadores, voltados para facilitar o uso de computadores e também para ampliar digitalmente documentos impressos [7, 9, 17]. Em relação à ampliação de documentos impressos, existem algumas soluções portáteis disponíveis, tal como o xLupa Embarcado, que é apresentado na Seção 3.2. Essas soluções portáteis são possíveis devido aos sistemas embarcados que vêm se tornando um elemento comum na vida das pessoas [41].

Este capítulo apresenta o xLupa (Seção 3.1), xLupa Embarcado (Seção 3.2), um ampliador de documentos impressos portátil em desenvolvimento que é baseado no projeto do xLupa e relaciona outros produtos com funcionalidades semelhantes.

3.1 xLupa

A tecnologia assistiva xLupa é um ampliador e leitor de tela para pessoas com visão subnormal [5, 9, 10]. O projeto do xLupa iniciou em 2004 após os autores constatarem, através do convívio no meio universitário e no ensino fundamental, o problema que envolve os alunos com baixa visão, que muitas vezes são tratados como videntes e são colocados no meio comum, sem atendimento ou auxílio especializado, e portanto, não conseguem o êxito desejado por suas instituições de ensino. Os autores observaram a necessidade do desenvolvimento de uma ferramenta de auxílio tanto para os alunos quanto para os professores em suas atividades pedagógicas [8, 9].

O xLupa é uma ferramenta de *software* livre desenvolvida em linguagem C e é executada sobre o ambiente Linux para plataforma x86. Com relação à ampliação de tela, o xLupa faz uso da biblioteca *X11* por meio do conjunto de ferramentas *GTK+* [24] e realiza uma ampliação total: após definir uma subárea da imagem, esta é ampliada para toda área disponível da tela. A leitura de tela é um recurso alternativo do xLupa que integra as linguagens de programação C e *Python* com o sintetizador de voz de código aberto e livre *Espeak*. Outra característica importante do xLupa é sua capacidade de se adaptar ao perfil do usuário através da técnica de inteligência artificial chamada de

Racionínio Baseado em Casos (RBC)[2] [5, 10].

Com o avanço da pesquisa sobre o xLupa, dois novos projetos forma iniciados: o Platmult e o xLupa Embarcado. O Platmult é uma solução integrada de *hardware* e *software* com o objetivo de prover um ambiente multisensorial em computadores ou quiosques para pessoas com baixa visão [47]. O xLupa Embarcado é um ampliador digital móvel também voltado para pessoas com baixa visão [25]. Este segundo projeto em particular utiliza uma arquitetura multiprocessada heterogênea e suas funcionalidades foram usadas como base para a plataforma proposta neste documento. Por este motivo, seu funcionamento e suas características são discutidas com maiores detalhes na Seção 3.2.

3.2 xLupa Embarcado

O xLupa Embarcado é um dispositivo de ampliação digital móvel baseado no projeto xLupa [17, 25]. O xLupa Embarcado é mostrado na Figura 3.1, adaptada de [17], e consiste em (a) uma base deslizante que contém uma câmera, (b) o dispositivo embarcado e (c) um monitor ou televisão como dispositivo de saída. Seu funcionamento requer que o usuário posicione o documento impresso a ser ampliado abaixo da base com a câmera (d), o dispositivo recupera a imagem capturada, realiza a ampliação da imagem e aplicação de contrastes e envia para o dispositivo de saída [17, 25, 48].

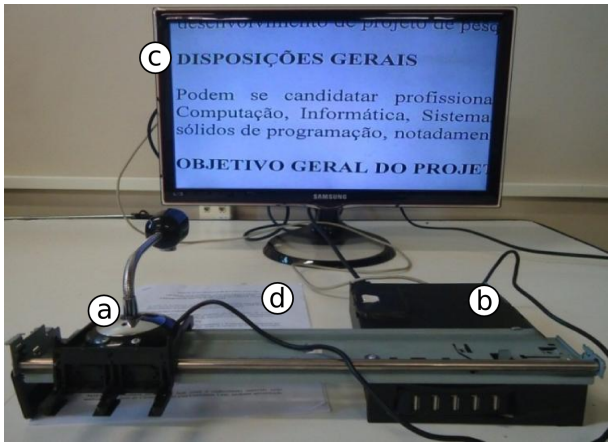


Figura 3.1: xLupa Embarcado. (a) Base com câmera, (b) dispositivo embarcado, (c) monitor/televisão e (d) documento a ser ampliado.

[2]Técnica computacional que explora a solução de problemas e o aprendizado do sistema por meio do rastreamento das experiências adquiridas com processamentos já realizados [58].

O dispositivo que realiza o tratamento das imagens foi desenvolvido com a plataforma OMAP. A adoção desta plataforma é justificada pelo baixo custo do ambiente de desenvolvimento que consiste em uma placa Beagleboard. A placa Beagleboard é composta por dois processadores, um ARM-Cortex A8 para processamento geral e distribuição de tarefas, e um processador secundário DSP C64x+, 256Mb de memória RAM, 256Mb de memória *flash*, *slot* para cartão SD (no qual pode ser instalado um sistema operacional) e provê conexões USB, HDMI, S-Video, entrada e saída de áudio estéreo, porta serial RS-232 e um conector JTAG [17, 25].

Como sistema operacional para a plataforma foi adotado o Ubuntu-Linux. A aplicação que executa as tarefas de captura das imagens, tratamento e envio para a saída é uma adaptação do código do xLupa. A adaptação do projeto original consistiu na remoção de algumas funcionalidades, tais como a capacidade adaptativa de perfis de usuários, cursores e possibilidade de configurações por *software*, tornando a aplicação mais simples e "leve" [17, 25, 48].

3.2.1 Captura e tratamento das imagens

O dispositivo deve capturar e tratar as imagens antes de enviá-las para a saída. Para capturar as imagens é utilizado a biblioteca Video 4 Linux 2 (V4L2) [42] que permite usar o nível de abstração de E/S apresentado no sistema Linux, sendo possível acessar as informações da câmera através do diretório /dev/video$X$, onde X é o número do dispositivo de E/S.

Após a captura as imagens são tratadas. O tratamento é divididos em 6 perfis classificados em dois grupos, ampliação e contraste. Com relação à ampliação, os perfis podem executá-la ou não, já o contraste é executado em uma das seguintes cores: verde, vermelho e cinza ou não executado. Existem duas exceções: não existem os perfis com contraste cinza sem aplicação de *zoom*, e nem contraste vermelho com aplicação de zoom [17, 25, 48]. Dois exemplos de execução de contraste são mostrados na Figura 3.2, adaptada de [17]. Nela é possível observar que o texto é apresentado com a cor preta e o fundo na cor definida pelo contraste.

(a) Contraste Vermelho (b) Contraste Verde

Figura 3.2: Exemplos de aplicação de contrastes

A ampliação das imagens é executada através do uso da biblioteca *Cairo* [11] em conjunto com o *GTK+*. *Cairo* é uma biblioteca gráfica para transformações 2D em imagens. O processo inicia-se com a captura do quadro armazenado pelo *GTK+* e uma transformação de escala é aplicado pelo *Cairo* através da função cairo_scale e o resultado é enviado para o *GTK+* para ser exibido em modo "tela cheia".

O contraste é executado através do cálculo do brilho da imagem (luminância) k, dado por $k = 0.299 * r + 0.587 * g + 0.114 * b$, sobre cada *pixel* da imagem e aplicado para determinar se o *pixel* é fundo de imagem ou texto através da comparação com o valor médio das componentes r (vermelho), g (verde) e b (azul). Para capturar cada *pixel* da imagem são utilizadas duas funções da biblioteca *GTK+*, a GET_PIXEL24 que recupera o *pixel* e a PUT_PIXEL24 que devolve *pixel* processado em sua posição original.

3.2.2 Desempenho do xLupa Embarcado

A avaliação de desempenho do xLupaEmbarcado foi realizada através de medições de tempo das tarefas de captura, processamento e exibição das imagens em duas versões de aplicação: ARM, que se utiliza unicamente do processador ARM Cortex-A8 e DSP que utiliza o processador DSP C64x+ para o processamento das imagens. Os resultados destas medições, dado em quadros por segundo, são apresentados na tabela 3.2, adaptada de [16].

Tabela 3.2: Desempenho do xLupa Embarcado (quadros por segundo)

	ARM	DSP
Ampliação Desabilitada	6,00	4,55
Ampliação Abilitada	3,09	3,16
Contraste Cinza	6,64	4,50
Contraste Verde	6,67	5,00
Contraste Verde com Ampliação	3,27	3,17
Contraste vermelho	6,64	4,98

3.2.3 Trabalhos relacionados ao xLupa Embarcado

Existem produtos semelhantes ao xLupa Embarcado disponíveis no mercado que variam em funções, estrutura e mobilidade. A seguir são descritos quatro produtos e suas principais diferenças com relação ao xLupa Embarcado.

O Vision Booster Magnifier, produzido por Rx Optics, é um ampliador sem função de aplicação de contrastes. Semelhante a um *mouse* convencional, porém com uma interface em seu topo para definição do fator de zoom, requer o uso de um computador com sistema operacional *Windows* para seu funcionamento [51].

Desenvolvido pela Carson Optical [32], o DR-200 ezRead Eletronic Reading Aid, mostrada na Figura 3.3a, requer apenas um televisor com entrada vídeo-RCA para funcionar. É semelhante ao Vision Booster Magnifier em funcionalidade e estrutura: tem um tamanho aproximado de um *mouse* e apenas amplia os documentos com um fator fixo [51].

O Electronic Reading Aid, produzido por Mattingly Low Vision [33], é semelhante ao DR-200, com funcionalidade de congelar e descongelar a imagem atual através de um único botão e a capacidade de aplicar dois tipos de alto contrastes, positivo e negativo [51].

A empresa Freedom Scientific produz algumas soluções de ampliação digital. A ONYX Swing-arm Portable Magnification Camera [53] é uma delas, mostrada na Figura 3.3b, consiste em um braço que contém uma câmera em seu extremo. O processamento é feito no próprio equipamento que requer apenas um televisor para seu funcionamento. Também permite quatro tipos de contrastes: colorido, inverso-colorido, preto em fundo branco e branco em fundo preto [53].

O *software* do cLUPA foi pensado tomando como base o xLupa Embarcado, porém algumas características dos ampliadores relatados nesta seção foram considerados.

(a) DR-200 ezRead (b) ONYX Swing-arm

Figura 3.3: Exemplos de trabalhos relacionados ao xLupa Embarcado

O Vision Booster Magnifier requer um computador com sistema operacional *Windows* para seu funcionamento, limitando sua portabilidade, que é uma característica importante para o cLUPA, assim, foi tomado a decisão de utilizar o *software* do cLUPA diretamente sobre *hardware*, evitando o uso de um sistema operacional e de bibliotecas, e consequentemente, evitando a carga computacional imposta pela utilização destes módulos.

Algumas pessoas com baixa visão podem sofrer também de algum tipo de deficiência física, limitando seus movimentos e impedindo o uso preciso dos ampliadores Vision Booster Magnifier, DR-200 e Eletronic Reading Aid. Assim, o braço com câmera presente no ONYX Swing-arm Portable Magnification Camera é um recurso que evita constantes movimentações do equipamento sobre o documento impresso, bastando posicionar o documento na área abaixo da câmera, algo semelhante ao xLupa Embarcado. Assim, optou-se por utilizar um braço com uma câmera na plataforma proposta do cLUPA.

CAPÍTULO 4

cLUPA - AMPLIADOR DIGITAL DE DOCUMENTOS IMPRESSOS

O cLUPA é um ampliador digital de documentos impressos baseado no xLupa Embarcado. A Figura 4.1 mostra a visão geral do sistema que consite em quatro elementos principais:

A: Documento impresso a ser tratado pelo cLUPA, posicionado abaixo da câmera;

B: Braço de sustentação com câmera;

C: Módulo de processamento do cLUPA que contém os botões que definem a aplicação de ampliação e contraste (representados pelas letras Z e C) e é ligado à câmera e ao monitor por cabos USB e VGA, respectivamente;

D: Monitor destinado a exibir a imagem capturada pela câmera e tratada por cLUPA.

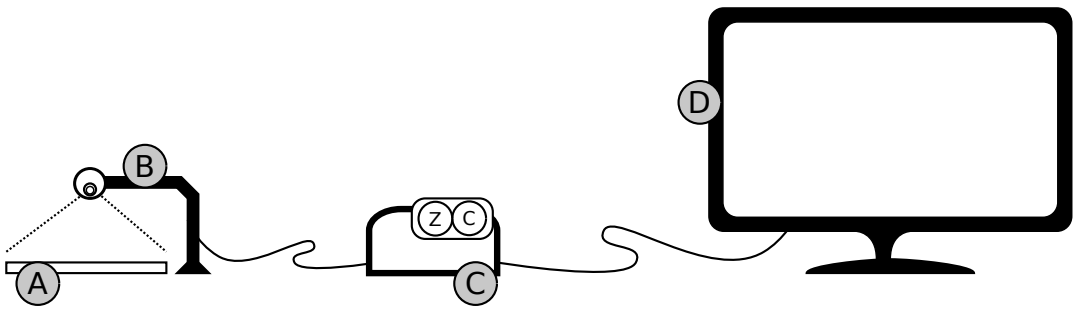


Figura 4.1: Visão geral da platafoma cLUPA.

O modo de usar o cLUPA consiste em posicionar o documento impresso abaixo da câmera, tarefa que é realizada pelo usuário. A câmera captura a imagem e envia ao módulo de processamento que verifica qual configuração de ampliação e contraste está ativa e a aplica, e ao fim do processamento, envia a imagem tratada para o monitor. O usuário deve movimentar o documento impresso no sentido vertical em relação à câmera de acordo com a posição desejada de leitura (início, meio ou fim do documento) e no sentido horizontal caso a configuração de ampliação esteja ativada.

O objetivo deste Capítulo é descrever o *hardware* (Seção 4.1) e *software* (Seção 4.2) do cLUPA, adições nos núcleos cMIPS (Seção 4.1.1), dispositivo de comunicação entre núcleos (Seção 4.1.2), dispositivos de acesso direto a memória (Seção 4.1.3), HPI (Seção 4.2.1), módulos de *software* que executam nos núcleos cMIPS (Seção 4.2.2) e algumas adaptações realizadas no código do xLupa Embarcado (Seção 4.2.3).

4.1 Hardware

A plataforma de *hardware* do cLUPA mostrada na Figura 4.1 é composta por três partes: a câmera de captura, o módulo de processamento cLUPA e o monitor de vídeo. O módulo de processamento é o componente que realiza a tarefa de receber as imagens da câmera, tratá-las de acordo com as configurações de contraste e ampliação, para depois enviá-las ao dispositivo de saída.

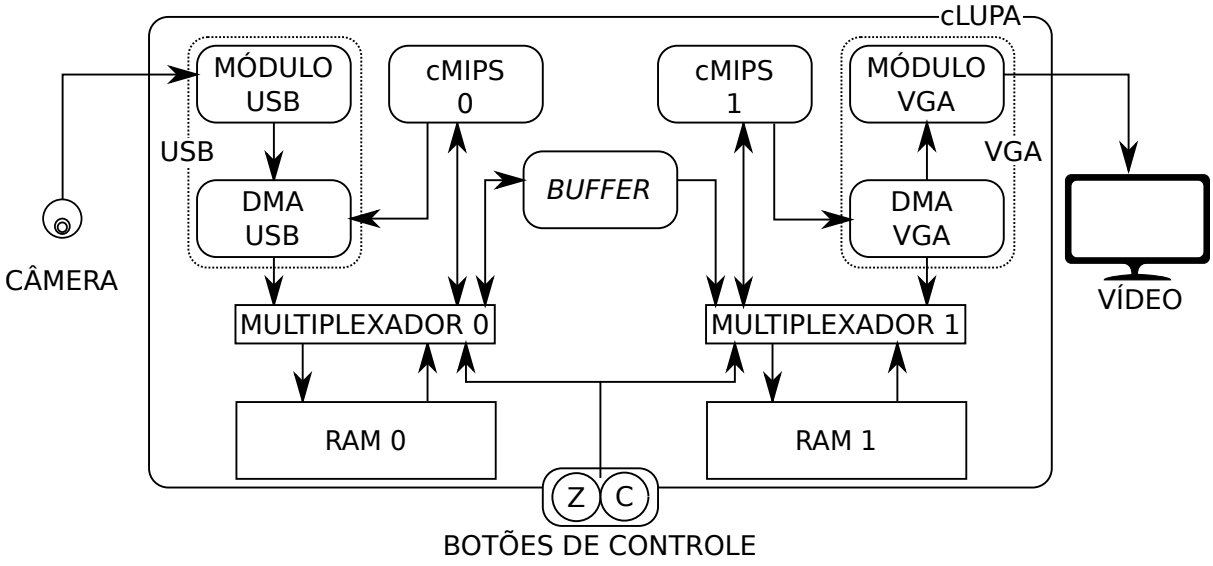


Figura 4.2: Plataforma de *hardware* do cLUPA.

O módulo de processamento do cLUPA, apresentado na Figura 4.2, é composto por doze componentes. O Módulo-USB efetua a comunicação da câmera com o DMA-USB e implementa o protocolo USB. O DMA-USB contém registradores que capturam quatro bytes vindos do Módulo-USB, que formam um *pixel*, e os envia para a memória RAM-0. Os componentes Módulo-USB e DMA-USB foram encapsulados em um único componente chamado USB. No caminho crítico entre o DMA-USB e RAM-0 encontra-se o árbitro do

MULTIPLEXADOR-0 que dá prioridade de acesso à memória para o componente DMA-USB.

Um conjunto de *pixels* gravados na RAM-0 é lido pelo núcleo cMIPS-0, esse conjunto é definido pela configuração de ampliação, na qual, *pixel*-a-*pixel* é aplicada a configuração de contraste, e após o processamento, são armazenados no BUFFER. Este é uma fila de dados que é usada como ponte de comunicação entre os dois núcleos, apresentado na Seção 4.1.2. O núcleo cMIPS-1 lê um *pixel* do BUFFER e calcula seu endereço para finalizar a aplicação de ampliação, se assim estiver configurado, e o armazena na memória RAM-1.

O DMA-VGA lê todos os *pixels* da imagem processada da memória RAM-1 e os transfere para o Módulo-VGA. No caminho crítico entre o DMA-VGA e RAM-1 existe o árbitro do MULTIPLEXADOR-1 que dá prioridade de acesso à memória para o componente DMA-VGA. O módulo-VGA contém os sinais analógicos VGA de temporização que permitem a exibição no dispositivo de vídeo. Os componentes Módulo-VGA e DMA-VGA foram encapsulados em um único componente chamado VGA.

O componente Botões-de-Controle é a forma com que o usuário configura o ampliação e o contraste. O botão de ampliação, indicado pela letra Z (zoom), ao ser pressionado uma vez ativa a configuração de ampliação, e se pressionado novamente, a desativa. Já o botão de contraste, indicado pela letra C, ao ser pressionado uma vez, altera a configuração de contraste para a próxima configuração de uma lista circular, seguindo a sequência: INATIVO, VERMELHO, VERDE e CINZA.

Os componentes de entrada e saída de dados Módulo-USB, DMA-USB, Módulo-VGA, DMA-VGA e Botões-de-Controle são discutidos na Seção 4.1.3.

4.1.1 Novos componentes adicionados à periferia do cMIPS

O cMIPS recebeu ligações em sua periferia (arquivo de simulação) para suportar os novos componentes de *hardware*. Essas adições são: cMIPS-0 recebeu ligações no MULTIPLEXADOR-0 para se comunicar com o componente DMA-USB e escrever palavras no BUFFER, e o cMIPS-1 recebeu ligações no MULTIPLEXADOR-0 para se comunicar com o componente DMA-VGA e ler palavras do BUFFER.

Foi necessário adicionar um árbitro, representado na Figura 4.3 por dois componentes, SEL-CD/DMA e MULTIPLEXADOR-DMA, entre o barramento principal MULTIPLEXADOR e a memória RAM, pois existe a concorrência com o dispositivo DMA. O componente SEL-CD/DMA dá prioridade de acesso à memória RAM para o módulo DMA, pois este se encontra em um caminho crítico: o fluxo de bytes originado do Módulo-USB e o fluxo de bytes entregue ao Módulo-VGA não podem ser interrompidos, podendo ocorrer falhas nas imagens caso a prioridade seja dada a outro componente. Desta forma, se o componente SEL-CD/DMA detectar os sinais mem_d_sel_dma e cpu_sel_ram ativos no mesmo ciclo de relógio, o SEL-CD/DMA ativa o sinal dma_sel, que faz com que o caminho DMA para RAM seja liberado no MULTIPLEXADOR-DMA.

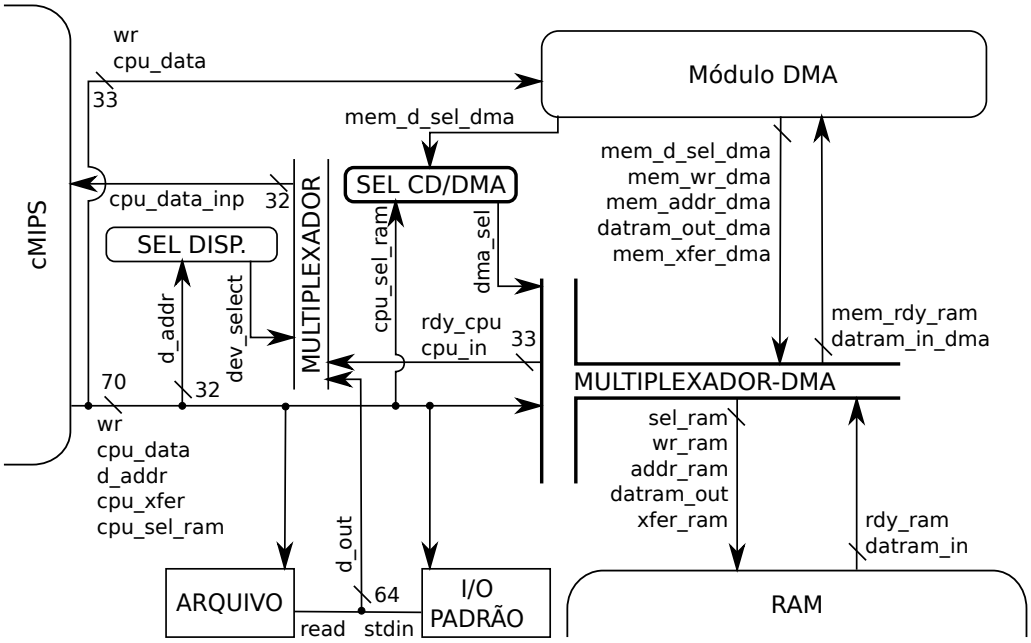


Figura 4.3: Diagrama de blocos do módulo de teste cMIPS com módulo DMA.

Quando houver requisições concorrentes para acesso à RAM pelo DMA e pelo cMIPS, o barramento envia sinais de controle ao núcleo cMIPS indicando que a memória RAM está ocupada. Esses sinais permanecem ativos até que o módulo DMA termine sua transferência.

Foram adicionados quatro novos dispositivos na unidade de decodificação de endereços, representada pelo módulo SEL-DISP na Figura 4.3: bcdW-Sel que indica o sinal de seleção do módulo de escrita de dados do BUFFER, bcdR-Sel que indica o sinal de seleção do

módulo de leitura de dados do BUFFER, dmaUSB-sel e dmaVGA-sel que indicam, respectivamente, os sinais de seleção dos módulos DMA-USB e DMA-VGA. A unidade de decodificação é replicada para os dois núcleos, porém o núcleo cMIPS-0 acessa apenas o módulo de escrita de dados do BUFFER e DMA-USB, e o núcleo cMIPS-1 apenas acessa o módulo de leitura de dado do BUFFER e DMA-VGA. Assim, é responsabilidade do desenvolvedor do software gerenciar os acessos corretamente, podendo gerar congelamentos durante a execução no evento de algum núcleo cMIPS referenciar um dispositivo a que não tem acesso.

Os demais componentes, apresentados na Figura 2.3, I/O-PADRÃO e ARQUIVO foram replicados com exceção do BUFFER, que é compartilhado entre os núcleos (Figura 4.2).

Como discutido na Seção 2.6, a memória *cache* de dados não foi implementada. Assim, todos os acessos realizados pelo cMIPS aos dispositivos e memória são transmitidos diretamente para o MULTIPLEXADOR.

4.1.2 *Buffer* de comunicação direta

O *Buffer* de Comunicação Direta (BCD), mostrado na Figua 4.4 (componente BUFFER), é o componente que permite a comunicação do núcleo cMIPS-0 para o núcleo cMIPS-1. O cLUPA utiliza o BCD para transmitir cada *pixel* processado pelo núcleo cMIPS-0 para o núcleo cMIPS-1, que o armazena em seu endereço final para exibição na tela.

O BCD é ligado aos núcleos cMIPS como sendo um periférico ligado aos barramentos principais de periféricos MULTIPLEXADOR-0 e MULTIPLEXADOR-1. O cMIPS-0 realiza gravação de palavras ou leitura de estado, enquanto que o cMIPS-1 realiza apenas leitura de palavras ou leitura de estado.

Os componentes do BCD são: RAM-FIFO é a memória com comportamento do tipo FIFO (*First-In-First-Out*) com dois sinais de dados, sendo um para gravação (ws_d_in) e outro para leitura (rs_d_out_A), dois sinais de endereçamento, um para o endereço da palavra que será gravada (ws_cw_addr) e outro para o endereço da palavra que será lida (rs_cw_addr), e sinais de controle que indicam quando uma gravação deve ocorrer (ws_we2) e quando uma leitura (rs_rd) deve ser realizada.

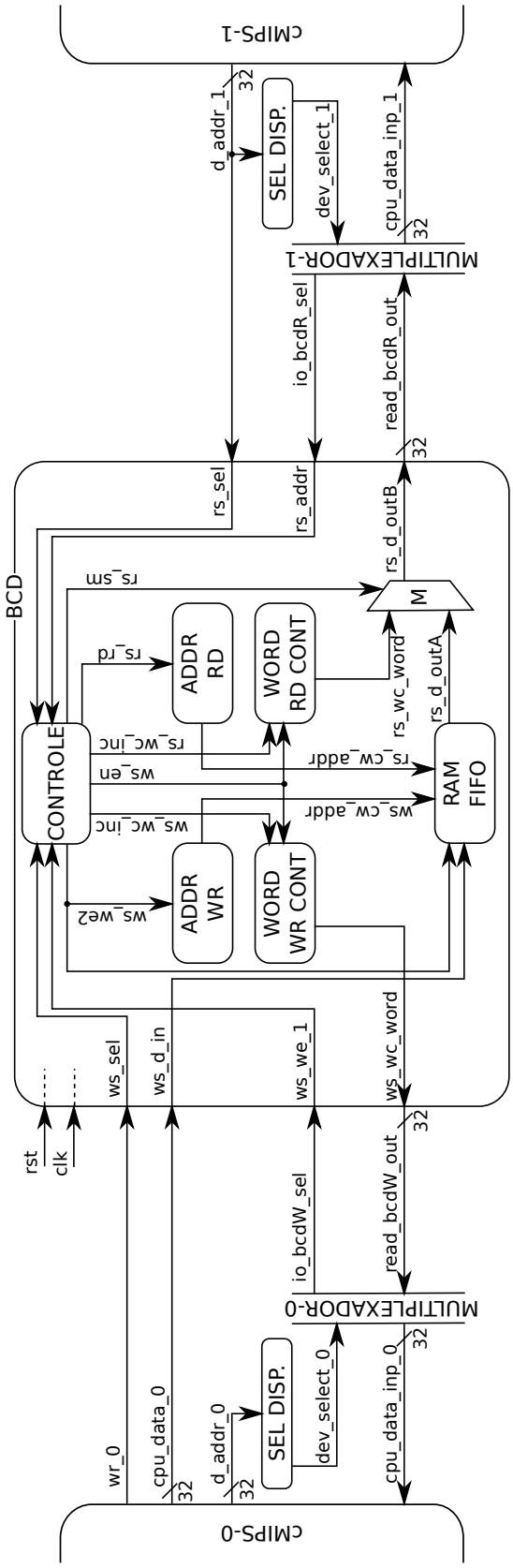


Figura 4.4: *Buffer* de Comunicação Direta.

Os componentes ADDR-WR e ADDR-RD são contadores cíclicos que mantém um endereço de gravação e um de leitura, respectivamente. Cada componente contém um sinal de saída de endereços ligada à RAM-FIFO (ws_cw_addr e rs_cw_addr) e sinais de controle que indicam período ativo para incremento de endereço (ws_we2 e rs_rd). O endereço de leitura sempre deve ser um valor anterior ao endereço de gravação, e durante um ciclo dos contadores, o endereço de leitura deve ser menor que o endereço de gravação. O desenvolvedor do *software* deve gerenciar, através das leituras de estados, as leituras e gravações, pois não foi implementado em *hardware* um controle que identifica endereçamento incorreto.

Com relação ao estado do BCD, dois componentes mantêm as informações de estado: WORD-WR-CONT e WORD-RD-CONT. WORD-WR-CONT refere-se ao número de palavras livres existentes na RAM-FIFO para gravação e é incrementado sempre que uma leitura é efetuada ou decrementado quando uma gravação é realizada. WORD-RD-CONT refere-se ao número de palavras gravadas na RAM-FIFO e disponíveis para leitura e é incrementado sempre que uma gravação é realizada e decrementado quando uma leitura é efetuada. WORD-RD-CONT concorre com a RAM-FIFO pelo acesso ao barramento MULTIPLEXADOR-1 e suas saídas de dados são selecionadas pelo componente multiplexador M.

O componente CONTROLE lê os sinais de ativação (w_sel e r_sel), sinais de endereçamento (r_addr) e leitura/gravação do BCD (w_we) originados dos núcleos e seleciona os sinais de controle de gravação da RAM-FIFO (ws_we2), sinais de incremento dos componentes de endereçamento (ws_we2 e rs_rd), sinais de incremento/decremento dos gerenciadores de estado (wc_en, ws_cw_inc e rs_cw_inc) e sinal de seleção de saída (rs_sm) para o compomente multiplexador M de acordo com as operações de escrita e/ou leitura do BCD.

4.1.3 Módulos de acesso direto à memória e botões de controle

A comunicação do cLUPA com o meio externo (câmera, monitor e usuário) é realizada através dos módulos USB, VGA e Botões-de-Controle, respectivamente, e foram implementados de modo a simular componentes reais.

O módulo de entrada de dados USB (Módulo-USB em conjunto com o módulo DMA-USB) foi criado a partir do componente FDTI FT245RQ disponível no conjunto de desenvolvimento Mercurio IV (Seção 2.5) e do barramento no cMIPS. A Figura 4.5 mostra este módulo com seus componentes e como estão ligados no arquivo de teste do cMIPS.

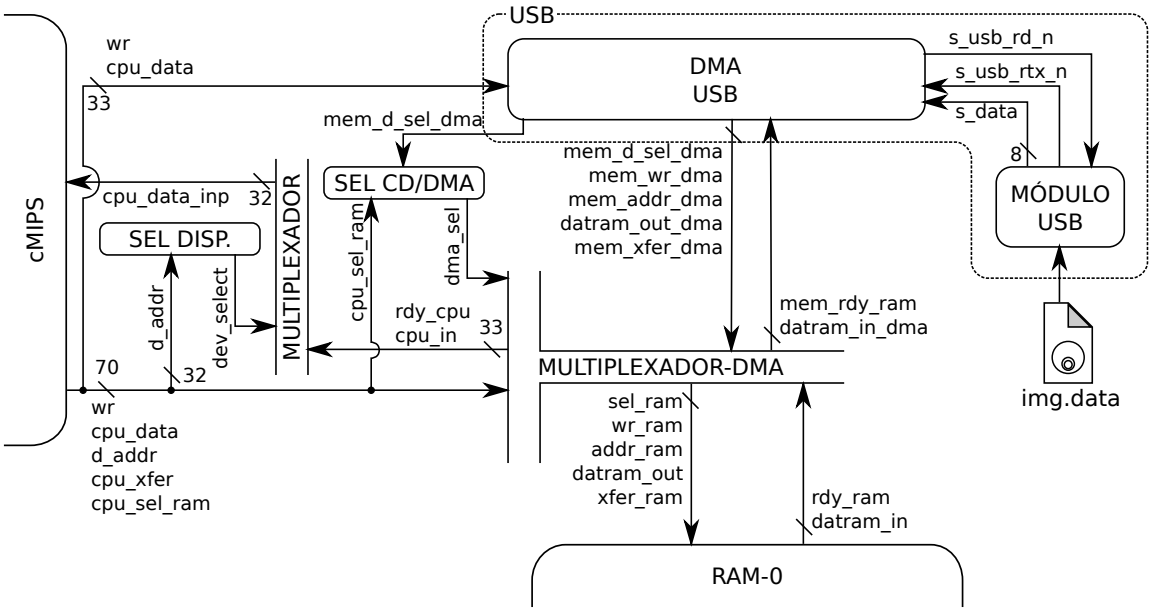


Figura 4.5: Módulo de entrada de dados USB.

O componente Módulo-USB transfere os *pixels* da câmera para os registradores do componente DMA-USB e foi implementado seguindo as especificações do componente FDTI FT245RQ. Assim, o Módulo-USB contém o sinal de controle s_usb_rd_n, que indica que existem bytes a serem lidos, e o sinal s_usb_rfx_n que inicia uma transferência de um byte usando o sinal de 8 bits s_data.

O componente DMA-USB trabalha como um *buffer* temporário para uma palavra de quatro bytes recebida do Módulo-USB. Assim que DMA-USB detecta um valor ativo no sinal s_usb_rd_n, inicia uma transferência de dados do Módulo-USB ativando o sinal s_usb_rfx_n. Cada byte é armazenado em uma posição de um registrador de quatro bytes, e após a conclusão desta transferência, que representa um *pixel*, esta palavra é enviada a um segundo registrador, liberando o primeiro para receber um novo *pixel*. Enquanto a transferência entre o Módulo-USB e DMA-USB acontece para o novo pixel, o DMA-USB se encarrega de enviar a palavra do segundo registrador para a memória RAM-0.

Para controlar todo o processo de transferência e armazenamento, o componente DMA-

USB conta com duas máquinas de estado, uma para receber os bytes do Módulo-USB e outra para enviar à RAM-0. É necessário que o núcleo cMIPS-0 inicie o DMA-USB com dois dados de entrada: o endereço inicial da RAM-0 onde deve ser armazenado o primeiro pixel e a quantidade de palavras (tamanho, em *pixel*, da imagem).

Para efeitos de simulação, o componente Módulo-USB recupera cada byte de um arquivo binário chamado img.data, que contém apenas os *pixels* de uma imagem no formato bitmap de 32 bits.

O módulo de saída de dados VGA (Módulo-VGA em conjunto com o componente DMA-VGA) foi criado a partir de componentes disponível no conjunto de desenvolvimento Mercurio IV: Controlador VGA e componente VGA, e do barramento do cMIPS. A Figura 4.6 apresenta o módulo criado e seus componentes.

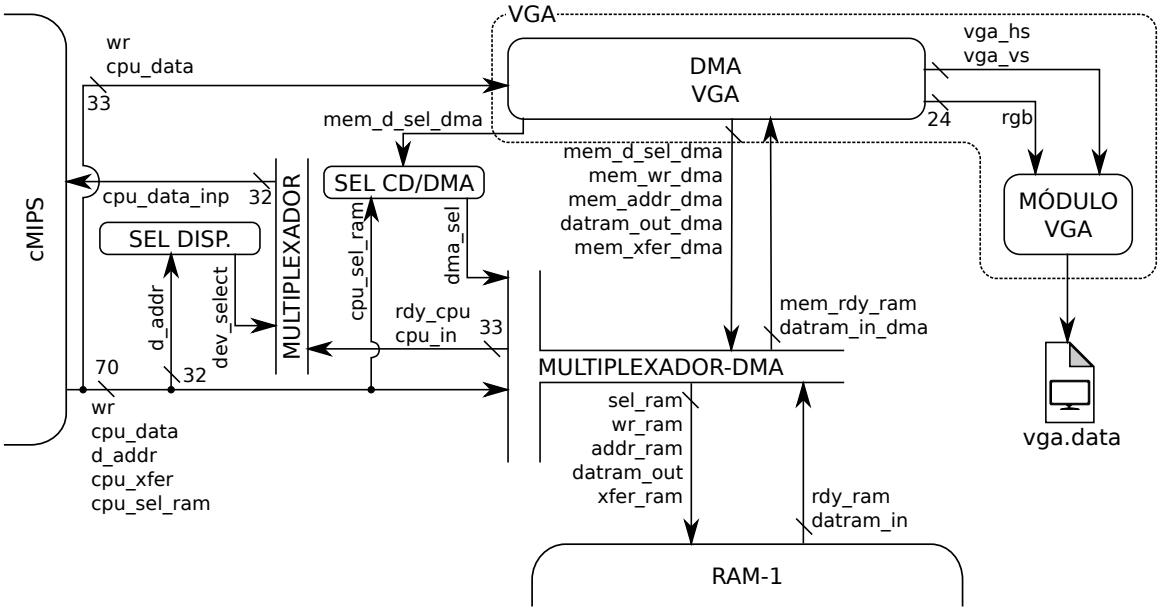


Figura 4.6: Módulo de saída de dados VGA.

O componente Módulo-VGA recebe três bytes, através do sinal rgb, que representam as componentes de cor verde, vermelho e azul (componente de transparência é ignorado) e converte em sinais analógicos reconhecidos pelo dispositivo de saída. Como sinais de controle, existem apenas dois sinais de entrada: vga_hs que representa o sincronismo horizontal e vga_vs que representa o sincronismo vertical. Esses sinais devem permanecer ativos durante o tempo em que os *pixels* são enviados ao dispositivo de saída. Para fins de simulação, o Módulo-VGA escreve os pixels em um arquivo binário chamado vga.data.

O componente DMA-VGA lê cada *pixel* processado e armazenado na RAM-1 e envia para o Módulo-VGA. Este componente contém encapsulado o Controlador-VGA disponível no conjunto de desenvolvimento Mercurio IV. Assim, somente uma máquina de estados é necessária, que efetua leitura da memória e permite que o *pixel* lido esteja disponível no registrador de envio para Módulo-VGA.

O componente de configuração Botões-de-Controle é a interface da plataforma cLUPA com o usuário e permite ativar ou desativar a aplicação de ampliação e também uma das configurações de contraste disponíveis e é composta por dois botões: Z e C. A Figura 4.7 mostra este componente e os módulos internos.

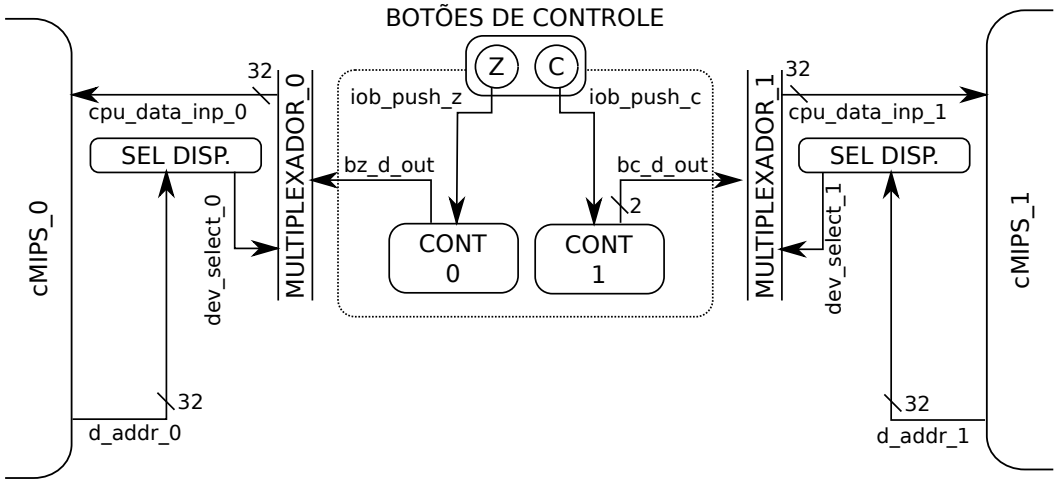


Figura 4.7: Módulo de interface com o usuário.

O componente Botões-de-Controle é composto por apenas dois contadores cíclicos CONT-0 e CONT-1. O contador CONT-0 contém um único bit para determinar se a configuração de ampliação está ativada (ou desativada) e é enviado ao núcleo cMIPS-0 através do sinal bz_d_out. O contador CONT-1 contém dois bits para determinar uma das quatro configurações de contraste disponíveis na ordem INATIVO, VERMELHO, VERDE e CINZA e é enviado ao núcleo cMIPS-1 pelo sinal de 2 bits bc_d_out. O comportamento dos contadores é simples: apenas incrementam seus valores ao encontrar um sinal ativo de algum dos botões Z e/ou C. O ato de manter algum dos botões pressionados não incrementa os contadores de forma indefinida, somente o ato de pressionar é que o fará. Para realização das simulações, o componente Botões-de-Controle foi desativado, no lugar é usado a entrada padrão do cMIPS, que é a leitura dos arquivos input.data e input1.data.

As entidades escritas em VHDL dos componentes `BCD`, `USB` e `VGA` encontram-se no Apêndice A.

4.2 Software

O *hardware* do cLUPA adiciona novos componentes ao cMIPS, consequentemente, adiciona novos recursos disponíveis ao *software*, desta forma é necessária a implementação de funções HPI, apresentadas na Seção 4.2.1.

No início do projeto do cLUPA foi avaliada a possibilidade de utilizar diretamente o *software* do xLupa Embarcado. Porém, o *hardware* do cLUPA foi considerado insuficiente para executar, de forma eficiente, um sistema operacional com suporte as bibliotecas GTK+, Cairo e V4L2. Assim, optou-se por implementar novo código baseado nas funcionalidade do xLupa Embarcado que executasse diretamente sobre o *hardware* do cLUPA. Este é apresentado na Seção 4.2.2.

4.2.1 HPI para os módulos USB, VGA e BUFFER

Para integrar os novos módulos de *hardware* ao *software*, foram adicionadas nove funções ao arquivo cMIPS.c: quatro funções para o uso do BUFFER, duas para o módulo USB e três para o módulo VGA. Essas funções são mostradas no Quadro 4.1.

Os núcleos cMIPS-0 e cMIPS-1 acessam todas as nove funções, porém, cMIPS-0 não efetua as chamadas para o módulo de leitura de dados do BUFFER e módulo VGA, enquanto o cMIPS-1 não efetua as chamadas para o módulo de escrita de dados do BUFFER e módulo USB, pois se assim o fizerem, suas execuções ficarão em espera à respostas de dispositivos a que não têm acesso.

A função bcdWWr(int n) escreve um inteiro n no BUFFER. É necessario verificar se existe ao menos uma palavra disponível com a função bcdWrSt(), que retornará o número de palavras que podem serem escritas. O Quadro 4.2 mostra um exemplo de código que impede a continuação da execução, e consequentemente a gravação no BUFFER, até uma palavra no BUFFER tornar-se disponível.

```
1  // BUFFER ==========================================================
2  void bcdWWr(int n); // Escreve um inteiro no módulo de escrita de dados
3  int  bcdWSt(void);  // Lê o estado do módulo de escrita de dados
4  int  bcdRRd(void);  // Lê um inteiro do módulo de leitura de dados
5  int  bcdRSt(void);  // Lê o estado do módulo de leitura de dados
6  // USB =============================================================
7  /*  Configura o módulo USB para transferência
8        int a: Endereço do primeiro byte
9        int w: Número de palavras a serem transferidas
10       int s: Tamanho, em bytes, de cada palavra */
11 void dmaUSB_init(int a, int w, int s);
12 int  dmaUSB_st();          // Estado do módulo USB
13 // VGA =============================================================
14 /*  Configura o módulo VGA para transferência
15       int a: Endereço do primeiro byte
16       int w: Número de palavras a serem transferidas
17       int s: Tamanho, em bytes, de cada palavra */
18 void dmaVGA_init(int a, int w, int s);
19 int  dmaVGA_st();          // Estado do módulo VGA
20 void dmaVGA_closeFile(); // Fecha arquivo de saída da simulação VGA'
21                          // 'Usado somente para simulações
```

Quadro 4.1: HPI para novos módulos CLUPA

A função bcdRRd() lê um inteiro n do BUFFER. É necessário verificar se existe algum valor válido gravado no BUFFER com a função bcdRSt(), que retornará o número de palavras válidas para leitura. O Quadro 4.3 mostra um exemplo de código para leitura de dados do BUFFER.

```
1 int bcd_Wr_max = bcdWSt();
2 // Retém execução até uma palavra estar disponível para escrita
3 while(bcd_Wr_max <= 0){
4   bcd_Wr_max = bcdWSt(); }
5 bcdWWRr(n); // Efetua a gravação no BUFFER
```

Quadro 4.2: Código para escrita de dados no BUFFER

```
bcd_Rd_max = bcdRSt();
// Retém execução até uma palavra estar disponível para leitura
while(bcd_Rd_max <= 0){
  bcd_Rd_max = bcdRSt();}
int n = bcdRRd(); // Efetua a leitura do BUFFER
```

Quadro 4.3: Código para leitura de dados do BUFFER

Os módulos DMA-USB e DMA-VGA são inicializados indicando o endereço inicial da memória RAM para receber ou ler os dados, o número de palavras escritas ou lidas e o tamanho, em bytes, de cada palavra. O Quadro 4.4 mostra o inicio de duas transferências: a primeira efetuada pelo módulo USB que guardará cem palavras de quatro bytes a partir do endereço 1.000 da RAM, em seguida, é aguardada a finalização da transferência antes de iniciar a execução do módulo VGA, que lerá quatrocentos bytes a partir do endereço 1.000.

```
int I = 1000; B = 4; T = 100;
// Transferência USB -> RAM no endereço I de T palavras de B bytes
dmaUSB_init(I, B, T);
dma_t_ok = dmaUSB_st();
while(dma_t_ok != 0){      // Aguarda transferência USB finalizar
  dma_t_ok = dmaUSB_st();  }
// Transferência RAM -> VGA no endereço I de T palavras de B bytes
dmaVGA_init(I, B, T);
```

Quadro 4.4: Código para inicialização dos módulos DMAs

As funções restantes para a leitura de estado dos módulos DMA retornam o valor "1" para transferência ativa, ou "0" para módulo ocioso e devem ser usadas caso seja necessário executar várias transferências consecutivas ou iniciar o processamento dos dados recentemente transferidos.

4.2.2 cLUPA

O código do cLUPA foi escrito com base nas funcionalidades do xLupa Embarcado e na divisão de tarefas apropriada ao *hardware* apresentado na Seção 4.1. Desta forma,

o cLUPA processa imagens nas seis configurações de ampliação e contraste possíveis do xLUPA Embarcado e mais duas configurações adicionais, totalizando oito configurações: ampliação ativa para contraste cinza, vermelho, verde ou contraste desativado e ampliação desativada para contraste cinza, vermelho, verde ou contraste desativado.

A aplicação das configurações foi dividida entre os dois núcleos cMIPS do cLUPA. O núcleo cMIPS-0 executa o módulo cLUPA-0 que controla o módulo USB, inicia os cálculos de endereços dos *pixels* (ampliação ativa), aplicação de contraste e armazenamento no BUFFER. O núcleo cMIPS-1 executa o módulo cLUPA-1 que lê do BUFFER, finaliza os cálculos de endereços dos *pixels* e controla o módulo VGA. Estas atividades são mostradas nas Figuras 4.8 e 4.9, respectivamente.

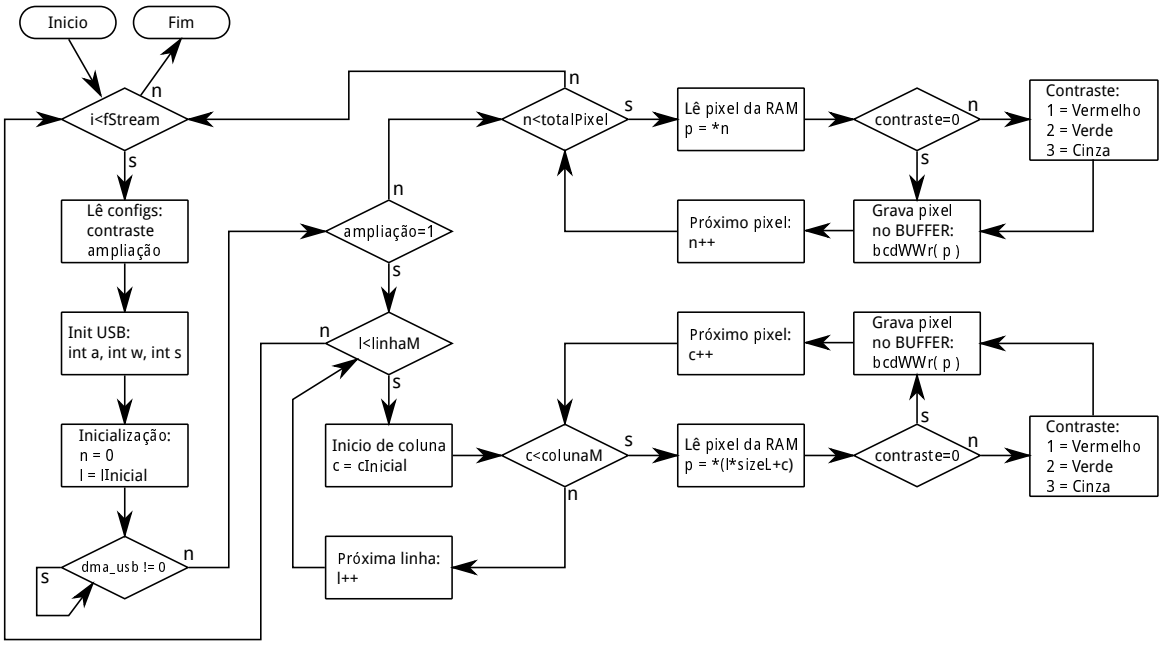


Figura 4.8: Fluxograma cLUPA-0 (Módulo do núcleo cMIPS-0).

Para as simulações, foi definido um número máximo i=3 de imagens a serem processadas, representado pelo laço de repetição fStream. O cLUPA-0 executa a leitura das configurações de ampliação e contraste e inicia o módulo USB invocando a função Init USB. Enquanto espera pelo final da tansmissão da primeira imagem, a aplicação realiza os ajustes de configuração inicial, entre estas configurações estão os valores iniciais de deslocamento do endereço dos *pixels* para imagem sem ampliação (n=0) e com ampliação (l=lInicial). Em seguida, é avaliado se a execução desta imagem sofrerá ampliação, re-

presentado pela decisão ampliação=1 na Figura 4.8. Caso a ampliação esteja desativada, todos os *pixels* da imagem são lidos através do deslocamento em n do endereço inicial da imagem. Caso a ampliação esteja ativa, os *pixels* lidos são reduzidos à 25% do total de *pixels* da imagem, isto é feito executando o cálculo de endereçamento dos *pixels* a serem lidos através da fórmula: $end.Pixel = (img + l * sizeL + c)$, onde l e c iniciam com valores lInicial (linha inicial do corte para ampliação) e cInicial (coluna inicial do corte), respectivamente, e finalizam com linhaM (linha final do corte) e colunaM (coluna final do corte). Mais detalhes na Seção 4.2.3

Ao efetuar uma leitura de um *pixel* da memória RAM, é avaliado qual contraste será aplicado. O contraste cinza é apenas a média aritimética das componentes vermelho, verde e azul do *pixel* e dispensa maiores detalhes. Os contrastes vermelho e verde são cálculados com base na fórmula $k = 0.299 * r + 0.587 * g + 0.114 * b$ usado no xLupa Embarcado (Seção 3.2.1). O cálculo original utiliza ponto-flutuante, porém o cMIPS suporta apenas valores interios, por esse motivo a fórmula empregada considera apenas valores inteiros e a aproximação utilizada é $k' = (2 * r + 4 * g + 1 * b)/6$, discutida na Seção 4.2.3.

Com o contraste aplicado no *pixel*, o cLUPA-0 o armazena no BUFFER e então lê o próximo. O ciclo de leitura, processamento e armazenamento encerra-se no último *pixel* da imagem, representado pelo laço n<totalPixel (sem ampliação) ou l<linhaM na Figura 4.8.

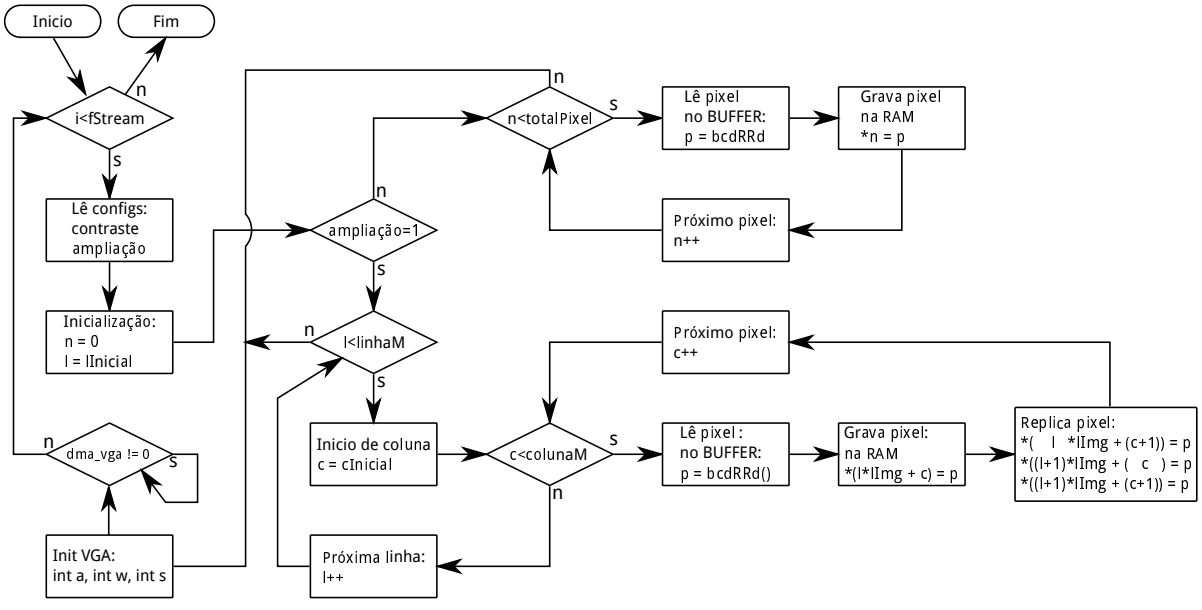


Figura 4.9: Fluxograma cLUPA-1 (Módulo do núcleo cMIPS-1).

O módulo cLUPA-1, que executa no núcleo cMIPS-1 é apresentado na Figura 4.9. Os componentes de controle fStream e totalPixel/linhaM são idênticos nos módulos. O cLUPA-1 difere do cLUPA-0 nos seguintes aspectos: Após a definição da configuração de ampliação, o *pixel* é buscado no BUFFER, seu endereço na RAM é calculado de forma idêntica ao utilizado no cLUPA-0 com configuração de ampliação desativada e em seguida este é gravado na memória RAM. Com a conclusão da leitura e armazenamento de todos os *pixels* da imagem, o cLUPA-1 ativa o módulo VGA para iniciar o processo de transferência e aguarda o seu término, para então iniciar o processo com uma nova imagem. Caso a configuração de ampliação esteja ativada, os cálculos replicam o *pixel* para as vizinhanças baseado na mesma fórmula $end.Pixel = (img + l * sizeL + c)$, esses cálculos de replicação são apresentados na Seção 4.2.3.

A Figura 4.10 é o diagrama de sequência que contém a troca de mensagens entre os módulos do cLUPA. O diagrama mostra o processamento de duas imagens delimitadas pela área cinza claro. O módulo BUFFER é o divisor da execução: do lado esquerdo (W) encontram-se as mensagens do módulo cLUPA-0 e do lado direito (R), as mensagens do módulo cLUPA-1. Foram desconsiderados os elementos de controle do *software* do cLUPA presentes na simulação (quantidade de imagens a executar, controle dos laços para processamento dos *pixels* e elementos de iniciação e finalização) visando deixar o diagrama de sequência mais "limpo", apresentando somente a troca de mensagens importantes para o processamento das imagens. Os elementos p0, pi e pn indicam *pixels* da imagem: p0 indica o primeiro *pixel* da imagem a ser processado, pn representa o último *pixel* e pi representa todos os demais *pixels* da imagem e contém um contorno em cinza escuro que agrupa as mensagens deste conjunto.

As mensagens trocadas entre os módulos do cLUPA são:

- cgf: configuração dos módulos DMA (DMA-USB e DMA-VGA). Indicam qual é o endereço inicial que os módulos devem utilizar para realizar a transferência dos *pixels* e a quantidade de *pixels* transferidas.

- rdy?: mensagem que avalia se o módulo DMA finalizou a transferência. Enquanto não receber a confirmação de término, o núcleo que emitiu esta mensagem não

executa processamento útil.

- rdy!: confirmação da mensagem rdy? emitido pelo módulo DMA quando a transferência dos *pixels* finalizar.

- ti: indica o início da transferência de um *pixel* entre a memória RAM, módulo DMA e seu respectivo "par" (MÓDULO-USB ou MÓDULO-VGA). O MÓDULO-USB emprega quatro mensagens de 8 bits cada para completar a transferência de um *pixel* para memória RAM-0, enquanto o MÓDULO-VGA recebe um *pixel* completo da memória RAM-1 por mensagem.

- tf: é a mensagem que contém o *pixel* (3 bytes) que está sendo escrito na memória RAM-0 por DMA-USB ou lido da memória RAM-1 por DMA-VGA.

- r0 e rn: leitura dos *pixels* efetuada pelo núcleo cMIPS-0 para processá-los e enviá-los ao BUFFER.

- w0 e wn: escrita dos *pixels* no BUFFER por cMIPS-0. É necessário avaliar se existe espaço disponível no BUFFER que responde de forma afirmativa e posteriormente recebe o *pixel*, totalizando três mensagens. O núcleo cMIPS-0 aguarda até o BUFFER responder de forma afirmativa.

- d0 e dn: leitura dos *pixels* efetuada pelo núcleo cMIPS-1. De forma similar às mensagens w0 e wn, é necessário avaliar se existe um *pixel* disponível no BUFFER que responde de forma afirmativa e posteriormente o envia. O núcleo cMIPS-1 aguarda até o BUFFER responder de forma afirmativa.

- e0 e en: escrita dos *pixels* na memória RAM-1 após o núcleo cMIPS-1 processá-los.

Na Figura 4.10 (Diagrama de sequência) é possível observar a sobreposição de tempo durante o processamento dos *pixels* entre os núcleos cMIPS-0 e cMIPS-0. Em um instante de tempo muito próximo em que o $pixel_n$ que está sendo processado por cMIPS-1, o $pixel_{n+1}$ está sendo processado por cMIPS-0. Outra sobreposição acontece quando o

cMIPS-0 finaliza o processamento da Imagem-0 envia as mensagens para início da transferência da Imagem-1. Durante este mesmo período de tempo, o núcleo cMIPS-1 processa os últimos *pixels* e envia as mensagens para início da transferência de RAM-1 para MÓDULO-VGA realizado por DMA-VGA.

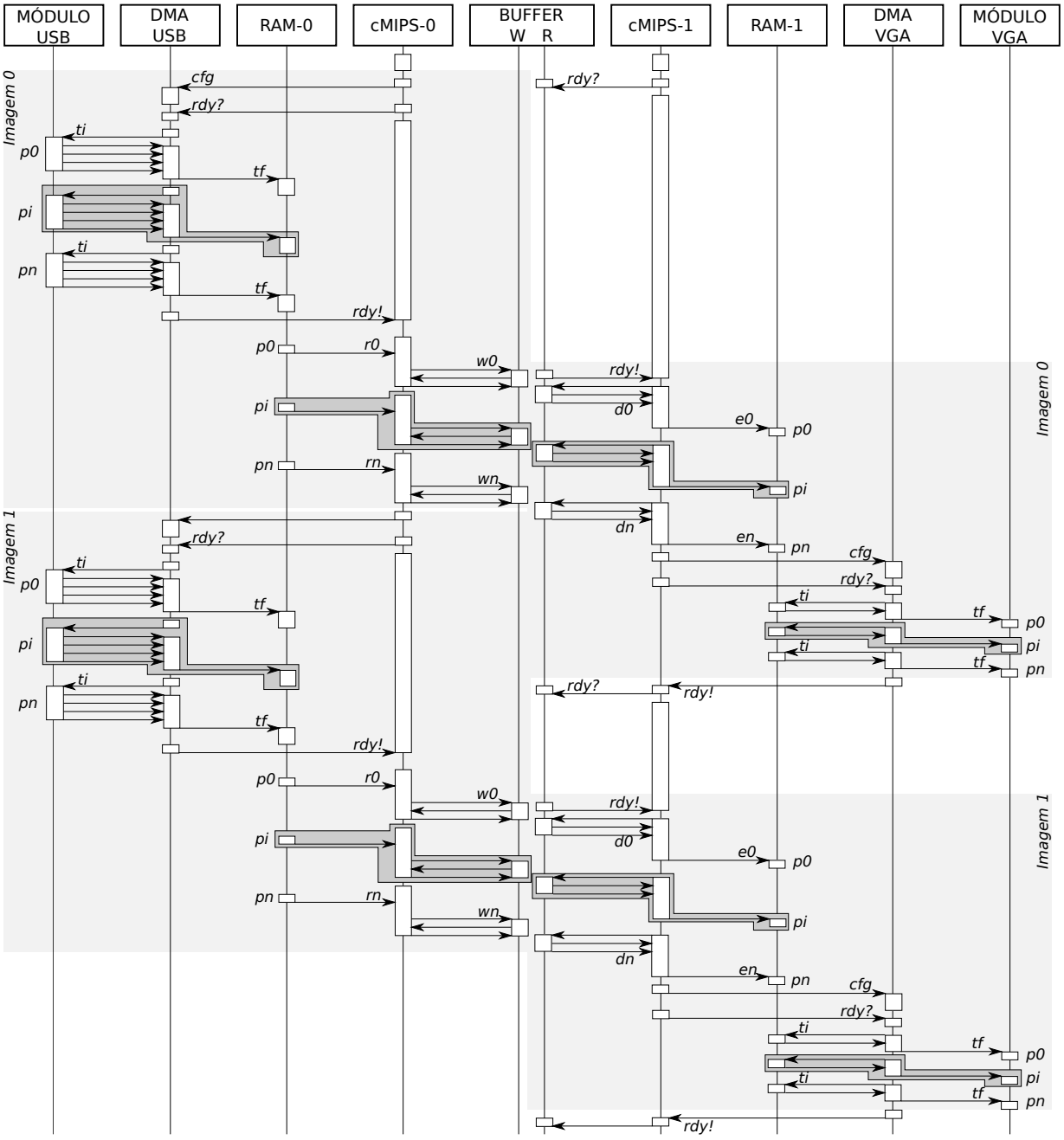


Figura 4.10: Diagrama de sequência do cLUPA.

Ao observar o comportamento dos módulos cLUPA-0 e cLUPA-1 percebe-se um desbalanceamento de processamento entre os núcleos cMIPS-0 e cMIPS-1. O núcleo cMIPS-0 aplica o contraste e, no caso da configuração de ampliação ativa, deve calcular os en-

dereços dos *pixels* buscados, enquanto o núcleo cMIPS-1 apenas calcula os endereços dos *pixels* em caso de ampliação ativa. Assim, o código do cLUPA foi reescrito para aplicar o contraste verde e vermelho nos dois núcleos com a configuração de ampliação desativada. Esta nova versão foi denominada cLUPAb e seus módulos cLUPAb-0 e cLUPAb-1.

O novo módulo cLUPAb-0 inicia a aplicação dos contrastes verde ou vermelho com a ampliação ativa e grava no BUFFER o resultado parcial de cada pixel. O contraste cinza com ampliação ativa e configurações de contraste para ampliação desativada não sofreram alterações. Para balancear o processamento, o módulo cLUPAb-1 completa a aplicação dos contrastes verde e vermelho no caso da configuração de ampliação ativa. As alterações encerraram-se neste ponto, pois o contraste cinza com apliação ativa e demais configurações de contraste para configuração de ampliação desativada não sofreram modificações. O Capítulo 5 apresenta o ganho de desempenho obtido com a otimização cLUPAb.

Os códigos escritos na linguagem de programação C dos componentes de *software* `cLupa-0`, `cLupa-1`, `cLupab-0`, `cLupab-1` encontram-se no Apêndice B.

4.2.3 Cálculos de endereçamento e de contraste

Uma imagem é representada por um espaço bidimensional com componentes que determinam linha (l) e coluna (c) de um *pixel*. Já a memória RAM é representada por uma única dimensão (*end.*) de *bytes*. Assim, o cálculo de endereços para acesso aos *pixels* pelos módulos do cLUPA é realizado da seguinte forma:

$$end.Pixel = (img + l * sizeL + c)$$

Onde:

- img é o endereço inicial da imagem na RAM,
- l é a linha do *pixel* alvo,
- c é a coluna do *pixel* alvo,
- $sizeL$ é a quantidade de *pixels* da linha (largura da imagem).

Por exemplo, se considerarmos $img = 1.000$, com uma imagem com largura de 640 *pixels* e altura de 480 *pixels*, então o *pixel* que se encontra na linha $l = 240$ e coluna $c = 320$ da imagem (*pixel* 78.120) se encontra no endereço 312.480 na memória RAM. O número 312.480 resulta do acesso alinhado em quatro bytes efetuado por cMIPS, assim, o primeiro *pixel* da imagem (*pixel 0*), no exemplo citado, está na posição 1.000, o segundo (*pixel* 1) está na posição 1.004 e o último *pixel* da imagem (o *pixel* 307.200) está na posição 1.228.800 da memória RAM.

O cLUPA processa a ampliação nas imagens e os cálculos são divididos entre os módulos cLUPA-0 e cLUPA-1. A ampliação utilizada no xLupa Embarcado depende das bibliotecas GTK+ e Cairo. Desta forma, a função de ampliação do cLUPA foi reescrita utilizando o método dos vizinhos próximos [12], um algoritmo que requer que cada *pixel* da entrada seja acessado uma única vez.

A ampliação executada por cLUPA é de área central, como mostra a Figura 4.11. A imagem a ser ampliada contém x *pixels* de largura (x colunas) e y *pixels* de altura (y linhas) e seus limites são definidos por a, b, c e d. A área a ser ampliada tem altura $y' = y/2$ e largura $x' = x/2$ e está localizada no centro da imagem limitada pelos *pixels* a', b', c' e d' que são calculados da seguinte forma:

Pixel $a_{(x,y)} = (x/4, y/4)$,

Pixel $b_{(x,y)} = (x/4, y/4 + y')$,

Pixel $c_{(x,y)} = (x/4 + x', y/4)$,

Pixel $d_{(x,y)} = (x/4 + x', y/4 + y')$.

Após os cálculos dos pontos serem realizados, cada *pixel* é recuperado da área central da imagem, processado de acordo com a configuração de contraste e replicado para quatro novas posições na imagem resultante.

Por exemplo, considere $y = 480$ e $x = 640$. Assim $y' = 240$ e $x' = 320$ e os pontos a', b', c' e d' são:

Ponto $a_{(x,y)} = (640/4, 480/4) = (160, 120)$,

Ponto $b_{(x,y)} = (640/4, 480/4 + 240) = (160, 360)$,

Ponto $c_{(x,y)} = (640/4 + 320, 480/4) = (480, 120)$,

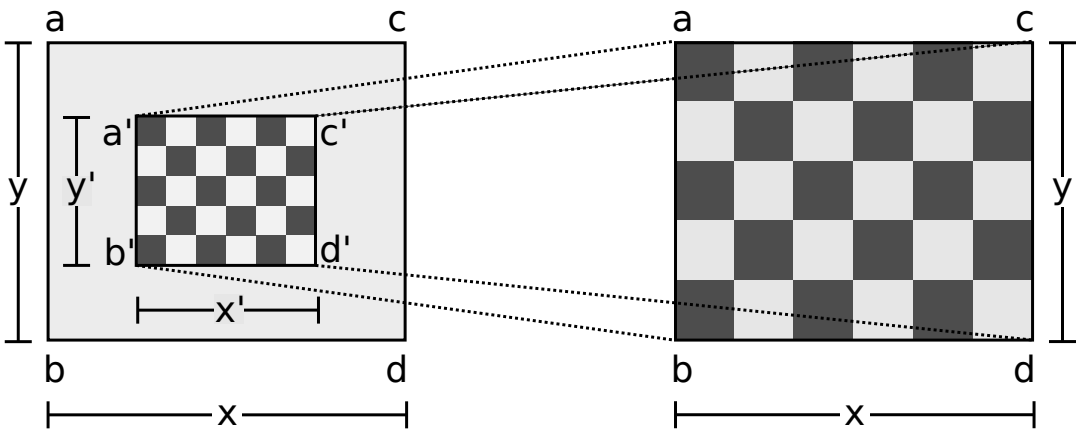


Figura 4.11: Ampliação de área central.

Ponto $d_{(x,y)} = (640/4 + 320, 480/4 + 240) = (480, 360)$.

Para exemplificar o método de replicação, mostrado na Figura 4.12, considere $y = 4$ e $x = 4$, como na Figura 4.12(a). Assim têm-se $y' = 2$, $x' = 2$ e a área central é determinada por $a(1,1)$, $b(1,2)$, $c(2,1)$ e $d(2,2)$, representados pelos *pixels* 5, 9, 6 e A, respectivamente.

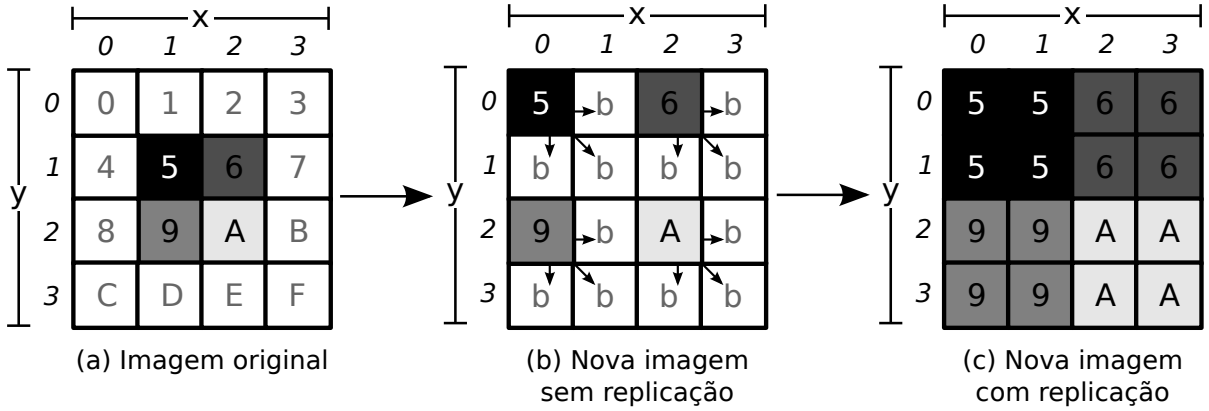


Figura 4.12: Ampliação utilizando o método dos vizinhos próximos.

A posição de cada *pixel* é determinado utilizando-se o cálculo $end.Pixel = (x' * 2, y' * 2)$, onde x' e y' são as coordenadas do *pixels* relativas à área central da imagem original. Por exemplo, o *pixel* 6 está na posição $(1, 0)'$ na área central da imagem original e sua nova posição é:

$end.Pixel(1, 0)' = (1 * 2, 0 * 2) = (2, 0)$.

A multiplicação por 2 ocorre da replicação dos *pixels* para seus vizinhos (*pixels* b na Figura 4.12(b)). Continuando com o exemplo, o *pixel* 6 é replicado para três vizinhos, dados por:

$b0.Pixel(1,0)' = (1*2+1, 0*2) = (3,0)$,

$b1.Pixel(1,0)' = (1*2, 0*2+1) = (2,1)$,

$b2.Pixel(1,0)' = (1*2+1, 0*2+1) = (3,1)$.

Desta forma, gera-se a imagem resultante da ampliação central com replicação dos *pixels* seguindo o método dos vizinhos próximos, como mostrado na Figura 4.12(c).

O contraste é baseado no cálculo do brilho da imagem e nos componentes de cada *pixel*. A fórmula original utilizada no xLupa Embarcado é $k = 0.299 * r + 0.587 * g + 0.114 * b$. Devido o cMIPs não suportar cálculos em ponto-flututante, é necessário adaptar o cálculo para utilizar apenas números inteiros da seguinte forma: $k' = (2 * r + 4 * g + 1 * b)/6$. Desta forma, os valores aproximados de cada constante das componentes r, g e b é $k'' = (0,333 * r + 0,666 * g + 0,166 * b)$.

Como k' é uma aproximação de k, k' retorna valores diferentes do cálculo original, gerando uma imagem final que difere da imagem final de k. Para mensurar esta diferença entre as imagens geradas, foram calculados as diferenças médias, diferença máxima e a quantidade de *pixels* diferentes da componente g, para contraste verde, ou r para contraste vermelho nas duas configurações de ampliação.

A diferença média é calculada por:

$$Media = \frac{\sum |p(c)_k - p(c)_{k'}|}{p}$$

Onde $p(c)_k$ é a componente r ou g do *pixel* p resultante do cálculo k e $p(c)_{k'}$ é a componente r ou g do *pixel* p resultante do cálculo k'.

A diferença máxima é dada por:

$$Max(c) = Maximo(|p(c)_k - p(c)_{k'})|$$

Onde $Max(c)$ é o valor máximo da diferença entre as componentes r ou g do *pixel* resultante dos cálculos k e k'

A tabela 4.1 apresenta os resultados das diferenças obtidas entre os cálculos k e k'. A diferença máxima encontrada para todos os casos foi 1, os valores das componentes r ou g para os cálculos k e k' diferem em 1 (1/255), quando são diferentes. A quantidade de

Tabela 4.1: Diferenças entre os cálculos k e $k´$.

	Ampliação Ativada			Ampliação Desativa		
Contraste	Média	Máx.	%.	Média	Máx.	%.
Verde	0,039	1	3,9%	0,029	1	2,87%
Vermelho	0,051	1	5,14%	0,037	1	3,80%

pixels diferentes variam de acordo com a configuração de contraste e é um número baixo se comparado com a totalidade dos *pixels*. Desta forma, a diferença entre os *pixels* das imagens geradas por k e k' é no máximo de 5,14%.

CAPÍTULO 5

SIMULAÇÕES COM O MODELO EM VHDL DO CLUPA

A avaliação de desempenho do cLUPA foi realizada através de simulações da implementação do modelo escrito em VHDL. Na Seção 5.1 são descritas as simulações realizadas, as métricas utilizadas na Seção 5.2, e os resultados são mostrados na Seção 5.3.

5.1 Simulações realizadas

Para avaliar o desempenho do cLUPA foram realizadas simulações de processamento de imagens em diversas configurações de contraste e ampliação. Como entrada das simulações foram utilizadas imagens no formato *bitmap* de 32 bits com uma resolução de 640 *pixels* de largura por 480 *pixels* de altura, totalizando 307.200 *pixels*. A escolha da resolução é justificada pelo uso do módulo CONTROLADOR-VGA, apresentado na Seção 2.5, que controla o módulo VGA configurado para a exibição de imagens nesta resolução.

As simulações do cLUPA utilizaram um ciclo de relógio de 20 ns ($50Mhz$). O período do relógio foi determinado para respeitar as especificações do conjunto de desenvolvimento Mercurio IV.

Com relação às configurações de contraste e ampliação, foram simuladas as quatro configurações de contraste com as duas configurações de ampliação, totalizando oito configurações, nomeadas da seguinte forma:

- SZ_CI: configuração de ampliação desativada (SZ) e sem aplicação de contraste (CI),
- SZ_CC: ampliação desativada (SZ) e com aplicação de contraste cinza (CC).
- SZ_CG: ampliação desativada (SZ) e com aplicação de contraste verde (CG).
- SZ_CR: ampliação desativada (SZ) e com aplicação de contraste vermelho (CR).
- CZ_CI: ampliação ativada (CZ) e sem aplicação de contraste (CI).

- CZ_CC: ampliação ativada (CZ) e com aplicação de contraste cinza (CC).
- CZ_CG: ampliação ativada (CZ) e com aplicação de contraste verde (CG).
- CZ_CR: ampliação ativada (CZ) e com aplicação de contraste vermelho (CR).

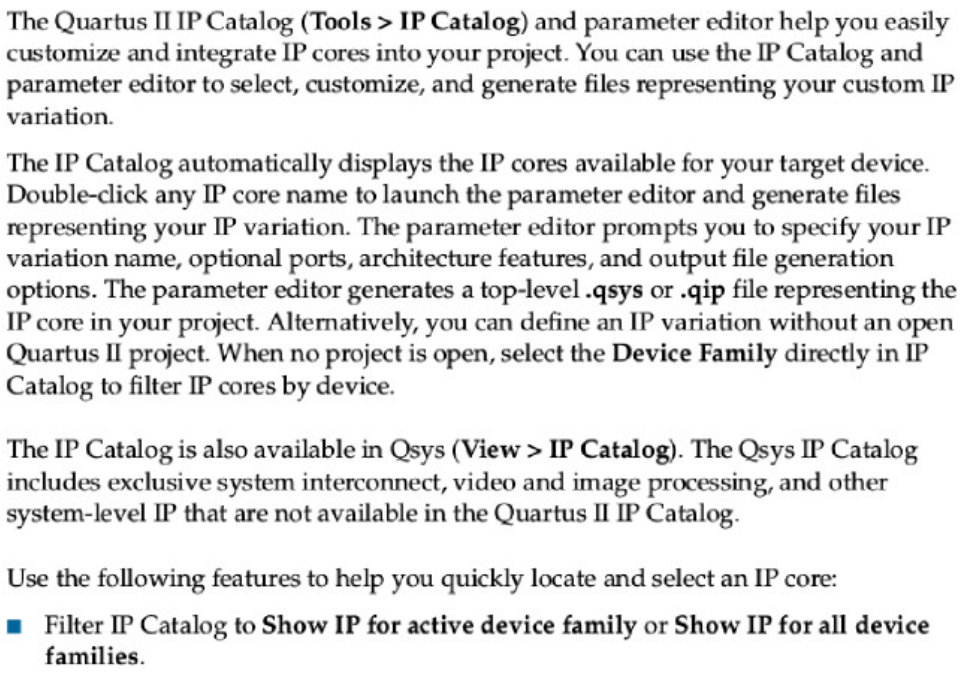
Os códigos dos módulos de *software* do cLUPA foram compilados utilizando dois níveis de otimização: O2 e O3. O nível de otimização O2 é usado por padrão pelos *scripts* de compilação dos arquivos de teste do cMIPS (Seção 2.1.1). O nível de otimização O3 foi utilizado para avaliar a possibilidade de ganho de desempenho com o uso das *flags* -finline-functions, que aloca o código das funções e procedimentos diretamente no código principal e -funswitch-loops, que remove saltos condicionais que não contém dependência de dados dos laços de repetição.

A imagem original usada para as simulações é apresentada na Figura 5.1, que é parte da página 1061 do *Stratix IV Handbook Device* [2]. Esta imagem tem como característica o fundo branco com texto escuro. Esta característica foi adotada pois não foi implementado na plataforma cLUPA uma funcionalidade que detecte a cor de fundo para depois processar o contraste. O arquivo de entrada da simulação img.data, descrito na Seção 4.1.3, é composto por três vetores de *pixels* que correspondem a três imagens no formato *bitmap* de 32 bits sem o cabeçalho[1] do arquivo original (primeiros 136 bytes).

Uma simulação processa três imagens do arquivo img.data, conforme as configurações de contraste e ampliação, e as imagens tratadas são gravadas no arquivo vga.data. O processamento de cada imagem é dividido em quatro operações:

- Transferência USB (T-USB): transferência da imagem do módulo MÓDULO-USB para RAM-0. Realizada pelo módulo DMA-USB.
- Processamento inicial (C-0): leitura da imagem escrita em RAM-0, execução da configuração de ampliação, execução da configuração de contraste e escrita no BUFFER. Realizado pelo núcleo cMIPS-0.

[1]Os primeiros 136 bytes do arquivo no formato *bitmap* de 32bits divide-se em dois grupos de informações: o primeiro, de 12 bytes, contém assinatura, tamanho do arquivo, *offset* para vetor de *pixels* e dois campos reservados, e o segundo, de 124 bytes, contém a 5ª versão do cabeçalho bitmap [43].



The Quartus II IP Catalog (**Tools > IP Catalog**) and parameter editor help you easily customize and integrate IP cores into your project. You can use the IP Catalog and parameter editor to select, customize, and generate files representing your custom IP variation.

The IP Catalog automatically displays the IP cores available for your target device. Double-click any IP core name to launch the parameter editor and generate files representing your IP variation. The parameter editor prompts you to specify your IP variation name, optional ports, architecture features, and output file generation options. The parameter editor generates a top-level **.qsys** or **.qip** file representing the IP core in your project. Alternatively, you can define an IP variation without an open Quartus II project. When no project is open, select the **Device Family** directly in IP Catalog to filter IP cores by device.

The IP Catalog is also available in Qsys (**View > IP Catalog**). The Qsys IP Catalog includes exclusive system interconnect, video and image processing, and other system-level IP that are not available in the Quartus II IP Catalog.

Use the following features to help you quickly locate and select an IP core:

- Filter IP Catalog to **Show IP for active device family** or **Show IP for all device families**.

Figura 5.1: Imagem usada como base para as simulações do cLUPA.

- Processamento final (C-1): leitura do BUFFER, execução da configuração de ampliação, execução da configuração de contraste e escrita da imagem em RAM-1. Realizado pelo núcleo cMIPS-1.

- Transferência VGA (T-VGA): transferência da imagem da RAM-1 para o módulo MÓDULO-VGA. Realizada pelo módulo DMA-VGA.

Para mensurar o tempo simulado de cada operação é utilizado o recurso assert do VHDL. assert é um comando condicional adicionado nos componentes modelados em VHDL que emite um relatório que permite verificar a corretude do modelo que está sendo simulado. O relatório contém uma mensagem personalizada e o momento em que esta mensagem é exibida no tempo simulado. O Quadro 5.1 mostra os assert utilizados, e suas representações na simulação do cLUPA.

Os relatórios nas linhas 8, 10 e 18 do Quadro 5.1 são gerados durante um evento específico e não requerem uma condição elaborada, assim usou-se o termo false para que, quando o evento ocorresse, o relatório fosse emitido. Os eventos específicos são: gravação de um *pixel* no módulo BUFFER (linha 8), leitura de um *pixel* do módulo BUFFER

(linha 10) e finalização da gravação do arquivo de saída vga.data (linha 18). Os relatórios restantes nas linhas 2, 5, 12 e 15 são gerados somente durante a mudança de estado dos módulos DMA e utilizam o sinal de controle initBycMips como condicional. O sinal initBycMips é ativado com valor lógico "1" quando o núcleo cMIPS envia as configurações para o módulo DMA, dando início à transferência de *pixels*, esta ativação é a mensagem cfg na Figura 4.10. Ao terminar, o sinal initBycMips é desativado com valor lógico "0" e indica ao núcleo cMIPS a finalização da transferência, representado pela mensagem rdy! da Figura 4.10.

```
1  -- Reporta início da transferência USB
2  assert (initBycMips'event and initBycMips = '0')
3    report "DMA_USB_transfer: init";
4  -- Reporta término da transferência USB
5  assert (initBycMips'event and initBycMips = '1')
6    report "DMA_USB_transfer: stop";
7  -- Reporta a gravação de um pixel no BUFFER
8  assert false report "BCD_write["&SLV32HEX(data)&"]";
9  -- Reporta a leitura de um pixel do BUFFER
10 assert false report "BCD_read ["&SLV32HEX(data)&"]";
11 -- Reporta início da transferência VGA
12 assert (initBycMips'event and initBycMips = '0')
13   report "DMA_VGA_transfer: init";
14 -- Reporta término da transferência VGA
15 assert (initBycMips'event and initBycMips = '1')
16   report "DMA_VGA_transfer: stop";
17 -- Reporta a finalização da gravação do arquivo de saída vga.data
18 assert false report "DMA_VGA_transfer: File Closed";
```

Quadro 5.1: Comandos assert utilizados nos modelos VHDL do cLUPA

Com os relatórios gerados pelos assert do Quadro 5.1 é possível determinar o tempo utilizado para a transferência de uma imagem realizada pelo módulo DMA-USB (relatórios das linhas 2 e 5), o tempo utilizado para a transferência de uma imagem realizada pelo módulo DMA-VGA (relatórios das linhas 12 e 15), o tempo de processamento do núcleo

cMIPS-0 para cada *pixel* (relatório da linha 8), o tempo de processamento do núcleo cMIPS-1 para cada *pixel* (relatório da linha 10) e o tempo total de uma simulação com o relatório da linha 18.

5.2 Métricas

Os tempos registrados nos relatórios criados a partir do comando assert foram analisados para avaliar o desempenho do cLUPA. A avaliação se dá com três métricas:

- Quadros por segundo (QPS): é a quantidade de imagens que a plataforma processa em 1 segundo,
- Tempo de processamento (TP): é o tempo médio necessário para executar uma determinada operação. Também é medido o tempo de processamento em ciclos de relógio (CP).
- Processamento de um *pixel* (IP): tempo médio de processamento de um *pixel*, dado em ciclos de relógio.

As unidades adotadas para representação dos valores são:

- milesegundos (ms) para representar TP;
- ciclos por operação (cpo) para representar CP;
- ciclos médios por *pixel* (cpp) para representar IP.

5.3 Resultados

Os resultados obtidos das simulações são apresentadas no que se segue. Inicialmente são mostrados os tamanhos dos arquivos compilados com os diferentes níveis de otimização. Após, as simulações de todas as configurações de contraste e ampliação do cLUPA. Finalmente, as simulações foram refeitas com a versão cLUPAb, com balanceamento de carga.

A Tabela 5.1 apresenta os tamanhos, em bytes, dos arquivos gerados pela compilação nos níveis de otimização O2 e O3 para o *software* do cLUPA e nível de otimização O3 para

o cMIPSb. É possível observar que o executável cLUPA do núcleo cMIPS-0 (prog0.bin) aumentou seu tamanho devido a expansão das funções diretamente no código. O código que executa no núcleo cMIPS-1 (prog1.bin) sofreu um aumento de apenas quatro bytes. O código com processamento balanceado cLUPAb sofreu um grande aumento em tamanho se comparado ao código original cLUPA. Isto foi causado pela alteração de alguns laços de controle que foram passados do código principal para as funções de contraste em conjunto com o parâmetro de compilação -finline-functions.

Tabela 5.1: Tamanhos, em bytes, dos arquivos executáveis cLUPA e cLUPAb

	cLUPA O2	cLUPA O3	cLUPAb O3
prog0.bin	2.424	2.832	5.892
prog1.bin	1.912	1.916	5.876

Os tempos simulados médios para o cLUPA com configuração de contraste verde e ampliação desativada utilizando códigos compilados em nível de otimização O2 (SZ_CG_O2) são apresentados na Tabela 5.2. A transferência do MÓDULO-USB para a RAM-0 custa 61,44 ms, o equivalente a 3.072.006 *cpo* que obtém uma média de 10 *cpp*. Este valor foi alcançado em todas as simulações, pois o módulo DMA-USB depende apenas do número total de *pixels* da imagem e da frequência de operação da simulação, que permaneceram constantes durante todas as simulações.

Tabela 5.2: Tempo das operações do cLUPA para SZ_CG com otimização O2.

	T_USB	C-0	C-1	T_VGA
TP (ms)	61,44	361,57	361,57	12,29
CP (*cpo*)	3.072.006	18.078.484	18.078.497	614.401
IP (*cpp*)	10	59	59	2

O módulo DMA-VGA gasta apenas 2 *cpp*, valor medido também em todas as simulações, pois compartilha da mesma propriedade do módulo DMA-USB: depender apenas do número total de *pixels*. As operações C-0 e C-1, que representam o processamento nos núcleos cMIPS, demoram aproximadamente 361,57 ms, ou o equivalente à 59 *cpp*.

O tempo simulado total SZ_CG_O2 de uma imagem foi de 435,30 ms ou 21.765.001 *cpp*, alcançado pela sobreposição de processamento existente entre as operações C-0 e C-1.

A Figura 5.2 apresenta o diagrama de tempo das operações do cLUPA com valores

dados em ciclos de relógio (*cpo*). O cLUPA-0 requer 77 ciclos de relógio, ou 1,54 μs, para iniciar a tarefa de transferência USB (T-USB). Ao finalizar a operação T-USB, é efetuada a leitura das configurações de contraste e ampliação e, em seguida, a operação de processamento C-0 é iniciada e o primeiro *pixel* é gravado em BUFFER. Levando em consideração que o custo médio de processamento de cada *pixel* em C-0 é de 59 ciclos, como mostrado na Tabela 5.2, e o tempo entre a finalização da operação T-USB e a gravação do primeiro *pixel* no BUFFER é de 73 ciclos, o custo da leitura das configurações é de, no máximo, 12 ciclos.

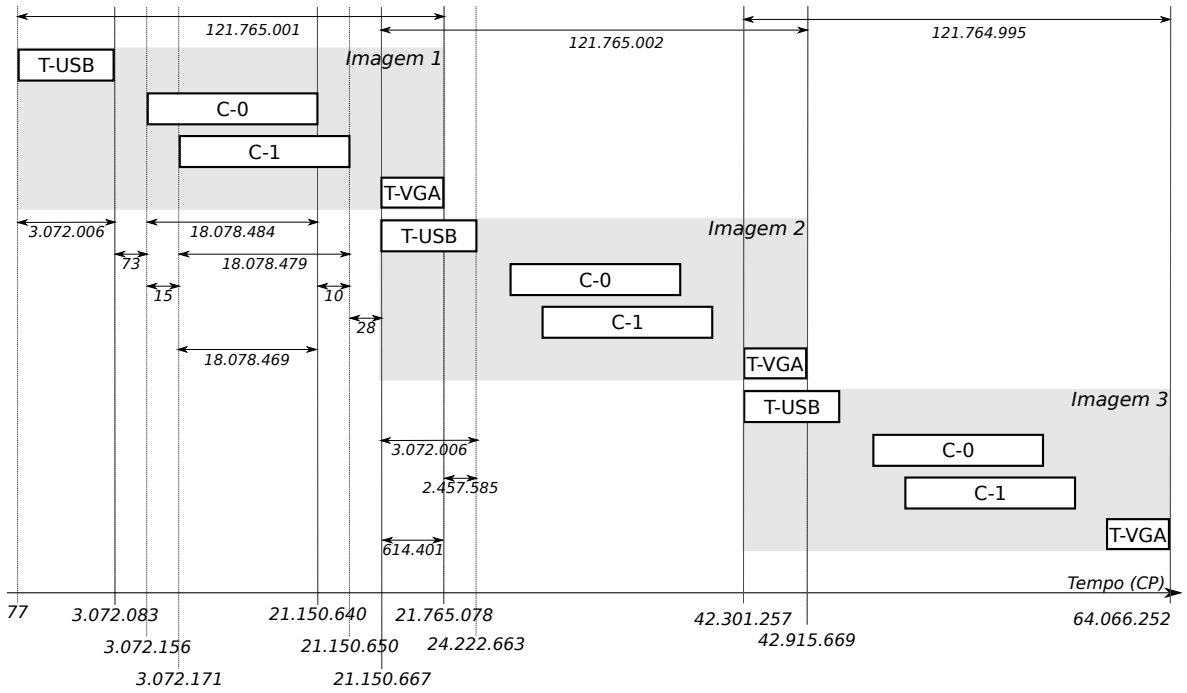


Figura 5.2: Diagrama de tempo das operações do cLUPA para SZ_CG em nível O2.

Após 15 ciclos da gravação do primeiro *pixel* no BUFFER pela operação C-0, que acontece no tempo 3.072.156, a operação C-1 é iniciada. Desta forma, as duas operações processam os *pixels* da imagem concorrentemente. A operação C-1 encerra-se no tempo 21.150.650 ciclos, 10 ciclos depois do encerramento da operação C-0. O custo para processar uma imagem pelas operações C-0 e C-1 foi 361,57 ms ou, respectivamente, de 18.078.484 ciclos e 18.078.497 ciclos.

É requerido um tempo de preparação e controle de 28 ciclos antes de iniciar a transferência VGA (T-VGA). A operação T-VGA é a operação mais rápida e consome apenas

614.401 ciclos de relógio.

No mesmo instante em que a operação T-VGA é iniciada, a operação T-USB para a nova imagem também é iniciada, o que gera uma sobreposição de processamento durante o tempo da operação T-VGA.

As sobreposição de processamento de uma imagem que ocorre entre as operações C-0 e C-1 e a sobreposição de transferências entre as operações T-VGA e T-USB acontecem em todas as simulações. Desta forma, serão mostrados apenas os tempos das operações para as demais simulações.

A Tabela 5.3 apresenta os tempos médios obtidos da simulação com configuração de contraste verde e ampliação ativada com otimização nível -O2 (CZ_CG_O2).

Tabela 5.3: Tempo das operações do cLUPA para CZ_CG em nível O2.

	T_USB	C-0	C-1	T_VGA
TP (*ms*)	61,44	85,75	85,75	12,29
CP (*cpo*)	3.072.006	4.287.837	4.287.837	614.401
IP (*cpp*)	10	56	56	2

Como já mencionado, os tempos para as transferências T-USB e T-VGA foram idênticos aos tempos obtidos da simulação SZ_CG_O2. O tempo de processamento para as operações C-0 e C-1 foi de 85,75 *ms*, equivalente a 24% da simulação com ampliação desativada SZ_CG_O2. Esta diferença se deve ao número de *pixels* processados pela operação C-0. As simulações com configuração de ampliação desativada (SZ) processam todos os 307.200 *pixels* da imagem, enquanto as simulações com configuração de apliação ativada (CZ) processam 76.800 *pixels*, 1/4 da total de *pixels* da imagem.

Outro fator que contribui para a diferença é a forma como o núcleo cMIPS-0 verifica a disponibilidade de espaço para escrita no BUFFER, apresentado no Quadro 4.2. A verificação de disponibilidade é dada por um laço de repetição que segura o processamento até encontrar, no mínimo, um espaço disponível para escrita do *pixel* processado. Como o conjunto de *pixels* a ser processado é 25% no caso da configuração de ampliação ativa, reduz nos mesmos 25% o acesso ao BUFFER, o que impacta no desempenho do cLUPA para configurações de ampliação ativada (CZ).

A Tabela 5.4 apresenta os tempos das simulações de contraste vermelho nas duas

configurações de ampliação com otimização nível O2 (SZ_CR_O2 e CZ_CR_O2).

Tabela 5.4: Tempo das operações C-0 e C-1 para CZ_CR e SZ_CR em nível O2.

	CZ_CR		SZ_CR	
	C-0	C-1	C-0	C-1
TP (*ms*)	84,52	85,52	356,35	356,35
CP (*cpo*)	4.226.096	4.226.097	17.817.542	17.817.543
IP (*cpp*)	55	55	58	58

De forma semelhante ao que acontece com os tempos das simulações de contraste verde (CG), as simulações com contraste vermelho (CR) resultam numa diferença entre as configurações de ampliação, que já era esperada. Ao se comparar as configurações de contraste CG e CR, nota-se uma redução de 1 *cpp* para o contraste vermelho (CR) tanto na configuração de ampliação ativa (CZ) quanto na configuração de ampliação desativada (SZ).

Esta redução é causada pela disposição das chamadas das funções de contraste, apresentada na Figura 4.8. A escolha do contraste encontra-se sobre a forma do comando switch-case da linguagem C interno a um laço de repetição. A instrução de alto nível switch-case é traduzida para a linguagem de montagem como uma sequência de instruções de salto beq que executa um salto condicional caso o conteúdo de dois registradores especificados sejam iguais. O salto condicional que determina a chamada para a função de aplicação de contraste vermelho aparece na posição anterior ao salto condicional da chamada da função de aplicação de contraste verde, causando a diferença de 1 *cpp* entre as simulações CG e CR.

A configuração de contraste cinza (CC) é a configuração que utiliza a média aritmética das componentes r, g e b para determinar a cor final do *pixel* processado. Sendo mais simples que os cálculos realizados para as configurações de contraste verde (CG) e vermelho (CR). A Tabela 5.5 mostra os tempos das operações C-0 e C-1 das simulações para configuração de contraste cinza nas configurações de contraste (SZ_CC_O2 e CZ_CC_O2).

As simulações SZ_CC_O2 e CZ_CC_O2 obtém resultados semelhantes às simulações CG_O2 e CC_O2. A simulação SZ_CC_O2 obtém um custo de ciclos por *pixel* (IP) menor do que as configurações de contraste vermelho e cinza devido aos cálculos simplificados e

Tabela 5.5: Tempo das operações C-0 e C-1 para CZ_CC e SZ_CC em nível O2.

	CZ_CC		SZ_CC	
	C-0	C-1	C-0	C-1
TP (ms)	59,94	59,94	258,04	258,04
CP (cpo)	2.997.312	2.997.317	12.902.358	12.902.359
IP (cpp)	39	39	42	42

a configuração de ampliação ativada (CZ_CC_O2) custou um número de ciclos de relógio menor do que a configuração de ampliação desativada (SZ_CC_O2).

A configuração de contraste desativada foi simulada para o código do cLUPA compilado com nível de otimização O2. A Tabela 5.6 apresenta os tempos obtidos das simulações

Tabela 5.6: Tempo das operações C-0 e C-1 para CZ_CI e SZ_CI em nível O2.

	CZ_CI		SZ_CI	
	C-0	C-1	C-0	C-1
TP (ms)	41,50	41,51	172,03	172,03
CP (cpo)	2.075.310	2.075.485	8.601.572	8.601.567
IP (cpp)	27	27	28	28

Como a configuração de contraste inativa (CI) não contém processamento de contraste sobre os *pixels*, apenas executa o acesso às memórias e ao BUFFER, tornando esta a configuração com o menor tempo simulado de execução.

Para avaliar o desempenho global do cLUPA com nível de otimização O2, foram processadas três imagens, iguais à Figura 5.1 e agrupadas no arquivo img.data, para cada configuração de contraste e ampliação. O tempo simulado é mostrado na Tabela 5.7.

Tabela 5.7: Tempo total simulado em nível O2.

ms	CZ	SZ
CI	321,13	712,71
CC	376,45	970,75
CG	453,89	1.281,32
CR	450,18	1.265,66

A configuração de ampliação gera um impacto no desempenho do cLUPA justificado pela quantidade de *pixels* processados nesta configuração, que é quatro vezes maior do que a quantidade de *pixels* processados na configuração de ampliação desativada. As configurações de contraste verde (CG) e vermelho (CR) são as mais custosas em termos de

tempo de processamento e diferem em um valor muito pequeno, devido ao posicionamento das chamadas das funções de contraste no código do *cLUPA*.

A Tabela 5.8 apresenta uma comparação, em ciclos médios de processamento (*cpp*), entre os resultados simulados do cLUPA com nível de otimização O2 para a operação C-0. Confirma-se que o custo, em termos de ciclos médios por *pixel*, é maior para a configuração de contraste verde (CG) e vermelho (CR).

Tabela 5.8: Ciclos por pixel para C-0 em nível O2.

cpp	CZ	SZ
CI	27	28
CC	39	42
CG	55	58
CR	56	59

A Tabela 5.9 apresenta uma comparação entre os resultados, em quadros por segundo, das configurações de contraste e ampliação simuladas do cLUPA com otimização O2.

Tabela 5.9: Quadros por segundo em nível O2.

qps	CZ	SZ
CI	9,34	4,20
CC	7,96	3,09
CG	6,60	2,34
CR	6,66	2,37

Observa-se que as configurações de contraste inativo e cinza alcançam os maiores valores de quadros por segundo, o que é esperado. Diferentemente, as configurações de contraste verde e vermelho foram as configurações que resultaram em desempenhos mais baixos, devido o custo de processamento por *pixel* elevado, com os cálculos de aplicação de contraste.

O códido do cLUPA também foi compilado usando nível de otimização O3. O tempo simulado é apresentado na Tabela 5.10.

A característica observável nos resultados das simulações com nível de otimização O3 é a semelhança dos tempos para as configurações de contraste verde (CG) e vermelho (CR) tanto para ampliação ativada quanto desativada. Na otimização de nível 2, foi detectado como causa da diferença o posicionamento das chamadas das funções na instrução

Tabela 5.10: Tempo total simulado em nível O3.

ms	CZ	SZ
CI	325,74	738,50
CC	376,45	970,75
CG	427,17	1.118,21
CR	427,17	1.118,21

case-switch dentro de um laço de repetição. A otimização de nível 3 ativa o parâmetro -funswitch-loops que remove os saltos condicionais do interior dos laços, eliminando a diferença de tempo de processamento entre os contrastes verde e vermelho.

A Tabela 5.11 apresenta o resultado das simulações com otmização de nível 3 para as configurações de contraste e ampliação.

Tabela 5.11: Quadros por segundo em nível O3.

qps	CZ	SZ
CI	9,20	4,06
CC	7,96	3,09
CG	7,02	2,68
CR	7,02	2,68

De maneira geral, as configurações de contraste verde (CG) e vermelho (CR) tiveram uma melhora de desempenho com a otimização de nível 3, além de equilibrar os tempos de execuções das configurações.

Independente do nível de otimização, todas as configurações de contrase e ampliação apresentaram uma característica relacionada ao tempo de execução das operações C-0 e C-1. A operação C-1, que é a realização dos cálculos finais de endereçamento dos *pixels*, tem tempo de execução muito próximo da operação C-0 que, além de incluir cálculos de endereços dos *pixels*, também realiza o processamento de contraste.

Foram realizadas simulações que não usam nenhum tipo de transferência com a memória RAM-0 e nem processamento pelo núcleo cMIPS-0, apenas a escrita de *pixels* que não representam nenhuma imagem no BUFFER com o objetivo de avaliar o gasto de tempo da operação C-1 nas configurações de contraste verde e vermelho nas configurações de ampliação realizada pelo núcleo cMIPS-1. A operação C-1 não depende da configuração de contraste, mas depende da configuração de ampliação. Com ampliação ativa, que requer cálculos para o endereçamento dos *pixels*, o tempo de execução da operação C-1 foi

de 28 *cpp*, ou 43,04 *ms* (para 76.800 *pixels*) enquanto que a execução da configuração de ampliação desativada foi de 25 *cpp*, ou 153,60 *ms* (para 307.200 *pixels*).

Assim, optou-se por reescrever o código do cLUPA a fim de balancear o processamento de contraste nas configurações verde e vermelho com configuração de ampliação desativada (CZ), pois estas representam as operações que mais demandam processamento.

Os resultados da simulação do cLUPAb apresentaram uma melhora de desempenho com relação as simulações do código cLUPA. A configuração de contraste verde (CG) e vermelho alcançaram 2,85 *qps* com o código balanceado do cLUPAb contra 2,68 *qps* com o código com nível de otimização 3. Esta melhora impactou no tamanho do arquivo executável gerado, que dobrou, como mostra a Tabela 5.1.

A Tabela 5.12 apresenta uma comparação, em quadros por segundo, dos resultados das simulações apresentados nas Tabelas 5.9 e 5.11 e dos resultados obtidos com a simulação do código cLUPAb.

Tabela 5.12: Quadros por segundo das simulações.

	cLUPA-O2		cLUPA-O3		cLUPAb	
qps	CZ	SZ	CZ	SZ	CZ	SZ
CI	9,34	4,20	9,20	4,06	-	-
CC	7,96	3,09	7,96	3,09	-	-
CG	6,60	2,34	7,02	2,68	-	2,85
CR	6,66	2,37	7,02	2,68	-	2,85

É possível observar a melhora de desempenho de acordo com as melhorias no código do cLUPA. As simulações que utilizam configuração de ampliação ativa (CZ) obtiveram os melhores resultados. Apesar de processarem 25% dos *pixels* de uma imagem, não alcançaram 4 vezes o número de quadros por segundo de seus pares com configuração de ampliação desativada, isto é devido aos cálculos relacionados ao endereçamento dos *pixels* em conjunto com as transferências dos *pixels* realizadas pelos módulos DMAs. A alteração do código cLUPAb, que realiza um balanceamento no processamento dos contrastes verde e vemelho obteve resultados um pouco melhores em troca de espaço de armazenamento, pois o tamanho do arquivo executável dobrou se comparado com os arquivos gerados pela compilação com otimização de nível O3.

Cumpre relembrar que as simulações foram executadas utilizando um ciclo de relógio

de 20 ns ($50Mhz$), que coincide com a frequência de operação do conjunto de desenvolvimento Mercurio IV.

Todas as simulações foram executadas no servidor de alto desempenho latrappe do Departamento de Informática da Universidade Federal do Paraná. Este servidor é um sistema *multicore* com 40 processadores Intel Xeon E5-2680 e 64 Gbytes de memória RAM. Cada núcleo tem *clock* de 2,8 GHz e três níveis de memória *cache*: 32 Kbytes para nível 1, 256 Kbytes para nível 2 e 25 Mbytes para nível 3. O simulador, apesar de não aproveitar a característica *multicore*, se beneficia da memória *cache* de nível 3 com grande capacidade. O tempo de execução do simulador variou entre 2 horas para contraste inativo e ampliação ativada (CZ_CI_O3) até 7 horas para contraste vermelho e ampliação desativada (SZ_CR_O3) para o código do cLUPA com otimização de nível O3. Cada simulação ocupava aproximadamente 12 Gbytes de memória RAM durante sua execução.

CAPÍTULO 6

PROTOTIPAÇÃO DO SISTEMA CLUPA COM MERCURIO IV

Todas as alterações listadas nesta Seção são sugestões para a implementação do protótipo do sistema cLUPA no conjunto de desenvolvimento Mercurio IV. Mudanças poderão ocorrer durante o processo de implementação de modo a possibilitar a implementação ou sanar problemas não previstos neste Capítulo.

O modelo do cLUPA foi pensado e projetado para a prototipação utilizando o conjunto de desenvolvimento Mercurio IV. A Figura 6.1 mostra os componentes idealizados para o protótipo do sistema cLUPA.

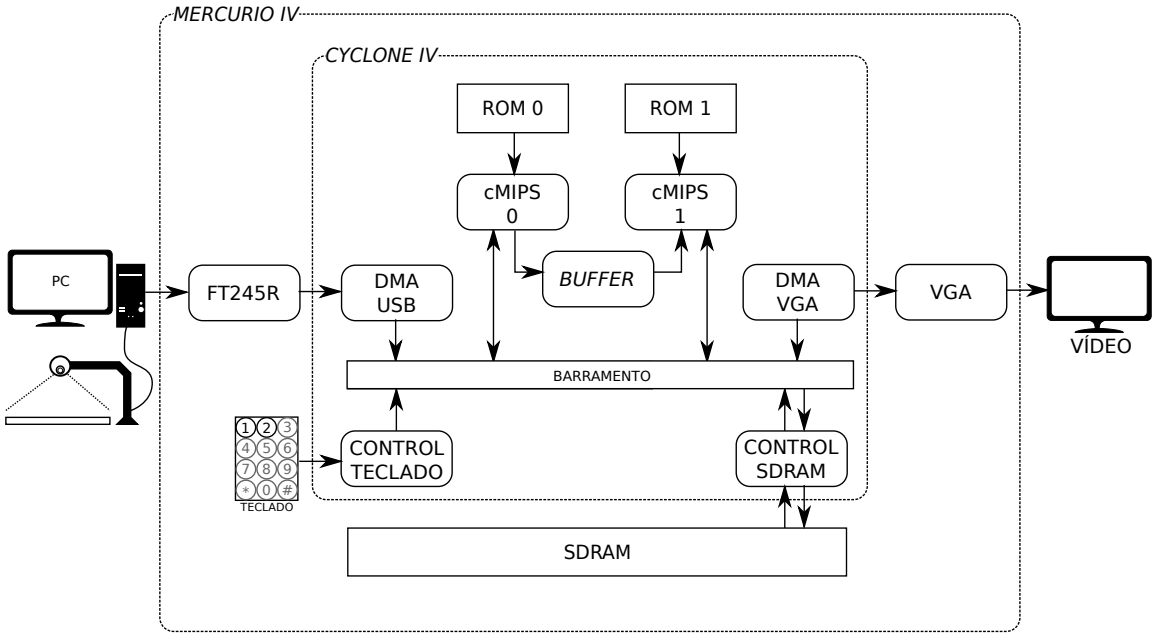


Figura 6.1: Diagrama de componentes do protótipo do cLUPA.

O conjunto de desenvolvimento Mercurio IV disponibiliza o componente FT245R para comunicação USB. Este componente está configurado pelo *hardware* como "escravo", assim, é necessária a utilização de um computador para capturar a imagen do documento impresso pela câmera e transmití-la para o protótipo no formato bitmap de 32 bits.

O MÓDULO-VGA do modelo cLUPA será substituído pelo componente VGA do conjunto de desenvolvimento Mercurio IV. O MÓDULO-VGA foi modelado com base no componente VGA e não necessita alteração no DMA-VGA com relação à esta comunicação, que utiliza o CONTROLADOR-VGA para gerenciar os sinais de controle para o componente VGA, mostrado na Figura 2.11.

O modelo cLUPA utiliza os arquivos prog0.bin e prog1.bin como arquivos de instruções e são carregados na memória ROM-0 e ROM-1 para os núcleos cMIPS-0 e cMIPS-1, respectivamente. As memórias ROM-0 e ROM-1 do protótipo serão implementadas utilizando os bits de armazenamento existentes no FPGA Cyclone IV. A memória ROM-0 terá a capacidade de 4 kbytes (32 Kbits) e a memória ROM-1 terá a capacidade de 2 Kbytes (16 Kbits), totalizando 48 Kbits, ou 8,2% dos 594 Kbits disponíveis. As memórias ROM-0 e ROM-1 serão inicializadas pelo compilador VHDL que gera o *bit stream* para a programação do FPGA Cyclone IV. As capacidades de armazenamento das memórias ROM-0 e ROM-1 foram definidas de modo a manter a relação $capacidade = 2^n$ e com base nos arquivos prog0.bin e prog1.bin do código do cLUPA com otimização de nível O3 que ocupam, respectivamente, 2,8 Kbytes e 1,9 Kbytes.

O BUFFER será implementado utilizando os bits de armazenamento do FPGA. A capacidade o BUFFER do modelo cLUPA é de 8 palavras de 32 bits e será mantida para a implementação do protótipo do cLUPA. A escolha dos bits de armazenamento do FPGA para a implementação do BUFFER se deve a capacidade de configuração destes bits como uma memória *dual-ported*.

A definição da configuração de ampliação e contraste utilizará o teclado numérico disponível no conjunto de desenvolvimento Mercurio IV. Substituíndo o Botões-de-Controle do modelo do cLUPA, o funcionamento permanecerá o mesmo, assim, o componente CONTROL-TECLADO será uma adaptação do Botões-de-Controle que terá como entrada somente os sinais que representam os botões “1” (contraste) e “2” (ampliação) do teclado numérico.

O componente SDRAM do conjunto de desenvolvimento Mercurio IV é uma memória com capacidade de armazenamento de 512 Mbits, interface de comunicação de 16 bits e

clock de 100 MHz. Na adaptação do modelo cLUPA, o componente SDRAM armazenará três imagens: imagemO, imagemP e imagemR, que ocuparão 29,5 Mbits (3,70 Mbytes) de SDRAM. A imagemO é a imagem recuperada e gravada pelo DMA-USB (imagem original), a imagemP é a imagem processada e gravada por cMIPS-1 e a imagemR é a última imagem processada que será enviada continuamente ao componente VGA. Quando o processamento de uma imagem finalizar, os ponteiros para as imagens imagemP e imagemR serão trocados, indicando ao DMA-VGA que a nova imagem já pode ser enviada ao VGA.

O componente CONTROL-SDRAM é responsável pelo acesso à memória SDRAM. Este componente é necessário devido a diferença de *clock* entre o FPGA Cyclone IV (50 MHz) e SDRAM (100 MHz) e à diferença da interface entre o cMIPS-1 (32 bits) e SDRAM (16 bits). Assim, o componente CONTROL-SDRAM fará o papel de um *buffer* que permitirá que dois acessos à SDRAM de 16 bits à um *clock* de 100 MHz torne-se um acesso de 32 bits à 50 MHz para o cMIPS e DMA-VGA. Outra responsabilidade do CONTROL-SDRAM é gerenciar o acesso à SDRAM pelos componentes DMA-USB, cMIPS-0, cMIPS-1 e DMA-VGA.

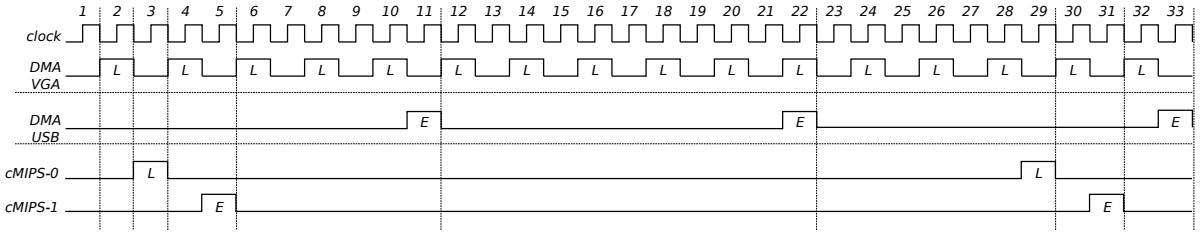


Figura 6.2: Diagrama de tempo para acesso à SDRAM (CZ_CI_O3).

A Figura 6.2 mostra o diagrama de tempo de acesso para escrita (E) e leitura (L) à memória SDRAM pelos componentes DMA-USB, cMIPS-0, cMIPS-1 e DMA-VGA. Os tempos foram estimados com base no resultado da simulação CZ_CI_O3, simulação de melhor desempenho (9,34 *qps*). Durante uma sequência de 2 ciclos, o componente DMA-VGA lê um *pixel* para exibição, ocupando apenas o segundo ciclo desta sequência, assim, o único momento em que outro componente pode acessar SDRAM é durante o primeiro ciclo. Desta forma, o CONTROL-SDRAM deve definir a prioridade de acesso para o DMA-VGA e os demais componentes competem pelo acesso no ciclo de relógio restante. O componente DMA-USB e cMIPS-0 não concorrem pelo acesso, pois estes não operam de forma paralela,

mas podem concorrer com cMIPS-1, então, o CONTROL-SDRAM, em caso de concorrência, dará prioridade de forma circular para DMA-USB ou cMIPS-0 e, posteriormente, cMIPS-1.

Em implementações preliminares de um único núcleo cMIPS utilizando o display LCD e o teclado numérico do conjunto de desenvolvimento Mercurio IV foram utilizados 4.892 dos 28.848 (17%) dos elementos lógicos do FPGA Cyclone IV, 4.429 funções combinacionais (15%), 1.631 registradores (6%) e 141.312 bits de memória (23%). Assim, estima-se que o sistema *multicore* cLUPA utilize aproximadamente 36% $\sim$ 40% dos elementos lógicos, 30% $\sim$ 35% das funções combinacionais, 12% $\sim$ 15% dos registradores e 46% $\sim$ 52% dos bits de memória disponíveis no FPGA.

CAPÍTULO 7

CONCLUSÃO

Este trabalho descreve o projeto do ampliador de documentos impressos cLUPA. O *hardware* do cLUPA é um SoC *multicore* com dois núcleos cMIPS que se comunicam através de uma fila, que foi modelado utilizando VHDL e compilado com GHDL. O *software* foi escrito com base nas funcionalidades de ampliação e aplicação de contraste presentes no xLUPA Embarcado e adaptado para arquitetura MIPS.

A avaliação de desempenho foi realizada através de simulações da execução do modelo gerado pelo GHDL em oito configurações: quatro configurações de contraste em duas configurações de ampliação. Através dos relatórios gerados com assert, é possível mensurar o tempo simulado de execução de cada operação para todas as configurações. As operações medidas foram: transferência da imagem do módulo MÓDULO-USB para RAM-0 (T-USB), operação de processamento inicial (C-0), operação de processamento final (C-1) e transferência da imagem da RAM-1 para o módulo MÓDULO-VGA (T-VGA).

Para todas as simulações, as operações T-USB e T-VGA apresentaram tempo de execução (simulado) sem variação. Isto acontece porque a utilização dos módulos do conjunto de desenvolvimento Mercurio IV no *hardware* do cLUPA que são independentes das configurações de contraste e ampliação. As operações C-0 e C-1 têm seus desempenhos dependentes das configurações. A configuração de ampliação é a configuração que mais impacta no desempenho geral do cLUPA. Como a ampliação ativada consiste em aumentar uma área central da imagem e utilizar a replicação dos *pixels* para seus vizinhos, o número total de *pixels* processados é de 76.800, equivalente a 1/4 do total de *pixels* da imagem que são processados com a ampliação desativada (307.200 *pixels*).

As configurações de contraste também afetam o desempenho do cLUPA nas operações C-0 e C-1. De maneira geral, a configuração de contraste desativada produz melhor desempenho, pois não realiza computação sobre os *pixels* e apenas os transfere entre as

memórias RAM-0 e RAM-1 através da fila de comunicação BUFFER. O desempenho intermediário foi obtido pela configuração de contraste cinza, que calcula a média aritmética das componentes r, g e b. As configurações de contraste verde e vermelho foram as configurações que demonstraram desempenho inferior às demais configurações de contraste devido à computação da luminosidade k, que é mais custosa do que para contraste cinza.

Para mensurar o desempenho foi utilizado a métrica de quadros por segundo (*qps*). Os resultados obtidos das simulações variam entre 9,34 *qps* para configuração de ampliação ativa e contraste inativo até 2,34 *qps* para configuração de contraste verde e ampliação desativada. O código executado no *hardware* do cLUPA foi compilado utilizando otimização de nível O2. É possível extrair uma melhora de desempenho para as configurações de contraste verde e vermelho com ampliação desativada utilizando uma versão do código do cLUPA compilado com otimização de nível O3, que elevou para 2,68 *qps* o desempenho destas configurações.

Através das simulações, também foi observado um desbalanceamento de processamento entre as operações C-0 e C-1 do *software* do cLUPA. O tempo de execução da operação C-1 ficou limitado ao tempo de execução de C-0. Assim, o código das configurações de desempenho inferior (contraste verde e vermelho para ampliação desativada) do cLUPA foi reescrito para uma nova versão visando balancear a carga de processamento. A versão cLUPAb tem melhor desempenho, alcançando 2,85 *qps*, equivalente à 1,05% melhor do que a versão cLUPA original.

As simulações foram realizadas com um relógio de 50 Mhz, coincidindo com a frequência do conjunto de desenvolvimento Mercurio IV, como discutido no Capítulo 6.

Trabalhos Futuros

Este trabalho tem como consequência das simulações, e na dependência do *hardware* que estiver disponível, uma série de experimentos que podem ser efetuados para testar as seguintes possibilidades:

Adaptação do modelo em VHDL e os códigos do cLUPA para o conjunto de desenvolvimento Mercurio IV, como apresentado na Capítulo 6.

O *hardware* do cLUPA não emprega uma memória *cache* de dados funcional. Assim, a modelagem de uma memória *cache* de nível 1 e realização de simulações para avaliar o desempenho é um trabalho futuro.

Os cálculos de contraste verde e vermelho utilizam somas em conjunto com divisão para simular uma soma saturada. A implementação de uma unidade de soma saturada no `cMIPS` e avaliação de desempenho é um trabalho futuro.

Os quadros 4.2, 4.3 e 4.4 apresentam os códigos para acesso ao BUFFER e DMA pelo núcleo cMIPS. Esses códigos utilizam a técnica de espera ocupada, através de laços de repetição while. Como trabalho futuro, a técnica de espera ocupada poderá ser substituída pelo uso de interrupções. Uma avaliação de desempenho comparando as duas técnicas deve ser realizada, pois interrupções no MIPS são relativamente custosas [56].

O padrão de imagem utilizado nas simulações contém resolução de 640x480 *pixels*. Esta resolução foi escolhida devido o conjunto de desenvolvimento Mercurio IV disponibilizar um controlador VGA que utiliza esta configuração. O padrão VGA suporta diversas configurações tais como 1280x720 *pixels* (resolução HD) e 1920x1080 *pixels* (resolução *Full*-HD). Assim, um trabalho que pode ser realizado é de alterar o módulo controlador-VGA, apresentado na Figura 2.11, para as especificações destas resoluções. Existe também a possibilidade de utilizar outro conjunto de desenvolvimento que utilize saída de vídeo no padrão HDMI (*High-Definition Multimedia Interface*), usado atualmente em uma grande variedade de dispositivos de vídeo, como monitores e televisores.

O *software* do cLUPA oferece espaço para otimização das operações que são executadas em paralelo. Para aproveitar melhor os recursos de *hardware* disponíveis, é necessário implementar o uso de *buffers* em memória principal para receber as imagens. No estado atual, a operação de transferência T-USB aguarda o término da operação C-0 para iniciar uma nova transferência devido o uso do mesmo espaço de memória. Este mesmo problema é encontrado na operação de transferência T-VGA. Com a adição de *buffers* de memória, estas operações de transferência podem ocorrer em paralelo as operações C-0 e C-1, escondendo o tempo de transferência.

A ampliação do cLUPA é feita através do método de replicação de vizinhos próximos.

Um algoritmo de interpolação que requer o acesso aos dados uma única vez. Existe a possibilidade de otimização de qualidade de imagem realizando um levantamento dos algoritmos de interpolação disponívels para ampliação e implementá-los no cLUPA.

BIBLIOGRAFIA

[1] Altera. *Cyclone IV Device Handbook*. Disponível em `http://www.altera.com/literature/lit-cyclone-iv.jsp`. Acessado em Nov/2014.

[2] Altera. *Stratix IV Device Handbook*. Disponível em `http://www.altera.com/literature/lit-stratix-iv.jsp`. Acessado em Nov/2014.

[3] Jan Axelson. *USB complete everything you need to develop custom USB peripherals*. Lakeview Research, Madison, WI, 2005.

[4] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, e Konrad Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, maio de 2005.

[5] Odair Moreira de Souza Balansin, Jorge Bidarra, Diego Rodrigo Hachmann, e Cleiton Fiatkoski. A Tecnologia Assistiva xLupa - Um software livre ampliador e leitor de tela para pessoas com baixa visão. *Revista Brasileira de Educação Especial*, Cascavel, Paraná, Brasil, 2010.

[6] M. Berekovic, S. Flugel, H.-J. Stolberg, L. Friebe, S. Moch, M.B. Kulaczewski, e P. Pirsch. HiBRID-SoC: a multi-core architecture for image and video applications. *ICIP 2003 - International Conference on Image Processing*, volume 3, páginas III–101–4 vol.2, 2003.

[7] Rita Bersch. Introdução à tecnologia assitiva. 2008. CEDI - Centro Especializado em Desenvolvimento Infantil.

[8] Jorge Bidarra, Clodis Boscarioli, e Sarajane Marques Peres. Avaliando a ferramenta xLupa como um recurso para educação especial e inclusiva. *Simpósio Brasileiro de Informática na Educação (XX SBIE)*, 2009.

[9] Jorge Bidarra, Clodis Boscarioli, e Sarajane Marques Peres. Software xLupa - um ampliador de tela para auxílio na educação de alunos com baixa visão. *Revista Brasileira de Educação Especial*, 17:151–172, 2011.

[10] Jorge Bidarra, Clodis Boscarioli, e Claudia Brandelero Rizzi. *Usabilidade, Acessibilidade e Inteligibilidade Aplicadas em Interfaces para Analfabetos, Idosos e Pessoas com Deficiência*, volume 1, capítulo xLupa - Um Ampliador de Tela com Interface Adaptativa para Pessoas com Baixa Visão, páginas 23–30. CPqD - Centro de Pesquisa e Desenvolvimento em Telecomunicações, 2009.

[11] Cairo. Disponível em `http://cairographics.org/`. Acessado em Dez/2013.

[12] Cambridge in Colour. Digital Image Interpolation. Disponível em `http://www.cambridgeincolour.com/tutorials/image-interpolation.htm`. Acessado em Ago/2014.

[13] Thiago N. C. Cardoso, Celina G. do Val, José A. Nacif, Antônio O. Fernandes, e Claudionor N. C. Jr. Memory aspects of dual core processor design. *VIII Microelectronics Students Forum*, 2008.

[14] Yao-Wen Chang, Martin D.F. Wong, e Chak-Kuen Wong. Programmable Logic Devices, 1999. Department of Computer and Information Science. National Chiao Tung University. Relatório Técnico.

[15] Macnica DHW. *Mercurio IV: Manual do Usuário.* Disponível em `http://www.macnicadhw.com.br/products/mercurion-4-devkit-board`. Acessado em Nov/2014.

[16] Fernando Fernandes dos Santos. Explorando recursos de paralelismo no MPSoC Heterogêneo DM3730, 2014. Trabalho de Conclusão de Curso. Bacharelado em Informática. Unioeste - Cascavel. Orientador: Marcio Seiji Oyamada.

[17] Fernando Fernandes dos Santos, Jorge Bidarra, e Marcio Oyamada. Implementação de um ampliador de documentos impressos embarcado. To be published.

[18] S. Edwards, L. Lavagno, E.A. Lee, e A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[19] Jr. Philip Enslow. Multiprocessor Organization – a Survey. *ACM Comput. Surv.*, 9(1):103–129, março de 1977.

[20] Liu Feng, Wang Chao, e Zhang Dong. Heterogeneous multi-core SOC architecture for MPEG decoding. *ICSICT 2008 - 9th International Conference on Solid-State and Integrated-Circuit Technology.*, páginas 2188–2191, 2008.

[21] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[22] GHDL. Disponível em `http://home.gna.org/ghdl/ghdl/index.html`. Acessado em Nov/2014.

[23] GNU. GNU Lesser General Public License - LGPL. Disponível em `http://www.gnu.org/copyleft/lesser.html`. Acessado em Dez/2013.

[24] GTK+. Disponível em `http://www.gtk.org/`. Acessado em Dez/2013.

[25] Diego Rodrigo Hachmann, Paulo Wesley Santiago, Jorge Bidarra, e Marcio Oyamada. Um ampliador de tela embarcado utilizando aquiteturas heterogêneas. *Fórum Internacional Software Livre - Workshop de Software Livre (XXII FISL)*, 2011.

[26] John Hennessy, Norman Jouppi, Steve Przybylski, Christopher Rowen, e Thomas Gross. Design of a High Performance VLSI Processor. Relatório Técnico 236, Stanford University, 1983.

[27] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, e John Gill. MIPS: A Microprocessor Architecture. *SIGMICRO Newsl.*, 13(4):17–22, outubro de 1982.

[28] John L. Hennessy e David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[29] Roberto André Hexsel. A VHDL model for the classical five stage pipeline. 2013. Universidade Federal do Paraná - Departamento de Informática. Relatório Técnico.

[30] M.D. Hill e M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[31] IBGE. Censo 2010 - Pessoas de 5 anos ou mais de idade, por existência ou não de pelo menos uma das deficiências investigadas e alfabetização, segundo o sexo e os grupos de idade. Disponível em `http://www.ibge.gov.br/`. Acessado em Nov/2013.

[32] Carson Optical Inc. DR-200 ezRead Eletronic Reading Aid. Disponível em `http://www.carsonoptical.com/videoDisplay/Magnifiers/Other/DR-200`. Acessado em Nov/2013.

[33] Mattingly Low Vision Inc. The Mattingly Wired/Wireless Mouse CCTV. Disponível em `http//www.mattinglylowvision.com/content.cfm?'n=mvideo`. Acessado em Nov/2013.

[34] Integrated Device Technology, Inc. *IDT R30xx Family Software Reference Manual*, 1994. Revisão 1.0.

[35] M.A. Kinsy, M. Pellauer, e S Devadas. Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System. *International Conference on Field Programmable Logic and Applications (FPL)*, páginas 356–362, 2011.

[36] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, e Parthasarathy Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, novembro de 2005.

[37] Ian Kuon, Russel Tessier, e Jonathan Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2007.

[38] Fabiany Lamboia e Roberto André Hexsel. An Analisys of Dynamic Instruction Usage with 32 Bit MIPS, PowerPC and SPARC Processors on Embedded Applications. Relatório Técnico RT DINF 001/2011, Universidade Federal do Paraná - Departamento de Informática, Curitiba, Paraná, Brasil, 2011.

[39] Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue*, 11(7):40, 2013.

[40] Future Technology Device Internacional Ltd. FT245R - USB FIFO IC. Disponível em `http://www.ftdichip.com/Products/ICs/FT245R.htm`. Acessado em Dez/2014.

[41] Peter Marwedel. *Embedded System Design*. Editora Springer, Secaucus, Nova Jersey, Estados Unidos da America, 2 edition, 2006.

[42] Linux media infrastructure API. V4L2 Specification. Disponível em `http://v4l2spec.bytesex.org/spec/`. Acessado em Nov/2013.

[43] Microsoft. BITMAPV5HEADER structure. Disponível em `https://msdn.microsoft.com/en-us/library/dd183381(v=vs.85).aspx`. Acessado em Nov/2013.

[44] S. Moch, M. Bereković, H. J. Stolberg, L. Friebe, M. B. Kulaczewski, A. Dehnhardt, e P. Pirsch. HIBRID-SoC: A Multi-core Architecture for Image and Video Applications. *SIGARCH Comput. Archit. News*, 32(3):55–61, setembro de 2003.

[45] Zainalabedin Navabi. A high-level language for design and modeling of hardware. *Journal of Systems and Software*, 18(1):5–18, 1992.

[46] Kunle Olukotun e Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, setembro de 2005.

[47] Marcio Oyamada, Jorge Bidarra, e Clodis Boscarioli. PlatMult: a multisensory platform with web accessibility features for low vision users. *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, páginas 62:1–62:2. ACM, 2013.

[48] Marcio Oyamada e Fernando Fernandes dos Santos. Correspondência privada. Em Out/2013 à Mar/2014.

[49] David A. Patterson e John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[50] Douglas L. Perry. *VHDL: Programming by Example*. McGraw-Hill, fourth edition, 2002.

[51] Ike Presley. A Comparison of Three Low-Cost, Hand-Held, Camera-Model Video Magnifiers: Vision Booster Magnifier, Carson DR-200 ezRead, and Wireless Electronic Reading Aid. *AFB AccessWorld Magazine*, 12, 2011.

[52] Qualcomm. Processadores Snapdragon 800. Disponível em `http://www.qualcomm.com.br/snapdragon/processors/800`. Acessado em Dez/2014.

[53] Freedom Scientific. ONYX Flexible arm Portable Magnification Camera. Disponível em `http://www.freedomscientific.com/products/lv/onyx-sa-product-page.asp`. Acessado em Nov/2013.

[54] Javier Serrano. Course on Digital Signal Processing. 2008. CAS - CERN Accelerator School.

[55] William Stallings. *Arquitetura de Computadores*. Pearson Pratice Hall, São Paulo, 8 edition, 2010.

[56] Dominic Sweetman. *See MIPS Run, Second Edition*. Morgan Kaufmann Publishers Inc., 2006.

[57] Jorge Tortato Junior. Projeto e Implementação de Multiprocessador em Dispositivos Lógicos Programáveis. Dissertação de mestrado, Informática, Setor de Ciências Exatas, Universidade Federal do Paraná, 2009.

[58] Christiane Gresse von Wangenheim e Aldo von Wangenheim. *Raciocínio Baseado em Casos*. Manole, 1 edition, 2003.

[59] WHO - World Health Organization. International Statistical Classification of Diseases and Related Health Problems. Disponível em `http://www.who.int/classifications/icd/en/`. Acesso em Jan/2015.

[60] Xilinx. Introduction to FPGA Design with Vivado High-Level Syntesis. 2013. Software Manual (UG998).

APÊNDICE A

ENTIDADES VHDL

Esta Seção apresenta as entidades criadas e adicionadas ao arquivo de testes do cMIPS original para a implementação do sistema *multicore* cLUPA.

Buffer de Comunicação direta

O Buffer de Comunicação Direta (BCD) é composto por cinco componentes, como apresentado na figura 4.4: RAM-FIFO, WORD-RD-CONT, WORD-WR-CONT, ADDR-RD e ADDR-WR. O componente CONTROLE é simbólico na figura 4.4 e é um conjunto de circuitos combinacionais e não é representado por uma entidade.

O componente RAM-FIFO, implementado com a entidade ram_dualBCD, é apresentado no Quadro A.1.

```
1  entity ram_dualBCD is
2    generic (N_WORDS : integer := NUM_WORDS_CB);
3    port (
4      data  : in reg32;                              -- Dado de escrita
5      raddr : in natural range 0 to N_WORDS - 1;     -- End. leitura
6      waddr : in natural range 0 to N_WORDS - 1;     -- End. escrita
7      we    : in std_logic;                          -- Sinal de escrita
8      rclk  : in std_logic;                          -- Clock leitura
9      wclk  : in std_logic;                          -- Clock escrita
10     rd    : in std_logic;                          -- Sinal de leitura
11     q     : out reg32                              -- Dado de leitura
12     );
13 end ram_dualBCD;
```

Quadro A.1: Entidades RAM-FIFO (ram_dualBCD)

RAM-FIFO contém duas portas independetes para dados de 32 bits: data (escrita) e q (leitura). Para endereçar esses dados, duas portas que recebem valores dos endereçadores são: waddr (escrita) e raddr (leitura). Dois sinais são usados para ativar os módulos de escrita (we) e leitura rd da memória (ativos com valor lógico 0). RAM-FIFO é uma memória síncrona com dois *clocks* independetes para leitura (rclk) e escrita (wclk), porém o módulo principal sincroniza esses dois *clocks* com um único sinal clk.

Para contar e endereçar os dados na memória RAM-FIFO é usada a mesma entidade counterAddrNat, apresentada no Quadro A.2, para criar os endereçadores ADDR-RD e ADDR-WR.

```
entity counterAddrNat is
  generic (INITIAL_VALUE: regLine := (others => '0'));
  port(clk: in  std_logic;
       rst: in  std_logic;
       en : in  std_logic;          -- Sinal de ativação
       Q  : out  natural range 0 to (NUM_WORDS_CB - 1)); -- End. para BCD
  attribute ASYNC_SET_RESET of rst : signal is true;
end counterAddrNat;
```

Quadro A.2: Entidades ADDR-RD e ADDR-WR (counterAddrNat)

A entidade counterAddrNat contém sinais de *clock* (clk) e *reset* (rst), bem como um sinal de ativação en e sua saída q é um número natural convertido para sua representação binária pelo VHDL. O funcionamento de counterAddrNat é contar, de forma circular, sempre na subida do *clock*, se e somente se, o sinal en estiver ativo (valor lógico 0).

Os componentes mantenedores de estados WORD-RD-CONT e WORD-WR-CONT são implementados pela entidade counterUD, apresentado no Quadro A.3. O objetivo do mantenedor é indicar ao núcleo quantas palavras podem ser gravadas (cMIPS-0) ou quantas palavras podem ser lidas (cMIPS-1). O funcionamento do contador counterUD é semelhante ao do counterAddrNat. A diferença é que existe um sinal para incremento (valor lógico 1) ou decremento do valor atual. Contém sinais de *clock* (clk) e *reset* (rst) e o sinal de ativação en.

```
entity counterUD is
  generic (INITIAL_VALUE: reg32 := (others => '0'));
  port(clk: in  std_logic;
       rst: in  std_logic;
       inc: in  std_logic;  -- Sinal para dec/incremento (0/1)
       en : in  std_logic;  -- Sinal de ativação
       Q  : out reg32);     -- status do BCD
  attribute ASYNC_SET_RESET of rst : signal is true;
end counterUD;
```

Quadro A.3: Entidade counterUD (WORD-RD-CONT e WORD-WR-CONT)

O BCD e seus componentes são implementados pela entidade circularBuffer, apresentada no Quadro A.4.

```
entity circularBuffer is
  port (
    clk    : in std_logic;
    clk4x  : in std_logic;
    rst    : in std_logic;
    -- cMIPS-0
    w_sel     : in  std_logic;  -- Sinal de seleção
    w_data_in : in  reg32;      -- Dados para gravação
```

```
9     w_we      : in  std_logic;  -- gravação (1) ou leitura de estado (0)
10    w_out_st  : out reg32;      -- Dados de estado (capacidade restante)
11    w_rdy     : out std_logic;  -- iwait
12    -- cMIPS-1
13    r_sel     : in std_logic;   -- Sinal de seleção
14    r_addr    : in  reg32;      -- Endereço para leitura de dados ou estado
15    r_out     : out reg32;      -- Dados de estado (palavras)
16    r_rdy     : out std_logic   -- iwait
17  );
18 end entity;
```

Quadro A.4: Entidade circularBuffer (BCD)

Cada núcleo cMIPS se comunica com apenas uma parte do módulo BCD. Desta forma, os sinais de comunicação com núcleo são mínimos para suas funções: cMIPS-0, responsável pela gravação de palavras no BCD contém os sinais de seleção (w_sel), dados para gravação (w_data_in), determinação de gravação ou leitura de estado (w_we), estado do BCD (w_out_st) e sinal de espera (w_rdy). O cMIPS-1 contém os sinais de seleção (r_sel), de endereço (r_addr), que determina a leitua de uma palavra ou estado, dados ou estado (r_out) e sinal de espera (r_rdy).

MÓDULO USB

O componente USB, apresentado na Figura 4.5, é composto por dois módulos: DMA-USB e MÓDULO-USB. Cada módulo foi criado a partir de uma entidade específica. O MÓDULO-USB é implementado a partir da entidade usb_fake, apresentada no Quadro A.5.

```
1 entity usb_fake is
2   generic (INPUT_FILE_NAME : string := "img.data");
3   port (rst       : in  std_logic;
4         clk       : in  std_logic;
5         usb_rd_n  : in  std_logic;  -- Sinal de leitura de dados
6         usb_rfx_n : out std_logic;  -- Sinal de dados disponíveis
7         data      : out reg8);      -- Dados
8 end usb_fake;
```

Quadro A.5: Entidade usb_fake (MÓDULO-USB)

O objetivo da entidade usb_fake é emular a leitura de um fluxo de bytes, vindos do arquivo img.data, e repassar, *byte-a-byte*, para o módulo DMA-USB. Além dos sinais de *clock* (clk) e *reset* (rst), a entidade contém um sinal de 1 byte (data) para transferência de dados e dois sinais de controle de comunicação com o DMA-USB: usb_rd_n que determina a passagem de bytes por data e usb_rfx_n que indica ao DMA-USB que existem bytes a serem lidos.

O módulo DMA-USB é implementado pela entidade dma_usb, apresentada no Quadro A.6. O multiplexador/árbitro MULTIPLEXADOR-DMA foi implementado dentro do módulo DMA-USB, desta forma, este módulo se comunica diretamente com o núcleo cMIPS-0 e a memória RAM-0.

```
entity dma_usb is
  port (rst        : in  std_logic;
        clk        : in  std_logic;
        phi1       : in  std_logic;
        -- USB
        usb_rd_n   : out std_logic;  -- Sinal de leitura de dados
        usb_rfx_n  : in  std_logic;  -- Sinal de "dados disponíveis"
        data       : in  reg8;       -- Dados
        -- RAM
        sel_ram    : out std_logic;
        rdy_ram    : in  std_logic;
        wr_ram     : out std_logic;
        addr_ram   : out reg32;
        data_ram   : out reg32;
        b_sel_ram  : out reg4;
        -- cMIPS
        c_sel      : in  std_logic;   -- Sinal de seleção
        c_data_in  : in  reg32;       -- Gravação / end. inicial
        c_data_out : out reg32;       -- Estado
        c_addr     : in  reg32;
        c_we       : in  std_logic;   -- Sinal de gravação(1)/leitura(0)
        c_rdy      : out std_logic    -- iwait
          );
end dma_usb;
```

Quadro A.6: Entidade dma_usb (DMA-USB)

A entidade dma_usb comunica-se com a entidade usb_fake pelos sinais conhecidos data, usb_rfx_n e usb_rfx_n. A comunicação com a memória RAM-0 é dada pelos sinais com sufixo _ram e a comunicação com o núcleo cMIPS-0 é dada pelos sinais com prefixo c_, todos os sinais são equivalente aos sinais apresentados na Figura 2.3 e não serão comentados nesta seção.

MÓDULO VGA

O componente VGA, apresentado na Figura 4.6, é composto por dois módulos: DMA-VGA e MÓDULO-VGA. O módulo MÓDULO-VGA é implementado a partir da entidade vga, apresentada no Quadro A.7.

```
entity vga is
  Generic(CLOCK_PERIOD      : integer := 20);
```

```
3     port(
4        signal CLOCK_50MHz_i : in std_logic;
5        signal phi2          : in std_logic; -- não existe em vga real, usado
             apenas para escrever em arquivo
6        signal closeFile     : in std_logic; -- não existe em vga real, usado
             apenas para fechar o arquivo  > '1' fecha e encerra
7        signal R_in          : in reg8;
8        signal G_in          : in reg8;
9        signal B_in          : in reg8;
10       signal A_in          : in reg8;
11       signal VGA_HS_i      : in std_logic;   -- VGA_controlador; indica
             nova linha
12       signal VGA_VS_i      : in std_logic);  -- VGA_controlador; indica
             novo frame
13 end vga;
```

Quadro A.7: Entidade vga (MÓDULO-USB)

Com o o propósito de emular o componente VGA presente no conjunto de desenvolvimento Mercurio IV, a entidade vga recebe os sinais das componentes de cor vermelho (R_in), verde (G_in) e azul (B_in) do *pixel* enviados pelo componente DMA-VGA. Para as simulações, a componente alfa/transparência (A_in) também é enviado. Como resultado da emulação, o *pixel* é gravado no arquivo vga.data. A gravação requer o sinal de clock secundário phi2 para ativar a escrita em arquivo, bem como o sinal closeFile para fechar o arquivo ao encerrar o fluxo de *pixels* das imagens processadas. Os sinais VGA_HS_i e VGA_VS_i são usados para controlar, respectivamente, as alterações de coluna e linha de *pixels* e são enviados pelo módulo DMA-VGA, que contém o CONTROLADOR-VGA disponível no conjunto de desenvolvimento Mercurio IV.

O módulo DMA-VGA é criado a partir da entidade dma_vga, apresentada no Quadro A.8.

```
1  entity dma_vga is
2    port (rst        : in  std_logic;
3          clk        : in  std_logic;
4          phi0       : in  std_logic;
5          phi1       : in  std_logic;
6          phi2       : in  std_logic;
7          -- RAM
8          sel_ram    : out std_logic;
9          rdy_ram    : in  std_logic;
10         wr_ram     : out std_logic;
11         addr_ram   : out reg32;
12         data_ram   : in  reg32;
13         b_sel_ram  : out reg4;
14         -- cMIPS
15         c_sel      : in  std_logic;   -- Sinal de seleção
```

```
16          c_data_in   : in  reg32;       -- Leitura / end. inicial
17          c_data_out  : out reg32;       -- Estado
18          c_addr      : in  reg32;
19          c_we        : in  std_logic;   -- Sinal de gravação(1)/leitura(0)
20          c_rdy       : out std_logic    -- iwait
21            );
22  end dma_vga;
```

Quadro A.8: Entidade dma_vga (DMA-USB)

A entidade dma_vga se comunica com a entidade vga através dos sinais R_in, G_in, B_in, VGA_HS_i e VGA_VS_i já apresentados. Semelhante à entidade dma_usb, a comunicação com a memória RAM-1 é dada pelos sinais com sufixo _ram e a comunicação com o núcleo cMIPS-1 é dada pelos sinais com prefixo c_, todos os sinais são equivalente aos sinais apresentados na Figura 2.3 e não serão comentados nesta seção.

APÊNDICE B

CÓDIGO DO CLUPA

Esta Seção apresenta as principais áreas do código das aplicações cLUPA e cLUPAb (com balanceamento de carga). O código do cLUPA é dividido em quatro arquivos: core0.c realiza as atividades do módulo cLUPA-0 (Figura 4.8), core1.c realiza as atividades do módulo cLUPA-1 (Figura 4.9), cLUPAContraste.h contém os algoritmos usados para aplicação de contraste e cLUPA.h contém as principais definições de configuração das imagens, espaço de memória e quantidade de imagens a serem simuladas. O código cLUPAb difere do código do cLUPA, principalmente, nos algoritmos de aplicação de contrastes, e nas chamadas das funções de contraste pelos módulos `cLUPAb-0` e `cLUPAb-1`. O arquivo de definições cLUPA.h é comum entre as duas aplicações.

cLUPA-0

```
1  for (framestream = 0; framestream < frameStream; framestream++){
2    readInt(&zoomCfg);
3    readInt(&contrastecfg);
4
5    // Endereço inicial, Palavras, # palavras
6    // inicia transferência DMA -> RAM
7    dmaUSB_init(dataImgAddressI, 4, totalPixelImagem);
8    dma_t_ok = dmaUSB_st();
9    while(dma_t_ok != 0){ // espera ativa
10     dma_t_ok = dmaUSB_st(); }
11
12   // Cálculo de end. inicial
13   img = (int*) dataImgAddressI/4;
14    if (zoomCfg == ZINACTIVE) { // Zoom inativo
15      for(l = 0; l < totalPixelImagem; l++){
16        bcd_max = bcdWSt();
17        while(bcd_max == 0){  // espera ativa
18          bcd_max = bcdWSt(); }
19        pixel = (int) *(img + l); // recupera pixel
20        // Aplica contraste
21        switch(contrastecfg){
22          case CINACTIVE:
23            // Não existe alteração do pixel; break;
24          case CVERMELHO:
25            pixel = contrasteVermelho(pixel); break;
26          case CVERDE:
27            pixel = contrasteVerde(pixel); break;
28          case CCINZA:
29            pixel = contrasteCinza(pixel); break;
30          }// switch(contrastecfg)
31         // Escrita no BCD
32         bcdWWr(pixel);
33     }// for(l = 0; l < totalPixelImagem; l++)
```

```
34      } else { // if (zoomCfg == ZINACTIVE)
35          // Zoom Ativo
36          linha   = initialYZoomed;
37          coluna  = initialXZoomed;
38          linhaM  = finalYZoomed;
39          colunaM = finalXZoomed;
40          sizeL = larguraImagem;
41
42          for(l = linha; l < linhaM; l++){
43            for(c = coluna; c < colunaM; c++){
44              bcd_max = bcdWSt();
45              while(bcd_max == 0){  // espera ativa
46                bcd_max = bcdWSt(); }
47              pixel = (int) *(img + l*sizeL + c);
48              // Aplica contraste
49              switch(contrastecfg){
50                case CINACTIVE:
51                    // Não existe alteração do pixel; break;
52                case CVERMELHO:
53                    pixel = contrasteVermelho(pixel); break;
54                case CVERDE:
55                    pixel = contrasteVerde(pixel); break;
56                case CCINZA:
57                    pixel = contrasteCinza(pixel); break;
58              }
59              // Escrita no BCD
60              bcdWWr(pixel);
61            } // for(c = coluna; c < colunaM; c++)
62          } // for(l = linha; l < linhaM; l++)
63      } // if (zoomCfg == ZINACTIVE) else
64  } // for (framestream = 0; framestream < frameStream; framestream++)
65    // Segura a execução do cMIPS_0 até cMIPS_1 finalizar
66    while(1){ // do nothing!  }
```

Quadro B.1: Código parcial do módulo cLUPA-0 (core0.c)

cLUPA-1

```
1   for (framestream = 0; framestream < frameStream; framestream++){
2     readInt(&zoomCfg);
3     readInt(&contrastecfg);
4
5     if (zoomCfg == ZINACTIVE) { // Zoom inativo
6         // escrita em memória sem ZOOM
7         for(l = 0; l < totalPixelImagem; l++){
8             bcd_max = bcdRSt();
9             while(bcd_max <= 0){  // espera ativa
10              bcd_max = bcdRSt(); }
11
12            // leitura do buffer
13            pixel = bcdRRd();
14            // escrita em memória
15            mem = img + l;
16            *mem = (int) pixel;
17          }
18    } else {                    // Zoom Ativo
```

```
19      for(l = 0; l < alturaImagem; l++){
20        for(c = 0; c < larguraImagem; c++){
21            bcd_max = bcdRSt();
22            while(bcd_max <= 0){  // pooling?
23              bcd_max = bcdRSt(); }
24
25            // leitura do buffer
26            pixel = bcdRRd();
27            // Replica para posição original
28            mem = img + l*larguraImagem + c;
29            *mem = (int) pixel;
30            // Replica para posição original+1
31            mem = img + l*larguraImagem + (c+1);
32            *mem = (int) pixel;
33            // Replica para próxima linha na posição original
34            mem = img + (l+1)*larguraImagem + c;
35            *mem = (int) pixel;
36            // Replica para próxima linha na posição original+1
37            mem = img + (l+1)*larguraImagem + (c+1);
38            *mem = (int) pixel;
39            c++;
40        } // for(c = 0; c < larguraImagem; c++)
41        l++;
42      } // for(l = 0; l < alturaImagem; l++)
43    } // if (zoomCfg == ZINACTIVE) else
44
45    // Envio para VGA
46    // dmaVGA_init(a, w, s);
47    dmaVGA_init(dataImgAddressI, 4, totalPixelImagem);
48    dma_t_ok = dmaVGA_st();
49    while(dma_t_ok != 0){
50      dma_t_ok = dmaVGA_st(); }
51  } // for (framestream = 0; framestream < frameStream; framestream++){
52  dmaVGA_closeFile(); // necessário fechar arquivo na simulação
```

Quadro B.2: Código parcial do módulo cLUPA-1 (core1.c)

Contrastes - cLUPA

```
1   // Pixel
2   typedef unsigned int Pixel[4];  // ARGB
3
4   int contrasteVerde(int j){
5     Pixel p;
6     int corPixel, r, g, b;
7
8     r = j; g = j; b = j;
9     p[RED]   = (unsigned int) ((r>>24) & 0xff);
10    p[GREEN] = (unsigned int) ((g>>16) & 0xff);
11    p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
12
13      // verde é mais sensível
14    corPixel = (int) (p[RED] + p[RED] + p[GREEN] +
15                      p[GREEN] + p[GREEN] + p[BLUE]) / 6;
16
17    p[RED]   = 0;    p[BLUE]  = 0;
```

```
18    p[GREEN] =(p[GREEN] * corPixel) / 255;
19
20    if(p[GREEN] < 127) p[GREEN] = 0;
21    j = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;  // 0xff alfa
22    return j;
23  }
24
25  int contrasteVermelho(int j){
26    Pixel p;
27    int corPixel, r, g, b;
28
29    r = j; g = j; b = j;
30    p[RED]   = (unsigned int) ((r>>24) & 0xff);
31    p[GREEN] = (unsigned int) ((g>>16) & 0xff);
32    p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
33
34    corPixel = (int) (p[RED] + p[RED] + p[GREEN] +
35                      p[GREEN] + p[GREEN] +  p[BLUE]) / 6;
36
37    p[RED]   = (p[RED] * corPixel) / 255;
38    p[GREEN] = 0; p[BLUE]  = 0;
39
40    if(p[GREEN] < 127) p[GREEN] = 0;
41    j = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;  // 0xff alfa
42    return j;
43  }
44
45  int contrasteCinza(int j){
46    Pixel p;
47    int corPixel, r, g, b;
48
49    r = j; g = j; b = j;
50    p[RED]   = (unsigned int) ((r>>24) & 0xff);
51    p[GREEN] = (unsigned int) ((g>>16) & 0xff);
52    p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
53
54    corPixel = p[RED] + p[GREEN] + p[BLUE];
55    corPixel = corPixel/3;
56    p[RED]   = corPixel;
57    p[GREEN] = corPixel;
58    p[BLUE]  = corPixel;
59
60    j = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;  // 0xff alfa
61    return j;
62  }
```

Quadro B.3: Código de contrastes do cLUPA (cLupaContraste.h)

cLUPAb-0

O código `cLUPAb-0` e `cLUPAb-1` diferem dos códigos `cLUPA-0` e `cLUPA-1`, respectivamente, apenas em duas regiões: a região que determina a chamada da função de contraste e aplicação de ampliação e a região de controle de acesso dos *pixels* na `RAM`, que foi movido para as funções de contraste. Desta forma, os Quadros B.4 e B.5 apresentam somente os

trechos de códigos diferentes referentes as chamadas das funções e aplicação de ampliação.

```
1  img = (int*) dataImgAddressI/4;
2  if (zoomCfg == ZINACTIVE) { // Zoom inativo
3      switch(contrastecfg){
4        case CINACTIVE:
5           contrasteInativoCore0(); break;
6          case CVERMELHO:
7              contrasteVermelhoCore0(); break;
8          case CVERDE:
9              contrasteVerdeCore0(); break;
10         case CCINZA:
11             contrasteCinzaCore0(); break;
12       } // switch(contrastecfg)
13   } else { // if (zoomCfg == ZINACTIVE)
14       switch(contrastecfg){
15         case CINACTIVE:
16             contrasteZInativoCore0(); break;
17         case CVERMELHO:
18             contrasteZVermelhoCore0(); break;
19         case CVERDE:
20             contrasteZVerdeCore0(); break;
21         case CCINZA:
22             contrasteZCinzaCore0(); break;
23       }
24   } // if (zoomCfg == ZINACTIVE) else
```

Quadro B.4: Código parcial do módulo cLUPAb-0 (core0b.c)

cLUPAb-1

```
1  for (framestream = 0; framestream < frameStream; framestream++){
2    if (zoomCfg == ZINACTIVE) { // Zoom inativo
3        switch(contrastecfg){
4          case CINACTIVE:
5              contrasteInativoCore1(); break;
6          case CVERMELHO:
7              contrasteVermelhoCore1(); break;
8          case CVERDE:
9              contrasteVerdeCore1(); break;
10         case CCINZA:
11             contrasteInativoCore1(); break;
12       }
13   } else {                    // Zoom Ativo
14       switch(contrastecfg){
15         case CINACTIVE:
16             contrasteZCore1(); break;
17         case CVERMELHO:
18             contrasteZVermelhoCore1(); break;
19         case CVERDE:
20             contrasteZVerdeCore1(); break;
21         case CCINZA:
22             contrasteZCore1(); break;
23       }
24   } // if (zoomCfg == ZINACTIVE) else
```

Quadro B.5: Código parcial do módulo cLUPAb-1 (core1b.c)

Contrastes - cLUPAb

```
1  // Pixel
2  typedef unsigned int Pixel[4];  // ARGB
3
4  unsigned int bcdW_max = 0;            // Capacidade inicial BCDW
5  unsigned int bcdR_max = BCDCapacity; // Capacidade inicial BCDR
6  unsigned int l, c;                    // indexadores de linhas e colunas
7  unsigned int linha, coluna;
8  unsigned int linhaM, colunaM;         // Max
9  unsigned int sizeL;                   // tamanho da linha
10
11 //========== VERDE S/Z
12 void contrasteVerdeCore0(){
13   Pixel p;
14   int pixel; corPixel, j, r, g, b;
15
16   for(l = 0; l < totalPixelImagem; l++){
17     bcdW_max = bcdWSt();
18
19     while(bcdW_max <= 1){  // espera ativa
20       bcdW_max = bcdWSt(); }
21
22     pixel = (int) *(img + l); // recupera pixel
23     r = pixel; g = pixel; b = pixel;
24     p[RED]   = (unsigned int) ((r>>24) & 0xff);
25     p[GREEN] = (unsigned int) ((g>>16) & 0xff);
26     p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
27
28    corPixel = (int) (p[RED]   + p[RED]   + p[GREEN] +
29                      p[GREEN] + p[GREEN] + p[BLUE]);
30   // escrita dupla no BCD
31   bcdWWr((int) p[GREEN]);
32   bcdWWr(corPixel);
33   } // for(l = 0; l < totalPixelImagem; l++)
34 } // contrasteVerdeCore0()
35
36 void contrasteVerdeCore1(){
37   Pixel p;
38   int corPixel, pixel, r, g, b;
39   unsigned int *mem;
40
41   for(l = 0; l < totalPixelImagem; l++){
42     bcdR_max = bcdRSt();
43     while(bcdR_max <= 1){  // espera ativa
44       bcdR_max = bcdRSt(); }
45
46     // leitura dupla do BCD
47     p[GREEN] = bcdRRd();
48     corPixel = bcdRRd();
49     corPixel = corPixel / 6;
50
51     p[RED]   = 0;
52     p[GREEN] = (p[GREEN] * corPixel) / 255;
53     p[BLUE]  = 0;
54
55     if(p[GREEN] < 127) p[GREEN] = 0;
56     // 0xff alfa
```

```
57      pixel = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;
58
59      mem = img + l;
60      *mem = (int) pixel;
61    } // for(l = 0; l < totalPixelImagem; l++)
62  } // contrasteVerdeCore1()
63
64  //============== VERMELHO S/Z
65  void contrasteVermelhoCore0(){
66    Pixel p;
67    int pixel, corPixel, j, r, g, b;
68
69    for(l = 0; l < totalPixelImagem; l++){
70      bcdW_max = bcdWSt();
71      while(bcdW_max <= 1){  // espera ativa
72        bcdW_max = bcdWSt(); }
73
74      pixel = (int) *(img + l); // recupera pixel
75      r = pixel; g = pixel; b = pixel;
76
77      p[RED]   = (unsigned int) ((r>>24) & 0xff);
78      p[GREEN] = (unsigned int) ((g>>16) & 0xff);
79      p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
80
81     corPixel = (int) (p[RED]   + p[RED]   + p[GREEN] +
82                       p[GREEN] + p[GREEN] + p[BLUE]);
83     // escrita dupla no BCD
84     bcdWWr((int) p[RED]);
85     bcdWWr(corPixel);
86    } // for(l = 0; l < totalPixelImagem; l++)
87  } // contrasteVermelhoCore0()
88
89  void contrasteVermelhoCore1(){
90    Pixel p;
91    int corPixel, pixel, r, g, b;
92    unsigned int *mem;
93
94    for(l = 0; l < totalPixelImagem; l++){
95      bcdR_max = bcdRSt();
96      while(bcdR_max <= 1){  // espera ativa
97        bcdR_max = bcdRSt(); }
98
99      // leitura dupla do buffer
100     p[RED]   = bcdRRd();
101     corPixel = bcdRRd();
102
103     corPixel = corPixel / 6;
104     p[RED]   = (p[RED] * corPixel) / 255;
105     p[GREEN] = 0;
106     p[BLUE]  = 0;
107
108     if(p[RED] < 127) p[RED] = 0;
109     // 0xff alfa
110     pixel = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;
111
112     mem = img + l;
113     *mem = (int) pixel;
114   } // for(l = 0; l < totalPixelImagem; l++)
115 } // contrasteVermelhoCore1()
```

```
116
117 //============== CINZA S/Z Core 0
118 void contrasteCinzaCore0(){
119   Pixel p;
120   int pixel, corPixel, j, r, g, b;
121
122   for(l = 0; l < totalPixelImagem; l++){
123     bcdW_max = bcdWSt();
124
125     while(bcdW_max <= 1){  // espera ativa
126       bcdW_max = bcdWSt();}
127
128     pixel = (int) *(img + l); // recupera pixel
129     r = pixel; g = pixel; b = pixel;
130
131     p[RED]   = (unsigned int) ((r>>24) & 0xff);
132     p[GREEN] = (unsigned int) ((g>>16) & 0xff);
133     p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
134
135     corPixel = p[RED] + p[GREEN] + p[BLUE];
136     corPixel = corPixel/3;
137     p[RED]   = corPixel;
138     p[GREEN] = corPixel;
139     p[BLUE]  = corPixel;
140     // 0xff alfa
141     pixel = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;
142
143     bcdWWr(pixel);
144   } // for(l = 0; l < totalPixelImagem; l++)
145 } // contrasteCinzaCore0()
146
147 //============== INATIVO S/Z
148 void contrasteInativoCore0(){
149   int pixel;
150
151   for(l = 0; l < totalPixelImagem; l++){
152     bcdW_max = bcdWSt();
153     while(bcdW_max <= 1){  // espera ativa
154       bcdW_max = bcdWSt(); }
155
156     pixel = (int) *(img + l); // recupera pixel
157     bcdWWr(pixel);
158   } // for(l = 0; l < totalPixelImagem; l++)
159 } // contrasteInativoCore0()
160
161 void contrasteInativoCore1(){
162   unsigned int *mem, pixel;
163
164   for(l = 0; l < totalPixelImagem; l++){
165     bcdR_max = bcdRSt();
166     while(bcdR_max <= 0){  // espera ativa
167       bcdR_max = bcdRSt(); }
168
169     pixel = bcdRRd();
170     mem = img + l;
171     *mem = (int) pixel;
172   } // for(l = 0; l < totalPixelImagem; l++)
173 } // contrasteInativoCore1()
174
```

```
175
176 //============== VERDE C/Z
177 void contrasteZVerdeCore0(){
178   Pixel p;
179   int pixel, corPixel, j, r, g, b;
180
181   linha   = initialYZoomed;
182   coluna  = initialXZoomed;
183   linhaM  = finalYZoomed;
184   colunaM = finalXZoomed;
185   sizeL = larguraImagem;
186
187   for(l = linha; l < linhaM; l++){
188     for(c = coluna; c < colunaM; c++){
189       bcdW_max = bcdWSt();
190       while(bcdW_max <= 1){  // espera ativa
191         bcdW_max = bcdWSt(); }
192
193       pixel = (int) *(img + l); // recupera pixel
194       r = pixel; g = pixel; b = pixel;
195
196       p[RED]   = (unsigned int) ((r>>24) & 0xff);
197       p[GREEN] = (unsigned int) ((g>>16) & 0xff);
198       p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
199
200       corPixel = (int) (p[RED]   + p[RED]   + p[GREEN] +
201                         p[GREEN] + p[GREEN] + p[BLUE]);
202       // escrita dupla no BCD
203       bcdWWr((int) p[GREEN]);
204       bcdWWr(corPixel);
205     } // for(c = coluna; c < colunaM; c++)
206   } // for(l = linha; l < linhaM; l++)
207 } // contrasteZVerdeCore0()
208
209 void contrasteZVerdeCore1(){
210   Pixel p;
211   int corPixel, pixel, r, g, b;
212   unsigned int *mem;
213
214   for(l = 0; l < alturaImagem; l++){
215     for(c = 0; c < larguraImagem; c++){
216       bcdR_max = bcdRSt();
217       while(bcdR_max <= 1){  // espera ativa
218         bcdR_max = bcdRSt(); }
219       // leitura dupla do buffer
220       p[GREEN] = bcdRRd();
221       corPixel = bcdRRd();
222
223       corPixel = corPixel / 6;
224       p[RED]   = 0;
225       p[GREEN] =(p[GREEN] * corPixel) / 255;
226       p[BLUE]  = 0;
227
228       if(p[GREEN] < 127) p[GREEN] = 0;
229       // 0xff alfa
230       pixel = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;
231
232       // Replica para posição original
233       mem = img + l*larguraImagem + c;
```

```
234         *mem = (int) pixel;
235         // Replica para posição original+1
236         mem = img + l*larguraImagem + (c+1);
237         *mem = (int) pixel;
238         // Replica para próxima linha na posição original
239         mem = img + (l+1)*larguraImagem + c;
240         *mem = (int) pixel;
241         // Replica para próxima linha na posição original+1
242         mem = img + (l+1)*larguraImagem + (c+1);
243         *mem = (int) pixel;
244         c++;
245       } // for(c = 0; c < larguraImagem; c++)
246       l++;
247     } // for(l = 0; l < alturaImagem; l++)
248   } // contrasteZVerdeCore1()
249
250   //============== VERMELHO C/Z
251   void contrasteZVermelhoCore0(){
252     Pixel p;
253     int pixel,corPixel, j, r, g, b;
254
255     linha   = initialYZoomed;
256     coluna  = initialXZoomed;
257     linhaM  = finalYZoomed;
258     colunaM = finalXZoomed;
259     sizeL = larguraImagem;
260
261     for(l = linha; l < linhaM; l++){
262       for(c = coluna; c < colunaM; c++){
263         bcdW_max = bcdWSt();
264         while(bcdW_max <= 1){  // espera ativa
265           bcdW_max = bcdWSt(); }
266
267         pixel = (int) *(img + l); // recupera pixel
268         r = pixel; g = pixel; b = pixel;
269
270         p[RED]   = (unsigned int) ((r>>24) & 0xff);
271         p[GREEN] = (unsigned int) ((g>>16) & 0xff);
272         p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
273
274         corPixel = (int) (p[RED]   + p[RED]   + p[GREEN] +
275                           p[GREEN] + p[GREEN] + p[BLUE]);
276
277          bcdWWr((int) p[RED]);
278          bcdWWr(corPixel);
279       } // for(c = coluna; c < colunaM; c++)
280     } // for(l = linha; l < linhaM; l++)
281   } // contrasteZVermelhoCore0()
282
283   void contrasteZVermelhoCore1(){
284     Pixel p;
285     int corPixel, pixel, r, g, b;
286     unsigned int *mem;
287
288     for(l = 0; l < alturaImagem; l++){
289       for(c = 0; c < larguraImagem; c++){
290         bcdR_max = bcdRSt();
291         while(bcdR_max <= 0){  // espera ativa
292           bcdR_max = bcdRSt(); }
```

```
293
294         // leitura dupla do BCD
295         p[RED] = bcdRRd();
296         corPixel = bcdRRd();
297
298         corPixel = corPixel / 6;
299         p[RED]   = (p[RED] * corPixel) / 255;
300         p[GREEN] = 0;
301         p[BLUE]  = 0;
302
303         if(p[RED] < 127) p[RED] = 0;
304         // 0xff alfa
305         pixel = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;
306
307         // Replica para posição original
308         mem = img + l*larguraImagem + c;
309         *mem = (int) pixel;
310         // Replica para posição original+1
311         mem = img + l*larguraImagem + (c+1);
312         *mem = (int) pixel;
313         // Replica para próxima linha na posição original
314         mem = img + (l+1)*larguraImagem + c;
315         *mem = (int) pixel;
316         // Replica para próxima linha na posição original+1
317         mem = img + (l+1)*larguraImagem + (c+1);
318         *mem = (int) pixel;
319         c++;
320       } // for(c = 0; c < larguraImagem; c++)
321       l++;
322     } // for(l = 0; l < alturaImagem; l++)
323 } // contrasteZVermelhoCore1()
324
325 //============== CINZA C/Z
326 void contrasteZCinzaCore0(){
327   Pixel p;
328   int pixel, corPixel, j, r, g, b;
329
330   linha   = initialYZoomed;
331   coluna  = initialXZoomed;
332   linhaM  = finalYZoomed;
333   colunaM = finalXZoomed;
334   sizeL = larguraImagem;
335
336   for(l = linha; l < linhaM; l++){
337     for(c = coluna; c < colunaM; c++){
338       pixel = (int) *(img + l); // recupera pixel
339       r = pixel; g = pixel; b = pixel;
340
341       p[RED]   = (unsigned int) ((r>>24) & 0xff);
342       p[GREEN] = (unsigned int) ((g>>16) & 0xff);
343       p[BLUE]  = (unsigned int) ((b>>8 ) & 0xff);
344
345       corPixel = p[RED] + p[GREEN] + p[BLUE];
346       corPixel = corPixel/3;
347       p[RED]   = corPixel;
348       p[GREEN] = corPixel;
349       p[BLUE]  = corPixel;
350       // 0xff alfa
351       pixel = (p[RED]<<24) | (p[GREEN]<<16) | (p[BLUE]<<8) | 0xff;
```

```
352
353         bcdW_max = bcdWSt();
354         while(bcdW_max <= 0){  // espera ativa
355           bcdW_max = bcdWSt(); }
356       bcdWWr(pixel);
357     } // for(c = coluna; c < colunaM; c++)
358   } // for(l = linha; l < linhaM; l++)
359 } // contrasteZCinzaCore0()
360
361 void contrasteZCore1(){ // Para Inativo e Cinza
362   int pixel;
363   unsigned int *mem;
364
365   for(l = 0; l < alturaImagem; l++){
366     for(c = 0; c < larguraImagem; c++){
367       bcdR_max = bcdRSt();
368       while(bcdR_max <= 0){  // espera ativa
369         bcdR_max = bcdRSt(); }
370
371       // leitura do buffer
372       pixel = bcdRRd();
373
374       // Replica para posição original
375       mem = img + l*larguraImagem + c;
376       *mem = (int) pixel;
377       // Replica para posição original+1
378       mem = img + l*larguraImagem + (c+1);
379       *mem = (int) pixel;
380       // Replica para próxima linha na posição original
381       mem = img + (l+1)*larguraImagem + c;
382       *mem = (int) pixel;
383       // Replica para próxima linha na posição original+1
384       mem = img + (l+1)*larguraImagem + (c+1);
385       *mem = (int) pixel;
386       c++;
387     } // for(c = 0; c < larguraImagem; c++)
388     l++;
389   } // for(l = 0; l < alturaImagem; l++)
390 } // contrasteZCore1()
391
392 //============== INATIVO C/Z
393 void contrasteZInativoCore0(){
394   int pixel;
395
396   linha   = initialYZoomed;
397   coluna  = initialXZoomed;
398   linhaM  = finalYZoomed;
399   colunaM = finalXZoomed;
400   sizeL = larguraImagem;
401
402   for(l = linha; l < linhaM; l++){
403     for(c = coluna; c < colunaM; c++){
404       bcdW_max = bcdWSt();
405       while(bcdW_max <= 0){  // espera ativa
406         bcdW_max = bcdWSt(); }
407
408       pixel = (int) *(img + l); // recupera pixel
409       bcdWWr(pixel);
410     } // for(c = coluna; c < colunaM; c++)
```

```
411     } // for(l = linha; l < linhaM; l++)
412 } // contrasteZInativoCore0()
```

Quadro B.6: Código de contrastes do cLUPAb (cLupaContrasteb.h)

Definições e configurações - Comum entre cLUPA e cLUPAb

```
1   #define HEAPSIZE 512    // Área para variáveis
2   // x_DATA_BASE_ADDR + Heap
3   #define dataImgAddressI x_DATA_BASE_ADDR + HEAPSIZE
4   // alterar para tamanho da imagem a ser tratada
5   #define larguraImagem          640
6   #define alturaImagem           480
7   // alterar para número de frames a executar
8   #define frameStream           3
9   #define totalPixelImagem       larguraImagem*alturaImagem
10  #define larguraZoom            larguraImagem / 2
11  #define alturaZoom             alturaImagem  / 2
12  #define initialXZoomed         larguraZoom   / 2
13  #define finalXZoomed           larguraZoom   / 2  + larguraZoom
14  #define initialYZoomed         alturaZoom    / 2
15  #define finalYZoomed           alturaZoom    / 2  + alturaZoom
16  #define tamanhoImagemZoomed    totalPixelImg / 4
17  // Ampliação
18  #define ZINACTIVE 0
19  #define ZACTIVE   1
20  // Contraste
21  enum contrasteCfg{
22  CINACTIVE, // Contraste desligado  0
23  CVERMELHO, // Contraste vermelho   1
24  CVERDE,    // Contraste verde      2
25  CCINZA   }; // Contraste cinza     3
26  // Definir com a capacidade do buffer de comunicação
27  #define BCDCapacity 8
28  // RGBA
29  #define RED   3 // Componente Vermelho
30  #define GREEN 2 //            Verde
31  #define BLUE  1 //            Azul
32  #define ALPHA 0 //            Alfa
33 #endif
```

Quadro B.7: Código de definições e configurações do cLUPA e cLUPAb (cLupa.h)

EDMAR ANDRÉ BELLORINI

CLUPA: UM AMPLIADOR DIGITAL DE DOCUMENTOS IMPRESSOS SOBRE UMA PLATAFORMA *MULTICORE*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2015