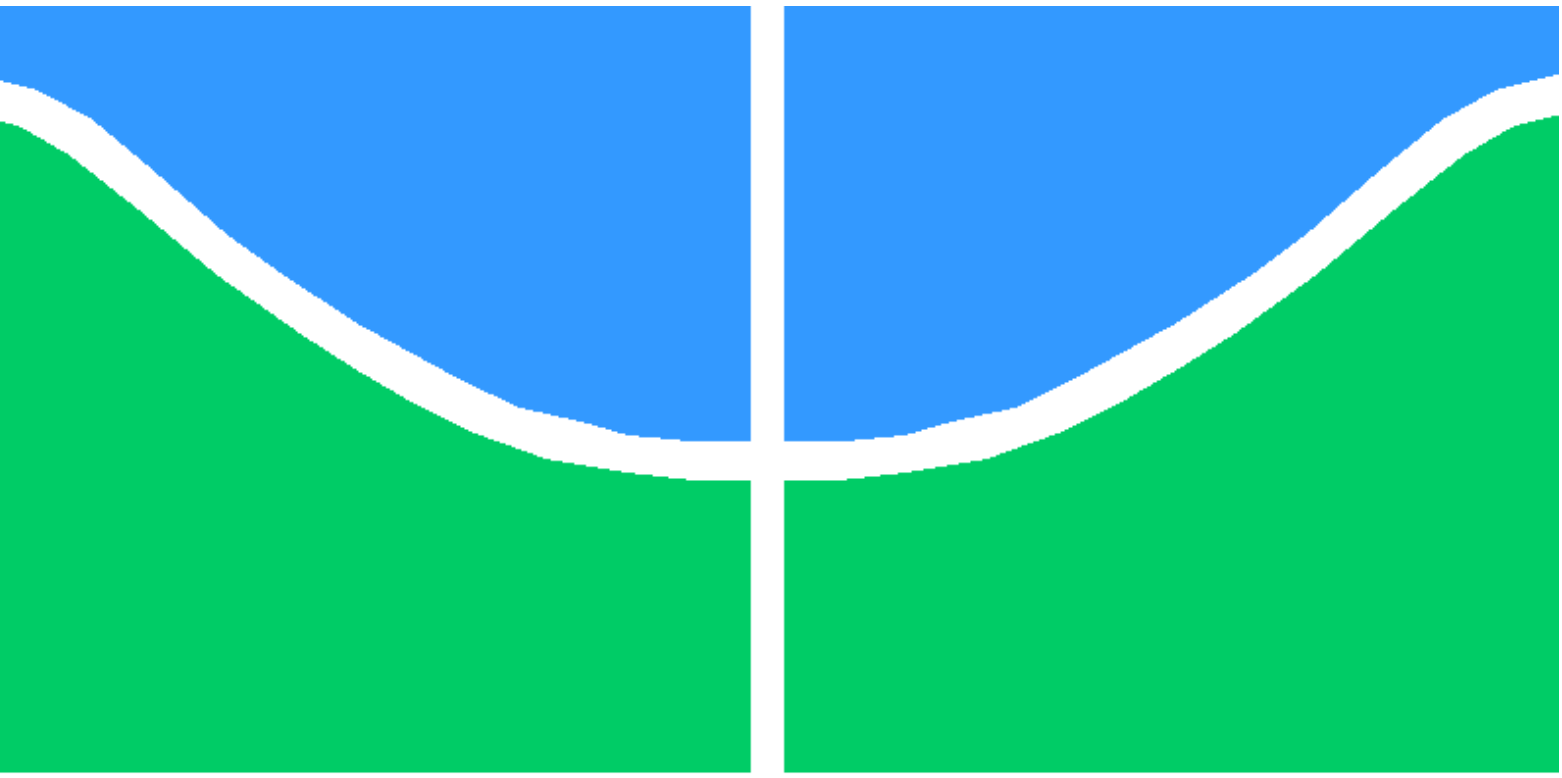

TRABALHO DE GRADUAÇÃO

MODELAGEM DE APS E MAPEAMENTO DE APLICAÇõES EM UMA PLATAFORMA RSOC VIRTUAL

João Lucas de Carvalho Carneiro

Brasília, julho de 2009

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

MODELAGEM DE APS E MAPEAMENTO DE APLICAÇõES EM UMA PLATAFORMA RSOC VIRTUAL

João Lucas de Carvalho Carneiro

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro Eletricista

Banca Examinadora

Prof. José Camargo da Costa _____
Orientador

Prof. Carlos Humberto Llanos _____
Examinador

Eng. José Edil Guimarães de Medeiros _____
Examinador

Dedicatória

À todos aqueles que, de uma forma ou de outra, se sacrificaram para que eu realizasse meus sonhos.

João Lucas de Carvalho Carneiro

Agradecimentos

Agradeço à Deus por me permitir alcançar mais um sonho, me abençoando com tantos exemplos vivos de vida:

- *Meus avôs e avós, pais por duas vezes, que a custo de muito sacrifício construíram as nossas vidas, e a quem muito devo;*
- *Meus padrinhos pelo notável apoio, em especial à minha madrinha, sendo meu ponto de sustentação desde que nasci;*
- *Irmãos e amigos, que se confundem, pelo importante papel desempenhado em minha formação;*
- *Colegas de curso, que em meio a felicidades e sofrimentos me ajudaram a ter uma verdadeira lição de vida em todos estes anos juntos.*
- *Meus pais, principais exemplos de vida, que sempre deram suas vidas para que eu me desenvolvesse, fornecendo sempre inegável apoio a todas as atividades para meu crescimento.*

Agradeço de forma especial à equipe do LPCI da UnB: professor José Camargo, Juan Eusse, Heider e Edil. Foram pessoas que me ajudaram muito durante a elaboração deste trabalho. Ao Gilmar, que praticamente me acompanha desde quando ingressei no curso, particularmente sou muito grato pela amizade e conhecimentos transmitidos. Sem esta equipe provavelmente este trabalho não teria sido elaborado.

João Lucas de Carvalho Carneiro

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

1.1 Contextualização

O recente desenvolvimento da tecnologia, desde a invenção do computador, sempre foi acompanhado por uma crescente necessidade por maior poder de processamento nos sistemas. Isto vem exigindo soluções cada vez mais sofisticadas para manter este crescimento. Como ilustração, a lei de Moore previa que o poder de processamento duplicaria a cada 18 meses. Hoje discute-se se esta lei perderá sua validade.

Sabe-se que sistemas eletrónicos modernos são formados por muitos subsistemas e componentes, prioritariamente divididos em hardware, software e algoritmos. Nestes sistemas modernos, cada uma destas disciplinas se tornou mais complexa [1]. A figura 1.1 ilustra a complexidade para projeto de hardware em um grande design *sistem on chip* (SoC). O gráfico separa quatro níveis de abstração: arquitetura, comportamental, RTL e em portas. Além disso, a figura 1.1 considera apenas um simples circuito integrado, não refletindo a grande complexidade de sistemas com vários chips.

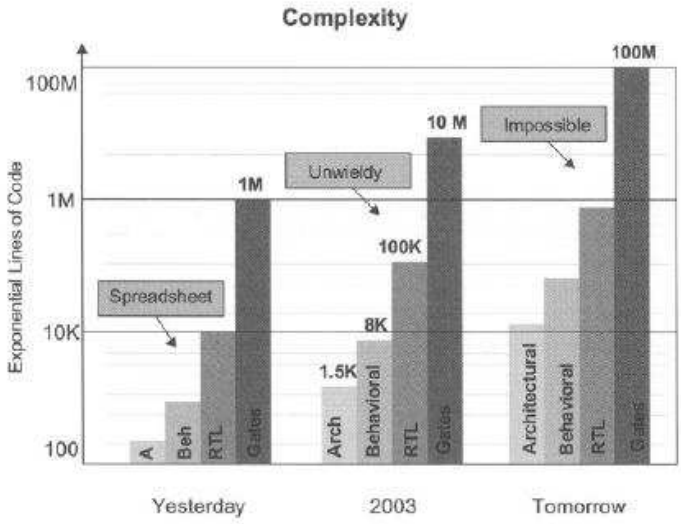

Figura 1.1: Complexidade no desenvolvimento de sistemas em diversas gerações [1]

Neste contexto, um SoC que utiliza arquitetura reconfigurável, que será descrita adiante, vêm sendo desenvolvido na UnB. O ponto inicial deste RSoC foi desenvolvido por um aluno de mestrado contendo apenas processador, memória e a parte reconfigurável. Este modelo foi descrito em uma linguagem de descrição de hardware de alto nível, de modo que seja possível o começo do desenvolvimento de aplicações para este sistema. Nesta plataforma serão acrescentados gradativamente outros componentes como sensor de imagens, barramento e conversor A/D. Outro aluno de doutorado possui como trabalho a finalização desta complexa plataforma, contendo ao menos um bloco analógico.

1.2 Definição do problema

O nível de abstração em que o hardware é desenhado tem aumentado significativamente com a adoção generalizada de Linguagens de Descrição de Hardware (Hardware Description Languages - HDLs) em relação ao formato da especificação, ou projeto dos pontos de entrada. Este procedimento tem conduzido a um enorme aumento da produtividade com relação a antiga metodologia de desenvolvimento baseada nos dados de entrada. O salto em termos de produtividade surgiu porque linguagens como VHDL e Verilog permitiram desenvolvedores especificar a complexa funcionalidade ao nível comportamental e de RTL (Register Transfer Level) de forma relativamente sucinta em relação a abordagem precedente que era unicamente estrutural. No entanto, após uma década de sucesso desta forma de implementação, parece que a atual geração de HDLs estão insuficientemente equipadas para tratar da crescente complexidade do desenvolvimento de hardware e do desenvolvimento a nível de sistema [2].

Para especificar, projetar e implementar sistemas muito complexos, incorporando funcionalidades implementadas em hardware e software, se é obrigado a ir além das HDLs antigas. Deve-se ir além do nível RTL de abstração usadas com estas HDLs, alcançando o que foi denominado *nível de sistema* de desenvolvimento. Para isso precisa-se de uma linguagem que possa apoiar este nível [5].

Várias linguagens surgiram para tratar os diferentes aspectos da concepção de sistemas eletrônicos. Embora Ada e Java terem provado seus valores, C/C++ é predominantemente utilizado hoje para softwares de sistemas embarcados. As linguagens VHDL e Verilog, são utilizadas para simular e sintetizar circuitos digitais. Vera é uma das opções de linguagens para verificação funcional de complexos circuitos integrados para aplicação específica (ASIC). SystemVerilog é uma linguagem relativamente nova que evoluiu da linguagem Verilog para tratar de muitos problemas de projeto em sistemas orientados a hardware. Matlab e várias outras ferramentas e linguagens como SPW e System Studio são amplamente utilizadas para captura de requisitos de sistemas e desenvolvimento de algoritmos processadores de sinais [1].

A figura 1.2 ressalta a aplicação de várias linguagens utilizadas para descrição de hardware. Cada linguagem possui suas peculiaridades, às vezes até mesmo extrapolando as utilidades de seu domínio primário, como pode-se observar nas sobreposições da mesma figura.

Também já não é mais produtivo para os desenvolvedores modelar ao nível de bits individuais, com isso surge a necessidade de uma abstração de dados mais sofisticada [2]. Além disso, o hardware

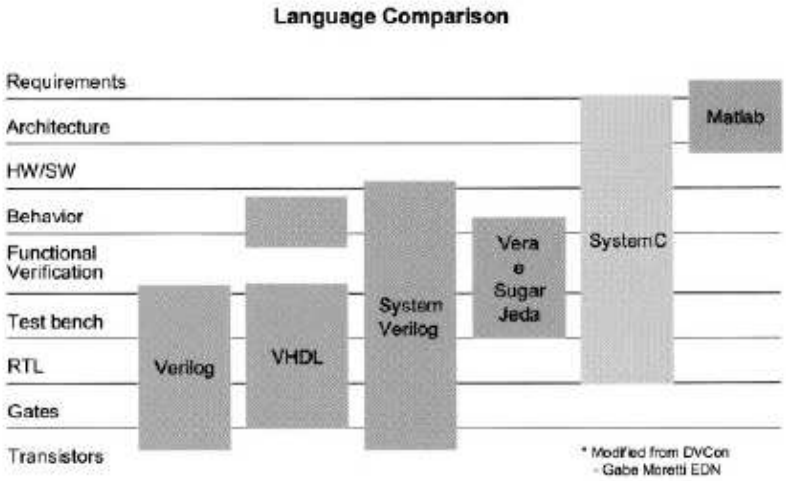

Figura 1.2: Comparação entre linguagens [1]

deixou de ser projetado como uma entidade independente. Módulos de hardware frequentemente coexistem em um mesmo chip com processador, software embarcado, e outros complexos blocos de IP, obrigando os desenvolvedores a realizar lentas e insuficientes co-simulações de partes do hardware e software quando tentam simular o funcionamento de todo o sistema integrado. Com isso torna-se necessário um mecanismo claro e objetivo para manipular componentes de software e hardware em um mesmo ambiente.

Além disso, a aparente estabilização do incremento de frequências de processamento em um mesmo núcleo de processador devido a barreiras físicas nos leva a procurar novos modos de aumentar o poder de processamento. Uma das alternativas é o processamento paralelo, seja pelo uso de vários núcleos em um mesmo processador, seja pelo uso de arquiteturas reconfiguráveis, que reúnem a versatilidade de processadores de uso geral e o desempenho de processadores de uso específico.

1.3 Objetivos do projeto

Utilizando uma plataforma reconfigurável já descrita em SystemC, busca-se mostrar a utilidade e versatilidade deste tipo desenvolvimento de circuitos em alto nível. Esta amostra procurará se basear em resultados de simulação modelando um sensor APS (Active Pixel Sensor) e integrando-o na arquitetura reconfigurável. Após pronto, procura-se executar uma aplicação de JPEG em imagens fornecidas pelo sensor. Deste modo será permitido comparar o desempenho de dois sistemas microprocessados, com e sem a arquitetura reconfigurável. Além disso, como o SystemC é um padrão de plataforma para modelagem baseado em C++ que aborda as questões discutidas acima e que possibilita abstração de design em níveis RTL, comportamental e de sistema, toma-se também como objetivo neste projeto o estudo, análise e aplicação da linguagem SystemC de forma a tornar possível a representação em alto nível desta arquitetura reconfigurável completa.

1.4 Apresentação do manuscrito

No capítulo 2 é feito um resumo com os diversos tópicos relevantes à este trabalho. Em seguida, o capítulo 3 descreve a metodologia empregada no desenvolvimento do projeto. A execução do projeto é discutida no capítulo 4. Os resultados obtidos experimentalmente são apresentados e discutidos no capítulo 5. Ao final, tem-se as conclusões e indicações de trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Como este trabalho reúne vários campos de conhecimento, este capítulo foi escrito com o objetivo de facilitar a compreensão do projeto. Para começar o desenvolvimento do sensor APS e da arquitetura reconfigurável, é imprescindível o entendimento da linguagem base SystemC e também do conceito de nível de modelagem a nível de sistema, abordados primeiramente. Em seguida aborda-se a descrição de sistemas reconfiguráveis. Devido a implementação do APS um estudo sobre sensores de imagem é exposto. Após isto, para o desenvolvimento da aplicação de compressão de imagem é necessário o estudo do JPEG, seção dentro da qual é feita uma introdução ao formato de arquivo utilizado no trabalho, o Portable Grey Map (PGM).

2.1 SystemC

SystemC é uma emergente linguagem de desenvolvimento de sistemas que evoluiu em resposta a uma profunda necessidade de uma linguagem que melhore produtividade global para criadores de sistemas eletrônicos. Diversos sistemas atuais contém hardware de aplicação específica e software. Além disso, o hardware e software são muitas vezes co-desenvolvidos obedecendo estreitos prazos, com estreitas restrições de performance para tempo real sendo que, quando concluido, a verificação funcional necessária para evitar falhas é por vezes cara e problemática. SystemC oferece ganhos reais de produtividade para os engenheiros por permitir que se desenvolvam juntos tanto componentes de hardware como de software do mesmo modo como existiriam no sistema final, mas em um elevado nível de abstração. Este alto nível de abstração possibilita à equipe de projeto uma compreensão fundamental do sistema, ainda no início do processo conceptivo de interações e complexidade do sistema como um todo, possibilitando encontrar erros de arquitetura em antecipação, se comparado ao trabalho com as HDLs tradicionais. Isto permite melhores mudanças de sistema, antecipações e aprimoramentos nas verificações e sobretudo ganhos de produtividade através da reutilização antecipada de modelos de sistemas [1].

O SystemC possibilita a abstração de design em níveis RTL, comportamental e de sistema. Além de mais simples, o projeto de sistemas usando o SystemC se torna menos custoso computacionalmente na medida em que a simulação em nível RTL consume muito mais recursos computacionais do que a simulação em nível de sistema. Consistindo em uma biblioteca de classes

e um núcleo de simulação, a linguagem é uma tentativa de padronização de uma metodologia de projeto em C/C++. É patrocinada pela Open SystemC Iniciative (OSCI), um consórcio de várias empresas do ramo de desenvolvimento de sistemas eletrônicos. Além das vantagens de modelagem disponíveis em C++ tais como abstração de dados, modularidade e orientação a objetos, as vantagens do SystemC incluem o estabelecimento de um ambiente comum de projeto consistindo em bibliotecas de C++, modelos e instrumentos, assim estabelecendo: uma base para o co-design hardware-software, a capacidade de permutar IP fácil e eficientemente e a capacidade de reusar testbenches entre os diferentes níveis de abstração em modelagem. O SystemC facilita a troca de módulos pré-fabricados que possuem direitos autorais, as propriedades intelectuais (Intelectual property - IP) na medida em que simplifica a elaboração de sistemas.

Para que uma linguagem seja aceitável para a concepção de vários níveis de abstração, é importante que seu poder expressivo corresponda ao das linguagens de descrição de hardware atuais. SystemC providencia mecanismos para modelar a típica funcionalidade de um hardware por meio de idéias análogas às das HDLs já conhecidas.

2.1.1 Arquitetura do SystemC

A estrutura global da biblioteca de classes do SystemC está resumida na figura 2.1. O núcleo de simulação, que é um leve agendador responsável por ativar e suspender os processos de SystemC é o fundamento da implementação e constitui a camada base. O mecanismo de eventos gerais abordado na seção 2.1.6.1, que constitui a base da sincronização, é introduzido na próxima camada. Com estas duas camadas como fundações, os elementos de comunicação (interfaces, canais, e portas) são definidos na camada sobresequente. O modelo é baseado no esquema interface-method-call (IMC). Este basicamente especifica que as portas acessam os canais apenas através das interfaces. O exemplo de canais primitivos fornecido pelo SystemC é construído nesta camada. Por último, a hierarquia e outros canais definidos pelo usuário são construídos na camada mais alta. A observação mais importante é que as camadas superiores são construídas de forma concisa sobre as camadas inferiores, e os projetistas podem utilizar os mecanismos de modelagem em qualquer um dos níveis.

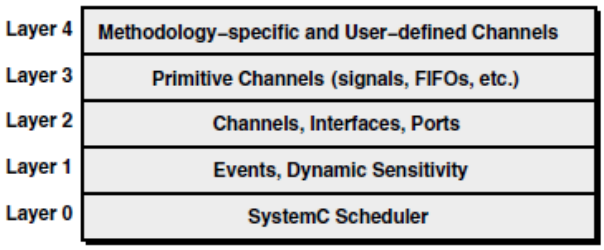

Figura 2.1: Arquitetura do SystemC. Camadas superiores são construídas de forma clara e concisa sobre as camadas inferiores. [2]

2.1.2 Simulação

O desenvolvedor pode escrever modelos em SystemC ao nível de sistema, comportamental ou RTL usando C/C++ acrescido da biblioteca de classes SystemC. Esta serve para dois importantes

propósitos. Em primeiro lugar, implementa muitos recursos que são específicos de hardware, tais como concorrência e hierarquia de módulos, portas e clocks. Segundo, ele contém um núcleo leve de agendamento de processos. O código em SystemC pode ser compilado e ligado com a biblioteca de classes fazendo uso de qualquer compilador C++ padrão (tal como o gcc do projeto GNU). Depois pode-se usar o próprio executável resultante como simulador para o projeto. O teste de corretude do modelo (testbench) é também escrito em SystemC e compilado juntamente com o sistema a ser desenvolvido. O executável pode ser depurado em qualquer ambiente familiar ao C++ (tal como o gdb do projeto GNU). Além disso, arquivos de monitoramento (trace files) podem ser gerados para visualizar a evolução de sinais selecionados usando alguma ferramenta de visualização de formas de onda. A figura 2.2 ilustra a típica metodologia de simulação em ambiente SystemC.

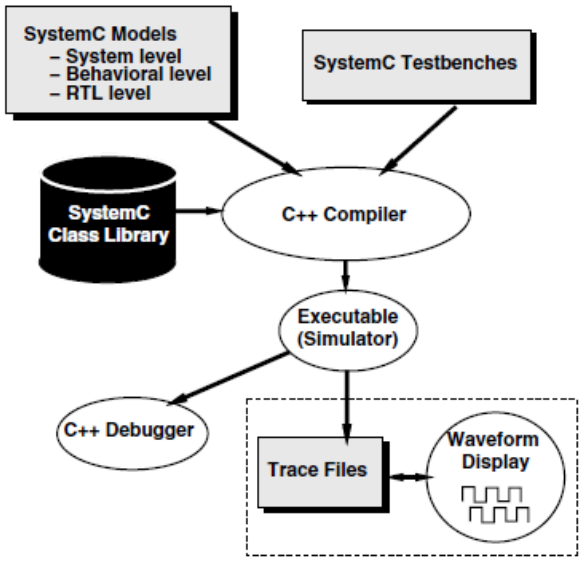

Figura 2.2: Metodologia de simulação em SystemC [2]

Trazer um ambiente tradicional de desenvolvimento de software para o cenário de desenvolvimento de hardware confere algumas poderosas vantagens. A sofisticada infraestrutura de desenvolvimento de programa já disponibilizada para C/C++ pode ser diretamente utilizado para tarefas de verificação e depuração em SystemC.

Para os projetistas de hardware já acostumados a visualizarem em forma de onda os dados de simulações, os arquivos de monitoração (trace files) oferecem uma interface amigável. Conceitualmente, a mais forte característica é que partes de hardware, software, e testbench do projeto podem ser simulados em um simples e unificado ambiente sem a necessidade de desajeitadas co-simulações utilizando diferentes paradigmas de modelagem.

2.1.3 Núcleo de simulação

O núcleo de simulação do SystemC segue o paradigma avaliar-atualizar que é comum em HDLs. O conceito de ciclos delta é implementado, onde múltiplas fases de avaliar-atualizar podem ocorrer ao mesmo tempo na simulação. Uma versão simplificada do algoritmo de simulação seria:

1. Início: Executa todos os processos para iniciar o sistema
2. Avaliação: executa um processo que está pronto para funcionar. Itera até que todos os processos prontos sejam executados. Os eventos que ocorrem durante a execução podem acrescentar novos processos para a lista de processos prontos.
3. Atualização: Executa qualquer chamada de atualização feita no passo 2.
4. Se notificações atrasadas estiverem pendentes , determina lista de processos prontos e procede para a fase de avaliação (passo 2).
5. Avança o tempo de simulação para a primeira notificação temporizada pendente. Se não existe tal caso, a simulação é terminada. Se existe, determina os processos prontos e prossegue ao passo 2.

2.1.4 Implementação e análise

A característica mais importante do fluxo de execução em SystemC é que a especificação é feita em linguagem comum tanto para partes de hardware quanto para software. Na modelagem SystemC em nível de sistema, as duas partes são ainda indistinguíveis durante a concepção dos objetivos de módulos para hardware ou software, que ainda não foram definidos. Assim, mudanças das diferentes partes do projeto de software ou hardware em execução podem ser exploradas de uma forma contínua, eliminando a necessidade de reimplementar cada módulo em C++ e em HDL. Isto também elimina a necessidade do uso de ferramentas de projeto de sistemas para compreender e analisar a sintaxe e semântica dos dois tipos diferentes de ambientes de modelagem.

Um típico fluxo de síntese/execução de cima para baixo (top-down) é ilustrado na figura 2.3. A entrada de projeto pode ser em qualquer nível de abstração: nível de sistema (que poderia ser um modelo funcional atemporal, ou um modelo em nível de transação(TLM)), nível comportamental, ou a nível RTL. A transição de um nível alto para um nível mais baixo de abstração poderá ser feito através de ferramentas de compilação e síntese automática (tais como particionamento hardware/software e ferramentas co-sintetizantes para determinar qual porção do projeto é sintetizada em portas e qual porção é compilado em software embarcado; e ferramentas de síntese comportamental), ou através de um processo manual de refinamento. Por último, o projeto em nível RTL, se gerado à mão, é a entrada para ferramentas de síntese lógica e física já conhecidas por desenvolvedores de hardware. A saída é uma netlist em nível de portas.

A linguagem de especificação permanece a mesma em todos os níveis de síntese, e as alterações no nível de abstração envolvem um refinamento em um maior detalhamento ainda dentro da mesma linguagem e ambiente de desenvolvimento. Isto permite, por exemplo, que o mesmo testbench seja utilizado para verificar o design em vários níveis, se cuidadosamente projetados, resultando em um ambiente de desenvolvimento que é rigorosamente integrado. Assim sendo, como C++ possui muitas estruturas que não estão relacionadas com hardware, adequados tipos de ferramentas terão que ser utilizados para síntese.

Com um refinamento de modelo, procedemos de uma especificação abstrata para uma mais detalhada, em termos de dados ou de comunicação. Refinamento de dados envolve determinação

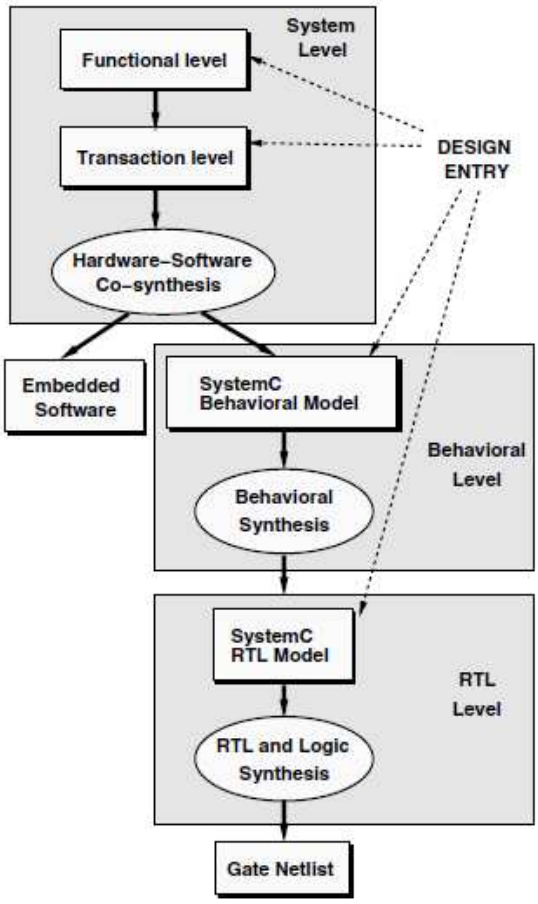

Figura 2.3: Fluxo de síntese em SystemC [2]

do exato número de bits de cada um dos itens de dados, e é tipicamente realizado no final da fase de projeto, isto é, nas fases de desenvolvimento comportamental e RTL. De modo oposto, o refinamento em comunicação tipicamente ocorre em antecipadas fases de concepção do sistema: inicialmente, os modelos de sistema podem empregar operações abstratas de comunicação que, após verificação, necessitam ser substituídas por implementação mais realista. SystemC provê um poderoso mecanismo de refinamento em comunicação, o qual pode ser facilmente realizado primeiro fixando a interoperabilidade dos canais de comunicação e então substituindo protocolos abstratos por implementações concretas. Funcionalidades como esta, que estão ausentes nas atuais HDLs, fazem do SystemC uma atrativa linguagem de desenvolvimento a nível de sistema.

2.1.5 Modelagem de Hardware

2.1.5.1 Estrutura e hierarquia

Módulos

A decomposição estrutural é um dos conceitos fundamentais em modelagem de hardware porque ajuda a fracionar projetos complexos em partes menores. Em SystemC, a decomposição estrutural é realizada com módulos, que são os elementos básicos de construção. Uma descrição em SystemC

consiste de uma série de módulos, cada um encapsulando algum comportamento ou funcionalidade. Módulos podem ser hierárquicos, contendo instâncias de outros módulos. O aninhamento de hierarquia pode ser arbitrariamente profundo, sendo um requisito importante para a representação do projeto estrutural.

Sinais e portas

O modo mais simples de se conectar diferentes módulos em SystemC é utilizando portas e sinais. Na realidade, a interface dos módulos com o mundo externo pode ser muito mais geral e sofisticado, e é descrito na seção sobre interfaces, mas a interface no seu mais baixo e primitivo nível compara-se aos típicos recursos já disponíveis nas atuais HDLs. Cada porta possui uma direção associada a ela, que pode ser: entrada, saída ou bidirecional.

A figura 2.4 mostra uma estrutura simples consistindo de um módulo C com instâncias hieráquicas de dois módulos A e B dentro dele (as instâncias identificadas como A1 e B1, respectivamente) com as seguintes características:

- Módulo A: portas de entrada a1 e a2 e porta de saída a3
- Módulo B: portas de entrada b1 e b2 e porta de saída b3
- Módulo C: portas de entrada c1 e c2 e porta de saída c3

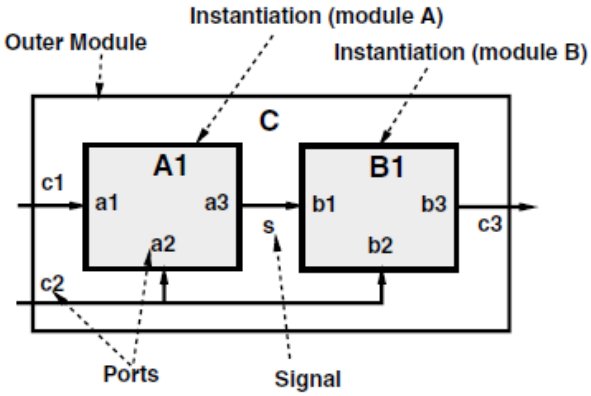

Figura 2.4: Exemplo da estrutura modular em SystemC [2]

As portas estão conectadas como mostrado na Figura 3. A descrição SystemC da estrutura acima possui o seguinte aspecto:

```
SC_MODULE (A) {          // declaração de módulo
        sc_in<bool> a1; // declaração de portas
        sc_in<bool> a2;
        sc_out<bool> a3;
        // comportamento omitido
};
SC_MODULE (B) {
        sc_in<bool> b1;
        sc_in<bool> b2;
        sc_out<bool> b3;
```

```
        // comportamento omitido
};
SC_MODULE (C) {
        sc_in<bool> c1;
        sc_in<bool> c2;
        sc_out<bool> c3;
        A *A1;
        B *B1;
        sc_signal<bool> s; // declaração de sinais
        SC_CTOR (C) {
                A1 = new A ("A1"); // instanciamento de módulo
                (*A1) (c1, c2, s); // mapemaneto de portas
                B1 = new B ("B1");
                (*B1) (s, c2, c3);
        }
};
```

Um módulo é declarado com a macro SC_MODULE, e as portas são especificadas com as expressões sc_in, sc_out, e sc_inout, com o parâmetro <bool> indicando que é do tipo booleano (único bit). Outros tipos de dados, incluindo aqueles definidos pelo usuário, podem ser utilizados como tipos de portas. A hierarquia estrutural é especificada dentro do construtor do módulo, especificado com a macro SC_CTOR. A declaração e chamada de ponteiro *new* para A1 e B1 estabelece uma instância do módulo. Os parâmetros de mapeamento de portas ligam as portas c1, c2, e o sinal s a três portas de A1. Assim, um sinal s serve como um fio conectando a porta de saída a3 de A1 para a porta de entrada b1 de B1.

2.1.5.2 Processos

As funcionalidades de um sistema em SystemC são descritas em processos. Análogos aos processos em VHDL, os processos em SystemC são utilizados para representar comportamento concorrente: múltiplos processos dentro de um módulo representam blocos executando em paralelo. Processos possuem uma lista de sensibilidade associada à uma lista de sinais que desencadeiam a execução do processo. Existem dois tipos importantes de processos:

Métodos

Um método se comporta como uma chamada a uma função e pode ser utilizado para modelar um comportamento combinacional simples. Não possui um fluxo próprio de execução, por isso, não pode ser suspenso. Essa característica peculiar permite alta eficiência na simulação.

Threads

Uma *thread* pode ser utilizada para modelar comportamento sequencial. Ela é associada a sua própria sequência de execução e por isso pode ser suspenso e depois reativado. Um simples exemplo que envolve métodos e *threads* é mostrado abaixo. As funções p e q (cujas definições foram omitidas) são registradas como um processo método e thread, respectivamente. A lista de sensibilidade é especificada usando a palavra *sensitive* e o operador «.

```
SC_MODULE (X) {
```

```
        sc_in<bool> a, b;
        void p();                // A definição da função foi omitida
        void q();
5       SC_CTOR (X) {
                SC_METHOD (p); sensitive << a;
                SC_THREAD (q); sensitive << a << b;
        }
};
```

2.1.5.3 Temporização e clock

Uma vez que os conceitos de tempo e clock são muito importantes em modelagem de hardware, SystemC provê um mecanismo para especificá-los. Um clock com período de 10 ns pode ser definido como:

```
sc_clock clk ("clk", 10, SC_NS);
```

De acordo com as palavras *sensitive, sensitive pos, e sensitive neg* pode ser especificada a sincronização com um pulso de clock e suas bordas positivas ou negativas, respectivamente. No exemplo de código a seguir, o processo x é ativado na borda positiva do clock.

```
SC_THREAD (x);
sensitive_pos << clk;
```

2.1.5.4 Testes de execução

O projeto de uma bateria de testes é uma parte importante da modelagem de hardware e consome uma quantidade significativa de tempo. Em SystemC, uma bateria de teste pode ser especificada usando SC_THREAD, como qualquer outro processo, sendo facilmente integrada à concepção global. Sofisticadas baterias de teste podem ser construídas utilizando todos os construtos disponíveis em C++, em contraposição com as capacidades relativamente primitivas do VHDL e Verilog com respeito a, por exemplo, arquivo de entrada e saída, abstração de dados e processamento de texto. Uma vez que o conjunto de testes não necessita ser sintetizada, não há necessidade de se adequar a nenhum subconjunto sintetizável de C++ enquanto se escreve os testes.

2.1.5.5 Tipos de dados

Além dos tipos de dados padrões em C++ como int, bool, char, etc. SystemC provê um rico conjunto de tipos de dados que podem ser utilizados para modelar conceitos específicos para hardware. Um tipo bastante útil de dados suportado é a lógica de quatro estados. Além dos tradicionais valores '0' e '1', seria útil fornecer um mecanismo para indicar que o valor de um bit é desconhecido. Isto permitiria identificar problemas de inicialização durante as simulações. Além disso, existe a necessidade de se especificar o estado de alta impedância (ou tristate) de sinais. Com isso, SystemC fornece o tipo sc_logic, um tipo com 4 estados, sendo '0' (ou false), '1' (ou true), 'X' (desconhecido), e 'Z' (alta impedï£¡ncia ou tristate). Um vetor de tipo de dados para

lógica, sc_lv, é utilizado para indicar dados maiores que um bit e que precisam de ser modelados usando a lógica com quatro estados. Por exemplo, um barramento pode ser inteiramente definido como alta impedância da seguinte forma:

```
sc_lv<8> data; // 8 bits wide
data = "ZZZZZZZZ"; // set to high impedance
```

2.1.6 Modelagem em nível de sistema

As características do SystemC revisadas na seção anterior fazem dele uma adequada linguagem de descrição de hardware. No entanto, as referidas capacidades apenas deixariam o SystemC com uma ligeira melhoria em relação às HDLs já existentes. A verdadeira vantagem do SystemC como uma linguagem de especificação reside no fato de que ela abrange todas as importantes características de modelagem de hardware, bem como provê poderosos recursos para o projeto a nível de sistema. Isto garante que a transição para uma metodologia baseada em SystemC não demanda nenhum compromisso em termos de força expressiva a níveis mais baixos de abstração, e proporciona ainda um enquadramento útil para a modelização em níveis mais altos.

As HDLs atuais carecem fortemente desta última capacidade. As características da modelagem em nível de sistema introduzidas no SystemC 2.0 incluem principalmente o apoio de meios de comunicação entre processos muito mais gerais e abstratos e um mecanismo mais geral para a sincronização de eventos.

2.1.6.1 Eventos e sensitividade

SystemC 2.0 introduz um mecanismo geral para especificação e notificação de eventos. Os eventos não são disparados pela mudança dos bits, mas são tipos abstratos que representam interações mais gerais e complexas. O tipo sc_event pode ser utilizado para declarar eventos que possam ser criados usando a palavra *notify* e ser sincronizados com declarações de espera (wait), conforme segue:

```
sc_event e1, e2;          // declaração de eventos
sc_time t (5, SC_NS);
e1.notify (t);            // notificar evento e1 depois de 5 ns
wait (e1);                // suspender execução até a
                          // ocorrência de e1
wait ();                  // aguarda até que um evento ocorra
                          // na lista de sensitividade de processos
wait (10, SC_NS, e1 | e2);
                          // aguarda a ocorrência dos eventos e1 ou e2
                          // mas no máximo por 10ns
```

2.1.6.2 Canais e interfaces

Na seção 2.1.5.1 ilustra-se um simples exemplo de módulos conectados através de sinais. Este quadro de comunicação onde a interação entre módulos está restrita à passagem de valores em fios

individuais está, no entanto, no mais baixo nível de abstração. Em nível de sistema precisa-se poder modelar usando paradigmas de comunicação mais abstratos, sofisticados e inteligentes. O SystemC 2.0 introduz o conceito de canais e interfaces que asseguram esta capacidade de modelagem. O projeto em nível de sistema em SystemC consiste de uma série de módulos e canais em alto nível. Simplicando, os módulos encobrem os aspectos relacionados com a funcionalidade do sistema e os canais modelam os aspectos relacionados com a comunicação. Canais podem ser muito gerais, implementando complexos algoritmos, por exemplo, um complexo protocolo de barramento com arbitragem. Canais podem ter uma estrutura hierárquica. A figura 2.5 mostra um simples projeto com três módulos conectados a um canal.

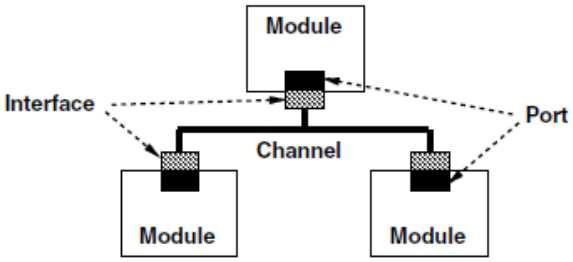

Figura 2.5: As portas dos módulos são ligadas a canais através de interfaces [2]

Uma interface consiste de um conjunto de declarações de métodos (não relacionadas a SC_METHODs) implementados pelo canal. Estes métodos são visíveis a uma porta que está conectada ao canal através da interface. Assim, a porta (e com isso, o módulo) está isolada dos detalhes de implementação do próprio canal. Esta arquitetura ajuda a manter as partes relacionadas à funcionalidade de um projeto separadas da comunicação e permite a modificação de um sem afetar o outro enquanto a interface não for mudada. A idéia de interface e canal foram inspiradas em conceitos similares da linguagem SpecC [6].

```
SC_MODULE (A) {
        sc_fifo_in<int> in;
        sc_fifo_out<int> out;
                        // resto omitido
};
...
A *M1, *M2;             //instâncias do módulo A
...
sc_fifo<int> q (5);     // cria um canal do tipo FIFO
                        // com tamanho 5 de buffer
M1->out (q);            // conecta porta de saída de M1 a q
M2->in (q);             // conecta porta de entrada de M2 a q
```

2.1.7 TLM

Para lidar com a crescente complexidade e pressão na velocidade de entrega ao mercado de projeto de sistemas em chip, a abstração de desenvolvimento foi elevada ao nível de sistema para aumentar a produtividade. O desenvolvimento a nível de sistema possibilita tomar decisões em

maiores níveis de abstração e reutilizando componentes de desenvolvimento. No nível de modelagem transacional (TLM), os detalhes de comunicação entre componentes computacionais são separados dos detalhes de implementação destes componentes. Comunicações são implementadas como canais e pedidos de transação são feitos chamando funções de interface destes modelos de canais. Detalhes desnecessários da comunicação e computação são ocultados pelo TLM e podem ser trabalhados em fases mais adiantadas do projeto. Modelagem a nível transacional possibilita o decréscimo do tempo de simulação, explorando e validando alternativas de implementação em níveis mais elevados de abstração [7].

O nível de modelagem transacional envolve a comunicação entre processos do SystemC usando chamadas de funções. Seu foco está mais na comunicação entre estes processos do que nos algoritimos internos ao processo. Assume-se que na modelagem do comportamento de um sistema enquanto alguns processos produzem dados, outros requisitam dados, alguns começam comunicações e outros passivamente respondem comunicações iniciadas por outros. TLM é especialmente desenvolvido para sistemas em chip mapeados em memória. Isto não quer dizer que o TLM é dedicado exclusivamente para este fim, mas que a maioria de seus recursos estão voltados para isto. O TLM possui uma estrutura em camadas, com as mais baixas sendo mais flexíveis e gerais, e as mais altas sendo específicas para modelagem de barramentos.

2.2 Arquiteturas reconfiguráveis

As duas abordagens tradicionais de arquiteturas de computadores são referentes aos processadores de propósito geral e aos dispositivos de propósito específico. Os processadores de propósito geral são também conhecidos como arquiteturas de Von Neumman. Esses dispositivos possuem um alto grau de flexibilidade sendo capazes de realizar quase todos os tipos de computação. Já os dispositivos de aplicação específica, ou ASICs (Application Specific Integrated Circuits), são direcionados à execução de um número restrito de aplicações. Por esse motivo, alcançam alto desempenho.

Embora os processadores de propósito geral possuam boa flexibilidade inerente, resultado da capacidade de executar diversos tipos de tarefas, não alcançam o alto desempenho dos dispositivos de aplicação específica. Os ASICs, por outro lado, não possuem a flexibilidade dos processadores de uso geral [8].

Arquiteturas reconfiguráveis são dispositivos de hardware capazes de mudar seu comportamento de acordo com as restrições da aplicação. Essa propriedade permite que dispositivos reconfiguráveis alcancem altas taxas de desempenho, que podem ser comparadas a arquiteturas de aplicação específica (ASICs), além de exibirem alto grau de flexibilidade.

Muitas arquiteturas reconfiguráveis podem ser encontradas na literatura. Mais detalhes sobre essas arquiteturas podem ser encontrados nas referências [9], [8] e [3]. A arquitetura utilizada neste trabalho agora é apresentada.

2.2.1 RoSA

RoSA (Reconfigurable Stream-based Architecture) é uma arquitetura reconfigurável de granularidade grossa que explora o paralelismo a nível de instrução de aplicações baseadas em fluxo de dados [9]. A figura 2.6 apresenta o diagrama de blocos da arquitetura.

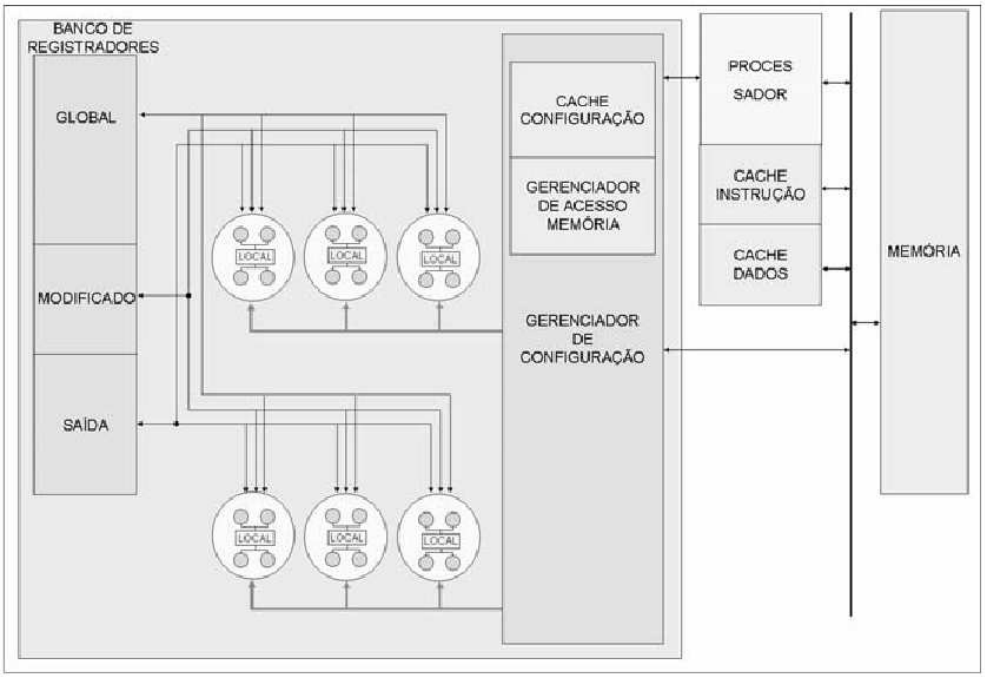

Figura 2.6: Diagrama de blocos do RoSA [3]

A arquitetura RoSA é formada por um bloco reconfigurável acoplado a um processador hospedeiro. O bloco reconfigurável é composto por: células, banco de registradores, um gerenciador de configuração e um gerenciador de acesso à memória, que serão detalhados a seguir.

2.2.1.1 Bloco reconfigurável

As células correspondem a unidades reconfiguráveis que se comunicam através do banco de registradores. Cada célula possui unidades funcionais (UFs) que executam as operações lógicas e aritméticas, um banco de registradores local e um componente que realiza o controle.

A arquitetura RoSA possui quatro bancos de registradores. O primeiro deles, denominado global, armazena as informações modificadas pelo processador apenas no início da execução da aplicação. Incluem-se neste banco os parâmetros das funções e as variáveis globais. O segundo banco, denominado modificado, armazena os dados que são alterados pelo processador em tempo de execução. O terceiro banco armazena as saídas das células. Por fim, o último banco de registradores é o local e armazena os cálculos intermediários realizados pelas UFs de cada célula.

O gerenciador de configuração é a figura principal da arquitetura, responsável por buscar a configuração no cache de configurações (ou na memória caso não a encontre) e distribuí-la entre as células, bem como gerenciar os registradores da arquitetura e as saídas das células. A configuração de cada célula corresponde a uma palavra de instrução longa (120 bits) e a configuração para toda

a arquitetura é uma VLIW (Very Long Instruction Word) de 720 bits, que inclui a configuração das seis células. O controle da célula tem a capacidade de receber a configuração e combiná-la com a arquitetura da célula, buscando os dados e executando as operações nas UFs correspondentes.

A Figura 2.7 ilustra o formato de uma instrução de configuração de todo o bloco reconfigurável e a sintaxe de configuração da célula. A descrição dos campos de cada configuração é apresentada a seguir.

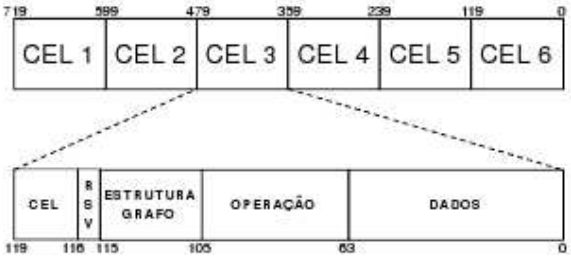

Figura 2.7: Instrução global de configuração [3]

- DADOS: indica o endereço dos registradores da lógica reconfigurável que armazenam os dados de entrada. Baseado na árvore de operações mais larga encontrada (ver figura 2.8), existem até 8 dados de entrada e são necessários 8 bits de endereçamento para cada dado. Portanto esse campo possui 64 bits.
- OPERAÇÃO: indica quais operações serão executadas na célula. Esse campo determina todas as operações de uma árvore de operações. Baseado na árvore mais larga, onde são executadas sete operações, e com 6 bits para indicar cada operação, a largura total desse campo é de 42 bits.
- CEL: possui 3 bits que indicam a qual das seis células a configuração pertence.
- ESTRUTURA DO GRAFO: possui 10 bits que indicam quantas operações são executadas em paralelo (limite máximo de 4 operações paralelas).

As especificações do RoSA foram baseadas em diversos estudos [3]. Desta forma convencionou-se que a mais larga estrutura de dados que uma célula pode receber seria de quatro operações de largura e três de profundidade. Do mesmo modo, adotou-se que a estrutura mais comprida possuiria 5 operações enfileiradas. Dessa forma, cada dois bits indicam quantas operações existem em cada nível. A combinação desse campo com o campo OPERAÇÃO determina o formato da estrutura de dados em relação à largura e altura. A Figura 2.8 exemplifica como os dois campos são combinados para obter a árvore de operações a ser executada na célula.

Inicialmente, a unidade de controle da célula lê o campo ESTRUTURA DO GRAFO e após conhecer o número de operações existentes no primeiro nível, essas operações são buscadas no campo correspondente, na ordem indicada na figura. Esse processo é feito repetidamente até que não existam mais níveis (indicado com 00). Quando o campo OPERAÇÃO não possui todas as sete operações, isso é indicado com um valor referente a NOP (no operation).

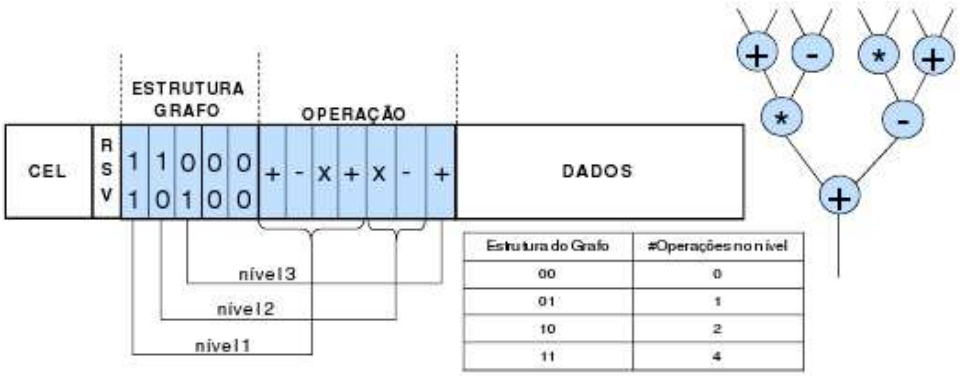

Figura 2.8: Configuração de célula com exemplo de estrutura de operações [3]

A unidade de controle da célula ao receber a configuração, combina essa informação com a arquitetura da célula (isto é feito através da busca de operações e dados nas UFs correspondentes) e associa os registradores de saída das UFs com os registradores de entrada.

Quando uma chamada ao bloco reconfigurável é encontrada durante a execução da aplicação, o processador hospedeiro carrega os dados nos bancos de registradores e chama o gerenciador de configuração para buscar e carregar a configuração no RoSA. O cache de configurações é verificada antes de buscar uma configuração na memória e atualizada quando esta não é encontrada (ocorre miss). O cache da arquitetura RoSA possui mapeamento associativo e sua política de substituição é a FIFO (First In First Out), onde o primeiro dado a entrar é o primeiro a sair. O tamanho do cache foi definido através da análise das aplicações utilizadas como estudo de casos. A partir da análise, constatou-se que um cache seria composto por 8 blocos.

O gerenciador de acesso à memória é responsável por atender as requisições de memória feitas pelas células. Esse componente recebe todas as requisições de leitura e escrita feitas pelas células e as armazena em uma fila de requisições. Essa abordagem evita incoerência entre os dados das células e da memória. Mais detalhes sobre a arquitetura podem ser encontrados em [9].

2.3 Sensores de imagem

Um sensor de imagem digital, agindo como a retina dos olhos, capta a luminosidade das imagens que são projetadas sobre ele continuamente e dá início ao processo de captura de uma instância ou de uma sequência de instâncias da imagem consecutivamente. Trata-se de um chip que pode contar com dezenas de milhões de transdutores fotossensíveis, cada um deles capaz de converter a energia luminosa de uma parte da imagem em carga elétrica para ser lida ou gravada posteriormente na forma de imagem digitalizada em valores numéricos.

Para captação de imagem a cores, é comum câmeras de vídeo usarem três sensores (sistema 3CCD), cada sensor com um filtro de uma tripla de filtros tricrômicos sobre ele, sendo que câmeras fotográficas geralmente contam com um único sensor de imagem que agrupa seus photosites sob um mosaico de filtros de luminosidade e de cor.

2.3.1 APS

Um sensor de pixel ativo (APS) é um sensor de imagem que consiste em um circuito integrado contendo uma sequência de sensores de pixel, com cada pixel contendo um fotodetector e um amplificador ativo. Apresenta como principais vantagens: custo reduzido, devido a tecnologia CMOS, e a possibilidade de implementação do circuito APS em conjunto com circuitos digitais, microprocessadores, memórias e circuitos analógicos [10]. Maiores detalhes sobre parâmetros, arquitetura e aplicações do sensor podem ser encontradas em [10].

2.4 JPEG

A interface gráfica é uma grande preocupação para programadores e desenvolvedores de aplicações. São gastas grandes quantias de tempo e esforço tentando acomodar a proliferação de interfaces gráfica do usuário (GUI). Milhões de horas humanas de trabalho e bilhões de dólares são alocados apenas para que sejam feitas melhorias no modo como os programas apresentam seus dados [4]. Neste contexto, um grande problema das imagens sem compressão é sua grande quantidade de dados. Para serem armazenadas consomem muitos recursos do disco. Uma imagem VGA de 256 cores, por exemplo, possui 200 linhas de 320 pixels cada uma, com cada pixel consumindo um byte. Isto significa que uma pequena imagem sem compressão já consumiria um mínimo de 64 KiloBytes. Não é difícil de imaginar aplicações que utilizam milhares de figuras ou figuras de maior resolução. Para solucionar este problema um grupo chamado Joint Photographic Experts Group (JPEG) foi criado para padronizar um formato com perdas para compressão de imagens.

O padrão JPEG de compressão atua resumidamente em três estágios, como ilustrado na figura 2.9.

Figura 2.9: Compressão com perdas no JPEG [4]

2.4.1 DCT

A chave para o processo de compressão discutido aqui é a transformação matemática conhecida como Transformada de Cosseno Discreta (DCT). A DCT pertence à mesma classe de operações matemáticas da Transformada de Fourier Rápida (FFT) e muitas outras. A função destas transformadas é mudar a maneira de representar sinais do domínio do tempo para uma representação no domínio da frequência.

Quando coletamos um conjunto de pontos amostrados de um sinal de áudio, obtemos uma representação no domínio do tempo. Isto é, temos uma coleção de pontos mostrando a amplitude do sinal em níveis de tensão de entrada em vários instantes. A FFT transforma este conjunto

de pontos em um conjunto de valores em frequência que descreve exatamente o mesmo sinal. A figura 2.10 mostra a clássica representação no domínio do tempo de um sinal analógico. Pode-se compor este sinal utilizando-se três diferentes ondas senoidais somadas para formar um simples e um pouco mais complexa forma de onda.

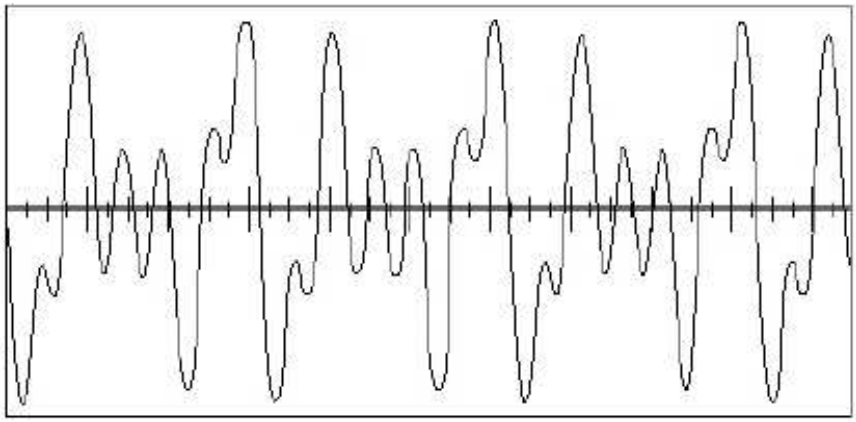

Figura 2.10: Clássica representação de um sinal no domínio do tempo [4]

A figura 2.11 mostra o que acontece ao mesmo conjunto de pontos depois do processamento da FFT. Esta representação mostra a amplitude e a frequência das ondas senoidais que podem ser usadas para recompor o sinal.

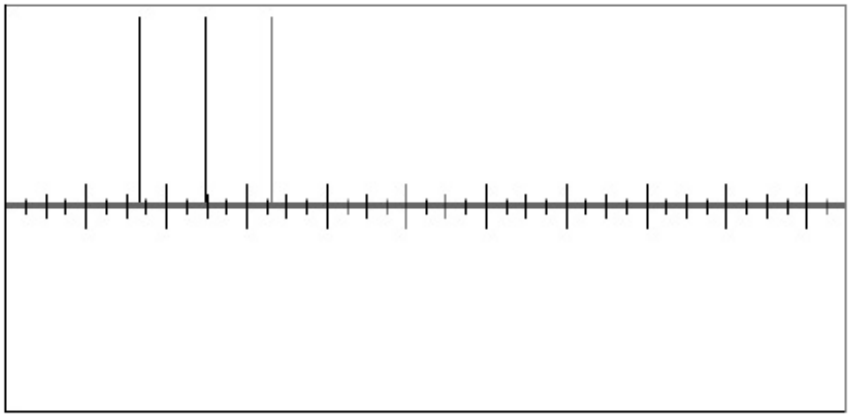

Figura 2.11: Representação do mesmo sinal no domínio da freqüência, depois de aplicada a FFT [4]

Dada esta interpretação da FFT, a figura 2.11 faz sentido imediato, mostrando que o sinal mostrado na figura anterior pode ser representado de uma forma mais econômica, com menos dados. Mostra que o sinal pode ser recomposto em três ondas de diferentes frequências que aparentam possuir a mesma magnitude. Dada esta informação, do mesmo modo pode-se facilmente recompor este sinal para reconstruir a onda da figura 2.10.

Um ponto importante deste tipo transformação é que ela é reversível. Os dois ciclos de transformação são essencialmente sem perdas, exceto pela perda da precisão de resultados das operações matemáticas envolvidas ou de erros de truncamento, muito comum em sistemas computacionais.

A DCT está intimamente relacionada com a Transformada de Fourier e produz um resultado

similar. Ela toma um conjunto de valores no domínio espacial e os transforma em uma representação identica no domínio da frequência; entretanto com uma complicação a mais, pois a DCT vai operar em um sinal tridimensinal plotado em eixos X, Y e Z.

Neste caso o "sinal" é uma imagem. Os eixos X e Y são as duas dimensões da tela. A amplitude do "sinal" neste caso é o valor do pixel de um ponto particular da tela, representado geralmente por um byte, ou seja, por um valor de 0 a 255. Então a imagem pode ser vista como um sinal tridimensional complexo, com o valor no eixo Z sendo a magnitude de cada pixel. Esta é a representação espacial do sinal.

A DCT pode ser usada para converter informação no espaço em informação em "frequência" ou informação "espectral", com os eixos X e Y representando frequências do sinal em duas diferentes dimensões. Assim como a FFT, existe uma DCT inversa (IDCT) que converte a representação espectral de volta a representação espacial.

A fórmula para a DCT bidimensional é mostrada na figura 2.12. Ela é aplicada em uma matriz N x N com valores de pixel, resultando em uma matriz N x N com coeficientes de frequência.

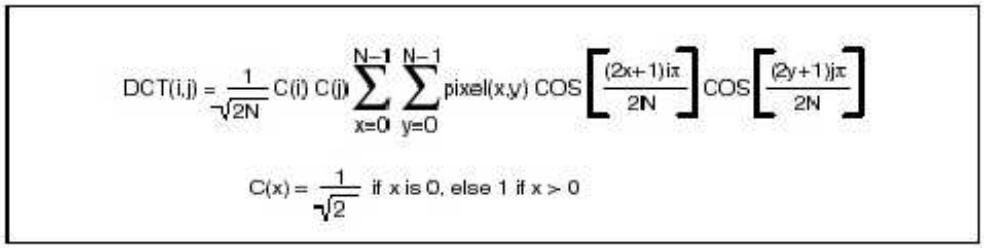

Figura 2.12: Transformada de cosseno discreta (DCT) [4]

Não parece ser óbvio a razão de se fazer esta operação com uma imagem. Depois de aplicar esta operação matemática os pixels são transformados em coeficientes de frequência, mas nós continuamos possuindo o mesmo número de elementos que anteriormente. Seria interessante se esta operação transformasse a matriz N x N em uma matriz N/2 x N/2.

Entretanto, a figura 2.11 indica algo de importante nesta transformada. A figura mostra que a representação do sinal de áudio no domínio da frequência toma toda a informação necessária para descrever a forma de onda e as compacta em três pontos não nulos. Então a priori podemos descrever todos os 512 pontos que compõem a forma de onda da entrada em apenas três pontos no ponto de vista da frequência.

A DCT produz um efeito similar quando é operada na matriz de dados. Na matriz N x N todos os elementos da linha 0 possuem um componente de frequência zero em uma direção do sinal e todos os da coluna 0 possuem um componente de frequência zero na outra direção. A medida que analisamos células se distanciando da linha 0 e da coluna 0, os coeficientes da matriz de dados após a DCT começam a representar componentes de altas frequências, sendo a maior delas representada na posição (N,N) da matriz.

Isto é significante pois a maioria das imagens apresentadas nas telas de computadores são compostas de informação de baixa frequência [4]. Os componentes que se encontram na linha 0 e na coluna 0 (os componentes contínuos ou DC) portam mais informações úteis sobre a imagem do

que os componentes de altas frequências.

Resumindo, a DCT identifica pedaços de informação no sinal que podem ser efetivamente desprezados sem comprometer seriamente a qualidade da imagem. Seria difícil imaginar como fazer a mesma coisa em uma imagem que não tenha sido transformada.

2.4.1.1 Implementação da DCT

Um dos primeiros problemas que surge ao tentar implementar o algoritmo da DCT é que o tempo demandado para calcular cada elemento da matriz é altamente dependente do tamanho da matriz. Como um *loop* duplamente encadeado é usado, o número de cálculos é um exponencial de N: a medida que N cresce, o número de cálculos cresce exponencialmente.

Uma das consequências disto é que seria impossível aplicar a DCT em uma matriz de imagem inteira. Fazer este cálculo com uma imagem pequena de 256 x 256 pixels levaria muito tempo. Para contornar este problema, a imagem é quebrada em blocos de 8 x 8 pixels para prosseguir com o cálculo da DCT. Esta solução foi a adotada pelo comite JPEG e é conhecida como codificação em blocos ("block coding").

A definição da DCT mostrada acima é relativamente direta, com um laço duplamente encadeado se implementado em código. Uma forma consideravelmente mais eficiente de se calcular a DCT é usando operações matriciais. Para executar esta operação primeiro cria-se uma matriz N x N conhecida como matriz da DCT, ou simplesmente C. Esta matriz é definida pela equação na figura 2.13.

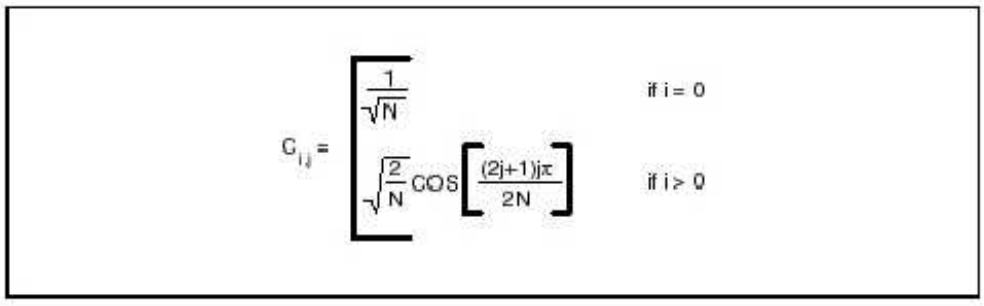

Figura 2.13: Matriz da DCT [4]

Uma vez construída, calculamos a sua transposta, referida como Ct. A construção destas matrizes pode ser feita com o seguinte trecho de código, em linguagem C:

```
for ( j = 0 ; j < N ; j++ ) {
    C[ 0 ][ j ] = 1.0 / sqrt( N );
    Ct[ j ][ 0 ] = C[ 0 ][ j ];
}
for ( i = 1 ; i < N ; i++ ) {
  for ( j = 0; j < N ; j ++ ) {
      C[ i ][ j ] = sqrt( 2.0 / N ) *
              cos( ( 2 * j + 1 ) * i * pi
               / ( 2.0 * N ) );
      Ct[ j ][ i ] = C[ i ][ j ];
```

```
    }
}
```

Com as matrizes prontas, podemos tirar vantagem da definição alternativa da DCT:

$$DCT = C * Pixels * Ct \tag{2.1}$$

Nesta operação o operador "*" se refere a multiplicação de matrizes. Cada fator corresponde a uma matriz 8 x 8, neste caso do algoritmo JPEG. Ao executar uma multiplicação de matrizes, o custo aritmético de cada elemento da matriz de saída será de 8 multiplicações e 8 adições. Como fazemos duas multiplicações, o custo será de 16 multiplicações e adições, um valor menor do que o obtido ao calcular a DCT utilizando a primeira definição apresentada.

2.4.2 Quantização

A figura 2.9 mostra a compressão JPEG como um procedimento de três etapas, com o primeiro sendo uma transformação DCT. A operação com a DCT é sem perda e sozinha não produz nenhuma compressão. Ela apenas prepara os dados para estágio que descarta dados, ou seja, para a quantização dos dados, no processo.

Quantização é simplesmente o processo de reduzir o número de bits necessários para armazenar um valor inteiro, reduzindo a sua precisão. Tendo a imagem passado pelo processo da DCT, pode-se continuamente reduzir a precisão dos coeficientes a medida que se afasta do coeficiente DC da origem. Quanto mais longe do elemento 0,0; menos o elemento contribui para a qualidade da imagem, então é necessário menos rigor na precisão de seu valor.

Um algoritmo JPEG implementa a quantização usando uma matriz de quantização. Para cada posição de elemento na matriz DCT existe um valor correspondente na matriz de quantização. O valor de quantização indica qual o tamanho do passo que será usado na precisão do valor na compressão da imagem. Desta forma os elementos mais importantes serão codificados com um passo menor, sendo 1 o passo de maior precisão. A lógica para quantizar valores é a seguinte:

Valor quantizado(i,j) = (DCT(i,j)/Valor de quantização(i,j)) arredondado para o inteiro mais próximo

Desta fórmula se torna claro que valores de quantização acima de vinte e cinco ou talvez cinquenta assegura que virtualmente todos os componentes de alta-frequência serão arredondados para zero. Apenas se o valor de alta-frequência for muito alto ele não será quantificado como zero. A desquantificação opera no modo reverso:

*DCT(i,j) = Valor quantizado(i,j) * Valor de quantização(i,j)*

Outra vez, percebe-se que quando altos valores de quantização são usados, se corre o risco de gerar grandes erros na recuperação do valor quantizado. Felizmente os erros gerados nos componentes de alta frequência durante a desquantificação normalmente não causam sérios efeitos na qualidade da imagem.

2.4.2.1 Implementação da matriz de quantização

Um grande número de esquemas podem ser usados para se definirem os valores da matriz de quantificação. Ao menos dois critérios podem ser usados para avaliar a escolha deste esquema de criação de valores. Um deles mede o erro matemático dos valores quantizados e posteriormente desquantizados, tentando determinar níveis aceitáveis de erro. O segundo critério julga os efeitos da descompressão aos olhos humanos, que nem sempre corresponde exatamente aos níveis matemáticos de erro. Como esta matriz pode ser obviamente definida com alguma rotina em tempo de execução, JPEG permite o uso de qualquer matriz de quantificação, entretanto existe um padrão ISO para a criação dos valores de quantização baseados em diversas pesquisas. Uma grande vantagem em utilizar matrizes de quantização definidas em tempo de execução é poder trocar quando quiser a taxa de compressão por qualidade de imagem. Escolhendo passos grandes obtém-se grandes taxas de compressão e uma pior qualidade de imagem. O oposto ocorre com pequenos passos. Isto proporciona grande flexibilidade ao usuário.

Uma matriz de quantização pode ser criada uutilizando-se um algoritmo bem simples. Para determinar o valor dos passos de quantização, o usuário fornece um único "fator de quantização" que deve variar de 1 até 25, pois a imagem já está extramente degradada neste valor.

```
for ( i = 0 ; i < N ; i++ )
    for ( j = 0 ; j < N ; j++ )
      Quantum[ i ][ j ] = 1 + ( ( 1 + i + j ) * quality );
```

Um exemplo de matriz de quantização usando um fator de quantização 2 é mostrado na figura 2.15.

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Figura 2.14: Matriz de quantização produzida com um fator de quantização 2 [4]

Para ilustrar este processo, na figura 2.15 é apresentada uma matriz logo após a DCT e a figura 2.16 apresenta a mesma matriz depois do processo de quantificação seguido da desquantificação.

2.4.3 Codificação

A etapa final do processo de compressão JPEG é a codificação para armazenar a imagem quantificada. Esta etapa combina três diferentes passos para comprimir a imagem. O primeiro muda o coeficiente DC em 0,0 de um valor absoluto para um valor relativo. Como blocos adjacentes em uma imagem exibem um alto grau de correlação, codificando o elemento DC como a diferença do elemento DC anterior tipicamente produz um número muito pequeno. Depois, os coeficientes da imagem são arranjados em "sequência zig-zag". Por último eles são codificados usando diferentes

92	3	-9	-7	3	-1	0	2
-39	-58	12	-17	-2	2	4	2
-84	62	1	-18	3	4	-5	5
-52	-36	-10	14	-10	4	-2	0
-86	-40	49	-7	17	-6	-2	5
-62	65	-12	-2	3	-8	-2	0
-17	14	-36	17	-11	3	3	-1
-54	32	-9	-9	22	0	1	3

Figura 2.15: Exemplo de matriz DCT antes da quantização [4]

90	0	-7	0	0	0	0	0
-35	-56	9	11	0	0	0	0
-84	54	0	-13	0	0	0	0
-45	-33	0	0	0	0	0	0
-77	-39	45	0	0	0	0	0
-52	60	0	0	0	0	0	0
-15	0	-19	0	0	0	0	0
-51	19	0	0	0	0	0	0

Figura 2.16: Matriz DCT anterior depois da quantificação e desquantificação [4]

mecanismos. O primeiro é a codificação para sequência de valores zero. O segundo é o que JPEG chama "Codificação de Entropia", que envolve código de Huffman ou códigos aritméticos, ficando a cargo do implementador escolher.

O motivo da compressão JPEG ser tão eficiente é que um grande número de coeficientes são truncados para zero durante a quantização. Como muitos valores serão zero, o comitê JPEG decidiu tratar os valores zero de um modo diferente dos outros valores. Um simples código foi desenvolvido para contar o número de zeros consecutivos na imagem. Como quase metade dos coeficientes são quantizados para zero, este método fornece uma excelente compressão.

Um meio de aumentar a sequência de zeros é reordenar os coeficientes em uma "sequência zig-zag". O caminho que esta sequência percorre é mostrada na figura 2.17, mostrando também o modo como os coeficientes serão reordenados. Quando ocorre a expansão da imagem, os coeficientes são reordenados na forma anterior ao zig-zag.

2.4.4 Formato de imagem PGM

O formato Portable Gray Map (PGM) foi designado para ser facilmente importado entre plataformas. Ele faz parte de um conjunto de tipos de arquivos chamado Portable Anymap Format (PNM). Fazem parte deste conjunto também os formatos PBM (Portable Bit Map) e PPM (Portable Pixel Map). Cada um deles difere dos outros no número de cores a ser representado:

- O PBM é monocromático;
- O PGM é escala de cinza;
- O PPM é colorido em RGB.

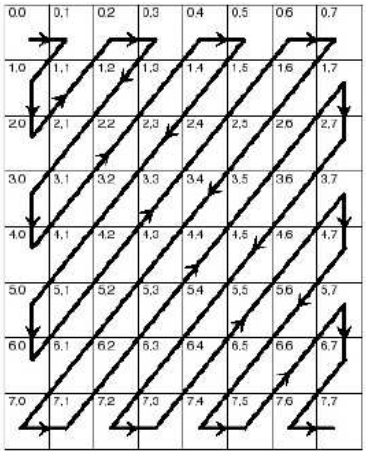

Figura 2.17: Sequência zig-zag [4]

Cada arquivo inicia com dois bytes de descrição (em ASCII) que especifica qual é o tipo de arquivo (PBM, PGM ou PPM) e sua codificação (ASCII ou binário). Os bytes de descrição são compostos pela letra "P" seguida de um único número. Deste modo as opções deste campo são:

- P1 para PBM com codificação ASCII
- P2 para PGM com codificação ASCII
- P3 para PPM com codificação ASCII
- P4 para PBM com codificação binária
- P5 para PGM com codificação binária
- P6 para PBM com codificação binária

Logo após a descrição do tipo de arquivo, existe um espaço para comentário, seguido do número de colunas e de linhas de pixels. Isto facilita muito a utilização em aplicações. O último item do cabeçalho no PGM é o máximo valor de brilho que a imagem pode usar, que varia de 0 (sem brilho) a 255 (com brilho máximo).

A codificação ASCII permite a verificação humana do código da imagem e facilita a adaptação para qualquer plataforma que consiga ler caracteres ASCII, enquanto a binária é mais eficiente por ocupar menos espaço e ser mais fácil de ser manipulada por programas devido a ausência de espaços entre os dados de pixels.

Quando se usa a codificação binária, o PBM usa 1 bit por pixel, o PGM usa 8 bits por pixel e o PPM 24 bits por pixel, 8 para cada cor.

Capítulo 3

Fluxo de projeto

Este capítulo apresentará o fluxo de projeto seguido no desenvolvimento de circuitos integrados analógicos e digitais. O fluxo utilizado no desenvolvimento deste trabalho também será detalhada.

3.1 Fluxos de projeto de circuitos integrados

A típico fluxo de projeto de circuitos integrados é mostrado na figura 3.1.

Um projeto tipicamente é iniciado com o objetivo com base nas especificações iniciais. Inicia-se a fase de concepção do projeto, levantando os objetivos e requisitos que devem ser obedecidos para atenderem as especificações. O sucesso nesta fase inicial deve ser verificada através de simulações comportamentais. Os blocos construtivos do sistema são identificados e descritos.

Para partes analógica e de rádio-freqüência (amplificadores, misturadores etc), segue-se o ramo esquerdo do fluxograma mostrado na figura 3.1. Especificando o layout de cada transistor e a interconexão entre eles, metodologia de projeto conhecida como customização completa (*full-custom*), este fluxo de projeto parte da especificação e segue com o projeto elétrico dos blocos e sua posterior implementação física.

Para partes digitais (processador, memórias etc), segue-se o ramo direito do fluxograma da figura 3.1. Usando uma abstração de projeto *standard-cell based*, este fluxo de projeto é iniciado com uma descrição do sistema em linguagem de descrição de hardware (HDL) e evolui para as fases finais através do uso de ferramentos de simulação e síntese lógica e física.

Todas as fases são executadas visando sempre a testabilidade do sistema. Na parte digital, modelam-se as falhas levantando vetores de teste necessários para caracterizar o sistema. Usa-se a técnica de *scan chain*, testando cada flip-flop do sistema de um modo mais ágil, e verificando os circuitos sequênciais. Para a parte analógica empregam-se técnicas para aumento da observabilidade dos circuitos.

As fases de layout são contempladas com técnicas de compatibilidade eletromágnetica de modo a isolar os sistemas de RF de sistemas analógicos de alta precisão (conversores analógico-digitais) e das seções digitais.

Um co-projeto é feito entre software de aplicação e hardware, onde os requerimentos de uma área impactam diretamente nas decisões de projeto tomadas na outra.

3.2 Inserção do trabalho no fluxo

Este projeto contempla todo o procedimento de projeto abordado, desde a etapa inicial do projeto, onde é realizada a concepção e especificação do sistema até as simulações e integração do sistema. De fato, uma das vantagens da metodologia em SystemC é justamente conseguir padronizar as etapas de desenvolvimento de sistemas em um mesmo ambiente. Unir não somente o projeto de hardware mas também o de software parece ser o caminho evolucionário de fluxo de projeto [1].

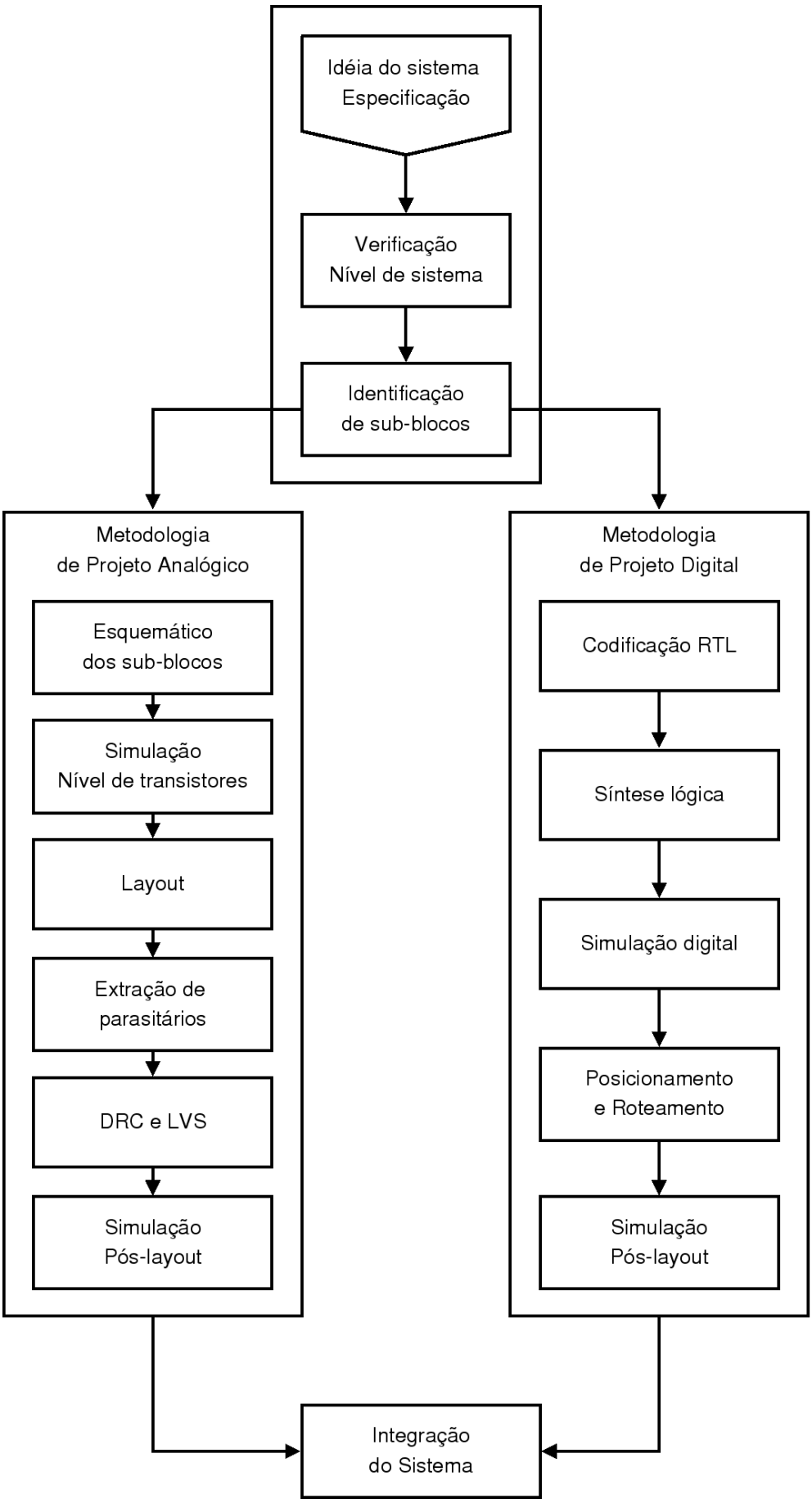

Figura 3.1: Fluxo de projeto de circuitos integrados.

Capítulo 4

Projeto

Este capítulo aborda a execução do projeto, com as relativas dificuldades e soluções de cada etapa. Este trabalho iniciou-se com o desenvolvimento do APS, passando pela sua integração à arquitetura RoSA, depois pelo desenvolvimento da aplicação JPEG e, finalmente, pela produção da aplicação JPEG que utiliza os recursos da arquitetura reconfigurável. A arquitetura reconfigurável utilizada neste projeto, RoSA, já havia sido implementada por outra pessoa, aluno de mestrado, e precisava apenas de alguns complementos, que foram feitos pelo autor, para entrar em pleno funcionamento.

4.1 APS

Sendo a primeira parte do projeto, a modelagem do Active Pixel Sensor (APS) foi precedida pelo estudo do SystemC e do TLM. A primeira dificuldade encontrada foi como modelar a entrada deste sensor. Depois de algumas pesquisas e discussões, foi visto que o melhor modo de implementá-la seria lendo arquivos de imagens. Com isso já fica bem claro o fluxo de dados no APS: um código que lesse cada pixel de uma imagem e os transferisse para algum componente que requisite estes dados. Possui um funcionamento similar ao de uma memória, respondendo pedidos de leitura, mas com dados das imagens.

A próxima etapa seria definir algum formato de imagem padrão para que os dados da imagem fossem lidos. Após pesquisa encontrou-se o formato PGM (Portable Grey Map) que possui uma estrutura simples e intuitiva de estruturar os dados e é destinada originalmente para envio de imagens via rede. Uma imagem PGM é constituída por pixels em graduação de cinza, com o valor 0 representando o preto e 255 o branco. É formada portanto por valores de cada pixel que variam de 0 a 255, formando uma matriz de bytes. Uma outra facilidade deste modelo é que em seu cabeçalho se imprimem os números de linhas e de colunas da imagem, possibilitando saber o tamanho da imagem. Baseado nestes informações foi criada uma rotina de leitura e escrita dos pixels nesse formato.

Neste ponto do projeto as dúvidas foram sanadas e o código do sensor foi escrito sem problemas. O código agora passa por adaptações à estrutura da arqitetura RoSA.

4.2 Integração do APS ao RoSA

Para a melhor compreensão da estrutura do sistema reconfigurável modelado em SystemC usando TLM, a figura 4.1. é apresentada.

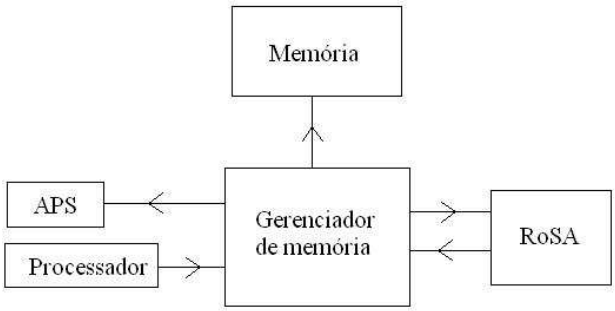

Figura 4.1: Estrutura da arquitetura reconfigurável

Originalmente o sistema, no qual o APS será inserido, era composto por 4 blocos. Este sistema é mapeado em memória, ou seja, cada bloco a ser acessado possui um endereço. O sistema possui endereçamento de 32 bits. As setas na figura 4.1 mostram o sentido de requisições de dados. Abaixo uma breve descrição dos componentes:

- Processador - É a fonte de requisições dos pedidos de dados, pois é dele que partem os pedidos de leitura e escrita em memória das aplicações. Portanto ele sempre começa a comunicação com o gerenciador de memória, que posteriormente responde com os dados desejados.
- Gerenciador de memória - Este componente faz o papel do barramento, recebendo todos os pedidos de acesso à memória e repassando o pedido ao componente mapeado no endereço requisitado. Após o envio da requisição de dados ao componente destinatário, espera sua resposta para repassar a quem requisitou a informação. Ocupa o papel principal da comunicação no sistema.
- Memória - Possui as funcionalidades de uma memória, recebendo requisições de leitura e escrita. Cada byte (8 bits) possui um endereço de 0x000000 até 0x400000. Pode receber pedidos de leitura e de escrita. Caso seja leitura, ele responde com uma mensagem de sucesso e o byte lido no endereço passado. Caso seja escrita a memória responde apenas com uma mensagem de sucesso de escrita. Se comunica sempre com o gerenciador de memória.
- RoSA - Está mapeado na faixa de endereços de 0x400000 a 0x4000FF, sendo que os registradores globais estão em 0x400000-0x40002F, os modificados em 0x400030-0x40005F, os de saída em 0x4000F9-0x004000FE e finalmente o gerenciador de configurações no endereço 0x4000FF. Este bloco pode receber e enviar pedidos de leitura em memória, pois o configurador de configurações ao ser ativado, busca as configurações em sua cache e se não as encontrar, as busca na memória, requisitando a leitura dos dados de configuração.
- APS - Foi mapeado no endereço 0x400100 e somente responde a pedidos de leitura, informando os dados da imagem.

Assim o primeiro desafio ao integrar o APS nesta estrutura foi projetar um modo simples e eficiente de passagem de dados. A melhor opção encontrada foi estipular um único endereço para o APS, de modo que quando o processador acesse este endereço o APS retorne 4 bytes (ou 4 pixels, pois cada pixel é um byte) lidos da imagem. Este número de 4 bytes por leitura foi escolhido devido à largura do espaço de dados na leitura em memória, já que cada endereço retorna um bloco de 32 bits, ou 4 bytes. Deste modo toda vez que seu endereço (0x400100) ele retorna ele retorna 4 bytes, funcionando como um componente serial no envio de dados.

O APS foi designado para ler um determinado nome de arquivo, que no código esta claramente exposto. Portanto ele lê o arquivo com o nome estipulado que estiver na mesma pasta que o executável da simulação da arquitetura RoSA. Além disso o APS foi projetado de uma forma que ao final da leitura dos últimos 4 bytes da imagem, ele novamente abre um arquivo de entrada com o nome estipulado no código, possibilitanto que este arquivo seja trocado antes que a imagem seja novamente lida.

O código completo da estrutura RoSA escrito pelo aluno de mestrado e complementado para entrar em uso encontra-se em anexo no material do CD. Já o código do APS integrado na estrutura está exposto anexo a este trabalho.

4.3 Aplicação JPEG

Primeiramente foi necessário um estudo do funcionamento e da implementação do JPEG na linguagem de programação C. A referência [4] foi usada nesta etapa, fornecendo não somente a teoria, mas um modelo pré-montado de compressão JPEG implementado em C. Deste modo o código precisou ser totalmente compreendido, adaptado e corrigido para começar a funcionar. A versão de compressão JPEG apresentada neste trabalho realiza as mesmas operações usadas em implementações comerciais, mas sem as muitas otimizações necessárias para que a compressão seja feita quase em tempo real.

Esta aplicação foi implementada para ser totalmente executada no processador hospedeiro, ou seja, sem nenhuma utilização do RoSA. Para a primeira etapa de testes da compressão JPEG usaram-se figuras em formato PGM de diversos números de pixels. Um parâmetro que sempre se pode variar nesta compressão é a qualidade da imagem comprimida através da variação da matriz de quantização da compressão.

Após testar seu funcionamento no processador hospedeiro, passou-se para sua implementação usando a arquitetura reconfigurável.

4.4 Aplicação JPEG mapeada no RoSA

Esta foi a etapa crítica do projeto, pois foi preciso corrigir e finalizar a estrutura do RoSA implementada em SystemC usando TLM 1.0. Alguns itens desta etapa incluíram:

- Correção dos endereços de memória dos registradores do RoSA e a forma como eram acessados

e escritos;

- Utilização do campo dados da configuração das células, que ainda não estava implementado;
- Implementação da rotina de leitura no RoSA, que no caso seriam leitura dos registradores de saída, que representam os únicos componentes capazes de produzir valores de saída na arquitetura reconfigurável.

Então com o sistema reconfigurável em condições de uso, o desafio passou a ser implementar a aplicação JPEG que utilizasse os recursos do RoSA. O primeiro item a ser avaliado foi ver em qual local do código o RoSA seria mais útil. Sabendo dos cálculos de multiplicação de matrizes, que geram um número considerável de operações para o processador, conclui-se que o melhor lugar seria passar a parte de multiplicação de matrizes 8x8 (como explicado na seção 2.4.1.1) da DCT para ser executada no RoSA. Para adquirir maior portabilidade, o cálculo de multiplicação de matrizes no RoSA foi implementada em uma função específica, sendo invocada no devido momento do algoritmo JPEG.

Devido a limitações da arquitetura, várias barreiras precisaram ser levadas em conta, que são aqui relacionadas:

- Escasso número de registradores de entrada de dados: 48 globais e 48 modificados;
- Limitação de quatro operações no máximo a serem feitas dentro de uma mesma célula; neste caso necessitando de 8 operandos (2 por operação) por célula por ciclo.
- Presença de apenas seis células, significando que se podem efetuar apenas seis configurações por ciclo.
- O cache da arquitetura RoSA armazena no máximo oito configurações de 720 bits (6 células)

Levando em consideração estas especificações, a solução proposta e implementada passa a ser exposta e analisada.

Um desafio foi pensar como seria montar estas configurações de forma clara e exequível, pois para cada 8 operandos serão necessários 120 bits de configuração. Deste modo, se fosse preciso digitar todas as configurações desta operação, sem qualquer forma de automação, seriam necessários formular e digitar um grande número de configurações. Isto se torna inviável por três motivos, primeiro pelo trabalho exagerado que seria demandado para serem feitas estas configurações, segundo pela grande chance de erro comprometer todo o cálculo e terceiro pela dificuldade de compreensão do código por terceiros.

Deste modo foi proposto escreverem-se apenas duas configurações, de modo que sejam sempre efetuadas operações entre registradores globais e modificados com o mesmo número, por exemplo o registrador global 4 com o modificado 4. Desta forma para usarem-se os 96 registradores foram necessários 2 configurações, pois cada uma opera seis células, que operam oito operandos por vez, resultando em 48 operandos por configuração.

Possuindo apenas estas duas configurações obtemos duas vantagens:

- A responsabilidade por criar a lógica das operações passa para a rotina de escrita dos registradores, pois será preciso somente salvar os valores nos registradores corretos;
- A velocidade das operações será aumentada devido a existir apenas duas configurações que estarão sempre armazenadas em cache. Será necessário carrega-las da memória apenas uma única vez, na primeira operação.

A figura 4.2 mostra, resumidamente, o cálculo do primeiro valor da matriz final 6x6.

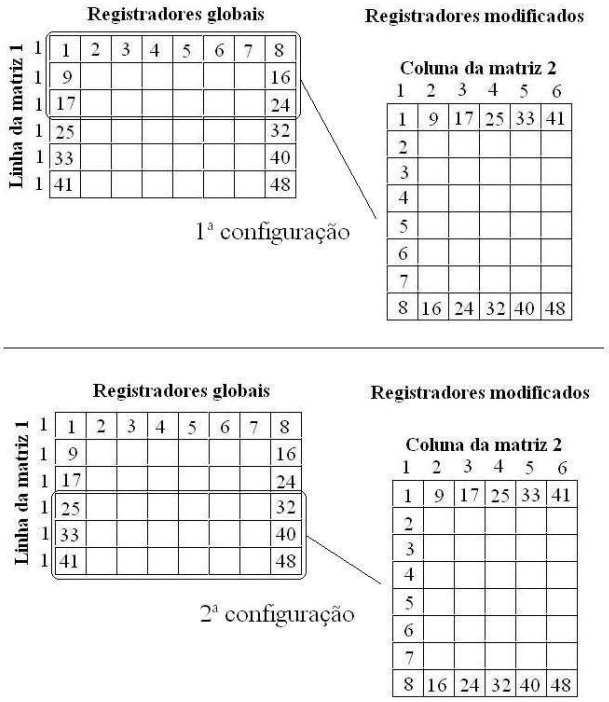

Figura 4.2: Sequência de cálculos das duas configurações do sistema

Outra dificuldade foi o escasso número de registradores para o número total de operandos envolvidos na multiplicação de matrizes. Em duas matrizes 8x8 o número de elementos seriam 128 (64+64) e o RoSA dispõe de somente 48 registradores globais e 48 modificados, em um total de 96 registradores disponíveis para a execução de operações. Portanto a reciclagem de valores nos registradores para a execução da multiplicação das matrizes é inevitável. Uma solução foi projetada de forma a tentar reduzir ao máximo a necessidade de se escreverem novos valores nos registradores.

A solução consistiu em dividir a matriz 8x8 resultante em quatro quadrantes, de tamanhos variados. Desta forma a matriz final foi dividida em 4 partes: uma matriz 6x6, outra 6x2, outra 2x6 e finalmente uma 2x2. Esta divisão é mostrada na figura 4.3.

O cálculo das matrizes é bastante similar e feito em ciclos de duas configurações, de forma que compreendendo o funcionamento de uma delas, compreende-se facilmente o cálculo das outras. Por isso abordaremos como calcular a maior, a matriz 6x6.

O primeiro passo é salvar a primeira linha da primeira matriz de entrada (chamada de matriz 1) seis vezes. Como temos 6 linhas de 8 componentes, precisa-se de 48 registradores. Escolheu-se

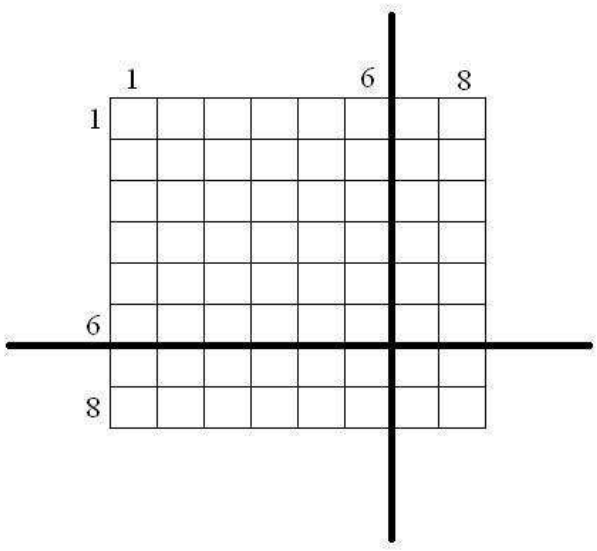

Figura 4.3: Solução ao problema dividindo a matriz em quatro partes

salvar estes valores da matriz 1 nos registradores globais, que com isso se esgotam. O segundo passo é salvar as seis primeiras colunas da segunda matriz de entrada (chamada de matriz 2). Deste modo, com as duas configurações das células expostas acima, a primeira linha da matriz 6x6 pode ser calculada no primeiro ciclo, multiplicando-se sempre a primeira linha da matriz 1 pelas diferentes colunas da matriz 2. Este procedimento é ilustrado na figura 4.4.

Registradores globais

Linha da matriz 1								
1	1	2	3	4	5	6	7	8
1	9							16
1	17							24
1	25							32
1	33							40
1	41							48

Registradores modificados

Coluna da matriz 2

	1	2	3	4	5	6
1	9	17	25	33	41	
2						
3						
4						
5						
6						
7						
8	16	24	32	40	48	

Figura 4.4: Conteúdo dos registradores para o cálculo da primeira linha da matriz 6x6 final

Quando terminada a primeira linha da matriz 6x6, o algoritmo prossegue com o segundo ciclo do cálculo repetindo seis vezes os elementos da segunda linha da matriz 1 nos registradores globais e mantendo os valores já existentes nos registradores modificados (pois já possuem os valores das seis primeiras colunas da matriz 2). Desta forma calcula-se a segunda linha da matriz final. Este procedimento prossegue até que todas as linhas da coluna 6x6 sejam calculados. Os ciclos de cálculo são ilustrados na figura 4.5.

Um outro problema é que cada célula consegue fazer apenas metade das operações necessárias para chegar a um elemento da matriz final. A entrada da célula só comporta oito elementos (quatro operações) e para o cálculo de cada elemento são necessários 16 elementos (oito operações). Após a multiplicação, os valores são somados dois a dois dentro da célula. Ao final ainda ficariam

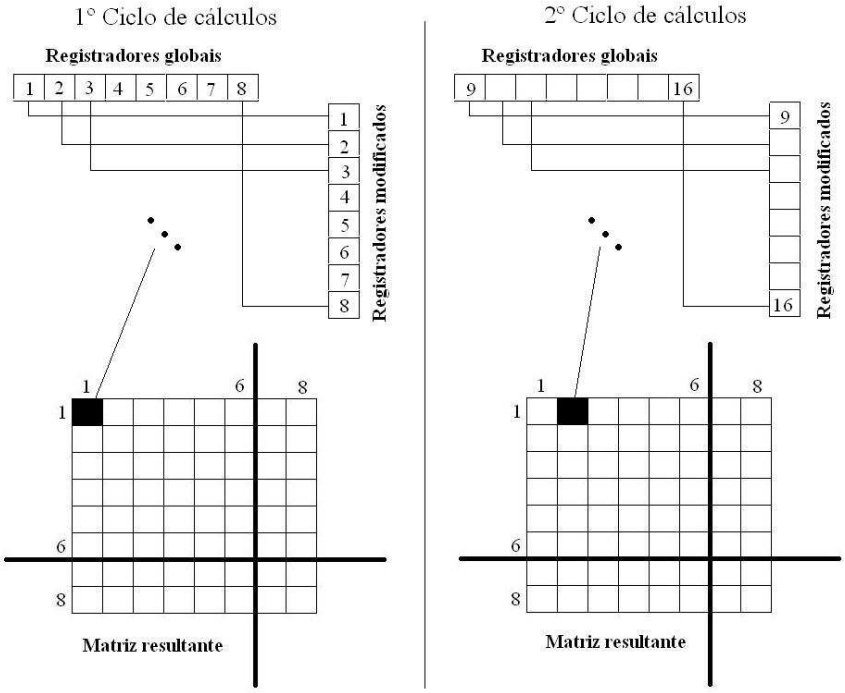

Figura 4.5: Ciclos de cálculos dos elementos da matriz final 6x6

faltando uma última soma entre os valores de duas céulas. Este cálculo foi deixado a cargo do processador hospedeiro, pois não justificava novamente reescreverem-se os registradores e uma nova configuração apenas para somar estes poucos valores. A sequência de operações e a divisão de operações que ocorrem no RoSA e no processador são mostradas na figura 4.6.

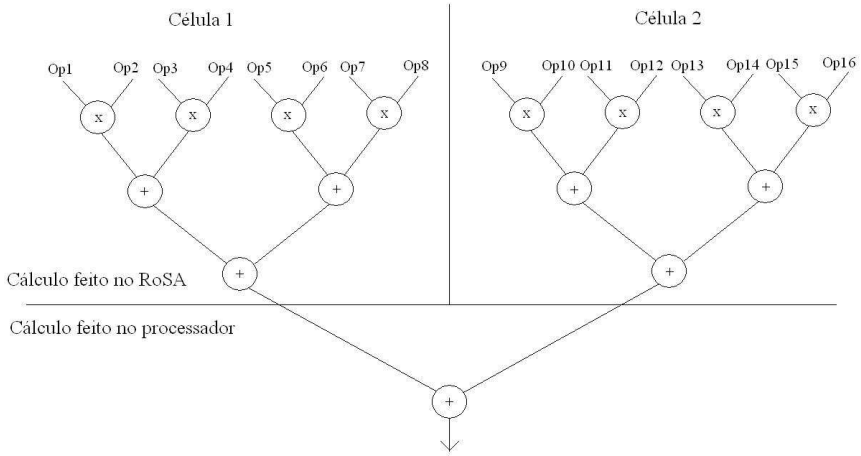

Figura 4.6: Árvore de operações no cálculo dos valores finais da matriz resultado

A parte de lógica dos cálculos encerra-se aqui, restando ainda obscuro somente o procedimento de comando do RoSA para o início dos cálculos qua agora é detalhado.

O comando é relativamente simples, possuindo como fundamento o recurso de ponteiros na linguagem de programação C. Para escrever ou ler qualquer registrador do sistema foi necessário apenas atribuir o endereço físico do resgistrador a algum ponteiro, e prosseguir com a sua chamada toda vez que for necessário salvar ou ler um valor de 32 bits.

Já para comandar o gerenciador de configurações na execução das configurações, o procedimento foi um pouco mais elaborado, já que é preciso primeiro escrever a configuração de 720 bits e depois criar um ponteiro que aponta para o endereço do gerenciador (0x4000FF) e passar como valor para este ponteiro o endereço em que foi escrita a configuração das células na memória. Ao fazer isso o gerenciador automaticamente dá início ao ciclo de cálculo, repassando as configurações para as células devidas, que iniciam o cálculo e disponibilizam os resultados em seus registradores de saída. Lembrando que a primeira configuração de 720 bits executada utiliza os 24 primeiros registradores de cada tipo (global e modificado) e a segunda opera os 24 registradores restantes de cada tipo.

A última dificuldade encontrada na implementação da multiplicação de matrizes da DCT foi que o RoSA não trabalha com tipos de variáveis float ou double, sendo que a aplicação JPEG projetada que roda inteiramente no processador utiliza o tipo double para armazenar os valores da matriz da transformada do DCT. Sabendo disso, a solução encontrada foi multiplicar os valores da matriz da transformada DCT por 100 e, como também se usa a sua transposta, teremos duas matrizes sendo multiplicadas, ao final divide-se o valor final das multiplicações por 10.000 (100*100). Esta foi a melhor forma encontrada para contornar esta limitação.

Capítulo 5

Análise e Resultados

Este experimento procurou implementar novos recursos à plataforma reconfigurável desenvolvida em SystemC usando TLM, validando-os através da aplicação JPEG. Deste modo pode-se esboçar uma avaliação da arquitetura que utiliza o RoSA fazendo uma comparação com os resultados obtidos com a arquitetura clássica, que utiliza apenas o processador e a memória.

5.1 APS

Durante a implementação foram utilizados vários testes para verificar a fidelidade da leitura das imagens. O formato PGM facilitou esta verificação pois, como visto na seção 2.4.4, a imagem pode ser escrita em binário ou em caracteres ASCII. Uma aplicação com o objetivo de ler os dados do APS foi escrita para possibilitar esta verificação. Com isso pode-se conferir os dados informados pelo APS com os dados no arquivo PGM original. Tirando o cabeçalho do arquivo PGM, a informação das duas fontes se mostraram idênticas, comprovando o sucesso na implementação.

Um dos possíveis defeitos do APS implementado é a necessidade de se informar para o código que utiliza a leitura do APS o número de vezes que devem ser lidos os dados até que a imagem chegue ao fim. Isto é feito no início do código da aplicação informando o número de linhas e de colunas de pixels que a imagem possui. Outro defeito seria que a imagem a ser lida precisa possuir um número de pixels múltiplo de 4, pois o APS funciona lendo blocos de 4 em 4 bytes (pixels). Caso isto não seja obedecido o final do arquivo de imagem pode chegar ao fim e o APS tentará continuar a leitura, causando um erro no sistema.

5.2 Aplicação JPEG

Esta aplicação foi projetada para ser executada no processador hospedeiro da arquitetura reconfigurável, que no caso foi um modelo do processador MIPS. Dois parâmetros podem ser variados para validar a aplicação: o arquivo de entrada e o fator de quantização, que é explicado na seção 2.4.2.1.

Sabe-se que quanto mais complexa a imagem em termos de variações de tons de cinza, menor

será a compressão, pois esta variação ocasiona uma grande quantidade de coeficientes diferentes de zero depois da DCT. Com isso usaram-se duas imagens de teste para serem comprimidas, uma simples de 32 x 32 pixels denominada blackV.pgm e uma outra, a ilidan.pgm, que é mais complexa, de 160 x 160 pixels, todas no formato PGM (e consequentemente em preto e branco).

As taxas de compressão são mostradas na tabela 5.1 para aplicações da compressão JPEG nas duas imagens, utilizando-se o fator de quantização como 1, 5 e 10 para comparação. Os dados de compressão são fornecidos diretamente pelo código executado, devido a inserção de uma rotina de medição do tamanho dos arquivos finais e iniciais.

Arquivo	Fator de quantização	Tamanho inicial (bytes)	Tamanho final (bytes)	Redução
ilidan.pgm	1	25600	7727	70%
ilidan.pgm	5	25600	2857	89%
ilidan.pgm	10	25600	1876	93%
blackV.pgm	1	1024	144	86%
blackV.pgm	5	1024	88	92%
blackV.pgm	10	1024	70	94%

Tabela 5.1: Comparação entre taxas de compressão

Analisando os dados, percebe-se que as informações comprovadas já eram esperadas: dependendo do fator de quantização a taxa de compressão varia. Aumentando o fator de quantização a qualidade diminui, mas se conseguem altíssimas taxas de compressão. Já escolhendo um fator baixo a qualidade aumenta e a taxa de compressão diminui, sendo que usando o valor 1 para este fator obteremos a melhor qualidade, mas ainda com uma alta taxa de compressão, 70%, na imagem de 160 x 160 pixels e de 86% na imagem de 32 x 32 pixels.

Uma última observação nesta aplicação se refere a eficácia do fator de quantização depender da complexidade da figura, definindo como complexa uma imagem com muitas variações de tonalidades em pixels adjacentes. Quanto mais complexa a imagem menor a taxa de compressão utilizando-se o fator de quantização 1 porque muitos coeficientes da matriz de dados após a DCT não serão zerados. Mas em compensação apresenta um aumento da taxa de compressão maior quando se varia o fator de quantização. a variação desta taxa usando os valores de compressão 1 e 10 foi de 23%.

O inverso se comprova na imagem menos complexa, de 32 x 32 pixels, apresentando a taxa de compressão inicial, com fator de quantização unitário bem elevada, com 86%. Mas já o crescimento da taxa de compressão com a variação do fator de quantização não acompanhou o da figura complexa, sendo de apenas 8%.

5.3 Aplicação JPEG mapeada no RoSA

Para testar esta aplicação utilizam-se as mesmas duas imagens da seção anterior, uma simples de 32 x 32 pixels e outra de 160 x 160 pixels. Para facilitar a análise, foram feitas comparações entre a aplicação JPEG executada na arquitetura reconfigurável com e sem o uso do RoSA. Os dados para análise entre as duas aplicações foram número de instruções executadas pelo processador e tempo de simulação dos códigos.

A tabela 5.2 expõe estes dados de desempenho da aplicação fazendo uso do RoSA para calcular a etapa de multiplicação de matrizes da DCT. Da mesma forma, a tabela 5.3 mostra os dados para o processamento sem a utilização do RoSA. Utilizou-se 1 e 10 como fatores de quantização em cada figura. Os dados de número de instruções executadas foram obtidos com a exploração de recursos do ArchC, que é uma linguagem de descrição de processadores com a qual o processador MIPS utilizado neste trabalho foi escrito. Já os dados de tempo de simulação foram fornecidos pelo comando "time" do linux.

Imagem	Fator de quantização	Tempo de simulação (s)	Número de instruções executadas
ilidan	1	15,949	68.830.995
ilidan	10	13,841	58.293.255
blackV	1	0,668	2.395.230
blackV	10	0,639	2.276.475

Tabela 5.2: Tempos de simulação e número de instruções executadas pelo processador utilizando o RoSA

Imagem	Fator de quantização	Tempo de simulação (s)	Número de instruções executadas
ilidan	1	98,715	470.409.602
ilidan	10	96,774	460.874.899
blackV	1	3,812	17.147.971
blackV	10	3,677	17.035.653

Tabela 5.3: Tempos de simulação e número de instruções executadas pelo processador não utilizando o RoSA

Observa-se uma grande redução de tempo de processamento e de instruções executadas pelo processador hospedeiro, comprovando a implementação e a eficácia do uso da arquitetura reconfigurável para a compressão JPEG.

Capítulo 6

Conclusões

Depois de feito um grande estudo sobre os diversos temas abordados por este trabalho, a implementação do APS em SystemC foi concluída e integrada à arquitetura RoSA. O sucesso desta etapa foi verificado através de simulações de leitura de imagens.

Do mesmo modo, uma aplicação JPEG para compressão de imagens foi implementada e posteriormente adaptada para utilizar os recursos da plataforma reconfigurável. Esta etapa foi bem-sucedida e apresentou dados que não somente validam o funcionamento da arquitetura reconfigurável, como também comprovam seu melhor desempenho nesta aplicação.

Possuindo em mente o objetivo deste trabalho, observa-se que este foi alcançado, obtendo vários resultados bem-sucedidos de simulações na plataforma reconfigurável que validam a implementação do sistema em SystemC. No caso específico da aplicação escolhida, a compressão JPEG, percebeu-se um alto ganho na performance do sistema utilizando-se o RoSA. Caso seja necessário aumentar esta performance, para algum desenvolvimento futuro, pode-se sugerir várias otimizações, como:

- Aumentar o número de registradores, de modo que sejam diminuídas o número de vezes que o processador precisa reescrevê-los;
- Aumentar o número de células, para que um maior número de elementos da matriz de saída sejam calculadas de cada vez;

Como trabalhos futuros podem-se sugerir:

- Aperfeiçoar do código da arquitetura reconfigurável em SystemC, com inclusão de rotinas de erro espalhados pelo código e melhor implementação dos registradores locais;
- Melhorar a maneira como o APS faz leitura de imagens;

REFERÊNCIAS BIBLIOGRÁFICAS

[1] BLACK, D.; DONOVAN, J. *SystemC: From the Ground Up*. [S.l.]: Springer, 2004.

[2] PANDA, P. Systemc: a modeling platform supporting multiple design abstractions. v. 30, p. 75–80, 2001.

[3] PEREIRA, M. M. Proposta e implementaÇÃo de uma arquitetura reconfigurÁvel hÍbrida para aplicaÇÕes baseadas em fluxo de dados. 2008.

[4] NELSON, M. *The data compression book*. [S.l.]: Henry Holt and Co., Inc. New York, NY, USA, 1991.

[5] GRÖTKER, T. *System Design with SystemC*. [S.l.]: Kluwer Academic Publishers, 2002.

[6] GAJSKI, D. *Specc: Specification Language and Methodology*. [S.l.]: Kluwer Academic Publishers, 2000.

[7] CAI, L.; GAJSKI, D. Transaction level modeling in system level design. *Center for Embedded Computer Systems*, 2003.

[8] BONDALAPATI, K.; PRASANNA, V. Reconfigurable computing systems. *Proceedings of the IEEE*, v. 90, n. 7, p. 1201–1217, 2002.

[9] PEREIRA, M.; OLIVEIRA, B. de; SILVA, I. RoSA: a reconfigurable stream-based architecture. In: ACM NEW YORK, NY, USA. *Proceedings of the 20th annual conference on Integrated circuits and systems design*. [S.l.], 2007. p. 159–164.

[10] BESERRA, G. S.; MARIANO, G. A.; GUIMARãES, H. H. CaracterÍsticas tÉcnicas desejadas para o hardware de um sensor de imagens aps para soc. 2008.

ANEXOS

I. CÓDIGOS

Códigos fonte utilizados no decorrer do projeto. O código da arquitetura reconfigurável está presente no CD que vêm junto com esta apresentação.

I.1 Código do APS inserido na arquitetura RoSA

```
/*
 * tlmMemory.h
 *
 *  Created on: Oct 20, 2008
 *      Author: root
 */

#ifndef APS_TLM1_H_
#define APS_TLM1_H_


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include"readppm.h"
#include "../comunication_structure/comunicationInterfaces.h"


namespace plataforma_unb{

template <class T>
class tlm_aps : public sc_module, public read_write_port_if<T>{

private:
        int cont;

public:
        int nlines, ncols, ncolors, Size_scan, end_addr;

        unsigned char *image_scanned, *image_pointer;

        sc_export<read_write_port_if<T> > apsExport;

        //leitura da imagem
        virtual ac_tlm_rsp memRead(const T &lectura){
                ac_tlm_rsp response;


                if(cont == 0){
                        ncolors = readppm(&nlines, &ncols, &image_scanned);
                        image_pointer = image_scanned;
```

```
                Size_scan = ncols*nlines;

                //endereÃğo de parada e o tamanho da imagem
                //como fazemos em blocos de 4 bytes, sera o tamanho
                    dividido por 4, arrendondando para cima.
                end_addr = Size_scan;
        };
        // Neste ponto nÃčo se aceita outro formato a nÃčo ser PGM
        if(ncolors == 1){
                if(cont < end_addr){
                        ((uint8_t*)&(lectura.data))[3]=image_pointer[cont
                            ++];
                }else{
                        ((uint8_t*)&(lectura.data))[3]=(uint8_t)0;
                };

                if(cont < end_addr){

                        ((uint8_t*)&(lectura.data))[2]=image_pointer[cont
                            ++];
                }else{

                        ((uint8_t*)&(lectura.data))[2]=(uint8_t)0;
                };

                if(cont < end_addr){

                        ((uint8_t*)&(lectura.data))[1]=image_pointer[cont
                            ++];
                }else{

                        ((uint8_t*)&(lectura.data))[1]=(uint8_t)0;
                };

                if(cont < end_addr){

                        ((uint8_t*)&(lectura.data))[0]=image_pointer[cont
                            ++];
                }else{

                        ((uint8_t*)&(lectura.data))[0]=(uint8_t)0;

                        cont=0;
                };

                response.data=lectura.data;
                response.status=SUCCESS;
        }else{
                printf("Unable to open file");
                response.status = ERROR;
        };
        return response;
};
```

```
90
        virtual ac_tlm_rsp memWrite(const T &escritura){
                        ac_tlm_rsp response;
                        response.status=SUCCESS;
                        return response;
95      }

        tlm_aps(sc_module_name name):sc_module(name), apsExport("CM_export_to_aps")
            {
                apsExport.bind(*this);
                ncolors=1;
100             end_addr=0;
                cont = 0;                       // serve para indicar se a imagem
                    ja foi lida completamente
        }
};

105 };

#endif /* APS_TLM1_H_ */
```

I.2 Código da função de leitura do formato PGM que o APS utiliza

```
#ifndef READPPM_H
#define READPPM_H

5  #include <stdio.h>
#include <stdlib.h>
#include <string.h>

int readppm(int *nlines, int *ncols, unsigned char **buf/*, char *filename*/)
10 {
        char    c, s[1000];
        FILE *f;
        int     depth, k, numread = 0, channels = 0;

15              if ((f = fopen("example.pgm", "rb")) == NULL)
                        return -2;

        //checa se Ã‘ PGM
        if (getc(f)!='P') return -1;
20      c = getc(f);
        if ( (c != '5') && (c != '6') ) return -1;
        if (c == '5') channels = 1; else channels = 3;
        while ( (c=getc(f)) != '\n') ;

25      // rotina que pula as linhas de comentario
        for (;;)
        {
                /* get a line */
```

```
                k = 0;
                while ( (c=getc(f)) != '\n')
                {
                        s[k++] = c;
                        if (k > 999) return -3;
                }
                s[k] = '\0';

                if (s[0] != '#')
                {
                        if (numread == 2)
                        {
                                numread += sscanf(s, "%d", &depth);
                                if (numread < 2) return -4;
                        }
                        if (numread == 1)
                        {
                                numread += sscanf(s, "%d %d", nlines, &depth);
                                if (numread < 1) return -4;
                        }
                        /* o programa comeÃğa por aqui, com numread = 0  e le ncols
                             e nlines, com sscanf retornando 2*/
                        /* se o brilho maximo estiver na mesma linha do numero de
                            colunas e linhas, nao havera problema pois sscanf
                            retorna 3*/
                        if (numread == 0)
                        {
                                numread += sscanf(s, "%d %d %d", ncols, nlines, &
                                    depth);
                                if (numread == EOF) return -4;
                        }
                        if (numread == 3) break;
                }
        }
        *buf = (unsigned char *) calloc( (*ncols) *  (*nlines) * channels, sizeof(
            unsigned char));
        fread(*buf,  channels * sizeof(unsigned char), (*ncols) * (*nlines), f);
        return channels;
}

#endif /* READPPM_H */
```

I.3 Código da rotina principal (main) que faz a compressão JPEG no processador

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bitio.h"
#include "errhand.h"
```

```
#include "main.h"
#ifdef ___STDC___
void usage_exit( char *prog_name );
void print_ratios( char *input, char *output );
long file_size( char *name );
#else
void usage_exit();
void print_ratios();
long file_size();
#endif

#define NCOLS 32
#define NLINS 32

/*************************************************************************/
/*
 *    Deve-se alterar o valor das linhas e colunas da imagem a ser lida para que
      funcione corretamente
 */

int main( argc, argv )
int argc;
char *argv[];
{
      BIT_FILE *output;
      FILE *input;
      int i, cont, *aps;
      setbuf( stdout, NULL );
      if ( argc < 3 )
            usage_exit( argv[ 0 ] );

      aps = (int*)0x00400100;
      i = 0;
      input = fopen(argv [ 1 ], "wb" );
      for(cont = 0; cont<((NCOLS*NLINS)/4); cont++){
                i=*aps;
                fwrite (&i , 4 , 1 , input );
                printf("Dados %d: %.8X \n", cont, i);
      };
      fclose( input );

      input = fopen(argv [ 1 ], "rb" );
      if ( input == NULL )
            fatal_error( "Error opening %s for input/n", argv[ 1 ] );
      output = OpenOutputBitFile( argv[ 2 ] );
      if ( output == NULL ){
            fatal_error( "Error opening %s for output/n", argv[ 2 ] );
      };
      printf( "\nCompressing %s to %s\n", argv[ 1 ], argv[ 2 ] );
      printf( "Using %s\n", CompressionName );
      argc -= 3;
      argv += 3;
      CompressFile( input, output, argc, argv );
```

```
        CloseOutputBitFile( output );
        fclose( input );
        argv -= 3;
        print_ratios( argv[ 1 ], argv[ 2 ] );

        return( 0 );
}

/*
* Esta rotina apenas mostra o modo de uso do programa, caso nÃčo sejam passados os
    parametros corretamente
*/
void usage_exit( prog_name )
char *prog_name;
{
      char *short_name;
      char *extension;
      short_name = strrchr( prog_name, '\\' );
      if (short_name == NULL )
            short_name = strrchr( prog_name, '/' );
      if (short_name == NULL )
            short_name = strrchr( prog_name, ':' );
      if (short_name != NULL )
            short_name++;
      else
            short_name = prog_name;
      extension = strrchr( short_name, '.' );
      if ( extension != NULL )
            *extension = '\0';
      printf( "\nUsage: %s %s\n", short_name, Usage );
      exit( 0 );
}
/*
* Esta funÃğÃčo faz a mediÃğÃčo de tamanho de arquivos
*/
#ifndef SEEK_END
#define SEEK_END 2
#endif
long file_size( name )
char *name;
{
      long eof_ftell;
      FILE *file;
      file = fopen( name, "rb");
      if ( file == NULL )
            return( 0 );
      fseek( file, 0, SEEK_END );
      eof_ftell = ftell( file );
      fclose( file );
      return( eof_ftell );
}
/*
* Esta rotina calcula e imprime a taxa de compressÃčo alcanÃğada
```

```
110 */
    void print_ratios( input, output )
    char *input;
    char *output;
    {
115     long input_size;
        long output_size;
        int ratio;
        input_size = file_size( input );
        if ( input_size == 0 )
120         input_size = 1;
        output_size = file_size ( output );
        ratio = 100 - (int) ( output_size * 100L / input_size );
        printf( "\nInput bytes:        %ld\n", input_size );
        printf( "Output bytes:      %ld\n", output_size );
125     if ( output_size == 0 ){
            output_size = 1;
        };
        printf( "Compression ratio: %d \n", ratio );
    };
```

I.4 Código da função de compressão JPEG

```
    /*
    * Este mÃşdulo implementa a compressÃčo JPEG de fato. Ela precisa ser ligada a
        mÃşdulos de suporte.
    *
    */
 5  #include <stdio.h>
    #include <stdlib.h>
    #include "bitio.h"
    #include "errhand.h"
    /*
10  * Para que a aplicaÃğÃčo funcione corretamente Ãľ necessÃąrio que o nÃžmero de
        linhas e de colunas da imagem sejam informados
    */
    #define ROWS 32
    #define COLS 32
    #define N 8
15  /*
    * Esta macro arredonda termos
    */
    #define ROUND( a )    ( ( (a) < 0 ) ? (int) ( (a) - 0.5 ) : \
                            (int) ( (a) + 0.5 ) )
20  char *CompressionName = "DCT compression";
    char *Usage      = "infile outfile [quality]\nQuality from 0-25";
    /*
    * ProtÃştipo de funÃğÃţes para funcionar em vÃąrias plataformas
    */
25  #ifdef __STDC__
    void Initialize( int quality );
```

```
void ReadPixelStrip( FILE *input, unsigned char strip[ N ][ COLS ] );
int InputCode( BIT_FILE *input );
void ReadDCTData( BIT_FILE *input, int input_data[ N ][ N ] );
void OutputCode( BIT_FILE *output_file, int code );
void WriteDCTData( BIT_FILE *output_file, int output_data[ N ][ N ] );
void WritePixelStrip( FILE *output, unsigned char strip[ N ][ COLS ] );
void ForwardDCT( unsigned char *input[ N ], int output[ N ][ N ] );
void InverseDCT( int input[ N ][ N ], unsigned char *output[ N ] );
void CompressFile( FILE *input, BIT_FILE *output,
                   int argc, char *argv[] );
void ExpandFile( BIT_FILE *input, FILE *output, int argc, char *argv[] );
#else
void Initialize();
void ReadPixelStrip();
int InputCode();
void ReadDCTData();
void OutputCode();
void WriteDCTData();
void WritePixelStrip();
void ForwardDCT();
void InverseDCT();
void CompressFile();
void ExpandFile();
#endif
/*
* Dados globais, com a tabela da DCT sendo especificada
*/
unsigned char PixelStrip[ N ][ COLS ];
double C[N][N]={ {0.353553 ,0.353553 ,0.353553 ,0.353553 ,0.353553 ,0.353553
    ,0.353553 ,0.353553}
                                ,{0.490393 ,0.415735 ,0.277785 ,0.097545
                                    ,-0.097545 ,-0.277785 ,-0.415735 ,-0.490393}
                                ,{0.461940 ,0.191342 ,-0.191342 ,-0.461940
                                    ,-0.461940 ,-0.191342 ,0.191342 ,0.461940}
                                ,{0.415735 ,-0.097545 ,-0.490393 ,-0.277785
                                    ,0.277785 ,0.490393 ,0.097545 ,-0.415735}
                                ,{0.353553 ,-0.353553 ,-0.353553 ,0.353553
                                    ,0.353553 ,-0.353553 ,-0.353553 ,0.353553}
                                ,{0.277785 ,-0.490393 ,0.097545 ,0.415735
                                    ,-0.415735 ,-0.097545 ,0.490393 ,-0.277785}
                                ,{0.191342 ,-0.461940 ,0.461940 ,-0.191342
                                    ,-0.191342 ,0.461940 ,-0.461940 ,0.191342}
                                ,{0.097545 ,-0.277785 ,0.415735 ,-0.490393
                                    ,0.490393 ,-0.415735 ,0.277785 ,-0.097545}};
double Ct[ N ][ N ];
int InputRunLength;
int OutputRunLength;
int Quantum[ N ][ N ];
struct zigzag {
  int row;
  int col;
} ZigZag[ N * N  ] =
{
```

```
    {0, 0},
    {0, 1}, {1, 0},
    {2, 0}, {1, 1}, {0, 2},
75  {0, 3}, {1, 2}, {2, 1}, {3, 0},
    {4, 0}, {3, 1}, {2, 2}, {1, 3}, {0, 4},
    {0, 5}, {1, 4}, {2, 3}, {3, 2}, {4, 1}, {5, 0},
    {6, 0}, {5, 1}, {4, 2}, {3, 3}, {2, 4}, {1, 5}, {0, 6},
    {0, 7}, {1, 6}, {2, 5}, {3, 4}, {4, 3}, {5, 2}, {6, 1}, {7, 0},
80  {7, 1}, {6, 2}, {5, 3}, {4, 4}, {3, 5}, {2, 6}, {1, 7},
    {2, 7}, {3, 6}, {4, 5}, {5, 4}, {6, 3}, {7, 2},
    {7, 3}, {6, 4}, {5, 5}, {4, 6}, {3, 7},
    {4, 7}, {5, 6}, {6, 5}, {7, 4},
    {7, 5}, {6, 6}, {5, 7},
85  {6, 7}, {7, 6},
    {7, 7}
    };
    /*
    * Esta rotina inicia os valores necessÃąrios para a compressÃčo
90  */
    void Initialize( quality )
    int quality;
    {
      int i;
95    int j;
      for ( i = 0 ; i < N ; i++ )
        for ( j = 0 ; j < N ; j++ )
          Quantum[ i ][ j ] = 1 + ( ( 1 + i + j ) * quality );
      OutputRunLength = 0;
100   InputRunLength = 0;

      for ( i = 1 ; i < N ; i++ ) {
        for ( j = 0 ; j < N ; j++ ) {
          Ct[ j ][ i ] = C[ i ][ j ];
105     }
      }
    }

    /*****************************************************************************
       */
110

    void ReadPixelStrip( input, strip )
    FILE *input;
    unsigned char strip[ N ][ COLS ];
    {
115

      int row;
      int col;
      unsigned char c;

120   for ( row = 0 ; row < N ; row++ )
          for ( col = 0 ; col < COLS ; col++ ) {
            c = (unsigned char) getc( input );
            if ( feof(input) )
```

```
            fatal_error( "Error reading input grey scale file/n" );
125         strip[ row ][ col ] = c;

        }

    }
130 /*
    * Para o entendimento desta rotina sugre-se a leitura do trabalho e da literatura
        recomendada
    *
    */
    int InputCode( input_file )
135 BIT_FILE *input_file;
    {
      int bit_count;
      int result;
      if ( InputRunLength > 0 ) {
140     InputRunLength--;
        return( 0 );
      }
      bit_count = (int) InputBits( input_file, 2 );
      if ( bit_count == 0 ) {
145     InputRunLength = (int) InputBits( input_file, 4);
        return( 0 );
      }
      if ( bit_count == 1 )
        bit_count = (int) InputBits( input_file, 1 ) + 1;
150   else
        bit_count = (int) InputBits( input_file, 2 ) +
                    ( bit_count << 2 ) - 5;
      result = (int) InputBits( input_file, bit_count );
      if ( result & ( 1 << ( bit_count - 1 ) ) )
155     return( result );
      return( result - ( 1 << bit_count ) + 1 );
    }

    /*************************************************************************
        */
160

    void ReadDCTData( input_file, input_data )
    BIT_FILE *input_file;
    int input_data[ N ][ N ];
    {
165   int i;
      int row;
      int col;
      for ( i = 0 ; i < ( N * N ) ; i++ ) {
        row = ZigZag[ i ].row;
170     col = ZigZag[ i ].col;
        input_data[ row ][ col ] = InputCode( input_file ) *
                        Quantum[ row ][ col ];
      }
    }
```

```
175
/*****************************************************************************
    */

void OutputCode( output_file, code )
BIT_FILE *output_file;
180 int code;
{
  int top_of_range;
  int abs_code;
  int bit_count;
185   if ( code == 0 ) {
    OutputRunLength++;
    return;                                 //aqui acaba a funÃğÃčo toda vez que for
        zero o numero a ser escrito
  }
  if ( OutputRunLength != 0 ) {
190     while ( OutputRunLength > 0 ) {
        if ( OutputRunLength <= 16 ) {
                OutputBits( output_file, (unsigned long) (OutputRunLength - 1 ), 4
                    );
                OutputRunLength = 0;
        } else {
195             OutputBits( output_file, 15L, 4 );
                OutputRunLength -= 16;
        }
    }
  }
200   if ( code < 0 )
          abs_code = -code;
  else
          abs_code = code;
  top_of_range = 1;
205   bit_count = 1;
  while ( abs_code > top_of_range ) {
          bit_count++;
          top_of_range = ( ( top_of_range + 1 ) * 2 ) - 1;
  }
210   if ( bit_count < 3 ) //aqui se o numero a ser escrito for menor que 3
      OutputBits( output_file, (unsigned long) ( bit_count + 1 ), 3 );
  else
      OutputBits( output_file, (unsigned long) ( bit_count + 5 ), 4 );
  if (code > 0 )
215     OutputBits( output_file, (unsigned long) code, bit_count );
  else
      OutputBits( output_file, (unsigned long) ( code + top_of_range ) ,bit_count
          ); //aqui se tratam os negativos
}

220 /*****************************************************************************
    */

void WriteDCTData( output_file, output_data )
```

```
BIT_FILE *output_file;
int output_data[ N ][ N ];
{
  int i;
  int row;
  int col;
  double result;
  for ( i = 0 ; i < ( N * N ) ; i++ ) {
    row = ZigZag[ i ].row;
    col = ZigZag[ i ].col;
    result = output_data[ row ][ col ] / Quantum[ row ][ col ];

    OutputCode( output_file, ROUND( result ) );

  }
}
/*
* This routine writes out a strip of pixel data to a GS format file.
*/

/*
    ********************************************************************************************
     */
void WritePixelStrip( output, strip )
FILE *output;
unsigned char strip[ N ][ COLS ];
{
  int row;
  int col;
  for ( row = 0 ; row < N ; row++ )
    for ( col = 0 ; col < COLS ; col++ )
      putc( strip[ row ][ col ], output );
}
/*
* Rotina que implementa a operaÃğÃčo principal da DCT:
*
*           DCT = C * pixels * Ct
*/
void ForwardDCT( input, output )
unsigned char *input[ N ];
int output[ N ][ N ];
{
  double temp[ N ][ N ];
  double temp1;
  int i;
  int j;
  int k;
// Neste ponto se pode definir se as contas serÃčo feitas no Rosa, utilizando as
    funÃğÃţes MatrixMultiply_ROSA
// para isto Ãľ necessÃąrio tirar este cÃşdigo do comentÃąrio e comentar o antigo
/*
int aux_input[N][N];
for(i=0;i<8;i++){
```

```
            for(j=0;j<8;j++){
                    aux_input[i][j]= *(input[j]+i);         //este passo foi
                        necessÃąrio para uniformizar a entrada
                                                            // da funÃğÃčo
                                                                MatrixMultiply_RoSA
            }
    }
    MatrixMultiply_ROSA(aux_input, Ct, temp, 1, 0);         //queremos que normalize os
         valores e nÃčo queremos que arredonde
  */
    for ( i = 0 ; i < N ; i++ ) {
      for ( j = 0 ; j < N ; j++ ) {
        temp[ i ][ j ] = 0.0;
        for ( k = 0 ; k < N ; k++ )
         temp[ i ][ j ] += ( (int) input [ i ][ k ] - 128 ) * Ct[ k ][ j ];
      }
    }
  // MatrixMultiply_ROSA(C, temp, output, 0, 1);  //nÃčo queremos que normalize os
      valores e queremos que arredonde
    for ( i = 0 ; i < N ; i++ ) {
      for ( j = 0 ; j < N ; j++ ) {
        temp1 = 0.0;
        for ( k = 0 ; k < N ; k++ )
         temp1 += C[ i ][ k ] * temp[ k ][ j ];
        output[ i ][ j ] = ROUND( temp1 );
      }
    }

  }
  /*
  * FunÃğÃčo que implementa a transformada inversa
  *
  *           pixels = C * DCT * Ct
  */
  void InverseDCT( input, output )
  int input[ N ][ N ];
  unsigned char *output[ N ];
  {
    double temp[ N ][ N ];
    double temp1;
    int i;
    int j;
    int k;
  /* MatrixMultiply( temp, input, C ); */
    for ( i = 0 ; i < N ; i++ ) {
      for ( j = 0 ; j < N ; j++ ) {
        temp[ i ][ j ] = 0.0;
        for ( k = 0 ; k < N ; k++ )
         temp[ i ][ j ] += input[ i ][ k ] * C[ k ][ j ];
      }
    }
  /* MatrixMultiply( output, Ct, temp ); */
    for ( i = 0 ; i < N ; i++ ) {
```

```
      for ( j = 0 ; j < N ; j++ ) {
        temp1 = 0.0;
        for ( k = 0 ; k < N ; k++ )
325      temp1 += Ct[ i ][ k ] * temp[ k ][ j ];
        temp1 += 128.0;
        if ( temp1 < 0 )
         output[ i ][ j ] = 0;
        else if ( temp1 > 255 )
330      output[ i ][ j ] = 255;
        else
         output[ i ][ j ] = (unsigned char) ROUND( temp1 );
      }
    }
335 }
    /*
    * Esta Ãľ a rotina principal de compressÃčo, que chama outras funÃğÃţes
        necessÃąrias deste mÃşdulo
    */
    void CompressFile( input, output, argc, argv )
340 FILE *input;
    BIT_FILE *output;
    int argc;
    char *argv[];
    {
345   int row;
      int col;
      int i;
      unsigned char *input_array[ N ];
      int output_array[ N ][ N ];
350   int quality;
      if ( argc-- > 0 )
        quality = atoi( argv[ 0 ] );
      else
        quality = 3;
355   if ( quality < 1 || quality > 50 )
        fatal_error( "Illegal quality factor. \n Quality must be > 1 and < 51. \n");
      printf( "Using quality factor of %d \n", quality );
      Initialize( quality );
      OutputBits( output, (unsigned long) quality, 8 );
360   for ( row = 0 ; row < ROWS ; row += N ) {
            ReadPixelStrip( input, PixelStrip );
        for ( col = 0 ; col < COLS ; col += N ) {
            for ( i = 0 ; i < N ; i++ ){
                    input_array[ i ] = PixelStrip[ i ] + col;
365         }
            ForwardDCT( input_array, output_array );
            WriteDCTData( output, output_array );
        }
      }
370   OutputCode( output, 1 );
            /* O Ãžltimo passo do programa Ãľ chamar esta rotina uma Ãžltima vez com
                algum valor para que se houver
```

```
         *  diferente de zero para que se existirem sequencia de zeros no final do
            programa, esta seja escrita no arquivo.
         */
    while ( argc-- > 0 )
375     printf( "Unused argument: %s\n", *argv++ );
}
/* Esta rotina opera o processo inverso da compressÃčo
*/
void ExpandFile( input, output, argc, argv)
380 BIT_FILE *input;
FILE *output;
int argc;
char *argv[];
{
385   int row;
    int col;
    int i;
    int input_array[ N ][ N ];
    unsigned char *output_array[ N ];
390   int quality;
    quality = (int) InputBits( input, 8 );
    printf( "\rUsing quality factor of %d\n", quality );
    Initialize( quality );
    for ( row = 0 ; row < ROWS ; row += N ) {
395   for ( col = 0 ; col < COLS ; col += N ) {
        for ( i = 0 ; i < N ; i++ )
          output_array[ i ] = PixelStrip[ i ] + col;
        ReadDCTData( input, input_array );
        InverseDCT( input_array, output_array );
400   }
      WritePixelStrip( output, PixelStrip );
    }
    while ( argc-- > 0 )
      printf( "Unused argument: %s\n", *argv++ );
405 }
```

I.5 Código da função que implementa a multiplicação de matrizes utilizando o RoSA

```
#include <stdio.h>
#include <stdlib.h>


5 void MatrixMultiply_ROSA(int in_mat_1[N][N], int in_mat_2[N][N], int output[N][N],
    int normal_in_mat_1, int round)

{
        int cont;
        int i;
10      int j;
```

```
        int k;
        unsigned int *global_conf[2];
/*************************************************************/
/*
 * ComeÃğo do uso da arquitetura reconfigurÃąvel
 *        1Âž: EndereÃğamento dos registradores globais, modificados, de saÃŋda e do gerenciador de configuraÃğÃţes
 *        2Âž: Criar as configuraÃğÃţes das celulas. O modo de criaÃğÃčo segue uma rotina progressiva, de modo que para
 * o cÃąlculo de cada item, um valor do registrador global e um do modificado sÃčo multiplicados. Desta forma
 * Ãľ preciso alterar os valores dos registradores de acordo com o objeto a ser calculado na tabela final.
 *    3Âž: Salvamos a primeira linha da primeira matriz seis vezes nos registradores, para que possamos multiplicar
 * pelas seis colunas da segunda matriz em duas configuraÃğÃţes de 720 bits. Pois em cada uma das conf. de 720,
 * calculam-se 3 valores finais.
 *        4Âž: Salva-se a segunda linha da primeira matriz seis vezes nos registradores, para calcular os valores finais
 * da segunda linha da matriz final.
 *        5Âž: Assim se prossegue atÃľ a sexta linha, para entÃčo passar para o calculo do segundo quadrante 6x2. Neste
 * cÃąlculo se salvam duas vezes a primeira linha da primeira tabela, depois duas vezes a segunda linha da primeira tabela,
 * e por fim duas vezes a terceira linha da primeira tabela.
 */
        int *glob_reg[48];
        int *mod_reg[48];
        //Aqui criamos os ponteiros para os endereÃğos dos registradores
        for( i=0; i<48; i++ ){
                glob_reg[i]=(unsigned int*)(0x00400000+i);
                mod_reg[i]=(unsigned int*)(0x00400030+i);
        };
        //Ponteiro para o configurador de memÃşria
        unsigned int *conf_m;
        conf_m=(unsigned int*)0x004000FF;

        //Ponteiro para os registradores de saÃŋda
        int *out_reg[6];
        for( i=0; i<6; i++){
                out_reg[i]=(unsigned int*)(0x004000F9+i);
        };

        //Aloca-se espaÃğo para as duas Ãžnicas configuraÃğÃţes genÃľricas
        global_conf[0]=(unsigned int*)malloc(23*(sizeof(unsigned int)));
        global_conf[1]=(unsigned int*)malloc(23*(sizeof(unsigned int)));

        //Zeram-se os valores das duas configuraÃğÃţes para posterior preenchimento
        for(i=0; i<2; i++){
                for(k=0; k<23; k++){
                        *(global_conf[i]+k)=0;
                };
```

```
55          };

            //Aqui escrevemos as duas configuraÃğÃţes genÃľricas para calcular toda a
                matriz


60  // ****************************************

            *(global_conf[0])=0x0E40E38E;
            *(global_conf[0]+1)=0x38208200;
            *(global_conf[0]+2)=0x30013102;
65          *(global_conf[0]+3)=0x3203330E;
            *(global_conf[0]+4)=0x40E38E38;
            *(global_conf[0]+5)=0x20820434;
            *(global_conf[0]+6)=0x05350636;
            *(global_conf[0]+7)=0x07370E40;
70          *(global_conf[0]+8)=0xE38E3820;
            *(global_conf[0]+9)=0x82083809;
            *(global_conf[0]+10)=0x390A3A0B;
            *(global_conf[0]+11)=0x3B0E40E3;
            *(global_conf[0]+12)=0x8E382082;
75          *(global_conf[0]+13)=0x0C3C0D3D;
            *(global_conf[0]+14)=0x0E3E0F3F;
            *(global_conf[0]+15)=0x0E40E38E;
            *(global_conf[0]+16)=0x38208210;
            *(global_conf[0]+17)=0x40114112;
80          *(global_conf[0]+18)=0x4213430E;
            *(global_conf[0]+19)=0x40E38E38;
            *(global_conf[0]+20)=0x20821444;
            *(global_conf[0]+21)=0x15451646;
            *(global_conf[0]+22)=0x17470000;
85
            *(global_conf[1])=0x0E40E38E;
            *(global_conf[1]+1)=0x38208218;
            *(global_conf[1]+2)=0x4819491A;
            *(global_conf[1]+3)=0x4A1B4B0E;
90          *(global_conf[1]+4)=0x40E38E38;
            *(global_conf[1]+5)=0x20821C4C;
            *(global_conf[1]+6)=0x1D4D1E4E;
            *(global_conf[1]+7)=0x1F4F0E40;
            *(global_conf[1]+8)=0xE38E3820;
95          *(global_conf[1]+9)=0x82205021;
            *(global_conf[1]+10)=0x51225223;
            *(global_conf[1]+11)=0x530E40E3;
            *(global_conf[1]+12)=0x8E382082;
            *(global_conf[1]+13)=0x24542555;
100         *(global_conf[1]+14)=0x26562757;
            *(global_conf[1]+15)=0x0E40E38E;
            *(global_conf[1]+16)=0x38208228;
            *(global_conf[1]+17)=0x5829592A;
            *(global_conf[1]+18)=0x5A2B5B0E;
105         *(global_conf[1]+19)=0x40E38E38;
            *(global_conf[1]+20)=0x20822C5C;
```

```
*(global_conf[1]+21)=0x2D5D2E5E;
*(global_conf[1]+22)=0x2F5F0000;

/***********Inicio do quadrante 6x6***********/
// Este laÃğo valora os registradores modificados com os valores das 6 primeiras colunas da matriz de saÃŋda
for ( k=0 ; k<6 ; k++ ) {
    for ( j=0 ; j<8 ; j++ ){
        *mod_reg[(k*8)+j]= in_mat_2[j][k];
    };
};

//AutomatizaÃğÃčo do calculo do primeiro quadrante, que Ãl' 6x6, com 36 valores
for(i=0; i<6; i++){
//      Neste laÃğo valoramos os registradores globais com os valores da primeira linha da matriz de entrada
        for ( j = 0 ; j < 6 ; j++ ) {
                for ( k = 0 ; k < 8 ; k++ ){
                        if(normal_in_mat_1){
                                *glob_reg[(j*8)+k]= (in_mat_1[i][k] -128);
                        } else {
                                *glob_reg[(j*8)+k]= in_mat_1[i][k];
                        };
                };
        };
// Efetuamos os primeiros cÃąlculos enviando para o gerenciador de configuraÃğÃţes o endereÃğo da configuraÃğÃčo

        *conf_m=(unsigned int)global_conf[0];

// Lemos os registradores de saida e somamos de dois em dois para obter os trÃłs primeiros valores finais
        if(round){
                output[i][0]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))/10000;
                output[i][1]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))/10000;
                output[i][2]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))/10000;
        }else{
                output[i][0]=(*out_reg[0]+*out_reg[1]);
                output[i][1]=(*out_reg[2]+*out_reg[3]);
                output[i][2]=(*out_reg[4]+*out_reg[5]);
        }
// Efetuamos os ultimos calculos da primeira linha da 6x6 e calculamos os ultimos tres valores
        *conf_m=(unsigned int)global_conf[1];

        if(round){
                output[i][3]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))/10000;
                output[i][4]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))/10000;
                output[i][5]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))/10000;
        }else{
                output[i][3]=(*out_reg[0]+*out_reg[1]);
                output[i][4]=(*out_reg[2]+*out_reg[3]);
```

```
                output[i][5]=(*out_reg[4]+*out_reg[5]);
155             }
        };
        /********** Fim do calculo do quadrante 6x6 **************/

        /***********Inicio do quadrante 2x6***********/
160
        //Aproveitam-se os registradores modificados do cÃąlculo anterior, pois
            tambem serÃčo as seis primeiras
        // da matriz in_mat_2

        //Automatizando o calculo do segundo quadrante, que Ãľ 2x6, com 12 valores
165     for(i=6; i<8; i++){
            //      Neste laÃğo valoramos os registradores globais com os
                valores da primeira linha da matriz de entrada
            for ( j = 0 ; j < 6 ; j++ ) {
                    for ( k = 0 ; k < 8 ; k++ ){
                        if(normal_in_mat_1){
170                         *glob_reg[(j*8)+k]= ( in_mat_1[i][k
                                ] -128 );
                        }else{
                            *glob_reg[(j*8)+k]= in_mat_1[i][k];
                        };
                    };
175         };
            // Efetuamos os primeiros cÃąlculos enviando para o gerenciador de
                configuraÃğÃţes o endereÃğo da configuraÃğÃčo

            *conf_m=(unsigned int)global_conf[0];

180         // Lemos os registradores de saida e somamos de dois em dois para
                obter os trÃłs primeiros valores finais
            if(round){
                output[i][0]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))/10000;
                output[i][1]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))/10000;
                output[i][2]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))/10000;
185         }else{
                output[i][0]=(*out_reg[0]+*out_reg[1]);
                output[i][1]=(*out_reg[2]+*out_reg[3]);
                output[i][2]=(*out_reg[4]+*out_reg[5]);
            };
190
            // Efetuamos os ultimos calculos da primeira linha da 6x6 e
                calculamos os ultimos tres valores
            *conf_m=(unsigned int)global_conf[1];

            if(round){
195             output[i][3]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))/10000;
                output[i][4]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))/10000;
                output[i][5]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))/10000;
            }else{
                output[i][3]=(*out_reg[0]+*out_reg[1]);
200             output[i][4]=(*out_reg[2]+*out_reg[3]);
```

```
                output[i][5]=(*out_reg[4]+*out_reg[5]);
        };

};
/********** Fim do calculo do quadrante 2x6 **************/


/***********Inicio do quadrante 6x2***********/

//Colocam nos registradores modificados os valores das 7 e 8 colunas da
    matriz in_mat_2
cont=0;
for(i=0; i<3; i++){
        for ( k = 6 ; k < 8 ; k++ ) {
                for ( j = 0 ; j < 8 ; j++ ){
                        *mod_reg[cont]= in_mat_2[j][k];
                        cont++;
                };
        };
};
//Automatizando o calculo do terceiro quadrante, que Ãl 6x2, com 12 valores
for(i=0; i<2; i++){
        cont=0;
        for (j=(i*3); j<(3+(i*3)); j++) {
                for (k=0; k<8; k++){
                        if(normal_in_mat_1){
                                *glob_reg[cont]= ( in_mat_1[j][k] -128 );
                                cont++;
                        }else{
                                *glob_reg[cont]= in_mat_1[j][k];
                                cont++;
                        };
                };
                for ( k=0; k<8; k++ ){
                        if(normal_in_mat_1){
                                *glob_reg[cont]= ( in_mat_1[j][k] -128 );
                                cont++;
                        }else{
                                *glob_reg[cont]= in_mat_1[j][k];
                                cont++;
                        };
                };
        };
        *conf_m=(unsigned int)global_conf[0];

        // Lemos os registradores de saida e somamos de dois em dois para
            obter os trÃłs primeiros valores finais
        if(round){
                output[0+(3*i)][6]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))
                    /10000;
                output[0+(3*i)][7]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))
                    /10000;
```

```
                output[1+(3*i)][6]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))
                    /10000;
        }else{
                output[0+(3*i)][6]=(*out_reg[0]+*out_reg[1]);
                output[0+(3*i)][7]=(*out_reg[2]+*out_reg[3]);
                output[1+(3*i)][6]=(*out_reg[4]+*out_reg[5]);
        }
        // Efetuamos os ultimos calculos da primeira linha da 6x6 e
            calculamos os ultimos tres valores
        *conf_m=(unsigned int)global_conf[1];

        if(round){
                output[1+(3*i)][7]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))
                    /10000;
                output[2+(3*i)][6]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))
                    /10000;
                output[2+(3*i)][7]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))
                    /10000;
        }else{
                output[1+(3*i)][7]=(*out_reg[0]+*out_reg[1]);
                output[2+(3*i)][6]=(*out_reg[2]+*out_reg[3]);
                output[2+(3*i)][7]=(*out_reg[4]+*out_reg[5]);
        };
};

/********** Fim do calculo do quadrante 6x2 **************/

/************Inicio do quadrante 2x2***********/
//Aproveitam-se os registradores modificados do cÃąlculo anterior (
    quadrante 6x2), pois tambem serÃčo as
// duas Ãžltimas colunas da matriz in_mat_2 que serÃčo necessarias para
    este calculo do quadrante 2x2.

//Automatizando o calculo do Ãžltimo quadrante, que Ãl 2x2, com 12 valores

cont=0;
for (j=6; j<8; j++) {
        for (k=0; k<8; k++){
                if(normal_in_mat_1){
                        *glob_reg[cont]= ( in_mat_1[j][k] -128 );
                        cont++;
                }else{
                        *glob_reg[cont]= in_mat_1[j][k];
                        cont++;
                };
        };
        for ( k=0; k<8; k++ ){
                if(normal_in_mat_1){
                        *glob_reg[cont]= (  in_mat_1[j][k] -128 );
                        cont++;
                }else{
                        *glob_reg[cont]=  in_mat_1[j][k];
                        cont++;
```

```
295                             };
                        };
                };
        *conf_m=(unsigned int)global_conf[0];

300     // Lemos os registradores de saida e somamos de dois em dois para obter os
            trÃłs primeiros valores finais
        if(round){
                output[6][6]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))/10000;
                output[6][7]=(/*ROUND*/(*out_reg[2]+*out_reg[3]))/10000;
                output[7][6]=(/*ROUND*/(*out_reg[4]+*out_reg[5]))/10000;
305     }else{
                output[6][6]=(*out_reg[0]+*out_reg[1]);
                output[6][7]=(*out_reg[2]+*out_reg[3]);
                output[7][6]=(*out_reg[4]+*out_reg[5]);
        };
310     // Efetuamos os ultimos calculos da primeira linha da 6x6 e calculamos os
            ultimos tres valores
        *conf_m=(unsigned int)global_conf[1];

        if(round){
                output[7][7]=(/*ROUND*/(*out_reg[0]+*out_reg[1]))/10000;
315     }else{
                output[7][7]=(*out_reg[0]+*out_reg[1]);
        };
        //os resultados das 4 ultimas celulas sÃčo rejeitados pois a matriz ja esta
             pronta

320     /********** Fim do calculo do quadrante 6x2 **************/
        /*
         * Fim do uso da arquitetura reconfigurÃąvel
         */
/*************************************************************/
325
};
```