Lucas Barbosa Sanches

ChipCFlow - *Partição e Protocolo de Comunicação no Grafo a Fluxo de Dados Dinâmico*

São Carlos - SP

Fevereiro / 2010

Lucas Barbosa Sanches

ChipCFlow - *Partição e Protocolo de Comunicação no Grafo a Fluxo de Dados Dinâmico*

Dissertação apresentada ao Instituto de Ciências Matemáticas e Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional.

Orientador:
Prof. Dr. Jorge Luiz e Silva

São Carlos - SP

Fevereiro / 2010

Agradecimentos

Em primeiro lugar, a Deus, por me conceder a bênção da saúde e do tempo para poder me dedicar a este trabalho. Ao meu orientador, professor Dr. Jorge, pela oportunide, e principalmente pela infinita paciência e apoio incondicional em todas as etapas, sem os quais não teria conseguido chegar até aqui. Ao Francisco, colega do mestrado que proporcionou as mais fecundas discussões em torno das questões do projeto. Sem ele este trabalho seria muitas vezes mais pobre. À Smar, empresa que me permitiu conduzir este trabalho de forma paralela à minha carreira profissional, sem prejuízo de um ou de outro. À minha esposa e à minha filha, que toleraram e apoiaram um marido e pai algumas vezes ausente, mas que nunca deixará de guardá-las com imenso amor no coração. Por fim, aos meus pais, alicerce fundamental sobre o qual pude sempre construir com tranquilidade e firmeza. Se pude chegar a algum lugar, devo isso a eles.

Resumo

Este trabalho descreve a prova de conceito de uma abordagem que utiliza o modelo de computação a fluxo de dados, inerentemente paralelo, associado ao modelo de computação reconfigurável parcial e dinamicamente, visando à obtenção de sistemas computacionais de alto desempenho. Mais especificamente, trata da obtenção de um modelo para o particionamento dos grafos a fluxo de dados dinâmicos e de um protocolo de comunicação entre suas partes, a fim de permitir a sua implementação em arquiteturas dinamicamente reconfiguráveis, em especial em FGPAs Virtex da Xilinx. Enquadra-se no contexto do projeto *ChipCFlow*, de escopo mais amplo, que pretende obter uma ferramenta para geração automática de descrição de *hardware* sintetizável, a partir de código em alto nível, escrito em linguagem C, fazendo uso da abordagem a fluxo de dados para extrair o paralelismo implícito nas aplicações originais. O modelo proposto é aplicado em um grafo a fluxo de dados dinâmico, e através de simulações sua viabilidade é discutida.

Abstract

This work describes the concept of an approach that uses dataflow computational model, inherently parallel, associated with de reconfigurable computing model, partial and dynamic, in order to obtain high performance computational systems. More specifically, it is about a model to the partitioning and communication between partitioned sectors of a CDFG (Control Dataflow Graph) in order to map these graphs on a partial reconfiguration FPGA fabric, in special Virtex II/II-Pro from Xilinx. It is part of the *ChipCFlow* project, that has a bigger scope, and that aims to automatically obtain syntetisable hardware descriptions, from high level code written in C and, by using a dataflow approach to extract implicit parallelism in original applications. The model obtained is extensively explained and applied to an example of CDFG, where by means of simulations its feasibility is discussed.

Sumário

Lista de Figuras

1 Introdução

O modelo de computação a fluxo de dados sempre despertou interesse como área de pesquisa por sua elegância ao extrair e representar de maneira simples o paralelismo presente em tarefas de computação. Houve muitas tentativas no passado de se implementar máquinas de propósito geral baseadas no modelo. No entanto, as dificuldades para se obter uma máquina a fluxo de dados de propósito geral sempre se mostraram enormes (LEE; HURSON, 1993). Já a partir do final da década de 80 o interesse pelo assunto parecia estar diminuindo, como mostram, por exemplo, as estatísticas em (NAJJAR; LEE; GAO, 1999). No entanto, devido ao seu potencial, o modelo nunca deixou totalmente de levantar interesse da comunidade científica.

Embora as pesquisas não tenham levado a uma arquitetura de propósito geral largamente comercial, o modelo a fluxo de dados é amplo o suficiente para ser aplicado em diversas áreas, como linguagens de programação, projeto de processadores, processamento digital de sinais e, mais recentemente, computação reconfigurável (NAJJAR; LEE; GAO, 1999) (COMPTON; HAUCK, 2002).

A computação reconfigurável, por sua vez, apesar de estar embasada em conceitos que não são novos, definiu-se mais claramente como área de pesquisa graças ao grande avanço das tecnologias de fabricação de circuitos integrados ocorrido nas últimas décadas, os quais permitiram o surgimento e a evolução dos FPGAs (*Field Programmable Gate Arrays*) ou matrizes de portas programáveis em campo. De acordo com (COMPTON; HAUCK, 2002) atualmente quando a expressão computação reconfigurável é usada, refere-se a sistemas que incorporam algum tipo de capacidade de programação de *hardware* utilizando certo número de pontos de controle que podem ser alterados, e que configuram como este *hardware* é usado. FPGAs e CPLDs são exemplos de dispositivos que concretizam esse conceito em um único circuito integrado.

Eles possibilitam a flexibilidade de se obter *hardware* específico para determinada aplicação de maneira rápida e barata. Rápida e barata por não exigir que este *hard-*

ware projetado seja fabricado, como circuito integrado de aplicação específica (ASIC - *Application-specific integrated circuit*), mas possa simplesmente ser carregado *em campo*, ou seja, no momento do uso, como uma configuração de um FPGA.

Normalmente, o circuito digital projetado é definido em uma linguagem de descrição de *hardware* (HDL - *Hardware Description Language*), sendo VHDL e Verilog as mais comumente usadas. Ferramentas de *software* adequadas sintetizam o circuito projetado, ou seja, obtêm a configuração correspondente a partir da entrada em linguagem de descrição.

Essa tecnologia permite, assim, a prototipação rápida de circuitos de propósito específico, sem a necessidade da fabricação. Além disso, abre campo para inúmeras outras aplicações.

Os FPGAs atuais agregam até milhões de portas lógicas, podendo ser usados para implementar, em um único circuito integrado, desde aplicações simples de lógica digital até sistemas inteiros, compostos de um processador (ou mais de um) e todos os seus periféricos necessários. Estes últimos conhecidos como SoCs - *System on a Programmable Chip*.

Além da possibilidade de se obter sistemas em *hardware* de grande complexidade para propósitos específicos de maneira rápida e barata, o que é de grande interesse para a indústria eletrônica e de computação em geral, os FPGAs fomentaram campos novos de pesquisa. Um desses campos é o que busca obter descrições de *hardware* eficientes a partir de programação em alto nível. Há ferramentas comerciais, como as citadas em (LOPES, 2007), que geram descrição de *hardware* que implementa algoritmos definidos a partir de linguagem de alto nível, como C, por exemplo.

Com essas novas possibilidades abertas, a computação reconfigurável, indiretamente, deu novo fôlego às pesquisas relacionadas ao modelo de computação a fluxo de dados. A fim de fazer uso do imenso potencial de paralelismo real oferecido pelas tecnologias de computação reconfigurável, o modelo a fluxo de dados se mostra bastante promissor, e é usado em algumas das ferramentas citadas anteriormente como representação intermediária, visando extrair o paralelismo implícito na codificação em alto nível. Foi revisitado também, em pesquisas de arquiteturas de processadores novas e bastante promissoras, como TRIPS e Wavescalar.

Este trabalho se enquadra neste contexto, e faz parte do projeto ChipCFlow, que tem por objetivo obter uma ferramenta de compilação capaz de gerar descrição de *hardware* a partir de código de alto nível (linguagem C) e que utilize como representação inter-

mediária o modelo a fluxo de dados dinâmico, visando extrair ao máximo o paralelismo de granularidade fina intrínseco ao código.

Numa primeira etapa do projeto, foram obtidos os operadores usados nos grafos a fluxo de dados do projeto (ASTOLFI; SILVA, 2007).

Numa segunda etapa, pretende-se obter implementações de grafos dinâmicos, fazendo uso de características de reconfiguração parcial dos FPGAs.

1.1 Objetivos do trabalho

A idéia de mapear grafos a fluxo de dados dinâmicos em FPGAs parcialmente reconfiguráveis é um dos pontos centrais do projeto ChipCFlow. Um dos objetivos deste trabalho é apresentar uma cadeia consistente e completa para o mapeamento dos grafos a fluxo de dados dinâmicos, incluindo seus operadores de *tag*, em FPGAs parcialmente reconfiguráveis. Consistente no sentido de garantir que o sistema execute de acordo com o modelo a fluxo de dados dinâmico, sem *deadlocks*, e completo no sentido de cobrir toda a cadeia de desenvolvimento para se obter o sistema final, tomando como ponto de partida os grafos dinâmicos obtidos por compilação. Esta é uma grande contribuição para o projeto. Para atingir o objetivo de mapear os grafos dinâmicos em uma arquitetura de FPGA parcialmente reconfigurável, é apresentado um modelo para o particionamento dos grafos e a comunicação entre suas partições. A proposta será aplicada a um exemplo, visando analisar suas características.

Não faz parte do escopo deste trabalho qualquer preocupação com a obtenção dos grafos a partir dos códigos em alto nível. Apenas admite-se que os grafos são entradas e que a saída é um sistema parcialmente reconfigurável que os implemente.

1.2 Organização da dissertação

No capítulo 2, são apresentados os conceitos e principais arquiteturas tradicionais e modernas que fazem uso do modelo de computação a fluxo de dados. As informações apresentadas são essenciais para o embasamento do projeto ChipCFlow.

No capítulo 3 é feito um levantamento das tecnologias envolvidas no campo da computação reconfigurável, bem como das principais arquiteturas baseadas nela. É apresentada, também, a tecnologia para reconfiguração parcial da Xilinx, primeiro fabricante a

fornecer dispositivos e ferramentas para essa nova tecnologia, ainda experimental.

O capítulo 4 apresenta em maiores detalhes o projeto ChipCflow, principalmente no que tange ao seu conceito de mapeamento dos grafos a fluxo de dados dinâmicos (CDFG) em FPGAs parcialmente reconfiguráveis.

No capítulo 5 é proposto um modelo para o particionamento e para o sistema de comunicação entre as partições previstas para os grafo a fluxo de dados dinâmicos do projeto. A implementação e simulação do modelo para um exemplo específico é mostrada, visando provar a viabilidade da proposta apresentada.

Por fim, o capítulo 6 traz discussões e conclusões a respeito do trabalho, além de propor um caminho para o projeto ChipCFlow a partir dos resultados.

2 Computação a Fluxo de Dados

O modelo de computação a fluxo de dados é um modelo de computação simples e poderoso, especialmente na representação do paralelismo em diversas tarefas de computação. Ele está baseado na idéia de que uma operação de computação, para ser executada, necessita apenas de que seus operandos estejam disponíveis num determinado instante. Dessa maneira, diversas operações podem ser executadas simultaneamente.

Neste modelo, os programas são grafos orientados cujos nós (em alguns casos chamados de atores (NAJJAR; LEE; GAO, 1999)) definem operações, e os arcos, a relação entre as operações, indicando a origem e o destino dos dados (também chamados de *tokens*). Como exemplo, o grafo da Figura 1 representa a computação A = (B - C) * (D + E).

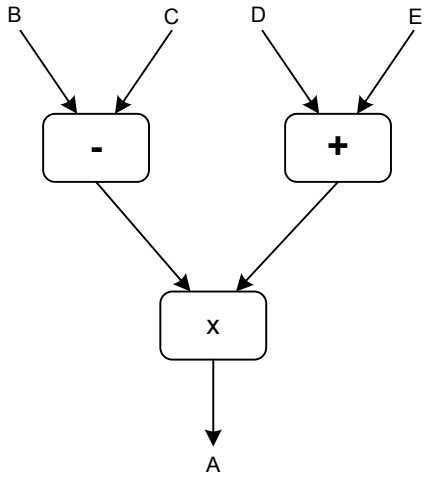


Figura 1: Representação na forma de um grafo a fluxo de dados da computação de A = (B-C)*(D+E)

No modelo, os dados, ou *tokens*, transitam entre os operadores através dos arcos. Todo operador possui arcos de entrada e arcos de saída. Desde que todos os dados necessários nas entradas estejam disponíveis, o operador executa (dispara) e gera *tokens* nos seus arcos de saída, consumindo os *tokens* de suas entradas. Dessa forma, as operações de subtração e de soma indicadas na Figura 1, por exemplo, podem ser executadas paralelamente.

Em arquiteturas a fluxo de dados não existe o conceito de uma memória global nem de um contador de programa (*program counter*), como nas arquiteturas Von Neumann tradicionais, e o controle das operações está distribuído entre os nós do grafo. O modelo potencialmente permite, como mostrado no exemplo anterior, extrair de maneira natural o paralelismo dos algoritmos, além de fornecer um mecanismo simples de sincronização entre as operações.

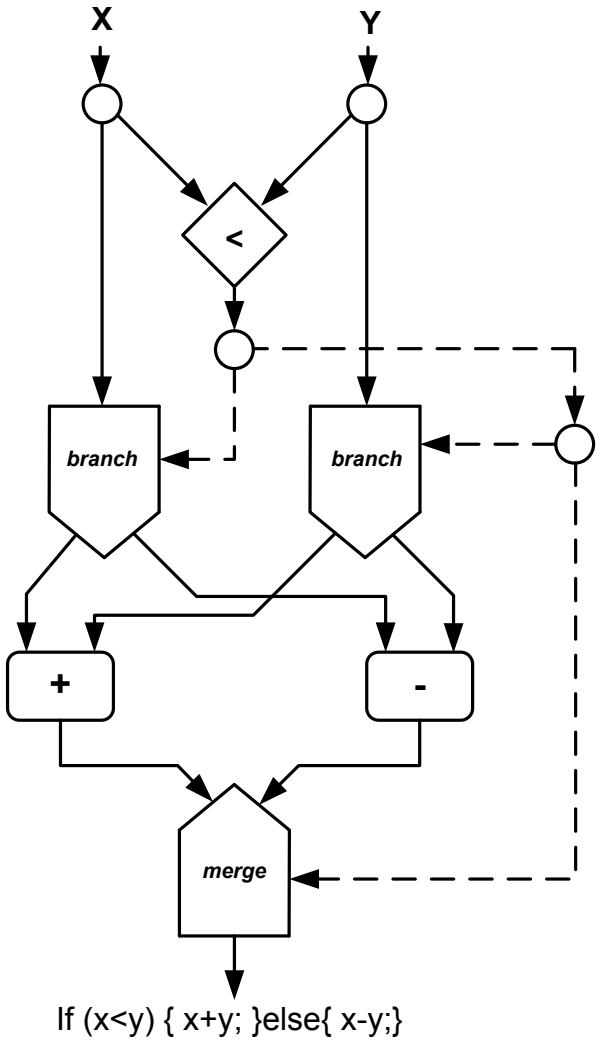


Figura 2: Grafo a fluxo de dados incorporando operadores de controle de fluxo (adaptado de (ARVIND; CULLER, 1986))

.

O modelo a fluxo de dados em princípio não define restrições para a complexidade dos operadores, cabendo aos projetistas de determinada arquitetura esta definição.

O grafo apresentado na Figura 1 é um grafo simples, que possui apenas operadores lógico matemáticos. Além desses operadores há os de controle e de decisão, que geram *tokens* que controlam o fluxo de dados em outros operadores. Em geral, as arestas que

conduzem estes *tokens* são representadas por linhas tracejadas, como forma de diferenciação (Figura 2).

No modelo adotado neste trabalho os nós podem pertencer a três classes: condicionais, de cópia e junção, ou funcionais. Entre os primeiros estão dois operadores: o de decisão (*decider*) e o de ramificação (*branch*). A Figura 3 ilustra os dois operadores antes e depois do disparo, enquanto a Figura 2 mostra um grafo simples que faz uso desses operadores.

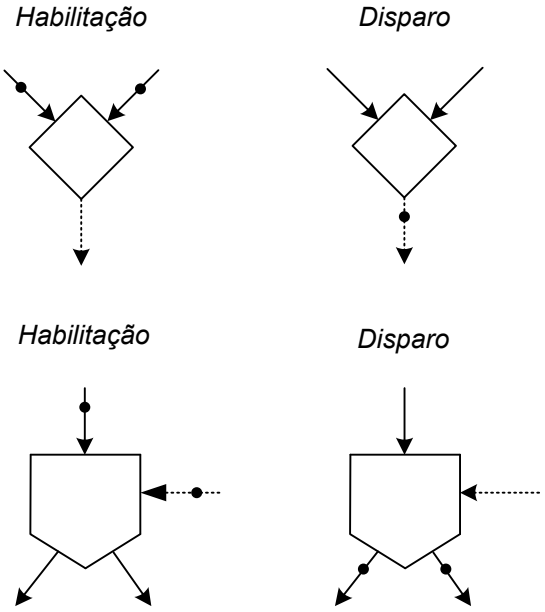


Figura 3: Operadores *Decider* e *Branch*, respectivamente.

O operador de decisão recebe dois valores e quando habilitado gera um dado (*token*) booleano verdadeiro ou falso de acordo com a aplicação de uma função aos dados de entrada.

O operador de ramificação, por sua vez, recebe um *token* em sua entrada que é direcionado para uma ou outra aresta de saída de acordo com um *token* de controle recebido em outra entrada.

Como operadores de cópia e junção tem-se os operadores junção determinística por valor (*deterministic merge*), junção não determinística (*non-deterministic merge*) e o operador de cópia (*copy*). A Figura 4 ilustra os três operadores antes e depois de seu disparo.

O operador de junção determinística recebe um valor de controle que seleciona uma ou outra entrada para ser colocada em sua saída após a sua habilitação. O operador de junção não determinística coloca em sua saída o primeiro dos dois valores possíveis que receber em suas entradas. O operador de cópia, por sua vez, duplica um dado recebido

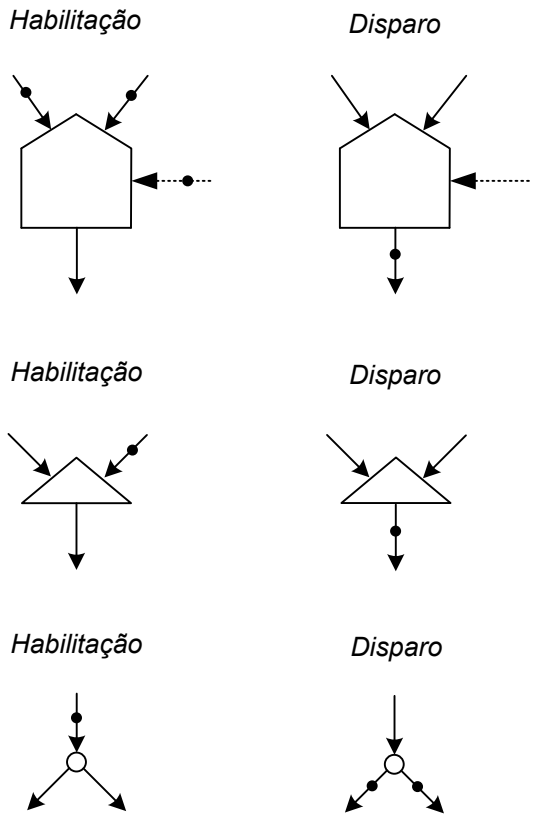


Figura 4: Operadores *Deterministic merge*, *non-deterministic merge* e *copy*. Através destes operadores há a criação ou eliminação de dados durante a execução de um grafo.

em sua entrada para as suas duas saídas.

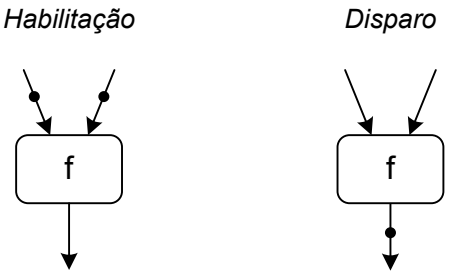


Figura 5: Operador funcional hipotético. Tradicionalmente é implementado um operador para cada função lógica ou aritmética básica

Os operadores funcionais (Figura 5) são os que efetivamente realizam as operações de computação desejadas com os dados. Os mais comuns são aqueles que implementam as operações lógicas e aritméticas básicas, como soma, subtração, 'e', 'ou', 'não', etc. No entanto, como afirmado anteriormente, a princípio não há restrições no modelo a fluxo de dados para a complexidade do operador.

2.1 Ciclos, iterações e reentrância em grafos a fluxo de dados

Quando um grafo contém ciclos, é possível que alguns problemas sejam originados. No grafo hipotético (a) da Figura 6 tem-se uma situação que gera um *deadlock*, enquanto em (b) tem-se uma situação em que o grafo não termina de executar nunca. As situações deste exemplo (VEEN, 1986), podem ocorrer em qualquer grafo cíclico, a menos que precauções especiais sejam tomadas.

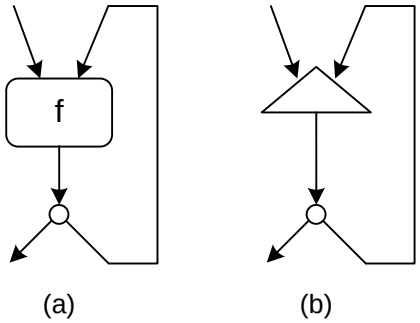


Figura 6: Problemas com estruturas cíclicas: o grafo da esquerda (a) fica em *deadlock*, o da direita (b) nunca termina de executar.

A Figura 7 mostra um exemplo de grafo cíclico, e a computação que implementa.

Trata-se de um *loop* implementado de forma insegura. Neste exemplo, é possível que um novo *token* chegue ao subgrafo 'h' antes de o primeiro ter sido processado.

Esta ocorrência é chamada de reentrância, e a forma que uma determinada arquitetura baseada no modelo a fluxo de dados lida com ela é um conceito fundamental(VEEN, 1986).

Uma forma de resolver a questão da reentrância é através do método da fechadura (*lock method*). Um exemplo de grafo cíclico seguro, que usa este método, está ilustrado na Figura 8.

Esta abordagem exige a inserção de novos operadores: o *branch* composto e o *merge* composto. Através do uso adequado destes operadores garante-se o determinismo do grafo. Eles fazem o sincronismo adequado dos dados, evitando a ocorrência de múltiplos *tokens* em uma entrada do subgrafo 'g' 8, e mantêm a condição de disparo estrita, ou seja, disparam apenas quando todos os *tokens* em suas entradas estiverem presentes.

Uma outra forma de lidar com a situação da reentrância é o uso de arcos de *acknowledge* (reconhecimento) entre os operadores produtores e consumidores. Este método garante que cada arco de um grafo armazene apenas um *token* em um determinado mo-

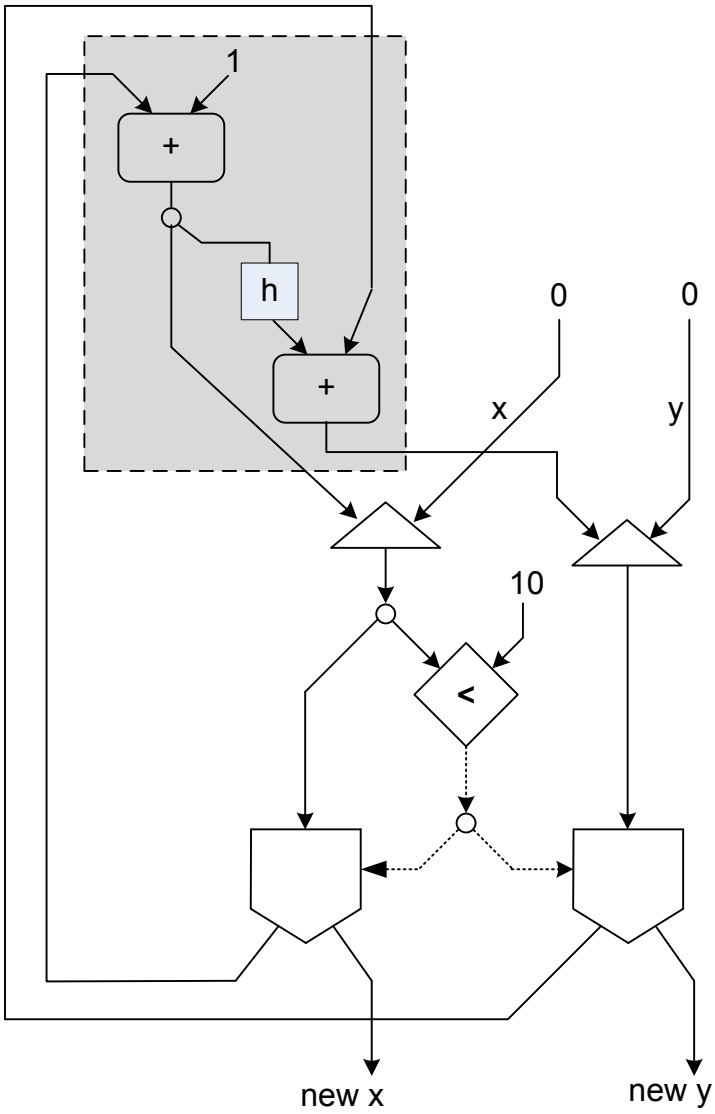


Figura 7: Uma maneira insegura de implementar um *loop*.

mento, mantendo o seu determinismo. Os produtores aguardam até que os consumidores sinalizem através do arco de reconhecimento o recebimento do *token*, e só então o produtor está apto a enviar novos *tokens*. Esta abordagem permite uma concorrência maior que o método da fechadura, ao custo de o número de arcos e *tokens* no mínimo dobrar.

No entanto, é interessante que se possibilite um maior nível de concorrência, permitindo que as iterações dos laços sejam executadas paralelamente. Uma forma de se conseguir isso é utilizando arquiteturas que de algum modo permitam cópia de código. Nesta abordagem, novas instâncias do grafo reentrante são criadas para cada iteração dos laços (*loops*) e a arquitetura precisa prover algum mecanismo de os *tokens* serem direcionados para a sua instância corretamente.

Uma segunda solução potencialmente mais eficiente de implementar a cópia de código

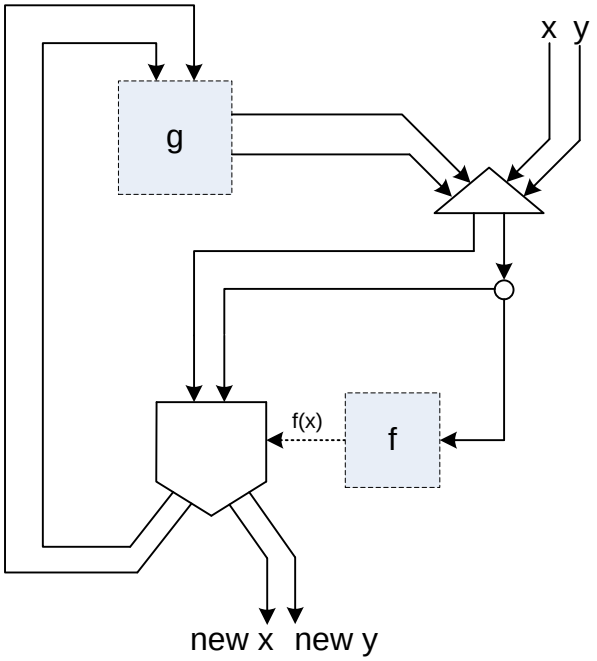


Figura 8: *Loop* seguro usando *branch* composto (VEEN, 1986)).

é alcançada através de arquiteturas que compartilham a estrutura do grafo reentrante original, mas distingue os dados de cada iteração através de uma etiqueta (*tag*), que define a qual instância um *token* pertence. Uma mudança é realizada na regra de disparo: aqui, para disparar, um operador precisa receber dados de um mesmo *tag* em suas entradas. Essas são as arquiteturas *tagged-token*.

A Figura 9 ilustra como a característica dos *tags* aparece em um programa para arquiteturas deste tipo. Além dos operadores tradicionais, outros que modificam *tags* estão presentes.

2.2 Arquiteturas tradicionais a fluxo de dados

Tradicionalmente a forma pela qual uma máquina a fluxo de dados lida com a reentrância define sua classificação como estática ou dinâmica. Aquelas que tratam a reentrância através de travamento ou de operadores com reconhecimento estão entre as primeiras, enquanto aquelas com mecanismos de cópia de código ou *tagged-token* estão entre as últimas.

Na abordagem estática uma única instância de um nó pode estar habilitada para disparar. Um operador só poderá disparar quando todas as suas entradas estiverem presentes e não houver *tokens* em nenhum de seus arcos de saída. Na abordagem dinâmica

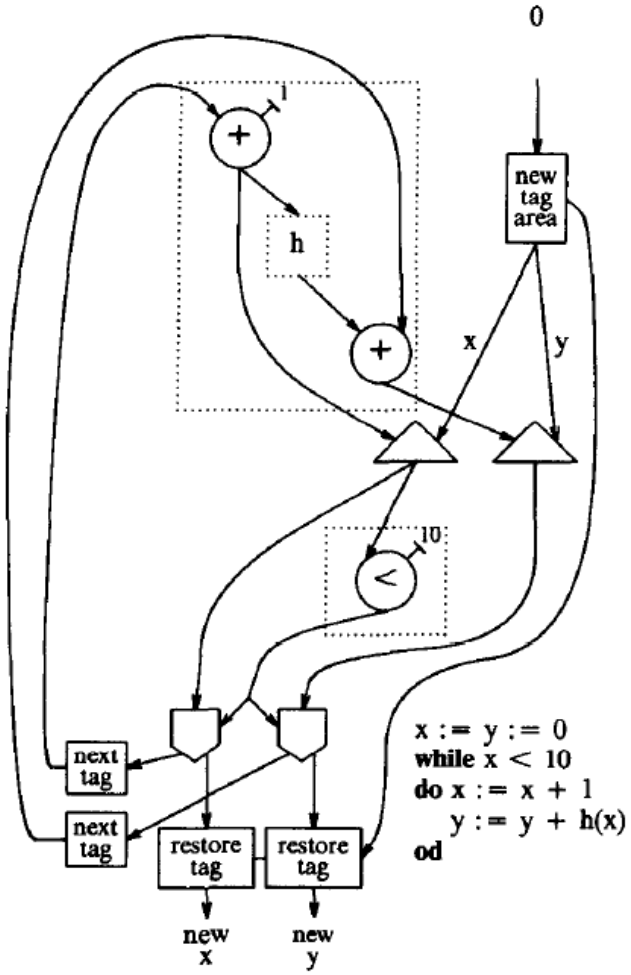


Figura 9: Uma implementação de um loop usando *tagged tokens*. No início do *loop* um novo *tag* é alocado (definindo uma nova *área* de *tokens*). *Tokens* de iterações consecutivas recebem *tags* indicando pertencerem a esta área. O *tag* anterior ao *loop* é restaurado em *tokens* que saem do *loop* (VEEN, 1986).

é possível o disparo de várias instâncias de um nó ao mesmo tempo *em tempo de execução*. O(*tag*), é usado para diferenciar as instâncias de um nó, e identifica o contexto em que um determinado *token* foi criado. Neste caso, um nó está habilitado quando suas entradas contem um conjunto de *tokens* com *tags* idênticos.

As máquinas a fluxo de dados clássicas, que serviram de base para desenvolvimentos subsequentes foram as máquinas estática de Jack Dennis (DENNIS; MISUNAS, 1974) e as máquinas dinâmicas do MIT(ARVIND; NIKHIL, 1990) e da Universidade de Manchester(GURD; KIRKHAM; WATSON, 1985).

2.2.1 Máquinas a fluxo de dados estáticas

A Figura 10 mostra a organização geral de uma máquina fluxo de dados estática. A memória de programa contém padrões que representam os nós em um grafo a fluxo de dados (Figura 11). Cada padrão de estrutura possui um código de operação, espaços para os operandos e endereços de destino. Para determinar a disponibilidade dos operandos, os espaços contem *bits* de presença. A unidade de atualização é responsável por detectar as instruções prontas para serem executadas. Quando esta condição é verificada, a unidade de atualização envia as instruções prontas para a unidade de *Fetch*. A unidade de *Fetch* envia um pacote completo de operação contendo o *opcode* correpondente, os dados e a lista de destinos para a unidade de processamento e também limpa os *bits* de presença. A unidade de processamento realiza a operação, monta o pacote resultante e o envia para a unidade de atualização. A unidade de atualização armazena cada resultado no espaço dos operandos apropriados e checa os *bits* de presença para determinar se a atividade está habilitada.

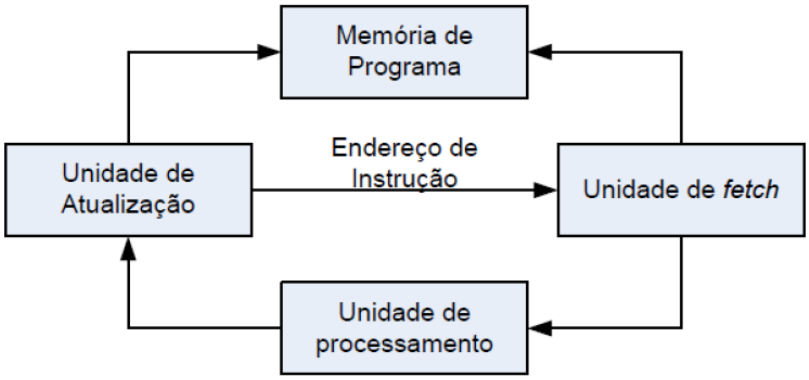


Figura 10: Organização básica de uma máquina fluxo de dados estática((LEE; HURSON, 1993))

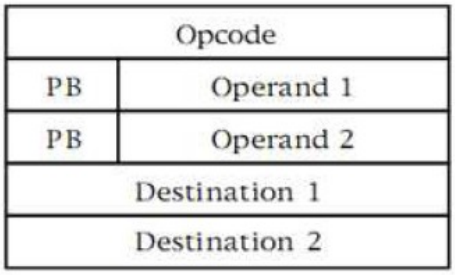


Figura 11: Formato de instrução de uma máquina fluxo de dados estática

2.2.2 Máquinas a fluxo de dados dinâmicas

O modelo a fluxo de dados dinâmico foi proposto por Arvind no MIT (ARVIND; NIKHIL, 1990) e por Gurd e Watson na Universidade de Manchester(GURD; KIRKHAM; WATSON, 1985). A organização básica da arquitetura a fluxo de dados dinâmica é mostrada na Figura 12. Os *tokens* são recebidos pela unidade de *Matching*, que é uma memória contendo uma coleção de *tokens* em espera. A operação básica da Unidade de *Matching* é colocar juntos *tokens* com *tags* idênticos. Se um acerto existe (*match*), o *token* correspondente é extraído da Unidade de *Matching* e este conjunto de *tokens* é passado para a unidade de *Fetch*. Na unidade de *Fetch* os *tags* do par de *tokens* identificam univocamente uma instrução a ser carregada da memória de programa. Um formato típico de instrução para o modelo a fluxo de dados consiste de um código de operador, um campo literal ou constante e um campo de destino. A instrução, em conjunto com o par de *tokens* formam a instrução habilitada, a qual é enviada para a Unidade de Processamento. A unidade de processamento executa as instruções habilitadas e produz *tokens* de resultado a serem enviados à unidade de *Matching*.

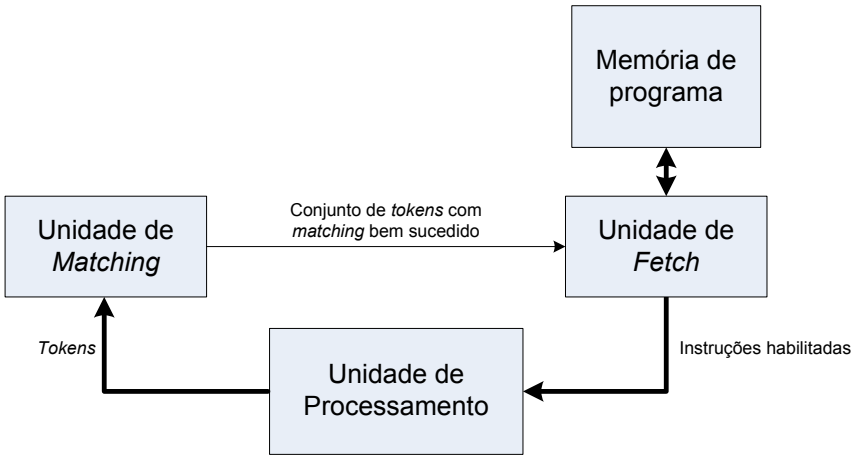


Figura 12: Esquema geral de uma arquitetura a fluxo de dados dinâmica.
(LEE; HURSON, 1993)

A maior vantagem do modelo dinâmico sobre o estático está no maior desempenho que pode ser obtido ao se permitir múltiplos *tokens* em um arco. Um laço, por exemplo, é desdobrado em tempo de execução, criando múltiplas instâncias do seu corpo e permitindo a execução de suas instâncias concorrentemente. O *overhead*, no entanto, envolvido na operação de *matching* sempre foi um grande problema. Para reduzir este custo em tempo de execução, as máquinas dinâmicas requerem a implementação de memórias associativas. Na prática, devido ao alto custo dessas memórias o que se usou foram mecanismos de *matching* pseudo-associativos, que requerem múltiplos acessos à memória. Estes acessos

tipicamente reduzem o desempenho e a eficiência das máquinas a fluxo de dados dinâmicas.

Outro problema típico dessas arquiteturas é a complexidade envolvida na alocação de recursos (regiões de memória). Cada operação de *matching* mal-sucedida provoca alocação de memória na unidade de *matching*. Em outras palavras, quando um bloco de código é mapeado para um processador, assume-se que a unidade de *matching* será capaz de suportar a alocação máxima de recursos do programa. Se este limite supera a capacidade da unidade, o programa pode entrar em *deadlock*. Por outro lado, esses recursos são muito caros para serem considerados infinitos.

2.2.2.1 A máquina de Manchester ((GURD; KIRKHAM; WATSON, 1985))

O grupo de Manchester desenvolveu o conceito de *tagged-tokens* para aumentar o paralelismo em grafos reentrantes independentemente do trabalho similar de Arvind e Gostelow(ARVIND; NIKHIL, 1990). A estrutura da arquitetura desenvolvida era similiar àquela mostrada na Figura 13. Tratava-se de um *pipeline* de quatro unidades: *token queue*, unidade de *matching*, unidade de *fetching* e unidade funcional (ou de processamento). Cada unidade trabalhava internamente de forma síncrona, mas se comunicava por meio de protocolos assíncronos.

Mais de 30 pacotes podiam ser processados simultaneamente nos vários estágios do *pipeline*. Para maximizar a velocidade de comunicação todos os dados eram transmitidos paralelamente, o que fazia com que cada pacote fosse transmitido totalmente de uma única vez. Consequentemente o tamanho dos pacotes, e portanto, dos *tokens*, era fixo. O *token queue* era simplesmente um *buffer* FIFO. Ele servia para regular as taxas irregulares de saída de dados de outras duas unidades no *pipeline*: a unidade de *matching* e a unidade funcional.

A unidade de *matching* aceitava *tokens* do *token queue* e enviava conjuntos completos de *tokens* de entrada para a unidade de *fetching*. Nessa máquina o número de arcos de entrada de um nó era limitado a dois, e um nó de destino poderia, portanto, ser apenas de uma entrada (*monadic*) ou de duas entradas (*dyadic*). Cada *token* levava a informação para distinguir os dois casos. No primeiro caso o *token* era simplesmente passado para a unidade de *fetching*. Para nós *dyadic*, uma operação de *matching* era realizada. Uma operação de *matching*poderia ou não resultar na produção de um pacote de saída. Isso explica a taxa variável desta unidade.

A unidade de *fetching* combinava o conjunto dos *tokens* de entrada com a descrição

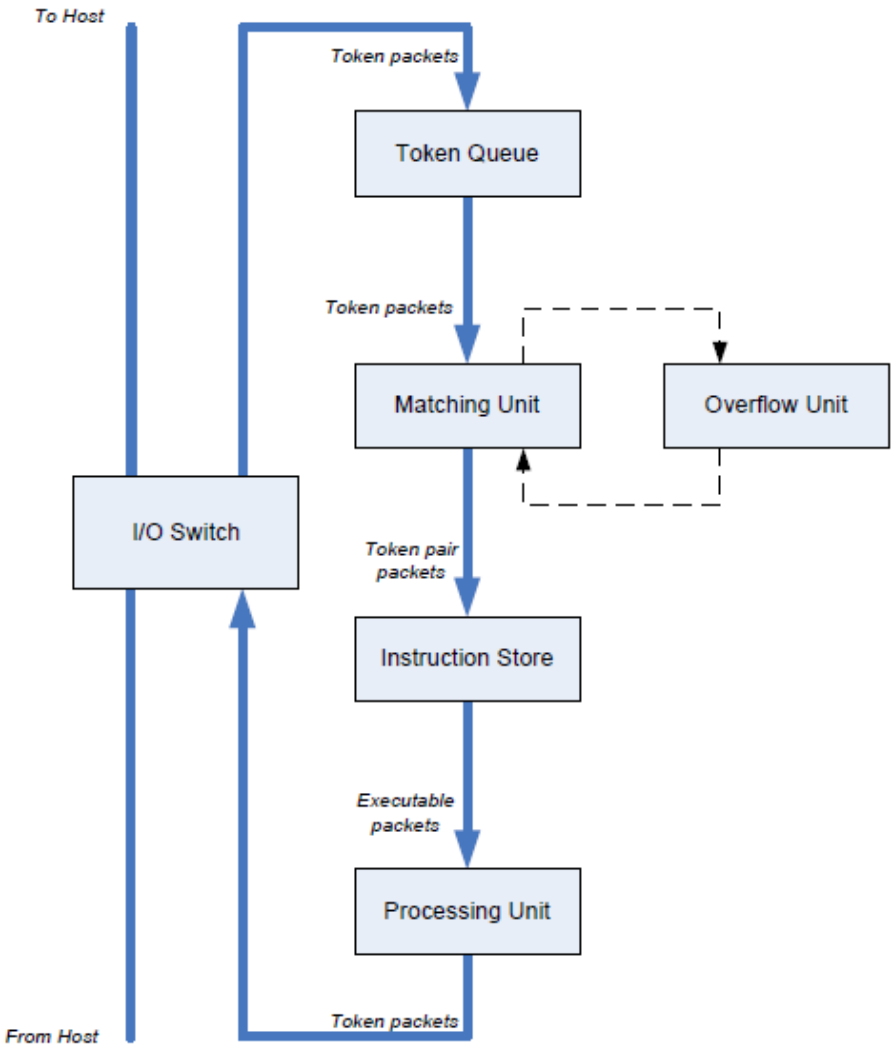


Figura 13: Visão geral da arquitetura de Manchester

do nó de destino em um pacote executável. Cada nó poderia conter até duas descrições de destino, cada uma consistindo de um endereço e uma indicação se o nó de destino era *monadic* ou *dyadic*. Uma instrução *dyadic* poderia se tornar *monadic* trocando-se uma das descrições de destino por uma entrada constante (como um literal) para uma das duas portas de entrada.

A unidade funcional consistia de um preprocessador e um conjunto de elementos funcionais conectados via um distribuidor e um árbitro. O preprocessador executava instruções que requeriam acesso ao contador de memória. A maior parte dos contadores era usada para monitorar desempenho. Um dos contadores, chamado de contador de nome de ativação (*activation name*) era usado para a geração de áreas de *tags* únicas e podia ser manipulado apropriadamente pelo programa. Embora esta não fosse uma operação funcional, o conjunto de instruções era tal que ele próprio não poderia levar

a programas não funcionais. Os elementos funcionais eram processadores divididos em pequenas larguras de *bits* microprogramados. O tempo de processamento por instrução variava de 3 a 30 microssegundos, com uma média de 6 microssegundos. Essa variação, combinada ao fato de uma instrução poder produzir nenhum, um ou dois *tokens*, explica a taxa irregular da unidade funcional.

2.2.2.2 A arquitetura Tagged-token do MIT (ARVIND; NIKHIL, 1990)

A máquina a fluxo de dados do MIT (*MIT Tagged-Token Dataflow Architecture*), assim como a máquina de Manchester foi implementada como um *pipeline* circular. Cada instrução pode ter até dois dados de entrada. Os *tokens* também possuíam *tags* que os associavam ao contexto de código a que pertenciam. Eram da forma $< c.s_p, v >$. O campo $c.s_p$ forma o *tag*. s identifica a instrução de destino, c indica a instância a que pertence o dado e p a porta de entrada do dado (esquerda ou direita).

Os *tokens* são agrupados na unidade de *matching*, cuja funcionalidade é a mesma da unidade de *matching* da máquina de Manchester. Os *tokens* com *matching* bem sucedido são então enviados à unidade de *Fetch*, onde o contexto é utilizado para endereçar um registrador de contexto. No registrador é obtido o endereço base do bloco de código (CBR), e pela soma do CBR com o *offset* s é obtido o endereço da memória de instrução onde estão armazenadas as seguintes informações: *opcode*, uma constante, um *offset* e no máximo dois destinos (s_1 e s_2). Há também associado com o registrador de contexto a memória de constantes (literais), os quais devem ser utilizados pelo bloco de código. Quando uma instrução especifica um *offset*, esse é utilizado para que seja buscado uma constante dentro da memória de constantes.

A ULA processa os valores de entrada da instrução e a constante, (se existir). Ao mesmo tempo são gerados os *tags* a serem utilizados pelos resultados da execução. Se forem produzidos mais *tokens* do que a Unidade de *Matching* pode processar, o excesso é armazenado na fila de *tokens* (*User Queue*).

2.3 Arquiteturas modernas inspiradas no modelo a fluxo de dados

Os conceitos da computação a fluxo de dados vêem sendo utilizados atualmente na concepção de algumas arquiteturas diferenciadas. Entre as mais eminentes estão Wavescalar e TRIPS.

2.3.1 Wavescalar(SWANSON et al., 2007)

WaveScalar é uma ISA (*Instruction Set Architecture*) e um modelo de execução projetado para processadores escalonáveis de baixa complexidade e alto desempenho. Diferente de máquinas a fluxo de dados anteriores, a arquitetura WaveScalar pode prover de maneira eficiente a semântica sequencial que as linguagens imperativas tradicionais (como C/C++) requerem. Para permitir aos programadores expressarem de maneira fácil o paralelismo, WaveScalar suporta o estilo *pthread*, de granularidade alta e o estilo fluxo de dados, de granularidade fina. Além disso, ela permite mesclar as duas abordagens dentro de uma aplicação ou mesmo de uma única função(SWANSON et al., 2007).

Para executar os programas da arquiteura WaveScalar, foi desenvolvido um processador de arquitetura *tile* escalável chamado WaveCache. À medida que um programa executa, o WaveCache mapeia as instruções do programa em seu vetor de elementos de processamento. As instruções permanecem em seus elementos de processamento por várias invocações, e de acordo com a mudança do conjunto de trabalho de instruções, o WaveCache remove as instruções não utilizadas e mapeia novas em seus lugares. As instruções comunicam-se entre si diretamente através de uma interconexão escalável, hierárquica, dentro do próprio *chip*, removendo a necessidade de longos fios e comunicações *broadcast*.

WaveScalar procura resolver a maior dificuldade em relação às arquiteturas a fluxo de dados: a ausência de ordenação de memória, como requerida pelas linguagens populares, como C. Wavescalar impõe essa ordenação de memória introduzindo um novo conceito conhecido como *wave-ordered memory*. Na memória *wave-ordered*, instruções de memória são anotadas com informação extra que as ordena relativamente a outras instruções em porções do grafo de programa chamadas *waves* (ondas). Presentemente há duas formas de acessar memória: a interface *wave-ordered* e a interface não ordenada.

Waves. Na arquitetura WaveScalar os *tags* são chamados *wave-numbers*. Um *token* com um *wave-number* w e um valor v é designado como $w.v$. Ao invés de associar diferentes *wave-number* a diferentes instâncias de instruções específicas (como fazem a maioria das máquinas fluxo de dados), WaveScalar os associa a porções delineadas pelo compilador chamadas *waves*. As *waves* possuem mais de uma entrada e podem conter junções controladas (semelhante aos operadores *deterministic merge*). Elas não podem conter *loops*. A Figura 14 mostra um exemplo de *loop* dividido em *waves*, como mostrado pelas linhas tracejadas.

No topo de cada *wave* há um conjunto de instruções WAVE-ADVANCE (pequenos lo-

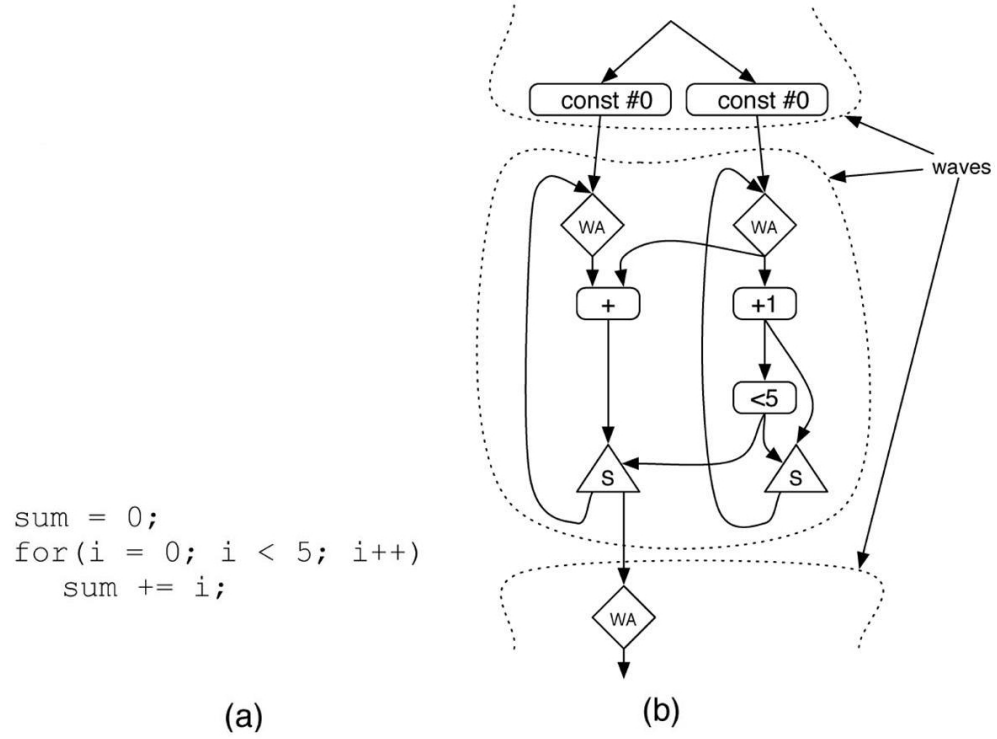


Figura 14: Um *loop* simples (a) e sua implementação em WaveScalar (b) (adaptado de (SWANSON et al., 2007))

.

sangos na figura). Cada qual incrementa o *wave-number* do *token* que passa por ele.

Chamadas de função. As chamadas de função em WaveScalar passam explicitamente os argumentos e um endereço de retorno para a função e disparam sua execução. Ao passar argumentos, cria-se uma dependência de dados entre a função chamadora (*caller*) e a função chamada (*callee*). Para funções indiretas, essas dependências não são estaticamente conhecidas e portanto o grafo fluxo de dados estático da aplicação não as contem. Ao contrário, WaveScalar provê um mecanismo para envio de valores de dados para uma instrução num endereço computado. A instrução que permite isso é chamada INDIRECT-SEND.

A instrução INDIRECT-SEND toma como entrada o dado a ser enviado, um endereço base para a instrução de destino (normalmente um *label*), e o *offset* a partir da base (como um dado imediato).

A Figura 15 contem o grafo para um pequena função e um local de chamada. As linhas tracejadas nos grafos representam as dependências que existem apenas em tempo de execução. A instrução LANDING_PAD provê um destino alvo para um dado enviado via INDIRECT-SEND. Para chamar a função, é necessário usar três instruções INDIRECT-SEND,

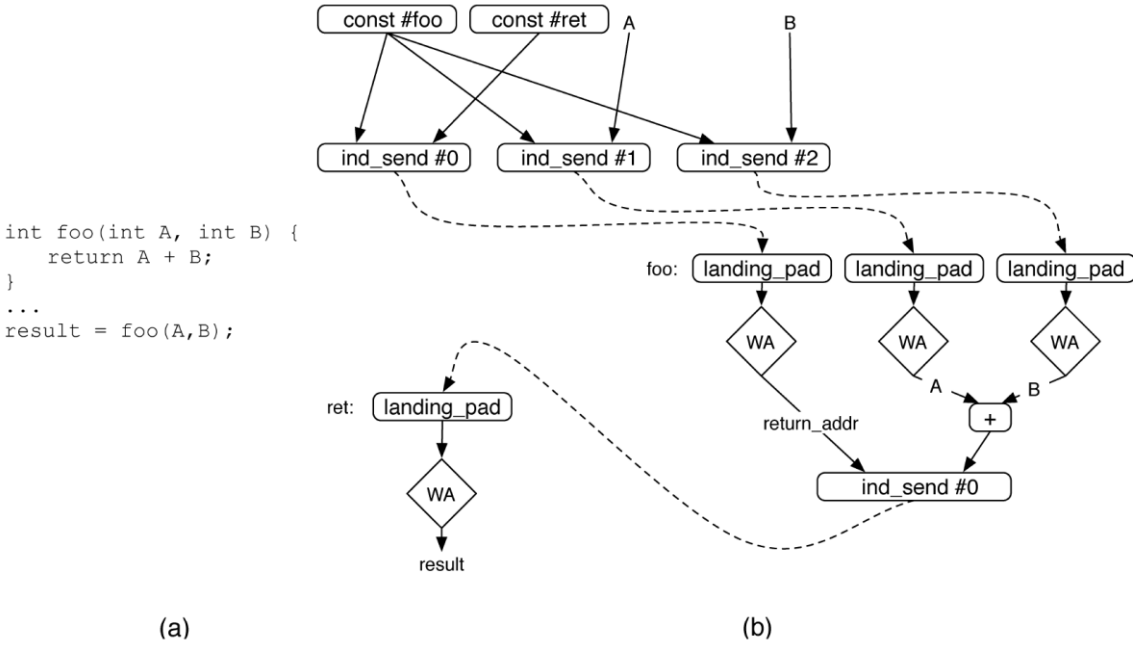


Figura 15: Chamada de função em WaveScalar

duas para A e B e uma para o endereço de retorno, que é o endereço do LANDING_PAD (label `ret` na Figura).

Quando os valores chegam em `foo`, as instruções LANDING_PAD as passam para as instruções WAVE-ADVANCE, que os repassam para o corpo da função (o *calee* inicia uma nova *wave*). Uma vez que a função foi terminada, eventualmente executando muitas *waves*, `foo` usa uma única instrução INDIRECT-SEND para retornar o resultado à instrução LANDING_PAD do chamador. Após uma chamada de função, o chamador inicia uma nova onda usando uma instrução WAVE-ADVANCE.

Memory Ordering. Uma ISA a fluxo de dados mantém a dependência de dados apenas no grafo, e não possui um mecanismo garantindo que as operações de memória ocorram na ordem do programa. A Figura 16 mostra um grafo que demonstra o problema de ordenação de memória. No grafo, o `load` precisa ser executado após o `store`. No entanto, o grafo não expressa essa dependência implícita entre as duas instruções (linhas pontilhadas).

A memória *wave-ordered* soluciona o problema de ordenação de memória usando *waves*. Dentro de cada *wave*, o compilador anota as instruções de acesso à memória para codificar as restriçoes de ordenação entre elas. Como os *wave-numbers* aumentam durante a execução do programa, eles provêem uma ordenação para as ondas em execução. Tomadas juntas, a ordenação de granularidade alta entre as *waves* (através de seus *wave-numbers*),

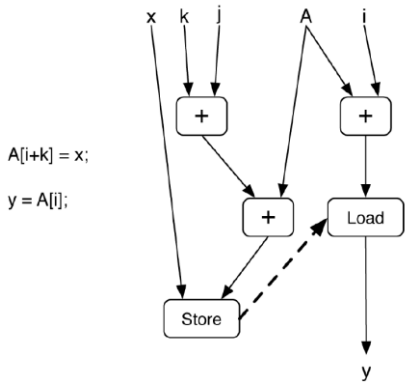


Figura 16: A linha tracejada representa uma potencial dependência de dados implícita entre as instruções LOAD e STORE(SWANSON et al., 2007).

combinadas com a ordenação de granularidade fina dentro de cada *wave*, provêem uma ordem total em todas as operações de memória do programa.

2.3.2 TRIPS (BURGER et al., 2004)

A arquitetura TRIPS *(The Tera-op, Reliable, Intelligently adaptive Processing System*[1]) está sendo desenvolvida com o intuito de se adequar à evolução dos semicondutores na próxima década, avançando para novos níveis de eficiência e desempenho(BURGER et al., 2004). Ela é a primeira instância do conjunto de instruções EDGE (Explicit Data Graph Execution[2]) uma nova classe de ISA.

O conjunto de instruções EDGE é definido por seu modo de execução em blocos atômicos assim como por sua característica de comunicação direta para as instruções dentro de um bloco. Isso sgnifica que um bloco de instruções pode ser carregado e executado como uma unidade. Com a comunicação direta, as instruções dentro de um bloco podem ser enviadas diretamente para instruções dependentes dentro de um mesmo bloco. Esse modo de execução é muito similar ao conceito de fluxo de dados: instruções dependentes esperam operandos de outras instruções antes de iniciar sua execução.

Os quatro objetivos com a implementação TRIPS são os seguintes:

- Aumentar a concorrência;
- Minimizar o consumo de energia tanto quanto possível;
- Minimizar os tempos de comunicação *e*

[1]Sistema de processamento de TERA operações, inteligentemente adaptativo, confiável.
[2]Execução explícita a grafo de dados

- Oferecer uma maior flexibilidade

A arquitetura TRIPS busca estes quatro objetivos combinando os elementos do modelo a fluxo de controle com os benefícios do modelo de execução a fluxo de dados.

Muitas das atuais arquiteturas caem em uma de duas categorias: sistemas de granularidade fina, de aplicação específica ou sistemas de granularidade grossa, para propósitos gerais. Ambas possuem desvantagens: sistemas de granularidade fina possuem um desempenho fraco em aplicações seriais ou fora do padrão, enquanto arquiteturas de granularidade grossa são incapazes de apresentar o mesmo desempenho nas aplicações paralelas. A arquitetura TRIPS busca unir os benefícios de ambas através de uma arquitetura polimórfica, ou seja, a arquitetura TRIPS pode trabalhar tanto em uma forma com granularidade fina, quanto em uma forma com granularidade grossa.

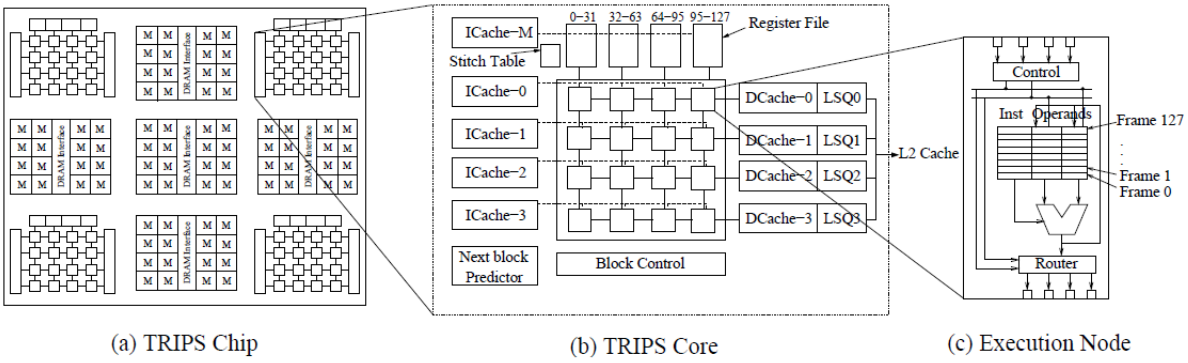


Figura 17: Visão geral da arquitetura TRIPS. Retirado de (SANKARALINGAM et al., 2003)

Essa flexibilidade é conseguida através do uso de núcleos grandes, de granularidade alta que podem ser divididos em núcleos menores para aplicações altamente paralelas. Com essa abordagem, a arquitetura pretende alcançar alto desempenho tanto com aplicações altamente paralelas quanto com aplicações sequenciais, sendo uma máquina de propósito geral.

Os *hiperblocos* são grupos de até 128 instruções, organizadas pelo compilador, que são mapeadas em um vetor de unidades de execução. Esses hiperblocos são criados de maneira que possam ser executados atomicamente, como definido pela arquiteura EDGE. Cabe ao compilador agrupar estes conjuntos de instruções para que os hiperblocos possam ser executados sem nenhuma potencial ramificação (*branch*). Uma vez iniciada a execução de um hiperbloco, as dependências entre as instruções são resolvidas pela comunicação direta dentro do bloco, representando um modelo de execução semelhante ao modelo a fluxo de dados.

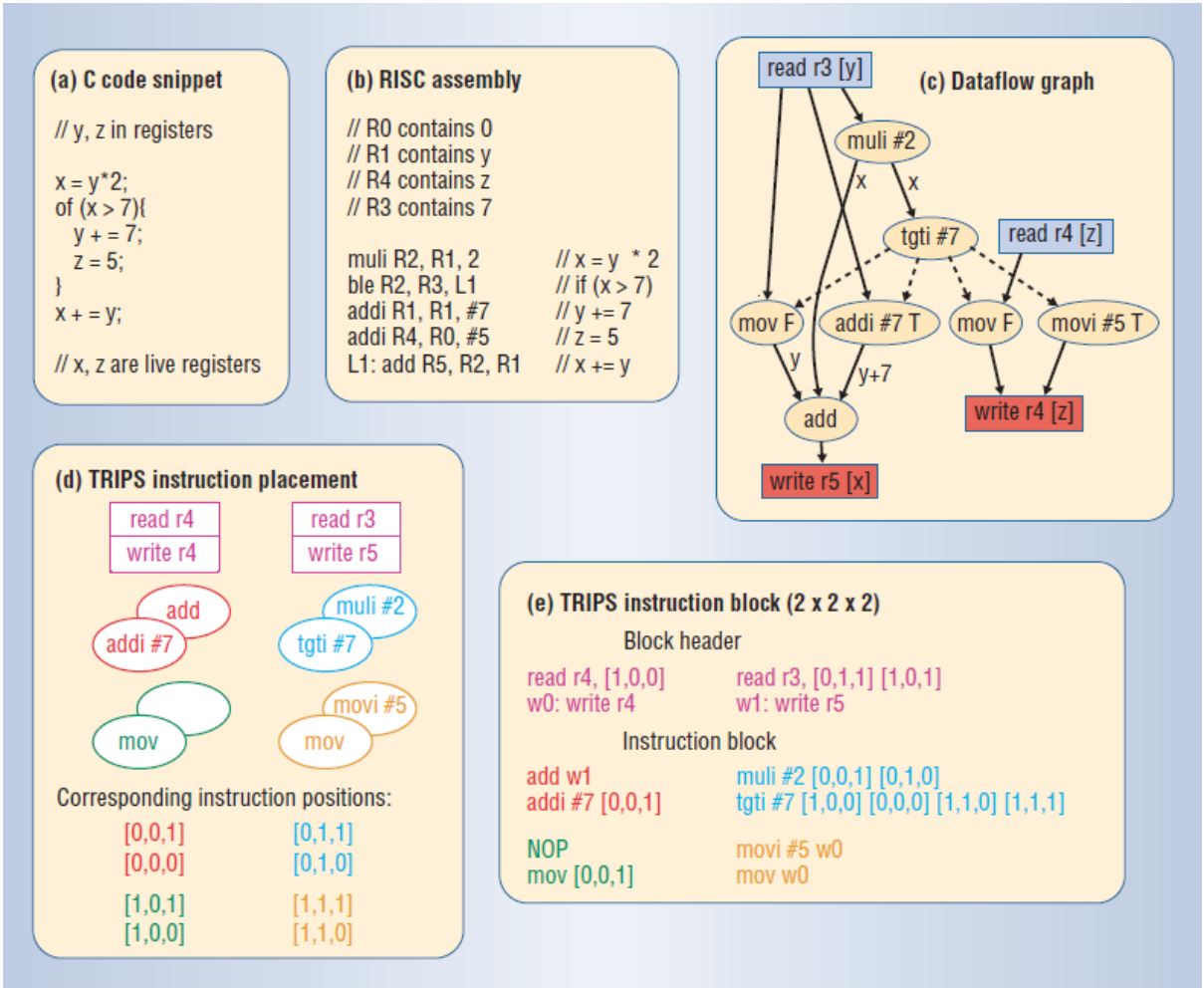


Figura 18: Geração de código para o TRIPS(SANKARALINGAM et al., 2003)

.

A Figura 18 é um exemplo de código no TRIPS. Neste pequeno trecho de código(a) em C, o compilador aloca a entrada y em um registrador(b). No código *assembly* semelhante ao MIPS, a variável x é salva no registro 5(c). O compilador converte o *branch* original para uma instrução de teste e usa o resultado para predicar as instruções dependentes de controle, que aparecem como linhas tracejadas no grafo(d). Cada nó 2x2 na matriz de nós guarda até duas instruções(e). O compilador gera no formato do processador alvo, correspondendo ao mapa de localizações de instruções na parte de baixo de (d).

Predicação de ramificação. *Branch predication* é o método de adiconar um outro operando, chamado de predicado, para toda instrução condicional - uma instrução que pode ou não executar, dependendo do resultado do *branch*. Instruções que produzem valores podem também produzir predicados, que podem então ser transmitidos para essas instruções condicionais. A instrução condicional não pode executar até receber o predicado, uma vez que ele é um operando para a instrução. Baseado no valor do predi-

cado, a instrução pode teminar ou ser executada. O que ocorre efetivamente, é que este mecanismo converte a dependência de controle resultante da instrução de ramificação condicional em dependências de dados para as instruções em seguida à ramificação.

Assim sendo, embora as dependências ainda existam, mais instruções podem ser carregadas sem a preocupação com as predições de ramificação (*branch prediction*). Uma vez que o compilador precisa encontrar instruções sem dependências de controle para agrupar em hiperblocos, este mecanismo de predicação de ramificação se integra à arquitetura TRIPS, aumentando bastante o número de instruções que podem ser agrupadas em um hiperbloco.

Dentro do hiperbloco, as ramificações são tratadas como dependências de dados num método semelhante ao modelo a fluxo de dados e com comunicação direta.

Memória. Uma vez que a implementação TRIPS opera em múltiplos hiperblocos de 128 instruções cada, um poderoso sistema de memória é necessário para prevenir a degradação do desempenho geral do sistema. O sistema de memória usa múltiplos bancos, *buffers* e tabelas para alcançar a alta velocidade necessária pelo sistema. Em particular, cada processador TRIPS possui quatro bancos de memória, que podem suportar até 256 operações de memória.

3 Computação Reconfigurável

A variedade de objetivos das tarefas computacionais levou ao desenvolvimento de três classes de arquiteturas de computadores (BOBDA, 2007):

- As arquiteturas *de propósito geral*, largamente usadas e fundamentalmente baseadas no modelo de Von Neumann;
- As arquiteturas para *domínios específicos*, em que características típicas de um campo de aplicação determinam o projeto de arquiteturas voltadas para o alto desempenho nesse tipo de aplicação. Entre os exemplos mais eminentes estão os DSPs (*Digital Signal Processors*) ou processadores digitais de sinais;
- As arquiteturas *específicas para aplicação*, compreendida pelos ASIP - *Application-Specific Instruction-Set Processors*, desenvolvidas especialmente para uma única aplicação. Em processamento de mídia por exemplo, processadores são desenvolvidos especialmente para algoritmos de um determinado padrão de compressão de vídeo. Estes processadores são incapazes de realizar quaisquer outras tarefas.

O gráfico da Figura 19, retirado de (BOBDA, 2007) mostra que em geral há um compromisso entre flexibilidade e desempenho entre essas classes. Os computadores de propósito geral, por exemplo possuem grande flexibilidade, o que na prática quer dizer que podem ser aproveitados para resolver virtualmente qualquer tarefa de computação, no entanto, com um desempenho relativamente baixo, quando comparados às outras classes, quando usadas para suas aplicações alvo. À medida que o campo de aplicação de um processador se restringe, diminui-se também sua flexibilidade, ou seja, mais difícies se tornam de ser aproveitados para outras tarefas, o que ocorre às custas de um maior desempenho nas tarefas visadas.

A computação reconfigurável se insere neste contexto como o estudo da computação realizada em dispositivos capazes de serem *configurados* e *reconfigurados* para executarem determinada tarefa. Por configuração entende-se o processo de se alterar a estrutura

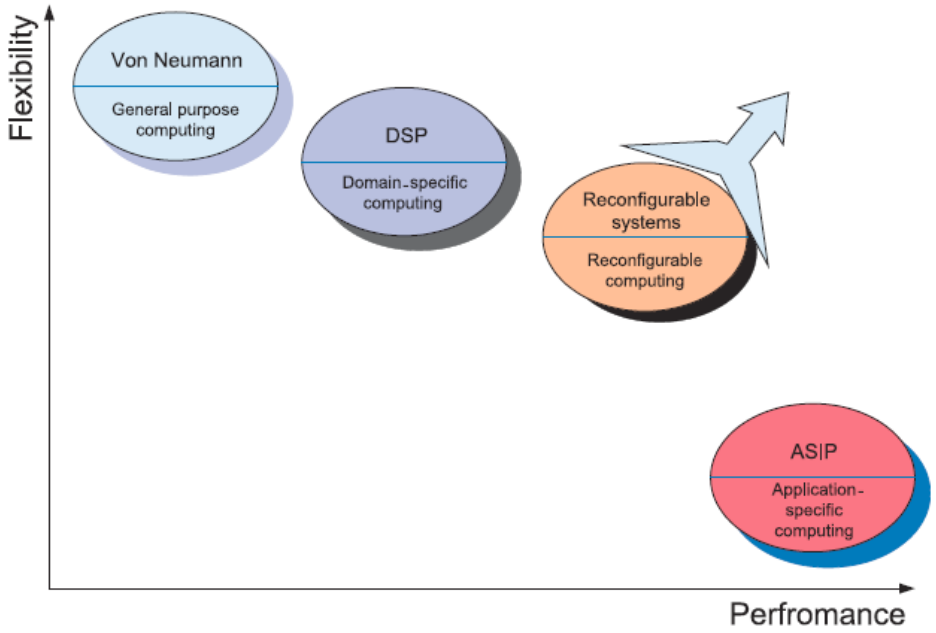


Figura 19: Gráfico mostrando a relação *desempenhoxflexibilidade* para as classes de máquinas computacionais.

interna do dispositivo, em geral ligando-se ou desligando-se determinados pontos de interconexão(COMPTON; HAUCK, 2002) de forma a atender da melhor forma possível uma determinada tarefa de computação, no momento de sua ligação. Respectivamente, por reconfiguração entende-se essa mesma alteração realizada em tempo de execução.

3.1 Arquiteturas de computação reconfigurável

Em (RADUNOVIC; MILUTINOVIC, 1998), um artigo relativamente antigo em relação aos últimos avanços da computação reconfigurável, é apresentada uma classificação das arquiteturas reconfiguráveis. A computação reconfigurável já existia mesmo antes do advento (e do grande avanço) dos FPGAs. Mas como já foi dito, apenas com estes últimos é que ganhou mais força. Para os autores, de uma forma geral, a arquitetura de um sistema reconfigurável pode ser descrita como *uma rede reconfigurável de células lógicas básicas.*

A taxonomia proposta por (RADUNOVIC; MILUTINOVIC, 1998) usa nomes da mitologia grega e faz a classificação dos sistemas usando os seguintes critérios:

- *Objetivo do sistema reconfigurável.* Duas ramificações foram identificadas aqui pelos autores: sistemas visando *tolerância a falhas* e sistemas visando *maior desempenho*;

- *Granularidade do bloco de lógica reconfigurável.* Os autores classificam os sistemas como de granularidade *grossa*, *média* e *fina*;

- *Acoplamento com o sistema* host. Aqui os sistemas podem ser *dinâmicos*, *estáticos fortemente acoplados* ou *estáticos fracamente acoplados.*

(MESQUITA, 2002) usou uma árvore para apresentar de maneira mais didática essa classificação, como se vê na Figura 20.

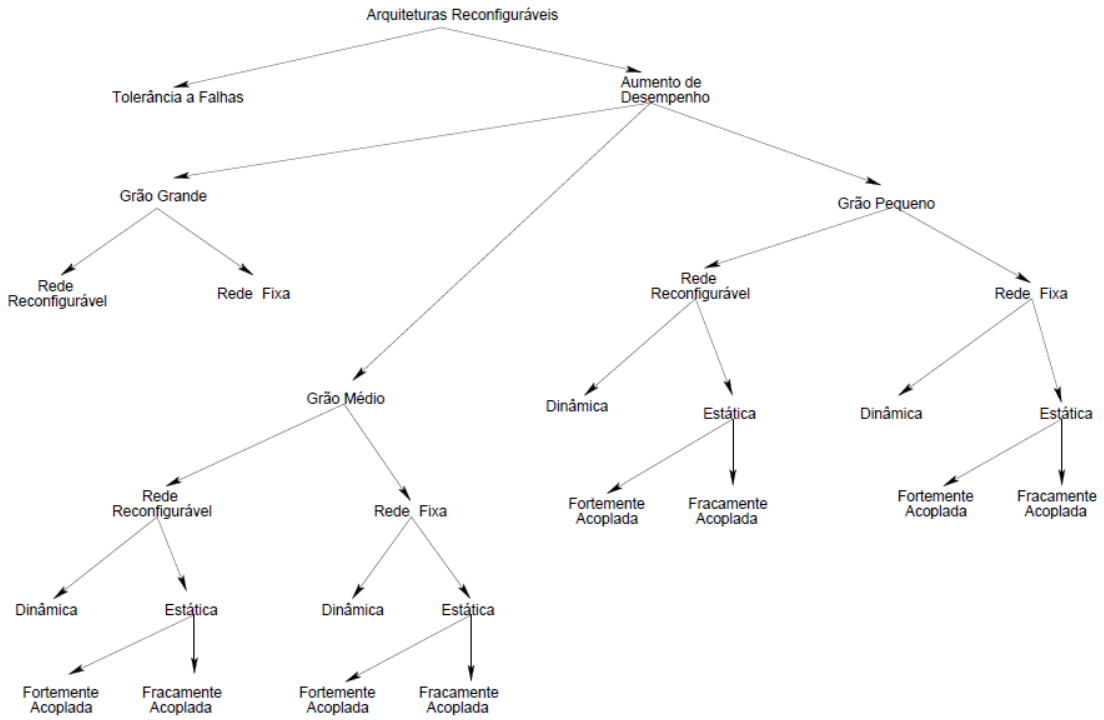


Figura 20: Classificação de (RADUNOVIC; MILUTINOVIC, 1998) em forma de árvore (retirado de (MESQUITA, 2002))

A próxima seção trata das arquiteturas classificadas por estes autores até a data de seu artigo, os quais trataremos por *primeiros sistemas reconfiguráveis*

3.1.1 Primeiros sistemas de computação reconfigurável

Os sistemas aqui apresentados excluem qualquer abordagem do ramo de *tolerância a falhas.* Todas as arquiteturas visam aumento de desempenho.

3.1.1.1 MATRIX (MIRSKY; DEHON, 1996)

O sistema MATRIX pode ser compreendido como sendo de granularidade média, com rede de interconexão externa, estático e fracamente acoplado(RADUNOVIC; MILUTINOVIC, 1998). MATRIX é a sigla para *Multiple ALU architecture with reconfigurable interconnect experiment*, ou arquitetura experimental de múltiplas ULAs com interconexão reconfigurável. Trata-se de um esforço acadêmico para uma unidade que compreende uma matriz de unidades funcionais básicas (BFUs), onde cada uma contém uma ULA de 8 *bits*, uma memória de 256 palavras e lógica de controle. A ULA contém as operações aritméticas e lógicas padrão, além de um multiplicador. Uma cadeia de *carry* configurável entre ULAs adjacentes pode ser configurada para operações de palavras maiores. Há ainda estruturas lógicas especiais para reconfiguração da conexão da ALU com a rede de comunicação. Com essas características, uma unidade funcional básica pode funcionar como uma memória de instruções, memória de dados, uma ULA combinada com banco de registros ou ainda como uma ULA independente. As instruções podem ser roteadas para várias ULAs. A infraestrutura de roteamento provê três níveis de barramento de 8-*bits*. Oito ligações aos vizinhos mais próximos, quatro aos segundos vizinhos mais próximos e linhas globais ao longo de uma coluna ou linha inteira. Uma desvantagem dessa arquitetura é seu baixo desempenho para operações em nível de *bit* e *byte*.

3.1.1.2 RAW (WAINGOLD et al., 1997)

De acordo com a classificação apresentada, este é um projeto voltado para mmaior desempenho, de granularidade grossa (os blocos configuráveis são processadores), com rede de interconexão externa. O projeto é baseado em uma unidade de processamento constituída de blocos[1] mais simples, altamente conectados e replicados. Cada um desses elementos possui também associado um *switch* com sua própria memória, controlando a reconfiguração da rede de interconexão (Figura 21).

O sistema RAW ocupa maior parte dos espaços com unidades de execução, procurando atingir maior grau de paralelismo e *clocks* mais altos. Isso também simplifica o processo de verificação. A presença de lógica de reconfiguração de granularidade fina (blocos CL da Figura 21) permite avaliação de operações em nível de *bit* e *byte* no próprio *tile*. Este sistema sofre de ineficiência no caso de computações altamente dinâmicas, ou seja, nos

[1]Chamados em inglês de *tiles*, literalmente: ladrilhos (ou azulejos). Dá a idéia de vários pedaços idênticos que juntos compõem a unidade. O termo em geral é usado quando se refere a uma unidade equivalente a um processador simples

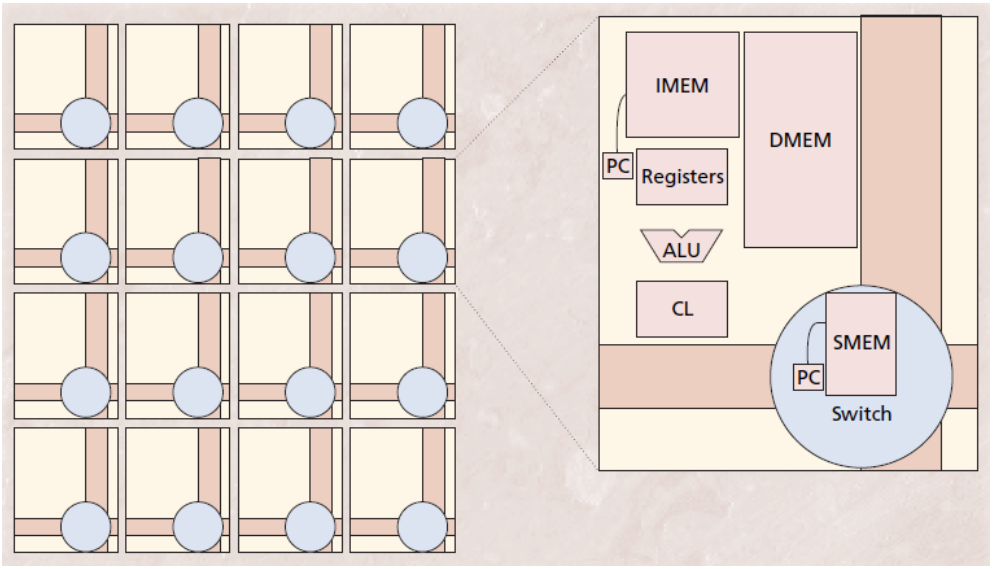


Figura 21: Um processador RAW é construído de múltiplos blocos idênticos. Cada bloco contém memória de instrução (IMEM), memória de dados (DMEM), unidade aritmética e lógica (ALU), registradores, lógica configurável(CL) e um *switch* programável associado com sua memória de instrução (SMEM).

casos em que as dependências de dados e de roteamento não podem ser resolvidas em tempo de compilação.

3.1.1.3 RaPiD(EBELING; CRONQUIST; FRANKLIN, 1996)

RaPiD (*reconfigurable pipelined datapath*) é um representante das arquiteturas fortemente acopladas de granularidade média. Ele consiste de um vetor de células localizadas no caminho de dados (*datapath*). Todos os elementos estão conectados ao barramento de dados. Cada célula possui ULA, registradores, multiplicadores e memória. Assim como as outras arquiteturas apresentadas até aqui, possuem um maior desempenho para operações aritméticas complexas, devido à sua granularidade maior do que a dos FPGAs(RADUNOVIC; MILUTINOVIC, 1998).

3.1.1.4 Firefly(GOEKE et al., 1997)

O sistema Firefly é um sistema evolutivo baseado no modelo de autômatos celulares, que tende a produzir oscilações uniformes. Os princípios de computação evolutiva já foram usados em *software*. Este é um dos primeiros trabalhos a implementar estes princípios em *hardware*. Esta arquitetura pode ser classificada como de granularidade fina, com conexão externa e dinâmica.

3.1.1.5 SPLASH-II(GOKHALE et al., 1990)

O sistema SPLASH-II é um sistema feito de vários FPGAs soldados em placas separadas e conectados para constituir um computador massivamente paralelo. Nessas placas, chamadas de *array boards*, os FPGAs podem ser reconfigurados para cumprir uma tarefa específica. Devido a esse tipo de ligação, trata-se de um sistema *fracamente acoplado*, porém de granularidade fina, com rede de interconexão externa e estático. Seu antecessor, SPLAH-I, foi um dos primeiros sistemas de computação reconfiguráveis. As *array boards* são conectadas a um *host* por uma placa de interface e um barramento de sistema. Cada placa possui 17 FPGAs, sendo 16 para processamento e 1 para implementação da interconexão.

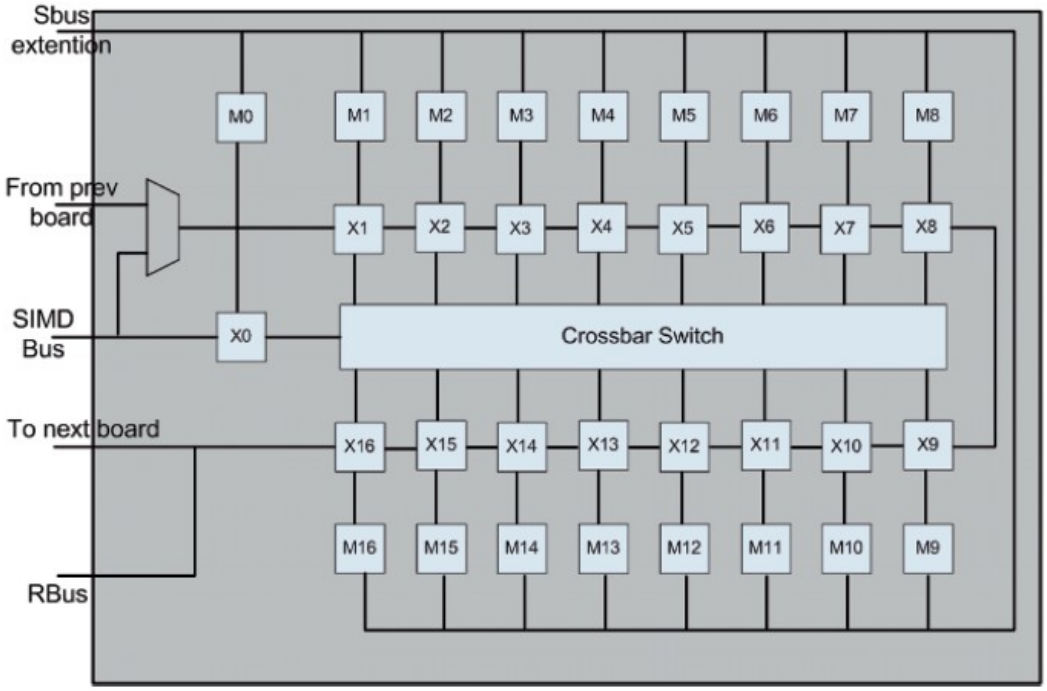


Figura 22: Arquitetura SPLASH-II.(retirado de (BOBDA, 2007))

3.1.1.6 DECPeRLe-1(VUILLEMIN et al., 2002)

DECPeRLe é também um projeto pioneiro da computação reconfigurável, que cai na mesma categoria do sistema SPLASH. É constituído por um *host*, conectado a uma placa com 16 FPGAs através de um barramento de entrada e saída. O acesso do *host* é similar a um acesso de memória, com escritas e leituras dos dados para processamento pelos FPGAs. A dificuldade de configuração e a falta de ferramentas de compilação são grandes desvantagens desse sistema(RADUNOVIC; MILUTINOVIC, 1998).

3.1.1.7 PRISM(ATHANAS; SILVERMAN, 1993)

O PRISM-I foi desenvolvido para provar a viabilidade de se usar instruções configuráveis com um processador de propósito de geral. Para tanto foi construída uma placa com FPGAs, ligada através de um barramento ao processador principal. Apesar de o desempenho não ser satisfatório, devido à baixa velocidade de comunicação, a viabilidade de se ter instruções configuráveis foi demonstrada. A versão PRISM-II atacou os problemas de velocidade de comunicação, conseguindo resultados melhores. Este sistema pode pode ser considerado na mesma categoria de SPLASH e DECPeRLe. Foi também um pioneiro que abriu grande campo para o estudo das instruções configuráveis.

O projeto RENCO, que aqui será apenas citado, seguiu uma abordagem semelhante à dos projetos SPLASH, DECPeRLe e PRISM, sendo constituído por uma placa externa com quatro FPGAs, ligado a um *host*.

3.1.1.8 GARP(HAUSER; WAWRZYNEK, 1997)

GARP é um exemplo de sistema com granularidade fina, rede de interconexão externa, estático, fortemente acoplado. Foi proposto para atacar os principais problemas das arquiteturas anteriores, baseadas em placas externas com FPGAs. Entre esses problemas pode-se citar a falta de memória interna para armazenar temporariamente os dados a serem computados, a lentidão para a configuração do sistema e a falta de um padrão para o acesso entre processador e lógica reconfigurável. Os autores sugeriram que o sistema estivesse em uma mesma pastilha. O sistema foi concebido como um processador *imerso* em lógica configurável. Apesar de um *chip* GARP nunca ter sido fabricado, é interessante salientar que os atuais FPGAs embarcados, como são conhecidos, herdam muitas das idéias dessa abordagem.

Além da classificação apresentada, há outras propostas para situar em perspectiva as arquiteturas de computação reconfigurável. Em (MöLLER, 2005) faz-se uma apresentação interessante sobre (HAUCK, 1998). A Figura 23 extraída de (HAUCK, 1998) mostra as diversas posições em que a lógica programável pode estar presente em um sistema reconfigurável. Este posicionamento pode ser usado para fazer uma classificação das várias arquiteturas.

Os primeiros três tipos de arquiteturas (A, B e C) apresentadas na Figura 2 são chamadas de processadores reconfiguráveis e possuem a lógica reconfigurável conectada a um processador mestre. As arquiteturas podem ser fortemente acopladas (RFUs (A)) ou

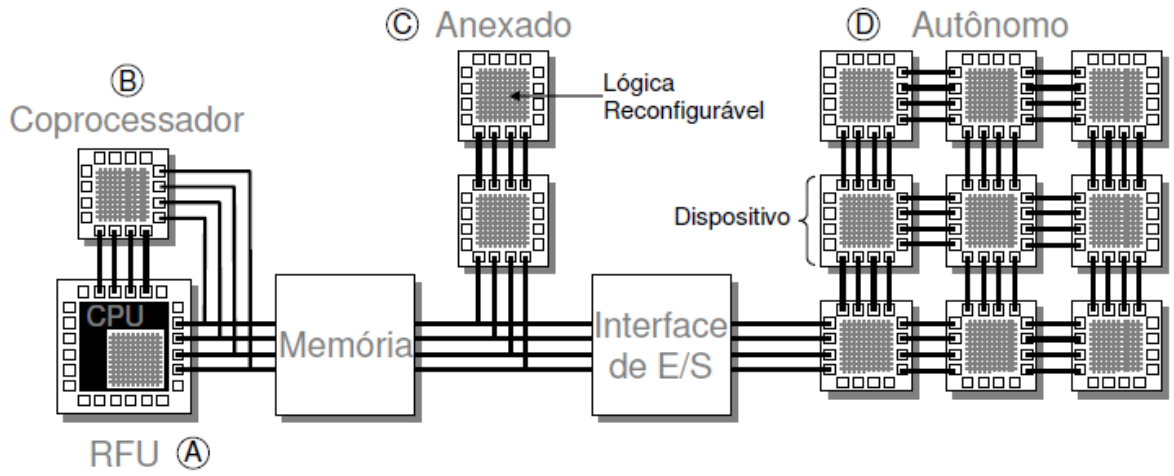


Figura 23: Posicionamento da lógica programável em arquiteturas de computação reconfigurável ((HAUCK, 1998))

fracamente acopladas (coprocessadores (B) e anexados (C)), devido à forma de ligação da lógica reconfigurável ao processador mestre. Em (A), a lógica reconfigurável é posicionada dentro do processador, visando estender seu conjunto de instruções e é acessada com baixa latência devido à proximidade entre a lógica reconfigurável e o processador. Em (B), a lógica reconfigurável possui uma conexão dedicada com o processador, sendo esta tratada como um coprocessador. Neste caso a lógica reconfigurável pode, opcionalmente, acessar diretamente a memória do processador ou recursos de entrada e saída. Em (C), a lógica reconfigurável não possui uma conexão dedicada com o processador. Nesse caso a comunicação é compartilhada com outros recursos do processador.

3.1.2 Sistemas mais recentes de computação reconfigurável

A rápida evolução dos FPGAs determinou o passo na pesquisa dos sistemas de computação reconfigurável. As primeiras abordagens, apresentadas na seção 3.1.1 são suplantadas atualmente por FPGAs comerciais, principlamente os conhecidos como FPGAs embarcados (BOBDA, 2007). Entre estes, o FPGA Virtex-4, da Xilinx é um eminente representante. Este tipo de circuito integrado provê os mesmos componentes de um FPGA típico moderno (blocos de lógica configurável, blocos de memória, blocos de entrada e saída) além de pelo menos um núcleo de processador, imerso na lógica programável.

Além dos FPGAs embarcados, há outras propostas de arquiteturas modernas de granularidade grossa. (BOBDA, 2007) as divide entre as que se baseiam em um modelo a fluxo de dados, e aquelas cujos blocos configuráveis se interonectam através de uma rede.

Entre as primeiras, pode-se citar a arquitetura PACT XPP, a arquitetura NEC DRP e o *picoCHIP*.

3.1.2.1 PACT XPP(BAUMGARTE et al., 2003)

A empresa PACT introduziu sua *eXtreme Processing Platform* (XPP) no ano de 2000. Desde 2006, a versão PACT XPP III está disponível no mercado. Trata-se de uma arquitetura de processamento de dados baseada em uma matriz hierárquica de elementos computacionais de granularidade grossa e configuráveis - PAE *Processing Array Elements*e uma rede interna de comunicação orientada a pacotes (PACT XPP TECHNOLOGIES, 2006).

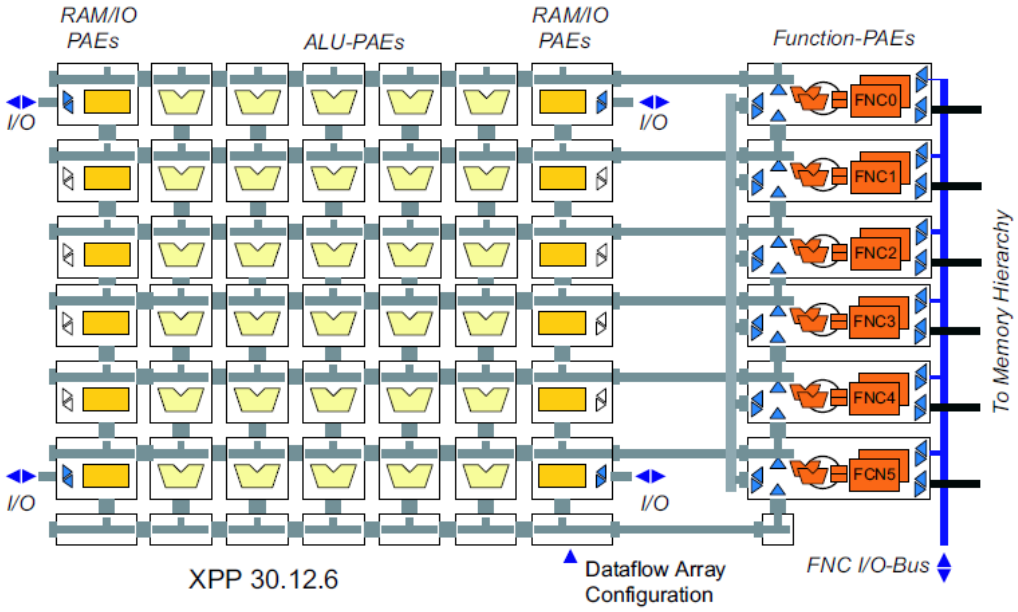


Figura 24: Arquitetura PACT XPP-III

Nessa arquitetura, o código irregular, predominantemente de controle (sem laços ou paralelismo em nível de *pipeline*) é mapeado em um ou vários elementos *Function-PAE* (FNC-PAE). Eles trabalham de forma concorrente e são constituídos de núcleos processadores sequenciais de 16 *bits*, otimizados para algoritmos sequenciais que requerem uma grande quantidade de ramificações (*branches*). Como exemplo tem-se decodificação de fluxos de *bits* ou encriptação. Um FNC-PAE executa até oito operações de ULA e uma operação especial (como multiplicação, por exemplo) em um ciclo. As operações podem ser encadeadas em até quatro níveis, ou seja, a saída de uma operação é diretamente a entrada de outra. Essa operação pode ser combinada também com predicação de *branches* (ver seção 2.3.2). Mecanismos especiais também podem fornecer *jumps* em apenas um ciclo. Essas unidades são capazes de fornecer um desempenho alto a uma frequência de operação baixa(PACT XPP TECHNOLOGIES, 2006).

Algoritmos sobre fluxos regulares de dados, como filtros e transformadas são eficientemente implementados na região a fluxo de dados da arquitetura. Grafos arbitrários podem ser diretamente mapeados em ULAs e conexões de roteamento, resultando em uma implementação paralela com *pipeline*. Sinais de eventos distribuídos dentro da matriz de fluxo de dados adicionam flexibilidade para algoritmos menos regulares uma vez que podem ser usados para controlar as operações em fluxo. Uma outra característica bastante importante é a presença de um mecanismo de reconfiguração dinâmica. Os PAEs podem ser reconfigurados enquanto seus vizinhos estão processando dados. Algoritmos inteiros podem executar independentemente em diferentes partes da matriz.

3.1.2.2 NEC DRP

O processador dinamicamnte reconfigurável (DRP - *Dynamically reconfigurable processor*), da NEC, possui uma estrutura composta de:

- Uma matriz de elementos de processamento orientados a *byte*;
- Uma rede reconfigurável de interconexão entre esses elementos;
- Um *sequencer* - controlador de transição de estados, que pode ser programado como uma máquina de estados finitos para controlar o processo de reconfiguração dinâmica no dispositivo;
- Blocos de memória, distribuídos ao redor do *chip*;
- Várias interfaces e controladores de memória.

A Figura 26 mostra o elemento de processamento(EP) do DRP. Ele contém uma ULA para aritmética e lógicas orientada a *byte*, um gerenciador para lidar com seleção, deslocamento, máscara e geração de constantes de *bytes*. O operando pode ser carregado do banco de registros do EP ou coletado diretamente de suas entradas. Os resultados podem ser armazenados no próprio banco ou enviados para fora. A configuração é realizada pelo controlador de transição de estados, que preenche um ponteiro para um registrador de instrução correspondente de acordo com o modo de operação da ULA. Ao possuir muitos registradores de configuração, torna possível armazenar dados de configuração diretamente nas proximidades dos EPs e permitir uma troca rápida de uma configuração para outra.

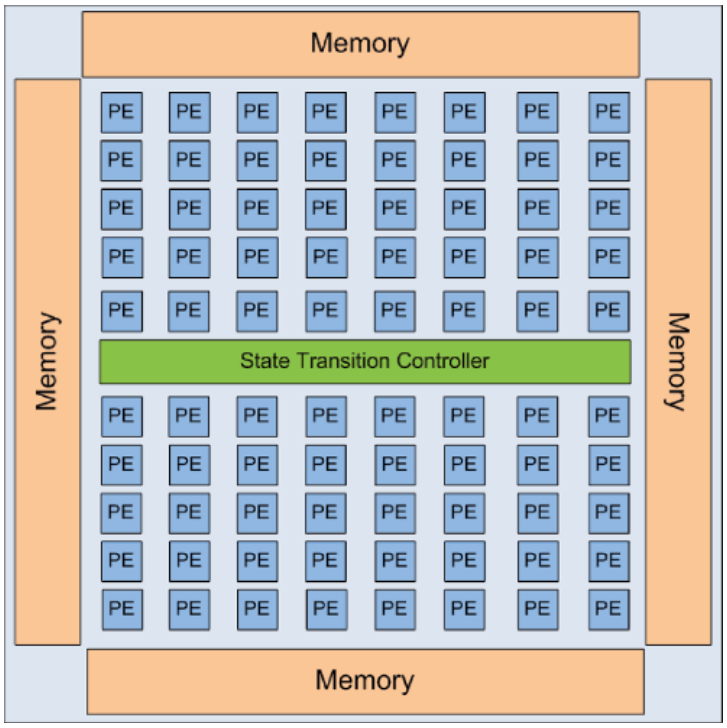


Figura 25: Arquitetura DRP da NEC. Retirado de (BOBDA, 2007)

3.1.2.3 *picoCHIP*

O *picoChip* consiste de uma matriz de *picoArray* e um conjunto de diferentes interfaces para conexão externa de vários módulos como memória e processadores. O núcleo *picoArray* possui uma estrutura similar ao PACT XPP e ao DRP. A diferença está no fato de a conexão entre os elementos de processamento se dar por interseções de colunas e linhas. Como exemplo, o *chip* PC102 contém centenas de *picoArrays*, cada um com um processador de 16 *bits* com memória de dados local conectadas por uma estrutura programável. A arquitetura é composta de quatro tipos de elementos de processamento, todos possuindo uma estrutura básica comum, mas otimizados para tarefas diferentes: o padrão (STAN), o de controle (CTRL), o de memória e a unidade de aceleração de função. Esta arquitetura foi desenvolvida visando as aplicações sem fio típicas de telefonia celular. Pode ser reconfigurada em tempo de execução, e múltiplos elementos da matriz podem ser agrupados para resolver determinada tarefa.

3.2 Arquitetura dos FPGAS

Atualmente, os FPGAs (*Field Programmable Gate Arrays*) - Matrizes de Portas Lógicas Programáveis em Campo - estão consolidados como os principais dispositivos

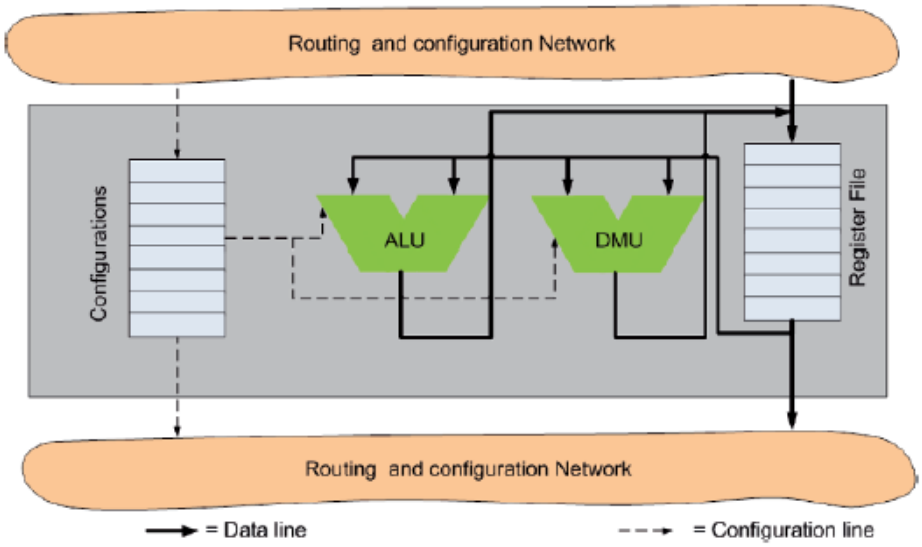


Figura 26: Um elemento de processamento da arquitetura DRP. Retirado de (BOBDA, 2007)

pelos quais se torna possível a computação reconfigurável. Graças ao grande avanço da tecnologia de fabricação de circuitos integrados nos últimos anos, hoje um único *chip* deste tipo pode conter milhões de portas lógicas.

Inicialmente usados para a implementação de circuitos simples de lógica digital, hoje a quantidade de recursos de um único FPGA, mesmo os pertencentes a linhas mais econômicas[2], é suficiente para implementar sistemas inteiros (*SoC - System on a Programmable Chip*).

Um FPGA é um circuito integrado caraterizado por ser composto por três elementos (Figura 27):

- Células de lógica programável;
- Uma rede de interconexão entre essas células;
- Um conjunto de células de entrada e saída ao redor do dispositivo.

As células lógicas, também chamadas de *blocos lógicos configuráveis* ou simplesmente *blocos lógicos* podem ser configuradas para implementar uma função lógica combinacional qualquer. Funções maiores são implementadas configurando-se outras células lógicas (portanto dividindo as funções) e interconectando-se essas células através da configuração da rede de interconexão interna ao dispositivo. As células lógicas também contêm elementos de memória, que podem ser desde simples *flip-flops* até blocos de memória.

[2]Linha Cyclone da Altera e linha Spartan da Xilinx, por exemplo

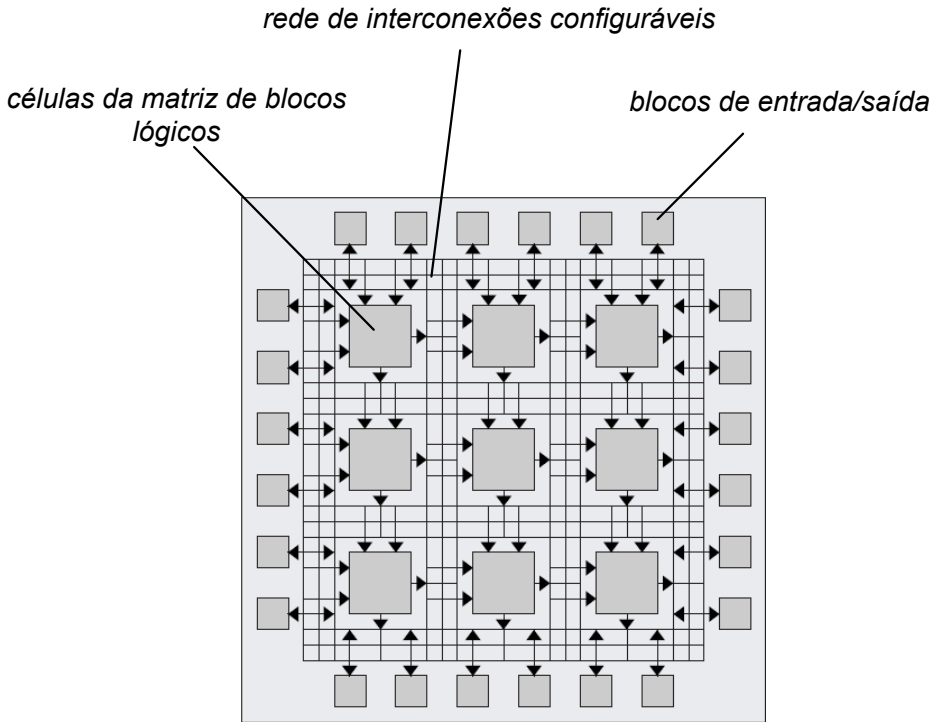


Figura 27: Visão abstrata da arquitetura de um FPGA (adaptado de (HAUCK; DEHON, 2008))

As células de lógica configurável dos FPGAs são compostas por elementos simples de computação que permitem a implementação de qualquer função lógica. Juntamente com os *flip-flops* ou blocos de memória, eles podem implementar a função de qualquer circuito digital, combinacional ou síncrono. Essa característica é a que torna os FPGAs os dispositivos mais usados na implementação de sistemas de computação reconfigurável.

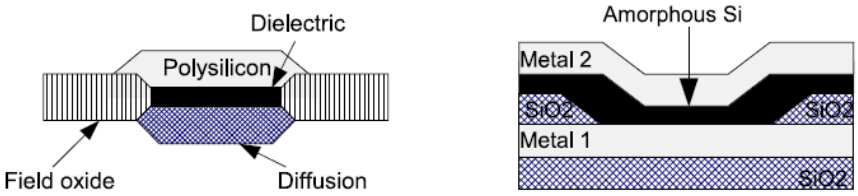
Os elemetos básicos de implementação de funções lógicas nos FPGAs, presentes nos seus blocos lógicos são os *multiplexadores* e as LUTs (*Look-Up Tables*). Qualquer função booleana complexa pode ser implementada por um conjunto de multiplexadores ligados entre si. Uma LUT, por sua vez, é um conjunto de células de memória que podem mapear diretamente a tabela verdade de uma função lógica combinacional. A entrada de *n bits* define um endereço de memória, e o valor contido no endereço constitui a saída da função para aquela entrada. Da mesma forma que os multiplexadores, funções combinacionais complexas podem ser implementadas pela combinação de LUTs menores.

3.2.1 Tecnologias de configuração dos FPGAs

Entre as tecnologias usadas para a realização física das interconexões em um FPGA estão a tecnologia *antifuse* e aquelas baseadas em memória.

3.2.1.1 *Antifuse*

Um FPGA baseado na tecnologia *antifuse*[3] possui elementos especiais em cada ponto de interconexão configurável. Tratam-se de elementos cujos os dois terminais de contato estão em camadas (*layers*) diferentes, separados por um material dielétrico. Nesse estado, não permitem portanto, a passagem de corrente, e comportam-se idealmente como chaves abertas. Uma tensão relativamente alta entre esses terminais é capaz de realizar a *fusão* do material dielétrico, fazendo com que a interligação entre os dois terminais se concretize. As ligações baseadas nessa tecnologia possuem impedância menor do que as realizadas por métodos de memória, o que implica em atrasos menores, permitindo ao circuito final operar em frequências maiores. Sua maior desvantagem, no entanto, é o fato de serem permanentes, ou sejam, a configuração só pode ocorrer uma vez. Este fato tornam os dispositivos dessa família pouco interessantes para as aplicações de computação reconfigurável.



(a) Q-Logic Vialink antifuse (b) Actel PLICE antifuse

Figura 28: Tecnologias *antifuse*. Em (a), *Vialink*, da empresa QuickLogic; em (b), PLACE, da Actel.(BOBDA, 2007)

3.2.1.2 Memória

A maneira mais utilizada em FPGAs comerciais para armazenar a informação de de configuração é através de RAM estática volátil, ou SRAM. Esse método tem sido popular por prover reconfiguração rápida e infinita em uma tecnologia bastante conhecida. Nesse método as interconexões podem estar abertas ou fechadas dependendo do estado dos transistores que a compõem. O estado dos transistores é determinado, por sua vez, por um *bit* de uma célula de memória. Além das interconexões, os *bits* de memória SRAM determinam o estado das LUTs e a seleção dos multiplexadores dos blocos lógicos, servindo, assim, também para a configuração da lógica do dispositivo. As desvantagens da SRAM aparecem na forma de consumo de potência e volatilidade dos dados. Comparada

[3]Uma possível tradução seria antifusível, uma vez que o conceito é o de um *fusível* normalmente aberto.

com as outras tecnologias, a célula de memória SRAM é grande (6-12 transistors)(HAUCK; DEHON, 2008) e dissipa potência estática significante devido às correntes de fuga parasitas. Outra desvantagem é que a SRAM não mantém o seu conteúdo sem alimentação. Isso significa que na partida o FPGA não está configurado e precisa ser programado com lógica e memória armazenada fora do dispositivo. Isso pode ser alcançado com dispositivos não voláteis externos que armazenam a configuração e - caso necessário - um microcontrolador para efetuar o procedimento de programação. A maioria dos FPGAs atuais, no entanto, possuem já implementada em sua estrutura a funcionalidade necessária para carregar de uma memória externa a configuração, sem a necessidade de microprocessadores ou outros mecanismos equivalentes. A memória não volátil externa, no entanto, ainda é imprescindível.

Embora menos popular do que SRAM, várias famílias de dispositivos usam memória Flash para manter a informação de configuração. As memórias Flash diferem por serem não voláteis e poderem ser escritas somente um número finito de vezes. Em contraste com os FPGAs baseados em SRAM, estes FPGAs não perdem sua configuração quando deixam de ser alimentados e estão prontos imediatamente após o momento da partida. Não necessitam também, de nenhum elemento externo para programação.

Uma célula de memória Flash pode ser feita com uma quantidade menor de transistores em relação a uma célula SRAM. Essa tecnologia pode trabalhar estaticamente com menos potência, por haver menos transistores para contribuírem com as correntes parasitas.

As desvantagens vêem na forma das técnicas necessárias para escrever nas memórias Flash. Elas possuem um ciclo de vida limitado de número de escritas e tipicamente são mais lentas. O número de ciclos varia com as tecnologias, mas é tipicamente de centenas de milhares a milhões de ciclos. A maioria requer também tensões maiores comparadas aos circuitos comuns. Eles necessitam de estruturas especiais no *chip*, como circuitos de elevação de tensão para serem capazes de realizar a escrita em Flash.

3.3 Reconfiguração parcial em FPGAs Xilinx

A Xilinx é a única empresa atualmente que suporta com ferramentas e dispositivos a *Reconfiguração Parcial* (PR - *Partial Reconfiguration*) de FPGAs. A reconfiguração parcial é útil para sistemas com múltiplas funções que podem compartilhar no tempo os recursos de um mesmo FPGA(XILINX, 2008). Nesses sistemas, uma seção do FPGA opera

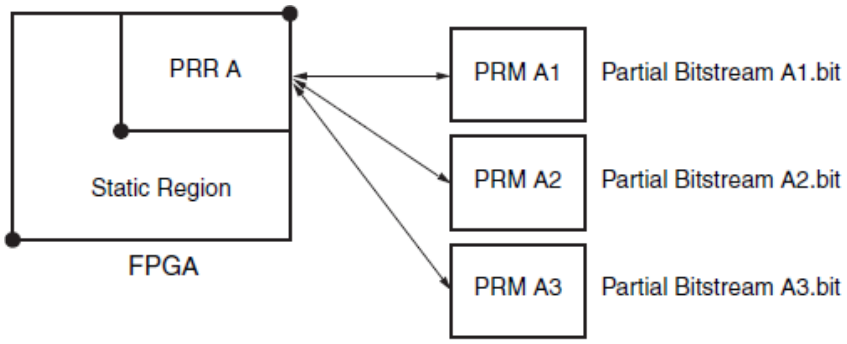


Figura 29: Uma região parcialmente reconfigurável (PRR) A pode ser carregada com os módulos parcialmente reconfiguráveis A1, A2, A3.

continuamente enquanto outras seções são desabilitadas e parcialmente reconfiguradas para prover nova funcionalidade.

A abordagem convencional de configuração de FPGAs, faz uso de múltiplos *bitstreams* completos para alterar a funcionalidade de um dispositivo. No entanto, *bitstreams* completos não podem ser carregados durante a execução, portanto, essa solução impede o FPGA de operar continuamente enquanto os recursos do dispositivo são reprogramados.

A Xilinx adota a seguinte terminologia na sua solução de reconfiguração parcial(XILINX, 2008):

- Região base: região do FPGA que não se altera durante a execução e pode conter lógica que controla o processo de reconfiguração dinâmica.
- PRR - *Partial Reconfiguration Region* - Região de Reconfiguração Parcial: o formato e o tamanho de uma região parcialmente reconfigurável é definido pelo usuário através de uma restrição (*constraint*) no fluxo de compilação das ferramentas do fabricante. Uma região desse tipo é uma região previamente delimitada que pode ser *reconfigurada* em tempo de execução, alterando sua funcionalidade dinamicamente.
- A cada PRR está associado um conjunto de PRMs - *Partial Reconfiguration Modules*, ou Módulos de Reconfiguração Parcial. Cada um deles é resultado de um projeto de lógica programável específico, elaborado para uma determinada PRR. A reconfiguração parcial consiste, efetivamente, na alternância de de PRMs instanciados em PRRs.

A Figura 29 ilustra o conceito de cada um destes termos.

O fluxo de projeto de reconfiguração parcial mais recente da Xilinx, chamado *early-access*, exige uma série de passos que usualmente não estão presentes no fluxo de projeto

FPGA normal. O fluxo normal envolve uma passada única através das ferramentas de implementação: (NGDBuild, MAP and PAR) enquanto o fluxo de reconfiguração parcial envolve a implementação do projeto base e de cada módulo de reconfiguração parcial (PRM) separadamente, seguido de um passo de mesclagem final. Os passos são apresentados a seguir.

- **HDL** Descrição em linguagem de *hardware*;
- **Constrain** AGs, temporização, E/S;
- **Implementação** AGs, temporização, E/S;
- **Timing/Placement** Analysis;
- **Implementação** Projeto estático;
- **Implementação** Módulos reconfiguráveis;
- **Mescla** Junção dos módulos estático e PR.

Descrição em HDL e síntese do projeto. A reconfiguração parcial requer uma abordagem de projeto hierárquica que precisa ser estritamente seguida durante o processo de codificação da descrição de *hardware*. Toda a lógica global, como entradas e saídas, *clocks* globais e DCMs precisam estar no módulo *top-level*. Os módulos base e os módulos parcialmente reconfiguráveis podem ser instanciados como caixas-preta no módulo topo.

A ferramenta de síntese deve ser configurada para não realizar otimizações entre os contornos hierárquicos. Além disso, a inserção de entradas e saídas precisa ser desabilitada para evitar que *IO buffers* sejam inseridos nos módulos de níveis mais baixos.

Os *NetLists* são gerados separadamente para cada módulo, PRMs e módulo base.

O módulo superior (*top level entity*) precisa conter apenas instâncias caixa-preta de módulos de mais baixo nível. Não pode haver lógica sintetizada no módulo *top-level*. Ele pode conter apenas:

- instâncias de entrada/saída;
- Instâncias de *clock* primitivas;
- Instâncias do módulo base;
- Instâncias de módulos PR;

- Declarações de sinais;
- Instâncias de *bus-macros*.

Todas os sinais não globais entre os módulos PRMs e o projeto base precisam passar por uma *bus-macro* (todos exceto os sinais de *clock*).

Os módulos base não podem conter qualquer primitiva relacionada a *clocks* ou *resets* (como por exemplo: BUFG, DCM, and STARTUP_VIRTEX2).

Assim como o projeto base, primitivas relativas a *clock* e *reset* não podem ser alocadas em PRMs. Todos os módulos parcialmente reconfiguráveis para uma determinada PRR precisam ser pino a pino compatíveis uns com os outros, isto é, ter a mesma definição de portas e nomes de entidades. A nomenclatura consistente permite que cada PRM seja conectado a partir da mesma entidade *top-level*. A estrutura de cada projeto de PRM deve ser mantida, inclusive nomes de arquivos e diretórios.

Configurção restrições de projeto. Após a síntese da descrição em HDL do projeto, o próximo passo é configurar as restrições (design constraints) para o posicionamento e o roteamento. Além das restrições usuais de temporização, os projetos PR precisam ser configurados com as restrições AREA GROUP, AREA GROUP RANGE, MODE e LOC. A restrição AREA GROUP evita de a lógica do projeto base ser combinada com a lógica dos PRMs. Essa restrição precisa ser definida para cada região de reconfiguração e para o projeto base. AREA GROUP RANGE indica o formato e a localização de cada região. A restrição MODE deve ser configurada para cada região que funciona como região de reconfiguração parcial, para indicar essa sua forma de operação. LOC deve ser configurada para os elementos do projeto cuja posição deve ser conhecida de antemão, que são as primitivas relacionadas a *clock* e os pontos de entrada e saída.

As ferramentas Floorplanner, PACE e o PlanAhead podem estimar o tamanho da região PRM. No entanto, essas ferramentas não podem fazer recomendações para o desenho ou posicionamento da região parcialmente reconfigurável. A extensão da região precisa ser suficiente para acomodar o maior PRM de uma região PRR. A região precisa ser desenhada e posicionada de forma a permitir ao projeto atingir as restrições de tempo desejadas. O desenho e o posicionamento ótimo para uma região PR é dependente do projeto. Em geral, regiões que imitam o desenho de um *chip* permitem uma maior flexibilidade no posicionamento e roteamento tanto do projeto base quanto dos módulos PR. Para o Virtex-II, regiões mais estreitas resultam em *bitstreams* parciais menores, devido ao fato de uma menor quantidade de colunas de coonfiguração precisarem ser reconfigu-

radas. Para a família Virtex 4 os elementos PR estão ao longo de todo o componente, portanto o *bitstream* não depende do desenho do região reconfigurável, apenas de seu tamanho. Todos os pinos, primitivas de *clock* e *bus-macros* precisam ser posicionadas de forma a estarem divididas entre os limites da região PR e e da região base. O posicionamento das *bus macros* possui uma implicação importante para o desempenho temporal, no entanto, a efetividade de um determinado posicionamento só pode ser conhecida após sua finalização. Os usuários precisam, assim, contar com sua experiência para propor um posicionamento inicial. Em geral, posicionar as *bus macros* tão próximo quanto possível ao longo de um lado da região PR e usar macros síncronas com *enable* sempre que possível colabora para melhores resultados.

Implementação do projeto usando um fluxo tradicional. Apesar de não ser um passo estritamente necessário, implementar o projeto usando um fluxo não PR antes de mover para uma implementação PR é recomendado. Este passo é crucial para a depuração do projeto e adiciona uma análise inicial de temporização e posicionamento. Pode ser útil também para determinar a melhor extensão das PRRs e o posicionamento das *bus macros*.

Análise temporal e de posicionamento. O próximo passo é analisar o *timing* e o posicionamento do projeto. Essa análise é crítica ao estabelecer o melhor desenho, tamanho e localização de uma região PR. Durante este passo, os projetistas determinam se uma *bus macro* está posicionada efetivamente e se o desenho e localização de uma região PR permite às ferramentas cumprir os requisitos temporais.

Implementação do projeto base. Após a análise de tempo e posicionamento estar completa, a implementação da região base pode ser realizada pela ferramenta, assim como dos módulos PR.

Mesclagem dos projetos base e PR.

Neste passo, um projeto completo é construído a partir de cada PRM e do módulo base. O número de *bitstreams* criados depende do número de PRMs do projeto. São criados *bitstreams* parciais para cada PRM e um *bitstream* completo para o PRM mesclado com o projeto base.

3.3.1 *Bus-macros* e comunicação interna em sistemas parcialmente reconfiguráveis

Os módulos posicionados dinamicamente em uma infraestrutura reconfigurável necessitam trocar dados uns com os outros, ou mesmo com áreas não reconfiguráveis. Sendo assim, cria-se a necessidade de canais de comunicação próprios para atender essa demanda dinâmica.

As *bus-macros* são elementos chaves na implementação de sistemas parcialmente reconfiguráveis usando a tecnologia da Xilinx. A implementação de canais de comunicação internos para esses FPGAs, que permitam a troca de dados entre as regiões passa necessariamente pelo uso desses elementos.

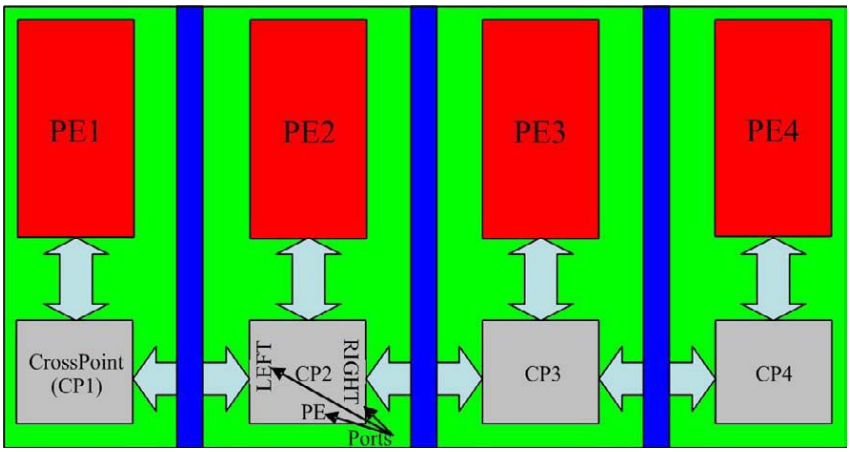


Figura 30: Divisão de um FPGA utilizando o sistema de comunicação RMBoC. Retirado de (AHMADINIA et al., 2005)

Uma solução promissora para a questão da comunicação foi apresentada por (AHMADINIA et al., 2005), chamada RMBoC (*Reconfigurable Multiple Bus on Chip*). Ela provê um canal de comunicação entre PRRs de um FPGA Virtex-II. Uma visão geral desse sistema é apresentada na Figura 30. Nela pode-se ver o FPGA subdivido em regiões verticais, separados por faixas também verticais mais estreitas. A comunicação se dá através de sinais trocados de uma região para outra, por meio de *bus-macros*, posicionadas nas faixas mais estreitas. Cada região pode conter elementos de processamento de acordo com a aplicação - um exemplo seriam microprocessadores. Os elementos se conectam ao sistema de comunicação através de módulos chamados de *cross points*. Este módulo é o mesmo para todas as regiões e é adicionado a uma região reconfigurável, como uma interface do elemento de processamento para os canais de comunicação. Tal abordagem padroniza a forma de acesso de todos os elementos ao canal de comunicação e não necessita de elementos centralizadores, como árbitros ou gerenciadores. Os *cross points* realizam a função

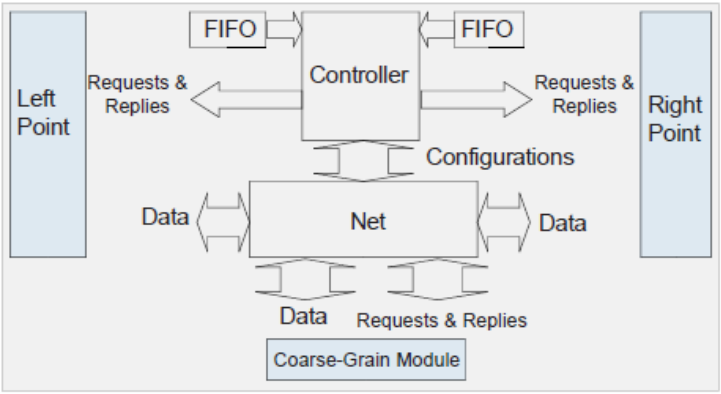


Figura 31: Um *crosspoint* do sistema RMBoC (retirado de (BOBDA, 2007)).

de roteadores reconfiguráveis, modificando sua interligação às linhas comuns do canal de comunicação e abrindo caminho para que cada elemento possa comunicar-se com outro *ponto a ponto*.

A Figura 31 ilustra a arquitetura dos *cross points*. Eles possuem três portas de ligação. Na figura, *left point* e *right point* são os pontos de ligação com os *cross-points* laterais, pelos quais recebe e envia pedidos de conexão. O controlador de cada *cross-point* trata estes pedidos e realiza o chaveamento da rede de ligação de forma a atendê-los. O atendimento das requisições iniciadas por um elemento de processamento provoca uma reconfiguração de todos os *cross-points* ao longo do caminho de interligação do módulo requisitante e do seu destino, concretizando uma ligação ponto a ponto.

4 *O projeto* ChipCFlow

ChipCFlow é o projeto de uma ferramenta capaz de gerar automaticamente descrições de *hardware* sintetizáveis em FPGAs que possuam capacidade de reconfiguração parcial e dinâmica, a partir de códigos genéricos de alto nível, escritos em linguagem C.

O processo de compilação envolve a representação dos programas como grafos a fluxo de dados dinâmicos, visando extrair o paralelismo em nível de instrução, sem a necessidade de preocupações específicas do programador. A partir do grafo, obtém-se uma descrição de *hardware*, em linguagem VHDL, correspondente a uma máquina a fluxo de dados dinâmica, específica para a execução do programa original. A natureza dinâmica dos grafos obtidos é mapeada diretamente na característica de reconfiguração parcial dos FPGAs alvo.

Inicialmente, o projeto visou o desenvolvimento em VHDL e a verificação do *hardware* dos operadores dos grafos, admitindo-se, no entanto, que os grafos eram estáticos (ASTOLFI; SILVA, 2007), (SILVA; MARQUES, 2006). A implementação obtida seguiu a abordagem de reconhecimento (*acknowledge*), como descrito na seção 2.1. Cada operador possuía, além dos arcos para a entrada e saída de dados, linhas de *strobe* e *acknowledge*, garantindo que num determinado momento, um único *token* estivesse presente em um arco. No entanto, ChipCFlow mira a obtenção de *hardware* que implemente grafos dinâmicos, aumentando o nível de concorrência durante a execução.

4.1 Mapeamento dos grafos dinâmicos em FPGAs parcialmente reconfiguráveis

A abordagem dinâmica no ChipCFlow baseia-se na solução *tagged-token* das máquinas a fluxo de dados dinâmicas tradicionais. Os *tokens* carregam, além do valor de interesse, um *tag* que os associa a um determinado contexto de execução - tipicamente *loops*, iterações ou chamadas de funções.

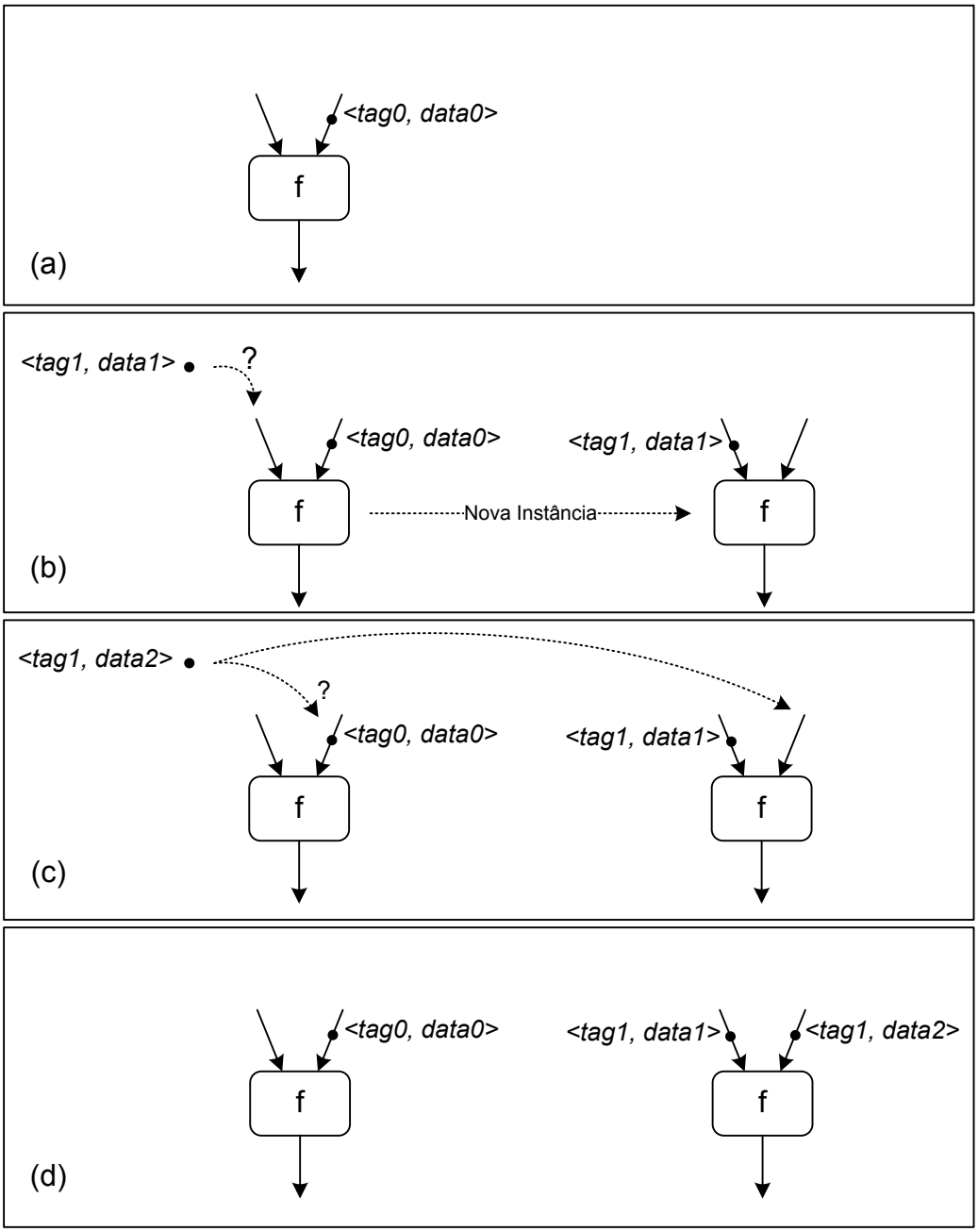


Figura 32: Evolução de um operador no modelo de instâncias. Em (a) o operador aguarda em seu segundo arco um operador com o mesmo tag do primeiro. Em (b) um novo dado chega, porém de tag diferente, provocando a instanciação de um novo operador(c). Em (d) um dado com o tag igual ao da segunda instância chega, e o matching é bem sucedido. A segunda instância está habilitada a disparar.

Em uma máquina *tagged-token* tradicional, os *tokens* e as instruções são processados por elementos de processamento dedicados (único ou múltiplos). No caso do ChipCflow a abordagem é outra, pois não há um elemento de processamento central, e a operação de *matching* em cada operador, quando não é bem sucedida, leva à instanciação de novos operadores, que podem ser vistos como elementos de processamento distribuídos. Cada operador do grafo original, portanto, pode possuir mais de uma instância dele mesmo em um instante de execução.

A figura 32 ilustra a idéia. Quando um *token* chega a um operador para execução, é necessário que os outros dados de entrada do operador que pertençam ao mesmo contexto - *mesmo tag* - estejam presentes nos outros arcos de entrada. Quando essa operação de *matching* não encontra um parceiro válido para um *token* de entrada, dispara a criação de uma nova instância do operador com o *tag* associado.

O modelo *tagged-token* requer que operadores específicos de *tag* sejam adicionados ao grafo a fluxo de dados que representa o programa de interesse. Estes operadores não atuam sobre os valores de dados, apenas sobre os *tags*. No projeto ChipCFlow há três operadores deste tipo. A figura 33 mostra um caso em que esses operadores são necessários. Eles são posicionados nas entradas e nas saídas dos *loops*. Na entrada de cada laço, é adicionado o operador *New Tag Manager* (NTM). Em um arco de realimentação (reentrância), um operador *New Iteration Generation* (NIG), e nos arcos de saída é adicionado o operador *New Tag Destruction* (NTD).

Para mapear esta abordagem no contexto do ChipCFlow, a proposta é dividir o FPGA alvo em um certo número de regiões parcialmente reconfiguráveis - PRRs - e fazer com que cada região possa conter, em um determinado instante de tempo, uma configuração correspondente a uma parte do grafo original dinâmico. Inicialmente esta partição pretendia ser composta de um único operador, e dessa forma a instanciação de outros operadores com novos *tags* equivaleria à reconfiguração de uma PRR disponível, que passaria a conter um novo operador com seu *tag* correspondente. A figura 34 ilustra a idéia de associar cada instância de operador a uma PRR.

O modelo para execução de um grafo dinâmico, na proposta original do projeto seria, portanto, o seguinte: Um FPGA dividido em n PRRs, cada uma delas igualmente capaz de ser configurada com um dos elementos de um conjunto de m Módulos de Reconfiguração Parcial - PRMs. Cada PRM equivaleria a um operador do grafo. O processamento portanto, estaria distribuído espacialmente entre os operadores, que seriam instanciados dinamicamente, de acordo com a evolução do programa. Como os *tokens*

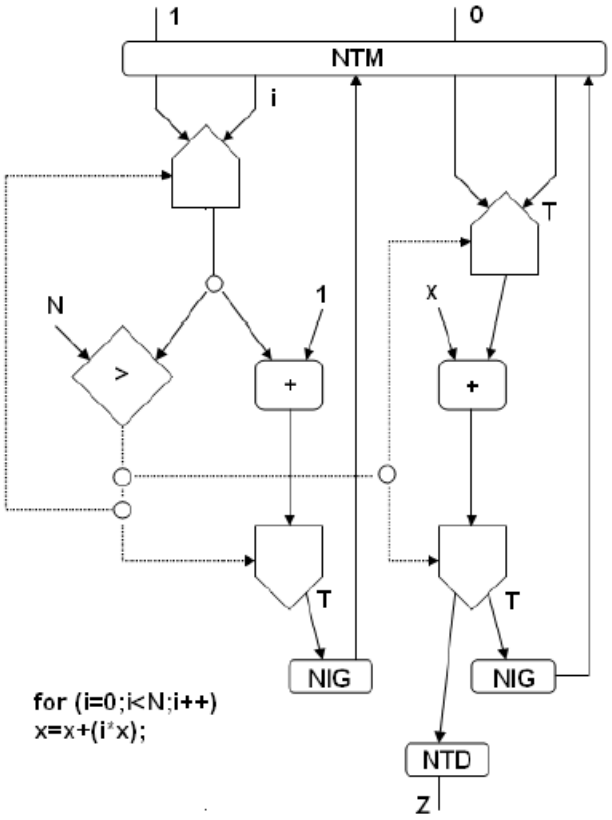


Figura 33: Um grafo *dataflow* dinâmico com os operadores de *tag*

precisam transitar de um operador para outro, um meio de comunicação garantiria que fossem trocados entre as regiões. O mecanismo de *matching*, portanto, seria distribuído entre os operadores.

4.1.1 ChipCFlow: fluxo de projeto

A partir das definições apresentadas na seção 4.1 é possível ter uma visão mais clara do projeto ChipCFlow. A figura 35 mostra um fluxo de projeto esperado com a ferramenta. Os passos mostrados para se obter um sistema em *hardware* que execute na forma de uma máquina a fluxo de dados dinâmica um programa originalmente em linguagem de alto nível são os seguintes:

- *1º Passo:* Obtenção do grafo a fluxo de dados a partir do código original em linguagem alto nível. Essa é essencialmente uma tarefa de compilação. Resultados iniciais já foram alcançados em (SILVA; COSTA; RODA, 2009), mas a solução ainda não é completa.

- *2º Passo:* Particionamento do grafo obtido. O grafo precisa ser particionado, de

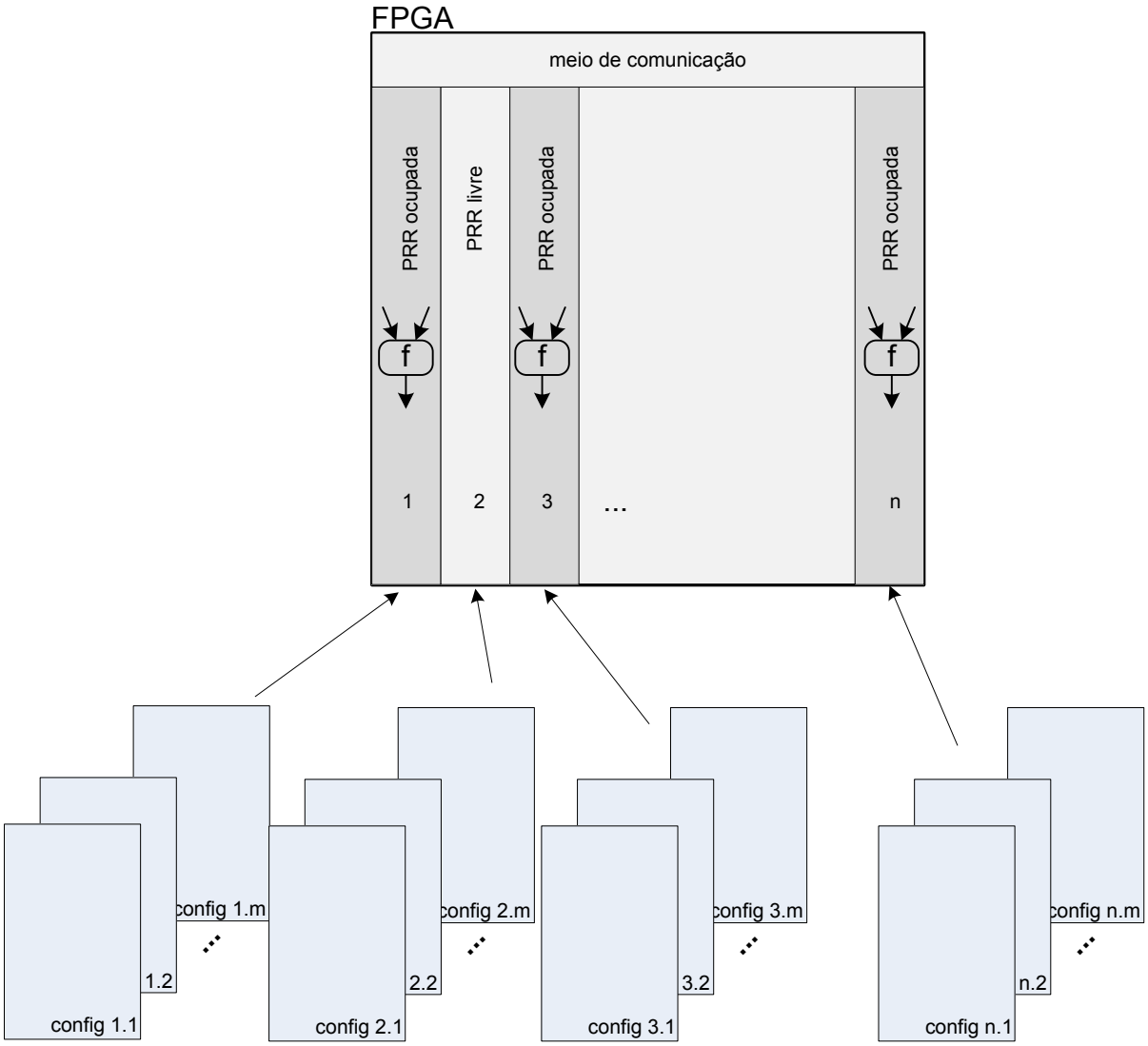


Figura 34: Originalmente o projeto ChipCFlow previa um operador por PRR

maneira que cada partição possa ser instanciada em qualquer das regiões de reconfiguração parcial. Cada partição passa a ser vista como um bloco de processamento com entradas e saídas correspondentes aos arcos dos operadores seccionados.

- *3º Passo:* Tradução das partições em descrição VHDL. A descrição VHDL de cada operador presente na partição é obtida de uma biblioteca de operadores, e a composição de uma partição do grafo é feita interligando-se os arcos dos operadores.

- *4º Passo:* Adição dos controladores de comunicação às descrições das partições. O controlador de comunicação é um bloco de *hardware* que padroniza a interface entre a partição (subgrafo) e um canal de comunicação que interliga todas as regiões de reconfiguração parcial.

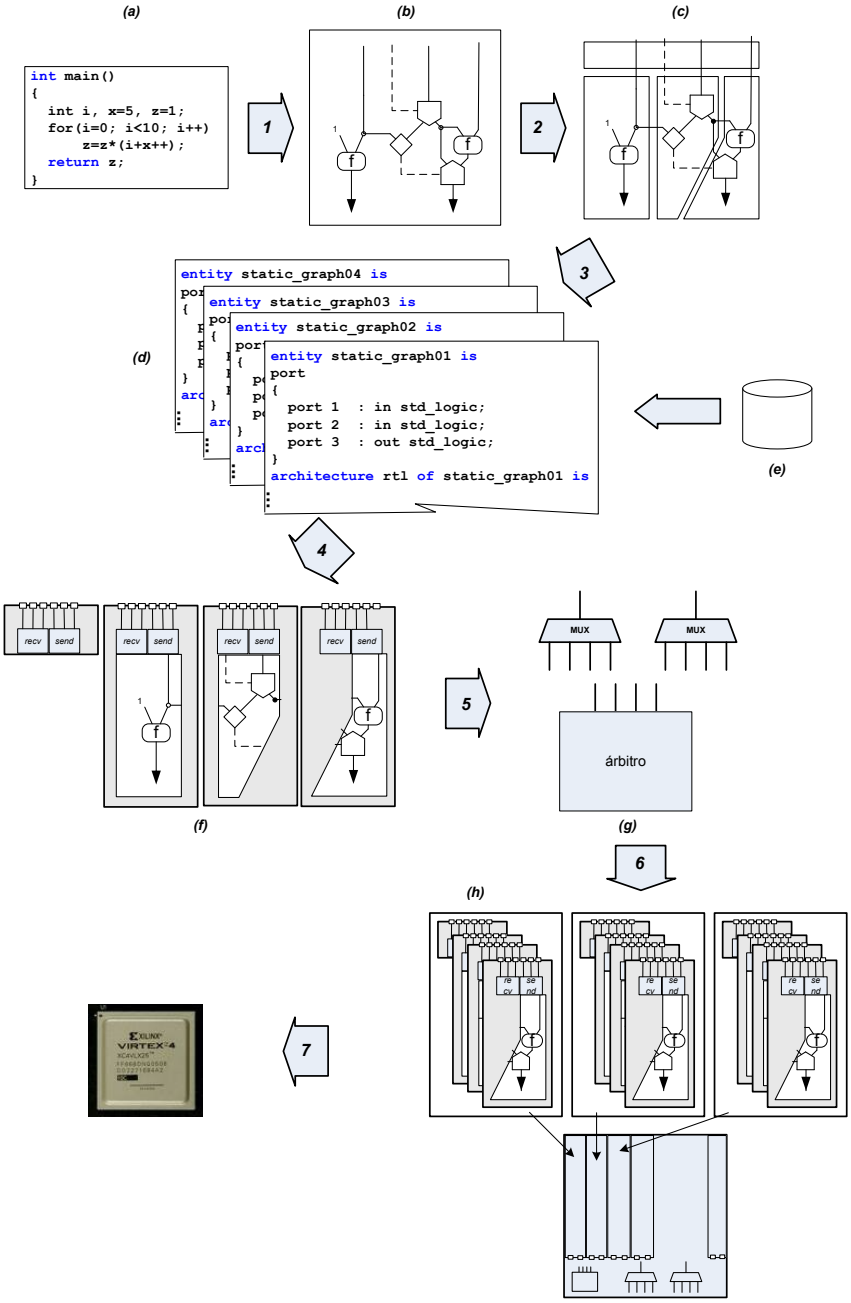


Figura 35: Fluxo de projeto ChipCFlow. A partir de um código em alto nível (a), obtém-se uma representação do programa em CDFG(b). O grafo é então particionado(c), e a partir de cada partição gera-se uma descrição VHDL para cada subgrafo estático(d). A cada subgrafo estático são adicionados controladores de comunicação(4), obtendo-se a descrição completa de cada partição(f). Além dos elementos de comunicação de cada partição, os elementos estáticos são adicionados(g). O projeto nas ferramentas EDA associa a cada PRR a possibilidade de conter qualquer uma das partições como configuração(h) e posiciona também os elementos estáticos fora das PRRs e ligados a elas pelas *bus-macros*. O projeto é compilado de acordo com as regras das ferramentas, obtendo-se os *bitstreams* finais de programação.

- *5º Passo:* Adição dos módulos que são localizados na região estática do FPGA: gerenciadores de configuração, canais de comunicação e árbitro. O elemento de gerenciamento é o elemento responsável pelo atendimento às requisições de reconfiguração geradas pelo sistema em execução. As possíveis configurações de cada região reconfigurável são conhecidas em tempo de compilação e devem estar disponíveis em uma memória não volátil para que o gerenciador de comunicação possa disparar o seu carregamento na região de reconfiguração parcial desejada. O canal de comunicação é composto por sinais que interligam as partições através de multiplexadores, enquanto o árbitro é um módulo que faz o escalonamento de acesso ao meio de comunicação. O funcionamento destes elementos será melhor detalhado no capítulo 5.

- *6º Passo:* Uso das ferramentas de EDA específicas para obtenção dos *bitstreams* de configuração do FGPA alvo. Para que se complete este passo é necessário adequar as descrições em VHDL obtidas na forma de uma aplicação que faz uso da reconfiguração parcial a ser compilada nas ferramentas específicas do fabricante do FPGA. Deve ser respeitado o fluxo de projeto das ferramentas. Nessa etapa os diferentes módulos citados anteriormente são associados às regiões específicas do FPGA. Obtém-se como saída os *bitstreams* de configuração e reconfiguração do FPGA.

- *7º Passo:* Carregamento do sistema em um FPGA alvo.

No capítulo seguinte(4) será apresentado o modelo de particionamento e comunicação para o projeto, com algumas mudanças com relação ao fluxo original apresentado aqui, visando aproveitamento melhor do recurso de reconfiguração.

5 *Modelo de Particionamento e Protocolo de Comunicação*

O particionamento dos grafos dinâmicos em uma granularidade no nível de operadores parecia promissor em resolver o seu mapeamneto em FPGAs parcialmente reconfiguráveis, como apresentado no capítulo 4. No entanto, esbarra em uma questão importante: A reconfiguração de uma PRR é um processo lento em relação aos tempos de execução esperados para os operadores. Isso significa que uma partição levaria muito mais tempo para ser alocada do que para executar.

A memória de configuração dos FPGAs Virtex da Xilinx é arranjada na forma de quadros verticais. A reconfiguração não é possível numa granularidade menor do que a de um quadro. O tempo nominal para reconfiguração de um único quadro na série Virtex II é de 12,5μs. No entanto, este tempo pode variar de acordo com os recursos utilizados em uma coluna. Resultados práticos (PAPADIMITRIOU; ANYFANTIS; DOLLAS, 2007) também indicam tempos possivelmente maiores.

Já o tempo de execução de apenas um operador é variável de acordo com o FPGA alvo, do resultado das etapas de posicionamento e de roteamento, realizadas pelas ferramentas de EDA, além de outros fatores. Assim mesmo, para que se possa fazer uma estimativa, um operador leva cerca de 3 a 4 ciclos de relógio para executar, a partir do momento em que os dados estão presentes em sua entrada. Considerando um relógio com 100MHz, que é um valor alcançado sem grandes problemas pelos FPGAs em questão, este tempo está em torno de 40 a 50ns, podendo ser ainda menor para relógios de frequência maior.

Este valor deve, no entanto, ser usado apenas como estimativa do esperado para execução do operador, devendo-se levar em conta também o tempo de espera dos dados necessários. Este tempo, por sua vez, dependerá da evolução em execução de cada programa. Apesar disso, é possível concluir que a relação *tempo em execução / tempo em reconfiguração* é muito baixa.

Para melhorar essa relação, aumentando a eficiência, uma possibilidade é fazer com que cada partição seja responsável pela execução de porções maiores de código, o que equivale a dizer que deve corresponder a *seções* maiores do CDFG original. Por outro lado, esse aumento pode fazer com que o nível de concorrência diminua, efeito contrário ao desejado quando se opta pelo modelo *tagged-token*. Devido a esse compromisso, um bom ponto de partida seria escolher partições com o máximo de operadores capazes de serem instanciados no menor tamanho de PRR. Dessa forma teria-se, a princípio, o menor tempo possível de reconfiguração com o máximo tempo de execução.

Em (JUNIOR et al., 2010) é mostrado que à medida que a largura de uma PRR cresce, juntamente com o número de operadores que acomoda também, há uma tendência para se ocupar uma única coluna com um número fixo de operadores ChipCFlow. Este resultado é indicativo do número de operadores que uma coluna em um FPGA VIrtex II Pro (de 30816 CLBs) comportaria, sem levar em consideração variações em sua interligação e a presença de outros elementos, necessários à partição, como o controlador de comunicação. Este número está entre 14 e 15 operadores, conforme a tabela 1.

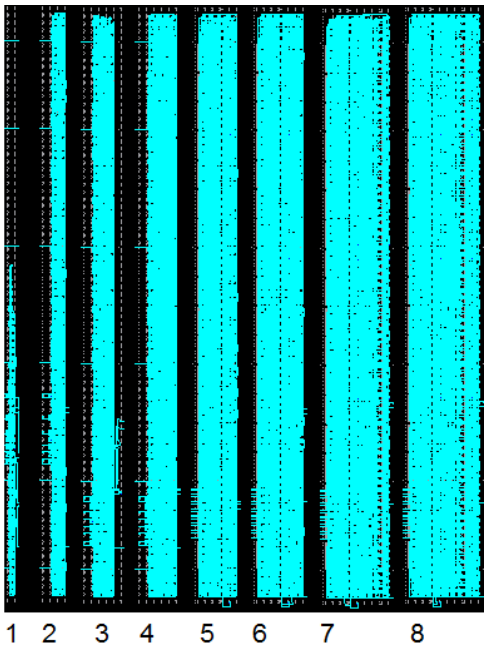


Figura 36: Alocação de colunas do FPGA em partições com diferentes números de operadores

Apenas para se ter um valor de referência, a soma do tempo de execução de cada um dos 15 operadores serialmente, levaria a valores em torno de 60 a 75 ciclos, que com um relógio de 100MHz levaria entre 600 e 750ns. Sabe-se no entanto, que tipicamente os operadores não serão executados serialmente, devido à própria construção dos grafos, o que torna esse número potencialmente menor.

Area size	Number of ops.	Number of ops. per col.
1	6	6
2	29	14.5
3	45	15
4	61	15.250
5	70	14
6	84	14
7	98	14
8	113	14.125

Tabela 1: Número de operadores por coluna (JUNIOR et al., 2010).

Ainda em (JUNIOR et al., 2010), os autores concluem pela necessidade de uma abordagem mais inteligente das áreas de reconfiguração do ChipCFlow, incluindo, possivelmente, mecanismos de reuso de uma partição sem a necessidade de reconfiguração, evitando-se o desperdício de tempo da reconfiguração.

Na determinação de um número de operadores ótimo para uma partição, há que se levar em consideração também que cada FPGA alvo terá um tamanho de coluna (em termos de quantidade de recursos disponíveis) diferente. Há ainda, outros fatores que devem ser pesados na obtenção de um tamanho ideal de partição, como por exemplo a taxa de reconfiguração esperada por um programa, e o fato de a maior partição determinar o tamnho mínimo de uma PRR, uma vez que todas as PRRs devem ser capazes de comportar todas as possíveis partições do grafo original.

É necessário também que se inclua os recursos ocupados pelo controlador de comunicação na soma de recursos de uma partição.

O presente trabalho não se ocupou da determinação dos melhores valores de tamanho de partição, apenas buscou fornecer uma infraestrutura necessária para que uma partição contivesse mais de um operador, uma vez percebida essa necessidade.

Espera-se, assim, que trabalhos futuros se ocupem em encontrar algoritmos de otimização do tamanho das partições, buscando diminuir os tempos de execução totais dos grafo a fluxo de dados dinâmicos.

Assim sendo, a questão do alto custo da reconfiguração direcionou o projeto no sentido de particionar os grafos a fluxo de dados dinâmicos em seções maiores, contendo mais do que um único operador, de forma a aproveitar melhor uma operação de reconfiguração. Dessa forma, o tempo de execução de cada partição seria potencialmente maior, eventualmente próximo dos tempos de reconfiguração.

Dividir o grafo em partições contendo mais de um operador leva a uma questão importante. O modelo inicial previa a instanciação de novos operadores quando uma operação de *matching* não era bem sucedida. Para o ChipCFlow, previa-se inicialmente, múltiplas instâncias de um único operador.

A solução encontrada foi a de considerar o grafo de uma partição como sendo estático. Os operadores internos às partições não possuem múltiplas instâncias, e a reentrância é evitada através do mecanismo de *acknowledge*. A granularidade do paralelismo devido à abordagem *tagged-token* passa a ser maior, sendo que as diversas 'instâncias' são agora de subgrafos estáticos e não mais de apenas um operador.

5.1 O modelo de particionamento

Dado um grafo a fluxo de dados dinâmico, composto dos operadores descritos na seção 2 e dos operadores de *tags* descritos na seção 4.1, é possível subdividí-lo em tantas partes quantas forem necessárias desde que obedecidas as seguintes restrições, necessárias para garantir a consistência na execução dos grafos dinâmicos:

- Um operador *New Tag Manager*, na entrada de um laço, deve estar totalmente contido em uma única partição.
- Excetuados os casos contidos no item anterior, o tamanho mínimo de uma partição é o de um operador, e o tamanho máximo é o de um grafo completo.
- Os outros operadores de *tag* (NIG e NTD) devem estar nos arcos de saída das partições.

É importante notar que essas restrições não impedem que o modelo original, que previa apenas um operador por partição, seja implementado.

De um ponto de vista conceitual, essas restrições são suficientes, ou seja, garantem o funcionamento adequado dos grafos. No entanto, para efeito de melhor desempenho é necessário considerar outros fatores na determinação das partições, devidas à plataforma e à tecnologia de reconfiguração parcial.

O particionamento do grafo está diretamente ligado à divisão em PRRs a ser adotada no FPGA alvo. Como no modelo adotado todas as PRRs devem ser capazes de receber a configuração de todas as partições, a partição do grafo de maior tamanho (em termos de recursos do FPGA) define o tamanho mínimo das PRRs do FPGA.

Como exemplo, a Figura 37 mostra um grafo e o código que implementa. Uma subdivisão em partições é mostrada, escolhida manualmente.

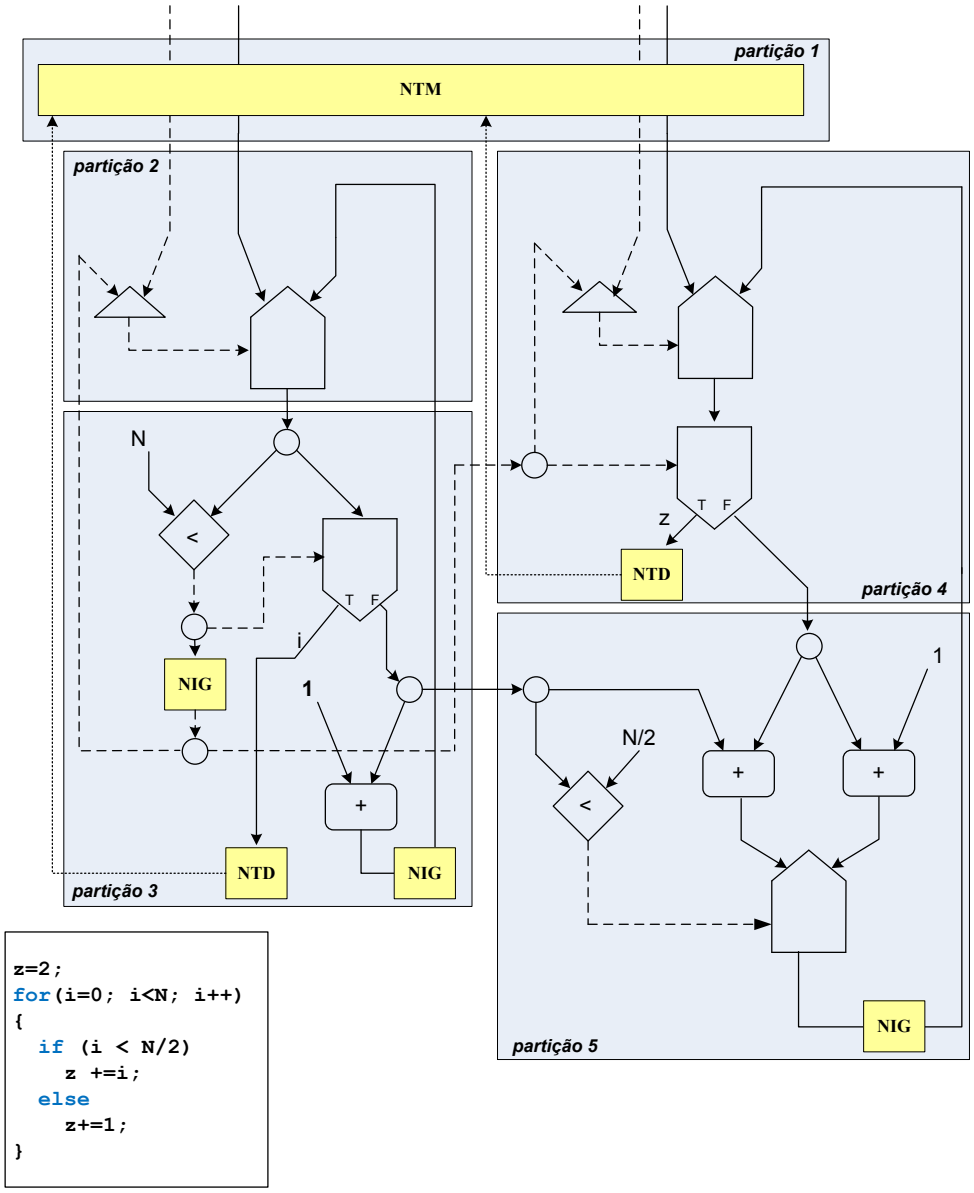


Figura 37: Exemplo de grafo particionado.

Note-se que as partições foram definidas obedecendo as restrições impostas pelo modelo.

5.2 Comunicação

A comunicação entre as partições é implementada através dos controladores de comunicação, do canal de comunicação e do árbitro do canal.

5.2.1 Controlador e canal de comunicação

Para que os *tokens* possam transitar entre as partições, propõe-se a comunicação através de um canal compartilhado. Os sinais de interligação entre cada partição instanciada em uma PRR e o canal não podem ser sinais comuns. O canal de comunicação deve permanecer inalterado durante a reconfiguração das diversas PRRs, sem afetar o funcionamento e a troca de dados das demais partições. Os PRMs precisam ser inseridos e retirados dinamicamente das PRRs sem afetar o restante do *hardware*. Além disso, a interface de cada módulo com o canal de comunicação precisa ser idêntica, para que a partição possa ser instanciada em qualquer PRR.

Para atender a essas necessidades, o canal deve estar em uma região de configuração estática do FPGA, fora das PRRs, que não se altera durante a execução. A interligação de um PRM com o canal precisa ser feita através dos elementos próprios para interligação entre regiões estáticas e regiões de reconfiguração parcial (PRR). Esses elementos são as chamadas *bus-macros*.

Para implementar a interface das partições com o barramento de comunicação, cada subgrafo a fluxo de dados estático da partição deve ser conectado a um módulo *controlador de comunicação*. Este módulo é responsável pela recepção e envio dos *tokens* de entrada e de saída de um determinado subgrafo (Figura 38) e se conecta ao subgrafo estático em suas entradas e saídas por um lado e ao canal de comunicação pelo outro, através de *bus macros*. Dessa maneira, atende-se ao requisito de se criar uma interface padrão para todas as partições.

A Figura 39 ilustra, para um grafo hipotético, o conceito da partição composta por um grafo estático e seu controlador de comunicação. Nesse exemplo o grafo possui quatro entradas e duas saídas.

O canal de comunicação foi concebido para ocupar uma região de configuração estática do FPGA e é composto de um grande multiplexador para os sinais de dados e uma porta lógica *ou*. A Figura 40 mostra a concepção desses dois elementos simples, usados para interligar as partições.

Além desses dois elementos existe um árbitro, que também fica posicionado na região estática e que gera os sinais que habilitam uma partição para o envio de dados, ou seja, que lhe dá acesso ao canal de comunicação. O árbitro gera um sinal de habilitação para cada partição separadamente, como se vê na figura. Além destes sinais, um outro sinal é enviado a todas as partições, indicando que uma transferência de dados irá iniciar.

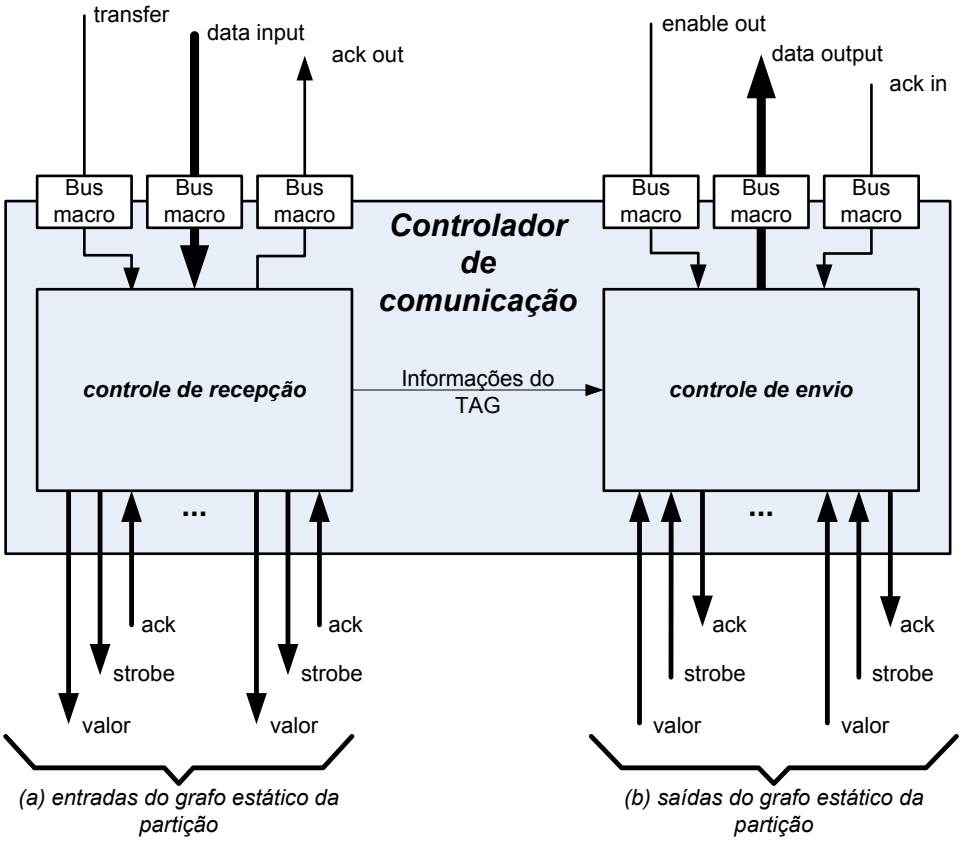


Figura 38: Controlador de comunicação.

A Figura 41 mostra o sistema de comunicação, apenas como exemplo, para um caso com 3 PRRs em um FPGA (não estão mostrados o árbitro nem o gerenciador de reconfiguração).

O controlador de comunicação foi elaborado para realizar a interface entre os subgrafos particionados e o canal de comunicação interno do FPGA alvo, como mostra a (Figura 38). A sua interface com o canal de comunicação é concretizada através das *bus macros*, elementos que realizam a passagem de sinais de uma região estática para uma região de reconfiguração parcial. A interface do controlador de comunicação com o grafo estático se dá através da ligação com os arcos de entrada e de saída do subgrafo, e segue o mecanismo de reconhecimento, adotado internamente nos subgrafos.

Internamente o controlador de comunicação está dividido em dois módulos, um para o controle do envio de dados e outro para a recepção. O módulo de controle de recepção recebe todas os quadros do canal de comunicação e dispara as ações necessárias na partição, quando estes quadros são endereçados à partição que pertence. Após a sua instanciação, uma partição, composta por seu subgrafo e pelo seu controle de comunicação, recebe um comando do tipo `0x03`, para configurar o *tag* com que irá operar. Esta mensagem é en-

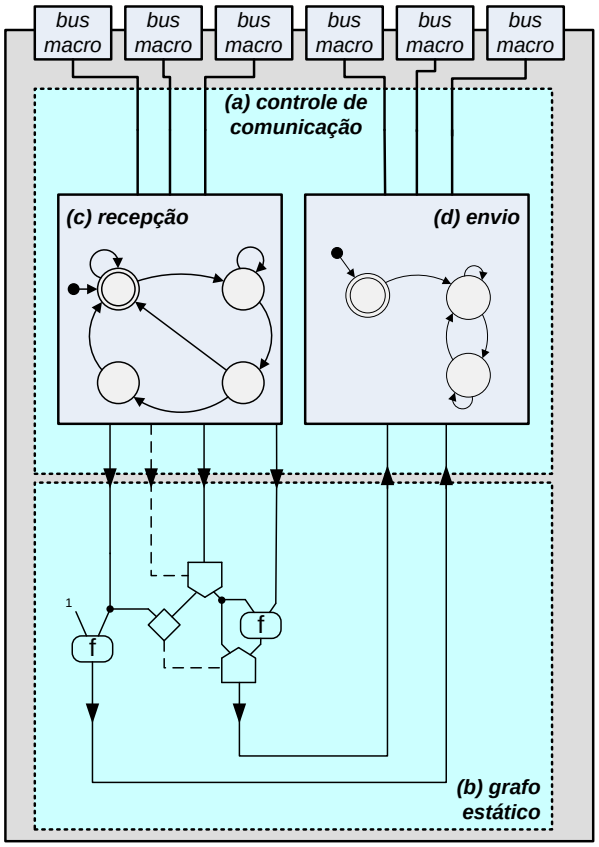


Figura 39: Diagrama mostrando uma partição hipotética composta de seu grafo estático e seu controlador de comunicação.

viada pelo controle de reconfiguração, logo após a reconfiguração da PRR. O módulo de controle de reconfiguração fica na região estática e será detalhado mais adiante. Este comando é necessário para que o controle de recepção seja capaz de distinguir os *tokens* que são direcionados para sua partição ou não, ou seja, para realizar a operação de *matching*.

A partir do momento que uma partição recebe esse comando, após sua instanciação, ela passa a operar normalmente, analisando a cada quadro de comunicação se a mensagem é dirigida a ela mesma. O controle de envio, por sua vez, armazena os *tokens* que chegam dos arcos de saída do subgrafo estático, e aguarda o momento em que o árbitro habilita a partição para envio.

A Figura 42 mostra as máquinas de estado de envio e de recepção do módulo.

A máquina de *recepção* aguarda no seu estado *Wait Transfer* até que o sinal *transfer* esteja alto. Este sinal é gerado pelo árbitro e igualmente conectado aos controladores de todas as partições. Após a indicação do sinal, o módulo permanece no estado *Receive*, até ter recebido todos os pacotes que compõem um quadro completo. Após o final do recebimento, o sinal de *transfer* volta a ficar inativo, fazendo com que a máquina transite

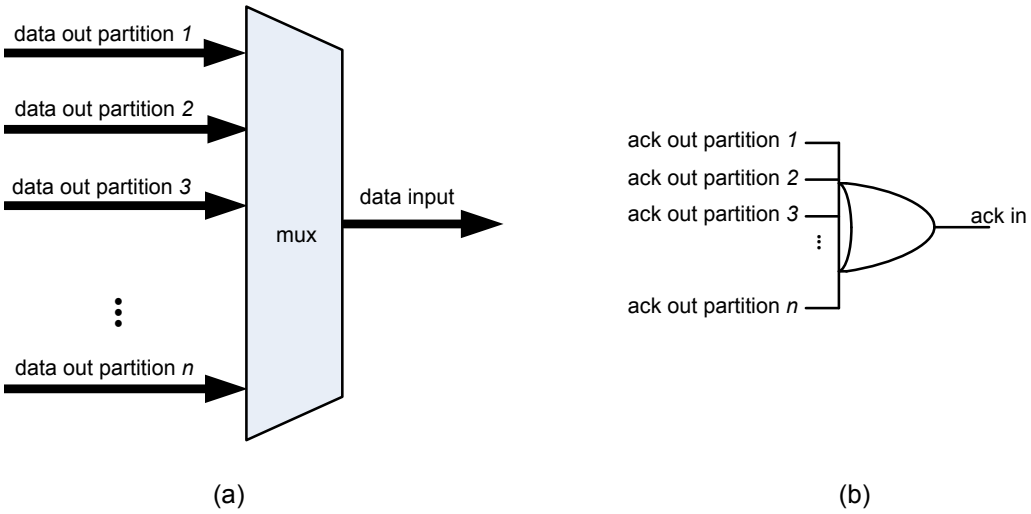


Figura 40: Elementos de interligação entre as partições que compõem o canal de comunicação. Em (a) um multiplexador dos dados; em (b) uma porta lógica OU para a indicação de recebimento de um quadro válido(*matching*).

então para o estado *process*, em que o quadro recebido é analisado. Caso o quadro seja dirigido à partição em questão *matching* a máquina transita para o estado de *ack*, onde o sinal *ack out* é gerado, indicando ao árbitro e ao controle de reconfiguração que a mensagem foi recebida. O árbitro então habilita outra partição, e o processo recomeça. Caso o *matching* não seja bem sucedido, o controlador de reconfiguração dispara a reconfiguração de uma nova PRR, visando a instanciação de uma partição ausente com o *tag* requerido pelo *token* enviado.

A máquina de envio permanece no estado *select frame* após a sua inicialização, enquanto não for habilitada pelo árbitro a enviar um de seus *tokens*. Assim que é habilitada, faz o envio. Ao término do envio, é esperado uma ativação do sinal *ack in*, indicando que o *token* foi recebido. Caso o sinal não seja ativado, o controlador desconsidera o envio, e aguarda uma próxima oportunidade de habilitação pelo árbitro para reenviar o *token*. O controlador de envio ainda possui uma segunda máquina de estados, que trabalha paralelamente. Essa segunda máquina faz o recolhimento dos dados de saída do grafo estático, armazenando-os em registros internos para que possam ser enviados no momento correto.

É importante notar que pelo fato de no interior de uma partição o grafo ser do tipo estático, não há a necessidade de os operadores levarem em consideração os *tags* dos *tokens*. A operação com os *tags* ocorre apenas quando os *tokens* chegam ou saem da partição em questão, ou seja, quando passam pelo controlador de comunicação. Na entrada de uma partição, quando um novo *token* chega, seu *tag* é comparado com o *tag* da partição. Se

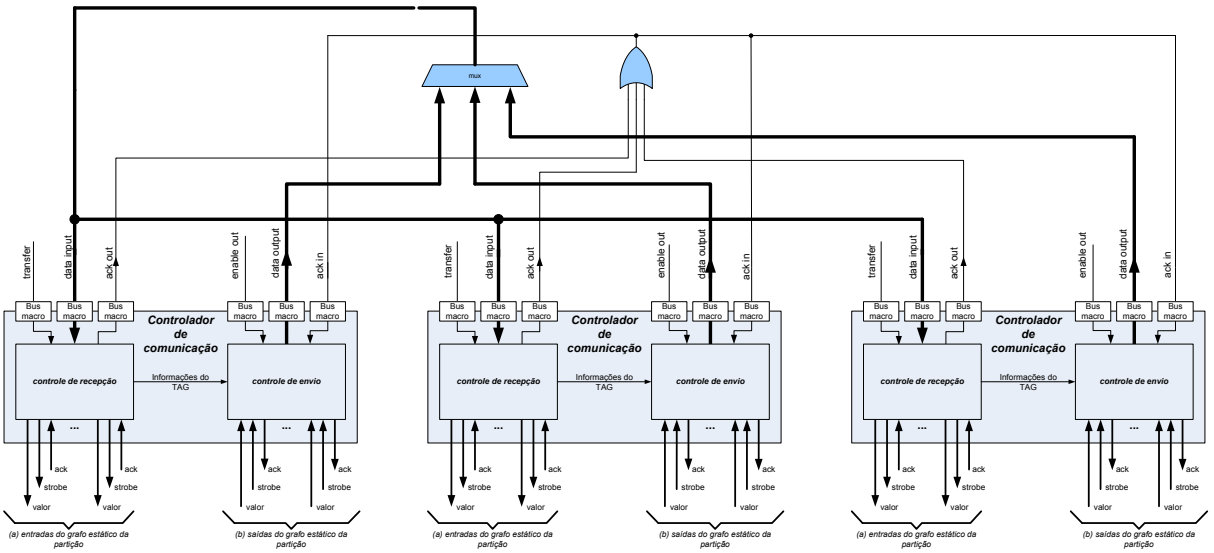


Figura 41: Exemplo de interligação dos controladores de comunicação de três partições com o canal de comunicação. O multiplexador e a porta lógica que se vêem ao alto são posicionados em uma região estática, enquanto a cada PRR associa-se uma partição.

o *matching* for bem sucedido, apenas o dado em si é interessante para os operadores do grafo estático. Ao sair de uma partição, os *tokens* devem carregar novamente o seu *tag*. Caso o *tag* necessite ser atualizado, devido à presença de operadores de *tag*, essas alterações podem ser realizadas somente na saída, ou seja, podem ser incorporadas pelo controlador de envio. Essa abordagem, que simplifica a implementação das partições, foi adotada, e justifica o fato de se realizar as partições de forma que os operadores de *tag* estejam sempre nos arcos de saída das partições.

5.2.2 Protocolo

A comunicação ocorre através de quadros enviados de uma partição para outra, que carregam os *tokens* de um arco de saída de uma partição para outro de entrada de uma outra partição.

A Figura 43 mostra um quadro de comunicação entre partições[1]. Ele contém os seguintes campos:

- *partition*: indica a qual partição se destina um token;
- *input*: indica a qual entrada da partição se destina o token;

[1]O quadro aparece dividido em duas partes por conveniência de espaço

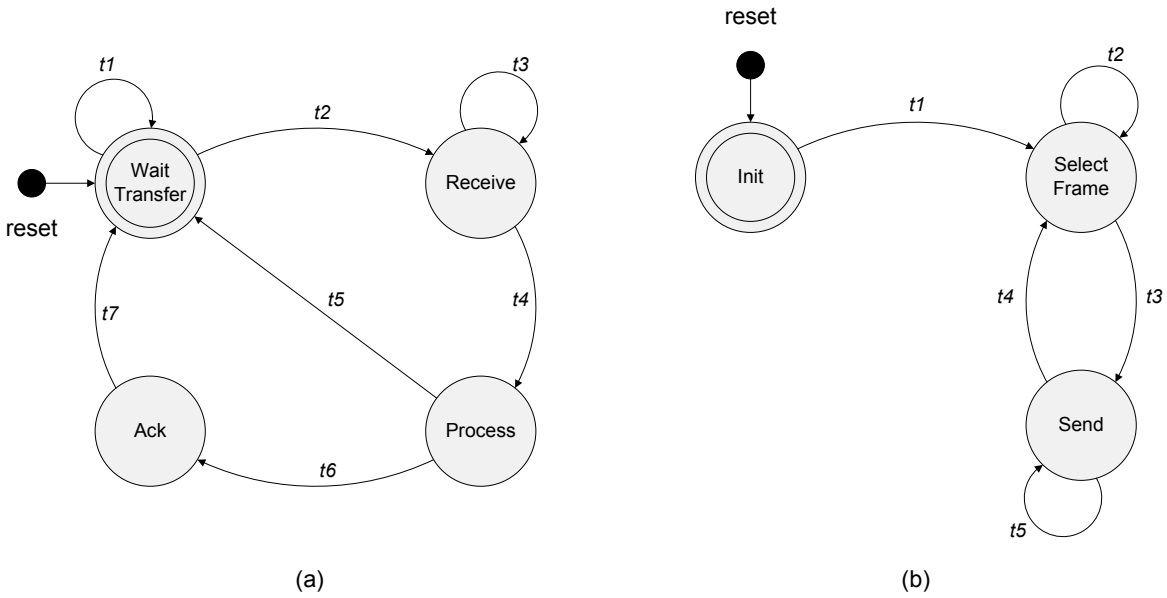


Figura 42: Máquinas de estado dos submódulos de recepção (a) e envio (b) do controlador de comunicação.

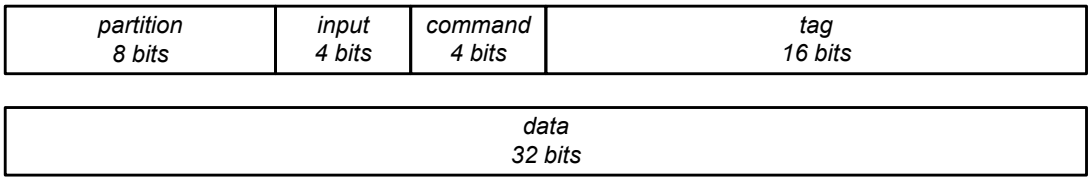


Figura 43: Quadro de comunicação.

- *command*: indica o tipo de comando que o frame está enviando;
- *tag*: tag do token;
- *data*: valor de interesse

Os quadros são enviados pelos elementos conectados ao canal de comunicação - as partições e o gerenciador de comunicação - quando estes estão habilitados pelo árbitro. Somente um elemento pode enviar um quadro por vez. O tempo necessário para o envio de um quadro depende da frequência de operação do relógio com que o circuito final poderá funcionar. Na implementação realizada optou-se por permitir que a largura do barramento de dados do controlador de comunicação fosse selecionada em tempo de compilação, através de um parâmetro da entidade de *hardware*. A largura pode estar, assim, entre 1 *bit* e 64 *bits*. No primeiro caso, a cada ciclo de relógio um *bit* do quadro é enviado, caracterizando assim um barramento tipicamente serial, com tempo de transferência igual a 48 ciclos de *clock*. São permitidas larguras intermediárias múltiplas de oito, fazendo com que o tempo de transferência vá caindo. Para o caso mais rápido 64 *bits* são transferidos

em um único ciclo de relógio.

A decisão de tornar a largura do barramento parametrizável veio do fato de o modelo ainda ser uma proposta que necessitará de mais avaliações, de desempenho e verificação, para que se possa concluir por um valor fixo.

A princípio a escolha recairia naturalmente no caso de apenas 1 ciclo para transferência, mas o custo que se paga é um ocupação maior do espaço do FPGA em que o programa será implementado. Mais linhas de roteamento entre os elementos naturalmente gera maior uso de recursos. É necessário, portanto, encontrar uma boa relação entre *velocidade de transferência* e *uso de recursos*.

O campo *command* do quadro de comunicação foi criado devido à necessidade de se inserir no protocolo algumas funções especiais, além da função mais comum de troca de *tokens* entre partições.

Os valores aceitos nesse campo são os seguintes:

- `0x0`: indica uma transação comum de envio de *token*;
- `0x1`: indica a finalização de execução de uma partição;
- `0x2`: indica a finalização de execução de uma ativação;
- `0x3`: indica o tag que uma partição deve assumir a partir de sua instanciação;

O quadro da Figura 44, por exemplo, indica o envio de um *token* para a partição 3, arco de entrada 2. O *tag* é o 22 e o valor do dado é 64. Este é um quadro de comando `0x0`, ou seja, indica um *token* comum, enviado de uma partição a outra.

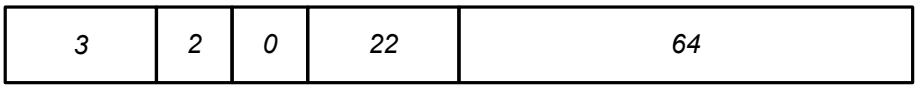


Figura 44: Exemplo de quadro trocado entre uma partição e outra, carregando um *token* e seu *tag*.

As mensagens com os outros comandos serão detalhadas na seção 5.2.3.

5.2.3 Gerência de reconfiguração

O gerenciador de reconfiguração é um módulo que deve ser adicionado ao sistema para controlar as configurações que cada região de reconfiguração parcial assume durante

a execução. Ele pode ser implementado através de um *hardware* específico ou então como *software* de um processador já presente na mesma pastilha do FPGA.

O módulo funciona conectado ao canal de comunicação, da mesma forma que as partições. Ele recebe todos os quadros de comunicação durante a execução, e aguarda pelo recebimento do sinal de reconhecimento (*ack*) de alguma partição. Caso o sinal não seja recebido, o gerenciador identifica pelo conteúdo da mensagem qual deve ser a nova partição a ser instanciada e qual deve ser seu *tag*. Ela é então colocada em uma fila de reconfigurações pendentes. O gerenciador trabalha mantendo um mapa das PRRs ocupadas ou não. Quando uma delas desocupa, o que é indicado pela chegada de uma mensagem de comando `0x02` ou `0x03`, essa PRR é marcada como desocupada, e o gerenciador então a ocupa com o primeiro elemento da fila. Logo após a reconfiguração, o gerenciador envia um comando do tipo `0x01`, para atribuir um *tag* à nova PRR configurada. Dessa forma ela passa a operar normalmente.

5.3 Exemplo de aplicação

Como exemplo de aplicação do modelo proposto, tomou-se o grafo da Figura 37.

Inicialmente, o grafo foi particionado manualmente, apenas com o cuidado de se respeitar as restrições impostas pelo modelo proposto. Optou-se por cinco partições, conforme ilustra a Figura 37.

O comunicador foi adicionado à descrição das partições, sendo adaptado ao número de arcos correspondentes. Os arcos de entrada foram conectados ao módulo de recepção e os arcos de saída ao módulo de envio. Através de simulações o comportamento do comunicador pode ser testado e depurado (Figura 45).

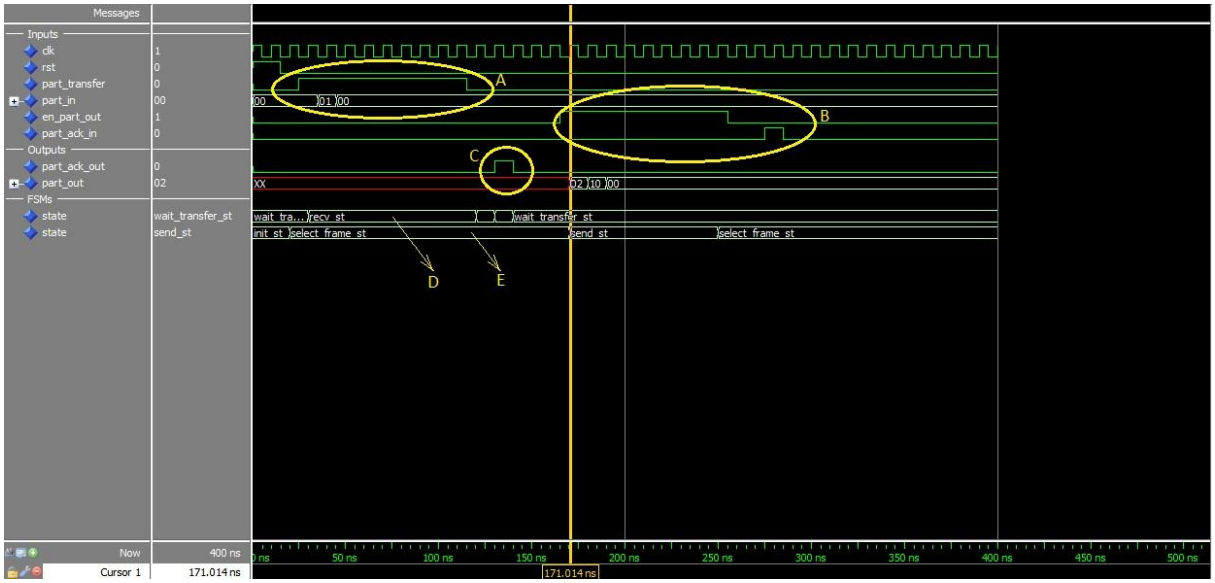


Figura 45: Tela de uma das simulações realizadas para verificar a implementação do gerenciador de comunicação na partição 1. Em (A) observa-se a janela de transferência em que o controle de recepção recebe um *token*. Após a recepção completa, o sinal de *acknowledge out* é ativado (C). Em (B) vê-se o sinal de habilitação de envio e posteriormente o sinal de *acknowledge in*, indicando que o pacote foi recebido. Em (D) e (E) é possível observar os estados das máquinas de recepção e de envio, conforme apresentado na seção 5.2.1.

6 Conclusão

Mesclando um dos temas mais novos da computação reconfigurável, a reconfiguração parcial, com um modelo de computação já largamente explorado, que é o da computação a fluxo de dados com *tagged-tokens*, o projeto ChipCFlow propõe uma abordagem inteiramente nova para a execução em *hardware* de programas descritos em CDFG, convertidos de programas originalmente escritos em C. Dentro deste contexto, o presente trabalho buscou apresentar um apanhado dos sistemas de computação reconfiguráveis e das arquiteturas baseadas no modelo a fluxo de dados, visando embasar uma proposta de particionamento e protocolo de comunicação nos grafos do projeto ChipCFlow. Há trabalhos extremamente promissores nas duas áreas, em especial os processadores híbridos WaveScalar e TRIPS.

No contexto do projeto ChipCFlow, há que se ressaltar que até então nenhum trabalho havia proposto uma solução completa e viável para o mapeamento dos grafos a fluxo de dados dinâmicos nos FPGAs parcialmente reconfiguráveis. Acredita-se que através da proposta apresentada, abriu-se pela primeira vez um caminho para um fluxo completo para obtenção das máquinas a fluxo de dados dinâmicas geradas a partir dos programas originais. Apresentou-se uma proposta para um mecanismo de comunicação completo entre as partições, bem como algumas regras simples de restrição na geração das partições.

De um ponto de vista puramente funcional, parece não haver grande empecilho na concretização da idéia do projeto, no entanto, um olhar mais atento na questão das temporizações, principalmente na relação entre o tempo de reconfiguração e o tempo de execução parece indicar a necessidade de uma abordagem para acelerar os tempos de reconfiguração ou ainda a adoção de outra abordagem para a viabilidade do projeto. É importante ressaltar que dentro do projeto ChipCFlow outras trabalhos já estão em andamento vizando uma análise mais profunda da proposta apresentada aqui.

Acredita-se, no entanto, que ainda assim as idéias centrais do projeto sejam promissoras, pois é bastante provável que a tecnologia de configuração caminhe no sentido de

obter tempos de reconfiguração cada vez menores, principalmente quando a tecnologia da reconfiguração parcial se consolidar comercialmente. Acredita-se que a redução do custo de chaveamento de configuração seja crucial para o futuro da reconfiguração parcial.

Trabalhos futuros certamente terão de lidar com a implementação completa da proposta aqui apresentada, principalmente para se obter conclusões a respeito da dinâmica de funcionamento da solução. Automatizar o processo de particionamento é também uma outra tarefa necessária, considerando o objetivo da ferramenta ChipCFlow.

Contrapondo os objetivo inicial apresentado na introdução, que consistia na apresentação de um modelo para particionamento e para comunicação dos grafos a fluxo de dados dinâmico, acredita-se que ele foi atendido, a menos, no entanto, de uma *validação* da proposta de forma mais acurada. Ressalta-se, todavia, que o próprio grupo de trabalho em torno do ChipCFlow optou por alocar essa tarefa ao escopo de um outro trabalho.

Espera-se que as descrições em VHDL produzidas para os comunicadores possa ser aproveitadas no restante do projeto, uma vez terem sido testadas funcionalmente como componente.

Referências

AHMADINIA, A. et al. A practical approach for circuit routing on dynamic reconfigurable devices. In: *In: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*. [S.l.: s.n.], 2005. p. 84–90.

ARVIND, K.; CULLER, D. E. Dataflow architectures. *Annual Review of Computer Science*, v. 1, p. 225–253, 1986.

ARVIND, K.; NIKHIL, R. S. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 39, n. 3, p. 300–318, 1990. ISSN 0018-9340.

ASTOLFI, V. F.; SILVA, J. L. e. Execution of algorithms using a dynamic dataflow model for reconfigurable hardware - commands in dataflow graph. *3rd Southern Conference on Programmable Logic, 2007. SPL '07*, p. 225–230, 18 June 2007.

ATHANAS, P. M.; SILVERMAN, H. F. Processor reconfiguration through instruction-set metamorphosis. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 26, n. 3, p. 11–18, 1993. ISSN 0018-9162.

BAUMGARTE, V. et al. Pact xpp—a self-reconfigurable data processing architecture. *J. Supercomput.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 26, n. 2, p. 167–184, 2003. ISSN 0920-8542.

BOBDA, C. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. 1. ed. [S.l.]: Springer, 2007. ISBN 1402060882.

BURGER, D. et al. Scaling to the end of silicon with EDGE architectures. *Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 37, p. 44–55, 2004. ISSN 0018-9162.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 34, n. 2, p. 171–210, 2002. ISSN 0360-0300.

DENNIS, J. B. Data flow supercomputers. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 13, n. 11, p. 48–56, 1980. ISSN 0018-9162.

DENNIS, J. B.; MISUNAS, D. P. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, v. 3, n. 4, p. 126–132, 1974.

EBELING, C.; CRONQUIST, D. C.; FRANKLIN, P. Rapid - reconfigurable pipelined datapath. In: *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*. London, UK: Springer-Verlag, 1996. p. 126–135. ISBN 3-540-61730-2.

GOEKE, M. et al. The firefly machine: Online evolware. In: *In Proc. of Int. Conf. on Evolutionary Computation (ICEC'97*. [S.l.]: IEEE Press, 1997. p. 181–186.

GOKHALE, M. et al. Splash: A reconfigurable linear logic array. In: *ICPP (1)*. [S.l.: s.n.], 1990. p. 526–532.

GURD, J. R.; KIRKHAM, C. C.; WATSON, I. The manchester prototype dataflow computer. *Commun. ACM*, ACM, New York, NY, USA, v. 28, n. 1, p. 34–52, 1985. ISSN 0001-0782.

HAUCK, S. The roles of fpgas in reprogrammable systems. In: *Proceedings of the IEEE*. [S.l.: s.n.], 1998. p. 615–638.

HAUCK, S.; DEHON, A. *Reconfigurable Computing:The Theory and Practice off FPGA-based computation*. 1. ed. [S.l.]: Morgan Kaufman Publishers, 2008. ISBN 9780123705228.

HAUSER, J. R.; WAWRZYNEK, J. Garp: A mips processor with a reconfigurable coprocessor. In: . [S.l.: s.n.], 1997. p. 12–21.

JUNIOR, F. de S. et al. Research and partial analysis of overhead of a partition model for a partially reconfigurable hardware in a data-driven machine - chipcflow. *6th Southern Conference on Programmable Logic 2010. SPL '10*, Ipojuca, PE, Brazil, 2010.

LEE, B.; HURSON, A. R. Issues in dataflow computing. *Advances in Computers*, v. 37, p. 285–333, 1993.

LOPES, J. J. *Estudos e avaliações de compiladores para arquiteturas reconfiguráveis*. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação (ICMC), São Carlos, SP, Brasil, maio 2007.

MESQUITA, D. G. *Contribuições para reconfiguração parcial, remota e dinâmica de FPGAs*. Dissertação (Mestrado) — Pontífice Universidade Católia do Rio Grande do Sul - Faculdade de Informática, Porto Alegre, RS, Brasil, março 2002.

MIRSKY, E.; DEHON, A. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In: *IEEE Symposium on FPGAs for Custom Computing Machines*. [S.l.: s.n.], 1996. p. 157–166.

MöLLER, L. H. *Sistemas Dinamicamente Reconfiguráveis com Comunicação via Redes Intra-Chip*. Dissertação (Mestrado) — Pontífice Universidade Católia do Rio Grande do Sul - Faculdade de Informática, Porto Alegre, RS, Brasil, dezembro 2005.

NAJJAR, W. A.; LEE, E. A.; GAO, G. R. Advances in the dataflow computational model. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v. 25, n. 13-14, p. 1907–1929, 1999. ISSN 0167-8191.

PACT XPP TECHNOLOGIES. *XPP-III Processor Overview*. [S.l.], July 2006. Disponível em: <http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf>.

PAPADIMITRIOU, K.; ANYFANTIS, A.; DOLLAS, A. Methodology and experimental setup for the determination of system-level dynamic reconfiguration overhead. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 335–336, 2007.

RADUNOVIC, B.; MILUTINOVIC, V. M. A survey of reconfigurable computing architectures. In: *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*. London, UK: Springer-Verlag, 1998. p. 376–385. ISBN 3-540-64948-4.

SANKARALINGAM, K. et al. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In: *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2003. p. 422–433. ISBN 0-7695-1945-8.

SILVA, J. L. e; COSTA, K. A. P. D.; RODA, V. O. The c compiler generating a source file in vhdl for a dynamic dataflow machine being executed direct into a hardware. *WSEAS Transactions on Computers*, v. 8, n. 12, p. 1908–1916, 2009. ISSN 1109-2750.

SILVA, J. L. e; MARQUES, E. Executing algorithms for dynamic dataflow reconfigurable hardware -the operators protocol. *Reconfigurable Computing and FPGAs, International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–7, 2006.

SWANSON, S. et al. The wavescalar architecture. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 25, n. 2, p. 1–54, 2007. ISSN 0734-2071.

VEEN, A. H. Dataflow machine architecture. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 18, n. 4, p. 365–396, 1986. ISSN 0360-0300.

VUILLEMIN, J. E. et al. Programmable active memories: reconfigurable systems come of age. Kluwer Academic Publishers, Norwell, MA, USA, p. 611–624, 2002.

WAINGOLD, E. et al. *Baring it all to Software: The Raw Machine*. Cambridge, MA, USA, 1997.

XILINX. *Early Access Partial Reconfiguration User Guide*. 1.2. ed. [S.l.], September 2008.

Anexo

Código de envio do comunicador adicionado à partição 1 do exemplo da dissertação, usado para validar o seu comportamento funcional.

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


library work;
use work.ccflow_pack.all;

  entity part_comm_control_out is
  generic
  (
    FRAME_WIDTH             : positive := 64;       --- data itself
    PACKET_WIDTH            : positive := 8;        --- width of communication lines beetween partitions
    INPUT_ARCS              : positive := 4;        ---
    OUTPUT_ARCS             : positive := 4;        ---
    OUTPUT_MASK             : std_logic_vector(15 downto 0) := x"000F";
    IS_ACTIVATION_PART      : boolean := true;      --- indicate that the
                                                    --- partition of this comm
                                                    --- controller is the first
                                                    --- one (activation entry
                                                    --- point)
    PARTITION_NUM           : std_logic_vector(7 downto 0) := x"01"
  );
  port
  (
    --- global signals
    clk                     : in std_logic;
    rst                     : in std_logic;
    --- communication input interconnection
    activationEnd           : in boolean;
    activation_num          : in std_logic_vector(7 downto 0);
    iteration_num           : in std_logic_vector(7 downto 0);
    --- comm outputs --------
    en_part_out             : in std_logic;
    part_ack_in             : in std_logic;
    part_out                : out std_logic_vector(PACKET_WIDTH-1 downto 0);
    ----
    part_ctrl_out_1         : in boolean;
    part_ctrl_out_1_str     : in std_logic;
    part_ctrl_out_1_ack     : out std_logic;

    part_dt_out_1           : in std_logic_vector(31 DOWNTO 0);
    part_dt_out_1_str       : in std_logic;
    part_dt_out_1_ack       : out std_logic;

    part_ctrl_out_2         : in boolean;
    part_ctrl_out_2_str     : in std_logic;
    part_ctrl_out_2_ack     : out std_logic;

    part_dt_out_2           : in std_logic_vector(31 DOWNTO 0);
    part_dt_out_2_str       : in std_logic;
    part_dt_out_2_ack       : out std_logic
  );
  end entity;


architecture rtl of part_comm_control_out is

-------------------------------------------------------------------------
--- States Declaration
type StateType is (INIT_ST, SELECT_FRAME_ST, SEND_ST);
signal State              : StateType;

-------------------------------------------------------------------------
--- registers
signal packets                          : integer range 0 to 48;
signal frame_reg                        : std_logic_vector(FRAME_WIDTH-1 downto 0);

--- indicates that a new data has arrived at partition output
signal graph_full_out                   : std_logic_vector(OUTPUT_ARCS-1 downto 0);
--- indicates that data has been sent from graph output
signal graph_sent_out                   : std_logic_vector(OUTPUT_ARCS-1 downto 0);

signal graph_out_0_reg                  : std_logic_vector(31 downto 0);
signal graph_out_1_reg                  : std_logic_vector(31 downto 0);
```

```
signal graph_out_2_reg                  : std_logic_vector(31 downto 0);
signal graph_out_3_reg                  : std_logic_vector(31 downto 0);

signal output_counter                   : integer range  0 to (2**OUTPUT_ARCS-1);

signal current_output_mask              : std_logic_vector(3 downto 0);

signal selected_reg                     : boolean;
signal executed_reg                     : boolean;
signal executed_condition               : boolean;

-------------------------------------------------------------------------
--- these must be defined prior to
--- compilation and dependes on graph structure
signal graph_out_0_dest                 : std_logic_vector(11 downto 0);
signal graph_out_1_dest                 : std_logic_vector(11 downto 0);
signal graph_out_2_dest                 : std_logic_vector(11 downto 0);
signal graph_out_3_dest                 : std_logic_vector(11 downto 0);

-------------------------------------------------------------------------
--- Combinatorial signals

--signal firstHeaderTag                   : std_logic_vector(HEADER_WIDTH+TAG_WIDTH-1 DOWNTO 0);


--signal part_out_aux                     : std_logic_vector(PACKET_WIDTH-1 downto 0);


 --- architecture begin
begin


-------------------------------------------------
--- Defines destination of outputs (Part of Header) (Partition (8 bits) + Input (4 bits))
--- These values must be defined previously, according to graph structure

graph_out_0_dest <= "00000010" & "0001";
graph_out_1_dest <= "00000010" & "0010";
graph_out_2_dest <= "00000011" & "0001";
graph_out_3_dest <= "00000011" & "0010";

-------------------------------------------------------------------------
--- Defines first header and tag  (TAG= activation (8 bits) + iteration (8 bits))
--firstHeaderTag <= graph_out_0_dest & NORMAL_DATA & "00000000" & "00000000";

-------------------------------------------------------------------------
--- partition output (packets goes out by here)
--part_out <= part_out_aux;

--- this condition must be defined for each partition
executed_condition <= true when graph_sent_out = "1111" ELSE false;

-------------------------------------------------------------------------
--- Finite State Machine Process

fsm: process (clk, rst, frame_reg, activation_num, iteration_num)


variable currentHeaderTag   : std_logic_vector(HEADER_WIDTH+TAG_WIDTH-1 DOWNTO 0)\
 := "00000000" & "0000" & NORMAL_DATA & activation_num & iteration_num;
variable currentData        : std_logic_vector(31 DOWNTO 0) := (OTHERS => '0');

begin
  if rst = '1' then
    current_output_mask <= (OTHERS => '0');
    --current_output_mask <= (OTHERS => '0');
    graph_full_out    <= (OTHERS => '0');
    graph_sent_out    <= (OTHERS => '0');
    --currentHeaderTag  <= "00000000" & "0000" & NORMAL_DATA & activation_num & iteration_num;
    --currentData       <= (OTHERS => '0');
    selected_reg      <= false;
    executed_reg      <= false;
    packets           <= 0;
    State <= INIT_ST;

  elsif rising_edge (clk) then

    --- store first output of the graph (it is a boolean)
    if part_ctrl_out_1_str = '1' then
      if graph_full_out(0) = '0' then
        if part_ctrl_out_1 then
          graph_out_0_reg <= (0 => '1', others=> '0');
        else
          graph_out_0_reg <= (others=> '0');
        end if;
        part_ctrl_out_1_ack <= '1';
        graph_full_out(0) <= '1';
      else
        part_ctrl_out_1_ack <= '0';
      end if;
    else
      part_ctrl_out_1_ack <= '0';
    end if;

    --- store second output of the graph
    if part_dt_out_1_str = '1' then
      if graph_full_out(1) = '0' then
        graph_out_1_reg <= part_dt_out_1;
```

```
    part_dt_out_1_ack <= '1';
    graph_full_out(1) <= '1';
  else
    part_dt_out_1_ack <= '0';
  end if;
else
  part_dt_out_1_ack <= '0';
end if;


--- store third output of the graph (it is a boolean)
if part_ctrl_out_2_str = '1' then
  if graph_full_out(2) = '0' then
    if part_ctrl_out_2 then
      graph_out_2_reg <= (0 => '1', others=> '0');
    else
      graph_out_2_reg <= (others=> '0');
    end if;
    part_ctrl_out_2_ack <= '1';
    graph_full_out(2) <= '1';
  else
    part_ctrl_out_2_ack <= '0';
  end if;
else
  part_ctrl_out_2_ack <= '0';
end if;

--- store fourth output of the graph
if part_dt_out_2_str = '1' then
  if graph_full_out(3) = '0' then
    graph_out_3_reg <= part_dt_out_2;
    part_dt_out_2_ack <= '1';
    graph_full_out(3) <= '1';
  else
    part_dt_out_2_ack <= '0';
  end if;
else
  part_dt_out_2_ack <= '0';
end if;


------------------------------------------------
--- Finite State Machine
case State is
  when INIT_ST =>
    graph_full_out    <= (OTHERS => '0');
    graph_sent_out    <= (OTHERS => '0');
    selected_reg      <= false;
    executed_reg      <= false;
    packets           <= 0;
    current_output_mask <= (0 => '1', OTHERS => '0');
    State <= SELECT_FRAME_ST;

  when SELECT_FRAME_ST =>
    if en_part_out = '1' then
      part_out <= frame_reg((PACKET_WIDTH*PACKET_NUM)-1 downto (PACKET_WIDTH*PACKET_NUM)-PACKET_WIDTH);
      packets <= 1;
      State <= SEND_ST;
    elsif not selected_reg then
      if not executed_reg then
        if (graph_full_out xor graph_sent_out) /= "0000" then
          graph_sent_out <= graph_sent_out OR current_output_mask;
          case current_output_mask is
            when "0001" =>
              if graph_full_out(0) = '1' and graph_sent_out(0) = '0' then
                currentHeaderTag := graph_out_0_dest & NORMAL_DATA & activation_num & iteration_num;
                currentData := graph_out_0_reg;
                selected_reg <= true;
              end if;
              current_output_mask <= "0010";

            when "0010" =>
              if graph_full_out(1) = '1' and graph_sent_out(1) = '0' then
                currentHeaderTag := graph_out_1_dest & NORMAL_DATA & activation_num & iteration_num;
                currentData := graph_out_1_reg;
                selected_reg <= true;
              end if;
              current_output_mask <= "0100";

            when "0100" =>
              if graph_full_out(2) = '1' and graph_sent_out(2) = '0' then
                currentHeaderTag := graph_out_2_dest & NORMAL_DATA & activation_num & iteration_num;
                currentData := graph_out_2_reg;
                selected_reg <= true;
              end if;
              current_output_mask <= "1000";

            when "1000" =>
              if graph_full_out(3) = '1' and graph_sent_out(3) = '0' then
                currentHeaderTag := graph_out_3_dest & NORMAL_DATA & activation_num & iteration_num;
                currentData := graph_out_3_reg;
                selected_reg <= true;
              end if;
              current_output_mask <= "0001";

            when others =>
              currentHeaderTag := "00000000" & "0000" & NORMAL_DATA & activation_num & iteration_num;
              currentData := (OTHERS => '0');
```

```
                selected_reg <= false;
                current_output_mask <= "0001";
            end case;
            frame_reg <= currentHeaderTag & currentData;
          else  --- there are no outputs to be sent yet
            currentHeaderTag := "00000000" & "0000" & NORMAL_DATA & activation_num & iteration_num;
            currentData := (OTHERS => '0');
            selected_reg <= false;
            frame_reg <= currentHeaderTag & currentData;
          end if;
        else --- executed_reg = '1' (it meens all the necessary outputs has been sent)
          ---- must send message of end of partition
          currentHeaderTag := partition_num & "0000" & PART_EXECUTED & activation_num & iteration_num;
          currentData := (OTHERS => '0');
          selected_reg <= true;
          frame_reg <= currentHeaderTag & currentData;
        end if;
      end if;

    when SEND_ST =>
      if packets = PACKET_NUM then
        if executed_condition then
          executed_reg <= true;
        end if;
        packets <= 0;
        State <= SELECT_FRAME_ST;
      else
        for n in 1 to PACKET_NUM-1 loop
          if packets = n then
            part_out <= frame_reg((PACKET_WIDTH*(PACKET_NUM - n))-1 downto ((PACKET_WIDTH*(PACKET_NUM - n))-PACKET_WIDTH));
          end if;
        end loop;
        packets <= packets + 1;
      end if;
      selected_reg <= false;

    when others =>
      State <= INIT_ST;
  end case;
 end if;
end process;


end architecture;
```

Código de recepção do comunicador adicionado à partição 1 do exemplo do texto, usado para validar o seu comportamento funcional.

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

library work;
use work.ccflow_pack.all;

  entity part_comm_control_in is
  generic
  (
    FRAME_WIDTH             : positive := 64;        ---
    PACKET_WIDTH            : positive := 8;        ---
    INPUT_ARCS              : positive := 4;        ---
    OUTPUT_ARCS             : positive := 4;        ---
    OUTPUT_MASK             : std_logic_vector(15 downto 0) := x"000F";
    IS_ACTIVATION_PART      : boolean := true;
    PARTITION_NUM           : std_logic_vector(7 downto 0) := x"01"
  );
  port
  (
    --- global input control signals
    clk                     : in std_logic;
    rst                     : in std_logic;
    --- comm inputs -----------------
    part_transfer           : in std_logic;
    part_in                 : in std_logic_vector(PACKET_WIDTH-1 downto 0);
    part_ack_out            : out std_logic;
    ---
    activationEnd           : out boolean;
    activation_num          : out std_logic_vector(7 DOWNTO 0);
    iteration_num           : out std_logic_vector(7 DOWNTO 0);
    -- graph connection points
    part_ctrl_in_1          : out boolean;
    part_ctrl_in_1_str      : out std_logic;
    part_ctrl_in_1_ack      : in  std_logic;

    part_dt_in_1            : out std_logic_vector(31 DOWNTO 0);
    part_dt_in_1_str        : out std_logic;
    part_dt_in_1_ack        : in  std_logic;

    part_ctrl_in_2          : out boolean;
    part_ctrl_in_2_str      : out std_logic;
    part_ctrl_in_2_ack      : in  std_logic;
```

```
    part_dt_in_2            : out std_logic_vector(31 DOWNTO 0);
    part_dt_in_2_str        : out std_logic;
    part_dt_in_2_ack        : in  std_logic

  );
  end entity;


architecture rtl of part_comm_control_in  is



--------------------------------------------------------------------------
--- States Declaration
type StateType is (WAIT_TRANSFER_ST, RECV_ST, PROCESS_ST, SEND_ACK_OUT_ST);
signal State              : StateType;

--------------------------------------------------------------------------
--- registers
signal packets                : integer range 0 to 48;
signal frame_reg              : std_logic_vector(FRAME_WIDTH-1 downto 0);
signal terminatedOutputs_reg  : std_logic_vector((2**INPUT_FRAME_FIELD_SIZE)-1 downto 0) := (OTHERS => '0');

signal activation_reg     : std_logic_vector(ACTIVATION_FRAME_FIELD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
signal iteration_reg      : std_logic_vector(ITERATION_FRAME_FIELD_SIZE-1 DOWNTO 0) := (OTHERS => '0');

--------------------------------------------------------------------------
--- Combinatorial signals
signal match              : boolean;

signal frame_aux          : std_logic_vector(FRAME_WIDTH-1 downto 0);

signal partition_field    : std_logic_vector(PARTITION_FRAME_FIELD_SIZE-1 DOWNTO 0);
signal input_field        : std_logic_vector(INPUT_FRAME_FIELD_SIZE-1 DOWNTO 0);
signal command_field      : std_logic_vector(COMMAND_FRAME_FIELD_SIZE-1 DOWNTO 0);
signal activation_field   : std_logic_vector(ACTIVATION_FRAME_FIELD_SIZE-1 DOWNTO 0);
signal iteration_field    : std_logic_vector(ITERATION_FRAME_FIELD_SIZE-1 DOWNTO 0);
signal data_field         : std_logic_vector(31 DOWNTO 0);

signal set_strobe         : std_logic;
signal rst_strobe         : std_logic;
signal strobe             : std_logic;

--signal activationEnd      : BOOLEAN;
signal set_iteration_end  : std_logic;

  --- architecture begin
begin

  partition_field   <= frame_reg(63 DOWNTO 56);
  input_field       <= frame_reg(55 DOWNTO 52);
  command_field     <= frame_reg(51 DOWNTO 48);
  activation_field  <= frame_reg(47 DOWNTO 40);
  iteration_field   <= frame_reg(39 DOWNTO 32);
  data_field        <= frame_reg(31 DOWNTO 0);

  activation_num    <= activation_reg;
  iteration_num     <= iteration_reg;


  match <= TRUE when partition_field = PARTITION_NUM and activation_field = activation_reg
                     and iteration_field = iteration_reg and IS_ACTIVATION_PART else
           TRUE when partition_field = PARTITION_NUM and activation_field = activation_reg
                     and NOT IS_ACTIVATION_PART else
           FALSE;


  markIterationEnd : process (clk, rst)
  begin
    if rst = '1' then
      terminatedOutputs_reg <= (OTHERS => '0');
    elsif rising_edge(clk) then
      if set_iteration_end = '1' THEN
        case input_field is
          when x"0" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0001";
          when x"1" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0002";
          when x"2" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0004";
          when x"3" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0008";
          when x"4" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0010";
          when x"5" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0020";
          when x"6" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0040";
          when x"7" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0080";
          when x"8" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0100";
          when x"9" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0200";
          when x"A" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0400";
          when x"B" => terminatedOutputs_reg <= terminatedOutputs_reg or x"0800";
          when x"C" => terminatedOutputs_reg <= terminatedOutputs_reg or x"1000";
          when x"D" => terminatedOutputs_reg <= terminatedOutputs_reg or x"2000";
          when x"E" => terminatedOutputs_reg <= terminatedOutputs_reg or x"4000";
          when x"F" => terminatedOutputs_reg <= terminatedOutputs_reg or x"8000";
          when others =>  null;
        end case;
      end if;
    end if;
  end process;


  activationEnd <= TRUE WHEN (terminatedOutputs_reg and OUTPUT_MASK) = OUTPUT_MASK else false;
```

```
--- debug:

    ---debug <= activationEnd;
--- frame <= frame_reg;
--- match_debug <= match;


    partitionInputs: process(clk, rst)
  begin
    if rst = '1' then
      frame_reg <= (OTHERS => '0');
      State <= WAIT_TRANSFER_ST;
      part_ack_out <= '0';

    elsif rising_edge(clk) then
      case State is
        when WAIT_TRANSFER_ST =>
          if part_transfer = '1' THEN
            frame_reg <= (OTHERS => '0');
            packets <= 1;
            State <= RECV_ST;
          end if;

        when RECV_ST =>
          if part_transfer = '0' then
            State <= PROCESS_ST;
          else
            packets <= packets + 1;
            frame_reg <= frame_aux OR frame_reg;
          end if;

        when PROCESS_ST =>
          case command_field is
            when NORMAL_DATA =>
              if match then
                part_ack_out <= '1';
                set_strobe <= '1';
              else
                set_strobe <= '0';
                part_ack_out <= '0';
              end if;
              ---
              --Agora precisa enviar os dados corretamente para a aresta de entrada
              State <= SEND_ACK_OUT_ST;

            when ITERATION_END =>
              if match then
                part_ack_out <= '1';
                set_iteration_end <= '1';
              else
                part_ack_out <= '0';
                set_iteration_end <= '0';
              end if;

              --- the partition should send an special command to reconfiguration manager
              State <= SEND_ACK_OUT_ST;

            when DESTROY_ACTIVATION =>
              --- should ignore the message (it only makes sense for reconfiguration manager)
              State <= WAIT_TRANSFER_ST;

            when SET_TAG =>
              activation_reg <= activation_field;
              iteration_reg  <= iteration_field;
              part_ack_out <= '1';
              --- should extract TAG received and store it as the TAG of activation for matching
              State <= WAIT_TRANSFER_ST;

            when others =>
              --- Error. Should ignore message
              State <= WAIT_TRANSFER_ST;
          end case;

        when SEND_ACK_OUT_ST =>
          part_ack_out <= '0';
          set_strobe <= '0';
          set_iteration_end <= '0';
          State <= WAIT_TRANSFER_ST;

        when others =>
          State <= WAIT_TRANSFER_ST;
      end case;
    end if;

  end process;

  aux: process(packets, part_in)
  variable integerAux : integer range 0 to 48;
  begin
    frame_aux <= (OTHERS => '0');
    for n in 1 to PACKET_NUM loop
      integerAux := (PACKET_NUM+1) - packets;
      if integerAux = n then
        frame_aux((PACKET_WIDTH*n)-1 downto ((PACKET_WIDTH*n)-PACKET_WIDTH)) <= part_in;
      end if;
    end loop;
  end process;
```

```
  strobeProcess: Process(clk, rst, part_ctrl_in_1_ack, part_dt_in_1_ack, part_ctrl_in_2_ack,
                         data_field, part_dt_in_2_ack, strobe, input_field)
    variable bool_aux : boolean := false;
-- this is a combinatorial process that must be altered according to the inputs of
-- current partition
  begin

    if rising_edge (clk) then
      if rst = '1' then
        strobe <= '0';
      else
        if set_strobe = '1' then
          strobe <= '1';
        elsif rst_strobe = '1' then
          strobe <= '0';
        end if;
      end if;
    end if;

    if data_field(0) = '0' then
      bool_aux := false;
    else
      bool_aux := true;
    end if;
    part_ctrl_in_1     <= bool_aux;
    part_dt_in_1       <= data_field;
    part_ctrl_in_2     <= bool_aux;
    part_dt_in_2       <= data_field;

    if strobe = '1' then
      --- envia o dado para a entrada correta do subgrafo
      -- este 'case' precisa ser alterado de acordo com as entradas de uma partição
      case input_field is
        when x"0" =>
          part_ctrl_in_1_str <= strobe;
          part_dt_in_1_str <= '0';
          part_ctrl_in_2_str <= '0';
          part_dt_in_2_str <= '0';

        when x"1" =>
          part_ctrl_in_1_str <= '0';
          part_dt_in_1_str <= strobe;
          part_ctrl_in_2_str <= '0';
          part_dt_in_2_str <= '0';

        when x"2" =>
          part_ctrl_in_1_str <= '0';
          part_dt_in_1_str <= '0';
          part_ctrl_in_2_str <= strobe;
          part_dt_in_2_str <= '0';

        when x"3" =>
          part_ctrl_in_1_str <= '0';
          part_dt_in_1_str <= '0';
          part_ctrl_in_2_str <= '0';
          part_dt_in_2_str <= strobe;

        when others =>
          part_ctrl_in_1_str <= '0';
          part_dt_in_1_str <= '0';
          part_ctrl_in_2_str <= '0';
          part_dt_in_2_str <= '0';
      end case;
    else  --- avoid infering latch
      part_ctrl_in_1_str <= '0';
      part_dt_in_1_str <= '0';
      part_ctrl_in_2_str <= '0';
      part_dt_in_2_str <= '0';
    end if;

    rst_strobe <= part_ctrl_in_1_ack or part_dt_in_1_ack or
                  part_ctrl_in_2_ack or part_dt_in_2_ack;
  end process;

end architecture;
```

Entidade top do comunicador da partição 1 do exemplo:

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


library work;
use work.ccflow_pack.all;


  entity part_comm_control is
  generic
  (
    FRAME_WIDTH         : positive := 64;       --- data+TAG+header
    PACKET_WIDTH        : positive := 8;        --- width of communication lines beetween partitions
    INPUT_ARCS          : positive := 4;        ---
```

```
    OUTPUT_ARCS         : positive := 4;        ---
    OUTPUT_MASK         : std_logic_vector(15 downto 0) := x"000F";
    IS_ACTIVATION_PART  : boolean := true;
    PARTITION_NUM       : std_logic_vector(7 downto 0) := x"01"
  );
  port
  (
    --- global input control signals
    clk                 : in std_logic;
    rst                 : in std_logic;
    --- comm inputs -----------------
    part_transfer       : in std_logic;
    part_in             : in std_logic_vector(PACKET_WIDTH-1 downto 0);
    en_part_out         : in std_logic;
    part_ack_in         : in std_logic;
    --- comm outputs --------
    part_ack_out        : out std_logic;
    part_out            : out std_logic_vector(PACKET_WIDTH-1 downto 0);
    ------------------------------------------------------------------------
    --- partition outputs (they are inputs for comm controller)--------------
    part_ctrl_out_1     : in boolean;
    part_ctrl_out_1_str : in std_logic;
    part_ctrl_out_1_ack : out std_logic;

    part_dt_out_1       : in std_logic_vector(31 DOWNTO 0);
    part_dt_out_1_str   : in std_logic;
    part_dt_out_1_ack   : out std_logic;

    part_ctrl_out_2     : in boolean;
    part_ctrl_out_2_str : in std_logic;
    part_ctrl_out_2_ack : out std_logic;

    part_dt_out_2       : in std_logic_vector(31 DOWNTO 0);
    part_dt_out_2_str   : in std_logic;
    part_dt_out_2_ack   : out std_logic;

    --executed            : in std_logic;

    --- partition inputs (they are outputs for comm controller)--------------
    part_ctrl_in_1      : out boolean;
    part_ctrl_in_1_str  : out std_logic;
    part_ctrl_in_1_ack  : in  std_logic;

    part_dt_in_1        : out std_logic_vector(31 DOWNTO 0);
    part_dt_in_1_str    : out std_logic;
    part_dt_in_1_ack    : in  std_logic;

    part_ctrl_in_2      : out boolean;
    part_ctrl_in_2_str  : out std_logic;
    part_ctrl_in_2_ack  : in  std_logic;

    part_dt_in_2        : out std_logic_vector(31 DOWNTO 0);
    part_dt_in_2_str    : out std_logic;
    part_dt_in_2_ack    : in  std_logic

---  debug               : out boolean ---_Vector(DATA_WIDTH-1 DOWNTO 0);
---    frame               : out std_logic_Vector(DATA_WIDTH-1 DOWNTO 0);
---    match_debug         : out boolean

  );
end entity;


architecture rtl of part_comm_control is


--- interconnection signals
signal activationEnd    : boolean;
signal activation_num   : std_logic_vector(7 downto 0);
signal iteration_num    : std_logic_vector(7 downto 0);


component part_comm_control_in is
  generic
  (
    FRAME_WIDTH         : positive := 64;       --- data itself
    PACKET_WIDTH        : positive := 8;        --- width of communication lines beetween partitions
    INPUT_ARCS          : positive := 4;        ---
    OUTPUT_ARCS         : positive := 4;        ---
    OUTPUT_MASK         : std_logic_vector(15 downto 0) := x"000F";
    IS_ACTIVATION_PART  : boolean := true;
    PARTITION_NUM       : std_logic_vector(7 downto 0) := x"01"
  );
  port
  (
    --- global input control signals
    clk                 : in std_logic;
    rst                 : in std_logic;
    --- comm inputs -----------------
    part_transfer       : in std_logic;
    part_in             : in std_logic_vector(PACKET_WIDTH-1 downto 0);
    part_ack_out        : out std_logic;
    ---
    activationEnd       : out boolean;
    activation_num      : out std_logic_vector(7 DOWNTO 0);
    iteration_num       : out std_logic_vector(7 DOWNTO 0);
    -- graph connection points
    part_ctrl_in_1      : out boolean;
```

```
    part_ctrl_in_1_str      : out std_logic;
    part_ctrl_in_1_ack      : in  std_logic;

    part_dt_in_1            : out std_logic_vector(31 DOWNTO 0);
    part_dt_in_1_str        : out std_logic;
    part_dt_in_1_ack        : in  std_logic;

    part_ctrl_in_2          : out boolean;
    part_ctrl_in_2_str      : out std_logic;
    part_ctrl_in_2_ack      : in  std_logic;

    part_dt_in_2            : out std_logic_vector(31 DOWNTO 0);
    part_dt_in_2_str        : out std_logic;
    part_dt_in_2_ack        : in  std_logic

  );
end component;


component part_comm_control_out is
  generic
  (
    FRAME_WIDTH             : positive := 64;       --- data itself
    PACKET_WIDTH            : positive := 8;        --- width of communication lines beetween partitions
    INPUT_ARCS              : positive := 4;        ---
    OUTPUT_ARCS             : positive := 4;        ---
    OUTPUT_MASK             : std_logic_vector(15 downto 0) := x"000F";
    IS_ACTIVATION_PART      : boolean := true;
    PARTITION_NUM           : std_logic_vector(7 downto 0) := x"01"
  );
  port
  (
    --- global signals
    clk                     : in std_logic;
    rst                     : in std_logic;
    --- communication input interconnection
    activationEnd           : in boolean;
    activation_num          : in std_logic_vector(7 downto 0);
    iteration_num           : in std_logic_vector(7 downto 0);
    --- comm outputs --------
    en_part_out             : in std_logic;
    part_ack_in             : in std_logic;
    part_out                : out std_logic_vector(PACKET_WIDTH-1 downto 0);
    ----
    part_ctrl_out_1         : in boolean;
    part_ctrl_out_1_str     : in std_logic;
    part_ctrl_out_1_ack     : out std_logic;

    part_dt_out_1           : in std_logic_vector(31 DOWNTO 0);
    part_dt_out_1_str       : in std_logic;
    part_dt_out_1_ack       : out std_logic;

    part_ctrl_out_2         : in boolean;
    part_ctrl_out_2_str     : in std_logic;
    part_ctrl_out_2_ack     : out std_logic;

    part_dt_out_2           : in std_logic_vector(31 DOWNTO 0);
    part_dt_out_2_str       : in std_logic;
    part_dt_out_2_ack       : out std_logic

    --executed                : in std_logic

  );
end component;




begin


comm_in: part_comm_control_in
  generic map
  (
    FRAME_WIDTH             => FRAME_WIDTH,
    PACKET_WIDTH            => PACKET_WIDTH,
    INPUT_ARCS              => INPUT_ARCS,
    OUTPUT_ARCS             => OUTPUT_ARCS,
    IS_ACTIVATION_PART      => IS_ACTIVATION_PART,
    PARTITION_NUM           => PARTITION_NUM
  )
  port map
  (
    --- global input control signals
    clk                     => clk,
    rst                     => rst,
    --- comm inputs -----------------
    part_transfer           => part_transfer,
    part_in                 => part_in,
    part_ack_out            => part_ack_out,
    ---
    activationEnd           => activationEnd,
    activation_num          => activation_num,
    iteration_num           => iteration_num,
    -- graph connection points
    part_ctrl_in_1          => part_ctrl_in_1,
```

```
    part_ctrl_in_1_str      => part_ctrl_in_1_str,
    part_ctrl_in_1_ack      => part_ctrl_in_1_ack,

    part_dt_in_1            => part_dt_in_1,
    part_dt_in_1_str        => part_dt_in_1_str,
    part_dt_in_1_ack        => part_dt_in_1_ack,

    part_ctrl_in_2          => part_ctrl_in_2,
    part_ctrl_in_2_str      => part_ctrl_in_2_str,
    part_ctrl_in_2_ack      => part_ctrl_in_2_ack,

    part_dt_in_2            => part_dt_in_2,
    part_dt_in_2_str        => part_dt_in_2_str,
    part_dt_in_2_ack        => part_dt_in_2_ack

  );


comm_out: part_comm_control_out
  generic map
  (
    FRAME_WIDTH             => FRAME_WIDTH,
    PACKET_WIDTH            => PACKET_WIDTH,
    INPUT_ARCS              => INPUT_ARCS,
    OUTPUT_MASK             => OUTPUT_MASK,
    IS_ACTIVATION_PART      => IS_ACTIVATION_PART,
    PARTITION_NUM           => PARTITION_NUM
  )
  port map
  (
    --- global signals
    clk                     => clk,
    rst                     => rst,
    ---
    activationEnd           => activationEnd,
    activation_num          => activation_num,
    iteration_num           => iteration_num,
    --- comm outputs --------
    en_part_out             => en_part_out,
    part_ack_in             => part_ack_in,
    part_out                => part_out,
    ----                      ----
    part_ctrl_out_1         => part_ctrl_out_1,
    part_ctrl_out_1_str     => part_ctrl_out_1_str,
    part_ctrl_out_1_ack     => part_ctrl_out_1_ack,

    part_dt_out_1           => part_dt_out_1,
    part_dt_out_1_str       => part_dt_out_1_str,
    part_dt_out_1_ack       => part_dt_out_1_ack,

    part_ctrl_out_2         => part_ctrl_out_2,
    part_ctrl_out_2_str     => part_ctrl_out_2_str,
    part_ctrl_out_2_ack     => part_ctrl_out_2_ack,

    part_dt_out_2           => part_dt_out_2,
    part_dt_out_2_str       => part_dt_out_2_str,
    part_dt_out_2_ack       => part_dt_out_2_ack

    --executed                => executed

  );


end architecture;
```