Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E
TECNOLÓGICAS Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Explorando recursos de paralelismo no MPSoC Heterogêneo DM3730

Fernando Fernandes dos Santos

CASCAVEL
2014

Fernando Fernandes dos Santos

Explorando recursos de paralelismo no MPSoC Heterogêneo DM3730

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Marcio Seiji Oyamada

CASCAVEL

2014

Fernando Fernandes dos Santos

Explorando recursos de paralelismo no MPSoC Heterogêneo DM3730

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Marcio Seiji Oyamada (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Adair Santa Catarina
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Alexandre Augusto Giron
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 8 de dezembro de 2014

"...É necessário sempre acreditar que o sonho é possível,
Que o céu é o limite e você truta é imbatível.
Que o tempo ruim vai passar é só uma fase,
E o sofrimento alimenta mais a sua coragem.
Que a sua família precisa de você
Lado a lado se ganhar pra te apoiar se perder.
Falo do amor entre homem, filho e mulher,
A única verdade universal que mantém a fé.
Olhe as crianças que é o futuro e a esperança,
Que ainda não conhecem, não sente o que é ódio e ganância.
Eu vejo o rico que teme perder a fortuna
Enquanto o mano desempregado, viciado se afunda
Falo do enfermo irmão, falo do são, então
Falo da rua que pra esse louco mundão
Que o caminho da cura pode ser a doença
Que o caminho do perdão às vezes é a sentença
Desavença, treta e falsa união.
A ambição como um véu que cega os irmão
Que nem um carro guiado na estrada da vida
Sem farol no deserto das trevas perdida
Eu fui orgia, ego louco, mas hoje ando sóbrio.
Guardo o revólver quando você me fala em ódio
Eu vejo o corpo, a mente, a alma, espírito.
Ouço o refém e o que diz lá no ponto lírico
Falo do cérebro e do coração
Vejo egoísmo preconceito de irmão pra irmão
A vida não é o problema é batalha desafio
Cada obstáculo é uma lição eu anuncio
É isso ai você não pode parar
Esperar o tempo ruim vir te abraçar
Acreditar que sonhar sempre é preciso
É o que mantém os irmãos vivos..."
(Trecho da música: A vida é um desafio – Racionais MC's)

AGRADECIMENTOS

A Deus que permitiu que tudo isso acontecesse, ao longo de minha vida, e não somente nestes anos como universitário, mas em todos os momentos (caro leitor, caso você seja um "*Neo-ateu Geração Toddynho*", os agradecimentos são meus e eu faço para quem eu quiser).

Agradeço a minha mãe Lucia, heroína que me deu apoio, incentivo nas horas difíceis, de desânimo e cansaço.

Ao meu pai Osmar que apesar de todas as dificuldades me fortaleceu e que para mim foi muito importante.

Obrigado meus irmãos, Osmar F. e José, que nos momentos de minha ausência dedicados ao estudo superior, sempre fizeram entender que o futuro é feito a partir da constante dedicação no presente.

Ao meu orientador Marcio, pelo empenho dedicado à elaboração deste trabalho.

Agradeço aos professores por me proporcionarem o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por proporcionarem meios para meu aprendizado A palavra mestre, nunca fará justiça aos professores dedicados os quais sem nominar terão o meu eterno agradecimento.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

MPSoC	*Multiprocessor System on Chip*
SE	Sistemas Embarcados
CPU	*Central Processing Unit*
ROM	*Read Only Memory*
RAM	*Random Access Memory*
MB	*Megabyte*
SoC	*System On Chip*
DSP	*Digital Signal Processor*
GPP	*General Purpose Processor*
SPP	*Specific Purpose Processor*
SIMD	*Single Instruction, multiple Data*
GPU	*Graphics Processing Unit*
DM3730	*Digital Media Processor* 3730
ARM	*Advanced Risc Machine*
SDRC	*SDRAM Controller*
GPMC	*General Purpose Memory Controller*
OTG	*On The Go*
RISC	*Reduced Instruction Set Computer*
CISC	*Complex Instruction Set Computer*
VLIW	*Very Long Instruction Word*
DMA	*Direct Memory Access*
IDMA	*Internal DMA*
EMC	*Eternal Memory Controller*
ULA	Unidade Lógica e Aritmética

PG	*Program address generate*
PS	*Program address send*
PW	*Program access ready wait*
PR	*Program fetch packet receive*
DP	*Instruction dispatch*
DC	*Instruction decode*
RAM	*Random Access Memory*
SRAM	*Static Random Access Memory*
ILP	*Instruction Level Parallelism*
RGB24	*Red, Green and Blue 24 bits format*
YUYV	*YUV Pixel Format*
FPS	*Frames Per Second*

Sumário

Resumo

Sistemas embarcados estão presentes no cotidiano das pessoas, em diversos produtos tais como smartphones, automóveis e aparelhos eletrodomésticos. Um dos principais requisitos dos sistemas embarcados é o desempenho, pois estes executam, em sua maioria tarefas críticas. MPSoCs (*Multiprocessor System on Chip*) heterogêneos têm sido usados para aumentar o desempenho em aplicações embarcadas. Este tipo de MPSoC utiliza a facilidade de processadores de propósito geral (GPP), com a eficiência dos processadores de propósito específico (SPP). Um tipo de processador de propósito específico, utilizado em aplicações multimídia embarcadas é o processador de sinais digitais, os DSPs. Estes são especializados em processamento de sinais digitais como áudio e vídeo entre outros sinais digitais em tempo real, o que torna seu uso importante para diversos fins. Este trabalho relata os esforços para melhorar o desempenho da lupa digital xLupa embarcado, utilizando o paralelismo no nível de tarefas na plataforma heterogênea DM3730, composta pelos processadores ARM Cortex A8 e DSP TMS320C64x (C64x+). Os resultados obtidos mostraram que dependendo do tipo de dado que se deseja processar em um processador DSP, pode tornar seu uso inviável, pois memórias não alinhadas e instruções de controle podem diminuir o desempenho do DSP. Outro ponto observado nesse trabalho foi no que se refere ao uso de paralelismo em nível de tarefas, utilizando-se da abordagem produtor/consumidor. Pode se observar que o uso de vários threads para o estudo de caso do xLupa embarcado não garantiu o aumento de desempenho da aplicação, em relação a uma versão totalmente sequencial.

Palavras-chave: Processadores heterogêneos, Processador DSP, MPSoC, DM3730, BeagleBoard.

Capítulo 1

Introdução

1.1 Motivações

Sistemas embarcados (SE) estão presentes no cotidiano das pessoas, seja fisicamente incorporado a uma função ou com algum objetivo puramente computacional. Estes sistemas podem ser *smarthphones*, controladores de fábricas, aparelhos eletrodomésticos, entre outros, cada sistema embarcado tem um objetivo principal em um determinado ambiente. São dispositivos que estão presentes no dia-a-dia das pessoas, muitas vezes imperceptíveis em um ambiente [1].

Uma característica dos SE, é sua construção, seu desenvolvimento tende a ser mais complexo do que outras aplicações, pois em sua maioria operam em ambientes dinâmicos, os quais não são inteiramente previsíveis. Este tipo de comportamento dá aos SEs uma dificuldade adicional no desenvolvimento, visto que linguagens de programação convencionais nem sempre conseguem descrever todos os cenários possíveis de um ambiente onde um SE operará.

Os SE são usados em situações críticas, onde fatores como confiança, segurança e, o mais importante, desempenho [2], são pontos cruciais para o sucesso da aplicação. Porém chegar a um sistema que atenda estes requisitos nem sempre é fácil, devido à limitação de recursos, principalmente no quesito desempenho.

Técnicas de ILP (*Instruction Level Parallelism*) para o aumento do desempenho, como pipeline e superescalarabilidade são usadas. Um exemplo de paralelismo em nível de instrução é a arquitetura VLIW (*Very long instruction word*), que é uma arquitetura de CPU capaz de executar um grupo de instruções diferentes ao mesmo tempo. A não dependência entre as instruções deve ser garantida pelo compilador para que as instruções possam ser processadas paralelamente, sem a perda da lógica do programa [3].

Porém estas técnicas possuem resultados limitados e dependentes das características da aplicação [3]. Por exemplo, o VLIW pode ser prejudicado por dependências de dados ou instruções de desvio.

Devido aos diversos problemas enfrentados na otimização em aplicações embarcadas, considerando as limitações impostas ao ILP, o uso de processadores de propósito específico associados aos processadores de propósito geral se faz necessário. O uso destas arquiteturas combinadas traz maior flexibilidade no desenvolvimento de aplicações embarcadas, uma vez que as tarefas podem ser divididas entre os processadores. Isto leva a separação de dois tipos de multiprocessadores existentes, MPSoC (*Multiprocessor system on chip*) homogêneo e heterogêneo.

Um multiprocessador heterogêneo é composto, em sua maioria, por processador(es) de propósito específico e processador(es) de propósito geral. Os processadores de propósito específico possuem vantagens na execução de tarefas específicas em relação aos de propósito geral, já que as instruções são implementadas diretamente em hardware ao invés de uso de microprogramas, usados comumente em processadores de propósito geral. A principal vantagem de se usar um MPSoC heterogêneo está na sua integração de diversos sistemas e subsistemas em um único circuito integrado, podendo adaptar-se a diversas aplicações na computação científica e na indústria [4]. Outra vantagem no uso de MPSoC heterogêneo é o uso de paralelismo em nível de tarefas, fazendo com que tarefas executem em paralelo em processadores com propósitos diferentes.

O uso de MPSoC também traz ganhos em relação ao consumo de energia; estes conseguem poupar energia de diferentes maneiras e em todos os níveis de abstração [4]. O programador pode usar diferentes tipos de processadores, podendo executar de forma eficiente as tarefas para quais ele foi projetado. Dispositivos móveis como smartphones fazem uso das vantagens que os MPSoCs trazem na execução de aplicações, por exemplo um processador de sinais digitais executa tarefas de processamento de vídeo ou de áudio, enquanto um processador de propósito geral pode gerenciar um protocolo de rede.

1.2 Metodologia

O MPSoC que será utilizado neste trabalho será o DM3730 [5]da Texas Instruments, que é composto por um processador ARM Cortex A8, e um processador de sinais digitais o DSP TMS320C64x+. Este MPSoC é parte integrante da plataforma de desenvolvimento Beagleboard XM [6] fabricada pela empresa Digi-Key [7] em parceria com a Texas Instruments.

A aplicação utilizada no projeto é o xLupa embarcado [8], desenvolvido em projetos anteriores pelo egresso Diego Hachmann. O xLupa embarcado é uma lupa digital para pessoas com baixa visão; o sistema pode ser conectado a uma TV ou monitor com entrada HDMI, possibilitando a ampliação de documentos impressos.

1.3 Objetivos

Este trabalho tem como objetivo explorar o paralelismo em uma plataforma heterogênea e analisar sua eficiência na implementação de uma nova versão da lupa digital xLupa embarcado.

Na implementação original, o xLupa embarcado possui um algoritmo sequencial, ou seja, a imagem é capturada e tratada de forma sequencial, sem a utilização de paralelismo. Este trabalho tem como objetivo a implementação de um algoritmo, utilizando paralelismo, que faça uso do processador ARM Cortex A8 e do DSP TMS320C64x presentes no DM3730, visando assim melhorias no desempenho no xLupa embarcado. O uso do DSP visa aproveitar melhor os recursos disponibilizados pela plataforma e extrair o paralelismo da aplicação.

1.4 Organização do Texto

No capítulo de introdução foram mostrados as principais motivações para o estudo do uso do paralelismo em nível de instrução, como também a justificativa do uso de MPSoCs heterogêneos. Foram também apresentados os objetivos deste trabalho.

No capítulo dois serão descritos os processadores presentes no DM3730, ARM Cortex A8 e DSP TMS320C64x. O capítulo também mostra um estudo de como é a comunicação entre

os processadores presentes no DM3730.

O capítulo três apresenta um estudo de caso da aplicação xLupa embarcado, neste foram analisados os algoritmos existentes como também as modificações feitas na aplicação. O capítulo mostra também as diferenças de execução da aplicação de forma sequencial e paralela.

No capítulo quatro são mostrados os resultados obtidos com as modificações feitas na aplicação, apresentadas no capítulo três. Para avaliar o desempenho obtido são apresentados os tempos de processamento por algoritmo de tratamento de imagens e também os FPS (*Frames Per Second*) obtidos comparando as diferentes versões desenvolvidas neste trabalho.

No capítulo cinco são apresentadas as conclusões obtidas do trabalho, bem como os trabalhos futuros que poderão ser desenvolvidos.

Capítulo 2

MPSoC Heterogêneo

2.1 MPSoCs em Aplicações Embarcadas

Segundo Marwedel [9] os sistemas embarcados podem ser definidos como sistemas de processamento de informações que são incorporados em um produto maior. Em suma os sistemas embarcados são definidos como sistemas que desenvolvem atividades específicas em um sistema maior. Os SE são usados em equipamentos de telecomunicações, sistemas de transportes, construções inteligentes, em robótica, segurança, em logística, aplicações militares, em eletrônicos de consumo como também em diversas aplicações onde necessite algum tipo de processamento.

A principal diferença dos SE para os computadores de propósito geral e que estes tem um objetivo na sua construção, ou seja, enquanto os computadores pessoais, notebooks e afins são maquinas que atendem uma vasta capacidade de objetivos, os SE são projetados para um específico [10].

Para atuar no meio em que estão inseridos os SE possuem periféricos que fazem o papel de sensores e atuadores. Os sensores são os periféricos onde o SE faz a captura das informações do ambiente onde está inserido, e com base nas informações faz o processamento necessário. Os atuadores são os dispositivos que o SE possui para intervir no ambiente onde está inserido, realizando ações que podem alterar o ambiente onde opera [10].

Os SE possuem algumas características que os diferem dos sistemas computacionais de uso geral, são elas [9]

a) **Segurança:** os SE estão inseridos em diversos contextos como em usinas nucleares, controle de trens e em diversos setores onde a segurança é de extrema importância, sendo assim é extremamente necessário que este tipo de sistema seja seguro. Métricas de segurança como confiabilidade, facilidade de manutenção e disponibilidade devem ser seguidas para que o SE seja considerado seguro;
b) **Eficiência:** sistemas embarcados devem ser eficientes, e também devem seguir algumas métricas para definir sua eficiência. Sendo estas: Consumo de energia eficiente, visto que grande parte deles é alimentado por baterias; Como todo o código da aplicação que irá executar em um sistema embarcado deve ficar no próprio sistema, sem o uso de um dispositivo de armazenamento ou de tamanho reduzido, o tamanho do código deve ser o menor possível; Um SE para ser considerado eficiente também deve ser implementado para consumir o mínimo de recursos. O hardware precisa ser dimensionado para que somente os recursos estritamente necessários para a execução da aplicação estejam presentes em um sistema embarcado; Peso e tamanho também são levados em consideração na eficiência dos SE, estes geralmente exibem requisitos restritos em relação ao peso e tamanho, como exemplo os celulares; Outra métrica de eficiência é o custo, para ter competividade, o sistema embarcado deve ter o menor custo possível, visto que esse é um fator para a sua aceitação no mercado;
c) **Sistemas em tempo real e reativos:** Os SE devem ser sistemas em tempo real e devem reagir por estímulos do controlador dentro de um determinado intervalo de tempo, caso isso não seja verdade em certas aplicações o resultado pode ser uma catástrofe;
d) **Dedicado a certa aplicação e interface dedicada:** Os SE são em sua maioria construídos para resolver os problemas de certo contexto, ou seja, são dedicados. Estes sistemas usualmente não possuem interfaces de entradas como teclado e mouse, a entrada de dados é feita por botões, volantes, pedais entre outros dispositivos de entrada dedicados ao sistema.

Devido a esse vasto campo de utilização dos SE o uso de SoCs (*System on chip*) se faz necessário para prover a integração das funções destes dispositivos. SoCs permitem que diversos componentes de processamento sejam colocados em um circuito integrado, devido a sua integração de circuitos em larga escala. Podendo associar sistemas como DSP,

processadores de propósito geral, decodificadores de áudio e vídeo e outros elementos processadores, em um único chip [11].

Algumas características dos SoCs fazem com que exista vantagens em seu uso. Maior velocidade devido à integração em um único chip, significativa redução de potência consumida e tamanho reduzido, pois não são necessários componentes adicionais. Ao invés de trilhas de barramento para conexão dos componentes processadores, a conexão é interna ao chip, aumentando assim a confiabilidade do sistema. A alta capacidade de integração permite que haja uma redução nos custos dos sistemas em um único chip [12].

Devido a necessidade de se integrar diversos elementos processadores com diversas funcionalidades foi necessário a criação de um tipo de SoC, o MPSoC (*Multiprocessor system on chip*) [13]. Um MPSoC é um sistema multiprocessador integrado em um único circuito, este incorpora todos ou quase todos os componentes necessários para uma aplicação, que utilize múltiplos processadores programáveis [4]. MPSoCs vêm sendo usados em grande quantidade em redes, comunicação, processamento de sinais e diversas aplicações multimídia.

MPSoCs podem fazer uso de uma rede complexa interna ao chip ou de barramentos de interconexão, para integrar diversos processadores programáveis, núcleos, memórias especializadas e componentes próprios em um único chip [14].

Devido as demandas de mercado por tecnologias embarcadas com características que cada vez mais necessitam de flexibilidade, baixo custo, baixo consumo e desempenho razoável, é necessário fazer uso dos MPSoCs para se conseguir atender as expectativas de consumidores [15].

2.2 Processadores heterogêneos GPPs x SPPs

Os processadores atuais podem ser categorizados em dois grupos diferentes: os de propósito geral GPPs (*General Purpose Processors*) ou os de propósito específico SPPs (*Specific Purpose Processor*) [16], que possuem características diferentes usadas para diferentes finalidades.

Os GPPs são processadores que não estão amarrados em nenhuma aplicação ou linguagem específica, ou seja, são destinados ao propósito geral, como computadores pessoais ou servidores de rede. Os processadores ARM Cortex, Intel Core i7, AMD Phenom são exemplos

de GPPs.

Entretanto os GPPs para determinadas tarefas como decodificação de sinais de áudio e vídeo, por exemplo, são menos eficientes que os SPPs construídos para este propósito. Os SPPs são processadores projetados para tarefas específicas, possuindo eficiência maior do que os GPPs nas tarefas para as quais foi projetado. SPPs são comumente usados em sistemas embarcados e em unidades de processamento gráfico, os processadores C64x DSP, GK110B e Vesuvius XT são exemplos de processadores de propósito específico.

Para atender os diversos requisitos das aplicações, as arquiteturas MPSoCs têm sido construídas fazendo uso somente de GPPs ou um conjunto formado por GPPs ou SPPs, e conforme apresentado na introdução isto acarreta na existência de dois tipos de MPSoCs, os homogêneos e os heterogêneos.

Em um MPSoC heterogêneo os elementos de processamento são distintos, em geral dedicados a tarefas específicas [4]. De acordo com HE *et. al* [15] um MPSoC heterogêneo é um tipo de MPSoC composto por processadores que podem ser GPP ou SPP, em um mesmo chip, estes podem ser configurados de forma a obterem o melhor desempenho de uma aplicação específica.

A categoria de MPSoC heterogêneo é destinada para aplicações heterogêneas onde a solução necessita que diferentes algoritmos para diferentes domínios executem, com elementos de processamento diferentes é possível obter vantagem desta diferença na execução destes algoritmos. A exemplo pode-se usar os dispositivos móveis, como celulares e smartphones que possuem diversas funcionalidades com diferentes características. Estes podem ao mesmo tempo ter processadores específicos para decodificar sequencias de áudio e vídeo enquanto outros processadores de propósito geral são responsáveis pela execução de aplicações de controle, como um editor de texto.

Além dos MPSoCs ditos heterogêneos, existe também a classe dos MPSoCs homogêneos, que segundo Wolf [4] possuem elementos de processamento idênticos, sendo este GPP ou SPP, a aplicação que utiliza-se de um MPSoC homogêneo pode ter um algoritmo executando em qualquer dos elementos de processamento. Segundo He [15] os processadores homogêneos possuem um suporte a uma grande faixa de aplicações e uma grande flexibilidade no uso, porém devido a estas características há uma queda em sua eficiência. Em geral estes são usados para aplicações que fazem uso intensivo de paralelismo nas operações sobre os dados, como por exemplo, em estações de rede wireless, onde é aplicado um mesmo algoritmo em vários fluxos de dados.

Para diferentes tipos de aplicações os processadores GPPs oferecem uma facilidade em seu uso muito grande, e em muitos casos possuem também a possibilidade da exploração do paralelismo de forma dinâmica ou em tempo de compilação, deixando o programador livre de tais preocupações. Porém tais processadores não possuem tanta eficiência quanto os SPPs em aplicações específicas, devido ao fato dos processadores de propósito específico possuírem grande parte das instruções necessárias para o domínio da aplicação implementadas diretamente em hardware.

Exemplos do uso de SPPs são as GPUs e os DSPs. As GPUs (*Graphics Processing Unit*) são processadores gráficos que possuem funções de iluminação, aplicação de texturas, transformações em vértices entre outras transformações gráficas incorporadas em hardware. Processadores DSP são processadores de sinais digitais que conseguem extrair um alto grau de paralelismo com o uso de instruções SIMD e funções implementadas em hardware como soma com saturação, média, cálculo do valor absoluto entre outras.

2.3 MPSoC DM3730

O MPSoC utilizado neste trabalho foi o Digital Media Processor DM3730, que é fabricado pela Texas Instruments [5], este é parte da plataforma de desenvolvimento Beagleboard XM [6] que será abordada posteriormente.

O DM3730 possui um processador de vídeo POWER SGX™ *Graphics Accelerator*, um processador de sinais digitais TMS320C64x+™ DSP e um processador de propósito geral o ARM® Cortex™-A8 (figura 2.1). O DM3730 possui um controlador de memória externa (SDRC) e um controlador de memória de propósito geral (GPMC) com suporte a memória flash.

O acesso direto a memória é feito pelo SDMA (*System Direct Memory Access*) que possui 32 canais lógicos, com possibilidade de configuração de prioridade. No quesito comunicação serial, o DM3730 possui um controlador de USB 2.0 OTG.

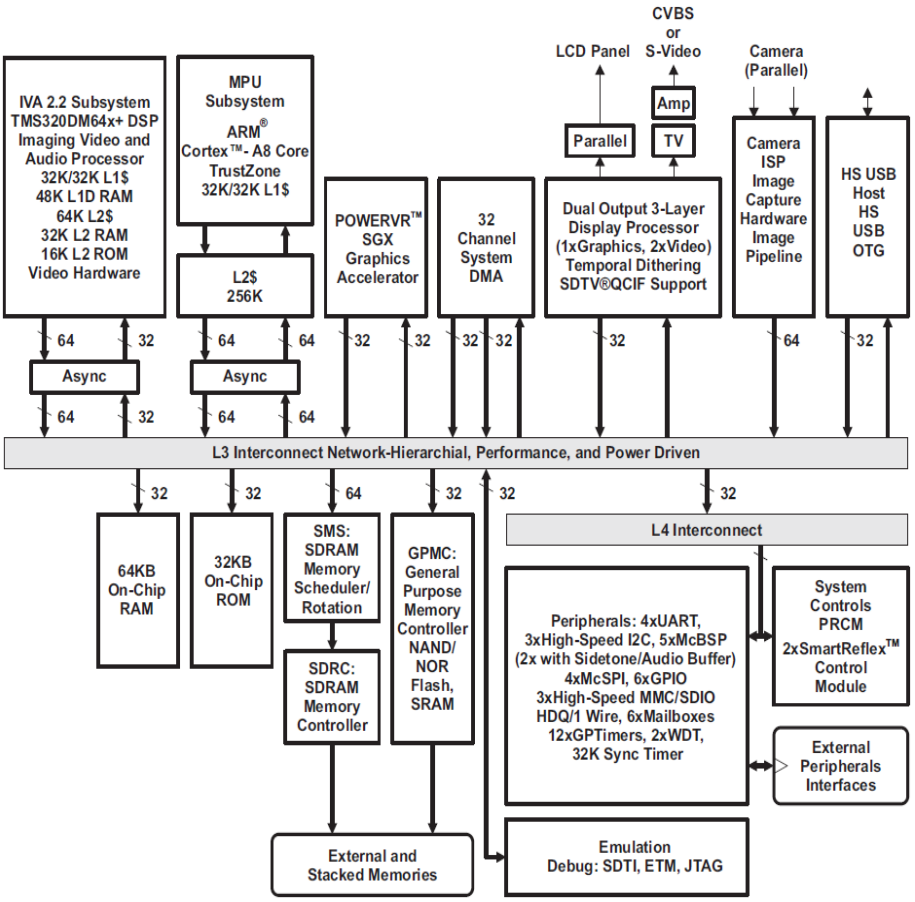

Figura 2.1: Estrutura em blocos do MPSoC DM3730 **[5]**

2.3.1 Processador de propósito geral ARM

ARM (*Advanced Risc Machine*) é uma arquitetura de processador comumente usada em sistemas embarcados. Seu uso se dá em diversas áreas da informática, pois o desenvolvimento da arquitetura visa o melhor desempenho possível, com grande simplicidade, tamanho pequeno e pouco gasto de energia.

Os processadores baseados na arquitetura ARM são conhecidos por serem simples, pois possuem um conjunto limitado de instruções. Estes processadores são encontrados em celulares, calculadoras, periféricos de computador, aplicações industriais e dispositivos de processamento digital [17].

A empresa ARM Holdings é a empresa responsável pelo projeto e licenciamento dos processadores com arquitetura ARM. A empresa foi fundada em 1990, e garante que mais de vinte bilhões de chips baseados na arquitetura ARM foram fabricados até a data do acesso ao

site oficial [18]. A empresa não fabrica os chips, ela somente possui a propriedade intelectual dos chips que projeta, segundo a ARM foram licenciados 800 processadores e vendidos para mais de 250 companhias, desde a fundação da empresa [18]. A empresa licencia a propriedade intelectual para diversos parceiros como Atmel, Broadcom, Dust Networks, Toshiba, Texas Instruments, Samsung, Freescale, Fujitsu entre outras [19].

2.3.2 RISC e CISC

A arquitetura ARM foi desenvolvida inicialmente com o conceito de arquitetura RISC (*Reduced Instruction Set Computer*), porém devido às particularidades dos processadores ARMs atuais, já não é possível enquadrar totalmente os processadores ARM como processador de conjunto de instruções reduzidas.

Na década de 70 devido às diferenças de velocidade entre memória e processador, compiladores pobres e pouco robustos foi necessária a criação de uma nova filosofia de desenvolvimento de arquiteturas [20], a arquitetura CISC, (*Complex Instruction Set Computer*) surgiu com o objetivo de solucionar problemas da época, como as diferenças de velocidades entre memória e processador. A arquitetura CISC possui algumas características como o uso de instruções complexas e de microprogramas.

Porém o uso de instruções complexas torna a tarefa de reorganizar o código, para permitir um uso do pipeline com mais eficiência, muito mais complexa, acarretando em perda de desempenho. Outra característica que torna o CISC menos eficiente, é segundo STALLINGS, que nem sempre o uso de instruções complexas resultará em código menor em relação ao uso de instruções reduzidas [17]. Além de que uma característica do CISC que acarreta em perda de desempenho, como existem mais instruções, *opcodes* maiores são necessários, acarretando instruções maiores, fazendo com que uma única instrução leve vários ciclos de relógio para sua completa execução.

Além disso, o aumento no tamanho dos microprogramas devido ao grande e crescente número de instruções, problemas de queda de desempenho e dificuldade na detecção e correção de erros em microcódigos fizeram os pesquisadores questionarem se o uso de instruções mais complexas seria bom ou ruim.

Com o passar do tempo os problemas que levaram a criação da arquitetura CISC não eram tão evidentes como anteriormente, levando ao questionamento dos pesquisadores, seria

necessário o uso de instruções complexas para tarefas simples? Em resposta a essa pergunta, na década de 80, uma nova filosofia de arquiteturas passou a existir, a arquitetura RISC. A arquitetura RISC se baseia no fato de que muitas operações podem ser feitas sem o uso de instruções complexas como na arquitetura CISC [20]. A arquitetura RISC possui as seguintes características [21]:

a) ***Menor quantidade de instruções e tamanho fixo:*** *A principal característica da arquitetura RISC é a quantidade reduzida de instruções. O tamanho destas instruções em bits é sempre o mesmo, facilitando sua busca;*
b) ***Execução otimizada de chamada de funções:*** *Nas máquinas CISC as chamadas de funções são feitas com operações de escrita e de leitura na memória, para manipulação de dados e passagem de parâmetros. Na arquitetura RISC este processo é feito diretamente no processador, utilizando-se de mais registradores do que as máquinas CISC, porém com um desempenho melhor;*
c) ***Menor quantidade de modos de endereçamento:*** *Enquanto a arquitetura CISC possui vários modos de endereçamento, as máquinas RISC no geral possuem somente dois modos LOAD e STORE;*
d) ***Modo de execução com Pipelining:*** *Devido à simplicidade das instruções o uso de pipeline nas maquinas RISC se torna mais produtivo.*

Como já dito, os processadores ARM possuem características da arquitetura RISC, porém algumas diferenças fazem com que os processadores ARM não possa ser enquadrado totalmente na filosofia RISC, as principais características são elas [22]:

a) ***Certas instruções têm ciclos variados de execução:*** *Não são todas as instruções que levam um ciclo para ser executada, isso depende da quantidade de registradores que estão sendo transferidos;*
b) ***Operações de deslocamento:*** *Operandos sofrem operações de deslocamento antes de chegarem a ULA, com o objetivo de melhorar o desempenho (diminuir o tamanho do código);*
c) ***Conjunto de instruções Thumb de 16 bits:*** *Um conjunto de instruções chamado de Thumb adicionado aos processadores ARM faz com que ele possa executar instruções de 16 bits ou 32 bits;*
d) ***Execução Condicional:*** *Uma instrução condicional é executada quando uma específica condição é satisfeita. Esta característica melhora o desempenho e a densidade do código, reduzindo a quantidade de linhas de execução;*

e) ***Instruções otimizadas complexas:*** *Instruções para sinais digitais foram adicionadas ao conjunto de instruções ARM para suportar operações como multiplicação 16 x 16 bits com saturação. Estas instruções permitem melhor desempenho no processador com dados multimídia;*

f) ***Multiplos loads e stores:*** *O modelo load/store para processamento de dados na arquitetura ARM, executa sobre registradores e não diretamente em memória. Esta diferença permite que o uso de instruções STM (Store multiple registers) e LDM (Load multiple registers) sobre múltiplos registradores seja possível.*

2.3.3 ARM Cortex A8

O processador ARM Cortex A8 (figura 2.2), é um processador baseado na arquitetura ARMv7-A Cortex com 2.0 DMIPS/MHZ no benchmark *Dhrystone Performance*. Sua frequência pode variar de 600MHz a 1GHz. Possui uma memória cache de níveis L1 e L2 de tamanhos variáveis, estes podem variar de acordo com o tamanho escolhido pelo fabricante, porém a especificação do processador traz tamanhos mínimos e máximos para cada nível. O nível L1 pode ser de 16 Kbytes ou 32 Kbytes de memória, e para o nível L2, de 0 Kbytes a 1 Mbyte.

O ARM Cortex A8, também possui um coprocessador NEON que opera sobre dados de 128 bits. Este é o processador de propósito geral utilizado no MPSoC DM 3730.

Pipeline do ARM Cortex A8

O processador ARM Cortex A8 é um bom exemplo de processador com características da arquitetura RISC, e também superescalar [17]. O A8 é um superescalar com despacho duplo, com escalonador estático e também detecção dinâmica de despacho, permitindo, por *clock*, a emissão de uma ou duas instruções [3].

O pipeline do Cortex A8 possui 14 estágios (figura 2.3) sendo estes divididos em [23]:

a) ***Instruction Fetch*****:** Busca da instrução, F0 ao F2, três ciclos, sendo o F0 o estágio em que são calculados os endereços quais serão buscados;

b) ***Instruction Decode*****:** Decodificação da instrução, D0 ao D4, cinco ciclos;

c) ***Instruction Execute*****:** Execução da instrução, E0 ao E5, seis ciclos.

O coprocessador NEON possui capacidade de executar instruções sobre dados de 128 bits,

possibilitando o aumento de desempenho em aplicações multimídia entre outros. O NEON possui seu próprio pipeline, que possui 10 estágios (figura 2.3), dividido em [23]:

a) ***Instruction Decode*:** Decodificação da instrução NEON, M0 ao M2, três ciclos;

b) ***Neon Register File*:** Transferência para arquivo de registro, M3, um ciclo;

c) ***NEON Register Writeback*:** Escrita de volta no arquivo de registro, N1 ao N6, seis ciclos.

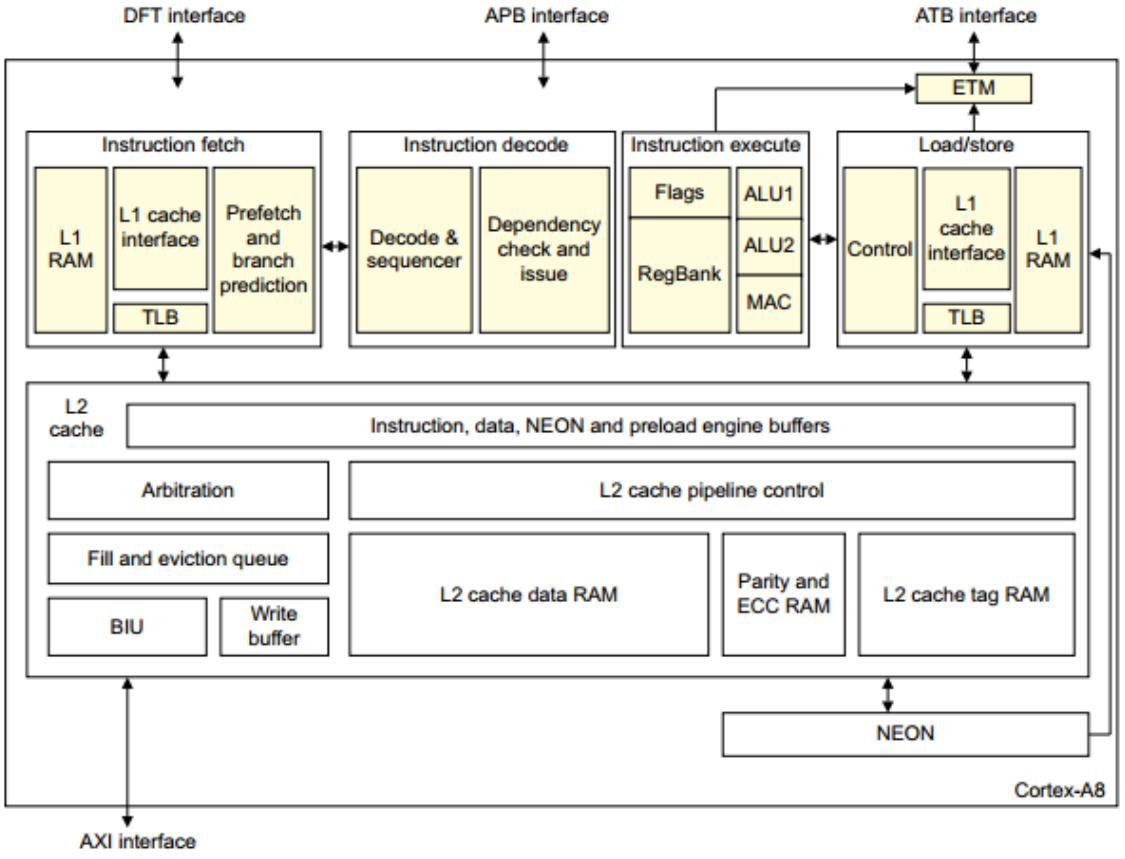

Figura 2.2: Estrutura em blocos do processador ARM Cortex A8 **[23]**

2.3.4 Processador de propósito específico DSP

Um sinal digital se refere a um sinal que é convertido em um padrão de bits. Em contrapartida do sinal analógico que é contínuo e contêm quantidades variadas no tempo, os sinais digitais têm valores discretos em cada ponto da amostragem [24]. Imagens, vídeo e áudio são exemplos de sinais digitais.

Dados em formatos digitais são utilizados intensamente no mundo atual, são dados que necessitam serem processados de uma forma rápida para atender as necessidades dos usuários.

Inserido nesta demanda por processamento estão os processadores de sinais digitais, os DSPs.

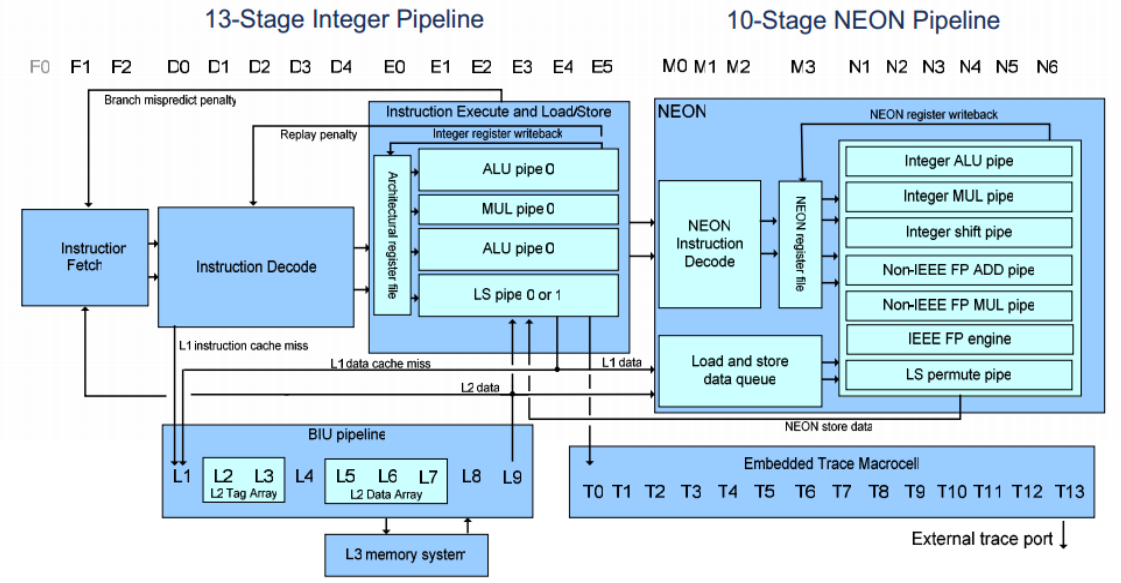

Figura 2.3: Pipeline ARM Cortex A8 **[25]**

2.3.5 Processador de sinais digitais TMS320C64x+

O processador de sinais digitais TMS320C64x+ [26], fabricado pela Texas Instruments™, pertence a plataforma C6000™ DSP da empresa. O processador é da família de processadores DSP TMS320, possui suporte para processamento de ponto fixo, com desempenho de 8000 milhões de instruções por segundo (MIPS).

O processador TMS320C64x+ (figura 2.4), também chamado de C64x+, tem como característica a existência de 64 registradores de propósito geral de 32 bits e oito unidades funcionais, sendo que destas unidades funcionais, são dois multiplicadores e seis ULAs (Unidade Lógica e Aritmética). Destes multiplicadores cada um consegue realizar duas multiplicações de 16 x 16 bits, ou quatro multiplicações 8 x 8 bits por ciclo de *clock*. Além destas características o C64x+ também possui DMA interna para transferência de dados na memória interna (IDMA), e um controlador de memória externa (EMC).

ILP no C64x+

O paralelismo em nível de instrução, ILP, pode ser obtido de duas formas, por meio de

escalonamento estático ou dinâmico. A principal diferença entre as duas formas está em como o paralelismo é extraído, no escalonamento dinâmico o paralelismo é extraído em tempo de execução pelo processador, ou seja, é capacidade do escalonador do processador tomar as decisões em tempo de execução. No estático o paralelismo é extraído em tempo de compilação, ou seja, escalonando instruções dos blocos básicos do código.

Um exemplo de arquitetura que explora o paralelismo de forma estática é a VLIW (*very long instruction word*), onde uma instrução longa abriga um conjunto pré-definido de instruções básicas sem dependências. A principal vantagem deste tipo de arquitetura é eliminar mecanismos complicados para analisar e resolver as dependências entre instruções do processador, e realizar tal análise em tempo de compilação.

Os processadores da plataforma C6000 DSP possuem a arquitetura de alto desempenho Velociti™, esta é uma arquitetura avançada para VLIW (*Very Long Instruction Word*) [26]. No geral os processadores da plataforma C6000 conseguem executar até oito instruções de 32 bits por ciclo, provendo recursos como [26]:

a) CPU VLIW com oito unidades funcionais, incluindo seis unidades aritméticas;

b) Empacotamento de instrução reduz a densidade do código, fetches de programas e consumo de energia;

c) Suporte a dados de 8, 16 e 32 bits, e também opção de precisão extra, até 40 bits, para aplicações de computação intensiva;

d) Opções de saturação e normalização, provendo suporte a diversas operações aritméticas;

e) Suporte a acesso de palavras não alinhadas de 32 e 64 bits;

Todas as unidades funcionais (.L, .M, .S, .D, figura 2.4) do C64x+ conseguem fazer as operações aritméticas de soma e subtração, devido a este fato é possível executar em uma instrução VLIW oito operações deste tipo em paralelo. Entretanto, operações como *load*, *shift* e multiplicação não são possíveis de serem feitas em todas as unidades funcionais, impondo assim uma limitação às operações que podem ser executadas em paralelo.

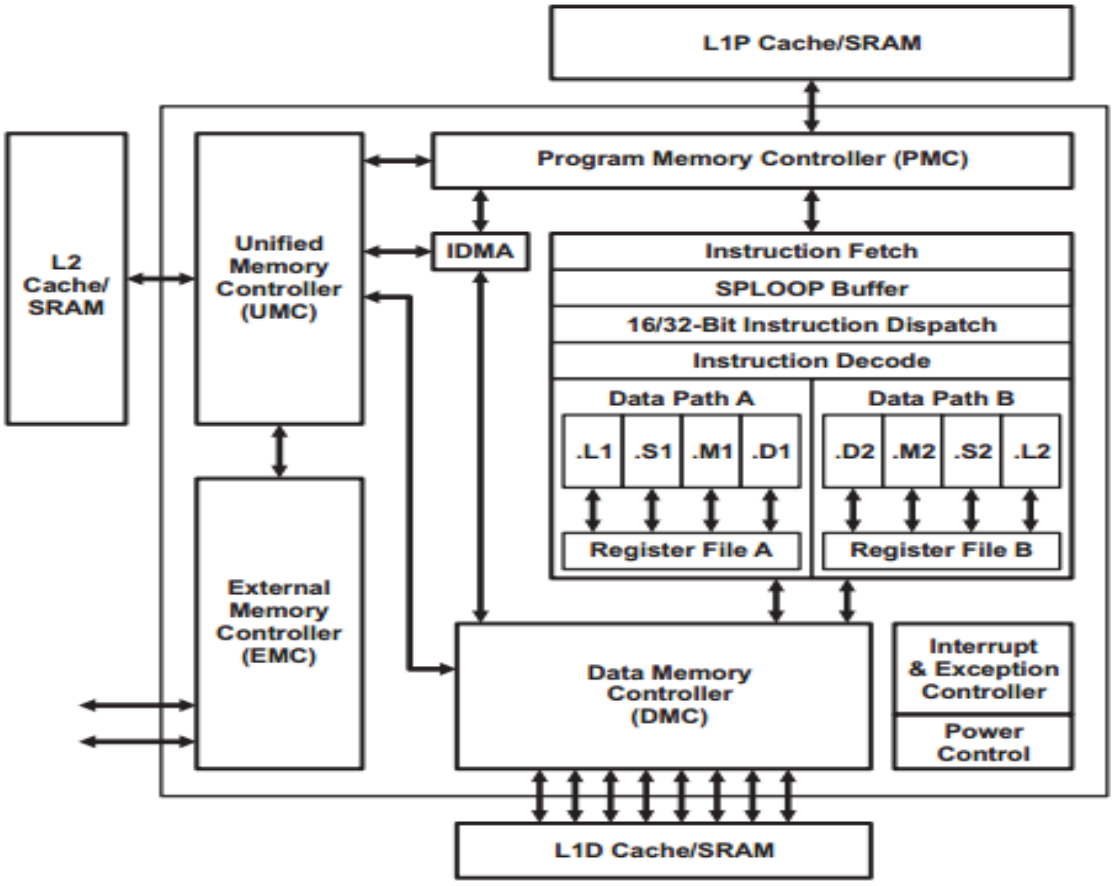

Figura 2.4: Estrutura em blocos do processador TMS320C64x+ DSP **[26]**

Pipeline

O pipeline do C64x+ possui três estágios: busca (*Fetch*), decodificação (*Decode*) e execução (*Execute*) (figura 2.5) [26]. O estágio *Fetch* possui quatro ciclos, sendo eles divididos por fases PG (*Program address generate*), PS *(Program address send*), PW (*Program access ready wait*) e PR (*Program fetch packet receive*). Sendo que na fase de PG o endereço do programa é gerado na CPU. Na fase de PS o endereço do programa é enviado à memória. Na fase de PW ocorre a leitura da memória. E na última fase, PR, a busca é recebida pela CPU.

O estágio *Decode* possui duas etapas. DP (*Instruction dispatch*) e DC (*Instruction decode*). Na fase de DP, os pacotes de instruções da busca são divididos em pacotes executáveis, ainda na fase de DP as instruções dos pacotes executáveis são designadas as unidades funcionais apropriadas. Na fase de DC, registradores são decodificados para a execução de instruções nas unidades funcionais.

O último estágio, *Execute*, possui cinco fases, porém nem todas as instruções chegam a usar todas as etapas, sendo que diferentes tipos de instruções necessitam de quantidades de passos diferentes para completar sua execução (figura 2.5).

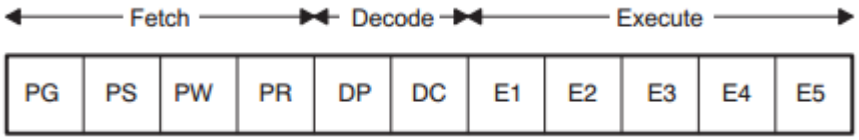

Figura 2.5: Estágios do pipeline do C64x+ **[26]**

Uma instrução no C64+ pode consumir de 7 a 11 ciclos no pipeline, porém em alguns casos pode ocorrer que um operando não seja encontrado na cache do processador, sendo necessário fazer protelação (*stall)* da instrução no pipeline. Caso o operando não seja encontrado em *cache* ele deve ser buscado na memória principal ocasionando bolha de pipeline.

Mesmo que na VLIW as dependências e *hazards*[1] sejam determinados pelo compilador, a falta do operando em memória cache também afeta a execução da VLIW, pois eventos não programados podem causar *stall* no processador de forma geral.

Memória Cache no C64x+

É um fato que a velocidade dos processadores aumentou durante um período de tempo, mais do que as velocidades das memórias, isso trouxe um problema para os projetistas, com velocidades diferentes mais ciclos de pipeline seriam gastos nas buscas de instruções e dados em memória. O problema a ser resolvido era, "como compatibilizar as velocidades da memória e dos processadores, de modo que *stalls* não aconteçam no pipeline?".

Uma solução plausível seria o aumento do número dos registradores em um processador, pois assim mais instruções e dados poderiam ser carregados e haveria uma diminuição nos *stalls* do pipeline. Porém devido ao alto custo monetário para inserção de mais registradores em um processador, a solução é inviável, devido o fato de que o registrador está no topo da hierarquia de memória. A melhor solução encontrada para este problema é o uso de memória cache, um tipo de memória mais barata que registradores, porém mais rápida do que as

[1] *Hazards* de pipeline acontecem quando não se é possível executar a próxima instrução em um próximo ciclo de clock, ou seja, são situações em que o pipeline precisa parar devido a alguma condição que não permite que este continue. *Hazards* podem ser de recurso, dados ou de controle [17].

memórias RAM [17]. A memória cache, é uma memória intermediária entre o processador e a memória principal, o ganho de eficiência do uso da memória cache se dá pelo principio de localidade de referência, que pode ser temporal ou espacial. A localidade de referência temporal diz que o acesso a informações tende a ser a mesma em um curto intervalo de tempo. A localidade de referência espacial diz que as informações próximas têm grande possibilidade de serem acessadas em um futuro próximo [17].

O processador DSP C64x+ possui dois níveis de cache, L1 e L2. O nível L1 é dividido em nível de cache de programa (L1P) e o nível de cache de dados (L1D), os dois com 16 Kbytes [27]. O acesso ao primeiro nível de cache pelo processador é feito sem *stalls*. O nível L2 de cache possui memória endereçável L2 SRAM, como também uma memória interna ao chip para acesso direto por parte do programa. O tamanho da cache no segundo nível é de 1 Mbyte (figura 2.6)

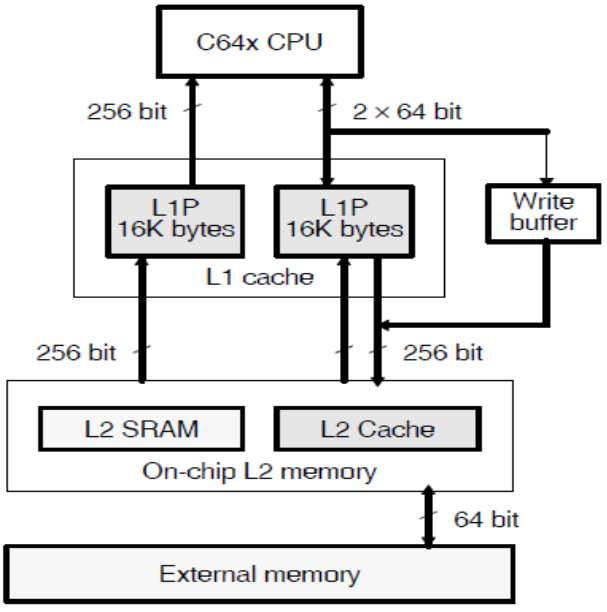

Figura 2.6: Estrutura da cache do C64x+ **[27]**

Programação usando funções intrínsecas no C64x+

É possível fazer uso de funções intrínsecas disponíveis nos compiladores para a família de processadores C6000, da Texas Instruments [28]. Funções intrínsecas são funções disponíveis para o uso em uma linguagem, que por ser usados constantemente, estão por definição incluídos no compilador. Em geral estas funções substituem uma chamada por uma sequencia

gerada automaticamente de instruções *assembler*. A tabela 2.1 mostra exemplos de funções intrínsecas para o compilador da Texas.

O uso das funções intrínsecas no compilador da Texas Instruments permite que se faça uso das instruções SIMD. Estas instruções executam sobre um conjunto de dados de 64 ou 32 bits, *double* ou *int*, respectivamente. Cada função intrínseca disponível pode operar em um ou dois pacotes de dados. Um exemplo é a função _saddu4 (tabela 2.1 [28]). Esta instrução faz a soma de quatro inteiros de 8 bits de uma vez, com saturação. A operação de soma de dois conjuntos de dados $\{8, 5, 45, 20\} e \{2, 10, 5, 2\}$ resultaria em um terceiro conjunto de dados {10, 15, 50, 22}.

Tabela 2.1: Exemplos de funções intrínsecas

Instrução	**Uso**
unsigned _saddu4 (unsigned src1, unsigned src2);	Faz a adição saturada entre pares de valores de 8 bits em src1 e src2.
int _abs2 (int src);	Calcula o valor absoluto para cada valor de 16 bits.
int _cmpgt2 (int src1, int src2);	Compara cada par de valores de 16 bits. Os resultados são empacotados dentro dos dois bits menos significantes do valor retornado.
int _dpint (double src);	Converte um *double* de 64 bits em um inteiro de 32 bits, usando o método de arredondamento.

Otimização de código para C64x+

Com objetivo de otimizar um código nos processadores DSP, diminuindo o tempo de processamento das operações, é possível usar instruções SIMD e/ou operações que façam uso da arquitetura VLIW. Para isso duas etapas devem ser feitas sobre o código que se deseja otimizar.

A primeira delas é a etapa de escrita do código para processadores de propósito geral, ou seja, o código é escrito sem nenhuma informação adicional ao compilador para aumentar o desempenho de execução. A figura 2.7 mostra um código de soma entre dois vetores escrito na linguagem C. São passados dois ponteiros de *char* não sinalizados (a e b) e um inteiro *n*.

Na função é feita a soma através do laço de repetição na linha 3.

```
1 void soma(unsigned char *a, unsigned char * b, int n){
2         int i;
3         for(i = 0; i < n; i++)
4            b[i] = a[i] + b[i];
5 }
```

Figura 2.7: Função para soma de dois vetores de tamanho n

Em segundo momento, para a otimização é necessário passar o máximo de informação possível para o compilador, informações como quantidade de iterações do laço, se vetores estão alinhados na memória, ou se certa posição de memória pode ser endereçada por mais de um ponteiro, assim é possível que o compilador faça otimizações necessárias. Porém nem sempre é possível que o compilador consiga fazer o máximo de otimização, sendo assim é necessário fazer uso de funções intrínsecas como as mostradas na tabela 2.1.

É possível no compilador disponibilizado pela Texas para o DSP C64x+ ativar um *feedback* sobre quais otimizações o compilador conseguiu realizar nas diferentes partes do código. Uma análise com mais detalhes sobre otimização obtida pelo compilador do DSP C64x+, foi feita por Hachmann [8].

Para avaliar o desempenho de diferentes otimizações no DSP C64x+ foram utilizados três códigos de soma entre dois vetores. Foram obtidos os tempos de execução considerando apenas a função de soma, sendo que os tempos de inicialização dos vetores não foram computados. Os tempos obtidos (tabela 2.2) são resultados da média de tempo de 100 execuções das funções de soma de vetores. O tamanho dos vetores somados é de 2764800, este tamanho foi escolhido como forma de anteceder o estudo de caso do capítulo 3 que faz cálculos sobre vetores com este tamanho.

O primeiro algoritmo foi o de soma padrão entre dois vetores (figura 2.7), sem nenhuma otimização adicional.

O segundo algoritmo é o soma com informação (figura 2.8), foram adicionados às informações de *const*, *restric* e *MUST_ITERATE*. A informação de *const* no vetor a informa ao compilador que o vetor *a* não será modificado na função. A palavra chave *restric* informa ao compilador que somente aquele ponteiro irá acessar suas posições de memória. A informação *MUST_ITERATE*, informa ao compilador qual é o mínimo de iterações que ele irá executar, o máximo, e também um múltiplo de iterações possíveis para o *laço*.

O terceiro algoritmo de soma é o soma intrínseca (figura 2.9), que além das informações adicionadas no segundo algoritmo, também adiciona as funções intrínsecas ao *laço*. A cada iteração do laço de repetição 8 bytes de dados são levados para os registradores, então são divididos em pacotes de 4 bytes. Logo após são somados e gravados na memória.

Tabela 2.2: Tempos obtidos nos algoritmos de soma

Algoritmo	Tempo em milissegundos
Soma padrão ARM	22.89 ms
Soma padrão DSP	53.38 ms
Soma com informação DSP	58.43 ms
Soma intrínseca DSP	10.79 ms

```
1 void soma_info(const unsigned char *restrict a, unsigned char *restrict b, const int n) {
2     int i;
3 #pragma MUST_ITERATE(921600, 2764800, 8)
4     for (i = 0; i < n; i++) {
5         b[i] = a[i] + b[i];
6     }
7 }
```

Figura 2.8 Função para soma com informação de dois vetores de tamanho n

```
1  void soma_intri(const unsigned char *restrict a, unsigned char *restrict b, const int n) {
2      unsigned int a1_a0, a3_a2, b1_b0, b3_b2;
3      unsigned int c1_c0, c3_c2, i;
4  #pragma MUST_ITERATE (921600, , 8)
5      for (i = 0; i < n / 8; i += 8) {
6          a3_a2 = _hi(_amemd8_const(&a[i])); //4 bytes mais altos de 8
7          a1_a0 = _lo(_amemd8_const(&a[i])); //4 bytes mais baxos de 8
8          b3_b2 = _hi(_amemd8_const(&b[i]));
9          b1_b0 = _lo(_amemd8_const(&b[i]));
10         c3_c2 = _saddu4(a3_a2, b3_b2); // soma 4 bytes sem sinal
11         c1_c0 = _saddu4(a1_a0, b1_b0);
12         _amemd8(&b[i]) = _itod(c3_c2, c1_c0); //empacote dois inteiros sem sinal
13         //em um double
14     }
15 }
```

Figura 2.9: Função para soma intrínseca de dois vetores de tamanho n

A tabela 2.2 mostra que houve uma perda de desempenho na execução do algoritmo de soma com informação no DSP em relação a soma padrão. Analisando-se o relatório do compilador sobre a otimização obtida dos dois algoritmos, observou-se que o algoritmo de soma com informação foi escalonado pelo compilador para que este extraísse maior

paralelismo do processador C64x+. Entretanto, ao passo que o compilador aumenta o nível de paralelismo em nível de instrução, este também informa que a proporção de falhas de cache pode aumentar, acarretando em perda de desempenho. Analisando os tempos obtidos pela soma com funções intrínsecas e soma padrão ARM, é possível observar que o uso das funções intrínsecas, em algoritmos com alto grau de paralelismo como a soma de vetores, traz um ganho de desempenho considerável.

2.4 Comunicação ARM-DSP

Para comunicação entre o processador ARM e o DSP é necessária uma infraestrutura de comunicação entre os núcleos. No Linux, a comunicação entre processadores pode ser realizada através do driver DSP Bridge [29] ou da biblioteca C6Run [30]. DSP Bridge provê funções de controle e comunicação entre os processadores habilitando processamento paralelo assimétrico.

A biblioteca C6Run permite a comunicação entre ARM-DSP e o uso dos dois núcleos de forma fácil e sem complicações. Outras ferramentas como OpenCL [31] tem sido desenvolvidas para comunicação em MPSoCs heterogêneos, porém não serão abordadas neste trabalho.

2.4.1 DSP Bridge

O driver DSP Bridge provê ferramentas de controle e comunicação entre os processadores GPPs e DSPs, habilitando processamento paralelo assimétrico, em aplicações multimídia. Da perspectiva do processador ARM, o DSP é visto como um dispositivo do sistema, assim o DSP Bridge possibilita a comunicação destes processadores de duas maneiras, por mensagem usando pacotes de tamanhos definidos ou *streaming* de dados, usando buffers de tamanhos variáveis.

As aplicações que se utilizam da DSP Bridge para comunicação com GPP podem usar o processador para atividades como: Iniciar tarefas de processamento no DSP; Trocar mensagens com as tarefas no DSP; Compartilhar buffers de dados com tarefas no DSP; Pausar, resumir e apagar tarefas no DSP; Consultar estados de recursos das tarefas.

Nodo

A DSP Bridge possui uma abstração para a representação de controle em um DSP que reúne códigos relacionados e dados em unidades funcionais do DSP. Este é o nodo, um código compilado para uma arquitetura que compartilha o mesmo espaço de memória física que os processadores GPPs, sendo possível alocar memória específica para o DSP (método de comunicação via mensagem) ou compartilhar memória com o ARM (método de comunicação via *streaming*).

Um nodo possui cinco estados de processamento, (figura 2.10), estes estados são:

a) *Allocated*: Quando a ciclo de vida de um nodo DSP inicia o GPP faz a chamada da função *DSPNode_Allocate*, esta fará a alocação das estruturas de dados para que o GPP consiga fazer a comunicação e controle com o nodo, abrindo assim uma conexão com o DSP. O estado de *Allocated* somente existe no GPP;

b) *Created*: Quando a função *DSPNode_Create* é chamada, cria-se o nodo em um estado de pré-execução no DSP, juntamente com isso serão alocados os recursos necessários para a execução de tarefas no DSP;

c) *Running:* A chamada de *DSPNode_Run* faz com que o DSP execute a função de processamento de sinais digitais;

d) *Paused:* A função *DSPNode_Pause* permite que o GPP coloque o DSP em um estado de pausa, podendo requisitar a execução novamente através de uma chamada de *DSPNode_Run*;

e) *Done:* Este estado pode ser obtido de duas formas, seja pelo término da tarefa no DSP, ou pela chamada da função *DSPNode_Terminate.* Após o término da tarefa o nodo DSP pode ser desalocado e destruído pela chamada da função *DSPNode_Delete*, tanto do lado do DSP quanto GPP.

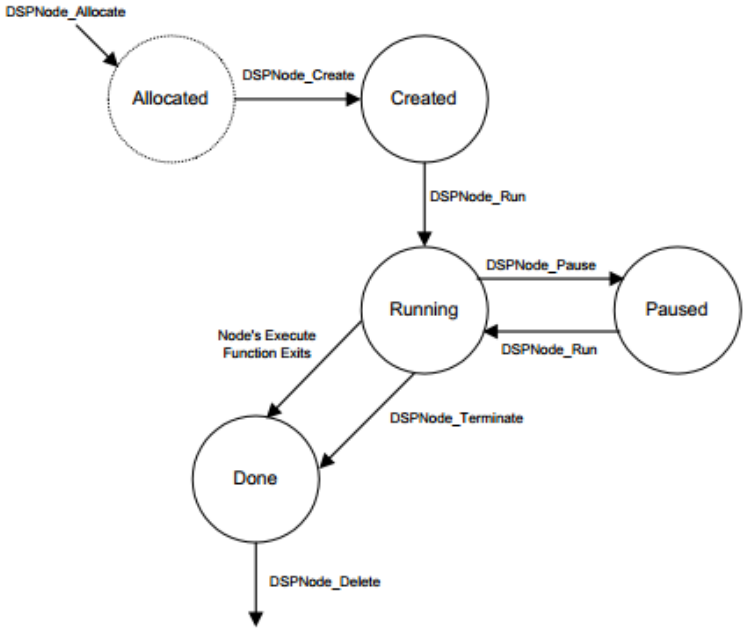

Figura 2.10: Estados possíveis de um nó DSP **[29]**

2.4.2 C6Run

Desenvolver algoritmos que utilizem processamento de forma paralela não é uma tarefa fácil tanto em arquiteturas homogêneas quanto heterogêneas. Junto a essa complexidade existente nas plataformas heterogêneas, também há dificuldade de se usar duas arquiteturas diferentes, GPP e SPP.

Bibliotecas como o DSP Brigde são úteis para a comunicação entre as arquiteturas, porém exige do programador conhecimento avançado na plataforma que se está usando. Isso implica em bibliotecas diferentes, funções intrínsecas diferentes, suporte da linguagem C em diferentes níveis como também ferramentas diferentes de programação. Como alternativa a isso a Texas Instruments desenvolveu a ferramenta *C6Run* [30] com três objetivos principais:

a) *Introduzir o desenvolvimento do DSP para programadores ARM com sistema Linux;*
b) *Simplificar o processo de utilização do DSP junto com o ARM;*
c) *Melhorar o desempenho do sistema utilizando os DSP MIPS.*

A proposta principal do C6Run é tornar o uso do DSP mais fácil de usar, mantendo o paradigma de que o DSP é um dispositivo escravo do processador ARM, porém sem a necessidade de uma chamada explícita como é feita na DSP Bridge (criação, utilização e

destruição do nodo DSP). Quando uma aplicação criada com C6Run executa o código no ARM, o DSP é inicializado juntamente com o código apropriado da aplicação (figura 2.11). Com o C6Run o DSP tem acesso ao sistema de arquivos e o console I/O do processador ARM, isso possibilita a programação de aplicações paralelas mais simples, pois não há alterações visíveis no código.

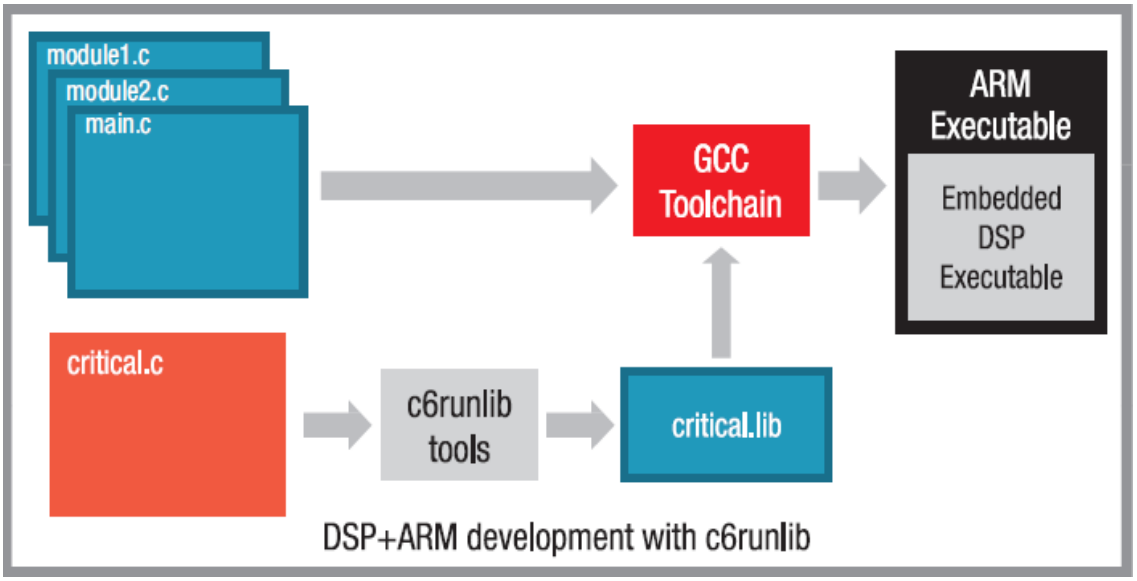

Figura 2.11: Biblioteca C6run **[30]**

Capítulo 3

Estudo de Caso xLupa Embarcado

O capítulo 3 apresenta um estudo de caso sobre o xLupa Embarcado, aplicação escolhida para ambiente de testes dos processadores citados no capítulo dois, esta foi desenvolvida em 2011 por Hackmann [8].

Primeiramente é apresentada a placa de desenvolvimento Beagleboard-XM, que foi utilizada neste trabalho. Logo após é explicado como é feita a captura dos frames de imagens pelo xLupa embarcado através da biblioteca V4L2 [32].

Também é mostrado neste capítulo o funcionamento do xLupa embarcado, e as alterações visando o aumento de desempenho feitas no software. Estas alterações estão divididas em dois tipos, alterações com uso de paralelismo de tarefas, e sem o uso.

3.1 Placa de desenvolvimento Beagleboard

A plataforma Beagleboard é um computador de baixo custo de tamanho reduzido, todos os seus periféricos e conexões, memória, armazenamento e processador estão embutidos em uma placa. Esta possui as principais características de um computador. O modelo utilizado neste projeto foi a Beagleboard XM que tem um custo em torno de U$$ 150,00, e possui as características indicadas na tabela 3.1.

Além do baixo custo a plataforma Beagleboard oferece suporte a diversos sistemas

operacionais Linux como Angstrom, Android, Ubuntu e XBMC. Outra característica que torna a Beagleboard XM viável para este trabalho é a existência de diversas conexões como HDMI, S-Video e porta serial.

A plataforma possui dois processadores que compõem o MPSoC DM3730, o processador de propósito geral ARM Cortex A8, da família AM37x, com um coprocessador NEON para instruções SIMD, e o um processador para sinais digitais o DSP TMS320C64x+.

Tabela 3.1: Especificação Beagleboard XM **[6]**

Processadores	ARM Cortex -A8 (Arquitetura RISC) com 1 GHz de frequência de operação TMS320C64x+ (Arquitetura DSP) com 800 MHz de frequência de operação
Memória	512-MB LPDDR RAM
Armazenamento	MICRO SD
Conexões	Entrada e Saída de áudio estéreo, 4xUSB 2.0 , Conector JTAG, HDMI, S-video, porta RS-232 Serial
Rede	10/100 Ethernet

O sistema instalado na plataforma foi o Ubuntu para processadores ARM, uma distribuição do Linux, na versão 11.10 a distribuição possui codinome Oneiric [33].

Originalmente a distribuição do Linux Ubuntu 11.10 possui kernel 2.6, porém para o uso do DSP nesta versão do kernel, é necessário fazer a compilação do mesmo. A versão 2.6 foi substituída pela 3.2.14 que já possui o driver para o processador DSP. Esta instalação foi feita via script fornecido pelo engenheiro da Digi-Key Robert Nelson [34].

3.2 Aplicação utilizada: ampliador digital xLupa embarcado

O xLupa embarcado é uma lupa digital para usuários de baixa visão e idosos. O sistema captura imagens e as trata para exibir em uma TV com entrada HDMI ou DVI. A principal característica do software é a opção de tratamento das imagens. Uma característica importante

a ser considerada no desenvolvimento de tecnologias assistivas para baixa visão é que cada indivíduo tem necessidades e patologias específicas, diferenciando-os em relação aos requisitos como a ampliação ou mesmo o contraste. Como meio de auxiliar estes usuários de baixa visão e idosos o xLupa embarcado possui diferentes perfis de configuração (figura 3.1).

O principal problema de execução do xLupa embarcado na plataforma Beagleboard é o alto consumo de processamento do processador ARM, que deixa a aplicação lenta. Alguns dos algoritmos causadores deste consumo de processamento são os de tratamento de imagens presentes no sistema. Os tópicos 3.6 e 3.7 mostram os algoritmos implementados neste trabalho.

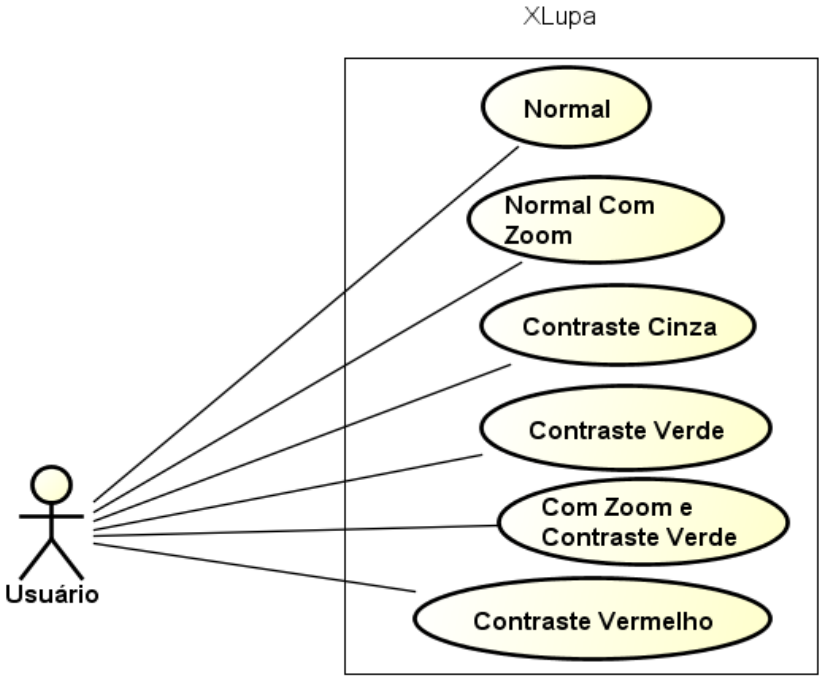

Figura 3.1: Perfis de imagem disponíveis no xLupa embarcado.

3.3 Versão original e modificações realizadas

O processo de discretização da imagem no xLupa embarcado é feito por meio do uso de uma webcam juntamente com a biblioteca V4L [32]. A webcam utilizada no sistema é a LifeCam HD-5000 fabricada pela Microsoft [35], com captura 720p e autofoco.

A *Video4Linux* ou V4L é uma biblioteca de captura e reprodução de vídeo para distribuições Linux, esta também é um *framework* de drivers para o kernel do Linux de suporte a webcams USB, sintonizadores de TV entre outros dispositivos de vídeo. V4L2 é a

segunda versão da biblioteca V4L, alguns erros foram corrigidos na segunda versão em relação à primeira.

O xLupa embarcado utiliza a V4L2 para fazer a captura dos frames de imagem da webcam. Esta captura é feita no formato YUYV pela biblioteca, com resolução da imagem capturada definida em 1280x720 pixels.

Os ponteiros dos frames obtidos por intermédio de chamadas de função de entrada e saída do sistema, as *ioctl*, são armazenados em uma fila circular para posterior tratamento pelo xLupa embarcado. É feita a chamada *ioctl* quando se deseja armazenar um novo frame de imagem com o uso da *flag* VIDIOC_QBUF (irá enfileirar um novo frame). Para informar à biblioteca que o frame de imagem pode ser substituído (irá desenfileirar o frame da fila) a flag VIDIOC_DQBUF é passada para a função *ioctl*.

Originalmente o xLupa embarcado era configurado para obter o frame de imagem da biblioteca V4L2 em formato RGB, entretanto como a captura dos frames é feita em formato JPEG, a biblioteca realizava a conversão para o formato RGB, acarretando em um tempo de captura maior (tabela 3.2). Observando a tabela 3.2 também é possível notar que devido ao fato do frame ser RGB, as operações de contraste cinza e limiarização se tornam mais custosas. Com o objetivo de aumentar o desempenho nestes quesitos, decidiu-se fazer a captura em YUYV, e com isso fazer somente as operações durante a conversão para o RGB conforme serão apresentadas na seção 3.3.2.

Tabela 3.2 Tempos dos processos do xLupa embarcado

Configuração	Tempo de captura ms	Tempo de processamento do algoritmo	Tempo de aplicação da ampliação	Tempo de saída	Tempo total
Normal	157,35 ms	0	0	75.21	232,56
Normal com zoom	157,35 ms	0	13,63	75.21	246,19
Contraste cinza	157,35 ms	98,21 ms	0	75.21	330,77
Contraste verde ou vermelho	157,35 ms	277,35 ms	0	75.21	509,91
Contraste verde e zoom	157,35 ms	277,35 ms	13,63	75.21	523,54

3.3.2 Algoritmos de processamento de imagem xLupa

Como a captura da imagem é feita em formato YUYV e a exibição desta necessita ser feita em formato RGB, então o algoritmo de conversão de imagens é necessário. Visando o aumento de desempenho em relação à versão do xLupa embarcado desenvolvida por Hachmann, onde é feita a conversão da imagem depois a aplicação do perfil, optou-se por fazer as operações de forma conjunta. A versão desenvolvida neste trabalho faz a captura, e de acordo com a escolha do usuário, faz somente a conversão YUYV/RGB ou a conversão YUYV/RGB + algoritmos de tratamento de imagens.

Os algoritmos usados no xLupa embarcado são de limiarização de imagem, conversão em nível de cinza e de ampliação de imagem. Sendo que os dois primeiros algoritmos e a conversão YUYV/RGB foram testados com execução tanto no ARM quanto no DSP, e o de ampliação somente no processador ARM.

O algoritmo de ampliação de imagem é feito pela função *cairo_scale* da biblioteca *Cairo Graphics* [36], que é uma biblioteca de processamento gráfico. Cairo foi desenvolvida para ser eficiente aproveitando a aceleração de hardware de exibição quando possível, produzindo saída consistente nos meios de saída. Baseado nos resultados de Hachmann [8], em que estes apontaram para o maior desempenho do processador ARM em relação ao DSP, no MPSoC OMAP 3530 [37] no algoritmo de *zoom* no xLupa embarcado, juntamente com a otmização da biblioteca *Cairo,* é preferível utilizar a função *cairo_scale* (função de ampliação da imagem) no processador ARM.

Algoritmo de conversão de YUYV/RGB

A conversão YUYV para RGB pode ser observada na figura 3.2, onde o algoritmo apresentado é o mesmo feito pela biblioteca V4L2. A função *sem_modificacao* recebe por parâmetro uma imagem de origem e destino. Nas linhas 2 a 5 é feita a declaração das variáveis que serão necessárias na conversão. O algoritmo de conversão é o mesmo para o processador ARM e DSP, sendo o *pragma* (linha 6) de uso somente do DSP.

O laço de repetição que percorre o vetor imagem (linha 7), processando cada conjunto de 4 bytes no vetor em YUYV e transformando em dois pixels (6 bytes) na imagem em RGB (linhas 8 a 20). A função CLIP (linhas 15 a 20) faz a saturação do resultado das transformações, pois se a operação retornar um número menor que 0 ou maior que 255, este

deve ser arredondado para 0 ou 255 respectivamente.

```
1  void inline sem_modificacao(unsigned char * restrict subimage, unsigned char* restrict dest) {
2      unsigned int j;
3      unsigned int i = 0;
4      unsigned int k = 0;
5      int u, v, u1, rg, v1;
6  #pragma MUST_ITERATE(8, ,8)
7      for (j = 0; j < 921600; j += 2, i += 6, k += 4) {
8          u = subimage[k + 1];
9          v = subimage[k + 3];
10         u1 = (((u - 128) << 7) + (u - 128)) >> 6;
11         rg = (((u - 128) << 1) + (u - 128) +
12                 ((v - 128) << 2) + ((v - 128) << 1)) >> 3;
13         v1 = (((v - 128) << 1) + (v - 128)) >> 1;
14         //O macro clip faz a saturação da operação para menor que zero ou maior que 255
15         dest[i] = CLIP(subimage[k] + v1);
16         dest[i + 1] = CLIP(subimage[k] - rg);
17         dest[i + 2] = CLIP(subimage[k] + u1);
18         dest[i + 3] = CLIP(subimage[k + 2] + v1);
19         dest[i + 4] = CLIP(subimage[k + 2] - rg);
20         dest[i + 5] = CLIP(subimage[k + 2] + u1);
21     }
22 }
```

Figura 3.2: Conversão YUYV para RGB

Algoritmo de conversão para níveis de cinza

No sistema de cores YUYV a componente Y representa os tons de cinza da imagem capturada, devido a esse fato para se converter a imagem capturada para cinza só é necessário fazer a atribuição da componente Y para as três componentes R, G e B (equação 3.1).

$$P_r \leftarrow Y, P_g \leftarrow Y \; e \; P_b \leftarrow Y \tag{3.1}$$

O algoritmo para conversão em níveis de cinza para o processador ARM (figura 3.3) é feito pela função *imagem_to_cinza*, sendo que esta recebe por parâmetro os vetores imagem de origem e destino. No laço logo em seguida, linha 5, é feito o percurso dos pixels da imagem fazendo as atribuições da equação 3.1 na imagem destino em RGB.

Para o processador DSP, o algoritmo utiliza-se da função intrínseca *_amem4 e _amem2* para fazer os *stores* e *loads* respectivamente dos dados dos vetores imagens (figura 3.4) de forma paralela. O primeiro passo feito no laço (linha 12) faz o *load* de quatro bytes do vetor

origem, sendo esses quatro bytes um conjunto YUVY. Com os quatro bytes é possível fazer a atribuição das componentes Y para dois pixels do vetor destino (linhas 23 a 25).

```
1  void imagem_to_cinza(const unsigned char * restrict subimage, unsigned char* restrict dest) {
2      unsigned int j;
3      unsigned int i = 0;
4      unsigned int k = 0;
5      for (j = 0; j < 921600; j += 2, i += 6, k += 4) {
6          dest[i] = subimage[k];
7          dest[i + 1] = subimage[k];
8          dest[i + 2] = subimage[k];
9          dest[i + 3] = subimage[k + 2];
10         dest[i + 4] = subimage[k + 2];
11         dest[i + 5] = subimage[k + 2];
12     }
13 }
```

Figura 3.3: Algoritmo de conversão em nível de cinza ARM

```
1  void imagem_to_cinza(const unsigned char * restrict subimage, unsigned char* restrict dest) {
2      unsigned int j = 0;
3      unsigned int i = 0;
4      unsigned int k = 0;
5      unsigned int de = 0;
6      unsigned short su1 = 0;
7      unsigned short su2 = 0;
8      unsigned short su3 = 0;
9
10     #pragma MUST_ITERATE(8, ,8)
11     for (j = 0; j < 921600; j += 2, i += 6, k += 4) {
12         de = _amem4_const(&subimage[k]);
13
14         ((unsigned char *) &su1)[0] = ((unsigned char *) &de)[0];
15         ((unsigned char *) &su1)[1] = ((unsigned char *) &de)[0];
16
17         ((unsigned char *) &su2)[0] = ((unsigned char *) &de)[0];
18         ((unsigned char *) &su2)[1] = ((unsigned char *) &de)[2];
19
20         ((unsigned char *) &su3)[0] = ((unsigned char *) &de)[2];
21         ((unsigned char *) &su3)[1] = ((unsigned char *) &de)[2];
22
23         _amem2(&dest[i]) = su1;
24         _amem2(&dest[i + 2]) = su2;
25         _amem2(&dest[i + 4]) = su3;
26     }
27 }
```

Figura 3.4: Algoritmo de conversão em nível de cinza DSP

Algoritmo de Limiarização

O processo de limiarização de uma imagem consiste em separar regiões de imagem em duas classes, fundo e objeto. O processo de limiarização, também usualmente chamado de

binarização, produz como resultado uma imagem binária separando a imagem em duas classes de objetos [38].

A limiarização no xLupa embarcado é a separação dos objetos em duas classes de cores, preto e as cores vermelho ou verde. Como o processo de limiarização da imagem é feito sobre a imagem em tons de cinza, é possível fazendo uso da componente Y do sistema de cores YUYV, fazer a limiarização utilizando somente uma instrução de controle como mostra a equação 3.2.

$$P_{RGB} = \begin{cases} R \leftarrow Y \; ou \; G \leftarrow Y \; ou \; B \leftarrow Y, & para \; Y > 127 \\ R \leftarrow 0, G \leftarrow 0 \; e \; B \leftarrow 0, & para \; Y \leq 127 \end{cases} \quad (3.2)$$

Semelhantemente ao algoritmo de conversão em níveis de cinza, o algoritmo de limiarização de imagem é muito parecido para os processadores ARM e DSP (figuras 3.4 e 3.5). Além do vetor de imagem passado por parâmetro, a função *limiar_imagem* recebe também a cor que se deseja de fundo, a imagem original e a imagem destino. Nas linhas 8 a 13 (figura 3.5) todas as componentes RGB de dois pixels são zeradas, para depois nas linhas 19 e 22 (figura 3.5) somente as componentes que são da cor passada por parâmetro receberem a componente Y, obtidas nas linhas 15 e 16 (figura 3.5).

A informação *MUST_ITERATE* (linha 6) é passada ao compilador (figura 3.6). Algoritmo para o limiar imagem para o processador DSP difere também na atribuição dos valores 0 da imagem (linhas 9 a 11, figura 3.6) onde é usada a função *_amem2* para fazer a atribuição de dois bytes em uma instrução.

```
1  void limiar_imagem(const unsigned char * restrict subimage, unsigned char* restrict dest, const unsigned char cor) {
2      unsigned int j;
3      unsigned int i = 0;
4      unsigned int k = 0;
5      unsigned char temp = 0, temp2 = 0;
6      for (j = 0; j < 921600; j += 2, i += 6, k += 4) {
7          //zera todas as componentes para o limiar
8          dest[i] = 0;
9          dest[i + 1] = 0;
10         dest[i + 2] = 0;
11         dest[i + 3] = 0;
12         dest[i + 4] = 0;
13         dest[i + 5] = 0;
14         //componente Y
15         temp = subimage[k];
16         temp2 = subimage[k + 2];
17
18         if (temp > 127)
19             dest[i + cor] = temp;
20
21         if (temp2 > 127)
22             dest[i + cor + 3] = temp2;
23     }
24 }
```

Figura 3.5: Algoritmo de limiarização de imagem para processador ARM

```
1  void inline limiar_imagem(const unsigned char * restrict subimage, unsigned char* restrict dest, const unsigned char cor) {
2      unsigned int j = 0;
3      unsigned int i = 0;
4      unsigned int k = 0;
5      unsigned char temp = 0, temp2 = 0;
6  #pragma MUST_ITERATE(8, ,8)
7      for (j = 0; j < 921600; j += 2, i += 6, k += 4) {
8          //zera todas as componentes para o limiar
9          _amem2(&dest[i]) = (unsigned short) 0;
10         _amem2(&dest[i + 2]) = (unsigned short) 0;
11         _amem2(&dest[i + 4]) = (unsigned short) 0;
12         //componente Y
13         temp = subimage[k];
14         temp2 = subimage[k + 2];
15
16         if (temp > 127)
17             dest[i + cor] = temp;
18
19         if (temp2 > 127)
20             dest[i + cor + 3] = temp2;
21
22     }
23 }
```

Figura 3.6: Algoritmo de limiarização de imagem para processador DSP

3.6 Execução do xLupa embarcado sem o uso de paralelismo de tarefas

Visando a comparação da eficiência dos algoritmos de processamento de imagem executando de forma sequencial, o xLupa embarcado foi modificado para executar sequencialmente utilizando o processador DSP. A captura e renderização da imagem continuam executando no processador ARM, somente os algoritmos de limiarização, conversão de imagem e em níveis de cinza foram colocados sob responsabilidade do DSP.

Após o início da execução o sistema faz a captura da imagem da webcam, por meio da biblioteca V4L2, logo em seguida é feita a aplicação do perfil ou somente a conversão da imagem YUYV para RGB, após isso a imagem é exibida ao usuário, caso o sistema continue executando a captura é feita novamente, caso contrário há o término da execução (figura 3.7).

Figura 3.7: Tratamento de imagem usando somente processador ARM

O tratamento de imagem utilizando o processador DSP é muito semelhante ao que usa somente o processador ARM. A diferença está no momento da aplicação do perfil ou a conversão da imagem YUYV para RGB, o frame de imagem é enviado juntamente com uma mensagem ao processador DSP, esta mensagem indica qual ação que o processador deve tomar. Após o término do processamento o DSP envia o frame juntamente com uma mensagem ao ARM, indicando que o processo foi concluído, o restante da execução é igual ao caso anterior (figura 3.8).

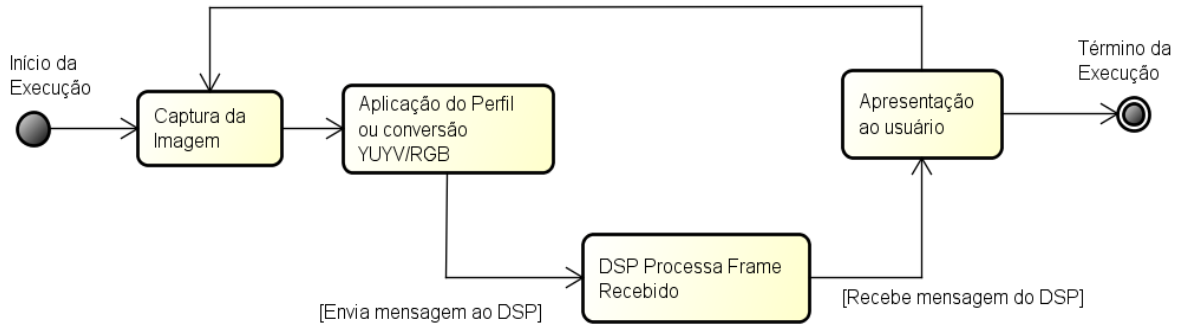

Figura 3.8: Tratamento de imagem usando processador DSP

3.7 Execução do xLupa embarcado com o uso de paralelismo de tarefas

A aplicação foi modificada para executar com o uso de paralelismo de tarefas, ou seja, foram adicionados threads para tentar aumentar o desempenho da aplicação. As implementações para ARM e ARM+DSP são praticamente idênticas (figuras 3.9 e 3.10), diferindo apenas na utilização do processador DSP na segunda implementação.

A figura 3.9 mostra como é a execução da aplicação com a utilização de três threads, sendo que thread *time_handler* é responsável pela captura do frame de imagem e armazenamento no buffer *captura_processa*. O thread *processa_imagem* é responsável por obter um frame de imagem do buffer *captura_processa* e fazer o tratamento de imagem conforme os algoritmos indicados no tópico 3.5.1, o resultado é armazenado no buffer *processa_exibe* (figura 3.9).

O resultado do processamento é exibido pelo thread *exibe_imagem*, onde é obtido do buffer *processa_exibe* a imagem já processada e convertida para RGB e passado para a

biblioteca GTK para exibição ao usuário. Caso seja necessário à aplicação de zoom na imagem a thread *exibe_imagem* também faz a execução do algoritmo *cairo_scale* (figuras 3.9 e 3.10). Todos os acessos aos buffers são protegidos por semáforos para que não ocorra o acesso indevido aos dados, sendo que os dois buffers utilizados têm o mesmo tamanho.

A principal diferença entre as implementações usando processador ARM e ARM+DSP está no momento do tratamento/conversão do frame de imagem (figura 3.10), onde na versão como o processador DSP a thread *processa_imagem* faz a chamada do DSP para a execução dos algoritmos.

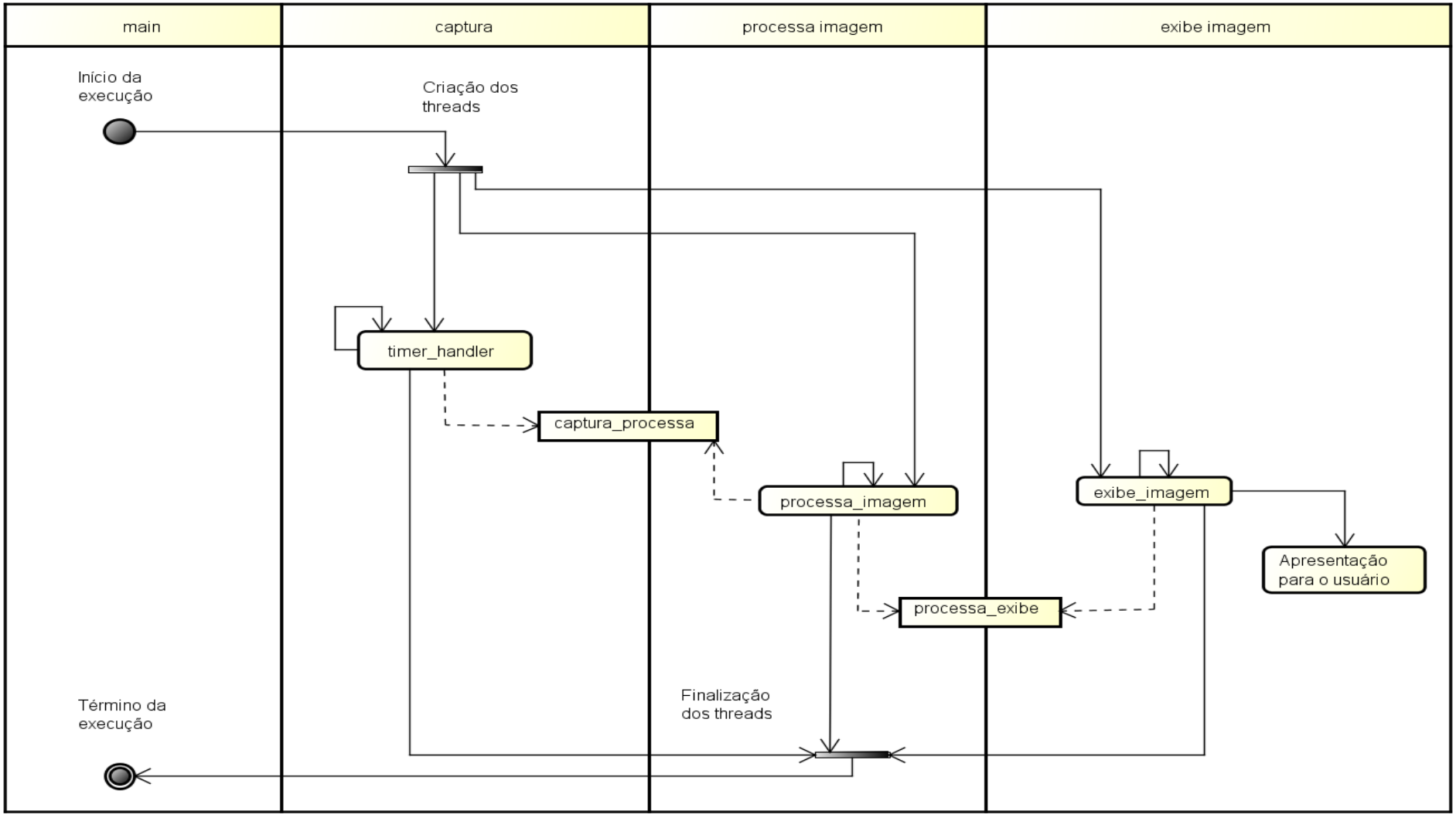

Figura 3.9: Tratamento de imagem com paralelismo utilizando processador ARM

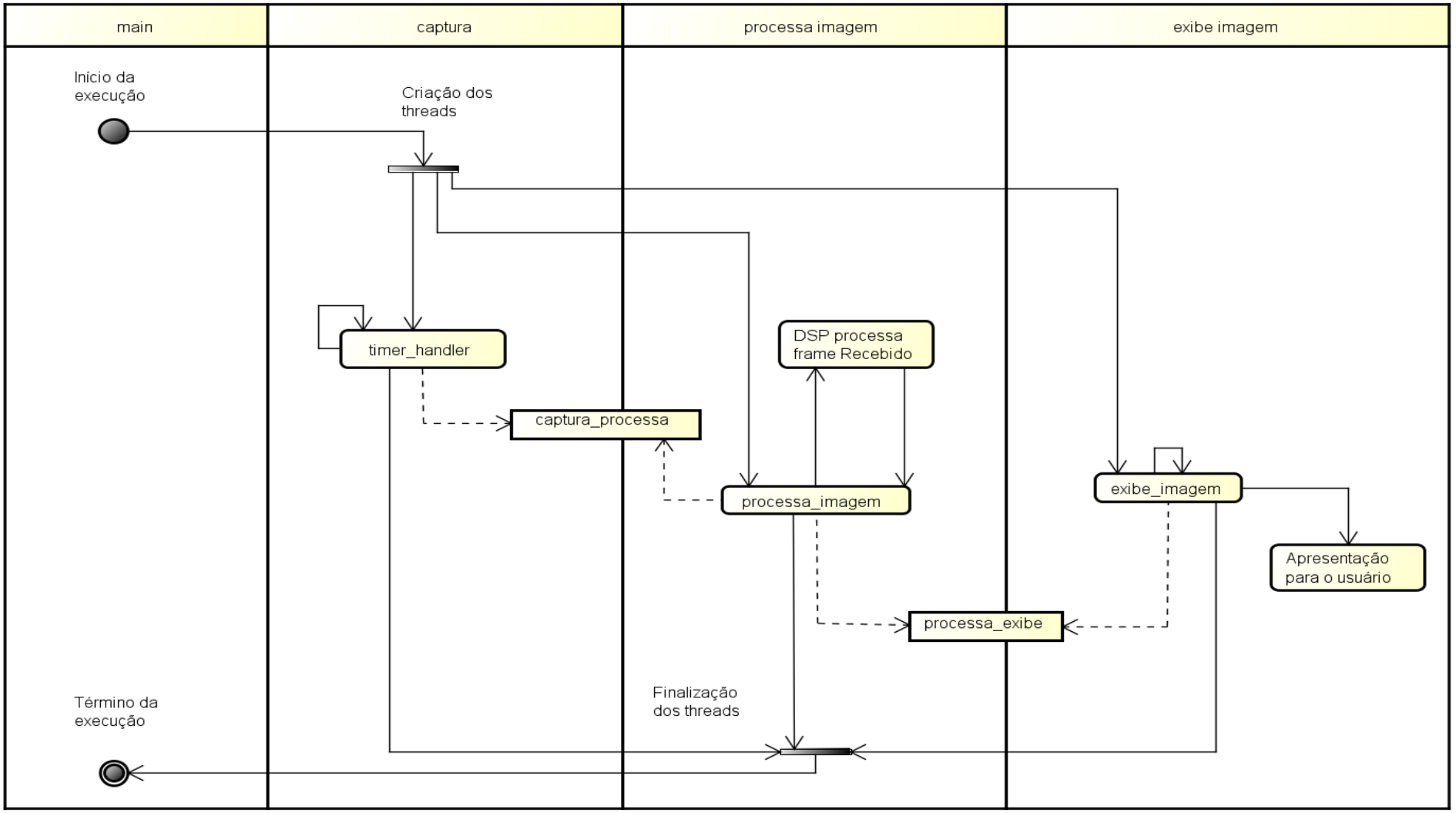

Figura 3.10: Tratamento de imagem com paralelismo utilizando processador ARM + DSP

Capítulo 4

Resultados

Neste capítulo serão apresentados os resultados obtidos com as implementações descritas no capítulo três. Foram levados em consideração os tempos de processamento de frame calculados e os FPS obtidos no momento da execução do xLupa embarcado.

As seções 4.2 e 4.3 irão relatar os resultados obtidos com as implementações usando os processadores ARM e DSP de modo sequencial e ARM+DSP como o uso de paralelismo de tarefas respectivamente.

4.1 Obtenção dos tempos de processamento de frame e quantidade de frames por segundo

As medidas para fim de comparação obtidas nos algoritmos foram divididas em duas, os tempos de processamento dos algoritmos e frames por segundo (FPS), ambos os tempos foram calculados usando somente o processador ARM, e usando ARM + DSP.

Tempo de processamento dos algoritmos

O tempo de processamento dos algoritmos se refere à diferença de tempo medida antes da

execução dos algoritmos de tratamento de imagem, conversão em escala de cinza, limiarização ou conversão YUVY/RGB, e o tempo logo após o término do processamento. Foram retiradas 100 amostras de tempo e feita média aritmética sobre elas.

Frames por segundo (FPS)

Os FPS medidos na execução do programa foram obtidos por intermédio da média de tempo entre 30 frames capturados. O contador de tempo era disparado no momento que antecedia a captura do frame 0, e recapturado logo após o frame 30 ser obtido, ao final da captura a proporção de tempo é calculada para cada frame . Para cada perfil foram feitos 10 cálculos de FPS, ao final da captura das amostras a média aritmética foi feita.

Para as implementações que se utilizam do paralelismo de tarefas foi necessário colocar um marcador em uma das posições dos buffers compartilhados, para que o FPS pudesse ser calculado. Sendo assim quando o frame é marcado o tempo começa a ser contado, sendo coletado quando o frame é exibido ao usuário.

4.2 Desempenho das implementações sequenciais

Após a coleta dos dados tanto da quantidade de frames por segundo quanto o tempo de processamento por algoritmo, foi possível fazer uma comparação entre as implementações sequenciais executando tanto no processador ARM e também no DSP. A principal desvantagem em usar somente o processador ARM para executar todo o processo de captura, de processamento de imagem e exibição ao usuário é a sobrecarga do processador. Por outro lado não existe o *overhead* de troca de mensagens com o processador DSP.

Algoritmos de tratamento de imagem

A tabela 4.1 mostra a diferença entre os tempos de processamento de imagem entre o ARM e o DSP para os algoritmos de aplicação de escala de cinza, contraste e conversão para RGB. É importante notar que em todos os algoritmos a imagem de entrada é YUV e a de saída é RGB. Os resultados mostram uma característica interessante, tanto no ARM quanto no DSP a aplicação do contraste de cinza é muito mais rápido que a conversão para RGB. Em relação ao desempenho ARM versus DSP, devido às características diferentes dos processadores,

como frequências de processamento, o tempo de processamento no DSP é até 535 % maior em relação ao ARM.

Tabela 4.1: Tempo de processamento dos algoritmos (100 amostras)

Configuração	ARM	DSP	Desempenho DSP/ARM
Contraste cinza	12,54 ms	79,60 ms	-535%
Contraste de cor	14,48 ms	79,10 ms	-446%
Conversão YUYV/RGB	35,71 ms	99,43 ms	-178%

Com as otimizações dos algoritmos, os tempos de captura e exibição dos frames de imagem mudaram (tabela 4.2). Um problema foi encontrado com a otimização da conversão YUYV/RGB juntamente com os algoritmos de processamento de imagem, apesar de este tempo ter diminuído, a exibição e ampliação consumiram mais tempo. Isso ocorre por existir uma concorrência com o servidor de vídeo Xorg, reduzindo-se a ocupação da CPU pelo xLupa (devido a redução do tempo de processamento dos algoritmos apresentado na Tabela 4.1), o sistema tende a realocar a CPU para o servidor X, fazendo com que mais frames sejam processados pelo servidor X e impactando no tempo de ampliação e saída na tela.

Tabela 4.2 Tempos dos processos no processador ARM do xLupa embarcado (sem os algoritmos da tabela 4.1)

Configuração	Tempo de captura ms	Tempo de aplicação da ampliação	Tempo de saída	Tempo total
Normal	1,15 ms	0 ms	109,18 ms	110,33 ms
Normal com zoom	1,15 ms	163,09 ms	109,18 ms	273,42 ms
Contraste cinza	1,15 ms	0 ms	109,18 ms	110,33 ms
Contraste verde ou vermelho	1,15 ms	0 ms	109,18 ms	110,33 ms
Contraste verde e zoom	1,15 ms	163,09 ms	109,18 ms	273,42 ms

Outro dado coletado durante as execuções é em relação ao overhead da cópia para a memória. Como a memória compartilhada entre os processadores ARM e DSP deve ser mapeada no momento da criação do nodo DSP, os dados que estão no buffer circular de captura de imagem não podem ser alterados no próprio buffer, estes devem ser copiados para a região compartilhada entre o ARM e o DSP, isso acarreta em um overhead de cópia de memória. O tempo de cópia de um vetor com o tamanho dos frames de imagem em RGB no ARM é em média 10.04 milissegundos (tempo obtido com 100 amostras).

Frames por segundo

A tabela 4.3 mostra os resultados da implementação sequencial em FPS, esta reflete as diferenças dos processadores observadas na medição do tempo de processamento. As diferenças nos perfis sem zoom e com zoom são muito pequenas visto que a sequência de processamento para estes casos não muda, ou seja, o processador DSP não chega a ser requisitado na implementação ARM + DSP para as configurações com e sem zoom, sendo responsabilidade somente do ARM esta etapa do processamento.

Tabela 4.3: Média de FPS

Configuração	FPS ARM	FPS DSP	Desempenho DSP/ARM
Sem zoom	6,00 fps	4,55 fps	-24%
Com zoom	3,09 fps	3,16 fps	2%
Cinza	6,64 fps	4,50 fps	-32%
Verde	6,67 fps	5,00 fps	-25%
Verde com zoom	3,27 fps	3,17 fps	-3%
Vermelho	6,64 fps	4,98 fps	-25%

4.3 Desempenho das implementações paralelas

Na implementação sequencial ARM+DSP pode-se notar que quando é passada a mensagem para o processador DSP solicitando o processamento de um frame o processador

ARM fica ocioso. Assim, o uso de paralelismo visa utilizar este tempo ocioso do processador ARM.

Para fins de comparação foram analisadas as implementações com o uso do paralelismo de tarefas com os processadores ARM e ARM+DSP.

A tabela 4.4 mostra a quantidade de frames por segundo obtidos na implementação paralela para o xLupa embarcado. Com a utilização de paralelismo pode-se notar a grande queda de desempenho na execução do sistema, isso se deve ao fato da diferença de tempo de processamento dos threads *time_handler, processa_imagem e exibe_imagem*.

Os threads *time_handler* e *processa_imagem* possuem os algoritmos de captura e processamento de imagem, com tempo de execução de 0,1 ms para captura de um frame e até 100 ms para processamento da imagem. Já o thread de exibição *exibe_imagem* possui o tempo de processamento de no mínimo 109 ms, podendo ser mais se este estiver executando o algoritmo de ampliação. Essa diferença de tempo de captura, processamento e exibição de imagem leva a um *overhead* de comunicação entre os threads, ou seja, o thread de exibição acaba segurando a execução dos outros, acarretando em perda de desempenho.

Outro resultado que é possível ser observado é que, a implementação ARM+DSP foi superior a implementação utilizando somente processador ARM mesmo ficando muito aquém do esperado.

Tabela 4.4: Média de FPS para implementações utilizando paralelismo de tarefas

Configuração	**FPS ARM**	**FPS ARM + DSP**	**Desempenho DSP/ARM**
Sem zoom	1,44 fps	1,70 fps	17,98%
Com zoom	0,65 fps	0,72 fps	10,97%
Cinza	1,52 fps	1,69 fps	11,13%
Verde	1,52 fps	1,64 fps	7,48%
Verde com zoom	0,67 fps	0,71 fps	5,68%
Vermelho	1,53 fps	1,63 fps	6,56%

O uso do paralelismo acarretou em outro problema, o overhead de comunicação entre os threads, deixando o desempenho da aplicação muito abaixo do esperado, a tabela 4.5 mostra

um comparativo entre as implementações. É possível observar (tabela 4.5 e figura 4.1) que em todos os perfis a implementação sequencial é no mínimo 60% mais rápida do que a utilizando o paralelismo, tornando inviável o uso de múltiplas threads no xLupa embarcado com MPSoC DM 3730, pois o processador ARM já está sobrecarregado, tanto com processos do sistema como também da aplicação.

Tabela 4.5: Tabela comparativa entre as implementações

Configuração	Seq. ARM	Seq. ARM+DSP	Par. ARM	Par. ARM +DSP	Sequencial/Paralelo ARM + DSP	Sequencial/ Paralelo ARM
Sem zoom	6,00 fps	4,55 fps	1,44 fps	1,70 fps	-63%	-76%
Com zoom	3,09 fps	3,16 fps	0,65 fps	0,72 fps	-77%	-79%
Cinza	6,64 fps	4,50 fps	1,52 fps	1,69 fps	-62%	-77%
Verde	6,67 fps	5,00 fps	1,52 fps	1,64 fps	-67%	-77%
Verde com zoom	3,27 fps	3,17 fps	0,67 fps	0,71 fps	-78%	-79%
Vermelho	6,64 fps	4,98 fps	1,53 fps	1,63 fps	-67%	-77%

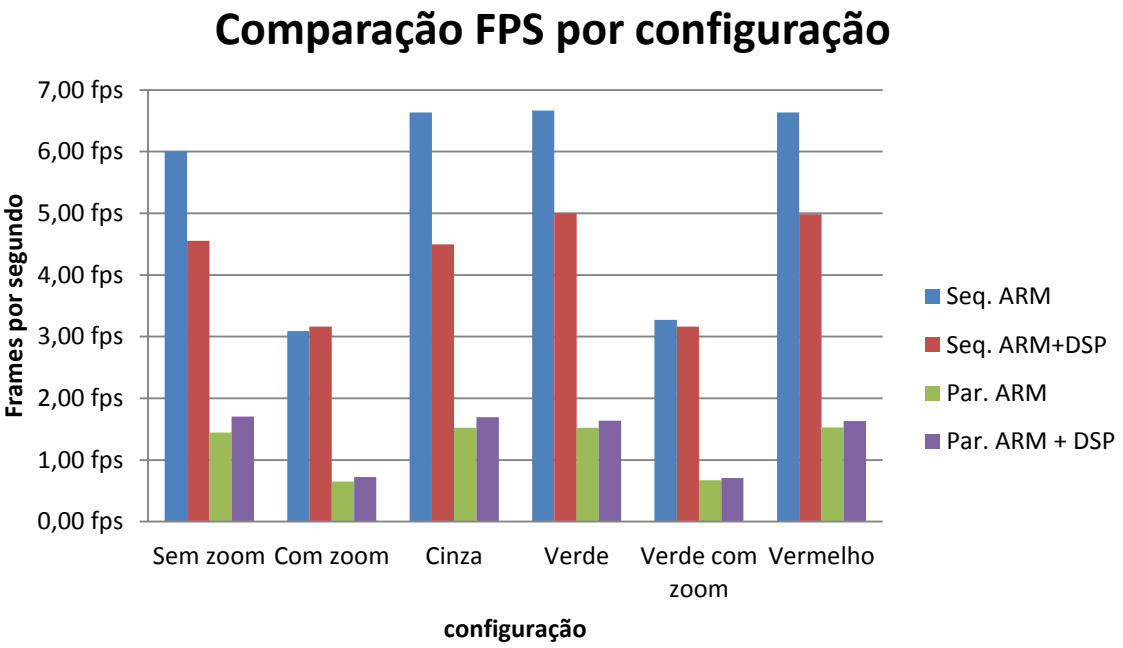

Figura 4.1: Comparação das implementações sequencial e paralelas

Capítulo 5

Conclusões e trabalhos futuros

Foram apresentadas as motivações para o uso das arquiteturas que possibilitam o uso de ILP e suas limitações. O estudo destas limitações foi feito por meio da implementação de algoritmos no MPSoC heterogêneo DM3730, que é parte integrante da placa de desenvolvimento Beagleboard.

O MPSoC DM3730 foi analisado visando a extração do máximo desempenho, nesta análise pode-se diferenciar as principais características dos dois processadores presentes no DM3730, o ARM Cortex A8 e DSP C64x+, e tentar extrair o máximo do potencial das duas arquiteturas.

As modificações feitas no estudo de caso do xLupa embarcado para a utilização do DSP mostraram as dificuldades de se usar uma arquitetura de propósito específico. Mesmo com a utilização da biblioteca DSP Bridge a programação para o processador DSP se mostrou mais complexa do que para o processador de propósito geral, ARM, pois ao se programar para um processador de propósito específico deve-se conhecer bem as suas características.

Apesar da execução dos algoritmos do xLupa embarcado com a utilização do DSP ter se mostrado, em alguns casos em torno de 500 % mais lenta, vale salientar que no momento da execução do algoritmo no DSP o processador ARM fica em estado ocioso, fato que permitiu o uso de paralelismo em nível de tarefas na plataforma.

O principal causador desta queda de desempenho entre a versão paralelizada e sequencial é o tempo que se gasta para fazer a renderização do resultado ao usuário, sendo assim deve se

encontrar outras maneiras de se mostrar o resultado. Outro fator que contribui para a perda de desempenho é a competição do processador pelos processos do xLupa e Xorg (servidor gráfico), que acaba diminuindo a quantidade de FPS.

Com este trabalho pode-se concluir que é viável o uso de processadores de sinais digitais para dados altamente paralelizáveis, ou seja, o acesso à memória pode ser feito de forma alinhada e o uso de instruções SIMD, como no exemplo da soma de vetores. Para se fazer uso do DSP é preciso levar em consideração também o tipo do algoritmo que se quer utilizar, onde este, não pode possuir muitas instruções de controle, pois a execução do algoritmo pode ser prejudicada por estas.

Outro ponto que pode ser observado é que, dependendo da aplicação uma implementação que utiliza-se de paralelismo a nível de tarefas pode não ser benéfica, pois o overhead causado pela comunicação dos threads pode acarretar em perda de desempenho.

Como pode ser observado neste trabalho, o processo de apresentar ao usuário o resultado do tratamento de imagem exige uma grande quantidade de processamento, sendo assim um trabalho futuro a este é a execução do xLupa embarcado sem o uso do servidor gráfico Xorg, realizando a escrita direta para o driver via framebuffer. Como observado nos resultados, grande parte do tempo de processamento se deve ao fato que o servidor gráfico do sistema compete como o xLupa na ocupação do processador, aumentando assim o tempo de processamento dos frames de imagem.

Um trabalho futuro que pode ser feito é avaliação do uso da biblioteca *TMS320C6000 Image Library*, esta que é uma biblioteca de processamento de imagem e vídeo para aplicações em tempo real para a família dos processadores TMS320C6000 [39]. A biblioteca possui os códigos já otimizados em linguagem de montagem, que segundo Wu o uso desta biblioteca traz um aumento de desempenho em relação à programação em linguagem C [40].

Outro trabalho futuro que deve-se citar, é viabilizar a portabilidade do software do xLupa embarcado para o sistema operacional Android [41], visando a utilização do sistema em dispositivos móveis, atingindo assim um grande número de usuários.

Referências Bibliográficas

1. MORIMOTO, C. E. Guia do Hardware. **Entendendo os sistemas embarcados**, 2007. Disponivel em: <http://www.hardware.com.br/artigos/entendendo-sistemas-embarcados/>. Acesso em: 25 Junho 2014.

2. BALARIN, F. et al. **Hardware-Software Co-Design of Embedded Systems The POLIS Approach**. 1ª. ed. Norwell: Kluwer Academic Publishers, v. I, 1997.

3. HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture A Quantitative Approach**. 5ª. ed. Waltham: Elsevier, v. I, 2011. ISBN 978-0123838728.

4. WOLF, W.; A., J. A.; MARTIN, G. Multiprocessor System-on-Chip (MPSoC) Technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, 19 Setembro 2008. p. 1-7.

5. TEXAS INSTRUMENTS. **DM3730 Digital Media Processor**. Disponivel em: <http://www.ti.com/product/dm3730>. Acesso em: 20 Junho 2014.

6. BEAGLEBOARD. Beagleboard.org. **BeagleBoard-xM**. Disponivel em: <http://beagleboard.org/beagleboard-xm>. Acesso em: 1 Julho 2014.

7. DIGI-KEY. Digi-Key Corporation. **Beagleboard and Beaglebone**. Disponivel em: <http://www.digikey.com/product-highlights/us/en/texas-instruments-beagleboard/685>. Acesso em: 1 Julho 2014.

8. HACHMANN, D. R. **Distribuição de tarefas em MPSoC Heterogêneo: estudo de caso no OMAP3530**. Dissertação (Trabalho de conclusão de curso) - UNIOESTE - Universidade Estadual do Oeste do Paraná. Cascavel-PR. Dezembro 2011.

9. MARWEDEL, P. **Embedded System Design**. 1ª. ed. Netherland: Springer, 2006. ISBN 978-0-387-29237-3.

10. DELAI, A. L. Guia do Hardware. **Sistemas Embarcados a computação invisível**, Rio de Janeiro, 2009. Disponivel em: <http://www.hardware.com.br/artigos/sistemas-

embarcados-computacao-invisivel/conceito.html>. Acesso em: 28 Setembro 2014.

11. BADAWY, W.; JULIEN, G. A. **System-on-Chip for Real-Time Applications**. 1ª. ed. Calgary: Kluwer Academic Publishers, v. I, 2003. ISBN 978-1-4615-0351-4.

12. JÚNIOR, V. C. Tecnologia SOC e o microcontolador PSoC (Programmable System on Chip). **Revista Integração**, Mauá-SP, v. I, n. 2, p. p. 251 - 253, Julho 2004.

13. MARTIN, G. **Overview of the MPSoC Design Challenge**, San Francisco, California, USA., 28 Julho 2006. p. 274 - 277.

14. RICHTER, K.; JERSAK, M.; ERNST, R. A Formal Approach to MpSoC Performance Verification. **IEEE Computer Society**, Braunschweig, 3 Abril 2003. 1.

15. HE, Y. et al. Efficient communication support in predictable heterogeneous MPSoC designs for streaming applications. **Journal of Systems Architecture**, Eindhoven, Netherland, 2 Maio 2013. p. 1 - 2.

16. BOSE, P. General-purpose versus application-specific processors. **IEEE computer Society**, 2004. p. 1.

17. STALLINGS, W. **Arquitetura e Organização de Computadores**. 8ª. ed. São Paulo - SP: Pearson, v. I, 2010. ISBN 978-85-7605-564-8.

18. ARM. Company Profile. **The world's leading semiconductor intellectual property (IP) supplier**, 2014. Disponivel em: <http://www.arm.com/about/company-profile/index.php>. Acesso em: 20 Junho 2014.

19. ARM. Cortex M series. **ARM Cortex**, 2014. Disponivel em: <http://www.arm.com/products/processors/cortex-m/index.php>. Acesso em: 20 Junho 2014.

20. SILVA, L. F.; ANTUNES, V. J. M. **Comparação entre as arquitecturas de processadores RISC e CISC**, Cidade do Porto, Portugal. p. 2 - 6.

21. MONTEIRO, M. A. **Introdução a Organização de Computadores**. 5ª. ed. Rio de Janeiro - RJ: LTC, v. I, 2007. ISBN 9788521615439.

22. SLOSS, A.; SYMES, D.; WRIGHT, C. **ARM System Developer's Guide:** Designing and Optimizing System Software. San Francisco, CA, USA: Elsevier, 2004. ISBN 1558608745.

23. TEXAS INSTRUMENTS. **Cortex™-A8 Revision: r3p2 Technical Reference Manual**. Texas Instruments Incorporated. [S.l.]. 2010. (ID060510).

24. CHEGG. Chegg study. **Definition of signal digital**, 2003. Disponivel em: <http://www.chegg.com/homework-help/definitions/digital-signal-4>. Acesso em: 28 Setembro 2014.

25. ARM. **The ARM Architecture With a focus on v7A and Cortex-A8**, 2014. Disponivel em: <http://www.arm.com/files/pdf/ARM_Arch_A8.pdf>. Acesso em: 25 Julho 2014.

26. TEXAS INSTRUMENTS. **TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide**. Texas Instruments Incorporated. [S.l.]. 2010. (SPRU732J).

27. TEXAS INSTRUMENTS. **TMS320C6000 DSP Cache User's Guide**. Texas Incoporated. [S.l.]. 2003. (SPRU656A).

28. TEXAS INSTRUMENTS. **TMS320C6000 Optimizing Compiler v7.6**. Texas Incorporated. [S.l.]. 2014. (SPRU187V).

29. OMAPPEDIA. WIKI. **DSPBridge Project**, 2010. Disponivel em: <http://omappedia.org/wiki/DSPBridge_Project#About_DSP_Bridge>. Acesso em: 20 Julho 2014.

30. ALLRED, D. C6Run DSP Software Development Tool. **White Paper**, Dallas, Texas, USA, 2010. p. 2 e 3.

31. TEXAS INSTRUMENTS. Texas Instruments Wiki. **OpenCL**, 2012. Disponivel em: <http://processors.wiki.ti.com/index.php/OpenCL>. Acesso em: 20 Julho 2014.

32. LINUXTV. Lib V4L2. **Video for Linux Two API Specification**, 2014. Disponivel em: <http://linuxtv.org/downloads/v4l-dvb-apis/>. Acesso em: 20 Julho 2014.

33. ELINUX. Elinux Wiki. **BeagleBoardUbuntu**, 2014. Disponivel em: <http://elinux.org/BeagleBoardUbuntu>. Acesso em: 20 Julho 2014.

34. NELSON, R. C. GitHub. **RoberCNelson**, 2009. Disponivel em: <https://github.com/RobertCNelson>. Acesso em: 20 Julho 2014.

35. MICROSOFT. Microsoft Hardware. **LifeCam HD-5000**, 2014. Disponivel em: <http://www.microsoft.com/hardware/pt-br/p/lifecam-hd-5000>. Acesso em: 20 Julho 2014.

36. CAIRO. Cairo Graphics. **Cairo API**, 2013. Disponivel em: <http://cairographics.org/>. Acesso em: 25 Julho 2014.

37. TEXAS INSTRUMENTS. **OMAP3530 Aplications Processor**, 2014. Disponivel em: <http://www.ti.com/product/omap3530>. Acesso em: 25 Julho 2014.

38. FILHO, O. M.; NETO, H. V. **Processamento Digital de Imagens**. 1ª. ed. Rio de Janeiro, RJ: Brasport, v. I, 1999. ISBN 8574520098.

39. TEXAS INSTRUMENTS. TMS320C6000 Image Library. **IMGLIB**, 2014. Disponivel em: <http://www.ti.com/tool/sprc264>. Acesso em: 20 Novembro 2014.

40. WU, T.; ZHOU, J.; PAN, J. A Research of DCT Algorithm Based on OMAP 3530. **Second International Workshop on Computer Science and Engineering**, Qingdao, China, 28 Outubro 2009. 1 a 5.

41. GOOGLE. http://www.android.com/. **Android**, 2005. Disponivel em: <http://www.android.com/>. Acesso em: 12 Outubro 2014.