Daniel Pereira Volpato e Leonardo Luiz Ecco

Projeto e Validação de um IP para o Padrão JPEG e sua Integração a uma Plataforma descrita no estilo TLM

Florianópolis – SC

Outubro / 2007

Daniel Pereira Volpato e Leonardo Luiz Ecco

Projeto e Validação de um IP para o Padrão JPEG e sua Integração a uma Plataforma descrita no estilo TLM

Monografia apresentada ao curso de Bacharelado em Ciências da Computação da Universidade Federal de Santa Catarina como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação.

Orientador:
Prof. Dr. Luiz Cláudio Villar dos Santos

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO

Florianópolis – SC

Outubro / 2007

Monografia de graduação sob o título *"Projeto e validação de um IP para o padrão JPEG e sua integração numa plataforma descrita no estilo TLM"*, defendida por Daniel Pereira Volpato e Leonardo Luiz Ecco e aprovada em 30 de outubro de 2007, em Florianópolis , Santa Catarina, pela banca examinadora constituída pelos professores:

Prof. Dr. Luiz Cláudio Villar dos Santos
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. José Luís Almada Güntzel
Universidade Federal de Santa Catarina
Membro da Banca

Prof. Dr. Olinto José Varela Furtado
Universidade Federal de Santa Catarina
Membro da Banca

RESUMO

Para a indústria de sistemas, o reúso de IPs e a engenharia baseada em modelos são mecanismos fundamentais para alcançar ganhos de produtividade significativos no projeto de sistemas integrados (SoCs) complexos. Nesse contexto, duas técnicas têm se mostrado promissoras: o projeto baseado em plataforma (PBD) e a modelagem baseada em transações (TLM).

Este trabalho explora conteitos de PBD e TLM para a concepção de um acelerador em hardware, projetado na forma de um bloco de propriedade intelectual (IP), para potencial reuso em plataformas multimídia.

Através do "profiling" de um programa de codificação no formato JPEG e da análise de potencial reuso, escolheu-se implementar em hardware a função "Discrete Cosine Transform" (DCT). O modelo funcional do IP selecionado foi integrado em uma plataforma, descrita no estilo TLM, a qual é capaz de realizar a compressão de "bitmaps" para o formato JPEG. A plataforma adotada compõe-se de um processador PowerPC, uma memória, um barramento e o IP sob projeto.

Validou-se o modelo funcional do IP integrado à plataforma, através de comparação dos resultados nela obtidos com os resultados esperados produzidos pelo programa de codificação JPEG (modelo de referência).

Assim, o resultado principal deste trabalho foi a caracterização do comportamento funcional da interface do IP, definido por um conjunto de estímulos aplicado às suas entradas e o respectivo conjunto de resultados às suas saídas. Tal caracterização é crucial para o projeto de validação de IP em nível RT com vistas à sua prototipação em FPGA e em silício, as quais são objeto de trabalho em andamento.

LISTA DE FIGURAS

1 Metodologia de validação utilizada . . . . . p. 16
2 Diagrama em blocos da plataforma . . . . . p. 17
3 Módulo forward_DCT . . . . . p. 23
4 A interface *simple_bus_slave_if* . . . . . p. 25
5 Implementação da função *read* . . . . . p. 26
6 Implementação da função *write* . . . . . p. 27
7 A interface *simple_bus_blocking_if* . . . . . p. 28
8 Trecho do código de main.cpp . . . . . p. 29
9 Arquitetura RTL do IP. . . . . . p. 30
10 Distribuição do tempo gasto na execução da plataforma . . . . . p. 34
11 Imagem “boy“, em tons de cinza . . . . . p. 39
12 Imagem “jpeg_testimg“ (que acompanha o software cjpeg) . . . . . p. 39
13 Imagem “mibench_small“ (contida no benchmark MiBench) . . . . . p. 40
14 Imagem “mibench_large“ (contida no benchmark MiBench) . . . . . p. 40
15 Imagem “mediabench“ (contida no benchmark MediaBench) . . . . . p. 41
16 Imagem “nasa“ . . . . . p. 41
17 Imagem “ufsc_ctc“ . . . . . p. 42

LISTA DE TABELAS

1 Conteúdo do arquivo $defs.arp$ mostrando a configuração da plataforma ...... p. 18

2 “Profiling“ do aplicativo cjpeg para a figura 13, 192Kb .... p. 21

3 “Profiling“ do aplicativo cjpeg para a figura 14, 768Kb .... p. 21

4 “Profiling“ do aplicativo cjpeg para a figura 16, 20Mb .... p. 21

5 “Profiling“ do aplicativo cjpeg para a figura 16, 183Mb .... p. 21

6 Sumário dos “profilings“ realizados sobre o aplicativo cjpeg .... p. 21

7 Entradas e saídas do módulo forward_DCT .... p. 23

8 Registradores do IP - “Offsets“ com relação ao endereço-base e tipo .... p. 24

9 Comparação de tempos de execução: plataforma *x* software .... p. 32

10 Sumário dos “profilings“realizados durante execução da plataforma........... p. 33

11 Configuração das figuras utilizadas para “profiling“ e validação............... p. 38

SUMÁRIO

1 MOTIVAÇÃO .... p. 8

2 BREVE REVISÃO BIBLIOGRÁFICA .... p. 10

2.1 PROJETO BASEADO EM PLATAFORMA .... p. 10

2.2 MODELAGEM EM NÍVEL DE TRANSAÇÃO .... p. 11

2.3 SYSTEMC .... p. 12

2.4 COMPRESSÃO JPEG .... p. 14

3 METODOLOGIA E INFRAESTRUTURA .... p. 15

3.1 METODOLOGIA DE PROJETO E VALIDAÇÃO .... p. 15

3.2 INFRA-ESTRUTURA DE MODELAGEM DA PLATAFORMA .... p. 16

3.2.1 ARCHC REFERENCE PLATFORM - ARP .... p. 16

3.2.2 MAPEAMENTO DA APLICAÇÃO DE COMPRESSÃO JPEG PARA UMA PLATAFORMA ARP .... p. 17

4 PROJETO DO IP .... p. 19

4.1 SELEÇÃO DO IP .... p. 19

4.1.1 IMPLEMENTAÇÃO DE COMPRESSÃO JPEG NO MIBENCH .... p. 19

4.1.2 A FERRAMENTA GPROF .... p. 19

4.1.3 “PROFILING“ E SELEÇÃO .... p. 20

4.2 MODELAGEM FUNCIONAL DO IP NA PLATAFORMA TLM .... p. 22

4.2.1 ENTRADAS E SAÍDAS DO MÓDULO FORWARD_DCT .... p. 22

4.2.2 MAPEAMENTO DE E/S EM MEMÓRIA DO MÓDULO FORWARD_DCT . . . p. 23

4.2.3 MODELAGEM DAS TRANSAÇÕES . . . p. 25

4.3 A INSTANCIAÇÃO DE COMPONENTES NA PLATAFORMA . . . p. 28

4.4 MODELAGEM RTL DO IP . . . p. 30

5 RESULTADOS EXPERIMENTAIS . . . p. 31

5.1 CONFIGURAÇÃO EXPERIMENTAL . . . p. 31

5.2 VALIDAÇÃO DO IP FUNCIONAL NA PLATAFORMA . . . p. 32

5.3 COMPARAÇÃO DE TEMPOS DE EXECUÇÃO ENTRE PLATAFORMA E SOFTWARE . . . p. 32

6 CONCLUSÃO . . . p. 35

6.1 TRABALHOS FUTUROS . . . p. 35

REFERÊNCIAS . . . p. 37

ANEXO A – IMAGENS UTILIZADAS PARA “PROFILING“ E VALIDAÇÃO . . . p. 38

APÊNDICE A – MANUAL DO IP FORWARD_DCT . . . p. 43

APÊNDICE A – CÓDIGO-FONTE . . . p. 48

1 MOTIVAÇÃO

A indústria de semicondutores vem vivendo há cerca de uma década a revolução dos sistemas integrados (SoCs) (MARTIN; CHANG, 2003). Um SoC é um sistema eletrônico completo em um único chip, integrando componentes de hardware e de software (GHENASSIA, 2005, p. 2). Os blocos fundamentais de um SoC são denominados *propriedades intelectuais* (IPs), blocos que realizam tarefas específicas, tais como a de periféricos ou aceleradores de funções críticas.

Os SoCs foram viabilizados com o adento das tecnologicas CMOS nanométricas, permitindo a realização de computadores embarcados, “sistemas de processamento de informação embutidos em um produto maior e geralmente não diretamente visíveis ao usuário“ (MARWEDEL, 2006, p. 1). Exemplos de aplicações que utilizam SoCs para computação embarcada são: telefones celulares, câmeras digitais, video-games, controladores de freio ABS, sistemas de GPS, sistemas anti-colisão em aviões, sistemas médicos, entre outros.

A crescente complexidade dos SoCs, o alto custo das máscaras de circuitos integrados para tecnologias CMOS nanométricas e o crescente custo de engenharia não recorrente deram origem à um novo paradigma de projeto denominado “Projeto baseado em Plataforma“ (SANGIOVANNI-VINCENTELLI; MARTIN, 2001). Essencialmente, uma plataforma consiste de um projeto genérico para um determinado domínio de aplicação, que pode ser personalizado pela seleção de IPs apropriados e estendido com a concepção de componentes adicionais. O projeto baseado em plataforma pode ser simplificado, então, como a utilização desta arquitetura de referência para um certo domínio associada com o reuso de IPs para composição e customização da mesma.

Os blocos que compõem uma plataforma podem ser descritos em diferentes níveis de abstração como, por exemplo: nível de transistores, nível de portas lógicas e nível de transferência entre registradores (RTL). Este último vinha sendo o mais usado pela indústria como ponto de partida para o projeto de circuitos integrados e descreve um componente como sendo formado geralmente por um *datapath* comandado por uma unidade de controle.

Além disso, a crescente necessidade da indústria de SoCs por um nível de abstração mais elevado levou ao surgimento da modelagem em nível de sistema eletrônico (ESL), que possibilita a descrição de módulos em diversos estilos, entre os quais está a modelagem em nível de transações (TLM).

TLM é a abstração dos detalhes de comunicação através dos meios de conexão para viabilizar a simulação de toda a plataforma, além de I/O mapeado em memória, o que facilita o uso do módulo pelo software rodando na plataforma. Desta forma, o TLM permite que a funcionalidade de um módulo seja separada da comunicação, o que implica no favorecimento do reuso. E traz vantagens ainda na implementação da comunicação, trazendo o foco para a funcionalidade da comunicação - quais dados, de onde e para onde são transmitidos -, e não nos detalhes do protocolo de comunicação, que são abstraídos (GHENASSIA, 2005).

No presente trabalho, realizamos um estudo sobre estas duas técnicas contemporâneas na área de projeto de SoCs: o projeto baseado em plataforma (PBD) e a modelagem transacional (TLM). A importância das técnicas mencionadas na atual conjuntura vivida pela indústria de SoCs foi o fator que nos motivou a estudá-las nesse trabalho.

O produto desse estudo foi uma plataforma descrita no estilo TLM, composta por um processador PowerPC, um bloco de memória e um IP interconectados através de um barramento, a qual é capaz de realizar compressão de "bitmaps" para o formato JPEG. As atividades aqui descritas estão no âmbito do projeto *APISCE: AUTOMAÇÃO, PROJETO E INTEGRAÇÃO DE SISTEMAS COMPUTACIONAIS EMBARCADOS*, em execução no Laboratório de Automação de Projeto de Sistemas (LAPS)[1] da UFSC.

Este trabalho é organizado da seguinte maneira: no capítulo 2, uma breve revisão bibliográfica abordando os tópicos de projeto baseado em plataforma, modelagem em nível de transação, SystemC e a compressão em formato JPEG. No capítulo 3, descrevemos a metodologia para desenvolvimento de IPs e a infra-estrutura de modelagem de plataformas utilizada neste trabalho. Já no capítulo 4, falamos do projeto do IP desenvolvido, dos critérios para seleção do mesmo (que fizeram uso da técnica de "profiling") e do processo de modelagem funcional do IP e sua integração na plataforma TLM utilizada. No capítulo 5, apresentamos os resultados experimentais obtidos, bem como a configuração utilizada para sua geração, e uma análise dos mesmos a partir dos conhecimentos adquiridos ao longo deste trabalho. Finalmente, no capítulo 6, demonstramos as conclusões as quais chegamos e os trabalhos futuros no contexto deste trabalho.

[1] Saiba mais sobre o Laboratório de Automação de Projeto de Sistemas em: `http://www.laps.inf.ufsc.br/`

2 BREVE REVISÃO BIBLIOGRÁFICA

Este capítulo faz uma análise dos principais trabalhos que abordam técnicas utilizadas na concepção de IPs. O objetivo dessa breve revisão é a de amparar conceitualmente o trabalho, através do reuso de técnicas consagradas ou promissoras. A revisão em profundidade da vasta literatura só faria sentido se a pretensão fosse a de desenvolver novas técnicas.

2.1 PROJETO BASEADO EM PLATAFORMA

O projeto baseado em plataforma surgiu como uma alternativa para lidar com três grandes problemas que a indústria microeletrônica vem enfrentando (SANGIOVANNI-VINCENTELLI et al., 2004):

- *Horizontalização* do modelo de negócios. Acostumadas a manter o controle sobre todo o processo de desenvolvimento do produto, as companhias viram-se forçadas a focar esforços em suas competências principais devido ao aumento da complexidade dos sistemas e ao número de tecnologias envolvidas. Assim, para continuar no mercado com produtos de boa qualidade, passaram a terceirizar partes do processo para outras empresas. Há, portanto, a necessidade de um mecanismo formal para a integração de toda a cadeia de projeto.
- Para garantir uma boa margem de lucros, o “time-to-market“ deve ser reduzido - apesar do crescimento exponencial da complexidade dos projetos. Por conseguinte, é necessário aumentar o reuso dos componentes em todos os níveis de abstração, e possibilitar uma certa flexibilidade de modo que mudanças de última hora possam ser tratadas.
- Aumento dos custos em engenharia não-recorrente, devido ao alto custo associado ao processo. Por exemplo, a fabricação de uma máscara pode custar milhões de dólares e a construção de unidades de produção envolve custos da ordem de bilhões de dólares. Isso

torna crucial que se obtenha um projeto funcional na primeira tentativa - construir um “chip“ que não funciona causaria prejuízos inaceitáveis.

Segundo Sangiovanni-Vincentelli e Martin (2001, p. 410), uma plataforma é uma “biblioteca de componentes junto com suas regras de composição“. Essa biblioteca contém tanto blocos que realizam a computação em si, como blocos de comunicação utilizados para interconexão. Assim, um projetista pode escolher um conjunto de componentes da biblioteca ou ajustar parâmetros de componentes reconfiguráveis e derivar, desse modo, sua instância da plataforma.

Dado que os blocos da biblioteca de uma plataforma podem estar em diferentes níveis de abstração, um bloco de alto nível pode ser refinado para um nível mais baixo e substituído na plataforma já existente e previamente validada. O bloco refinado será validado nesse ambiente já certificado, permitindo que os erros eventualmente encontrados sejam facilmente localizados e corrigidos, pois só podem estar no novo bloco. Execuções desses passos para todos os blocos visam o refinamento da plataforma como um todo para uma instância em nível mais baixo, e que servirá de base para os próximos passos até a fabricação do “chip“.

Além disso, a exploração do espaço de projeto (*design-space exploration*) é favorecida numa plataforma de alto nível pois componentes podem ser substituídos e simulados mais rapidamente. Isso permite, por exemplo, testar diferentes modelos de processador e verificar qual atenderia melhor as restrições do sistema.

2.2 MODELAGEM EM NÍVEL DE TRANSAÇÃO

Como já foi mencionado, a indústria de SoCs vêm enfrentando pressões para reduzir o “time-to-market“, além de ter de lidar com a crescente complexidade e custo dos sistemas projetados. Para tanto, uma estratégia promissora é a de antecipar os ciclos de desenvolvimento de software e de exploração de arquitetura durante o projeto. Isso culmina na necessidade de um aumento no nível de abstração utilizado para descrição dos sistemas, para viabilizar a simulação de sistemas complexos completos.

De acordo com (GHENASSIA, 2005, p. 25), três critérios precisam ser levados em consideração quando se procura uma solução intermediária entre uma descrição algorítmica (onde não existe uma noção de particionamento de hardware e software) e uma descrição RTL (uma descrição precisa do hardware que, entretanto, torna as simulações mais lentas). São eles:

- **Velocidade:** Em simulações de modelos muito detalhados, o tempo acaba se tornando um fator proibitivo.

- **Precisão:** A simulação precisa fornecer resultados confiáveis.
- **Pequeno custo de modelagem:** O esforço de modelagem gasto precisa ser pequeno para não aumentar o custo total do projeto..

Assim, utilizar modelagem "bit-true" com precisão de ciclos não é uma boa alternativa dados os propósitos mencionados anteriormente, uma vez que a velocidade de simulação desse tipo de modelo nos dá somente um ganho de cerca de uma ordem de magnitude em relação à simulação RTL. Ademais, os detalhes envolvidos na construção do modelo precisam ser extraídos da descrição RTL - um processo lento. Tampouco é interessante utilizar modelagem temporal, dado que apesar de ser bastante útil para análise de desempenho, não é preciso o suficiente para que se possa iniciar a etapa de desenvolvimento de software.

Nesse contexto surge a modelagem em nível transacional (TLM) - um estilo de descrição onde os detalhes da comunicação são abstraídos em transações. O estilo TLM é um meio-termo entre o modelo "bit-true" com precisão de ciclos e uma descrição algorítmica (não temporizada). A simulação TLM é rápida, a interface dos componentes é "bit-true" e o esforço de modelagem é razoavelmente pequeno.

A característica mais interessante de se utilizar um modelo TLM é que ele funciona como um modelo de referência tanto para as equipes que desenvolvem software quanto para as equipes que desenvolvem hardware de um sistema.

Tão logo o modelo esteja disponível, o software pode começar a ser desenvolvido (dada a interface "bit-true" dos componentes do modelo) - ou seja, o ciclo de desenvolvimento de software é trazido para as etapas iniciais do projeto.

Ao mesmo tempo que o software está sendo desenvolvido, a equipe de hardware pode trabalhar em uma descrição RTL do hardware. Quando ela finalmente for obtida, basta confrontá-la com o modelo TLM para validá-la. Ou seja, a utilização de uma plataforma TLM de referência torna possível o co-projeto de software e hardware, reduzindo o tempo necessário para disponibilizar o produto no mercado.

2.3 SYSTEMC

O uso de linguagens de programação convencionais para modelar componentes tanto de hardware como de software tornaria possível um ganho significativo de produtividade. Algumas das vantagens dessa abordagem são((BLACK; DONOVAN, 2004):

- **Exploração de particionamento software/hardware**
- **Verificação funcional desde os primeiros estágios do projeto**
- **Especificação executável do sistema que se está projetanto**

Entretanto, existem três razões principais pelas quais linguagens de programação convencionais são inadequadas para modelagem de componentes de hardware:

- **Noção de eventos sequenciados no tempo:** Não há uma noção de tempo.
- **Concorrência:** Componentes de hardware são inerentemente concorrentes. Em geral, as linguagens de programação não dão suporte nativo a concorrência.
- **Tipos de dados inadequados de hardware:** Faltam tipos de dados adequados para a modelagem de hardware.

Uma solução encontrada para lidar com esses problemas foi extender a linguagem C++ com uma biblioteca de classes e um “kernel“ de simulação. O “kernel“ de simulação introduz a noção de tempo e a biblioteca possui os tipos de dados necessários para a descrição de componentes de hardware, além de dar suporte à concorrência. A essa estrutura de modelagem foi dado o nome de SystemC.

SystemC foi anunciado em setembro de 1999 por um conjunto de companhias líderes nos setores de Automação Eletrônica de Projetos (EDA), propriedade intelectual (IP) e semicondutores. Por ser aberto, as empresas de EDA tem total acesso a sua implementação, de modo que a construção de ferramentas que operem sobre modelos funcionais descritos em SystemC fica fácil. Além disso, nenhum tipo de licença para uso é necessária, tornando-o atrativo à indústria.

SystemC tem se mostrado uma boa alternativa para a modelagem em nível de sistema (ESL). Em especial, o estilo TLM foi recentemente padronizado e tem se mostrado bastante promissor para a diminuição do “time-to-market“ e para o projeto concorrente de hardware e software, conforme relatos da indústria (GHENASSIA, 2005).

Para desenvolver um modelo, a infra-estrutura necessária resume-se às ferramentas e ambiente de desenvolvimento C++ da preferência do projetista. Durante a escrita do modelo, basta incluir os *headers* (cabeçalhos contendo a declaração das classes que modelam os componentes) da biblioteca de SystemC nos arquivos-fonte. Na hora da compilação é necessário linkar o projeto com as classes pré-compiladas da biblioteca e com o kernel de simulação do SystemC.

Vale salientar a possibilidade de descrever diferentes componentes do sistema em diferentes níveis de abstração (é possível ter componentes descritos em nível funcional, que especificam o que o componente deve fazer, cooperando com componentes descritos em nível RTL, que especificam como um componente realiza uma tarefa).

2.4 COMPRESSÃO JPEG

O padrão JPEG foi definido pelo *Joint Photographic Experts Group* no ano de 1992 e rapidamente tornou-se uma referência compressão de imagens fotográficas. De acordo com (AGOSTINI; SILVA; BAMPI, 2001), a compressão de uma imagem para o formato JPEG é um processo de cinco etapas:

- **Conversão do o espaço de cores:** Conversão do espaço de cores RGB (que possui apenas componentes de cor) para um espaço de cores do tipo luminância e crominância. Isso é necessário porque existe um grau elevado de correlação entre as componentes R, G e B, dificultando o processamento individual delas.
- ***Downsampling*:** O olho humano é menos sensível às informações de crominância do que de luminância. Na etapa de *downsampling*, parte da informação de crominância é eliminada, com o intuito de aumentar a taxa de compressão da imagem.
- **DCT 2D:** A transformada discreta do cosseno em duas dimensões é utilizada para transformar a representação da informação do domínio espacial para o domínio das frequências.
- **Quantização:** Nessa etapa, as frequências mais elevadas são atenuadas ou até mesmo eliminadas, uma vez que elas tendem a contribuir menos com a informação da imagem.
- **Codificação de entropia:** Consiste na aplicação de um conjunto de técnicas para compressão sem perdas nas matrizes de coeficientes já quantizados (resultado do passo anterior), de modo que seja obtida uma representação com o menor número possível de bits.

Os dois primeiros passos não são necessários quando se comprime imagens em preto e branco.

3 METODOLOGIA E INFRAESTRUTURA

Neste capítulo trataremos da metodologia para desenvolvimento de IPs e também da infraestrutura para plataformas TLM utilizadas nesse trabalho.

3.1 METODOLOGIA DE PROJETO E VALIDAÇÃO

Depois que se obtém a descrição de um componente de hardware, o passo seguinte - independentemente do nível da descrição - é verificar que as funcionalidades e requisitos estão de acordo com o esperado. Gerar manualmente um conjunto de estímulos e resultados - para verificar a corretude do modelo - é extremamente trabalhoso e pouco produtivo, além de bastante propício a erros.

Para lidar com esse problema, usaremos o conceito de "golden model". Um "golden model" é um modelo de referência que, garantidamente, possui as funcionalidades e cumpre os requisitos previamente especificados.

Portanto, no projeto de um componente de hardware, é útil partir de uma descrição algorítmica não temporizada - que modela funcionalidade. Essa descrição servirá, posteriormente, como "golden model" para a verificação funcional de modelos descritos em outros níveis de abstração, como por exemplo TLM e RTL.

A figura 1 ajuda a explicar a metodologia apresentada. Estímulos são utilizados para excitar o modelo de referência ("golden model") e a plataforma. Uma vez que os resultados do processamento estão disponíveis, é feita uma comparação. Caso os resultados obtidos, tanto para a plataforma quanto para o "golden model" sejam iguais, significa que, para um dado estímulo, a plataforma implementa as mesmas funcionalidades que o "golden model". Caso contrário, existem um ou mais erros na implementacao da plataforma. Assim que os erros forem corrigidos, uma nova rodada de execuções acontece e novamente é feita uma comparação. Esse processo é repetido até que os resultados sejam idênticos. É uma boa prática testar diferentes estímulos

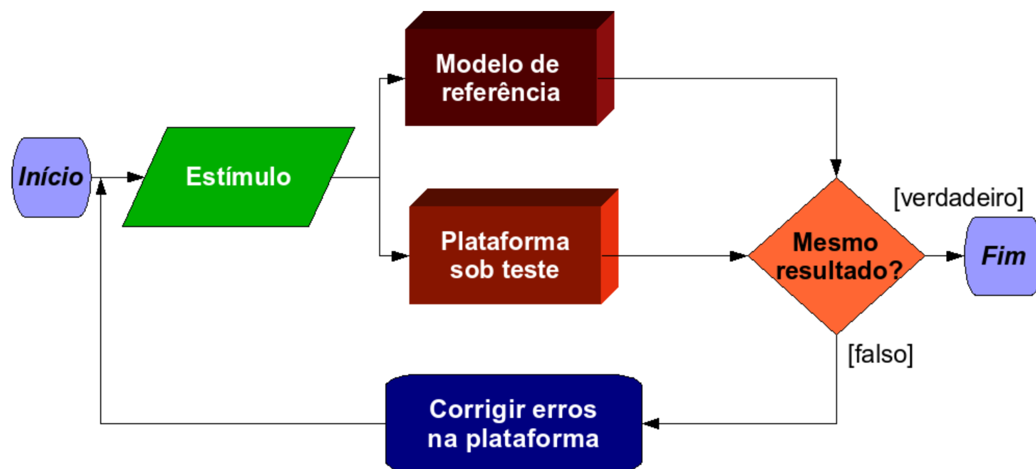

Figura 1: Metodologia de validação utilizada

com grande variabilidade, ou seja, aumentar a abrangência dos casos de teste.

3.2 INFRA-ESTRUTURA DE MODELAGEM DA PLATAFORMA

Segundo (GHENASSIA, 2005, p. 63), o primeiro passo para um fluxo de projeto e verificação à nível de sistema é o uso de modelos cuja comunicação foi modelada usando transações (TLM) - uma vez que essa abordagem fomenta a reusabilidade.

Para facilitar o reuso, deve ser fornecida uma infraestrutura responsável por organizar o código-fonte e o gerenciamento de diferentes tipos de componentes. É recomendável que cada projeto seja organizado em uma hierarquia definida por diretórios, de modo que componentes de tipos diferentes sejam alocados em diretórios diferentes.

Neste trabalho foi usada como base para a plataforma a infraestrutura fornecida pela *ArchC Reference Platform* (ARP) [1].

3.2.1 ARCHC REFERENCE PLATFORM - ARP

A ARP é um framework para a construção de modelos de plataformas que usam simuladores ArchC 2.0[2] (AZEVEDO et al., 2005). Para tanto, é fornecida uma estrutura básica composta de espaços reservados para componentes comuns em modelos de plataformas heterogêneas e *scripts* pra construir e simular essas plataformas.

[1]Mais informações em `http://www.archc.org` na seção *Platforms*.
[2]Mais informações em: `http://www.archc.org/`

IPs descritos em SystemC podem ser plugados na ARP e observados interagindo com outros componentes (processadores, memórias, outros IPs).

Plataformas seguindo o framework ARP são organizadas em diretórios, separando os diferentes tipos de componentes. Existem também diretórios de apoio, como para os scripts da ARP e documentação. Os principais são:

- **platforms** Armazena as configurações das plataformas criadas pelo usuário, incluindo seu arquivo executável.
- **processors** Para os simuladores gerados pelo ArchC.
- **is** Para estruturas de interconexão, como barramentos.
- **ip** Blocos de propriedade intelectual.
- **sw** Contém o código dos software embutido que rodará na plataforma.
- **wrappers** Armazena adaptadores entre os mais diversos tipos de componentes, e até mesmo, diferentes níveis de abstração.

3.2.2 MAPEAMENTO DA APLICAÇÃO DE COMPRESSÃO JPEG PARA UMA PLATAFORMA ARP

Como já foi mencionado no capítulo 1, foi desenvolvida uma plataforma (descrita em TLM) que realiza compressão de “bitmaps“ para arquivos no formato JPEG. O mapeamento da aplicação compressora em uma plataforma ARP é ilustrado na figura 2.

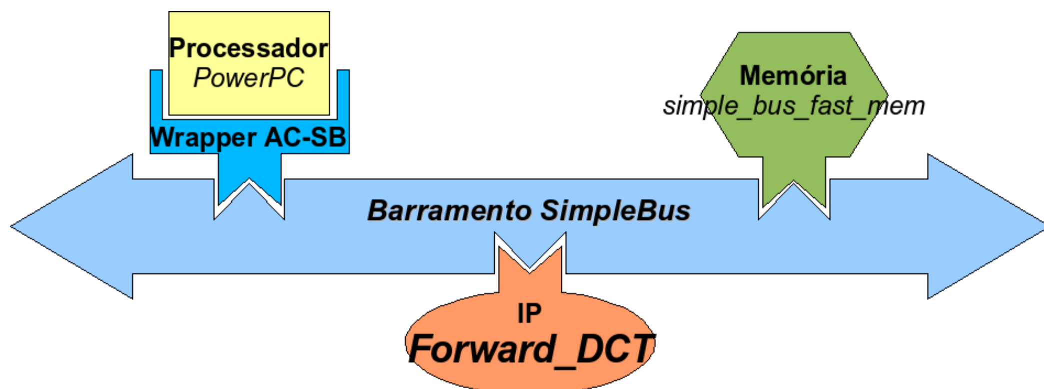

Figura 2: Diagrama em blocos da plataforma

A plataforma é composta de um processador PowerPC (cujo simulador foi obtido a partir de um modelo descrito em ArchC) interconectado pelo barramento SimpleBus à um bloco de

memória *simples_bus_fast_mem*, no qual o software de compressão JPEG compilado para esse processador, foi carregado. A descrição do processador não suporta a interface TLM do barramento SimpleBus, portanto, foi necessário um "wrapper" que faz a conversão das transações.

Também foi plugado à plataforma um IP que implementa parte da funcionalidade do software (mais detalhes no capítulo 4), cujos registradores são mapeados em memória. O software foi modificado para utilizar a funcionalidade do IP.

A configuração da plataforma descrita acima é especificada através do arquivo *defs.arp* dentro do diretório da plataforma na hierarquia da ARP, cujo conteúdo apresenta-se na tabela 1.

```
IP := sb_forward_DCT
IS := simple_bus_ua
PROCESSOR := powerpc
SW := cjpeg
WRAPPER := ac.simple_bus_ua.01
```

Tabela 1: Conteúdo do arquivo *defs.arp* mostrando a configuração da plataforma

4 PROJETO DO IP

Nesse capítulo trataremos do “profiling“ realizado no algoritmo de compressão JPEG, bem como da seleção da funcionalidade do IP que foi implementado, além da integração do mesmo numa plataforma TLM.

4.1 SELEÇÃO DO IP

Para definirmos a funcionalidade do software que o IP implementaria realizamos um “profiling“ do software de compressão JPEG incluído no benchmark MiBench usando a ferramenta gprof.

4.1.1 IMPLEMENTAÇÃO DE COMPRESSÃO JPEG NO MIBENCH

O MiBench[1] (GUTHAUS et al., 2001) é um benchmark gratuito organizado em grupos de programas (automotivo, rede, consumidor e outros). Dentro do grupo *consumidor* está a implementação do compressor JPEG (release 6a, de 07 de fevereiro de 1996, do “The Independent JPEG Group”). Essa implementação foi utilizada tanto para a definição do IP quanto para a validação (“golden model“).

4.1.2 A FERRAMENTA GPROF

O gprof é o profiler que compõe o *GNU Binary Utilities* (coleção de ferramentas utilizadas para manipulação de código em vários formatos mantido pela GNU). Com ele é possível verificar quantas vezes e quais funções estão sendo chamadas, além do tempo gasto em cada uma e a porcentagem correspondente ao total do tempo de execução, facilitando a identificação de gargalos de processamento no código.

[1] Mais informações em `http://www.eecs.umich.edu/mibench/`

4.1.3 “PROFILING“ E SELEÇÃO

O “profiling“ da aplicação de compressão JPEG foi realizado principalmente em cima das figuras 13, 14, 15 e 16, apresentadas no anexo A, pág. 38. As figuras possuem grande variação de tamanho e configuração (conforme tabela 11, do mesmo anexo), de modo a garantir que os resultados da análise não sejam facilmente influenciados por condições criadas a partir das características de uma determinada imagem.

As tabelas 2, 3, 4 e 5 mostram as funções mais chamadas durante a execução do aplicativo para algumas das figuras, resultado este extraído do “profiling“ realizado com o gprof. Já a tabela 6 mostra um sumário de todos os profilings realizados, e também foi gerado pelo gprof. O significado de cada coluna da tabela é mostrado a seguir:

- **% time:** A porcentagem de tempo usada pela função com relação ao tempo total.
- **self seconds:** O tempo que a função levou para executar.
- **calls:** O número de vezes que a função foi chamada.
- **self {s,ms,us}/call:** A média de tempo (em segundos, milissegundos ou microsegundos) gasta em cada chamada da função, não levando em conta o tempo gasto por seus descendentes (as funções chamadas por esta).
- **total {s,ms,us}/call:** A média de tempo (em segundos, milissegundos ou microsegundos) gasta em cada chamada da função, incluindo seus descendentes.
- **name:** O nome da função.

Analisando as tabelas, percebem-se quatro funções que se destacam como candidatas à terem suas funcionalidades implementadas em um IP. São elas: *rgb_ycc_convert*, *encode_mcu_huff*, *forward_DCT* e *jpeg_fdct_islow*. As duas primeiras são, a priori, boas candidatas, porém, analisando-se o fluxo de execução, percebe-se que as funções *forward_DCT* e *jpeg_fdct_islow* são, respectivamente, mãe e filha, e ainda que a função *jpeg_fdct_islow* é ’folha’, ou seja, não é chamada por mais ninguém.

Embutindo o código da função *jpeg_fdct_islow* dentro de *forward_DCT* obtemos uma função que consome aproximadamente 36% do tempo de processamento, o que a torna uma melhor candidata do que as duas funções anteriores. O único problema desta função é que ela é chamada muitas vezes, o que pode gerar um gargalo dependendo da velocidade do barramento,

Tabela 2: “Profiling“ do aplicativo cjpeg para a figura 13, 192Kb

% time	self seconds	calls	self ms/call	total ms/call	name
0.00	0.00	1536	0.00	0.00	jpeg_fdct_islow
0.00	0.00	1024	0.00	0.00	forward_DCT
0.00	0.00	625	0.00	0.00	emit_byte
0.00	0.00	384	0.00	0.00	expand_right_edge
0.00	0.00	271	0.00	0.00	pre_process_data

Tabela 3: “Profiling“ do aplicativo cjpeg para a figura 14, 768Kb

% time	self seconds	calls	self us/call	total us/call	name
50.00	0.01	6144	1.63	1.63	jpeg_fdct_islow
50.00	0.01	512	19.53	19.53	rcc_ycc_convert
0.00	0.00	4096	0.00	0.00	forward_DCT
0.00	0.00	1024	0.00	0.00	encode_mcu_huff
0.00	0.00	768	0.00	0.00	expand_right_edge

Tabela 4: “Profiling“ do aplicativo cjpeg para a figura 16, 20Mb

% time	self seconds	calls	self ms/call	total ms/call	name
38.99	0.23	3000	0.08	0.08	rgb_ycc_convert
27.12	0.16	28200	0.01	0.01	encode_mcu_huff
18.65	0.11	112650	0.00	0.00	forward_DCT
6.78	0.04	168900	0.00	0.00	jpeg_fdct_islow
5.09	0.03	3000	0.01	0.01	h2v2_downsample

Tabela 5: “Profiling“ do aplicativo cjpeg para a figura 16, 183Mb

% time	self seconds	calls	self ms/call	total ms/call	name
29.94	1.44	1000000	0.00	0.00	forward_DCT
27.86	1.34	250000	0.01	0.01	encode_mcu_huff
20.58	0.99	8000	0.12	0.12	rgb_ycc_convert
17.26	0.83	1500000	0.00	0.00	jpeg_fdct_islow
3.53	0.17	8000	0.02	0.02	h2v2_downsample

Tabela 6: Sumário dos “profilings“ realizados sobre o aplicativo cjpeg

% time	self seconds	calls	self ms/call	total ms/call	name
31.19	2.96	22844	0.13	0.13	rgb_ycc_convert
28.03	2.66	501213	0.01	0.01	encode_mcu_huff
24.76	2.35	2004420	0.00	0.00	forward_DCT
11.59	1.10	3005851	0.00	0.00	jpeg_fdct_islow
3.48	0.33	22844	0.01	0.01	h2v2_downsample

já que um grande número de transações é necessário. Essa potencial desvantagem, porém, é superada pela maior perspectiva de reuso, pois esta função implementa uma transformada discreta do cosseno bi-dimensional - DCT-2D -, que é usada para várias aplicações, principalmente do tipo multimídia, o que aumenta as chances de reusabilidade do IP. Por isso, escolheu-se a função *forward_DCT* e sua filha *jpeg_fdct_islow* para serem aceleradas em hardware, na forma de um bloco de propriedade intelectual.

4.2 MODELAGEM FUNCIONAL DO IP NA PLATAFORMA TLM

Como mencionado na seção 4.1.3, a função *forward_DCT* foi escolhida para implementação. Esta função faz um pré-processamento nos valores de entrada, chama o método *jpeg_fdct_islow*, para efetivamente realizar a DCT, e finalmente faz a quantização e diminuição de escala dos coeficientes da DCT. O protótipo da função é o seguinte:

```
      forward_DCT (j_compress_ptr cinfo, jpeg_component_info
*compptr, SAMPARRAY sample_data, JBLOCKROW coef_blocks,
JDIMENSION start_row, JDIMENSION start_col, JDIMENSION
num_blocks);
```

Os parâmetros recebidos são, em geral, ponteiros para estruturas de dados nas quais os dados a serem processados se encontram. Uma explicação mais detalhada sobre estes se encontra na seção 4.2.2.

A função *forward_DCT* foi transformada num módulo SystemC, conforme mostra a figura 3. A figura mostra as entradas e saídas e também o mapeamento de E/S em memória do módulo, que serão mostrados em detalhes nas seções 4.2.1 e 4.2.2, respectivamente. O bloco FDCT-2D, que será transformado em IP será demonstrado na seção 4.4, pág. 30.

4.2.1 ENTRADAS E SAÍDAS DO MÓDULO FORWARD_DCT

O módulo *forward_DCT* possui um total de 64 pinos. Todos são utilizados tanto para entrada como para saída, e são divididos em dois sinais de 32 pinos cada, conforme demonstram a tabela 7 e a figura 3.

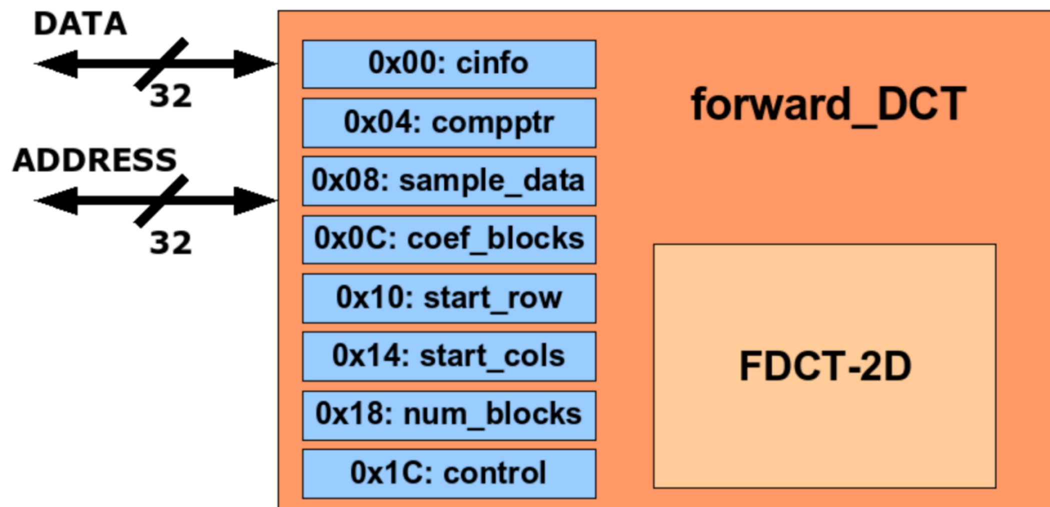

Figura 3: Módulo forward_DCT

Tabela 7: Entradas e saídas do módulo forward_DCT

Sinal	Pinos	Descrição
DATA	32	Um valor lido da memória ou um valor a ser escrito nela
ADDRESS	32	O endereço que deverá ser lido ou escrito na memória

4.2.2 MAPEAMENTO DE E/S EM MEMÓRIA DO MÓDULO FORWARD_DCT

O módulo *forward_DCT* possui oito registradores de 32 bits cada. Sete destes têm relação direta com a função *forward_DCT*, apresentada acima, pois representam os parâmetros da mesma. Já o último é um registrador de controle do módulo. Segue abaixo o nome de cada registrador, seu respectivo tipo - (E) para entrada, (S) para saída e (E/S) para entrada e saída -, e uma breve descrição:

- **cinfo (E):** Ponteiro para uma "struct" do tipo *j_compress_ptr* que contém diversas informações utilizadas durante a compressão da imagem, como: largura e altura da imagem, número de componentes de cor, parâmetros relacionados à otimização e qual o método de DCT desejado, que é o dado que realmente importa para nós.

- **compptr (E):** Ponteiro do tipo *jpeg_component_info** para uma estrutura que contém informações relacionadas à um componente (canal de cor) da imagem tais como: identificador do componente, número de amostras de um bloco de DCT e um ponteiro para a tabela de quantização que será utilizada após o cálculo da DCT.

- **sample_data (E):** Ponteiro para array bidimensional, tipo *JSAMPARRAY*, com as amostras de entrada.
- **coef_blocks (E/S):** Ponteiro do tipo *JBLOCKROW* para um bloco de coeficientes. Também é usado como saída do módulo, onde os coeficientes de resultado da transformada discreta do cosseno já quantizados e em sua escala normal são colocados.
- **start_row (E):** Inteiro que representa a linha inicial, na matriz *sample_data*, do bloco de valores para os quais será calculado a DCT.
- **start_col (E):** Inteiro que representa a coluna inicial, na matriz *sample_data*, do bloco de valores para os quais será calculado a DCT.
- **num_blocks (E):** Número de elementos que compõem a linha de um bloco.
- **control (E/S):** Variável inteira de controle. Deve valer 1 quando o processamento deve iniciar. Após o término do processamento, o próprio módulo irá setá-la com valor 0, indicando o fim do processamento.

Como os registradores são mapeados em memória, para cada registrador foi alocado um determinado endereço. Os “offsets“ podem ser observados na tabela 8. Tendo em vista que são 8 registradores de 32 bits cada, a faixa de endereços (o endereço do último registrador menos o endereço-base) reservada para o IP não pode ser menor do que 0x20.

Tabela 8: Registradores do IP - “Offsets“ com relação ao endereço-base e tipo

Endereço	Registrador
0x00	cinfo
0x04	compptr
0x08	sample_data
0x0C	coef_blocks
0x10	start_row
0x14	start_col
0x18	num_blocks
0x1C	control

O funcionamento do módulo se dá da seguinte forma: todos os registradores (com exceção do registrador control) são carregados com seus respectivos valores. A carga dá-se colocando escrevendo o valor desejado no respectivo endereço (o que se traduz em colocar o valor no sinal DATA e o endereço no sinal ADDRESS), e geralmente será feita pelo software que chamará o módulo. Após a carga de todos os registradores, o registrador control é setado com valor 1 e o módulo inicia o processamento. Assim que ele terminar seu trabalho, o valor de control é

setado pelo próprio módulo para 0, e o software que ficava aguardando este valor via “pooling“ pode continuar.

4.2.3 MODELAGEM DAS TRANSAÇÕES

O IP desenvolvido precisa atuar tanto como mestre quanto como escravo do barramento ao qual está conectado (SimpleBus). Isso porque o processador escreverá valores nos registradores do IP (IP atua como escravo) e o IP lerá e escreverá valores na memória como resultado de seu processamento (IP atua como mestre).

IP COMO ESCRAVO DO BARRAMENTO

Para que o IP atue como escravo do barramento, o modelo foi declarado como uma subclasse de *simple_bus_slave_if*, cuja declaracao aparece na figura 4.

```
 1class simple_bus_slave_if
 2  : public simple_bus_direct_if
 3{
 4    public:
 5        virtual simple_bus_status read(int *data, unsigned int address) = 0;
 6        virtual simple_bus_status write(int *data, unsigned int address) = 0;
 7
 8        virtual unsigned int start_address() const = 0;
 9        virtual unsigned int end_address() const = 0;
10};
```

Figura 4: A interface *simple_bus_slave_if*

A classe *simple_bus_slave_if* contém um conjunto de funções puramente virtuais que precisam ser implementadas por componentes que devem atuar como escravos do barramento SimpleBus.

As funções *start_address* e *end_address* retornam o endereço base e o endereço final, respectivamente, da faixa de memória aonde os registradores do IP foram mapeados.

As funções *read* e *write* são utilizadas pelo barramento SimpleBus para ler e escrever dados nos registradores do IP. Elas recebem como parâmetros um ponteiro para uma área de dados e um endereço de memória.

A implementação feita para o IP da função *read* pode ser vista na figura 5. A idéia é simples, primeiro se descobre de qual registrador de dados os dados devem ser lidos (através

do argumento *address*) e depois copia-se o valor desse registrador para a área apontada pelo ponteiro *data*.

```
 1simple_bus_status Forward_DCT::read(int *data, unsigned int address)
 2{
 3    switch (address - m_start_address)
 4    {
 5        case 0x00:
 6            (*data) = (int) cinfo;
 7        break;
 8        case 0x04:
 9            (*data) = (int) compptr;
10        break;
11        case 0x08:
12            (*data) = (int) sample_data;
13        break;
14        case 0x0C:
15            (*data) = (int) coef_blocks;
16        break;
17        case 0x10:
18            (*data) = (int) start_row;
19        break;
20        case 0x14:
21            (*data) = (int) start_col;
22        break;
23        case 0x18:
24            (*data) = (int) num_blocks;
25        break;
26        case 0x1C:
27            (*data) = (char) control;
28        break;
29        default:
30            return SIMPLE_BUS_ERROR;
31        break;
32    }
33    return SIMPLE_BUS_OK;
34}
```

Figura 5: Implementação da função *read*

No caso da função write (cuja implementação é mostrada na figura 6, os dados são copiados da área apontada pelo ponteiro *data* para o registrador (selecionado através do argumento *address*). Além disso, caso o registrador de controle receba o valor 1, o evento *wakeup* é notificado, o que faz com que o IP inicie o seu processamento.

```
simple_bus_status Forward_DCT::write(int *data, unsigned int address)
{
    switch (address - m_start_address)
    {
        case 0x00:
            cinfo = (j_compress_ptr) (*data);
        break;
        case 0x04:
            compptr = (jpeg_component_info *) (*data);
        break;
        case 0x08:
            sample_data = (JSAMPARRAY) (*data);
        break;
        case 0x0C:
            coef_blocks = (JBLOCKROW) (*data);
        break;
        case 0x10:
            start_row = (JDIMENSION) (*data);
        break;
        case 0x14:
            start_col = (JDIMENSION) (*data);
        break;
        case 0x18:
            num_blocks = (JDIMENSION) (*data);
        break;
        case 0x1C:
            control = (char) (*data);
        break;
        default:
            return SIMPLE_BUS_ERROR;
        break;
    }

    if (control)
    {
        wakeup.notify();
    }

    return SIMPLE_BUS_OK;
}
```

Figura 6: Implementação da função *write*

IP COMO MESTRE DO BARRAMENTO

Para que o IP atue como mestre do barramento, é necessário adicionar ao modelo um atributo do tipo *sc_port<simple_bus_blocking_if>*.

Posteriormente, na hora de instanciar a plataforma, esse atributo precisa ser 'ligado' a um canal de comunicação (que no caso será o barramento SimpleBus) que implemente as funci-

onalidades descritas pela interface *simple_bus_blocking_if* - cujo código é mostrada na figura 7.

```
 1class simple_bus_blocking_if : public virtual sc_interface
 2{
 3    public:
 4        virtual simple_bus_status burst_read(unsigned int unique_priority
 5                              , int *data
 6                              , unsigned int start_address
 7                              , unsigned int length = 1
 8                              , bool lock = false) = 0;
 9        virtual simple_bus_status burst_write(unsigned int unique_priority
10                           , int *data
11                           , unsigned int start_address
12                           , unsigned int length = 1
13                           , bool lock = false) = 0;
14};
```

Figura 7: A interface *simple_bus_blocking_if*

4.3 A INSTANCIAÇÃO DE COMPONENTES NA PLATAFORMA

Parte do arquivo *main.cpp* - onde está o código para instanciar a plataforma TLM, é mostrado na figura 8.

Nas linhas 7 e 10 são instanciados o processador PowerPC e o "wrapper" que torna possível a conexão com o barramento SimpleBus, respectivamente.

Nas linhas 13 e 14 são instanciados o barramento SimpleBus e também o árbitro do mesmo.

Nas linhas 17, 18 e 19 são instanciados dois blocos de memória e também o IP que implementa a função $forward_DCT$. Dentre os argumentos passados para os construtores estão o nome dos componentes e também as faixa de endereços reservados no espaço de endereçamento.

A partir da linha 22, são feitas as conexões entre os componentes da plataforma. O *clock* e o árbitro são atribuídos ao barramento. Os dois blocos de memória e o IP são conectados em portas de escravo do barramento.

Na linha 28, o atributo do tipo *sc_port<simple_bus_blocking_if>* do IP(que é utilizado para que ele atue como mestre do barramento) é conectado ao barramento.

Na linha 32, o software (previamente compilado) é carregado na memória. Depois o processador é inicializado. A linha 39 é responsável pelo início da simulação.

```
int sc_main(int ac, char *av[])
{
  // Bus clock
  sc_clock bus_clock;

  // Processor instance
  powerpc powerpc_proc1("powerpc");

  // Wrapper that maps memory with offset
  w_sb wrap1("wrapper1",1, 0x000000, false, 0);

  // Bus instance
  simple_bus bus("bus",false);
  simple_bus_arbiter arbiter("arbiter");

  // Address space
  simple_bus_fast_mem mem_fast1("mem_fast1", 0x000000, 0x2FFFFF);
  Forward_DCT forward_DCT_ip("forward_DCT_ip", 2, 0x300000, 0x3FFFF);
  simple_bus_fast_mem mem_fast2("mem_fast2", 0x400000, 0x9FFFFF);

  // Bindings
  bus.clock(bus_clock);
  bus.arbiter_port(arbiter);
  bus.slave_port(mem_fast1);
  bus.slave_port(mem_fast2);
  bus.slave_port(forward_DCT_ip);
  wrap1.bus_port(bus);
  forward_DCT_ip.bus_port(bus);
  powerpc_proc1.MEM_port(wrap1.target_export);

  // Load elf before start
  load_elf(powerpc_proc1, mem_fast1, "cjpeg.x",0x000000);

  powerpc_proc1.set_instr_batch_size(1);

  // Prepare processors
  powerpc_proc1.init();

  sc_start();

  powerpc_proc1.PrintStat();

  return powerpc_proc1.ac_exit_status;
}
```

Figura 8: Trecho do código de main.cpp

4.4 MODELAGEM RTL DO IP

Após análise mais detalhada do IP em nível funcional, percebemos que o mesmo era específico demais. Todos os registradores de entrada estão intimamente relacionados com o código do JPEG, e além disso as funcionalidades do IP vão além do cálculo da DCT, devido ao pré-processamento e a quantização também realizados. A parte do IP que realmente implementa uma DCT-2D genérica é a parte equivalente ao código da função $jpeg_fdct_islow$.

Após pesquisa sobre o funcionamento de uma DCT bi-dimensional e comparação de algumas arquiteturas propostas na literatura (BHASKARAN; KONSTANTINIDES, 1999), optamos pela arquitetura apresentada na figura 9, proposta por (AGOSTINI, 2002). Esta baseia-se numa propriedade desta transformada, o princípio da separabilidade, que mostra que o cálculo pode ser realizado através de duas 2 DCTs unidimensionais, a primeira sobre as linhas da matrix $8x8$ e a segunda sobre as colunas da matriz resultante da primeira. O componente denominado “buffer de transposição“ é quem permitirá que a segunda transformada opere sobre as colunas da matriz.

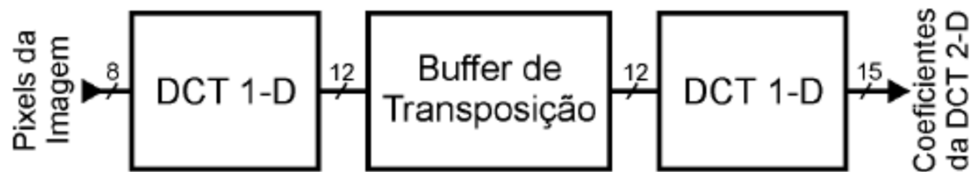

Figura 9: Arquitetura RTL do IP.
Fonte: (AGOSTINI, 2002)

A modelagem em RTL da DCT-2D está em andamento. Mais informações podem ser encontradas na seção 6.1.

5 RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados experimentais e a configuração necessária para reproduzí-los.

5.1 CONFIGURAÇÃO EXPERIMENTAL

O computador utilizado para a obtenção dos resultados aqui demonstrados possui a seguinte configuração: processador Intel Core Duo, 1.66GHz de frequência, cache L2 de 2 MB e 667MHz de FSB; memória principal de 2 GB; e roda sistema operacional GNU/Linux Ubuntu 7.04, kernel 2.6.20 SMP.

As ferramentas usadas durante o processo, e suas respectivas versões, estão listadas abaixo:

- **ArchC** , versão 2.0;
- **SystemC** , versão 2.2.0;
- **SystemC TLM** , versão 1.0 (2005-04-08);
- **Compilador GCC** , versão 4.1.2;
- **Cross-compiler GCC para PowerPC** , versão 3.1;
- **cjpeg** , versão 6a (07 de fevereiro de 1996), do The Independent JPEG Group - IJG.

Além disso, o modelo funcional de PowerPC integrado na plataforma, conforme seção 3.2.2 é a versão 0.7.3, para o ArchC 2.0. O software de compressão JPEG que roda em cima deste modelo foi compilado com o “cross-compiler“. Tanto o modelo do processador quanto o compilador cruzado estão disponíveis na página do projeto ArchC.

5.2 VALIDAÇÃO DO IP FUNCIONAL NA PLATAFORMA

A metodologia utilizada para a validação do IP foi descrita na seção 3.1. Os seguintes insumos foram utilizados:

- **Estímulos** : Figuras 11, 12, 13, 14, 15, 17.
- **“golden model“** : Implementação para compressão JPEG do benchmark MiBench.
- **Utilitário *diff*** : Utilitário que compara dois arquivos bit a bit.

Para cada uma das figuras listadas, a plataforma e o “golden model“ foram executados. Os arquivos de saída (imagens comprimidas em JPEG) foram comparados com o utilitário *diff* - o que garante que as saídas são rigorosamente iguais. Várias imagens foram utilizadas para garantir uma certa variabilidade dos vetores de entrada, aumentando, assim, a confiabilidade do processo de validação.

5.3 COMPARAÇÃO DE TEMPOS DE EXECUÇÃO ENTRE PLATAFORMA E SOFTWARE

Naturalmente, a execução da plataforma para comprimir imagens leva muito mais tempo do que a execução de um software qualquer que realiza a mesma função. Entretanto, achamos interessante mensurar de quantas ordens de grandeza é essa diferença.

A tabela 9 compara os tempos de execução da plataforma desenvolvida com os tempos do software cjpeg. Por exemplo, para a figura mibench_large, a execução da plataforma levou 132,94 segundos, enquanto que a execução do software levou apenas 0,020 segundos.

Índice da Figura	Nome da Imagem	Tempo Plataforma	Tempo Software
11	boy	36.23s	0.012s
12	jpeg_testimg	21.16s	0.008s
15	mediabench	196.71s	0.052s
14	mibench_large	132.94s	0.020s
13	mibench_small	36.86s	0.008s
17	ufsc_ctc	283.34s	0.052s
SOMATÓRIO		707.24s	0.152s
TOTAL		707.572s	

Tabela 9: Comparação de tempos de execução: plataforma *x* software

Como se pode perceber, o tempo de execução da plataforma é muito superior, algo em torno de 3 à 4 ordens de grandeza. Buscamos, então, identificar o porquê dessa diferença tão grande. Para tanto, executamos um “profiling“ da execução da plataforma desenvolvida sobre as mesmas figuras da tabela 9 e, finalmente, um sumário desses seis “profilings“, de maneira análoga ao procedimento descrito na seção 4.1.3.

Parte do resultado deste “profiling“ está demonstrado na tabela 10, que relaciona as cinco funções mais chamadas durante as simulações com suas respectivas porcentagens sobre o tempo total e o tempo gasto em cada uma. Analisando a tabela, percebe-se que estas funções, sozinhas, são responsáveis por mais da metade do tempo de processamento. Também pode-se notar que a maior parte do tempo consumido pelas mesmas está relacionado com o SystemC, o que pode ser notado pelo *namespace* sc_core no início do nome das funções.

Porcentagem do tempo	Tempo gasto (s)	Nome da Função
19.74	107.58	sc_core::sc_event::trigger()
18.01	98.16	sc_core::sc_simcontext::crunch()
7.29	39.74	sc_core::sc_simcontext::simulate(sc_core::sc_time const&)
3.83	20.89	powerpc::behavior()
3.07	16.71	sc_core::wait(sc_core::sc_time const&,sc_core::sc_simcontext*)
51.94	283.08	*TOTAL*

Tabela 10: Sumário dos “profilings“realizados durante execução da plataforma

Realizando esta separação por *namespaces* em todo o sumário, pudemos chegar ao gráfico da figura 10. Ele nos mostra como o tempo gasto na execução na plataforma está distribuído.

Como pode-se perceber, em torno de 74% do tempo é gasto com funções do “kernel“ de simulação e componentes da biblioteca do SystemC. Chamadas a funções do modelo do barramento SimpleBus ficam em torno de 11%. Outros 5% são gastos na simulação do processador PowerPC (mais 2% são gastos com o “wrapper“do processador). O resto do tempo (representado pela categoria *Outros*) é gasto com a simulação do IP $forward_DCT$, biblioteca STL do C++ além de um pequeno overhead relacionado ao uso de TLM.

Esses resultados deixam claro que existe muito a ser feito no que diz respeito a otimizações na execução das simulações, dado que a maior parte do tempo é gasta para se prover a infraestrutura necessária para a execução do código dos modelos.

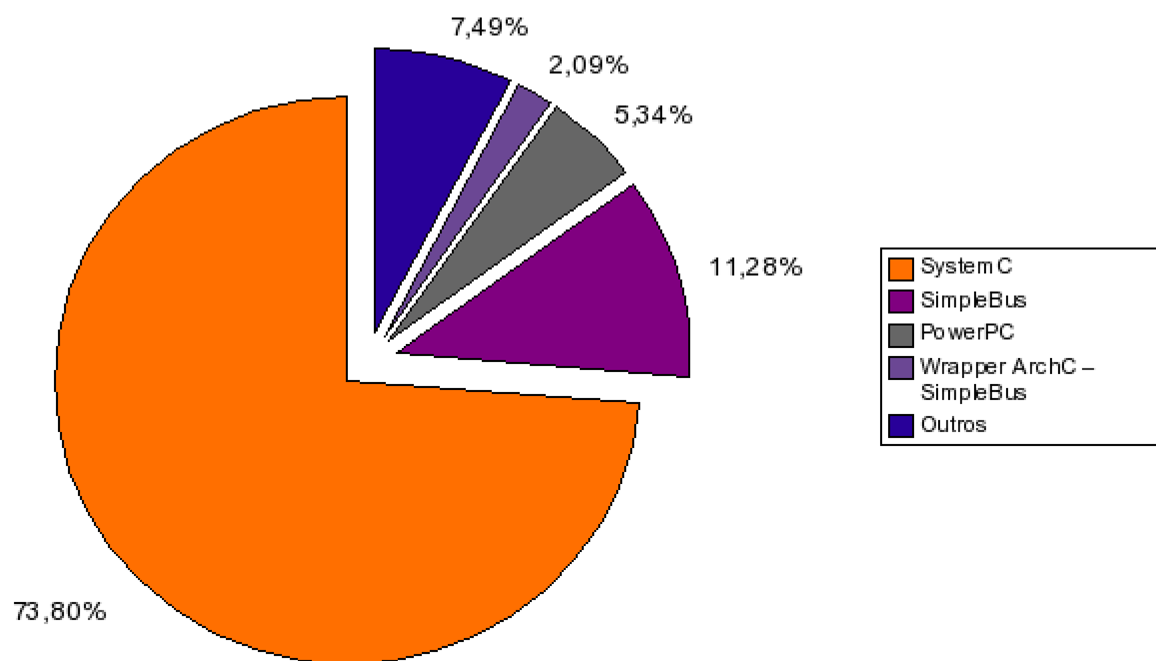

Figura 10: Distribuição do tempo gasto na execução da plataforma

6 *CONCLUSÃO*

A utilização de modelagem em nível transacional parece indubitavelmente fundamental para que se lide com a crescente complexidade dos sistemas projetados e prazos cada vez mais apertados. TLM parece ter identificado o nível correto de abstração para que se possa realizar o co-projeto de hardware e software, uma vez que a modelagem toma pouco tempo - se comparado ao tempo total do projeto, e mesmo assim os resultados são precisos o suficiente para que o desenvolvimento de software possa ser antecipado, executando-o em um modelo supostamente preciso do futuro hardware.

O uso se TLM se torna particularmente interessante quando fazemos uso de outra abordagem ganhando espaço na indústria - o projeto baseado em plataforma. Modelar a comunicação como transações torna mais fácil integrar novos componentes a um projeto genérico - uma plataforma, de modo que a exploração do espaço do projeto é favorecida devido ao grande número de variações que podem ser testadas.

Juntas, as duas técnicas mencionadas proporcionam um significativo ganho de produtividade, conforme pode ser comprovado durante o desenvolvimento do IP e também da sua integração na plataforma descrita em TLM.

6.1 TRABALHOS FUTUROS

Como trabalhos futuros, objetivamos chegar à uma descrição RTL do IP proposto em 4.4. Para atingirmos este objetivo, modelaremos cada componente descrito na arquitetura da figura 9 também em nível de transferência de registradores.

Para tanto, cada componente passará pelo mesmo processo de projeto e validação descrito neste trabalho. Primeiramente será gerado um “golden model“ e um IP descrito em nível funcional para cada bloco. As saídas dos dois serão confrontadas para que o IP seja considerado validado. E então serão estudadas possíveis arquiteturas e a que apresentar as características mais atraentes será escolhida para descer o nível de cada componente para RTL.

O maior grau de dificuldade estará na implementação da DCT *1D*, que envolverá uma grande quantidade de somas e multiplicações. Portanto, deverá ser escolhida uma arquitetura eficiente em velocidade de processamento - dado que boa parte das aplicações candidatas à uso do IP são do tipo multimídia, como compressores de aúdio e vídeo, e impõe restrições de tempo real -, mas não deixando de lado a preocupação com o tamanho de área utilizado.

REFERÊNCIAS

AGOSTINI, L. V. *Projeto de Arquiteturas Integradas para a Compressão de Imagens JPEG*. Dissertação (Mestrado) — UFRGS, mar 22 2002.

AGOSTINI, L. V.; SILVA, I. S.; BAMPI, S. Pipelined fast 2-D DCT architecture for JPEG image compression. 2001.

ARP. *ArchC Reference Platform*. Disponível em: <http://www.archc.org/>.

AZEVEDO, R. et al. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, Kluwer Academic Publishers, v. 33, n. 5, p. 453–484, October 2005. ISSN 0885-7458.

BHASKARAN, V.; KONSTANTINIDES, K. *Image and Video Compression Standards*: Algorithms and architectures. [S.l.]: Kluwer Academic Publishers, 1999.

BLACK, D.; DONOVAN, J. *SystemC From the ground up*. [S.l.]: Springer, 2004.

GHENASSIA, F. *Transaction-Level Modeling with SystemC*: TLM concepts and applications for embedded systems. [S.l.]: Springer, 2005.

GUTHAUS, M. et al. MiBench: A free, commercially representative embedded benchmark suite. In: . [S.l.: s.n.], 2001. p. 3–14.

LEE, C.; POTKONJAK, M.; MANGIONE-SMITH, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: *International Symposium on Microarchitecture*. [S.l.: s.n.], 1997. p. 330–335.

MARTIN, G.; CHANG, H. *Winning the SOC Revolution*: Experiences in real design. [S.l.]: Kluwer Academic Publishers, 2003.

MARWEDEL, P. *Embedded system design*. [S.l.]: Springer, 2006.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and challenges for platform-based design. In: *Proceedings of the 41st Annual conference on Design Automation (DAC-04)*. New York: ACM Press, 2004. p. 409–414.

SANGIOVANNI-VINCENTELLI, A. L.; MARTIN, G. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, v. 18, n. 6, p. 23–33, 2001.

ANEXO A – IMAGENS UTILIZADAS PARA “PROFILING“ E VALIDAÇÃO

Para a realização do “profiling“ foram utilizadas diversas imagens, das quais as principais são apresentadas aqui. Em sua maioria, estas foram retiradas de benchmarks como MiBench e MediaBench. Cabe ressaltar que algumas das imagens apresentadas abaixo tiveram seus tamanhos alterados para melhorar sua apresentação neste documento.

As configurações das imagens utilizados são apresentadas na tabela 11. A figura 16 possui diversas instâncias, com configurações diferentes, de modo a melhorar a acurácia do “profiling“.

Figura	Nome	Formato	Largura (px)	Altura (px)	Tamanho aprox.
11	boy	PPM	200	281	164.7 KB
12	jpeg_testimg	PPM	227	149	99.1 KB
13	mibench_small	PPM	256	256	192 KB
14	mibench_large	PPM	512	512	768 KB
15	mediabench	PPM	704	576	1,2 MB
16	nasa	PPM	2400	3000	20,6 MB
16	nasa	PPM	4500	4500	57,9 MB
16	nasa	PPM	6000	6000	103 MB
16	nasa	PPM	8000	8000	183,1 MB
17	ufsc_ctc	PPM	800	600	1,4 MB

Tabela 11: Configuração das figuras utilizadas para “profiling“ e validação

Figura 11: Imagem “boy“, em tons de cinza

Figura 12: Imagem “jpeg_testimg“ (que acompanha o software cjpeg)

Figura 13: Imagem “mibench_small“ (contida no benchmark MiBench)

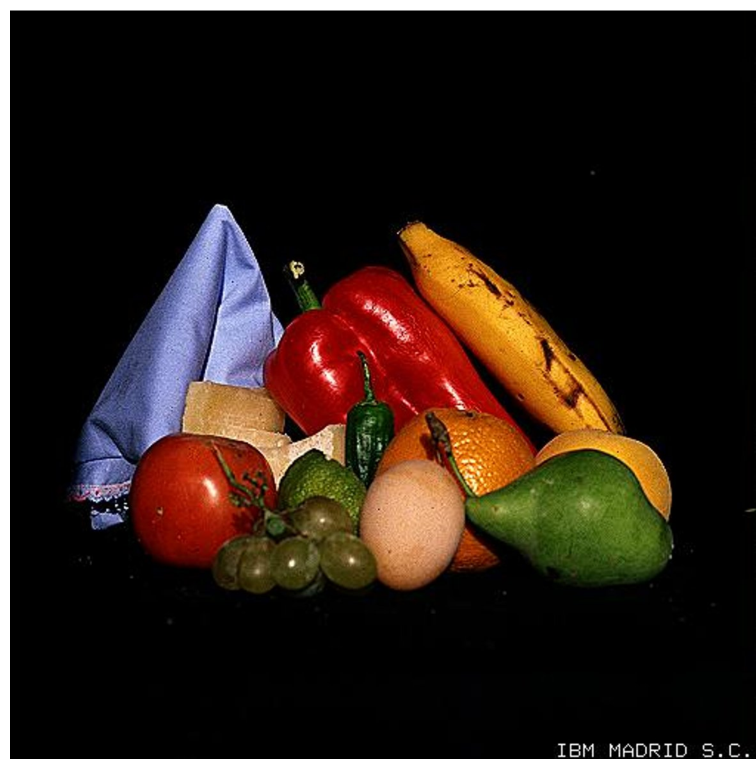

Figura 14: Imagem “mibench_large“ (contida no benchmark MiBench)

Figura 15: Imagem “mediabench“ (contida no benchmark MediaBench)

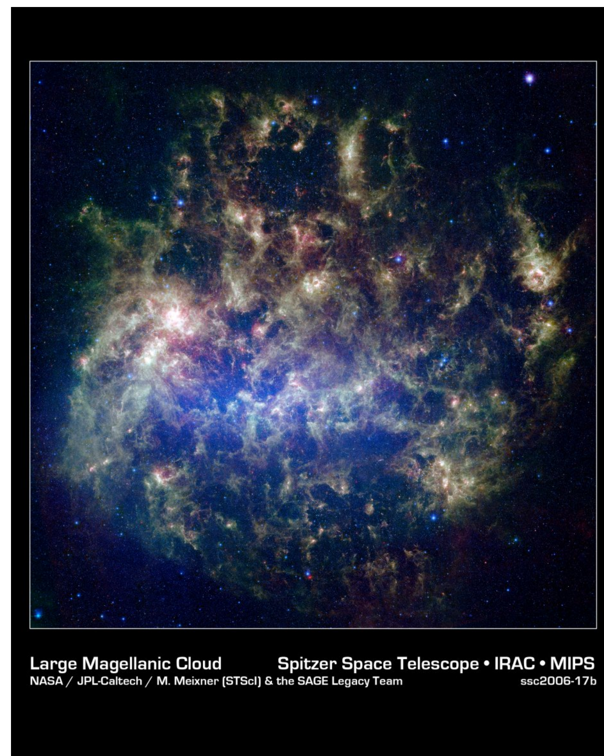

Figura 16: Imagem “nasa“

Figura 17: Imagem “ufsc_ctc“

APÊNDICE A – MANUAL DO IP FORWARD_DCT

A seguinte documentação foi gerada durante o desenvolvimenteo do IP Forward-DCT

```
******************************************************************
                           MANUAL IP forward_DCT
******************************************************************

Autores: Daniel P. Volpato (danielpv@inf.ufsc.br)
         Leonardo L. Ecco  (ecco@inf.ufsc.br)


Conteúdo
========

1. Função implementada
  1.1. Comportamento
  1.2. Origem do golden-model
2. Entradas
3. Saídas
4. Mapeamento de I/O em memória
5. Modificações no software aplicativo
6. Status da Validação
7. Release notes
8. Sugestões de melhoria
```

1. Função implementada
==

1.1. Comportamento

A função escolhida foi a

```
   forward_DCT (j_compress_ptr, jpeg_component_info *, SAMPARRAY,
        JBLOCKROW, JDIMENSION, JDIMENSION, JDIMENSION)
```

Essa função não é folha, porém chama uma função folha, que é determinada dependendo do argumento -dct passado como parâmetro de linha de comando.

Para o IP rodar, o software deve ser modificado para sempre considerar o argumento "-dct int", ou então, passar esse argumento como parâmetro na inicialização do software.

1.2. Origem do golden-model

O código utilizado para desenvolvimento desse IP provém do benchmark Mibench.

A implementação do compressor JPEG utilizada no Mibench é o release 6a (07/Fev/96) do JPEG software do The Independent JPEG Group (IJG)

2. Entradas
==

O IP possui as seguintes entradas:

```
  * j_compress_ptr cinfo
  * jpeg_component_info * compptr
  * JSAMPARRAY sample_data
  * JBLOCKROW coef_blocks
  * JDIMENSION start_row
  * JDIMENSION start_col
  * JDIMENSION num_blocks



3. Saídas
========================================

A matriz apontada por coef_blocks fornece endereços de estruturas de
dados onde o resultado do processamento será escrito.



4. Mapeamento de I/O em memória
========================================

A faixa de endereço usada pelo IP é dada pelos endereços base e final
fornecidos através do construtor do IP.

Essa faixa não deve ser menor do que 0x20, que é o espaço ocupado
pelos registradores do IP.

Segue abaixo o offset de cada um dos registradores do IP:

    0x00 : cinfo
    0x04 : compptr
    0x08 : sample_data
    0x0C : coef_blocks
    0x10 : start_row
    0x14 : start_col
```

```
    0x18 : num_blocks
    0x1C : control



5. Modificações no software aplicativo
========================================

Arquivo:    jcdctmgr.c
Função:     forward_DCT(...)
Descrição: O código da função foi substituído pelos seguintes passos:
  - Setar os registradores de entrada do IP com os valores corretos;
  - Setar o registrador de controle "control" para 1.
  - Busy waiting no registrador de controle até que seu valor seja 0.

Arquivo:    cjpeg.c
Função:     main (int argc, char **argv)
Descrição: Modificada para forçar a seguinte linha de comando:
   "-dct int -opt -outfile output.jpeg input.ppm",
   o que habilita o uso do IP.



6. Status da Validação
========================================

Foram usados como estímulos para o IP as imagems no formato PPM que
vieram junto com os benchmarks mibench e mediabench.

Para verificar se o IP codificou a imagem para o format JPEG
corretamente, o arquivo gerado pelo IP é comparado com um arquivo gerado
pelo software cjpeg (que recebeu o mesmo PPM de entrada).

A comparação foi feita utilizando o 'diff'.
```

```
7. Release notes
========================================

* sb_forward_dct:
  - Barramento: Simple Bus
  - Descrição: Funciona como master e slave do barramento simple bus.
    Validada contra o golden model.

* ahb_forward_dct:
  - Barramento: Simple Bus
  - Descrição: (Ainda) Não atua como master do sistema, portanto não é
    capaz de realizar o processamento.



8. Sugestões de melhoria
========================================
  - Modificar AMBA para usar as funções definidas pela OSCI Core TLM
    Interface.
  - Documentação do AMBA deixa a desejar.
  - Toy examples demonstrando como fazer um IP atuar como slave e como
    master do AMBA.
```

APÊNDICE A – CÓDIGO-FONTE

forward_DCT.h

```
*********************************************************************

/**
 * @file      forward_DCT.h
 * @author    Daniel Pereira Volpato
 * @author    Leonardo Luiz Ecco
 */

#ifndef FORWARD_DCT_H_
#define FORWARD_DCT_H_

#include <systemc.h>

#include "simple_bus_types.h"
#include "simple_bus_blocking_if.h"
#include "simple_bus_slave_if.h"

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

// required for function jdct_islow
#include "jdct.h"

class Forward_DCT : public sc_module, public simple_bus_slave_if
{
```

```
public:
    /// Bus master port
    sc_port<simple_bus_blocking_if> bus_port;

    /// Wake up IP action
    sc_event wakeup;

    SC_HAS_PROCESS(Forward_DCT);

    /**
     * Destructor
     **/
    ~Forward_DCT();

    /**
     * Constructor
     * @param unique_priority To be passed to bus actions (read/write)
     * @param start_address Start address of IP
     * @param end_address End address of IP
     **/
    Forward_DCT(sc_module_name name_, unsigned int unique_priority,
            unsigned int start_address, unsigned int end_address);

    // Slave direct interface
    // Not used, just for Simple Bus compatibility
    bool direct_read(int *data, unsigned int address);
    bool direct_write(int *data, unsigned int address);

    // Slave blocking interface
    // To act like a slave
    simple_bus_status read(int *data, unsigned int address);
    simple_bus_status write(int *data, unsigned int address);

    // Functions to comply with SimpleBus standard
    unsigned int start_address() const;
```

```
    unsigned int end_address() const;

    // Main IP thread, holds all functionality
    void main_action();

    // Aiding function to act like master
    // All those are made on bus, not locally
    uint32_t rmem32u(unsigned int address);
    uint8_t rmem8u(unsigned int address);
    void wmem8u(unsigned int address, uint8_t data);
    void wmem16u(unsigned int address, uint16_t data);

private:
    // Those variables are the interface to processor
    // External sources access just those to configure and trigger
    // the IP
    j_compress_ptr cinfo;
    jpeg_component_info * compptr;
    JSAMPARRAY sample_data;
    JBLOCKROW coef_blocks;
    JDIMENSION start_row;
    JDIMENSION start_col;
    JDIMENSION num_blocks;
    char control;

    // Holds essential information
    unsigned int m_unique_priority;
    unsigned int m_start_address;
    unsigned int m_end_address;

}; /* end class Forward_DCT */

//
// Slave Direct Interface
//
```

```
inline bool Forward_DCT::direct_read(int *data, unsigned int address)
{
    return (read(data, address) == SIMPLE_BUS_OK);
}

inline bool Forward_DCT::direct_write(int *data, unsigned int address)
{
    return (write(data, address) == SIMPLE_BUS_OK);
}

/*
 * Constructor
 */

inline Forward_DCT::Forward_DCT(sc_module_name name_,
        unsigned int unique_priority, unsigned int start_address,
        unsigned int end_address) :
    sc_module(name_), m_unique_priority(unique_priority),
            m_start_address(start_address),
            m_end_address(end_address), control(0)
{
    // Main function that sleeps until the IP is configured correctly
    SC_THREAD(main_action);
}

/*
 * Destructor
 */
inline Forward_DCT::~Forward_DCT()
{
    // Empty
}
```

```
//
// Necessary for SimpleBus Standard
//

inline unsigned int Forward_DCT::start_address() const
{
    return m_start_address;
}

inline unsigned int Forward_DCT::end_address() const
{
    return m_end_address;
}

#endif /*FORWARD_DCT_H_*/
```

forward_DCT.cpp

```
**********************************************************************

/**
 * @file      forward_DCT.cpp
 * @author    Daniel Pereira Volpato
 * @author    Leonardo Luiz Ecco
 */

#include "forward_DCT.h"

#include <stdio.h> //DEBUG ONLY
#include "jdct.h"       /* Private declarations for DCT subsystem */

#define GETADDR(ptr) ((unsigned int) &(ptr))
```

```
/*
 * Macros for jpeg_fdct_islow(..).
 */
#ifdef DCT_ISLOW_SUPPORTED

/*
 * This module is specialized to the case DCTSIZE = 8.
 */

#if DCTSIZE != 8
Sorry, this code only copes with 8x8 DCTs. /* deliberate syntax err */
#endif

/*
 * The poop on this scaling stuff is as follows:
 *
 * Each 1-D DCT step produces outputs which are a factor of sqrt(N)
 * larger than the true DCT outputs.  The final outputs are therefore
 * a factor of N larger than desired; since N=8 this can be cured by
 * a simple right shift at the end of the algorithm.  The advantage of
 * this arrangement is that we save two multiplications per 1-D DCT,
 * because the y0 and y4 outputs need not be divided by sqrt(N).
 * In the IJG code, this factor of 8 is removed by the quantization
 * step (in jcdctmgr.c), NOT in this module.
 *
 * We have to do addition and subtraction of the integer inputs, which
 * is no problem, and multiplication by fractional constants, which is
 * a problem to do in integer arithmetic.  We multiply all the
 * constants by CONST_SCALE and convert them to integer constants
 * (thus retaining CONST_BITS bits of precision in the constants).
 * After doing a multiplication we have to divide the product by
 * CONST_SCALE, with proper rounding, to produce the correct output.
 * This division can be done cheaply as a right shift of CONST_BITS
 * bits.  We postpone shifting as long as possible so that partial
```

```
 * sums can be added together with full fractional precision.
 *
 * The outputs of the first pass are scaled up by PASS1_BITS bits so
 * that they are represented to better-than-integral precision. These
 * outputs require BITS_IN_JSAMPLE + PASS1_BITS + 3 bits; this fits in
 * a 16-bit word with the recommended scaling. (For 12-bit sample
 * data, the intermediate array is INT32 anyway.)
 *
 * To avoid overflow of the 32-bit intermediate results in pass 2, we
 * must have BITS_IN_JSAMPLE + CONST_BITS + PASS1_BITS <= 26.  Error
 * analysis shows that the values given below are the most effective.
 */

#if BITS_IN_JSAMPLE == 8
#define CONST_BITS  13
#define PASS1_BITS  2
#else
#define CONST_BITS  13
#define PASS1_BITS  1  /* lose a little precision to avoid overflow */
#endif

/* Some C compilers fail to reduce "FIX(constant)" at compile time,
 * thus causing a lot of useless floating-point operations at run
 * time. To get around this we use the following pre-calculated
 * constants. If you change CONST_BITS you may want to add appropriate
 * values. (With a reasonable C compiler, you can just rely on the
 * FIX() macro...)
 */

#if CONST_BITS == 13
#define FIX_0_298631336  ((INT32)  2446)    /* FIX(0.298631336) */
#define FIX_0_390180644  ((INT32)  3196)    /* FIX(0.390180644) */
#define FIX_0_541196100  ((INT32)  4433)    /* FIX(0.541196100) */
#define FIX_0_765366865  ((INT32)  6270)    /* FIX(0.765366865) */
#define FIX_0_899976223  ((INT32)  7373)    /* FIX(0.899976223) */
```

```
#define FIX_1_175875602  ((INT32)  9633)   /* FIX(1.175875602) */
#define FIX_1_501321110  ((INT32)  12299)  /* FIX(1.501321110) */
#define FIX_1_847759065  ((INT32)  15137)  /* FIX(1.847759065) */
#define FIX_1_961570560  ((INT32)  16069)  /* FIX(1.961570560) */
#define FIX_2_053119869  ((INT32)  16819)  /* FIX(2.053119869) */
#define FIX_2_562915447  ((INT32)  20995)  /* FIX(2.562915447) */
#define FIX_3_072711026  ((INT32)  25172)  /* FIX(3.072711026) */
#else
#define FIX_0_298631336  FIX(0.298631336)
#define FIX_0_390180644  FIX(0.390180644)
#define FIX_0_541196100  FIX(0.541196100)
#define FIX_0_765366865  FIX(0.765366865)
#define FIX_0_899976223  FIX(0.899976223)
#define FIX_1_175875602  FIX(1.175875602)
#define FIX_1_501321110  FIX(1.501321110)
#define FIX_1_847759065  FIX(1.847759065)
#define FIX_1_961570560  FIX(1.961570560)
#define FIX_2_053119869  FIX(2.053119869)
#define FIX_2_562915447  FIX(2.562915447)
#define FIX_3_072711026  FIX(3.072711026)
#endif

/* Multiply an INT32 variable by an INT32 constant to yield an INT32
 * result.
 * For 8-bit samples with the recommended scaling, all the variable
 * and constant values involved are no more than 16 bits wide, so a
 * 16x16->32 bit multiply can be used instead of a full 32x32
 * multiply. For 12-bit samples, a full 32-bit multiplication will be
 * needed.
 */

#if BITS_IN_JSAMPLE == 8
#define MULTIPLY(var,const)  MULTIPLY16C16(var,const)
#else
#define MULTIPLY(var,const)  ((var) * (const))
```

```
#endif

/*
 * Perform the forward DCT on one block of samples.
 *
 * COPIED FROM jfdctint.c
 */
inline GLOBAL(void) jpeg_fdct_islow(DCTELEM * data)
{
    INT32 tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    INT32 tmp10, tmp11, tmp12, tmp13;
    INT32 z1, z2, z3, z4, z5;
    DCTELEM *dataptr;
    int ctr;
SHIFT_TEMPS

    /* Pass 1: process rows. */
    /* Note results are scaled up by sqrt(8) compared to a true DCT;*/
    /* furthermore, we scale the results by 2**PASS1_BITS. */

    dataptr = data;
    for (ctr = DCTSIZE-1; ctr >= 0; ctr--)
    {
        tmp0 = dataptr[0] + dataptr[7];
        tmp7 = dataptr[0] - dataptr[7];
        tmp1 = dataptr[1] + dataptr[6];
        tmp6 = dataptr[1] - dataptr[6];
        tmp2 = dataptr[2] + dataptr[5];
        tmp5 = dataptr[2] - dataptr[5];
        tmp3 = dataptr[3] + dataptr[4];
        tmp4 = dataptr[3] - dataptr[4];

        /* Even part per LL&M figure 1 --- note that published figure
         * is faulty;
         * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
```

```
 */

tmp10 = tmp0 + tmp3;
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;

dataptr[0] = (DCTELEM) ((tmp10 + tmp11) << PASS1_BITS);
dataptr[4] = (DCTELEM) ((tmp10 - tmp11) << PASS1_BITS);

z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
dataptr[2] = (DCTELEM)
    DESCALE(z1 + MULTIPLY(tmp13, FIX_0_765366865),
        CONST_BITS-PASS1_BITS);
dataptr[6] = (DCTELEM)
    DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
        CONST_BITS-PASS1_BITS);

/* Odd part per figure 8 --- note paper omits factor of
 * sqrt(2).
 * cK represents cos(K*pi/16).
 * i0..i3 in the paper are tmp4..tmp7 here.
 */

z1 = tmp4 + tmp7;
z2 = tmp5 + tmp6;
z3 = tmp4 + tmp6;
z4 = tmp5 + tmp7;
z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

/* sqrt(2) * (-c1+c3+c5-c7) */
tmp4 = MULTIPLY(tmp4, FIX_0_298631336);
/* sqrt(2) * ( c1+c3-c5+c7) */
tmp5 = MULTIPLY(tmp5, FIX_2_053119869);
/* sqrt(2) * ( c1+c3+c5-c7) */
```

```
    tmp6 = MULTIPLY(tmp6, FIX_3_072711026);
    /* sqrt(2) * ( c1+c3-c5-c7) */
    tmp7 = MULTIPLY(tmp7, FIX_1_501321110);
    /* sqrt(2) * (c7-c3) */
    z1 = MULTIPLY(z1, - FIX_0_899976223);
    /* sqrt(2) * (-c1-c3) */
    z2 = MULTIPLY(z2, - FIX_2_562915447);
    /* sqrt(2) * (-c3-c5) */
    z3 = MULTIPLY(z3, - FIX_1_961570560);
    /* sqrt(2) * (c5-c3) */
    z4 = MULTIPLY(z4, - FIX_0_390180644);

    z3 += z5;
    z4 += z5;

    dataptr[7] = (DCTELEM)
        DESCALE(tmp4 + z1 + z3, CONST_BITS-PASS1_BITS);
    dataptr[5] = (DCTELEM)
        DESCALE(tmp5 + z2 + z4, CONST_BITS-PASS1_BITS);
    dataptr[3] = (DCTELEM)
        DESCALE(tmp6 + z2 + z3, CONST_BITS-PASS1_BITS);
    dataptr[1] = (DCTELEM)
        DESCALE(tmp7 + z1 + z4, CONST_BITS-PASS1_BITS);

    dataptr += DCTSIZE; /* advance pointer to next row */
}

/* Pass 2: process columns.
 * We remove the PASS1_BITS scaling, but leave the results scaled
 * up by an overall factor of 8.
 */

dataptr = data;
for (ctr = DCTSIZE-1; ctr >= 0; ctr--)
{
```

```
tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];

/* Even part per LL&M figure 1 --- note that published figure
 * is faulty;
 * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
 */

tmp10 = tmp0 + tmp3;
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;

dataptr[DCTSIZE*0] = (DCTELEM) DESCALE(tmp10 + tmp11,
        PASS1_BITS);
dataptr[DCTSIZE*4] = (DCTELEM) DESCALE(tmp10 - tmp11,
        PASS1_BITS);

z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
dataptr[DCTSIZE*2] = (DCTELEM)
        DESCALE(z1 + MULTIPLY(tmp13, FIX_0_765366865),
        CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*6] = (DCTELEM)
        DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
        CONST_BITS+PASS1_BITS);

/* Odd part per figure 8 --- note paper omits factor of
 * sqrt(2).
 * cK represents cos(K*pi/16).
```

```
 * i0..i3 in the paper are tmp4..tmp7 here.
 */

z1 = tmp4 + tmp7;
z2 = tmp5 + tmp6;
z3 = tmp4 + tmp6;
z4 = tmp5 + tmp7;
z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

/* sqrt(2) * (-c1+c3+c5-c7) */
tmp4 = MULTIPLY(tmp4, FIX_0_298631336);
/* sqrt(2) * ( c1+c3-c5+c7) */
tmp5 = MULTIPLY(tmp5, FIX_2_053119869);
/* sqrt(2) * ( c1+c3+c5-c7) */
tmp6 = MULTIPLY(tmp6, FIX_3_072711026);
/* sqrt(2) * ( c1+c3-c5-c7) */
tmp7 = MULTIPLY(tmp7, FIX_1_501321110);
/* sqrt(2) * (c7-c3) */
z1 = MULTIPLY(z1, - FIX_0_899976223);
/* sqrt(2) * (-c1-c3) */
z2 = MULTIPLY(z2, - FIX_2_562915447);
/* sqrt(2) * (-c3-c5) */
z3 = MULTIPLY(z3, - FIX_1_961570560);
/* sqrt(2) * (c5-c3) */
z4 = MULTIPLY(z4, - FIX_0_390180644);

z3 += z5;
z4 += z5;

dataptr[DCTSIZE*7] = (DCTELEM) DESCALE(tmp4 + z1 + z3,
        CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*5] = (DCTELEM) DESCALE(tmp5 + z2 + z4,
        CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*3] = (DCTELEM) DESCALE(tmp6 + z2 + z3,
        CONST_BITS+PASS1_BITS);
```

```
        dataptr[DCTSIZE*1] = (DCTELEM) DESCALE(tmp7 + z1 + z4,
                CONST_BITS+PASS1_BITS);

        dataptr++; /* advance pointer to next column */
    }
}

#endif /* DCT_ISLOW_SUPPORTED */

/* Private subobject for this module */
typedef struct
{
    struct jpeg_forward_dct pub; /* public fields */

    /* Pointer to the DCT routine actually in use */
    forward_DCT_method_ptr do_dct;

    /* The actual post-DCT divisors --- not identical to the quant
     * table entries, because of scaling (especially for an
     * unnormalized DCT).
     * Each table is given in normal array order.
     */
    DCTELEM * divisors[NUM_QUANT_TBLS];

#ifdef DCT_FLOAT_SUPPORTED
    /* Same as above for the floating-point case. */
    float_DCT_method_ptr do_float_dct;
    FAST_FLOAT * float_divisors[NUM_QUANT_TBLS];
#endif
} my_fdct_controller;

typedef my_fdct_controller * my_fdct_ptr;

/**
 * The IP functionallity
```

```
 * A SystemC process that sleeps until a valid data is programed into
 * IP
 * Process all input data without loss and untimed
 * (behaviour simulation)
 **/
void Forward_DCT::main_action()
{
    while (true)
    {
        wait(wakeup);

        // IP trigger (start signal)
        if (control)
        {
            // Function forward_DCT(...) from jcdctmgrc.c :

            //my_fdct_ptr fdct = (my_fdct_ptr) cinfo->fdct;
            my_fdct_ptr fdct = (my_fdct_ptr)
                rmem32u( GETADDR(cinfo->fdct));

            //No need for this one anymore, jpeg_fdct_islow is gonna
            //be called.
            //forward_DCT_method_ptr do_dct = fdct->do_dct;

            int quant_tbl_no_aux = rmem32u(
                    GETADDR(compptr->quant_tbl_no));
            DCTELEM * divisors = (DCTELEM *) rmem32u(
                    GETADDR(fdct->divisors[quant_tbl_no_aux]));

            /* work area for FDCT subroutine */
            DCTELEM workspace[DCTSIZE2];
            JDIMENSION bi;

            /* fold in the vertical offset once */
            sample_data += start_row;
```

```
        /* Load data into workspace, applying unsigned->signed
         * conversion */
        for (bi = 0; bi < num_blocks; bi++, start_col += DCTSIZE)
        {
            {
                register DCTELEM *workspaceptr;
                register JSAMPROW elemptr;
                register int elemr;

                workspaceptr = workspace;
                for (elemr = 0; elemr < DCTSIZE; elemr++)
                {
                    elemptr = (JSAMPROW) rmem32u(
                            GETADDR(sample_data[elemr]))
                            + start_col;

#if DCTSIZE == 8        /* unroll the inner loop */

                    *workspaceptr++
                            = GETJSAMPLE( rmem8u(
                                    GETADDR(*elemptr++) ))
                                    - CENTERJSAMPLE;
                    *workspaceptr++
                            = GETJSAMPLE( rmem8u(
                                    GETADDR(*elemptr++) ))
                                    - CENTERJSAMPLE;
                    *workspaceptr++
                            = GETJSAMPLE( rmem8u(
                                    GETADDR(*elemptr++) ))
                                    - CENTERJSAMPLE;
                    *workspaceptr++
                            = GETJSAMPLE( rmem8u(
                                    GETADDR(*elemptr++) ))
                                    - CENTERJSAMPLE;
```

```
                        *workspaceptr++
                                = GETJSAMPLE( rmem8u(
                                        GETADDR(*elemptr++) ))
                                        - CENTERJSAMPLE;
                        *workspaceptr++
                                = GETJSAMPLE( rmem8u(
                                        GETADDR(*elemptr++) ))
                                        - CENTERJSAMPLE;
                        *workspaceptr++
                                = GETJSAMPLE( rmem8u(
                                        GETADDR(*elemptr++) ))
                                        - CENTERJSAMPLE;
                        *workspaceptr++
                                = GETJSAMPLE( rmem8u(
                                        GETADDR(*elemptr++) ))
                                        - CENTERJSAMPLE;

#else
                        {
                            register int elemc;
                            for (elemc = DCTSIZE; elemc > 0; elemc--)
                            {
                                *workspaceptr++ = GETJSAMPLE(
                                        rmem8u( GETADDR(*elemptr++) ))
                                        - CENTERJSAMPLE;
                            }
                        }
#endif
                    }
                }

                /* Perform the DCT */
                jpeg_fdct_islow(workspace);

                /* Quantize/descale the coefficients, and store into
```

```
                 * coef_blocks[] */
                {
                    register DCTELEM temp, qval;
                    register int i;
                    register JCOEFPTR output_ptr = coef_blocks[bi];

                    for (i = 0; i < DCTSIZE2; i++)
                    {
                        qval = (DCTELEM) rmem32u (
                                GETADDR(divisors[i]));
                        temp = workspace[i];

                        /* Divide the coefficient value by qval,
                         * ensuring proper rounding.
                         * Since C does not specify the direction of
                         * rounding for negative quotients, we have
                         * to force the dividend positive for
                         * portability.
                         *
                         * In most files, at least half of the output
                         * values will be zero (at default
                         * quantization settings, more like
                         * three-quarters...)
                         * so we should ensure that this case is fast.
                         * On many machines, a comparison is enough
                         * cheaper than a divide to make a special
                         * test a win.  Since both inputs will be
                         * nonnegative, we need only test
                         * for a < b to discover whether a/b is 0.
                         * If your machine's division is fast enough,
                         * define FAST_DIVIDE.
                         */
#ifdef FAST_DIVIDE
#define DIVIDE_BY(a,b)  a /= b
#else
```

```
#define DIVIDE_BY(a,b)  if (a >= b) a /= b; else a = 0
#endif
                        if (temp < 0)
                        {
                            temp = -temp;
                            temp += qval>>1; /* for rounding */
                            DIVIDE_BY(temp, qval);
                            temp = -temp;
                        }
                        else
                        {
                            temp += qval>>1; /* for rounding */
                            DIVIDE_BY(temp, qval);
                        }

                        //output_ptr[i] = (JCOEF) temp;
                        wmem16u( GETADDR(output_ptr[i]), (short)temp);
                    }
                }
            }
        } // End of if(control)

        // Tell software job is done
        control = 0;

    } // End of while (true)
}

/**
 * Receives incoming read requests
 * IP is acting like a slave
 * @param data pointer to buffer that will receive data
 * @param address the address to be read
 * @returns Success or Error
 **/
```

```
simple_bus_status Forward_DCT::read(int *data, unsigned int address)
{
    switch (address - m_start_address)
    {
    case 0x00:
        (*data) = (int) cinfo;
        break;

    case 0x04:
        (*data) = (int) compptr;
        break;

    case 0x08:
        (*data) = (int) sample_data;
        break;

    case 0x0C:
        (*data) = (int) coef_blocks;
        break;

    case 0x10:
        (*data) = (int) start_row;
        break;

    case 0x14:
        (*data) = (int) start_col;
        break;

    case 0x18:
        (*data) = (int) num_blocks;
        break;

    case 0x1C:
        (*data) = (char) control;
        break;
```

```
    default:
        return SIMPLE_BUS_ERROR;
        break;
    }

    return SIMPLE_BUS_OK;
}

/**
 * Receives incoming write requests
 * IP is acting like a slave
 * @param data pointer to buffer that hold the data
 * @param address the address to be written
 * @returns Success or Error
 **/
simple_bus_status Forward_DCT::write(int *data, unsigned int address)
{
    switch (address - m_start_address)
    {
    case 0x00:
        cinfo = (j_compress_ptr) (*data);
        break;

    case 0x04:
        compptr = (jpeg_component_info *) (*data);
        break;

    case 0x08:
        sample_data = (JSAMPARRAY) (*data);
        break;

    case 0x0C:
        coef_blocks = (JBLOCKROW) (*data);
        break;
```

```
    case 0x10:
        start_row = (JDIMENSION) (*data);
        break;

    case 0x14:
        start_col = (JDIMENSION) (*data);
        break;

    case 0x18:
        num_blocks = (JDIMENSION) (*data);
        break;

    case 0x1C:
        control = (char) (*data);
        break;

    default:
        return SIMPLE_BUS_ERROR;
        break;
    }

    if (control)
    {
        wakeup.notify();
    }

    return SIMPLE_BUS_OK;
}

//
//////////////////////////////////////////////////////////////////
/** Reads 32bits from the bus address space
 * IP is acting like a master
 * @param address the address to be read
```

```
 * @returns The data read as unsigned int
 **/
uint32_t Forward_DCT::rmem32u(uint32_t address)
{
    uint32_t tmp;
    bus_port->burst_read(m_unique_priority, (int*) &tmp, address, 1,
            false);
    return tmp;
}

/** Reads 8bits from the bus address space
 * IP is acting like a master
 * @param address the address to be read
 * @returns The data read as char
 **/
uint8_t Forward_DCT::rmem8u(uint32_t address)
{
    uint32_t tmp;
    // Actually it can only read 32 bits
    // The trick on address is for big/endian sake
    bus_port->burst_read(m_unique_priority, (int*) &tmp, address
            -address%4, 1, false);
    return (tmp>>(24-(8*(address%4))))&0x000000FF;
}

/** Writes 8bits from the bus address space
 * IP is acting like a master
 * @param address the address to be written
 * @param data the address to be written
 **/
void Forward_DCT::wmem8u(unsigned int address, uint8_t data)
{
    uint32_t tmp;
    // Actually it can only write 32 bits
    // The trick on address is for big/endian sake
```

```
    // First we make a read
    bus_port->burst_read(m_unique_priority, (int*) &tmp, address
            -(address%4), 1, false);
    // Next pack the data in a convenient format
    switch (address%4)
    {
    case 0:
        tmp = (tmp&0x00FFFFFF)|((data&0xFF)<<24);
        break;
    case 1:
        tmp = (tmp&0xFF00FFFF)|((data&0xFF)<<16);
        break;
    case 2:
        tmp = (tmp&0xFFFF00FF)|((data&0xFF)<<8);
        break;
    case 3:
        tmp = (tmp&0xFFFFFF00)|((data&0xFF));
        break;
    }
    // Now write the 32 bits back with the data packed
    bus_port->burst_write(m_unique_priority, (int*) &tmp, address
            -(address%4), 1, false);
}

void Forward_DCT::wmem16u(unsigned int address, uint16_t data)
{
    uint8_t lsb = (char) (data & 0x00FF);
    uint8_t msb = (char) ((data>>8) & 0x00FF);

    wmem8u(address, msb);
    wmem8u(address+1, lsb);
}
```