MARCOS VINÍCIUS DA SILVA

EXPLORAÇÃO DO ESPAÇO DE PROJETO DE ARQUITETURAS RECONFIGURÁVEIS EM ARRANJOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2006

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
S586e
2006

Silva, Marcos Vinícius da, 1973-
Exploração do espaço de projeto de arquiteturas reconfiguráveis em arranjos / Marcos Vinícius da Silva. – Viçosa : UFV, 2006.
xiv, 109f. : il. ; 29cm.

Inclui anexos.
Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 84-88.

1. Arquitetura de computador. 2. Computadores - Projetos e construção. 3. Algoritmos genéticos. I. Universidade Federal de Viçosa. II.Título.

CDD 22.ed. 004.22

MARCOS VINÍCIUS DA SILVA

EXPLORAÇÃO DO ESPAÇO DE PROJETO DE ARQUITETURAS RECONFIGURÁVEIS EM ARRANJOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 28 de agosto de 2006

Dr. Carlos de Castro Goulart
(Co-Orientador)

Dr. Vladimir Oliveira di Iorio
(Co-Orientador)

Dr. Marcus Vinícius Alvim Andrade

Dr. Luigi Carro

Dr. Ricardo dos Santos Ferreira
(Orientador)

À minha esposa Rosana e aos filhos Inácio, Iara e Yuri que, apesar das dificuldades, sempre apoiaram e me ampararam nos momentos mais difíceis.
Aos meus pais José Inácio e Cirene.

AGRADECIMENTOS

- Ao meu orientador, Professor Ricardo dos Santos Ferreira pelo grande apoio e pela dedicação, mesmo nos longos finais de semana que precediam a submissão de artigos.
- Ao Professor Antonio Machado Filho por conseguir condições que me permitiram alcançar este objetivo.
- Às pessoas do Departamento de Informática da Universidade Federal de Viçosa que direta ou indiretamente apoiaram meu trabalho, especialmente a Altino Alves, sempre solícito e disposto a resolver os maiores problemas no menor tempo possível.
- Ao Centro Universitário do Leste de Minas Gerais, que através da Fundação Geraldo Perlingeiro de Abreu concedeu-me bolsa de estudos, tornando possível esta conquista.
- À Universidade Federal de Viçosa.

BIOGRAFIA

Marcos Vinícius da Silva, nascido a primeiro de abril de 1973 em Muriaé-MG, concluiu seus estudos para técnico em eletrônica pelo Colégio Técnico de Coronel Fabriciano no final de 1991. No ano seguinte ingressou no curso de Bacharelado em Informática da Universidade Federal de Viçosa, formando-se em fevereiro de 1996. Desde 1997 é professor do Centro Universitário do Leste de Minas Gerais, onde atua nos cursos de Sistemas de Informação e Engenharia Elétrica. Em agosto de 2006 defendeu sua dissertação, tornando-se Mestre em Ciência da Computação pela Universidade Federal de Viçosa.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

SILVA, Marcos Vinícius da, M.Sc., Universidade Federal de Viçosa, agosto de 2006. **Exploração do espaço de projeto de arquiteturas reconfiguráveis em arranjos**. Orientador: Ricardo dos Santos Ferreira. Co-Orientadores: Carlos de Castro Goulart e Vladimir Oliveira di Iorio.

As aplicações atuais de processamento de sinais em sistemas embarcados demandam soluções de hardware com alto poder de de processamento e baixo consumo de energia. Além disso, estas soluções devem ser escaláveis permitindo fácil adaptação para acompanhar os avanços tecnológicos na construção de circuitos integrados. As arquiteturas de arranjos reconfiguráveis de processadores de grão grosso permitem atender a esta demanda e atualmente são foco de pesquisas que propõem várias maneiras de se realizar sua implementação, com variação de características como topologia, elementos de processamento e capacidade de roteamento, dentre outras. Este trabalho apresenta um algoritmo de posicionamento de unidades funcionais em arranjos de processadores capaz de lidar com variações dessas características, permitindo avaliar a priori seu desempenho para várias arquiteturas. Foi elaborado um conjunto de testes com vários benchmarks de processamento de sinais para validação. Os experimentos mostraram que a solução é rápida, flexível e escalável, permitindo a exploração de uma ampla gama de arquiteturas antes de sua implementação física.

ABSTRACT

SILVA, Marcos Vinícius da, M.Sc., Universidade Federal de Viçosa, August, 2006. **Design space exploration for array reconfigurable architectures**. Adviser: Ricardo dos Santos Ferreira. Co-advisers: Carlos de Castro Goulart and Vladimir Oliveira di Iorio.

Current applications of signal processing in embedded systems demand hardware solutions with high processing power and low consumption. Moreover, these solutions must be scaled allowing easy adaptation in order to follow technological advances in the construction of integrated circuits. Coarse-grained reconfigurable array processors architectures allow to attend this demand, and nowadays are research focus that considers several ways to deal with it's implementation, with variation of characteristics as topology, processing elements and routing capacity, among others. This work presents a placement algorithm in arrangements of processors capable to deal with variations of these characteristics, allowing early evaluation of its performance for several architectures. A test set with several DSP benchmarks was elaborated for validation. The experiments shows that the solution is fast, flexible and scalable, allowing exploration of an ample architecture spectrum before physical implementation.

1 INTRODUÇÃO

1.1 Motivação

O mercado de sistemas embarcados vem atualmente demandando uma grande capacidade de processamento, incluindo aplicações multimídia envolvendo áudio, vídeo e processamento de sinais. O dinamismo deste mercado exige que as soluções de software e hardware atendam aos requisitos de consumo reduzido de energia e alta capacidade de processamento. Além disto, estas soluções devem ser escaláveis para poder aproveitar melhor os avanços da tecnologia de construção dos circuitos integrados, flexíveis para poder acomodar mudanças nas aplicações e geradas em tempo reduzido para inserção mais rápida no mercado.

A implementação de sistemas embarcados atuais pode ser feita através de sistemas baseados em software específico executado em em microprocessadores de uso geral, construção de circuitos integrados ASIC (*Application Specific Integrated Circuit* — Circuito integrado para aplicação específica) e mais recentemente através de circuitos reconfiguráveis como os FPGA (*Field Programmable Gate Array* - Arranjos de portas programáveis por efeito de campo). (DEHON; WAWRZYNEK, 1999).

A solução por software tem como vantagens a flexibilidade e o tempo de desenvolvimento reduzido, enquanto a solução por chips ASIC privilegia o desempenho. As desvantagens da abordagem ASIC são o tempo prolongado, o alto custo e a complexidade do desenvolvimento do projeto. Já para os sistemas de software, há a dificuldade de explorar operações paralelas inerentes a vários problemas de processamento de sinais multimídia. Ambas as soluções têm uma longa história de exploração e pesquisa. Segundo Keutzer, Malik e Newton (2002), soluções intermediárias como as arquiteturas ASIP (*Application Specific Instruction Set Processor*) vêm surgindo graças ao aparecimento de plataformas de hardware reconfiguráveis após a fabricação. Existem também soluções comerciais de alto desempenho, em geral, arquitetu-

ras fechadas, compostas por clusters de processadores superescalares com placas aceleradoras baseadas em hardware reconfigurável, como o supercomputador DeChyper da Timelogic, (http://www.timelogic.com) para aplicações de bioinformática.

Neste contexto, as soluções intermediárias surgem tentando aliar a flexibilidade das soluções baseadas em software, com o desempenho das soluções de hardware (KEUTZER; MALIK; NEWTON, 2002). Dentre as opções, aparecem atualmente como foco de várias pesquisas, as arquiteturas reconfiguráveis de grão grosso que provêem a possibilidade de execução paralela de tarefas, conseguindo ganhos de desempenho e eficiência em termos de operações realizadas *versus* potência consumida. E por possuírem a capacidade de reconfiguração, são flexíveis assim como as soluções por software. Pelo mesmo motivo, o tempo de desenvolvimento é reduzido em comparação com ASIC, pois são baseados em sistemas de prototipagem rápida. Uma primeira versão da aplicação pode ser mapeada na arquitetura, validada e posteriormente melhorada.

A idéia central para tais arquiteturas é a distribuição das computações de um algoritmo para vários elementos de processamento (PE - *Processing Element*), de forma que sejam executadas em paralelo. Um enfoque de projeto muito utilizado é a combinação de um arranjo de PE's interligados com um processador (CPU). O processador executa as tarefas com pouco grau de paralelismo, enquanto os núcleos centrais dos algoritmos (como *loops* em matrizes e vetores) são executados no arranjo(SAWITZKI; GRATZ; SPALLEK, 1998).

A busca por arquiteturas mais apropriadas para processamento intensivo de dados baseadas nas arquiteturas com arranjos e na utilização de fluxo de dados no lugar do fluxo de instruções vem se tornando uma importante opção para se obter desempenho adequado, sem consumo elevado de energia em comparação com a geração atual de processadores superescalares (WAN et al., 1999). Estes processadores têm estruturas complexas, demandando grande quantidade de área para os circuitos integrados. Para obter o desempenho desejado, são necessárias freqüências de operação elevadas, resultando em alto consumo de energia. A escalabilidade é um fator chave, já que a tecnologia oferece a cada ano, um aumento na densidade de integração (ALLAN et al., 2002).

O projeto de arquiteturas reconfiguráveis é atualmente uma área de pesquisa que tem atraído vários pesquisadores interessados em investigar soluções de hardware que possam integrar alto desempenho e baixo consumo de energia(RABAEY, 1997). Além disso, tais arquiteturas pos-

sibilitam a implementação de uma solução regular e escalável (HARTENSTEIN, 2001b), explorando o paralelismo intrínseco das aplicações (VENN, 1986).

1.2 Modelos de Computação

Segundo Hartenstein (2001a), pode-se classificar a evolução dos sistemas seguindo o paradigma de Tredennick em três fases: *hardwired* (Década 60), programação baseada em instruções (Década 70, 80 e 90), programação estrutural (a partir de 1997). A primeira fase tem como dominante os sistemas onde tanto os recursos quando os algoritmos eram fixos em hardware. A segunda fase é marcada pelo uso do modelo Von Neumann, processador-memória e variações, onde os recursos são fixos (hardware), mas o algoritmo é flexível (software). Na terceira fase temos tanto o algoritmo flexível como também os recursos, deixando de ser um modelo orientado pelo fluxo de instruções, passando a ser um modelo orientado pelo fluxo de dados. Em termos de implementação, iniciou-se com o surgimento dos circuitos reconfiguráveis de granularidade fina (bits), ou FPGAs que apareceram no início dos anos 90. Atualmente, os circuitos com grão grosso também denotados por *Coarse Grained ou rDPA* (*reconfigurable DataPath Array*), vêm sendo pesquisados como uma solução que agrega flexibilidade, desempenho e redução do consumo de energia (COMPTON; HAUCK, 2002).

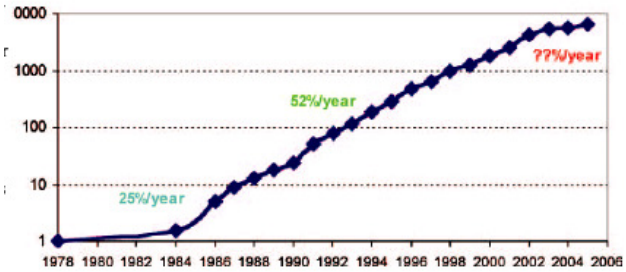


Figura 1: Evolução do desempenho dos processadores por ano, extraída de (PATTERSON, 2006)

Em um artigo recente, David Patterson (PATTERSON, 2006) avalia o potencial dos FPGAs e a importância dos currículos dos cursos de ciência da computação incorporarem o assunto. Primeiro, a lei de Moore continua válida para o aumento da taxa de integração, porém a evolução dos processadores deixou de acompanhar tal lei devido a perdas de desempenho como ilustra a Figura 1. Patterson aponta como principais fatores para esta degradação: o consumo de

energia, a exploração do paralelismo, a latência das memórias. Na década de 80-90 presenciamos uma taxa de crescimento da ordem de 50% ao ano, graças ao uso intensivo de pipeline, as técnicas de exploração de paralelismo (escalonamento dinâmico, previsão de desvio, hierarquia de memória de caches em vários níveis e funções). Entretanto, desde o final dos anos 90, com a redução da taxa de crescimento devido a uma série de fatores, duas tendências vem sendo mais exploradas: os multiprocessadores e os FPGAs. Os multiprocessadores, como em um chip *dual core*, buscam a exploração do paralelismo a nível de threads. Já os FPGA podem explorar o paralelismo em um grão mais fino e/ou mudar completamente o paradigma de implementação baseado em fluxo de instruções. Esta solução vem se expandindo, como mostra a indústria de FPGA que tem obtido um faturamento da ordem de 1 bilhão de dólares por ano (PATTERSON, 2006).

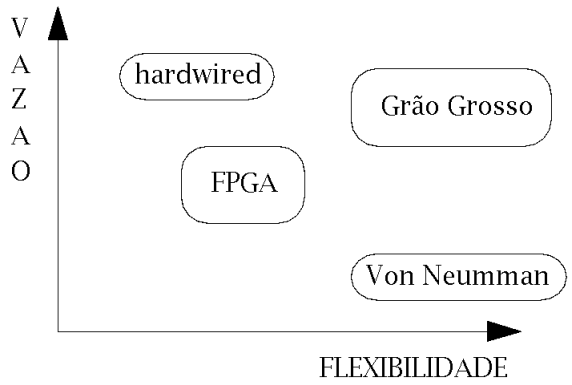


Figura 2: Vazão *versus* flexibilidade para os paradigmas de computação

A Figura 2 mostra a relação entre vazão e flexibilidade para os modelos de computação. A solução *hardwired* perde em flexibilidade, mas como é feita sob medida para a aplicação pode ter uma vazão muito alta. No outro extremo temos a solução baseada no modelo de Von Neumman, onde a flexibilidade é total, basta alterar o software, porém perde em vazão, pois o hardware e o conjunto de instruções é fixo. Já um FPGA ou um grão grosso possibilitam um meio termo entre vazão e flexibilidade. Como o hardware é gerado pelo compilador e pode ser alterado dinamicamente, o sistema é flexível, o que permite gerar um hardware específico para a aplicação, que por sua vez irá explorar o paralelismo e gerar uma vazão alta. Uma das diferenças importantes entre a granularidade fina (bit) ou grossa (palavras) é a complexidade para a síntese e/ou mapeamento das aplicações.

Os FPGA necessitam de ferramentas a serem desenvolvidas para serem mapeados de forma rápida (flexibilidade). Segundo dados extraídos de (CONG; XU, 2000), em 1988 um FPGA da *Altera* (http://www.altera.com)tinha o equivalente a 3000 portas lógicas reconfiguráveis, e uma década após, já tinha o equivalente a 1 milhão em meio de portas. Hoje, a tecnologia dos FPGA já está próxima da barreira de 100 milhões de portas, que vem refletindo nos últimos anos em um maior taxa de crescimento das indústrias de FPGA em comparação com as de fabricantes de outras soluções. No período de 1994 a 1998, enquanto a Intel teve um crescimento de 24% por ano, a Altera teve um crescimento de 36% por ano, e a indústria de ASIC's teve uma taxa de 16%. As indústrias de semicondutores como um todo tiveram um crescimento de 28% no mesmo período. Segundo Cong e Xu (2000), um dos autores com mais publicações em algoritmos de mapeamento para FPGA, já existia desde do final da década de 90, uma enorme demanda para ferramentas de síntese para explorar o potencial dos FPGA de alta densidade e alto desempenho. Cong sugere o uso de hierarquia e recursos heterogêneos. No final da década de 90, o atraso para conectar elementos lógicos fora das células base dominava o atraso consumido durante o cálculo dentro da célula. Por exemplo, o atraso de uma porta lógica configurável era de 2ns e uma conexão local tinha o atraso de 0,4ns enquanto uma conexão fora da célula passando por linhas ou colunas de roteamento do FPGA, gerava um atraso da ordem de 5 a 10ns. Os FPGAs começaram a incorporar recursos heterogêneos com granularidade maior como, por exemplo, células de memória e multiplicadores distribuídos pelo circuito, enquanto a granularidade da célula básica continua baixa (1-4 bits). Devido ao tamanho do espaço de projeto, o grão muito fino leva a uma complexidade muito alta em todos os níveis do processo de projeto, da síntese ao mapeamento tecnológico.

Por outro lado, os circuitos baseados em granularidade maior (1 palavra de 16 ou 32 bits), por serem próximos do paradigma atual onde um processador que possui várias unidade de cálculo, demandam poucas alterações nos compiladores e nas linguagens para poder oferecer uma flexibilidade tanto no nível de ferramentas de síntese, e ao mesmo tempo por serem mais fáceis de fabricar e tirar proveito da alta densidade de integração que as tecnologias de silício oferecem.

Um outro ponto importante neste aspecto é a eficiência em termos de energia, ou bilhões de operações de ponto flutuante por watt, denotado por Gflops/W. Como exemplo, o processador Pentium 4 da Intel com *clock* de 3,8 GHZ consegue desempenho de pico de 12 Gflops consu-

mindo 80 watts, o que resulta em uma eficiência energética de 0,15 Gflop/watt. O processador baseado em arquitetura reconfigurável *Imagine*, voltado para aplicações de processamento de dados em *streamming*, atinge a eficiência energética de 2,4 Gflops/watt (KAPASI et al., 2003).

A computação reconfigurável vem ganhando espaço nos últimos anos (TESSIER; BURLESON, 2000; COMPTON; HAUCK, 1999). Recentemente os arranjos de grão grosso vêm sendo acoplados a processadores superescalares (HARTENSTEIN et al., 2000; BOSSUET; GOGNIAT; PHILIPPE, 2003), acelerando a execução de aplicações e reduzindo o consumo de energia. Por exemplo, as novas gerações de rede de sensores e sistemas sem fio necessitam de processamento digital de sinais que possibilitem alto desempenho, baixo consumo de energia, flexibilidade para se adaptarem aos novos padrões em formação. Os telefones celulares são outro exemplo onde existe a demanda de um uso intensivo de processamento de sinais para a comunicação, recursos multimídia e outros.

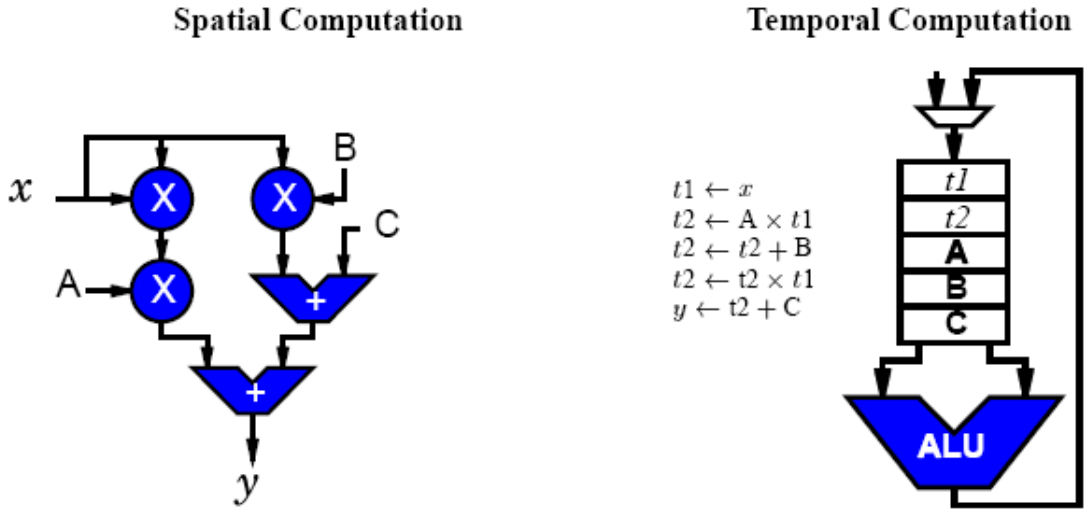


Figura 3: Computação espacial *versus* computação temporal (DEHON; WAWRZYNEK, 1999)

As arquiteturas em arranjos de elementos de processamento (PE) possibilitam a exploração da computação espacial e temporal (DEHON; WAWRZYNEK, 1999). Os PEs podem ser configurados para executarem várias operações ao longo do tempo e podem trabalhar concorrentemente, gerando um alto grau de paralelismo. A Figura 3 exemplifica o conceito para o cálculo da equação $y = ax^2 + bx + c$ onde x é a entrada e y a saída. No modelo de computação espacial, cada operador está localizado em um ponto no espaço permitindo aproveitar as características de paralelismo das operações. No modelo de computação temporal um pequeno

número de recursos computacionais de uso geral são compartilhados no tempo.

Os arranjos de grão grosso podem tirar proveito das altas taxas de integração (BUDIU; GOLDSTEIN, 2002) e usar modelos de conexões locais ponto a ponto. Um aspecto importante a ser explorado são as ligações regulares heterogêneas. Linhas longas com saltos podem ter o atraso similar a conexões locais. Por exemplo, saltar uma ou cinco células pode ter o mesmo custo (TAGHAVI et al., 2004). Nos reconfiguráveis a distância física entre dois nodos e o atraso não tem um correlação direta como nos circuitos dedicados. Um FPGA Virtex-II da Xilinx (http://www.xilinx.com) já incorpora um padrão octal, 1-hop e saltos de 2 em 2. Um ponto muito importante é a exploração das topologias escaláveis.

1.3 Arquiteturas Reconfiguráveis

Uma solução para se melhorar o desempenho dos processadores é a construção de software/hardware que permita a exploração da execução de tarefas em paralelo, aumentando o desempenho geral do sistema. Dentre as várias formas de se implementar o paralelismo em computadores (DUNCAN, 1990), as arquiteturas de fluxo de dados parecem ser uma boa escolha para auxiliarem as arquiteturas usuais em aplicações específicas com processamento intensivo de dados.

Arquiteturas de processadores em arranjos têm sido propostas como extensões para sistemas baseados em microprocessador. Arquiteturas em arranjos, usam células contendo elementos de processamento e/ou memória inter-conectadas por padrões regulares. Cada uma das células de um arranjo pode ser programada para executar certas operações aritméticas, lógicas, deslocamento ou condicionais. Várias células podem executar concorrentemente, com isto pode-se obter altos índices de paralelismo ao nível de instrução (*Instruction Level Paralelism — ILP*) (GALANIS; DIMITROULAKOS; GOUTIS, 2005).

De acordo com a granularidade de cada célula, arquiteturas reconfiguráveis em arranjos são classificadas como “grão fino” ou “grão grosso”. Como exemplos de arquiteturas de grão fino, podemos citar as arquiteturas de FPGA (*Field Programmable Gate Array* - Arranjo de Portas Programados por Campo), largamente utilizadas para implementação de circuitos digitais. Estas arquiteturas consistem de células com pequeno número de bits de entrada e saída, da ordem de 1 a 4 bits. As arquiteturas de grão grosso possuem células base que podem manipular um

maior número de bits (16, 32, etc.). As arquiteturas KressArray (HARTENSTEIN; KRESS; HEINIG, 1994), XPP (CARDOSO; WEINHARDT, 2002) e Morphosys (SINGH et al., 1998) são classificadas como arquiteturas de grão grosso, onde cada célula base implementa operações em valores com vários bits. Outra tendência são os NoC (Network on Chip), que tiram proveito da grande área do chip para implementar vários processadores em um único chip (BJERREGAARD; MAHADEVAN, 2006; BENINI; MICHELI, 2002). A granularidade dos NoC é bem maior que a de grão grosso, e os problemas de roteamento usam a abordagem dinâmica da comunicação entre os processadores, semelhantes às redes de computadores e clusters. Porém, novas topologias de interconexão podem ser exploradas e implementadas de forma eficiente dentro de um único chip.

Arquiteturas de grão grosso são mais apropriadas para implementar aplicações com processamento orientado ao fluxo de dados (COMPTON; HAUCK, 2002). Tais aplicações apresentam paralelismo natural em suas operações. O controle pode ser distribuído para cada célula utilizando um protocolo do tipo "*ready-acknowledge*" (HOLZMANN, 1991). Por não necessitar de controle centralizado, grande parte da complexidade do projeto é removida.

As operações no modelo de computação de fluxo de dados são escalonadas dinamicamente pelo próprio fluxo dos dados (VENN, 1986), com isto estruturas de controle centralizadas não são necessárias. Como vantagem adicional, o processamento assíncrono dos dados leva a um menor consumo de energia. As unidades só são ativadas quando existirem os operandos necessários para sua computação.

1.4 Arquiteturas de Grão Grosso

Os arranjos com grão grosso podem ser baseados em modelos mapeamento estático de grafos de fluxo de dados das aplicações. Exemplos como o sistema comercial da XPP (PACT-XPP-TECHNOLOGIES, 2002) e o KressArray (HARTENSTEIN et al., 2000). O controle do fluxo é distribuído e gerencia caminhos desbalanceados. A implementação pode ser assíncrona, dispensando o uso de controle centralizado com escalonamento. Os arranjos são escaláveis, graças ao desenho regular e a estrutura simétrica de conexões. Um alto grau de paralelismo, redução do consumo de energia, tolerância a falhas e redução do ciclo de projeto podem ser obtidos em arranjos regulares de grão grosso. Entretanto, muitos sistemas tem sido apresentados

sem justificarem como e porque decisões de projetos foram tomadas.

O trabalho dessa dissertação aborda o tema de exploração do espaço de projeto, que vem sendo estudado por vários pesquisadores (BANSAL et al., 2003; ZABEL et al., 2005; MA JOHN KNIGHT, 2005), e que com raras exceções têm explorado vários aspectos dos arranjos de grão grosso (HARTENSTEIN et al., 2000; MEI et al., 2005; NAGELDINGER, 2001; FERREIRA et al., 2005; THAM; MASKELL, 2005). Alguns trabalhos usam o algoritmo de *module scheduling* para o mapeamento espacial e temporal (GALANIS; DIMITROULAKOS; GOUTIS, 2005; BANSAL et al., 2003), permitindo a exploração de alguns parâmetros, mas ficam limitados a avaliação de arranjos com dimensões de 16 a 64 elementos. Em (BOSSUET; GOGNIAT; PHILIPPE, 2003, 2005), um modelo baseado em um grafo reduzido é apresentado para avaliar arquiteturas hierárquicas. O grafo reduzido é gerado a partir do grafo da aplicação e serve de métrica para comparar arquiteturas hierárquicas como ilustrado na Figura 4.

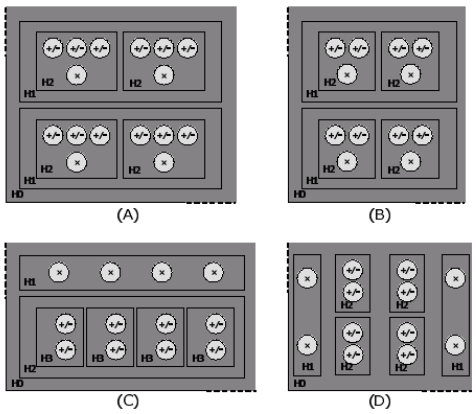


Figura 4: Quatro arquiteturas com níveis hierárquicos para distribuição de recursos e conexões. (Extraído de (BOSSUET; GOGNIAT; PHILIPPE, 2003))

Um arranjo pode ser definido como um conjunto de PEs (*Processing Elements* - Elementos de processamento) inter-conectados por diferentes topologias. Cada PE pode ter internamente uma ou várias unidades funcionais (FUs - Functional Units), ou pode ser simplesmente um elemento de roteamento. O arranjo pode estar acoplado a um processador e a uma unidade de controle do arranjo. Esta unidade pode ter uma cache de configurações e um sistema de gerência, possibilitando o mapeamento espacial e a partição temporal das aplicações, como o XPP (PACT-XPP-TECHNOLOGIES, 2002), quando o número de recursos necessários excede a capacidade do arranjo.

Alguns ambientes comerciais para grão grosso, incluem ferramentas de simulação como as XDS da PACT (PACT-XPP-TECHNOLOGIES, 2002). O usuário pode especificar o número e largura dos barramentos, o número de PEs, etc. Um outro exemplo é o RAW (TAYLOR et al., 2004) do MIT, que apresenta uma arquitetura em fatias ou estágios. Na arquitetura do RAW a capacidade de memória, de processamento, comunicação e processamento de cada estágio pode ser configurada, mas o ambiente não permite a exploração de diferentes topologias.

Já o KressArray Xplorer é capaz de tomar decisões de projeto usando um algoritmo de inferência baseado em lógica nebulosa (*fuzzy*). A Figura 5 ilustra vários tipos de topologias explorados no KressArray (HARTENSTEIN et al., 2000). Cada PE pode ter um ou dois operadores, as ligações com os vizinhos podem ser uni ou bi-direcionais, barramentos verticais e horizontais podem ser definidos. O algoritmo de posicionamento é baseado em *simulated annealing* e o roteamento no algoritmo *pathfinder*.

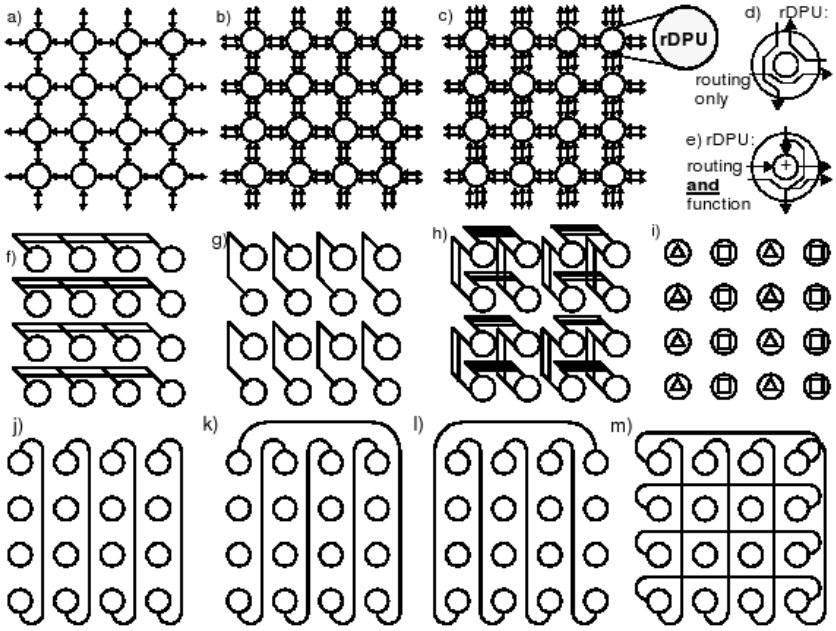


Figura 5: Várias topologias do KressArray (extraído de (HARTENSTEIN et al., 2000)).

Neste contexto, podemos afirmar que a combinação do modelo de computação de fluxo de dados com arquiteturas de arranjo de processadores em grão grosso é muito promissora para implementar soluções em várias áreas de aplicação.

O desenvolvimento de arquiteturas eficientes para o processamento no modelo de fluxo de dados implica em uma série de decisões de projeto, dentre as quais podemos citar:

1. Linguagens de Descrição das Arquiteturas/Aplicações;
2. Particionamento Software/Hardware Reconfigurável;
3. Características da Arquitetura:
   (a) Escolha da topologia de interconexões das células;
   (b) Posicionamento e Roteamento das células no arranjo de processadores;
   (c) Determinação da arquitetura interna de cada célula (protocolo de comunicação, tamanho das filas de entrada e saída, processamento, etc.).

1.5 Posicionamento e Roteamento

O posicionamento das unidades funcionais e o roteamento das ligações entre estas unidades são pontos cruciais no mapeamento das aplicações em uma arquitetura alvo. Várias técnicas para resolução destes problemas foram desenvolvidas para chips ASIC e posteriormente para FPGA. Geralmente, os dois problemas são tratados de forma separada por algoritmos independentes. Entretanto, alguns trabalhos (CRONQUIST, 1995) realizam o posicionamento e o roteamento simultaneamente. Algumas das técnicas mais utilizadas para posicionamento e roteamento são descritas brevemente, com o objetivo de contextualizar o trabalho desenvolvido nesta dissertação. A topologia com a qual foram arranjadas os PE's no arranjo é um fator determinante para o posicionamento e o roteamento. Topologias com maior grau de conectividade entre células geram resultados melhores. Entretanto, disponibilizar mais conexões implica em maior custo para implementação do circuito (BANSAL et al., 2003).

Posicionamento

O posicionamento das células é um problema que consiste em arranjar o mais próximo possível células relacionadas, facilitando sua posterior interligação. Quanto mais próximo estiverem as células relacionadas, menores serão as ligações entre elas e, portanto, menor também será o atraso do sinal e menos recursos de roteamento (fios) serão necessários. O posicionamento de células é um problema conhecidamente NP-completo (GUDISE; VENAYAGAMOORTHY, 2004). Devido a isto, técnicas de otimização combinatória são largamente utilizadas para sua solução. A técnica de têmpera simulada (*simulated annealling*) é uma das escolhas

mais freqüentes (COMPTON; HAUCK, 2002) e consiste de partir de um posicionamento inicial aleatório e realizar movimentações das unidades buscando um ganho geral na distância entre PE's relacionados. Os algoritmos genéticos são outra escolha para solucionar o problema (HARTENSTEIN, 2001b). Uma população inicial de indivíduos que representam posicionamentos é evoluída através de cruzamentos e mutações para encontrar uma solução melhor adaptada que representa um melhor posicionamento (KAHNE, 1997).

Alguns algoritmos propostos para máquinas MIMD e SIMD com organizações em arranjos ou máquinas de fluxo de dados podem ser semelhantes ao contexto dos arranjos reconfiguráveis de grão grosso, como por exemplo os trabalhos apresentados por (KOREN et al., 1988; ROBIC; SILC, 1994b). Um algoritmo apresentado por Koren et al. (1988), baseado na busca em profundidade do grafo de aplicação, (ROBIC; SILC, 1994b) propõe uma otimização no trabalho anterior (KOREN et al., 1988) com a técnica de *simulated annealing*. Entretanto, ambos os trabalhos fazem apenas a avaliação da topologia hexagonal com graus fixos de interconexão. Topologias organizadas de forma hierárquica foram avaliadas em (BOSSUET; GOGNIAT; PHILIPPE, 2003), porém apenas o mapeamento em pequenos arranjos com menos de 20 PEs heterogêneos com alocação dinâmica de recursos foi avaliado. O foco principal é a relação entre os atrasos das conexões em níveis diferentes da hierarquia. Um trabalho de avaliação de topologias em grid, 1-hop e 2-hop foi apresentado em (BANSAL et al., 2004a), mostrando que a topologia 1-hop gera os melhores resultados. Entretanto, outras topologias como octal ou hexagonal não foram avaliadas. Além disso, os arranjos tem da ordem de 16-32 elementos, e o mapeamento é espacial-temporal com *module scheduling*. Recentemente, um algoritmo polinomial (LAI; LAI; YEH, 2005) foi proposto para a arquitetura rDPA ilustrada na Figura 6. O maior problema desta solução é a geração de um posicionamento com baixa densidade de ocupação do arranjo. Além disso, o algoritmo polinominal proposto por Lai, Lai e Yeh (2005) é limitado a uma única topologia onde os PEs têm apenas duas entradas e duas saídas. Outro exemplo de mapeamento espacial e temporal com *module scheduling* e exploração das características de algumas topologias com grid, 1-hop e agrupamentos foi apresentado por Mei et al. (2005). Arranjos heterogêneos e restrições de conexão também são avaliados no referido trabalho. Os resultados mostram que a topologia 1-hop apresenta resultados melhores que os agrupamentos baseados na arquitetura Morphosys que consiste em agrupamentos locais de 4, 8 ou 16 PEs onde todos são conectados com todos e num segundo nível existem conexões globais entre os agrupamentos. Um algoritmo genético é apresentado em (MA JOHN KNIGHT, 2005),

em que vários parâmetros do algoritmo são avaliados porém a arquitetura alvo é fixa e composta por um número reduzido de PEs.

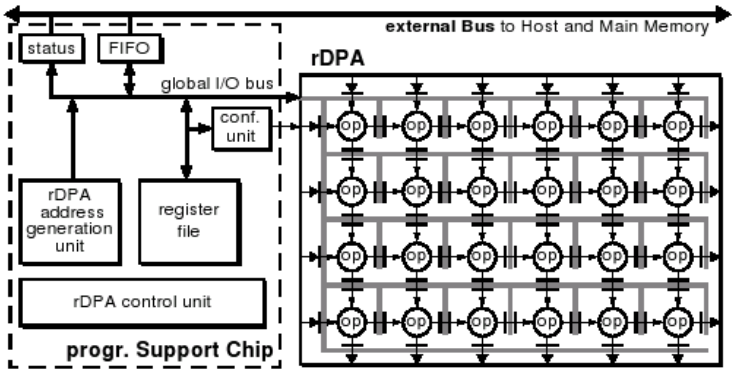


Figura 6: Reconfigurable Datapath Array - rDPA (Extraído de (LAI; LAI; YEH, 2005))

Um bom posicionamento deixa mais próximos PE's relacionados, reduzindo o uso de recursos de roteamento e implicando em maior possibilidade de sucesso para o posterior roteamento das células (COMPTON; HAUCK, 2002).

Roteamento

O roteamento consiste em realizar as ligações entre as células relacionadas. Cada uma das ligações pode ser implementada como uma conexão direta entre células adjacentes ou como uma ligação indireta, passando por outras células intermediárias. O problema do roteamento também é NP-completo (COMPTON; HAUCK, 2002) e para sua resolução, assim como no problema de roteamento, são utilizadas heurísticas.

O algoritmo *pathfinder* (MCMURCHIE; EBELING, 1997) faz um balanceamento entre o desempenho e a capacidade de roteamento e tenta chegar na solução ótima que atenda aos dois quesitos. Uma estratégia de negociação, onde sinais mais críticos têm prioridade é utilizada para reduzir o atraso.

O roteamento hierárquico divide a área do arranjo em duas partes e para cada ligação que cruza a divisão entre as partes é utilizada uma estratégia de negociação semelhante à do algoritmo *pathfinder*. Depois, repete-se o processo recursivamente para cada uma das partes. Quando cada parte ficar pequena o bastante, utiliza-ze um algoritmo guloso para rotear as ligações restantes (MCMURCHIE; EBELING, 2001).

1.6 Escopo da Dissertação

Este trabalho visa implementar soluções para o problema de posicionamento das unidades funcionais em arquiteturas reconfiguráveis de grão grosso a partir de uma descrição funcional de um algoritmo representado por um grafo de fluxo de dados. O mapeamento deve ser flexível, para poder lidar com diferentes topologias e tipos de elementos de processamento.

Várias topologias foram utilizadas para o mapeamento, permitindo uma comparação direta entre elas para verificar a sua aplicabilidade a diversos tipos de problemas reais. As implementações realizadas foram submetidas a testes com vários *benchmarks* como algoritmos de processamento de imagens, áudio e vídeo. Os resultados obtidos foram comparados com os de outros trabalhos correlatos.

A abordagem desta dissertação difere das precedentes em alguns aspectos. O ambiente de exploração não é restrito a arquiteturas específicas (isto é, baseado em malhas) e pode portanto suportar uma ampla variação de padrões de conexão e topologias. O ambiente faz uso do paradigma de orientação a objetos e a carga dinâmica de classes, características disponíveis na linguagem Java. O uso de dialetos XML para a especificação das propriedades com geração automática de código simplifica a extensão do ambiente dando suporte a aperfeiçoamentos incrementais. Os próximos capítulos apresentam o ambiente de exploração, o espaço de projeto das várias topologias, os algoritmos de posicionamento e seus resultados.

Esta dissertação está organizada em Capítulos brevemente descritos a seguir. O Capítulo 2 descreve o ambiente EDA, onde foram integradas as soluções de posicionamento descritas nesta dissertação. O Capítulo 3 descreve as várias topologias utilizadas para organização dos arranjos de processadores. No Capítulo 4, são apresentadas as soluções de posicionamento de unidades funcionais em um arranjo, posteriormente avaliadas para várias topologias. Será apresentada uma solução de posicionamento por níveis e duas outras novas soluções de posicionamento: posicionamento do caminho crítico e posicionamento por algoritmo genético. Os resultados obtidos na avaliação das soluções apresentadas foram exibidos e discutidos no Capítulo 5. O Capítulo 6 traz as conclusões derivadas deste trabalho e sugestões de trabalhos futuros. O Anexo 1 lista os códigos para os *benchmarks* utilizados e o Anexo 2 exibe os grafos de fluxos de dados gerados a partir desses códigos.

2 AMBIENTE DE EXPLORAÇÃO DE ARQUITETURAS - EDA

2.1 Introdução

Como mencionado anteriormente, várias arquiteturas vêm sendo apresentadas e avaliadas como soluções para implementações reconfiguráveis de grão grosso (HARTENSTEIN, 2001a). Porém, poucos ambientes de projeto foram propostos para explorar as diversas características que estas arquiteturas podem apresentar (NAGELDINGER, 2001; HARTENSTEIN et al., 2000; MEI et al., 2005). Neste capítulo será apresentado o ambiente de projeto EDA (environment for EXPLORATION of DATA-DRIVEN ARRAY Architectures)(FERREIRA; CARDOSO; NETO, 2004; FERREIRA et al., 2005), que serviu de base para implementação dos algoritmos de posicionamento propostos nesta dissertação. O termo EDA é muito usado na literatura para descrever a categoria de ferramentas automatizadas para projeto de circuitos eletrônicos em geral, sendo a sigla de Electronic Design Automation (EDA).

O ambiente EDA é baseado em uma abordagem incremental. Desta forma novas estratégias de posicionamento e roteamento, bem como novas topologias podem ser implementadas, uma vez que a exploração de arquiteturas reconfiguráveis é uma área muito dinâmica. A metodologia de desenvolvimento faz uso do paradigma de orientação a objetos e da definição de formatos intermediários, que foram implementados em Java e XML, respectivamente.

A Figura 7 mostra o diagrama de blocos do ambiente EDA e suas principais características:

- Especificação da Aplicação;
- Definição de Operadores (granularidade, funcionalidade, atraso, consumo de energia, tamanho das filas, etc.);
- Definição da Arquitetura (topologia, padrão de conexões, atraso, consumo de energia,

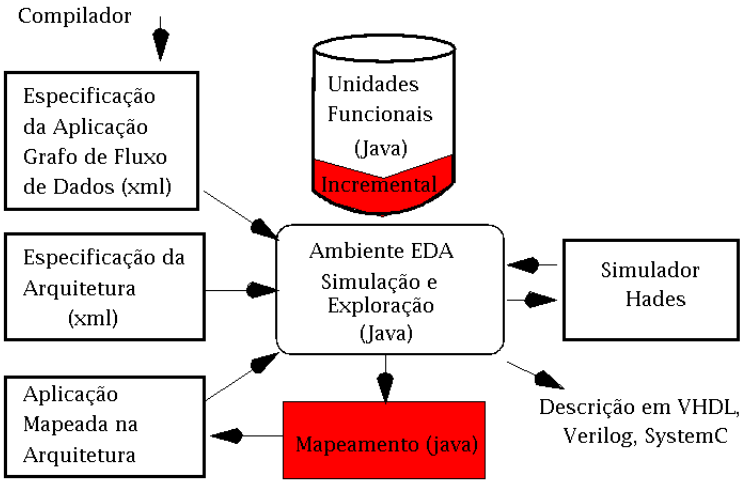


Figura 7: Diagrama de blocos do ambiente EDA

etc.);

- Simulação da Aplicação com ou sem restrições de mapeamento;
- Inclusão de novos algoritmos de posicionamento e roteamento.

Uma descrição XML foi utilizada para fazer a especificação da aplicação. Esta descrição pode ser gerada por um compilador. A definição de um formato XML possibilita portabilidade e capacidade de incorporação de extensões futuras. Como é um ambiente de exploração, a arquitetura também é especificada por uma descrição no formato XML, permitindo a definição de vários parâmetros. O ambiente avalia as arquiteturas através da simulação das aplicações, usando uma abordagem quantitativa (PATTERSON; HENNESSY, 1996). A simulação utiliza como base o simulador Hades (HENDRICH, 2000), que é implementado em Java. Através de uma biblioteca incremental de componentes, novas características das arquiteturas podem ser facilmente adicionadas, como por exemplo novos operadores, protocolos ou topologias. A simulação se faz em dois níveis, antes e depois do mapeamento. A simulação da aplicação antes do mapeamento possibilita uma avaliação das características independente da tecnologia alvo. Posteriormente, a aplicação é mapeada na arquitetura alvo e pode ser avaliada levando em consideração as restrições impostas pela arquitetura e pelo mapeamento. O mapeamento é realizado em duas etapas. Primeiro os vértices do grafo que descreve a aplicação são posicionados,

posteriormente é realizado o roteamento, conectando os operadores.

Esta dissertação aborda os algoritmos de posicionamento, bem como a variação e especificação de topologias para exploração das arquiteturas. Neste capítulo serão detalhadas as características gerais do ambiente EDA, e nos capítulos 3 e 4 serão abordados os temas de definição das topologias e dos algoritmos de posicionamento.

2.2 Especificação da Aplicação

Uma aplicação que será implementada em um arquitetura reconfigurável pode ser descrita em diversas formas e níveis de abstração. Neste trabalho foram considerados algoritmos de processamento de sinais com implementações em linguagem C. Porém, como o formato de descrição e especificação é aberto, expansões futuras podem ser facilmente acrescentadas. O código C pode ser mapeado em um grafo de fluxo de dados através de técnicas de compilação conhecidas, como as apresentadas em (BUDIU; GOLDSTEIN, 2002). Normalmente, os núcleos ou cernes do algoritmo dominam o tempo de execução da aplicação. Este núcleos são compostos por laços com manipulação intensiva de vetores e matrizes de dados. Em geral, os códigos possuem um alto grau de paralelismo a ser explorado.

Nesta seção são detalhados os grafos de fluxos de dados e seus operadores. Em seguida, dois exemplos clássicos de processamento de sinais serão descritos para ilustrar como o paralelismo pode ser explorado. O primeiro exemplo é um filtro de sinais, muito utilizado para tratamento de sinais de áudio e telecomunicações. Sua implementação faz uso de dados implementados em vetores. O segundo exemplo é uma transformada discreta, muito usada no tratamento de imagens, e faz uso de dados representados por matrizes. Finalmente, será detalhado como o grafo de fluxo de dados pode ser avaliado e explorado antes da implementação em uma arquitetura específica.

2.2.1 Grafo de Fluxo de Dados e Operadores

Nesta seção são abordados os grafos de fluxo de dados com mapeamento estático da aplicação. O fluxo de controle é implementado juntamente com o fluxo de dados. Os operadores considerados são assíncronos, ou seja, são disparados, ou realizam uma computação, quando os dados estão disponíveis em suas entradas, e após um atraso especificado, o resultado é propa-

gado para as saídas. Existem operadores que implementam a função de controle, e que eventualmente, consomem os dados e não geram saídas. A sincronização dos operadores podem ser implementada por protocolos de *Ready/Ack*, como será apresentado na seção 2.3.5, e ilustrado pela Figura 8. Filas podem ser usadas nas entradas e saídas para balanceamento dos caminhos de dados.

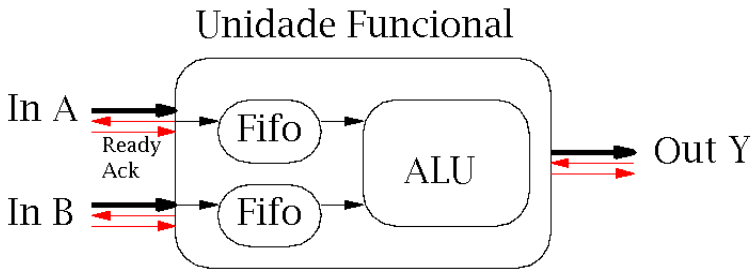


Figura 8: Uma unidade funcional com 2 filas na entrada e protocolo assíncrono com *Ready/Ack*

Os operadores podem ser classificados como operadores lógicos/aritméticos e operadores de controle. Os operadores lógico/aritméticos implementam operações padrões como Soma, Subtração, *And*, *Or*, Comparações, etc. Em geral são unários ou binários, mas a priori, não existem restrições sobre o número de entradas e/ou saídas.

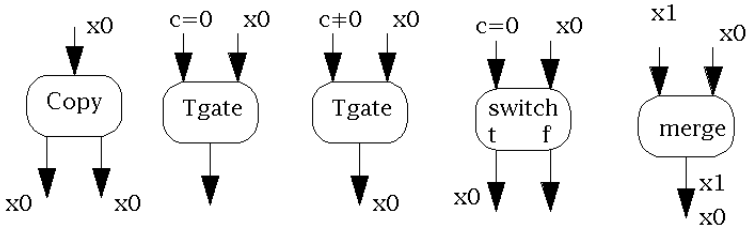


Figura 9: Operadores de controle

A Figura 10 ilustra os principais operadores de controle. O operador *Copy*, duplica o fluxo, quando um dado chega na entrada é propagado para ambas as saídas. Nos exemplos a seguir, é suposto que os dados sejam números inteiros, que são tipicamente usados na manipulação e representação de sinais digitais discretos. Vamos supor também que uma condição é um número inteiro e é considerada falsa se for igual a zero, caso contrário é considerada verdadeira.

O operador condicional *Tgate*, possui duas entradas: dado e condição. O dado é propagado

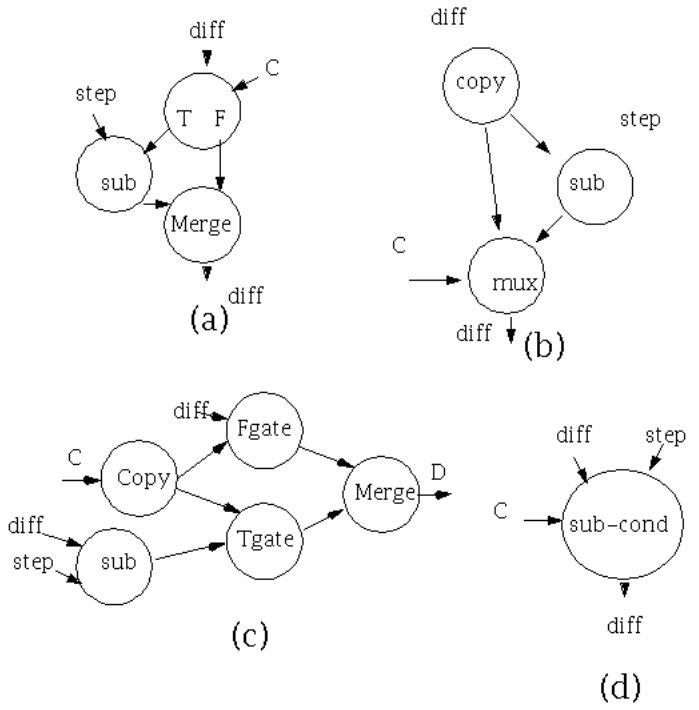


Figura 10: Quatro implementações de um comando condicional if (C) diff -= step: (a) Baseado em Merge; (b) Baseado em Mux; (c) Baseado em Fgate e Tgate; (d) Integrado

para a saída se a condição for verdadeira, caso contrário é absorvido pelo operador. O operador *Fgate* é semelhante, mas propaga o dado se a condição for falsa. O operador *switch* possui duas entradas e duas saídas. O dado é propagado para a saída *T* se a condição for verdadeira, e para a saída *F* se a condição for falsa. O operador *Mux*, possui três entradas e uma saída. A saída receberá o dado da entrada *T* se a condição for verdadeira, caso contrário propaga o dado da entrada *F*. O operador *Merge* possui duas entradas e propaga os dados para a saída. Se os dados chegarem ao mesmo tempo nas duas entradas, primeiro será propagado o dado de uma entrada e depois da outra, de maneira a evitar *starvation*. Os operadores *Tgate*, *Fgate*, *Merge*, *Switch*, *Mux* e *Copy* podem ser usados para implementação de comandos condicionais como ilustra a Figura 10.

Os operadores *Se-Par* e *Par-se*, que fazem a transformação Série-Paralelo e Paralelo-Série, foram propostos por Cardoso (2005) como uma solução para a exploração de paralelismo em laços, uso de técnicas de *self-loop pipelining*, gerenciamento de memória. Este último ponto é importante, pois a manipulação de dados em uma seqüência contínua facilita o controle do fluxo durante o acesso a memória, o que normalmente é um dos maiores gargalos na exploração

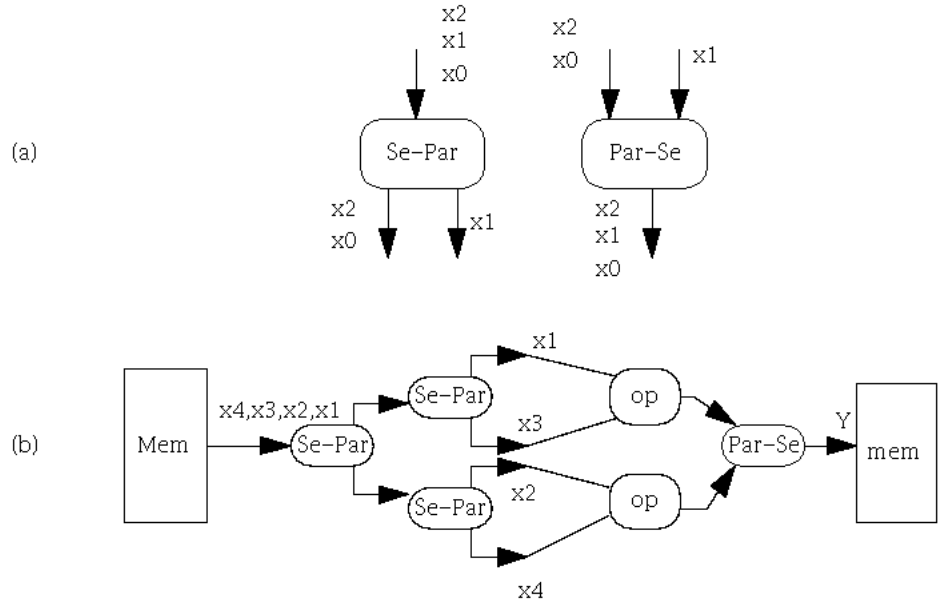


Figura 11: Operadores série-paralelo: (a) Operadores; (b) Exemplo de controle de fluxo

do paralelismo. O operador Se-Par funciona como um alternador, onde a entrada é propagada para a saída 0, depois para a saída 1, depois novamente para 0, para 1, etc. Já o operador Par-Se serializa os dados das entradas 0 e 1, propagando entrada 0, entrada 1, entrada 0, entrada 1, etc. A Figura 11 ilustra como os operadores Se-Par e Par-Se podem ser usados para gerenciar as entradas e saídas como um fluxo contínuo. A Figura 11a mostra os operadores, e a Figura 11b mostra um fluxo contínuo saindo de um módulo de memória, passando por uma estrutura de cálculo com 4 elementos de cada vez, e agrupando em outro módulo de memória. Por exemplo, a operação $y[2*j] = x[i]$ op $x[i+2]$ e $y[2*j+1] = x[i+1]$ op $x[i+3]$ poderia ser implementada com o grafo da Figura 11b.

2.2.2 Filtro de Resposta a Impulsos Finitos- FIR

Nesta seção iremos mostrar como a técnica de expansão ou linearização de laços (*loop unrolling*) pode ser usada para obter um grafo do fluxos de dados para implementações de algoritmos de filtros digitais. O objetivo desta seção é ilustrar como o código pode ser mapeado nos operadores de fluxo de dados, explorando o paralelismo intrínseco da aplicação e fazendo uso de controle distribuído, além de mostrar um exemplo da especificação em XML que é utilizada na versão atual da ferramenta EDA. Considere o filtro FIR (*Finite Impulse Response*)

que tem muitas aplicações na área de processamento de sinais e imagens, sendo amplamente estudado e utilizado.

Considere o código na linguagem C mostrado abaixo:

```
/*  TEXAS INSTRUMENTS, INC.
 *  USAGE This routine is C Callable and can be called as:
 *  void fir4(short *x, short *h, short *y, int N, int M)
 *  x = input array
 *  h = coefficient array
 *  y = output array
 *  N = number of coefficients (MULTIPLE of 4 >= 4)
 *  M = number of output samples (EVEN >= 2)
 */
void fir4(short x[], short h[], short y[], int N, int M)
{
   int i, j, sum;

   for (j = 0; j < M; j++) {
      sum = 0;
      for (i = 0; i < N; i++)
          sum += x[i + j] * h[i];
      y[j] = sum >> 15;
   }
}
```

O laço interno implementa um somatório onde h são os coeficientes do filtro aplicados a seqüência de dados discretos do sinal x. Temos o sinal de saída do filtro representado pelo vetor $y_j = \sum_i x_{i+j} * h_i$. Vamos supor um caso mais simples onde o laço interno trata apenas 4 coeficientes, ou seja um filtro de ordem 4. Se usarmos a técnica de linearização do laço interno teremos

```
void fir4(short x[], short h[], short y[], int N, int M)
{
   int i, j, sum;

   for (j = 0; j < M; j++) {
      sum = 0;
      sum = x[j] * h[0]+ x[j+1] * h[1] + x[j+2] * h[2]+ x[j+3] * h[3];
      y[j] = sum >> 15;
   }
}
```

ou na forma de equações:

$$
\begin{aligned}
y_0 &= x_0 * h_0 + x_1 * h_1 + x_2 * h_2 + x_3 * h_3 \\
y_1 &= x_1 * h_0 + x_2 * h_1 + x_3 * h_2 + x_4 * h_3
\end{aligned}
$$

$$
\begin{aligned}
y_2 &= x_2 * h_0 + x_3 * h_1 + x_4 * h_2 + x_5 * h_3 \\
y_3 &= x_3 * h_0 + x_4 * h_1 + x_5 * h_2 + x_6 * h_3 \\
&\ldots.
\end{aligned}
$$

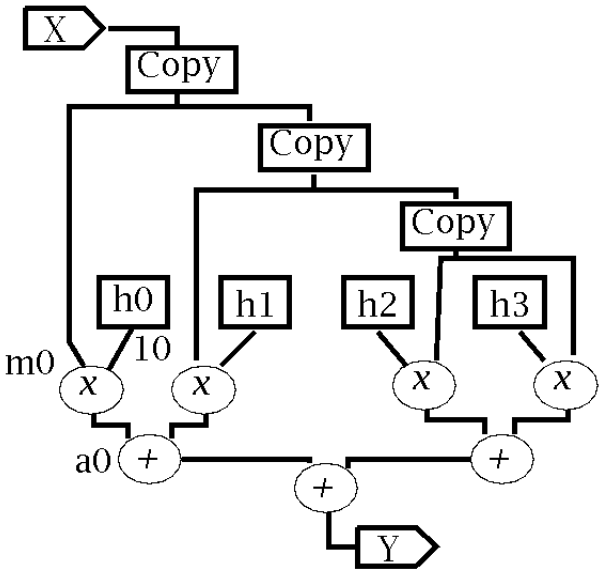


Figura 12: Representação do fluxo de dados de um filtro de resposta a impulsos (FIR) de ordem 4

Podemos notar que apenas x_0, x_1 e x_2 são os termos iniciais a circular pelo grafo. Posteriormente, os termos x_3 em diante irão passar por todos os coeficientes (seguindo a diagonal). Tirando proveito desta propriedade regular, podemos propor o grafo de fluxo de dados da Figura 12. O dados do vetor *X* entram em uma seqüência de nodos do tipo *Copy* que serve para duplicar o fluxo. Neste exemplo, o laço externo está implicitamente controlado pelo vetor de entrada que tem a função de injetar os valores continuamente.

Para obter portabilidade e conectividade com outras ferramentas, um formato XML para a especificação do fluxo de dados foi proposto no EDA (FERREIRA; CARDOSO; NETO, 2004). O formato é incremental e permite extensões futuras. A especificação permite descrever características adicionais como as constantes de entrada. É possível também definir o atraso, tamanho da fila de entrada ou saída, ou outros aspectos que podem ser incorporados ao ambiente EDA. Um trecho da especificação XML é apresentado a seguir. Dois operadores são instanciados *m*0 e *a*0. O operador *m*0 tem uma entrada constante com o valor 10, que implementa a cons-

tante *h*0 do filtro. Além dos dois operadores, a descrição inclui a especificação dos sinais de interligações, como ilustra o sinal *wire*3.

```
 ....
<COMPONENT op="IMUL" name="m0">
   <PORT name="A" value="10"/>
</COMPONENT>
<COMPONENT op="IADD" name="a0" />
<SIGNAL name="wire3">
   <SOURCE name="m0" port="Y"/>
   <SINK name="a0" port="A"/>
</SIGNAL>
...
```

O exemplo do FIR é bem simples. Porém não é necessário linearizar todo o laço, se a ordem do filtro for alta, implicará em um número alto de multiplicadores. Os multiplicadores têm implementação com custo maior que outros operadores, tanto em área como atraso e potência. O compartilhamento e implementação em pipeline desses operadores é possível sem que haja prejuízo no desempenho. Os operadores *Se-Par* e *Par-Se* podem ser usados para seqüencializar e compartilhar multiplicadores. Operadores acumuladores também podem ser muito úteis nestes casos. Como o escopo da dissertação é voltado para o problema do posicionamento, não foi aprofundado o estudo do espaço de projeto dos operadores. Porém, este é um ponto importante a ser explorado para as arquiteturas de grão grosso. Uma revisão e classificação detalhada sobre os operadores de fluxo de dados pode ser encontrada em (VEEN, 1986). Na próxima seção, será ilustrada uma situação comum em códigos de processamento de sinais, onde os operadores *Se-Par* e *Par-Se* simplificam o fluxo de controle e gerência de memória.

2.2.3 FDCT

Considere um subconjunto do código de uma implementação Transformada Rápida de Cosenos, ou FDCT (Fast Discrete Cosine Transform). O apêndice A.3 mostra o trecho completo para o laço abaixo. No trecho apenas a varredura vertical de um bloco 8x8 é ilustrada. No código completo, duas varreduras sobre cada bloco 8x8 da matriz de pixels são realizadas. A varredura vertical e horizontal tem características muito semelhantes no corpo do laço. As diferenças ficam restritas a entrada de dados, sendo que na varredura vertical são injetados 8 pontos (um de cada linha), enquanto que na varredura horizontal é injetado um dado para cada coluna. Uma pequena diferença no cálculo na saída do laço também existe, na operação final de

deslocamento realizada antes de gravar os dados calculados pelo corpo do laço.

```
for (j = 0; j < N; j++) {
   /* ------------------------------------------------ /
   /  Load the spatial-domain samples.                 /
   / ------------------------------------------------ */
   f0 = dct_io_ptr[ 0+i_1];
   f1 = dct_io_ptr[ 8+i_1];
   f2 = dct_io_ptr[16+i_1];
   f3 = dct_io_ptr[24+i_1];
   f4 = dct_io_ptr[32+i_1];
   f5 = dct_io_ptr[40+i_1];
   f6 = dct_io_ptr[48+i_1];
   f7 = dct_io_ptr[56+i_1];

   /* ------------------------------------------------ /
   /  Stage 1:  Separate into even and odd halves.     /
   / ------------------------------------------------ */
   g0 = f0 + f7;               h2 = f0 - f7;
   g1 = f1 + f6;               h3 = f1 - f6;
   h1 = f2 + f5;               g3 = f2 - f5;
   h0 = f3 + f4;               g2 = f3 - f4;
....
   F1 = c7 * Q1 + c1 * S1;     F7 = c7 * S1 - c1 * Q1;
   F5 = c3 * Q0 + c5 * S0;     F3 = c3 * S0 - c5 * Q0;

   /* ------------------------------------------------ /
   /  Store the frequency domain results.              /
   / ------------------------------------------------ */
   dct_io_tmp[ 0+i_1] = F0;
   dct_io_tmp[ 8+i_1] = F1 >> 13;
   dct_io_tmp[16+i_1] = F2 >> 13;
   dct_io_tmp[24+i_1] = F3 >> 13;
   dct_io_tmp[32+i_1] = F4;
   dct_io_tmp[40+i_1] = F5 >> 13;
   dct_io_tmp[48+i_1] = F6 >> 13;
   dct_io_tmp[56+i_1] = F7 >> 13;

   //dct_io_ptr++;
   i_1++;
}
```

Neste exemplo, oito elementos do vetor `dct_io_ptr` são injetados em paralelo no fluxo de dados e depois armazenados no vetor temporário `dct_io_tmp`. Pode-se notar que existem muitas operações que podem ser realizadas em paralelo, como o primeiro estágio de somas e subtração. Entretanto o fluxo poderia ficar limitado a um módulo sequencial de memória, sendo necessário um controle com muitos nodos condicionais ou nodos de *load*. Alguns operadores podem ser usados para facilitar esta tarefa. Suponha que um gerador de endereços seja aplicado

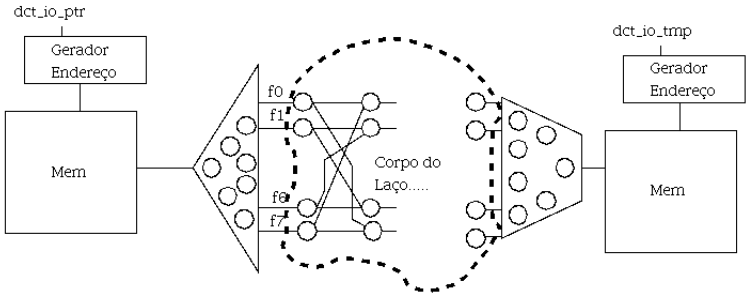


Figura 13: Representação simplificada do fluxo de dados da varredura vertical de uma FDCT 8x8 com operadores série-paralelo

ao vetor, resultando em um fluxo dos elementos de 8 em 8. No caso específico do DCT, seria gerado um fluxo dos elementos do vetor `dct_io_ptr`. Uma solução seria fazer uso dos operadores definidos como série paralelo e paralelo série. O fluxo proveniente do vetor `dct_io_ptr` pode ser aplicado a uma estrutura em árvore de operadores *Se-Par* que irá gerar os sinais f_0 a f_7 nas suas respectivas folhas. O processo complementar, ou seja, uma árvore invertida, usando operadores Par-Se pode coletar os sinais F_0 a F_7 e armazenar no vetor `dct_io_tmp`. A Figura 13 ilustra como os operadores e o gerador de endereços podem simplificar o controle.

As duas varreduras têm os mesmos operadores internos, com poucas exceções, como mostra o código da varredura horizontal apresentado a seguir. Por terem operadores internos coincidentes, uma estrutura compartilhada poderia ser usada para o caso das varreduras verticais e horizontais do FDCT. A Figura 14 ilustra esta abordagem.

```
// ------------------------------------------------- /
//  Load the spatial-domain samples.                 /
// ------------------------------------------------- /
f0 = dct_io_tmp[0+i_1];
f1 = dct_io_tmp[1+i_1];
f2 = dct_io_tmp[2+i_1];
f3 = dct_io_tmp[3+i_1];
f4 = dct_io_tmp[4+i_1];
f5 = dct_io_tmp[5+i_1];
f6 = dct_io_tmp[6+i_1];
f7 = dct_io_tmp[7+i_1];

// ------------------------------------------------- /
//  Stage 1:  Separate into even and odd halves.     /
// ------------------------------------------------- /
g0 = f0 + f7;              h2 = f0 - f7;
g1 = f1 + f6;              h3 = f1 - f6;
h1 = f2 + f5;              g3 = f2 - f5;
h0 = f3 + f4;              g2 = f3 - f4;
```

```
  // ------------------------------------------------- /
  //  Stage 2                                          /
  // ------------------------------------------------- /
  p0 = g0 + h0;               r0 = g0 - h0;
  p1 = g1 + h1;               r1 = g1 - h1;
  q1 = g2;                    s1 = h2;

.....

  F0r = (F0 + 0x0006) >>  3;
  F1r = (F1 + 0x7FFF) >> 16;
  F2r = (F2 + 0x7FFF) >> 16;
  F3r = (F3 + 0x7FFF) >> 16;
  F4r = (F4 + 0x0004) >>  3;
  F5r = (F5 + 0x7FFF) >> 16;
  F6r = (F6 + 0x7FFF) >> 16;
  F7r = (F7 + 0x7FFF) >> 16;

  // ------------------------------------------------- /
  //  Store the results                                /
  // ------------------------------------------------- /
  dct_o[0+i_1] = F0r;
  dct_o[1+i_1] = F1r;
  dct_o[2+i_1] = F2r;
  dct_o[3+i_1] = F3r;
  dct_o[4+i_1] = F4r;
  dct_o[5+i_1] = F5r;
  dct_o[6+i_1] = F6r;
  dct_o[7+i_1] = F7r;
```

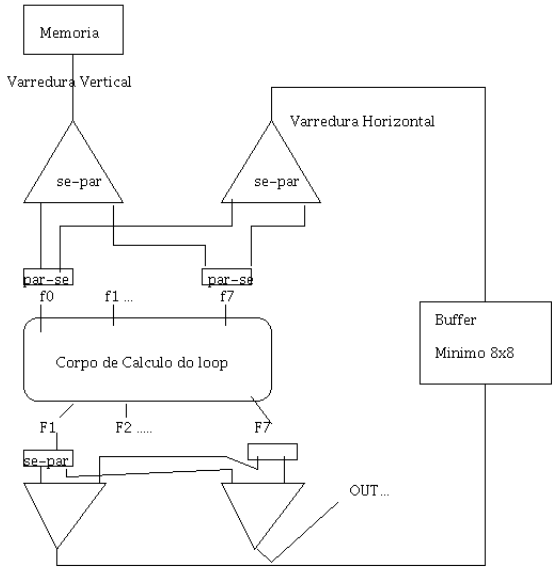


Figura 14: Representação simplificada do FDCT compartilhando a parte de cálculo para as varreduras verticais e horizontais com operadores série-paralelo

2.2.4 Simulação

O simulador HADES foi escolhido pela sua simplicidade e portabilidade. O simulador foi desenvolvido para ser uma ferramenta de ensino de sistemas lógicos digitais em vários níveis desde de transistores e portas lógicas a sistemas embarcados. Possui uma biblioteca de mais de 500 componentes. A orientação a objetos facilita muito a adição de novos componentes. Todos os sinais também fazem uso da orientação a objetos, tendo sinais para valores analógicos, bit, palavras de N bits, *strings* e até imagens. Será mostrado que os *strings* são muito úteis na simulação de hardware reconfigurável.

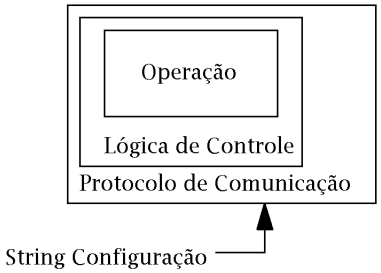


Figura 15: Modelagem das classes de um unidade funcional

A Figura 15 mostra o modelo usado para a implementação de um operador. Foi criada uma classe FU que implementa uma unidade funcional. A classe possui três objetos: protocolo, controle e operação. Estes objetos tem seu código carregado dinamicamente durante a execução da simulação, o que facilita o desenvolvimento e modelagem de novos operadores. A classe protocolo define o protocolo assíncrono de comunicação (ex: fase do *ready/ack*). A classe controle define a lógica de disparo do operador. Por exemplo, o operador *Merge* repassa uma das entradas para a saída assim que chega algo em qualquer uma das duas entradas. Um operador ALU binário, irá repassar um valor somente se ambas as entradas tiverem recebido dados. A operação que é executada é definida em outra classe, por exemplo, uma ALU pode carregar um ADD ou um IF_LT. O código Java completo para cada operação ocupa poucas linhas como ilustra a operação de ADD no código abaixo. O sinal *string* é usado para passar dinamicamente o módulo que será carregado para a FU.

```
// arquivo: OP_ISUB.java
package eda.fu;
import hades.models.StdLogicVector;
```

```
/**
ISUB Operator: A - B
*/
public class  OP_ISUB extends AluOperation {
    public StdLogicVector compute(StdLogicVector A,StdLogicVector B) {
        return A.sub(B);
    }
    public double getDelay() {
        return Delays.ISUB;
    }
}; // ALU class
```

Outra facilidade do ambiente é a interface gráfica para depuração do código para definição de novos operadores, técnicas de controle e protocolos.

Uma vez definidos os operadores e acrescentados à biblioteca do Hades, o simulador está apto para explorar os grafos de aplicações. O arquivo de entrada com a especificação do grafo de fluxo de dados é transformado por um script em um código Java que gera uma descrição estrutural no formato HDS (formato de entrada do simulador HADES). Um script de simulação também é gerado. A simulação pode fazer uso da interface gráfica ou rodar em batch. Durante a simulação do grafo, vários parametros podem ser explorados e avaliados como a vazão, a latência, o atraso dos operadores, o tamanho médio das filas, a energia consumida em médio ou instantânea, o grau de uso do paralelismo, etc.

Considere o gráfico da Figura 16 extraído de (FERREIRA et al., 2005) que foi gerado para o mostrar o número de unidades que estão em atividades para a aplicação FDCT com cálculo de varredura compartilhado. No início, o primeiro bloco 8x8 de dados passa por uma varredura vertical. Depois, em paralelo, o primeiro bloco passa pela varredura horizontal e o segundo bloco passa pela varredura vertical. Pode-se observar que teremos em torno de 40-50 unidades trabalhando em paralelo ao longo do tempo, sendo 10-15 operadores aritméticos e 20-25 operadores de controle. O número total de unidades para este exemplo é de 139. Ou seja, em torno de 35% das unidades trabalham em paralelo.

O diagrama na Figura 17 mostra o corpo de cálculo sendo compartilhado pelas varreduras verticais e horizontais de dois blocos distintos em pipeline. Enquanto o primeiro bloco, que já realizou a varredura vertical, realiza a varredura horizontal, o segundo bloco realiza a varredura vertical. Cada varredura faz uso de quatorze multiplicadores que neste caso são compartilhados, bem como os outros operadores lógicos, aritméticos e de controle (*copa* no caso).

Para ter uma avaliação da aplicação em uma arquitetura reconfigurável, é necessário definir

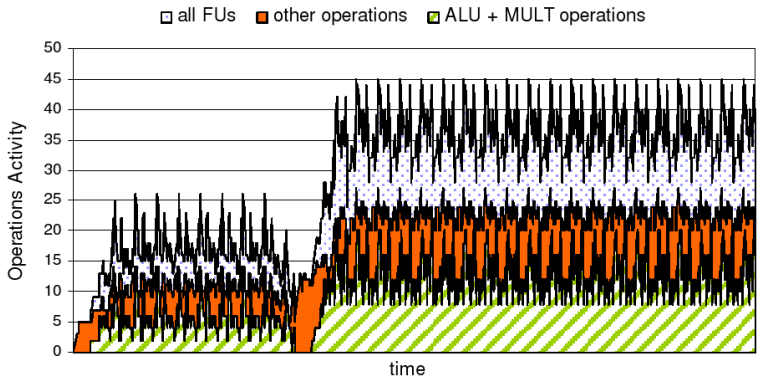


Figura 16: Gráfico da atividade (número de unidades ativas em paralelo) para o FDCT

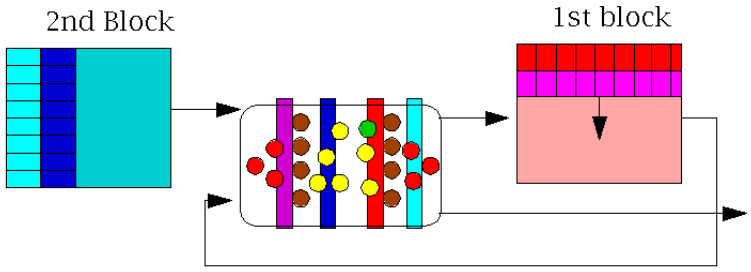


Figura 17: Diagrama de blocos do FDCT com compartilhamento

a arquitetura e realizar o mapeamento. Na próxima seção, será mostrada a flexibilidade do ambiente EDA para gerenciar arquiteturas, mapeamento e simulação na exploração do espaço de projeto.

2.3 Mapeamento

O mapeamento de uma aplicação em um arquitetura reconfigurável em geral é dividido em duas fases: posicionamento e roteamento. Para possibilitar a exploração de diversas arquiteturas, o ambiente EDA tem como entrada uma especificação XML da topologia e outros parâmetros. Além disso, uma abordagem aberta (FERREIRA et al., 2005) foi implementada, possibilitando a inclusão de algoritmos de posicionamento e roteamento. A solução adotada envolve três classes principais. A classe *arranjo* que define os algoritmos de posicionamento e roteamento. A classe *PE* que define a topologia ou o padrão de ligação entre os *PEs*, que irá se comportar como uma moldura para abrigar uma ou mais unidades funcionais. Finalmente a

classe *Borda* que define a interface entre duas células ou *PEs*.

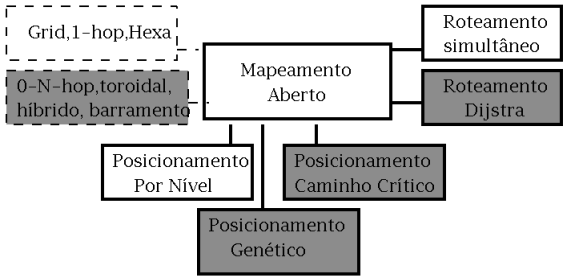


Figura 18: Diagrama de blocos do mapeamento no ambiente EDA

A Figura 18 mostra um diagrama de blocos da estrutura de mapeamento do EDA. A versão inicial não possuía mapeamento automatizado (FERREIRA; CARDOSO; NETO, 2004). A segunda versão (FERREIRA et al., 2005), incluiu uma avaliação das topologias hexagonais, grid e 1-hop, um algoritmo de posicionamento baseados em níveis e um algoritmo de roteamento guloso onde todos os caminhos eram construídos ao mesmo tempo. A versão apresentada nesta dissertação acrescentou mais dois algoritmos de posicionamento, avaliou um conjunto mais amplo de topologias, e acrescentou também um algoritmo de roteamento local (GARCIA; FERREIRA, 2006) como adaptação do algoritmo de Dijkstra.

2.3.1 Bordas e Padrão de Conexão Local

A Figura 19 mostra alguns exemplos de *PEs* com variação do padrão de conexão entre as bordas, isto é, o número de entradas e saídas. O primeiro exemplo mostra um *PE* com uma entrada e uma saída. O segundo exemplo mostra um *PE* com duas entradas e duas saídas por borda. Finalmente, o terceiro exemplo ilustra um caso como dois sinais bi-direcionais por borda, podendo se comportar como entrada ou saída.

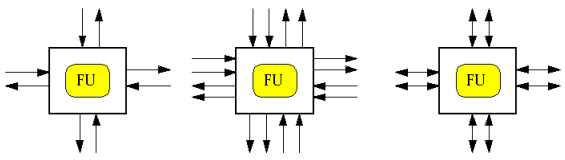


Figura 19: Diferentes configurações de Bordas

Além dos número de entradas e saídas é necessário definir o número de bordas do *PE*. Nos exemplos da Figura 19, todos os três casos tinham apenas 4 bordas. A Figura 20 mostra a comparação da capacidade de roteamento para dois *PE* em função do seu número de bordas, 4 e 6 para os *PEs* grid e hexagonal, respectivamente. Quando limitado a uma entrada e saída, uma célula de grid com uma unidade funcional não pode realizar roteamento, se internamente for implementado um operador com duas entradas e uma saída (Figura 20a). No caso de duas entradas/saídas por borda, além de conectar a unidade funcional, o *PE* pode realizar até dois roteamentos. Já uma célula hexagonal, pode realizar 1 ou 4 roteamentos se tem uma ou duas entradas e saídas por borda, respectivamente (Figura 20b)

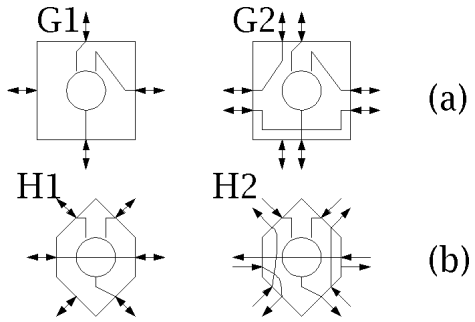


Figura 20: Comparação entre células com uma ou duas entradas/saídas: (a) Grid (b) Hexagonal

2.3.2 Arquitetura

A arquitetura envolve a definição das bordas e suas conexões, mas também envolve outras definições. O arquivo XML que define a arquitetura inclui também a topologia que pode ser regular como um grid, ou aleatória. A primeira implementação do EDA (FERREIRA et al., 2005) suportava apenas a topologia hexagonal para simulação. Porém, para o mapeamento, as topologias grid, 1-hop e hexagonal foram avaliadas. Podendo ser estendidas para tratar as outras possibilidades que foram adicionadas como contribuição deste trabalho de dissertação. Além de definir a topologia, as características como o tipo de *FU* que cada *PE* irá implementar, e outros parâmetros podem ser acrescentadas. A Figura 21 ilustra uma arquitetura heterogênea com topologia hexagonal com 16 elementos dispostos em um arranjo 4x4. As *FU* das duas primeiras linhas implementam *ALU*, assim como a terceira linha, exceto a primeira *FU* que implementa uma unidade de entrada e saída. A quarta linha implementa duas unidades de memória e duas unidades de entrada e saída.

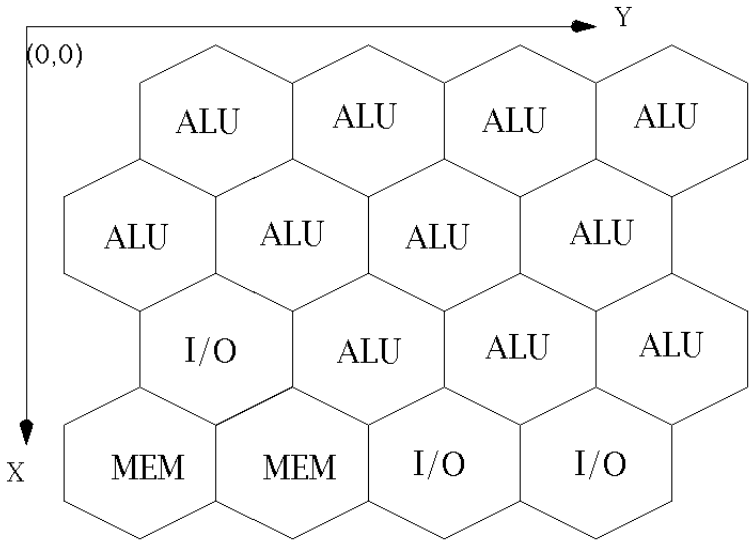


Figura 21: Exemplo de um arranjo hexagonal 4x4

O trecho XML abaixo ilustra parte da definição da arquitetura mostrada na Figura 21.

```
<CELL name="ALU" symbol="ALU"/>
....
<LAYOUT length="4" width="4"
                 symbol="ALU"/>
....
<!-- override special cells -->
<LAYOUT x="2" y="0" symbol="I/O"/>
...
<!-- line 3 : 2 MEM and 2 I/O -->
<LAYOUT x="3" y="0" symbol="MEM"/>
<LAYOUT x="3" y="1" symbol="MEM"/>
<LAYOUT x="3" y="2" symbol="I/O"/>
<LAYOUT x="3" y="3" symbol="I/O"/>
```

2.3.3 Posicionamento

A primeira versão do algoritmo de posicionamento (TOLEDO; FERREIRA, 2004; FERREIRA et al., 2005) foi baseada em (KOREN et al., 1988) que faz uma busca em profundidade no grafo de fluxos de dados e posiciona os vértices por níveis. Posteriormente um ajuste baseado no centro de massa é aplicado. A adaptação disponível na primeira versão do EDA (TOLEDO; FERREIRA, 2004), fez a inclusão de *PE* desocupados em cada linha durante a otimização e

troca de linha durante a fase de ajuste. Porém os resultados geravam posicionamentos alongados na forma de diamante. No caso de grafo com ciclos, como o FDCT compartilhado (Figura 22), alguns vértices eram posicionados fisicamente muitos distantes.

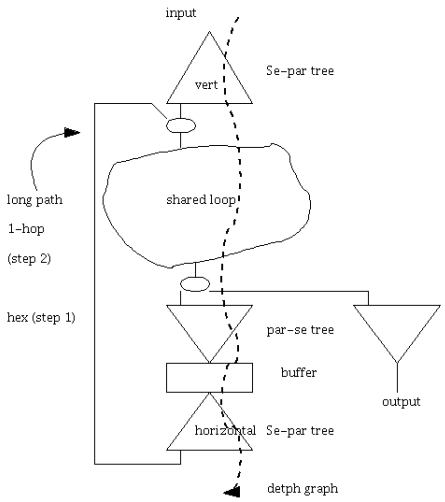


Figura 22: FDCT compartilhado com posicionamento alongado

A baixa densidade de ocupação dos arranjos e a presenças de caminhos muito longos motivou o estudo, avaliação e inclusão de novos algoritmos de mapeamento que serão apresentados no capítulo 4.

2.3.4 Roteamento

A Figura 23 mostra um exemplo para um filtro de resposta a impulsos finitos de ordem 4. A qualidade do roteamento depende do posicionamento e da conectividade da arquitetura. No exemplo, como o grafo de fluxo de dados é pequeno e as ligações têm muita localidade, uma topologia hexagonal gera bons resultados.

A primeira versão do algoritmo de roteamento (TOLEDO; FERREIRA, 2004; FERREIRA et al., 2005) também foi baseada em (KOREN et al., 1988) que sugere que todos as conexões sejam construídas simultaneamente. Uma lista com todos os arcos do grafo de fluxo de dados é construída. Para cada elemento da lista, uma conexão começa a ser construída a partir do *PE* de origem , onde está posicionado o vértice de origem. A cada passo, as interconexões avançam um *PE*. Quando uma conexão atinge o *PE* de destino, ela é concluída e retirada da lista. O algoritmo utiliza o enfoque guloso e é rápido, porém para alguns casos, em função do posicio-

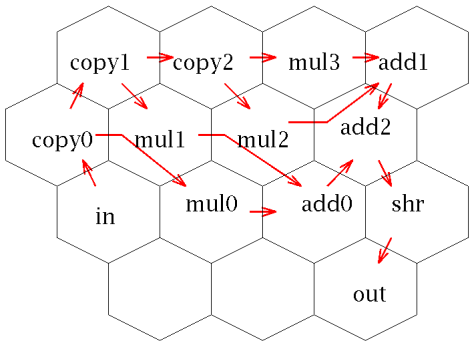


Figura 23: Exemplo de posicionamento e roteamento do filtro de ordem 4 em um arranjo hexagonal

namento, nenhuma solução é encontrada, ou seja, algumas conexões não são completadas.

A versão atual do roteamento (SILVA et al., 2006; GARCIA; FERREIRA, 2006) inclui uma variação do algoritmo de Dijkstra para topologias regulares com grid, N-hop e 0-N-hop. A cada passo, vários *PE* são incluídos na lista de vizinhos, uma vez que muitas conexões tem o mesmo custo.

2.3.5 Simulação

Para realizar a simulação das arquiteturas, um componente parametrizado foi criado. Este componente funciona como um moldura e é reconfigurável para os seguintes parâmetros:

- Número de bordas;
- Número e tipo das conexões em cada borda
- Capacidade de roteamento;
- Atraso e potência dissipada;
- Monitoramento;
- Número e tipo de conexões com as unidades funcionais.

A Figura 24 ilustra um diagrama do componente que serve como moldura para a implementação de um *PE*. Neste exemplo, na Figura 24a há uma moldura com 4 bordas, onde cada

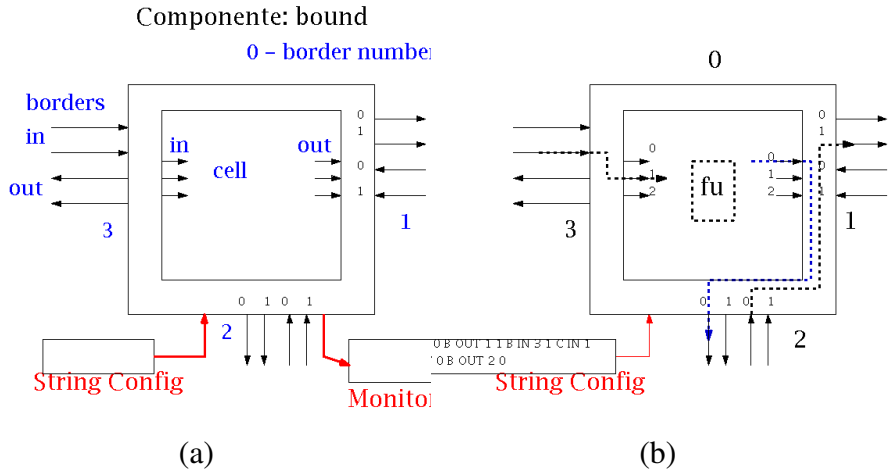


Figura 24: Moldura de um *PE*: (a) Estrutura de Conexões Fixas (b) Exemplo de Configuração

borda tem duas conexões de entrada e duas conexões de saída. Foram definidas também três conexões de entrada internas e três conexões de saída. O sinal `StringConfig` faz a configuração do roteamento e o sinal `Monitor` serve para capturar os eventos que ocorrem no *PE*. Ambos utilizam *strings*, o que facilita a modelagem e implementação do código para exploração das arquiteturas. A Figura 24a apresenta três configurações de conexão. Um sinal que entrada 1 pela borda 3 e se conecta a entrada 1 da *FU*. Um sinal da saída 0 da *FU* que se conecta a saída 0 da borda 2, e um sinal que passa pelo *PE* que chega pela entrada 0 da borda 2 e é roteado para saída 1 da borda 1.

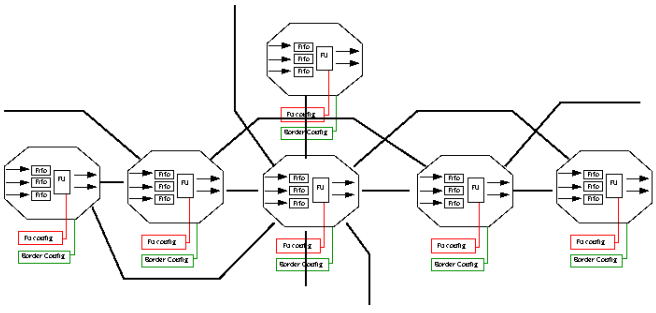
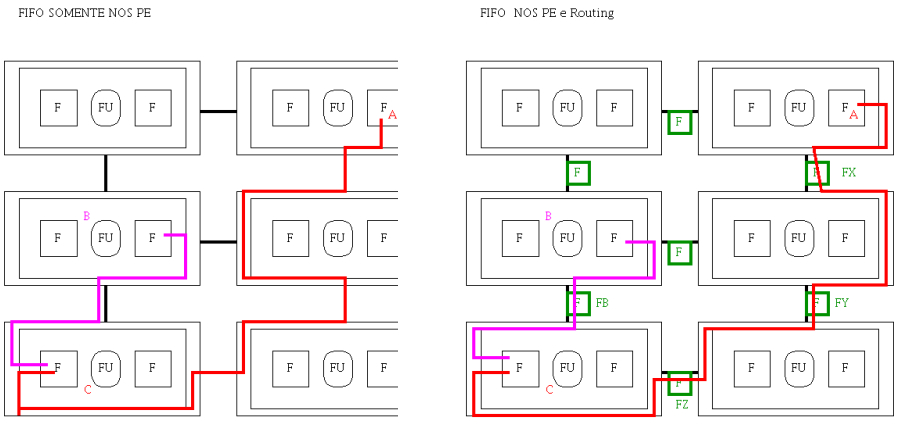


Figura 25: Exemplo de uma arquitetura 1-hop com uma *FU* por *PE* e três filas nas entradas da *FU*

A Figura 25 ilustra um exemplo de um arranjo com conexões no padrão 1-hop. O número e tipo de conexões não está destacada na figura, apenas o padrão de conexão. Além disso, cada *PE* tem uma estrutura interna com uma unidade funcional e três filas de entrada.



(a) (b)

Figura 26: Arranjo em grade 3x2 com: (a) Filas no interior do *PE* (b) Filas dentro e fora dos *PEs*.

Outra característica importante em arranjos que implementam modelos assíncronos é o número, posição e capacidade das filas. A Figura 26 mostra dois exemplos de um arranjo 3x2 na topologia grid, onde existem filas somente no interior dos *PE* (Figura 26a) ou além de filas no interior de cada *PE*, existe uma fila em cada conexão local entre os vizinhos (Figura 26b). Apesar de aumentar a latência, as filas nas conexões externas, são escaláveis, aumentam a vazão e servem para balancear os caminhos. A Figura 26 ilustra também uma configuração de roteamento. O arranjo é reconfigurável tanto para o roteamento, quanto para a funcionalidade da *FU* e/ou filas que foram definidas.

A integração com o simulador HADES foi direta pela simplicidade da abordagem e pelos recursos disponíveis no simulador. Durante a simulação tanto as conexões quanto as operações podem ser modificadas, como também o próprio arranjo. Detalhes de sinais não são abordados aqui, mas os resultados apresentados nesta dissertação levam em consideração uma protocolo assíncrono de 4 fases com uma palavra de 32 bits para transmissão dos dados, um sinal binário para *Ready*, um sinal binário para *Ack*.

A Figura 27 mostra um exemplo de um detalhe de um arranjo hexagonal heterogêneo com o protocolo assíncrono.

Finalmente, para mostrar a interface gráfica que pode ser usada na depuração e no ensino de arranjos reconfiguráveis, a Figura 28 foi extraído do simulador HADES e mostra um arranjo hexagonal onde cada *PE* tem uma unidade funcional e três filas, além de destacar as conexões de roteamento que estão configuradas.

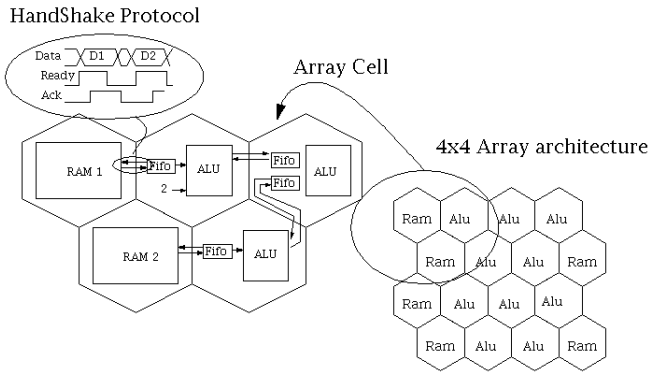


Figura 27: Arranjo hexagonal 4x4 heterogêneo com protocolo assíncrono

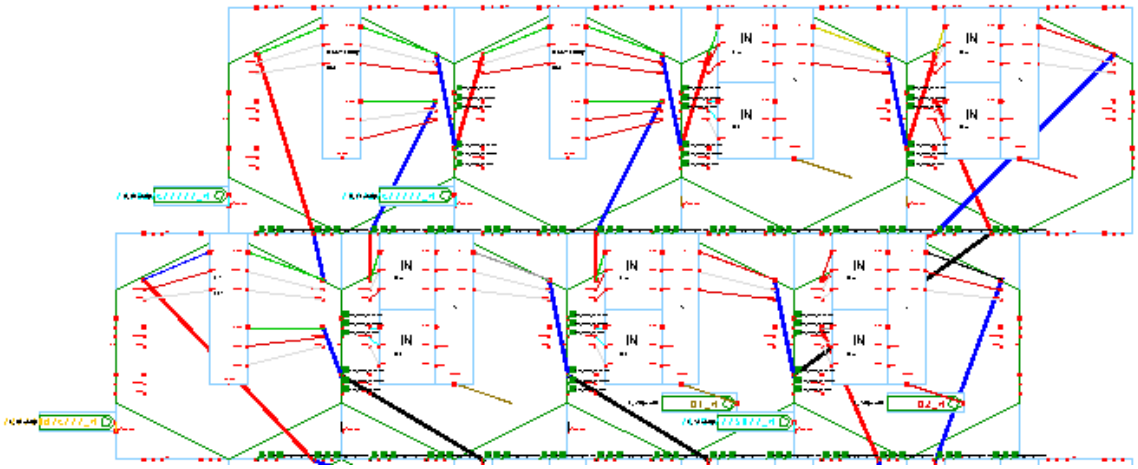


Figura 28: Imagem extraída de um simulação do HADES para um arranjo hexagonal

2.4 Conclusão

Este capítulo apresentou uma visão geral do ambiente EDA para exploração do espaço de projeto de arquiteturas baseadas em arranjos bi-dimensionais para implementação de grafos de fluxo de dados. O ambiente é aberto, usa orientação a objetos e formatos intermediários que facilitam a integração com outras ferramentas e inclusão de novos recursos. A principal contribuição dessa dissertação foi a inclusão de novos algoritmos de posicionamento e a exploração de topologias diversas, temas que serão abordados nos próximos capítulos.

3 TOPOLOGIAS DE ARRANJOS DE PROCESSADORES

3.1 Introdução

Neste capítulo serão apresentadas as principais topologias em arranjos bi-dimensionais, enfatizando as topologias em malha, que vêm sendo utilizadas e avaliadas para implementação de arquiteturas reconfiguráveis. Vale ressaltar que estas topologias vêm sendo estudadas a várias décadas em outros contextos de implementação, como nos *array processors* da década de 70-80 que eram máquinas padrão SIMD, assim como nos multiprocessadores ou máquinas padrão MIMD. Os FPGA's têm uma estrutura padrão em um plano bi-dimensional, em geral usam conexões locais, conexões em cascata para propagação em estruturas regulares, barramentos, etc. Recentemente, as NoC (Network on chip) também vêm explorando este espaço de solução, em geral com topologias em grid, porém os arranjos de NoC têm dimensões pequenas (16 a 32 elementos). Estas topologias têm como principal vantagem o fato de serem escaláveis graças ao padrão de conexão local.

Uma arquitetura reconfigurável pode ser definida como um conjunto de elementos de processamento (PE) inter-conectados por uma rede. No caso das arquiteturas de grão grosso, um PE pode também ser denominado por Unidade Funcional (FU) ou ALU, pois implementam um conjunto simples de operações com largura de N bits. Algumas arquiteturas de grão grosso já consideram uma capacidade para incorporar um poder de processamento com memória local e um pequeno conjunto de instruções em cada PE. Neste capítulo será usado o termo PE para denotar um elemento do arranjo. A rede de interconexões pode ser regular ou não, ter níveis hierárquicos, ser homogênea ou heterogênea, existindo portanto um grande espaço a ser explorado. Nesta dissertação serão abordadas as topologias em malha com algumas variações. No

final do capítulo será apresentado um modelo geral que pode ser usado para exploração de um conjunto bem mais amplo de topologias. Nas próximas seções serão apresentadas as topologias convencionais em malha, as topologias padrão N-Hop, as topologias 0_N-hop e variações.

3.2 Topologias Regulares em Malha

As topologias convencionais possuem um padrão local de conexão, onde os PE estão dispostos em um plano bi-dimensional como ilustra a Figura 29a-c que apresenta respectivamente as topologias de um arranjo 3x3 dispostas em grid, hexagonal e octal. A topologia em grid é a uma das mais simples, mas muito utilizada. As ligações com os quarto vizinhos podem ser bi-direcionais ou unidirecionais. A topologia hexagonal, que é geralmente apresentada na forma de favos de mel, como apresentado anteriormente, pode também ser disposta em um plano semelhante ao grid com algumas ligações na diagonal. No contexto de multiprocessadores, alguns autores argumentam que a topologia hexagonal possibilita características interessantes para tolerância a falhas (BOSSART, 1994) e capacidade de roteamento dinâmico, sem perder a simplicidade da implementação física. Um exemplo de arquitetura reconfigurável com topologia hexagonal é o Kress Array (HARTENSTEIN et al., 2000). A topologia octal já possui uma padrão maior de conectividade com os 8 vizinhos diretos, e um maior custo associado.

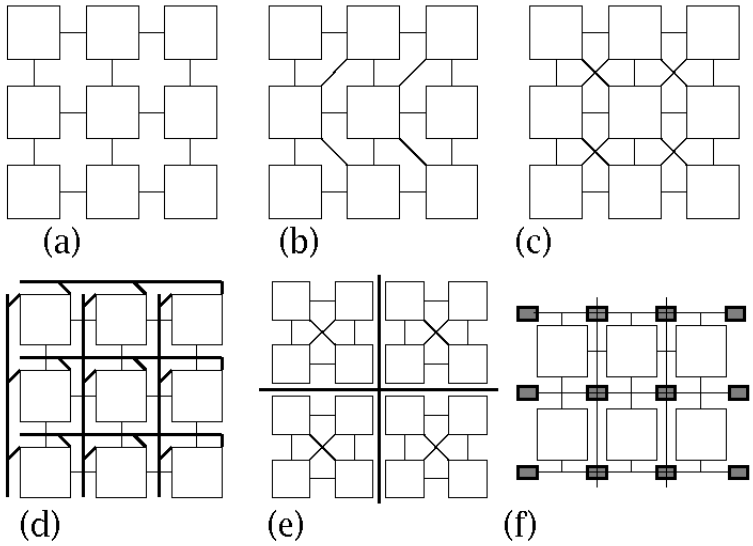


Figura 29: Padrões regulares em malha

Para melhorar o grau de conectividade sem aumentar muito os custos, algumas arquiteturas reconfiguráveis fazem uso de barramentos como ilustra a Figura 29d. O barramento pode ter um árbitro e uma política de escalonamento dinâmica, ou pode ser mais simples, e usar apenas escalonamento estático. O barramento pode possibilitar ligações diretas de PEs que estão fisicamente distantes nos arranjos. Entretanto, barramentos introduzem atrasos e têm consumo de energia maior que conexões ponto-a-ponto.

Outra opção são os agrupamentos ou clusters com níveis hierárquicos (SWANSON et al., 2003; BOSSUET; GOGNIAT; PHILIPPE, 2003; SINGH et al., 1998), para possibilitar ligações locais e globais. A Figura 29e ilustra um arranjo de 16 PEs com quatro agrupamentos de quatro elementos. Dentro de cada agrupamento, cada um dos PEs tem conexão direta com todos os outros PEs. Para conectar os agrupamentos, uma rede de conexão global com um barramento horizontal e outro vertical é utilizada.

Uma solução semelhante ao barramento porém com maior flexibilidade é o uso de elementos de chaveamento, como por exemplo uma rede *cross-bar* ilustrada na Figura 29f. Uma linha horizontal ou vertical pode ser segmentada e permitir mais de uma conexão, aumentando o grau de conectividade da rede.

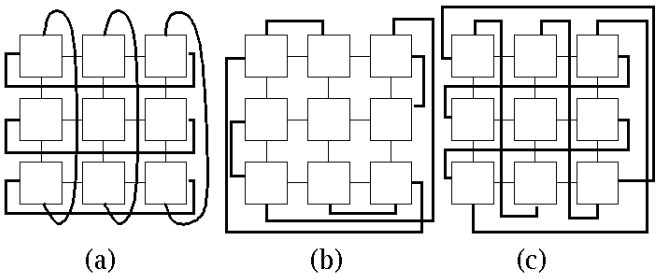


Figura 30: Padrões regulares em malha com grau 4 em cada PE e ligações periféricas

As topologias também podem ter ligações em forma de toróide (ou anel) nas extremidades. O uso dessas ligações reduz significativamente o tamanho médio das conexões. Três padrões de ligações são ilustrados na Figura 30 para uma topologia grid. A implementação física pode fazer uso de camadas extras de metal para preservar a escalabilidade e baixo custo da conexão ponto a ponto. A Figura 30a mostra um toróide convencional com conexões entre as extremidades de cada linha e coluna. A Figura 30b usa ligações entre as diagonais do arranjo e ligações locais duplicadas entre PE das extremidades situados no centro do arranjo. A Figura 30c mostra

ligações em *zig-zag* entre as extremidades, e duas ligações em diagonal nos cantos do arranjo. Todas as três propostas tem quatro ligações locais em cada PE.

3.3 Padrões *Hop*

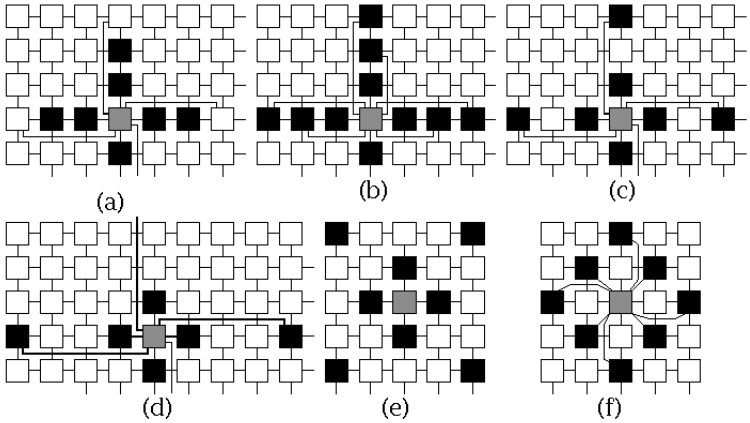


Figura 31: Padrões N-Hop e 0-N-Hop:(a) 1-Hop (b) 2-Hop (c) 0-2Hop (d) 0-3Hop (e) 0-2HOP Octal (f) 1-Hop Toroidal

Uma variação das topologias em grid pode ser gerada com a incorporação de ligações com saltos (*hops*). Um exemplo é mostrado na Figura 31a, que apresenta o padrão de conexão 1-Hop em grid. Considere o PE com coloração cinza, na parte central da Figura 31a. Além de 4 vizinhos diretos (com coloração preta), na mesma linha ou coluna com distância 1, existem mais 4 ligações com distância dois nas direções horizontal e vertical. Ou seja, de maneira mais formal o $PE_{i,j}$ possui uma conexão direta com o $PE_{i+1,j}$, o $PE_{i-1,j}$, o $PE_{i,j+1}$, e o $PE_{i,j-1}$, que correspondem às ligações diretas e as ligações que saltam um elemento com o $PE_{i+2,j}$, o $PE_{i-2,j}$, o $PE_{i,j+2}$, o $PE_{i,j-2}$, onde i representa a linha e j a coluna. Este tipo de topologia vem mostrando resultados interessantes para aplicações de processamento de sinal mapeadas em arranjos reconfiguráveis (BANSAL et al., 2004a). Nas próximas seções serão abordadas as variações das topologias com saltos (*hop*). Um salto serve para reduzir ligações longas, que conduz a resultados interessantes para as aplicações de processamento de sinal, como demonstrarão os experimentos realizados nesta dissertação.

A topologia 1-Hop é uma classe da topologia N-Hop, onde N representa o número máximo de elementos que a conexão pode saltar. Por exemplo, uma topologia 2-Hop pode saltar 0, 1 ou 2 elementos na mesma linha ou coluna como ilustra a Figura 31b. Em uma topologia grid

N-Hop, cada PE terá 4*(N+1) conexões locais, ou seja para 1-hop teremos 8 vizinhos diretos, e para o 2-hop teremos 12 vizinhos. Apesar de ter um custo linear em função de N, o número total de conexões pode ser bem alto quando comparado com o número de conexões utilizadas pelas aplicações mapeadas como será mostrado nos resultados experimentais no capítulo 5.

Com o objetivo de reduzir a complexidade (número) de ligações locais, e possibilitar uma comparação direta com outros padrões com a mesma complexidade, iremos considerar um padrão denotado por 0_N-Hop. Vale ressaltar, que outros trabalhos consideram apenas os padrões N-Hop (BANSAL et al., 2004a; HAUCK; BORRIELLO; EBELING, 1998).

Em um padrão 0_N-Hop, cada $PE_{i,j}$ possui uma conexão com o $PE_{i+1,j}$, o $PE_{i-1,j}$, o $PE_{i,j+1}$, e o $PE_{i,j-1}$, que correspondem as ligações diretas, e as ligações que saltam N elementos com o $PE_{i+N,j}$, o $PE_{i-N,j}$, o $PE_{i,j+N}$, o $PE_{i,j-N}$. A Figura 31c ilustra o padrão 0_2-Hop para um grid, e a Figura 31d ilustra o padrão 0_3-Hop. A vantagem do padrão 0_N-hop é manter o custo local em 8 conexões que é equivalente as topologias 1-hop e Octal, e ter também conexões com saltos longos.

Outras variações podem ser propostas e avaliadas. Por exemplo, o padrões Hop podem ser incorporados a uma topologia hexagonal ou octal. A Figura 31e ilustra um misto de octal e 1-Hop, como quatro ligações 0-hop nas linhas e colunas, e quatro ligações que saltam 1 elemento nas diagonais. A intenção é poder ter um salto longo em diagonal. Um outra variação seria usar 0-hop na diagonal e saltar um elemento nas linhas e colunas como mostra a Figura 31f. Porém este padrão gera sub-conjuntos desconectados no arranjo. Os PE's com a coloração branca (ou vazada) não possuem conexão com os PE's em preto ou o PE central em cinza. Este padrão poderia ser interessante se complementado com ligações adicionais para garantir conectividade extra entre os sub-conjuntos.

A Figura 32 mostra as topologias 1-Hop e 0_2-Hop com conexões toroidais nas extremidades. Apenas as conexões do *PE* do canto superior esquerdo são mostradas. Como iremos mostrar nos resultados, estas topologias apresentam um ganho considerável pois geram uma redução significativa do custo do roteamento, aumentam a conectividade do arranjos, tornam a solução mais regular pois cada PE tem 8 conexões locais, reduzem o tráfego de roteamento no centro do arranjo, dentre outras.

Uma variação do padrão 0_N-Hop seria manter oito conexões locais com o N variável.

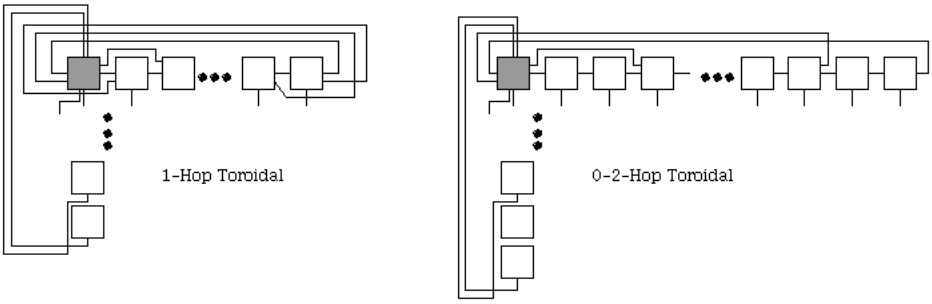


Figura 32: Padrões 1-Hop e 0_2-Hop com ligações toroidais

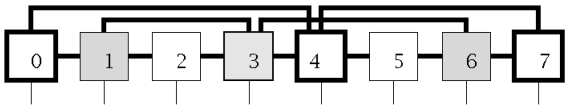


Figura 33: Subconjunto de *PE*s com o Padrão 0_N-Hop Híbrido e Assimétrico

Esta topologia tem relação com a topologia *hash-nets* descrita por Chemij (1994) no contexto de computação paralela. Este padrão será denominado nesta dissertação como 0_N-hop híbrido não simétrico. O termo híbrido pelo fato das conexões terem saltos variados. Considere um sub-conjunto de PE's mostrado na Figura 33. O PE_0 tem uma conexão que salta três *PEs*, enquanto que o PE_1 tem uma conexão que salta 1 *PE*. O termo não simétrico é devido ao fato dos saltos em direções diferentes para o mesmo *PE* não serem idênticos. Por exemplo, o PE_3 salta um *PE* para esquerda e dois *PE*s para a direita.

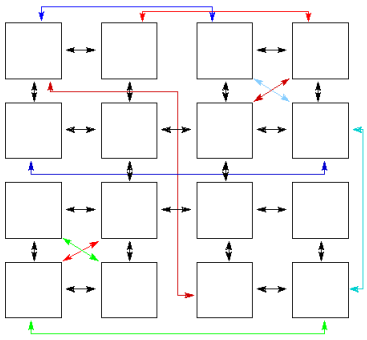


Figura 34: Arranjo 4x4 com padrão 0_N-Hop híbrido não simétrico

Além disso, variações destes padrões com menos restrições podem ser geradas. Considere a Figura 34 para visualizar um arranjo com 16 elementos, ou 4x4, como o padrão híbrido

não simétrico, onde nem todos os elementos têm conexões com os vizinhos diretos, além das ligações em saltos na mesma linha e coluna, também é permitido ligações entre um elemento qualquer $PE_{i,j}$ com outro elemento $PE_{k,l}$, sendo $i \neq k$ e $j \neq l$.

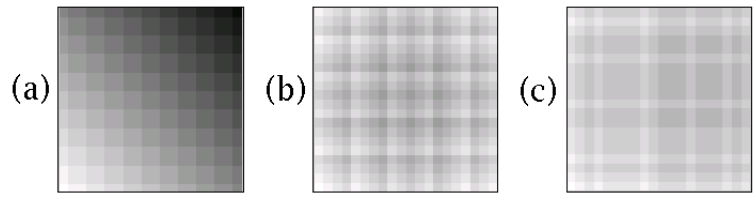


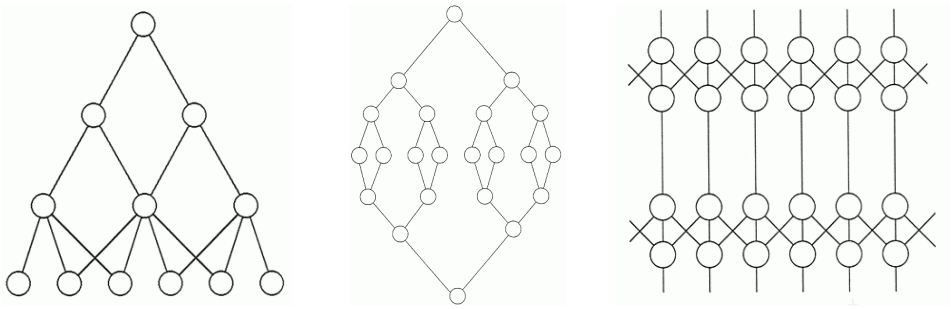
Figura 35: Distância de todos os *PEs* em relação ao *PE* do canto Inferior Esquerdo: (a) 1-hop (b) 0-3Hop Toroidal (c) Híbrido não Simétrico

Foi feita uma avaliação das distâncias entre os *PE* para visualizar o potencial de conexão das topologias. A Figura 35 ilustra com tons de cinza, variando do mais claro (mais próximo) ao mais escuro (mais distante), a avaliação de três topologias: 1-hop, 0_3-hop toroidal e 0_N-Hop híbrido não simétrico. A Figura 35a mostra que o 1-hop gera os piores resultados. Os padrões 0_3-hop toroidal e 0_N-Hop híbrido não simétrico apresentam resultados semelhantes na média, sendo que os pontos claros que são os conectados pelos saltos diretos.

Uma topologia com o padrão 0_N-Hop híbrido não simétrico pode ser especificada por vetores de distâncias relativas. Considere o exemplo mostrado na Figura 33, o vetor $v_1 = \{3,1,-,2,2,-,-\}$, onde o PE_0 é conectado ao PE_4 portanto salta 3 elementos, o PE_1 é conectado ao PE_3, saltando 1 elemento, e assim sucessivamente. Uma vez especificado o padrão, um algoritmo de cálculo de distância de todos para todos pode ser executado e uma função local de distância pode ser obtida para uso em algoritmos de posicionamento e roteamento. O padrão da Figura 35c usa apenas dois vetores, um para definir o padrão da linha e outro para o padrão da coluna, por isso tem uma distribuição de tons de cinza regular no sentido horizontal e vertical.

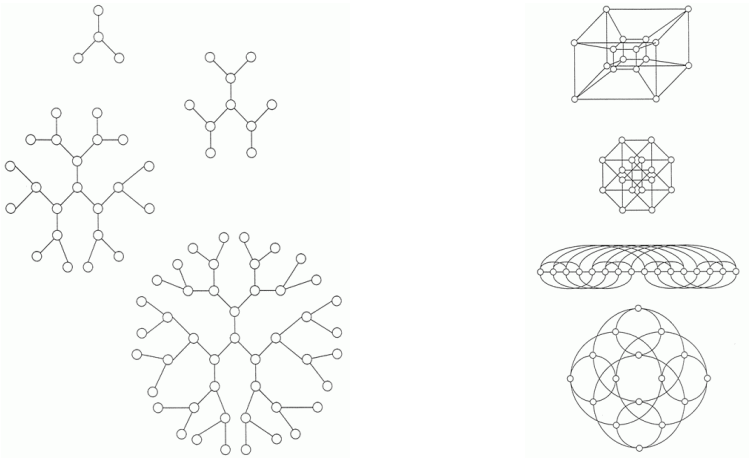
3.4 Modelo Genérico

Existe um grande espaço de topologias a ser explorado. A seguir são citados alguns modelos propostos para máquinas paralelas ao longo décadas de 60 a 80. As figuras 36, 37 e 38 foram extraídas de um estudo sobre máquinas paralelas disponível em (CHEMIJ, 1994).



(a) (b) (c)

Figura 36: Topologias em árvore (a) Grafos derivados de árvores (b) Forma de diamante; (c) Híbrido de grid (extraído de (CHEMIJ, 1994))



(a) (b)

Figura 37: Topologias: (a) Árvore n-ária (b) 4 grafos isomorfos de hipercubo dimensão 4 (CHEMIJ, 1994).

Como muitos grafos de fluxos de dados têm estrutura próxima à de uma árvore, variações deste formato poderiam ser exploradas, como ilustra a Figura 36. Os padrões das figuras 36a e 36b, mostram grafos derivados de árvores. A forma derivada de árvore serve para distribuir dados como ilustra a Figura 36a, e as conexões extras nas folhas aumentam a conectividade. A forma de diamante do grafo da Figura 36b é bem semelhante a muitos grafos de fluxo de dados, onde os dados são gerados por poucas fontes, distribuídos durante várias fases de cálculo e depois convergem para gerarem os resultados. A Figura 36c mostra um híbrido de conexões de grid e árvore com grau 3 nos dois sentidos. A Figura 37a mostra uma árvore com a raiz posicionada no centro e ramificações em todas as direções. A topologia hexagonal e a topologia octal com as conexões na diagonal possibilitam a exploração da conectividade no padrão árvore (diagonal) juntamente com padrão grid (horizontal e vertical).

A topologia hipercubo, juntamente com o grid, já foi muito estudada. Os hipercubos apresentam características interessantes de conectividades, que já foram exploradas por facilitarem a implementação algoritmos de roteamento, tolerância a falhas, etc. A Figura 37b ilustra alguns grafos isomorfos para um hipercubo de dimensão 4.

Além das variações acima citadas, várias outras podem ser geradas a partir de combinações ou mesmo o uso de padrões aleatórios, como mostra a Figura 38.

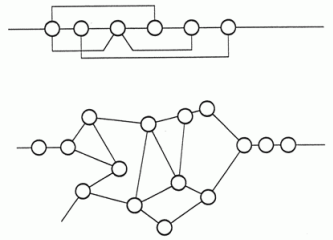


Figura 38: Dois padrões aleatórios de conexão (CHEMIJ, 1994).

Para derivar um modelo mais genérico que possa ser explorado sistematicamente por um ambiente de projeto, como o ambiente EDA, pode-se fazer uso de um grafo para especificar a topologia. Um barramento pode ser modelado como uma seqüência de vértices que executam apenas roteamento, sem implementar uma operação com um *PE*. Ou seja, o grafo pode ter vértices com capacidade distintas de roteamento. A Figura 39 mostra um grid com barramento e a modelagem do barramento como vértices do grafo. No exemplo, o $PE_{1,1}$ é conectado ao $PE_{1,2}$ com uma ligação local, e usa o barramento para conectar com o $PE_{2,4}$.

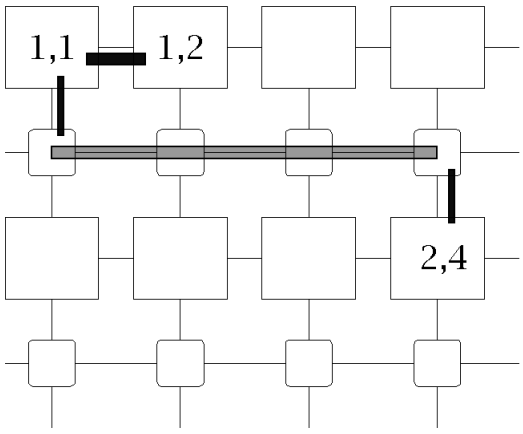


Figura 39: Grade com barramento modelado como vértices.

3.5 Conclusão

Este capítulo procurou ilustrar diferentes propostas de topologias em duas dimensões e outras variações. Além das topologias mais tradicionais como o grid, árvores e hipercubo, as topologias hexagonal, octal e toroidal também foram evidenciadas. Mais recentemente, as topologias N-Hop vem mostrando resultados interessantes pois possibilitam ligações locais que estão presentes nas aplicações como reduzem as ligações longas com o uso dos saltos. Entretanto se o valor de N for alto, o custo e a complexidade de implementar as conexões pode aumentar significativamente. Por outro lado, as topologias 0_N-Hop se mostram interessantes por preservarem o custo local e garantirem uma redução de ligações entre PEs distantes fisicamente graças aos saltos. Finalmente, mesmo como padrões aleatórios e irregulares, o custo das conexões pode ser baixo e uma redução das distâncias entre os PEs pode ser obtida.

4 POSICIONAMENTO DE CÉLULAS

4.1 Introdução

Na implementação de arquiteturas de fluxo de dados em arranjos de processadores, o posicionamento das unidades funcionais nas células de processamento é uma tarefa de grande importância e sua discussão tem sido feita em uma série de trabalhos.

Deve-se levar em consideração as facilidades e restrições inerentes à topologia escolhida para conseguir um bom resultado de posicionamento.

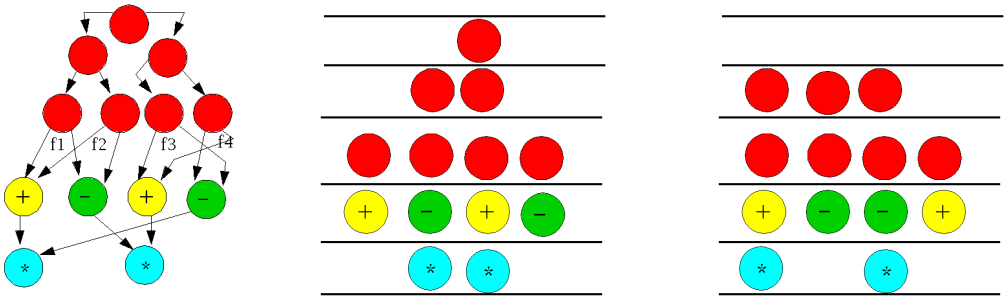
Neste capítulo, serão apresentadas algumas soluções para o problema de posicionamento. Inicialmente, será apresentada a primeira versão implementada no ambiente EDA (FERREIRA et al., 2005), onde o posicionamento é realizado por níveis, baseado em uma adaptação do algoritmo apresentado por Koren et al. (1988). Em seguida, será apresentada uma solução de posicionamento priorizando o caminho crítico. Finalmente, uma solução baseada em algoritmo genético é descrita. Esta solução mostrou eficiência e flexibilidade em relação às soluções anteriores.

4.2 Posicionamento Por Níveis

O posicionamento proposto por Koren et al. (1988) e posteriormente adaptado por (TOLEDO; FERREIRA, 2004; FERREIRA et al., 2005), foi a primeira versão implementada no ambiente EDA, onde os nodos são posicionados por níveis com uma heurística baseada na busca em profundidade. Os resultados obtidos demonstram que a técnica utilizada consegue bons resultados com relação ao comprimento médio das ligações, mas que a densidade de ocupação do arranjo é baixa. A maior contribuição da primeira versão foi a estrutura geral do mapeamento de forma independente da arquitetura, dando flexibilidade para incorporar novas

soluções de posicionamento (SILVA et al., 2006) e roteamento (GARCIA; FERREIRA, 2006). A Figura 18 mostra o fluxograma do ambiente de mapeamento. O uso da linguagem Java para a modelagem das classes (algoritmo, arranjo, célula e borda) e dos formatos intermediários em XML facilitaram as expansões e inclusão de novos algoritmos.

Para ilustrar o funcionamento do posicionamento por níveis, suponha o grafo de fluxo de dados apresentado na Figura 40(a). O algoritmo faz uma busca em profundidade no grafo organizando os nodos por níveis no arranjo (Figura 40b). O grafo deste exemplo tem 5 níveis em profundidade. Posteriormente, para cada nível é calculado seu possível centro de massa com o objetivo de se fazer ajustes no posicionamento horizontal dos nodos (Figura 40c). Como já mencionado, o posicionamento por níveis tem bons resultados para o comprimento médio das conexões em vários grafos. Entretanto, além da baixa densidade dos nodos, o algoritmo tende a necessitar de arranjos com dimensões maiores devido a utilização de maior número de linhas em relação ao número de colunas.



(a) Grafo de fluxo de dados

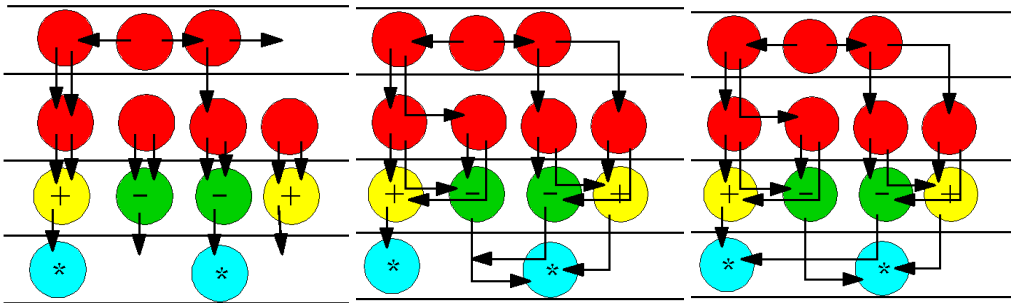
(b) Posicionamento por níveis e profundidade

(c) Ajuste dos níveis

Figura 40: Posicionamento por níveis do *benchmark* Fir2.

O roteamento também foi baseado na proposta de Koren et al. (1988) e adaptado por Toledo e Ferreira (2004). Uma lista de todos os arcos do grafo é construída. Depois, saindo do vértice de origem e usando a posição de destino como orientação, o algoritmo vai acrescentando uma conexão por vez. Desta forma, todos os caminhos são construídos ao mesmo tempo. A Figura 41 mostra os 3 passos necessários para rotear o grafo da Figura 40, primeiro todos os caminhos de comprimento um, ou seja, que tiveram um posicionamento ótimo, são completados, restando as conexões que necessitam de mais de uma ligação local. Posteriormente, mais uma ligação é adicionada a todos os caminhos incompletos. Os caminhos que atingiram o seu destino com tamanho 2 são completados e retirados da lista. O algoritmo prossegue assim até

completar todos os caminhos.



(a) Comprimento 1 (b) Comprimento 2 (c) Comprimento 3

Figura 41: Roteamento de todos os caminhos ao mesmo tempo.

Como ajuste é feito apenas dentro de cada linha, e o posicionamento é feito por níveis, os arranjos tendem a ter a forma de um losango ou diamante, como mostra a Figura 42.

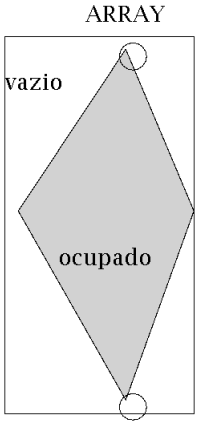


Figura 42: Ocupação do arranjo de processadores com o posicionamento por profundidade

Posteriomente ao algoritmo apresentado por (KOREN et al., 1988), outros autores continuaram a explorar a topologia hexagonal, tentando melhorar a ocupação e realizando um número maior de movimentações tanto no sentido vertical quando horizontal (ROBIC; SILC, 1993, 1994a; ROBIC; VILFAN, 1995a). Para isso, implementações do algoritmo de otimização *simulated annealing* foram adicionadas ao algoritmo de posicionamento. A Figura 43 ilustra um posicionamento inicial e sua otimização. Os resultados apresentados por (ROBIC; SILC, 1993, 1994a; ROBIC; VILFAN, 1995a) foram arranjos mais densos e menos alongados. Porém devido ao posicionamento inicial alongado, a densidade de ocupação continuou baixa, prevalecendo o formato de diamante.

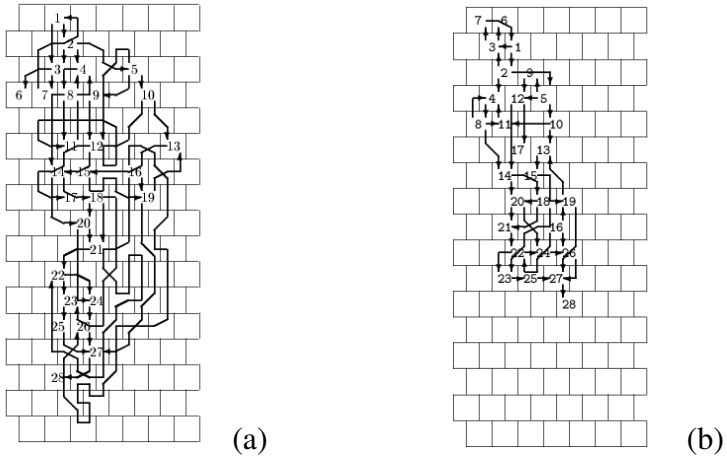


Figura 43: Posicionamento por níveis otimizado pela técnica de *simulated annealing*. Extraído de (ROBIC; SILC, 1994a)

4.3 Posicionamento por Caminho Crítico

Conforme Micheli (1994) o posicionamento de células baseadas na mobilidade dos nodos de um grafo é uma técnica amplamente utilizada no projeto de circuitos digitais, visando reduzir a latência e melhorar a vazão. Para determinar a mobilidade dos nodos, é necessário realizar dois escalonamentos no grafo: ASAP (*As Sooner As Possible*) e ALAP (*As Later As Possible*). O escalonamento ASAP determina o menor tempo possível para escalonar cada um dos nodos, respeitando as dependências. Enquanto o escalonamento ALAP determina o maior tempo para escalonamento dos nodos. De posse dos valores ASAP e ALAP para cada nodo, calcula-se sua mobilidade (ALAP - ASAP), isto é, qual a "folga" de escalonamento possível para o nodo. Nodos com mobilidade igual a zero são integrantes do caminho crítico do grafo. Quanto maior a mobilidade de um nodo, O exemplo da Figura 44 mostra o caminho crítico (em cinza) para o grafo do filtro fir4. Os rótulos dos nodos contêm a descrição da FU e entre parênteses os escalonamentos ALAP, ASAP e a mobilidade, nesta ordem.

Os nodos *imult_0*, *imult_3* e *iadd_0* são os únicos, no exemplo, que não fazem parte do caminho crítico. Estes nodos têm mobilidade diferente de zero. O nodo *imult_0* tem ASAP igual a 3 e ALAP igual a 5, o que significa que pode ocupar uma posição nos níveis 3, 4 ou 5 do grafo sem alterar a latência final da implementação. Todos os demais nodos têm mobilidade zero e, caso sejam movidos de nível, um atraso será adicionado.

A técnica de utilização do caminho crítico no posicionamento das FU's consiste em pri-

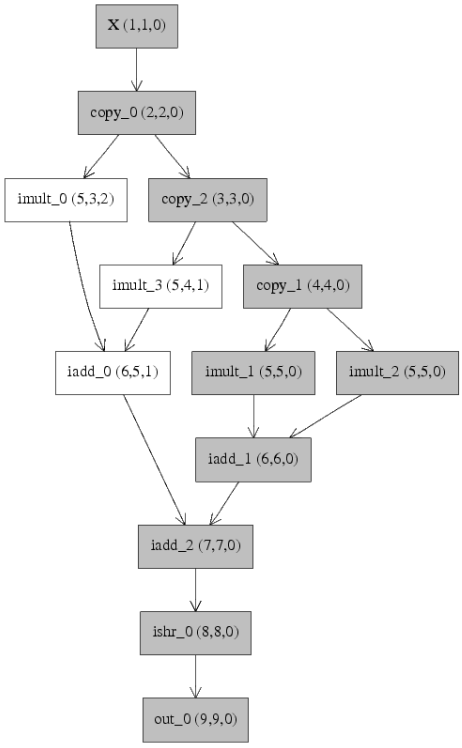


Figura 44: Mobilidade dos nodos do grafo Fir de ordem 4 (caminho crítico representado em cinza)

meiro posicionar de maneira compacta o caminho crítico no arranjo de processadores. Isto foi feito como proposto em (BANSAL et al., 2004b). Depois foi feito o mapeamento dos nodos restantes em ordem crescente de mobilidade, de forma a colocar cada um deles na posição mais próxima à ideal. Caso a posição ideal já esteja ocupada, uma outra posição vazia nas adjacências desta é procurada, utilizando-se o enfoque "guloso" (*greedy*), como ilustra da Figura 45

X	C0	C2	C1
	M0	M3	M1
		A0	A1
	Oo	S0	A2

Figura 45: Posicionamento por caminho crítico e mobilidade (ASAP-ALAP) em forma de U

Posicionamento do caminho crítico

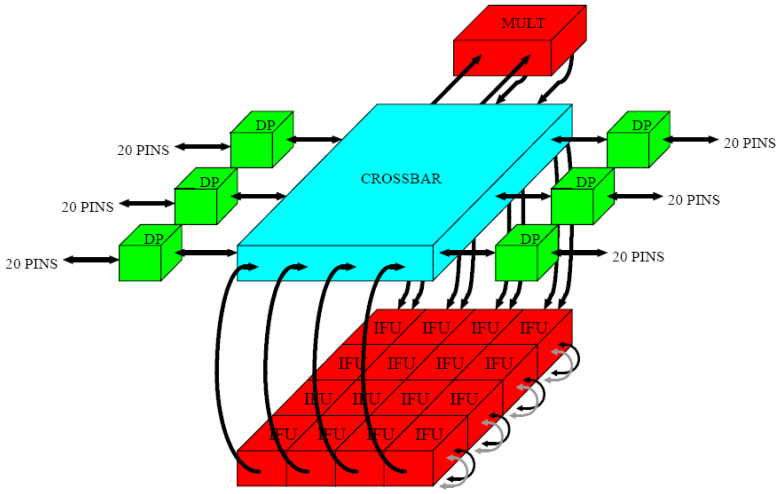


Figura 46: Diagrama da Arquitetura *Colt* extraído de (BITNER, 1996)

4.4 Algoritmo Genético

Como já foi mencionado, o posicionamento por nível, por caminho crítico e mesmo o baseado em *simulated annealing* (ROBIC; VILFAN, 1995b) geram soluções muito esparsas na ocupação dos arranjos. Nesta seção, será apresentada a implementação de um algoritmo genético como solução para melhorar a ocupacão do arranjos. A utilização de métodos evolutivos de busca é bastante conhecida em problemas de otimização de projeto de circuitos e escalonamento de tarefas (RUSSELL; NORVIG, 2004, p. 117). No escopo deste trabalho, esta solução deve se mostrar flexível o bastante para explorar várias arquiteturas regulares ou não, homogêneas ou não, etc.

Uma abordagem interessante é a utilizada por (KAHNE, 1997) para implementar o posicionamento na arquitetura *Colt* (BITNER, 1996), que consiste basicamente de uma matriz 4*x*4 com dezesseis unidades funcionais conectadas, seis unidades de entrada e seis de saída, um multiplicador e uma rede de *crossbar* para ligações entre as unidades. Conforme mostrado na Figura 46. O algoritmo de posicionamento é um algoritmo genético que determina, com base na distância média das ligações, uma solução para o problema.

A abordagem da implementação nesta dissertação é baseada no trabalho anterior proposto em (KAHNE, 1997), mas com alterações para permitir flexibilidade no tamanho do arranjo, nas topologias utilizadas, e avaliações com *benchmarks* maiores.

Para realizar o posicionamento das unidades funcionais no arranjo de processadores, inici-

almente é calculado o lado l de um quadrado mínimo do arranjo necessário para alojar todas as unidades necessárias para mapear o conjunto de v vértices do grafo. Este cálculo é feito de acordo com a Equação 4.1.

$$l = \lceil \sqrt{v} \rceil \tag{4.1}$$

Somente com este procedimento, já se consegue a compactação ótima para o posicionamento. A tarefa do algoritmo genético é evoluir o posicionamento dos vértices dentro dessa área, reduzindo o comprimento das conexões entre vértices relacionados.

Alternativamente, o sistema aceita que o usuário defina as dimensões de uma área irregular para conter o posicionamento. Esta escolha deve ser feita de maneira tal que a área disponibilizada possa conter todos os nodos do grafo.

O usuário pode informar também a localização do posicionamento dentro arranjo, com esta possibilidade pode-se mapear mais de um grafo em um arranjo de processadores, inclusive utilizando áreas não contíguas.

4.4.1 Representação

Ao longo das próximas seções, serão apresentadas várias técnicas de posicionamento. O grafo de fluxo de dados de um filtro de resposta a impulsos de segunda ordem (Fir2), ilustrado na Figura 47, será usado para ilustrar as diferentes estratégias.

Dentre as várias opções apresentadas por Mitchell (1996, p. 117) para descrever indivíduos em um algoritmo genético, foi escolhida a representação *Many-Characters* (vários caracteres) onde um indivíduo é descrito por uma seqüência de caracteres de algum alfabeto. Neste caso, o alfabeto escolhido foi o conjunto de valores de 1 a l^2. Os valores são armazenados em um vetor onde unidades funcionais relativas aos vértices do grafo são associadas às suas posições, como mostra a Figura 48. Em (a), temos a representação do indivíduo para o algoritmo genético: um vetor contendo as FU's do grafo de fluxo de dados. A representação deste indivíduo no arranjo de processadores é mostrada em (b). Há uma ligação direta entre cada caractere do indivíduo (FU) com uma posição no arranjo. Com isto, é fácil determinar onde estão as unidades funcionais e as distâncias entre elas.

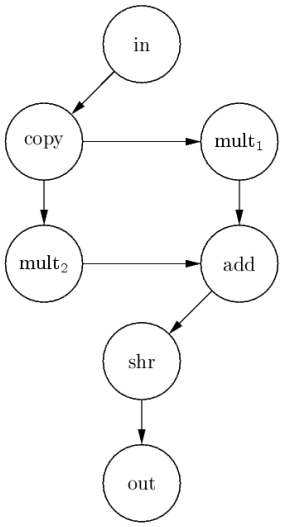


Figura 47: Grafo de fluxo de dados de um filtro Fir2

$mult_2$	$mult_1$	in	add		out		shr	copy

(a) Para o algoritmo genético

$mult_2$	$mult_1$	in
add		out
	shr	copy

(b) no arranjo de processadores

Figura 48: Exemplo da representação do posicionamento

Os vértices do grafo são distribuídos no vetor utilizando algum critério para gerar a população inicial para o algoritmo genético. Um dos critérios faz a distribuição puramente aleatória dos vértices: cada indivíduo da população inicial é uma permutação particular das unidades funcionais gerada de maneira aleatória. Outro critério é gerar a população inicial de uma maneira que procure respeitar, mesmo a "grosso modo", a localidade das ligações presentes nos grafos de fluxo de dados para diminuir o tempo de convergência do algoritmo genético.

4.4.2 O Algoritmo Genético

O algoritmo genético de posicionamento é baseado em uma implementação genérica em Java desenvolvida por (IORIO et al., 2004), com algumas alterações para adaptar e otimizar o desempenho para o problema de posicionamento. O algoritmo básico é descrito na Figura 49.

O desempenho da busca realizada por um algoritmo genético é fortemente acoplado a de-

```
Entradas: População inicial, Descrição da topologia,
          Grafo de Fluxo de dados.
Saída: Melhor indivíduo encontrado.

1    geraçõesSeguidasSemMelhoria = 0;
2    tempoDecorrido = tempo_atual();
3    ordene os indivíduos da população em ordem decrescente das aptidões
4    melhorAptidão = 0
5    enquanto (tempoDecorrido > tempo_atual() + tempoConcedido e
               geraçõesSeguidasSemMelhoria < maxGerações) faça
5.1         ordene a população em ordem decrescente das aptidões
5.2         se (aptidão(melhorIndividuo) <= melhorAptidão)
5.2.1            então geraçõesSeguidasSemMelhoria++
5.2.2            senão geraçõesSeguidasSemMelhoria = 0
5.2         fimSe
5.3         selecione indivíduos para exclusão
5.4         selecione indivíduos para cruzamento
5.5         realize os cruzamentos gerando novos indivíduos.
5.6         escolha indivíduos para mutação e as realize.
5    fimEnquanto
6    retorne o melhor indivíduo encontrado
```

Figura 49: Descrição do algoritmo genético

terminados aspectos, como a representação escolhida para os indivíduos (RUSSELL; NORVIG, 2004) e a qualidade da função de aptidão utilizada (MITCHELL, 1996).

4.4.3 Função de aptidão

Para descrever a aptidão de cada indivíduo, as estratégias utilizadas foram a minimização do custo total (a somatória das distâncias), e valorização dos indivíduos que apresentam maior número de ligações com distâncias menores ou iguais a um limite pré-determinado. A função de aptidão (*Fitness*) deve representar de maneira unificada os dois objetivos citados anteriormente. A equação 4.2 mostra uma função utilizada para representar a aptidão de um posicionamento descrito por um indivíduo.

Cada topologia tem um comportamento diferente com relação à distância entre células. A Tabela 1 mostra fórmula de distâncias entre duas células $A(x,y)$ e $B(x,y)$ para as topologias: grid, octal, 1-hop, 0_2-Hop e 2-Hop. Os valores d_x e d_y representam as expressões $|A_x - B_x|$ e $|A_y - B_y|$, respectivamente.

A Figura 50 representa matrizes de distâncias a partir da célula central e ilustra o alcance possível de ligações para cada topologia. Pode-se notar a maior facilidade para o roteamento nas topologias 1hop e octal, demonstrada com a existência de maior quantidade de células com

Tabela 1: Cálculo de distâncias entre células em algumas topologias

Topologia	**Distância**
Grid	$D_{(A,B)} = d_x + d_y$
Octal	$D_{(A,B)} = \min\{d_x, d_y\} + \lvert d_x - d_y \rvert$
1-Hop	$D_{(A,B)} = d_x \operatorname{div} 2 + d_y \bmod 2 + d_y \operatorname{div} 2 + d_y \bmod 2$
0_2-Hop	$D_{(A,B)} = d_x \operatorname{div} 2 + d_y \bmod 2 + d_y \operatorname{div} 2 + d_y \bmod 2$
2-Hop	$D_{(A,B)} = d_x \operatorname{div} 3 + d_y \bmod 3 + d_y \operatorname{div} 3 + d_y \bmod 3$

distâncias 1 ou 2.

4	3	2	3	4
3	2	1	2	3
2	1	**0**	1	2
3	2	1	2	3
4	3	2	3	4

(a) Grid

3	2	2	2	3
2	2	1	1	2
2	1	**0**	1	2
2	2	1	1	2
3	2	2	2	3

(b) Hexagonal

2	2	2	2	2
2	1	1	1	2
2	1	**0**	1	2
2	1	1	1	2
2	2	2	2	2

(c) Octal

2	2	1	2	2
2	2	1	2	2
1	1	**0**	1	1
2	2	1	2	2
2	2	1	2	2

(d) 1-Hop

Figura 50: Matrizes de distâncias em algumas topologias

$$\text{Fitness}(D) = \sum_{1}^{n} D_i P_i \tag{4.2}$$

Considerando a função de aptidão baseada na equação 4.2, onde D indica o vetor de distâncias para a qual é calculada a aptidão, o indivíduo tem n genes e para cada um deles é associado um peso P_i. O vetor de pesos escolhido para o cálculo da aptidão é descrito na Tabela 2. Os valores foram obtidos por meio de testes do algoritmo, realizados com alguns grafos escolhidos empiricamente.

O algoritmo para cálculo da aptidão do indivíduo é listado na Figura 51 e deve retornar maiores valores para indivíduos que sejam mais aptos. A determinação da menor distância entre duas células é feita de acordo com as fórmulas listadas na Tabela 1. A complexidade

Tabela 2: Associação de pesos às distâncias dos arcos

Distância	**Peso**
1	5000
2	500
3	50
4	5
> 5	-50 d^2

do algoritmo é proporcional ao número de arestas. Entretanto, o custo não é proibitivo e sua utilização só é necessária para uma fração dos indivíduos a cada iteração do algoritmo. Esta fração é composta apenas pelos novos indivíduos gerados ou para indivíduos da população que sofrem mutação.

```
Entradas: Grafo de fluxo de dados, vetor de bônus
Saídas: valor da aptidão

1       total = 0;
2       totalBonus = 0;
3       maior = 0;
4       hist = histograma com os valores das distâncias iguais a zero.
5       para cada aresta a no grafo faça
5.1           distancia = menor_distância(a.fonte, a.destino);
5.2           hist[distancia]++;
5.3           se distância > maior então maior = distância
5.4           total += distância
5       fimPara
6       para i=1 até maior faça
6.1           se i < 4
6.1.1              então totalBonus += hist[i] * bonus[i];
6.1.2              senão totalBouns += hist[i] * bonus[4];
6.1           fimSe
6       fimPara
7       aptidão = -(total - totalBonus)
```

Figura 51: Algoritmo para cálculo da aptidão do indivíduo

O cálculo da menor distância é feito utilizando somente informações sobre a topologia e não leva em consideração as restrições impostas durante o roteamento, como células com excesso de ligações.

4.4.4 Operadores genéticos

Cruzamento

A operação de cruzamento em um algoritmo genético visa mesclar características de dois indivíduos para gerar novos indivíduos. Com isto, espera-se aproveitar características boas dos indivíduos pais para gerar filhos com melhor aptidão.

Na implementação corrente, a técnica utilizada para fazer o cruzamento consiste em selecionar dois indivíduos da população com maior probabilidade de escolha para indivíduos com maior aptidão. Posteriormente, aleatoriamente é definido um ponto de cruzamento P no intervalo $[2, n-1]$. A Figura 52 ilustra o cruzamento de dois indivíduos (pais) para gerar dois novos indivíduos (filhos). O ponto escolhido para quebrar a representação dos dois indivíduos foi após o terceiro cromossomo (unidade funcional). A parte **a** do primeiro pai (em cinza) é copiada para o primeiro filho, o restante do primeiro filho, é composto por elementos do segundo pai (em branco) completados por elementos da parte inicial quando o cromossomo da posição já estiver repetido na primeira parte do indivíduo filho. Desta maneira, cada indivíduo herda características dos pais mas sem perder a propriedade de possuir todos cromossomos diferentes uns dos outros. O segundo filho é gerado de maneira análoga ao primeiro, com o primeiro segmento vindo do segundo pai e o segundo segmento do primeiro pai, evitando repetições.

$mult_2$	$mult_1$	in		add	out		shr	copy
	copy	shr	$mult_2$	in	add	out		mult1

(a) Pais

$mult_2$	$mult_1$	in		copy	add	out		shr
	copy	shr		add	out	$mult_2$	$mult_1$	in

(b) Filhos

Figura 52: Representação de um cruzamento entre indivíduos

Mutação

A operação de mutação nesta implementação do algoritmo genético é mostrada na Figura 53 e consiste simplesmente em escolher aleatoriamente duas posições dentre os genes do indivíduo (os cromossomos escolhidos para a mutação estão em negrito) e realizar a troca destas posições. A escolha desta estratégia de mutação possibilita manter a unicidade dos cromossomos que des-

crevem os vértices do grafo. Após a realização da mutação, a aptidão do indivíduo é calculada novamente para refletir o novo estado dos genes.

$mult_2$	**$mult_1$**	in		add	**out**		shr	copy

(a) Indivíduo original

$mult_2$	**out**	in		add	**$mult_1$**		shr	copy

(b) Indivíduo após mutação.

Figura 53: Mutação em um indivíduo

Mutações em algoritmo genéticos geram "perturbações" com o intuito de retirar de possíveis mínimos locais a solução descrita por uma população .

4.4.5 Seleção de Indivíduos

A cada iteração do algoritmo genético, devem ser escolhidos indivíduos para cruzamento, gerando novos indivíduos e descartando uma parte da população. Dentre as técnicas utilizadas para este fim, foi escolhida nesta implementação a seleção proporcional baseada na aptidão (método de seleção da roleta - *roulette wheel selection*). Este método utiliza a aptidão de cada indivíduo para calcular a probabilidade dele participar de um cruzamento ou ser descartado. Quanto maior a aptidão de um dado indivíduo, maior a chance dele cruzar com outro para gerar novos indivíduos e menor sua chance de ser descartado.

Este método comprovadamente exibe bons resultados, entretanto exige mais processamento pois a população deve ser percorrida para realizar as escolhas, dificultando a utilização deste algoritmo em um ambiente onde seja necessária a reconfiguração do arranjo em tempo de execução. Porém, o ambiente é flexível e permite que outros métodos mais rápidos possam ser escolhidos futuramente, como o método da seleção por torneio (*tournament selection*), que escolhe aleatoriamente um grupo de indivíduos e faz um "torneio" entre eles onde o de melhor aptidão tem uma certa probabilidade p de ser escolhido, o segundo tem probabilidade $p(1-p)$, o terceiro $p(1-p)^2$ e assim por diante.

4.4.6 Parâmetros do algoritmo genético

Para tentar otimizar o desempenho da busca por uma boa solução, o algoritmo genético possui alguns parâmetros ajustáveis:

Tamanho da população Número de indivíduos presentes na população a ser evoluída pelo algoritmo.

Percentual de mutações Descreve o percentual da população que sofrerá mutação em uma iteração do algoritmo.

Razão entre cruzamento e eliminação Valor utilizado na escolha de indivíduos da população para cruzamento ou eliminação. Indica a razão de probabilidade de escolha entre o indivíduo mais adaptado da população para cruzamento e o indivíduo menos adaptado da população para eliminação.

Percentual de renovação Percentual de indivíduos que serão descartados a cada iteração do algoritmo e substituídos por outros gerados por cruzamento de indivíduos melhor adaptados.

Estes parâmetros têm influência direta no tempo de convergência e na possibilidade de alcance de mínimos locais ao invés de globais (MITCHELL, 1996, p. 129). O tamanho da população interfere no tempo de convergência pois, quanto maior a população, mais tempo será gasto para se realizar uma iteração. Uma população muito pequena tem maior probabilidade de estacionar em mínimos locais. O valor da taxa de renovação da população também tem um impacto no tempo de convergência.

4.4.7 Critérios de Parada

Dois critérios de parada são utilizados para terminar uma busca:

Tempo de execução Quantidade máxima de tempo permitida para a execução do algoritmo genético.

Iterações sem melhoria Número máximo de iterações do algoritmo genético (gerações) sem melhoria na aptidão da população em seqüência.

O primeiro critério de parada é atingido quando o tempo atribuído para execução do algoritmo chega ao fim. O segundo é alcançado quando a aptidão do melhor indivíduo da população evoluída pelo algoritmo atingiu um patamar por um determinado número de iterações.

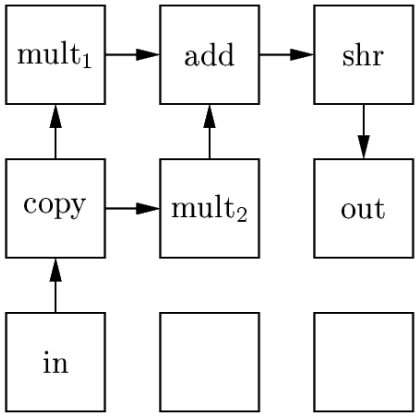
A Figura 54 mostra um exemplo de posicionamento ótimo para o grafo do filtro Fir2 em uma topologia de grade (*grid*).

$mult_1$	add	shr	copy	$mult_2$	out	in		

(a) Representado no algoritmo genético

$mult_1$	add	shr
copy	$mult_2$	out
in		

(b) No arranjo de processadores



(c) Representação gráfica com as conexões utilizadas

Figura 54: Um exemplo de posicionamento ótimo

4.4.8 Arranjos Heterogêneos

Uma das contribuições deste trabalho é explorar arranjos heterogêneos. Alguns operadores implementados em elementos de processamento no arranjo de processadores têm complexidade maior que outros operadores mais simples. Multiplicadores e memórias tendem a gastar maior área para sua implementação. Entretanto, estes recursos mais "caros" não necessitam ser implementados em todas as posições do arranjo. As arquiteturas podem distribuir tais elementos em determinadas posições tornando-as fixas (MEI et al., 2005; AHN et al., 2006; CONG; XU, 2000). Dessa forma, as outras posições do arranjo podem ocupar menor área por implementarem operações mais simples.

A Figura 55 mostra duas possíveis organizações para arranjos heterogêneos: com operadores mais complexos distribuídos segundo algum critério ou em colunas fixas.

Para permitir arranjos com recursos heterogêneos, o operador de cruzamento entre dois

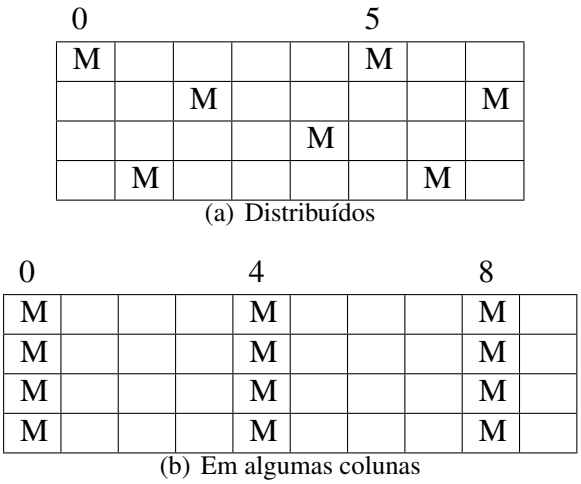


Figura 55: Exemplos de arranjos heterogêneos com posicionamento diferenciado de multiplicadores

indivíduos deve ser alterado para que realize a permutação de forma que cada unidade funcional somente seja permutada por uma de tipo equivalente, caso ocorra repetição. De forma análoga, operador de mutação somente deverá trocar de posição unidades funcionais de tipos equivalentes.

4.4.9 População Inicial

População Inicial Aleatória

A escolha dos indivíduos para compor a população inicial foi implementada, em um primeiro momento, através da geração aleatória de permutações de valores contidos no intervalo $[1, n]$, onde n é o número de PE's disponíveis para o mapeamento. Esta escolha, apesar de ser facilmente implementável, gera soluções com grande distanciamento entre nodos relacionados. No tópico a seguir é discutida uma melhoria para a população inicial.

População Inicial Melhorada

O tempo de convergência do algoritmo genético é bastante sensível à qualidade da população inicial. A geração aleatória de indivíduos para composição desta população tal como descrito na seção 4.4.1, torna pouco provável a ocorrência de indivíduos com boas características. Com isto é necessário mais tempo para que o algoritmo genético ache um posicionamento aceitável.

Uma estratégia implementada para melhorar a qualidade da população inicial, diminuindo o tempo de busca, é a geração de indivíduos para a população inicial de forma a respeitar a localidade das ligações entre nodos mas com algum comportamento aleatório. O grafo é percorrido em níveis (*breadth first*) ou em profundidade (*depth first*) e as unidades funcionais correspondentes aos nodos são mapeadas no arranjo utilizando técnicas de varredura tais como "zig-zag" horizontal ou vertical, espiral, forma de "L", forma de "U", dentre outras. Os indivíduos gerados desta forma são combinados com outros gerados de maneira aleatória para formar a população inicial do algoritmo genético. Com estas alterações, a convergência do algoritmo genético passou a ser mais rápida em comparação com a população inicial puramente aleatória.

A Figura 56 mostra algumas opções para realizar este mapeamento. No alto, três opções de varredura do arranjos são ilustradas: em zig-zag, espiral ou zig-zag na diagonal. No meio da figura, um grafo de fluxo de dados é percorrido em profundidade, enquanto que o arranjo é varrido em zig-zag horizontal. Finalmente, na parte de baixo da Figura 56, o grafo é percorrido em largura e uma varredura em zig-zag diagonal é realizada no arranjo para implementar o posicionamento inicial.

4.5 Considerações Finais

Este capítulo apresentou várias técnicas de posicionamento de unidades funcionais em um arranjo de processamento. Por ser um problema NP-Completo (LAI; LAI; YEH, 2005), a resolução do problema do posicionamento exige que se encontre boas soluções heurísticas e de busca exaustiva.

Outro fator importante com relação ao posicionamento é a dependência do algoritmo de roteamento de um bom posicionamento para conseguir realizar a ligação entre as unidades funcionais. Caso o posicionamento não seja bom, onde nodos com ligações diretas sejam posicionados em células distantes, o roteamento em topologias com mais restrições de recursos de ligações como grid e hexagonal será dificultado.

A minimização dos recursos utilizados para roteamento tem impacto no atraso das computações (HAUCK; BORRIELLO; EBELING, 1998). Quanto mais saltos de interconexão de células forem utilizados, maior o atraso.

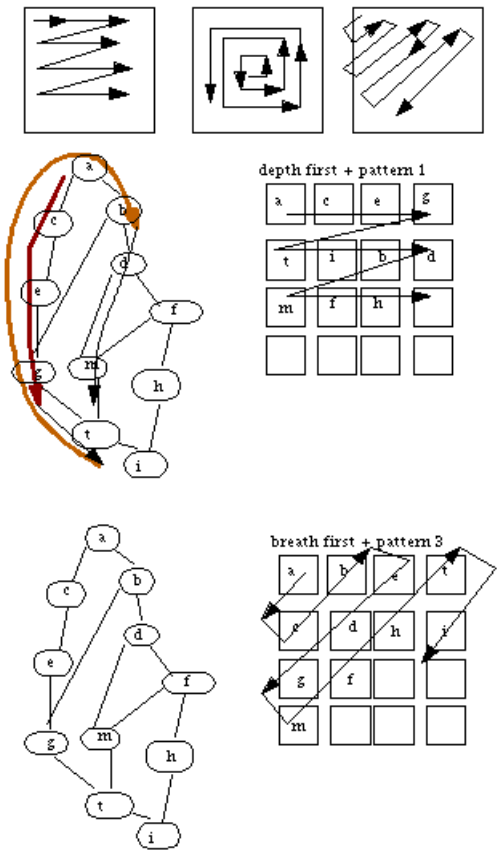


Figura 56: Percurso dos grafos e varredura dos arranjos

5 RESULTADOS

Neste capítulo, será feita a avaliação das extensões propostas ao ambiente EDA sob os aspectos do posicionamento e roteamento de FU's utilizando vários *benchmarks* mapeados em um conjunto de topologias.

As métricas utilizadas para avaliar os mapeamentos foram o comprimento médio, máximo e total das ligações, a compactação do posicionamento no arranjo e a utilização dos recursos disponíveis de roteamento. As medidas de comprimento são importantes pois caminhos mais longos tender a demandar filas maiores para balanceamento de caminhos no fluxo de dados. A compactação do posicionamento determina o tamanho do arranjo necessário para implementação do algoritmo. Esta métrica pode ser obtida com o número de linhas *versus* o número de colunas ($l \times c$) de um arranjo necessárias para se realizar o mapeamento.

Inicialmente, serão apresentadas as arquiteturas avaliadas e os algoritmos utilizados para *benchmark*. Serão apresentados resultados do posicionamento do caminho crítico, do algoritmo genético com população inicial totalmente aleatória e população inicial melhorada. Sempre que possível, os resultados obtidos neste trabalho foram comparados com os de trabalhos anteriores.

5.1 Aplicações e Arquiteturas Avaliadas

5.1.1 Conjunto de *Benchmarks*

Para avaliar o desempenho do ambiente, vários *benchmarks* da área de processamento de sinais tiveram o núcleo do algoritmo descrito como um grafo de fluxo de dados baseado em técnicas de compilação descritas em (BUDIU; GOLDSTEIN, 2002), e mapeados em diferentes arquiteturas baseadas em arranjos de grão grosso. Todas unidades funcionais que implementam os vértices do grafo têm largura de barramento de 32 bits, mas esta é configurável, e pode

ser facilmente alterada e definida na especificação da arquitetura em XML. Como protocolo de comunicação, os operadores foram implementados para trabalharem de forma assíncrona, usando um protocolo típico de 4 fases com os sinais de Ready/Ack(HOLZMANN, 1991).

Os algoritmos utilizados foram traduzidos manualmente para uma representação em grafo de fluxo de dados. O conjunto de algoritmos para *benchmark* está listado na Tabela 3. Na definição deste conjunto, foram escolhidos algoritmos que abrangessem aplicações usuais de multimídia, como processamento de imagens e codificação de áudio.

Tabela 3: *Benchmarks* utilizados e sua breve descrição

Benchmark	Descrição
FIR	Filtro de resposta finita ao impulso
CPLX	Implementação do filtro FIR com aritmética de números complexos
FDCT	Transformada discreta em cossenos rápida
Paeth	Rotina de codificação do filtro *paeth* do padrão PNG
FilterRGB	Filtro para destacar uma imagem clareando ou escurecendo *pixels*
ADPCM	Codificador PCM adaptativo disponível no repositório *MediaBench*
SNN	Filtro de Imagem por vizinhança
Gouraud	Algoritmo de sombreamento de imagens

Foram usadas versões dos filtros FIR de ordens 8, 16, 64 e 128. O filtro FIR foi escolhido por ter um padrão bem definido que pode ser facilmente parametrizado e posicionado em três linhas se o grafo de somadores for desbalanceado. Se o grafo do FIR for uma árvore balanceada, existem mais soluções equivalentes. Como uma solução ótima é conhecida, pode-se avaliar a qualidade do posicionamento. Na implementação da transformada em cossenos, além do grafo completo (FDCT), foram considerados também os grafos para cálculo com componentes verticais (FDCTa) e horizontais (FDCTb).

5.1.2 Topologias

Foi escolhido um conjunto de topologias para serem exploradas. A escolha das topologias de Grid, Hexagonal, Octal é justificada por sua ampla utilização em vários trabalhos relacionados. As topologias N-hop foram incluídas por possibilitarem grande número de conexões com ligações curtas, mas com custo de implementação maior por necessitarem de maior número de fios para interconexões. As topologias 0_N-hop são alternativas de menor custo do que as N-hop mas, mantendo a possibilidade de alcançar células distantes com poucos saltos. Topologias

toroidais, onde as bordas do arranjo são ligadas diretamente às suas bordas opostas também foram avaliadas. Nestas topologias, o roteamento tem mais liberdade para ligar elementos nas bordas do arranjo. Algumas estatísticas foram geradas também para as topologias anteriores com a adição de barramentos. Isto permite que as ligações com maior número de saltos sejam removidas e trocadas por uma ligação de menor custo.

5.2 Ocupação do Arranjo

É interessante avaliar como se comportam os posicionamentos quanto à ocupação dos arranjos de processadores. Este quesito é crucial no mapeamento de grafos grandes, pois quanto maior a taxa de ocupação, menor serão as dimensões do arranjo necessárias para a implementação. A Tabela 4 mostra os resultados obtidos para a ocupação e dimensões do arranjo necessárias para filtros FIR de variados tamanhos. A coluna LxC mostra respectivamente o número de linhas e colunas do arranjo. A coluna *Ocup %* mostra a taxa de ocupação percentual. O posicionamento do algoritmo genético obteve os melhores resultados, conseguindo ocupação ótima nos grafos pequenos e se aproximando dela para grafos maiores. O posicionamento por níveis gera resultados muito baixos para arranjos grandes, utilizando somente 7% das unidades disponíveis no Fir128. As dimensões para o posicionamento do algoritmo genético obtiveram também os melhores resultados, enquanto o posicionamento por níveis chega a necessitar de um arranjo com 133 linhas para realizar o mapeamento.

Tabela 4: Percentual de ocupação dos mapeamentos

Benchmark	Posicionamento					
	Por níveis		Cam. crítico		Alg. genético	
	L×C	Ocup %	L×C	Ocup %	L×C	Ocup %
Fir8	13 × 4	38,4	6 × 6	59,5	5 × 5	100,0
Fir16	21 × 7	29,1	10 × 9	49,0	7 × 7	100,0
Fir64	69 × 22	12,1	17 × 19	56,7	14 × 14	98,5
Fir128	133 × 39	7,2	24 × 24	61,6	20 × 20	96,3

5.3 Exploração do Número de Bordas e Tipo de Conexões

Considere o exemplo de grafo de fluxo de dados da Figura 57 e seu posicionamento e roteamento em um grid 2x2. O grafo tem 4 arcos que foram roteados em dois caminhos de

custo 1 (A,B) e (C,D), mais dois caminhos de custo 2 (A,C) e (B,D). O valor médio é $x = \frac{1+1+2+2}{4} = 1.5$. A coluna M da Tabela 6 mostra o valor do maior caminho. Para o nosso exemplo da Figura 57, o valor máximo é 2.

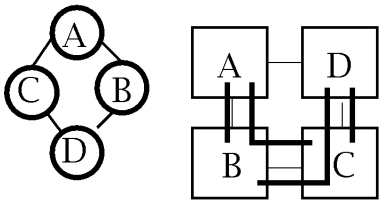


Figura 57: Exemplo de um grafo mapeado em uma grade 2x2

A Tabela 5 mostra resultados do posicionamento por níveis (FERREIRA et al., 2005) para as topologias grid, hexagonal e 1-Hop. Foram gerados mapeamentos para células com duas configurações: 2 entradas e 2 saídas por borda e 2 entradas/saídas por borda.

Os resultados mostram que o algoritmo de roteamento não foi capaz de realizar o seu trabalho em algumas situações onde os *benchmarks* tinham uma maior quantidade de nodos e ligações (FDCTa, FDCTb e FDCT) em algumas topologias (grid e octal), uma das motivações para implementação dos algoritmos de posicionamento dessa dissertação.

Tabela 5: Resultados do mapeamento em profundidade e roteamento (FERREIRA et al., 2005)

EX	N/E	P&R	Grid				Grid 1-hop				Hexagonal			
			2,2,0		0,0,2		2,2,0		0,0,2		2,2,0		0,0,2	
			P	M	P	M	P	M	P	M	P	M	P	M
FIR2	7/7	PR1	**1.28**	**2**	**1.28**	**2**	**1.28**	**2**	**1.28**	2	**1.28**	**2**	**1.28**	**2**
		PR2	1.28	**2**	1.28	**2**	1.28	**2**	1.28	**2**	**1.14**	**2**	**1.14**	**2**
FIR4	13/15	PR1	1.86	4	1.60	3	1.40	2	1.40	**2**	**1.33**	**2**	**1.33**	**2**
		PR2	1.60	3	1.60	3	1.46	**2**	1.46	**2**	**1.26**	**2**	**1.26**	**2**
FIR8	25/31	PR1	1.96	6	2.03	5	1.58	3	1.58	**3**	**1.54**	4	**1.54**	4
		PR2	1.83	5	1.83	5	1.54	3	1.54	**3**	**1.51**	4	**1.51**	4
FIR16	49/63	PR1	2.25	9	2.26	11	1.71	**5**	1.69	**5**	**1.55**	7	**1.55**	7
		PR2	2.19	9	2.15	11	**1.71**	**5**	1.73	**5**	**1.71**	8	**1.71**	8
CPLX4	22/28	PR1	1.71	6	1.75	6	1.46	**3**	1.46	**3**	**1.39**	5	**1.39**	5
		PR2	1.71	6	1.71	6	**1.46**	**3**	**1.46**	**3**	1.50	4	1.50	4
CPLX8	46/60	PR1	2.46	10	2.31	11	**1.73**	**5**	**1.73**	**5**	1.75	7	1.76	7
		PR2	2.13	10	2.21	11	**1.61**	**6**	**1.61**	**6**	1.80	8	1.80	8
FDCTa	92/124	PR1	-	-	2.49	14	**1.83**	**6**	2.09	**7**	2.08	8	2.32	9
		PR2	-	-	2.41	10	1.83	5	**1.96**	10	2.07	10	2.10	9
FDCTb	102/134	PR1	-	-	2.32	14	**1.75**	6	1.94	**7**	2.01	10	2.24	9
		PR2	-	-	2.42	12	1.76	**5**	**1.85**	8	1.97	10	2.02	10
FDCT	136/186	PR1	-	-	-	-	3.19	15	3.31	21	4.61	22	4.31	28
		PR2	-	-	-	-	**2.91**	**13**	**3.01**	**16**	3.71	20	4.04	21

A coluna N/E mostra o número de vértices (N) e arcos (E) do grafo de fluxo de dados. As colunas P e M mostram respectivamente o tamanho médio dos caminhos e o tamanho má-

ximo. As colunas rotuladas com 0,0,2 significam nenhuma conexão de entrada unidirecional, nenhuma conexão de saída unidirecional, e duas conexões de entrada/saída bi-direcionais. As colunas rotuladas com 2,2,0 têm duas conexões de entrada unidirecional, duas conexões de saída unidirecional e nenhuma conexão bi-direcional. As linhas PR1 e PR2 são duas estratégias de posicionamento com e sem inserção de nodos falsos para gerar um maior grau de ajustes nas linhas, para maiores detalhes, consulte (FERREIRA et al., 2005).

5.4 Posicionamento do Caminho Crítico

O algoritmo de posicionamento do caminho crítico foi testado para um conjunto de *benchmarks* e comparado com o posicionamento por níveis com relação ao comprimento médio e máximo das conexões e a compactação das células no arranjo. O tempo de execução do algoritmo é menor que um segundo para qualquer *benchmark*. A Tabela 6 mostra os resultados obtidos em vários filtros Fir. A coluna $\bar{x}$ mostra o valor médio em número de ligações locais. Os resultados com "$-$" significam que o posicionamento gerado não foi adequado para o algoritmo de roteamento que implementa uma estratégia gulosa e, eventualmente, para topologias com mais restrições de conexões locais não consegue rotear todos os sinais. Podemos notar que as topologias Hop, por usarem saltos, reduzem bem o tamanho médio bem como o tamanho máximo.

Tabela 6: Resultados para o posicionamento e roteamento do caminho crítico

Benchmark	Grid		Hex		Octal		1-Hop		0_2-Hop		2Hop	
	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M
Fir8	2,74	9	2,54	8	2,51	9	1,87	5	1,96	5	1,61	4
Fir16	-	-	4,40	19	4,26	20	2,55	6	2,34	6	2,07	5
Fir64	-	-	-	-	-	-	5,14	33	3,98	17	3,47	10
Fir128	-	-	-	-	-	-	-	-	-	-	5,03	17

5.5 Posicionamento com Algoritmo Genético

5.5.1 População Inicial Aleatória

O posicionamento com o algoritmo genético proposto, de acordo com a Tabela 8, mostrou ser bastante eficiente com relação a fazer o mapeamento de maneira compacta, sem deixar de

obter bons resultados no comprimento das conexões. Os testes foram executados utilizando-se um computador equipado com um processador Pentium 4, 2.4 GHz e 512 MB de RAM. O sistema operacional utilizado foi o Debian Linux (kernel 2.6.14) e a máquina virtual Java j2vm 1.5.0_06.

Tabela 7: Escolha dos parâmetros para o algoritmo genético

	Parâmetro	Valor
Configuração	Tamanho da população	50 indivíduos
	Percentual de mutação	5%
	Percentual de renovação	20 %
	Razão entre cruzamento e eliminação	10 ×
Parada	Gerações sem melhoria	60 iterações
	Tempo de Execução	Entre 0.1s e 100s

A configuração dos parâmetros para o algoritmo genético, conforme mostrado na seção 4.4.6, é listada na Tabela 7. Os valores dos parâmetros foram determinados dentro de uma faixa conforme propôs Mitchell (1996) e por testes de execução do sistema. Entretanto, é necessário que mais testes sejam feitos para um ajuste mais fino, pois foi notada uma sensibilidade muito grande da resposta à variação dos parâmetros.

Tabela 8: Resultados do posicionamento com população inicial aleatória

Bench.	#N	#A	D	O	Grid		Hex		Octal		1-Hop		0_2-Hop		2-Hop	
					$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M	$\bar{x}$	M
ADPCM	88	124	10	88.0	2.97	13	2.21	12	1.57	8	1.55	7	1.52	6	**1.36**	**4**
CPLX8	46	60	7	93.9	2.23	10	2.02	7	1.52	6	1.23	**3**	1.38	4	**1.17**	**3**
FDCTa	92	124	10	92.0	3.16	14	2.41	11	2.06	9	1.53	9	1.53	5	**1.40**	**4**
FCDTb	102	134	11	84.3	2.92	17	2.44	12	2.02	10	1.62	8	1.52	**5**	**1.34**	6
FDCT	139	186	12	96.5	3.85	27	3.19	14	2.56	11	1.98	10	1.82	**6**	**1.65**	**6**
FilterRGB	57	70	8	89.1	1.97	10	2.13	9	1.26	4	1.39	8	1.44	3	**1.11**	**2**
Paeth	25	34	5	100.0	1.74	7	1.82	5	1.29	4	1.18	3	1.24	**2**	**1.09**	3
FIR8	25	31	5	100.0	1.77	7	1.27	3	1.20	4	**1.07**	**2**	1.27	3	**1.07**	*2*
FIR16	49	63	7	100.0	2.02	8	2.06	7	1.50	6	1.32	6	1.48	4	**1.18**	**2**
FIR64	193	255	14	98.5	2.47	15	2.49	15	1.85	10	1.51	6	2.18	6	**1.29**	**5**
FIR128	385	511	20	96.3	3.19	32	2.96	11	2.46	10	1.82	9	1.73	**7**	**1.45**	**7**

As colunas *#N* e *#A* indicam o número de nodos e arestas do grafo respectivamente. A coluna *D* demonstra a dimensão do arranjo de processadores. Por exemplo, o valor 7 indica que o arranjo tem tamanho 7×7. A ocupação percentual do arranjo é indicada pela coluna *O*. Os mapeamentos para as arquiteturas de Grid, Hexagonal, Octal, 1-Hop, 0_2-Hop e 2-Hop foram avaliados e os resultados do tamanho médio ($\bar{x}$) e máximo (M) das ligações após o roteamento

estão descritos nas demais colunas da tabela.

Pode-se notar que Grid gera os piores resultados. A topologia hexagonal melhora os resultados, porém o ganho é bem maior para as topologias com 8 conexões, sendo que para estas topologias, os padrões 1-hop e 0-2hop têm um ganho significativo em relação ao Octal. Finalmente, a 2-hop apresenta um custo de conexão local maior, mas gera a maior redução no caminho médio e máximo. Em muitos casos, o tamanho médio é bem próximo do ótimo (1.00).

5.5.2 Visualização do Posicionamento

A Figura 58 mostra graficamente um resultado obtido pelo algoritmo genético para o *benchmark* FDCTa em um arranjo de PE's 10×10.

add_Q1	pas_3	pas_5	mul_Q1F7	sub_g3	copy_S0	sub_q0a	sep_1	sep_2	copy_s0
	sub_F7	pas_2	copy_q0	copy_g3	sub_Q0	shr_F1	sep_0	copy_f_0	
copy_g1	sub_r1		copy_g2		mul_S0F3	out_0	add_F5	shr_s0	add_s0
copy_h1	pas_1	sep_4	mul_S0F5	sub_S0	copy_h0	mul_S1F7	copy_S1	copy_f_3	mul_q0a
add_F0	copy_Q1	pas_4	pas_6	shr_F2	add_g0	add_g1	mul_S1F1	add_S1	sep_3
copy_f_2	copy_p1	shr_F7	copy_r1	sub_g2	mul_Q1F1	copy_g0	add_F1	X	shr_F5
copy_p0	sub_F4	sub_F3	copy_f_6	mul_Q0F5	sep_6		mul_s0a	shr_q0	add_q0
pas_0	mul_Q0F3	copy_Q0		sub_r0	sub_h3	copy_h2	sub_h2	copy_f_4	add_p0
mul_r0F6	mul_r1F6	add_p1	shr_F6	add_h1	copy_f_7	copy_f_1	sep_5	add_h0	copy_h3
sub_F6		copy_r0	mul_r0F2		mul_r1F2	add_F2	copy_f_5	add_s0a	shr_F3

Figura 58: Representação gráfica do posicionamento do *benchmark* FDCTa em um arranjo 10 × 10

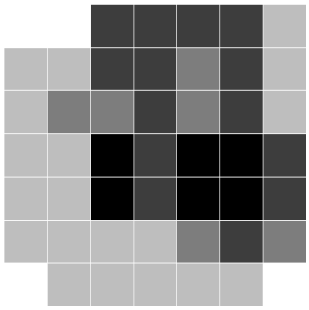


Figura 59: Superfície representando a utilização dos recursos de roteamento para o mapeamento do benchmark FilterRGB

A Figura 59 mostra a utilização de recursos de roteamento em cada PE no arranjo para o posicionamento e roteamento do *benchmark* FilterRGB em um arranjo 8×8 da topologia grid

onde cada PE tem 1 entrada e 1 saída. A utilização varia da mínima (0) para células em branco à máxima (4) para células em preto. É possível notar que em células mais centrais são utilizados mais recursos de roteamento.

5.5.3 Topologias Híbridas: Grade+*CrossBar*

Os *benchmarks* foram executados também para avaliar o posicionamento em diferentes topologias modificadas para incluir, entre as linhas e colunas do arranjo, um ou dois barramentos do tipo *crossbar*. Os caminhos com maior comprimento são retirados do mapeamento e são implementados com a alocação estática de 1 ou 2 barramentos. Os resultados são exibidos na Tabela 9.

A coluna *Tempo* indica o tempo de execução do posicionamento. A escolha dos valores foi feita com base no número de nodos dos grafos dos *benchmarks*.

A coluna *0 bar* demonstra o resultado obtido após o posicionamento sem a utilização de barramentos. As colunas *1 bar* e *2 bar* mostram o resultado com 1 e 2 barramentos entre as linhas e colunas do arranjo.

A utilização dos barramentos tem impacto positivo na redução dos comprimentos das ligações em todas as topologias, embora implique em maior custo para a implementação dos barramentos. Em alguns casos, os resultados obtidos para topologias com menos recursos de roteamento (grid, hexagonal) se aproxima daquelas com mais possibilidade de ligações (4-Hop, 5-Hop). Quando o resultado é próximo de 1, significa que todas as ligações são diretas. Podemos notar que o Octal que tem um padrão de 8 vizinhos diretos, com o recurso extra do cross-bar gera resultados muito bons. Como as ligações não são toroidais, muitas vezes arranjos pequenos não tiram proveito dos saltos 0-N-hop para N=4 ou 5.

Para efeito de comparação, foi elaborado um algoritmo que, partindo da mesma área predeterminada para o posicionamento do algoritmo genético, busca aleatoriamente o melhor posicionamento para os PE's. O tempo de execução deste algoritmo é o mesmo do algoritmo genético e está mostrado na coluna rotulada *Tempo*. A função para avaliação do posicionamento obtido é a mesma função de aptidão de um indivíduo utilizada para o algoritmo genético.

As colunas *Aleatório* e *Aumento %* indicam o resultado obtido após o posicionamento das células para as topologias sem barramentos e o aumento do comprimento das ligações do posi-

Tabela 9: Mapeamento com barramentos *crossbar* de alocação estática

Benchmark	Tempo	Topologia	0 bar.	1 bar.	2 bar.	Aleatório	Aumento %
Fir8	60s	0_1-Hop	**1,22**	**1,00**	**1,00**	1,97	38
		0_2-Hop	1,45	1,12	**1,00**	2,19	34
		0_3-Hop	1,45	1,12	**1,00**	2,23	35
		0_4-Hop	1,54	1,19	1,03	2,74	44
		0_5-Hop	1,87	1,32	1,03	2,68	30
		Grid	1,83	1,29	1,03	3,32	45
		Hex	1,61	1,19	**1,00**	2,35	32
		Octal	1,29	**1,00**	**1,00**	2,16	40
Fir16	120s	0_1-Hop	1,88	1,50	1,22	2,57	27
		0_2-Hop	1,82	1,49	1,25	2,51	27
		0_3-Hop	**1,69**	1,38	1,15	2,73	38
		0_4-Hop	1,82	1,44	1,20	2,89	37
		0_5-Hop	2,04	1,66	1,34	3,40	40
		Grid	2,90	2,15	1,57	4,10	29
		Hex	2,25	1,69	1,30	3,75	40
		Octal	1,74	**1,33**	**1,11**	3,33	48
Cplx8	120s	0_1-Hop	1,95	1,60	1,30	2,65	26
		0_2-Hop	**1,78**	**1,45**	**1,21**	2,57	31
		0_3-Hop	1,86	1,51	1,25	2,65	30
		0_4-Hop	1,91	1,53	1,26	3,27	42
		0_5-Hop	2,16	1,70	1,35	3,32	35
		Grid	2,61	1,85	1,46	4,25	39
		Hex	2,43	1,78	1,38	3,37	28
		Octal	1,98	1,58	1,26	2,8	29
paeth	120s	0_1-Hop	1,35	**1,05**	**1,00**	1,82	26
		0_2-Hop	1,41	1,11	**1,00**	1,97	28
		0_3-Hop	1,44	1,17	1,02	2,18	34
		0_4-Hop	1,82	1,32	1,08	2,65	31
		0_5-Hop	1,73	1,26	**1,00**	2,68	35
		Grid	2,11	1,47	1,11	2,82	25
		Hex	1,47	1,08	**1,00**	2,29	36
		Octal	**1,32**	1,08	**1,00**	2,06	36
filterRGB	180s	0_1-Hop	2,05	1,58	**1,28**	3,01	32
		0_2-Hop	**1,92**	**1,57**	1,30	2,91	34
		0_3-Hop	1,98	1,62	1,32	2,86	31
		0_4-Hop	2,02	1,67	1,41	3,01	33
		0_5-Hop	2,41	1,90	1,57	3,31	27
		Grid	3,25	2,32	1,74	5,11	36
		Hex	2,71	2,05	1,62	4,14	35
		Octal	2,27	1,70	1,34	3,49	35
FDCTb	240s	0_1-Hop	3,14	2,62	**2,21**	3,91	20
		0_2-Hop	3,07	2,63	2,29	3,69	17
		0_3-Hop	**2,90**	**2,55**	2,23	3,38	14
		0_4-Hop	3,20	2,78	2,45	3,7	14
		0_5-Hop	3,20	2,78	2,44	3,93	19
		Grid	5,60	4,50	3,65	6,92	19
		Hex	4,26	3,44	2,91	5,84	27
		Octal	3,38	2,77	2,29	4,80	30
FDCT	300s	0_1-Hop	3,80	3,31	2,87	4,38	13
		0_2-Hop	3,50	3,10	2,78	3,91	11
		0_3-Hop	3,22	2,90	2,64	3,52	9
		0_4-Hop	**3,01**	**2,68**	**2,41**	3,75	20
		0_5-Hop	3,36	3,00	2,72	4,06	17
		Grid	6,31	5,39	4,66	7,92	20
		Hex	5,33	4,60	4,00	6,49	18
		Octal	4,43	3,85	3,34	5,70	22

cionamento puramente aleatório em relação ao genético (coluna *0 bar*).

Os resultados obtidos mostram que, para todos os *benchmarks*, o posicionamento puramente aleatório obtém resultados entre 8% e 48% piores em relação ao posicionamento por

algoritmo genético. Isto comprova que o algoritmo genético consegue convergir mais rapidamente para uma solução em comparação com um posicionamento puramente aleatório.

A Figura 60 exibe um gráfico dos resultados obtidos para o posicionamento do *benchmark* FDCTa em topologia hexagonal em três situações: sem barramentos, com 1 barramento em cada linha e coluna e com 2 barramentos para cada linha e cada coluna. O impacto da utilização dos barramentos nas distâncias entre os vértices mapeados no arranjo é positivo visto que, além da redução do valor médio das distâncias, a maior distância também sofre redução. Mesmo para o mapeamento sem barramentos, a maioria das ligações entre FU's (88) é roteada com distância igual a 1 (a menor possível).

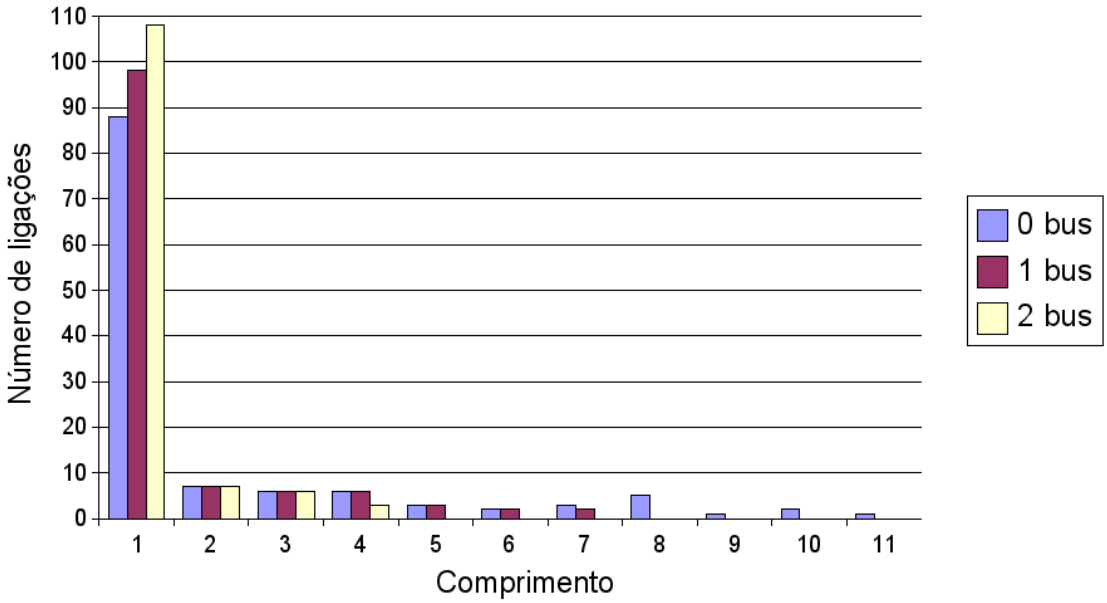


Figura 60: Impacto dos barramentos na distância entre células no *benchmark* FDTCa mapeado em topologia hexagonal

5.5.4 Utilização dos Recursos de Roteamento

Como já demonstrado por (BANSAL et al., 2004b) os resultados obtidos confirmam que a topologia 1-Hop consegue melhores resultados em relação ao grid. Entretanto no trabalho anterior, a topologia 0_2-Hop não foi avaliada. Esta pode ser uma escolha interessante por usar menos recursos de roteamento em comparação à 2-Hop quando vemos que o ganho da segunda foi de apenas 16,3% no comprimento médio e 12,4% no comprimento máximo. A Tabela 10 mostra os resultados médios obtidos após o posicionamento e roteamento dos grafos em seis

topologias. Sendo que a unidade funcional na topologia grid tem 4 vizinhos, na hexagonal tem 6, na octal, 1-Hop e 0-2Hop tem 8, e finalmente o 2-Hop tem 12 vizinhos, e conseqüentemente mais recursos de roteamento e um custo mais elevado. Os resultados mostram o tamanho médio das ligações, o caminho máximo e a comparação na utilização dos recursos de roteamento. Por exemplo, o Grid usa apenas 44,3% dos fios disponíveis. Já as outras topologias como possuem um número bem maior de conexões, a utilização é muito baixa. As duas últimas colunas da Tabela 10 fazem uma comparação relativa ao grid. Por exemplo, o hexagonal usa 77,8% de conexões locais em relação ao grid, mas tem 50% mais recursos (1,5).

Tabela 10: Resultados médios obtidos após o roteamento

Topologia	Média. do comp. médio	Média do maior comp.	Utilização (%) Roteamento	Recursos utilizados (%) em relação ao Grid	Recursos disponíveis (%) em relação ao Grid
Grid	2.57	14.55	44.3	100.0	1.0
Hex	2.25	9.18	26.2	77.8	1.5
Octal	1.75	7.45	14.9	57.7	2.0
1-hop	1.47	6.45	14.0	50.7	1.9
0_2-Hop	1.49	4.64	15.3	52.8	1.7
2-Hop	1.29	4.09	15.5	44.6	2.6

A Tabela 11 mostra os resultados do tamanho médio das conexões e da maior conexão das várias topologias em relação ao Grid. Podemos notar em termos de tamanho médio, o hexagonal reduz 10%, já o Octal reduz de 30%. Entretanto, o 1-hop e 0-2hop com o mesmo número de vizinhos do octal, já tem uma redução no patamar de 40%. Finalmente, o 2-hop que tem 12 vizinhos e uma redução de 50%. Se considerarmos o maior caminho, o 0-2hop passa a ser interessante em relação ao 1-hop pois o salto é de 3 em 3.

Tabela 11: Ganhos relativos da média e maior comprimento das ligações

Topologia	Redução média (%)	Redução máxima (%)
Hex	-10,9	-31,3
Octal	-31,4	-44,3
1-Hop	-41,4	-52,1
0_2-Hop	-39,5	-64,8
2-Hop	-48,3	-69,7

Tabela 12: Resultados com a população inicial melhorada

Topologia	Aleatório				Melhorado			
	0,1s	1s	10s	100s	0,1s	1s	10	100
1-hop	0	13%	28%	38%	38%	42%	45%	48%
1-hop toroidal	0	12%	24%	33%	45%	48%	50%	52%

5.5.5 População Inicial Melhorada

A geração da população inicial foi alterada de forma a obter uma parte dos indivíduos através do caminhamento no grafo de fluxo de dados de maneira diversa com intenção de reduzir o tempo de convergência do algoritmo genético para uma boa solução. A Tabela 12 mostra os resultados obtidos com a geração de uma população inicial com 15 elementos obtidos por caminhamento no grafo e outros 35 obtidos aleatoriamente frente a uma população totalmente aleatória.

Os resultados descrevem o ganho médio obtido no comprimento total das ligações de todos os *benchmarks*. Os tempos para execução foram ajustados em 0,1s, 1s, 10s e 100s e os resultados descrevem o percentual de ganho, normalizado em relação ao menor valor (população inicial aleatória com tempo de 0,1s).

Conforme mostraram os resultados, o tempo de convergência é reduzido em 3 ordens de magnitude: o melhor ganho obtido com população aleatória em 100s é menor em comparação ao pior ganho obtido com população inicial melhorada em 0,1s.

5.5.6 Exploração de Topologias e Ligações Toroidais

Bansal et al. (2004a) mostram que o padrão 1-hop supera a topologia em grid e que são pequenos os ganhos obtidos com 2-hop. Foram feitos experimentos em topologias com conectividade limitada a 8 células vizinhas como octal e 1-hop, sendo este utilizado como referência.

A Tabela 13 (SILVA et al., 2006) mostra a soma de todas as conexões para cada um dos *benchmarks* mapeados em várias topologias, onde a coluna 1-T agrupa os resultados para a topologia 1-hop toroidal e 0-3-T indica a topologia 0_3-hop toroidal. A última linha mostra o ganho médio para todos *benchmarks* comparado à topologia 1-Hop. As topologias 1-hop toroidal e 2-hop obtiveram os melhores resultados. Entretanto, para os núcleos maiores (p. ex.

fir128, fir64), as topologias 0_3-hop toroidal, 0_3-hop e 2-hop foram melhores.

Tabela 13: Comprimento das ligações para as topologias toroidais

Benchmark	Octal	1-hop	1-T	0-3hop	0-3T	2-hop
fir8	39	38	*36*	*36*	41	*36*
fir16	98	92	82	82	95	*76*
fir64	684	490	456	449	448	*385*
fir128	1560	1015	977	890	878	*797*
cplx8	104	95	83	83	93	*80*
paeth	48	43	*38*	*38*	47	38
filterRGB	107	106	94	117	114	*94*
Snn	859	710	606	588	549	*535*
Gouraud	31	29	*28*	*28*	33	*28*
FDCT	546	432	353	408	*362*	367
FDCTa	301	254	*216*	262	233	*216*
Ganho	-18%	0	10%	7%	3%	15%

A Tabela 14 mostra o ganho obtido pelo algoritmo genético na soma das distâncias relativo à topologia 1-hop, considerando somente os grafos com número de nodos em torno de 200 ou maiores: Fir64, Fir64T(fir com estruturas de somadores em árvore), Fir128, Fir128T e SNN, e as topologias: Octal (Oc), Octal 1-hop (Oh), 1-hop (1h), 0_3-hop (3h), 1-hop toroidal (1t), 0_3-hop toroidal (3t), hop híbrido assimétrico (hns) e hop híbrido assimétrico toroidal (tns). O padrão octal tem os piores resultados, sendo que o padrão Octal 1-hop é melhor que o anterior mas ainda assim é pior que o 1-hop. As topologias 0_3-hop e 0_3-hop toroidal obtiveram os melhores resultados. O maior ganho relativo (20%) foi obtido pela topologia assimétrica para o *benchmark* SNN. Os resultados mostram que as topologias assimétricas e 0_3-hop são mais indicadas para grafos grandes organizados em árvore (fir64t, fir128t e SNN). A linha **Ganho** mostra o ganho médio para todos os algoritmos em comparação à topologia 1-hop.

Tabela 14: Exploração das topologias com a inclusão de topologias toroidais

Benchmark	oc	oh	3h	hns	1t	3t	tns
fir128	-64%	-19%	-5%	-4%	0%	-4%	-4%
fir128T	-76%	-16%	12%	9%	4%	14%	9%
fir64	-64%	-20%	-5%	-22%	-4%	-4%	-21%
fir64T	-74%	-17%	10%	12%	3%	11%	11%
SNN	-38%	0	15%	19%	6%	18%	19%
Ganho	-63%	-14%	6%	3%	2%	7%	3%

5.5.7 Mapeamento Heterogêneo

Considerando os *benchmarks* FDCT, FDCTa, Fir65, Fir128 e SNN com 14, 14, 64, 128 e 48 multiplicadores, respectivamente e os arranjos heterogêneos mostrados na Figura 55, a Tabela 15 sumariza o ganho relativo quando comparamos cada topologia com 1-hop. As colunas *oct*, *0-3-h*, *0-3-T* e *T1-hop* indicam, respectivamente, as topologias octal, 0_3-hop, 0_3-hop toroidal e 1-hop toroidal. As linhas rotuladas com *mod5* indicam que o arranjo utilizado tem um multiplicador a cada posição módulo 5, enquanto aquelas rotuladas com *col 0,4,8* têm os multiplicadores fixos nas colunas $0, 4, 8, \ldots$. Os melhores resultados foram conseguidos pela topologia 0_3-hop toroidal seguida pela topologias 0_3-hop e 1-hop toroidal. Embora a topologia 2-hop tenha 12 conexões locais por PE, a topologia 0_3-hop consegue melhores resultados mesmo com 8 conexões locais.

Tabela 15: Arranjos heterogêneos com multiplicadores

Array	Bench	Oct	0-3-H	0-3-T	1-T	2Hop
mod5	FDCT	-17%	6%	19%	15%	11%
	FDCTa	-18%	13%	15%	15%	4%
	Fir64	-12%	14%	27%	19%	11%
	Fir128	-32%	26%	36%	19%	21%
	SNN	-19%	11%	16%	8%	6%
Ganho		-20%	14%	22%	15%	11%
col 0,4,8	FDCT	-20%	7%	23%	23%	3%
	FDCTa	-2%	17%	22%	26%	19%
	Fir64	-19%	7%	18%	14%	9%
	Fir128	-25%	29%	36%	17%	20%
	SNN	-22%	13%	22%	15%	10%
Ganho		-17%	15%	24%	19%	12%

5.5.8 Mapeamento de Vários Grafos no mesmo Arranjo

A reconfiguração em tempo de execução é uma característica essencial de uma arquitetura de grão grosso. O mapeamento deve ser rápido e flexível de modo a caber em uma parte do arranjo como um quadrado, polígono ou polígono descontínuo. Como o arranjo é facilmente escalável, grandes arranjos são aceleradores de hardware promissores e podem simultaneamente implementar várias aplicações. A Figura 61 mostra uma seqüência de mapeamento de vários núcleos. Inicialmente três deles são mapeados, depois o núcleo k_3 é terminado e k_4, k_5 e k_6 são mapeados no espaço liberado. Então os núcleos k_1, k_2 e k_4 são terminados e k_7 é mapeado em um polígono irregular.

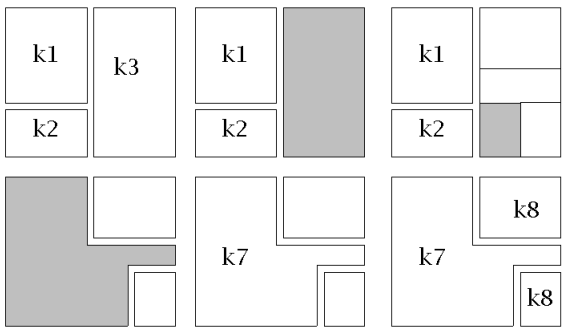


Figura 61: Um exemplo de seqüência de mapeamento em tempo de execução

Considerando a Figura 61 e seguindo a seguinte seqüência de grafos: FDCT. FilterRGB, Fir64, ADPCM, Cplx8, Paeth, SNN e FDCT, a Tabela 16 resume os resultados obtidos com o mapeamento desta seqüência em duas topologias. A coluna K mostra o número de seqüência do núcleo (ou Kernel) alocado, seguido do espaço alocada em linhas$\times$ colunas, depois vem o nome do núcleo, e os resultados do total de conexões usadas e do comprimento da maior conexão para o 1hop e para o 1-hop toroidal. A linha "Non-R"apresenta um polígono irregular e a linha "Non-C"uma alocação discontínua.

Tabela 16: Resultados para seqüência com vários mapeamentos

K	Tam.	bench	1hop	max	1hT	máx.
1	14x8	FDCT	387	8	333	6
2	6x10	FilterRGB	106	7	83	4
3	20x10	Fir64	673	9	582	7
4	10x10	Adpcm	198	6	176	4
5	5x10	Cplx8	88	5	69	3
6	5x5	Paeth	42	3	35	2
7	Non-R	SNN	662	10	543	9
8	Non-C	FDCT	539	10	453	8

6 CONCLUSÃO

6.1 Considerações Finais

As arquiteturas de arranjo de processadores podem ser extensões para microprocessadores eficientes para lidar com a demanda por alto desempenho em sistemas de computação futuros. Considerando que as arquiteturas de grão grosso usam um percentual maior da área em silício para as operações, elas podem ser vistas como soluções mais eficientes do que as arquiteturas reconfiguráveis de grão fino. Embora resultados promissores tenham sido publicados, há a necessidade de se explorar as características principais das arquiteturas apresentadas, e dessa forma, obter dos arranjos orientados por fluxo de dados um desempenho que se aproxime do ótimo.

Foram exibidos os resultados obtidos com a avaliação de um conjunto de arquiteturas de arranjos com diferentes topologias. Estes resultados mostram a capacidade da topologia 0_N-hop em obter menores comprimentos das ligações. Como informado, o ambiente não é restrito e pode ser estendido para avaliar novas topologias como árvores, hipercubos, etc. A estratégia de melhorar a população inicial obteve bons resultados, conseguindo ganhos de até três ordens de magnitude no tempo de convergência, mesmo lidando com grafos ocupando grandes arranjos. Embora novas melhorias estejam planejadas, relacionadas ao mapeamento inicial, à função de aptidão e ao algoritmo de roteamento, os resultados experimentais indicam uma implementação bastante flexível e eficiente. A topologia 0_3-hop se mostra como uma solução com maior escalabilidade para arranjos grandes com recursos homogêneos ou heterogêneos, bem como para o mapeamento dinâmico de vários núcleos. Levando em conta que avanços na tecnologia levam à criação de arquiteturas de grão grosso de grande capacidade, ferramentas de exploração do espaço de projeto têm crucial importância.

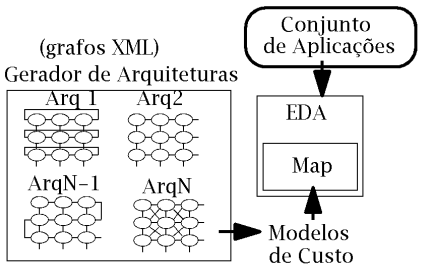


Figura 62: Exploração de arquiteturas em alto nível

6.2 Contribuições

A contribuição principal do trabalho foi a implementação de um algoritmo genético para o mapeamento de unidades funcionais, flexível o bastante para ser expandido para novas topologias e *benchmarks*. O mapeamento inicial para a população do algoritmo genético demonstrou ser capaz de reduzir significativamente o tempo de busca de uma solução satisfatória para o posicionamento. A flexibilidade do algoritmo permitiu sua extensão para mapeamento de arranjos com recursos heterogêneos. Arranjos grandes e compartilhados com vários núcleos de algoritmos também são mapeados pelo algoritmo, permitindo inclusive re-mapeamento de grafos. Os resultados obtidos comprovam que, apesar da complexidade computacional do problema de posicionamento ótimo ser proibitiva, pois é um conhecido problema NP-Completo, o uso da heurística implementada teve bom comportamento, gerando boas soluções em tempo aceitável.

Como contribuição adicional, foram avaliadas várias topologias e variações delas, como as topologias Hop, toroidais e não simétricas com relação ao posicionamento e roteamento de células. Isto possibilitou verificar a aplicabilidade de cada uma delas em várias situações.

6.3 Trabalhos Futuros

A inclusão de exploração automática de arquiteturas pode permitir a geração de topologias de roteamento para certos tipos de aplicação, como mostra a Figura 62. A busca de modelos analíticos associados às topologias pode ser um importante auxílio para guiar os passos de exploração das arquiteturas. Outras características de exploração devem ser adicionadas ao ambiente como estimativa de dissipação de potência e elementos de processamento heterogêneos na simulação. A definição do formato da configuração para as células e sua geração automática

em VHDL permitirá avaliar em FPGA as soluções obtidas. A longo prazo, a construção de um *front end* para o ambiente facilitará o estudo das diversas características de mapeamento em aplicações complexas.

A seguir, são apresentadas sugestões pontuais para melhoria do ambiente EDA sob os aspectos posicionamento e roteamento.

Sugestões para o posicionamento:

- Fazer uma análise mais detalhada dos parâmetros do algoritmo genético (taxa de cruzamento, taxa de mutação, etc.) e de seus critérios de parada, buscando valores que possam reduzir o tempo de convergência.
- Implementar e testar novas funções de aptidão que melhorem o desempenho do algoritmo genético.
- Utilizar técnicas de Otimização e Inteligência Artificial, possivelmente até mesmo um algoritmo genético, para determinar automaticamente qual topologia melhor se adapta a um determinado *benchmark*.
- Implementar o algoritmo de *simulated annealing* no ambiente EDA, utilizando uma estratégia similar à do algoritmo genético, para manter o posicionamento dentro de certos limites no arranjo e verificar seu desempenho em relação aos demais.

Sugestões para o roteamento:

- Implementar outros algoritmos de roteamento como o *Maze Router* e *PathFinder* e avaliar o seu comportamento frente aos já existentes no ambiente EDA.
- Utilizar informações do roteamento, por exemplo, as posições de células com recursos de roteamento esgotados. Isto seria útil para resolver dificuldades de se rotear uma determinadas ligações, alterando o posicionamento das células envolvidas.

REFERÊNCIAS BIBLIOGRÁFICAS

AHN, M. et al. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: *DATE'06 - Design, Automation and Test in Europe Conference & Exhibition*. Munich, Germany: European Design and Automation Association, 2006.

ALLAN, A. et al. 2001 technology roadmap for semiconductors. *IEEE Computer*, v. 35, n. 1, p. 42–53, Jan 2002.

BANSAL, N. et al. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In: *Design, Automation and Test in Europe Conference*. Washinghton DC, USA: IEEE Computer Society, 2004. p. 474–479.

BANSAL, N. et al. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In: *Design, Automation and Test in Europe Conference*. Washinghton DC, USA: IEEE Computer Society, 2004. p. 474–479.

BANSAL, N. et al. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In: *Design, Automation and Test in Europe Conference*. Washinghton DC, USA: IEEE Computer Society, 2004. p. 474–479.

BENINI, L.; MICHELI, G. D. Networks on chips: A new soc paradigm. *IEEE Computer*, v. 35, n. 1, p. 70–78, Jan 2002.

BITNER, R. *Wormhole Run-time Reconfiguration: Conceptualization and VLSI design of a High Performance Computing System*. Tese (Doutorado) — Virginia Polytechnic Institute and State University, 1996.

BJERREGAARD, T.; MAHADEVAN, S. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, v. 38, n. 1, 2006.

BOSSART, T. C. *Generalized Measures for Reconfigurable Hexagonal Systolic Arrays*. [S.l.], 1994.

BOSSUET, L.; GOGNIAT, G.; PHILIPPE, J. Fast Design Space Exploration Method for Reconfigurable Architectures. In: *The International Conference on Engineering of Reconfigurable Systems and Algorithm, ERSA'03*. Las Vegas, Nevada, USA: [s.n.], 2003.

BOSSUET, L.; GOGNIAT, G.; PHILIPPE, J. L. Generic design space exploration for reconfigurable architectures. In: *IPDPS*. Denver, Colorado, USA: IEEE Computer Society, 2005.

BUDIU, M.; GOLDSTEIN, S. C. Compiling Application-Specific Hardware. In: SPRINGER-VERLAG (Ed.). *Proceedings 12th Int'l Conference on Field Programmable Logic and Applications (FPL'02)*. Montpelier, France: s.n., 2002. LNCS 2438, p. 853–863.

CARDOSO, J. M. P. Dynamic Loop Pipelining in Data-Driven Architectures. In: *ACM Int'l Conference on Computing Frontiers (CF'05)*. Ischia, Italy: ACM Press, 2005.

CARDOSO, J. M. P.; WEINHARDT, M. Xpp-vc: A c compiler with temporal partitioning for the pact-xpp architecture. In: *International Conference on Field Programmable Logic and Applications*. Montpelier, France: Springer Verlag, 2002. p. 864–878.

CHEMIJ, W. *Parallel Computer Taxonomy*. Dissertação (Mestrado) — Aberystwyth University, 1994.

COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, v. 34, n. 2, p. 171–210, Jun. 2002.

COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, v. 34, n. 2, p. 171–210, Jun. 2002.

CONG, J.; XU, S. Synthesis challenges for nextgeneration high-performance and high-density plds. In: *Asia and South Pacific Design Automation Conf.* Yokohama, Japan: ACM Press, 2000. p. 157–162.

CRONQUIST, D. C. *Simultaneous Place and Route for Wire-Constrained FPGAs*. [S.l.], 1995. Disponível em: <citeseer.ist.psu.edu/cronquist95simultaneous.html>.

DEHON, A.; WAWRZYNEK, J. Reconfigurable computing: What, why, and implications for design automation. In: *DAC - ACM/IEEE Design Automation Conference*. New Orleans, USA: ACM Press, 1999. p. 610–615.

DUNCAN, R. A survey of parallel computer architetures. *ACM Computer*, v. 23, n. 2, p. 365–396, Feb. 1990.

FERREIRA, R. et al. Data-driven regular arrays: Design space exploration and mapping. In: SPRINGER-VERLAG (Ed.). *Int'l Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS V)*. Samos, Greece: Springer-Verlag, 2005. LNCS 3553, p. pp. 41–50.

FERREIRA, R.; CARDOSO, J. M. P.; NETO, H. C. An Environment for Exploring Data-driven Architectures. In: SPRINGER-VERLAG, L. (Ed.). *14th International Conference on Field Programmable Logic and Applications*. Antwerp, Belgium: Springer-Verlag, 2004. p. 1022–1026.

GALANIS, M. D.; DIMITROULAKOS, G.; GOUTIS, C. E. Speedups from partitioning critical software parts to coarse-grain reconfigurable hardware. In: *ASAP*. Washington DC, USA: IEEE Computer Society, 2005. p. 50–59. ISBN 0-7695-2407-9.

GARCIA, A.; FERREIRA, R. S. Coarse-grained reconfigurable architectures. In: *Microeletronic Students Forum - SForum*. Ouro Preto, Brazil: s.n., 2006.

GUDISE, V. G.; VENAYAGAMOORTHY, G. K. Fpga placement and routing using particle swarm optimization. In: *IEEE Computer society Annual Symposium on VLSI*. Lafayette, LA, USA: IEEE Computer Society, 2004. p. 307–308.

HARTENSTEIN, R. A Decade of Reconfigurable Computing: a Visionary Retrospective. In: *Int'l Conf. on Design, Automation and Test in Europe (DATE'01)*. Munich, Germany: IEEE Press, 2001. pp. 642–649.

HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In: *Proceedings of the conference on Design, automation and test in Europe*. Munich Germany: IEEE Press, 2001. p. 642–649.

HARTENSTEIN, R. et al. Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array. In: *Proceedings of the 2000 conference on Asia South Pacific design automation*. Yokohama, Japan: ACM Press, 2000. p. 163–168. ISBN 0-7803-5974-7.

HARTENSTEIN, R.; KRESS, R.; HEINIG, H. A dynamically reconfigurable wavefront array architecture. In: *Proceedings International Conference on Application Specific Array Processors*. San Francisco, USA: IEEE Press, 1994. p. 404–414.

HAUCK, S.; BORRIELLO, G.; EBELING, C. Mesh routing topologies for multi-fpga systems. *IEEE Transactions on VLSI Systems*, v. 6, n. 3, p. 400–408, Sep. 1998.

HENDRICH, N. A Java-based Framework for Simulation and Teaching. In: PUBLISHERS, K. A. (Ed.). *3rd European Workshop on Microelectronics Education (EWME'00)*. Aix en Provence, France: [s.n.], 2000. p. pp. 285–288.

HOLZMANN, G. J. *Design and Validation of Computer Protocols*. [S.l.]: Prentice Hall, 1991.

IORIO, V. O. D. et al. *Implementação em Java de um algoritmo genético*. Dez. 2004. DPI/UFV em convênio com CPqD.

KAHNE, B. C. *A Genetic Algorithm-Based Place-and-Route Compiler For A Run-time Reconfigurable Computing System*. Dissertação (Mestrado) — Virginia Polytechnic Institute and State University, 1997.

KAPASI, U. J. et al. Programmable stream processors. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 8, p. 54–62, 2003. ISSN 0018-9162.

KEUTZER, K.; MALIK, S.; NEWTON, A. R. From asic to asip: The next design discontinuity. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. Freiburg, Germany: IEEE Press, 2002. p. 84–90.

KOREN, I. et al. A data-driven vlsi array for arbritrary algorithms. *IEEE Computer*, v. 20, n. 10, p. 30–43, Oct. 1988.

LAI, Y.-T.; LAI, H.-Y.; YEH, C.-N. Placement for the reconfigurable datapath architecture. In: *IEEE International Symposium on Circuits and Systems, ISCAS*. Kobe, Japan: IEEE Press, 2005. v. 2, p. 1875 – 1878.

MA JOHN KNIGHT, C. P. F. Physical resource binding for coarse-grain reconfigurable array using evolutionary algorithms. *IEEE Transctions on Very Large Integration (VLSI)*, v. 13, n. 5, May 2005.

MCMURCHIE, L.; EBELING, C. *PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs*. [S.l.]: ACM Press, Feb. 1997. 111-117 p.

MCMURCHIE, L.; EBELING, C. *PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs*. [S.l.]: ACM Press, 2001. 29-36 p.

MEI, B. et al. Architecture exploration for a reconfigurable architecture template. *IEEE Design and Test of Computers*, v. 22, n. 2, p. 90–101, Mar/Apr 2005.

MICHELI, G. de. *Synthesis and optimization of digital circuits*. New York: McGraw-Hill, 1994.

MITCHELL, M. *An Introduction to Genetic Algorithms*. Cambridge, Massachusetts: MIT Press, 1996.

NAGELDINGER, U. *Coarse-Grained Reconfigurable Architecture Design Space Exploration*. Tese (Doutorado) — University of Kaiserslautern, 2001.

PACT-XPP-TECHNOLOGIES. The xpp white paper: A technical perspective. Release 2.1,http://www.pactcorp.com/. March 2002.

PATTERSON, D.; HENNESSY, J. *Computer Architecture: A Quantitative Approach*. 2nd. ed. [S.l.]: Morgan Kaufmann Publishers, 1996.

PATTERSON, D. A. Computer science education in the 21st century. *Communications of the ACM*, ACM Press, New York, NY, USA, v. 49, n. 3, p. 27–30, 2006. ISSN 0001-0782.

RABAEY, J. Reconfigurable processing: The solution to low-power programmable dsp. In: *IEEE International Conference on Acoustics, Speech, and Signal Processing*. [S.l.]: IEEE Computer Society, 1997. p. 275.

ROBIC, B.; SILC, J. High-performance computing on a honeycomb architecture. *Lecture Notes in Computer Science*, v. 734, p. 1–??, 1993. ISSN 0302-9743.

ROBIC, B.; SILC, J. *Mapping irregular parallel algorithms in VLSI processor arrays*. [S.l.], fev. 15 1994. Sat, 22 Jun 1996 10:22:05 GMT. Disponível em: <ftp://ftp-csd.ijs.si/Reports/CSD-TR-94-1.ps.gz>.

ROBIC, B.; SILC, J. Using parallel simulated annealing in the mapping problem. In: HALATSIS, C. et al. (Ed.). *PARLE*. [S.l.]: Springer, 1994. (Lecture Notes in Computer Science, v. 817), p. 797–800. ISBN 3-540-58184-7.

ROBIC, B.; VILFAN, B. *Improved schemes for mapping arbitrary algorithms onto processor meshes*. [S.l.], mar. 15 1995. Sat, 22 Jun 1996 10:22:05 GMT. Disponível em: <ftp://ftp-csd.ijs.si/Reports/CSD-TR-95-3.ps.gz>.

ROBIC, B.; VILFAN, B. *Improved schemes for mapping arbitrary algorithms onto processor meshes*. [S.l.]: Computer Systems Department - Institut Jozef Stefan, Mar. 1995. Technical Report CSD-95-3.

RUSSELL, S.; NORVIG, P. *Inteligência Artificial trad. da segunda edição*. Rio de Janeiro: Elsevier, 2004.

SAWITZKI, S.; GRATZ, A.; SPALLEK, R. G. Increasing microprocessor performance with tightly-coupled reconfigurable logic arrays. In: HARTENSTEIN, R. W.; KEEVALLIK, A. (Ed.). *Field-Programmable Logic: From FPGAs to Computing Paradigm*. Springer-Verlag, Berlin, 1998. p. 411–415. Disponível em: <citeseer.ist.psu.edu/sawitzki98increasing.html>.

SILVA, M. V. et al. Mesh mapping exploration for coarse-grained reconfigurable array architectures. In: *International Conference on Reconfigurable Computing and FPGA's*. San Luis Potosi, Mexico: IEEE Computer Society Press, 2006. p. To appear.

SINGH, H. et al. Morphosys: An integrated re-configurable architecture. In: *Proceedings of the NATO Symposium on System Concepts and Integration*. Hsinchu, Taiwan, China: ACM Press, 1998.

SWANSON, S. et al. Wavescalar. In: *36th International Symposium on Microarchitecture*. San Diego, USA: IEEE Press, 2003.

TAGHAVI, T. et al. Innovate or perish: Fpga physical design. In: *ISPD - International Symposium on Physical Design* . Phoenix, Arizona, USA: IEEE Press, 2004.

TAYLOR, M. B. et al. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In: *ISCA - International Symposium on Computer Architecture*. Munich, Germany: IEEE Computer Society, 2004. p. 2–13. ISBN 0-7695-2143-6.

TESSIER, R.; BURLESON, W. *Reconfigurable computing for digital processing: A survey*. 2000.

THAM, K. S.; MASKELL, D. L. Software-oriented system-level simulation for design space exploration of reconfigurable architectures. In: *Asia-Pacific Computer Systems Architecture Conference*. Singapore: Springer-Verlag, 2005. p. 391–404.

TOLEDO, A.; FERREIRA, R. dos S. Data-driven mapping on a reconfigurable array. In: *Student Forum 2004 CHIP*. Porto de Galinhas, PE, Brazil.: s.n., 2004.

VEEN, A. Dataflow Machine Archirecture. *ACM Computing Surveys*, v. 18, p. 365–396, 1986.

VENN, A. H. Dataflow machine architecture. *ACM Computing Surveys*, v. 18, n. 4, p. 5–16, Dec. 1986.

WAN, M. et al. A low-power data-driven dsp system. In: *SIPS - IEEE Workshop on Signal Processing Systems, Design and Implementation*. Taipei, Taiwan: IEEE Press, 1999. p. 191–200.

ZABEL, M. et al. Design space exploration of coarse-grain reconfigurable dsps. In: *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2005)*. San Luis Potos, Mexico: IEEE Press, 2005.

ANEXO A – CÓDIGOS DOS *BENCHMARKS*

A.1 Gouraud

```
/*
*       USAGE This routine is C callable and can be called as
*
*             void gouraud(unsigned int n, unsigned int rd, unsigned int r,
*                          unsigned int gd, unsigned int g, unsigned int bd,
*                          unsigned int b, int p[])
*
*             n   --- half of the pixels on a line              (input)
*             rd  --- increment/decrement of the red color      (input)
*             r   --- red color intensity                       (input)
*             gd  --- increment/decrement of the green color    (input)
*             g   --- green color intensity                     (input)
*             bd  --- increment/decrement of the blue color     (input)
*             b   --- blue color intensity                      (input)
*             p[] --- array of pixel's color intensity          (output)
*
*             If the routine is not to be used as a C callable function,
*             then all instructions relating to stack should be removed.
*             Refer to comments of individual instructions. You will also
*             need to initialize values for all the values passed as these
*             are assumed to be in registers as defined by the calling
*             convention of the compiler, (refer to the C compiler reference
*             guide.)
*
*       C CODE
*             This is the C equivalent of the Assembly Code without the
*             assumptions listed below. Note that the assembly code is hand
*             optimized and assumptions apply.
*
*             void gouraud(unsigned int n, unsigned int rd, unsigned int r,
*                          unsigned int gd, unsigned int g, unsigned int bd,
*                          int b, int p[])
*             {
*                 unsigned int    mask =3D 0xF800F800, i;
*                 for (i =3D 0; i < n; i++) {
*                   r +=3D rd;
*                   g +=3D gd;
*                   b +=3D bd;
*                   p[i] =3D (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
*                 }
*             }
*
*/

#include "XPP.h"

#define TP

#define N 200

// array of pixel's color intensity          (output)
```

```
int p[N];
#pragma extern p 1

unsigned int n;
unsigned int rd;
unsigned int r;
unsigned int gd;
unsigned int g;
unsigned int bd;
int b;

main() {
  /*void gouraud(unsigned int n, unsigned int rd, unsigned int r,
                         unsigned int gd, unsigned int g, unsigned int bd,
                         int b, int p[])
  */
  unsigned int    mask = 0xF800F800, i;
  for (i = 0; i < N; i++) {
    r += rd;
    g += gd;
    b += bd;
    p[i] = (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
  }
}
```

A.2 Fir CPLX

```
/* ========================================================================= */
/*  TEXAS INSTRUMENTS, INC.                                                  */
/*                                                                           */
/*  NAME                                                                     */
/*      DSP_fir_cplx_d.c -- Example file for DSP_fir_cplx                    */
/*                                                                           */
/*  USAGE                                                                    */
/*      This file contains code for demonstrating the C64x DSPLIB            */
/*      DSP_fir_cplx function. All inputs to the function contain random     */
/*      data. The assembly version of DSP_fir_cplx is called and the         */
/*      output is compared with the reference C code using mem_compare()     */
/*      in support.c.                                                        */
/* ------------------------------------------------------------------------- */
/*          Copyright (C) 2003 Texas Instruments, Incorporated.              */
/*                          All Rights Reserved.                             */
/* ========================================================================= */

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

/* Header file for the memory compare function */
int mem_compare(const void *ptr1, const char *name1,
                const void *ptr2, const char *name2, int len);

/* Header file for the DSPLIB function */
#include "DSP_fir_cplx.h"

/* Header file for the C function */
void DSP_fir_cplx_c(const short * x, const short * h, short * r, int nh, int nr);


/* ========================================================================= */
/*  Constant dataset.                                                        */
/* ========================================================================= */
#define NH   (10)
#define NH2  (20)
#define NR   (20)
#define NR2  (40)
#define NX   (58)
```

```
/* ======================================================================== */
/*  Initialize arrays with random test data.                                */
/* ======================================================================== */
int     test = 0;
const short  x[NX] =
{
     0x2D98, -0x074B,  0x68EE, -0x2D98,  0x59B7, -0x247B, -0x4333,  0x1DBB,
    -0x63AA,  0x1C1F, -0x005E,  0x08AA,  0x6321, -0x1FBC, -0x4417,  0x7B0D,
    -0x1CEA, -0x2208, -0x30DF, -0x2FF6, -0x470D, -0x350C,  0x5319, -0x3553,
    -0x7996,  0x0A3D, -0x3B26,  0x6A74, -0x6322, -0x564E,  0x47DF, -0x6259,
    -0x4E4C, -0x1FE0, -0x5DC6, -0x0865,  0x7385,  0x34AF,  0x7A8F, -0x63F3,
    -0x546E,  0x1891, -0x1E99,  0x200B, -0x25F1, -0x66B9,  0x63A3, -0x26DE,
     0x5033, -0x0F78,  0x56F7, -0x51C7,  0x6812,  0x054A, -0x1AAD,  0x0243,
     0x37F1,  0x37EB
};

const short  h[NH2] =
{
     0x6D1B,  0x18A4,  0x489F, -0x1B14,  0x3167,  0x53DF, -0x5F2B, -0x647F,
     0x161B, -0x3C81, -0x5840, -0x2A70,  0x51B0,  0x0282, -0x68CC, -0x667C,
     0x3317,  0x496E,  0x2895, -0x414F
};

short  r_asm[NR2];
short  r_c[NR2];



int main()
{

/* ======================================================================== */
/*  Call hand-coded assembly version (located in DSP64x.lib library         */
/*  archive - see Project -> Build Options -> Linker -> Include Libraries)  */
/*  Note that x should point to x[2*(NH-1)].                                */
/* ======================================================================== */
    DSP_fir_cplx(x+2*(NH-1), h, r_asm, NH, NR);

/* ======================================================================== */
/*  Call natural-C version                                                  */
/*  Note that x should point to x[2*[NH-1)].                                */
/* ======================================================================== */
    DSP_fir_cplx_c(x+2*(NH-1), h, r_c, NH, NR);

/* ======================================================================== */
/*  Compare outputs using mem_compare() in support.c.  If different,        */
/*  mem_compare() displays location of failures.                            */
/* ======================================================================== */
    test = mem_compare(r_asm, "r_asm", r_c, "r_c", sizeof(r_c));
    if (test == 0)
    {
      printf("r_asm vs. r_c");
      printf("\nVerification successful.\n");
    }

}

/* ======================================================================== */
/*  End of file:  dsp_fir_cplx_d.c                                          */
/* ------------------------------------------------------------------------ */
/*          Copyright (C) 2003 Texas Instruments, Incorporated.             */
/*                          All Rights Reserved.                            */
/* ======================================================================== */
```

A.3 FDCT

```
/*
  This is the 8x8 DCT (Discrete Cosine Transform) algorithm.
```

```
  Source: ftp://ftp.ti.com/pub/tms320bbs/c62xfiles/dct.asm

  DSP IMPLEMENTATION
  Cycles = 48 + 160 * num_fdcts
  for num_fdcts >= 1
  208 cycles or 1.04 microseconds for one 8x8
  Block of Data


  MAIN differences:
  - pointer-free version
  - added two auxiliar arrays:
  dct_io_ptr -> |ROW| -> dct_io_tmp -> |COL| -> dct_o

  An improvement can be done in order to use always the array dct_io_tmp of size 64
  and so mapped to internal memories (faster accesses).
  However, such improvement needs that the image are traversed with an outer loop
  which is not so good since the DCT does not fit entirely in a realistic XPP.

 */

//#define SW

#ifndef SW

#include "XPP.h"
#define TP

#else

#include <sys/types.h>
#include <time.h>

#endif

#define N 8 // fixed
#define M N*N

#define num_fdcts 4 // 5400 NTSC: 720x480 => 90 x 60 blocks of 8x8; if 1 then 8x8 DCT

#define SIZE num_fdcts*M

short    dct_io_ptr[SIZE];
#ifdef EXT_ALLOC
#pragma extern dct_io_ptr 1 // external memory #2
#endif

short    dct_io_tmp[SIZE];
#ifdef EXT_ALLOC
#pragma extern dct_io_tmp 3 // external memory #2
#endif

short    dct_o[SIZE];
#ifdef EXT_ALLOC
#pragma extern dct_o 2 // external memory #2
#endif


main() {
  int tmp;

  //void fdct_8x8(short dct_data, unsigned num_fdcts)

  /* ------------------------------------------------------- /
     /  Set up the cosine coefficients c0..c7.                /
     / ------------------------------------------------------- */
  const unsigned short c1 = 0x2C62, c3 = 0x25A0;
  const unsigned short c5 = 0x1924, c7 = 0x08D4;
  const unsigned short c0 = 0xB505, c2 = 0x29CF;
  const unsigned short c6 = 0x1151;

  /* ------------------------------------------------------- /
```

```
    /  Intermediate calculations.                          /
    / ------------------------------------------------------ */
  short f0, f1, f2, f3,f4, f5, f6, f7;       // Spatial domain samples.
  int   g0, g1, h0, h1,p0, p1;               // Even-half intermediate.
  short r0, r1;               // Even-half intermediate.
  int   P0, P1, R0, R1;       // Even-half intermediate.
  short g2, g3, h2, h3;       // Odd-half intermediate.
  short q0a,s0a,q0, q1,s0, s1;               // Odd-half intermediate.
  short Q0, Q1, S0, S1;       // Odd-half intermediate.
  int   F0, F1, F2, F3, F4, F5, F6, F7;       // Freq. domain results.
  int   F0r,F1r,F2r,F3r,F4r,F5r,F6r,F7r;      // Rounded, truncated results.

  /* ------------------------------------------------------- /
    /  Input and output pointers, loop control.             /
    / ------------------------------------------------------ */
  unsigned i, j, i_1;

#ifdef SW

  time_t  t0, t1; /* time_t is defined on <time.h> and <sys/types.h> as long */
  clock_t c01, c11; /* clock_t is defined on <time.h> and <sys/types.h> as int */

  for (i = 0; i < num_fdcts*M; i++) {
    dct_io_ptr[i] = i;
  }

  printf ("using UNIX function time to measure wallclock time ... \n");
  printf ("using UNIX function clock to measure CPU time ... \n");


  t0 = time(NULL);
  c01 = clock();

  printf ("\tbegin (wall):               %ld\n", (long) t0);
  printf ("\tbegin (CPU):                %d\n", (int) c01);

#endif

  /* ------------------------------------------------------- /
    /  Outer vertical loop -- Process each 8x8 block.       /
    / ------------------------------------------------------ */
  //dct_io_ptr = dct_data;
  i_1 = 0;
  for (i = 0; i < num_fdcts; i++) {
    /* ---------------------------------------------------- /
      /  Perform Vert 1-D FDCT on columns within each block. /
      / ---------------------------------------------------- */
    for (j = 0; j < N; j++) {
      /* ------------------------------------------------ /
 /  Load the spatial-domain samples.               /
 / ------------------------------------------------ */
      f0 = dct_io_ptr[ 0+i_1];
      f1 = dct_io_ptr[ 8+i_1];
      f2 = dct_io_ptr[16+i_1];
      f3 = dct_io_ptr[24+i_1];
      f4 = dct_io_ptr[32+i_1];
      f5 = dct_io_ptr[40+i_1];
      f6 = dct_io_ptr[48+i_1];
      f7 = dct_io_ptr[56+i_1];

      /* ------------------------------------------------ /
 /  Stage 1:  Separate into even and odd halves.   /
 / ------------------------------------------------ */
      g0 = f0 + f7;               h2 = f0 - f7;
      g1 = f1 + f6;               h3 = f1 - f6;
      h1 = f2 + f5;               g3 = f2 - f5;
      h0 = f3 + f4;               g2 = f3 - f4;

      /* ------------------------------------------------ /
 /  Stage 2                                       /
 / ------------------------------------------------ */
      p0 = g0 + h0;               r0 = g0 - h0;
      p1 = g1 + h1;               r1 = g1 - h1;
```

```
      q1 = g2;                    s1 = h2;

      s0a= h3 + g3;               q0a= h3 - g3;
      s0 = (s0a * c0 + 0x7FFF) >> 16;
      q0 = (q0a * c0 + 0x7FFF) >> 16;

      /* ---------------------------------------------- /
 /  Stage 3                                          /
 / ---------------------------------------------- */
      P0 = p0 + p1;             P1 = p0 - p1;
      R1 = c6 * r1 + c2 * r0;     R0 = c6 * r0 - c2 * r1;

      Q1 = q1 + q0;               Q0 = q1 - q0;
      S1 = s1 + s0;               S0 = s1 - s0;

      /* ---------------------------------------------- /
 /  Stage 4                                          /
 / ---------------------------------------------- */
      F0 = P0;                    F4 = P1;
      F2 = R1;                    F6 = R0;

      F1 = c7 * Q1 + c1 * S1;     F7 = c7 * S1 - c1 * Q1;
      F5 = c3 * Q0 + c5 * S0;     F3 = c3 * S0 - c5 * Q0;

      /* ---------------------------------------------- /
 /  Store the frequency domain results.              /
 / ---------------------------------------------- */
      dct_io_tmp[ 0+i_1] = F0;
      dct_io_tmp[ 8+i_1] = F1 >> 13;
      dct_io_tmp[16+i_1] = F2 >> 13;
      dct_io_tmp[24+i_1] = F3 >> 13;
      dct_io_tmp[32+i_1] = F4;
      dct_io_tmp[40+i_1] = F5 >> 13;
      dct_io_tmp[48+i_1] = F6 >> 13;
      dct_io_tmp[56+i_1] = F7 >> 13;

      //dct_io_ptr++;
      i_1++;
    }
    /* ---------------------------------------------------- /
       /  Update pointer to next 8x8 FDCT block.              /
       / ---------------------------------------------------- */
    //dct_io_ptr += 56;
    i_1 += 56;
  }

#ifdef TP
  XPP_next_conf();
#endif

  /* ---------------------------------------------------- /
     /  Perform Horizontal 1-D FDCT on each 8x8 block.          /
     / ---------------------------------------------------- */
  //dct_io_ptr = dct_data;
  i_1 = 0;
  for (i = 0; i < N*num_fdcts; i++) {
    // ---------------------------------------------- /
    //  Load the spatial-domain samples.              /
    // ---------------------------------------------- /
    f0 = dct_io_tmp[0+i_1];
    f1 = dct_io_tmp[1+i_1];
    f2 = dct_io_tmp[2+i_1];
    f3 = dct_io_tmp[3+i_1];
    f4 = dct_io_tmp[4+i_1];
    f5 = dct_io_tmp[5+i_1];
    f6 = dct_io_tmp[6+i_1];
    f7 = dct_io_tmp[7+i_1];

    // ---------------------------------------------- /
    //  Stage 1:  Separate into even and odd halves.  /
    // ---------------------------------------------- /
    g0 = f0 + f7;               h2 = f0 - f7;
    g1 = f1 + f6;               h3 = f1 - f6;
```

```
    h1 = f2 + f5;               g3 = f2 - f5;
    h0 = f3 + f4;               g2 = f3 - f4;

    // ---------------------------------------------------- /
    //  Stage 2                                             /
    // ---------------------------------------------------- /
    p0 = g0 + h0;               r0 = g0 - h0;
    p1 = g1 + h1;               r1 = g1 - h1;
    q1 = g2;                    s1 = h2;

    s0a= h3 + g3;               q0a= h3 - g3;
    q0 = (q0a * c0 + 0x7FFF) >> 16;
    s0 = (s0a * c0 + 0x7FFF) >> 16;

    // ---------------------------------------------------- /
    //  Stage 3                                             /
    // ---------------------------------------------------- /
    P0 = p0 + p1;               P1 = p0 - p1;
    R1 = c6 * r1 + c2 * r0;     R0 = c6 * r0 - c2 * r1;

    Q1 = q1 + q0;               Q0 = q1 - q0;
    S1 = s1 + s0;               S0 = s1 - s0;

    // ---------------------------------------------------- /
    //  Stage 4                                             /
    // ---------------------------------------------------- /
    F0 = P0;                    F4 = P1;
    F2 = R1;                    F6 = R0;

    F1 = c7 * Q1 + c1 * S1;     F7 = c7 * S1 - c1 * Q1;
    F5 = c3 * Q0 + c5 * S0;     F3 = c3 * S0 - c5 * Q0;

    // ---------------------------------------------------- /
    //  Round and truncate values.                          /
    //                                                      /
    //  Note: F0 and F4 have different rounding since no    /
    //  MPYs have been applied to either term.  Also, F0's  /
    //  rounding is slightly different to offset the        /
    //  truncation effects from the horizontal pass (which  /
    //  does not round).                                    /
    // ---------------------------------------------------- /
    F0r = (F0 + 0x0006) >>  3;
    F1r = (F1 + 0x7FFF) >> 16;
    F2r = (F2 + 0x7FFF) >> 16;
    F3r = (F3 + 0x7FFF) >> 16;
    F4r = (F4 + 0x0004) >>  3;
    F5r = (F5 + 0x7FFF) >> 16;
    F6r = (F6 + 0x7FFF) >> 16;
    F7r = (F7 + 0x7FFF) >> 16;

    // ---------------------------------------------------- /
    //  Store the results                                   /
    // ---------------------------------------------------- /
    dct_o[0+i_1] = F0r;
    dct_o[1+i_1] = F1r;
    dct_o[2+i_1] = F2r;
    dct_o[3+i_1] = F3r;
    dct_o[4+i_1] = F4r;
    dct_o[5+i_1] = F5r;
    dct_o[6+i_1] = F6r;
    dct_o[7+i_1] = F7r;

    // ---------------------------------------------------- /
    //  Update pointer to next FDCT row.                    /
    // ---------------------------------------------------- /
    //dct_io_ptr += 8;
    i_1 += 8;
  }


#ifdef SW
   t1 = time(NULL);
```

```
   c11 = clock();

   //for (i = 0; i < num_fdcts*M; i++) {
   //printf ("\timage values:                 %d\n", dct_o[i]);
   //}

   printf ("\tend (wall):                  %ld\n", (long) t1);
   printf ("\tend (CPU);                   %d\n", (int) c11);
   printf ("\telapsed wall clock time (s): %f\n", (float) (t1 - t0));
   printf ("\telapsed CPU time (s):        %f\n", (float) (c11 - c01)/CLOCKS_PER_SEC);
   printf ("\telapsed CPU time (ccs):      %ld\n", (long) (c11 - c01));
   printf ("\tclocks per second:           %d\n", CLOCKS_PER_SEC);

   printf("%d ", dct_o[num_fdcts*M-1]);
   /*for (i = 0; i < num_fdcts*M; i++) {
    printf("%d ", dct_o[i]);
    }*/
   //#include "time.end"
#endif

}
```

A.4 FilterRGB

A.5 Paeth

```
int predict(int a, int b, int c) {
   int pas, pbs, pcs;
   bool test_1, test_2, test_3;

   pas = b - c;
   pbs = a - c;
   pcs = a | b - 2 * c;
   test_1 = abs(pas) <= abs(pbs);
   test_2 = abs(pas) <= abs(pcs);
   test_3 = abs(pbs) <= abs(pcs);

   if (test_1 && test_2)
      return a;
   if (test_3)
      return b;
   return c;
}
```

A.6 ADPCM

```
/***********************************************************
Copyright 1992 by Stichting Mathematisch Centrum, Amsterdam, The
Netherlands.

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the names of Stichting Mathematisch
Centrum or CWI not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
```

```
FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

******************************************************************/

/*
** Intel/DVI ADPCM coder/decoder.
**
** The algorithm for this coder was taken from the IMA Compatability Project
** proceedings, Vol 2, Number 2; May 1992.
**
** Version 1.2, 18-Dec-92.
**
** Change log:
** - Fixed a stupid bug, where the delta was computed as
**   stepsize*code/4 in stead of stepsize*(code+0.5)/4.
** - There was an off-by-one error causing it to pick
**   an incorrect delta once in a blue moon.
** - The NODIVMUL define has been removed. Computations are now always done
**   using shifts, adds and subtracts. It turned out that, because the standard
**   is defined using shift/add/subtract, you needed bits of fixup code
**   (because the div/mul simulation using shift/add/sub made some rounding
**   errors that real div/mul don't make) and all together the resultant code
**   ran slower than just using the shifts all the time.
** - Changed some of the variable names to be more meaningful.
*/

#include "adpcm.h"
#include <stdio.h> /*DBG*/

#ifndef __STDC__
#define signed
#endif

/* Intel ADPCM step variation table */
static int indexTable[16] = {
    -1, -1, -1, -1, 2, 4, 6, 8,
    -1, -1, -1, -1, 2, 4, 6, 8,
};

static int stepsizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
    19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
    50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
    130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
    876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
    2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
    5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

void
adpcm_coder(indata, outdata, len, state)
    short indata[];
    char outdata[];
    int len;
    struct adpcm_state *state;
{
    short *inp; /* Input buffer pointer */
    signed char *outp; /* output buffer pointer */
    int val; /* Current input sample value */
    int sign; /* Current adpcm sign bit */
    int delta; /* Current adpcm output value */
    int diff; /* Difference between val and valprev */
    int step; /* Stepsize */
    int valpred; /* Predicted output value */
    int vpdiff; /* Current change to valpred */
    int index; /* Current step change index */
    int outputbuffer; /* place to keep previous 4-bit value */
```

```
    int bufferstep; /* toggle between outputbuffer/output */

    outp = (signed char *)outdata;
    inp = indata;

    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];

    bufferstep = 1;

    for ( ; len > 0 ; len-- ) {
val = *inp++;

/* Step 1 - compute difference with previous value */
diff = val - valpred;
sign = (diff < 0) ? 8 : 0;
if ( sign ) diff = (-diff);

/* Step 2 - Divide and clamp */
/* Note:
** This code *approximately* computes:
**    delta = diff*4/step;
**    vpdiff = (delta+0.5)*step/4;
** but in shift step bits are dropped. The net result of this is
** that even if you have fast mul/div hardware you cannot put it to
** good use since the fixup would be too expensive.
*/
delta = 0;
vpdiff = (step >> 3);

if ( diff >= step ) {
    delta = 4;
    diff -= step;
    vpdiff += step;
}
step >>= 1;
if ( diff >= step  ) {
    delta |= 2;
    diff -= step;
    vpdiff += step;
}
step >>= 1;
if ( diff >= step ) {
    delta |= 1;
    vpdiff += step;
}

/* Step 3 - Update previous value */
if ( sign )
  valpred -= vpdiff;
else
  valpred += vpdiff;

/* Step 4 - Clamp previous value to 16 bits */
if ( valpred > 32767 )
  valpred = 32767;
else if ( valpred < -32768 )
  valpred = -32768;

/* Step 5 - Assemble value, update index and step values */
delta |= sign;

index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;
step = stepsizeTable[index];

/* Step 6 - Output value */
if ( bufferstep ) {
    outputbuffer = (delta << 4) & 0xf0;
} else {
    *outp++ = (delta & 0x0f) | outputbuffer;
```

```
}
bufferstep = !bufferstep;
    }

    /* Output last step, if needed */
    if ( !bufferstep )
      *outp++ = outputbuffer;

    state->valprev = valpred;
    state->index = index;
}

void
adpcm_decoder(indata, outdata, len, state)
    char indata[];
    short outdata[];
    int len;
    struct adpcm_state *state;
{
    signed char *inp; /* Input buffer pointer */
    short *outp; /* output buffer pointer */
    int sign; /* Current adpcm sign bit */
    int delta; /* Current adpcm output value */
    int step; /* Stepsize */
    int valpred; /* Predicted value */
    int vpdiff; /* Current change to valpred */
    int index; /* Current step change index */
    int inputbuffer; /* place to keep next 4-bit value */
    int bufferstep; /* toggle between inputbuffer/input */

    outp = outdata;
    inp = (signed char *)indata;

    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];

    bufferstep = 0;

    for ( ; len > 0 ; len-- ) {

/* Step 1 - get the delta value */
if ( bufferstep ) {
    delta = inputbuffer & 0xf;
} else {
    inputbuffer = *inp++;
    delta = (inputbuffer >> 4) & 0xf;
}
bufferstep = !bufferstep;

/* Step 2 - Find new index value (for later) */
index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;

/* Step 3 - Separate sign and magnitude */
sign = delta & 8;
delta = delta & 7;

/* Step 4 - Compute difference and new predicted value */
/*
** Computes 'vpdiff = (delta+0.5)*step/4', but see comment
** in adpcm_coder.
*/
vpdiff = step >> 3;
if ( delta & 4 ) vpdiff += step;
if ( delta & 2 ) vpdiff += step>>1;
if ( delta & 1 ) vpdiff += step>>2;

if ( sign )
  valpred -= vpdiff;
else
  valpred += vpdiff;
```

```
/* Step 5 - clamp output value */
if ( valpred > 32767 )
  valpred = 32767;
else if ( valpred < -32768 )
  valpred = -32768;

/* Step 6 - Update step value */
step = stepsizeTable[index];

/* Step 7 - Output value */
*outp++ = valpred;
    }

    state->valprev = valpred;
    state->index = index;
}
```

A.7 Filtro SNN

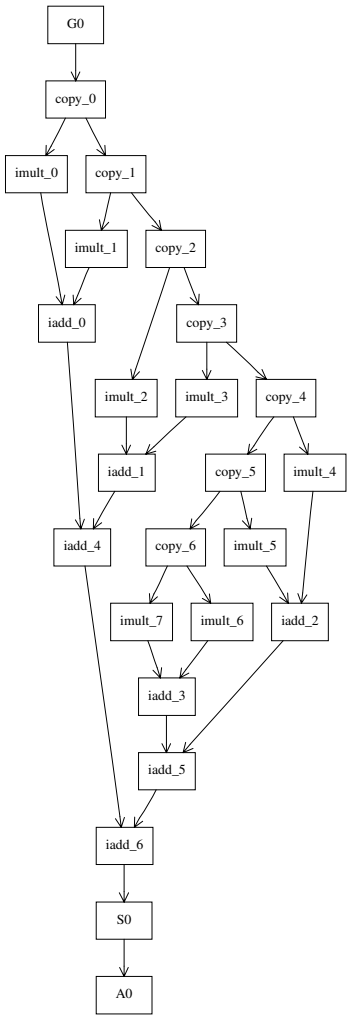
Trecho extraído do código disponível em http://www.uni-koblenz.de/ lb/lb_downloads/

```
/*

This "SOFTWARE" is a free software.

You are allowed to download, use, modify and redistribute this software.
The software is provided "AS IS" without warranty of any kind.

Copyright: University of Koblenz-Landau, Dirk Balthasar

*/

#include <ToolsImage.h>
namespace tools
{
   /// Symmetric nearest neighbour, help function
   /// \author Dirk Balthasar
   template <class ImageType, class AccessRGB>
      void FilterSNNcore(ImageType &Input, ImageType &Output, const tools::CROI &ROI,
                         std::vector<tools::CROI>& coord)
   {

      int sz = coord.size();
      for (unsigned int y = ROI.y1(); y < ROI.y2(); ++y)
      {
         for (unsigned int x = ROI.x1(); x < ROI.x2(); ++x)
         {
            int mr = 0, mg = 0, mb = 0;
            for (char a = 0; a < sz; ++a)
            {
               unsigned char
                  cr = AccessRGB::GetR(Input(x,y)),
                  cg = AccessRGB::GetG(Input(x,y)),
                  cb = AccessRGB::GetB(Input(x,y)),
                  r1 = AccessRGB::GetR(Input(x+coord[a].x1(),y+coord[a].y1())),
                  g1 = AccessRGB::GetG(Input(x+coord[a].x1(),y+coord[a].y1())),
                  b1 = AccessRGB::GetB(Input(x+coord[a].x1(),y+coord[a].y1())),
                  r2 = AccessRGB::GetR(Input(x+coord[a].x2(),y+coord[a].y2())),
                  g2 = AccessRGB::GetG(Input(x+coord[a].x2(),y+coord[a].y2())),
                  b2 = AccessRGB::GetB(Input(x+coord[a].x2(),y+coord[a].y2()));
               if (tools::DistRGBEuklidPow2(cr, cg, cb, r1, g1, b1) <
                     tools::DistRGBEuklidPow2(cr, cg, cb, r2, g2, b2))
               { // 1
                  mr += r1;
                  mg += g1;
                  mb += b1;
               }else
```

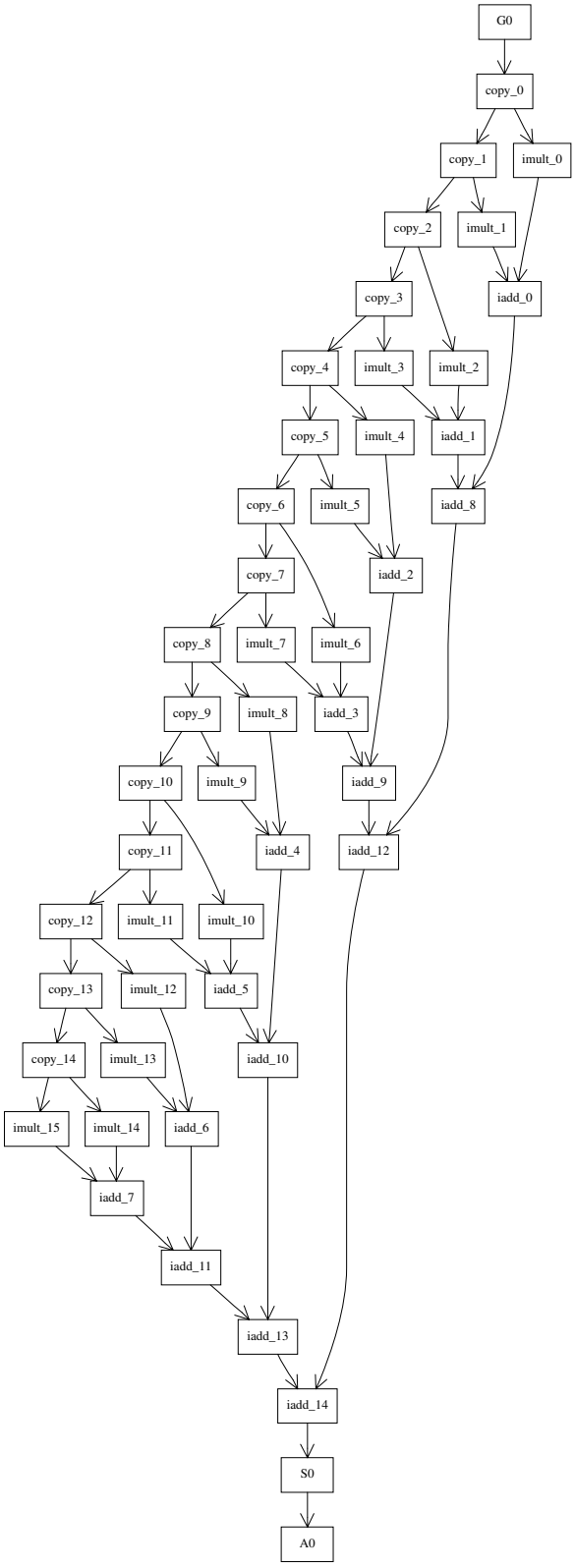
```
            { // 2
               mr += r2;
               mg += g2;
               mb += b2;
            }
         }
         // replace color by mean of r, g, b
         Output(x,y)   = AccessRGB::PackRGB(mr/4, mg/4, mb/4);
      }
   }
}

/// Filter Symmetric nearest neighbour
/// \author Dirk Balthasar
template <class ImageType, class AccessRGB>
   void FilterSNN(ImageType &Input, ImageType &Output)
{
   int w = Input.GetWidth(),
      h = Input.GetHeight();

   if (Output.GetWidth() != w || Output.GetHeight() != h)
      Output.SetSize(w,h);

   std::vector<tools::CROI> coord;

   // core of image
   coord.resize(4);
   coord[0] = tools::CROI(-1, -1, +1, +1);
   coord[1] = tools::CROI( 0, -1,  0, +1);
   coord[2] = tools::CROI(+1, -1, -1, +1);
   coord[3] = tools::CROI(-1,  0, +1,  0);
   FilterSNNcore<ImageType, AccessRGB>(Input, Output, tools::CROI(1, 1, w-1, h-1), coord);
   // corner dots (just clone)
   for (int y = 0; y < h; y++)
   {
      Output(0,y) = Input(0,y);
      Output(w-1,y) = Input(w-1,y);
   }
   for (int x = 0; x < w; x++)
   {
      Output(x,0) = Input(x,0);
      Output(x,h-1) = Input(x,h-1);
   }
}
};
```

ANEXO B – GRAFOS DE FLUXO DE DADOS

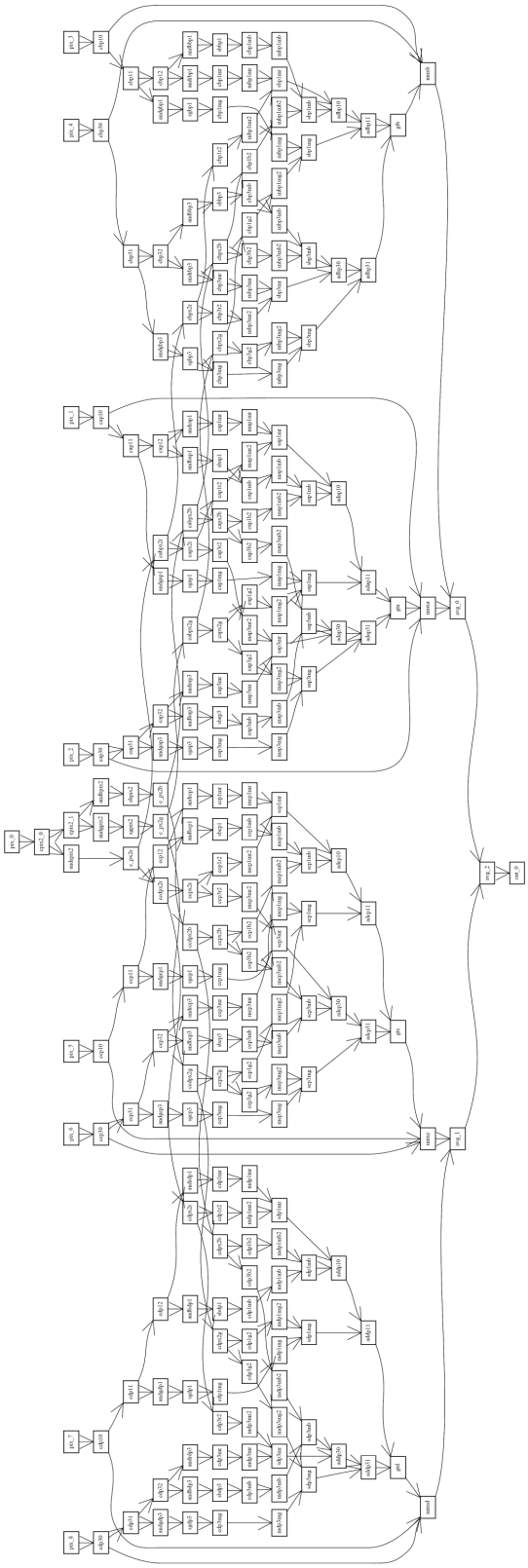
B.1 Fir8



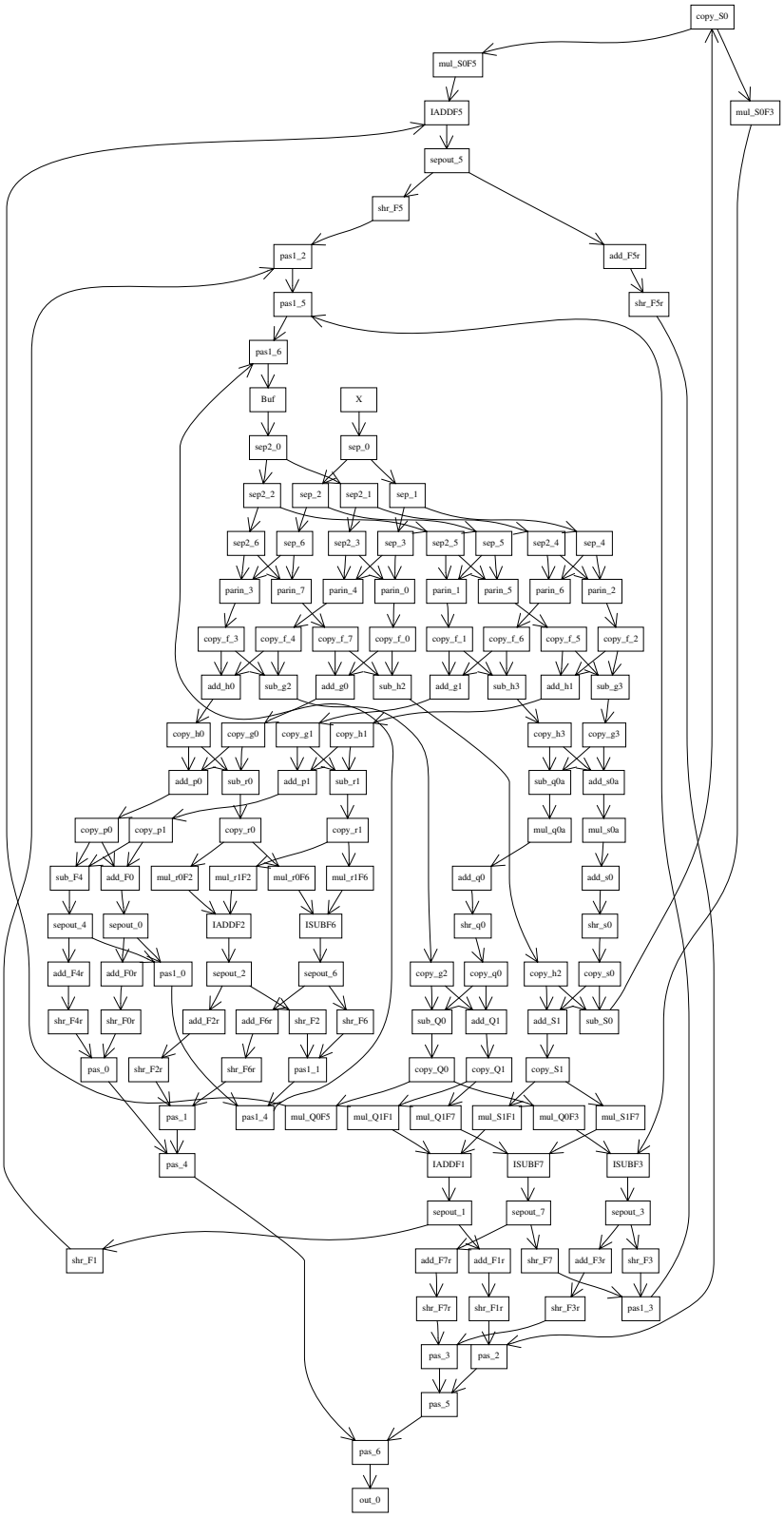
B.2 Fir16


G0
copy_0
copy_1
imult_0
copy_2
imult_1
copy_3
iadd_0
copy_4
imult_3
imult_2
copy_5
imult_4
iadd_1
copy_6
imult_5
iadd_8
copy_7
iadd_2
copy_8
imult_7
imult_6
copy_9
imult_8
iadd_3
copy_10
imult_9
iadd_9
copy_11
iadd_4
iadd_12
copy_12
imult_11
imult_10
copy_13
imult_12
iadd_5
copy_14
imult_13
iadd_10
imult_15
imult_14
iadd_6
iadd_7
iadd_11
iadd_13
iadd_14
S0
A0

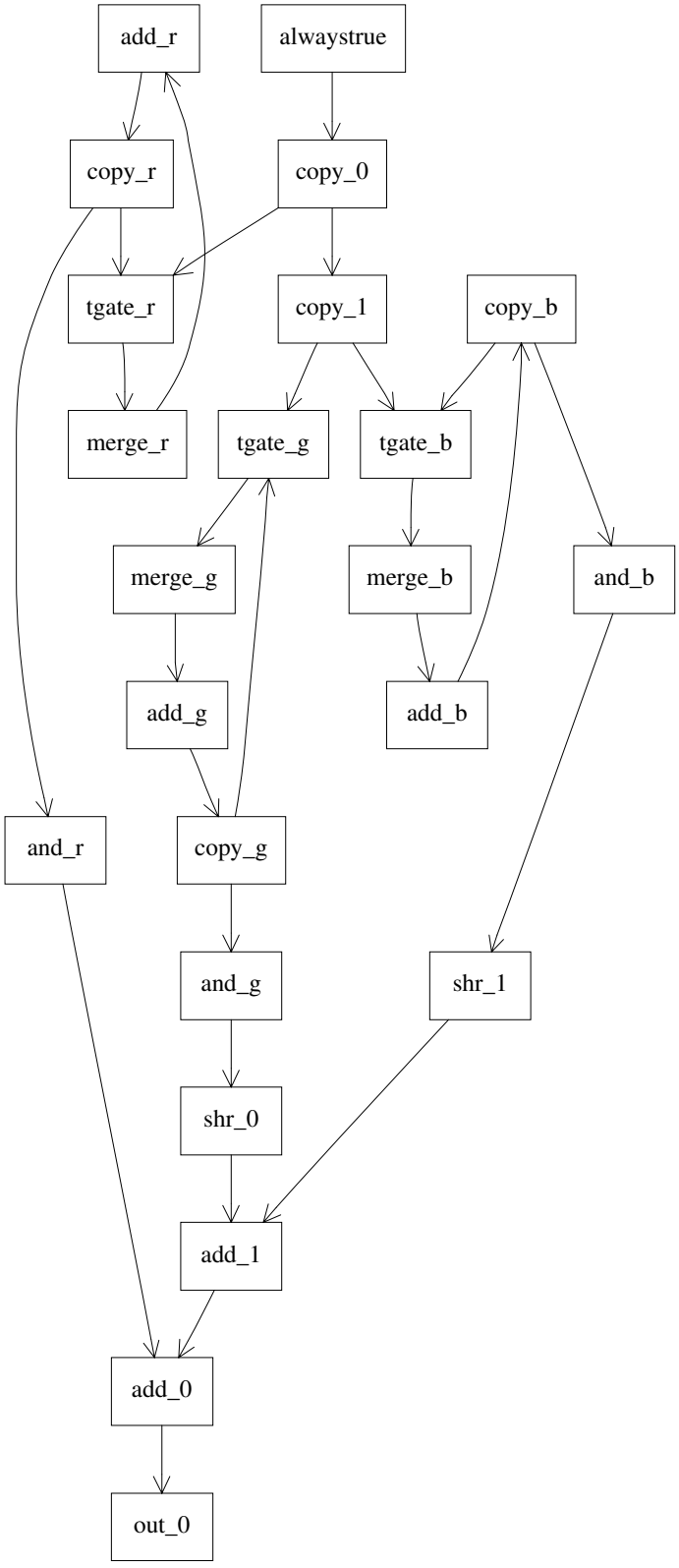
B.3 SNN



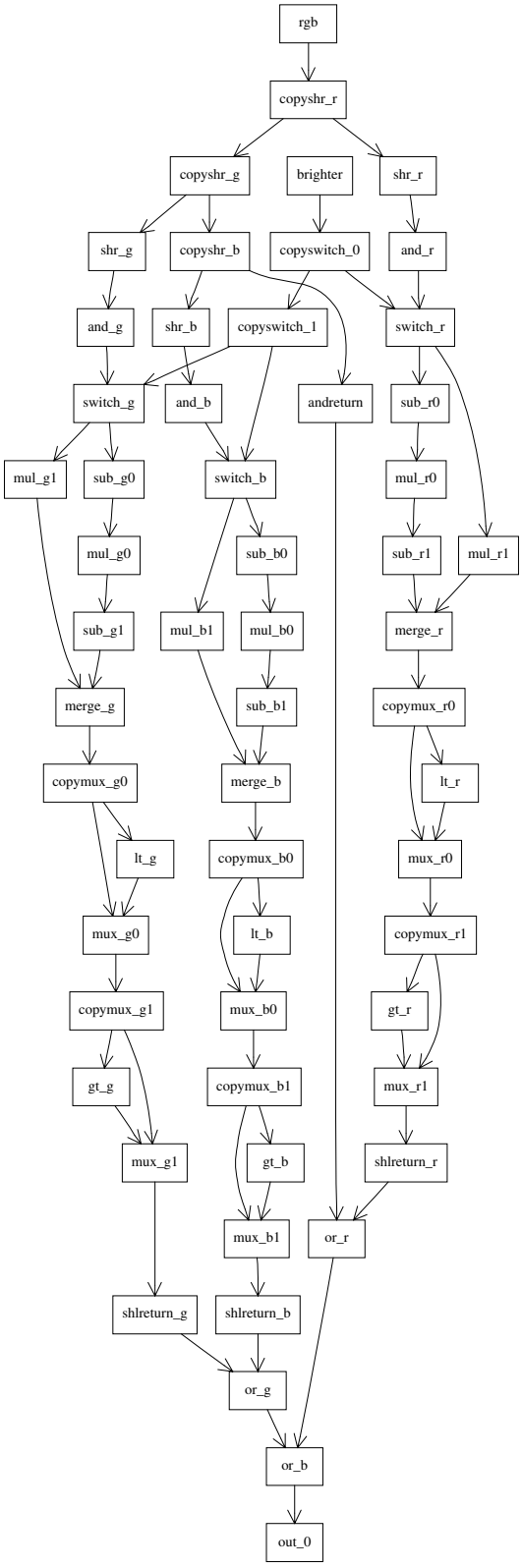
B.4 FDCT


copy_S0
mul_S0F5
mul_S0F3
IADDF5
sepout_5
shr_F5
pas1_2
add_F5r
pas1_5
shr_F5r
pas1_6
Buf
X
sep2_0
sep_0
sep2_2
sep_2
sep2_1
sep_1
sep2_6
sep_6
sep2_3
sep_3
sep2_5
sep_5
sep2_4
sep_4
parin_3
parin_7
parin_4
parin_0
parin_1
parin_5
parin_6
parin_2
copy_f_3
copy_f_4
copy_f_7
copy_f_0
copy_f_1
copy_f_6
copy_f_5
copy_f_2
add_h0
sub_g2
add_g0
sub_h2
add_g1
sub_h3
add_h1
sub_g3
copy_h0
copy_g0
copy_g1
copy_h1
copy_h3
copy_g3
add_p0
sub_r0
add_p1
sub_r1
sub_q0a
add_s0a
copy_p0
copy_p1
copy_r0
copy_r1
mul_q0a
mul_s0a
sub_F4
add_F0
mul_r0F2
mul_r1F2
mul_r0F6
mul_r1F6
add_q0
add_s0
sepout_4
sepout_0
IADDF2
ISUBF6
shr_q0
shr_s0
add_F4r
add_F0r
pas1_0
sepout_2
sepout_6
copy_g2
copy_q0
copy_h2
copy_s0
shr_F4r
shr_F0r
add_F2r
add_F6r
shr_F2
shr_F6
sub_Q0
add_Q1
add_S1
sub_S0
pas_0
shr_F2r
shr_F6r
pas1_1
copy_Q0
copy_Q1
copy_S1
pas_1
pas1_4
mul_Q0F5
mul_Q1F1
mul_Q1F7
mul_S1F1
mul_Q0F3
mul_S1F7
pas_4
IADDF1
ISUBF7
ISUBF3
sepout_1
sepout_7
sepout_3
shr_F1
add_F7r
add_F1r
shr_F7
add_F3r
shr_F3
shr_F7r
shr_F1r
shr_F3r
pas1_3
pas_3
pas_2
pas_5
pas_6
out_0

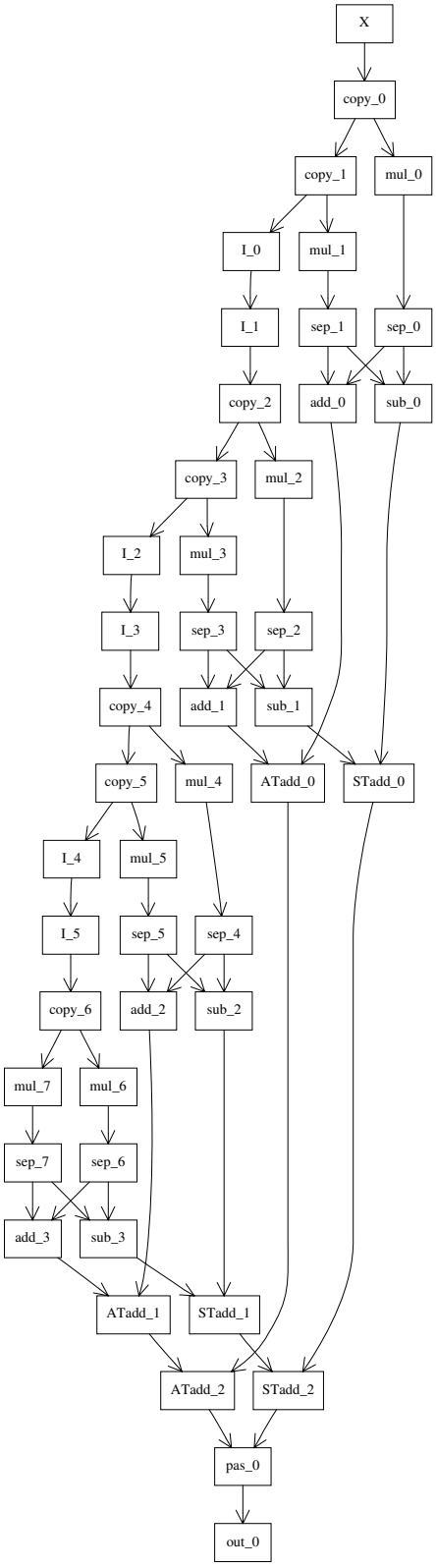
B.5 Gouraud


add_r
alwaystrue
copy_r
copy_0
tgate_r
copy_1
copy_b
merge_r
tgate_g
tgate_b
merge_g
merge_b
and_b
add_g
add_b
and_r
copy_g
and_g
shr_1
shr_0
add_1
add_0
out_0

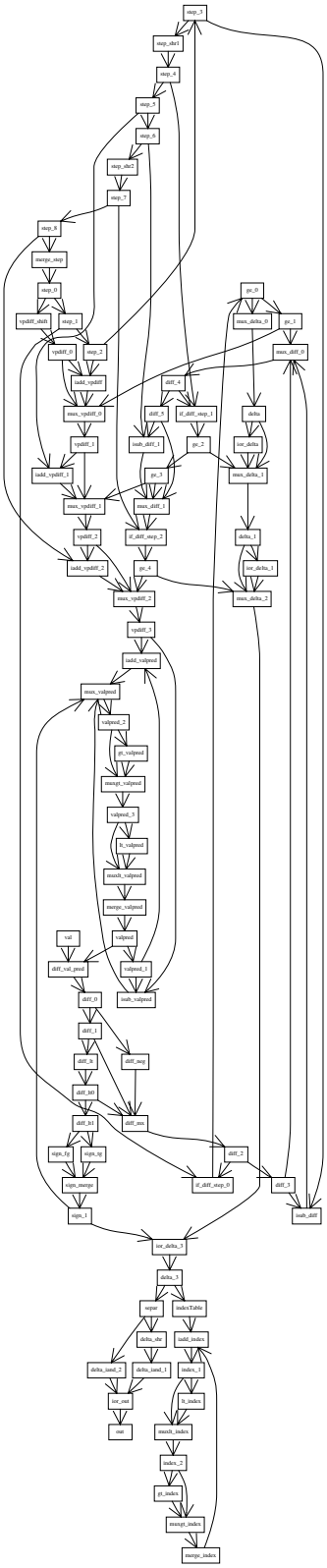
B.6 FilterRGB



B.7 CPLX


X
copy_0
copy_1
mul_0
I_0
mul_1
I_1
sep_1
sep_0
copy_2
add_0
sub_0
copy_3
mul_2
I_2
mul_3
I_3
sep_3
sep_2
copy_4
add_1
sub_1
copy_5
mul_4
ATadd_0
STadd_0
I_4
mul_5
I_5
sep_5
sep_4
copy_6
add_2
sub_2
mul_7
mul_6
sep_7
sep_6
add_3
sub_3
ATadd_1
STadd_1
ATadd_2
STadd_2
pas_0
out_0

B.8 ADPCM


step_3
step_shr1
step_4
step_5
sign_0
step_shr2
step_7
step_8
merge_step
step_9
vpdiff_shift
step_1
vpdiff_0
step_2
inlcd_vpdiff
mux_vpdiff_0
vpdiff_1
iadd_vpdiff_1
mux_vpdiff_1
vpdiff_2
iadd_vpdiff_2
mux_vpdiff_2
vpdiff_3
iadd_valpred
mux_valpred
valpred_2
gt_valpred
merge_valpred
valpred_3
lt_valpred
mux0_valpred
merge_valpred
valpred
val
diff_val_pred
valpred_1
isub_valpred
diff_0
diff_1
diff_3
diff_b0
diff_b1
diff_step
diff_sm
sign_0g
sign_1g
sign_merge
sign_1
ge_0
mux_delta_0
ge_1
mux_diff_0
diff_4
diff_5
if_diff_step_1
delta
sub_diff_1
ge_2
ior_delta
ge_3
mux_delta_1
mux_diff_1
if_diff_step_2
delta_1
ior_delta_1
ge_4
mux_delta_2
diff_2
if_diff_step_3
diff_3
sub_diff
ior_delta_3
delta_3
equal
indexTable
delta_shr
iadd_index
delta_iand_2
delta_iand_1
index_3
ior_out
lt_index
out
mux0_index
index_2
gt_index
merge_index
merge_index