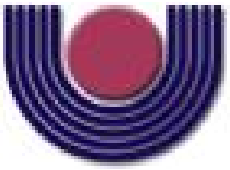

Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Utilização de instruções SIMD (NEON) no ARM para otimização do xLupa embarcado

Deivide Possamai

CASCAVEL
2014

Deivide Possamai

Utilização de instruções SIMD (NEON) no ARM para otimização do xLupa embarcado

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: Marcio Seiji Oyamada

CASCAVEL
2014

Deivide Possamai

Utilização de instruções SIMD (NEON) no ARM para otimização do xLupa embarcado

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

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

Prof. Guilherme Galante
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Jeferson José Baqueta
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 8 de dezembro de 2014

“Milagre? Algo como isso nunca vai acontecer se você sempre estiver esperando. Milagre é algo que você tem que trabalhar para merecer” Raimon Taro

“O importante é ganhar. Tudo e sempre. Essa história que o importante é competir não passa de demagogia” Ayrton Senna

AGRADECIMENTOS

Serei eternamente grato a meus pais Ervino Estevão Possamai e Zilda Lopes Pereira Possamai pelo apoio em todos os momentos da minha vida. Ao meu irmão Gabriel Possamai por sempre me apoiar nos momentos difíceis do TCC. Agradeço ao meu orientador Marcio Seiji Oyamada pelos 3 anos de projeto de IC, por orientar meu estágio e pela ajuda incondicional, proporcionando um imenso aprendizado que levarei por toda minha carreira profissional. Agradeço aos meus amigos de Toledo em especial ao Alison Fernando, André Felipe, Michel Weirich, Matheus Giaretta, André Pastório e Giovani Airton Timm que sempre me apoiaram e me proporcionaram vários momentos de felicidade. A amizade de vocês não há riqueza que pague.

Agradeço a todos os amigos e colegas que fiz durante estes quatro anos de graduação em especial ao Charles Giovane de Salles, Fernando Fernandes dos Santos, Thales Bertaglia, Marcelo Fudo Rech e João Paulo Colling. Agradeço também a todos os integrantes do PETComp e aos meus colegas de estágio. Serei eternamente grato a todos.

A todos vocês, muito obrigado.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABS	*Anti-lock Breaking System*
CISC	*Complex Instruction Set Computer*
CMYK	*Cyan, Magenta, Yellow, Black Key*
CPU	*Central Processing Unit*
EX	*Execution*
FPS	*Frames Per Second*
GPS	*Global Positioning System*
HSV	*Hue, Saturation, Value*
ID	*Instruction Decode*
IF	*Instruction Fetch*
IP	*Intelectual Property*
MEM	*Memory*
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MPSoC	*Multiprocessor System-on-Chip*
PC	*Program Counter*
PC	*Personal Computer*
RGB	*Red, Green, Blue*
RISC	*Reduced Instruction Set Computer*
SIMD	*Single Instruction Single Data*
SISD	*Single Instruction Single Data*
TLB	*Translation Look Aside Buffer*
V4L	*Video For Linux*
VLIW	*Very Long Instruction Word*
WB	*Write-Back*

Sumário

Resumo

O uso dos sistemas embarcados vem crescendo e se tornando cada vez mais presentes na vida das pessoas. Contudo, para desenvolver aplicações para esses sistemas, é necessário levar em consideração alguns requisitos como a área de aplicação, o processamento disponível, a quantidade de energia gasta, o tamanho da plataforma e o preço. Todos esses fatores influenciam na qualidade do produto final. Este trabalho de conclusão de curso apresenta otimizações realizadas em uma lupa digital para usuários baixa visão denominada xLupa embarcado. O objetivo deste trabalho é explorar o paralelismo utilizando instruções SIMD (*Single Instruction Multiple Data*), chamadas de NEON disponíveis nos processadores da família ARM Cortex-A. O trabalho foca em duas partes: a conversão de cores YUV4:2:2 para RGB24 e o aprimoramento do método de diferença de imagens. Com o uso das instruções NEON, o xLupa obteve um aumento em média de 8,48% na sua taxa de *frames* por segundo. O método de diferença de imagens não obteve um ganho significativo.

Palavras-chave: SIMD (*Single Instruction Multiple Data*), software embarcado, processador embarcado, NEON.

Capítulo 1

Introdução

1.1 Contextualização

O projeto xLupa é um ampliador de tela inteligente, voltado, mas não de uso exclusivo, para pessoas com baixa visão, que vem sendo desenvolvido pelo grupo GIA da UNIOESTE [1]. O xLupa embarcado é uma variação do ampliador de tela visando prover uma plataforma móvel para ampliação de documentos impressos para pessoas com baixa visão. Ele consiste de uma plataforma embarcada e uma *webcam* que faz a captura do texto impresso, realiza um pré-processamento e envia a saída para uma TV ou monitor. Os sistemas embarcados (como o xLupa) podem ser definidos como sistemas de processamento de informação incorporados em produtos como carros, telecomunicação ou em equipamentos eletrônicos [2]. Segundo [2] o número de processadores nos sistemas embarcados já excede o número de processadores em PCs (*Personal Computers*).

1.2 Motivação

Embora os sistemas embarcados estejam cada vez mais presentes na vida das pessoas, existe uma dificuldade em desenvolver aplicações para esse tipo de sistema, pois são diversas variáveis que influenciam no desempenho e qualidade do produto final. Além disso, nem sempre é fácil explorar o paralelismo existente nas aplicações, pois deve-se considerar fatores como a comunicação e sincronização. Entretanto para aplicações multimídia onde existe uma independência no nível de dados pode-se utilizar o paralelismo comumente chamado de SIMD, que será apresentado com mais detalhes na Seção 2.6.2.

A motivação desta pesquisa surgiu de um trabalho anterior [3], onde foi verificado que o

formato de captura escolhido pelo programador tem um impacto direto no desempenho. A *web-cam* utilizada no projeto[1] suporta dois tipos de formato YUV4:2:2 e MJPEG. Tanto o xLupa embarcado quanto o GTK utilizam o formato RGB24 para realizar o pré-processamento e apresentação na tela. Desta forma, quando é solicitado o formato RGB24, o driver seleciona o formato MJPEG para captura, e faz a conversão internamente para RGB24. No trabalho anterior, foi detectado que se a captura fosse realizada em YUV4:2:2 e a conversão para RGB24 fosse feita na aplicação, resultaria em uma execução aproximadamente 20% mais rápida. Esta diferença é resultado da maior complexidade de decodificação do formato JPEG se comparado com a conversão YUV4:2:2 para RGB. Desta forma, optou-se por capturar *frames* em formato YUV4:2:2 e realizar a conversão para RGB explicitamente no xLupa embarcado.

Além disso, em [3] foi aplicado um algoritmo de diferença de imagens que tinha como objetivo evitar o processamento desnecessário quando não houvesse a necessidade de atualizar da saída na tela. Nesse método a cada nova captura é realizado um cálculo de diferença *pixel* a *pixel* entre a imagem anterior e a atual. Assim, a captura da imagem é feita continuamente em YUV4:2:2 fazendo a conversão explícita para RGB24 apenas quando o limiar de diferença for maior que 80%. Após a conversão, o xLupa faz a aplicação do perfil e por fim envia para um monitor ou uma TV. O perfil é uma configuração (*zoom*, contraste verde, constraste em cinza, etc) escolhida pelo usuário ao utilizar o xLupa.

1.3 Objetivos

O objetivo deste trabalho é explorar o paralelismo em nível de dados no xLupa Embarcado através da utilização de instruções SIMD, disponíveis no processador ARM Cortex-A8, visando aumentar o desempenho do xLupa Embarcado que na sua versão original é de aproximadamente 3,135 *frames* por segundo [3](o ideal para seria próximo de 25 *frames* por segundo, pois é uma taxa próxima a padrões como NTSC e PAL). As instruções SIMD serão utilizadas em dois pontos principais, na otimização do algoritmo de conversão de cores e no método de diferença de imagens.

[1] Microsoft HD-5000 [4]

1.4 Organização do Texto

Este capítulo inicial demonstrou os objetivos e as motivações deste trabalho. No Capítulo 2 são apresentados conceitos sobre arquitetura de computadores, como surgiram as arquiteturas RISC e CISC e suas diferenças, conceitos sobre paralelismo em nível de instrução assim como seus problemas e limites e conceitos sobre a taxonomia de Flynn. Além disso, neste mesmo capítulo será apresentada a arquitetura ARM, e as características do processador ARM Cortex-A8 e das instruções NEON.

No Capítulo 3 são abordados conceitos sobre sistemas embarcados e suas características. Além disso, apresenta-se a plataforma embarcada utilizada neste trabalho e o xLupa embarcado e suas características, como o sistema operacional e a câmera utilizada.

No Capítulo 4 são apresentados os resultados do método aprimorado com as instruções NEON e do método aprimorado de diferença de imagens. São feitas comparações de ambas as implementações, com a versão anterior do xLupa Embarcado. No Capítulo 5 são apresentados as conclusões e trabalhos futuros. O Apêndice A apresenta os conceitos sobre os espaços de cores, mais especificamente sobre o RGB24 e o YUV4:2:2. No Apêndice B encontra-se os códigos do método de conversão YUV4:2:2 para RGB24 original escrito em C, o otimizado utilizando instruções NEON, o de diferença de imagens sequencial e com NEON. No Apêndice C encontra-se os resultados relativos a diferença de imagens para todos os perfils.

Capítulo 2

Arquitetura de Computadores

2.1 CISC vs RISC

Arquitetura de computadores refere-se aqueles atributos que são visíveis ao programador, ou seja, aqueles atributos que possuem influência sobre a execução lógica do programa [5]. Entre os exemplos de atributos arquiteturais estão: o conjunto de instruções, mecanismos de E/S (Entrada/Saída), técnicas para o endereçamento de memória, entre outros [5].

Nessa área existem dois conceitos que historicamente registraram o tipo de arquitetura que os computadores possuem. São elas a arquitetura CISC e RISC [6]. A arquitetura CISC (*Complex Instruction Set Computer*), é uma arquitetura que como o próprio nome diz possui um conjunto complexo de instruções [6]. Essa arquitetura surgiu nos anos 70. Nessa época, o *software* era mais caro que o *hardware*. Dessa forma, os projetistas pensaram em transferir algumas funcionalidades mais utilizadas em *softwares* para o *hardware* [6].

> "As principais razões para a adoção dessa arquitetura foi: reduzir as dificuldades de escrita de compiladores, reduzir o custo global do sistema, reduzir os custos de desenvolvimento de *software*, reduzir drasticamente o *software* do sistema, reduzir a diferença semântica entre linguagens de programação de máquina, fazer com que os programas escritos em linguagens de alto nível corressem mais facilmente, melhorar a compactação do código e facilitar a detecção e correção de erros" [6].

Por outro lado a arquitetura RISC (*Reduced Instruction Set Computer*) surgiu em contrapartida da complexidade gerada pela arquitetura CISC, que apresentou problemas inerentes ao microcódigo cada vez maiores e difíceis de depurar e testar [6]. Além desses motivos, a transfe-

rência da complexidade do *software* para o *hardware* usufruía dos poucos recursos disponíveis da época: os transistores [6]. A ideia do RISC é fazer o contrário, transferir a complexidade do *hardware* para o *software* de maneira que as instruções fossem rápidas e pequenas [6]. Seu surgimento também está ligado a estudos sobre quais instruções são mais frequentes em um programa de alto nível.

O resultado desses estudos mostraram que instruções de atribuição e condicionais são bastante utilizadas, sugerindo uma implementação otimizada para elas [5]. Em relação aos operandos, foi verificado que eles são utilizados com bastante frequência, comprovando a importância de se utilizar uma arquitetura que provê um acesso rápido a eles [5]. Outros estudos mostraram que a chamada de procedimentos em linguagens de alto nível consomem bastante tempo, comprovando assim, a importância do desenvolvimento dessas operações de forma eficiente [5]. Dessa forma, surgiram-se três elementos que caracterizaram a arquitetura RISC: a utilização de um grande número de registradores ou o uso de um compilador para a otimizar o uso de registradores, a implementação de um *pipeline* de instruções eficiente e a utilização de um conjunto de instruções reduzido [5].

2.2 Características da Arquitetura RISC

A arquitetura RISC possui um conjunto reduzido de instruções que possuem algumas características como [5]:

a) *Uma instrução por ciclo*;

b) *Operações registrador-para-registrador*;

c) *Modos de endereçamento simples*;

d) *Formatos de instruções simples* [5].

O uso de uma instrução por ciclo diminui quase totalmente a necessidade de microcódigo, pois as instruções devem ser simples e executar quase ao mesmo tempo. As operações devem ser em sua maioria de registrador para registrador, deixando apenas operações básicas como *load* e *store* para o acesso a memória. Isso acaba simplificando o conjunto de instruções e consequentemente a unidade de controle do processador. Outra característica é o uso de modos

de endereçamento simples. Modos mais complexos de endereçamento podem ser incluídos também, por exemplo, a geração de cálculo de endereço via *software*. Por fim o RISC possui formatos de instruções simples onde o tamanho da instrução é fixo assim como os seus opcodes. Essas características também simplificam a unidade de controle [5].

Essa arquitetura também é caracterizada pela utilização de um *pipeline* de instruções que será detalhado na Seção 2.3.

2.3 Paralelismo em Nível de Instruções

O *pipelining* de instruções é uma técnica que explora os estados intermediários de uma instrução para tirar proveito do paralelismo por meio da sobreposição de instruções. O *pipeline* é análogo a uma linha de montagem. Na linha de montagem, existem vários estágios que contribuem para a construção de um carro. Cada vez que um estágio é terminado, o estágio anterior fica ocioso, podendo assim ser utilizado na construção de outro carro. Essa é a ideia do *pipeline* de instruções que gera uma redução no tempo médio de execução por instrução, tendo a vantagem de não ser visível ao programador [7].

Em [7] é apresentado um *pipeline* de 5 estágios baseado na arquitetura RISC. O primeiro é o ciclo de busca de instrução (*Instruction Fetch* - IF) que utiliza o *Program Counter* para fazer a busca da instrução na memória. O segundo é a decodificação de instrução/ciclo de busca de registradores (*Instruction Decode* - ID), que decodifica a instrução e lê os registradores correspondentes. O terceiro ciclo é o de execução/ciclo de endereço efetivo (*Execution* - EX) que utiliza a unidade lógica e aritmética para fazer as operações sobre os operandos especificados na instrução. O quarto ciclo é o de acesso à memória (*Memory* - MEM) que verifica se é um *load* ou um *store* para fazer a leitura ou a gravação respectivamente. O quinto e último ciclo é o de *write-back* (*Write-Back* - WB) que armazena o resultado no banco de registradores. A Figura 2.1 mostra um fluxo simples de instruções sendo executado, a qual espera todos os ciclos se completarem para iniciar a próxima instrução. Já a Figura 2.2 mostra um fluxo com um *pipeline* de instruções.

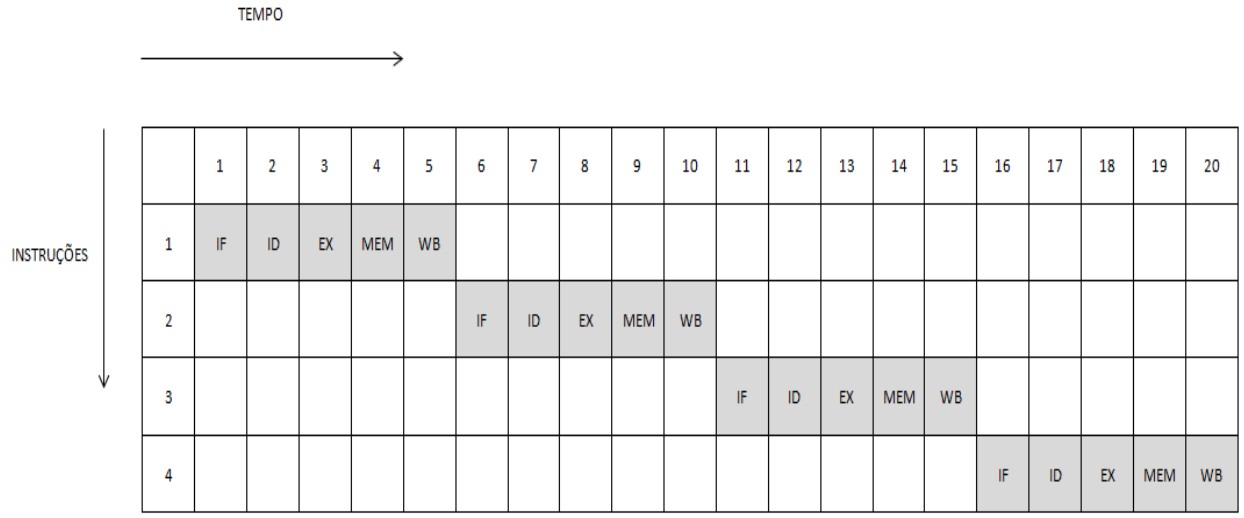

Figura 2.1: Fluxo de instruções simples [5]

Essa implementação baseada em um fluxo simples espera até o último ciclo ser executado para então começar outra instrução. No exemplo da Figura 2.1 com cinco estágios utilizando quatro instruções, é necessário vinte unidades de tempo para a execução completa das instruções. Na implementação com *pipeline*, quando um ciclo é completado e fica ocioso, a próxima instrução pode utilizar desse ciclo para executar sem interferir na instrução anterior. A Figura 2.2 mostra o mesmo exemplo apresentado anteriormente, porém com *pipeline*. são necessárias apenas oito unidades de tempo para a execução completa das 4 instruções.

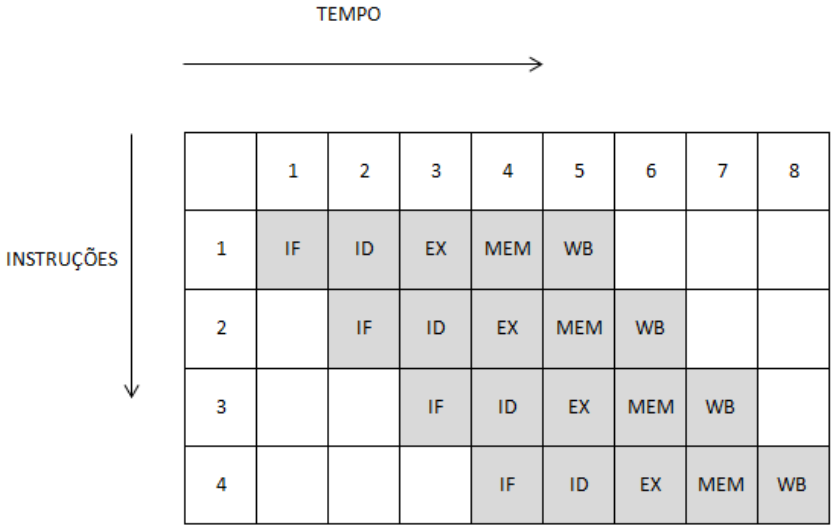

Figura 2.2: Fluxo de instruções com *Pipeline* [5]

2.4 Dependências no Pipeline

Apesar de aumentar o rendimento da CPU, o *pipeline* acaba por aumentar e não diminuir o tempo de execução individual de cada instrução, devido ao tempo gasto no controle do *pipeline* [7]. Assim alguns problemas começam a aparecer como desequilíbrio entre os estágios de *pipeline* e *overhead* do *pipelining*. O primeiro causa redução no desempenho, pois o *clock* fica dependente do estágio de *pipeline* mais lento. O segundo é consequência do atraso nos registradores de *pipeline* e do *clock skew* que é "o atraso máximo da chegada do *clock* a dois registradores" [7]. Além desses problemas, o *pipeline* na realidade, deve considerar que as instruções podem depender umas das outras causando "*Hazards*" de *pipeline*.

Hazards são situações que impedem a execução normal do *pipeline* diminuindo o ganho de velocidade obtido pelo mesmo. Existem três classes de *Hazards* [7]:

a) a) *Hazards* estruturais ocorrem quando há conflitos de recursos, onde o *hardware* não pode aceitar todas as combinações possíveis de instruções simultaneamente na execução sobreposta;

b) b) *Hazards* de dados surgem quando uma instrução depende dos resultados de uma ins-

trução anterior, impedindo a execução do *pipeline*;

c) c) *Hazards* de controle surgem através de *pipelining* de desvios e outras instruções que acabam mudando o PC (*Program Counter*).

O *Hazard* estrutural ocorre, por exemplo, quando um *pipeline* quer realizar duas operações de escrita em um ciclo de *clock*, em um processador que possui apenas uma porta de escrita no banco de registradores. Desta forma o *pipeline* é adiado em um ciclo de *clock*.

Os *Hazards* estruturais são permitidos pelos projetistas, pois utilizar o *pipeline* de todas as unidades funcionais ou sua duplicação pode ser cara. Assim se esse problema for raro pode não ser viável o custo de evitá-lo [8]. Um exemplo de *Hazard* de dados é mostrada na Figura 2.3 [7]:

```
1  DADD    R1,R2,R3
2  DSUB    R4,R1,R5
3  AND     R6,R1,R7
4  OR      R8,R1,R9
5  XOR     R10,R1,R11
```

Figura 2.3: *Hazard* de Dados [7]

É possível perceber na Figura 2.3 que as instruções a partir da linha 2 dependem todas do resultado do valor armazenado em R1. Isso causa um *Hazard* de dados. Para minimizar problemas como esse, é utilizado um método chamado de adiantamento (do inglês *forwarding*), a Figura 2.4 mostra um exemplo de adiantamento.

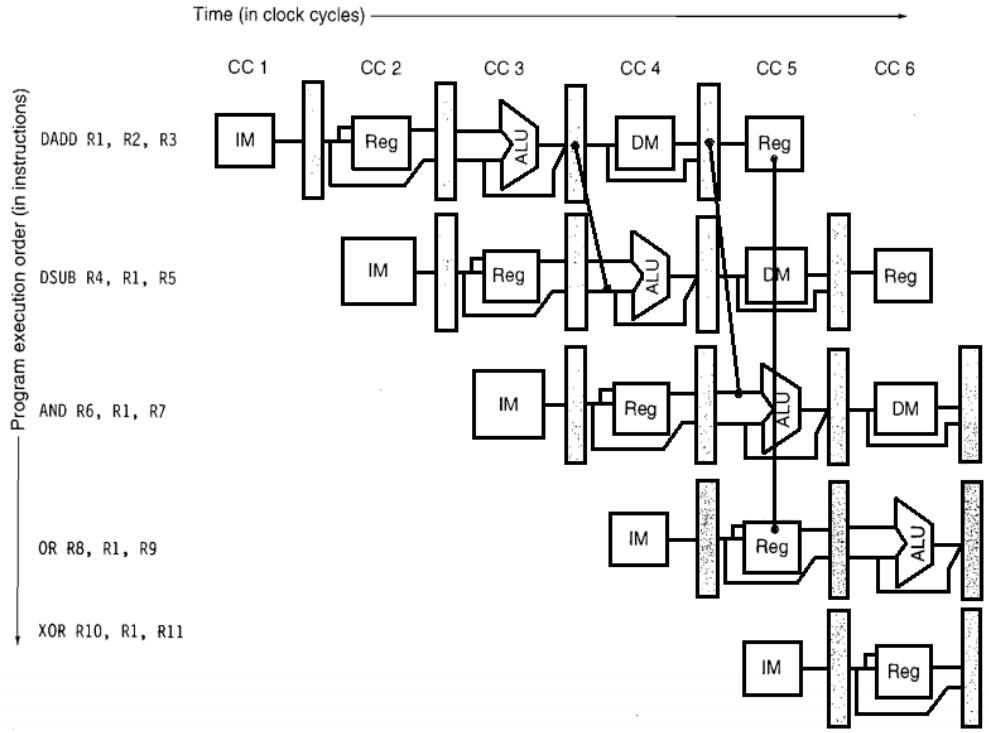

Figura 2.4: Exemplo da utilização de adiantamento [7]

Na Figura é verificado que a instrução DSUB na linha 2 não necessita imediatamente do resultado de DADD. Dessa forma são utilizados registradores temporários onde é possível pegar o valor já carregado da primeira instrução e colocar no registrador de *pipeline* da segunda instrução antes do cálculo aritmético, evitando um *stall* (parada do *pipeline*) [7].

Entretanto, no *hazard* de dados há casos em que o adiantamento não resolve como demonstra a Figura 2.5 [7].

```
1  LD      R1,0(R2)
2  DSUB    R4,R1,R5
3  AND     R6,R1,R7
4  OR      R8,R1,R9
```

Figura 2.5: *Hazard* de dados que exige *stall* [7]

A instrução LD carrega dados para R1 mais tardiamente no ciclo de *pipeline* não conseguindo repassar esses dados previamente para DSUB. A Figura 2.6 mostra esse problema.

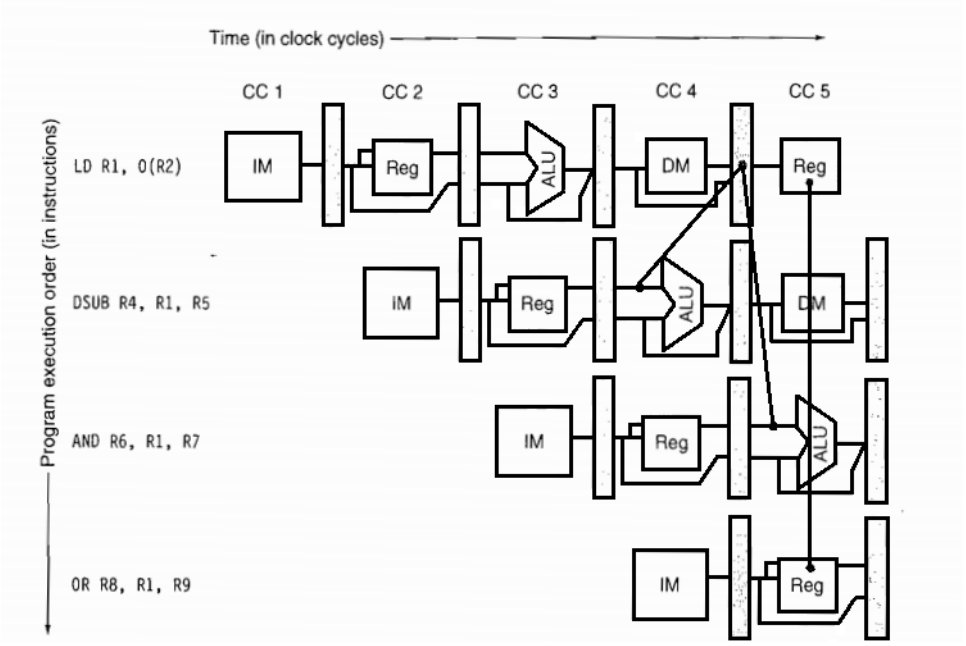

Figura 2.6: Exemplo onde o adiantamento não resolve a dependência de dados [7]

Esse tipo de instrução não pode ser tratada pelo adiantamento, pois segundo [7] “isso significaria encaminhar o resultado em ‘tempo negativo’ ”,“uma capacidade ainda não disponível aos projetistas de computadores”[7]. É necessário a utilização de um *hardware* chamado de *interlock*. O *interlock* detecta o *Hazard* e atrasa o *pipeline* introduzindo um *stall* como no *hardware* estrutural[7].

Quando um desvio é executado, ele pode alterar ou não o PC da aplicação. Se não alterar, o desvio é dito tomado. Se alterar o desvio é dito não-tomado. O problema dos desvios é que o *pipeline* não sabe quais as instruções ele deve carregar antes da execução do desvio. As consequências disso são *stalls* de cinco estágios no pipeline(no caso do *pipeline* de 5 estágios). Para resolver o problema de *hazard* de desvio, várias abordagens são utilizadas. A mais simples delas é fazer o congelamento ou o esvaziamento do *pipeline*, podendo manter ou excluir posteriores instruções após o desvio até que seu destino seja conhecido. A vantagem dessa abordagem é a simplicidade tanto para o *hardware* quanto para *software* [7].

Outra alternativa é tratar o desvio como não tomado ou como tomado. Em ambos os casos o compilador pode melhorar o desempenho conforme a necessidade do *hardware* [7]. É possível ainda utilizar preditores de desvios, que podem ser de dois tipos: estáticos ou dinâmi-

cos. Os preditores estáticos dependem de informações disponíveis em tempo de compilação. Já os preditores de desvios dinâmicos dependem do comportamento do programa durante sua execução.

2.5 Limites do Paralelismo em Nível de Instrução

Nem sempre é possível explorar totalmente o paralelismo em nível de instrução disponível em uma aplicação. Isso porque existem problemas como os *Hazards* de *pipeline* que diminuem esse ganho. Dessa forma, ao desenvolver um *pipeline* deve-se analisar seu custo para justificar seu uso [7]. Devido a esses problemas e a outros relacionados ao desempenho, espaço e potencia dissipada em um processador que dificultam a exploração do paralelismo em nível de instrução, [7] afirma que "o desempenho hoje é o fardo dos programadores", ou seja, os programadores já não podem mais depender dos projetistas de *hardware* para melhorar o desempenho de suas aplicações. Assim, é necessário que os programadores explorem o paralelismo de seus programas se quiserem um aumento no desempenho.

2.6 Taxonomia de Flynn

Em sistemas de computação, o paralelismo pode ser explorado de diversas maneiras, porém ele depende do quanto uma aplicação pode ser paralelizável. Dentro desse contexto, [8] introduziu uma taxonomia para caracterizar diferentes tipos de paralelismo disponíveis. A Figura 2.7 mostra esse esquema:

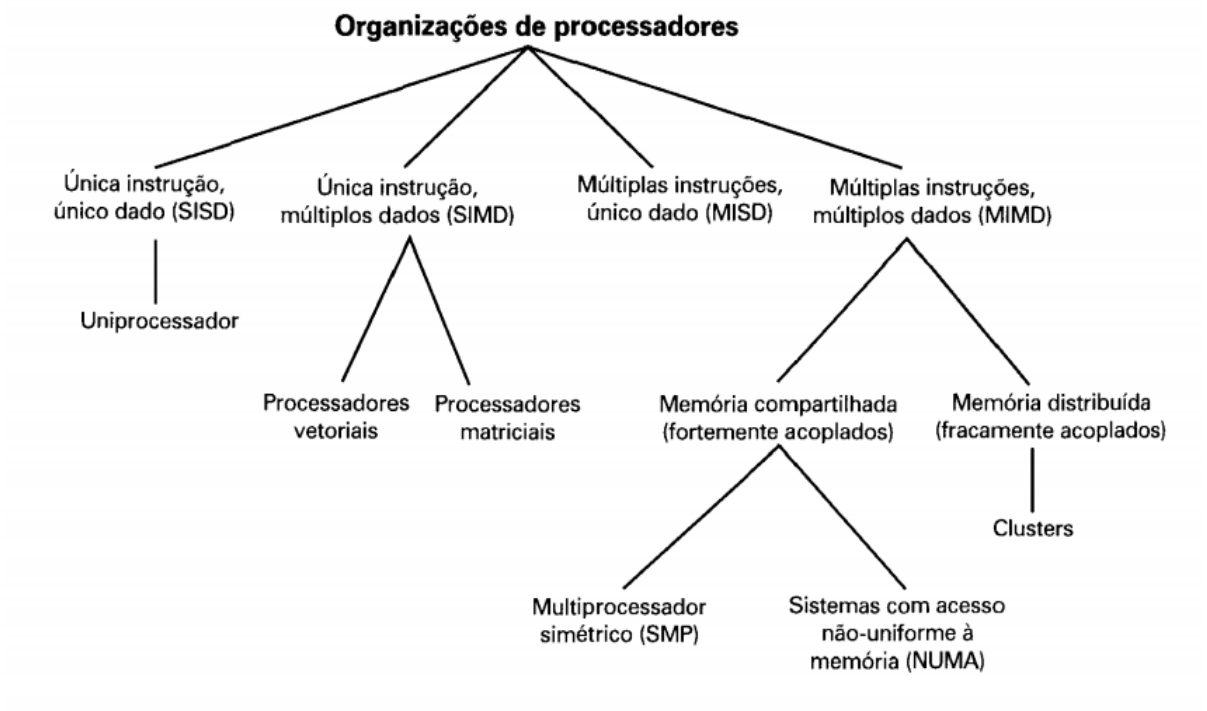

Figura 2.7: Taxonomia de Flynn. Adaptado de [5]

Os processadores são organizados conforme a forma que executam suas instruções. Processadores que executam uma única instrução e um único dado, também chamados de SISD (*Single Instruction Single Data*) são os uniprocessadores. Aqueles que executam uma única instrução sobre múltiplos dados são chamados de SIMD (*Single Instruction, Multiple Data*) no qual se enquadram os processadores vetoriais e matriciais. Existem também uma organização chamada de MISD (*Multiple Instruction, Single Data*) que utiliza múltiplas instruções para processar um único dado. E por fim existem processadores que executam múltiplas instruções sobre múltiplos dados MIMD (*Multiple Instruction, Multiple Data*), estes são divididos em processadores com memória compartilhada (fortemente acoplados) e memória distribuída (fracamente acoplados, por exemplo *Clusters*). Os processadores com memória compartilhada são divididos em duas classes: os multiprocessadores simétricos (SMP - *Symmetric Multi-Processing*) e os sistemas com acesso não-uniforme à memória (NUMA - *Non-Uniform Memory Access*) [5].

2.6.1 SISD

Nas arquiteturas SISD (*Single Instruction Single Data*), as instruções operam sobre dados escalares. Nesta arquitetura são utilizadas técnicas que exploram o paralelismo em nível de instrução, como o *superscalar* (“máquina que é projetada para melhorar o desempenho da execução de instruções escalares, através da execução de instruções independentes e concorrentes em *pipelines* diferentes”) e o VLIW (*Very Long Instruction Word*) (“arquitetura que coloca várias instruções em uma única palavra”, normalmente construída pelo compilador) [5].

2.6.2 SIMD

O SIMD (*Single Instruction, Multiple Data*) é um conjunto de instruções que permite fazer operações sobre dados matriciais e vetoriais onde existem parcelas independentes de dados que podem ser operados separadamente na mesma instrução [5]. Dessa forma uma única instrução pode lançar várias operações de dados, sendo potencialmente mais eficiente em questão de energia comparado com a MIMD (*Multiple Instruction, Multiple Data*) o qual precisa buscar e executar uma instrução por operação de dados [7].

Uma vantagem desse tipo de arquitetura é que os compiladores podem dizer aos programadores se uma seção de código será ou não vetorizada dizendo muitas vezes o motivo de ele não ter sido vetorizado no código. Isso permite que o programador aprenda a identificar no seu código trechos que podem ser melhorados para que o compilador possa gerar um código otimizado. Além disso, comparado ao MIMD, o SIMD possui a vantagem de que o programador continua a pensar sequencialmente mesmo utilizando operações paralelas. A desvantagem dessa arquitetura é que o programador precisa pensar em como extrair o paralelismo a nível de dados [7]. A Figura 2.8 mostra o esquema SIMD.

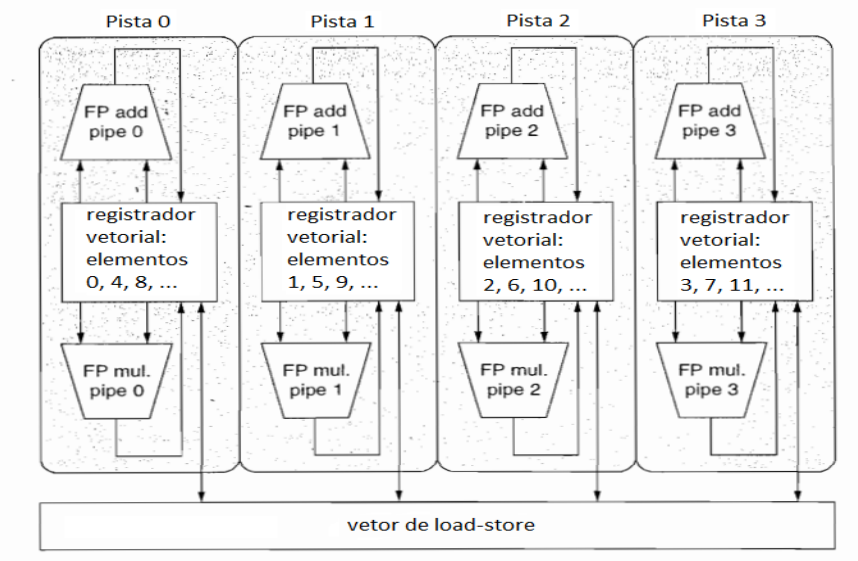

Figura 2.8: *Single instruction multiple data* [7]

O trabalho correlato obtido em [9] apresenta resultados relativos a utilização das instruções NEON. Neste caso, a ideia foi realizar a otimização no decodificador de video H264, explorando o paralelismo existente entre os *pixels* nas etapas do processo de decodificação. Para os testes, foram utilizadas amostras de sequencia de vídeo públicas como o *Foreman*, *Akiyo* e o *Mobile-Calendar*. O ganho obtido em FPS para o *Akiyo*, *Foreman* e o *mobile-calendar* foram de 1,68, 2,12 e 1,91 respectivamente. Os dados apresentados mostram ganhos consideráveis, sem a necessidade de reorganização do fluxo de execução que seria necessário caso um ambiente multiprocessado fosse utilizado.

2.6.3 MISD

O MISD (*Multiple Instruction, Single Data*) utiliza várias instruções sobre um único dado. Neste caso, várias unidades funcionais realizam diferentes operações sobre o mesmo conjunto de dados. Essa arquitetura não é implementada comercialmente [5] [8]. A Figura 2.9 mostra seu funcionamento.

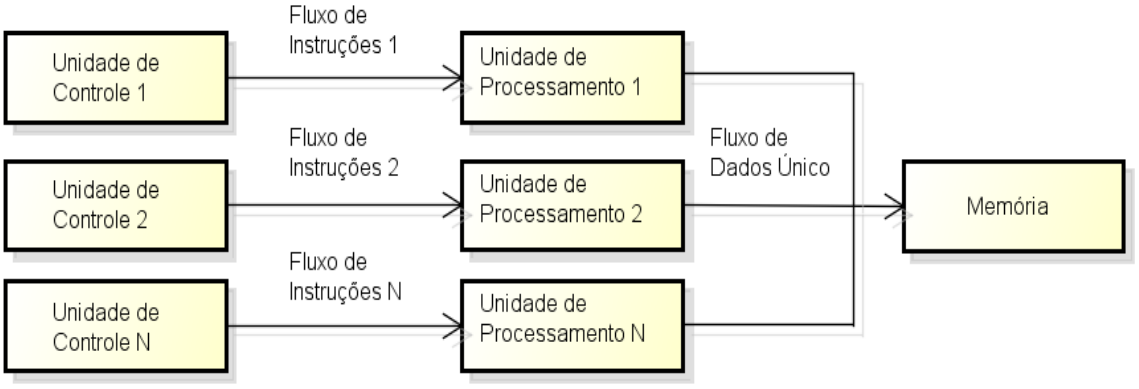

Figura 2.9: MISD [5]

2.6.4 MIMD

O MIMD (*Multiple Instruction, Multiple Data*) utiliza várias instruções sobre vários dados. Essa arquitetura consiste de vários processadores conectados que executam independentemente um mesmo programa. Ao implementá-la é necessário levar em consideração dois problemas significantes: a consistência da memória e a coerência de cache. A consistência da memória deve ser verificada pelo programador da aplicação, controlando a concorrência entre os fluxos de execução sobre dados compartilhados. A coerência de cache, no entanto, necessita de sincronização por parte dos processadores. A consistência da memória pode ser resolvida por *hardware* ou por *software*. A coerência de cache apenas por técnicas de *hardware* [8]. A Figura 2.10 apresenta o MIMD com memória compartilhada:

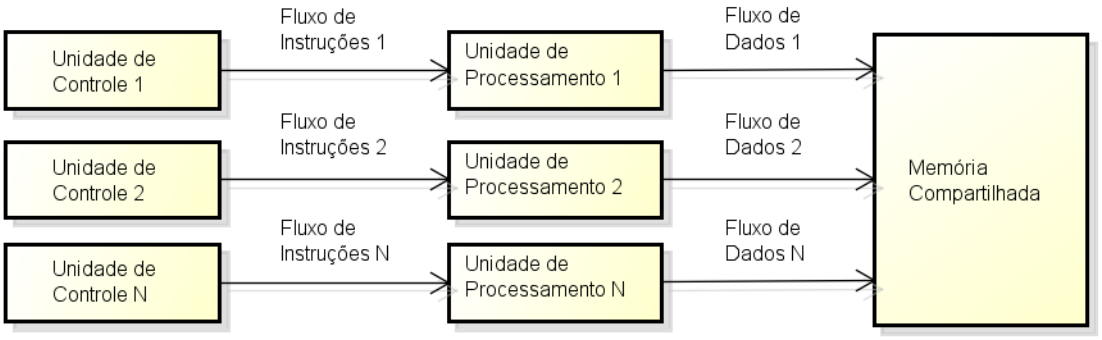

Figura 2.10: MIMD [5]

2.7 Processador ARM Cortex-A8

A arquitetura ARM é produzida pela empresa ARM Holdings, com sede em Cambridge no Reino Unido. A ARM não produz chips semicondutores, ela faz o *design* e licencia seu IP (*Intelectual Property* - Propriedade Intelectual) para fabricantes de semicondutores [10].

O ARM Cortex-A8 é um processador de alto desempenho, de baixa potência que fornece capacidade total de memória virtual [11]. Baseado na arquitetura ARMv7, o ARM Cortex-A8 pode regular a frequência do *clock* de 600MHz a mais de 1 GHz. A Figura 2.11 apresenta o *pipeline* do ARM Cortex-A8.

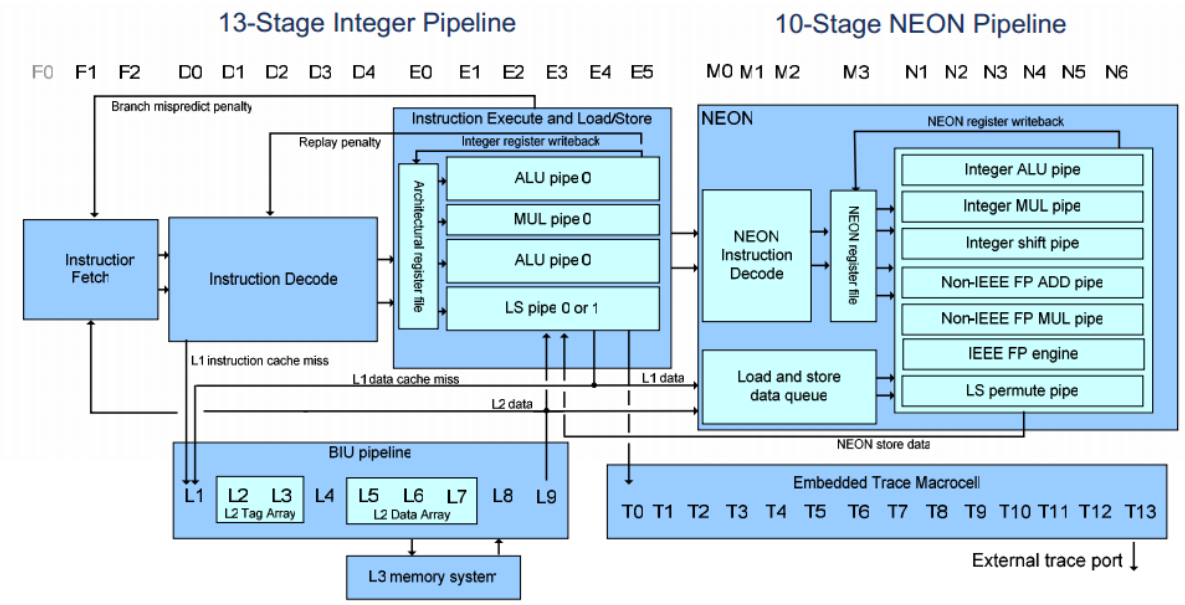

Figura 2.11: *Pipeline* do ARM Cortex-A8 [12]

Esse processador possui um *pipeline* de 3 fases, busca de instrução (*Instruction Fetch*), decodificação de instrução (*Instruction Decode*) e execução de instrução (*Instruction Execute*), divididos em 13 estágios. A busca de instrução utiliza 3 ciclos, a decodificação de instrução utiliza 4 ciclos e a execução de instrução utiliza 6 ciclos [7]. Além disso, o ARM Cortex-A8 possui a unidade NEON utilizada para operações vetoriais (SIMD). Essa unidade possui um *pipeline* de 10 estágios [5]. A tecnologia NEON será apresentada na Seção 2.7.1.

2.7.1 Tecnologia NEON

O conjunto de instruções NEON é uma tecnologia da ARM para SIMD, feito especialmente para operações multimídia. O NEON utiliza registradores de 64 e 128 *bits* para o processamento das informações [13]. A Figura 2.12 mostra o registrador de 128 *bits* do NEON e algumas das maneiras de se acessar seus dados. O registrador de 128 *bits* pode ser utilizado para armazenar e computar diversos tipos de dados como 16 inteiros de 8 *bits*, 8 inteiros de 16 *bits* , 4 inteiros de 32 *bits*, 4 valores de ponto flutuante de 32 *bits* ou 2 valores de ponto flutuante de 64 *bits* [14].

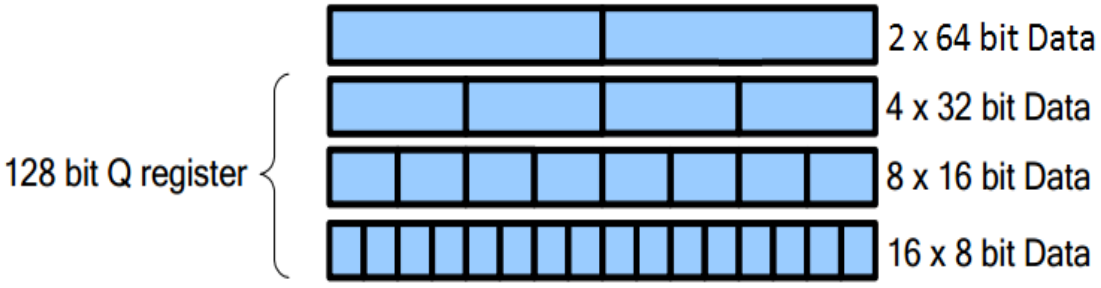

Figura 2.12: Registradores de 128 *bits* [13]

O SIMD como apresentado na Seção 2.6.2, é a utilização de uma única instrução para o processamento de múltiplos dados. Esse conjunto de instruções possui várias operações que envolvem vetores como soma, subtração, multiplicação, multiplicação por escalar, entre outros [15]. A Figura 2.13 mostra como é feito uma operação NEON.

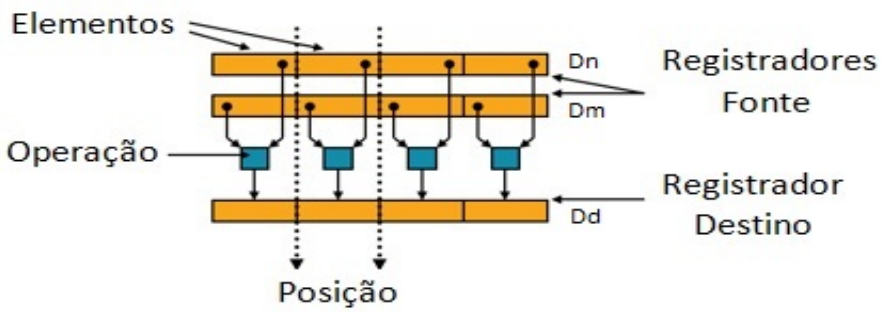

Figura 2.13: Exemplo de operação sobre vetores [16]

Cada registrador fonte (*source*) possui vários elementos (*elements*), onde para cada posição do vetor (*lane*), é feito uma operação (*operation*) sobre os elementos de cada registrador (soma, subtração, etc) e armazenado em um registrador destino (*destination register*), podendo operar sobre diferentes tipos de dados como *unsigned int*, *signed int*, *float* e *poly* (polinomial). No NEON, a declaração do tipo dos vetores bem como seu tamanho e número de posições é definida conforme a Figura 2.14 [17]:

```
<type><size>x<number of lanes>_t
```

Figura 2.14: Padrão para a utilização das *intrinsics* NEON [16]

Onde *type* representa o tipo (*int*, *uint*, *float* e *poly*), *size* representa o tamanho (8, 16, 32, 64) e o *number of lanes* é a quantidade de posições do vetor. A tabela 2.1 mostra os tipos disponíveis [17]:

Tabela 2.1: Tipos de Dados [16]

int8x8_t	int8x16_t
int16x4_t	int16x8_t
int32x2_t	int32x4_t
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_
poly8x8_t	poly8x16_t
poly16x4_t	poly16x8_t

Além disso, algumas instruções podem conter um *array* de vetores, que representam um conjunto de registradores NEON, sendo definidas da conforme a Figura 2.15 [17]:

```
<type><size>x<number of lanes>x<length of array>_t
```

Figura 2.15: Padrão para a utilização das *intrinsics* NEON com *array* [17]

Um exemplo é a declaração *int16x4x2_t* que é definida pela estrutura mostrada na Figura 2.16 [17]:

```
1  struct int16x4x2_t
2  {
3      int16x4_t val[2];
4  };
```

Figura 2.16: Instrução *int16x4x2_t* [17]

Essa estrutura armazena dois vetores de 16 *bits* sinalizado de tamanho 4. O acesso ocorre através da utilização da variável e do campo *val* da estrutura, por exemplo, uma variável definida como *int16x4x2_t* foo, pode acessar o primeiro vetor da seguinte forma: *foo.val[0]*.

A Tabela 2.2 mostra as instruções utilizadas nesse trabalho e quais são suas funções.

Tabela 2.2: Instruções Utilizadas [16]

Instrução	Descrição
vld4_u8	Faz a extração de dados da memória e armazena em registradores (vetores) com intercalação de 4 componentes.
vld2_u8	Faz a extração de dados na memória e armazena em registradores (vetores) com intercalação de 2 componentes.
vzip_s16	Faz a intercalação entre os componentes de um vetor
vcombine_s16	Combina dois vetores distintos em um único vetor
vmovl_u8	Reinterpreta os dados entre um vetor de tamanho 8 para outro com tamanho 16.
vaddq_s16	Faz a adição entre dois vetores 16 *bits* de tamanho 8.
vsubq_s16	Faz a subtração entre dois vetores 16 *bits* de tamanho 8.
vdupq_n_s16	Inicializa um vetor com uma constante
vshrq_n_s16	Efetua um *shift right* em um vetor com a quantidade especificada
vshlq_n_s16	Efetua um *shift left* em um vetor com a quantidade especificada
vmaxq_s16	Efetua uma operação de máximo entre dois vetores.
vminq_s16	Efetua uma operação de mínimo entre dois vetores.

vmovn_u16	Reinterpreta os dados entre um vetor de tamanho 16 para outro com tamanho 8.
vst3_u8	Faz a transferência dos dados armazenados em registradores para a memória com intercalação de 3 componentes.
vmaxq_s16	Faz a saturação. Se o primeiro elemento for maior ou igual ao segundo elemento então é colocado o primeiro. Caso contrário é colocado o segundo.
vminq_s16	Também faz a saturação. Neste caso, se o primeiro for o maior ou igual ao segundo então é colocado o segundo. Caso contrário é colocado o primeiro
vceqq_u8	Se o primeiro elemento for igual ao segundo, o elemento resultante terá todos os bits setados em 1. Caso contrário, é setado zero
vpaddlq_u8	Faz a adição par a par resultando em um vetor com precisão de 16 *bits*
vpaddlq_u16	Faz a adição par a par resultando em um vetor com precisão de 32 *bits*
vpaddlq_u32	Faz a adição par a par resultando em um vetor com precisão de 64 *bits*
vgetq_lane_u64	Extrai o valor de uma posição específica do vetor (64 *bits*)
vget_low_s16	Pega a parte mais baixa do vetor (16 *bits*)
vget_high_s16	Pega a parte mais alta do vetor (16 *bits*)

Este Capítulo apresentou conceitos sobre arquitetura de computadores e organização paralela. Como o objetivo do trabalho é a utilização de instrução SIMD, foi dado um maior foco em explicar conceitos sobre o SIMD do que as outras organizações paralelas. Em relação as instruções apresentadas, é possível encontrar várias outras no site oficial da ARM. O próximo Capítulo apresenta o xLupa, a webcam e a plataforma embarcada utilizada para realização deste trabalho.

Capítulo 3

Sistemas Embarcados

Neste capítulo será apresentado algumas áreas de aplicação dos sistemas embarcados e as características que os diferem dos sistemas *desktop*. Além disso, será apresentado a plataforma embarcada utilizada, o xLupa embarcado e as ferramentas utilizadas no desenvolvimento do mesmo.

3.1 Áreas de aplicação dos sistemas embarcados

Os sistemas embarcados podem ser aplicados em diferentes áreas, tais como eletrônica automotiva, eletrônica de aeronaves, trens, telecomunicações, sistemas médicos, aplicações militares, sistemas de autenticação, eletrônicos de consumo, equipamentos de fabricação, edifícios inteligentes e robótica [2]. Na eletrônica automotiva os carros só podem ser vendidos a países tecnologicamente avançados se possuírem uma boa quantidade de produtos eletrônicos como sistemas ABS (*Anti-lock Breaking System*), programas de estabilidade eletrônico, recursos de segurança como GPS (*Global Positioning System*), proteção anti-roubo, etc. Sistemas implantados em aeronaves possuem controle de voo, sistema anti-colisão, sistema de informação piloto, entre outros. Todos esses sistemas precisam ser altamente confiáveis [2]. Ao desenvolver uma aplicação para esse tipo de sistema é necessário levar em consideração suas características limitadas de memória e processamento. Eles também devem ser rígidos em relação à segurança como o controle de usinas nucleares, carros, trens, etc [2]. A segurança em sistemas embarcados também engloba aspectos como confiança, manutenção, disponibilidade e proteção [2]. Além disso, sistemas embarcados devem ser eficientes em relação à utilização da energia disponível, tamanho do código executado no sistema, eficiência em tempo real, ao peso e ao custo.

3.2 Características dos Sistemas Embarcados

As diferenças entre arquiteturas *desktop* (x86, etc) e embarcadas (ARM) como objetivos de otimização, modelo de computação, relevância em tempo real, etc são apresentados na Tabela 3.1 retirada de [2].

Tabela 3.1: Diferença entre sistemas *desktop* tipicamente CISC e aplicações embarcadas [2]

	Embarcado	PC
Arquiteturas	Frequentemente heterogênea, Muito compacta	Principalmente homogênea, Não compacta (x86 etc)
Compatibilidade X86	Menos relevante	Muito relevante
Arquitetura fixa	As vezes não	Sim
Modelo de Computação (MoCs)	Código C mais múltiplos modelos (fluxo de dados, eventos discretos,...)	Principalmente *Von Neumann* (C,C++,Java)
Objetivos de Otimização	Vários (energia, tamanho,...)	média de desempenho é o fator principal
Relevância de Tempo Real	Sim e muito!	Dificilmente
Aplicações	Vários aplicativos simultâneos (em alguns casos)	Principalmente uma única aplicação
Aplicativos conhecidos em tempo de *design*	A maioria	Apenas alguns

Na tabela 3.1 é possível perceber a diferença entre as aplicações embarcadas e *desktop*. Nela são apresentadas algumas características que são importantes em sistemas embarcados como por exemplo os objetivos de otimização em relação a energia e tamanho, os quais são pouco relevantes para os PCs. Outra característica citada é que para aplicações em tempo real, existe uma relevância muito maior para sistemas embarcados do que PCs.

3.3 Plataforma Embarcada Beagleboard-xM

Neste trabalho, foi utilizada a placa BeagleBoard-xM [18] que é construída utilizando a tecnologia MPSoC(*Multiprocessor System-on-Chip*) DM 3730, que possui dois processadores no mesmo chip. Um deles é o processador ARM Cortex-A8 e o outro é um processador de sinais digitais (DSP) TMS320C64x+. A Tabela 3.2 apresenta as especificações da plataforma e a Figura 3.1 mostra a BeagleBoard-xM.

Tabela 3.2: Descrição dos recursos da Beagleboard xM [18]

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

A Beagleboard-xM possui um tamanho reduzido de 8,509 x 8,763 centímetros o que possibilita a montagem de protótipos portáteis. É de baixo custo em torno de $149 dólares, baixo consumo de potência o que evita a necessidade de um sistema de resfriamento [19].

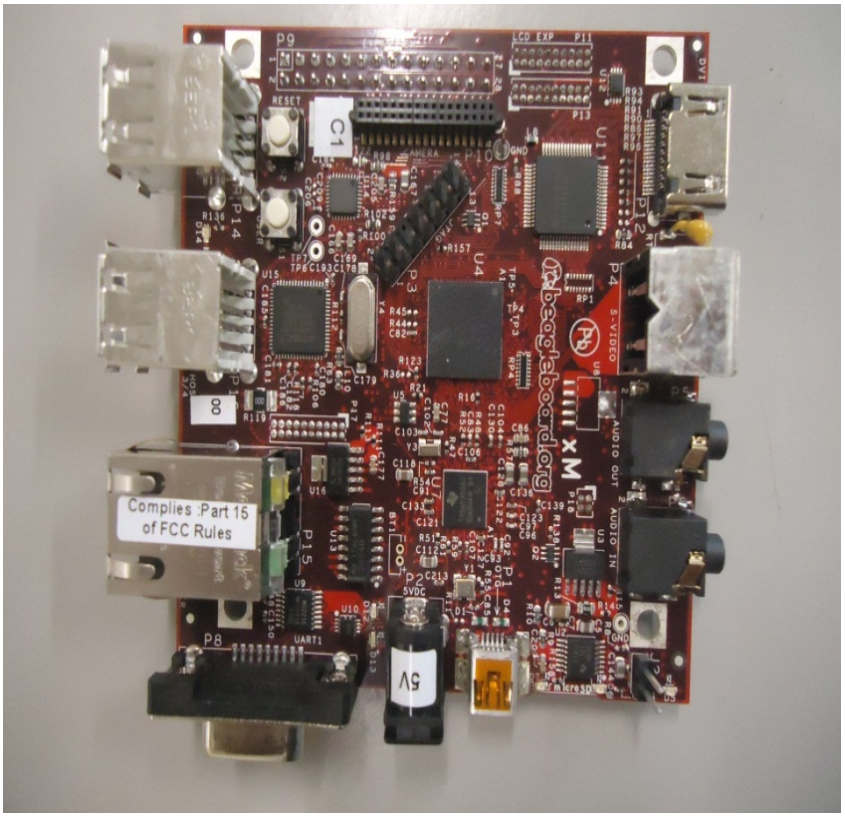

Figura 3.1: Beagleboard-xM

Além disso, essa plataforma possui suporte a sistemas operacionais *open source* (Ubuntu, Fedora, Debian, Angstrom) o que deixa mais flexível o projeto de aplicações, pois há uma gama de bibliotecas disponíveis nos repositórios desses sistemas [19].

3.4 Características do xLupa Embarcado em sua versão atual

Um trabalho anterior [3] mostrou que a conversão explícita do xLupa é mais eficiente do que a conversão feita internamente pelo *driver*, obtendo-se um desempenho de 5,71 FPS contra 4,4 FPS da versão antiga. Esse tempo foi obtido com a mudança no fluxo de execução do xLupa embarcado conforme mostra a Figura 3.2.

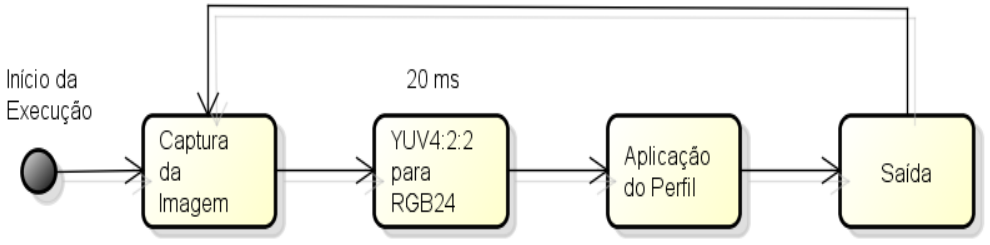

Figura 3.2: Fluxo do xLupa Embarcado [3]

O xLupa embarcado primeiramente faz a captura da imagem utilizando a biblioteca V4L (*Video For Linux*), realiza um pré-processamento, aplicando um perfil escolhido pelo usuário e após envia a imagem para a saída em um monitor ou uma TV. Neste trabalho, foram definidos 5 perfis. O perfil 1 é o perfil normal sem nenhuma modificação na imagem, o perfil 2 possui *zoom*, o perfil 3 tem escala em cinza e *zoom*, o perfil 4 possui contraste verde sem *zoom* e o perfil 5 possui contraste verde com *zoom*. A Figura 3.3 apresentado um exemplo do xLupa embarcado em execução.

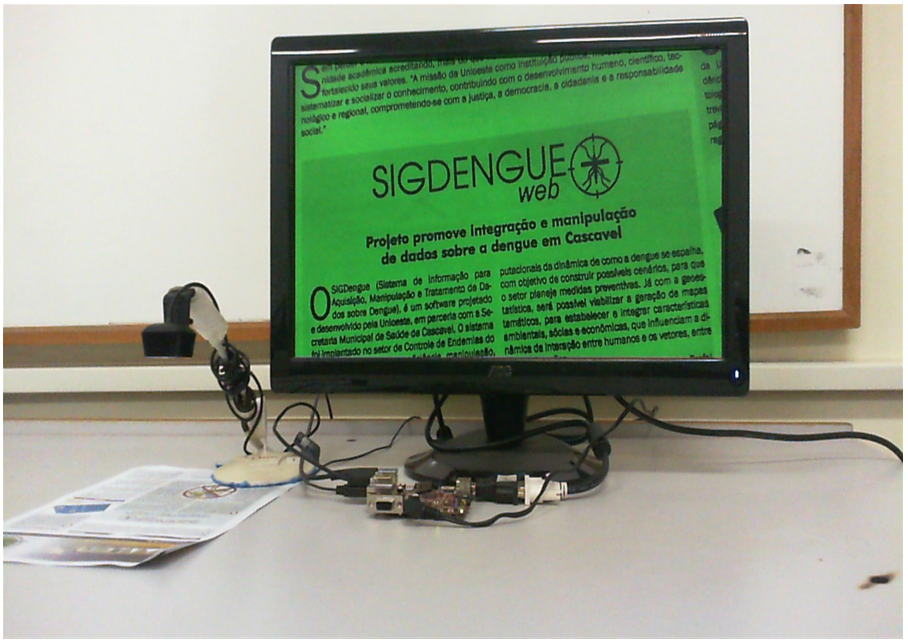

Figura 3.3: xLupa Embarcado

A Tabela 3.3 apresenta detalhadamente o tempo que cada perfil possui em milissegundos e as características envolvidas no processo.

Tabela 3.3: Tempo de execução de cada perfil em milissegundos

	Captura	Cinza	Ampliação	Contraste	Saída	Total
Perfil 1	157,35	0	0	0	75,21	252,36
Perfil 2	157,35	0	13,63	0	75,21	246,19
Perfil 3	157,35	98,21	13,63	0	75,21	344,4
Perfil 4	157,35	0	0	277,35	75,21	509,91
Perfil 5	157,35	0	13,63	277,35	75,21	523,54

As colunas representam respectivamente os tempos de captura, cinza, ampliação, contraste, saída e total. É possível notar que o perfil com maior custo é 5, que envolve a aplicação do contraste e também da ampliação, sendo que o tempo total é mais que o dobro do perfil mais básico onde ocorre apenas a captura e o envio pela tela de saída.

3.5 Distribuição Linux Utilizada

O Ubuntu é um sistema operacional baseado em Linux (Debian) desenvolvido pela Canonical Ltd [20]. A distribuição Ubuntu foi criada para oferecer um sistema operacional amigável,

visando a fácil utilização por pessoas de diferentes nacionalidades [20]. É importante frisar que todas as bibliotecas necessárias para compilação do xLupa foram encontradas no repositório do Ubuntu no formato binário, não necessitando portanto de nenhuma compilação ou adaptação. O xLupa foi compilado sem a necessidade de modificações no código fonte original.

3.6 Microsoft LifeCam HD-5000

A câmera utilizada para a captura das imagens é a LifeCam HD-500 da Microsoft que possui a característica de autofoco. Além disso, ela suporta captura na resolução 720p que possibilita uma melhor definição principalmente nos contornos das letras [4]. Essa *webcam* tem suporte a captura nativa nos formatos YUV4:2:2 e MJPEG. A Figura 3.4 mostra a LifeCam.

Figura 3.4: LifeCam HD-500 com foco automático [21]

Neste capítulo foi apresentado o projeto xLupa e suas características assim como os materiais utilizados para a realização deste trabalho. O xLupa já está sendo utilizado em Cascavel, e auxilia bastante os usuários baixa visão. O próximo Capítulo apresenta os resultados relativos a implementação do método de conversão (YUV4:2:2 para RGB24) e do método de diferença de imagens com NEON.

Capítulo 4

Implementação e Análise dos Resultados

Neste capítulo será apresentado como foi desenvolvido as implementações assim como seus resultados e impactos na execução do xLupa Embarcado. O formato YUV 4:2:2 é um formato definido em termos de uma luminância (Y) e duas crominâncias (UV). O formato RGB (*Red, Green, Blue*) é um sistema de cor aditivo definido em termos de vermelho, verde e azul, sendo o formato utilizado pelo xLupa para a renderização da imagem capturada (Mais detalhes sobre os formatos no Apêndice A).

4.1 Método Aprimorado: Conversão YUV4:2:2 para RGB24 utilizando NEON

No método sequencial, disponível no apêndice B, a conversão de YUV4:2:2 para RGB24 é definida conforme mostra as equações 4.1, 4.2 e 4.3 [22].

$$R = Y + ((V - 128) << 1 + (V - 128)) >> 1 \tag{4.1}$$

$$G = Y - (((U - 128) << 1) + (U - 128) + (V - 128) << 2 + ((V - 128)) << 1)) >> 3 \tag{4.2}$$

$$B = Y + ((U - 128) << 7 + (V - 128)) >> 6 \tag{4.3}$$

O método recebe como argumento dois vetores de *unsigned char*, um representando a imagem capturada em YUV4:2:2 e o outro representando a saída em RGB24, além também, da largura e da altura da imagem. O xLupa embarcado utiliza uma resolução de 1280x720. Dessa

forma, o tamanho da imagem em YUV4:2:2 é 1843200 *bytes* (1280x720x2) (o YUV4:2:2 possui dois *bytes* por *pixel*, conforme a especificação disponível no apêndice A). Nesse método, para cada 4 *bytes* YUV4:2:2 são convertidos 6 *bytes* RGB24, ou seja, para cada 2 *pixels* YUV4:2:2, são convertidos 2 *pixels* RGB24. Dessa forma, o cálculo de conversão apresentado nas equações 4.1 a 4.3 é realizado dentro de um laço com 460800 iterações, pois são processados dois *pixels* YUV4:2:2 de uma vez (O algoritmo de conversão sequencial e com NEON estão disponíveis no Apêndice B).

O método aprimorado NEON segue a mesma ideia do método sequencial para fazer a conversão, porém explorando o paralelismo entre as componentes. Dessa forma, foi possível definir a conversão dos 3 componentes RGB de uma maneira paralela. O componente *Red* foi definido conforme as equações 4.4 à 4.11:

$$R0 = Y0 + ((V0 - 128) << 1 + (V0 - 128)) >> 1 \tag{4.4}$$

$$R1 = Y1 + ((V0 - 128) << 1 + (V0 - 128)) >> 1 \tag{4.5}$$

$$R2 = Y2 + ((V1 - 128) << 1 + (V1 - 128)) >> 1 \tag{4.6}$$

$$R3 = Y3 + ((V1 - 128) << 1 + (V1 - 128)) >> 1 \tag{4.7}$$

$$R4 = Y4 + ((V2 - 128) << 1 + (V2 - 128)) >> 1 \tag{4.8}$$

$$R5 = Y5 + ((V2 - 128) << 1 + (V2 - 128)) >> 1 \tag{4.9}$$

$$R6 = Y6 + ((V3 - 128) << 1 + (V3 - 128)) >> 1 \tag{4.10}$$

$$R7 = Y7 + ((V3 - 128) << 1 + (V3 - 128)) >> 1 \tag{4.11}$$

A componente *Green* foi definida conforme as equações 4.12 à 4.19:

$$G0 = Y0 - (((U0 - 128) << 1) + (U0 - 128) + (V0 - 128) << 2 + ((V0 - 128)) << 1)) >> 3 \tag{4.12}$$

$$G1 = Y1 - (((U0 - 128) << 1) + (U0 - 128) + (V0 - 128) << 2 + ((V0 - 128)) << 1)) >> 3 \tag{4.13}$$

$$G2 = Y2 - (((U1 - 128) << 1) + (U1 - 128) + (V1 - 128) << 2 + ((V1 - 128)) << 1)) >> 3 \tag{4.14}$$

$$G3 = Y3 - (((U1 - 128) << 1) + (U1 - 128) + (V1 - 128) << 2 + ((V1 - 128)) << 1)) >> 3 \tag{4.15}$$

$$G4 = Y4 - (((U2 - 128) << 1) + (U2 - 128) + (V2 - 128) << 2 + ((V2 - 128)) << 1)) >> 3 \tag{4.16}$$

$$G5 = Y5 - (((U2 - 128) << 1) + (U2 - 128) + (V2 - 128) << 2 + ((V2 - 128)) << 1)) >> 3 \tag{4.17}$$

$$G6 = Y6 - (((U3 - 128) << 1) + (U3 - 128) + (V3 - 128) << 2 + ((V3 - 128)) << 1)) >> 3 \tag{4.18}$$

$$G7 = Y7 - (((U3 - 128) << 1) + (U3 - 128) + (V3 - 128) << 2 + ((V3 - 128)) << 1)) >> 3 \tag{4.19}$$

A componente *Blue* foi definida conforme as equações 4.20 à 4.27:

$$B0 = Y0 + ((U0 - 128) << 7 + (V0 - 128)) >> 6 \tag{4.20}$$

$$B1 = Y1 + ((U0 - 128) << 7 + (V0 - 128)) >> 6 \tag{4.21}$$

$$B2 = Y2 + ((U1 - 128) << 7 + (V1 - 128)) >> 6 \tag{4.22}$$

$$B3 = Y3 + ((U1 - 128) << 7 + (V1 - 128)) >> 6 \tag{4.23}$$

$$B4 = Y4 + ((U2 - 128) << 7 + (V2 - 128)) >> 6 \tag{4.24}$$

$$B5 = Y5 + ((U2 - 128) << 7 + (V2 - 128)) >> 6 \tag{4.25}$$

$$B6 = Y6 + ((U3 - 128) << 7 + (V3 - 128)) >> 6 \tag{4.26}$$

$$B7 = Y7 + ((U3 - 128) << 7 + (V3 - 128)) >> 6 \tag{4.27}$$

Pode-se perceber nas equações 4.4 à 4.27 que cada componente U e V faz parte da computação de dois *pixels* RGB. Também é possível perceber que a componente Y muda a cada *pixel*. As equações 4.4 à 4.27 serão executadas todas ao mesmo tempo com as instruções NEON. Com base nessas informações, foi desenvolvido um método que utilizasse o máximo de paralelismo disponível entre esses dados. Os próximos parágrafos explicam como funciona esse método. As Figuras 4.3 à 4.15 mostram como as instruções NEON realizam as operações de conversão de maneira paralela.

Inicialmente, dois ponteiros de *unsigned int* foram apontados para a primeira posição do vetor de imagem YUV4:2:2 como mostra a Figura 4.1.

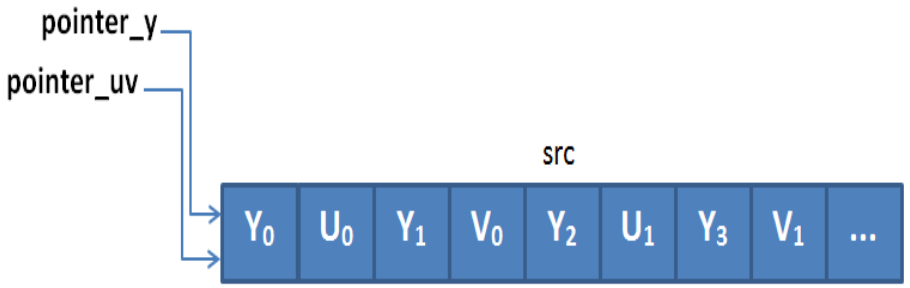

Figura 4.1: Posicionamento dos ponteiros no vetor de dados YUV4:2:2

A variável src é um vetor de *unsigned char* que representa a imagem capturada pela câmera em YUV4:2:2. O *pointer_y* é responsável por apontar a área de dados que será utilizada por uma instrução NEON para a captura dos componentes *y's*. O *pointer_uv* é responsável por apontar para a área que será utilizada por uma instrução NEON para a captura dos componentes *u* e *v*. O ponteiro *vdest* é responsável por apontar para uma aréa de dados (vetor) onde será gravado os valores convertidos para RGB conforme mostra a figura 4.2.

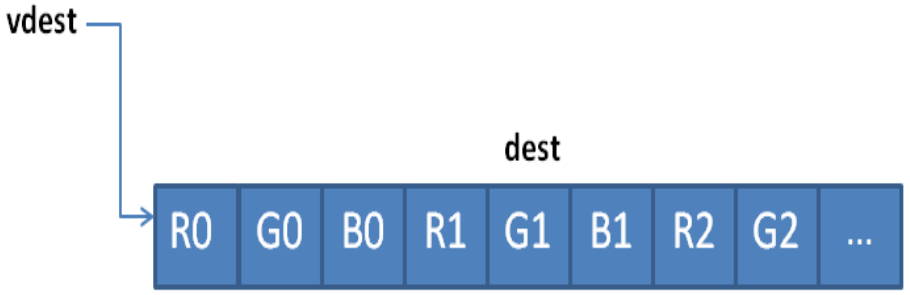

Figura 4.2: Posicionamento do ponteiro vdest no vetor de dados destino RGB24

As próximas instruções são realizadas dentro de um laço de repetição com 57600((1280x720x2)/32 = 57600) iterações ao invés de 921600 ((1280x720x2)/2) como é definido no método sequencial. Isso ocorre por que são processados 32 bytes de uma só vez com as instruções NEON. A figura 4.3 mostra como funciona a instrução *vld4_u8(pointer_uv)*. Essa instrução faz uma intercalação com 4, separando os componentes do vetor src YUV4:2:2 em quatro vetores. Essa extração intercalada é importante, pois assim é possível manipular os componentes separadamente. Analogamente, a instrução *vld2_u8(pointer_y)* extrai do vetor *src* dois

vetores contendo valores *y's* e valores *u's* e *v's* juntos, conforme mostra a Figura 4.4.

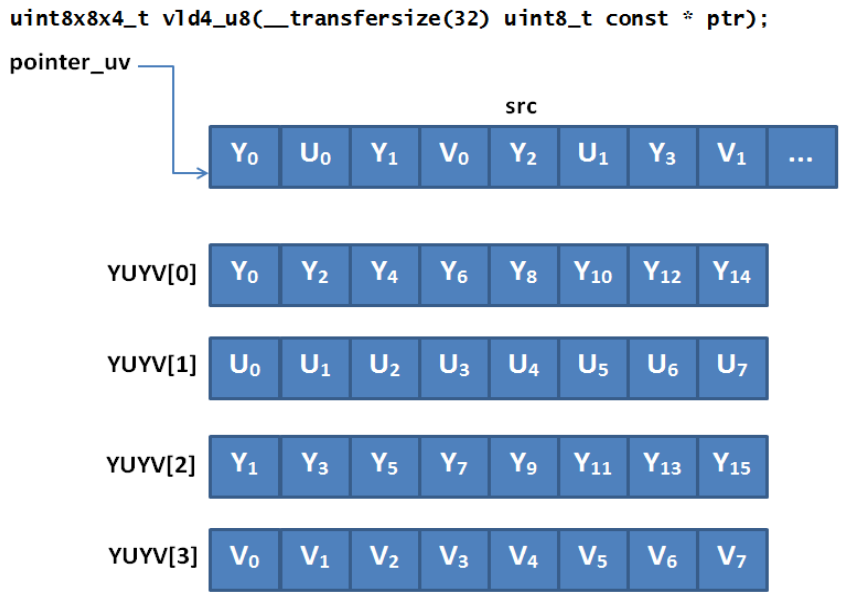

Figura 4.3: Execução da instrução *vld4_u8*

Na instrução *vld4_u8* são extraídos 32 *bytes* de informação do vetor *src*, onde apenas são utilizados os vetores YUYV[1] e YUYV[3], pois como mostra a figura, as componentes *u* e *v* estão dispostas de um modo onde será possível (com a instrução *vzip*, explicada posteriormente) intercalar seus valores.

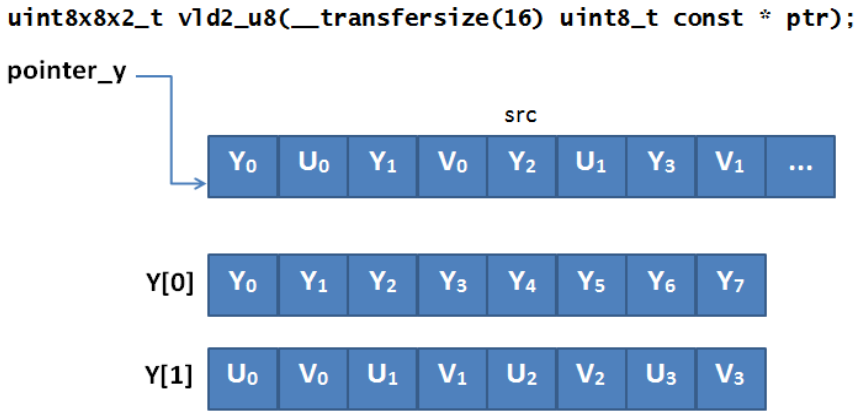

Figura 4.4: Execução da instrução *vld2_u8*

Nas próximas instruções foram feitas intercalações dos componentes *u's* e *v's* extraídos

do *src*, através da instrução *vzip_s16* conforme mostra as Figuras 4.5 e 4.6. Essas operações são necessárias, para que seja possível executar a conversão de maneira paralela. Com isso é possível calcular 24 componentes (8 *red*, 8 *green* e 8 *blue*) de uma só vez. É interessante observar também, que apesar dos componentes YUV4:2:2 serem de 8 *bits* (*uint_8*), as operações intermediárias para a conversão (como *shift right*, *shift left*, soma e subtração) exigem ao menos uma precisão de 16 *bits*, sendo necessário utilizar um vetor com tamanho menor mas com uma precisão maior (*int16x8* com 8 posições e 16 bits de precisão).

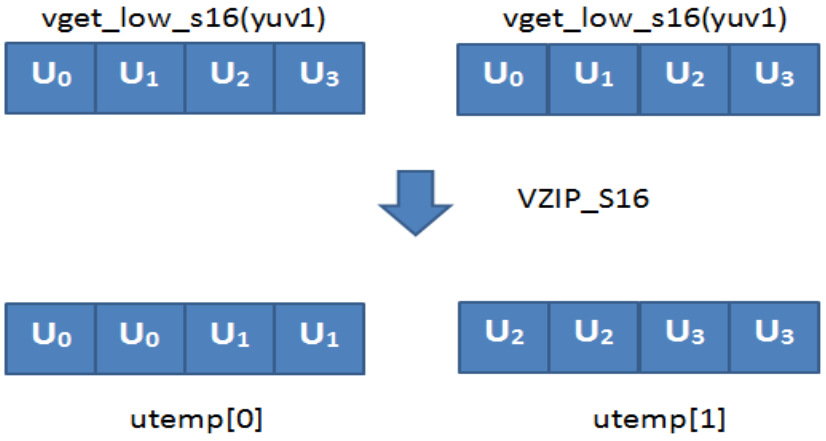

Figura 4.5: Execução da instrução *vzip_s16* no componente u.

Analogamente, a instrução *vzip_s16* também é utilizada para os componentes v's, conforme mostra a figura 4.6:

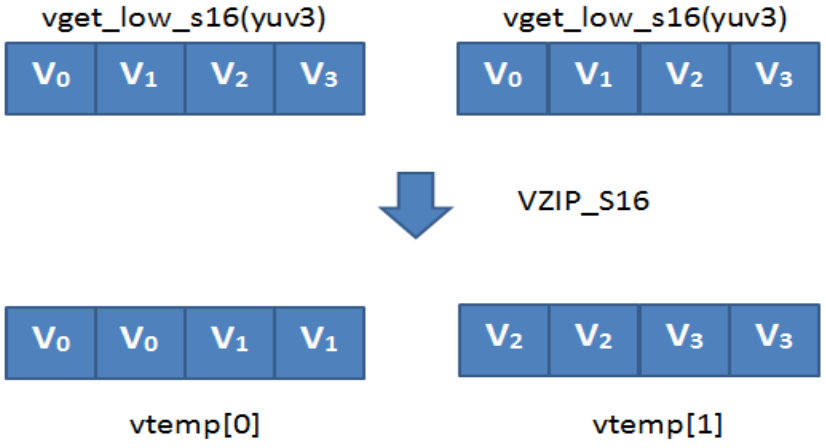

Figura 4.6: Execução da instrução *vzip_s16* no componente v.

Após o *vzip*, foi necessário fazer o *vcombine* para unir os dois vetores em um único de 128 *bits*, conforme mostra a Figura 4.7

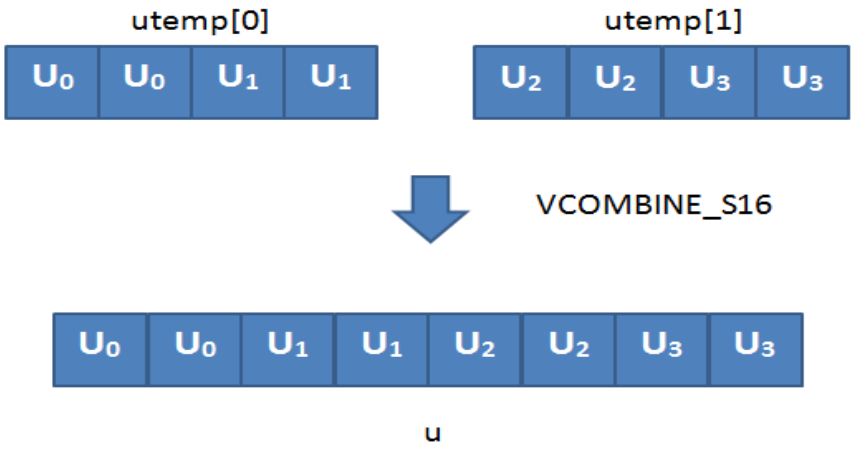

Figura 4.7: Execução da instrução *vcombine_s16* no componente u.

Analogamente, o *vcombine_s16* também é feito ao componente v, conforme mostra a Figura 4.8.

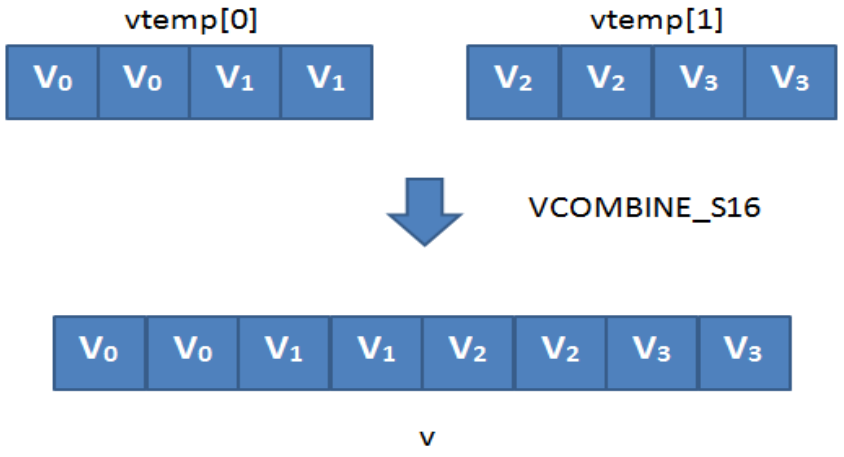

Figura 4.8: Execução da instrução *vcombine_s16* no componente v.

Após a extração e a organização dos componentes *y's*, *u's* e *v's*, o próximo passo foi fazer os cálculos de conversão. As Figuras 4.9 e 4.10 mostram os cálculos intermediários das componentes *u* e *v*. O cálculo do *vintermediario* é representado como *V-128* nas equações 4.1 e 4.2 e o cálculo do *uintermediário* é representado como o *U-128* nas equações 4.2 e 4.3.

$$R = Y + \left(((\boldsymbol{V} - \boldsymbol{128}) \ll 1) + (\boldsymbol{V} - \boldsymbol{128})\right) \gg 1$$

$$G = Y - \left(\begin{matrix} ((U - 128) \ll 1) + (U - 128) + \\ (\boldsymbol{V} - \boldsymbol{128}) \ll 2 + ((\boldsymbol{V} - \boldsymbol{128}) \ll 1) \end{matrix} \right) \gg 3$$

$$B = Y + \left(((U - 128) \ll 7) + (U - 128)\right) \gg 6$$

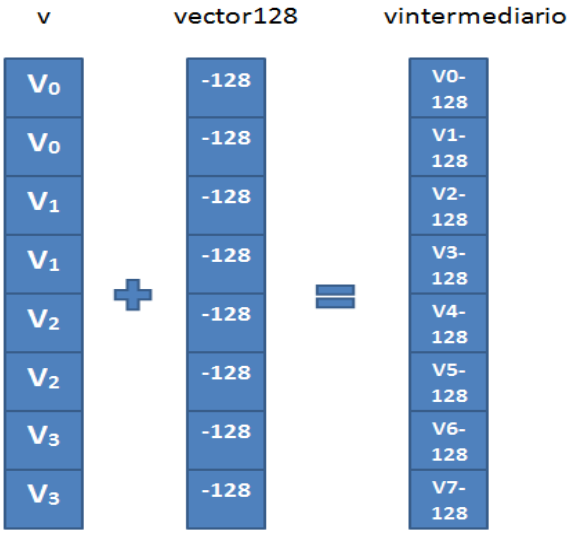

Figura 4.9: Cálculo intermediário envolvendo a componente *v*

Nesse cálculo intermediário é feito a soma posição a posição de *v* com o *vector128* e atribuído à posição respectiva de *vintermediario*, ou seja, *vintermediario[i] = v[i] + vector128[i]* sendo *i* uma variável que representa a posição do vetor.

$$R = Y + \left(\left((V - 128) \ll 1\right) + (V - 128)\right) \gg 1$$

$$G = Y - \left(\begin{matrix}\left((\boldsymbol{U} - \boldsymbol{128}) \ll 1\right) + (\boldsymbol{U} - \boldsymbol{128}) + \\ (V - 128) \ll 2 + \left((V - 128) \ll 1\right)\end{matrix}\right) \gg 3$$

$$B = Y + \left(\left((\boldsymbol{U} - \boldsymbol{128}) \ll 7\right) + (\boldsymbol{U} - \boldsymbol{128})\right) \gg 6$$

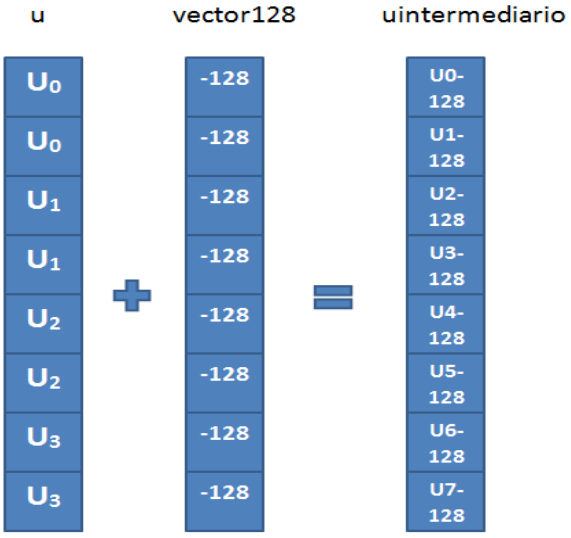

Figura 4.10: Calculo intermediário envolvendo a componente *u*

Análogamente ao calculo da variável *vintermediário*, o calculo do *uintermediario* é feito através da soma entre o vetor *u* e o *vector128* (*uintermediario[i] = u[i] + vector128[i]*).

Esses cálculos intermediários são efetuados uma vez e armazenados para evitar recálculos posteriores. Na Figura 4.11 é demonstrado o cálculo da variável *v1*. Na Figura 4.11 é realizado uma operação de *shift left* (posição a posição) sobre o *vintermediario*, somado com outro vetor *vintermediario* e por fim, realizado um *shift right* sobre o vetor resultante.

$$R = Y + \Big(\big((\boldsymbol{V} - \mathbf{128}) \ll \mathbf{1}\big) + (\boldsymbol{V} - \mathbf{128})\Big) \gg \mathbf{1}$$

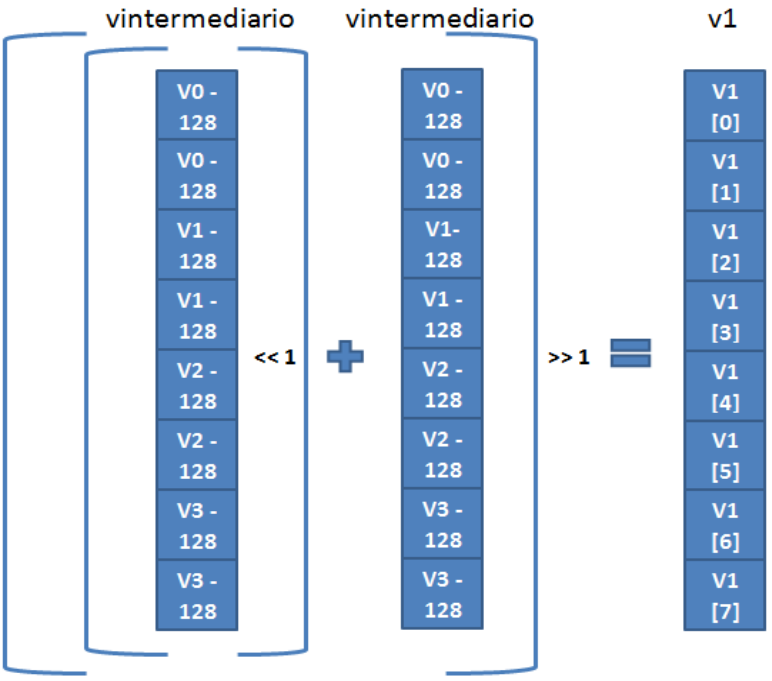

Figura 4.11: Calculo para a obtenção do vetor *v1*

Na Figura 4.12 é demonstrado o cálculo da variável *u1*. Nesta parte é realizado um *shift left* de 7 sobre o vetor *uintermediario*, somado esse resultado a outro *uintermediario* e sobre o resultado final, é realizado um *shift right* de 6.

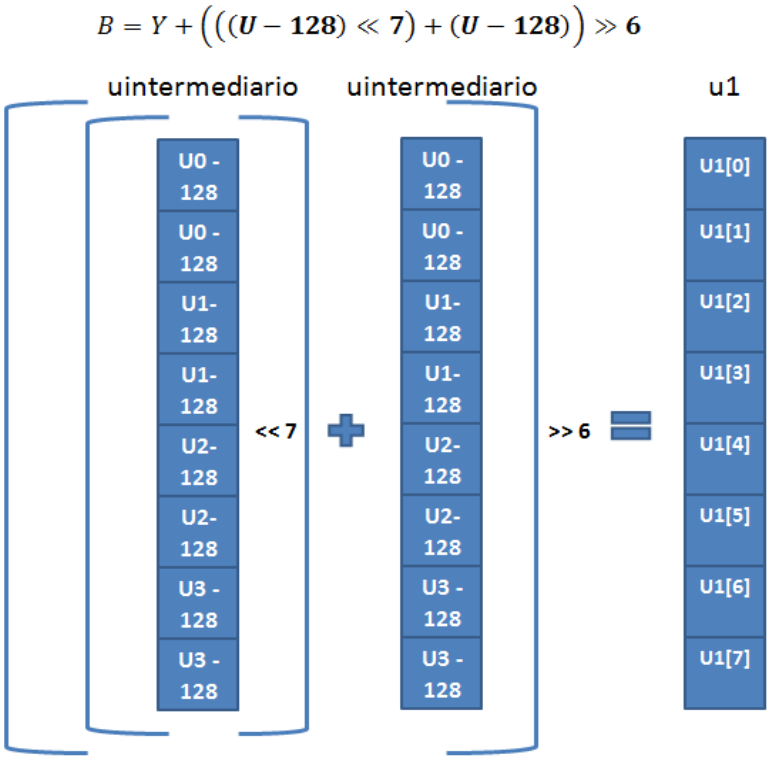

Figura 4.12: Calculo para a obtenção do vetor *u1*

Na Figura 4.13 é demonstrado o cálculo da variável *rgIntermediário*. Nesse cálculo é feito o *shift left* por 2 sobre o *vintermediario*, somado ao *shift left* por 1 sobre outro vetor *vintermediario*.

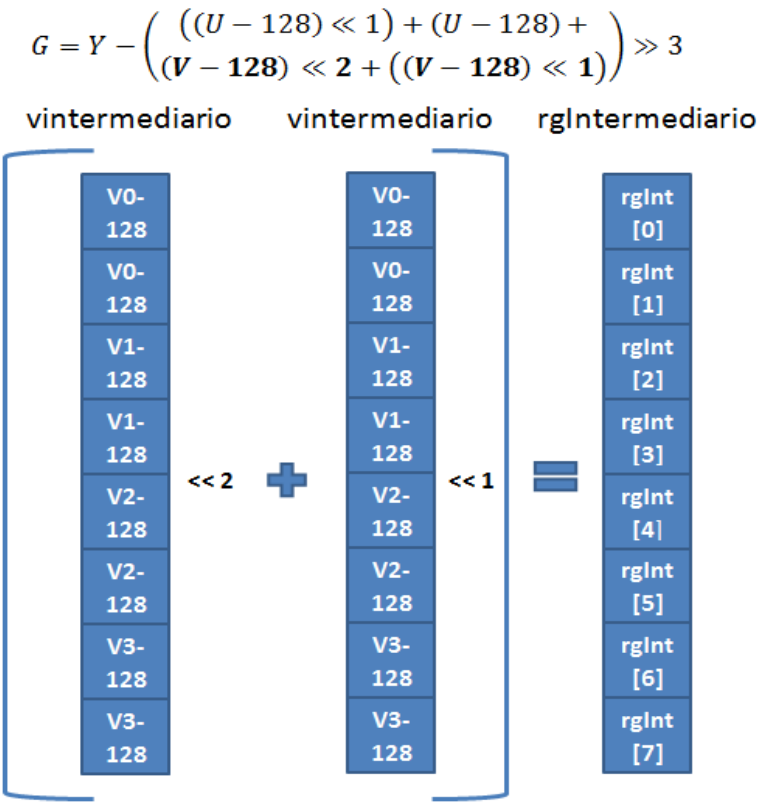

Figura 4.13: Cálculo para a obtenção do vetor *rgIntermediario*

O *rgIntermediário* é utilizado no cálculo do *rg* conforme a Figura 4.14. O *rg* é definido como o *shift left* por 1 de *vintermediario*, somado a outro *vintermediario* que é somado ao *rgintermediario* calculado anteriormente. Ainda sobre esse resultado, é feito um *shift right* por 3.

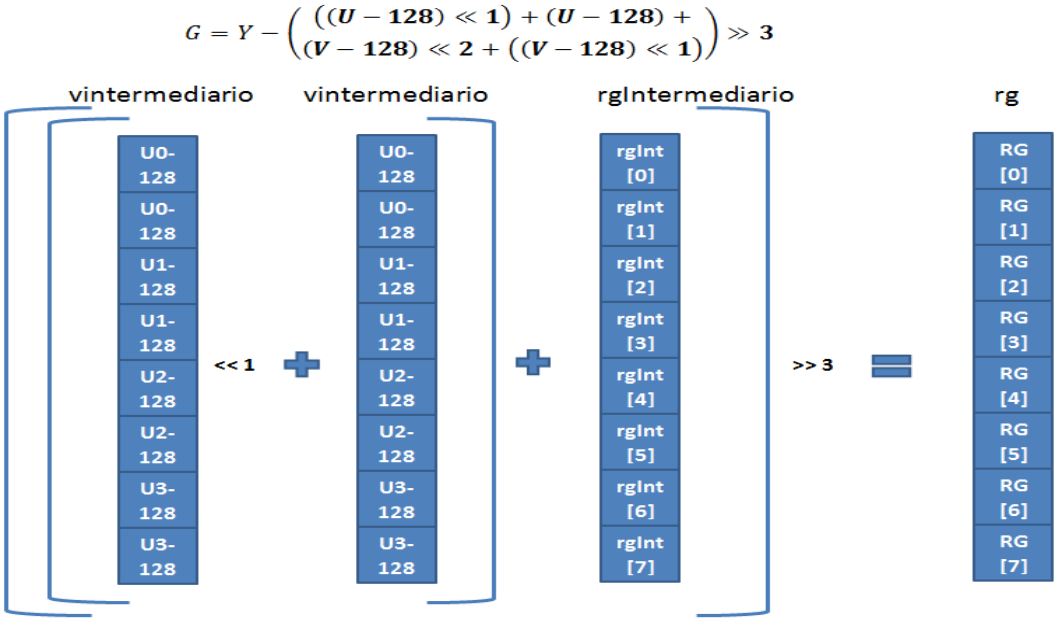

Figura 4.14: Cálculo para a obtenção do vetor *rg*

Após o cálculo do rg é necessário fazer a soma com os *y's* e a saturação. Para isso, foram utilizadas instruções *vmaxq_s16* e *vminq_s16* as quais verificam qual é o máximo e qual é o mínimo entre os vetores. Os resultados desses cálculos possuem uma precisão de 16 *bits*. Assim foi feito a conversão desses 3 vetores de 16 *bits* para 8 *bits* utilizando a instrução *vmovn_u16*. Por fim é feita a gravação dos componentes RGB no vetor destino. Para isso foi utilizado a instrução *vst3_u8* que transfere os dados armazenados nos registradores (3 vetores) para memória, já na representação RGB. A Figura 4.15 demonstra esse esquema:

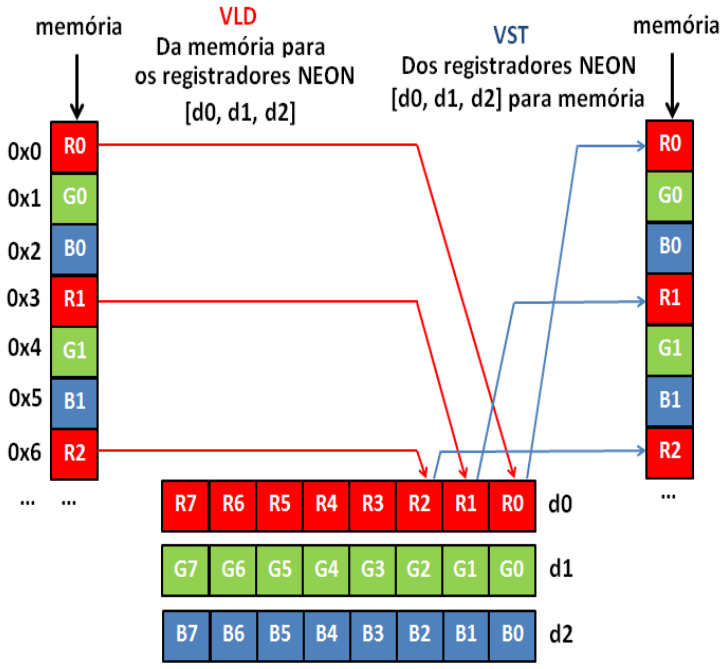

Figura 4.15: Exemplo da execução de Instruções VLD (*load*) e VST (*store*) com RGB. Nesse caso é utilizado uma intercalação de 3 dados, bastante útil para computação do RGB [23]

Visto que não é possível utilizar todos os componentes *u's* e *v's* obtidos de uma vez (pois a intercalação (*vzip*) gera 16 elementos sendo que somente 8 podem ser processados por vez) todas as instruções desde a Figura 4.4 até a Figura 4.15 precisam ser executadas duas vezes, utilizando uma vez a parte baixa do vetor (*u* e *v*) e outra vez a parte alta.

Pode-se observar que para instrução de movimentação de dados entre memória/registrador e registrador/memória é mantido a sequência dos componentes. Isso é possível sem a necessidade de várias movimentações de dados, pois o NEON disponibiliza esse recurso em uma única instrução.

4.2 Comparação do tempo obtido com o tempo da versão atual

A Tabela 4.1 compara os *frames* por segundo da conversão original e o outro utilizando NEON.

Tabela 4.1: Comparação entre a conversão sequencial e a conversão com NEON

	TEMPO YUV-RGB(fps)	TEMPO NEON(fps)	Ganho(%)
Perfil 1	5,713	6,663	16,628%
Perfil 2	2,97	3,20	7,74%
Perfil 3	2,57	2,75	7%
Perfil 4	2,70	2,89	7,03%
Perfil 5	1,83	1,91	4,37%
Média	3,15	3,48	8,55%

É possível observar que o ganho obtido diminuiu conforme aumenta a complexidade do perfil (a descrição de cada perfil está disponível na Seção 3.4), pois o tempo de processamento é dominado pelo tempo de aplicação do perfil. Pode-se observar que as instruções NEON aumentaram em média 8,48% a taxa de *frames* por segundo sendo que o fator máximo foi de 16,628% e o fator mínimo de 4,37%. Esses resultados demonstraram que apesar da disponibilidade de registradores de até 128 *bits* para os cálculos e a possibilidade de realizar 8 operações em paralelo sobre dados de 16 *bits*, nem sempre é possível obter um ganho próximo do paralelismo teórico (8x mais rápido), pois na prática é necessário realizar movimentações de dados entre registradores e memória que consequentemente eleva o processamento.

4.3 Método de Diferença de Imagens

O método de diferença de imagens implementado em [3] utiliza a diferença componente a componente de *pixel* de toda a imagem. Os componentes neste caso são do formato YUV4:2:2, ou seja, estão dispostos conforme a especificação disponível no Apêndice A. O objetivo deste método é melhorar a experiência do usuário com baixa visão com o software xLupa e evitar processamento desnecessário quando não há necessidade de atualização de uma nova imagem. A captura é realizada em YUV4:2:2 e a conversão é realizada somente se necessária. Neste caso, a imagem que está sendo apresentada na tela é armazenada em um *buffer*. A cada nova captura, é realizado um cálculo de diferença componente a componente entre a imagem anterior

e a atual. Após a comparação, se necessário, é feita a conversão para RGB24 e o processamento da imagem. Caso contrário, uma nova captura é realizada. O limiar escolhido para determinar se haverá a troca da imagem foi a alteração de 80% do total de componentes. Esse valor foi escolhido, através dos testes realizados com textos sobre a *webcam*.

O método desenvolvido possibilita a escolha da quantidade da imagem que será utilizada para fazer a diferença. A partir do meio da imagem, é feito a divisão da porcentagem de imagem escolhida para fazer a diferença. Por exemplo, para 50%, a diferença é realizada 25% acima do meio da imagem e 25% abaixo do meio. As Figuras 4.16 a 4.18 mostram como esse método se comporta para 25%, 50% e 75% dos compoenentes em uma imagem de 1280x720 *pixels*.

Figura 4.16: Diferença de Imagens utilizando 25% dos componentes de *pixels*

Figura 4.17: Diferença de Imagens utilizando 50% dos componentes de *pixels*

Figura 4.18: Diferença de Imagens utilizando 75% dos componentes de *pixels*

4.3.1 Diferença de Imagens Utilizando NEON

Além da diferença (sequencial, componente a componente de *pixel*) foi implementado um método que fizesse essa diferença utilizando as instruções NEON, visto que cada componente é

independente e pode ser operado separadamente. Este método também possibilita a escolha da área da imagem para fazer a diferença conforme os exemplos mostrados nas Figuras 4.16, 4.17 e 4.18. Com NEON, é possível fazer 16 operações em paralelo, pois a diferença componente a componente não ultrapassa a precisão de 8 *bits*. Ele possui um laço de repetição que executa a diferença de 16 elementos de uma vez, ou seja, a quantidade de iterações é 16 vezes menor que a versão sequencial. Dentro do laço, são carregados dois vetores *V1* e *V2* de 16 posições cada um. Para isso, foi utilizado a instrução *vld1q_u8* que faz uma cópia sem intercalação de 16 elementos da memória para o registrador (vetor). A Figura 4.19 mostra como é realizado essa cópia:

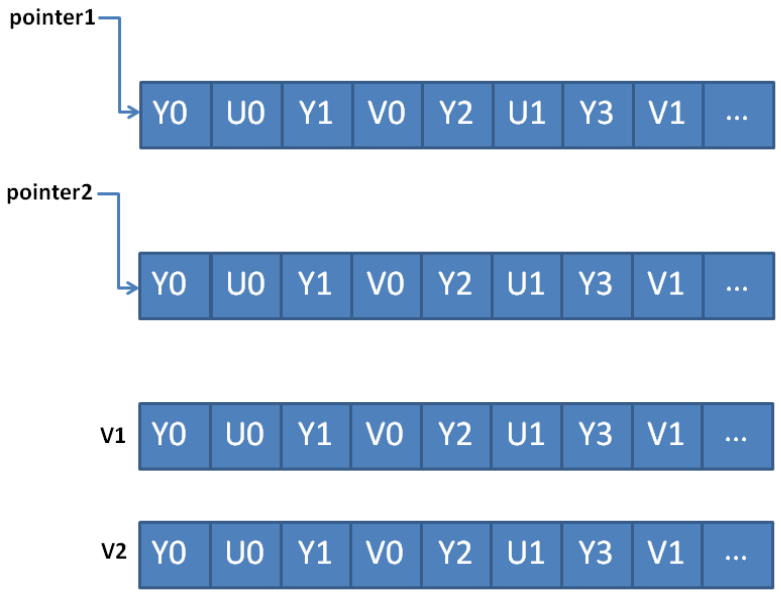

Figura 4.19: Execução da instrução *vld1q_u8*. É feita uma cópia simples da memória (pointer1 e pointer2) para os registradores (V1 e V2)

Após carregado esses valores nos vetores *V1* e *V2*, foi realizado uma operação de comparação utilizando a instrução *vceqq_u8* que compara dois vetores posição a posição, retornando para cada uma, 255 (todos os *bits* ligados) caso os elementos sejam iguais e 0 caso os elementos sejam diferentes. A Figura 4.20 mostra como a instrução *vceqq_u8* se comporta.

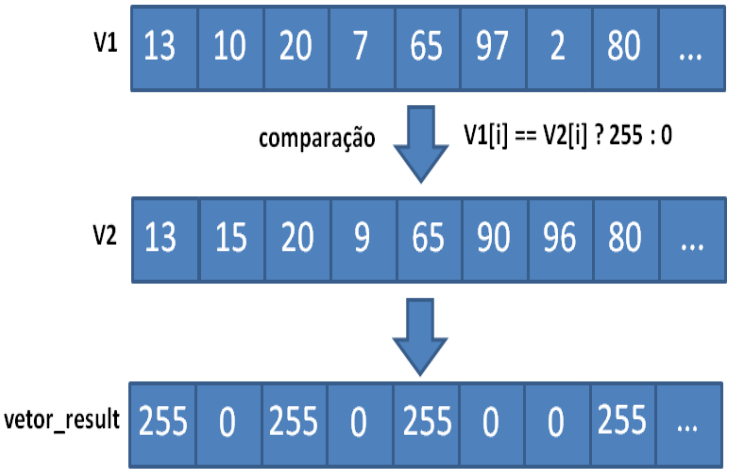

Figura 4.20: Execução da instrução *vceqq_u8* para alguns valores aleatórios

Após, foi necessário armazenar quantos componentes de *pixels* eram iguais. O vetor nesse estágio possui valores 0 e 255, onde o zero representa um valor diferente e o 255 representa um valor igual. Dessa forma, foi realizado a soma de todas as posições do vetor e feita a divisão por 255 para se obter a quantidade de componentes de *pixels* iguais. Para isso, foi necessário utilizar as instruções *vpaddlq_u8*, *vpaddlq_u16* e *vpaddlq_u32* que fazem a adição entre os pares de elementos de um vetor. A Figura 4.21 mostra a utilização dessas instruções.

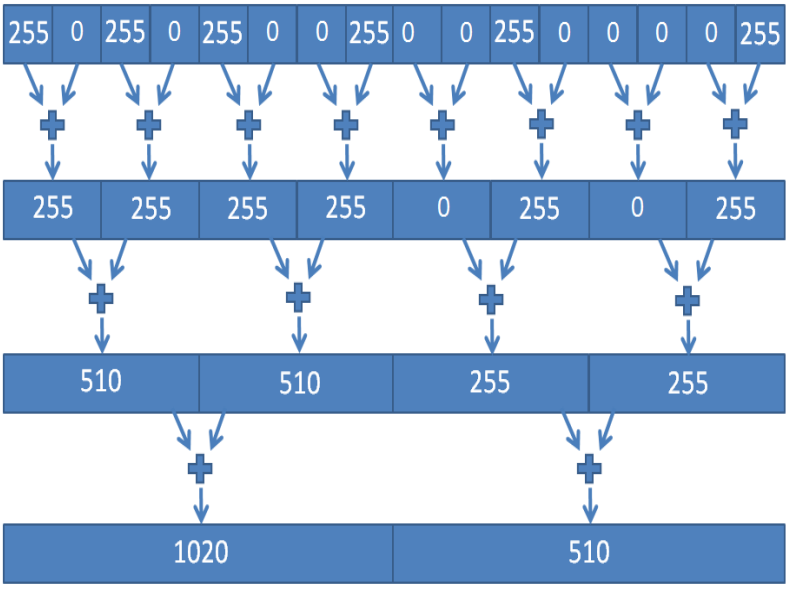

Figura 4.21: Exemplo de execução das instruções *vpaddlq_u8*, *vpaddlq_u16* e *vpaddlq_u32*

Essas 3 instruções são necessárias, pois a adição ocorre par a par e não existe uma instrução disponível que faça tudo de uma só vez. O vetor resultante possui duas posições com 64 *bits* de precisão nas quais foram obtidas as quantidades de componentes de *pixels* iguais. Assim a soma ocorre a cada duas posições e o resultado é dividido por 255, retornando a quantidade de componentes de *pixels* iguais.

O objetivo do uso das instruções NEON, foi realizar a diferença entre os componentes de *pixels* de maneira que fosse possível diferenciar quais ficaram iguais e quais ficaram diferentes e assim armazenar, através da soma em pares, a quantidade de componentes de *pixels* iguais, se essa quantidade estiver abaixo de 80% a imagem não é convertida para RGB.

4.3.2 Comparação entre o método de diferença de imagens sequencial e NEON

A Figura 4.22 faz uma comparação entre o método de diferença de imagens sequencial e NEON.

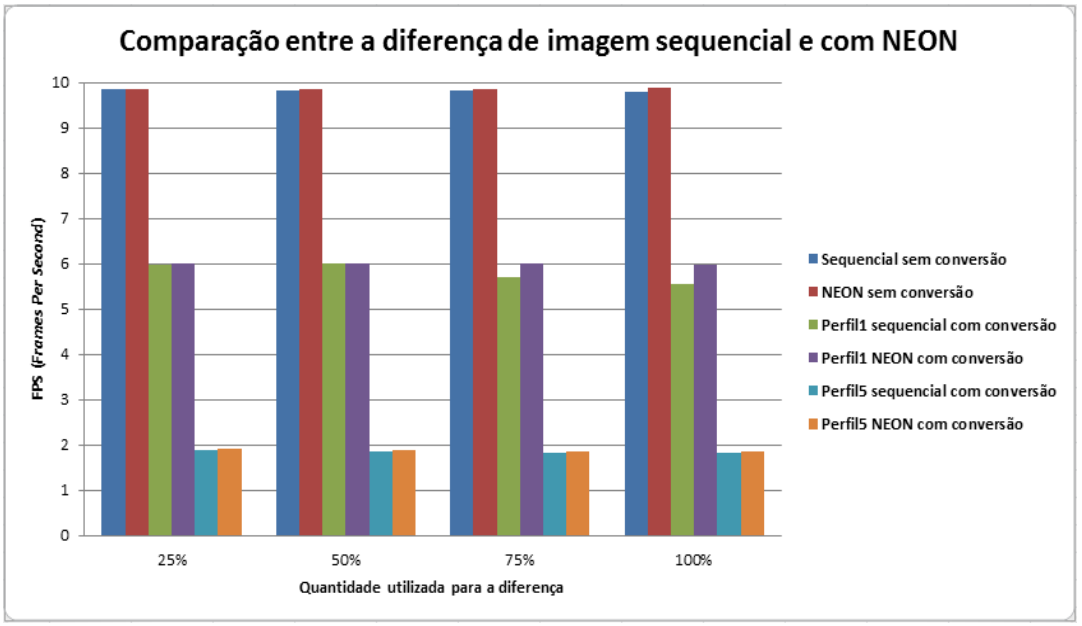

Figura 4.22: Resultados Obtidos

No gráfico da Figura 4.22, foi utilizado para comparação o perfil 1 e o perfil 5 (limite máximo e mínimo de FPS respectivamente) com e sem conversão. O tempo obtido sem a conversão foi de 9,87 FPS com NEON e de 9,8 FPS com o sequencial, utilizando em ambos a diferença sobre todos os componentes da imagem). O tempo obtido do perfil 1 sequencial com conversão sobre 100% dos componentes da imagem, foi de 5,54 FPS contra 5,98 FPS do NEON. Em relação aos outros tamanhos, é possível perceber que não houve uma mudança significativa em relação aos valores obtidos com 100%. Além disso, o ganho obtido com o método NEON foi pouco (menos de 1 FPS). O tempo obtido com a captura em MJPEG sem o processamento da imagem foi de 6,19 FPS contra 9,87 FPS com captura em YUV4:2:2 e sem conversão (NEON). Essa diferença ocorre pelo fato da complexidade da conversão MJPEG para RGB ser maior do que a YUV4:2:2 para RGB24.

No NEON utilizando 25% da imagem, o tempo foi de 9,86 FPS. Já para 100% o tempo foi de 9,87 FPS. Portanto, analisando o custo benefício, a melhor forma de se utilizar o método de diferença de imagens é fazer a diferença entre todos os componentes (100%) utilizando NEON, pois além de possuir um FPS maior, a diferença é realizada sobre a imagem toda.

Capítulo 5

Conclusões e Trabalhos Futuros

Desenvolver aplicações para sistemas embarcados não é uma tarefa fácil. É preciso levar em consideração requisitos que geralmente não são cogitados durante o desenvolvimento de uma aplicação para *desktop*.

Neste trabalho, foram mostradas características desse tipo de sistema, e em particular sobre o xLupa embarcado. Além disso, foram realizados estudos sobre as instruções SIMD que podem ser de grande utilidade para aplicações nas quais não existem dependência entre os dados, como foi o caso da conversão YUV4:2:2 para RGB24 e o método de diferença de imagens utilizados pelo xLupa embarcado.

O processador ARM Cortex-A8 disponível na plataforma embarcada BeagleBoard-xM, possui um conjunto de instruções SIMD chamado de NEON. Com o NEON, foi possível flexibilizar a programação de maneira que diversas operações da conversão sequencial pudessem ser feitas em poucas instruções sobre vários conjuntos de dados.

Além do desenvolvimento de um método aprimorado de conversão YUV4:2:2 para RGB24, também foi implementado um método de diferença de imagens sequencial e outro com NEON que utiliza (para a diferença) apenas algumas partes da imagem a ser renderizada. É importante observar que o compilador não detectou as instruções NEON automaticamente (através das *Flags* -O3 e *-ftree-vectorize*), ou seja, a otimização em ambos os casos (da conversão e da diferença de imagens) foi realizada manualmente.

Os resultados referentes ao método de conversão YUV4:2:2 para RGB24 não foram satisfatórios, pois foi necessário fazer operações de movimentação de dados e reinterpretação desses dados (8 *bits* para 16 *bits* e 16 *bits* para 8 *bits*).

Em relação ao método de diferença de imagens, não houve ganhos significativos quando há

mudança no tamanho da imagem a ser verificada. Além disso, os ganhos relativos a utilização do NEON para a diferença de imagems também não foram satisfatórios, pois foi necessário utilizar instruções que fazem a soma par a par (como mostrado na Figura 4.21) que aumentaram o processamento.

Algumas alternativas que podem ser exploradas com as instruções SIMD no xLupa embarcado estão relacionados com a aplicação de contraste e escala de cinza. Estas operações são independentes e podem ser exploradas de maneira paralela através das instruções NEON.

Apêndice A

Espaço de Cores

A cor é a reação do cérebro a determinados estímulos luminosos. Ela é relativa, ou seja, depende de fatores físicos como o comprimento de onda e a frequência do espectro de luz, e também da fisiologia do indivíduo que a observa. Um espaço de cor é um método no qual é possível especificar, criar e visualizar a cor. Por exemplo, um papel impresso pode representar uma cor em termos de reflexão e absorção de cores como ciano, magenta, amarelo e preto. Um computador pode representar cores em seu monitor através da emissão de fosforo vermelho, verde e azul [24] [25]. Como exemplo de espaços de cores conhecidos, pode-se citar o RGB, CMYK, HSV e o YUV. Os espaços de cores de interesse deste trabalho são RGB e o YUV. O RGB (Red, Green, Blue) é um sistema de cor aditivo definido em termos de vermelho, verde e azul. Cada um desses componentes possuem valores que variam entre 0 e 255, onde valores próximos de 0 significam pouca intensidade e valores próximos a 255 significam maior intensidade. Assim pode-se definir a cor preta como a ausência de todos os componentes (0, 0, 0) e a cor branca a intensidade máxima de todos os componentes (255, 255, 255) [26] [27] [28]. A Figura A.1 mostra o espaço de cor RGB.

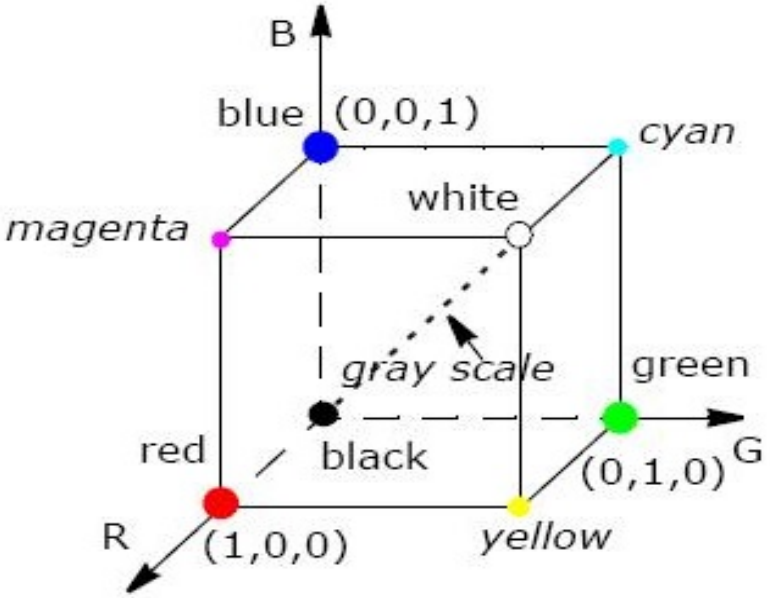

Figura A.1: Espaço de cor YUV [29]

O RGB é bastante utilizado e importante por que ele se relaciona intimamente com a forma como o olho humano percebe a cor. Além disso, monitores gráficos utilizam bastante as cores vermelho, verde e azul para criar a cor desejada [29]. O sistema de cor YUV é definido em termos de uma luminância (Y) e duas crominâncias (UV). Ele foi desenvolvido para permitir reduzir a largura de banda dos componentes de crominância e erros de transmissão que são mascarados da percepção humana do que o uso do sistema RGB. Além disso, era necessário compatibilizar com os antigos sistemas de preto e branco que utilizam apenas a componente Y [30]. A Figura A.2 mostra o espaço de cor YUV.

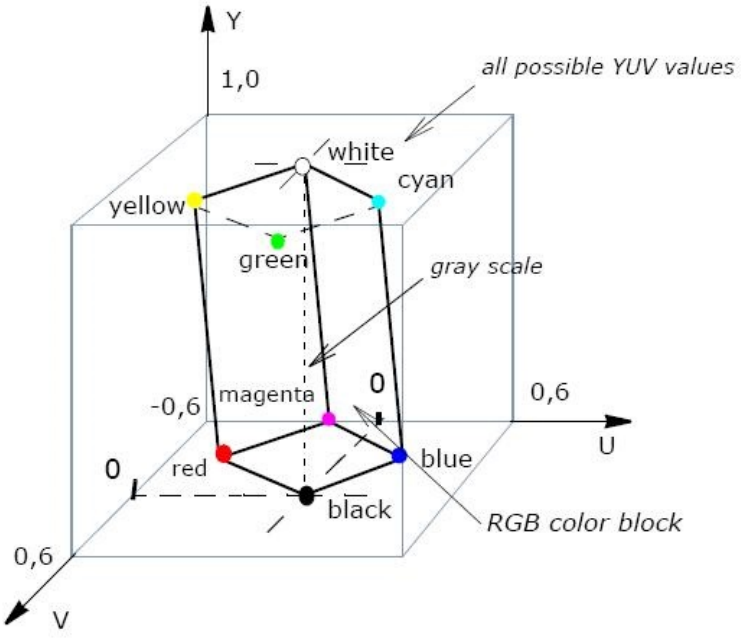

Figura A.2: Espaço de cor YUV [29]

Pode-se perceber que nem todos os valores YUV são validos dentro do espaço RGB.

A.1 Formato RGB24

O formato RGB24 (24bits por pixel) é utilizado pelo xLupa para o desenho da imagem em um monitor ou uma TV. A tabela A.1 abaixo mostra a disposição dos pixels de uma imagem 4x4 com RGB24 [28].

Tabela A.1: Disposição dos bytes RGB24 [29]

Start + 0	R00	G00	B00	R01	G01	B01	R02	G02	B02	R03	G03	B03
Start + 12	R10	G10	B10	R11	G11	B11	R12	G12	B12	R13	G13	B13
Start + 24	R20	G20	B20	R21	G21	B21	R22	G22	B22	R23	G23	B23
Start + 36	R30	G30	B30	R31	G31	B31	R32	G32	B32	R33	G33	B33

A.2 Formato YUV 4:2:2

No formato YUV 4:2:2 cada 4 bytes são dois pixels. Cada Y vai a um pixel e o Cb e o Cr pertence a ambos os pixels [31], estes últimos possuindo metade da resolução horizontal do

componente Y. A tabela A.2 abaixo mostra a disposição dos pixels de uma imagem 4x4 com YUV 4:2:2 [31].

Tabela A.2: Disposição dos bytes YUV 4:2:2 [30]

Start + 0	Y00	Cb00	Y01	Cr00	Y02	Cb01	Y03	Cr01
Start + 8	Y10	Cb10	Y11	Cr10	Y12	Cb11	Y13	Cr11
Start + 16	Y20	Cb20	Y21	Cr20	Y22	Cb21	Y23	Cr21
Start + 24	Y30	Cb30	Y31	Cr30	Y32	Cb31	Y33	Cr31

Apêndice B

Códigos Implementados

B.1 Código YUV 4:2:2 para RGB24 na Linguagem C

```
#define CLIP(color) (unsigned char)(((color)>0xFF)?
0xff:(((color)<0)?0:(color)))

void v4lconvert_yuyv_to_rgb24(const unsigned char *src, unsigned char *
int width, int height)
{
    int j;
    int u,v,u1,rg,v1;
    int screen_size = width*height;
    for (j = 0; j < screen_size; j += 2) {
        u = src[1];
        v = src[3];
        u1 = (((u - 128) << 7) + (u - 128)) >> 6;
        rg = (((u - 128) << 1) + (u - 128) +
        ((v - 128) << 2) + ((v - 128) << 1)) >> 3;
        v1 = (((v - 128) << 1) + (v - 128)) >> 1;
        *dest++ = CLIP(src[0] + v1);
        *dest++ = CLIP(src[0] - rg);
        *dest++ = CLIP(src[0] + u1);
        *dest++ = CLIP(src[2] + v1);
        *dest++ = CLIP(src[2] - rg);
        *dest++ = CLIP(src[2] + u1);
        src += 4;
    }
}
```

B.2 Código YUV 4:2:2 para RGB24 em NEON

```
void v4lconvert_yuyv_to_rgb24_optimized_NEON_test(const unsigned char *
unsigned char *dest,
int width, int height)
{
    int j,i=0;
```

```
int x;
int screen_size_pixels = width*height;//1280x720
int screen_size_bytes = screen_size_pixels*2;
uint8_t *pointer_y = src;
uint8_t *pointer_uv = src;
uint8_t *vdest = dest; //ponteiro para gravar no destino
uint8x8x2_t y; //captura os y's e os u's e v's
uint8x8x4_t yuyv; //armazena os 4 componentes porem utiliza
apenas os u's e v's
uint8x8x3_t rgb; //armazena os 3 componentes finais para a
gravacao
int16x4x2_t utemp; //variavel temporaria para o u, utilizada
para intercalar os valores
int16x4x2_t vtemp; //variavel temporaria para o v, utilizada
para intercalar os valores
int16x8_t u;
int16x8_t v;
int16x8_t vIntermediario; //armazena valores intermediarios
int16x8_t uIntermediario;//armazena valores intermediarios
int16x8_t v1;
int16x8_t u1;
int16x8_t rgIntermediario;//armazena valores intermediarios
int16x8_t rg;
int16x8_t red;
int16x8_t green;
int16x8_t blue;
uint8x8_t convert;
/* vetores de constantes*/
int16x8_t vector128 = vdupq_n_s16(-128);
int16x8_t vector0 = vdupq_n_s16(0);
int16x8_t vector255 = vdupq_n_s16(255);
int16x8_t y0;
int16x8_t yuv1;
int16x8_t yuv3;
```

```
for (j = 0; j < screen_size_bytes; j += 32) {
    /* Apenas os u's e os v's sao utilizados*/
    yuyv = vld4_u8(pointer_uv);
    yuv1 = vmovl_u8(yuyv.val[1]);
    yuv3 = vmovl_u8(yuyv.val[3]);
    for(i = 0;i<2;i++){
        rgb = vld3_u8(vdest); //carrega rgb para os
        registradores (3)
        /* Apenas os y's sao utilizados*/
        y = vld2_u8(pointer_y);
        y0 = vmovl_u8(y.val[0]);
        /* combina os dois vetores de maneira a intercalar
         os valores
        * [u0,u0,u1,u1,...] e [v0,v0,v1,v1,...]*/
        if(i == 0){
            utemp = vzip_s16(vget_low_s16(yuv1),
```

```
                vget_low_s16(yuv1));
                vtemp = vzip_s16(vget_low_s16(yuv3),
                vget_low_s16(yuv3));
            }else{
                utemp = vzip_s16(vget_high_s16(yuv1),
                vget_high_s16(yuv1));
                vtemp = vzip_s16(vget_high_s16(yuv3),
                vget_high_s16(yuv3));
            }
            u = vcombine_s16(utemp.val[0],utemp.val[1]);
            v = vcombine_s16(vtemp.val[0],vtemp.val[1]);
            //faz o calculo do v1
            vIntermediario = vaddq_s16(v,vector128);
            uIntermediario = vaddq_s16(u,vector128);
            v1 = vshrq_n_s16(vaddq_s16(vshlq_n_s16(vIntermediario,1)
            ,vIntermediario),1);
            //faz o calculo do u1
            u1 = vshrq_n_s16(vaddq_s16(vshlq_n_s16(uIntermediario,7)
            ,uIntermediario),6);



            //faz o calculo do rg
            rgIntermediario =
                vaddq_s16(vshlq_n_s16(vIntermediario,2),
                vshlq_n_s16(vIntermediario,1));
            rg =
                vshrq_n_s16(vaddq_s16(vaddq_s16(
                vshlq_n_s16(uIntermediario,1),uIntermediario),
                rgIntermediario),3);
            //faz a soma e a saturacao
            red = vmaxq_s16(vector0,vminq_s16(vector255,
            vaddq_s16(y0,v1)));
            green = vmaxq_s16(vector0,vminq_s16(vector255,
            vsubq_s16(y0,rg)));
            blue = vmaxq_s16(vector0,vminq_s16(vector255,
            vaddq_s16(y0,u1)));
            convert = vmovn_u16(red);
            rgb.val[0] = convert;
            convert = vmovn_u16(green);
            rgb.val[1] = convert;
            convert = vmovn_u16(blue);
            rgb.val[2] = convert;
            vst3_u8(vdest,rgb);
            //move o ponteiro destino (rgb)
            vdest+=24;
            //move o ponteiro do vetor yuv
            pointer_y+=16;
        }
        //move o ponteiro do auxiliar
        pointer_uv+=32;
    }
}
```

B.3 Diferença de Imagens Sequencial

```
int image_diff_section(const unsigned char *yuv1,
const unsigned char *yuv2,int first_offset, int end_offset){
    int i;
    int pixels = 0;
    for(i = first_offset; i<end_offset;i++){
        if((int)((int)yuv1[i] - (int)yuv2[i]) != 0)
            pixels++;
    }
    return pixels;
}
```

B.4 Diferença de Imagens NEON

```
int image_diff_section_NEON(const unsigned char *yuv1,
const unsigned char *yuv2,int first_offset,
int end_offset,int width,int height){
    int i;
    uint8x16_t resultado;
    uint64x2_t scalarResult;
    uint8x16_t vetor_result;
    uint8x16_t v1;
    uint8x16_t v2;
    int iguais = 0;
    int partSize = end_offset - first_offset;
    int j = 0;
    if((end_offset - first_offset) % 16 == 0){
        const uint8_t *pointer1 = &yuv1[first_offset];
        const uint8_t *pointer2 = &yuv2[first_offset];
        for(i = first_offset; i<end_offset;i+=16){
            v1 = vld1q_u8(pointer1);
            v2 = vld1q_u8(pointer2);
            vetor_result = vceqq_u8(v1,v2);
            scalarResult = vpaddlq_u32(vpaddlq_u16(
                vpaddlq_u8(vetor_result)));
            iguais += ((int)vgetq_lane_u64(
                scalarResult,0) +
                (int)vgetq_lane_u64(scalarResult,1))/255;
            pointer1+=16;
            pointer2+=16;
        }
        return (partSize - iguais);
    }else{
        printf("Tamanho não é multiplo de 16.");
        exit(0);
    }
}
```

Apêndice C

Resultados da Diferença de Imagens

A Tabela C.1 compara os resultados relativos a diferença de imagens sequencial e NEON.

Tabela C.1: Comparação dos Resultados da Diferença de Imagens Sequencial e NEON em FPS

	25%	50%	75%	100%
Sequencial sem conversão	9,84026	9,81	9,82022	9,8
NEON sem conversão	9,86	9,84	9,86	9,87
Perfil 1 sequencial com conversão	5,982	5,999	5,699	5,549
Perfil 1 NEON com conversão	6	6	6	5,98
Perfil 2 sequencial com conversão	3,03	2,95	2,91	2,86
Perfil 2 NEON com conversão	3,02	2,97	2,91	2,9
Perfil 3 sequencial com conversão	2,6	2,56	2,52	2,47
Perfil 3 NEON com conversão	2,63	2,57	2,54	2,52
Perfil 4 sequencial com conversão	2,71	2,66	2,63	2,59
Perfil 4 NEON com conversão	2,74	2,71	2,66	2,64
Perfil 5 sequencial com conversão	1,88	1,86	1,84	1,82
Perfil 5 NEON com conversão	1,91	1,88	1,87	1,85

Glossário

Desktop	Termo utilizado na computação relacionando a um computador de mesa.
Load	Termo que em computação, significa carregar dados da memória.
Store	Em computação significa gravar dados na memória.

Referências Bibliográficas

[1] XLUPA. *XLupa Recursos and Funcionamento*. 2014. Consultado na INTERNET: http://projetos.unioeste.br/campi/xlupa, 2014.

[2] MARWEDEL, P. *Embedded System Design*. 2. ed. [S.l.]: Springer, 2010.

[3] POSSAMAI, D.; SANTOS, F. F. dos; OYAMADA, M. S. Implementação de uma lupa digital baseada em captura de imagens. In: *22º EAIC: Encontro Nacional de Iniciação Científica*. Foz do Iguaçu: [s.n.], 2013. p. 1–4.

[4] MICROSOFT. *Microsoft Corporation - LifeCam HD-5000*. 2014. Consultado na INTERNET: http://www.microsoft.com/hardware/pt-br/p/lifecam-hd-5000/7ND-00002, 2014.

[5] STALLINGS, W. *Arquitetura e Organização de Computadores*. 8. ed. Reading: Pearson Pratice Hall, 2010.

[6] SILVA, L. F.; ANTUNES, V. J. M. *Comparação entre as arquitecturas de processadores RISC e CISC*. 2014. Consultado na INTERNET: http://necc.di.uminho.pt/lib/exe/fetch.php?media=ap:1ano:sc:ap_sc_risc_cisc.pdf, 2014.

[7] HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 5. ed. Reading: Morgan Kaufmann, 2012.

[8] FLYNN, M.; RUDD, K. Parallel architectures. In: *ACM Computing Surveys*. [S.l.: s.n.], 1996. p. 1–4.

[9] PUJARA, C. et al. H.264 video decoder optimization on arm cortex-a8 with neon. In: *Samsumg India Software Operations Pvt. Ltd. Bangalore*. Bangalore: [s.n.], 2009. p. 1–4.

[10] ARM. *Company Profile*. 2014. Consultado na INTERNET: http://www.arm.com/about/company-profile/index.php, 2014.

[11] ARM. *Cortex-A8 Tecnical Reference Manual*. 2007. Consultado na INTERNET: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344d/DDI0344D_cortex_a8_r2p1_trm.pdf, 2014.

[12] ARM. *The ARM Architecture with focus on v7A and Cortex-A8*. 2014. Consultado na INTERNET: http://www.arm.com/files/pdf/ARM_Arch_A8.pdf, 2014.

[13] ARM. *ARM NEON support in the ARM compiler*. 2008. Consultado na INTERNET: http://www.arm.com/files/pdf/neon_support_in_the_arm_compiler.pdf, 2014.

[14] NUHI, A. *ARM NEON Development*. 2014. Consultado na INTERNET: http://www.add.ece.ufl.edu/4924/docs/arm/ARM%20NEON%20Development.pdf, 2014.

[15] ARM. *Intrinsics*. 2014. Consultado na INTERNET: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/CIHJBEFE.html, 2014.

[16] ARM. *NEON*. 2014. Consultado na INTERNET: http://www.arm.com/products/processors/technologies/neon.php, 2014.

[17] ARM. *Vector Data Types*. 2014. Consultado na INTERNET: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/CIHJBEFE.html, 2014.

[18] BEAGLEBOARD.ORG. *BeagleBoard Reference Manual*. 2014. Consultado na INTERNET: http://beagleboard.org, 2014.

[19] HACKMANN, D. et al. Um ampliador de tela embarcado utilizando arquiteturas heterogêneas. In: *XII Workshop de Software Livre*. [S.l.: s.n.], 2011. p. 1–6.

[20] UBUNTU. *O que é o Ubuntu?* 2014. Consultado na INTERNET: http://www.ubuntu-br.org/, 2014.

[21] TUSEQUIPOS.COM. *Microsoft LifeCam Hd-5000*. 2014. Consultado na INTERNET: http://www.tusequipos.com/wp-content/uploads/2010/03/Microsoft-LifeCam-HD-50001.jpg, 2014.

[22] BROWSER, S. C. *YUV4:2:2 to RGB24*. 2014. Consultado na INTERNET: http://sourcecodebrowser.com/libv4l/0.5.8/libv4lconvert-priv_8h.html#ab280783fd1c254c0869453f296900aff, 2014.

[23] MARTYNARM. *Coding for NEON - Part 1: Load and Stores*. 2014. Consultado na INTERNET: http://community.arm.com/groups/processors/blog/2010/03/17/coding-for-neon–part-1-load-and-stores, 2014.

[24] FORD, A.; ROBERTS, A. *Colour Space Conversions*. 1998. Consultado na INTERNET: http://www.poynton.com/PDFs/coloureq.pdf, 2014.

[25] MELCHIADES, F.; BOSCHI, A. Cores e tonalidades em revestimentos cerâmicos. In: *Laboratório de Revestimentos Cerâmicos*. [S.l.: s.n.], 1999. p. 1–6.

[26] FOURCC. *RGB Pixel Formats*. 2014. Consultado na INTERNET: http://www.fourcc.org/rgb.php, 2014.

[27] SIGNIFICADOS.COM.BR. *O que é RGB?* 2014. Consultado na INTERNET: http://www.significados.com.br/rgb/, 2014.

[28] DIRKS, B. et al. *Packed RGB Formats*. 2014. Consultado na INTERNET: http://linuxtv.org/downloads/v4l-dvb-apis/packed-rgb.html, 2014.

[29] INTEL. *O que é RGB?* 2014. Consultado na INTERNET: https://software.intel.com/sites/products/documentation/hpc/ipp/ippi/ippi_ch6/ch6_color_models.html, 2014.

[30] CHANEY, A. J. B. *YUV*. 2014. Consultado na INTERNET: https://www.princeton.edu/ achaney/tmve/wiki100k/docs/YUV.html, 2014.

[31] DIRKS, B. et al. *YUV Formats*. 2014. Consultado na INTERNET: http://linuxtv.org/downloads/v4l-dvb-apis/V4L2-PIX-FMT-YUYV.html, 2014.