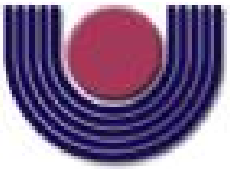


Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Distribuição de tarefas em MPSoC Heterogêneo: estudo de caso no OMAP3530

Diego Rodrigo Hachmann

CASCAVEL
2011

Diego Rodrigo Hachmann

Distribuição de tarefas em MPSoC Heterogêneo: estudo de caso no OMAP3530

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. Jorge Bidarra

CASCAVEL
2011

DIEGO RODRIGO HACHMANN

DISTRIBUIÇÃO DE TAREFAS EM MPSOC HETEROGÊNEO: ESTUDO DE CASO NO OMAP3530

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. Jorge Bidarra (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Márcio Seiji Oyamada (Co-orientador)
Colegiado de Ciência da Computação,
UNIOESTE

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

Cascavel, 16 de novembro de 2011

"O que me assusta, não é a violência de poucos, mas o silêncio de muitos"

(-Martin Luther King)

“Às vezes a vida te bate com um tijolo na cabeça. Não perca a fé. Estou convencido de que a única coisa que me fez continuar foi que eu amava o que eu fazia. Você precisa encontrar o que você ama. E isso vale para o seu trabalho e para seus amores. Seu trabalho irá tomar uma grande parte da sua vida e o único meio de ficar satisfeito é fazer o que você acredita ser um grande trabalho. E o único meio de se fazer um grande trabalho é amando o que você faz. Caso você ainda não tenha encontrado, continue procurando. Não pare. Do mesmo modo como todos os problemas do coração, você saberá quando encontrar. E, como em qualquer relacionamento longo, só fica melhor e melhor ao longo dos anos. Por isso, continue procurando até encontrar, não pare".

(-Steve Jobs, discurso durante formatura em Stanford, 2005)

AGRADECIMENTOS

Serei eternamente grato aos meus pais, Nilson Erno Hachmann e Maria Tânia Bertolini, pelo apoio incondicional em todos os momentos.

Agradeço a minha namorada por todo amor, amizade, e por me entender nos momentos de ausência.

Agradeço aos meus irmãos, meus tios e minhas avós por me apoiarem sempre, apesar da distância.

Agradeço ao meu orientador, Jorge Bidarra, pela orientação deste trabalho e pelos três anos de IC, onde nunca mediu esforços diante dos desafios e das dificuldades. Agradeço também por sua amizade e companheirismo durantes todos estes anos. Serei eternamente grato, obrigado.

Agradeço os meus companheiros de IC. Aos que estavam comigo deste o inicio, Odair Moreira de Souza e Cleiton Fiatkoski Balansin, e a Dener Júnior Ribeiro da Cunha que veio integrar o projeto mais tarde.

Agradeço a todos os companheiros da Maratona de Programação, em especial aos *coaches* Josué Pereira de Castro e Adriana Postal, e aos integrantes da minha primeira equipe, Evaristo Wychoski Benfatti e Tiago Sipert, e da segunda equipe, Dener Júnior Ribeiro da Cunha e Anderson Roberto Slivinski.

Agradeço ao meu co-orientar Marcio Seiji Oyamada e a Adair Santa Catarina pela disponibilidade e sugestões em todos os momentos de dúvidas na realização deste trabalho. A Paulo Wesley, pela ajuda na instalação do ambiente de trabalho.

Agradeço a todos os amigos e colegas que fiz durante estes cinco anos de graduação. A amizade de vocês não há riqueza que pague. Serei eternamente grato a todos.

Agradeço também a Dennis Ritchie por criar a linguagem C, e a Linus Torvalds por ter criado o Linux.

A todos vocês, muito obrigado.

Sumário

Sumário vi

Lista de Figuras viii

Lista de Tabelas ix

Lista de Abreviaturas e Siglas x

Resumo xi

1 Introdução 1
- 1.1 Contextualização . . . 1
- 1.2 Problemas . . . 3
- 1.3 Motivações . . . 3
- 1.4 Objetivos . . . 4
- 1.5 Organização do Texto . . . 4

2 MPSoC Heterogêneos 6
- 2.1 MPSoC em Aplicações Embarcadas . . . 6
- 2.2 Processadores Heterogêneos : GPPs x SPPs . . . 7
- 2.3 MPSoC OMAP3530 . . . 9
- 2.4 Processadores de Propósito Geral : ARM . . . 10
  - 2.4.1 ARM como arquitetura RISC . . . 12
  - 2.4.2 Particularidades da Arquitetura ARM . . . 14
  - 2.4.3 Coprocessadores . . . 15
  - 2.4.4 Processador ARM Cortex-A8 . . . 16
- 2.5 Processadores de Propósito Específico : DSP . . . 18
  - 2.5.1 Processamento de Sinais Digitais . . . 18
  - 2.5.2 Processadores de Sinais Digitais . . . 19
  - 2.5.3 Processador DSP TMS320C64X+ . . . 20
  - 2.5.4 Programação do Processador DSP . . . 29
- 2.6 Comunicação ARM-DSP . . . 33
  - 2.6.1 Aspectos Gerais . . . 33
  - 2.6.2 Nodo . . . 34
  - 2.6.3 Estados do Processamento . . . 35

3 Ampliador digital: ambiente e testes **37**
3.1 Placa de Desenvolvimento BeagleBoard 37
3.2 Distribuição Linux Utilizada 38
3.3 Obtenção dos tempos para análise 39
3.4 Captura de imagem da webcam 40
3.5 Aplicação desenvolvida: ampliador xLupa embarcado 41
3.6 Algoritmos de tratamento de imagem 43
3.6.1 Algoritmo escala de cinza 43
3.6.2 Algoritmo brilho 44
3.6.3 Algoritmo binarização 45
3.6.4 Algoritmo ampliação 46

4 Resultados **51**
4.1 Algoritmos de tratamento de imagem 51
4.1.1 Algoritmo escala de cinza 51
4.1.2 Algoritmo brilho 52
4.1.3 Algoritmo binarização 53
4.1.4 Algoritmo ampliação 54
4.2 Tempos teóricos do DSP 56
4.3 Tempos da aplicação 57

5 Conclusão e Trabalhos Futuros **62**
5.1 Conclusão 62
5.2 Trabalhos Futuros 64

Referências Bibliográficas **65**

Lista de Figuras

1.1 ILP em um processador perfeito para seis dos *benchmarks* SPEC92 [1] 2

2.1 Processador OMAP3530. Dimensões: 63.5 x 35 x 6.4 mm 10
2.2 Diagrama de blocos do processador OMAP 3530 10
2.3 Diagrama de blocos do processador OMAP 5430 12
2.4 Diagrama de blocos do processador ARM Cortex-A8 17
2.5 Pipeline Processador ARM Cortex-A8 18
2.6 Filtro FIR e Transformada Discreta de Fourier 19
2.7 Fluxo de instruções no processador DSP c64x+ 22
2.8 Formáto básico de um pacote de instruções 24
2.9 Pipeline cheio no c64x+ 25
2.10 Arquitetura da memória cache no c64x+ 27
2.11 Instrução SADDU4 sobre dois pacotes de 32 bits 28
2.12 Instrução auxiliar de divisão SUBC 29
2.13 Implementação da divisão em uma chamada C 30
2.14 Troca de mensagens entre ARM e DSP 34
2.15 Estados de Processamento de um Nodo 35

3.1 Placa BeagleBoard e seus recursos. 38
3.2 Ambiente montado para os testes 42
3.3 Fluxo de dados no xLupa embarcado 42
3.4 Algoritmo escala de cinza aplicado sobre uma imagem 44
3.5 Algoritmo brilho aplicado sobre uma imagem 45
3.6 Algoritmo binarização (preto/branco) aplicado sobre uma imagem 46
3.7 Algoritmo do vizinho mais próximo. 47
3.8 Processo de Ampliação com zoom de 2 vezes 47
3.9 Sentido do processamento do algoritmo de ampliação. 48
3.10 Algoritmo ampliação aplicado sobre uma imagem 48

4.1 Algoritmo escala de cinza : tempo x tamanho da imagem 52
4.2 Algoritmo brilho : tempo x tamanho da imagem 53
4.3 Algoritmo binarização : tempo x tamanho da imagem 54
4.4 Algoritmo ampliação com fator igual 2x: tempo x tamanho da imagem (100 amostras) 55
4.5 Algoritmo Binarização : Tempo x Tamanho da imagem 58
4.6 Imagem 640x480 : tempo x algoritmo 59
4.7 Fluxo de Execução do xLupa Embarcado. 60

Lista de Tabelas

2.1 Exemplos de GPPs e SPPs 8
2.2 Dispositivos embarcados que contêm a plataforma OMAP 3 9
2.3 ARM e Intel: Desempenho(DMIPS) e Consumo de Energia [2, 3, 4] 11
2.4 Dispositivos embarcados que utilizam o processador ARM Cortex-A8 16
2.5 Desempenho ARM x DSP 21
2.6 Recursos do Processador TMS320C64x+ 21
2.7 Unidades funcionais e operações executadas 23
2.8 Instruções e unidades funcionais em que podem ser executadas 23
2.9 Latência e Throughput no DSP c64x+ 26
2.10 Exemplos de instruções parapelas no processador DSP c64x+ 28
2.11 Feedback do compilador para três implementações diferentes de um mesmo algoritmo 33

4.1 Algoritmo escala de cinza: média e desvião padrão por imagem (100 amostras) 52
4.2 Algoritmo brilho: tempo de execução (100 amostras) 53
4.3 Algoritmo binarização: média e desvião padrão por imagem (100 amostras) . . 54
4.4 Algoritmo ampliação (fator=2): média e desvião padrão por imagem (100 amostras) 55
4.5 Tempo teórico: *Feedback* do compilador do DSP 57
4.6 Tempo Processamento (640x480): Captura e Renderização (100 amostras) . . . 57
4.7 Algoritmo da aplicação: quatro abordagens 61

Lista de Abreviaturas e Siglas

DSP	Digital Signal Processor
GPP	General Purpose Processor
SPP	Specific Purpose Processor
MAC	Multiply-and-Accumulate
c64x+	TMS320C64x+
ILP	Instruction Level Parallelism
FIR	Finite Impulse Response
SoC	System-on-Chip
MPSoC	Multiprocessor System-on-Chip
SE	Sistema Embarcado
PC	Personal Computer
TI	Texas Instruments
CPU	Central Processing Unit
MIPS	Million Instruction Per Second
SIMD	Simple Instruction Multiple Data
FFT	Fast Fourier Transform
RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computing
ULA	Unidade Lógica e Aritmética
MMU	Memory Management Unit
DP	Instruction Dispatch
DC	Instruction Decode
USB	Universal Serial Bus
V4L2	Video For Linux 2
SO	Sitema Operacional
LCD	Liquid Crystal Display
RAM	Random-Access Memory
ROM	Read-Only Memory

Resumo

A evolução da tecnologia de fabricação dos circuitos integrados permitiram que dois ou mais processadores fossem colocados em um único chip. Dispositivos embarcados utilizam desta tecnologia, MPSoC(multiprocessor system-on-chip), para colocar em um único chip um ou dois processadores de propósito geral(GPPs) e alguns processadores de propósito específico(SPPs). O uso destes processadores heterogêneos permitem que os requisitos dos dispositivos embarcados, como poder de processamento com baixo consumo de energia, sejam satisfeitos.

Apesar dos MPSoCs estarem presentes em diversos dispositivos, os processadores de propósito específico muitas vezes não são utilizados, pois o projeto, desenvolvimento e testes de software em tais arquiteturas, utilizando processadores heterogêneos, é diferente do desenvolvimento utilizando unicamente um processador de propósito geral.

Neste trabalho, foi avaliado a viabilidade da utilização de dois processadores heterogêneos, o ARM Cortex-a8, de propósito geral, e o DSP TMS320C64x+, para processamento de sinais digitais, presentes no MPSoC OMAP3530. Para tanto, foi utilizado a placa de desenvolvimento embarcado *BeagleBoard* onde foi instalado o Ubuntu 10.10. Um ampliador digital foi desenvolvido com o objetivo de avaliar o ganho de desempenho utilizando o processador DSP.

Foram implementados quatro algoritmos diferentes de tratamento de imagem. O tempo de execução de cada algoritmo no DSP em relação ao ARM variou de 50% a 200%. Há ainda outras partes da aplicação que não podem ser executados no DSP como é o caso da operações de entrada e saída. No ampliador como um todo, o uso do DSP levou a uma redução de tempo de processamento de 13.7% e diminuiu a ocupação do processador ARM em 23.6%. Em outro cenário, onde o ARM era utilizado para I/O e o DSP para a execução dos algoritmos, a redução no tempo de execução foi de apenas 6.1%, porém liberou a carga de processamento do ARM em 37.6%.

Palavras-chave: Processadores heterogêneos, Processador DSP, MPSoC, OMAP3530, BeagleBoard.

Capítulo 1

Introdução

1.1 Contextualização

Os sistemas embarcados (SEs) vem se tornando um elemento comum no cotidiano das pessoas, estando presentes em diversos equipamentos como micro-ondas, automóveis, celulares e televisores. Eles possuem vários componentes presentes em um computador como processador, memória, dispositivo de armazenamento e interfaces, porém, eles são projetados para executar, geralmente, tarefas específicas. Seu projeto, diferente dos computadores *desktop* tradicionais, possuem requisitos particulares como desempenho, potência, custo de produção, *design*, tolerância à falhas e tempo de projeto [5].

Em relação ao desempenho, diferentes sistemas embarcados podem necessitar diferentes taxas de processamento. Aplicações de tempo real, como aplicações multimídia, necessitam de um alto desempenho para processar em um curto espaço de tempo uma grande quantidade de dados. Este alto poder de processamento deve estar alinhado ao baixo consumo de energia. Isso não acontece, por exemplo, em sistemas orientados a entrada e saída, como é o caso dos editores de texto, onde eventualmente o processador é ocupado.

Sistemas embarcados possuem cada vez computações mais complexas, requerendo maior poder de processamento. Práticas que exploram o paralelismo no nível de instruções (ILP), como o pipeline e o aumento das unidades funcionais, tornaram-se muito comuns para resolver a demanda por processamento. Porém, atualmente estas técnicas não têm levado a um ganho significativo de desempenho, pois este nível de paralelismo é limitado pelo tamanho de um bloco básico de instruções[1, 6].

A Figura 1.1 mostra esse limite através da quantidade de instruções emitidas por ciclo do

relógio considerando um processador perfeito com número infinito de registradores, onde não há dependência de controle e nem dependência verdadeira de dados. Se um processador teoricamente perfeito possui um limite para a quantidade de instruções emitidas por *clock*, um processador real é ainda mais limitado.

Os dados da figura 1.1 estão distantes da maioria dos processadores, onde a emissão de duas instruções por *clock* é comum. Por outro lado, processadores vetoriais podem alcançar números melhores que os processadores tradicionais, ainda assim menores que os valores mostrados na Figura 1.1. Portanto, há um limite claro para o paralelismo de instruções, sendo o uso de processadores heterogêneos uma saída para se obter ganho de desempenho.

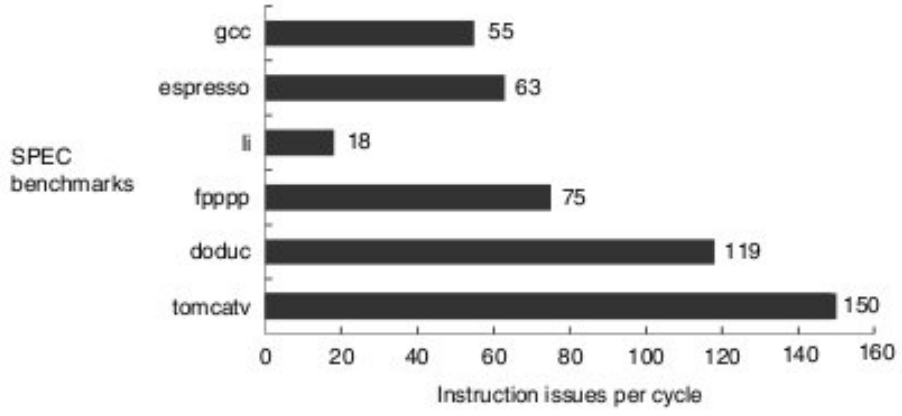


Figura 1.1: ILP em um processador perfeito para seis dos *benchmarks* SPEC92 [1]

Sistemas de computadores com processadores heterogêneos tem sido propostos como uma maneira de reduzir o consumo de energia e aumentar o desempenho em tarefas específicas. Um sistema de computador com processamento heterogêneo é composto, normalmente, por um processador de propósito geral, onde o sistema operacional é executado, e um ou mais processadores de propósito específico como processadores gráficos e processadores de sinais digitais. Processadores de propósito específico possuem vantagens em desempenho e consumo de energia quando comparado aos de propósito geral, pois a arquitetura e organização são melhores utilizados para tarefas específicas, realizadas algumas vezes diretamente pelo hardware ao invés do software ou microcódigo.

1.2 Problemas

Diversas pesquisas[7, 8, 9, 10, 11] têm sido realizadas recentemente com o objetivo de utilizar com maior eficiência processadores heterogêneos. Porém, utilizar processadores heterogêneos de forma eficiente é normalmente complicado e ainda não há uma forma consolidada de geração automática de código que utilize de forma eficaz todos eles. Desta forma, muitas aplicações não utilizam todos os recursos do hardware onde é executada.

Desenvolvedores de software foram treinados durante décadas a pensar em modo sequencial. A natureza dos MPSoCs faz necessário um modelo de programação paralela onde *deadlock*, *starvation*, condição de corrida e incoerência de dados podem ocorrer. Apesar do cuidado na hora do projeto e da codificação para evitar estes acontecimentos, problemas sempre irão ocorrer e será preciso depurar o programa. Se depurar *threads* em um simples processador já é uma tarefa complexa, depurar *threads* em um ambiente multiprocessado é ainda pior [12].

Com todas estas dificuldades é inevitável o aumento da equipe em projetos que envolvam ambientes heterogêneos, o que consequentemente acarretará aumento nos custos. No entanto, se os benefícios forem visíveis os clientes estarão dispostos a pagar por ele. Por outro lado, se o benefício em desempenho não for satisfatório para que valha a pena o investimento, provavelmente todo o potencial do hardware, que foi pago, não será explorado.

Na área de processamento de sinais digitais, por exemplo, menos de dois terços dos algoritmos de processamento de sinais digitais são executados por processadores DSP especializados neste tipo de processamento. Isto está relacionado à falta de ferramentas automáticas e eficazes que façam de modo transparente a divisão de processamento entre o processador principal e os processadores DSPs dedicados[13].

1.3 Motivações

Com a possibilidade de produzir um único chip com vários processadores, memória e periféricos, através da tecnologia *(System-on-Chip)*, cada vez mais processadores de propósito geral são apoiados por processadores de propósito específico, em especial na indústria de entretenimento com equipamentos como *smarphones* e *tablets*, onde uma grande quantidade de processamento em tempo real é exigido. Esses equipamentos vem se tornando parte do cotidi-

ano das pessoas, porém os diversos processadores heterogêneos normalmente não são utilizados pelos motivos colocados acima.

Diferentes trabalhos analisam o desempenho dos processadores de propósito geral e processadores de propósito específico com base em *benchmarks*. Seus resultados, avaliações e conclusões são focadas, muitas vezes, apenas nos processadores, individualmente, e não num contexto geral que envolva dois ou mais processadores, o sistema operacional e a comunicação entre eles.

1.4 Objetivos

Este trabalho tem como objetivo avaliar a viabilidade na utilização de processadores de propósito geral junto com processadores de propósito específicos em arquiteturas heterogêneas. Para isto será utilizado o MPSoC OMAP3530 que contém um processador de propósito geral ARM Cortexv8, e um processador de propósito específico DSP TMS320C64x+, especializado em processamento de sinais digitais .

Será desenvolvida uma aplicação completa, um ampliador digital, onde serão analisados os tempos de processamento de um passo completo no laço de *refresh* do programa, que envolve a captura de imagem da *webcam*; a aplicação de até quatro algoritmos de tratamento de imagem: ampliação, binarização, escala de cinza e aumento de brilho; e a renderização da imagem tratada.

1.5 Organização do Texto

Neste capítulo foram apresentados os motivos para a utilização de processadores heterogêneos, os problemas advindos da utilização deste novo ambiente, motivações e objetivos deste trabalho.

No capítulo 2 são apresentados os dois processadores heterogêneos ARM e DSP, presentes no MPSoC *OMAP3530*, utilizado neste trabalho, bem como a comunicação entre eles.

No capítulo 3, após o estudo inicial dos dois processadores utilizados, são apresentados os componentes de software e hardware utilizados no desenvolvimento da aplicação, bem como os algoritmos implementados.

No capítulo 4, são apresentados os resultados dos tempos obtidos na execução do ampliador digital. Ele está dividido em uma parte que trata individualmente do desempenho de cada algoritmo descrito no capítulo 3, uma análise do tempo teórico fornecido pelo DSP, e outra que trata dos tempos totais da execução do software.

Por fim, no capítulo 5, como finalização deste estudo, serão apresentados as conclusões e análises a respeito dos resultados obtidos no capítulo 4 e propostas de trabalhos futuros.

Capítulo 2

MPSoC Heterogêneos

2.1 MPSoC em Aplicações Embarcadas

Antes de iniciar o capítulo é necessário definir o que é um sistema embarcado. Um sistema embarcado (SE), segundo [14], pode ser definido como sendo qualquer dispositivo que tenha um computador programável, mas que não tenha como objetivo ser um computador de propósito geral. *Smartphones*, *tablets*, *MP3 players*, semáforos e microondas são exemplos de equipamentos que são ou que utilizam sistemas embarcados. Eles são projetados para realizarem tarefas predefinidas, com recursos específicos. Deste modo, sua engenharia pode otimizar o projeto reduzindo o tamanho, diminuindo o custo do produto e melhorando o desempenho para o grupo de tarefas que ele deve executar.

Com a crescente inovação na área dos SEs, seus hardwares estão ficando cada vez mais complexos, possuindo diversos componentes como amplificadores, controlador LCD, conversor analógico-digital, lógica de entrada e saída, memória RAM, memória ROM e vários níveis de memória *cache*. Há algum tempo atrás esses diversos elementos deveriam ser implementados em *chips* separados e soldados em uma placa de circuito impresso. A evolução da tecnologia de fabricação dos circuitos integrados permitiu que diversos estes diversos componentes fossem colocados em um único chip. A essa tecnologia dá-se o nome de *system-on-chip* (SoC). Entre as diversas vantagens dos SoCs, estão [15]:

- Aumento da velocidade de operação do sistema, uma vez que na integração *"on chip"* o fluxo de dados entre o processador e outros componentes também *"on chip"* pode ser maximizado.

- Significativa redução na potência consumida, graças às menores tensões requeridas devido ao alto grau de integração.
- Redução do tamanho e da complexidade dos produtos para os usuários finais, uma vez que o número de componentes adicionais diminui drasticamente.
- Substituição das trilhas de uma placa de circuito impresso pelas conexões internas em um único chip, aumentando extremamente a confiabilidade do sistema.
- Todas estas vantagens, dentre outras, possibilitam obter dispositivos cada vez mais baratos, mais eficientes, confiáveis e menores.

Infelizmente os projetos de SoCs estão ficando cada vez mais complexos e não estão acompanhando a lei de Moore, co-fundador da Intel, que enuncia que a capacidade de integração dos transistores em um único chip dobraria a cada 18 meses[15]. Além disso, os sistemas modernos necessitam integração e comunicação entre os diversos sistemas embarcados. Para atender a demanda por desempenho e as restrições de tempo para a colocação do produto no mercado, *time-to-market*, os projetistas dos SoCs tem buscado novos métodos de projeto no nível de sistema, que possibilitem o desenvolvimento concorrente de hardware e software de forma eficaz [16].

Uma nova tecnologia SoC, usada atualmente, consiste no uso de múltiplos processadores em um único circuito integrado (MPSoC, do inglês *Multiprocessor System-on-Chip*). A tecnologia MPSoC não consiste em apenas inserir diversos processadores em um único núcleo como acontece, por exemplo, nos conhecidos processadores *dual core*. Ao invés disso os processadores são heterogêneos, otimizados para aplicações específicas, onde blocos computacionais desnecessários são removidos, o que acaba gerando uma economia de energia e diminuindo a área do circuito.

2.2 Processadores Heterogêneos : GPPs x SPPs

O aumento na demanda por processamento, impulsionados principalmente pelas aplicações multimídias, alinhado ao requisito de baixo consumo de energia, tem levado a utilização de plataformas MPSoC nos sistemas embarcados. Nelas, um ou dois processadores de propósito

geral (GPP, do inglês *Gernal Purpose Processor*) são combinados com alguns processadores de propósito específico (SPP, do inglês *Specific Purpose Processor*). Enquanto os GPPs possuem alto grau de flexibilidade, sendo portanto, capazes de realizar qualquer tipo de computação, os SPPs tratam de tarefas específicas como processamento de áudio e vídeo.

Um exemplo muito comum são os telefones celulares e *smartphones*. Eles podem ter de 4 a 8 processadores heterogêneos embarcados, incluindo um ou dois processadores de propósito geral RISC para a interface com o usuário, processamento da pilha de protocolos e controle de outras funções, e outros processadores de propósito específico como processador DSP (*Digital Signal Processor*) para codificação e decodificação multimídia e acelerador de vídeo para renderização 3D.

A Tabela 2.1 mostra alguns exemplos das duas classes de processadores. Dentre os de propósito geral (GPP) estão os processadores ARM, presentes nos dispositivos embarcados, e os x86, dominantes no mercado dos computadores pessoais (PCs). Já entre os de propósito específico (SPP) tem-se o DSP para processamento de sinais digitais e o POWERVR para aceleração gráfica, ambos presentes em equipamentos embarcados.

Tabela 2.1: Exemplos de GPPs e SPPs

GPP	**SPP**
MIPS	IVA-HD video accelerator
ARM	POWERVR SGX544-MPx 3D Graphics
x86	C64x DSP
SPARC	TI Graphics 2D

Os SPPs possuem instruções complexas no domínio de sua aplicação. As unidades de processamento gráficas (GPUs), por exemplo, realizam operações sobre vértices e arestas presentes na renderização de imagens 3D. Já os processadores DSPs, utilizados em aplicações multimídias, contém instruções SIMD(*Single Instruction Multiple Data*), que operam em paralelo sobre um conjunto de dados, bem como instruções que aumentam o desempenho no processamento de sinais digitais como soma-e-acumula(MAC), saturação e média, dentre outras. Esta classe de processadores explora, principalmente, o alto grau de paralelismo no nível de dados. Em algumas operações sobre imagens, por exemplo, os *pixels* são independentes uns dos outros e é possível operar sobre um conjunto deles ao mesmo tempo.

O conjunto de instruções, que caracteriza o processador do ponto de vista do programador, são apoiados pelo hardware e pela arquitetura nos SPPs. Parte do processamento implementado em software nos GPPs é executado diretamente em hardware, o que permite aumentar o desempenho e diminuir o consumo de energia. Além disto, devido ao maior número de ULAs e a capacidade de utilização de todas em paralelo, os SPPs alcançam, em geral, um desempenho aritmético maior que os GPPs.

Os GPPs, ao contrário dos SPPs, são processadores com alto grau de flexibilidade, capazes de realizar qualquer tipo de computação. Sua arquitetura geralmente segue a arquitetura de Von Neumann onde há uma memória que armazena tanto instruções quanto dados, e um processador que busca as instruções e opera sobre os dados da memória. Como o tipo de processamento neles é bastante variado, eles fazem uso de recursos dinâmicos como cache, predição de desvio e execução superescalar dinâmica para aumento de desempenho.

2.3 MPSoC OMAP3530

O OMAP3530 (*Open Media Application Plataform* é um MPSoC destinado a aplicações multimídias móveis e portáveis desenvolvida pela Texas Instruments (TI). Ele, bem como outros MPSoCs da mesma plataforma, a OMAP 3, são usadas em diversos dispositivos embarcados, como mostra a Tabela 2.2.

Tabela 2.2: Dispositivos embarcados que contêm a plataforma OMAP 3

Dispositivo	Tipo	MPSoC
Touch Book	netbook	OMAP 3530
Pandora	video game portátil	OMAP 3530
DevKit8000	Kit de avaliação	OMAP 3530
BeagleBoard	Kit de avaliação	OMAP 3530
Motorola Milestone	Smartphone	OMAP 3430
Nokia N9000	Smartphone	OMAP 3430
Samsung i8910	Smartphone	OMAP 3430
Galaxy S	Smartphone	OMAP 3630
Droid 2	Smartphone	OMAP 3630
Milestone 2	Smartphone	OMAP 3630

Ele possui área de aproximadamente 6 centímetros quadrados (Figura 2.1) e contém, além de outros componentes, um processador de propósito geral ARM CortexA8, com frequência

Figura 2.1: Processador OMAP3530. Dimensões: 63.5 x 35 x 6.4 mm

de 720Mhz e um processador para processamento de sinais digitais, DSP C64x+, que opera a frequência de 520Mhz, como pode ser visto na Figura 2.2. Esses dois processadores serão abordados com mais detalhes nas próximas seções.

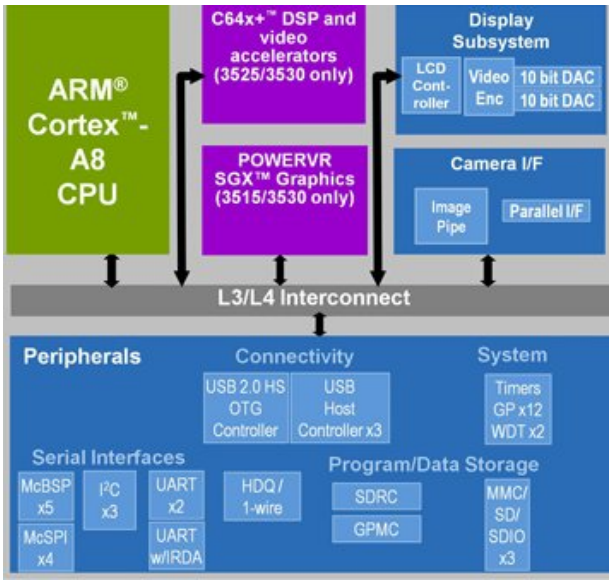


Figura 2.2: Diagrama de blocos do processador OMAP 3530

2.4 Processadores de Propósito Geral : ARM

ARM (*Advanced Risc Machine*) é uma arquitetura RISC (*Reduced Instruction Set Computer*) de 32 bits, com baixo consumo de energia, desenvolvida pela empresa britânica *ARM Holdings*, usada na grande maioria dos sistemas embarcados atuais, com bilhões em vendas nos últimos anos. Ela possui os direitos autorais sobre a arquitetura e a licencia para outras companhias que desejam produzi-los, ficando sob a responsabilidade da ARM Holdings e evolução e desenvolvimento de novas arquiteturas.

As licenças são de dois tipos: de implementação e de arquitetura. A licença de implementação provê completa informação requerida para o projeto e produção de circuitos integrados contendo o processador ARM. Já a licença de arquitetura licencia o desenvolvimento do próprio processador compatível com o conjunto de instruções (ISA) ARM.

Diversos fatores tornaram o processador ARM diferente de outros processadores de propósito geral. Com seu núcleo RISC e arquitetura simples ele pode ser fabricando usando um menor número de transistores e consequentemente possui uma menor área e um menor custo. Com um simples e poderoso pipeline ele possui a mesma performance que outros processadores de propósito geral com um consumo de energia consideravelmente menor. Ele também é altamente modular. Seu núcleo contém apenas um pipeline com número inteiros, sendo possível incluir outros componentes como MMU, unidade de ponto flutuante e outros coprocessadores. Todos estes fatores destacam o processador ARM no meio dos dispositivos embarcados.

A relação entre desempenho e potência dissipada entre processadores ARM, para sistemas embarcados, e processadores Intel, para sistemas *desktop*, pode ser visto na Figura 2.3. O *benchmark Dhrystone* utilizado para medir o desempenho, contém somente operações sobre números inteiros, sendo a medida DMIPS (*Dhrystone Million Instruction per Second*) a taxa de instruções por segundo, em milhões, do *benchmark* em questão. Com a mesma frequência e aproximadamente o mesmo desempenho, os processadores para embarcados consomem menos da milésima parte da energia que os processadores x86 da Intel.

Tabela 2.3: ARM e Intel: Desempenho(DMIPS) e Consumo de Energia [2, 3, 4]

Processador	**Frequência**	**Desempenho**	**Energia Dissipada**
ARM Cortex A8	1Ghz	2000 DMIPS	300mW
ARM Cortex A9	2Ghz	5000 DMIPS/core	250mW
Intel Pentium 3	1Ghz	1000 DMIPS	35W
Intel Core 2 Duo	2Ghz	5379 DMIPS/core	65W

Sendo a arquitetura ARM licenciável, os processadores ARM são produzidos por outras empresas além da ARM Holdings, o que permitiu uma rápida popularização da tecnologia. As empresas que fabricam e utilizam os processadores ARM em seus dispositivos não precisam pesquisar ou evoluir a arquitetura do processador. Elas apenas pagam uma taxa para poder usar uma arquitetura madura e já desenvolvida. Algumas das empresas que têm ou já tiveram uma

licença incluem: Atmel, Broadcom, Freescale, Intel, LG, Marvell Technology Group, NEC, NVIDIA, NXP (antiga Philips), Samsung, Sharp, Texas Instruments e Yamaha [13]. Algumas destas desenvolvem soluções próprias, integrando o processador ARM com processadores de propósitos específicos, controladores auxiliares, dentre outros recursos em um único chip.

A Figura 2.3, a título de exemplo, ilustra os blocos do núcleo do processador OMAP5430, fabricado pela Texas Instruments. Com o uso da tecnologia SoC(System-on-Chip) que permite diversos componentes eletrônicos em um único chip, ele possui 4 processadores ARM, um processador de sinais digitais, um processador gráfico e um processador de áudio, que são ativados quando necessário, dentre outros componentes.

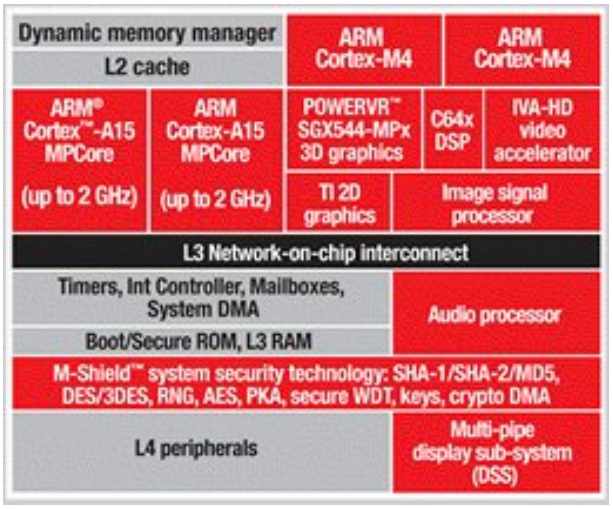


Figura 2.3: Diagrama de blocos do processador OMAP 5430

Nesta última década mais de dois bilhões de processadores ARM foram vendidos[17]. Eles estão nos mais variados dispositivos como PDA's, smartphones, telefones celulares, calculadoras, roteadores, dentre outros. Em 2009 a arquitetura ARM estava presente em 90% de todos os sistemas embarcados RISC [17].

2.4.1 ARM como arquitetura RISC

Até o início dos anos 80, a tecnologia colocava diversas restrições no desenvolvimento das arquiteturas de processadores. A memória de alto custo e baixo desempenho, por exemplo, forçavam a arquitetura a reduzir o tamanho dos programas e a quantidade de acessos à memória principal. Como a maioria dos programas eram escritos em linguagem de máquina, um conjunto de instruções que permitisse aos programadores serem criativos e eficientes, e ao mesmo

tempo facilitar a escrita de compiladores, era necessário. Estas e outras limitações estiveram presentes em aproximadamente meio século, desde o surgimento dos computadores, e levaram a criação de uma arquitetura com um conjunto complexo de instruções que exigia muito mais do hardware do que do software, pois todas as instruções complexas deveriam ser interpretadas por micro-instruções [18]. Essa arquitetura é conhecida como arquitetura CISC. Atualmente, os processadores CISC possuem um núcleo RISC, onde instruções complexas são traduzidas para instruções simples [18].

Com o avanço tecnológico e a evolução dos compiladores, as principais deficiências que levaram a criação da arquitetura CISC deixavam de existir. As memórias se tornaram maiores e mais baratas e os compiladores se tornavam disponíveis em diversas famílias de processadores. Isto permitiu o surgimento das arquiteturas RISC. A principal filosofia das arquiteturas RISC era tornar as instruções simples para explorar o paralelismo no nível de instruções.

Instruções simples tornam possível a sobreposição das instruções em execução, que não é possível com instruções complexas. O uso de pipeline permite que instruções sejam executadas a taxa de quase uma instrução por ciclo de relógio. Na verdade, a latência ou o tempo de execução real da instrução é de vários ciclos, mas após um pipeline cheio, uma instrução é terminada a cada ciclo. Além disso, instruções mais simples levam a circuitos mais simples e consequentemente a um menor tempo na estabilização dos sinais elétricos. Desta forma, é possível aumentar a frequência em que o processador pode operar.

Para permitir que fosse possível buscar instruções futuras antes da decodificação da instrução atual, uma arquitetura RISC possui instruções de tamanho fixo. Isso contrasta com as instruções CISC que tem tamanho variável. O compilador ou programador sintetiza uma complicada operação (como a divisão) dividindo-a em diversas instruções simples.

Outra característica das instruções RISC é que há somente duas instruções para acesso à memória: LOAD e STORE. Isto faz com que o programador ou o compilador tenha que trazer os dados da memória para os registradores, de forma explícita, antes de operar sobre eles. Esta política visa diminuir os acessos a memória, incentivando o programador ou compilador a usar um operando a máxima quantidade de vezes possível antes de gravar o resultado na memória e descartá-lo. Para que isto seja possível, processadores RISC contém mais registradores de propósito geral, normalmente 32, contra 4 nos processadores x86 com arquitetura CISC.

2.4.2 Particularidades da Arquitetura ARM

A arquitetura ARM possui muitas características da arquitetura RISC, porém ela não é inteiramente RISC. Quando os primeiros computadores RISC surgiram, seus objetivos eram diminuir o tempo de processamento geral através do aumento da frequência do processador e do uso de pipeline. Questões como consumo de energia, tamanho do processador e baixo custo de produção não eram requisitos importantes na época, apesar da arquitetura RISC contribuir em alguns deles por causa de sua simplicidade.

Os processadores ARM são destinados a dispositivos embarcados e seus requisitos levam em conta outros requisitos como os citados acima. Um dispositivo embarcado deve consumir a menor quantidade de energia possível pois a duração de sua bateria é um fator crítico. O tamanho do dispositivo e o seu custo final também são fatores importantes em um mercado tão competitivo.

Processadores RISC não levavam em consideração o tamanho do executável gerado pelo compilador. As memórias, na época, estavam cada vez maiores e mais baratas. Com os dispositivos embarcados esse requisito de memória muda, e se parece muito com a filosofia CISC onde havia pouco memória disponível. Além disso, algumas instruções não simples foram adicionadas as instruções ARM visando melhorar o desempenho em sistemas embarcados.

Tudo isto levou os processadores ARM a não adotarem uma arquitetura puramente RISC. Sloss, Symes e Wright [17] sumarizam as principais diferenças entre os processadores puramente RISC e processadores ARM:

- **Ciclos de execução variados para certas instruções** – Nem todas instruções ARM executam em um simples ciclo. A quantidade de registradores envolvidos em uma instrução e o tipo de acesso à memória, sequencial ou aleatória, contribuem na quantidade de ciclos de cada instrução.

- **Shift preprocessamento** – Antes de chegar a ULA, um operando pode ser modificado com operações de deslocamento, expandindo o operando antes dele ser usado. Isto melhora o desempenho e a densidade do código.

- **Conjunto de instruções Thumb de 16-bits** – É permitido executar um código com instruções de 16 ou 32 bits. Isso aumenta a densidade do código em aproximadamente 30%

quando comparado a instruções de tamanho fixo de 32 bits.

- **Execução condicional** - Uma instrução pode ser executada quando uma condição especifica é satisfeita. Isso evita instruções de desvio explícitas, melhorando o desempenho e a densidade do código.

- **Instruções específicas** – Foram adicionadas algumas instruções para processamento de sinais digitais como a multiplicação de inteiros de 16 bits com saturação. Isto permite maior desempenho, principalmente em processamento multimídia.

- **Load/Store Múltiplos** - Instruções de acesso a memória podem operar sobre múltiplos registradores.

2.4.3 Coprocessadores

Uma característica que torna os processadores ARM ideais para sistemas embarcados é a possibilidade de estender o conjunto de instruções através da adição de coprocessadores. Os coprocessadores são processadores de propósito específico, destinados a ampliar as funcionalidades do processador ou melhorar o desempenho para determinados grupos de instruções. O núcleo ARM, por exemplo, não contém instruções para ponto flutuante, sendo necessária a emulação por software destas operações. Porém, é possível adicionar um coprocessador de ponto flutuante à arquitetura melhorando o desempenho em operações que envolvam números reais.

Essa possibilidade de expansão de funcionalidade dos processadores ARM torna a arquitetura flexível para grande parte dos sistemas embarcados. Um dispositivo que não necessite de cálculos em ponto flutuante pode usar um processador ARM sem a unidade de processamento de ponto flutuante. Isso diminui o custo final e o tamanho do processador. Por outro lado, um dispositivo que execute um grande número de operações em ponto flutuante pode incluir ao núcleo ARM um processador VFP que executa operações em ponto flutuante, com simples e dupla precisão, seguindo as normas da IEEE.

São definidos 16 coprocessadores na arquitetura ARM, CP0 à CP15. Suas instruções fazem parte do conjunto de instruções ARM, ou seja, todas as operações disponíveis nos coprocessadores são acessadas através de instruções assembler, assim como as instruções para o núcleo ARM. Isto contrasta com outros processadores que se encontraram no mesmo chip que o pro-

cessador ARM, como é o caso dos processadores OMAP que possuem juntos, num mesmo chip, um processador ARM Cortex-a8 e um processador DSP c64x+. Neles, cada programa é compilado para seu processador, com compiladores diferentes, e a comunicação entre eles é realizada através de um *device driver* do sistema operacional. Portanto, coprocessadores são mais fáceis de utilizar que outros processadores cujas instruções não fazem parte do conjunto de instruções ARM.

Como dito anteriormente, a ARM Holdings apenas licencia a arquitetura para a fabricação dos processadores ARM por terceiros. O fabricante então produz seus processadores ARM, incluindo mais ou menos coprocessadores ao núcleo ARM, conforme custos e nicho que pretende atuar. Por padrão, o compilador ARM não utiliza os coprocessadores para gerar o código de máquina, porém, é possível informar ao compilador quais os coprocessadores estão disponíveis para que ele faça uso deles quando possível.

2.4.4 Processador ARM Cortex-A8

O processador ARM Cortex-A8 é baseado na arquitetura ARMv7 e sua frequência pode variar de 600Mhz até 1Ghz, dependendo do modelo do processador. Seu consumo de energia é em torno de 300mW, resultando num consumo baixo de energia aliado a um alto desempenho. A Figura 2.4 mostra a estrutura do processador ARM Cortex-A8 e a Tabela 2.4 mostra alguns dispositivos que o utiliza.

Tabela 2.4: Dispositivos embarcados que utilizam o processador ARM Cortex-A8

Apple Iphone 3GS
Apple IPad
BeagleBoard
Motorola Milestoned
Samsung i8910
Nokia N900
Texas Instruments OMAP 3xxx
Samsung Galaxy S i9000
LG Optimus Black

Um dos elementos particulares da arquitetura ARM é o coprocessador NEON. Com instruções que operam sobre vetores de dados de 128 bits, ele acelera o processamento multimídia de

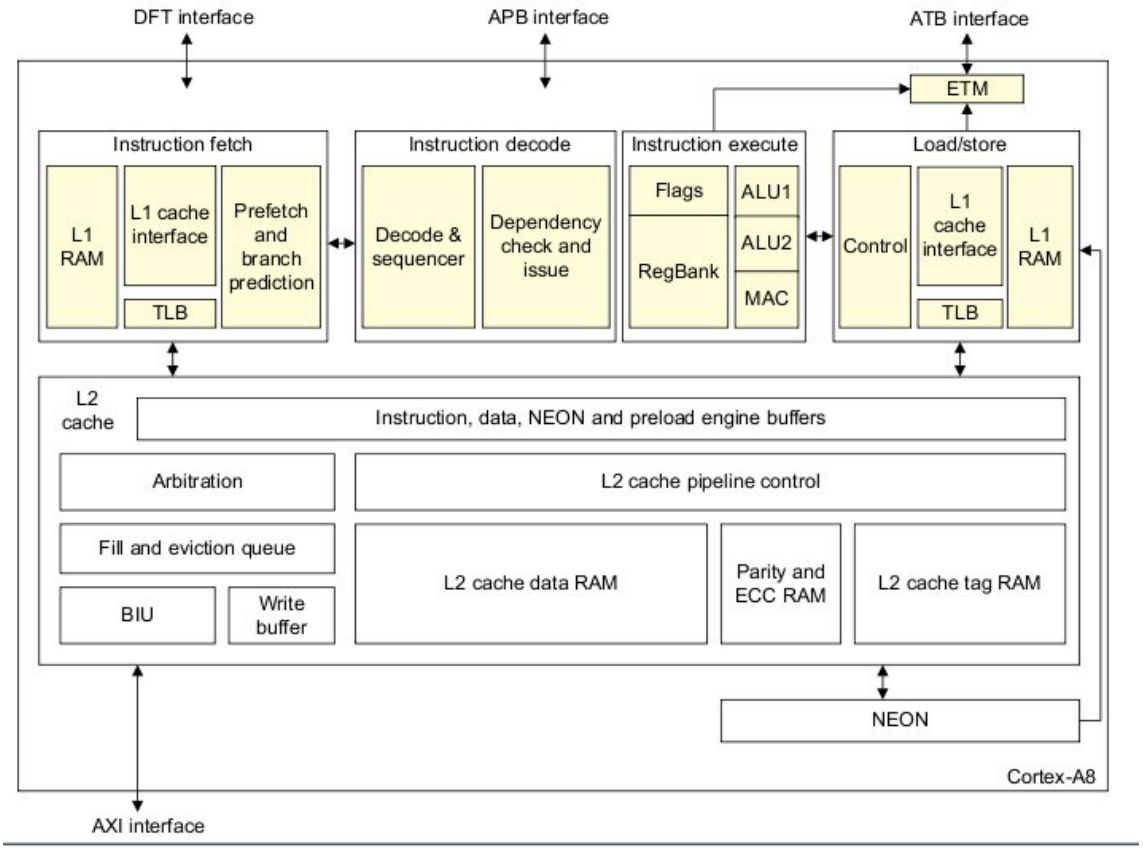


Figura 2.4: Diagrama de blocos do processador ARM Cortex-A8

áudio e de vídeo, entre outros. Ele tem seu próprio pipeline e registradores, que permite uma maior flexibilidade e alta performance para aplicações orientadas a processamento.

O Pipeline principal do ARM Cortex-A8 possui 3 estágios(Figura 2.5):

- Busca (3 ciclos)
- Decodificação(4 ciclos)
- Execução (6 ciclos)

O estágio de execução do pipeline possui três unidades funcionais, onde duas delas são unidades lógicas e aritméticas (ULAs). Isso diminui a ocorrência de protelação no pipeline por falta de recursos e permite a execução de dois *pipelines* simultaneamente (*superescalar*).

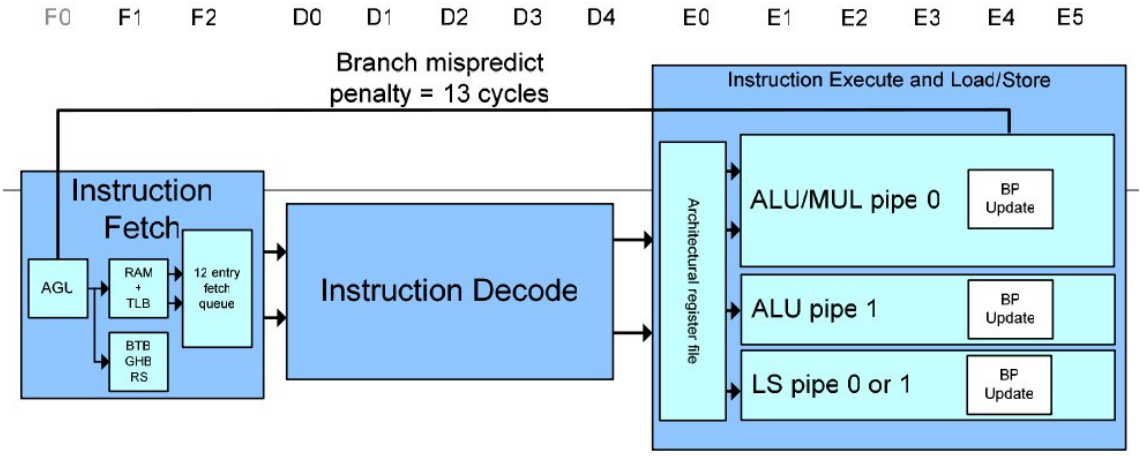


Figura 2.5: Pipeline Processador ARM Cortex-A8

2.5 Processadores de Propósito Específico : DSP

2.5.1 Processamento de Sinais Digitais

O mundo em que vivemos cada vez mais vem se tornando dependente de dados. São dados financeiros, dados médicos, notícias e um vasto conjunto de informações relacionadas à área de entretenimento. Uma parcela destes dados, seja áudio, vídeo ou texto, tem uma coisa em comum: precisam ser entregues e processados rapidamente. Um modem para acesso a internet precisa decodificar o sinal vindo da rede a uma taxa proporcional ao volume de dados sendo transmitido. Um vídeo que é executado em um *smartphone* precisa ser decodificado numa taxa proporcional a quantidade de *frames* por segundo.

No mundo real, a grande maioria dos dados costumam ser representados por sinais, variações físicas que podem ser mensuradas [19], como luz e temperatura. Sinais estes que são analógicos por natureza e caracterizados por serem contínuos ao longo do tempo. Para tratá-los computacionalmente é necessário representá-los em um formato digital.

Um sinal digital, nesse caso, nada mais é do que uma sequência discreta de estados que codifica uma mensagem. Uma imagem JPEG, um vídeo AVI e um pacote TCP/IP são alguns exemplos de sinais digitais. Em geral, funções de processamento digitais são operações matemáticas em sinais de tempo real, repetitivas e numericamente intensivas [20]. Para tratar essa classe de processamento, surgiram os processadores de sinais digitais.

2.5.2 Processadores de Sinais Digitais

Os Processadores de Sinais Digitais (DSP, do inglês *Digital Signal Processor*) são processadores especializados que tratam sinais das mais diversas naturezas (dados, vídeo, áudio, etc). Eles surgiram como dispositivos programáveis que tem vantagens em termos de desempenho, custo e consumo de energia [21, 22]. Em contraste com os processadores de propósito geral (GPP), eles são moldados pelos algoritmos DSP, cuja computação é de alguma maneira facilitada pelos recursos específicos presentes nos processadores DSP [21].

Operações como multiplicar-e-acumular (MAC) e operações SIMD(*Single Instruction Multiple Data*), que operam em paralelo em um conjunto de dados, são suportadas pelos processadores DSP. Um recurso útil para filtros com o FIR (*Finite Impulse Response*) e transformada de fourier (FFT), mostrados na Figura 2.6, bem como na multiplicação de matrizes.

$$y(n) = \sum_{i=0}^{M} c_i x(n-i) \qquad F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-j2\pi/N}$$

(a) Filtro FIR (b) Transformada Discreta Fourier

Figura 2.6: Filtro FIR e Transformada Discreta de Fourier

Uma vez que os processadores DSP são projetados para executar com eficiência algoritmos de processamento de sinais digitais, com recursos limitados, sua arquitetura é diferente das encontradas nos GPPs. Em suas versões mais antigas os processadores DSPs diferenciavam bastante dos processadores de propósito geral, com o uso de aritmética de ponto fixo ao invés de ponto flutuante e a arquitetura de *Harvard* ao invés da tradicional arquitetura de Von Neumann[22]. Tais diferenças deixaram de existir nas últimas gerações dos DSPs. Processadores DSPs agora implementam aritmética de ponto flutuante como os da geração TMS320C67xTM da Texas Instruments (TI), e as arquiteturas Harvard que possuem *caches* separadas para dados e instruções, agora são utilizados por muitos GPPs. Porém, a maioria dos processadores DSP ainda utilizam aritmética em ponto fixo, com o objetivo de diminuir a complexidade do hardware e consumir menos energia. Isso não implica em grandes danos, pois sinais digitais por terem uma amplitude bem definida podem ser discretizados sem ou com pouca perda de informações. Tais mudanças sobre estes processadores podem ser divididas em três gerações.

A primeira geração surgiu nos começo dos anos 80 com uma arquitetura Harvard que sepa-

rava as instruções dos dados na *cache*, e executavam somente computações em ponto-fixo. Na segunda geração, entre o final dos anos 80 e o início dos anos 90, recursos de pipeline, unidade aritmética lógica (ULA), unidade de ponto flutuante e acumuladores foram acrescentados a arquitetura para melhorar a performance. No que é considerado a terceira geração, o projeto dos DSPs passaram a incorporar muitas das funcionalidades dos GPPs. Recentemente, estão sendo lançados no mercado processadores DSPs que executam instruções SIMD, longas instruções (VLIW) e operações superescalares [21].

2.5.3 Processador DSP TMS320C64X+

O Processador TMS320C64x+, ou apenas c64x+, é um processador de alto desempenho para processamento de sinais digitais em ponto-fixo, fabricado pela Texas Instruments(TI). Pertence a geração TMS320C6000TM e possui uma arquitetura VLIW. Ele faz parte da família c64x que possuem um conjunto de instrução (ISA) compatíveis entre si[23].

Com um relógio de frequência de 520Mhz ele alcança o desempenho de até 4160 milhões de instruções por segundo (MIPS). Esta taxa de processamento é alcançada através de seus 64 registradores de 32 bits de propósito geral e oitos unidades funcionais. Com a máxima paralelização possível, cada unidade funcional irá executar uma das 8 instruções contidas numa instrução longa VLIW a cada ciclo de relógio, totalizando 8 instruções por ciclo. Com 520 milhões de ciclos por segundo, a expressiva quantia de 4160 MIPS é alcançada. Como os instruções SIMD podem executar até quatro operações em paralelo, então o limite teórico de processamento vai além das 4160 operações por segundo.

No geral, é muito difícil para o programador ou para o compilador paralelizar totalmente um código e utilizar todas as unidades funcionais em um único ciclo, devido as dependências entre os dados. Fazendo-se uma comparação com um processador de propósito geral que opera a 520 MHZ e que executa uma instrução por ciclo de CPU, é possível chegar a conclusão de que o processador DSP é oito vezes mais rápido que o processador GPP. No entanto, em aplicações reais estes ganhos muitas vezes não passam de quatro vezes. No processador c55x da Texas Instruments, que possui recursos semelhantes ao c64x+, os ganhos em desempenho são de 3 vezes em média, com mínimo de 1.1 vezes e máximo de 6 vezes, quando comparado a processadores RISC de propósito geral conforme mostra Tabela 2.5 [24]. A codificação de

áudio e vídeo em formato MPEG4 no padrão H263, por exemplo, consome 41 milhões de ciclos por segundo do processador DSP e no mínimo 153 milhões de ciclos por segundo entre os dois processadores RISC, o que dá uma relação de desempenho de mínima de 3.73 vezes em favor do DSP. Com isto, esta codificação ocupa muito menos o processador DSP do que o processador ARM, ou em outros palavras, ela é executada em menos tempo no processador DSP do que no processador ARM

Tabela 2.5: Desempenho ARM x DSP

	GPP		**DSP**	**Desempenho**
	ARM9E[1]	**StrongARM 1100**[1]	**TMS320C5510**[1]	**DSP/ARM**[2]
Cancelamento de Eco 16bits (32ms - 8Khz)	24	39	4	6x
Cancelamento de Eco 32bits (32ms - 8Khz)	37	41	15	2.46x
Decodificação MPEG4/H263 decodificação QCIF @ 15 fps	33	34	17	1.94x
‘ Codificação MPEG4/H263 QCIF @ 15 fps	179	153	41	3.73x
Decodificação JPEG (QCIF)	2.1	2.06	1.2	1.71x
Decodificação MP3	19	20	17	1.11x
Média de ciclos proporcional ao C5510TM	3.1	3	1	

[1]Unidade de medida: milhões de ciclos por segundo
[2]Desempenho proporcional do DSP em relação ao ARM

A Figura 2.7 mostra o fluxo de instruções no DSP c64x+ da Texas Instruments e a Tabela 2.6 os principais recursos dele.

Tabela 2.6: Recursos do Processador TMS320C64x+

Frequência Clock	700 Mhz
Instruções por segundo	até 5600 Milhões(MIPS)
Cache	L1: 256Kb L2: 640Kb
Registradores	64 de propósito geral
Unidades Funcionais	2 multiplicadores 6 ULA
Instrução buscadas por ciclos	8-14 instruções : 256 bytes
Operandos buscados por ciclo	4-8 operandos : 256 bytes
Aritmética	Ponto Fixo

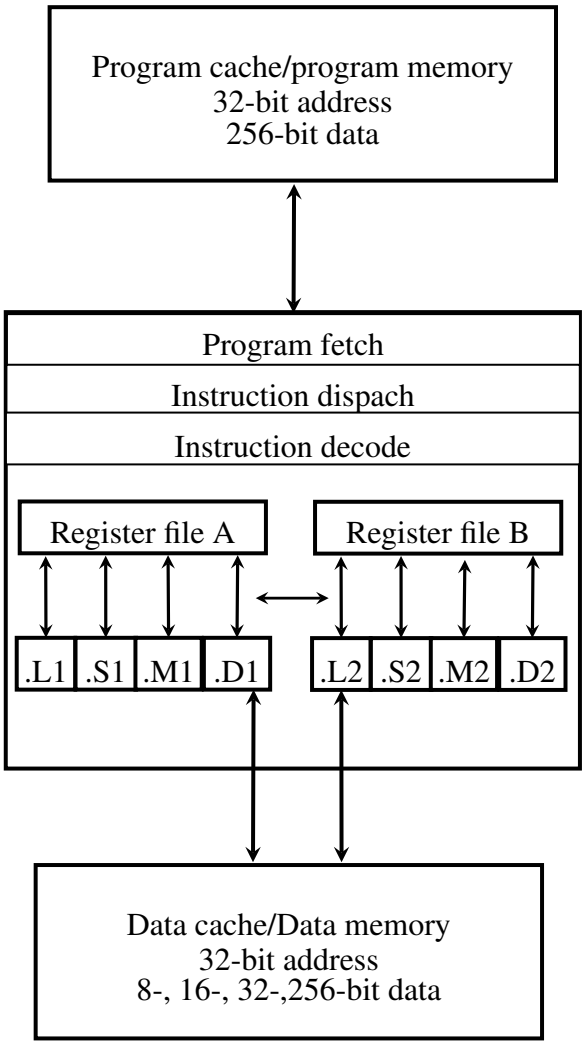


Figura 2.7: Fluxo de instruções no processador DSP c64x+

Unidades Funcionais

As oito unidades funcionais (L, S, M, D) do processador são dividas em dois caminhos de dados(*data path*) com quatro unidades cada (Figura 2.7). Cada caminho de dados tem o seu conjunto de 32 registradores sobre os quais operam diretamente, mas também é possível a comunicação com o outro caminho de dados com uma penalidade de alguns ciclos. A Tabela 2.7 mostras as principais operações executadas por cada unidade funcional. Além da duplicação de recursos que permite no mínimo duas operações do mesmo tipo operarem em paralelo, várias operações podem ser executadas em mais que uma unidade funcional. Uma operação de soma, por exemplo, pode ser executada nas unidades .L1(.L1 e .L2), .S(.S1 e .S2) e .D(.D1 e .D2). A Tabela 2.8 mostra outras operações e as unidades funcionais que podem executá-las.

Tabela 2.7: Unidades funcionais e operações executadas

Unidade Funcional	**Operações em ponto fixo**
Unidade .L (.L1 e .L2)	Operações aritméticas de 32 bits
	Operações lógicas de 32 bits
	Deslocamento de bytes
	Empacotamento e desempacotamento de dados
	Duas operações aritméticas de 16 bits
	Quatro operações artiméticas de 8 bits
	Duas operações mínimo/máximo de 16 bits
	Quatro operações mínimo/máximo de 8bits
Unidade .S (.S1 e .S2)	Operações aritméticas de 32 bits
	Operações lógicas de 32 bits
	Deslocamento de bytes
	Empacotamento e desempacotamento de dados
	Duas operações de comparação de 16 bits
	Quatro operações de comparação de 8 bits
	Duas operações de deslocamento de 16 bits
	Duas operações de deslocamento de 16 bits com saturação
	Quatro operações de deslocamento de 16 bits com saturação
Unidade .M (.M1 e .M2)	Operação de multiplicação 32 x 32 bits
	Duas Operações de multipliação 16 x 16 bits
	Quatro Operações de multipliação 16 x 16 bits
	Duas Operações de multipliação 16 x 16 bits com saturação
	Quatro Operações de multipliação 8 x 8 bits com saturação
	Operações de deslocamento
Unidade .D (.D1 e .D2)	Adição e Subtração de 32 bits linear e circular
	Load e Store de palavras com 64 bits alinhadas
	Load e Store de palavras com 64 bits desalinhadas
	Operações lógicas de 32 bits

Tabela 2.8: Instruções e unidades funcionais em que podem ser executadas

		Unidade Funcional			
Instrução	**Descrição**	**Unidade .L**	**Unidade .M**	**Unidade .S**	**Unidade .D**
ABS	Um valor absoluto 32 bits	✓			
ADD	Uma soma	✓		✓	✓
ADD2	Duas somas	✓		✓	✓
ADD4	Quatro somas	✓			
MPY2	Duas multiplicações		✓		
SADD	Duas somas c/ saturação		✓	✓	
SMPY2	Duas multiplicações c/ saturação		✓		
LDDW	Load de 128 bits			✓	

Formato e Busca das Instruções

O formato das instruções no DSP c64x+ são baseadas nas arquiteturas RISC e VLIW. Cada instrução tem um formato fixo de 32 bits, porém o compilador pode dividir esta instrução em

duas de 16 bits quando possível. Com a arquitetura VLIW, o compilador é responsável por rearranjar a ordem de execução das instruções e avisar ao processador quando elas podem ser executadas em paralelo, ao contrário do que acontece nas arquiteturas superescalares onde o processador gerencia a ordem das instruções e decide quando elas podem ou não serem executadas em paralelo, por causa das dependências. Com isto, há uma economia de hardware, tempo e também de consumo de energia, pois todo o trabalho realizado pelo hardware em arquiteturas superescalares é feito pelo compilador nas arquiteturas VLIW. Este trabalho é executado apenas uma vez na compilação e não possui restrição de tempo para otimizações.

As instruções são buscadas (*fetched*) na memória em pacotes de oito instruções por vez (ou até 14 se o compilador conseguir compactar as instruções para 16 bits). O formato das instruções é mostrado na Figura 2.8. O campo p-bit determina se uma instrução pode executar em paralelo com outras instruções. Os p-bits são lidos da direita para a esquerda. Se o p-bit de uma instrução I é 1, então a instrução I+1 pode ser executada em paralelo com a instrução I. Se o p-bit de uma instrução é zero, então a instrução I+1 é executada um ciclo depois da instrução I. Desta forma, até oito instruções podem ser executadas em paralelo, onde cada uma deve ocupar uma unidade funcional diferente.

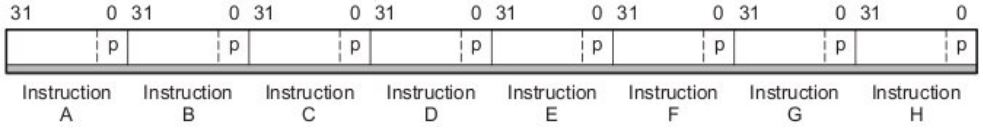


Figura 2.8: Formáto básico de um pacote de instruções

Pipeline

O pipeline no DSP c64x+ é dividido em três estágios [25]:

- Busca (4 ciclos)
- Decodificação (2 ciclos)
- Execução (5 ciclos)

No estágio de busca um pacote de oito palavras com instruções é buscada na memória. Ele possui quatro fases para todas as instruções: PG(*program address generate*), PS (*Program*

address send), PW (*Program access ready wait*) e PR (*Program fetch packet receive*). Durante a fase PG, o endereço de programa é gerado. Na fase PS, o endereço de programa é enviado para a memória. Na fase PW, a leitura de memória ocorre. Finalmente, na fase PR, o pacote buscado é recebido na CPU.

No estágio de decodificação o pacote de instruções é dividido em pacotes executáveis. Pacotes executáveis consistem de uma a oito instruções que podem ser executadas em paralelo. Ela possui duas fases: DP (*Instruction dispatch*) e DC (*Instruction decode*). Durante a fase DP, as instruções de um pacote de execução é atribuída às unidades funcionais. Durante a fase DC, os registradores são decodificados para a execução das instruções nas unidades funcionais.

O estágio de execução é dividida em cinco fases (E1-E5). Diferentes tipos de instruções requerem diferente números de fases para execução. Uma multiplicação de 16 bits requer duas fases. Uma instrução *store* requer três fases, enquanto uma instrução *load* requer cinco fases.

Portanto, uma instrução pode levar de 7 a 11 ciclos de CPU para ser executada. Esse é um limite mínimo imposto pela arquitetura. Instruções poderão levar mais ciclos devido a protelação(*stalls*), que podem ocorrer durante sua execução. Se um operando não se encontra na memória *cache*, ele deverá ser buscando da memória principal, que tem um tempo de acesso maior. Isso provoca uma espécie de bolha no pipeline, onde uma instrução não está fazendo um trabalho útil, mas está ocupando a CPU. Essa bolha retarda toda a execução das instruções que vem antes dela no pipeline e só some quando ela sair do pipeline. A Figura 2.9 mostra um pipeline cheio no DSP c64x+ onde não ocorrem bloqueios.

	Clock cycle												
Fetch packet	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5		
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3
n+5						PG	PS	PW	PR	DP	DC	E1	E2
n+6							PG	PS	PW	PR	DP	DC	E1
n+7								PG	PS	PW	PR	DP	DC
n+8									PG	PS	PW	PR	DP
n+9										PG	PS	PW	PR
n+10											PG	PS	PW

Figura 2.9: Pipeline cheio no c64x+

É possível notar que apesar de cada instrução levar no máximo 11 ciclos para ser executada, depois que a primeira instrução deixou o pipeline, cada nova instrução termina de ser executada

em apenas um ciclo de CPU. Isto leva a dois novos conceitos: Latência e *Throughput*.

Latência e Throughput

Latência é a quantidade de ciclos de CPU que uma instrução leva para ser executada, desde o momento em que ela entra no pipeline até o momento em que ela sai dele. Throughput é a quantidade de ciclos de CPU que uma segunda instrução leva para sair do pipeline após uma primeira instrução ser finalizada.

O *throughput* é uma medida muito mais importante que a latência. Num pipeline de instruções a primeira instrução consome um tempo de CPU igual sua latência. Após a primeira instrução deixar o pipeline, uma nova instrução deixa o pipeline a cada *h* ciclos de CPU, onde *h* é o valor do *throughput*.

No processador DSP c64x+, as etapas de busca (*fetch*) e decodificação (*decode*) consomem a mesma quantidade de ciclos de CPU para qualquer instrução, quatro e dois respectivamente. Já na etapa de execução (execute) a quantidade de ciclos vária conforma cada instrução. Os tempos de *throughput* e latência no processador c64x+ são mostradas na Tabela 2.9, supondo um pipeline cheio (sem *stalls*).

Tabela 2.9: Latência e Throughput no DSP c64x+

Instrução	Fetch	Decode	Execute	Latência	Throughput
Adição	4	2	1	7	1
Multiplicação	4	2	4	10	1
Store	4	2	1	7	1
Load	4	2	5	11	1
Divisão	4	2	18-42	24-38	-

Caches L1 e L2

Durante muito tempo a velocidade dos processadores cresceram muito mais rapidamente que a velocidade das memórias. Como consequência, atualmente existe uma lacuna entre o desempenho das CPUs e o desempenho das memórias. O ideal seria uma grande quantidade de memória junto com o chip do processador, mas o custo de tal memória tornaria o projeto inviável. Como solução a este impasse, a maioria dos sistemas computacionais usam dois ou

três níveis de memória *cache*. Elas são memórias pequenas e rápidas que atuam entre a CPU e a memória principal.

No DSP c64x+ há dois níveis de memória *cache*: L1 e L2. A *cache* L1 é dividida em L1P que contém instruções e L1D que contém dados, cada uma com 32Kb. Ela é acessada pela CPU sem bloqueio (*stalls*). Já o segundo nível de *cache*(L2) é configurável e pode ser configurada em memória não *cache* (acesso sequencial) e memória *cache* com até 256K bytes[26]. A Figura 2.10 mostra a arquitetura da memória no processador DSP c64x+.

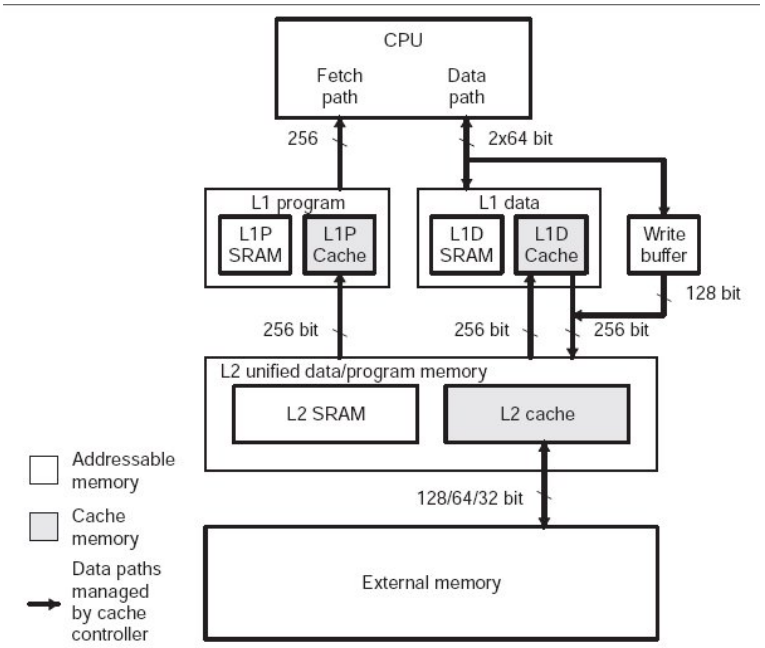


Figura 2.10: Arquitetura da memória cache no c64x+

As instruções são buscadas em pacotes de 256 *bits* na *cache* de instruções (L1P). Já os dados são buscados na *cache* de dados (L1D) em dois canais de 64 *bits*. Em caso de *miss cache* na L1 os dados ou instruções são acessados no segundo nível de *cache* e trazidos em blocos de 256 *bits*. Como dito anteriormente, a CPU consegue buscar um dado na *cache* L1 em apenas um ciclo. Se o dado não estiver no primeiro nível da *cache* então ocorre um bloqueio no pipeline e são necessários 5 ciclos de CPU para buscar o dado no segundo nível de *cache*. Caso o dado não esteja no segundo nível, então um acesso a memória principal é feito e o tempo de acesso vai depender da frequência que a memória primária opera. Quando menor for a frequência da memória principal em relação a frequência interna da CPU, mais ciclos serão necessários [27].

Instruções

Como citado anteriormente, os processadores DSP, em geral, possuem instruções que operam em paralelo sobre um conjunto de dados (SIMD) e instruções especiais muito comuns ao processamento de sinais digitais, como as operações multiplica-e-acumula (MAC).

As instruções SIMD operam sobre um conjunto de dados organizados em pacotes de 32 bits (inteiro) ou 64 bits (double). Cada pacote pode conter dois ou quatro dados. As instruções binárias, que necessitam de dois operadores, executam sobre dois destes pacotes. A Tabela 2.10 mostra algumas das operações suportadas pelo processadores DSP c64x+ e a Figura 2.11 mostra o resultado da execução da instrução SADDU4, que soma 4 inteiros de 1 byte com saturação. A instrução de saturação é executada direta em *hardware*, ao contrário das implementação em *software* que resultam, no mínimo, em uma instrução de comparação e uma de atribuição.

Tabela 2.10: Exemplos de instruções parapelas no processador DSP c64x+

Instrução	**Descrição**
uint _ saddu4(int src1, int src2);	Executa adição saturada entre pares de valores 8-bit sem sinal em src1 e src2.
double _ mpy2(int src1, int src2);	Retorna o produto dos valores baixo e altos de 16-bit em src1 e src2.
int _ subabs4 (int src1, int src2);	Calcula o valor absoluto da diferença para cada pacote de 8 bits.
uint _ avgu4(uint, uint);	Calcula a média para cada par de valores de 8-bit.
uint _ swap4 (uint src);	Troca pares de bytes dentro de cada valor de 16-bit.

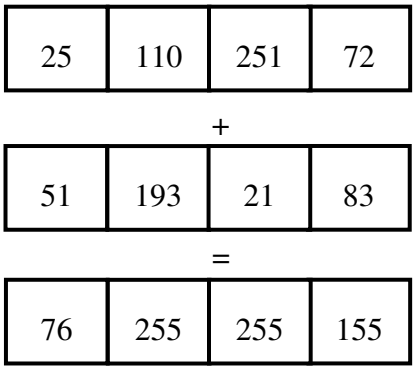


Figura 2.11: Instrução SADDU4 sobre dois pacotes de 32 bits

Operação de divisão

Uma vez que a operação de divisão não é comum no processamento de sinais digitais, o processador c64x+ não prove uma instrução simples implementada em hardware, devido a complexidade do hardware e consequentemente aos custos de tal implementação. Ao invés disso, ele a emula de modo iterativo em software. Como forma de otimizar o desempenho ele oferece a instrução auxiliar de divisão SUBC (Figura 2.12), implementada em hardware, que executa uma subtração condicional baseado nos valores do numerador e do denominador da divisão[28]. Isto acelera a execução da função de divisão (Figura 2.13) e diminui a quantidade total de ciclos consumidos por ela. Como resultado, o tempo mínimo de uma operação de divisão no processador DSP c64x+, é de 18 ciclos e o tempo máximo de 42 ciclos [28].

Algorithm 1 Instrução SUBC

```
IF (cond) {
      IF (src1 >= src2)
            dst = ((src1-src2)<<1)+1
      ELSE dst = src1<<1
}
ELSE nop
```

Figura 2.12: Instrução auxiliar de divisão SUBC

2.5.4 Programação do Processador DSP

Quando utilizamos um processador DSP desejamos obter o melhor desempenho possível, pois tempo é um fator crítico em aplicações embarcadas. É possível melhorar o desempenho através de uma instrução que executa uma operação SIMD e/ou executando várias instruções em paralelo, utilizando as várias unidades funcionais através de uma instrução VLIW. São basicamente três etapas básicas para criar um código para um processador DSP. A primeira etapa consiste em codificar um código C padrão, como os para processadores de propósito geral, identificando os loops mais importantes em termos de MIPS (milhões de instruções por segundo).

Na segunda fase busca-se otimizar o código para obter o desempenho desejado. Nela devemos repassar ao compilador o máximo de informações possíveis, como o número de iterações

Algorithm 2 Implementação da divisão em uma chamada C

```
unsigned int udiv(unsigned int num, unsigned int den)
{
        int i, shift;
        if (den > num) return (0);
        if (num == 0) return (0);
    if (den == 0) return (-1);
        shift = _lmbd(1, den) - _lmbd(1, num);
        den <<= shift;
        for (i=0; i<=shift; i++)
                num = _subc(num, den);
        return (num << (32-(shift+1))) >> (32-(shift+1));
}
```

Figura 2.13: Implementação da divisão em uma chamada C

mínimas e máximas de um laço, se os vetores de dados estão alinhados na memória ou se uma posição na memória pode ser endereçada por mais de um ponteiro. Com estas informações o compilador pode, por exemplo, desenrolar um laço (*loop*) e transformar quatro instruções de soma em uma instrução SIMD equivalente, bem como determinar as instruções que podem ou não ser executadas em paralelo. Se o compilador não conseguir agrupar várias instruções em instruções SIMD é possível usar funções intrínsecas, que invocam diretamente instruções *assembler*. Por exemplo, a função intrínseca _sadd4, que soma 4 inteiros de 1 byte com saturação, na linguagem C, é traduzida para a instrução assembler SADD4. É possível também diminuir fluxo de dados entre memória/*cache* e CPU através de instruções intrínsecas que leem e gravam até 8 bytes de uma só vez na memória.

A última etapa consiste em escrever código *assembler* para os principais funções. Com a codificação *assembler* é possível decidir quais instruções devem ser executadas e em quais unidades funcionais. Assim evita-se o gargalo na obtenção de recursos de hardware, além de passar mais informações para as ferramentas de compilação. Esta etapa depende muito do conhecimento dos desenvolvedores sobre a arquitetura e normalmente não é necessária, pois um desempenho satisfatório é alcançado na segunda etapa.

No compilador do processador DSP c64x+ é possível ativar um *feedback* para as funções compiladas que informa, dentre outros, os recursos usados, a profundidade do pipeline e a quan-

tidade de ciclos de execução. A otimização é feita em cima dos laços de repetição, pois ali está o maior potencial de paralelização. Para analisar como o desempenho pode ser melhorado nas funções DSP serão apresentados três diferentes códigos com diferentes graus de desempenho. Todos eles executam a soma de dois vetores de inteiros de 1 byte sem sinal de n posições e armazenam o resultado num terceiro vetor.

A primeira função, *soma_padrao*, apresentando no Algoritmo 3, é a mais simples. Ela pode ser compilada em qualquer compilador C que siga os padrões ANSI. Nenhuma informação extra é passada ao compilador.

Na função *soma_info*, Algoritmo 4, são passadas algumas informações para o compilador. A palavra-chave *const* indica que os vetores que serão somados não serão alterados dentro da função; a palavra-chave *restrict* indica que nenhum outro ponteiro aponta para aquela posição de memória ou seja, não há sobreposição de endereços de memória. Com essas duas palavras-chave o compilador reduz dependências entre os dados que serão processados. O *pragma* MUST_ITERATE informa o compilador a quantidade mínima de vezes que o laço irá executar, a quantidade máxima e a multiplicidade da quantidade de iterações. Assim o compilador pode, de forma segura, desenrolar o laço de repetição.

Na função *soma_intrinsica*, Algoritmo 5, são adicionadas funções intrínsecas que executam 8 operações de soma a cada repetição do laço. A cada passo do laço 8 bytes de dados de cada vetor são levados da memória para os registradores. Antes de serem processados, cada conjunto de oito bytes são quebrados em pacotes de 4 bytes para então serem serem somados e o resultado gravado na memória.

Algorithm 3 Algoritmo soma padrão

```
void soma_padrao(unsigned char *a, usigned char *b,
        unsigned char *res, int n){
        int i;
        for(i=0; i<n; i++){
                res[i] = a[i] + b[i];
        }
}
```

Algorithm 4 Algoritmo soma com informação

```
void soma_info(const unsigned char * restrict a,
                        const unsigned char * restrict b,
                        unsigned char *restrict c, const int n)
        int i;
        #pragma MUST_ITERATE (512, 1024, 8)
        for(i=0; i<n; i++){
                res[i] = a[i] + b[i];
        }
}
```

Algorithm 5 Algoritmo soma intrínseca

```
void soma_intrinsic(const unsigned char * restrict a,
                        const unsigned char * restrict b,
                        unsigned char *restrict c, const int n){
   #pragma MUST_ITERATE (512/8, 1024/8, 8)
   for(i=0; i<k/8; i+=8){
      unsigned int a1_a0, a3_a2;
          unsigned int b1_b0, b3_b2;
          unsigned int c1_c0, c3_c2;
          a3_a2 = _hi(_amemd8_const(&a[i])); /*4 bytes mais altos de 8

          a1_a0 = _lo(_amemd8_const(&a[i])); /*4 bytes mais baixos de 8
                                              valores lidos*/
          b3_b2 = _hi(_amemd8_const(&b[i]));
          b1_b0 = _lo(_amemd8_const(&b[i]));
          c3_c2 = _saddu4(a3_a2, b3_b2); /*soma 4 bytes sem sinal*/
          c1_c0 = _saddu4(a1_a0, b1_b0);
          _amemd8(&c[i])= _itod (c3_c2,c1_c0) ; */empacote dois inteiros
                                                  sem sinal em um double*/
   }
}
```

O *feedback* do compilador para o três algoritmos pode ser visto na Tabela 2.11. Esses valores são teóricos e consideram condições ideais como dados sempre presentes na memória *cache* e podem apresentar diferenças numa execução real (a comparação entre o tempo teórico e o real pode ser visto no capítulo 4). O desempenho da função *soma_intrínsenca* foi aproximadamente 7.5 vezes melhor que a função *soma_info*, que por sua vez é aproximadamente de 2 vezes mais rápida que a função *soma_normal*. O significado dos dados são apresentados a seguir, e representam informações após a desenrolação dos laços de repetição (*loop unroling* e paralelização

das instruções.

- **Desenrolação do loop**: Indicam quantas vezes o loop original foi desenrolado.
- **Quantidade Mínima de iterações** e **Quantidade máxima de iterações**: indicam quantas vezes o loop irá executar.
- **Ciclos de cada iteração**: quantidade de ciclos de cara iteração do loop.
- **Iterações em paralelo no pipeline**: Indica a profundidade do pipeline do loop.
- **Total de Ciclos**: Quantidade de ciclos total que a função consome.

Tabela 2.11: Feedback do compilador para três implementações diferentes de um mesmo algoritmo

	soma_padrao	**soma_info**	**soma_intrinsic**
Desenrolação do loop	não consegiu	2x	2x
Quantidade Mínima de iterações	desconhecido	256	32
Quantidade Máxima de iterações	desconhecido	512	64
Intervalo entre cada iteração no loop	2	3	3
Iterações em parapelo no pipeline	5	4	3
Total Ciclos	8+iteracoes*2	8+iterações*3	6+iteracoes*3
Ciclos p/ n=512	1030	777	192
Ciclos p/ n=1024	2054	1545	198

2.6 Comunicação ARM-DSP

2.6.1 Aspectos Gerais

Como dito anteriormente os processadores ARM e DSP tem objetivos diferentes. Enquanto o primeiro é um processador de propósito geral onde é executado o sistema operacional, o segundo é dedicado ao processamento digital de sinal em tempo real. O ARM é considerado o processador principal ou hospedeiro, enquanto o DSP é visto do lado do ARM como um periférico, como um impressora ou um modem, com a diferença de ser especializado em processamento de sinais digitais. Assim como o ARM, o DSP contém um sistema operacional especifico para gerenciar seus recursos, sendo a ligação entre os dois SOs feita através da *DSP Bridge* [29].

Em seu mais baixo nível a DSP Bridge constitui de um *device driver* que executa no núcleo do sistema operacional sobre o ARM e se comunica com o Servidor Gerenciador de Recursos (*Resource Manager Server*) no lado do DSP, habilitando assim a comunicação entre o ARM e o DSP através de chamadas *ioctl* ao *device driver* no lado do GPP. Em um nível mais alto, há uma interface para programação de aplicações (API) que utiliza o *device driver* mas que abstrai os recursos e as tarefas sobre eles, como :

- Iniciar tarefas de processamento de sinais no DSP;
- Trocar mensagens com tarefas DSP;
- Criar um fluxo de dados de e para tarefas no DSP;
- Pausar, resumir e deletar tarefas no DSP; e
- Realizar consultas sobre estados dos recursos.

2.6.2 Nodo

Uma das principais abstrações da *DSP Bridge* é o Nodo, que agrupa blocos de código relacionados e dados juntos em unidades funcionais. Um Nodo é um código de máquina, compilado para o processador DSP que pode ser carregado na memória, iniciado e então deletado quando não mais necessário, através da troca de mensagens iniciadas no lado do GPP (Figura 2.14). Como o ARM e o DSP compartilham a mesma memória física é possível tanto alocar uma memória especificamente para o DSP quanto compartilhar ela entre os dois processadores.

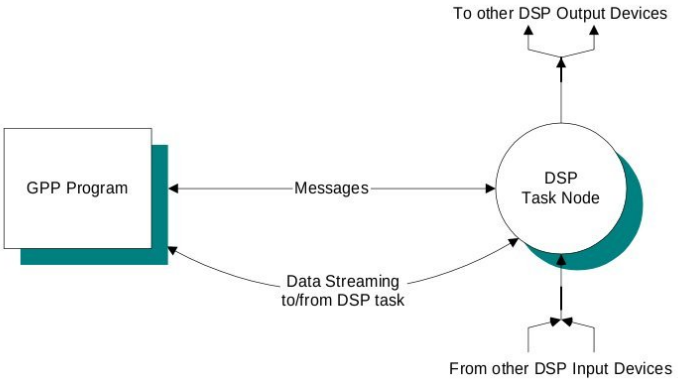


Figura 2.14: Troca de mensagens entre ARM e DSP

2.6.3 Estados do Processamento

Os estados do processamento podem ser vistos na Figura 2.15. Um nodo começa seu ciclo de execução quando um cliente GPP chamada a função **DSPNode_Allocate** que alocará uma estrutura de dados na GPP para permitir controle e comunicação com o nodo recém criado. No estado alocado o nodo existe somente na GPP. Uma vez que o nodo tenha sido criado na GPP, uma chamada a **DSPNode_Create** irá criar um nodo em um estado de pré-execução no lado do DSP.

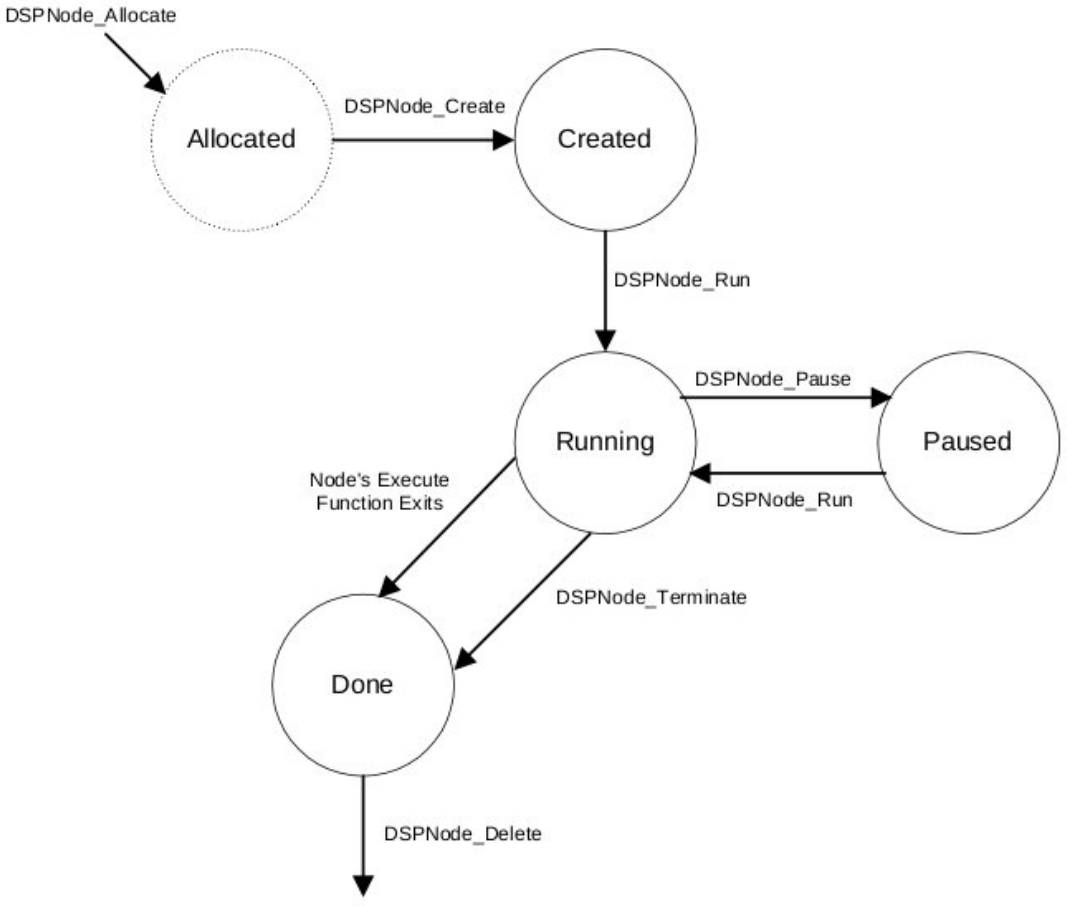


Figura 2.15: Estados de Processamento de um Nodo

Quando a GPP chama **DSPNode_Run** o nodo irá entrar na fase de execução. Nesse estado ele executa suas funções de processamento de sinais digitais. O GPP pode temporariamente suspender a execução de uma tarefa chamando **DSPNode_Pause**, e então resumir a execução chamando **DSPNode_Run** novamente.

O nodo irá terminar sua execução, com transição para o estado Done, quando ele sai de sua fase de execução. A saída pode ocorrer ou porque um nodo completou seu processamento, ou porque o cliente GPP fez uma chamada a **DSPNode_Terminate** para sinalizar o termino da execução.

Para terminar o clico de vida do nodo, ele deverá ser deletado com uma chamada a **DSPNode_Delete**, que irá iniciar no lado do DSP a função responsável pela deleção do nodo. Uma vez que a fase de deleção do nodo tenha executada, todos os recursos do lado do GPP e do lado DSP serão liberados, causando a destruição do nodo.

Capítulo 3

Ampliador digital: ambiente e testes

Para testar o desempenho dos processadores heterogêneos apresentados anteriormente foi desenvolvido um ampliador digital que é executado na placa *BeagleBoard*. Ele captura imagens da *webcam*, trata elas com algoritmos computacionais e exibe o resultado na saída de vídeo. Dentre outros motivos para a escolha desta aplicação, ela trouxe diversas variáveis a serem consideras em um aplicação real que utiliza processador de propósito específico, como será visto nas próximas seções.

3.1 Placa de Desenvolvimento BeagleBoard

A BeagleBoard é uma placa para desenvolvimento de sistemas embarcados, de baixo custo e baixo consumo de energia, que contém processador, memória e entradas e saídas. Ela contém um OMAP3530 *System-On-Chip* que possui um processador ARM Cortex-A8 e um processador TMS320C64x DSP para aceleração de áudio e vídeo, ambos descritos no capítulo anterior.

As principais motivações para a adoção da BeagleBoard neste trabalho foram: (i) baixo custo, em torno de $125; (ii) disponibilidade de diversas interfaces como HDMI, USB e Serial RS232; (iii) Disponibilidade de sistemas operacionais e *device drivers* e (iv) disponibilidade de dois processadores heterogêneos, um de propósito geral e outro de propósito especifico. A Tabela 3.1 mostra alguns dos recursos disponíveis da BeagleBoard, versão C4.

Processadores	- 720MHz ARM Cortex-A8 (Arquitetura RISC) - TMS320C64x+ (520MHz) (Arquitetura DSP)	
Memória	256MB DDR RAM 256MB NAND memória flash	
Periféricos e conexões	- HDMI - S-Video - USB - Entrada e saída Stereo - Porta RS232 - Connector JTAG	
Armazenamento	Slot Cartão SD	

Figura 3.1: Placa BeagleBoard e seus recursos.

Seu tamanho reduzido, de aproximadamente 3x3”, permite o desenvolvimento de protótipos bastante reduzidos. Ela satisfaz muito bem o requisito de baixo consumo de energia dos sistemas embarcados com um consumo de potência em torno de 2W de pico, dependendo dos periféricos conectados na porta USB. Como ela tem somente uma saída USB foi necessário utilizar um hub USB para a conexão de outros periféricos, como mouse, teclado e *webcam*.

Há vários sistemas operacionais disponíveis para a BeagleBoard, como Angstron, Ubuntu, Debian e Android. Eles são distribuídos em forma de código fonte ou em forma de binário, o que facilita o processado de instalação. Além disso cada distribuição tem seus próprios repositórios com aplicações e bibliotecas em formato binário ou código fonte, como compiladores e gerenciadores de janelas, que acabam facilitando o desenvolvimento de novas aplicações, como também permite a adaptação de aplicações já desenvolvidas para outras arquiteturas.

3.2 Distribuição Linux Utilizada

Entre os vários sistemas operacionais disponíveis, optou-se pela distribuição Ubuntu por ser familiar à equipe envolvida neste trabalho, além de ser uma distribuição bem popular. A versão 10.10 de codinome *Maverick*, que foi instalada na BeagleBoard, possui a versão 2.6.36 do *kernel* Linux que era a última versão lançada da linha Ubuntu quando os trabalhos foram iniciados.

Sua instalação é feita num cartão SD em qualquer computador com entrada para este tipo de

mídia. Há vários *wikis* com tutorias extremamente úteis e com roteiros e *links* para download dos arquivos necessários. A forma mais simples de instalação, a qual foi usada neste trabalho, foi baixar um pacote com todos os arquivos necessários com um script de instalação [30]. Entre os arquivos necessários para a instalação estão o *kernel* do Linux e arquivos de *boot*. O sistema de janelas X11 não vem junto com o pacote e foi necessário instalá-lo separado através do gerenciador de pacotes *apt-get*. A instalação dos *device drivers* para o processador DSP foi a parte mais complexa de toda a instalação do ambiente na BeagleBoard, a qual demandou várias tentativas.

3.3 Obtenção dos tempos para análise

O Sistema operacional Linux é um sistema de tempo compartilhado onde diversos processos compartilham o acesso a CPU. Para medir o tempo de execução dos algoritmos, de forma precisa, deve-se deixar apenas o processo alvo, o qual deseja-se obter os tempos de execução, ocupando a maior parte da CPU. Deve-se evitar que outros processos concorram pelo acesso a CPU com o processo alvo, salvo os *daemons* que ocupam uma parte insignificante de tempo de execução na CPU.

Apesar de servir para uma ampla gama de testes que precisam medir o tempo de processamento, há casos em que não há como evitar que outro processo concorra pelo uso da CPU. Essa situação ocorre, por exemplo, quando utilizamos aplicações comnsinterfaces gráficas, como neste trabalho, onde tratamos com processamento e exibição de imagem. O processo referente ao sistemas de janelas X inevitavelmente estará em execução, e quanto mais atualizações forem feitas em alguma janela, mais processamento será demandado ao servidor X.

Para medir os tempos de processamento da aplicação que será executado sobre os dois processadores, será utilizado o **tempo real** que contabiliza o tempo total entre dois pontos do programa, independente se de quanto tempo o processo esteve escalonado na CPU. Com isto, os tempos em que o *device driver* da *webcam* e o servidor X ocupa a CPU serão contabilizados no tempo total da aplicação, que é mais correto no caso deste trabalho.

3.4 Captura de imagem da webcam

A captura de imagens da *webcam* no Linux é feita pela interface *V4L2*. Video For Linux 2 (V4L2) é uma interface de programação para captura de vídeo no Linux, usadas por *webcans* USB e *TV Turnes* (placa para captura de sinais de TV), que impõe um padrão de comunicação para classes de dispositivos. Ela não é apenas uma biblioteca comum do espaço do usuário, mas também um conjunto de *device drivers* padronizados para o *kernel* do Linux, que fazem todo o trabalho de comunicação com o dispositivo.

Os *device drivers* V4L2 são implementados como módulos do *kernel* e plugados dentro do módulo *videodev*. O módulo *videodev* é como se fosse uma classe abstrata com algumas operações básicas. Cada *device driver* V4L2 deve implementar algumas rotinas padrões deste módulo, que acesse e interaja com um dispositivo específico. A comunicação do aplicativos no lado do usuário com os *drivers* no lado do *kernel* é feita através de chamadas *ioctl*. Chamadas *ioctl* são compostas por um identificador do dispositivo, de um número que indica a operação desejada e um ponteiro para os dados que serão transmitidos entre o usuário e o *kernel*.

Num nível superior as chamadas *ioctl* há uma pequena biblioteca que atua como uma camada fina entre a aplicação e as chamadas de sistema para o driver. Chamadas como *open* e *close*, para abrir e fechar um dispositivo, que poderiam ser feitas diretamente, são feitas através das chamadas *v4l2_open* e *v4l2_close*. Entre as chamadas v4l2 e as chamadas de sistema, são verificadas algumas condições e feitas algumas configurações.

Uma série de etapas são necessárias antes capturar imagens da *webcam*. Primeiramente o dispositivo, mapeado em /dev/videoX, deve ser aberto através de uma chamada a *v4l2_open*. Então através de uma chamada *ioctl* é enviado para o *driver* o formato de imagem desejada, as dimensões dela e outras informações. Neste trabalho foram usadas imagens em formato RGB de dimensões 640x480 (a mais alta resolução disponível pela *webcam* usada). Como o formato nativo da webcam é YUYV, foi solicitado ao *device driver* que fizesse a conversão para RGB, a qual demanda um certo tempo. O último passo é fazer o mapeamento em memória de um endereço no espaço do usuário para um endereço no espaço do *kernel* para a troca de informações entre o *device driver* e a aplicação.

No processo de obtenção das imagens, elas são retiradas da frente de uma fila de 4 *buffers*. Após utilizado, o *buffer* é colocado no fim da fila e o *device driver* trata de preencher o *buffer*

com uma nova imagem. Desta forma, uma chamada para a obtenção de uma imagem da *web-cam* dificilmente será bloqueada pois sempre haverá uma imagem disponível no início da fila. Consequentemente isso irá diminuir os tempos de processamento uma vez que o *device driver* trabalha em segundo plano preenchendo os *buffers* de imagem (apesar de estar sendo executado em segundo plano, o tempo dele acaba sendo contabilizado no tempo total medido pois irá tirar outro processo da CPU para executar). Ao finalizar a aplicação uma chamada a *v4l2_close* deve ser feita para liberar o dispositivo. Vale ressaltar que os drives não suportam múltiplas aplicações lendo ou escrevendo em um mesmo dispositivo. Isto deve ser feito do lado do usuário através de um *proxy*.

3.5 Aplicação desenvolvida: ampliador xLupa embarcado

O ampliador digital desenvolvido neste trabalho, aqui chamado de xLupa Embarcado (para saber mais sobre o projeto xLupa e seu ampliador para *PCs*, consulte [31]), além dos objetivos deste trabalho, tem como objetivo auxiliar pessoas com baixa visão em atividades do seu cotidiano, como ler um livro, jornal ou revista, ver uma figura, dentre outras. Mais informações sobre este e outros trabalhos desenvolvidos pelo grupo, poderão ser encontrados futuramente em *http://projetos.unioeste.br/campi/xlupa/*.

A Figura 3.2 mostra o ambiente com a placa de desenvolvimento *BeagleBoard* executando a aplicação desenvolvida neste trabalho. Além da placa há uma *webcam* que captura a imagem a ser processada, um *hub* USB com alimentação externa, e um monitor com entrada VDI (é utilizado um conversor VDI/HDMI) onde o resultado final, a imagem processada, é exibida.

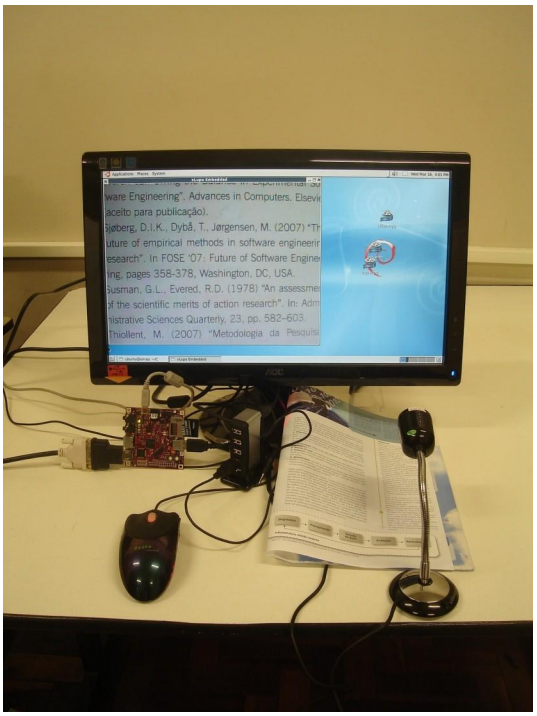


Figura 3.2: Ambiente montado para os testes

O fluxo macro da aplicação é mostrado na Figura 3.3. A *webcam* captura as imagens que são transmitidas para a *BeageBoard*, onde há o Linux instalado, através de uma conexão USB. Através da interface *V4L2* o software obtém as imagens. Uma vez que as imagens estão disponíveis elas são processadas com algoritmos de tratamento de imagens (descritos na próxima seção). O resultado final é renderizado no sistema de janelas X11 através do *toolkit* GTK. A saída de vídeo pode ser qualquer *display* com entrada HDMI (a imagem também pode ser acessada através de uma conexão sem fio com redirecionamento de imagem do servidor X).

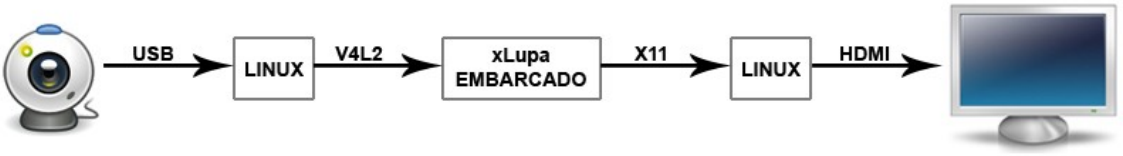


Figura 3.3: Fluxo de dados no xLupa embarcado

3.6 Algoritmos de tratamento de imagem

Nas próximas subseções são apresentados os algoritmos implementados para o tratamento de imagens: Algoritmo Escala de Cinza, Brilho, Binarização e Ampliação. Seus resultados individuas e o impacto na aplicação como um todo serão analisados no próximo capítulo.

3.6.1 Algoritmo escala de cinza

Este algoritmo visa transformar uma imagem colorida em cinza. Foi utilizado na implementação o método da média simples que consiste na média dos canais R, G e B de cada pixel (Equação 3.1).

$$PixelCinza = \frac{R + G + B}{3} \tag{3.1}$$

Tanto a implementação para o processador ARM quanto para o processador DSP são muito semelhantes. No DSP, como a média depende de exatamente três valores e as funções intrínsecas do DSP atuam sobre dois, quatro ou oito valores, optou-se por deixar que o compilador otimizasse o algoritmo (alguns testes foram feitos usando várias operações intrínsecas para se chegar ao mesmo resultado mas eles obtiveram um tempo inferior). Uma restrição de multiplicidade de 8 *pixels* nas colunas foi imposta como forma de fornecer mais informações ao compilador para que ele fizesse otimizações.

As implementações para o ARM e para o DSP podem ser vistas nos Algoritmos 6 e 7.

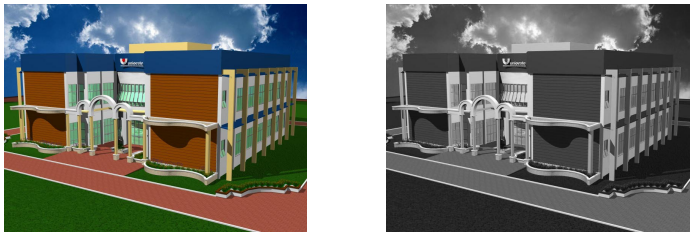
Algorithm 6 Algoritmo Cinza - ARM

```
void arm_imagem_cinza(int tamanho, char *imagem){
        for(int i=0; i<tamanho; i+=3){
                char cinza = (imagem[i] + imagem[i+1] + imagem[i+2])/3;
                imagem[i] = imagem[i+1] = imagem[i+2] = cinza;
        }
}
```

Algorithm 7 Algoritmo Cinza - DSP

```
void dsp_imagem_cinza(int tamanho, char* restrict imagem){
        #MUST_ITERATE(8,0,8)
        for(int i=0; i<tamanho; i+=3){
                char cinza = (imagem[i] + imagem[i+1] + imagem[i+2])/3;
                imagem[i] = imagem[i+1] = imagem[i+2] = cinza;
        }
}
```

O resultado da aplicação do algoritmo pode ser visto na Figura 3.4



(a) Imagem original (b) Imagem cinza

Figura 3.4: Algoritmo escala de cinza aplicado sobre uma imagem

3.6.2 Algoritmo brilho

O algoritmo brilho visa aumentar ou diminui o brilho de uma imagem. A equação dele, assim como a equação para se chegar ao nível de cinza, consiste na média simples dos canais R, G e B. Desta forma, o brilho é dado pela expressão abaixo:

$$Brilho = \frac{R + G + B}{3} \tag{3.2}$$

O aumento ou diminuição de brilho consiste em aumentar a média das componentes R, G e B, como mostra a Equação 3.3.

$$Brilho + k = \frac{(R + k) + (G + k) + (B + k)}{3} \tag{3.3}$$

Desta forma, no aumento ou na diminuição de brilho, o valor de cada componente RGB depende apenas dela mesma e não da tupla RGB como ocorre nos algoritmos de limiarização e escala de cinza. Isto torna possível utilizar as funções intrínsecas do DSP, pois agora quatro ou oito valores podem ser processados em paralelo. Os algoritmos 8 e 9 mostram a implementação do algoritmo de aumento de brilho nos dois processadores.

Algorithm 8 Algoritmo Brilho ARM

```
void arm_imagem_aumenta_brilho(int tamanho, char brilho, char *data){
        for(i=0; i<tamanho; i++){
                int novo = data[i]+brilho;
                if(novo>255)
                        data[i] = 255;
                else if(novo[i]<0)
                        data[i]=0;
                else data[i] += novo;
        }
}
```

Algorithm 9 Algoritmo Brilho DSP

```
void dsp_imagem_aumenta_brilho(int tamanho, char brilho, char * restrict imagem){
        unsigned int pack = brilho | brilho<8 | brilho<16 | brilho<24;
        #pragma MUST_ITERATE (8, , 8)
        for(int i=0; i<tamanho; i+=8){
                unsigned int ui1 = _hi(_memd8(&imagem[i]));
                unsigned int ui2 = _lo(_memd8(&imagem[i]));
                unsigned int d1 = _saddu4 (pack, ui1);
                unsigned int d2 = _saddu4 (pack, ui2);
                _memd8(&imagem[i]) = _itod (d1,d2);
        }
}
```

O resultado da aplicação do algoritmo pode ser visto na Figura 3.5

(a) Imagem original (b) Diminuição de brilho (c) Aumento de brilho

Figura 3.5: Algoritmo brilho aplicado sobre uma imagem

3.6.3 Algoritmo binarização

O algoritmo de binarização tem por objetivo transformar uma imagem cinza ou colorida em uma imagem de apenas duas cores (binária). Um limiar estabelece o limite de divisão entre as

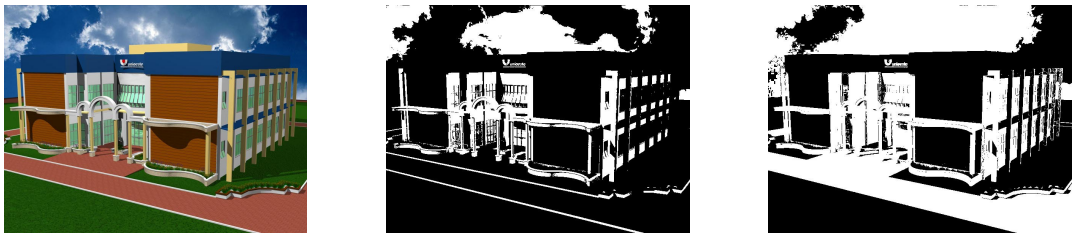
cores. Aos valores com brilho menores que o limiar é atribuído uma cor, e aos valores maiores ou iguais, outra.

O Algoritmo 10 mostra a implementação do algoritmo no processador ARM. A implementação para o processador DSP é semelhante (como no algoritmo escala de cinza), incluindo apenas a palavra-chave *restrict* e o *pragma MUST_ITERATE* no laço de repetição.

Algorithm 10 Algoritmo de Binarização ARM/DSP*

```
/*O algoritmo para o processador DSP inclui apenas a palavra chave restrict no ponteiro
imagem e o pragma MUST_ITERATE(8,,8) no laco de repeticao.*/
void arm_imagem_binaria(tamanho, int limiar, char cor1[3], char cor2[3], char *imagem){
        for(int i=0; i<tamanho; i+=3){
                int media = (data[i] + data[i+1] + data[i+2])/3;
                if(media < limite){
                        imagem[i]=cor1[0];
                        imagem[i+1]=cor1[1];
                        imagem[i+2]=cor1[2];
                }else{
                        imagem[i] = cor2[0];
                        imagem[i+1] = cor2[1];
                        imagem[i+2] = cor2[2];
                }
        }
}
```

O resultado da aplicação do algoritmo, com diferentes limiares, pode ser vistos na Figura 3.6.



(a) Imagem original (b) Limiar = 128 (c) Limiar = 80

Figura 3.6: Algoritmo binarização (preto/branco) aplicado sobre uma imagem

3.6.4 Algoritmo ampliação

O algoritmo ampliação tem como objetivo aumentar o tamanho de uma imagem. Ele pode ser aplicado a uma imagem completa ou apenas uma parte dela, dependendo do tamanho da

imagem de saída desejada. O algoritmo mais simples, com boa eficiência e baixa suavidade nos contornos é o algoritmo do vizinho mais próximo, utilizado neste trabalho. Outros algoritmos se baseiam na forma como os *pixels* são interpolados e em geral são mais custosos e envolvem um compromisso entre desempenho, suavidade e nitidez.

O algoritmo do vizinho mais próximo consiste basicamente em replicar os *pixels* sem qualquer tipo de interpolação. A Figura 3.7 ilustra o funcionamento do algoritmo com ampliação de duas vezes. Na Figura mais a esquerda é mostrado uma porção de imagem original de 4x4 *pixels* de dimensão. A imagem do meio mostra a imagem original duas vezes maior, mas algumas lacunas, que são os *pixels* a serem preenchidos. A Figura mais a direita mostra a imagem original com as lacunas preenchidas pelo algoritmo do vizinho mais próximo. Outra visão do algoritmo é mostrado na Figura 3.8.

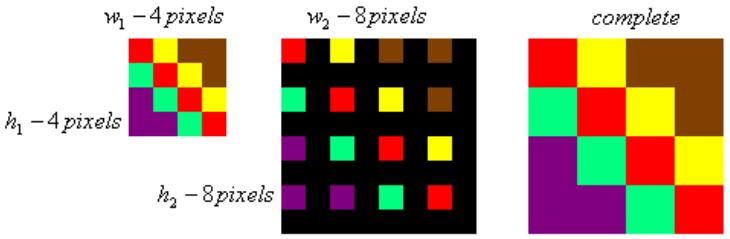
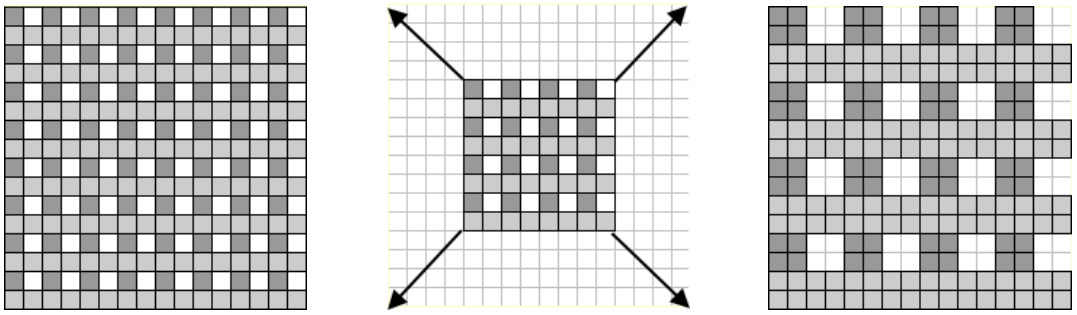


Figura 3.7: Algoritmo do vizinho mais próximo.



(a) Imagem Original (b) Porção a ser ampliada (c) Porção Ampliada

Figura 3.8: Processo de Ampliação com zoom de 2 vezes

De forma a permitir que o algoritmo possa gravar a saída (imagem ampliada) na mesma matriz que de entrada (imagem original), como nos outros algoritmos, ele é executado das bordas em direção ao cento no sentido vertical, e na linha do central é processado das bordas para o centro no sentido horizontal, como mostrado da Figura 3.9. O resultado da aplicação do

algoritmo implementando neste trabalho pode ser visto na Figura 3.10

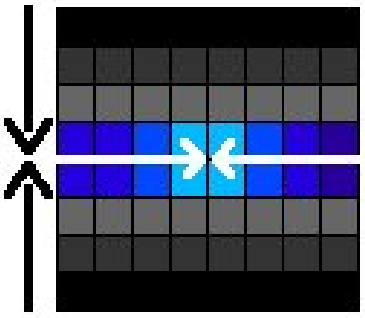


Figura 3.9: Sentido do processamento do algoritmo de ampliação.

O resultado da aplicação do algoritmo pode ser visto na Figura 3.10.

(a) Imagem original (b) Ampliada 2x (c) Ampliada 3x

Figura 3.10: Algoritmo ampliação aplicado sobre uma imagem

O algoritmo implementado no ARM é apresentado abaixo (Algoritmo 11 e 12). O algoritmo para o DSP apenas inclui a palavra-chave *restrict* e o *pragma* MUST_ITERATE nos lações de repetição.

Algorithm 11 Algoritmo Zoom - parte 1

```
/*Duas imagens do mesmo tamanho − zoom no centro
  arm_nearest_neighbor_interpolation*/
void arm_vizinho_mais_proximo_invert(GdkPixbuf *src_pixbuff, int scale){
        /*restrict */guchar* src = gdk_pixbuf_get_pixels (src_pixbuff);
        int width = gdk_pixbuf_get_width (src_pixbuff);
        int height = gdk_pixbuf_get_height (src_pixbuff);
        int dst_x = width/2 − width/scale/2;
        int dst_y = height/2 − height/scale/2;
        int i, j, k, cnt=0;
        int p_src=(dst_x+dst_y*width)*3;
        int p_dst=0;
        int ant=p_src;

        for(i=0; i<height/2−1; i++){
                //linha superior
                p_dst=i*width*3;
                p_src=(dst_x+dst_y*width)*3+(i/scale)*width*3;
                /*MUST_ITERATE (8,0,8)*/
                for(j=0; j<width; ){
                        for(k=0; k<scale;k++,j++){
                                src[p_dst] = src[p_src];
                                src[p_dst+1] = src[p_src+1];
                                src[p_dst+2] = src[p_src+2];
                                p_dst+=3;
                        }
                        p_src+=3;
                }
                //linha inferior
                p_dst=(height−i−1)*width*3;
                p_src=(dst_x+dst_y*width)*3+((height−i−1)/scale)*width*3;
                for(j=0; j<width; ){
                        for(k=0; k<scale;k++,j++){
                                src[p_dst] = src[p_src];
                                src[p_dst+1] = src[p_src+1];
                                src[p_dst+2] = src[p_src+2];
                                p_dst+=3;
                        }
                        p_src+=3;
                }
        }
```

Algorithm 12 Algoritmo Zoom - parte 2

```
        //linha do meio
        for(i=height/2−1; i<height/2+1;i++){
                //Da direita ao centro
                p_dst=i∗width∗3;
                p_src=(dst_x+dst_y∗width)∗3+(i/scale)∗width∗3;/
                /∗MUST_ITERATE (8,0,8)∗/
                for(j=0; j<width/2; ){
                        for(k=0; k<scale;k++,j++){
                                src[p_dst] = src[p_src];
                                src[p_dst+1] = src[p_src+1];
                                src[p_dst+2] = src[p_src+2];
                                p_dst+=3;
                        }
                        p_src+=3;
                }

                //Da esquerda ao centro
                p_dst=i∗width∗3+width∗3−3;
                p_src=(dst_x+dst_y∗width)∗3+(i/scale)∗width∗3+(width/scale)∗3−3;
                for(j=width/2−1; j>=0; ){
                        for(k=0; k<scale;k++,j−−){
                                src[p_dst] = src[p_src];
                                src[p_dst+1] = src[p_src+1];
                                src[p_dst+2] = src[p_src+2];
                                p_dst−=3;
                        }
                        p_src−=3;
                }
        }
}
```

Capítulo 4

Resultados

Neste capítulo serão apresentados os resultados obtidos neste trabalho. Primeiramente será feito uma abordagem individual entre os algoritmos Brilho, Binarização, Cinza e Zoom em ambos os processadores para diferentes tamanhos de imagens. Após, uma comparação entre os tempos reais, obtidos na execução dos algoritmos, e os teóricos, disponibilizados pelo compilador do DSP, será realizada. Em seguida, uma abordagem mais geral será feita analisando os algoritmos de processamento de imagens com outros tempos de processamentos envolvidos na aplicação como a **captura** da imagem da *webcam* e a **renderização** da imagem já processada no dispositivo de saída.

As entradas dos algoritmos executados no processador DSP possuem memória compartilhada com o processador ARM. Desta forma, não é necessário uma transferência explícita para trocar dados entre os dois processadores. Além disso, a imagem resultante da execução dos algoritmos são escritas sobre as imagens de entrada.

4.1 Algoritmos de tratamento de imagem

4.1.1 Algoritmo escala de cinza

A implementação do algoritmo escala de cinza no processador DSP foi semelhante a do processador ARM, pois é necessário processar uma componente RGB de 3 valores a cada passo do laço de repetição, porém as funções intrínsecas disponíveis operam sobre dois, quatro ou oito valores. Algumas tentativas de aproximação foram feitas mas elas apresentaram um desempenho pior. Optou-se então por deixar o compilador otimizar o código.

Os resultados são apresentados na Tabela 4.1 e o gráfico na Figura 4.1. O desvio padrão

no ARM foi bem superior ao DSP. Isto pode ser atribuído, além de outros fatores, ao próprio tempo que o *kernel* ocupa a CPU para gerenciar os processos, fazendo assim oscilar os tempos medidos no ARM, o que não ocorre no DSP com apenas um processo.

O ganho no processador de sinais digitais foi inferior ao de propósito geral apenas para a menor imagem. Para as outras imagens o desempenho no DSP foi superior, chegando a executar o algoritmo escala de cinza 35% mais rápido que o ARM para a maior imagem.

Tabela 4.1: Algoritmo escala de cinza: média e desvião padrão por imagem (100 amostras)

	ARM		**DSP**		**Desempenho**
Tamanho imagem	média (ms)	d.p (ms)	média (ms)	d.p(ms)	DSP/ARM
320x240	2.90	0.39	7.37	0.58	+154.14%
640x480	14.57	2.07	13.56	0.55	-6.93%
800x600	22.76	2.40	18.26	0.88	-19.77%
1024x780	37.86	3.83	26.67	0.84	-29.56%
1280x800	48.64	4.10	32.58	0.78	-33.02%
1440x900	61.50	4.17	39.89	0.73	-35.14%

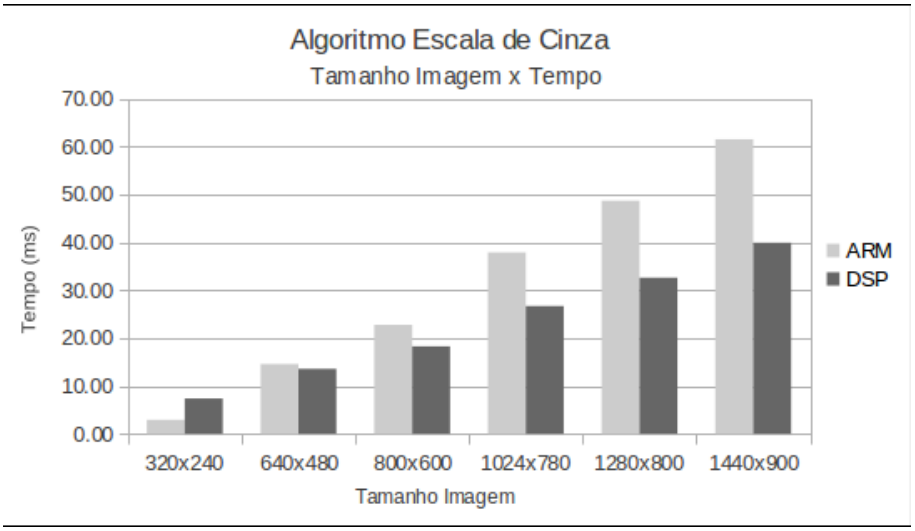


Figura 4.1: Algoritmo escala de cinza : tempo x tamanho da imagem

4.1.2 Algoritmo brilho

A implementação do algoritmo brilho no processador DSP pode ser considerada perfeita pois utilizada somente funções intrínsecas e processa oito bytes por vez a cada passo do laço de

repetição (Algoritmo 9). Com isto, o algoritmo brilho precisou menos da metade do tempo para executar no DSP, apesar do ARM operar a uma frequência superior (720Mhz x 520Mhz).

Abaixo, pode ser visto a média e o desvio padrão dos tempos de execução, tanto no ARM quanto no DSP, além do percentual de tempo reduzido com o uso do DSP (Tabela 4.2 e Figura 4.2).

Tabela 4.2: Algoritmo brilho: tempo de execução (100 amostras)

	ARM		**DSP**		**Desempenho**
Tamanho imagem	média (ms)	d.p (ms)	média (ms)	d.p(ms)	DSP/ARM
320x240	5.15	1.12	2.88	0.74	-44.08%
640x480	21.11	2.86	9.45	0.48	-55.23%
800x600	32.38	2.76	14.39	0.50	-55.56%
1024x780	53.91	3.81	23.50	0.72	-56.41%
1280x800	68.79	4.35	30.02	0.79	-56.35%
1440x900	86.81	5.41	37.83	0.66	-56.42%

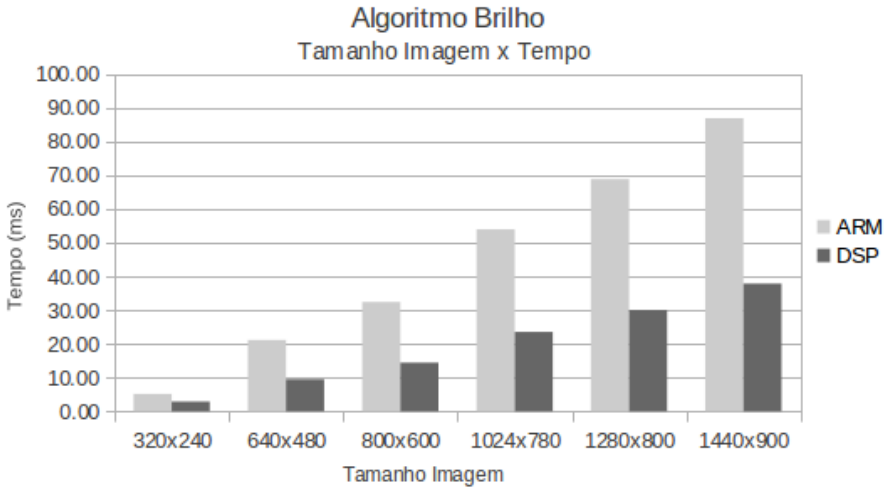


Figura 4.2: Algoritmo brilho : tempo x tamanho da imagem

4.1.3 Algoritmo binarização

A implementação do algoritmo no processador DSP, assim como no algoritmo escala de cinza, é semelhante a implementação para o processador ARM. Os resultados, para diferentes tamanhos de imagens podem ser vistos na Tabela 4.3 e na Figura 4.3. Nota-se que a partir das

imagens com tamanho 800x600 os valores são semelhantes aos obtidos no algoritmo escala de cinza.

O processador DSP ainda tem maior desempenho em relação ao ARM, consumindo aprimadamente 40% menos tempo para a execução do mesmo algoritmo. Apesar disto, o desempenho é menor que o obtido no algoritmo brilho.

Tabela 4.3: Algoritmo binarização: média e desvião padrão por imagem (100 amostras)

	ARM		**DSP**		**Desempenho**
Tamanho imagem	média (ms)	d.p (ms)	tempo (ms)	d.p(ms)	DSP/ARM
320x240	4.17	1.05	3.37	0.52	-20.18%
640x480	17.52	2.49	11.49	0.57	-38.68%
800x600	27.12	3.19	17.37	0.44	-35.95%
1024x780	44.51	3.56	28.47	0.63	-36.04%
1280x800	58.15	3.96	36.22	0.58	-37.71%
1440x900	73.92	6.28	45.75	0.61	-38.11%

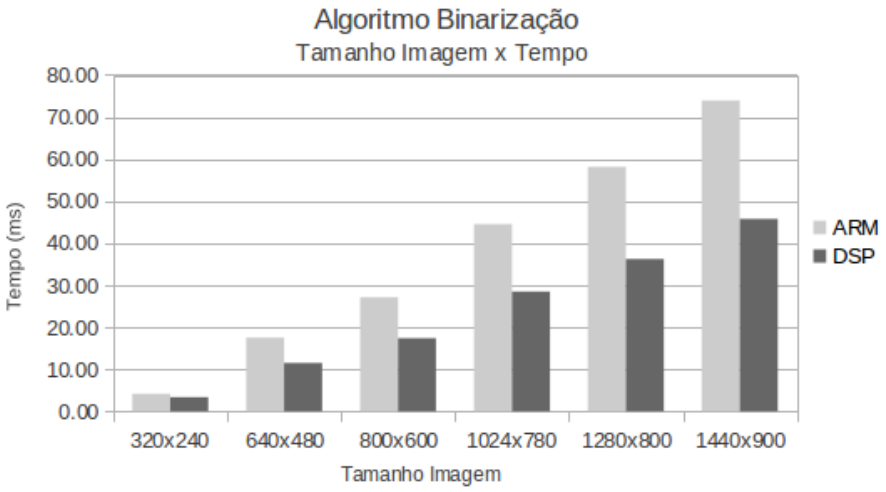


Figura 4.3: Algoritmo binarização : tempo x tamanho da imagem

4.1.4 Algoritmo ampliação

O algoritmo ampliação foi o único, dentro os três, que teve o desempenho no DSP pior que no ARM, com um tempo superior ao dobro do tempo do ARM. Um dos motivos foi o não uso das funções intrínsecas, deixando ao compilador a tarefa de otimizar o código. Outro fator que influenciou é o comportamento espacial com o qual opera o algoritmo. Ao contrário dos demais

algoritmos, a cada passo do laço de repetição mais interno ele acessa várias linhas diferentes da matriz de imagem. Isso faz com que aumentem as chances de ocorrerem falhas de cache.

A Tabela 4.4 e a Figura 4.4 mostram os resultados obtidos para várias imagens quando aplicado um fator de ampliação igual a dois. Outros fatores de ampliação obtiveram tempos aproximados e podem ser vistos na seção 4.3;

Tabela 4.4: Algoritmo ampliação (fator=2): média e desvião padrão por imagem (100 amostras)

	ARM		**DSP**		**Desempenho**
Tamanho imagem	média (ms)	d.p (ms)	tempo (ms)	d.p(ms)	DSP/ARM
320x240	2.09	1.68	4.62	1.03	+121.05%
640x480	8.30	3.14	16.93	1.77	+103.98%
800x600	13.71	4.39	25.79	1.42	+88.11%
1024x780	21.54	6.02	42.59	1.39	+97.75%
1280x800	26.88	5.93	54.37	1.54	+102.34%
1440x900	36.89	9.40	67.89	1.20	+84.03%

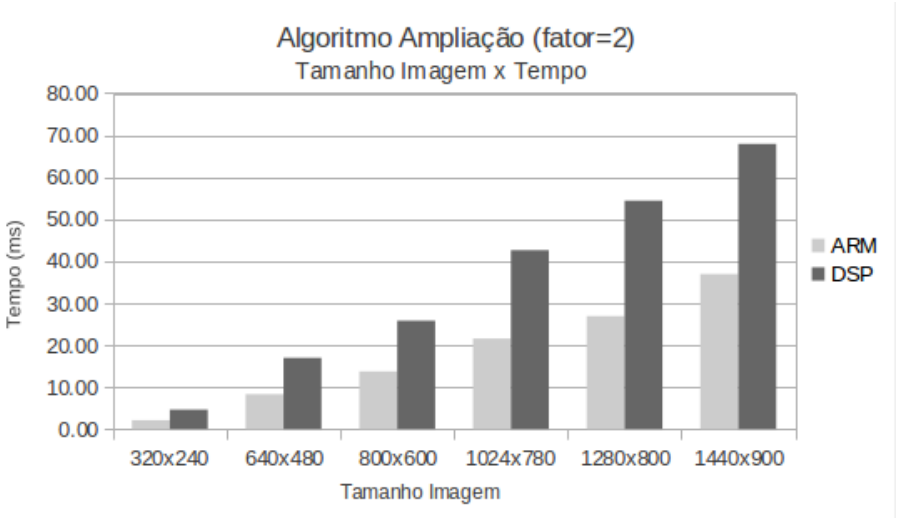


Figura 4.4: Algoritmo ampliação com fator igual 2x: tempo x tamanho da imagem (100 amostras)

Isto mostra que nem sempre a escolha pelo processador DSP é a melhor. No caso do algoritmo de ampliação caso o objetivo seja reduzir o tempo de processamento ele deve ser executado no ARM. Se ao invés disso, o objetivo é diminuir a carga de processamento no ARM e possivelmente o consumo de energia, então o DSP seria a melhor escolha.

4.2 Tempos teóricos do DSP

Na fundamentação, capítulo 1, foram mostradas algumas informações que o compilador do processador DSP disponibiliza. Serão comparados aqui os tempos teóricos fornecidos pelo DSP e os tempos obtidos na prática pela execução dos algoritmos.

Os dados podem ser vistos na Tabela 4.5, onde **ii (intervalo de iteração)** é número de ciclos entre o início de sucessivas iterações do *loop* principal do algoritmo e **ppi** é quantidade de *pixels* processados em cada iteração. A partir destes dois dados é calculado o tempo teórico como a quantidade de *pixels* processados por ciclos (*hertez*). Como cada nova iteração do *loop* é iniciada em *ii* ciclos, para um número alto de iterações o tempo de cada uma pode ser aproximada como sendo o tempo de intervalo entre elas, e portanto a quantia de ciclos por *herz* é dada pela razão entre ii e ppi.

O tempo prático, obtido com os resultados das execuções dos algoritmos, é calculado como sendo a razão entre a quantidade de *pixels* processados por segundo e a frequência do processador DSP (520 Mhz).

A quantidade de *pixels* processados por *hertz* é inferior na prática, para a maioria dos casos, em relação a teoria, chegando a uma diferença de 7x no caso do algoritmo escala de cinza. Dois itens muito importantes que foram desconsiderados são o tempo de acesso à memória principal, que opera a uma frequência de até três dígitos abaixo do processador e as falhas de cache, ou seja, o compilador considera que os dados estão sempre presentes na cache *L1*. Assim, é muito vaga a comparação entre os tempos fornecidos pelo compilador do DSP e o tempo real de execução.

No entanto, este tempo pode ser útil para comparar os tempos teóricos entre si. Porém, estes valores não foram totalmente condizentes com os tempos obtidos na prática. O algoritmo escala de cinza, segundo o compilador, seria tão eficiente quanto o algoritmo brilho, o que não se mostrou verdadeiro nos testes realizados. O algoritmo binário é o quarto mais eficiente na teoria e o terceiro na prática. O algoritmo ampliação por conter um código mais complexo, com dois laços aninhados, já era de se esperar um tempo teórico não condizente com a realidade.

Tabela 4.5: Tempo teórico: *Feedback* do compilador do DSP

			pixel/Hz		Relação
	ii[1]	ppi[2]	Teórico	Prático	Teórico/Prático
Escala de Cinza	3	1	0.333	0.043	7.74x
Brilho	8	8/3	0.333	0.065	5.12x
Binário	8	1	0.125	0.051	2.45x
Ampliação	6	1	0.166	0.034	0.48x

[1]Intervalo de iteração: número de ciclos entre o início de sucessivas iterações do *loop*
[2]Pixels processados por iteração: pixels processados em cada iteração do *loop*

4.3 Tempos da aplicação

Além dos algoritmos de processamento de imagem, a aplicação necessita de mais dois processamentos que demandam uma certa quantia de tempo e influenciam no desempenho geral da aplicação : captura da imagem da *webcam* e a renderização da imagem processada em uma janela de aplicação (GUI). Será utilizado, a partir de agora, o formato de imagem 640x480 que é disponibilizado pela webcam utilizada neste trabalho. Os tempos dos processamentos citados acima podem ser vistos na Tabela 4.6.

Tabela 4.6: Tempo Processamento (640x480): Captura e Renderização (100 amostras)

Processamento	**Tempo (ms)**	**d.p (ms)**
Captura	24.81	0.31
Renderização	41.6	0.22

O algoritmo de ampliação variou de tempo conforme fator de ampliação desejado. Quanto maior é o fator de ampliação, para o mesmo tamanho de saída da imagem, menor é a área que será ampliada, logo, há menos acessos à memória para buscar as componentes RGB. Os tempos podem ser vistos na Figura 4.5, sendo o fator de ampliação igual a dois o que teve o maior tempo. Desta forma, este fator de ampliação é o escolhido para medir o tempo total da aplicação, pois ele possui o tempo limitante entre todos os fatores de ampliação.

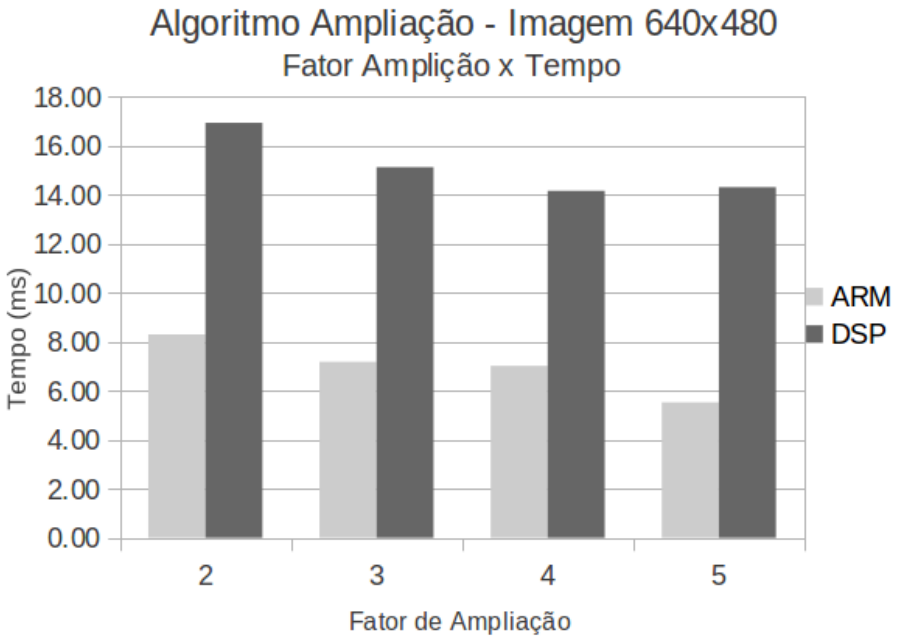


Figura 4.5: Algoritmo Binarização : Tempo x Tamanho da imagem

Na Figura 4.6 são mostrados os tempos dos algoritmos de processamento de imagem, tanto para o processador ARM quanto para o processador DSP, utilizados na aplicação (imagem 640x480). O algoritmo ampliação executado no ARM é aproximadamente duas vezes mais rápido que quando executado no DSP, enquanto que para o algoritmo brilho o DSP é mais de duas vezes mais eficiente que o ARM. Nos outros dois algoritmos, o DSP é ligeiramente mais rápido que o ARM.

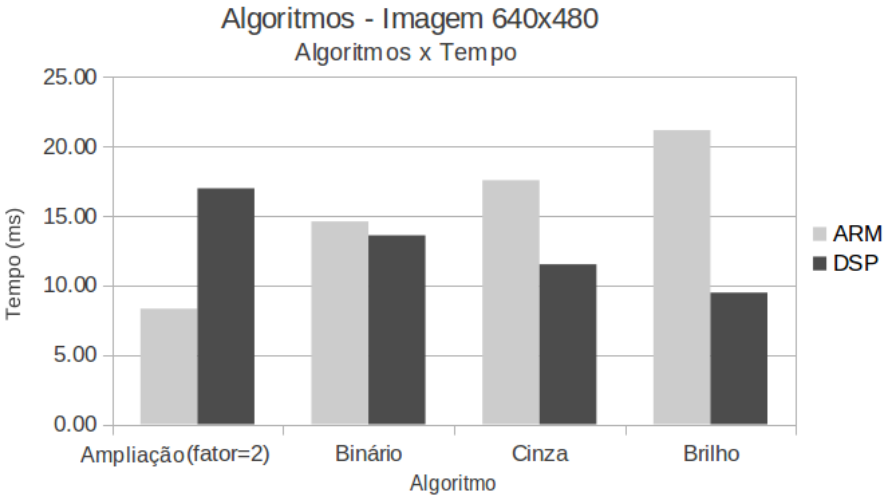


Figura 4.6: Imagem 640x480 : tempo x algoritmo

O diagrama de fluxo de toda a aplicação que envolve os dois processadores pode ser visto da Figura 4.7. Um losangolo representa uma decisão sobre qual processador usar (ARM ou DSP) para determinado algoritmo ou se o algoritmo não será aplicado a imagem. Um retângulo branco indica que o algoritmo será processado no ARM e um retângulo cinza que o algoritmo será processado no DSP. Uma linha vermelha representa uma decisão independente do tempo de processamento, ou seja, é uma decisão do usuário. Finalmente, a linha azul indica uma escolha pelo processador mais eficiente em termos de tempo para a execução do algoritmo em questão. Nas arestas há os tempos de processamento com uma variação para mais ou para menos o desvio padrão.

Se o diagrama da Figura 4.7 for analisado como um grafo ponderado, então é possível verificar qual o pior caso de execução, ou seja, qual é o tempo máximo de execução (tempo limitante), verificando o maior caminho entre o início do processo de *refresh* e o fim dele. Nesta analise o processador DSP ficou com o segundo menor tempo e o ARM em último lugar com os piores tempos. A única exceção é o algoritmo ampliação.

São possíveis três abordagens diferentes na combinação dos dois processadores. Os tempos são apresentados na Tabela 4.7. Nela há o tempo de execução para cada abordagem e a carga que cada processador recebe durante a execução de um processamento completo de *refresh*. A primeira linha da tabela mostra os resultados quando somente o processador ARM é

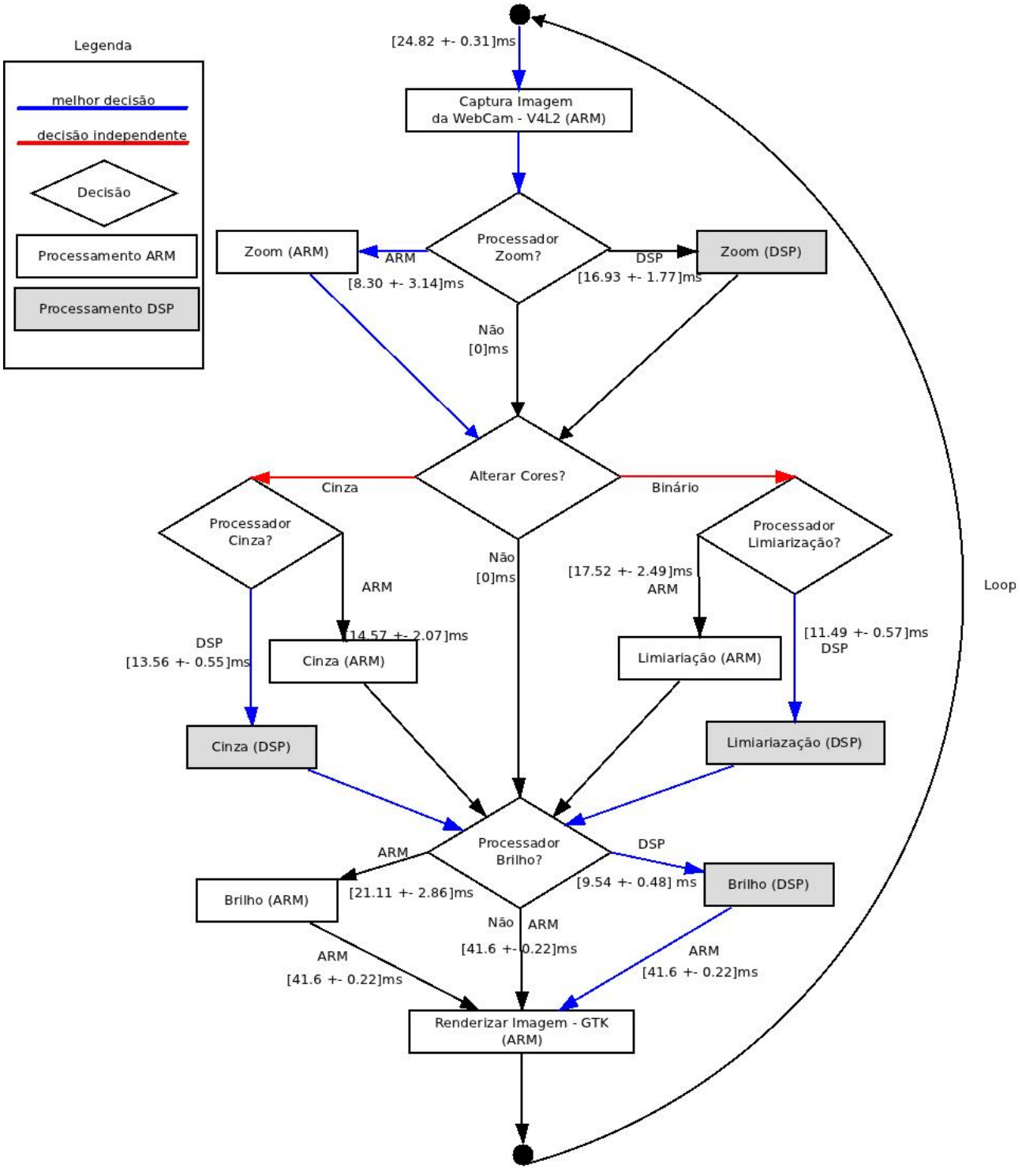


Figura 4.7: Fluxo de Execução do xLupa Embarcado.

utilizado. A segunda quando o processador DSP é utilizado para os algoritmos de tratamento de imagem, e o processador ARM nos processamentos que o DSP não consegue realizar (captura e renderização). A terceira linha , *ARM | DSP*, corresponde ao uso de apenas um processador

por vez mas optando pelo processador mais rápido quando há dois disponíveis para o mesmo processamento.

Tabela 4.7: Algoritmo da aplicação: quatro abordagens

Algoritmo	**Tempo**	**Ocupação Processador**		**Ganho de Desempenho**
		ARM	DSP	
ARM	113.35ms	100%	0%	0%
DSP	106.47ms	62.4%	37.6%	6.1%
ARM | DSP	97.82ms	76.4%	23.6%	13.7%

A execução dos algoritmos de processamento de imagem no DSP não trazem muita vantagem em tempo de execução, justamento por causo do algoritmo ampliação que tem um tempo muito elevado em ralação ao ARM. Apesar de não obter um ganho significativo neste quesito, ele alivia a carga de processamento do ARM em 37.6%.

Se o processador DSP for utilizado apenas onde ele obtém o menor tempo (ARM | DSP), então o ganho de tempo sobe para 13.7%, enquanto que a carga no ARM aumenta para 76.4% do tempo da solução inicial (ARM).

Capítulo 5

Conclusão e Trabalhos Futuros

5.1 Conclusão

Neste trabalho foi analisada a viabilidade no uso do processadores heterogêneos através do MPSoC OMAP3530 que contém com um processador ARM e um processador DSP. Este MPSoC está disponível na placa de desenvolvimento embarcada *BeagleBoard*, onde foram instalados o sistema operacional Linux, o sistema de janelas X11 e o *device driver* para o processador DSP.

Um ampliador digital foi desenvolvido para testar o ambiente com os dois processadores. Ele é um sistema de tempo real que fica em *loop*, capturando, processando e exibindo a imagem de uma *webcam*. O tempo para este processamento é crítico pois influencia na taxa de *refresh* da tela e consequentemente na experiência do usuário. Com um tempo de processamento de 500ms para o *loop* da aplicação, por exemplo, é possível a taxa de duas atualizações por segundo, enquanto que se o tempo for de 100ms é possível chegar apenas a uma taxa de 10 atualizações por segundo.

De toda a implementação do software, a codificação dos algoritmos para processador DSP foi a parte que mais demandou tempo. Como não foi utilizado nenhum ambiente de emulação e não era possível o DSP emitir mensagens para o console, a depuração se tornou uma tarefa complexa.

No ganho de desempenho geral da aplicação com o uso do processador DSP, três abordagens diferentes no uso dos processadores foram feitas. A melhor delas, processando os algoritmos no processador que se mostrava mais eficiente para tal, permitiu uma redução de tempo de até 13.7%, quando comparado ao uso do ARM isoladamente, e uma redução de 23.6% na ocupação

do processador principal.

Em outra abordagem, utilizando o processador DSP para o processamento dos algoritmos e o ARM para captura e renderização de imagens, foi possível reduzir em 6.1% o tempo total da aplicação e em 27.6% a ocupação do processador hospedeiro (ARM). Esta técnica é bastante útil quando for necessário executar outra aplicação no ARM que concorra pela CPU.

Nem todos os algoritmos puderam extrair o máximo de desempenho do DSP, devido ao modo em que ele processa a entrada. Algoritmos que operam sobre potência de dois valores a cada passo do laço de repetição principal, podem ser facilmente modificados para utilizar as funções intrínsecas do DSP. Caso contrário, a melhor solução pode ser deixar que o ARM otimize o código.

Analisando o tempo de execução dos quatro algoritmos de processamento de imagem, o algoritmo brilho que usou o máximo dos recursos do DSP obteve o melhor desempenho. Outros dois algoritmos, que tinham apenas um laço de repetição e o código era semelhante ao do ARM, obtiveram menor ganho que o algoritmo brilho, porém, eles ainda foram mais rápidos que a execução somente com o ARM. O algoritmo zoom, que tem dois laços de repetições aninhados e código semelhante ao ARM, obteve o pior desempenho, sendo mais de duas vezes mais lento que o ARM. É válido ressaltar que o DSP opera a uma menor frequência que o ARM (520Mhz x 720Mhz).

Com isto, concluímos que, dependendo do tipo de aplicação que irá fazer uso do DSP, ela pode executar de forma mais rápida ou mais lenta que no ARM. Se essa não for uma restrição, é possível utilizar o DSP para amenizar a carga de trabalho no ARM e com isto permitir que outras aplicações executem de maneira mais eficiente nele.

Uma comparação dos tempos teóricos fornecidos pelo compilador do processador DSP e os tempo obtidos na prática foi feita. Como os tempos teóricos não levam em consideração falha de cache e acesso a memória principal, seus tempos ficaram muito distantes dos tempos reais.

Uma vantagem do DSP que não foi abordada neste trabalho é o consumo de energia, que é descrita na literatura como sendo um dos seus diferenciais. Logo, outra vantagem na utilização do DSP ocorreria em ambientes sem alimentação externa, típicos em ambientes embarcados.

5.2 Trabalhos Futuros

Como trabalhos futuros sugere-se:

1. Utilizar do ambiente CCS (*Code Compose Studio*) da *Texas Instruments* para aumentar a velocidade de desenvolvimento e depuração para o processador DSP. Com ele é possível emular a execução dos algoritmos, depura-los e obter estatísticas através de simulações e descobrir gargalos que levam o processador a um baixo desempenho.

2. Modelar uma interface gráfica (GUI) para o controle dos diversos parâmetros disponíveis no código já implementado. É possível também usar a I/O da *BeagleBoard* para se comunicar com a aplicação e alterar os parâmetros dela, evitando com isto o uso de mouse ou teclado, características importantes para um produto embarcado.

3. Paralelizar a execução dos algoritmos de tratamento de imagem entre os processadores ARM e DSP. O algoritmo que gerencia a paralelização divide a imagem original em duas porções, não necessariamente de tamanhos iguais, e envia cada parte para uma unidade de processamento. Ele detém o histórico dos tempos de processamento em cada unidade e faz o balanceamento de carga (quanto de cada imagem cada processador irá processar) em tempo de execução. Para otimizar este processo deve-se utilizar o método produtor-consumidor com um *buffer* que contém os processamentos demandados.

4. Analisar a relação entre os tempos teóricos e práticos obtidos com o DSP. Que elementos, precisamente, devem ser levados em conta na análise teórica para obter uma aproximação do tempo real. Há uma documentação vasta sobre as otimizações e análises do código DSP [32, 33], fornecida pela *Texas Instruments*, que pode ser utilizada. O emulador CCS pode ser de grande utilidade nesse estudo.

5. Abordar o consumo de energia no DSP para os algoritmos propostos neste trabalho. Ainda nesta linha, produzir um circuito de alimentação através de bateria. Um trabalho onde foi desenvolvido um kit de alimentação produzido para a *Beagleboard* pode ser encontrado em [34].

Referências Bibliográficas

[1] HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 0123704901.

[2] ARM. *Cortex-A8 Processor*. 2011. Consultado na INTERNET: http://www.arm.com/products/processors/cortex-a/cortex-a8.php.

[3] ARM. *The ARM Cortex-A9 Processors*. 2011. Http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

[4] LONGBOTTOM, R. *Dhrystone Benchmark Results On PCs*. 2011. Http://homepage.virgin.net/roy.longbottom/dhrystone%20results.htm.

[5] MARWEDEL, P. *Embedded System Design*. Netherland: Springer, 2006. ISBN 978-0-387-29237-3.

[6] KARKOWSKI, I.; CORPORAAL, H. Design space exploration algorithm for heterogeneous multi-processor embedded system design. In: *Proceedings of the 35th annual Design Automation Conference*. New York, NY, USA: ACM, 1998. (DAC '98), p. 82–87. ISBN 0-89791-964-5. Disponível em: <http://doi.acm.org/10.1145/277044.277060>.

[7] MANOLACHE, S.; ELES, P.; PENG, Z. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 7, p. 19:1–19:35, January 2008. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/1331331.1331343>.

[8] GHIASI, S.; KELLER, T.; RAWSON, F. Scheduling for heterogeneous processors in server systems. In: *Proceedings of the 2nd conference on Computing frontiers*. New

York, NY, USA: ACM, 2005. (CF '05), p. 199–210. ISBN 1-59593-019-1. Disponível em: <http://doi.acm.org/10.1145/1062261.1062295>.

[9] KOBAYASHI, Y. et al. Methodology for operation shuffling and l0 cluster generation for low energy heterogeneous vliw processors. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 12, September 2007. ISSN 1084-4309. Disponível em: <http://doi.acm.org/10.1145/1278349.1278354>.

[10] YUYAMA, Y. et al. An soc architecture and its design methodology using unifunctional heterogeneous processor array. In: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 2004. (ASP-DAC '04), p. 737–742. ISBN 0-7803-8175-0. Disponível em: <http://portal.acm.org/citation.cfm?id=1015090.1015290>.

[11] EL-MAHDY, A.; EL-SHISHINY, H. Efficient parallel selective separable-kernel convolution on heterogeneous processors. In: *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*. New York, NY, USA: ACM, 2010. (IFMT '10), p. 7:1–7:6. ISBN 978-1-4503-0008-7. Disponível em: <http://doi.acm.org/10.1145/1882453.1882463>.

[12] MARTIN, G. Overview of the mpsoc design challenge. In: *Design Automation Conference, 2006 43rd ACM/IEEE*. [S.l.: s.n.], 2006. p. 274 –279. ISSN 0738-100X.

[13] PAULIN, P. et al. Dsp design tool requirements for embedded systems: A telecommunications industrial perspective. *The Journal of VLSI Signal Processing*, Springer Netherlands, v. 9, p. 23–47, 1995. ISSN 0922-5773. 10.1007/BF02406469. Disponível em: <http://dx.doi.org/10.1007/BF02406469>.

[14] WOLF, W. *Computers as components: principles of embedded computing system design*. Morgan Kaufmann Publishers, 2005. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780123694591. Disponível em: <http://books.google.com/books?id=JjvSkmd8zsAC>.

[15] JúNIOR, V. C. Tecnologia soc e o microcontrolador psoc. *Revista Integração*, São Paulo, v. 1, n. 42, p. 251,257, Outubro 2005.

[16] FILHO, S. J.; PONTES, J.; LEITHARDT, V. Multiprocessor system on chip. In: . Porto Alegre: [s.n.].

[17] SLOSS, A.; SYMES, D.; WRIGHT, C. *ARM System Developer's Guide: Designing and Optimizing System Software*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558608745.

[18] TANENBAUM, A. S. *Structured Computer Organization (5th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005. ISBN 0131485210.

[19] HAM, J. A system-theory perspective for signal theory. *Circuit Theory, IRE Transactions on*, v. 3, n. 4, p. 208 – 209, dec 1956. ISSN 0096-2007.

[20] TAN, E. J.; HEINZELMAN, W. B. Dsp architectures: past, present and futures. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 31, p. 6–19, June 2003. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/882105.882108>.

[21] EYRE, J.; BIER, J. The evolution of dsp processors. *Signal Processing Magazine, IEEE*, v. 17, n. 2, p. 43 –51, mar 2000. ISSN 1053-5888.

[22] MOSHE, Y.; PELEG, N. Implementations of h.264/avc baseline decoder on different digital signal processors. In: *ELMAR, 2005. 47th International Symposium*. [S.l.: s.n.], 2005. p. 37 – 40.

[23] TEXAS INSTRUMENTS. *Fixed-Point Digital Signal Processor*. Texas, 2009.

[24] TEXAS INSTRUMENTS. *OMAPTM Technology Overview*. Texas, 2000.

[25] TEXAS INSTRUMENTS. *TMS320C64x/C64x+ DSP: CPU and Instruction Set - Reference Guide*. Texas, 2009.

[26] TEXAS INSTRUMENTS. *Cache Memory Architecture Overview*. Texas, 2009.

[27] TEXAS INSTRUMENTS. *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide*. Texas, 2002.

[28] TEXAS INSTRUMENTS. *TMS320C6000 Integer Division*. Texas, 2000.

[29] TEXAS INSTRUMENTS. *Developing Core Software Technologies for TI's OMAPTM Platform*. Texas, 2002.

[30] WIKI. *BeagleBoard Ubuntu*. 2011. Consultado na INTERNET: http://elinux.org/BeagleBoardUbuntu.

[31] BIDARRA CLODIS BOSCARIOLI, S. M. P. J. Avaliando a ferramenta xlupa como recurso para a educação especial inclusiva. In: *XX Simpósio Brasileiro de Informática na Educação*. Santa Catarina: [s.n.], 2009.

[32] TEXAS INSTRUMENTS. *TMS320C6000 Optimizing Compiler v 6.1*. Texas, 2008.

[33] TEXAS INSTRUMENTS. *TMS320C6000 Programmer's Guide*. Texas, 2010.

[34] ANDRADE, S. A. Implementação e avaliação de um cluster de beableboards ambientadas em linux utilizando mpi. *UFSC Universidade Federal de São Carlos*, São Carlos - SP, junho 2010.