UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

TRANSFORMADA RÁPIDA DE FOURIER PARALELA EM SISTEMA COMPUTACIONAL HÍBRIDO RECONFIGURÁVEL

TRABALHO DE GRADUAÇÃO

Vitor Conrado Faria Gomes

Santa Maria, RS, Brasil

2010

TRANSFORMADA RÁPIDA DE FOURIER PARALELA EM SISTEMA COMPUTACIONAL HÍBRIDO RECONFIGURÁVEL

por

Vitor Conrado Faria Gomes

Trabalho de Graduação apresentado ao Curso de Ciência da Computação
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Profª. Andrea Schwertner Charão (UFSM)

Co-orientador: Dr. Haroldo Fraga de Campos Velho (INPE)

Trabalho de Graduação N. 296

Santa Maria, RS, Brasil

2010

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

TRANSFORMADA RÁPIDA DE FOURIER PARALELA EM SISTEMA COMPUTACIONAL HÍBRIDO RECONFIGURÁVEL

elaborado por
Vitor Conrado Faria Gomes

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Profª. Andrea Schwertner Charão (UFSM)
(Presidente/Orientador)

Prof. Giovani Baratto (UFSM)

Dr. Adriano Petry (INPE)

Santa Maria, 05 de Janeiro de 2010.

“Aprender sem pensar é tempo perdido.”
— KUNG-FU-TSE

AGRADECIMENTOS

Gostaria de agradecer primeiramente à minha família, em especial: aos meus pais, Miriam Helena e Mario Sergio, que me serviram de exemplo, me apoiaram, deram todo o suporte necessário para atingir meus objetivos e sempre me incentivaram na realização dos meus estudos; à minha namorada Luciane, que me apoiou durante todo o período de realização da graduação, me deu força nos momentos difíceis, me aturou nos momentos de estresse e comemorou comigo os momentos de alegria; à Rosa que me ajudou e deu todo o apoio durante a minha estadia em Santa Maria; ao meu irmão Tiago e a minha tia Marisa, que sempre estavam atentos ao que acontecia na minha vida acadêmica.

Agradeço aos professores do curso de Ciência da Computação, em especial ao Cesar Pozzer, Benhur Stein e Andrea Charão, que além do conhecimento, compartilharam comigo a amizade. Agradeço novamente à Andrea Charão, minha orientadora, que me despertou o gosto por pesquisas e reacendeu meu interesse em ser professor. Agradeço pela dedicação, paciência, pelas noites escrevendo artigos e sobretudo pela amizade.

Ao meu orientador Haroldo de Campos Velho, que me orientou, me motivou e soube sempre dar boas dicas nos momentos em que foram necessárias.

Aos meus colegas do LSC, do PET, do DACC, do INPE e da Florestal, que compartilharam muitas horas durante os 6 anos que estive na UFSM. Em especial ao Vinicius Cogo, Gustavo Rissetti, Robson Disarz e Marcelo Bernál, que além de colegas foram grandes companheiros nessa jornada.

E aos demais amigos, parentes e conhecidos que de uma forma ou de outra ajudaram no decorrer dessa jornada, muito obrigado!

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

TRANSFORMADA RÁPIDA DE FOURIER PARALELA EM SISTEMA COMPUTACIONAL HÍBRIDO RECONFIGURÁVEL

Autor: Vitor Conrado Faria Gomes
Orientador: Profª. Andrea Schwertner Charão (UFSM)
Co-orientador: Dr. Haroldo Fraga de Campos Velho (INPE)
Local e data da defesa: Santa Maria, 05 de Janeiro de 2010.

A computação híbrida reconfigurável vem se destacando no cenário de computação de alto desempenho, sendo considerada uma promissora opção para sistemas que requerem processamento intensivo. Com esta abordagem é possível utilizar coprocessadores especializados que cooperam com processadores de propósito geral na solução de aplicações. O Cray XD1 é um dos primeiros sistemas comerciais que interconecta *FPGAs* a nós multiprocessados, formando um sistema computacional híbrido reconfigurável. Neste trabalho, tem-se por objetivo o projeto e implementação de uma FFT unidimensional que tira proveito desta arquitetura, explorando o paralelismo e distribuindo computação entre CPU e FPGA. Esta FFT é validada para variados tamanhos de dados de entrada e apresenta ganho de desempenho face a uma implementação baseada somente em software.

Palavras-chave: FFT, Computação Híbrida Reconfigurável, Cray XD1.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
CLB	Configuration Logical Blocks
CPU	Central Processing Unit
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
IOB	Input/Output Block
SGI	Silicon Graphics, Inc.
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit

SUMÁRIO

1 INTRODUÇÃO

Uma das motivações para o desenvolvimento do computador eletrônico foi a previsão numérica. Nos anos 30, John von Neumann viu que o progresso da hidrodinâmica seria bastante acelerado se houvesse um meio de resolver equações numéricas complexas (1). Reconheceu ainda que a previsão meteorológica era um exemplo de problema suficientemente grande e cientificamente interessante para o computador automático.

Desde a época de von Neumann, a computação científica teve um grande progresso, permitindo resolver sistemas complexos e realizar previsões numéricas com grande precisão, que constantemente exigem mais recursos computacionais para a sua resolução. De maneira geral, a solução de problemas científicos necessita de uso intensivo de processamento.

Visando melhorar o desempenho de aplicações na área de computação científica, diversas estratégias têm sido desenvolvidas, sendo a computação paralela a mais difundida. Nesta abordagem, são utilizados diversos processadores de propósito geral para a computação de uma aplicação em paralelo, sendo necessária a utilização de técnicas e ferramentas especiais para a paralelização de problemas, nem sempre envolvendo tarefas triviais (2; 3; 4).

Nos últimos anos, surge uma nova abordagem, conhecida como Computação Híbrida Reconfigurável, em que são utilizados processadores de propósito geral em cooperação com *Field Gate Programmable Arrays* (FPGAs). FPGAs são dispositivos lógicos programáveis que funcionam como circuitos dedicados. Sua vantagem é ter a flexibilidade de soluções baseadas em software com o desempenho de soluções implementadas em hardware. Além disso, FPGAs permitem aumentar significativamente a densidade computacional e consomem menos energia que microprocessadores (5; 2).

Em sistemas híbridos reconfiguráveis de alto desempenho, o objetivo é acelerar a

execução de tarefas críticas de uma aplicação através da configuração do FPGA como um coprocessador especializado. Esta abordagem, devido à sua grande importância para computação científica, vem sendo investigada por diversos trabalhos recentes que exploram o poder computacional de CPU e FPGA combinados.

A Transformada de Fourier é uma transformada linear usada em diversas aplicações científicas. Esta transformada é usualmente encontrada como núcleo de aplicações que vão desde processamento de imagens até simulações atmosféricas. A Transformada Rápida de Fourier (FFT) é um algoritmo que computa a Transformada de Fourier reduzindo sua complexidade de $O(N^2)$ para $O(NLogN)$. Esta forma de computar a Transformada de Fourier é considerada um dos *Top 10 algoritmos do século 20* (6; 7), comprovando sua importância para a computação científica. Entretanto, apesar de reduzir a complexidade desta transformada, a FFT continua sendo um algoritmo computacionalmente intensivo e, portanto, alvo de estudos que visam o aumento do desempenho de sua implementação (8; 9; 10; 11; 12).

Diante deste cenário, este trabalho apresenta uma implementação de FFT paralela para um sistema híbrido reconfigurável. O objetivo principal é acelerar a execução desta transformada através da utilização combinada de FPGA e CPU na sua computação. O ambiente de testes utilizado é um Cray XD1 que incorpora FPGAs em sua arquitetura para permitir a aceleração de aplicações críticas. A implementação visa aproveitar os recursos disponíveis neste equipamento e estabelecer um perfil da execução da FFT utilizando CPU e FPGA.

No restante deste texto, são apresentados os conceitos e as tecnologias utilizados no desenvolvimento deste trabalho. Na sequência, apresenta-se o projeto da FFT híbrida e paralela desenvolvido e sua implementação. No capítulo 5 é apresentada uma análise das implementações, enquanto que o capítulo 6 apresenta o estado atual deste trabalho.

2 CONCEITOS E TECNOLOGIAS

Neste capítulo são apresentados os conceitos e as tecnologias utilizados para o desenvolvimento deste trabalho.

2.1 Computação Híbrida Reconfigurável

De maneira formal, a Computação Híbrida, também conhecida como Computação Heterogênea, pode ser definida como a estratégia de utilizar vários tipos de elementos de processamento em um único fluxo de trabalho, permitindo que cada dispositivo execute as tarefas às quais está mais adaptado. Este modelo pode empregar processadores especializados como processadores vetoriais, GPUs, DSPs, FPGAs, etc., que cooperam com processadores de propósito geral na execução de uma aplicação (2). A Computação Híbrida Reconfigurável é uma subárea da Computação Híbrida que utiliza dispositivos reconfiguráveis (FPGAs) como coprocessadores especializados.

O conceito de computação híbrida, apesar de antigo, está se tornando cada vez mais importante no cenário de computação de alto desempenho. Nós últimos anos, fabricantes de sistemas de alto desempenho, como Cray, SGI e SRC, introduziram sistemas de computação híbrida como Cray XD1, XT3, XT4, XT5h, SGI RASC e SRC-6 MAP. Estes sistemas tem sido explorados em diversos trabalhos (13; 14; 15; 16; 17) e representam uma boa alternativa para problemas que exigem grande esforço computacional para a solução.

Em sistemas híbridos reconfiguráveis de alto desempenho, o objetivo é acelerar a execução de tarefas críticas através da configuração do FPGA como um coprocessador especializado. Aplicações para estes sistemas têm o potencial de obter um grande aumento de desempenho em relação às soluções baseadas somente em software. A grande vantagem desta abordagem é o paralelismo próprio de dispositivos reconfiguráveis em

conjunto com o processamento em CPU. Entretanto, é grande a complexidade no desenvolvimento de aplicações para esta arquitetura, devido ao duplo paradigma do sistema, o que dificulta a identificação de gargalos e a obtenção do desempenho desejado (18). Isto é agravado pelas características de comunicação entre os dispositivos de computação, que são específicas para cada sistema de computação híbrida reconfigurável.

Field-Programmable Gate Arrays (FPGAs) são dispositivos lógicos programáveis compostos por blocos lógicos configuráveis que, diferentemente de circuitos dedicados, podem ser reconfigurados diversas vezes. Sua utilização visa obter o desempenho de soluções implementadas em hardware e a flexibilidade de soluções baseadas em software (19; 5; 20; 21). A flexibilidade é dada pela facilidade de configuração através de uma descrição de hardware, geralmente escrita em VHDL ou Verilog. Essas linguagens permitem a descrição do comportamento de um circuito lógico e facilitam a criação de novas aplicações em hardware devido ao nível de abstração que fornecem ao programador.

O uso de FPGAs tem algumas vantagens sobre sistemas paralelos de computadores, como no caso de *clusters*. Primeiramente, porque FPGAs consomem menos energia que os microprocessadores. Além disso, FPGAs permitem aumentar significativamente a densidade computacional e o desempenho de tarefas (19; 22; 23). No entanto, nem todas aplicações podem ter um aumento de desempenho com o uso de hardware reconfigurável, em especial aquelas que possuem pouca oportunidade de paralelização.

A figura 2.1 ilustra o funcionamento interndo de um Field-Programmable Gate Array, o qual é constituído por blocos lógicos configuráveis incorporados em uma estrutura de roteamento programável. Estes blocos lógicos contém elementos para a realização de lógica combinacional simples e *flip-flops* para a realização de lógica sequencial. Estas unidades lógicas são implementadas utilizando blocos de memória, operações booleanas simples e normalmente possuem de 4 a 6 entradas. A estrutura de roteamento programável permite a configuração de ligações arbitrárias, de modo que os elementos lógicos podem ser conectados na forma desejada (23). Um possível implementação para um bloco lógico pode ser vista na figura 2.2.

Para permitir a construção flexível de sistemas complexos, FPGAs atuais contam com milhões de portas lógicas e podem operar em centenas de Megahertz. Para aumentar o desempenho e capaciadade destes dispositivos, elementos especiais podem ser incorpora-

dos, como memórias, multiplicadores, funções lógicas e até mesmo microprocessadores. Com estes recursos adicionais, é possível implementar sistemas completos em um único FPGA (23).

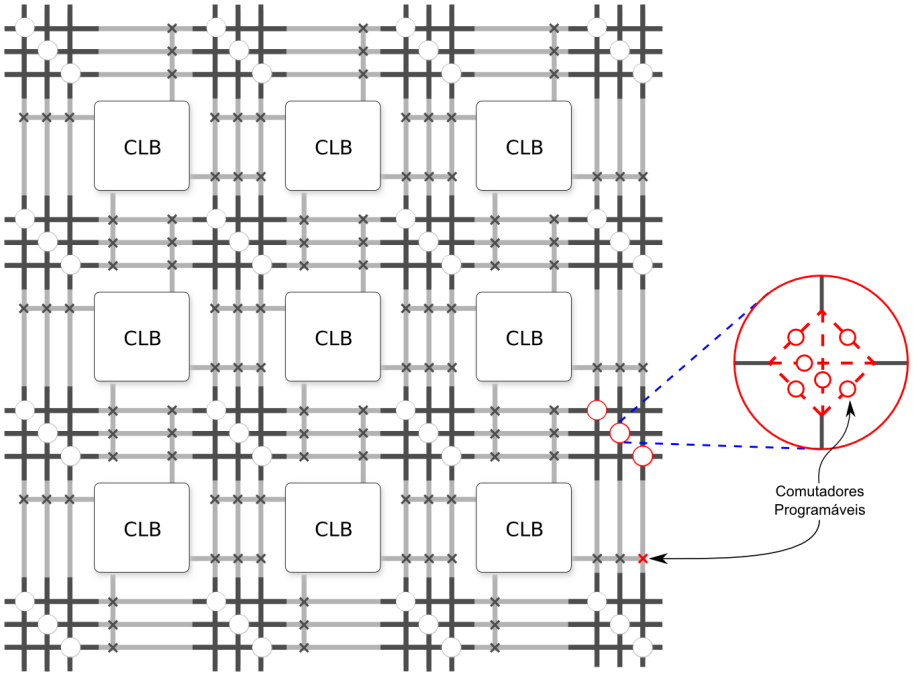


Figura 2.1: Arquitetura Básica FPGA

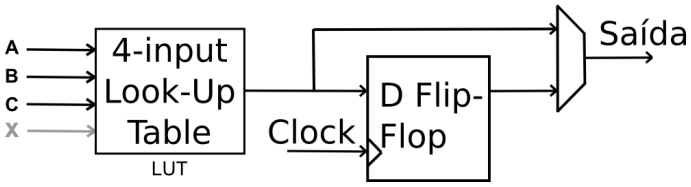


Figura 2.2: Bloco Lógico Configurável

O desenvolvimento de aplicações para FPGAs é feito através de linguagens de descrição de Hardware. Linguanges como VHDL e Verilog, permitem a descrição do comportamento lógico do sistema e o fluxo interno de dados. A conversão da descrição abstrata do hardware para configuração do FPGA segue os seguintes passos: Primeiramente o processo de síntese transforma a lógica em alto nível e o código comportamental em portas lógicas interconectadas. Na sequência, o processo de mapeamento separa as portas lógicas em grupos para poderem ser melhor adaptadas aos recursos lógicos do FPGA. O roteamento vem na sequência, este processo indica em qual bloco logico cada grupo de portas lógicas vai ser configurado e determina as interconexões que irão transportar os

sinais (23).

2.2 Transformada de Fourier

A Transformada de Fourier foi formulada por Jean-Baptiste Joseph Fourier e é uma transformada linear usada em diversas aplicações científicas. Esta transformada permite a expressão de funções não periódicas por integrais de funções senoidais de frequências diferentes sendo multiplicadas por coeficientes próprios. Esta transformada é aplicável em regiões onde a curva da função é finita.

A Transformada de Fourier $F(t)$ de uma função contínua $f(t)$ é definida por:

$$F(f) = \int_{-\infty}^{\infty} f(t)e^{-j2\pi ft}dt \tag{2.1}$$

e sua inversa por:

$$f(t) = \int_{-\infty}^{\infty} F(f)e^{j2\pi ft}df \tag{2.2}$$

onde $e^{j\theta} = cos(\theta) + jsen(\theta)$ e $j = \sqrt{-1}$.

Em sua formulação discreta, esta transformada é usualmente núcleo computacional de aplicações como processamento de sinais e solução de equações parciais (8; 11; 10). A Transformada Discreta de Fourier (DFT) de uma sequência de N números pode ser computada como:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \qquad k = 0, 1, ..., N-1 \tag{2.3}$$

e sua inversa por:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)W_N^{-kn} \qquad k = 0, 1, ..., N-1 \tag{2.4}$$

onde $W_N = e^{-2\pi\sqrt{-1}/N}$ é um coeficiente trigonométrico conhecido como fator de giro.

Na versão discreta da transformada, tanto para a direta quanto para a inversa, para cada posição do vetor é necessário realizar N multiplicações complexas e N-1 somas complexas. Para a computação total da série a ser transformada é realizado um esforço computacional na ordem de $O(N^2)$.

2.2.1 Transformada Rápida de Fourier (FFT)

O algoritmo da Transformada Rápida de Fourier (24) foi proposto para otimizar a computação da Transformada de Fourier, reduzindo sua complexidade de $O(N^2)$ para

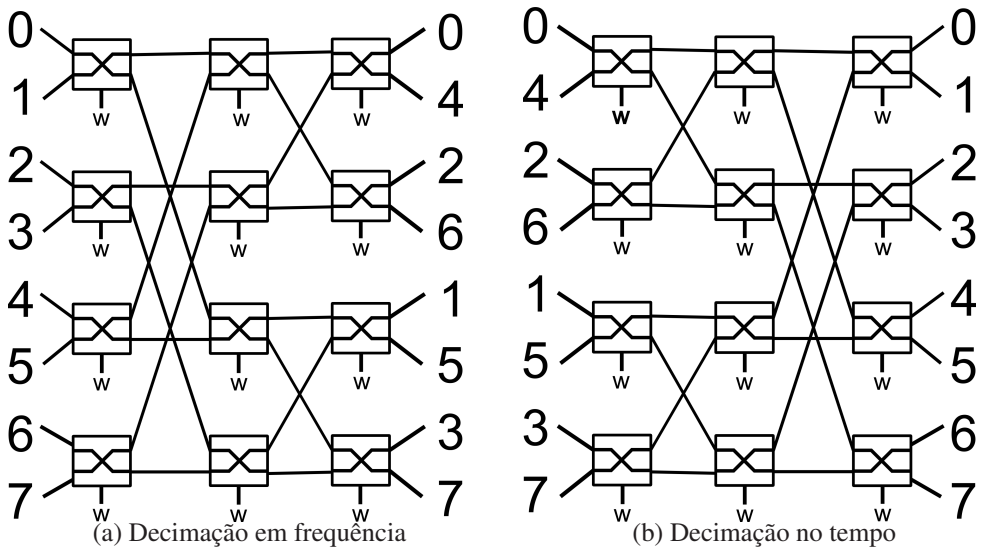
$O(NlogN)$. Esta redução é possível através da computação eficiente de operações com a eliminação de multiplicações por 1 que são encontradas nesta transformada.

A estratégia utilizado na computação da Transformada Rápida de Fourier é dividir a transformada em transformadas menores recursivamente para reduzir o esforço computacional. Existem várias formas de estruturar o algoritmo da FFT, sendo uma variante conhecida como radix-2, que opera sobre um vetor de N elementos, onde N é potência de 2. Devido a sua representação gráfica, a operação básica do algoritmo radix-2 é conhecida como "borboleta" e consiste de duas somas e uma multiplicação complexa. O algoritmo de radix-2 realiza cada operação sobre dois pontos, fornecendo a menor unidade computacional possível para a FFT (25). Esta configuração permite maior flexibilidade para a avaliação deste algoritmo, em especial quanto a avaliação de espaço em dispositivos reconfiguráveis. A figura 2.3 ilusta dois possíveis fluxos para a computação de uma FFT em 8 pontos. A operação borboleta é representada por cada quadrado branco desta figura e W representa o fator de giro.

A computação da FFT usando a versão radix-2 pode ser aplicada no domínio de frequência (*decimation-in-frequency*) ou no domínio de tempo (*decimation-in-time*). O fluxo de computação da transformada de decimação na frequência pode ser visto na figura 2.3a. Nesta figura é possível observar que a saída está ordenada na ordem reversa dos bits, ou seja, o índice 1 (001_b) aparece na posição 4 (100_b). O inverso acontece na decimação no tempo, onde a entrada deve ser rearranjada antes da computação das borboletas e a saída é obtida ordenada. Por causa desta ordenação no início da computação da FFT em decimação no tempo, esta algoritmo também é conhecido por *binary-exchange*. Na figura 2.3b é apresentado o fluxo para a computação da FFT radix-2 em decimação no tempo.

2.2.2 FFT Paralela

Apesar de reduzir a complexidade da computação da Transformada de Fourier, a Transformada Rápida de Fourier continua sendo uma operação computacionalmente intensiva. Visando aumentar o desempenho da computação destra transformada, diversas estratégias foram propostas para permitir a aceleração de sua computação. Alguns esforços visam a implementação desta operação em hardware (11; 12), enquanto outros buscam o aumento de desempenho através de estratégias de paralelização em soft-



(a) Decimação em frequência (b) Decimação no tempo

Figura 2.3: Fluxo Computacional da FFT de 8 pontos

ware (8; 9; 10; 26).

A paralelização da FFT utilizando o algoritmo *decimation-in-time* minimiza as comunicações entre os processos e facilita o particionamento de processamento entre as unidades funcionais (9). Esta estrutura fornece uma complexidade ideal de $O(LogN)$ quando computada com N processos. Na figura 2.3b pode ser visto que a cada passo da FFT ocorrem operações em pontos que aumentam a distância de seus índices. No último passo, as borboletas operam nos pontos i e $(N/2)+i$, onde i varia de 0 até $(N/2)-1$. No caso de particionar a FFT em duas tarefas paralelas (cinza escuro e cinza claro na figura 2.4), o último passo da FFT não pode ser obtido sem que ocorra troca de dados entre os processos. Esta dependência de dados é encontrada antes com o aumento da quantidade de processos paralelos que computam a FFT (9). Quando o custo de comunicação é alto, pode ser vantajoso atribuir a computação de N/P pontos para cada processo e calcular o restante sequencialmente em um único processo, conforme destacado na figura 2.4.

2.3 Cray XD1

O Cray XD1 é um sistema híbrido reconfigurável de alto desempenho lançado em outubro de 2004, trazendo no momento de seu lançamento algumas inovações tecnológicas. Entre elas está a rede de interconexão de alto desempenho RapidArray, otimizações no sistema Linux e a inclusão de FPGAs em seu chassis. Na figura 2.5 apresenta-se um *rack* com diversos equipamentos XD1 e um equipamento em destaque.

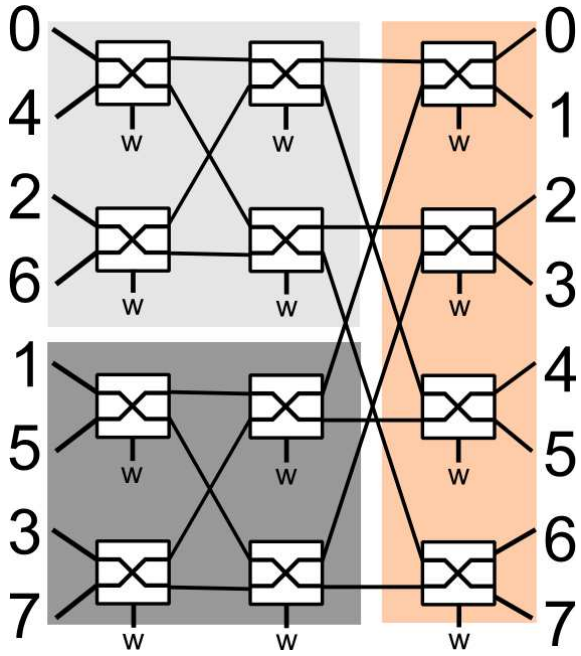


Figura 2.4: Fluxo computacional para FFT Paralela de 8 pontos

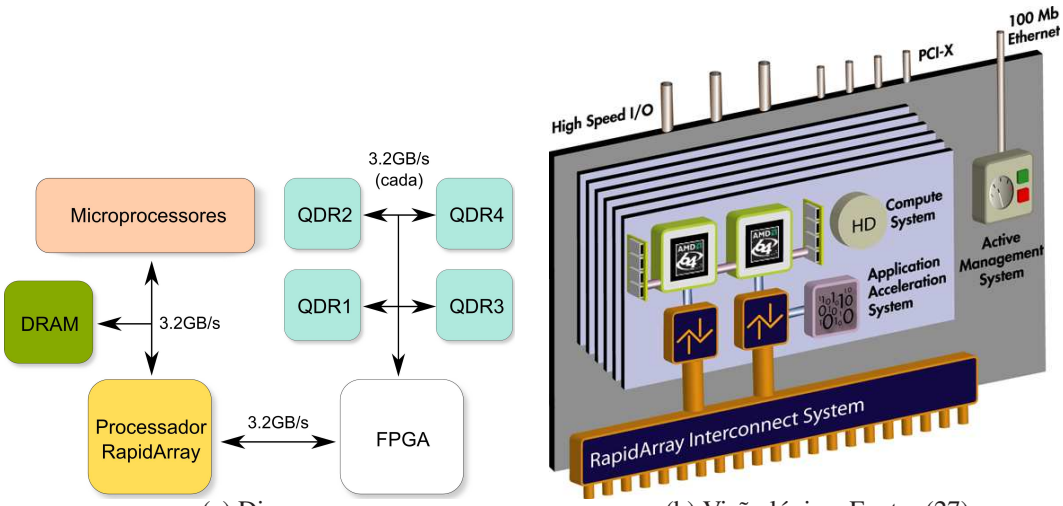
Cada sistema Cray XD1 é composto por seis nós (*blades*), cada um contendo dois processadores de propósito geral AMD Opteron 64bits e um FPGA Xilinx Virtex II Pro. A arquitetura de um *blade* do Cray XD1 pode ser vista na figura 2.6. É possível observar que o dispositivo reconfigurável tem acesso direto a quatro bancos de memória QDR II SRAM, que possuem 4MB cada. O RapidArray Processor permite que os processadores enviem dados para o FPGA e que o FPGA leia dados da DRAM (28; 29; 30). No desenvolvimento de aplicações híbridas para este sistema, existem duas questões chaves que devem ser observadas: a transferência de dados entre os dispositivos e o uso eficiente dos diferentes níveis de memória disponíveis no sistema.

2.3.1 Hierarquia de Memória

Conforme a figura 2.6a, o FPGA de um *blade* do XD1 tem acesso a diferentes tipos de memória. Além dos quatro blocos de SRAM e da memória DRAM, o FPGA pode utilizar blocos de memória internos ao hardware reconfigurável.

O uso destes três níveis de memória deve ser planejado no desenvolvimento de uma aplicação, para evitar a redução de eficiência pelo gargalo no acesso à memória. Cada recurso disponível tem forma de acesso e quantidades distintas. A DRAM é o mais alto nível, com a maior quantidade de memória disponível e com latência de leitura não constante. Esta memória pode ser acessada usando uma interface de comunicação disponibi-

Figura 2.5: Cray XD1. Fonte: (27)



(a) Diagrama (b) Visão lógica. Fonte: (27)

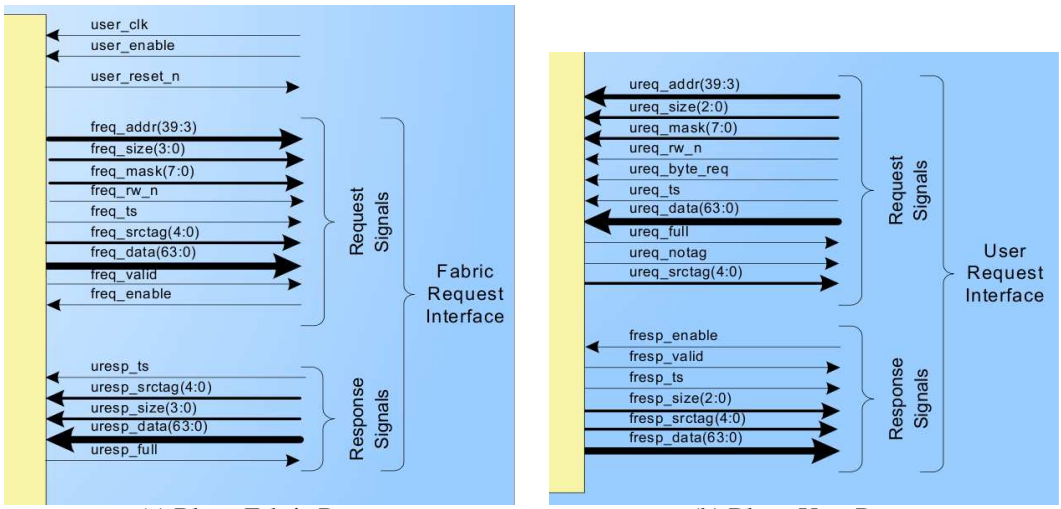
Figura 2.6: Arquitetura do Blade do Cray XD1

lizada pela Cray, o *RapidArray Transport Core*. Em um nível mais baixo estão os bancos de memória QDR II SRAM com 4MB cada. A latência de acesso a esses bancos é de 8 ciclos para a leitura, e o acesso é feito utilizando o *QDR II SRAM Core*, também disponibilizado pelo fabricante (28). Além desses, a família de FPGAs Vitex Pro possuem bancos de memória internos que podem ser acessados diretamente em um único ciclo.

Normalmente a DRAM é utilizada para compartilhar dados com o FPGA, sendo carregados para a QDR II SRAM para serem acessados durante a execução da aplicação. A memória interna do FPGA, disponível em menor quantidade, é utilizada para registradores e cache de dados. Cada problema necessitará de recursos e formas de acesso diferentes que devem ser otimizadas conforme a necessidade.

2.3.2 Comunicação

A comunicação entre CPU e FPGA também é um elemento chave que precisa ser considerado durante o desenvolvimento de aplicações híbridas. Para a comunicação entre dispositivos em um *blade*, a Cray disponibiliza a API RapidArray Transport Core, que é um componente usado na descrição do algoritmo em VHDL. Esta entidade é composta por dois blocos denominados Fabric Request e User Request, que permitem duas abordagem de comunicação (29). A figura 2.7 mostra um diagrama com os sinais destes blocos.



(a) Bloco Fabric Request (b) Bloco User Request

Figura 2.7: Blocos do *RapidArray Transport Core*. Fonte: (29)

A abordagem que utiliza o bloco Fabric Request é denominada *push* e permite que o programa executado nos processadores envie e requisite dados para o FPGA. Nesta abordagem a CPU é responsável por gerenciar a comunicação, ficando ocupada durante a transferência de dados.

Para permitir a comunicação, a Cray disponibiliza a biblioteca *enlib*, a qual abstrai ao programa o FPGA como um arquivo. Deste modo, a transferência de dados entre os processadores e o FPGA é realizada através de leituras e gravações pelo programa em C neste 'arquivo'. Com a realização de uma leitura ou escrita, o FPGA recebe, através do RapidArray Transport Core, uma requisição que deve ser tratada pela aplicação do FPGA. Em caso de leitura, deverá ser retornado um valor ao RapidArray Transport Core para que ocorra o retorno da função chamada pelo programa em linguagem de alto nível. Somente é permitida a manipulação de um *quadword* (64 bits) por requisição utilizando o bloco Fabric Request (29). O sinais denominados *Request Signals* (figura 2.7a) são ativados

quando uma requisição é feita pelo programa, enquanto que através dos sinais do grupo *Response Signals* o FPGA envia dados para o programa.

A segunda abordagem utiliza o bloco User Request e é conhecida como *pull*. Diferentemente da Fabric Request, mantém os processadores livres durante a transferência de dados entre o programa em C e o FPGA. Para isso, este bloco permite que o FPGA realize leituras e escritas em um espaço de memória compartilhado do programa. O endereço para a região de memória, que é compartilhada utilizando a *einlib*, é enviado ao FPGA através do bloco Fabric Request. Estando disponível o endereço, a aplicação descrita para o FPGA é capaz de fazer até 32 requisições sequenciais ao RapidArray Transport Core. Cada requisição pode solicitar até 8 posições contíguas da memória do programa. Os retornos das solicitações ao RapidArray Transporte Core não são necessariamente na ordem em que foram realizadas. O *core* garante somente a ordem das 8 posições contíguas de cada requisição (29). O sistema descrito para o FPGA deve ordenar os dados através do auxílio de *tags* disponibilizadas durante a requisição e o retorno.

As requisições são feitas através dos sinais do grupo *Request Sinals* (figura 2.7b) e os dados da memória chegam através dos sinais *Response Signals* do bloco User Request.

2.3.3 Fluxo Geral de Desenvolvimento

O fluxo de desenvolvimento de aplicações híbridas para o Cray XD1 é dividido em duas tarefas (figura 2.9) devido ao duplo paradigma enfrentado para a implementação de aplicações para este tipo de arquitetura.

Na figura 2.9, o fluxo da esquerda representa o desenvolvimento da aplicação para o FPGA. Um importante ponto neste fluxo é a utilização da interface de comunicação que é disponibilizada pelo fabricante do sistema. Como mencionado anteriormente, esta entidade fornece os sinais que devem ser manipulados para realizar a comunicação com os recursos do sistema (memória, CPU, FPGAs, etc). Além disso, outras características do FPGA são específicas para o XD1, como frequência, sinais de controle e acesso a bancos de memória. Para facilitar este desenvolvimento, a Cray disponibiliza um *template*, ilutrado na figura 2.8, com a estrutura de componentes necessários para o desenvolvimento da aplicação para o FPGA. Neste *template*, o componente *prog_clk_gen* gerencia os sinais de controle do FPGA (*reset_n*, ready, *clock*, etc.), o componente *rt_core* permite a comunicação com o programa executado em CPU, e o componente *qdr2_core* faz a inter-

face com os quatro blocos de memória DRAM. A aplicação é desenvolvida utilizando o componente *user_app*, que não possui arquitetura (*architecture*) implementada.

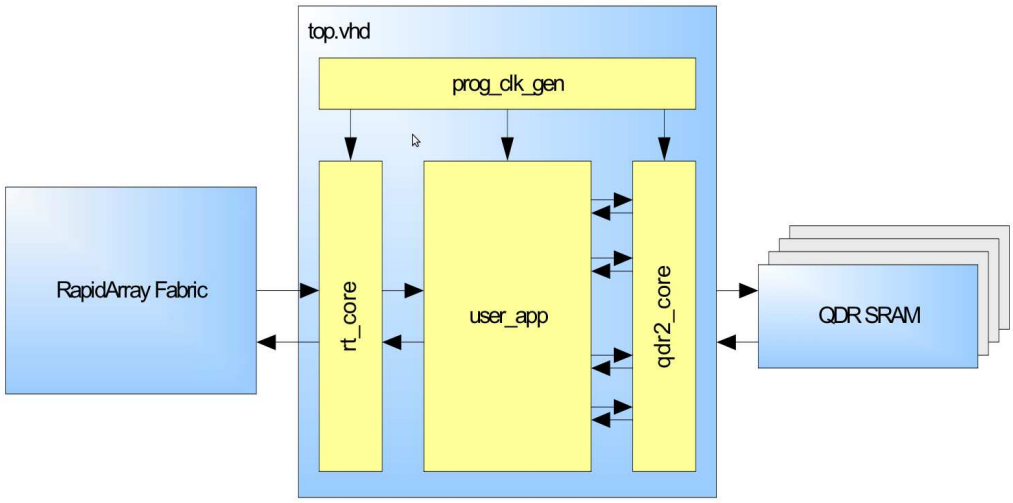


Figura 2.8: *Template* para desenvolvimento no Cray XD1. Fonte: (28)

O fluxo do lado direito define o desenvolvimento da aplicação em linguagem de alto nível. Não existe restrição quanto a linguagem de programação, mas o programa deve ser capaz de ser integrado à biblioteca *einlib* que é diponibilizada em linguagem C. Atualmente compiladores como gcc e intel permitem a utilização de bibliotecas em C por programas Fortran.

Durante a execução, a aplicação deve carregar o arquivo de configuração (*bitstream*) no FPGA e gerenciar, quando necessário, a transferência de dados entre os dispositivos.

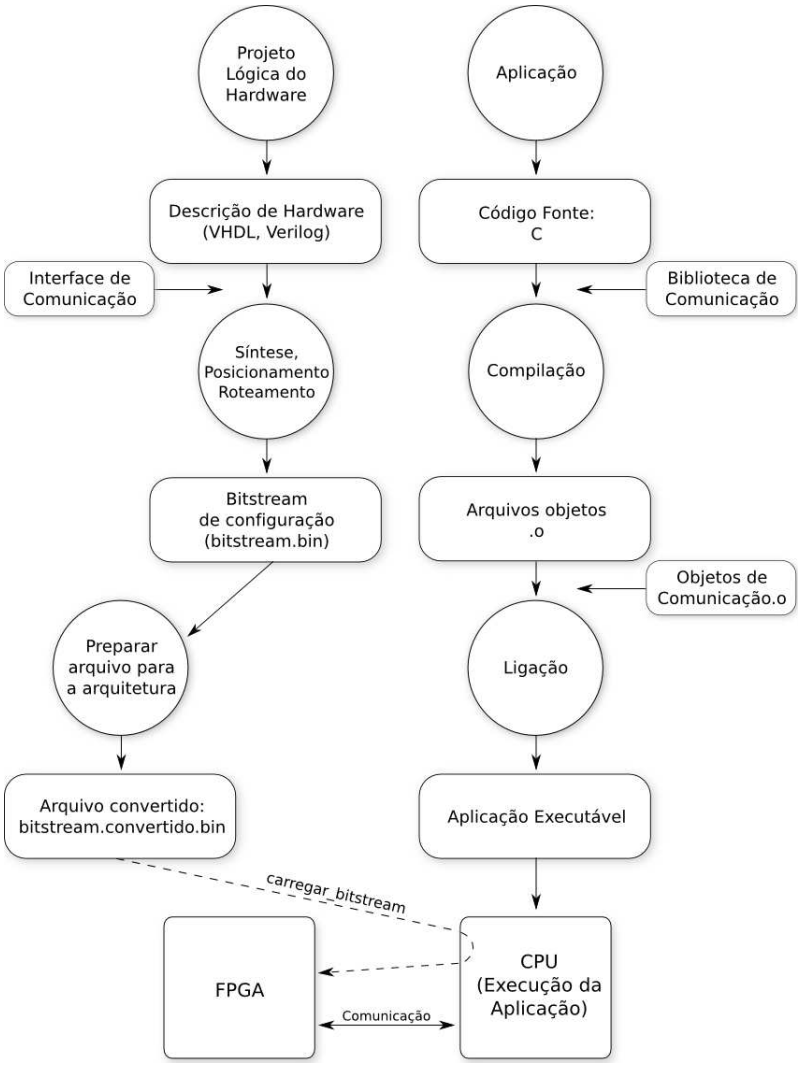


Figura 2.9: Fluxo Geral de Desenvolvimento. Adaptado de (28)

3 PROJETO DA FFT HÍBRIDA E PARALELA

Este capítulo apresenta a descrição da FFT Híbrida e Paralela desenvolvida para sistema computacional híbrido reconfigurável.

3.1 Requisitos

Para o desenvolvimento do projeto de uma FFT paralela para um sistema híbrido reconfigurável foram considerados alguns requisitos. O primeiro é quanto a utilização de ponto flutuante como representação numérica dos dados a serem processados. A representação em ponto fixo, apesar de ser mais simples, tem uma faixa mais estreita de valores em relação a representação em ponto flutuante, o que restringiria a utilização da solução proposta neste trabalho por diversas aplicações. Entretanto, FPGAs não possuem suporte nativo a operações em ponto flutuante, as quais necessitam ser implementadas assim como o restante da lógica da aplicação. Felizmente existem bibliotecas que implementam esta representação e suas operações e que geralmente estão em conformidade com o padrão IEEE754 utilizados pela maioria dos processadores (31).

Outro requisito importante no desenvolvimento de aplicações de alto desempenho, são os custos envolvidos na movimentação de dados. Como mencionado na seção 2.3.2, o Cray XD1 possui duas abordagens (*push* e *pull*) para a realização de comunicação entre CPU e FPGA que, em estudos anteriores (32), apresentaram taxas diferentes de transferência para quantidades distintas de dados. Ambas os valores estão abaixo do limite teórico informado pelo fabricante, e observa-se que a utilização de técnicas para sobrepor estes custos de comunicação com computação devem ser utilizadas em um projeto eficiente.

Além destes requisitos, existe a necessidade de elaborar uma descrição que aproveite o paralelismo próprio de dispositivos reconfiguráveis, pois está é a principal forma de se

obter melhores desempenhos em um dispositivo reconfigurável que irá operar em uma frequência inferior (MHz) a de uma processador de propósito geral (GHz).

3.2 Arquitetura

Considerando os requisitos do projeto da FFT híbrida e paralela, a arquitetura desenvolvida é focada no aproveitamento máximo de oportunidades de paralelismo de operações e tarefas e também na sobreposição de custos de comunicação. Para isso, aproveita o poder computacional de CPU e FPGA combinados para o cálculo da FFT utilizando o algoritmo *binary-exchange* (descrito na seção 2.2.2). Nesta abordagem, cada dispositivo calcula a FFT em uma faixa dos valores de entrada e os passos finais são computados em CPU para evitar maiores custos de transferência de dados.

A arquitetura desenvolvida, considera um único *blade* do XD1, pois este trabalho está focado nas interações entre CPU e FPGA. Para esconder o custo de comunicação, o controle de comunicação é feito pelo FPGA, o qual, devido ao seu paralelismo inerente, pode sobrepor este custo iniciando a computação enquanto os dados são transferidos. Esta solução mantém a CPU livre para colaborar com a computação da FFT. O diagrama de blocos da FFT descrita para FPGA pode ser visto na figura 3.1. Nas subseções a seguir, são descritos os componentes da arquitetura da FFT para FPGA. O código fonte de cada entidade pode ser visto no Apêndice A.

3.2.1 Unidade de Comunicação

A Unidade de Comunicação da figura 3.1 é a interface entre a CPU/DRAM e as demais unidades da arquitetura. Esta entidade é responsável por gerenciar as requisições vindas da CPU, como os sinais de início e os registradores de estado. Além disso, esta unidade é responsável por transferir os fatores de giro da memória do *blade* para a SRAM e o vetor de entrada da FFT.

Para sobrepor o custo da movimentação de dados com computação, esta unidade fornece para a Unidade de Controle um registrador de estado que indica o progresso da cópia dos dados. Desta forma, é possível iniciar a computação da FFT enquanto os dados são transferidos. Ao final da transferência dos dados, esta unidade aguarda um sinal da Unidade de Controle que indica o fim da computação. A partir deste momento, esta unidade copia o resultado da DRAM para a memória do *blade* e altera seu estado para finalizado.

3.2.2 Unidade de Controle

A Unidade de Controle gerencia a computação da FFT. Ela é responsável por endereçar os pontos que serão processados na Unidade de Borboletas. Também é responsável por obter o fator de giro do Pré-Buscador de Fatores de Giro. Na figura 3.2 é mostrado o pseudocódigo desta unidade. Este algoritmo é similar ao utilizado em implementações baseadas em software. Uma importante característica nesta implementação é que esta unidade não espera a finalização de uma operação de borboleta para buscar os próximos dados. Esta abordagem permite realizar operações em paralelo e manter o máximo possível a Unidade de Borboletas ocupada.

3.2.3 Unidade de Borboletas

A Unidade de Borboletas é o núcleo computacional da arquitetura. Esta entidade recebe requisições da Unidade de Controle com dois pontos e um fator de giro para serem computados. Para permitir operações em paralelo, esta unidade possui múltiplos núcleos de borboleta que implementam operações radix-2. Estes núcleos podem operar em paralelo pois não existem dependências entre as operações de borboleta em um único passo da FFT. Para operar com múltiplos núcleos, esta unidade faz a distribuição entre as componentes de borboleta de forma a ter sempre pelo menos uma disponível para receber outra requisição da Unidade de Controle.

Ao final da computação, cada núcleo de borboleta gera dois novos pontos que são enviados para a SRAM.

3.2.4 Controladores de Memória

A arquitetura possui duas unidades que controlam o acesso à memória, chamadas de Pré-Buscador de Fator de Giro e Manipulador de Dados. Essas unidades visam reduzir a latência de acesso aos blocos da SRAM.

O Pré-Buscador de Fator de Giro realiza o pré-carregamento dos fatores de giro. Isto é possível pois os fatores de giro são armazenados na ordem de utilização. Esta técnica reduz de 8 para 1 ciclo o acesso a esses dados.

O Manipulador de Dados gerencia o acesso aos itens intermediários calculados. Seu objetivo é permitir o acesso a duas posições de memória por acesso. Para isso, utiliza dois blocos de SRAM, reduzindo o tempo de acesso aos dados para manter por mais tempo as

borboletas ocupadas.

3.2.5 Multiplexadores

É possível ver na figura 3.1, que existem três unidades distintas que precisam solicitar leituras e escritas para o Manipulador de Dados. Para permitir estas operações concorrentes, são usados dois multiplexadores controlados pela Unidade de Comunicação, a qual muda o fluxo de dados conforme necessário.

3.3 Funcionamento

A computação da FFT utilizando a arquitetura descrita anteriormente acontece da seguinte maneira:

1. O FPGA recebe da CPU o endereço onde se encontram os dados do vetor de entrada;

2. A **Unidade de Comunicação** copia estes dados para os bancos de memória através do **Manipulador de Dados** e inicia a cópia dos fatores de giro;

3. A **Unidade de Controle**, que acompanha o estado da transferência dos fatores de giro, solicita dois pontos para o **Manipulador de Dados** e recebe um fator de giro do **Pré-Buscador de Fator de Giros**;

4. A **Unidade de Controle** envia os dados para a **Unidade de Borboletas** e volta a requisitar dados;

5. A **Unidade de Borboleta** utiliza um de seus núcleos para computar a operação de borboleta e envia o resultado para o **Manipulador de Dados**.

Os itens 3, 4 e 5 se repetem em paralelo até que seja finalizada toda a computação. O item 2 também acontece paralelamente às demais operações até que todos os fatores de giro sejam copiados para o banco de memória do FPGA. Na figura 3.3 é apresentada a linha de tempo (em azul) da computação da FFT em FPGA, onde se observa as operações que acontecem paralelamente. Considerando a linha de execução em CPU (em vermelho), esta figura ilustra o funcionamento da execução híbrida e paralela.

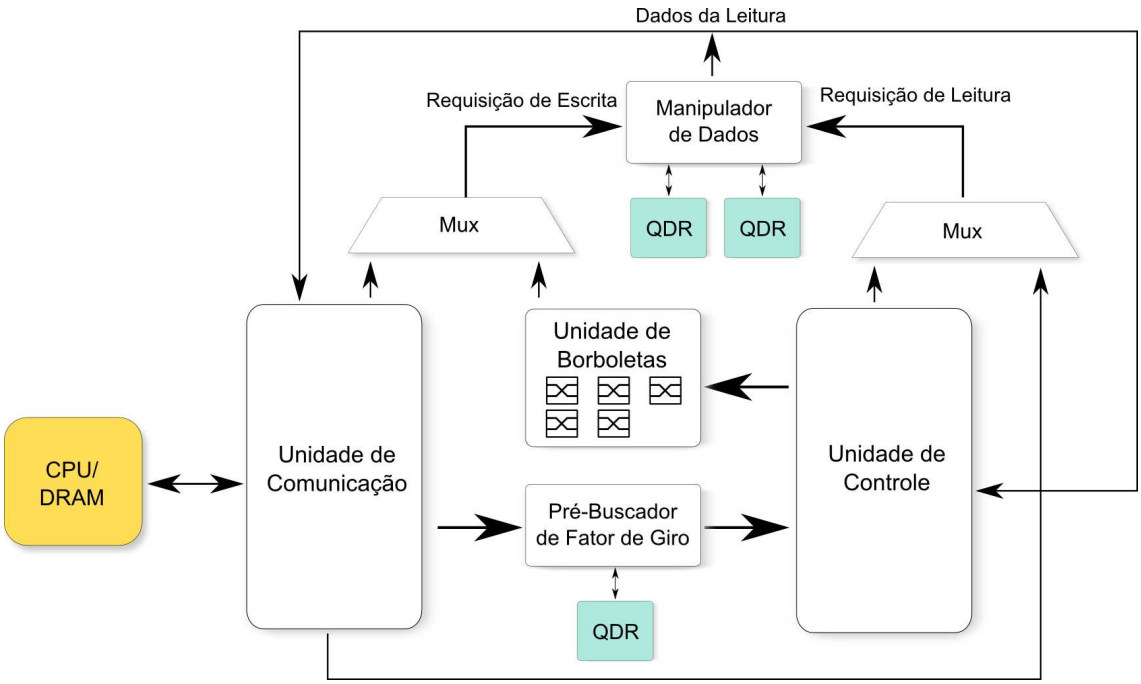


Figura 3.1: Diagrama de blocos da FFT para FPGA

```
for passo from 1 to N by 2
   for j from 0 to passo by 1
      leia fator de giro
      for i from j to n by passo*2
         leia ponto[i]
         leia ponto[i + passo]
         enviar pontos para borboletas
      end for
   end for
   while borboletas ocupadas do
      aguarde
   end while
end for
```

Figura 3.2: Pseudocódigo da Unidade de Controle

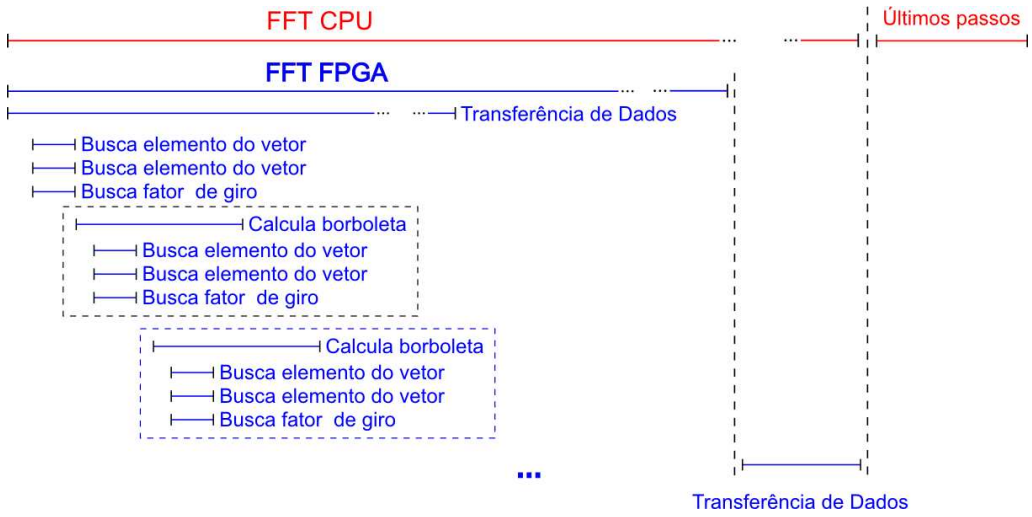


Figura 3.3: Linha de Tempo da computação da FFT Híbrida

4 IMPLEMENTAÇÃO

Para validar e testar o desempenho da arquitetura desenvolvida neste trabalho, foram implementados em VHDL os componentes descritos no capítulo 3.2. Além disso, foi desenvolvido em linguagem C um conjunto de funções para operar a execução da FFT em FPGA e para calcular a FFT em CPU. A implementação em software da Transformada Rápida de Fourier é usada na execução híbrida (CPU e FPGA) e para realizar comparações de desempenho entre os dispositivos. As próximas seções apresentam detalhes destas implementações para FPGA e CPU.

4.1 Descrição de Hardware

Para o FPGA foram implementadas as unidades descritas na seção 3.2 usando VHDL. O desenvolvimento e as simulações foram feitas utilizando a ambiente de desenvolvimento Ise Foundation 10.1 da Xilinx. Esta ferramenta dá suporte completo ao desenvolvimento de aplicações para o FPGA disponível no XD1 (Virtex II Pro, modelo XC2VP50). Para permitir uma maior flexibilidade da implementação quanto à utilização combinada com software, foi utilizada a representação em ponto flutuante do padrão IEEE 754 para a representação de números complexos. Cada ponto é composto por 32 bits da parte real e 32 bits da parte imaginária. As subseções a seguir descrevem os principais componentes do projeto da FFT para FPGA.

4.1.1 Unidade de Comunicação

A Unidade de Comunicação é implementada pela entidade *fabric_request_control* e sua descrição é apresentada no anexo A.5. Através de uma máquina de estado, este componente realiza o tratamento dos sinais oriundos do *RapidArray Transporte Core*. Além disso, através de sinais de estado indica o início da computação para a Unidade de Con-

trole e envia o resultado final para a memória do *blade* após receber um sinal que indica o fim da operação. Para permitir que a computação aconteça durante a cópia dos dados, esta entidade altera o estado do sinal *start* antes de concluir toda a transferência do vetor e fornece o estado atual da transação através do sinal *fg_count*.

4.1.2 Unidade de Borboletas

A descrição em VHDL da Unidade de Borboletas pode ser vista no anexo A.5. Esta entidade foi implementada com 5 núcleos de borboleta (anexo A.11), onde cada uma realiza a operação borboleta radix-2 usando um somador complexo, um subtrator complexo e um multiplicador complexo, respectivamente os anexos A.13, A.14 e A.12. Não foi possível incorporar mais unidades de borboleta pois esta configuração ocupa atualmente 97% da área total do FPGA. Para as operações em ponto flutuante foi utilizada a biblioteca Xilinx Floating-Point Core que implementa o padrão IEEE 754. Esta escolha foi feita baseada em estudos anteriores (31) que avaliaram a acurácia de bibliotecas que implementam estas operações.

Para a distribuição de tarefas entre os núcleos de borboleta, todas as borboletas são conectadas em um barramento de tarefas e cada um possui um sinal de ocupado. Quando uma requisição de computação chega, cada borboleta que se encontra desocupada avalia se as borboletas anteriores estão ocupadas. Em caso positivo, esta aceita a tarefa e marca seu estado como ocupado. Esta abordagem simplifica a distribuição de tarefas entre as borboletas, enquanto evita o uso de mais recursos do FPGA.

4.1.3 Unidade de Controle

A Unidade de Controle do projeto da FFT para FPGA é implementada utilizando uma máquina com 9 estados para a realização dos dois laços encontrados no algoritmo da FFT e para o controle de estado inicial e final. A implementação desta unidade pode ser vista no anexo A.10.

4.1.4 Manipulador de Dados

O Manipulador de Dados é implementado através da entidade *ram*. Esta entidade, conforme é visto no código do anexo A.8, possui duas máquinas de estado que são responsáveis pela leitura e escrita nos blocos de memória, permitindo que estas operações ocorram de maneira independente e em paralelo. Além disso, como comentado na seção

3.2, esta unidade manipula dois blocos de memória para permitir a leitura de duas posições de memória por vez. Devido o limite de armazenamento dos blocos de memória, a implementação da FFT para FPGA opera em vetore com até 2^{19} pontos.

4.1.5 Pré-Buscador de Fator de Giro

O Pré-Buscador de Fator de Giro é implementado através de três máquinas de estados, duas para realizar a leitura e a escrita no banco de memória e uma para tratar os pedidos de fatores de giro. Conforme apresentado no anexo A.4, a máquina de estado de leitura, realiza uma leitura logo que possível, deixando o valor disponível para ser prontamente entregue pela máquina de estado *pedidos* (linhas 102 até 117) em caso de solicitação. Com esta técnica, é possível reduzir o tempo de acesso a memória percebido pela Unidade de Controle.

4.1.6 Outras Unidades

As demais entidades que compõe a implementação da FFT para FPGA podem ser encontradas nos anexos A.1, A.2 e A.3. De maneira geral, estas entidades realizam uma implementação estrutural através da instanciação de componentes e a conexão de sinais. A organização destas operações em componentes facilita a organização e abstração da descrição do hardware.

4.2 Software

A implementação para CPU foi desenvolvida em linguagem C. Esta implementação é usada para a computação híbrida e também para a execução somente em CPU, que serve como base para comparação de desempenho. O programa é organizado da seguinte forma:

- *soft_fft*: esta função implementa a FFT apresentada no pseudocódigo da figura 3.2 e é utilizada para computar a parte da CPU na execução híbrida. Esta função também é a utilizada na execução somente em CPU para efeitos de comparação. Diferentemente da implementação em FPGA, esta implementação não tem limite para o tamanho do vetor de entrada;

- *fpga_fft_start*: esta função manipula a execução baseada em FPGA. Ela envia o endereço da memória compartilhada entre CPU e FPGA, o tamanho do vetor de entrada e muda o estado de um registrador para indicar o início da FFT em FPGA;

- *fpga_fft_wait*: esta função aguarda a finalização da computação da FFT em FPGA através de um *polling* em um registrador de estado do FPGA;

- *soft_last_steps_fft*: esta função computa os últimos passos de uma FFT de N pontos. Esta função é necessária para juntar os dados da execução híbrida.

Enquanto a execução somente em software utiliza somente a função *soft_fft*, a execução híbrida e paralela chama as funções nesta ordem: *fpga_fft_start*, *soft_fft*, *fpga_fft_wait* e *soft_last_steps_fft*.

Apesar do implementação para FPGA suportar vetores de até 2^{19} pontos, a implementação em software limita a execução híbrida em até 2^{18} pontos. Esta limitação é devido ao tamanho máximo da memória compartilhada através da biblioteca einlib fornecida pela Cray.

5 AVALIAÇÃO

A avaliação da implementação desenvolvida foi feita usando um sistema Cray XD1 disponível no Laboratório Associado de Matemática e Computação Aplicada do Instituto Nacional de Pesquisas Espaciais (LAC/INPE). Essa análise é focada em um único *blade* do XD1 e utiliza uma CPU, pois este trabalho está interessado em conhecer as interações entre CPU e FPGA durante uma execução híbrida. Os testes foram realizados utilizando as implementações descritas na seção 4. Para o programa escrito em linguagem C, foi utilizado o compilador GCC 3.3.3 (compilador *default* no equipamento disponível). A descrição em VHDL foi sintetizada, mapeada e roteada usando as ferramentas do ambiente ISE Foundation 10.1 da Xilinx, implementada em um FPGA Virtex II Pro e executada a 160MHz. A aplicação em software foi executada em uma CPU AMD Opteron 64bits a 2,4GHz.

A análise está organizada em dois conjuntos de testes. No primeiro, investiga-se o particionamento de trabalho entre CPU e FPGA para a solução híbrida e paralela da FFT, com o objetivo de determinar a distribuição de tarefas que permita uma utilização máxima dos dois dispositivos. No segundo conjunto, realiza-se uma comparação entre a implementação somente em software com a solução híbrida. As próximas subseções apresentam e discutem esses experimentos com mais detalhes.

5.1 Particionamento de Carga

Uma maneira simples de particionar carga em um ambiente híbrido reconfigurável é executar a parte computacionalmente intensiva no FPGA e usar o processador somente para o controle da execução. Entretanto, neste caso o poder de computação do processador é desperdiçado (17). O particionamento entre FPGA e CPU deve considerar a utilização máxima dos dispositivos. Esta operação não é uma tarefa trivial e é uma decisão

importante do ponto de vista do desempenho (14).

Para avaliar esta questão para o projeto da FFT desenvolvido neste trabalho, foram medidos os tempos de execução da implementação baseada em FPGA para tamanhos de vetores de entrada de 2^{10} até 2^{18} pontos. É importante notar que esta FFT é executada principalmente em FPGA, sendo que a CPU somente gera os fatores de giro e coordena o início e o fim da computação. Também foram medidos os tempos de execução da aplicação para CPU para entradas de 2^{10} até 2^{20} amostras. Os tempos medidos são apresentados na tabela 5.1. Vale observar que não foram executados testes com a implementação para FPGA com 2^{19} e 2^{20} amostras devido ao limite desta versão, ocasionado pelos recursos disponíveis no sistema, conforme apresentado no capítulo 4.

Observando a tabela 5.1, verifica-se que os tempos de execução no FPGA são sempre maiores que o tempo de execução somente em CPU. Além disso, nota-se, conforme apresentado na figura 5.1, que os tempos medidos para a versão para FPGA formam uma reta com crescimento aproximadamente linear, enquanto a versão para CPU apresenta um aumento acentuado na inclinação da curva em 131.072 amostras (eixo X). Esta variação pode estar associada ao uso da memória cache do processador, que não acontece com a execução em FPGA, o que permite ter um crescimento linear.

Tabela 5.1: Tempos da FFT em CPU e da FFT em FPGA

$log_2 N$	Amostras	Tempo (ms)	
		CPU	FPGA
10	1.024	$0,19$	$0,94$
12	4.096	$0,79$	$3,45$
14	16.384	$4,33$	$14,68$
16	65.536	$20,59$	$63,76$
17	131.072	$47,59$	$132,73$
18	262.144	$191,25$	$276,76$
19	524.288	$461,59$	–
20	1.048.576	$979,56$	–

Com os tempos medidos, foi possível determinar uma comparação de desempenho na execução da FFT para tamanhos variados de amostras entre a implementação para FPGA e para CPU. A relação de tempo entre os tempos pode ser vista na terceira coluna da tabela 5.2. Nesta tabela, também é apresentado o particionamento de carga entre CPU e FPGA

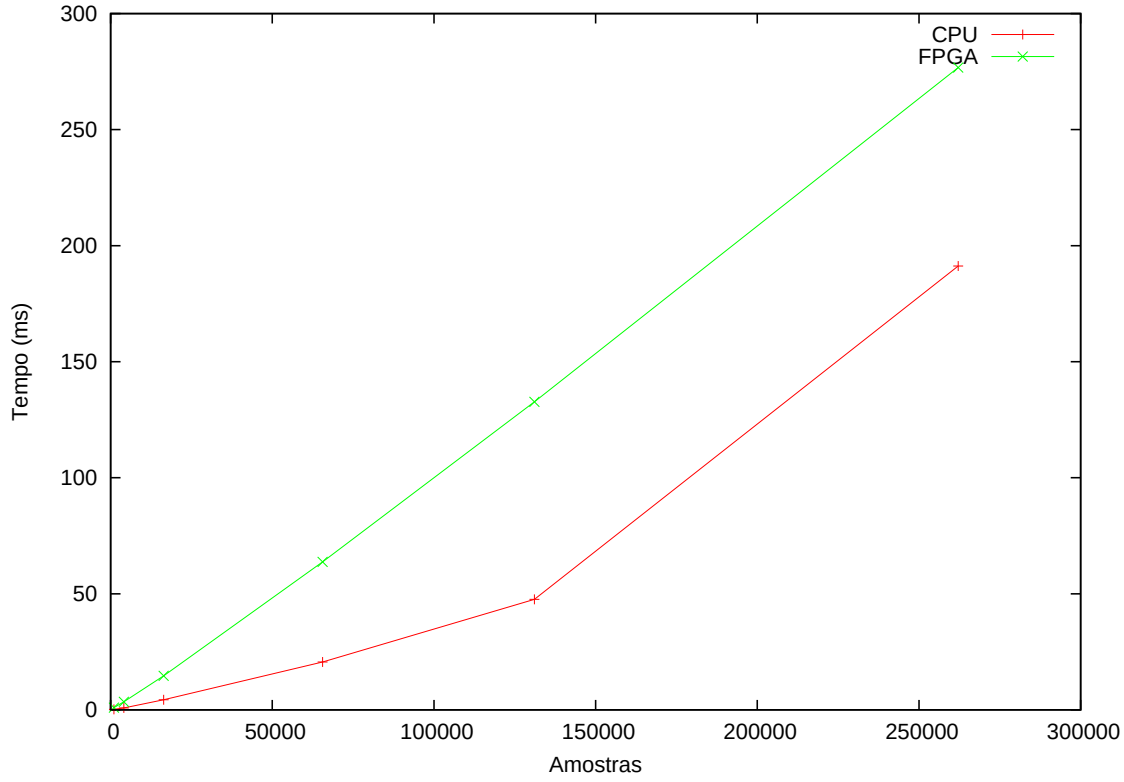


Figura 5.1: Tempos CPU e FPGA

para cada quantidade de pontos do vetor de entrada. Esta decisão de particionamentofoi tomada considerando a razão calculada entre os tempos de cada versão e considerando o tempo de execução de cada parte da aplicação executando em cada dispositivo. O objetivo nesta etapa foi manter os dispositivos ocupados o máximo possível.

Para exemplificar o particionamento apresentado na última coluna da tabela 5.2, consideremos uma execução da FFT com 131.072 pontos, cujo particionamento determinado foi 3:1. Esse particionamento significa que três-quartos da FFT serão executados no processador de propósito geral e um-quarto no FPGA. Tal proporção foi definida considerando o tempo necessário para o cálculo de cada parte da FFT em CPU e em FPGA. No caso da FFT com 131.072 pontos, o particionamento não é 1:1 (65.536 pontos por unidade funcional) pois o tempo de computação de um vetor com 65.536 pontos em FPGA é superior ao tempo da computação total (131.072 pontos) da FFT em software. Esta escolha levaria a computação em $63,76ms$ para um improvável melhor caso, sendo pior que a computação somente em software ($47,58ms$). Esta escolha é ainda validada pela razão entre o tempo em CPU e em FPGA que é de 2,79 vezes.

Para evitar maiores custos de comunicação, os últimos passos da computação da FFT

híbrida são computados em CPU (conforme apresentado na seção 2.2.2). O número de últimos passos calculados em CPU é $log_2(P)$, onde P é o número de partes em que os dados de entrada são divididos. Cada parte de trabalho significa uma FFT de $\frac{N}{P}$ pontos.

Tabela 5.2: Particionamento

$log_2 N$	Amostras	$\frac{\text{Tempo FPGA}}{\text{Tempo CPU}}$	Particionamento
10	1.024	$4,94$	$3:1$
12	4.096	$4,36$	$3:1$
14	16.384	$3,39$	$3:1$
16	65.536	$3,09$	$3:1$
17	131.072	$2,79$	$3:1$
18	262.144	$1,45$	$1:1$
19	524.288	$-$	$1:1$
20	1.048.576	$-$	$3:1$

Considerando ainda a tabela 5.2, podemos observar que no pior caso temos a execução em FPGA aproximadamente 5 vezes mais lenta que a execução em CPU. No melhor caso, a razão está na ordem de 1,45 vezes, sugerindo que com uma quantidade maior de amostras, a FFT em FPGA pode ser mais rápida que em CPU. Este comportamento pode ser explicado através da razão entre o custo de comunicação e o custo de computação da FFT. Para computar a FFT são enviados para o FPGA N pontos do vetor, N fatores de giro e são transferidos do FPGA para a memória da CPU N pontos, totalizando a movimentação de $3N$ pontos, que representam complexidade de $O(N)$ para o custo de comunicação. A complexidade de computação da FFT, conforme já apresentado, é de $O(NlogN)$. A relação entre esses custos ($logN$) representa a razão pela qual a influência do custo de comunicação reduz com o aumento da quantidade de pontos processados.

Ressalta-se ainda que a frequência de operação do FPGA é 15 vezes menor que a frequência da CPU. Este fato comprova uma melhor eficiência do FPGA no cálculo da FFT, considerando a computação da FFT por ciclo de relógio, devido a sua especialização para a computação desta transformada.

5.2 Desempenho da FFT Híbrida e Paralela

Considerando os resultados apresentados na seção anterior, realizou-se uma execução híbrida da FFT utilizando CPU e FPGA usando o particionamento apresentado na tabela

5.2. Devido à divisão de trabalho entre os dispositivos, foi possível realizar testes com vetores de 2^{10} até 2^{20} pontos. A tabela 5.3 apresenta os tempos para a execução somente em CPU e para a versão híbrida. A quinta coluna desta tabela apresenta a aceleração, a qual é obtida pela razão entre o tempo da execução em CPU pelo tempo da execução híbrida ($\frac{\text{Tempo CPU}}{\text{Tempo Hibrido}}$). Existe um ganho de desempenho quando a aceleração é maior que 1, ou seja, o tempo de execução híbrida é menor que o tempo de execução em CPU.

Tabela 5.3: Tempos da FFT em CPU e da FFT em FPGA e *Speedup*

$log_2 N$	Amostras	Tempo (ms)		*Aceleração*
		CPU	Híbrido	
10	1.024	$0,19$	$0,66$	$0,29$
12	4.096	$0,79$	$1,72$	$0,46$
14	16.384	$4,33$	$5,66$	$0,77$
16	65.536	$20,59$	$23,29$	$0,88$
17	131.072	$47,59$	$49,77$	$0,96$
18	262.144	$191,25$	$150,63$	$1,27$
19	524.288	$461,59$	$310,48$	$1,49$
20	1.048.576	$979,56$	$813,57$	$1,20$

Nestes resultados, observa-se que a execução somente em CPU é mais rápida que a execução híbrida para vetores com até 2^{17} pontos. Nota-se que foi obtido aceleração com a versão híbrida e paralela para 2^{18}, 2^{19} e 2^{20} pontos e que existe uma melhora no desempenho da versão híbrida com o aumento da quantidade de amostras de entrada. A redução da aceleração para o teste com 2^{20} pontos é explicado pela granularidade da tarefa, pois neste caso foi utilizado particionamento 3:1 entre CPU e FPGA. Outra distribuição não foi possível pela limitação do tamanho do vetor de entrada na implementação para FPGA.

Na figura 5.2 são apresentadas duas curvas para as acelerações obtidas e uma curva para a aceleração ideal, devendo-se observar que a escala é logarítmica. A curva em vermelho representa os testes utilizando o particionamento 3:1, a curva em verde o particionamento 1:1 e a curva em azul a aceleração ideal, considerando que duas unidades executando a FFT poderia executar esta transformada na metade do tempo da execução usando somente a CPU. Observa-se que a aceleração mantém-se crescente em ambos os casos medidos, tendo uma maior inclinação para a curva verde, que se aproxima mais da aceleração ideal. Um melhor desempenho neste caso é explicado pelo fato do FPGA contribuir mais para a computação da FFT do que com o particionamento 3:1. Observando

as inclinações das curvas das acelerações para ambos os particionamentos, podemos projetar que com o aumento da quantidade de pontos do vetor computado, as acelerações se aproximariam mais da reta de aceleração ideal.

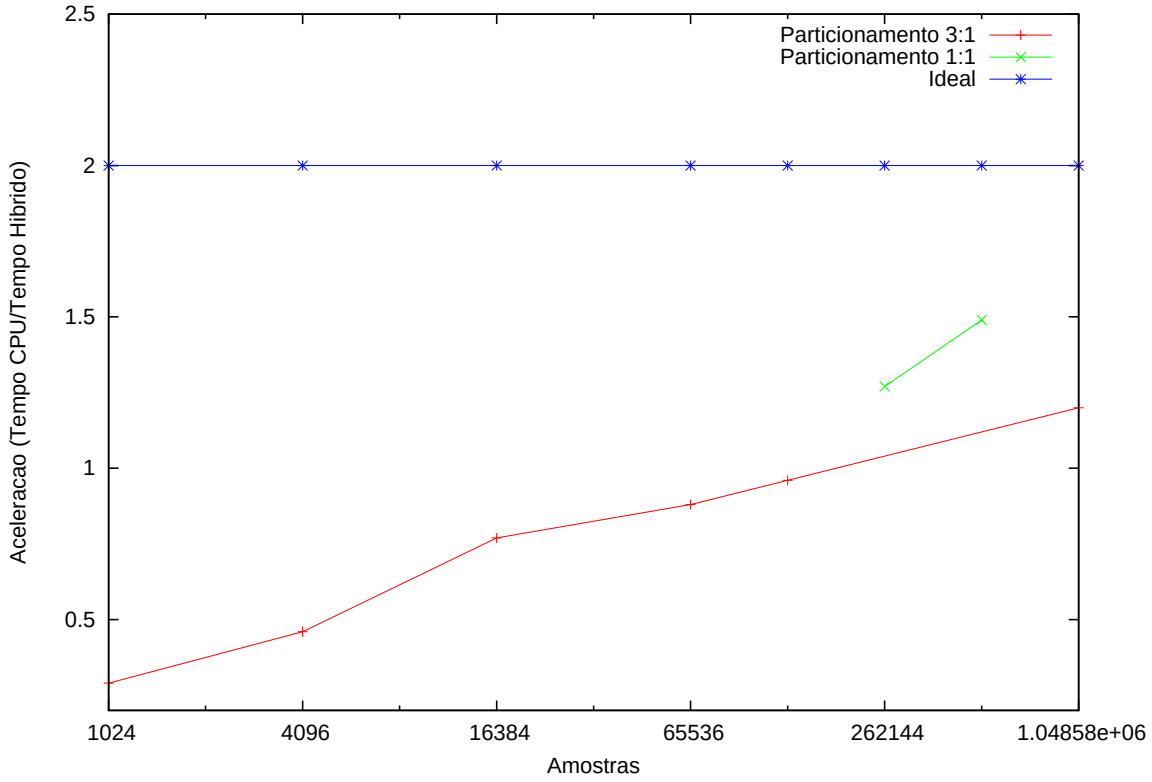


Figura 5.2: Aceleração entre execução em CPU e execução Híbrida

6 CONCLUSÃO

A proposta deste trabalho foi o projeto e a implementação de uma Transformada Rápida de Fourier Paralela para um sistema computacional híbrido reconfigurável. Além disso, teve-se por objetivo o estudo de algoritmos de paralelização da FFT, a exploração de um sistema híbrido reconfigurável e o estabelecimento do perfil de execução da FFT neste tipo de plataforma.

Com o desenvolvimento deste trabalho, confirma-se que a Computação Híbrida Reconfigurável é uma promissora alternativa para a solução de aplicações que exigem de alto desempenho. Além disso, que é possível obter aumento de desempenho para a computação da Transformada Rápida de Fourier neste tipo de arquitetura.

Quanto ao projeto de aplicações para esta arquitetura híbrida, conclui-se que as técnicas chaves são a sobreposição de custos de movimentação de dados e o paralelismo em múltiplos níveis, ambos entre CPU e FPGA e internos ao FPGA. A solução apresentada utiliza CPU e FPGA cooperando na computação da FFT, contrastando com a abordagem usual onde o uso do FPGA supera o da CPU e não são considerados os custos de comunicação.

Como principal contribuição, este trabalho apresenta o projeto e implementação da Transformada Rápida de Fourier para um ambiente híbrido reconfigurável que aproveita-se de técnicas como sobreposição de custos de comunicação, cache, pré-busca em memória, concorrência de acesso à memória, paralelização de operações em FPGA e da FFT em ambiente de memória distribuída.

Através dos resultados, conclui-se que o custo de comunicação ainda é alto em relação ao processamento em FPGA, o que resulta em melhores tempos para vetores com maiores quantidades de amostras.

Alguns trabalhos futuros são propostos com o objetivo de melhorar o desempenho

do projeto e da implementação desenvolvida neste trabalho. A lista a seguir apresenta algumas técnicas que poderão ser incorporadas em uma nova versão.

- Utilização do algoritmo radix-4 na implementação da borboleta, considerando que vetores com poucas amostras são pouco usados em aplicações científicas reais e que esta técnica pode reduzir a utilização de recursos do FPGA, tornando possível a utilização de mais núcleos de borboleta na descrição;
- Desenvolvimento de um núcleo de borboleta com *pipeline* para aumentar o fluxo de dados que um núcleo desta operação poderá computar;
- Desenvolver um componente para a geração dos fatores de giro no FPGA. Esta técnica eliminaria a necessidade de transferir estes dados para o FPGA, reduzindo assim o custo da movimentação de dados;
- Particionamento automático de carga entre CPU e FPGA, que atualmente é feito de forma manual.

REFERÊNCIAS

[1] LYNCH, P. The origins of computer weather prediction and climate modeling. *J. Comput. Phys.*, Academic Press Professional, Inc., San Diego, CA, USA, v. 227, n. 7, p. 3431–3444, 2008. ISSN 0021-9991.

[2] SHAN, A. Heterogeneous processing: a strategy for augmenting moore's law. Http://www.linuxjournal.com/article/8368. Acessado em 10/09/2009. 2006.

[3] FOSTER, I. *Designing and building parallel programs: concepts and tools for parallel software engineering.* [S.l.]: Addison-Wesley, 1995.

[4] CRAD-RS. *Caderno dos Cursos Permanentes.* [S.l.]: SBC, 2006.

[5] WAIN, R.; AL. et. An overview of FPGAs and FPGA programming; initial experiences at Daresbury. In: . Daresbury, Cheshire, UK: Council for the Central Laboratory of the Research Councils, 2004. p. 2–4. ISSN 1362-0207.

[6] MULLIN, L. R.; SMALL, S. G. Four easy ways to a faster fft. *Journal of Mathematical Modelling and Algorithms*, Springer Netherlands, v. 1, n. 3, p. 193–214, 2002. ISSN 1572-9214.

[7] CIPRA, B. A. The best of the 20th century: Editors name top 10 algorithms. *SIAM News*, Society ofr Industrial and Applied Mathematics, v. 33, n. 4, 2000.

[8] AGARWAL, R. C.; GUSTAVSON, F. G.; ZUBAIR, M. A high performance parallel algorithm for 1-D FFT. In: *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 34–40. ISBN 0-8186-6605-6.

[9] GUPTA, A.; KUMAR, V. The scalability of FFT on parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 4, n. 8, p. 922–932, 1993. ISSN 1045-9219.

[10] PALMER, J. M. *The Hybrid Architecture Parallel Fast Fourier Transform (HAPFFT)*. Dissertação (Mestrado) — Brigham Young University, 2005.

[11] HE, H.; GUO, H. The realization of FFT algorithm based on FPGA co-processor. *Intelligent Information Technology Applications, 2007 Workshop on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 3, p. 239–243, 2008.

[12] CHAO, C. et al. Design of a high performance FFT processor based on FPGA. In: *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2005. p. 920–923. ISBN 0-7803-8737-6.

[13] BONDHUGULA, U. et al. Hardware/software integration for FPGA-based all-pairs shortest-paths. In: *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2006. p. 152–164. ISBN 0-7695-2661-6.

[14] KINDRATENKO, V. V.; STEFFEN, C. P.; BRUNNER, R. J. Accelerating scientific applications with reconfigurable computing: Getting started. *Computing in Science and Engg.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 9, n. 5, p. 70–77, 2007. ISSN 1521-9615.

[15] ZHUO, L.; PRASANNA, V. K. High performance linear algebra operations on reconfigurable systems. In: *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005. p. 2. ISBN 1-59593-061-2.

[16] KINDRATENKO, V.; POINTER, D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In: *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2006. p. 13–22. ISBN 0-7695-2661-6.

[17] ZHUO, L.; PRASANNA, V. K. Scalable hybrid designs for linear algebra on reconfigurable computing systems. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 57, n. 12, p. 1661–1675, 2008. ISSN 0018-9340.

[18] KOEHLER, S.; CURRERI, J.; GEORGE, A. D. Performance analysis challenges and framework for high-performance reconfigurable computing. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 34, n. 4-5, p. 217–230, 2008. ISSN 0167-8191.

[19] CHAMBERLAIN, R. D.; LANCASTER, J. M.; CYTRON, R. K. Visions for application development on hybrid computing systems. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 34, n. 4-5, p. 201–216, 2008. ISSN 0167-8191.

[20] SHIRAZI, N.; WALTERS, A.; ATHANAS, P. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In: *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1995. p. 155. ISBN 0-8186-7086-X.

[21] GOKHALE, M.; GRAHAM, P. S. Reconfigurable computing: Accelerating computation with field-programmable gate arrays. In: *Reconfigurable Computing*. [S.l.: s.n.], 2005. ISBN 0-387-26106-0.

[22] UNDERWOOD, K. FPGAs vs. CPUs: trends in peak floating-point performance. In: *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays'*. New York, NY, USA: ACM, 2004. p. 171–180. ISBN 1-58113-829-6.

[23] HAUCK, S.; DEHON, A. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon) (Systems on Silicon)*. [S.l.]: Morgan Kaufmann, 2007. Hardcover. ISBN 0123705223.

[24] COOLEY, J. W.; TUKEY, J. W. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, v. 19, n. 90, p. 297–301, 1965. Disponível em: <http://dx.doi.org/10.2307/2003354>.

[25] BRACEWELL, R. N. *Fourier Transform and Its Applications*. McGraw-Hill Education, 1980. Hardcover. ISBN 0070661960. Disponível em: <http://www.worldcat.org/isbn/0070661960>.

[26] FRIGO, M.; JOHNSON, S. G. Parallel FFTW. Http://www.fftw.org/parallel/parallel-fftw.html. 2009.

[27] Cray Inc. *Cray XD1 Datasheet*. Mendota, MN, USA, 2005.

[28] Cray Inc. *Design of Cray XD1 QDR II SRAM Core*. Mendota, MN, USA, 2005. 3–7 p.

[29] Cray Inc. *Design of Cray XD1 RapidArray Transport Core*. Mendota, MN, USA, 2005. 3–7 p.

[30] Cray Inc. *Cray XD1 System Overview*. Mendota, MN, USA, 2005. 3–7 p.

[31] GOMES, V. C. F.; CHARAO, A.; VELHO, H. C. Avaliação de uma biblioteca de ponto flutuante para fpga no supercomputador cray xd1. In: *WSCAD-CTIC 2008*. [S.l.: s.n.], 2008.

[32] GOMES, V. C. F.; CHARAO, A.; VELHO, H. C. Avaliação de abordagens de comunicação com fpga no supercomputador cray xd1. In: *ERAD 2009 - IC*. [S.l.: s.n.], 2008.

APÊNDICE A CÓDIGOS FONTE

A.1 Pacotes

Listing A.1: Pacote Principal

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use ieee.numeric_std.all;

package user_pkg is

  ------------------------------------------------------------------------
  -- Declare Constants
  ------------------------------------------------------------------------

  constant c_tmpl_base : std_logic_vector(15 downto 0) := x"0008";
  constant c_tmpl_var  : std_logic_vector(7 downto 0)  := x"01";
  constant c_tmpl_rev  : std_logic_vector(7 downto 0)  := x"01";
  constant c_tmpl_comp : std_logic_vector(7 downto 0)  := x"06";


end package user_pkg;

package body user_pkg is
end package body user_pkg;
```

Listing A.2: Pacote da Biblioteca FFT

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use ieee.numeric_std.all;

------------------------------------------------------------------------------
-- basic_pkg
-- Pacote com Tipos e constants usadas na implementacao da FFT
------------------------------------------------------------------------------

package basic_pkg is
   -- Tipo complexo
   type t_complex is
      record
         creal       : std_logic_vector(31 downto 0);
         cimag       : std_logic_vector(31 downto 0);
      end record t_complex;

   -- Tipo para requisicao para o modulo borboleta_control.vhd
   type t_req_borboleta_control is
      record
         in_1           : t_complex;                     -- Entrada 1
         in_2           : t_complex;                     -- Entrada 2
         fg             : t_complex;                     -- Fator de giro
         addr_in_1      : std_logic_vector(19 downto 0); -- Endereco destino do 1'
         addr_in_2      : std_logic_vector(19 downto 0); -- Endereco destino do 2'
         enable         : std_logic;                     -- Habilita a requisicao
      end record t_req_borboleta_control;

   -- Tipo para requisicao de leitura para o modulo ram.vhd
   type t_req_read_ram is
      record
         addr_r_1     : std_logic_vector(19 downto 0);    -- Endereco de leitura 1
         addr_r_2     : std_logic_vector(19 downto 0);    -- Endereco de leitura 2
         enable       : std_logic;                        -- 1 para habilitar a requisicao
      end record t_req_read_ram;

   -- Tipo de resposta do modulo ram.vhd
   type t_resp_read_ram is
      record
         data_1         : t_complex;                     --std_logic_vector(63 downto 0); -- dados do endereco 1
```

```
        data_2          : t_complex;                    --std_logic_vector(63 downto 0); -- dados do endereco 2
        done            : std_logic;                    -- 1 para dados validos
    end record t_resp_read_ram;

-- Tipo para solicitacao de escrita no modulo ram.vhd
type t_req_write_ram is
    record
        addr            : std_logic_vector(19 downto 0);  -- Endereco para escrita
        data            : t_complex;                      -- Dados
        enable          : std_logic;                      -- 1 para habilitar requisicao
    end record t_req_write_ram;

-- Tipo para solicitacao de fator de giro
type t_req_fator_giro is
    record
        posicao         : std_logic_vector(19 downto 0);  -- Posicao do fator de giro
        enable          : std_logic;
    end record t_req_fator_giro;

-- Tipo para resposta do fator de giro
type t_resp_fator_giro is
    record
        fator_giro      : t_complex;                      -- O fator de giro
        done            : std_logic;                      -- Fator de giro pronto
    end record t_resp_fator_giro;

-- Tipo para solicitacao de gravacao no fator de giro
type t_req_write_fator_giro is
    record
        posicao         : std_logic_vector(19 downto 0);
        data            : t_complex;
        enable          : std_logic;
    end record t_req_write_fator_giro;

-- Constant da latencia de leitura da QDRAM (8 ciclos)
constant c_latencia_qdr : std_logic_vector(3 downto 0) := "1000";

end package basic_pkg;

package body basic_pkg is
-- Vazio
end package body basic_pkg;
```

A.2 Componentes Principais

Listing A.3: Componente Principal

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.user_pkg.all;
library basic;
use basic.basic_pkg.all;


entity user_app is
  port (
    -- Global signals
    reset_n         : in  std_logic;
    user_clk        : in  std_logic;
    user_enable     : in  std_logic;
    rt_ready        : in  std_logic;
    qdr_ready       : in  std_logic;
    -- QDR II RAM Interface
    -- RAM 1
    dr_1            : in  std_logic_vector(71 downto 0);  -- read data
    ar_1            : out std_logic_vector(19 downto 0);  -- read address
    aw_1            : out std_logic_vector(19 downto 0);  -- write address
    r_n_1           : out std_logic;        -- Read strobe
    w_n_1           : out std_logic;        -- Write strobe
    bw_n_1          : out std_logic_vector(7 downto 0);   -- byte write
    dw_1            : out std_logic_vector(71 downto 0);  -- write data
    -- RAM 2
    dr_2            : in  std_logic_vector(71 downto 0);  -- read data
    ar_2            : out std_logic_vector(19 downto 0);  -- read address
    aw_2            : out std_logic_vector(19 downto 0);  -- write address
    r_n_2           : out std_logic;        -- Read strobe
    w_n_2           : out std_logic;        -- Write strobe
    bw_n_2          : out std_logic_vector(7 downto 0);   -- byte write
    dw_2            : out std_logic_vector(71 downto 0);  -- write data
    -- RAM 3
    dr_3            : in  std_logic_vector(71 downto 0);  -- read data
    ar_3            : out std_logic_vector(19 downto 0);  -- read address
    aw_3            : out std_logic_vector(19 downto 0);  -- write address
    r_n_3           : out std_logic;        -- Read strobe
    w_n_3           : out std_logic;        -- Write strobe
    bw_n_3          : out std_logic_vector(7 downto 0);   -- byte write
    dw_3            : out std_logic_vector(71 downto 0);  -- write data
    -- RAM 4
    dr_4            : in  std_logic_vector(71 downto 0);  -- read data
    ar_4            : out std_logic_vector(19 downto 0);  -- read address
    aw_4            : out std_logic_vector(19 downto 0);  -- write address
```

```
    r_n_4          : out std_logic;       -- Read strobe                                  47
    w_n_4          : out std_logic;       -- Write strobe                                 48
    bw_n_4         : out std_logic_vector(7 downto 0);   -- byte write                    49
    dw_4           : out std_logic_vector(71 downto 0);  -- write data                    50
    -- RT Interface                                                                        51
    -- User Request Interface                                                              52
    ureq_srctag    : in  std_logic_vector(4 downto 0);   -- request srctag                 53
    ureq_full      : in  std_logic;       -- request buffer full                          54
    ureq_notag     : in  std_logic;       -- request out of source tags                   55
    ureq_addr      : out std_logic_vector(39 downto 3);  -- request address               56
    ureq_size      : out std_logic_vector(2 downto 0);   -- request size                  57
    ureq_mask      : out std_logic_vector(7 downto 0);   -- request byte mask             58
    ureq_rw_n      : out std_logic;       -- request read/write                           59
    ureq_data      : out std_logic_vector(63 downto 0);  -- request write data            60
    ureq_ts        : out std_logic;       -- request transfer start                       61
    ureq_byte_req  : out std_logic;       -- request type (byte or double word)           62
    -- Fabric Response Interface                                                           63
    fresp_valid    : in  std_logic;       -- response signals valid                       64
    fresp_ts       : in  std_logic;       -- response transfer start                      65
    fresp_size     : in  std_logic_vector(2 downto 0);   -- response size                 66
    fresp_srctag   : in  std_logic_vector(4 downto 0);   -- response srctag               67
    fresp_data     : in  std_logic_vector(63 downto 0);  -- response data                68
    fresp_enable   : out std_logic;       -- enable responses                             69
    -- Fabric Request Interface                                                            70
    freq_addr      : in  std_logic_vector(39 downto 3);  -- request address               71
    freq_size      : in  std_logic_vector(3 downto 0);   -- request size                  72
    freq_srctag    : in  std_logic_vector(4 downto 0);   -- request srctag                73
    freq_mask      : in  std_logic_vector(7 downto 0);   -- request byte mask             74
    freq_rw_n      : in  std_logic;       -- request read/write                           75
    freq_ts        : in  std_logic;       -- request transfer start                       76
    freq_valid     : in  std_logic;       -- request valid                                77
    freq_data      : in  std_logic_vector(63 downto 0);  -- request write data            78
    freq_enable    : out std_logic;       -- enable request interface                     79
    -- User Response Interface                                                             80
    uresp_full     : in  std_logic;       -- response buffer full                         81
    uresp_ts       : out std_logic;       -- response transfer start                      82
    uresp_size     : out std_logic_vector(3 downto 0);   -- response size                 83
    uresp_srctag   : out std_logic_vector(4 downto 0);   -- response srctag               84
    uresp_data     : out std_logic_vector(63 downto 0)   -- response data                 85
    );                                                                                     86
                                                                                           87
end entity user_app;                                                                       88
                                                                                           89
architecture rtl of user_app is                                                            90
                                                                                           91
  ------------------------------------------------------------------------                 92
  -- Components.                                                                           93
  ------------------------------------------------------------------------                 94
component fg_control is                                                                    95
  port(                                                                                    96
    -- public                                                                              97
    requisicao     : in t_req_fator_giro;                                                  98
    user_clk       : in  std_logic;                                                        99
    reset_n        : in  std_logic;                                                        100
    system_rdy     : in std_logic;                                                         101
    requisicao_w   : in t_req_write_fator_giro;  -- requisicao de gravacao                 102
    resposta       : out t_resp_fator_giro; -- resposta do fator de giro                   103
    cache_miss     : out std_logic_vector(31 downto 0); -- total de cache miss             104
    fg_count       : in std_logic_vector(19 downto 0); -- total de fatores de giros gravados 105
    --private                                                                             106
    -- RAM 3                                                                             107
    dr_3           : in  std_logic_vector(71 downto 0);  -- read data                      108
    ar_3           : out std_logic_vector(19 downto 0);  -- read address                   109
    aw_3           : out std_logic_vector(19 downto 0);  -- write address                  110
    r_n_3          : out std_logic;       -- Read strobe                                  111
    w_n_3          : out std_logic;       -- Write strobe                                 112
    bw_n_3         : out std_logic_vector(7 downto 0);   -- byte write                    113
    dw_3           : out std_logic_vector(71 downto 0);  -- write data                    114
    -- RAM 4                                                                             115
    dr_4           : in  std_logic_vector(71 downto 0);  -- read data                      116
    ar_4            : out std_logic_vector(19 downto 0);  -- read address                  117
    aw_4           : out std_logic_vector(19 downto 0);  -- write address                  118
    r_n_4          : out std_logic;       -- Read strobe                                  119
    w_n_4          : out std_logic;       -- Write strobe                                 120
    bw_n_4         : out std_logic_vector(7 downto 0);   -- byte write                    121
    dw_4           : out std_logic_vector(71 downto 0)  -- write data                     122
  );                                                                                       123
  end component fg_control;                                                                124
  for all: fg_control use entity basic.fg_control(com_cache2);                             125
                                                                                           126
                                                                                           127
component fabric_request_control is                                                        128
  port(                                                                                    129
    -- public                                                                              130
    -- dados                                                                               131
    vector_size    : out std_logic_vector(19 downto 0); -- tamanho do vetor a ser calculada a fft 132
    start          : out std_logic;                     -- iniciar calculo                133
    buf_ptr        : out std_logic_vector(39 downto 3); -- ponteiro destino dos dados da fft 134
    req_w_fg       : out t_req_write_fator_giro;        -- requisicao para gravacao no fator de giro 135
    req_w_ram      : out t_req_write_ram;               -- requisicao para gravacao do vetor na ram 136
    req_r_ram      : out t_req_read_ram; -- requisicao de leitura da ram;                  137
    resposta_ram   : in t_resp_read_ram;                                                   138
    fft_done       : in std_logic;                                                         139
    select_mux_w_ram : out std_logic;                                                      140
    select_mux_r_ram : out std_logic;                                                      141
    requisicao_fg  : out t_req_fator_giro;                                                 142
    resposta_fg    : in t_resp_fator_giro; -- debug                                        143
    cache_miss     : in std_logic_vector(31 downto 0); -- total de cache miss              144
    fft_debug      : in std_logic_vector(31 downto 0);                                     145
```

```
      rodadas_borb    : in std_logic_vector(31 downto 0);
      -- controle
      user_clk        : in  std_logic;
      reset_n         : in  std_logic;
      user_enable     : in std_logic;
      rt_ready        : in std_logic;
      qdr_ready       : in std_logic;
      fg_count        : out std_logic_vector(19 downto 0); -- total de fatores de giros gravados
      last_cicle      : in std_logic; -- indica que o fft control esta no ultimo ciclo
      borb_req_w_ram : in t_req_write_ram; -- requisicao de gravacao na ram pela borboleta (baseado no protocolo de
          coerencia de cache Snoopy)
      -- private
      -- Fabric Request Interface
      freq_addr      : in  std_logic_vector(39 downto 3);  -- request address
      freq_size      : in  std_logic_vector(3 downto 0);   -- request size
      freq_srctag    : in  std_logic_vector(4 downto 0);   -- request srctag
      freq_mask      : in  std_logic_vector(7 downto 0);   -- request byte mask
      freq_rw_n      : in  std_logic;      -- request read/write
      freq_ts        : in  std_logic;      -- request transfer start
      freq_valid     : in  std_logic;      -- request valid
      freq_data      : in  std_logic_vector(63 downto 0); -- request write data
      freq_enable    : out std_logic;      -- enable request interface
      -- User Response Interface
      uresp_full     : in  std_logic;      -- response buffer full
      uresp_ts       : out std_logic;      -- response transfer start
      uresp_size     : out std_logic_vector(3 downto 0);   -- response size
      uresp_srctag   : out std_logic_vector(4 downto 0);   -- response srctag
      uresp_data     : out std_logic_vector(63 downto 0);   -- response data
      -- RT Interface
      -- User Request Interface
      ureq_srctag    : in  std_logic_vector(4 downto 0);   -- request srctag
      ureq_full      : in  std_logic;      -- request buffer full
      ureq_notag     : in  std_logic;      -- request out of source tags
      ureq_addr      : out std_logic_vector(39 downto 3);  -- request address
      ureq_size      : out std_logic_vector(2 downto 0);   -- request size
      ureq_mask      : out std_logic_vector(7 downto 0);   -- request byte mask
      ureq_rw_n      : out std_logic;      -- request read/write
      ureq_data      : out std_logic_vector(63 downto 0); -- request write data
      ureq_ts        : out std_logic;      -- request transfer start
      ureq_byte_req : out std_logic;      -- request type (byte or double word)
      -- Fabric Response Interface
      fresp_valid    : in  std_logic;      -- response signals valid
      fresp_ts       : in  std_logic;      -- response transfer start
      fresp_size     : in  std_logic_vector(2 downto 0);   -- response size
      fresp_srctag   : in  std_logic_vector(4 downto 0);   -- response srctag
      fresp_data     : in  std_logic_vector(63 downto 0); -- response data
      fresp_enable   : out std_logic      -- enable responses
   );
end component fabric_request_control;
   for all: fabric_request_control use entity basic.fabric_request_control(basic);

component multiplexador_write_ram is
   port(
      entrada0    : in t_req_write_ram;
      entrada1    : in t_req_write_ram;
      saida       : out t_req_write_ram;
      seletor     : in std_logic
   );
end component multiplexador_write_ram;
   for all: multiplexador_write_ram use entity basic.multiplexador_write_ram(basic);

component multiplexador_read_ram is
   port(
      entrada0    : in t_req_read_ram;
      entrada1    : in t_req_read_ram;
      saida       : out t_req_read_ram;
      seletor     : in std_logic
   );
end component multiplexador_read_ram;
   for all: multiplexador_read_ram use entity basic.multiplexador_read_ram(basic);

component ram is
   port(
      -- public
      requisicao_r       : in  t_req_read_ram;
      requisicao_w       : in  t_req_write_ram;
      resposta           : out t_resp_read_ram;
      user_clk           : in  std_logic;
      reset_n            : in  std_logic;
      --private
      -- RAM 1
      dr_1            : in  std_logic_vector(71 downto 0);  -- read data
      ar_1            : out std_logic_vector(19 downto 0);  -- read address
      aw_1            : out std_logic_vector(19 downto 0);  -- write address
      r_n_1           : out std_logic;      -- Read strobe
      w_n_1           : out std_logic;      -- Write strobe
      bw_n_1          : out std_logic_vector(7 downto 0);   -- byte write
      dw_1            : out std_logic_vector(71 downto 0);  -- write data
      -- RAM 2
      dr_2            : in  std_logic_vector(71 downto 0);  -- read data
      ar_2             : out std_logic_vector(19 downto 0);  -- read address
      aw_2            : out std_logic_vector(19 downto 0);  -- write address
      r_n_2           : out std_logic;      -- Read strobe
      w_n_2           : out std_logic;      -- Write strobe
      bw_n_2          : out std_logic_vector(7 downto 0);   -- byte write
      dw_2            : out std_logic_vector(71 downto 0)  -- write data

   );
end component ram;
```

```
   for all: ram use entity basic.ram(basic);


component borboleta_control is
   port(
      -- public
      requisicao        : in  t_req_borboleta_control;
      busy              : out std_logic;
      user_clk          : in  std_logic;
      reset_n           : in  std_logic;
      -- private
      req_w_ram         : out t_req_write_ram; -- requisicao de gravacao na ram
      rodadas           : out std_logic_vector(31 downto 0);
      working           : out std_logic
   );
end component borboleta_control;
   for all: borboleta_control use entity basic.borboleta_control(quintupla);

component fft_control is
   port(
      user_clk        : in std_logic;
      reset_n         : in std_logic;
      -- registradores
      vector_size     : in std_logic_vector(19 downto 0);
      select_mux_r_ram : out std_logic;
      start           : in std_logic;
      borb_busy       : in std_logic;
      done            : out std_logic;
      last_cicle      : out std_logic;
      req_read_ram    : out t_req_read_ram;
      resposta_ram    : in t_resp_read_ram;
      req_read_fg     : out t_req_fator_giro;
      resposta_fg     : in t_resp_fator_giro;
      req_borb        : out t_req_borboleta_control;
      start_dma       : out std_logic;
      fft_debug       : out std_logic_vector(31 downto 0);
      borb_ctrl_working : in std_logic
   );
end component fft_control;
   for all: fft_control use entity basic.fft_control(basic);



   -- sinais do fg
   signal s_next_fg           : std_logic;
   signal s_requisicao_w_fg   : t_req_write_fator_giro;
   signal s_resposta_fg       : t_resp_fator_giro;
   signal s_cache_miss        : std_logic_vector(31 downto 0);
   -- saidas do fabrig_request
   signal s_vector_size       : std_logic_vector(19 downto 0);
   signal s_start             : std_logic;
   signal s_buf_ptr           : std_logic_vector(39 downto 3);
   signal s_mux_write_entrada0  : t_req_write_ram;
   signal s_mux_write_entrada1  : t_req_write_ram;
   signal s_select_mux_write_ram   : std_logic;
   signal s_requisicao_fg     :  t_req_fator_giro;
   signal s_requisicao_r_ram : t_req_read_ram;
   signal s_requisicao_w_ram : t_req_write_ram;
   signal s_fg_count         : std_logic_vector(19 downto 0);
   -- saida fft_control
   signal s_fft_done          : std_logic;
   signal s_mux_read_ram0  : t_req_read_ram;
   signal s_last_cicle     : std_logic;
   -- mux
   signal s_mux_ram_saida     : t_req_write_ram;


   -- ram write
   signal s_resposta_ram   : t_resp_read_ram;

   -- mux read
   signal s_select_mux_read_ram   : std_logic;

   -- borb
   signal s_req_borb_ctrl  : t_req_borboleta_control;
   signal s_borb_busy: std_logic;
   signal s_rodadas_borb: std_logic_vector(31 downto 0);
   signal s_borb_working   : std_logic;

   -- fft ctrl
   signal s_mux_read_ram1 : t_req_read_ram;
   signal s_fft_debug      : std_logic_vector(31 downto 0);
   ------------------------------------------------------------------------
   -- Signals.
   ------------------------------------------------------------------------


begin  -- architecture rtl

   ------------------------------------------------------------------------
   -- Instancias
   ------------------------------------------------------------------------
   fg_cntl: fg_control
   port map(
      requisicao     => s_requisicao_fg,
      user_clk       => user_clk,
      reset_n        => reset_n,
      system_rdy     => s_start,
      requisicao_w   => s_requisicao_w_fg,
```

```
      resposta        => s_resposta_fg,
      cache_miss      => s_cache_miss,
      fg_count        => s_fg_count,
      --private
      -- RAM 3
      dr_3            => dr_3,
      ar_3            => ar_3,
      aw_3            => aw_3,
      r_n_3           => r_n_3,
      w_n_3           => w_n_3,
      bw_n_3          => bw_n_3,
      dw_3            => dw_3,
      -- RAM 4
      dr_4            => dr_4,
      ar_4            => ar_4,
      aw_4            => aw_4,
      r_n_4           => r_n_4,
      w_n_4           => w_n_4,
      bw_n_4          => bw_n_4,
      dw_4            => dw_4
   );

   fabric_request_cntl: fabric_request_control
   port map(
      -- public
      -- dados
      vector_size       => s_vector_size,
      start             => s_start,
      buf_ptr           => s_buf_ptr,
      req_w_fg          => s_requisicao_w_fg,
      req_w_ram         => s_mux_write_entrada0,
      req_r_ram         => s_mux_read_ram0,
      resposta_ram      => s_resposta_ram,
      fft_done          => s_fft_done,
      select_mux_w_ram  => s_select_mux_write_ram,
      select_mux_r_ram  => s_select_mux_read_ram,
      requisicao_fg     => open,
      resposta_fg       => s_resposta_fg, -- debug
      cache_miss        => s_cache_miss,
      fft_debug         => s_fft_debug,
      rodadas_borb      => s_rodadas_borb,
      fg_count          => s_fg_count,
      last_cicle        => s_last_cicle,
      borb_req_w_ram    =>s_mux_write_entrada1,
      -- controle
      user_clk          => user_clk,
      reset_n           => reset_n,
      user_enable       => user_enable,
      rt_ready          => rt_ready,
      qdr_ready         => qdr_ready,
      -- private
      -- Fabric Request Interface
      freq_addr         => freq_addr,
      freq_size         => freq_size,
      freq_srctag       => freq_srctag,
      freq_mask         => freq_mask,
      freq_rw_n         => freq_rw_n,
      freq_ts           => freq_ts,
      freq_valid        => freq_valid,
      freq_data         => freq_data,
      freq_enable       => freq_enable,
      -- User Response Interface
      uresp_full        => uresp_full,
      uresp_ts          => uresp_ts,
      uresp_size        => uresp_size,
      uresp_srctag      => uresp_srctag,
      uresp_data        => uresp_data,
      -- RT Interface
      -- User Request Interface
      ureq_srctag       => ureq_srctag,
      ureq_full         => ureq_full,
      ureq_notag        => ureq_notag,
      ureq_addr         => ureq_addr,
      ureq_size         => ureq_size,
      ureq_mask         => ureq_mask,
      ureq_rw_n         => ureq_rw_n,
      ureq_data         => ureq_data,
      ureq_ts           => ureq_ts,
      ureq_byte_req     => ureq_byte_req,
      -- Fabric Response Interface
      fresp_valid       => fresp_valid,
      fresp_ts          => fresp_ts,
      fresp_size        => fresp_size,
      fresp_srctag      => fresp_srctag,
      fresp_data        => fresp_data,
      fresp_enable      => fresp_enable
   );

mux_write_ram: multiplexador_write_ram
   port map(
      entrada0    => s_mux_write_entrada0,
      entrada1    => s_mux_write_entrada1,
      saida       => s_requisicao_w_ram,
      seletor     => s_select_mux_write_ram
   );

mux_read_ram: multiplexador_read_ram
   port map(
      entrada0    => s_mux_read_ram0,
```

```
      entrada1    => s_mux_read_ram1,
      saida       => s_requisicao_r_ram,
      seletor     => s_select_mux_read_ram
   );


borb_ctrl: borboleta_control
   port map (
      -- public
      requisicao        => s_req_borb_ctrl,
      busy              => s_borb_busy,
      user_clk          => user_clk,
      reset_n           => reset_n,
      -- private
      req_w_ram         => s_mux_write_entrada1,
      rodadas           => s_rodadas_borb,
      working              => s_borb_working
   );


fft_ctrl: fft_control
   port map(
   user_clk             => user_clk,
      reset_n           => reset_n,
      -- registradores
      vector_size       => s_vector_size,
      start             => s_start,
      borb_busy         => s_borb_busy,
      done              => s_fft_done,
      last_cicle        => s_last_cicle,
      req_read_ram      => s_mux_read_ram1,
      resposta_ram      => s_resposta_ram,
      req_read_fg       => s_requisicao_fg,
      resposta_fg       => s_resposta_fg,
      req_borb          => s_req_borb_ctrl,
      fft_debug         => s_fft_debug,
      borb_ctrl_working    => s_borb_working
   );



ram_comp: ram
   port map(
      -- public
      requisicao_r      => s_requisicao_r_ram,
      requisicao_w      => s_requisicao_w_ram,
      resposta          => s_resposta_ram,
      user_clk          => user_clk,
      reset_n           => reset_n,
      --private
      -- RAM 1
      dr_1              => dr_1,
      ar_1              => ar_1,
      aw_1              => aw_1,
      r_n_1             => r_n_1,
      w_n_1             => w_n_1,
      bw_n_1            => bw_n_1,
      dw_1              => dw_1,
      -- RAM 2
      dr_2              => dr_2,
      ar_2              => ar_2,
      aw_2              => aw_2,
      r_n_2             => r_n_2,
      w_n_2             => w_n_2,
      bw_n_2            => bw_n_2,
      dw_2              => dw_2
   );


end architecture rtl;
```

Listing A.4: Pré-buscador de Fator de Giro

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;

-------------------------------------------------------------------------------
-- Entidade Fator de Giro
-- Responsavel por buscar e retornar o fator de giro baseado no j e no step
-------------------------------------------------------------------------------

entity fg_control is
   port(
      --public
      requisicao     : in t_req_fator_giro;          -- Requisicao de leitura
      user_clk       : in  std_logic;
      reset_n        : in  std_logic;
      requisicao_w   : in t_req_write_fator_giro;    -- Requisicao de gravacao
      system_rdy     : in std_logic;                 -- Indica que o sistema esta pronto (inicia a carga na
            cache)
      resposta       : out t_resp_fator_giro;        -- Resposta para a requisicao de leitura
```

```
      cache_miss     : out std_logic_vector(31 downto 0);  -- Total de cache miss, o cache miss ocorre quando o fator de giro nao esta' em cache ainda
      fg_count       : in std_logic_vector(19 downto 0);   -- Total de fatores de giros gravados
      --private
      -- RAM 3
      dr_3           : in  std_logic_vector(71 downto 0);  -- Read data
      ar_3           : out std_logic_vector(19 downto 0);  -- Read address
      aw_3           : out std_logic_vector(19 downto 0);  -- Write address
      r_n_3          : out std_logic;                      -- Read strobe
      w_n_3          : out std_logic;                      -- Write strobe
      bw_n_3         : out std_logic_vector(7 downto 0);   -- Byte write
      dw_3           : out std_logic_vector(71 downto 0);  -- Write data
      -- RAM 4
      dr_4           : in  std_logic_vector(71 downto 0);  -- Read data
      ar_4           : out std_logic_vector(19 downto 0);  -- Read address
      aw_4           : out std_logic_vector(19 downto 0);  -- Write address
      r_n_4          : out std_logic;                      -- Read strobe
      w_n_4          : out std_logic;                      -- Write strobe
      bw_n_4         : out std_logic_vector(7 downto 0);   -- Byte write
      dw_4           : out std_logic_vector(71 downto 0)   -- Write data
   );
end entity fg_control;

architecture com_cache2 of fg_control is
   type t_read_state is (read_idle, read_wait_data, read_wait_req); -- Definicao dos estados para a ma'quina de estado
-- RAM 3
   signal s_aw_3      : std_logic_vector(19 downto 0);              -- Write address
   signal s_r_n_3     : std_logic;                                  -- Read strobe
   signal s_w_n_3     : std_logic;                                  -- Write strobe
   signal s_bw_n_3    : std_logic_vector(7 downto 0);               -- Byte write
   signal s_dw_3      : std_logic_vector(63 downto 0);              -- Write data
   signal s_ar_3      : std_logic_vector(19 downto 0);              -- Read adress
   -- saida
   signal s_resposta : t_resp_fator_giro;                           -- Resposta para a requisicao de leitura
   -- controle
   signal s_count_wait     : std_logic_vector(3 downto 0);          -- Contador esperar a ram devido a latencia
   signal s_read_state : t_read_state;                              -- Sinal para a ma'quina de estado
   signal s_cache_miss : std_logic_vector(31 downto 0);             -- Sinal para o cache miss
   signal s_contador : std_logic_vector(19 downto 0);               -- Contador para os enderecos na ram
   signal s_entregar : std_logic;                                   -- Sinal que diz ja' pode entregar a requisicao
   signal s_entregue : std_logic;                                   -- Sinal que diz que foi entregue a requisicao
begin
   main : process(user_clk, reset_n) is
   begin
      if (reset_n = '0') then
         -- Zera os valores
         s_resposta.done   <= '0';
         s_contador <= (others => '0');
         s_entregue <= '0';
      elsif (user_clk'event and user_clk = '1') then
         s_r_n_3 <= '1';                                            -- Desabilita leituras
         s_resposta.done <= '0';
         s_entregue <= '0';
         -- Como o acesso aos valores do fator de giro e' sequencial, essa ma'quina de estado deixa a resposta preparada
         -- e fica em estado de espera ate' que seja liberado a entrega.
         case (s_read_state) is                                     -- Ma'quinda de estado
            when read_idle =>
               if(system_rdy = '1' and fg_count > s_contador) then  -- Carrega fator de giro
                  s_ar_3 <= s_contador;                           -- Seta o endereco de leitura a partir do contador
                  s_r_n_3 <= '0';                                 -- Habilita leituras
                  s_count_wait <= c_latencia_qdr;                 -- Devido a latencia da ram, usa-se esse contador de espera
                  s_read_state <= read_wait_data;
               end if;
            when read_wait_data =>
               -- Se acabou a espera, entao manda como resposta os valores do fator de giro, e ativa a resposta(done)
               if(s_count_wait = 0) then
                  s_resposta.fator_giro.cimag <= dr_3(63 downto 32);
                  s_resposta.fator_giro.creal <= dr_3(31 downto 0);
                  s_resposta.done <= '1';
                  s_read_state <= read_wait_req;
               else
                  s_count_wait <= s_count_wait - 1;
               end if;
            when read_wait_req =>                                  -- Esse estado fica esperando que seja liberada
               if(s_entregar = '1') then                           -- a entrega da resposta.
                  s_resposta.done <= '0';                          -- Assim que liberado, o estado volta a idle e a
                  s_contador <= s_contador + 1;                    -- posicao do contador e' incrementada pois o
                  s_read_state <= read_idle;                       -- proximo acesso sera' sempre a proxima posicao.
                  s_entregue <= '1';
               end if;
               s_resposta.done <= '1';                             -- Diz que a resposta ja' esta pronta
         end case;
      end if;
   end process main;

   -- Processa os pedidos
   pedidos : process(user_clk, reset_n) is
   begin
      if (reset_n = '0') then
         -- Zera os valores
         s_entregar <= '0';
         s_cache_miss <= (others => '0');
      elsif (user_clk'event and user_clk = '1') then
         -- Se a requisicao esta ativa, entao libera para entregar
         if(requisicao.enable = '1') then
            s_entregar <= '1';
         -- Desativa o sinal para entregar se foi entregue
```

```
        elsif(s_entregue = '1') then
           s_entregar <= '0';
        end if;
        -- Se a requisicao esta ativa mas a resposta nao esta pronta(nao esta em cache)
        -- entao ocorre um cache miss
        if(requisicao.enable = '1' and s_resposta.done = '0') then
           s_cache_miss <= s_cache_miss + 1;
        end if;
     end if;
  end process pedidos;

  -- Processo que grava os dados na RAM
  write_qdr : process(user_clk, reset_n) is
  begin
     if (reset_n = '0') then
        s_w_n_3             <= '1';                                  -- Desabilita gravacao
        s_bw_n_3            <= (others => '0');
     elsif (user_clk'event and user_clk = '1') then
        s_w_n_3       <= '1';                                        -- Desabilita escrita
        -- Se a requisicao de escrita esta ativada, entao envia para a RAM 3 os dados
        if(requisicao_w.enable = '1') then
           s_aw_3 <= requisicao_w.posicao;                           -- Endereco
           s_w_n_3 <= '0';                                           -- Habilita gravacao
           s_dw_3 <= requisicao_w.data.cimag & requisicao_w.data.creal; -- Dados gravados
        end if;
     end if;
  end process write_qdr;

  -- Conecta os sinais com a RAM
  ar_3            <= s_ar_3;
  aw_3            <= s_aw_3;
  r_n_3           <= s_r_n_3;
  w_n_3           <= s_w_n_3;
  bw_n_3          <= s_bw_n_3
  dw_3            <= x"00" & s_dw_3;
  -- Conecta o sinal da resposta do fator de giro
  resposta        <= s_resposta;
  -- Conecta os valores do cache miss
  cache_miss      <= s_cache_miss;
end architecture com_cache2;
```

Listing A.5: Unidade de Comunicação

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;

--------------------------------------------------------------------------------
-- fabric_request_control
-- Faz a interface entre a CPU e FPG
--------------------------------------------------------------------------------

entity fabric_request_control is
   port(
      fft_done        : in std_logic;
      vector_size     : out std_logic_vector(19 downto 0); -- Tamanho do vetor a ser calculada a fft
      start           : out std_logic;                     -- Iniciar calculo
      buf_ptr         : out std_logic_vector(39 downto 3); -- Ponteiro destino dos dados da fft
      req_w_fg        : out t_req_write_fator_giro;        -- Requisicao para gravacao no fator de giro
      req_w_ram       : out t_req_write_ram;               -- Requisicao para gravacao do vetor na ram
      req_r_ram       : out t_req_read_ram;                -- Requisicao de leitura da ram;
      resposta_ram    : in t_resp_read_ram;                -- Resposta da requisicao da ram
      select_mux_w_ram : out std_logic;                    -- Seleciona o mux de escrita na ram
      select_mux_r_ram : out std_logic;                    -- Seleciona o mux de leitura na ram
      requisicao_fg   : out t_req_fator_giro;              -- Requisicao de escrita/leitura na memoria de Fator de Giro
      resposta_fg     : in t_resp_fator_giro;              -- Resposta de Fator de Giro
      cache_miss      : in std_logic_vector(31 downto 0); -- Debug: Total de vezes que a unidade de controle
           requisitou um fator de giro e o valor ainda nao estava na cache de fator de giro
      fft_debug       : in std_logic_vector(31 downto 0); -- Debug: Total de ciclos que a unidade de controle pela
           unidade de borboletas estar livre para receber uma nova requisicao
      rodadas_borb    : in std_logic_vector(31 downto 0);  -- Debug: Total de requisicoes recebidas pela unidade de
           borboletas
      fg_count        : out std_logic_vector(19 downto 0);  -- Total de fatores de giros gravados. Permite que a
           unidade de controle inicie a execucao antes de terminar toda a copia de fatores de giro
      -- Controle
      user_clk        : in  std_logic;
      reset_n         : in  std_logic;
      user_enable     : in std_logic;
      rt_ready        : in std_logic;
      qdr_ready       : in std_logic;
      -- Fabric Request Interface
      freq_addr       : in  std_logic_vector(39 downto 3); -- request address
      freq_size       : in  std_logic_vector(3 downto 0);  -- request size
      freq_srctag     : in  std_logic_vector(4 downto 0);  -- request srctag
      freq_mask       : in  std_logic_vector(7 downto 0);  -- request byte mask
      freq_rw_n       : in  std_logic;       -- request read/write
      freq_ts         : in  std_logic;       -- request transfer start
      freq_valid      : in  std_logic;       -- request valid
      freq_data       : in  std_logic_vector(63 downto 0); -- request write data
      freq_enable     : out std_logic;       -- enable request interface
      -- User Response Interface
      uresp_full      : in  std_logic;       -- response buffer full
      uresp_ts        : out std_logic;       -- response transfer start
```

```
      uresp_size    : out std_logic_vector(3 downto 0);   -- response size
      uresp_srctag  : out std_logic_vector(4 downto 0);   -- response srctag
      uresp_data    : out std_logic_vector(63 downto 0);   -- response data
      -- RT Interface
      -- User Request Interface
      ureq_srctag   : in  std_logic_vector(4 downto 0);   -- request srctag
      ureq_full     : in  std_logic;      -- request buffer full
      ureq_notag    : in  std_logic;      -- request out of source tags
      ureq_addr     : out std_logic_vector(39 downto 3); -- request address
      ureq_size     : out std_logic_vector(2 downto 0);   -- request size
      ureq_mask     : out std_logic_vector(7 downto 0);   -- request byte mask
      ureq_rw_n     : out std_logic;      -- request read/write
      ureq_data     : out std_logic_vector(63 downto 0); -- request write data
      ureq_ts       : out std_logic;      -- request transfer start
      ureq_byte_req : out std_logic;      -- request type (byte or double word)
      -- Fabric Response Interface
      fresp_valid   : in  std_logic;      -- response signals valid
      fresp_ts      : in  std_logic;      -- response transfer start
      fresp_size    : in  std_logic_vector(2 downto 0);   -- response size
      fresp_srctag  : in  std_logic_vector(4 downto 0);   -- response srctag
      fresp_data    : in  std_logic_vector(63 downto 0); -- response data
      fresp_enable  : out std_logic      -- enable responses
   );
end entity fabric_request_control;

architecture basic of fabric_request_control is
   -- Estados da ma'quina de estado de manipulacao de registradores
   type t_req_state is ( idle, read, read_blocked, write);
   -- Estados da ma'quina de estados de copia e escreve via User Request da memo'ria do programa para a memo'ria do
       FPGA
   type t_cpy_state is ( cpy_idle, cpy_req_ur, cpy_wait_ur, cpy_wait_fft_done, cpy_read, cpy_wait_data, cpy_done,
       cpy_cheio, cpy_fg, cpy_wait_ur_fg);

   -- Controle
   signal s_state_enable         : std_logic; -- Inidica que o fpga esta' pronto para iniciar a computacao

   -- Fabric Request Interface
   signal s_freq_addr     :  std_logic_vector(39 downto 3); -- request address
   signal s_freq_size     :  std_logic_vector(3 downto 0);   -- request size
   signal s_freq_mask     :  std_logic_vector(7 downto 0);   -- request byte mask
   signal s_freq_rw_n     :  std_logic;      -- request read/write
   signal s_freq_ts       :  std_logic;      -- request transfer start
   signal s_freq_valid    :  std_logic;      -- request valid
   signal s_freq_data     :  std_logic_vector(63 downto 0); -- request write data
   signal s_freq_srctag   :  std_logic_vector(4 downto 0);
   signal s_freq_enable   :  std_logic;      -- s_enable request interface
   -- User Response Interface
   signal s_uresp_full    :  std_logic;      -- response buffer full
   signal s_uresp_ts      :  std_logic;      -- response transfer start
   signal s_uresp_size    :  std_logic_vector(3 downto 0);   -- response size
   signal s_uresp_data    :  std_logic_vector(63 downto 0); -- response data
   signal s_uresp_srctag  :  std_logic_vector(4 downto 0);   -- response source tag

   signal s_ureq_addr     : std_logic_vector(39 downto 3); -- request address
   signal s_ureq_size     : std_logic_vector(2 downto 0);   -- request size
   signal s_ureq_mask     : std_logic_vector(7 downto 0);   -- request byte mask
   signal s_ureq_rw_n     : std_logic;      -- request read/write
   signal s_ureq_data     : std_logic_vector(63 downto 0); -- request write data
   signal s_ureq_ts       : std_logic;      -- request transfer start
   signal s_ureq_byte_req : std_logic;      -- request type (byte or double word)

   signal s_fresp_enable  : std_logic;      -- enable responses

   -- Contador para gravacao de resposta
   signal s_count_buf        : std_logic_vector(39 downto 3);

   -- Registradores
   signal s_app_cfg       : std_logic_vector(63 downto 0); -- Registrado de estado do programa (Envia start)
   signal s_vector_size   : std_logic_vector(19 downto 0); -- Tamanho do vetor que sera' computado
   signal s_start         : std_logic;                     -- Sinal de start
   signal s_buf_ptr       : std_logic_vector(39 downto 3); -- Endereco para leitura/escrita dos dados (in place)
   signal s_buf_ptr2      : std_logic_vector(39 downto 3); -- Endereco para leitura dos Fatores de Giro
   signal s_ciclos        : std_logic_vector(63 downto 0); -- Debug: Total de ciclos. Iniciado em start = 1 e
       finalizado quando done = 1
   signal s_debug         : std_logic_vector(63 downto 0); -- Debug: Inidica estados que o FPGA ja' esteve
   signal s_req_w_fg      : t_req_write_fator_giro;        -- requisicao de gravacao no fator de giro
   signal s_req_w_ram     : t_req_write_ram;               -- requisicao de gravacao na RAM

   -- Controle
   signal s_req_state      : t_req_state; -- Ma'quina de estado
   signal s_cpy_state      : t_cpy_state; -- Ma'quina de estado
   signal s_fft_done       : std_logic;   -- Final da FFT (deve iniciar transferencia de dados)
   signal s_cpy_done       : std_logic;   -- Final da copia dos dados de saida para a memoria do programa (buf_ptr2)
   signal s_cpy_vet_done   : std_logic;   -- Final da copia dos dados de entrada para a memo'ria do FPGA
   signal s_cpy_fg_done    : std_logic;   -- Final da copia dos Fatores de Giro para a memo'ria do FPGA
   signal s_requisicao_fg  : t_req_fator_giro; -- Requisicao de fator de giro
   signal s_init_transf    : std_logic;   -- Auxiliar na copia via User Request (indica que foi inicia a tranferencia
       de um bloco)

   -- Contadores para a co'pia de dados
   signal s_count_addr     : std_logic_vector(39 downto 3);
   signal s_count_ram      : std_logic_vector(19 downto 0);

   -- Sinais de Debug
   signal s_resposta_fg : t_resp_fator_giro; -- Sinal de resposta de fator de giro
   signal s_req_r_ram   : t_req_read_ram; -- Requisicao de leitura da ram;
   signal s_resposta_ram: t_resp_read_ram; -- Resposta da ram (conferencia)
begin
   -- freq_reg
```

```
-- Realiza o armazenamento dos dados do fabric request que serao analizados no prcess acess_control
freq_reg : process(user_clk, reset_n) is
begin -- process freq_reg
   if (reset_n = '0') then
      s_state_enable <= '0';
      s_ciclos             <= (others => '0');
   elsif (user_clk'event and user_clk = '1') then
      -- Fabric request input registers.
      -- Armazena a requisicao
      if (s_freq_enable = '1') then         -- s_freq_enable e' ativado/desativado no processo de controle e se
          conecta diretamenta a freq_enable
         -- Copia sinais do Fabric Request
         s_freq_ts     <= freq_ts;
         s_freq_size   <= freq_size;
         s_freq_srctag <= freq_srctag;
         s_freq_addr   <= freq_addr;
         s_freq_rw_n   <= freq_rw_n;
         s_freq_valid  <= freq_valid;
         s_freq_mask   <= freq_mask;
         s_freq_data   <= freq_data;
      end if;
      -- Sinal de habilitaÃ§ao da maquina de estados
      s_state_enable <= user_enable and rt_ready and qdr_ready;
      s_fft_done <= fft_done;
      -- Contabiliza quantos ciclos foram necessarios para realizar a coputacao da FFT, iniciando em start=1 e
          finalizando em fft_done = 1
      if(s_start = '1' and s_fft_done = '0') then
         s_ciclos <= s_ciclos + 1;
      end if;
   end if;
end process freq_reg;

-- acess_control
-- Realiza o controle de acesso via Fabric Request aos registrados internos
-- Manipula os sinais do Fabric Request
access_control : process(user_clk, reset_n) is
begin
   if (reset_n = '0') then
      s_vector_size        <= (others => '0');
      s_buf_ptr            <= (others => '0');
      s_debug(31 downto 0) <= (others => '0');
      s_requisicao_fg.enable <= '0';
      s_freq_enable <= '0';
      s_req_state <= idle;
   elsif (user_clk'event and user_clk = '1') then
      -- Atualiza sinal de buffer do Fabric Request cheio
      s_uresp_full <= uresp_full;
      -- Limpa sinais de ciclo Ã°nico
      s_uresp_ts <= '0';
      s_requisicao_fg.enable <= '0';
      -- Sinais de debug
      s_debug(0) <= s_fft_done;
      s_debug(1) <= s_cpy_done;
      s_debug(2) <= s_cpy_vet_done;

      if(s_state_enable = '1') then
         -- Ma'quina de estados que manipula as requisiÃ§oes do Fabric Request
         case (s_req_state) is
            when idle =>
               if (s_freq_ts = '1' and s_freq_valid = '1') then        -- Dado valido
                  if (s_freq_rw_n = '1') then                          -- Leitura
                     if (s_uresp_full = '0') then                      -- Buffer vazio
                        s_req_state <= read;
                     else
                        s_req_state <= read_blocked;
                     end if;
                  else                                                 -- Escrita
                     s_req_state <= write;
                  end if;
                  s_freq_enable <= '0';       -- Desabilita novas requisicoes
               else                                                    -- Dado invalido
                  -- Mantem em idle, e mantem habilitada novas requisicoes
                  s_req_state <= idle;
                  s_freq_enable <= '1';
               end if;
            when read =>
               if (s_freq_addr(24)  = '1') then -- Ler sinais de debug fft_debug & rodadas_borb
                  if (s_uresp_full = '0') then
                     s_uresp_data <= fft_debug & rodadas_borb;
                     s_uresp_ts <= '1';
                     s_uresp_size <= "0000";
                     s_uresp_srctag <= s_freq_srctag;
                     -- Habilita novas requisicoes e vai para idle
                     s_req_state <= idle;
                     s_freq_enable <= '1';
                  else
                     s_req_state <= read_blocked;
                  end if;
               elsif (s_freq_addr(26) = '1') then -- Ler registradores
                  if (s_uresp_full = '0') then
                     case (s_freq_addr(6 downto 3)) is
                        when "0000" => -- 0x00UL => APP_CFG
                           s_uresp_data <= s_app_cfg;
                        when "0001" => -- 0x08UL => VECTOR_SIZE
                           s_uresp_data <= x"00000000000" & s_vector_size;
                        when "0010" => -- 0x10UL => BUF_PTR1
                           s_uresp_data <= "000" & x"000000" & s_buf_ptr;
                        when "0011" => -- 0x18UL => CACHE_MISS
                           s_uresp_data <= x"00000000" & cache_miss;
```

```
                        when "0100" => -- 0x20UL => CICLOS
                            s_uresp_data <= s_ciclos;
                        when "0101" => -- 0x28UL => DEBUG
                            s_uresp_data <= s_debug;
                        when "0110" => -- 0x10UL => BUF_PTR1
                            s_uresp_data <= "000" & x"000000" & s_buf_ptr2;
                        when others => null;
                    end case;
                    -- Envia o registrador requisitado
                    s_uresp_ts <= '1';
                    s_uresp_size <= "0000";
                    s_uresp_srctag <= s_freq_srctag;
                    -- Habilita novas requiscoes e vai para idle
                    s_req_state <= idle;
                    s_freq_enable <= '1';
                else
                    s_req_state <= read_blocked;
                end if;
            end if;
        when write =>
            if (s_freq_addr(26)  = '1') then -- Gravar registradores
                -- escrita nos registradores
                case (s_freq_addr(6 downto 3)) is
                    when "0000" => -- 0x00UL => APP_CFG
                        s_app_cfg <= s_freq_data;
                    when "0001" => -- 0x08UL => VECTOR_SIZE
                        s_vector_size <= s_freq_data(19 downto 0);
                    when "0010" => -- 0x010UL => BUF_PTR1
                        if (s_freq_mask(0) = '1') then
                            s_buf_ptr(7 downto 3 ) <= s_freq_data(7 downto 3);
                        end if;
                        if (s_freq_mask(1) = '1') then
                            s_buf_ptr(15 downto 8 ) <= s_freq_data(15 downto 8);
                        end if;
                        if (s_freq_mask(2) = '1') then
                            s_buf_ptr(23 downto 16 ) <= s_freq_data(23 downto 16);
                        end if;
                        if (s_freq_mask(3) = '1') then
                            s_buf_ptr(31 downto 24 ) <= s_freq_data(31 downto 24);
                        end if;
                        if (s_freq_mask(4) = '1') then
                            s_buf_ptr(39 downto 32 ) <= s_freq_data(39 downto 32);
                        end if;
                    when "0110" => -- 0x010UL => BUF_PTR2
                        if (s_freq_mask(0) = '1') then
                            s_buf_ptr2(7 downto 3 ) <= s_freq_data(7 downto 3);
                        end if;
                        if (s_freq_mask(1) = '1') then
                            s_buf_ptr2(15 downto 8 ) <= s_freq_data(15 downto 8);
                        end if;
                        if (s_freq_mask(2) = '1') then
                            s_buf_ptr2(23 downto 16 ) <= s_freq_data(23 downto 16);
                        end if;
                        if (s_freq_mask(3) = '1') then
                            s_buf_ptr2(31 downto 24 ) <= s_freq_data(31 downto 24);
                        end if;
                        if (s_freq_mask(4) = '1') then
                            s_buf_ptr2(39 downto 32 ) <= s_freq_data(39 downto 32);
                        end if;
                    -- when "0011" => -- 0x18UL => CACHE_MISS somente leitura
                    -- when "0100" => -- 0x20UL => CICLOS somente leitura
                    -- when "0101" => -- 0x28UL => DEBUG somente leitura
                    when others => null;
                end case;
            end if;
            -- Habilita novas requiscoes e vai para idle
            s_req_state <= idle;
            s_freq_enable <= '1';
        when read_blocked =>
            if (s_uresp_full = '0') then
                s_req_state <= read;
            else
                s_req_state <= read_blocked;
            end if;
        when others => null;
    end case;
  end if;
end if;
end process access_control;

cpy_read_ram : process(user_clk, reset_n) is
begin
  if (reset_n = '0') then
    s_count_buf <= (others => '0');
    s_count_addr <= (others => '0');
    s_count_ram <= (others => '0');
    s_cpy_state <= cpy_idle;
    s_cpy_done <= '0';
    s_cpy_vet_done <= '0';
    s_debug(63 downto 32) <= (others => '0');
    s_fresp_enable <='0';
  elsif (user_clk'event and user_clk = '1') then
    -- Limpa sinais de ciclo Ãºnico
    s_req_r_ram.enable <= '0';
    s_ureq_ts <= '0';
    s_req_w_ram.enable <= '0';
    s_req_w_fg.enable <= '0';
    -- Ma'quina de estados com trÃªs momentos principais: LÃª vetor de entrada; Aguarda finalizacao da FFT; Envia
        dados para a memo'ria do programa
```

```
case (s_cpy_state) is
   when cpy_idle => -- Aguarda ate' sinal de start
      if (s_start = '1') then
         s_debug(63) <= '1';
         s_cpy_vet_done <= '0';
         s_count_addr <= (others => '0');
         s_count_ram <= (others => '0');
         s_cpy_state <= cpy_req_ur; -- Iniciar copia dos dados
         s_fresp_enable <='1';
      end if;
   when cpy_req_ur => -- Requisita dados (vetor) via User Request ate' que a qtd requisita seja igual ao tamanho do vetor
      if(s_count_addr /= s_vector_size) then
         if(ureq_full = '0' and ureq_notag = '0') then
            s_ureq_addr <= s_buf_ptr + s_count_addr;
            s_ureq_rw_n <= '1'; -- Lendo
            s_ureq_size <= "111";
            s_ureq_ts <= '1';
            s_count_addr <= s_count_addr + 8;
            s_cpy_state <= cpy_wait_ur;
            s_init_transf <= '0';
            s_debug(62) <= '1';
         end if;
         s_debug(61) <= '1';
      else
         -- Co'pia do vetor finalizada, inicia copia dos fatores de giro
         s_cpy_state <= cpy_fg;
         s_cpy_vet_done <= '1';
         s_count_addr <= (others => '0');
         s_count_ram <= (others => '0');
      end if;
   when cpy_wait_ur =>  -- Espera retorno dos dados do User Request
      if(fresp_valid = '1') then
         -- Grava na ram
         s_req_w_ram.addr        <= s_count_ram;
         s_req_w_ram.data.cimag  <= fresp_data(63 downto 32);
         s_req_w_ram.data.creal  <= fresp_data(31 downto 0);
         -- Habilita gravacao
         s_req_w_ram.enable <= '1';
         s_count_ram <= s_count_ram + 1;
         s_init_transf <= '1';
         s_debug(59) <= '1';
      elsif(s_init_transf = '1') then
         s_cpy_state <= cpy_req_ur;
         s_debug(58) <= '1';
      end if;
      s_debug(57) <= '1';
   when cpy_fg => -- requisita fatores de giro via User Request ate' que a qtd requisita seja igual ao tamanho do vetor (que e' igual ao total de fatores de giro)
      if(s_count_addr /= s_vector_size) then
         if(ureq_full = '0' and ureq_notag = '0') then
            s_ureq_addr <= s_buf_ptr2 + s_count_addr;
            s_ureq_rw_n <= '1'; -- Lendo
            s_ureq_size <= "111";
            s_ureq_ts <= '1';
            s_count_addr <= s_count_addr + 8;
            s_cpy_state <= cpy_wait_ur_fg;
            s_init_transf <= '0';
         end if;
      else
         s_cpy_state <= cpy_wait_fft_done;
         s_debug(60) <= '1';
         s_fresp_enable <='0';
      end if;
   when cpy_wait_ur_fg =>
      if(fresp_valid = '1') then
         -- Grava no fator de giro
         s_req_w_fg.posicao      <= s_count_ram;
         s_req_w_fg.data.cimag   <= fresp_data(63 downto 32);
         s_req_w_fg.data.creal   <= fresp_data(31 downto 0);
         -- Habilita gravacao
         s_req_w_fg.enable <= '1';
         s_count_ram <= s_count_ram + 1;
         s_init_transf <= '1';
      elsif(s_init_transf = '1') then
         s_cpy_state <= cpy_fg;
      end if;
   when cpy_wait_fft_done => -- Aguarda ate' que seja finalizada a FFT
      if(fft_done = '1') then
         s_debug(56) <= '1';
         s_cpy_state <= cpy_read;
         s_count_buf <= (others => '0');
      end if;
   when cpy_read => -- LÃª dado da memo'ria do FPGA para enviar para a memo'ria do Programa
      if(s_count_buf /= s_vector_size) then
         s_req_r_ram.addr_r_1 <= s_count_buf(22 downto 3);
         s_req_r_ram.enable <= '1';
         -- Espera ram
         s_cpy_state <= cpy_wait_data;
      else
         s_cpy_state <= cpy_done;
      end if;
   when cpy_wait_data =>  -- Espera o dado da memo'ria do FPGA e envia para a memo'ria do Programa (via User request)
      if(s_resposta_ram.done = '1') then
         if(ureq_full = '0') then
            s_ureq_addr <= s_buf_ptr + s_count_buf;
            s_ureq_size <= "000";
            s_ureq_rw_n <= '0'; -- Gravando
```

```
                    s_ureq_ts <= '1';
                    s_ureq_data <= s_resposta_ram.data_1.cimag & s_resposta_ram.data_1.creal;
                    -- Habilita novas requiscoes e vai para idle
                    s_cpy_state <= cpy_read;
                    s_count_buf <= s_count_buf + 1;
                else
                    s_cpy_state <= cpy_cheio;
                end if;
            end if;
        when cpy_cheio => -- Aguarda ate' buffer disponÃvel e envia dados para a memo'ria do Programa
            if(ureq_full = '0') then
                s_ureq_addr <= s_buf_ptr + s_count_buf;
                s_ureq_size <= "000";
                s_ureq_rw_n <= '0'; -- Gravando
                s_ureq_ts <= '1';
                s_ureq_data <= s_resposta_ram.data_1.cimag & s_resposta_ram.data_1.creal;
                -- Habilita novas requiscoes e vai para idle
                s_cpy_state <= cpy_read;
                s_count_buf <= s_count_buf + 1;
            else
                s_cpy_state <= cpy_cheio;
            end if;
        when cpy_done => -- Sinaliza Terminado
            s_cpy_done <= '1';
    end case;
  end if;
end process cpy_read_ram;

-- Fabric Request
-- Conexao dos sinais de saida
freq_enable <= s_freq_enable;
uresp_ts     <= s_uresp_ts;
uresp_srctag <= s_uresp_srctag;
uresp_size   <= s_uresp_size;
uresp_data   <= s_uresp_data;
fg_count     <= s_count_ram;
s_start      <= s_app_cfg(0);
start        <= s_start and s_cpy_vet_done;
vector_size  <= s_vector_size;
buf_ptr      <= s_buf_ptr;
req_w_fg     <= s_req_w_fg;

select_mux_r_ram <= not s_fft_done; -- Quando esta' pronto a unidade de comunicacao ja' pode ler
select_mux_w_ram <= s_start and s_cpy_vet_done; -- Depois do start e da co'pia dos vetores o FFT control pode
      escrever na memo'ria do FPGA
-- Debug
s_resposta_fg <= resposta_fg;
s_resposta_ram <= resposta_ram;
requisicao_fg    <= s_requisicao_fg;
req_w_ram        <= s_req_w_ram;
req_r_ram        <= s_req_r_ram;

-- Sinais Constantes
s_ureq_mask <= x"FF";
s_ureq_byte_req <= '0';

--RT Core (conexao)
ureq_addr   <= s_ureq_addr;
ureq_size   <= s_ureq_size;
ureq_mask   <= s_ureq_mask;
ureq_rw_n   <= s_ureq_rw_n;
ureq_data   <= s_ureq_data;
ureq_ts     <= s_ureq_ts;
ureq_byte_req  <= s_ureq_byte_req;
fresp_enable <= s_fresp_enable;
end architecture basic;
```

Listing A.6: Multiplexador de escrita na RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;


-- multiplexador_write_ram
-- Multiplexador usado para a selecao dos requisicoes de escrita na ram do FPGA

entity multiplexador_write_ram is
    port(
        entrada0    : in t_req_write_ram;   -- Entrada 0
        entrada1    : in t_req_write_ram;   -- Entrada 1
        saida       : out t_req_write_ram;  -- Saida
        seletor     : in std_logic          -- Seletor
    );
end entity multiplexador_write_ram;


architecture basic of multiplexador_write_ram is
begin
    saida <= entrada0 when seletor = '0' else
             entrada1;
end architecture basic;
```

Listing A.7: Multiplexador de leitura na RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;

-- multiplexador_read_ram
-- Multiplexador usado para a selecao dos requisicoes de leitura na ram do FPGA

entity multiplexador_read_ram is
   port(
      entrada0    : in t_req_read_ram;  -- Entrada0
      entrada1    : in t_req_read_ram;  -- Entrada1
      saida       : out t_req_read_ram; -- Saida
      seletor     : in std_logic        -- Seletor
   );
end entity multiplexador_read_ram;


architecture basic of multiplexador_read_ram is
begin
   saida <= entrada0 when seletor = '0' else
            entrada1;
end architecture basic;
```

Listing A.8: Manipulador de Dados

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;

-------------------------------------------------------------------------------
-- ram
-- Manipulador de Dados
-- Utiliza dois blocos de memo'ria do FPGA para permitir a leitura de duas posicoes de mem'oria por vez( em paralelo)
-------------------------------------------------------------------------------

entity ram is
   port(
      requisicao_r      : in  t_req_read_ram;             -- Requisicao de leitura
      requisicao_w      : in  t_req_write_ram;            -- Requisicao de escrita
      resposta          : out t_resp_read_ram;            -- Resposta de uma requisicao de leitura
      user_clk          : in  std_logic;
      reset_n           : in  std_logic;
      -- Sinais da QDRAM
      -- RAM 1
      dr_1              : in  std_logic_vector(71 downto 0); -- Read data
      ar_1              : out std_logic_vector(19 downto 0); -- Read address
      aw_1              : out std_logic_vector(19 downto 0); -- Write address
      r_n_1             : out std_logic;                     -- Read strobe
      w_n_1             : out std_logic;                     -- Write strobe
      bw_n_1            : out std_logic_vector(7 downto 0);  -- Byte write
      dw_1              : out std_logic_vector(71 downto 0); -- Write data
      -- RAM 2
      dr_2              : in  std_logic_vector(71 downto 0); -- Read data
      ar_2               : out std_logic_vector(19 downto 0); -- Read address
      aw_2              : out std_logic_vector(19 downto 0); -- Write address
      r_n_2             : out std_logic;                     -- Read strobe
      w_n_2             : out std_logic;                     -- Write strobe
      bw_n_2            : out std_logic_vector(7 downto 0);  -- Byte write
      dw_2              : out std_logic_vector(71 downto 0)  -- write data
   );
end entity ram;

architecture basic of ram is
   type t_read_state is (read_idle, read_wait);       -- Estados da ma'quinas de estados para leitura de dados

   -- RAM 1 e RAM 2 (sinais em comum entre os dois blocos de memo'ria)
   signal s_aw       : std_logic_vector(19 downto 0);       -- Endereco de gravacao (ram1 <= addr, ram2 <= addr)
   signal s_r_n      : std_logic;                           -- Habilita leitura em 0
   signal s_w_n      : std_logic;                           -- Habilita escrita em 0
   signal s_bw_n     : std_logic_vector(7 downto 0);        -- Bytes habilitados
   signal s_dw       : std_logic_vector(63 downto 0);       -- Dados a serem gravados (ram1<=data, ram2<=data)
   -- Sinais individuais
   -- RAM 1
   signal s_ar_1     : std_logic_vector(19 downto 0);       -- Endereco de leitura da ram 1
   signal s_dr_1     : std_logic_vector(71 downto 0);       -- Dados lidos da ram 1
    -- RAM 2
   signal s_ar_2     : std_logic_vector(19 downto 0);       -- Endereco de leitura da ram 2
   signal s_dr_2     : std_logic_vector(71 downto 0);       -- Dados lidos da ram 2
   -- Controle
   signal s_read_state  : t_read_state;                     -- Maquina de estados
   signal s_count_wait     : std_logic_vector(3 downto 0); -- Contador de espera para a leitura de dados da memo'ria
   -- Saida
   signal s_resposta : t_resp_read_ram;                     -- Saida da requisicao de leitura da ram
   signal s_requisicao_r : t_req_read_ram;                  -- Armazena a requisicao
begin
   -- read
   -- Ma'quina de estados de leitura da ram
```

```
   read : process(user_clk, reset_n) is
   begin
      if (reset_n = '0') then
         s_resposta.done <= '0';
         s_r_n <= '1';
         s_read_state <= read_idle;
      elsif (user_clk'event and user_clk = '1') then
         s_resposta.done <= '0';
         -- Limpa sinais de ciclo u'nico
         s_r_n <= '1'; -- Desabilita leitura
         s_requisicao_r <= requisicao_r;
         case (s_read_state) is
            when read_idle => -- Aguardando requisicao
               if(s_requisicao_r.enable = '1') then -- Realiza requisicao
                  s_ar_1 <= requisicao_r.addr_r_1;
                  s_ar_2 <= requisicao_r.addr_r_2;
                  -- Habilita leitura
                  s_r_n <= '0';
                  s_count_wait <= c_latencia_qdr;
                  s_read_state <= read_wait;
               end if;
            when read_wait => -- Aguarda resposta
               if(s_count_wait = 0) then -- Retorna resposta de leitura
                  s_resposta.data_1.cimag <= dr_1(63 downto 32);
                  s_resposta.data_1.creal <= dr_1(31 downto 0);
                  s_resposta.data_2.cimag <= dr_2(63 downto 32);
                  s_resposta.data_2.creal <= dr_2(31 downto 0);
                  s_resposta.done <= '1';
                  s_read_state <= read_idle;
               else
                  s_count_wait <= s_count_wait - 1;
               end if;
         end case;
      end if;
   end process read;

   -- write
   -- Ma'quina de estados para escrita na ram
   write : process(user_clk, reset_n) is
   begin
      if (reset_n = '0') then
         s_w_n <= '1'; -- Desabilita gravacao
         s_bw_n <= (others => '0');
      elsif (user_clk'event and user_clk = '1') then
         -- Limpa sinais de ciclo u'nico
         s_w_n <= '1';
         if(requisicao_w.enable = '1') then -- Grava dados nos blocos (em ambos os blocos)
            s_aw <= requisicao_w.addr;
            s_dw <= requisicao_w.data.cimag & requisicao_w.data.creal;
            s_w_n <= '0';
         end if;
      end if;
   end process write;

   -- Conexao de sinais
   -- RAM 1
   ar_1            <= s_ar_1;
   aw_1            <= s_aw;
   r_n_1           <= s_r_n;
   w_n_1           <= s_w_n;
   bw_n_1          <= s_bw_n;
   dw_1            <= x"00" & s_dw;
   -- RAM 2
   ar_2            <= s_ar_2;
   aw_2            <= s_aw;
   r_n_2           <= s_r_n;
   w_n_2           <= s_w_n;
   bw_n_2          <= s_bw_n;
   dw_2            <= x"00" & s_dw;
   --Saida
   resposta <= s_resposta;
```

Listing A.9: Unidade de Borboletas

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;

------------------------------------------------------------------------------
-- borboleta_control
-- Entidade que controla a quintupla de borboletas
------------------------------------------------------------------------------

entity borboleta_control is
   port(
      requisicao        : in  t_req_borboleta_control;       -- Requisicao com 2 pontos + fator de giro para ser
          computada
      busy              : out std_logic;                     -- Indica que todas borboletas estao trabalhando
      user_clk          : in  std_logic;
      reset_n           : in  std_logic;
      req_w_ram         : out t_req_write_ram;               -- Requisicao de gravacao na ram (para envio dos dados
          para a RAM)
      rodadas           : out std_logic_vector(31 downto 0); -- Quantidades de requisicoes tratadas (total de
          borboletas computadas)
```

```
      working           : out std_logic                       -- Indica se tem alguma borboleta trabalhando (usado
          para aguardar finalizacao de um ciclo da FFT)
   );
end entity borboleta_control;

architecture quintupla of borboleta_control is

component borboleta is
   port(
      a_in         : in t_complex;   -- Entrada a
      b_in         : in t_complex;   -- Entrada b
      fator_giro   : in t_complex;   -- Fator de giro (w)
      en           : in std_logic;   -- Enable
      clk          : in std_logic;
      a_out        : out t_complex;  -- Saida a
      b_out        : out t_complex;  -- Saida b
      done         : out std_logic;  -- Indica que o calcula esta' pronto
      reset_n      : in std_logic
   );
end component borboleta;
   for all: borboleta use entity basic.borboleta(arch_min);

   type t_borb_state is (borb_idle, borb_wait, borb_ciclo2); -- Maquina de estado que manipula cada borboleta

   -- Sanais de estado das borboletas
   signal s_borb0_state      : t_borb_state;
   signal s_borb1_state      : t_borb_state;
   signal s_borb2_state      : t_borb_state;
   signal s_borb3_state      : t_borb_state;
   signal s_borb4_state      : t_borb_state;

   -- Armazena requisicao
   signal s_requisicao        : t_req_borboleta_control;

   -- Sinais de ocupado de cada borboleta
   signal s_busy0             : std_logic;
   signal s_busy1             : std_logic;
   signal s_busy2             : std_logic;
   signal s_busy3             : std_logic;
   signal s_busy4             : std_logic;

   -- Saida de cada borboleta
   signal s_req_w_ram0: t_req_write_ram;
   signal s_req_w_ram1: t_req_write_ram;
   signal s_req_w_ram2: t_req_write_ram;
   signal s_req_w_ram3: t_req_write_ram;
   signal s_req_w_ram4: t_req_write_ram;

   -- Sinais para instanciacao de cada borboleta
   --borboleta0
   signal s_borb0_a_in   : t_complex;
   signal s_borb0_b_in   : t_complex;
   signal s_borb0_fg     : t_complex;
   signal s_borb0_a_out  : t_complex;
   signal s_borb0_b_out  : t_complex;
   signal s_borb0_b_out_tmp : t_complex;
   signal s_borb0_done   : std_logic;
   signal s_borb0_en     : std_logic;
   --borboleta1
   signal s_borb1_a_in   : t_complex;
   signal s_borb1_b_in   : t_complex;
   signal s_borb1_fg     : t_complex;
   signal s_borb1_a_out  : t_complex;
   signal s_borb1_b_out  : t_complex;
   signal s_borb1_b_out_tmp : t_complex;
   signal s_borb1_done   : std_logic;
   signal s_borb1_en     : std_logic;
   --borboleta2
   signal s_borb2_a_in   : t_complex;
   signal s_borb2_b_in   : t_complex;
   signal s_borb2_fg     : t_complex;
   signal s_borb2_a_out  : t_complex;
   signal s_borb2_b_out  : t_complex;
   signal s_borb2_b_out_tmp : t_complex;
   signal s_borb2_done   : std_logic;
   signal s_borb2_en     : std_logic;
   --borboleta3
   signal s_borb3_a_in   : t_complex;
   signal s_borb3_b_in   : t_complex;
   signal s_borb3_fg     : t_complex;
   signal s_borb3_a_out  : t_complex;
   signal s_borb3_b_out  : t_complex;
   signal s_borb3_b_out_tmp : t_complex;
   signal s_borb3_done   : std_logic;
   signal s_borb3_en     : std_logic;
   --borboleta4
   signal s_borb4_a_in   : t_complex;
   signal s_borb4_b_in   : t_complex;
   signal s_borb4_fg     : t_complex;
   signal s_borb4_a_out  : t_complex;
   signal s_borb4_b_out  : t_complex;
   signal s_borb4_b_out_tmp : t_complex;
   signal s_borb4_done   : std_logic;
   signal s_borb4_en     : std_logic;


   signal s_rodadas: std_logic_vector(31 downto 0);    -- Debug: total de requisicoes recebidas

   -- Enderecos de destino de cada requisicao recebida (onde serao gravados os pontos apos a computacao)
```

```
   -- borboleta0
   signal s_destino0_1    : std_logic_vector(19 downto 0);
   signal s_destino0_2    : std_logic_vector(19 downto 0);
   -- borboleta1
   signal s_destino1_1    : std_logic_vector(19 downto 0);
   signal s_destino1_2    : std_logic_vector(19 downto 0);
   -- borboleta2
   signal s_destino2_1    : std_logic_vector(19 downto 0);
   signal s_destino2_2    : std_logic_vector(19 downto 0);
   -- borboleta3
   signal s_destino3_1    : std_logic_vector(19 downto 0);
   signal s_destino3_2    : std_logic_vector(19 downto 0);
   -- borboleta4
   signal s_destino4_1    : std_logic_vector(19 downto 0);
   signal s_destino4_2    : std_logic_vector(19 downto 0);

begin
borb_0: borboleta
   port map(
      a_in         => s_borb0_a_in,
      b_in         => s_borb0_b_in,
      fator_giro   => s_borb0_fg,
      en           => s_borb0_en,
      clk          => user_clk,
      a_out        => s_borb0_a_out,
      b_out        => s_borb0_b_out,
      done         => s_borb0_done,
      reset_n      => reset_n
   );


borb_1: borboleta
   port map(
      a_in         => s_borb1_a_in,
      b_in         => s_borb1_b_in,
      fator_giro   => s_borb1_fg,
      en           => s_borb1_en,
      clk          => user_clk,
      a_out        => s_borb1_a_out,
      b_out        => s_borb1_b_out,
      done         => s_borb1_done,
      reset_n      => reset_n
   );

borb_2: borboleta
   port map(
      a_in         => s_borb2_a_in,
      b_in         => s_borb2_b_in,
      fator_giro   => s_borb2_fg,
      en           => s_borb2_en,
      clk          => user_clk,
      a_out        => s_borb2_a_out,
      b_out        => s_borb2_b_out,
      done         => s_borb2_done,
      reset_n      => reset_n
   );

borb_3: borboleta
   port map(
      a_in         => s_borb3_a_in,
      b_in         => s_borb3_b_in,
      fator_giro   => s_borb3_fg,
      en           => s_borb3_en,
      clk          => user_clk,
      a_out        => s_borb3_a_out,
      b_out        => s_borb3_b_out,
      done         => s_borb3_done,
      reset_n      => reset_n
   );

borb_4: borboleta
   port map(
      a_in         => s_borb4_a_in,
      b_in         => s_borb4_b_in,
      fator_giro   => s_borb4_fg,
      en           => s_borb4_en,
      clk          => user_clk,
      a_out        => s_borb4_a_out,
      b_out        => s_borb4_b_out,
      done         => s_borb4_done,
      reset_n      => reset_n
   );

   -- b0
   -- Manipula a borboleta0
   b0 : process(user_clk, reset_n) is
   begin
      if (reset_n = '0') then
         s_req_w_ram0.enable <= '0';
         s_busy0 <= '0';
         s_borb0_state <= borb_idle;
         s_rodadas(31) <= '0';
         s_rodadas(23 downto 0) <= (others => '0');
      elsif (user_clk'event and user_clk = '1') then
         -- Limpa sinais de ciclo u'nico
         s_req_w_ram0.enable <= '0';
         s_borb0_en <= '0';
         if(requisicao.enable = '1') then -- Conta total de requisicoes recebidas
            s_rodadas(23 downto 0) <= s_rodadas(23 downto 0) + 1;
```

```
        end if;
        case(s_borb0_state) is
           when borb_idle =>
              if(requisicao.enable = '1') then -- Ao receber requisicao, configura borboleta 1 para computar
                 -- Copia requisicao para entrada da borboleta
                 s_borb0_a_in <= requisicao.in_1;
                 s_borb0_b_in <= requisicao.in_2;
                 s_borb0_fg   <= requisicao.fg;
                 s_destino0_1  <= requisicao.addr_in_1;
                 s_destino0_2  <= requisicao.addr_in_2;
                 s_borb0_state <= borb_wait;
                 s_borb0_en <= '1';
                 s_busy0 <= '1';
                 s_rodadas(31) <= '1';
              end if;
           when borb_wait => -- Aguarda finalizacao da computacao da borboleta
              if(s_borb0_done = '1') then
                 s_req_w_ram0.addr <= s_destino0_1;
                 s_req_w_ram0.data <= s_borb0_a_out;
                 s_req_w_ram0.enable <= '1';          -- Grava o ponto a
                 s_borb0_b_out_tmp <= s_borb0_b_out; -- Guarda o b pois a gravacao na RAM e' um ponto por vez
                 s_borb0_state <= borb_ciclo2;
              end if;
           when borb_ciclo2 =>                      -- Grava o ponto b
              s_req_w_ram0.addr <= s_destino0_2;
              s_req_w_ram0.data <= s_borb0_b_out_tmp;
              s_req_w_ram0.enable <='1';
              s_borb0_state <= borb_idle;
              s_busy0 <= '0';
        end case;
     end if;
  end process b0;

  -- b1
  -- Manipula a borboleta1
  b1 : process(user_clk, reset_n) is
  begin
     if (reset_n = '0') then
        s_req_w_ram1.enable <= '0';
        s_busy1 <= '0';
        s_borb1_state <= borb_idle;
         s_rodadas(30) <= '0';
     elsif (user_clk'event and user_clk = '1') then
        s_req_w_ram1.enable <= '0';
        s_borb1_en <= '0';
        case(s_borb1_state) is
           when borb_idle =>
              if(requisicao.enable = '1' and s_busy0 ='1') then -- Se tiver requisicao e a borboleta0 estiver ocupada
                 -- Copia requisicao para entrada da borboleta
                 s_borb1_a_in <= requisicao.in_1;
                 s_borb1_b_in <= requisicao.in_2;
                 s_borb1_fg   <= requisicao.fg;
                 s_destino1_1  <= requisicao.addr_in_1;
                 s_destino1_2  <= requisicao.addr_in_2;
                 s_borb1_state <= borb_wait;
                 s_borb1_en <= '1';
                 s_busy1 <= '1';
                 s_rodadas(30) <= '1';
              end if;
           when borb_wait =>
              if(s_borb1_done = '1') then
                 s_req_w_ram1.addr <= s_destino1_1;
                 s_req_w_ram1.data <= s_borb1_a_out;
                 s_req_w_ram1.enable <= '1';
                 s_borb1_b_out_tmp <= s_borb1_b_out;
                 s_borb1_state <= borb_ciclo2;
              end if;
           when borb_ciclo2 =>
              s_req_w_ram1.addr <= s_destino1_2;
              s_req_w_ram1.data <= s_borb1_b_out_tmp;
              s_req_w_ram1.enable <='1';
              s_borb1_state <= borb_idle;
              s_busy1 <= '0';
        end case;
     end if;
  end process b1;

  -- b2
  -- Manipula a borboleta2
  b2 : process(user_clk, reset_n) is
  begin
     if (reset_n = '0') then
        s_req_w_ram2.enable <= '0';
        s_busy2 <= '0';
        s_borb2_state <= borb_idle;
        s_rodadas(29) <= '0';
     elsif (user_clk'event and user_clk = '1') then
        s_req_w_ram2.enable <= '0';
        s_borb2_en <= '0';
        case(s_borb2_state) is
           when borb_idle =>
              if(requisicao.enable = '1' and s_busy0 ='1' and s_busy1 = '1') then -- Se tiver requisicao e a
                   borboleta0 e borboleta1 estiverem ocupadas
              -- Copia requisicao para entrada da borboleta
                 s_borb2_a_in <= requisicao.in_1;
                 s_borb2_b_in <= requisicao.in_2;
                 s_borb2_fg   <= requisicao.fg;
                 s_destino2_1  <= requisicao.addr_in_1;
                 s_destino2_2  <= requisicao.addr_in_2;
```

```
                s_borb2_state <= borb_wait;
                s_borb2_en <= '1';
                s_busy2 <= '1';
                s_rodadas(29) <= '1';
            end if;
        when borb_wait =>
            if(s_borb2_done = '1') then
                s_req_w_ram2.addr <= s_destino2_1;
                s_req_w_ram2.data <= s_borb2_a_out;
                s_req_w_ram2.enable <= '1';
                s_borb2_b_out_tmp <= s_borb2_b_out;
                s_borb2_state <= borb_ciclo2;
            end if;
        when borb_ciclo2 =>
            s_req_w_ram2.addr <= s_destino2_2;
            s_req_w_ram2.data <= s_borb2_b_out_tmp;
            s_req_w_ram2.enable <='1';
            s_borb2_state <= borb_idle;
            s_busy2 <= '0';
    end case;
  end if;
end process b2;

-- b3
-- Manipula a borboleta3
b3 : process(user_clk, reset_n) is
begin
  if (reset_n = '0') then
    s_req_w_ram3.enable <= '0';
    s_busy3 <= '0';
    s_borb3_state <= borb_idle;
    s_rodadas(28) <= '0';
  elsif (user_clk'event and user_clk = '1') then
    s_req_w_ram3.enable <= '0';
    s_borb3_en <= '0';
    case(s_borb3_state) is
      when borb_idle =>
        if(requisicao.enable = '1' and s_busy0 ='1' and s_busy1 = '1' and s_busy2 = '1') then -- Se tiver
            requisicao e a borboleta0, borboleta1 e borboleta2 estiverem ocupadas
          -- Copia requisicao para entrada da borboleta
          s_borb3_a_in <= requisicao.in_1;
          s_borb3_b_in <= requisicao.in_2;
          s_borb3_fg   <= requisicao.fg;
          s_destino3_1  <= requisicao.addr_in_1;
          s_destino3_2  <= requisicao.addr_in_2;
          s_borb3_state <= borb_wait;
          s_borb3_en <= '1';
          s_busy3 <= '1';
          s_rodadas(28) <= '1';
        end if;
      when borb_wait =>
        if(s_borb3_done = '1') then
          s_req_w_ram3.addr <= s_destino3_1;
          s_req_w_ram3.data <= s_borb3_a_out;
          s_req_w_ram3.enable <= '1';
          s_borb3_b_out_tmp <= s_borb3_b_out;
          s_borb3_state <= borb_ciclo2;
        end if;
      when borb_ciclo2 =>
        s_req_w_ram3.addr <= s_destino3_2;
        s_req_w_ram3.data <= s_borb3_b_out_tmp;
        s_req_w_ram3.enable <='1';
        s_borb3_state <= borb_idle;
        s_busy3 <= '0';
    end case;
  end if;
end process b3;


-- b4
-- Manipula a borboleta4
b4 : process(user_clk, reset_n) is
begin
  if (reset_n = '0') then
    s_req_w_ram4.enable <= '0';
    s_busy4 <= '0';
    s_borb4_state <= borb_idle;
    s_rodadas(27) <= '0';
  elsif (user_clk'event and user_clk = '1') then
    s_req_w_ram4.enable <= '0';
    s_borb4_en <= '0';
    case(s_borb4_state) is
      when borb_idle =>
        if(requisicao.enable = '1' and s_busy0 ='1' and s_busy1 = '1' and s_busy2 = '1' and s_busy3 = '1') then
            -- Se tiver requisicao e a borboleta0, borboleta1, borboleta2 e borboleta3 estiverem ocupadas
          -- Copia requisicao para entrada da borboleta
          s_borb4_a_in <= requisicao.in_1;
          s_borb4_b_in <= requisicao.in_2;
          s_borb4_fg   <= requisicao.fg;
          s_destino4_1  <= requisicao.addr_in_1;
          s_destino4_2  <= requisicao.addr_in_2;
          s_borb4_state <= borb_wait;
          s_borb4_en <= '1';
          s_busy4 <= '1';
          s_rodadas(27) <= '1';
        end if;
      when borb_wait =>
        if(s_borb4_done = '1') then
          s_req_w_ram4.addr <= s_destino4_1;
```

```
                s_req_w_ram4.data <= s_borb4_a_out;
                s_req_w_ram4.enable <= '1';
                s_borb4_b_out_tmp <= s_borb4_b_out;
                s_borb4_state <= borb_ciclo2;
            end if;
        when borb_ciclo2 =>
            s_req_w_ram4.addr <= s_destino4_2;
            s_req_w_ram4.data <= s_borb4_b_out_tmp;
            s_req_w_ram4.enable <='1';
            s_borb4_state <= borb_idle;
            s_busy4 <= '0';
    end case;
  end if;
end process b4;

-- req
-- Responsa'vel por fazer o tratamento das requisicoes das borboletas para o envio dos dados para a memoria do FPGA
req : process(user_clk) is
begin
  if(reset_n = '0') then
      -- Sinais de debug
      s_rodadas(26) <= '0';
      s_rodadas(25) <= '0';
      s_rodadas(24) <= '0';
  elsif (user_clk'event and user_clk = '1') then
    -- Seleciona requisicao que serÃ¡ enviada para a RAM
    if(s_req_w_ram0.enable = '1') then
      req_w_ram <= s_req_w_ram0;
    elsif(s_req_w_ram1.enable = '1') then
      req_w_ram <= s_req_w_ram1;
    elsif(s_req_w_ram2.enable = '1') then
      req_w_ram <= s_req_w_ram2;
    elsif(s_req_w_ram3.enable = '1') then
      req_w_ram <= s_req_w_ram3;
    else
      req_w_ram <= s_req_w_ram4;
    end if;
  end if;
  -- Verifica a existencia de colisao enter requisicoes das borboletas (nunca ocorre)
  if(s_req_w_ram0.enable = '1' and s_req_w_ram1.enable = '1') then
    s_rodadas(26) <= '1';
  end if;
  if(s_req_w_ram1.enable = '1' and s_req_w_ram2.enable = '1') then
    s_rodadas(25) <= '1';
  end if;
  if(s_req_w_ram2.enable = '1' and s_req_w_ram3.enable = '1') then
    s_rodadas(24) <= '1';
  end if;
  if(s_req_w_ram3.enable = '1' and s_req_w_ram4.enable = '1') then
    s_rodadas(24) <= s_rodadas(24) or '1';
  end if;
end process req;

busy <= s_busy0 and s_busy1 and s_busy2 and s_busy3 and s_busy4; -- Se todas ocupadas
rodadas <= s_rodadas;
working <=  (s_busy0 or s_busy1 or s_busy2 or s_busy3 or s_busy4); -- Se alguma ocupada
end architecture quintupla;
```

Listing A.10: Unidade de Controle

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;
library basic;
use basic.basic_pkg.all;

--------------------------------------------------------------------------------
--fft_control
-- Unidade de Controle, responsa'vel por realizar o laco referente ao algoritmo da FFT: Efetua a leitura de dois
     pontos, um fator de giro e envia para a unidade de borboletas. Sincroniza ao final de cada passo para garantir
     que a nao leitura de dados invalidados da memoria RAM
--------------------------------------------------------------------------------

entity fft_control is
  port(
    user_clk        : in std_logic;
    reset_n         : in std_logic;
    -- Registradores
    vector_size     : in std_logic_vector(19 downto 0);  -- Tamanho do vetor
    start           : in std_logic;                      -- Start (iniciar)
    borb_busy       : in std_logic;                      -- Borboletas ocupadas
    done            : out std_logic;                     -- Sinaliza final da computacao
    req_read_ram    : out t_req_read_ram;                -- Requisicao de leitura na RAM
    resposta_ram    : in t_resp_read_ram;                -- Resposta da RAM
    req_read_fg     : out t_req_fator_giro;              -- Requisicao de leitura no Fator de Giro
    resposta_fg     : in t_resp_fator_giro;              -- Leitura do FAtor de Giro
    req_borb        : out t_req_borboleta_control;       -- Requisicao para unidade de borboletas
    fft_debug       : out std_logic_vector(31 downto 0); -- Debug: total de ciclos que esta unidade aguardo ate que a
          unidade de borboletas estivesse disponivel
    borb_ctrl_working : in std_logic                     -- Borboletas estao trabalhando (usado na sincronizacao no
          final de cada passo)
  );
end entity fft_control;

architecture basic of fft_control is
```

```
    -- estados da maquina de estados
    type t_fft_state is ( fft_idle, fft_loop_step, fft_loop_j, fft_wait_fg, fft_loop_i, fft_wait_ram, fft_wait_borb,
        fft_wait_fim_step, fft_done);

    -- controle
    signal s_done         : std_logic;                    -- Pronto
    signal s_req_read_ram : t_req_read_ram;               -- Requisicao de leitura da ram
    signal s_req_read_fg  : t_req_fator_giro;             -- Requisicoa de leitura de fator de giro
    signal s_req_borb     : t_req_borboleta_control;      -- Requisicao de calculo de borboleta (enviado para unidade
        de borboletas)
    signal s_fft_state    : t_fft_state;                  -- Estado da maquina de estados

    signal s_contador_fg: std_logic_vector(19 downto 0);  -- Contador de fatores de giro (os fatores de giro estao
        ordenados na memoria)
    signal s_fft_debug: std_logic_vector(31 downto 0);    -- Debug: total de ciclos que esta unidade aguardo ate que
        a unidade de borboletas estivesse disponivel

    -- Indices usados nos lacos (confome algoritmo de Cooley-Tukey, ver pseudocodigo monografia)
    signal s_step  : std_logic_vector(19 downto 0); --
    signal s_j     : std_logic_vector(19 downto 0);
    signal s_n     : std_logic_vector(19 downto 0);
    signal s_i     : std_logic_vector(19 downto 0);
begin
    -- main
    -- Ma'quina de estados principal que realiza os lacos do algoritmo
    main: process(user_clk, reset_n) is
    begin
        if (reset_n = '0') then
            s_done <= '0';
            s_req_read_ram.enable <= '0';
            s_req_read_fg.enable <= '0';
            s_req_borb.enable <= '0';
            s_fft_state <= fft_idle;
            s_contador_fg <= (others => '0');
            s_fft_debug <= (others => '0');
        elsif (user_clk'event and user_clk = '1') then
            -- Limpa sinais de ciclo u'nico
            s_req_read_ram.enable <= '0';
            s_req_read_fg.enable <= '0';
            s_req_borb.enable <= '0';
            case(s_fft_state) is
                when fft_idle =>
                    if(start = '1') then                  -- Aguarda start
                        s_n <= vector_size;
                        s_contador_fg <= (others => '0');
                        s_step <= x"00001";               -- 1
                        s_done <= '0';
                        s_fft_state <= fft_loop_step;
                    end if;
                when fft_loop_step =>                     -- Laco mais externo
                    if(s_step /= s_n) then
                        s_j <= (others => '0');
                        s_fft_state <= fft_loop_j;
                    else
                        s_fft_state <= fft_done;
                    end if;
                when fft_loop_j =>                        -- Segundo laco
                    if(s_j /= s_step) then
                        -- Requisita pro'ximo fator de giro
                        s_req_read_fg.posicao <= s_contador_fg;
                        s_req_read_fg.enable <= '1';
                        s_fft_state <= fft_wait_fg;
                    else
                        s_step <= s_step(18 downto 0) & '0';  -- step*=2;
                        s_fft_state <= fft_wait_fim_step;
                    end if;
                when fft_wait_fim_step =>                 -- Aguarda final do passo
                    if(borb_ctrl_working = '0') then      -- Sincronizacao com as borboletas
                        s_fft_state <= fft_loop_step;
                    end if;
                when fft_wait_fg =>                       -- Aguarda fator de giro (se ele nao estiver em cache
                    demorara' alguns ciclos para ser entregue)
                    if(resposta_fg.done = '1') then
                        s_req_borb.fg <= resposta_fg.fator_giro;
                        s_contador_fg <= s_contador_fg + 1;
                        s_fft_state <= fft_loop_i;
                        s_i <= s_j;
                    end if;
                when fft_loop_i =>                        -- Laco mais interno
                    if(s_i < s_n) then
                        -- Requisita posicoes da ram
                        s_req_read_ram.addr_r_1 <= s_i;
                        s_req_read_ram.addr_r_2 <= s_i + s_step;
                        s_req_read_ram.enable <= '1';
                        -- Organiza a requisicao para a borboleta
                        s_req_borb.addr_in_1    <= s_i;
                        s_req_borb.addr_in_2    <= s_i + s_step;
                        s_fft_state <= fft_wait_ram;
                    else
                        s_j <= s_j + 1;                   -- j++
                        s_fft_state <= fft_loop_j;
                    end if;
                when fft_wait_ram =>                      -- Espera os dados virem da RAM
                    if(resposta_ram.done = '1') then
                        s_req_borb.in_1 <= resposta_ram.data_1;
                        s_req_borb.in_2 <= resposta_ram.data_2;
                        -- Espera borboleta estar pronta; (otimizar)
                        s_fft_state <= fft_wait_borb;
                    end if;
```

```
            when fft_wait_borb =>                          -- Espera borboletas ficarem livres para enviar a
                 requisicao (atualmente sempre estao livres)
               if(borb_busy = '0') then
                  s_req_borb.enable <= '1';
                  s_i <= s_i + (s_step(18 downto 0) & '0');-- i+=2*step = i = i + 2*step
                  s_fft_state <= fft_loop_i;
               else
                  s_fft_debug <= s_fft_debug + 1;
               end if;
            when fft_done =>                               -- Sinaliza final da computacao
               s_done <= '1';
         end case;
      end if;
   end process main;

   -- conecta sinais de saida
   done <= s_done;
   req_read_ram  <= s_req_read_ram;
   req_read_fg <= s_req_read_fg;
   req_borb <= s_req_borb;
   fft_debug  <= s_fft_debug ;
end architecture basic;
```

A.3 Componentes Internos

Listing A.11: Borboleta

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;

-------------------------------------------------------------------------------
-- borboleta
-- Entidade da borboleta, nele sao usados compentes para as operacoes matematicas complexas. A maquinda de estado
    controla a multiplicacao e soma complexa
-------------------------------------------------------------------------------

entity borboleta is
   port(
      a_in         : in t_complex; -- Entrada a
      b_in         : in t_complex; -- Entrada b
      fator_giro   : in t_complex; -- Fator de giro (w)
      en           : in std_logic; -- Enable
      clk          : in std_logic;
      a_out        : out t_complex; -- Saida a'
      b_out        : out t_complex; -- Saiba b'
      done         : out std_logic; -- Pronto
      reset_n      : in std_logic
   );
end entity borboleta;


architecture arch_min of borboleta is
   -- Componentes
   component complex_multiplicacao is
   port(
      z            : in t_complex;
      w            : in t_complex;
      r            : out t_complex;
      en           : in std_logic;
      clk          : in std_logic;
      done         : out std_logic;
      reset_n      : in std_logic
   );
   end component complex_multiplicacao;

   component complex_soma is
   port(
      a            : in t_complex;
      b            : in t_complex;
      r            : out t_complex;
      en           : in std_logic;
      clk          : in std_logic;
      done         : out std_logic;
      reset_n      : in std_logic
   );
   end component complex_soma;

   component complex_subtracao is
   port(
      a            : in t_complex;
      b            : in t_complex;
      r            : out t_complex;
      en           : in std_logic;
      clk          : in std_logic;
      done         : out std_logic;
      reset_n      : in std_logic
   );
   end component complex_subtracao;
   -- Cconfiguracao
   for all: complex_soma use entity basic.complex_soma(arch_fp_1_somador);
```

```
    for all: complex_subtracao use entity basic.complex_subtracao(arch_fp_1_subtrador);
    for all: complex_multiplicacao use entity basic.complex_multiplicacao(arch_fp_2_multiplicacao);
    -- Sinais
    type t_req_state is (idle, multiplicando, add_sub);
    signal s_req_state : t_req_state;

    signal s_done: std_logic;
    signal s_a_in: t_complex;
    signal s_b_in: t_complex;
    signal s_fator_giro: t_complex;
    signal s_a_out: t_complex;
    signal s_b_out: t_complex;
    signal s_wb: t_complex;
    signal s_wb_r: t_complex;
    signal s_mul_en: std_logic;
    signal s_mul_done: std_logic;
    signal s_add_sub_en: std_logic;
    signal s_add_done: std_logic;
    signal s_sub_done: std_logic;

begin
    -- Multiplicador complexo
    mult_complexo: complex_multiplicacao
        port map(
            w      => s_fator_giro,
            z      => s_b_in,
            r      => s_wb_r,
            en     => s_mul_en,
            clk    => clk,
            done   => s_mul_done,
            reset_n => reset_n
        );
    -- Somador complexo
    add_complexo: complex_soma
        port map(
            a      => s_a_in,
            b      => s_wb,
            r      => s_a_out,
            en     => s_add_sub_en,
            clk    => clk,
            done   => s_add_done,
            reset_n => reset_n
        );
    -- Subtrador complexo
    sub_complexo: complex_subtracao
        port map(
            a      => s_a_in,
            b      => s_wb,
            r      => s_b_out,
            en     => s_add_sub_en,
            clk    => clk,
            done   => s_sub_done,
            reset_n => reset_n
        );

    -- main
    -- Processo principal que realiza a computacao de uma borboleta
    main: process(reset_n, clk) is
    begin
        if(reset_n = '0') then
          s_req_state <= idle;
          s_done <= '0';
          s_mul_en <= '0';
          s_add_sub_en <= '0';
          s_a_in.creal <= (others => '0');
          s_a_in.cimag <= (others => '0');
          s_b_in.creal <= (others => '0');
          s_b_in.cimag <= (others => '0');
          s_fator_giro.creal <= (others => '0');
          s_fator_giro.cimag <= (others => '0');
        elsif(clk'event and clk='1') then
          -- Limpa sinais de ciclo u'nico
          s_done <= '0';
          s_mul_en <= '0';
          s_add_sub_en <= '0';
          case (s_req_state) is
              when idle =>
                  if (en = '1') then -- Inicia computacao quando habilitada
                      -- Copia dados de entrada
                      s_a_in <= a_in;
                      s_b_in <= b_in;
                      s_fator_giro <= fator_giro;
                      s_mul_en <= '1'; -- Habilita multiplicador
                      s_req_state <= multiplicando;
                  end if;
              when multiplicando => -- Aguarda fim a multiplicacao
                  if(s_mul_done = '1') then
                      s_wb <= s_wb_r;
                      s_add_sub_en <= '1'; -- Habilita soma e subtracao (acontecem em paralelo)
                      s_req_state <= add_sub;
                  end if;
              when add_sub =>
                  if(s_add_done = '1') then -- Aguarda fim da soma e subtracao
                      s_done <= '1';
                      s_req_state <= idle;
                  end if;
              when others => null;
          end case;
        end if;
```

```
    end process main;
    -- Conexao com sinais de saida
    a_out <= s_a_out;
    b_out <= s_b_out;
    done <= s_done;
end architecture arch_min;
```

Listing A.12: Multiplicação Complexa

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;

-------------------------------------------------------------------------------
-- complex_multiplicacao
-- Realiza uma multiplicacao complexa
-------------------------------------------------------------------------------

entity complex_multiplicacao is
    port(
        z           : in t_complex; -- Entrada 1
        w           : in t_complex; -- Entrada 2
        r           : out t_complex;-- Saidas
        en          : in std_logic; -- Habilitar
        clk         : in std_logic;
        done        : out std_logic;-- Pronto
        reset_n     : in std_logic
    );
end entity complex_multiplicacao;

architecture arch_fp_2_multiplicacao of complex_multiplicacao is
    -- Componentes
    component subtrador_fp32 is
        port (
            a: in std_logic_vector(31 downto 0);
            b: in std_logic_vector(31 downto 0);
            operation_nd: in std_logic;
            clk: in std_logic;
            result: out std_logic_vector(31 downto 0);
            rdy: out std_logic
        );
    end component;
    component somador_fp32 is
        port (
            a: in std_logic_vector(31 downto 0);
            b: in std_logic_vector(31 downto 0);
            operation_nd: in std_logic;
            clk: in std_logic;
            result: out std_logic_vector(31 downto 0);
            rdy: out std_logic
        );
    end component;
    component multiplicador_fp32 is
        port (
            a: in std_logic_vector(31 downto 0);
            b: in std_logic_vector(31 downto 0);
            operation_nd: in std_logic;
            clk: in std_logic;
            result: out std_logic_vector(31 downto 0);
            rdy: out std_logic
        );
    end component;

    signal s_result: t_complex; -- Resultado
    signal s_done: std_logic;   -- Pronto

    -- Sinais para o somador
    signal s_add_a: std_logic_vector(31 downto 0);
    signal s_add_b: std_logic_vector(31 downto 0);
    signal s_add_result: std_logic_vector(31 downto 0);
    signal s_add_en: std_logic;
    signal s_add_done: std_logic;

    -- Sinais para o subtrador
    signal s_sub_a: std_logic_vector(31 downto 0);
    signal s_sub_b: std_logic_vector(31 downto 0);
    signal s_sub_result: std_logic_vector(31 downto 0);
    signal s_sub_en: std_logic;
    signal s_sub_done: std_logic;


    -- Sinais para o multiplicador 1
    signal s_mul_1_a: std_logic_vector(31 downto 0);
    signal s_mul_1_b: std_logic_vector(31 downto 0);
    signal s_mul_1_result: std_logic_vector(31 downto 0);
    signal s_mul_1_en: std_logic;
    signal s_mul_1_done: std_logic;


    -- Sinais para o multiplicador 2
    signal s_mul_2_a: std_logic_vector(31 downto 0);
    signal s_mul_2_b: std_logic_vector(31 downto 0);
    signal s_mul_2_result: std_logic_vector(31 downto 0);
    signal s_mul_2_en: std_logic;
    signal s_mul_2_done: std_logic;
```

```
                                                                                                                    89
                                                                                                                    90
  -- Sinais para armazenar dados de entrada                                                                         91
  signal s_z: t_complex;                                                                                            92
  signal s_w: t_complex;                                                                                            93
                                                                                                                    94
  type t_req_state is (idle, mul_fase_1, mul_fase_2, add_sub); -- Estados da maquina de estados para multiplicacao  95
      complexa
  signal s_req_state : t_req_state;                            -- Sinal do estado da maquina de estado            96
begin                                                                                                               97
  -- Somador (Ponto flutuante 32 bits)                                                                            98
  somador_inst: somador_fp32                                                                                        99
    port map(                                                                                                       100
      a => s_add_a,                                                                                                 101
      b => s_add_b,                                                                                                 102
      result => s_add_result,                                                                                       103
      operation_nd => s_add_en,                                                                                     104
      clk => clk,                                                                                                   105
      rdy => s_add_done                                                                                             106
    );                                                                                                              107
  -- Subtrador (Ponto flutuante 32 bits)                                                                          108
  subtrador_inst: subtrador_fp32                                                                                    109
    port map(                                                                                                       110
      a => s_sub_a,                                                                                                 111
      b => s_sub_b,                                                                                                 112
      result => s_sub_result,                                                                                       113
      operation_nd => s_sub_en,                                                                                     114
      clk => clk,                                                                                                   115
      rdy => s_sub_done                                                                                             116
    );                                                                                                              117
  -- Multiplicadores (Ponto flutuante 32 bits)                                                                    118
  mult_inst_1: multiplicador_fp32                                                                                   119
    port map(                                                                                                       120
      a => s_mul_1_a,                                                                                               121
      b => s_mul_1_b,                                                                                               122
      result => s_mul_1_result,                                                                                     123
      operation_nd => s_mul_1_en,                                                                                   124
      clk => clk,                                                                                                   125
      rdy => s_mul_1_done                                                                                           126
    );                                                                                                              127
  mult_inst_2: multiplicador_fp32                                                                                   128
    port map(                                                                                                       129
      a => s_mul_2_a,                                                                                               130
      b => s_mul_2_b,                                                                                               131
      result => s_mul_2_result,                                                                                     132
      operation_nd => s_mul_2_en,                                                                                   133
      clk => clk,                                                                                                   134
      rdy => s_mul_2_done                                                                                           135
    );                                                                                                              136
  -- main                                                                                                         137
  -- Ma'quina de estados para a multiplicacao complexa, contem 3 fases, sendo duas de multiplicacao e uma de soma e 138
      subtracao. Estas ultimas acontecem em paralelo
  main: process(reset_n, clk) is                                                                                    139
  begin                                                                                                             140
    if(reset_n = '0') then                                                                                          141
      s_done <= '0';                                                                                                142
      s_add_en <= '0';                                                                                              143
      s_sub_en <= '0';                                                                                              144
      s_mul_1_en <= '0';                                                                                            145
      s_mul_2_en <= '0';                                                                                            146
      s_result.creal <= (others => '0');                                                                            147
      s_result.cimag <= (others => '0');                                                                            148
      s_req_state <= idle;                                                                                          149
      s_z.creal <= (others => '0');                                                                                 150
      s_z.cimag <= (others => '0');                                                                                 151
      s_w.creal <= (others => '0');                                                                                 152
      s_w.cimag <= (others => '0');                                                                                 153
    elsif(clk'event and clk='1') then                                                                               154
      -- Limpa sinais de ciclo u'nico                                                                               155
      s_add_en <= '0';                                                                                              156
      s_sub_en <= '0';                                                                                              157
      s_mul_1_en <= '0';                                                                                            158
      s_mul_2_en <= '0';                                                                                            159
      s_done <= '0';                                                                                                160
      case (s_req_state) is                                                                                         161
        when idle =>                                                                                                162
          if (en = '1') then -- Aguarda habilitacao                                                                 163
            -- Copia dados de entrada                                                                               164
            s_z <= z;                                                                                               165
            s_w <= w;                                                                                               166
            -- Configurando mul 1                                                                                   167
            s_mul_1_a <= z.creal;                                                                                   168
            s_mul_1_b <= w.creal;                                                                                   169
            s_mul_1_en <='1';                                                                                       170
            -- Configurando mul 2                                                                                   171
            s_mul_2_a <= z.cimag;                                                                                   172
            s_mul_2_b <= w.creal;                                                                                   173
            s_mul_2_en <='1';                                                                                       174
            -- Prox estado                                                                                          175
            s_req_state <= mul_fase_1;                                                                              176
          end if;                                                                                                   177
        when mul_fase_1 =>                                                                                          178
          if(s_mul_1_done = '1') then   -- As multiplicacoes terminam juntas                                        179
            -- Armazena resultados                                                                                  180
            s_sub_a <= s_mul_1_result;                                                                              181
            s_add_a <= s_mul_2_result;                                                                              182
            -- Configurando mul 1                                                                                   183
            s_mul_1_a <= s_z.cimag;                                                                                 184
            s_mul_1_b <= s_w.cimag;                                                                                 185
```

```
                    s_mul_1_en <='1';
                    -- Configurando mul 2
                    s_mul_2_a <= s_z.creal;
                    s_mul_2_b <= s_w.cimag;
                    -- Prox estado
                    s_req_state <= mul_fase_2;
                end if;
            when mul_fase_2 =>
                if(s_mul_1_done = '1') then -- As multiplicacoes terminam juntas
                    s_sub_b <= s_mul_1_result;
                    s_add_b <= s_mul_2_result;
                    s_add_en <= '1';
                    s_sub_en <= '1';
                    s_req_state <= add_sub;
                end if;
            when add_sub =>
                if (s_add_done = '1') then -- add e sub terminam juntos
                    -- Copia resultados
                    s_result.creal <= s_sub_result;
                    s_result.cimag <= s_add_result;
                    s_req_state <= idle;
                    s_done <= '1'; -- Indica final da computacao
                end if;
            when others => null;
        end case;
    end if;
end process main;

-- Conexao dos sinais de saida
r <= s_result;
done <= s_done;
end architecture arch_fp_2_multiplicacao;
```

Listing A.13: Soma Complexa

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;

-------------------------------------------------------------------------------
-- complex_soma
-- Realiza uma soma complexa
-------------------------------------------------------------------------------

entity complex_soma is
    port(
        a           : in t_complex; -- Entrada 1
        b           : in t_complex; -- Entrada 2
        r           : out t_complex;-- Saida
        en          : in std_logic; -- Habilitar
        clk         : in std_logic;
        done        : out std_logic;-- Pronto
        reset_n     : in std_logic  -- Nivel logico baixo
    );
end entity complex_soma;

architecture arch_fp_1_somador of complex_soma is
    -- Componentes
    component somador_fp32 is
        port (
            a: in std_logic_vector(31 downto 0);
            b: in std_logic_vector(31 downto 0);
            operation_nd: in std_logic;
            clk: in std_logic;
            result: out std_logic_vector(31 downto 0);
            rdy: out std_logic
        );
    end component;

    -- Sinal para o resultado da soma
    signal s_result: t_complex;
    -- Para armazenar os dados de entrada
    signal s_r: std_logic_vector(31 downto 0);
    signal s_a: std_logic_vector(31 downto 0);
    signal s_b: std_logic_vector(31 downto 0);
    -- Sinais para o somador
    signal s_soma_done: std_logic;
    signal s_soma_en: std_logic;
    signal s_done: std_logic;

    type t_req_state is (idle, soma_real, soma_imag); -- Difinicao dos estados para a ma'quina de estados
    signal s_req_state : t_req_state;  -- Sinal para a maquina de estado

begin
    -- Somador (Ponto flutuante 32 bits)
    somador_inst: somador_fp32
        port map(
            a => s_a,
            b => s_b,
            result => s_r,
            operation_nd => s_soma_en,
            clk => clk,
            rdy => s_soma_done
        );
    process(clk,reset_n) is
```

```
    begin
        if(reset_n = '0') then
            -- Zera os valores
            s_soma_en <= '0';
            s_a <= (others => '0');
            s_b <= (others => '0');
            s_req_state <= idle;
        elsif(clk'event and clk='1') then
            s_soma_en <= '0';
            s_done <= '0';
            -- Ma'quina de estado para a soma complexa. Como Ã© usado apenas um somador, o processo fica divido em duas fases, a soma dos numeros reais,
            -- e a soma dos numero imaginarios.
            case s_req_state is
                when idle =>
                    if(en = '1') then
                        -- Adiciona ao somador os valores reais
                        s_a <= a.creal;
                        s_b <= b.creal;
                        s_soma_en <= '1';
                        s_req_state <= soma_real;
                    end if;
                when soma_real =>
                    if(s_soma_done = '1') then
                        -- Pega o resultado da soma dos valores reais
                        s_result.creal <= s_r;
                        -- Adiciona ao somados os valores imaginarios
                        s_a <= a.cimag;
                        s_b <= b.cimag;
                        s_soma_en <= '1';
                        s_req_state <= soma_imag;
                    end if;
                when soma_imag =>
                    if(s_soma_done = '1') then
                        -- Pega o resultado da soma dos valores reais
                        s_result.cimag <= s_r;
                        s_done <= '1'; -- Marca a soma complexa como pronta
                        s_req_state <= idle;
                    end if;
                when others => s_req_state <= idle;
            end case;
        end if;
    end process;

    -- Conexao dos sinais de saida
    r <= s_result;
    done <= s_done;
end architecture arch_fp_1_somador;
```

Listing A.14: Subtração Complexa

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.basic_pkg.all;

--------------------------------------------------------------------------------
-- complex_subtracao
-- Realiza uma subtracao complexa
-- r = a - b;
--------------------------------------------------------------------------------

entity complex_subtracao is
    port(
        a           : in t_complex;
        b           : in t_complex;
        r           : out t_complex;
        en          : in std_logic;
        clk         : in std_logic;
        done        : out std_logic;
        reset_n     : in std_logic
    );
end entity complex_subtracao;


architecture arch_fp_1_subtrador of complex_subtracao is
    -- Componentes
    component subtrador_fp32 is
        port (
            a: in std_logic_vector(31 downto 0);
            b: in std_logic_vector(31 downto 0);
            operation_nd: in std_logic;
            clk: in std_logic;
            result: out std_logic_vector(31 downto 0);
            rdy: out std_logic
        );
    end component;

    -- Sinais
    signal s_result: t_complex; -- Resultado da subtracao

    -- Valores de resultados parciais
    signal s_r: std_logic_vector(31 downto 0);
    signal s_a: std_logic_vector(31 downto 0);
    signal s_b: std_logic_vector(31 downto 0);
    signal s_sub_done: std_logic;                   -- Subtracao pronta (componente subtracao)
```

```
    signal s_sub_en: std_logic;                    -- Habilita subtracao
    signal s_done: std_logic;                      -- Subtracao complexa pronta

    type t_req_state is (idle, sub_real, sub_imag); -- Ma'quina de estados da subtracao (acontece em duas partes,
         primeiro a subtracao da parte real e depois a da parte imaginaria)
    signal s_req_state : t_req_state;              -- Estado da ma'quina de estados

begin
    -- Subtador (Ponto Flutuante 32 bits)
    sutrador_inst: subtrador_fp32
        port map(
            a => s_a,
            b => s_b,
            result => s_r,
            operation_nd => s_sub_en,
            clk => clk,
            rdy => s_sub_done
        );

    -- main
    -- M'aquina resposa'vel por controlar as subtacoes (parte real e imaginaria)
    main: process(clk) is
    begin
        if(reset_n = '0') then
            -- Zera Valores
            s_sub_en <= '0';
            s_a <= (others => '0');
            s_b <= (others => '0');
            s_req_state <= idle;
        elsif(clk'event and clk='1') then
            -- Limpa sinais de ciclo u'nico
            s_sub_en <= '0';
            s_done <= '0';
            case s_req_state is -- Ma'quina de estados
                when idle =>
                    if(en = '1') then -- Aguarda habilitacao
                        -- Carrega valores reais para subtracao
                        s_a <= a.creal;
                        s_b <= b.creal;
                        s_sub_en <= '1'; -- Habilita subtacao
                        s_req_state <= sub_real;
                    end if;
                when sub_real =>
                    if(s_sub_done = '1') then -- Aguarda fim a subtracao real
                        s_result.creal <= s_r; -- Copia resultado
                        -- Carrega valores imaginais
                        s_a <= a.cimag;
                        s_b <= b.cimag;
                        s_sub_en <= '1'; -- Habilita subtracao
                        s_req_state <= sub_imag;
                    end if;
                when sub_imag =>
                    if(s_sub_done = '1') then -- Aguarda fim da subtracao imaginaria
                        s_result.cimag <= s_r; -- Copia resultado
                        s_done <= '1'; -- Sinaliza fim
                        s_req_state <= idle;
                    end if;
                when others => s_req_state <= idle;
            end case;
        end if;
    end process main;
    -- Conexao de sinais
    r <= s_result;
    done <= s_done;
end architecture arch_fp_1_subtrador;
```