ChipCflow - uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável - operadores e grafos a fluxo de dados

Vasco Martins Correia

ChipCflow - uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável - operadores e grafos a fluxo de dados

Vasco Martins Correia

Orientador: ***Prof. Dr. Jorge Luiz e Silva***

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

USP - São Carlos
Janeiro/2009

Agradecimentos

À Deus, por ter me dado saúde, capacidade, persistência e sabedoria para realizar este trabalho.

Ao meu orientador prof. Dr. Jorge Luiz e Silva, que por seus conselhos, incentivo e constante bom humor, se tornou mais do que um orientador, um amigo. Agradeço pelo apoio dado nesta conquista.

Aos amigos do LabES e LCR, pela amizade, compreensão e ajuda ao longo do mestrado.

À todos professores que contribuíram para minha formação acadêmica.

À minha família, pelo apoio incondicional dado em todas as etapas de minha vida.

À minha noiva, Alexandra, amiga e companheira incansável, fonte de carinho e força nos momentos mais difíceis. Pessoa fundamental na minha trajetória e na concretização deste trabalho.

Agradaço a todas as pessoas que diretamente ou indiretamente contribuíram para o desenvolvimento deste trabalho.

Ao CNPq pelo apoio financeiro.

"Minha crença pessoal é que se a computação reconfigurável pretende ter sucesso ela deve criar uma metodologia para converter automaticamente um programa numa linguagem de programação padrão para o hardware do sistema."Scott Hauck

Resumo

ChipCflow é o projeto de uma ferramenta para execução de algoritmos escritos em linguagem C utilizando o modelo a fluxo de dados dinâmico em *hardware* com reconfiguração parcial. O objetivo principal do projeto *ChipCflow* é a aceleração da execução de programas por meio da execução direta em *hardware*, aproveitando ao máximo o paralelismo considerado natural do modelo a fluxo de dados. Em particular nesta parte do projeto, realizou-se a prova de conceito para a programação a fluxo da dados em *hardware* reconfigurável. O modelo de fluxo de dados utilizado foi o estático em plataforma sem reconfiguração parcial, dada a complexidade desse sistema, que faz parte de outro módulo em desenvolvimento no projeto *ChipCflow*.

Abstract

In order to convert C Language into hardware, a ChipCflow project, is a fundamental element to be used. In particular, dynamic dataflow architecture can be generated to produce a high level of parallelism to be executed into a partial reconfigurable hardware. Because of the complexity of the partial reconfigurable system, in this part of the project, a poof-of-concept was described as a program to be executed in a static reconfigurable hardware. The partial reconfiguration is a focus on another part of the ChipCflow project.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ASH	-	Application Specific Hardware
ASIC	-	Application Specific Integrated Circuit
CASH	-	Compiler Application Specific Hardware
CLB	-	Configurable Logic Block
CI	-	Integrated Circuit
CPLD	-	Complex Programmable Logic Device
DDFG	-	Dynamic Dataflow Graph
DPR	-	Dynamic Partial Reconfiguration
FIFO	-	First in first out
EEPROM	-	Electrically-Erasable Programmable Read-Only Memory
EPROM	-	Erasable Programmable Read-Only Memory
FPGA	-	Field Programmable Gate Array
ICMC	-	Instituto de Ciências Matemáticas e de Computação
ISE	-	Integrated Software Environment
MPGA	-	Mask Programmable Gate Array
NIG	-	New Iteration Generation
NTD	-	New Tag Destructor
NTM	-	New Tag Manager
PAL	-	Programmable Array Logic
PE	-	Processor Elements
PLA	-	Programmable Logic Array
PLD	-	Programmable Logic Device
RTR	-	Run Time Reconfiguration
SPLD	-	Simple Programmable Logic Device
SRAM	-	Static Random Access Memory
ULA	-	Unidade Lógica e Aritmética
USP	-	Universidade de São Paulo
VHDL	-	Very high-speed integrated circuit Hardware Description Language
VLSI	-	Very Large Scale Integration

CAPÍTULO

1

Introdução

1.1 Contextualização

Introduzido em 1985 pela empresa *Xilinx*, um *FPGA* (*Field Programmable Gate Array*) é um dispositivo reconfigurável que consiste em três partes principais: um conjunto de células lógicas programáveis, também chamados de blocos lógicos, uma rede de interconexões programáveis e um grupo de células de entrada/saída dispostos em volta do dispositivo (Bobda, 2007).

Todos os dispositivos *FPGAs* são por definição programáveis, ou seja, configurável pelo menos uma única vez. Os conceitos de reconfiguração estão relacionados com a possibilidade de reconfigurar o dispositivo muitas vezes ou constantemente se necessário.

Nos sistemas paralelos tradicionais um caminho para se obter maior desempenho é a exploração do paralelismo da aplicação por meio de múltiplos microprocessadores; nos sistemas computacionais reconfiguráveis o caminho é a implementação em *hardware* das partes do programa de aplicação computacionalmente mais intensas. Neles, a exploração do paralelismo existente em

uma aplicação tem a característica de ser um modelo intrinsecamente concorrente (*hardware*) em vez dos modelos baseados na estrutura seqüencial de *von-Neumann* utilizados pela maioria dos sistemas de suporte ao processamento paralelo (Dehon, 1996).

Durante os anos 70 até meados dos anos 90 existiram algumas implementações buscando máquinas cujas organizações eram "naturalmente"paralelas, é o caso das máquinas com arquiteturas a fluxo de dados (Dennis e Misunas, 1974; Gurd et al., 1985; Shimada et al., 1986; Veen, 1986; Sato et al., 1992; Silva, 1992; Swanson et al., 2003; Arvind, 2005).

A diferença fundamental entre as arquiteturas tradicionais *von Neumann* e as arquiteturas a fluxo de dados está exatamente no controle do fluxo de informações que no caso das máquinas *von Neumann* é baseado no controle seqüencial, enquanto que nas máquinas a fluxo de dados o controle é executado pela presença dos dados (Cappelli et al., 2004; Swanson et al., 2003).

A arquitetura a fluxo de dados explora, de modo simples e natural, paralelismo de granularidade fina, uma vez que execução de cada instrução é dirigida somente pela disponibilidade de seus valores de entrada. Por este motivo, as máquinas a fluxo de dados têm execução assíncrona, que é uma característica desejável para o processamento paralelo.

Um grafo a fluxo de dados é um conjunto de operadores interligados por arcos. A presença dos dados em cada arco de um operador é que irá disparar a execução desse operador.

Existem dois modelos de arquiteturas a fluxo de dados: estático e dinâmico. No modelo estático apenas um dado pode estar presente em um arco esperando pelos seus dados parceiros. Um protocolo deve garantir o sincronismo entre os operadores para garantir que apenas um dado esteja presente em cada arco. Consequentemente o paralelismo fica limitado ao conjunto de operadores cujos arcos possuem apenas um dado.

No modelo dinâmico mais que um dado pode estar presente no arco. Consequentemente o paralelismo fica limitado ao conjunto de operadores cujos arcos possuem apenas um dado. No modelo dinâmico mais que um dado pode estar presente no arco. Um protocolo deve assegurar que os dados presentes nos arcos disparem as operações cujos dados parceiros também estejam presentes. O *"Tagged-Token"* é usado no protocolo para controlar os dados parceiros em cada arco (Arvind, 2005). Neste caso, o paralelismo acontece quando da execução de vários operadores cujos dados estejam todos disponíveis.

1.2 Motivação

Segundo (Cardoso, 2000), acredita-se que a comunidade de *software* tem o predomínio no desenvolvimento de sistemas eletrônicos digitais. Enquanto não existir um suporte a implementação de sistemas reconfiguráveis a partir de algoritmos em alto nível, os programadores de *software* não se sentirão atraídos pelo desenvolvimento de aplicações neste modelo. Para tornar esse processo mais atrativo, seriam necessárias ferramentas que a partir de uma descrição de uma aplicação em uma linguagem de alto nível gerasse o código objeto para ser executado em um microprocessador embarcado e os arquivos de descrição de *hardware* necessários para programação em sistemas reconfiguráveis.

O *ChipCflow* é o projeto de uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em *hardware* reconfigurável. Tem como principal objetivo utilizar o modelo de arquitetura a fluxo de dados, associado ao conceito de arquiteturas reconfiguráveis para acelerar a execução de algoritmos em alto nível, tais como programas escritos em linguagem Ansi-C. Essa aceleração vai acontecer por meio da execução direta em *hardware*, aproveitando ao máximo o paralelismo considerado natural do modelo a fluxo de dados.

A partir de algoritmos em alto nível escritos originalmente na linguagem C, se extrai os grafos a fluxo de dados do programa de aplicação tendo como referência para os operadores, uma base de grafos a fluxo de dados a ser gerada especificamente para essa ferramenta. O grafo a fluxo de dados gerado já estará otimizado. O grafo então é convertido em *VHDL* (*Very High Speed Integrated Circuit Hardware Description Language*), tendo como base todo o conjunto de operadores propostos para o *ChipCflow*, previamente implementados em *VHDL*. O código *VHDL* é então sintetizado e simulado em ferramenta *EDA* comercial, em particular o trabalho será desenvolvido no *ISE* (*Integrated Software Environment*) da *Xilinx*, e o *bitstream*[1] será então gerado para ser executado direto no *hardware*.

[1] Arquivo de configuração do *FPGA*, gerado pelo ISE da Xilinx.

1.3 Objetivo

O *ChipCflow* é o projeto de uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em *hardware* reconfigurável. Tem como principal objetivo utilizar o modelo de arquitetura a fluxo de dados associado ao conceito de arquiteturas reconfiguráveis, com a intenção de acelerar a execução de algoritmos em alto nível escritos em linguagem ansi-C.

Em particular, pretende-se validar conceitos no modelo de programação a fluxo de dados, sendo o principal objetivo deste trabalho o desenvolvimento dos operadores de máquinas a fluxo de dados propostos no projeto *ChipCflow* e sua execução em *hardware* através da implementação de grafos a fluxo de dados estáticos.

1.4 Organização

No capítulo 2 é apresentada a evolução dos circuitos digitais, tecnologias de fabricação de circuitos digitais, as características de reconfigurabilidade dos *FPGAs*, tecnologias de programação, a arquitetura interna dos blocos lógicos dos *FPGAs* e por fim será mostrada a tecnologia para reconfiguração parcial e dinâmica.

No capítulo 3 são apresentadas as máquinas a fluxo de dados e algumas arquiteturas existentes.

No capítulo 4 é apresentado o projeto *ChipCflow*, a estrutura do modelo a fluxo de dados, a estrutura *matching* de dados e a ferramenta de conversão C em grafos a fluxo de dados.

No capítulo 5 é descrito o desenvolvimento do trabalho, detalhando a implementação dos operadores utilizados no projeto *ChipCflow* e a construção de grafos a fluxo de dados.

No capítulo 6 são apresentadas as considerações finais e trabalhos futuros.

CAPÍTULO

2

Computação Reconfigurável

2.1 Evolução dos FPGAs

A tecnologia de circuitos digitais vem se desenvolvendo rapidamente nas últimas décadas ocasionando uma ampla transformação em todo o processo de desenvolvimento de *hardware*. Os elementos empregados em projetos de sistemas evoluíram de transistores a circuitos integrados *VLSI* (*Very Large Scale Integration*) (Souza, 1998).

Os *CIs* (*Integrated Circuit*) digitais podem ser construídos utilizando-se de diversas tecnologias diferentes, a escolha da tecnologia adequada deve ser realizada com base no tipo de projeto que se pretente executar (Ribeiro, 2002). A Figura 2.1 apresenta as tecnologias para projetos de sistemas digitais.

As tecnologias de implementação de circuitos digitais, conforme a Figura 2.1 podem ser agrupadas em dois grandes grupos: circuitos customizados (*ASICs*) e circuitos semi-customizados.

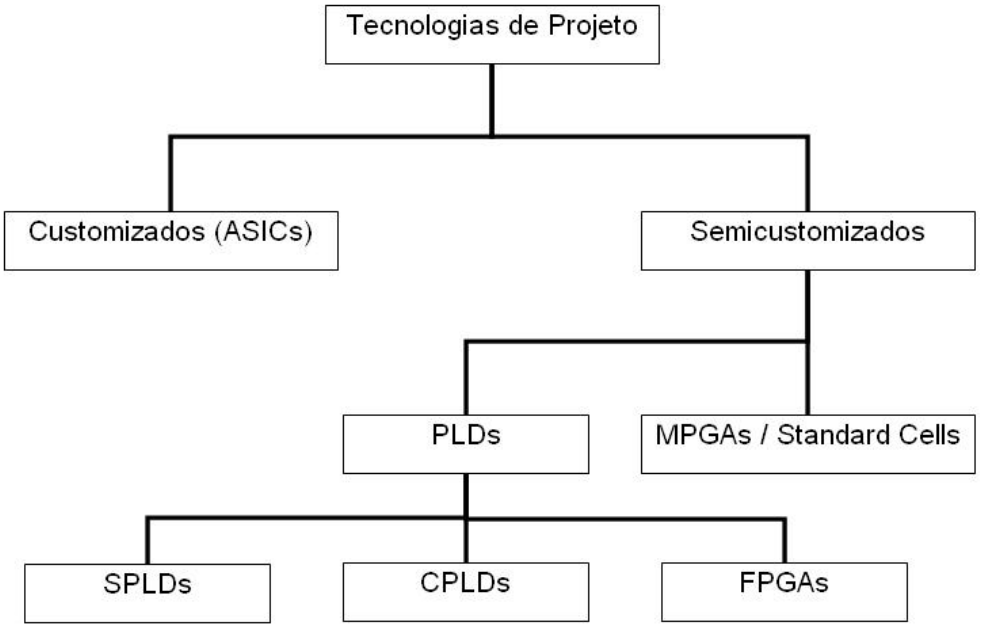


Figura 2.1: Tecnologias para Projetos de Sistemas Digitais.

Os circuitos customizados são indicados para aplicações e sistemas que necessitam de grande desempenho, baixo consumo de energia. As características desse tipo de implementação são os custos de projeto extremamente altos e o tempo de desenvolvimento longo. Em aplicações que requerem um grande volume de produção, o alto custo do projeto e dos testes é amortizado.

Os circuitos semi-customizados, se dividem ainda em outras duas categorias: *PLDs* (*Programmable Logic Devices*), *MPGAs* (*Mask-programmable Gate Array*)/*Standard Cells*.

Os *MPGAs* e *Standard Cells* foram agrupadas, devido suas semelhanças. Em comparação aos circuitos customizados, esses circuitos são menos eficientes em tamanho e desempenho, entretanto, seu custo de desenvolvimento é menor. Os *PLDs* possuem como principal característica a capacidade de programação (configuração) pós-fabricação pelo usuário, facilitando assim as eventuais mudanças de projetos. Eles Possibilitam colocar muitas portas em um único circuito integrado e controlar eletronicamente a conexão entre elas. Em comparação com outras tecnologias, os *PLDs* apresentam ciclo de projeto muito curto com baixos custos de desenvolvimento. Os *PLDs* se dividem em três categorias: *SPLDs* (*Simple Programmable Logic Device*), *CPLDs* (*Complex Programmable Logic Device*) e *FPGAs* (Ribeiro, 2002; Souza, 1998).

Os *SPLDs* correspondem a categoria de todos os pequenos *PLDs* como *PLAs* (*Programmable Logic Array*), *PALs (Programmable Array Logic*). As características mais importantes dessa ca-

tegoria é o baixo custo e alto desempenho. Esses dispositivos possuem apenas algumas centenas de portas lógicas. Os *CPLDs*, são constituídos de múltiplos *SPLDs* integrados em um único *chip*, apresentam interconexões programáveis para conectar os blocos *SPLDs*. Apesar de possuírem uma capacidade maior, comparados aos *SPLDs*, os *CPLDs* ainda são muito pequenos para serem utilizados em dispositivos de computação reconfigurável (Bobda, 2007)

Já os *FPGAs* são constituídos de um arranjo de elementos de circuitos não conectados: blocos lógicos e recursos de interconexão. A configuração é feita pelo usuário. A função a ser implementada no *FPGA* é dividida em módulos, cada qual pode ser implementada em um bloco lógico. Os blocos lógicos são então conectados juntos usando a interconexão programada (Bobda, 2007; Lopes, 2008).

2.2 Tecnologia dos FPGAs

No ano de 1985 foi introduzido pela Xilinx Inc. o *FPGA*, um dispositivo reconfigurável que consiste como os *CPLDs*, de três partes principais: um conjunto de células lógicas programáveis, também chamados de blocos lógicos, uma rede de interconexões programáveis e um grupo de células de entrada/saída dispostos em volta do dispositivo (Bobda, 2007; Lopes, 2008).

Os *FPGAs* são dispositivos reprogramáveis em campo, podendo ser reconfigurado muitas vezes ou constantemente se necessário. Muitas aplicações emergentes em multimídia e processamento de imagens necessitam que suas funcionalidades permaneçam flexíveis mesmo depois que o sistema tenha sido manufaturado. Tal flexibilidade é fundamental, uma vez que os requisitos das aplicações e as características dos sistemas podem mudar durante a vida do produto. Essa flexibilidade propicia novas abordagens de projeto voltadas para ganhos de desempenho, redução dos custos do sistema e/ou redução do consumo geral de energia.

Os *FPGAs* fornecem um *chip* pré-fabricado e totalmente reprogramado por milhares de vezes, essa capacidade é sua maior característica. Na realidade, um mesmo chip poderia assumir arquiteturas e funções completamente diferentes, sem a necessidade de mudanças do mesmo, em função apenas da necessidade do usuário, dessa forma facilita as mudanças em projetos, o que possibilita um curto ciclo de projeto e consequentemente baixo custo. Essas características garantem a atuali-

zação constante do produto com um mesmo estoque de CIs. Como nos processadores, *FPGAs* são programados após a fabricação para solucionar virtualmente qualquer tarefa computacional, isto é, qualquer tarefa que caiba nos recursos finitos do dispositivo (Souza, 1998).

2.2.1 Arquitetura dos FPGAs

Segundo (Ribeiro, 2002), a arquitetura básica de um *FPGA*, apresentada na Figura 2.2, consiste de um arranjo 2-D de blocos lógicos. A comunicação entre blocos é feita por meio dos recursos de interconexão (interconexão programável). A borda externa do arranjo consiste de blocos especiais capazes de realizar operações de entrada e saída (blocos de I/O), esses blocos fazem a interface do *FPGA* com o mundo externo, em placas de prototipação, eles fazem a interface do *FPGA* com os demais recursos da placa, como por exemplo, memórias externas.

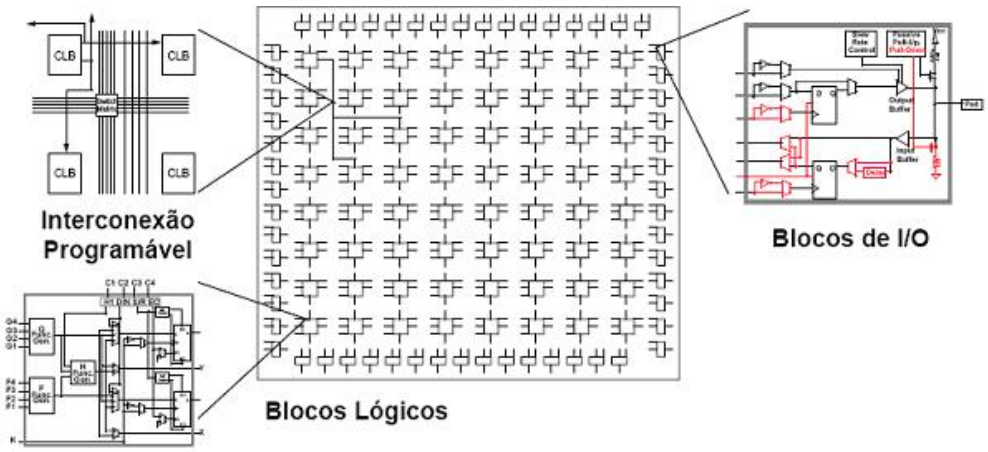


Figura 2.2: Arquitetura de um FPGA XILINX família 4000.

A arquitetura de um *FPGA* está, basicamente, composta de 3 partes. A primeira são **Blocos Lógicos Configuráveis** (Blocos lógicos). Um bloco lógico de um FPGA pode ser tão simples como um transistor ou tão complexo quanto um microprocessador. Ele é capaz de implementar várias funções lógicas combinacionais - tipicamente por meio de *LUT* (*Lookup Tables*) ou seqüenciais (por meio de *flip-flops*). As funções lógicas são implementadas no interior dos Blocos Lógicos. Em algumas arquiteturas os Blocos Lógicos possuem recursos seqüenciais tais como *flip-flop* ou registradores. O fabricante Xilinx chama seu Bloco Lógico de CLB (*Configurable Logic Block*).

Cada bloco lógico está conectado a um número determinado de matrizes de conexões programáveis, que, por sua vez, estão ligadas a um número de matrizes de chaveamento programáveis. Programando as conexões apropriadas, cada bloco pode fornecer uma variedade de funções lógicas combinacionais e/ou sequenciais, ou seja, qualquer função lógica desejada.

Como mostra a Figura 2.3, o bloco computacional básico nos *FPGAs* da Xilinx consistem de uma *LUT* com um variado número de entradas, um grupo de multiplexadores, lógica aritmética e elementos de armazenamento (Bobda, 2007).

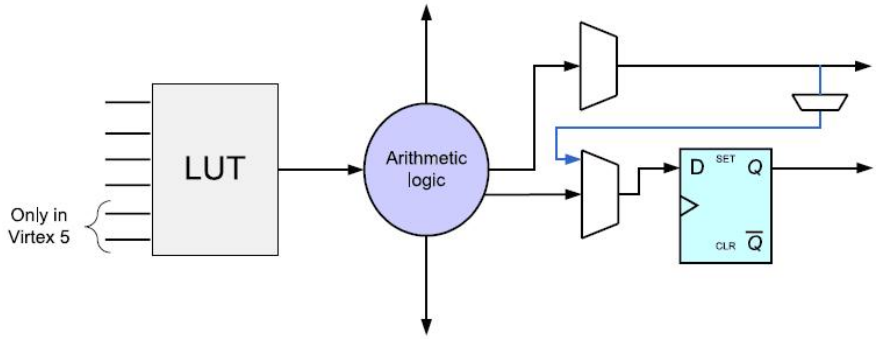


Figura 2.3: Bloco básico dos *FPGAs* Xilinx.

A *LUT* é um agrupamento de células de memória do tipo *SRAM* (*Static RAM*), que contém todos os resultados possíveis de uma dada função para um dado grupo de valores de entrada. Assim, uma *LUT* é capaz de implementar qualquer função booleana de até "n"variáveis, fazendo todas as possíveis combinações de valores entre elas. Os valores das funções são armazenados de tal modo que eles podem ser restaurados pelos correspondentes valores de entradas (Bobda, 2007).

Essa memória é carregada com os valores dados pelo arquivo de configurações *bitstream*. Esse arquivo é gerado pelas ferramentas de síntese de *FPGAs* e é composto pelas funcionalidades implementadas pelo usuário nessas ferramentas.

O número de blocos básicos em um *CLB* varia de dispositivo para dispositivo. No *FPGA Virtex-II Pro* e *Virtex-4*, os *CLBs* são divididos em quatro *slices* [1] cada qual contém dois blocos básicos. Os *CLBs* do *Virtex-5* contêm apenas dois *slices*, cada qual contém quatro blocos básicos (Bobda, 2007; Lopes, 2008).

A segunda parte que compõem a arquitetura de um *FPGA* são as **Interconexões Programáveis** (*Programmable Interconnect*). As matrizes de conexões programáveis são usadas para estabelecer

[1] Na família *Virtex* da Xilinx, a menor unidade lógica configurável ou unidade lógica básica é denominada *slice*.

ligações entre entradas e saídas dos blocos lógicos, enquanto as matrizes de chaveamento programáveis são empregadas para rotear os sinais entre as várias matrizes de conexões, conforme a Figura 2.4. Nos dispositivos *Xilinx*, os *CLBs* são embutidos em estruturas de roteamento que consiste de linhas verticais e horizontais. Cada *CLB* está amarrado a uma matriz de conexão para acessar a estrutura de roteamento geral. A matriz de conexão provê multiplexadores programáveis, os quais são utilizados para selecionar os sinais de um dado canal de roteamento que deveria ser conectado aos terminais dos *CLBs*. A matriz de conexão pode também conectar linhas horizontais e verticais, assim permitindo o roteamento completo no *FPGA* (Bobda, 2007).

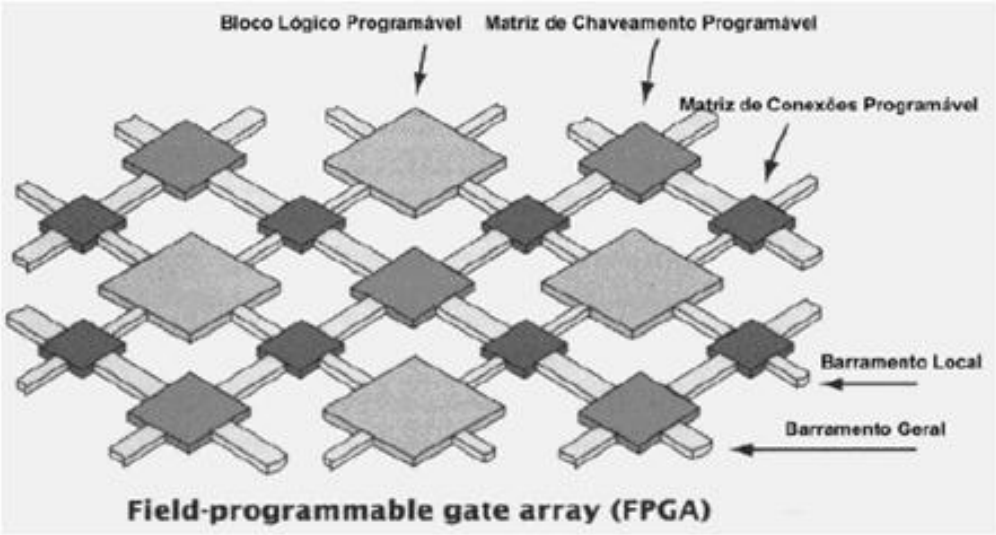


Figura 2.4: Conexões entre blocos lógicos.

A terceira parte da arquitetura do *FPGA* são os **Blocos de Entrada/Saída Configuráveis** (*I/O Blocks*). Os *FPGAs* possuem componentes de entrada/saída chamados de *IOB* (*I/O Block*), formados por estruturas bidirecionais que incluem *buffer*, *flip-flop* de entrada, *buffer tri-state* [2] e *flip-flop* de saída. Ou seja, assim como os *CLBs*, cada pino de *E/S* do componente pode ser programado pelo usuário. Um *FPGA* pode dispor de dezenas a centenas de pinos programáveis.

Estes aspectos influenciam diretamente no desempenho e na densidade das diferentes arquiteturas de *FPGAs*, entretanto não se pode afirmar que há uma melhor arquitetura, e sim a mais adequada para uma determinada aplicação (Ribeiro, 2002; Lopes, 2008). Os componentes de en-

[2] Em um barramentos de dados somente um dispositivo pode operar por vez. Os buffers tri-state são utilizados para isolar os dispositivos que não estão em uso do acesso ao barramento.

trada/saída são localizados em volta, na periferia do dispositivo e permitem a comunicação do dispositivo com os dispositivos localizados fora do *chip*.

Os *IOBs* (*input/output block*) podem ser utilizados de forma independente um dos outros como entradas e/ou saídas ou eles podem ser combinados em grupos de dois para ser utilizados como pares diferenciais diretamente conectados na matriz de conexão (Bobda, 2007).

As famílias de FPGAs diferem principalmente nas formas de realizar a programação, nas formas de organização dos condutores de interconexão e no funcionamento dos blocos lógicos.

2.2.2 Tecnologias de Programação

Um *FPGA* é programado usando comutadores programáveis eletricamente. As propriedades desses comutadores, tais como tamanho, resistência, capacitância, tecnologia, afetam principalmente o desempenho e definem características como volatilidade e capacidade de reprogramação, que devem ser avaliadas na fase inicial do projeto para a escolha do dispositivo (Souza, 1998; Ribeiro, 2002). Em todos os tipos de *FPGAs* os comutadores programáveis ocupam uma grande área. Existem basicamente dois tipos de tecnologias de programação principais: *Antifuse* e *SRAM* (Bobda, 2007).

O *antifuse* é um dispositivo de dois terminais, que no estado não programado apresenta uma alta impedância entre seus terminais. Quando aplicado uma tensão entre 11 e 20 Volts o antifuse "queima", criando uma conexão de baixa impedância, não permitindo reprogramações posteriores.

Enquanto a tecnologia *antifuse* está limitada a realização das interconexões, a tecnologia *SRAM* é usada para a computação, bem como para interconexão (Bobda, 2007; Lopes, 2008).

Entre as vantagens do *antifuse* está o seu tamanho reduzido (Bobda, 2007). A desvantagem está no espaço extra gasto para conseguir isolar os transistores no circuito de programação, já que eles trabalham com tensão de até 20V.

A tecnologia de programação *SRAM*, utiliza células memória *SRAM* para controlar transistores de passagem ou multiplexadores. A maior vantagem, segundo (Bobda, 2007) é que os *FPGAs* com essa tecnologia podem ser reprogramados indefinidamente, pois é necessário apenas modificar os valores nas células *SRAM* para realizar uma nova conexão ou uma nova função.

Como desvantagem, a tecnologia *SRAM*, ao contrário da tecnologia *antifuse*, é uma memória volátil, portanto perde sua configuração na ausência de eletricidade, necessitando de uma memória externa, como uma *EPROM* ou *EEPROM* ou um dispositivo não volátil de armazenamento, tal como o *CPLD* para guardar suas configurações na ausência de eletricidade. Outra desvantagem é que essa tecn'ologia ocupa muito espaço no *chip*, pois para cada comutador estão associados pelo menos 6 transistores.

Os recentes *FPGAs* e dispositivos reconfiguráveis utilizam a tecnologia de programação *SRAM*. Porém, Segundo (Compton e Hauck, 2002; Lopes, 2008), o termo *SRAM* é tecnicamente incorreto para muitas arquiteturas de *FPGAs*, isto porque, a memória de configuração nem sempre suporta acesso randômico. Segundo (Bobda, 2007), um FPGA pode ser programado uma ou várias vezes, dependendo da tecnologia utilizada.

2.2.3 Arquitetura dos blocos lógicos

Os *FPGAs* possuem uma grande variedade de tamanhos e com muitas combinações diferentes de características internas e externas. O que eles têm em comum é o fato de serem compostos por blocos lógicos configuráveis. Em um *FPGA*, a estrutura interna dos blocos lógicos, como por exemplo, *LUTs*, registradores, multiplexadores, podem ser replicadas milhares de vezes para construir um grande dispositivo reconfigurável (Compton e Hauck, 2002; David e Scott, 2005).

Em *FPGAs* mais complexos, esses blocos lógicos são combinados com lógicas digitais maiores, como lógicas aritméticas, estruturas de controles, tais como multiplicadores e contadores.

Com a finalidade de classificar os *FPGAs* quanto à capacidade lógica dos blocos lógicos, pode-se dividi-los em três categorias de granularidade: fina, media e grossa (Ribeiro, 2002; Compton e Hauck, 2002).

Cada bloco lógico pode ser simples como um *LUT* de três entradas ou complexo como uma ULA (Unidade Lógica Aritmética) de 4 bits. Essas diferenças entre os blocos podem ser referenciadas como granularidade do bloco lógico, no qual uma *LUT* de três entradas é um exemplo de um bloco lógico de granularidade fina e uma ULA de 4 bits é um exemplo de bloco lógico de granularidade grossa (Compton e Hauck, 2002).

Os blocos de granularidade fina são úteis para manipulações em nível de bits. Eles manipulam apenas 2 ou 3 valores de 1 bit.

O melhor exemplo para um bloco de granularidade fina seria um bloco contendo alguns transistores interconectáveis ou portas lógicas básicas. Para se implementar um somador, deve-se utilizar várias dessas células (Cardoso, 2000). Como vantagem, esse tipo de bloco é quase totalmente utilizado, fornecendo um alto grau de funcionalidade com um número relativamente pequeno de transistores. A desvantagem principal é que por serem muito pequenos, eles tem baixa capacidade lógica, assim, podem ser requeridos muitos blocos lógicos em uma determinada aplicação, elevando a quantidade de trilhas de conexões e comutadores programáveis o que sobrecarrega o roteador. Um roteador desse tipo de *FPGA* se torna lento e ocupa grande área do *chip*. Assim, a tecnologia *antifuse* é a mais adequada para a fabricação desse tipo de *FPGA* devido o seu tamanho reduzido. Um exemplo de um *FPGA* fabricado com essa granularidade é o *FPGA XC6200* da *Xilinx* (Ribeiro, 2002).

Os blocos de granularidade media operam com 2 ou mais dados com 4 bits de tamanho. Isso aumenta o número total de linhas de entrada do circuito e prove estruturas computacionais mais eficientes para problemas computacionais mais complexos. Similares aos blocos lógicos com granularidade fina, porém, esses podem executar operações mais complexas com um grande número de entradas. Esses blocos podem implementar operações lógico-aritméticas (Cardoso, 2000). Esse tipo de estrutura também pode ser usada para implementar operações mais complexas, tais como máquinas de estado finito (Compton e Hauck, 2002).

Os *FPGAs* da família *Virtex* da *Xilinx* são exemplos de dispositivos de granularidade média (Ribeiro, 2002).

Os blocos de granularidade grossa são úteis para aplicações que manipulam grandes quantidades de bits, porque o bloco lógico é otimizado para aplicações computacionais grandes, por isso eles executam as operações dessas aplicações muito rapidamente, consumindo menos área. Porém, devido a sua composição ser estática, eles são incapazes de ser otimizados para se adequar aos tamanhos dos operandos. Se, por exemplo, somadores, multiplicadores forem compostos por 16 bits e apenas valores de 1 bit forem processados, então o uso dessa arquitetura não oferece

vantagens, devido ao fato de todos os 16 bits serem computados e o gasto desnecessário de área e *overhead* (Compton e Hauck, 2002).

Segundo (Cardoso, 2000), esses blocos são verdadeiros núcleos de processamento, que muitas vezes incorporam processadores e memórias acoplados a uma matriz de lógica reconfigurável com arquiteturas que podem ser blocos de granularidades finas e medias.

Os blocos de granularidade grossa requerem menos *overhead* de comunicação entre os processos no *FPGA*. Se cada processo mantiver sua memória local e tiver delineado uma tarefa para ser executada, então, a aplicação poderá ser facilmente dividida entre diferentes áreas do *FPGA*.

2.2.4 FPGAs Dinâmicamente Reconfiguráveis

Atualmente, alguns *FPGAs* suportam a reconfiguração parcial, onde um *bitstream* menor pode ser utilizado para reconfigurar uma área do *FPGA*. Reconfiguração parcial dinâmica, em inglês *DPR* (*Dynamic Partial Reconfiguration*) é feito enquanto o dispositivo está ativo: certas áreas do dispositivo podem ser reconfiguradas enquanto que outras áreas continuam em operação e não são afetadas pela reconfiguração (Tadigotla et al., 2006).

Em (Bobda, 2007), o termo *run-time reconfiguration* é utilizado para especificar reconfiguração parcial e dinâmica. A implementação de um sistema no modo *run-time reconfiguration*, a computação e a seqüência de configuração do projeto não são conhecidas em tempo de compilação, assim, os requisitos para implementar uma dada tarefa é conhecida no tempo de execução. *Run-time reconfiguration* é um procedimento difícil, pois deve-se lidar com fatores de efeitos colaterais, tais como desfragmentação do dispositivo e comunicação entre os módulos recentemente colocados nele (Bobda, 2007).

A reconfiguração parcial e dinâmica de *FPGAs* exige dois tipos de particionamento:

1. Particionamento espacial: forma espacial com que os módulos de hardware serão distribuídos nos recursos de lógica reconfigurável;

2. Particionamento temporal: definição dos intervalos de tempo em que cada módulo de hardware deverá está presente no FPGA, já que algum módulo reconfigurável pode ocupar recursos no dispositivo, já ocupado por outro (Bobda, 2007).

Conforme ilustra a Figura 2.5, a administração do dispositivo reconfigurável é usualmente feitas por um escalonador e um *placer*[3] que podem ser implementados como parte de um sistema operacional executando em um processador, que pode residir dentro ou fora do *chip* (Bobda, 2007).

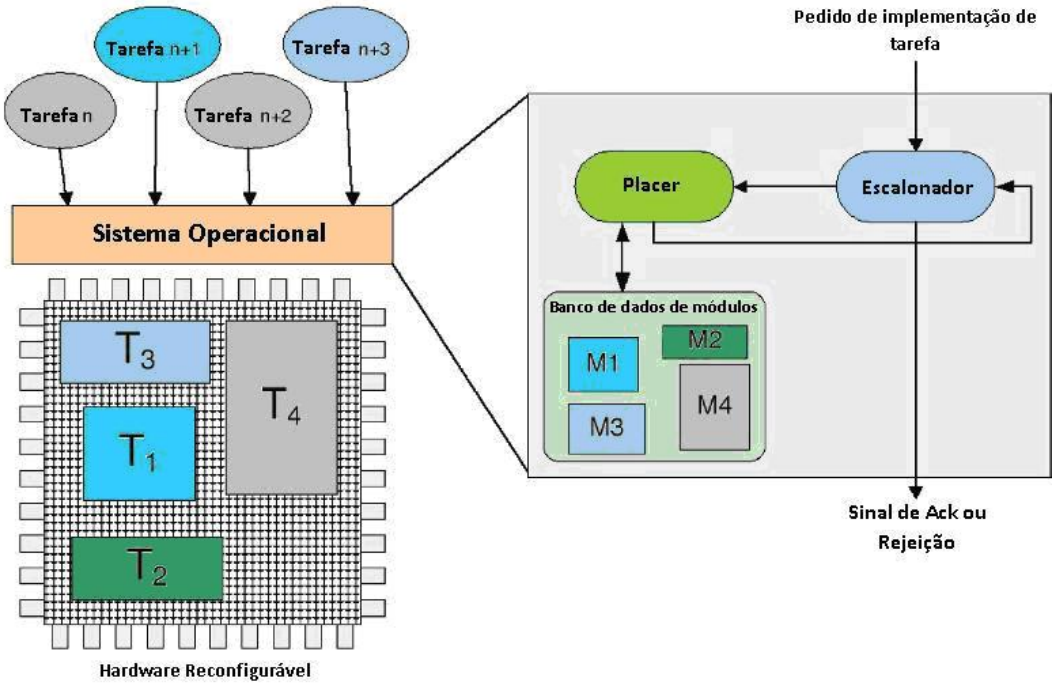


Figura 2.5: Arquitetura de um sistema em um ambiente reconfigurável em tempo de execução.

Em sistemas embutidos, que fazem uso de sistemas reconfiguráveis, os processadores são geralmente integrados no dispositivo reconfigurável e são utilizados mais para o propósito de administração do que para computação. Os *FPGAs* atuam como co-processadores com uma variedade de grupos de instruções que são acessadas pelo processador em uma chamada de função. No inicio da computação, o *host* processador configura o dispositivo reconfigurável e então envia os segmentos de dados para serem processados pelo *FPGA*. O *host* processador e o *FPGA* podem então processar em paralelo seus segmentos de dados. No final da computação, o *host* processador coleta os resultados do processamento na memória do *FPGA*.

O escalonador administra as tarefas e decide quando uma delas deveria ser executada. As tarefas estão disponíveis como dados de configuração em uma base de dados e são caracterizadas pelo tamanho da área que ela ocupará no *FPGA* e seu tempo de execução[4] (Bobda, 2007).

[3] *Placer* determina a posição que cada módulo reconfigurável deve ocupar no *FPGA*
[4] Tempo em que ele deverá substituir outro módulo reconfigurável no *FPGA*

Conforme a Figura 2.5, o escalonador determina qual tarefa deveria ser executada no *FPGA* e então, a entrega para o *placer* que irá tentar alocá-la no dispositivo. Se o *placer* não for capaz de encontrar uma localização para a execução da nova tarefa no *hardware*, ele irá devolvê-la ao escalonador que poderá decidir reenviá-la mais tarde e assim irá enviar uma nova tarefa para o *placer*. Nesse caso, diz-se que a tarefa foi rejeitada (Bobda, 2007).

Segundo (Bobda, 2007), existem três abordagens de projetos utilizadas para a reconfiguração parcial nos *FPGAs* da família *Virtex* da *Xilinx*. A abordagem *JBits*, a abordagem *Modular Design Flow* e a abordagem *Early Access Design Flow*, conhecida como *EAPR Design Flow* (Tadigotla et al., 2006).

Como mostra a Figura 2.6, a reconfiguração parcial é útil quando um projeto muito grande não precisa ocupar todo o *FPGA*, em um dado tempo ou quando apenas uma parte do projeto necessita ser modificada. Em uma configuração parcial, os *bits* de programação que necessitam ser modificados no *FPGA* funcionam como um código em uma memória *RAM*. Um endereço é utilizado para especificar a exata localização desses *bits* no dispositivo reconfigurável. Quando um sistema não necessita de todo o espaço físico de configuração de um *FPGA*, um número de diferentes configurações podem ser alocadas nas áreas não utilizadas do *hardware* em diferentes tempos. Em (Compton e Hauck, 2002) é mostrado alguns fabricantes de *FPGAs* que suportam a reconfiguração parcial e dinâmica.

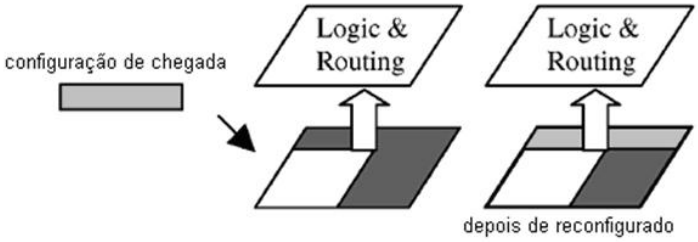


Figura 2.6: Reconfiguração Parcial.

O *Jbits* é utilizado apenas para algumas famílias *Virtex* antigas; as abordagens mais utilizadas para os *FPGAs Virtex* atualmente são *EAPR Design Flow* e a *Modular Design Flow* (Bobda, 2007).

FPGAs Virtex como outros *FPGAs* da *Xilinx* são organizados com arranjos bidimensionais de *CLBs* contendo certa quantidade de lógica. Esses *FPGAs* são configurados com dados de configu-

ração chamados de *bitstream*, os quais podem ser descarregados no dispositivo por uma porta de configuração.

A idéia por trás da reconfiguração parcial é realizar a reconfiguração apenas mudando algumas partes da configuração sendo executadas no *FPGA*. Fragmentos do *bitstream* chamados de pacotes (*packets*) são enviados para o dispositivo para reconfigurar as partes necessárias dele. Uma cópia da configuração existente no *FPGA* é mantida em uma parte dedicada da memória do processador, chamada de memória de configuração (*configuration memory*). A reconfiguração parcial é feita pela sincronização entre a memória de configuração e o dispositivo reconfigurável. As mudanças feitas entre a última configuração e a presente são marcadas e então enviadas para o dispositivo para a reconfiguração parcial.

A arquitetura dos *FPGAs Virtex* da *Xilinx* é organizada em *frames* ou colunas de *slices*. Os *frames* são pequenas unidades de reconfiguração. A Figura 2.7 mostra a arquitetura de configuração utilizada nos dispositivos *Virtex-II* e *Virtex-II Pro*. Os dados de configuração são carregados nos *frames* do *FPGA Virtex*.

As últimas gerações de *FPGAs Virtex*, tais como os da família *Virtex-4*, marcam uma significativa mudança na arquitetura, com relação às famílias anteriores. Como mostra a Figura 2.8, a arquitetura de configuração é baseada em *frames*, como nas famílias anteriores, porém, nessas famílias de *FPGAs*, as colunas são constituídas de múltiplos *frames* independentes, ou seja, um *frame* transpõe apenas 16 linhas de blocos lógicos, ao invés de uma coluna inteira (Craven, 2008). Isso significa que a reconfiguração de um componente presente em um bloco afeta apenas os componentes dele, os quais compartilham uma coluna comum com o módulo reconfigurável (Bobda, 2007).

O maior problema da reconfiguração parcial é justamente produzir o *bitstream* parcial. Há duas formas principais para fazer a extração do *bitstream* parcial, representando um dado módulo: a abordagem construtiva, representada na Figura 2.9-a, permite fazer a implementação de um dado módulo separadamente usando ferramentas comuns de desenvolvimento, tal como o *ISE* da *Xilinx* e então, juntar esse módulo no *bitstream* completo ou construí-lo como somas de módulos parciais. Essa é a abordagem seguida pelos *EAPR Design Flow* e o *Modular Design Flow*. A segunda possibilidade, representada na Figura 2.9-b consiste primeiramente implementar o *bitstream* com-

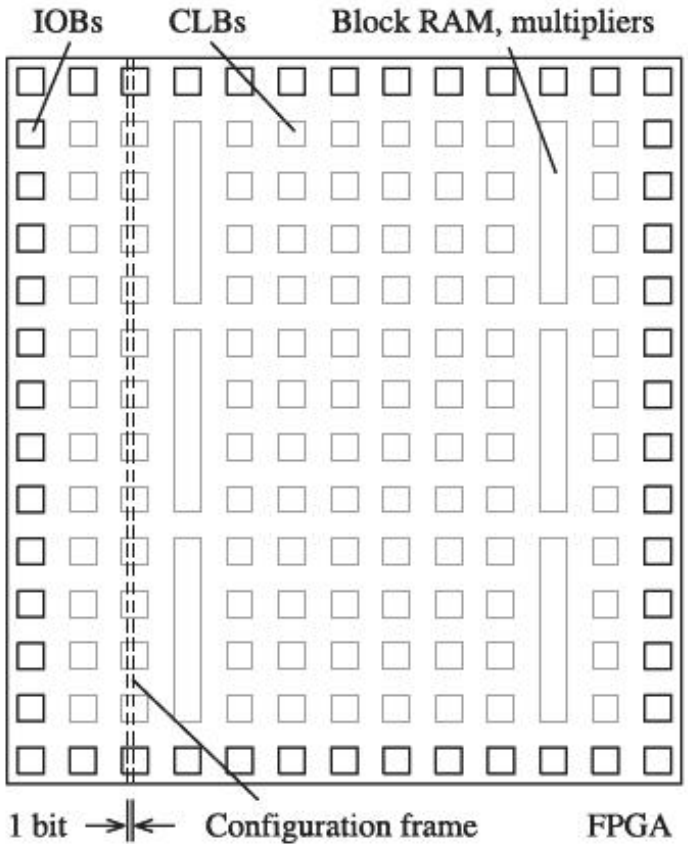


Figura 2.7: Arquitetura de Reconfiguração do Virtex-II.

pleto separadamente. As partes fixas bem como as partes reconfiguráveis são implementadas como componentes e juntadas em outro *bitstream*. As diferenças dos dois *bitstream* estão nas partes reconfiguráveis e, então, os dois *bistreams* são computadas para se obter o *bitstream* parcial (Bobda, 2007).

Segundo (Zheng et al., 2005), *Jbits* é uma *API* (*Application Programming Interface*) desenvolvida pela *Xilinx*. *Jbits* é formado por um grupo de classes *Java*, métodos e ferramentas que podem ser utilizadas para modificar o conteúdo de uma *LUT* bem como interconexões dentro do *FPGA*, sem a necessidade de outra ferramenta de configuração. Segundo (Bobda, 2007), isso é tudo o que é exigido para implementar funções em um *FPGA*.

JBits possui centenas de *cores*[5] pré-definidos, tais como somadores, subtratores, multiplicadores, entre outros. Eles também podem ser combinados para gerar *cores* mais complexos (Bobda, 2007; Lopes, 2008).

[5]Núcleos de propriedade intelectual, núcleos *IP* (*Intellectual Property*), ou simplesmente *cores*

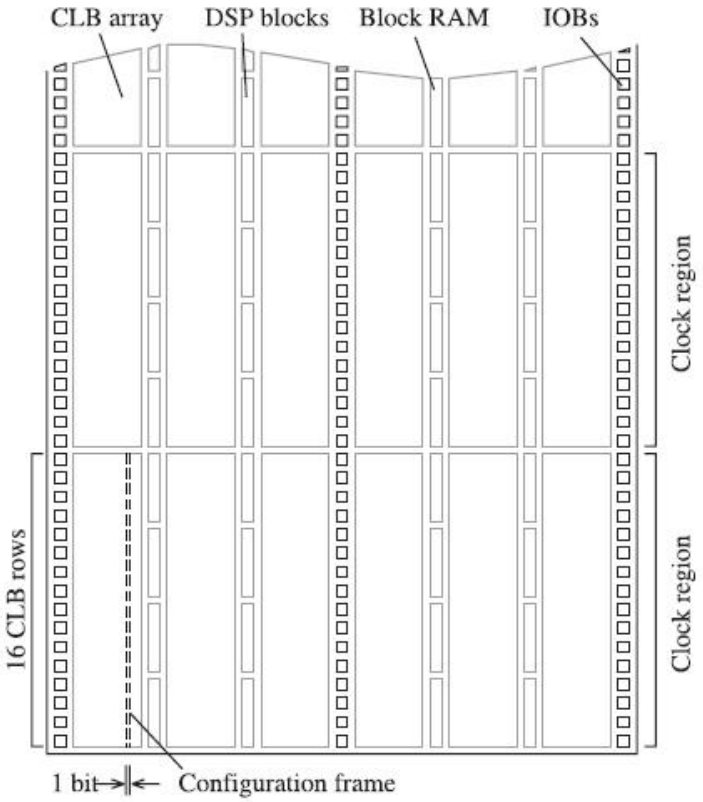


Figura 2.8: Arquitetura de Reconfiguração do Virtex-4.

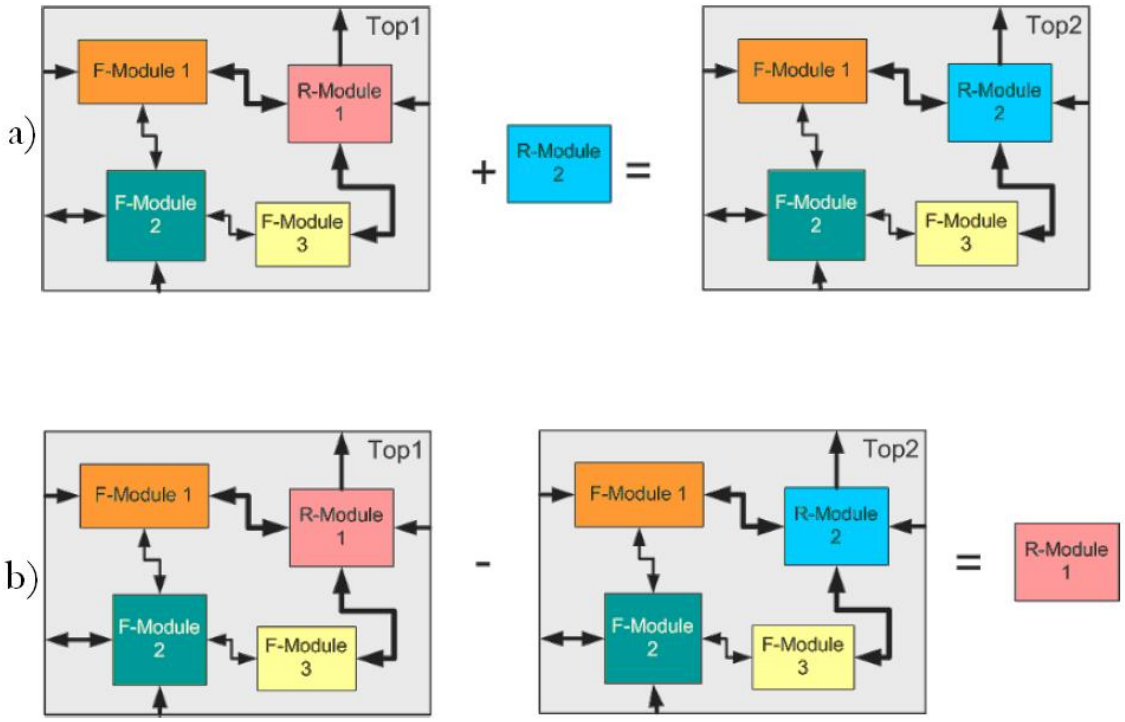


Figura 2.9: Geração do *bitstream* para reconfiguração parcial.

Embora qualquer função possa ser implementada inteiramente com o *JBits*, ele não é a melhor abordagem utilizada para reconfiguração parcial e dinâmica porque ele torna muito complexo a construção de componentes a nível de portas lógicas e interconexões. Segundo (Zheng et al., 2005) o *JBits* possui uma baixa abstração, comparando-se com a abstração utilizada na síntese lógica de alto nível, encontrada nas ferramentas de descrição de *hardware*. Além disso, o suporte para dispositivos *Virtex*, relacionado ao *JBits*, está limitado a família *Virtex-II*, até o presente momento.

No método *Difference-based flow*, a reconfiguração parcial é realizada fazendo-se pequenas mudanças em um projeto e então, é gerado um *bitstream* baseado apenas na diferença desse projeto com o original. A troca de configuração de um módulo de uma implementação para outra é muito rápido, devido às pequenas modificações feitas no projeto (Montminy et al., 2007).

Segundo (Upegui e Sanchez, 2005), nesse método o projetista deve fazer manualmente as mudanças no projeto, ao nível das interconexões internas do *FPGA*, utilizando, por exemplo, o *FPGA Editor*, uma ferramenta de edição nesse nível. Essa ferramenta vem com o *ISE* da *Xilinx*. Nessa ferramenta, o projetista pode mudar as configurações dos componentes tais como: equações nas *LUTs*, conteúdos na *RAM* interna, multiplexadores, inicialização de *flip-flops*, roteamentos, entre outros. Depois de editado as mudanças, um *bitstream* parcial é gerado, contendo apenas as diferenças entre o projeto original e as modificações. Para projetos complexos, esse método seria impraticável devido a edição no nível de interconexões no arquivo *bistream*.

Segundo (Bobda, 2007), uma das principais desvantagens de JBits é a dificuldade de prover canais de roteamento fixos para uma conexão direta entre dois módulos. Segundo (Tadigotla et al., 2006) a abordagem *Modular Design Flow* é mais dinâmica e flexível do que a abordagem usando o *Difference-base flow*. Por isso, ela é mais utilizada nos processos de reconfiguração parcial.

Isso é importante porque deve-se garantir que sinais não serão roteados por caminhos errados depois da reconfiguração. A Figura 2.10 ilustra um exemplo de roteamento em caminhos diferentes. Na reconfiguração, o projeto provavelmente não irá funcionar corretamente, porque foi criada uma situação não prevista no projeto.

Na Figura 2.10 são mostrados dois projetos: *Top1* e *Top2*. Após a reconfiguração dinâmica, o módulo *R-Module 1* em *Top1* foi substituído pelo módulo *R-Module 2*, gerando assim o projeto *Top2*. Tantos os módulos fixos quanto os reconfiguráveis foram colocados nas mesmas posições

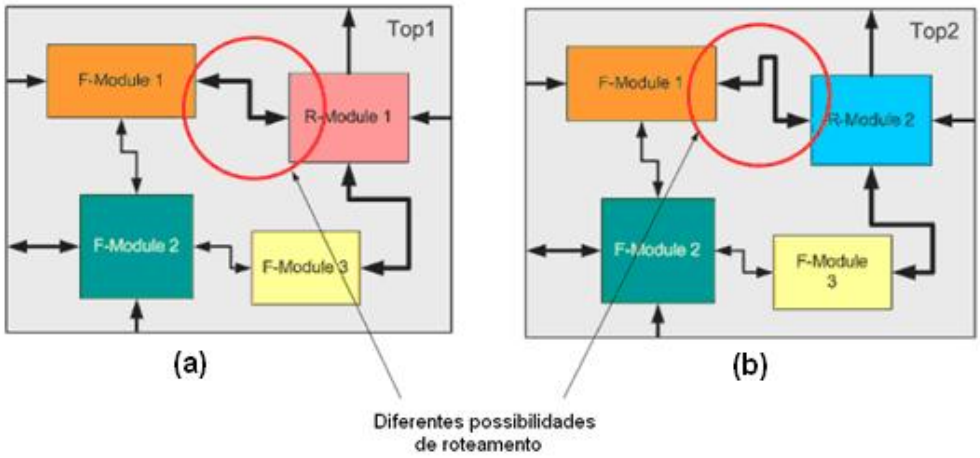


Figura 2.10: Roteamento em dois caminhos diferentes.

originais, como em *Top1*. A conexão *F-Module 1* <-> *R-Module 1* está roteada em um caminho diferente em relação a conexão *F-Module 1* <-> *R-Module 2*.

Uma das principais contribuições da abordagem modular é o uso da primitiva *Bus Macro* que garante canais de comunicação fixos entre componentes que serão reconfigurados em tempo de execução e os componentes fixos do projeto (Bobda, 2007).

Ainda segundo (Bobda, 2007), a abordagem *modular* não foi desenvolvida inicialmente para dar suporte à reconfiguração parcial. Ela foi desenvolvida para permitir a cooperação de engenheiros que trabalham no mesmo projeto, ou seja, ela permite que o projeto completo seja dividido em módulos, os quais podem ser estáticos e/ou dinâmicos. Por exemplo, em um grupo de projetistas, o responsável pelo projeto identifica todos os componentes que farão parte do projeto, estima a quantidade de recursos que será consumido por cada componente, define a localização física do componente no dispositivo e, então, distribui cada componente para um engenheiro projetar. Depois, esses componentes são integrados no projeto final e gerado um arquivo de configuração final.

Baseado em áreas especificas, módulos fixos e reconfiguráveis serão alocados em regiões no *FPGA*. Quando um módulo reconfigurável necessitar comunicar-se com um módulo estático ou dinâmico, isto deve ser feito por meio do *Bus Macro*. O *Bus Macro* mantém de forma correta as conexões entre os dois módulos durante a reconfiguração parcial. Um projeto com módulos fixos

e reconfiguráveis é mostrado na Figura 2.11, onde o *Bus Macro* fornece a comunicação necessária entre os módulos (Tadigotla et al., 2006).

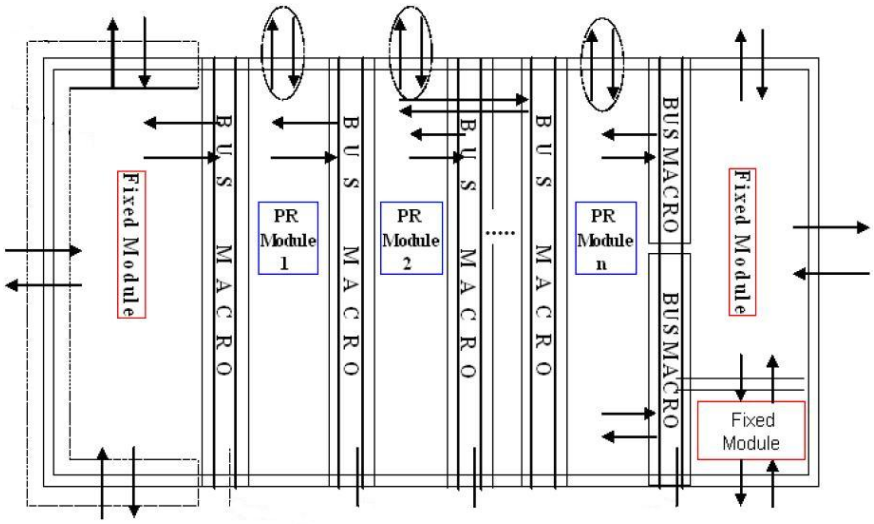


Figura 2.11: Esboço de um projeto básico com dois ou mais módulos reconfiguráveis.

Segundo (Tadigotla et al., 2006; Bobda, 2007) o *EAPR Design Flow* é um *upgrade* da abordagem projeto *modular* e as principais mudanças estão relacionadas com a usabilidade e atenuação de algumas restrições rígidas da abordagem anterior, o que reduz significativamente a complexidade de reconfiguração parcial.

Um dos principais benefícios dessa abordagem é que ela permite ao projetista realizar a reconfiguração parcial em regiões de qualquer tamanho regular, removendo assim um dos requisitos de reconfiguração parcial das abordagens anteriores (Tadigotla et al., 2006). Isto prove a possibilidade para dispositivos *Virtex-4* e superiores.

Quando uma pequena região é configurada, a coluna inteira é escrita, mas o controlador de configuração no *FPGA* verifica se a reconfiguração deveria modificar o conteúdo dos *CLBs*. Assim a execução da reconfiguração é feita apenas onde as mudanças são necessárias, conseqüentemente, a reconfiguração não irá afetar os blocos que deveriam manter-se inalterados (Bobda, 2007).

A mais importante característica do *EAPR Desing* é que ela permite a comunicação entre diferentes regiões do *FPGA*. Na abordagem *modular*, quando um sinal de um módulo estático necessitasse passar por meio de um módulo reconfigurável, era necessário fazê-lo por meio do *bus macro* e sempre quando esse módulo era configurado, os sinais de roteamento interno também eram o

que provocava quebra de comunicação entre alguns módulos estáticos no sistema. Na abordagem *EAPR*, os sinais que apenas atravessam módulos reconfiguráveis, sem interagir com eles, não terão que passar mais por meio do *bus macro*. O algoritmo de roteamento é capaz agora de determinar esses sinais e usar longos sinais de comunicação que não serão afetados pela reconfiguração parcial (Bobda, 2007), o que possibilita aumento do desempenho de tempo e simplifica o processo de construção de um projeto parcialmente reconfigurável.

CAPÍTULO

3

Máquinas a Fluxo de Dados

Arquiteturas de alto desempenho em geral estão centradas em conceitos de processamento paralelo. Segundo (Silva, 1992), máquinas a fluxo de dados foi uma dessas frentes de pesquisa, devido a natureza do paralelismo encontrado nesses sistemas, uma vez que o processo de execução de programas nestas maquinas é determinado pela disponibilidade dos dados, contrário às estruturas tradicionais onde o fluxo de dados é determinado por um controle explicito. Considerando que diversas operações poderão estar processando em paralelo, dependendo da aplicação, teremos um paralelismo natural conforme os dados vão sendo gerados e processados (Dennis e Misunas, 1974; Veen, 1986).

Segundo (Arvind e Culler, 1986), o modelo de computação a fluxo de dados oferece um simples, porém, poderoso formalismo para descrever computação paralela. Existiram um número considerável de arquiteturas a fluxo de dados diferentes.

Segundo (Veen, 1986), máquinas a fluxo de dados são computadores programáveis de granularidade fina, nos quais o *hardware* é otimizado para computação paralela dirigida pelos dados.

Em 1969, Dennis desenvolveu o esquema do modelo a fluxo de dados. Eles chamaram esse esquema de grafo a fluxo de dados. O primeiro projeto de uma máquina a fluxo de dados é encontrado em (Dennis e Misunas, 1974).

Segundo (Veen, 1986), não há uma definição exata para se distinguir máquinas a fluxo de dados de todos os outros computadores. Considera-se que máquinas a fluxo de dados são todos os computadores programáveis dos quais o *hardware* é otimizado para computação paralela de granularidade fina e computação dirigida por dados.

Em arquiteturas a fluxo de dados não existe o conceito de memória como nas arquiteturas *von Neumann*. O conceito de memória em arquiteturas *von Neumann* torna esse modelo seqüêncial, pois a passagem de dados entre as instruções é feita através de atualizações em áreas de memória e um ponteiro de instruções é responsável por estabelecer o fluxo de controle de uma instrução para outra. Devido a esse fato, o modelo *von Neumann* também é conhecido como modelo de fluxo de controle ou *control flow* (Treleaven et al., 1982).

Conforme já mostrado, nas máquinas a fluxo de dados não existe o conceito de armazenamento de dados em memória, eles simplesmente são produzidos por uma instrução e consumido por outra (Veen, 1986). A chegada dos dados serve como sinal para habilitar a execução de uma instrução, eliminando a necessidade de controle de fluxo.

Em máquinas *datafow* cada instrução é considerada como um processo separado (Veen, 1986). Para facilitar a execução, cada instrução que produz dados como resultado contém ponteiros para todos os seus consumidores. Visto que uma instrução em um programa a fluxo de dados somente contém referências para outras instruções, ele pode ser visto como um grafo (Veen, 1986).

Essa notação de grafo refere-se como um grafo a fluxo de dados dirigidos, no qual cada nó representa uma operação (instrução) e os arcos que interligam esses nós representam a dependência entre as operações (Dennis e Misunas, 1974; Veen, 1986).

Nas arquiteturas a fluxo de dados, as instruções são conhecidas como nós e os dados como *tokens*. Um nó produtor é conectado a um nó consumidor por um arco e o "ponto"no qual o arco entra no nó é chamado porta de entrada. A execução de uma instrução é chamada de disparo do nó. O disparo de um nó apenas pode acontecer se o nó estiver habilitado, o que acontece por sua vez por meio de uma regra de habilitação, que geralmente é a disponibilidade de todos os tokens

nas entradas de um nó (Veen, 1986). O termino de execução de uma operação liberam valores ou decisões para os nós do grafo, cuja execução depende deles.

Um exemplo de um programa na linguagem a fluxo de dados pode ser visto na Figura 3.1.

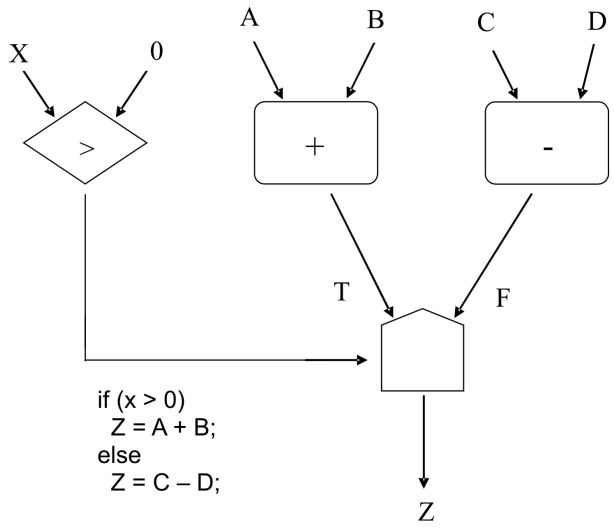


Figura 3.1: Programa básico na linguagem a fluxo de dados.

A Figura 3.1 ilustra um grafo a fluxo de dados, composto por quatro nós. Quando há tokens nas entradas dos nós eles podem ser processados e geram um token de saída. Os nós habilitados podem ser disparados em tempos não especificados, em qualquer ordem ou concorrência. O disparo envolve a remoção dos tokens de entrada e a computação do resultado.

Para representar programas na linguagem a fluxo de dados os enlaces e operadores são descritos nas Figuras 3.2 e 3.3 (Dennis e Misunas, 1974).

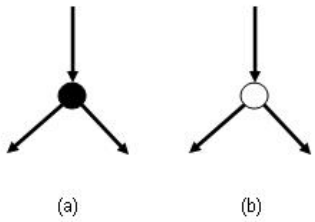
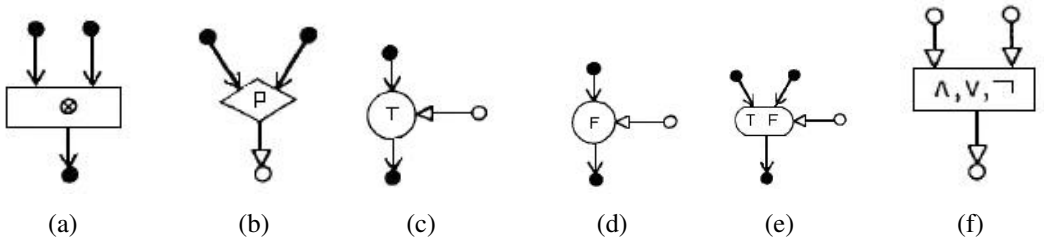


Figura 3.2: Links para a linguagem a fluxo de dados. (a) Link de dados. (b) Link de controle.

O Link de Dados é responsável em transmitir números inteiros, reais ou complexos e o Link de Controle de transmitir valores *booleanos* (*true* ou *false*). Os dados de controle são gerados por um operador *decider*, que ao receber dados de seus arcos de entrada, é aplicado um "predicado"que gera um dado de controle com um valor *true* ou *false*.



(a) (b) (c) (d) (e) (f)

Figura 3.3: Operadores para a linguagem a fluxo de dados. (a) operator (b) decider (c) T-gate (d) F-gate (e) merge (f) boolean operator

Para representar computação interativa ou condicional em grafos a fluxo de dados, são utilizados os operadores *decider*, *T-gate*, *F-gate* e *merge*. O operador *decider* é utitilzado para a aplicação de uma operação lógica sobre os seus arcos de entrada e gera um valor resultante contendo *true* ou *false*. O *operator* do grafo a fluxo de dados requer também um dado para cada arco de entrada e produz um dado como resultado da aplicação de uma operação aritmética ou lógica sobre os dados recebidos.

Dados de controle representam valores booleanos (*true* e *false*) que controlam o fluxo de dados através dos operadores *T-gate*, *F-gate* e *merge*. O operador *T-gate* transmite um dado de seu arco de entrada para o seu arco de saída somente quando ele recebe um valor *true* no seu arco de entrada, caso o valor recebido em seu arco de entrada seja *false*, ele nao transmitirá o dado para o arco de saída. O operador *F-gate* possui um comportamento similar, exceto que o valor do controle é reverso. Um operador *merge* possui arcos de entrada de dados *true* e *false* e um arco de controle. Quando um dado *true* é recebido no arco de controle, o operador *merge* posiciona o dado do arco de entrada T no seu arco de saída e o dado no arco de entrada não usado é descartado. O mesmo acontece se o dado *false* for recebido no arco de controle, o operador *merge* inverte o processo e o dado de entrada presente na entrada F é passado para o arco de saída.

Uma situação que pode ocorrer nos grafos a fluxo de dados é a reentrância, ou seja, um nó pode disparar repetidamente. O modo que a reentrância é manipulada define uma classificação das maquinas a fluxo de dados (Veen, 1986; Arvind e Culler, 1986).

Há quatro maneiras distintas de se tratar o problema da reentrância em um sub-grafo: *lock*, *acknowledge*, *code copying*, e *tagged tokens*. Máquinas a fluxo de dados que manipulam a reentrância por *lock* ou *acknowledge* são chamadas de estáticas e aquelas que manipulam a reentrância por *code copying* ou *tagged tokens* são chamadas de dinâmicas (Veen, 1986).

Um grafo a fluxo de dados é uma linguagem atrativa para uma maquina paralela, desde que todos os nós que não tem dependência de dados podem disparar concorrentemente (Veen, 1986).

Há duas implementações de arquitetura do modelo abstrato a fluxo de dados: a arquitetura estática e a dinâmica.

No modelo estático (Dennis e Misunas, 1974; Davis, 1978), permite apenas um *token* por arco no grafo a fluxo de dados. Um modelo de sincronismo para garantir a presença de um único dado no arco pode ser implementado por um protocolo específico que implementa a comunicação entre os operadores. Conseqüentemente o paralelismo neste caso é limitado pela seqüência de dados que vão sendo colocados nos diversos arcos, além do paralelismo existente entre os operadores.

As máquinas dinâmicas (Shimada et al., 1986; Gurd et al., 1985; Kishi et al., 1983; Grafe et al., 1989; Papadopoulos e Culler, 1990) são uma evolução das máquinas estáticas, por nelas permitirem que mais de um *token* esteja presente em um arco, o que significa utilizar toda a potencialidade da programação a fluxo de dados, exigindo, no entanto um tratamento mais complexo na implementação do sistema, pois é preciso além de armazenar os dados, identificar os parceiros dos dados que disparam uma operação.

Um alto nível de paralelismo é obtido quando cada iteração do grafo a fluxo de dados é executada em uma instância (ou cópia) separada de um sub-grafo reentrante. Para que isso aconteça é necessário existir na máquina algumas facilidades para criar uma nova instância de um sub-grafo e direcionar os *tokens* para as novas instâncias. Um modo eficiente de se implementar essas facilidades seria compartilhar a descrição dos nós entre diferentes instâncias de um grafo, sem que haja confusão nos tokens que pertencem a instâncias separadas. Isso é feito adicionando-se um rótulo (*tag*) para cada *token*, para identificar a instância do nó que ele será direcionado. Estas máquinas são chamadas de arquitetura *tagged-token* e possuem uma regra de habilitação que determina que um nó está habilitado se cada arco de entrada do nó possuir um *token* com um rótulo idêntico. Nessa máquina, não há mais do que um *token* com o mesmo rótulo.

3.1 Arquitetura das máquinas a Fluxo de Dados

Uma máquina a fluxo de dados típica consiste de um número de elementos de processamento, os quais podem comunicar-se uns com os outros. Os nós de um programa a fluxo de dados são freqüentemente armazenados na forma de um gabarito contendo a descrição do nó e espaço para os *tokens* de entrada (Veen, 1986). A Figura 3.4 mostra um gabarito correspondendo ao operador "soma"(Silva, 1992).

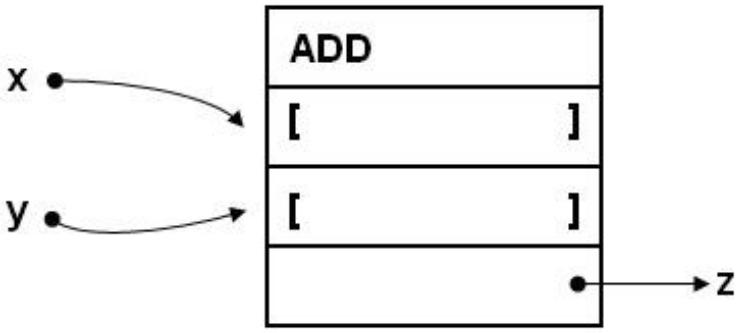


Figura 3.4: Gabarito da operação soma.

Existem quatro campos no gabarito da Figura 3.4: um para o código de operação especificando a operação a ser executada; dois receptores, que esperam para ser preenchidos com valores de operandos; e campo destino, que especifica o que fazer com o resultado da operação sobre os operandos.

Pode-se pensar na movimentação de um *token* entre dois nós como uma atividade local de um processo. Nas máquinas a fluxo de dados a coordenação dos *tokens* consumidos nos nós resume-se a administração da regra de habilitação desses nós que requerem mais que uma entrada (Veen, 1986).

A Figura 3.5 ilustra um diagrama funcional de um elemento de processamento. Alguns autores chamam a unidade que administra o armazenamento dos *tokens* de unidade de habilitação (*enabling unit*). Ela seqüencialmente aceita um *token* e o armazena no endereço do nó na memória. Se isso fizer o nó, o qual o *token* está endereçado, se tornar habilitado (ex: cada porta de entrada contém um *token*), esses *tokens* de entrada são extraídos da memória e juntos com uma cópia do nó, formam um pacote executável e são enviados a unidade funcional (*functional unit*), onde eles

são processados. Esse pacote consiste de valores dos *tokens* de entrada, código do operando e uma dista de destino. A unidade funcional processa os valores de saída e os combinam com os endereços de destino dentro dos *tokens*. Os *tokens* são enviados de volta a unidade de habilitação, onde eles podem habilitar outros nós. Desde que a unidade de habilitação e a unidade funcional trabalhem concorrentemente, elas são freqüentemente referenciadas como *pipeline* circular. Os módulos dedicados a *buffering* ou comunicação foram deixados de fora do diagrama exposto na Figura 3.5 (Veen, 1986).

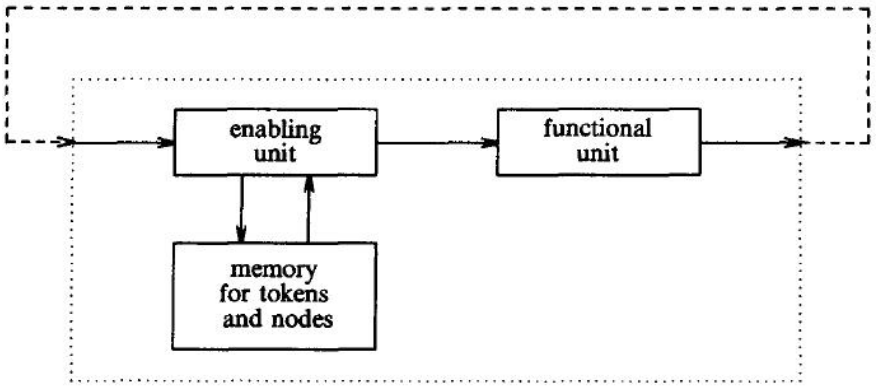


Figura 3.5: Diagrama funcional do elemento de processamento.

Em algumas máquinas a fluxo de dados, o elemento de processamento não tem que ser tão potente e ele consiste de uma memória conectada a uma unidade que liga o armazenamento de *tokens* e a execução dos nós (Veen, 1986).

Como discutido no início desse capítulo, as máquinas a fluxo de dados podem diferenciar nas suas arquiteturas. Segundo (Veen, 1986), em outras máquinas, o *pipeline* circular consiste de mais unidades concorrentes.

Em algumas máquinas a unidade de habilitação é separada em duas unidades: a unidade de *matching store* e a unidade de *fetching*, usualmente organizadas como mostra a Figura 3.6 (Veen, 1986).

A *matching unit* armazena *tokens* em suas memórias e verifica se uma instância do nó destino está habilitada. Esta operação requer uma compatibilidade de endereço de destino e rótulo. Os *tokens* são armazenados na memória conectada a *matching unit*. Para cada *token* que a *matching unit* aceita, ela tem que verificar se o endereço do nó está habilitado. Na maioria das máquinas *tagged-token* isto é facilitado limitando-se o número de arcos de entrada para dois e disponibi-

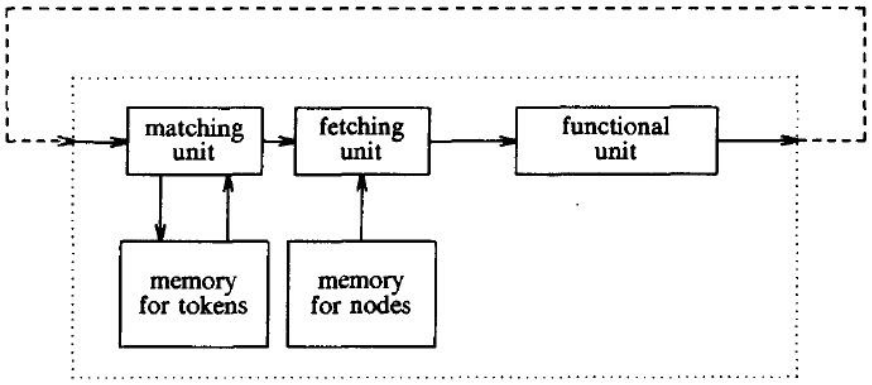


Figura 3.6: Diagrama funcional do elemento de processamento.

lizando para cada *token* um *bit* extra que indica se o nó endereçado possui uma entrada ou duas. Apenas para nós com duas entradas, o *matching unit* tem que verificar se a memória já contém um *token* parceiro, que é um *token* com o mesmo destino e um rótulo (*tag*), que indicam suas posições locais nos arcos (Arvind e Culler, 1986). Quando todos os *tokens* para uma instância de um nó em particular chegam, eles são enviados a *fetching unit*, a qual os combina com uma cópia da descrição do nó dentro de um pacote executável a ser enviado para a unidade funcional.

Conceitualmente, segundo (Veen, 1986), uma *matching unit* simplesmente combina endereço e rótulo em um endereço e verifica se o local denotado pelo endereço contém um *token*.

O grupo de localizações endereçáveis pelo rótulo e o endereço de destino formam um espaço chamado de *matching space*. Administrar esse espaço e representá-lo em uma memória física é um dos problemas chaves na arquitetura a fluxo de dados *tagged-token* (Veen, 1986).

3.2 Descrição de algumas arquiteturas existentes

A seguir são descritas arquiteturas a fluxo de dados tradicionais e em seguida são descritas com maiores detalhes aquiteturas a fluxo de dados contemporâneas.

3.2.1 Arquiteturas a fluxo de dados tradicionais

Como citado anteriormente, (Dennis e Misunas, 1974) foram os pioneiros nas pesquisas de computadores a fluxo de dados. A idéia era a construção de um sistema de computador altamente

paralelo, que executasse simultaneamente vários fragmentos de um programa, porém esse alto grau de atividades concorrentes deveria ser encontrado sem nenhum sacrifício.

Com as características acima, um grande números de projetos de arquiteturas a fluxo de dados foram desenvolvidos e construídos ao longo dos anos. Um desses projetos de máquina a fluxo de dados começou com o *Massachusetts Institute of Technology - MIT* (Arvind e Kathail, 1981).

Na máquina a fluxo de dados do *MIT* somente um *token* pode ocupar um arco a cada instante. Isto implica que a regra de disparo é tal que uma instrução é habilitada se um *token* está presente sobre cada um de seus arcos de entrada e nenhum *token* está presente sobre qualquer um de seus arcos de saída. Na organização de programas da *MIT* existem *tokens* de controle e de dados, que contribuem para a habilitação de uma instrução. *Tokens* de controle atuam como sinal de reconhecimento quando os *tokens* de dados são removidos dos arcos de saída. Sua arquitetura consiste de cinco partes conectadas por canais através dos quais são enviados *tokens* usando um protocolo de comunição assíncrono, sendo uma Seção de Memória que contém instruções e seus operandos; uma Seção de Processamento que realiza operações sobre os *tokens* de dados; uma Rede de Arbitragem que distribui as instruções da seção de memória para a seção de processamento; uma Rede de Controle que distribui *tokens* de controle da seção de processamento para a seção de memória e por último uma Rede de Distribuição que distribui *tokens* de dados da seção de processamento para a seção de memória.

Um outro projeto de uma máquina a fluxo de dados dinâmico foi desenvolvido na Universidade de *Manchester* denominada *MMDM* (*Manchester Multi-Ring Dataflow Machine*) (Gurd et al., 1985). Ela consiste de cinco unidades principais:

- Comutador: responsável em providenciar as entradas e saídas para o sistema;
- Pilha de Token de dados: responsável em realizar o armazenamento temporário de um token de dados;
- Unidade de Disparo: responsável em igualar par de token de dados;
- Memória de Instruções: memória que mantém os programas a fluxo de dados;
- Unidade de Processamento: consite de um número de Elementos de Processamento idênticos, que executam as instruções.

O comutador é usado para passar *tokens* de dados para dentro ou fora do sistema. Para iniciar a execução de um programa, os tokens de iniciação são inseridos através de comutadores e dirigidos por suas *tags* para as instruções de início da computação. No final do programa são inseridos intruções especiais que geram pacotes de dados de saída.

Em (Sato et al., 1992) é apresentada uma máquina a fluxo de dados híbrida denominada *EM-4*, onde sua arquitetura *multithread*[1] pode explorar tanto programas *von Neumann* como programas a fluxo de dados.

Do ponto de vista de uma execução sequencial de *threads*, *EM-4* pode ser visto como multiprocessador com memória distribuída. Cada elemento de processamento possui sua própria memória local e as *threads* dentro de um elemento de processamento operam somente sobre a memória local.

No modelo a fluxo de dados utilizado na *EM-4*, os grafos a fluxo de dados são classificados dentro de duas categorias: arcos normais e arcos fortemente conectados. Um conjunto de nós interligados por arcos fortemente conectados é chamado de *SCB* (*strongly connected block*). Arcos fortemente conectados possuem regras de disparo dentro de um *SCB*.

Um *SCB* é implementado como uma sequencia de instruções e executado dentro de um elemento de processamento, controlado por um contador de programa. Do ponto de vista do modelo a fluxo de dados *SCB* são armazenados em registradores em vez de serem transferidos como *tokens*, reduzindo o custo de comunicação.

A unidade de toda a comunicação no *EM-4* é o pacote. O *token* de uma computação a fluxo de dados é entregue como um pacote. Uma parte do endereço do pacote especifica o *SCB* que será invocado para aquele pacote.

Cada instrução possui um *bit* para indicar se a execução de um *SCB* deve continuar ou não. Este *bit* corresponde aos arcos fortemente conectados. Uma vez que um *SCB* é executado, sua execução continua até que uma instrução possua este *bit* configurado para encerrar a execução do *SCB*.

Um outro projeto de uma máquina a fluxo de dados é o *DDM1* (Davis, 1978), implementada na Universidade de *Utah*. Tanto o programa como a arquitetura da máquina são organizados

[1] Em um ambiente de múltiplos threads

utilizando os conceitos de recursão. O computador é composto de elementos computacionais estruturados de forma hierárquica (pares processador-memória) onde cada elemento é logicamente recursivo. Fisicamente a arquitetura do computador é estruturada em árvore, onde cada elemento computacional é conectado a um elemento superior e a oito elementos inferiores.

Na máquina a fluxo de dados de *Utah* não existem *tokens* de controle, os *tokens* de dados providenciam toda comunicação entre as instruções. Nesta organização, os arcos do grafo a fluxo de dados, são vistos como uma fila *FIFO* (*First-in First-out*), onde os *tokens* de dados são armazenados à medida que vão chegando mas não são imediatamente consumidos pelas instruções. Para isso cada instrução consiste de um código de operação e uma lista de endereços distintos para os resultados, junto com um número variável de conjuntos de *tokens* de dados esperando para ser consumidos pela instrução.

Na Universidade de *Newcastle* foi desenvolvido um projeto de uma máquina a fluxo de dados denominada *Jumbo* (Treleaven et al., 1982). O objetivo era estudar a integração entre a computação a fluxo de dados e a fluxo de controle. Ela consiste em três componentes principais interconectadas por *buffers FIFO*. O primeiro componente é a Unidade de Disparo, que controla a habilitação de instruções igualando um conjunto de *tokens* de dados, que são enviados para a Unidade de Memória quando executadas; o segundo componente é a Unidade de Memória responsável em armazenar dados e instruções; o último componente é a Unidade de Processamento, que oferece suporte para execução de instruções e a distribuição de resultados.

3.2.2 Arquiteturas a fluxo de dados contemporâneas

Um modelo a fluxo de dados foi apresentado por (Budiu e Goldstein, 2002) denominado *ASH* (*Application-Specific Hardware*). *ASH* é uma estrutura arquitetural para implementar aplicação específica em hardware. *ASH* é baseado na síntese automática de linguagens de alto nível.

O *ASH* utiliza o compilador *CASH*, um compilador escalonado, que gera *hardware* a partir de programas escritos em C. O compilador explora paralelismo ao nível de instruções usando técnicas como *aggressive speculation* e *dynamic scheduling*.

Os circuitos que o *ASH* gera podem ser usados de forma autônoma, com a implementação da aplicação inteira nele ou com um processador de propósito geral (Budiu et al., 2005).

O principal componente do sistema *ASH* é o compilador CASH.

A estrutura do *ASH* é ilustrada na Figura 3.7. Na Figura 3.7(a) é descrito o fluxo de execução da estrutura, onde programas escritos em linguagens de alto nível são as entradas para o compilador *CASH*, que depois são convertidos para *hardware*. Na Figura 3.7(b) são descritos os três tipos de objetos que o compilador constrói para cada procedimento do programa (circuito, memória local e rede de interconexões) (Budiu e Goldstein, 2002).

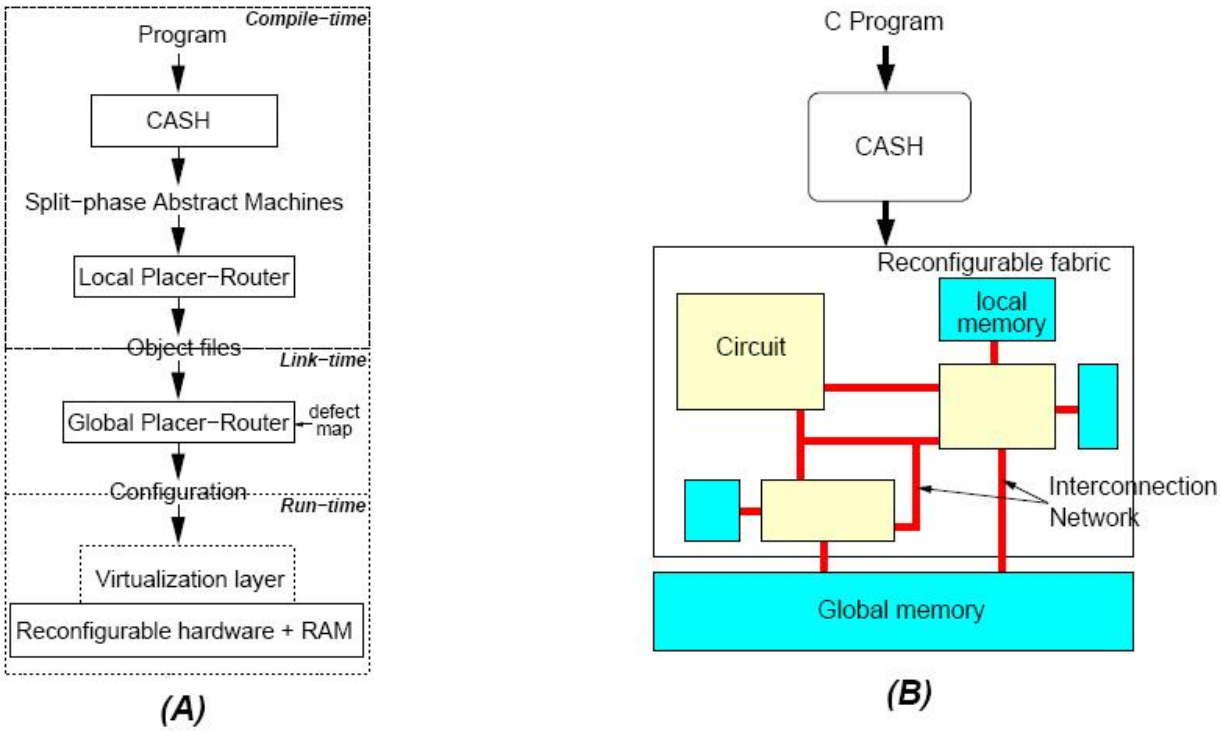


Figura 3.7: (A) Fluxo da ferramenta *ASH* (B) Tradução do programa em *hardware*.

A partir do compilador *CASH* na Figura 3.7(a), para cada procedimento do programa o compilador cria três diferentes tipos de objetos: estruturas de computação, interconexões e memória local. O resultado é uma arquitetura para hardware reconfigurável, Figura 3.7(b). Cada procedimento é independentemente otimizado, sintetizado, alocado e roteado. Os procedimento comunicam-se uns com os outros de forma assíncrona. Todos os sinais de um procedimento tem latência previsível, incluindo o acesso a memória local. Todavia, os procedimentos podem invocar operações remotas, as quais têm latência imprevisível.

Sempre que um procedimento necessita executar uma operação que tem latência imprevisível, ele usa a rede de interconexões: acesso de memória remoto e transferências de controle de fluxo

são conceitualmente transformados em mensagens, as quais podem ser roteadas dinamicamente em uma rede (Budiu e Goldstein, 2002).

O compilador *CASH* divide cada procedimento em uma coleção de *hyperblocks*, transformando cada *hyperblock* em linhas de código direta (sem laços), usando uma técnica em compiladores chamada *speculation*[2], em seguida traduz cada *hyperblocks* em circuitos a fluxo de dados.

Uma vez que um *hyperblock* começa a executar, cada uma de suas operações são executadas exatamente uma vez.

O circuito produzido pelo sistema *ASH* funciona da mesma forma que um sistema produtor/-consumidor. Uma vez que o dado é consumido ele não estará mais disponível.

Em geral um operador é rígido, pois não computa a menos que todos os dados estejam presentes em suas entradas.

Segundo (Budiu e Goldstein, 2002), circuitos a fluxo de dados podem ser facilmente utilizados para expressar códigos diretos. Para permitir a implementação de construções a fluxo de controle (ramificações e chamadas a procedimentos), *ASH* aumenta o grupo de operações a fluxo de dados com duas construções especiais: nós *merge* e *eta*.

Esses nós são usados entre os *hyperblocks* e são suficientes para sintetizar circuitos correspondentes a um fluxo de controle arbitrário, incluindo grafos irregulares. Os nós *merge* são denotados por um triangulo que aponta para cima, enquanto *eta* é denotado por um triangulo que aponta para baixo.

O operador *eta* possui duas entradas, uma para o dado e outra para um predicado e uma saída de dados. Se o predicado é verdadeiro *true* o dado da entrada é copiado para a saída, porém, se o predicado for falso, o dado de entrada é apenas consumido e nenhuma saída é gerada. O operador *merge* possui "n"entradas e uma saída. Para que ele possa executar não é necessário que todas as suas entradas estejam disponíveis, ele simplesmente copia uma entrada disponível para a saída. O operador *merge* recebe dados de múltiplos operadores, mas somente um deles pode ativá-lo.

[2]Existem dois tipos de especulação: a de dados e a de controle. Com a especulação, o compilador antecipa uma operação de forma que sua latência, ou seja, tempo gasto seja retirada do caminho crítico. A especulação é uma forma de permitir ao compilador evitar que operações lentas atrapalhem o paralelismo das instruções. A especulação de controle é a execução de uma operação antes do desvio que a precede. Por sua vez, a especulação de dados é a execução de uma carga da memória (*load*) antes de uma operação de armazenagem (*store*) que a precede e com a qual pode estar relacionada.

A maior abstração utilizada ao nível de programa é o *hyperblock*. Um *hyperblock* é parte de um grafo de fluxo de controle *CFG*, com um único ponto de entrada, mas múltiplas saídas possíveis.

Na Figura 3.8 é visto que compilador gera três *hyperblocks* para a seqüência de *Fibonacci*, marcados do lado direito da figura por linhas pontilhadas. O conjunto de hyperblocks são convertidos e executados diretamente no *hardware*.

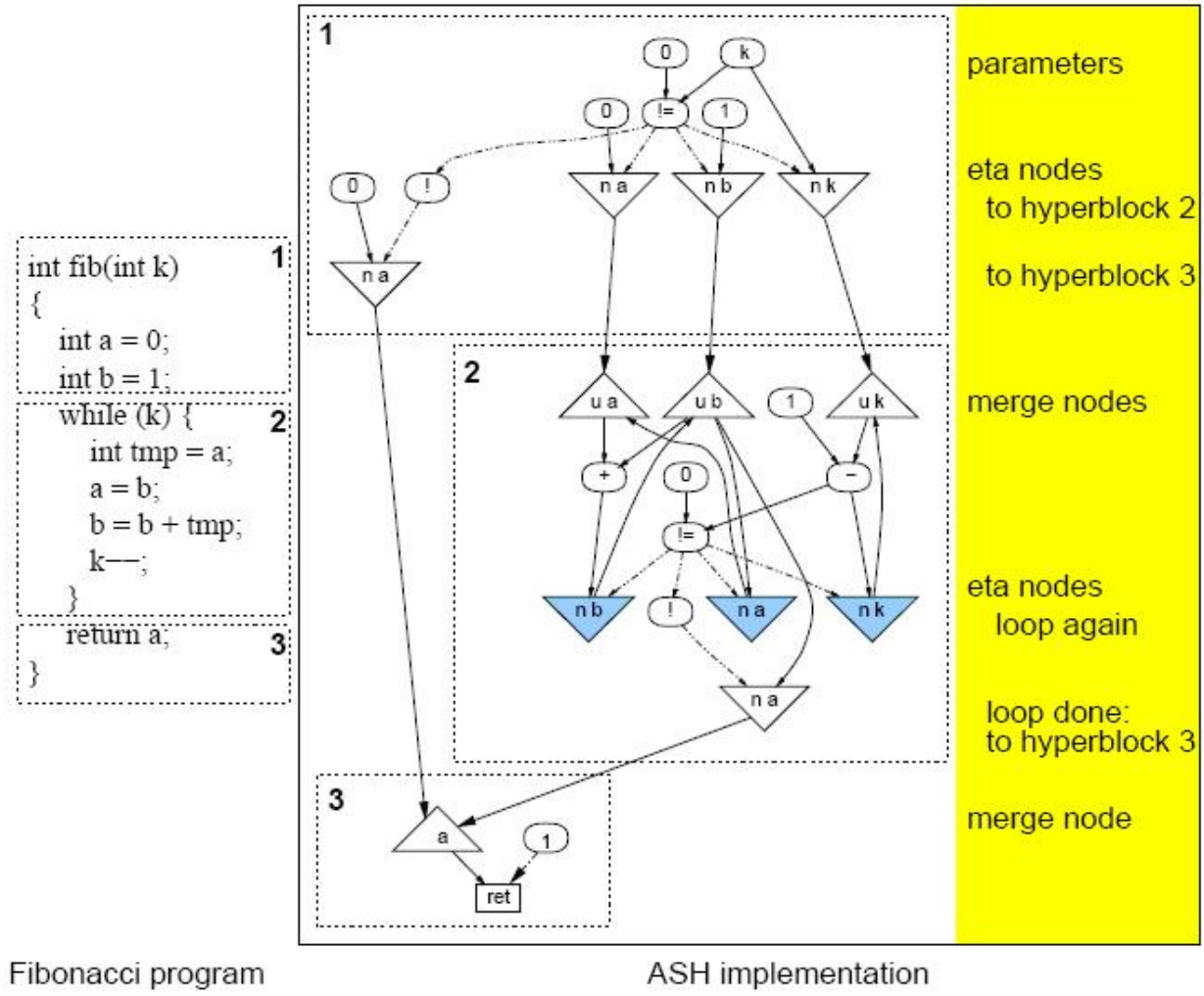


Figura 3.8: Programa *Fibonacci* e sua implementação em *ASH*.

Como mostra a Figura 3.8, há nós *merge* nos *hyperblocks* 2 e 3. Os nós *merge* no *hyperblock* 2 aceitam dados do *hyperblock* 1 e do próprio *hyperblock* 2, controlando o laço *while*. Os nós *merge* no hyperblock 3 podem aceitar dados de controle dos *hyperblock* 1 ou do *hyperblock* 2.

Os circuitos gerados para os *hyperblocks* são ligados juntos usando os nós *merge* e *eta*. Para cada variável ativa na entrada de um *hyperblock* é criado um operador *merge*; para cada variável

ativa na saída do *hyperblock* é criado um operador *eta*. A saída do operador *eta* é conectado a entrada do operador *merge* correspondente no *hyperblock* que sucede (Budiu et al., 2005).

A principal desvantagem do *ASH* é a exigência para recursos de *hardware*. Porém, isso pode ser suavizado pelo particionamento *hardware/software* (Budiu e Goldstein, 2002).

O circuito gerado pelo *ASH* está relacionado a máquinas a fluxo de dados, mas são destinados a ser implementado diretamente em *hardware* e não interpretado nas máquinas a fluxo de dados usando passagem de *tokens* (Budiu e Goldstein, 2002).

Em (Swanson et al., 2003, 2007) é descrito a arquitetura a fluxo de dados, chamada de arquitetura *WaveScalar*. Essa arquitetura possui um grupo de instruções a fluxo de dados e modelo de execução projetado para ser escalonado, ter baixa complexidade e alto desempenho de processamento.

Diferente das máquinas a fluxo de dados existentes, *WaveScalar* pode fornecer de forma eficiente as semânticas de memória seqüencial que as linguagens imperativas exigem. Para permitir aos programadores expressar facilmente o paralelismo, *WaveScalar* suporta estilos de programação *pthread*[3], *multithread* e a fluxo de dados. No *WaveScalar*, como em toda arquitetura a fluxo de dados, o programa é representado para o processador como grafos a fluxo de dados. Cada nó no grafo é uma instrução e os arcos entre os nós codificam estaticamente a dependência de dados entre instruções.

Além disso, *WaveScalar* permite combinar os dois estilos de programação em uma aplicação ou até em uma simples função (Swanson et al., 2007).

Para executar programas *WaveScalar*, foi projetado uma arquitetura de processador baseada em *tile*[4], chamada *WaveCache*. Esse processador possui uma arquitetura escalonada e descentralizada. A medida que um programa executa, o *WaveCache* organiza as instruções de programa sobre seus arranjos de elementos de processamentos *PEs* (*Processor Elements*).

As instruções continuam em seus elementos de processamento para muitas invocações futuras e conforme ocorre o processamento de um programa o grupo de instruções muda. Então

[3]Interface de manipulação de threads padronizada em 1995 pelo IEEE (IEEE POSIX 1003.1c)

[4]Na indústria de processadores, muitos projetistas estão abandonando o conceito de processadores monolíticos, complexos e de alto desempenho em favor de elementos de processamento (PE), mais simples, replicados dentro do *chip*. Estas arquiteturas são denominadas *tiled architectures*.

o *WaveCache* remove as instruções ociosas e organiza novas instruções em seus lugares. As instruções comunicam-se diretamente com as outras por meio de uma rede hierárquica de interconexão escalonada *on-chip*, evitando assim longos fios.

O projeto *WaveScalar* foi construído no modelo de execução a fluxo de dados e explora as propriedades desse modelo, tais como descentralização e escalonamento (Swanson et al., 2007).

Além disso, como já mostrado nos capítulos anteriores, o modelo de execução a fluxo de dados permite aos programadores e compiladores expressar explicitamente o paralelismo, em vez de depender de um *hardware* para extraí-lo (Swanson et al., 2007).

Os sistemas a fluxo de dados antigos não podiam executar de forma eficiente a semântica seqüencial de memória que as linguagens imperativas, tais como *C*, *C++* e *Java* exigem. A execução nessa ordem é necessária para preservar as dependências entre as instruções de leitura e escrita, por exemplo. Segundo (Swanson et al., 2007), linguagens imperativas modernas basicamente fornecem semântica de memória idêntica, sendo que os sistemas a fluxo de dados antigos usavam linguagens a fluxo de dados especiais que limitavam a sua utilidade (Swanson et al., 2007).

WaveScalar explora as propriedades do modelo a fluxo de dados e supera essas limitações por meio de um esquema de ordenação de memória, chamado de *wave-ordered memory*, necessário para linguagens imperativas. Usando esse esquema de ordenação de memória, *WaveScalar* suporta estilos de programação *pthread*, *multi-thread*. Os programadores podem combinar esses diferentes modelos de *threads* no mesmo programa ou na mesma função.

Segundo (Swanson et al., 2007), aplicar diversos estilos de *threads* em um único programa pode expor de forma significante o paralelismo no código, que de outra forma seria difícil de ser completamente paralelizável. Expor o paralelismo é apenas o primeiro passo. O processador deve então traduzir esse paralelismo em desempenho.

O *WaveCache* é baseado no modelo descentralizado da execução a fluxo de dados, assim, ele não tem nenhuma unidade de processamento central. Ao invés disso, o *WaveCache* consiste de nós de processamento, que substituem o processador central e o cachê de instruções de um sistema convencional. A utilização de técnicas de *threads* em programas *WaveScalar*, necessita de um compilador próprio para a arquitetura. *WaveScalar* é uma arquitetura a fluxo de dados dinâmica. Nele, os *tags* são chamados de *waves numbers*. Os *tokens* são simbolizados com *waves numbers*

"w"e valor "v"como "w.v"(Swanson et al., 2007). Ao invés de atribuir diferentes *wave numbers* para diferentes instâncias de uma instrução específica (como faz a maioria das máquinas a fluxo de dados dinâmicas), o *WaveScalar* os atribui na porção delimitada do compilador do grafo a fluxo de dados, chamada *waves*. Os *waves* são similares a *hyperblocks*[5], mas são mais gerais, visto que eles podem conter fluxo de controle juntos e ter mais que uma entrada. Eles não contém *loops*, pois eles são divididos em *waves*, como mostra a Figura 3.9.

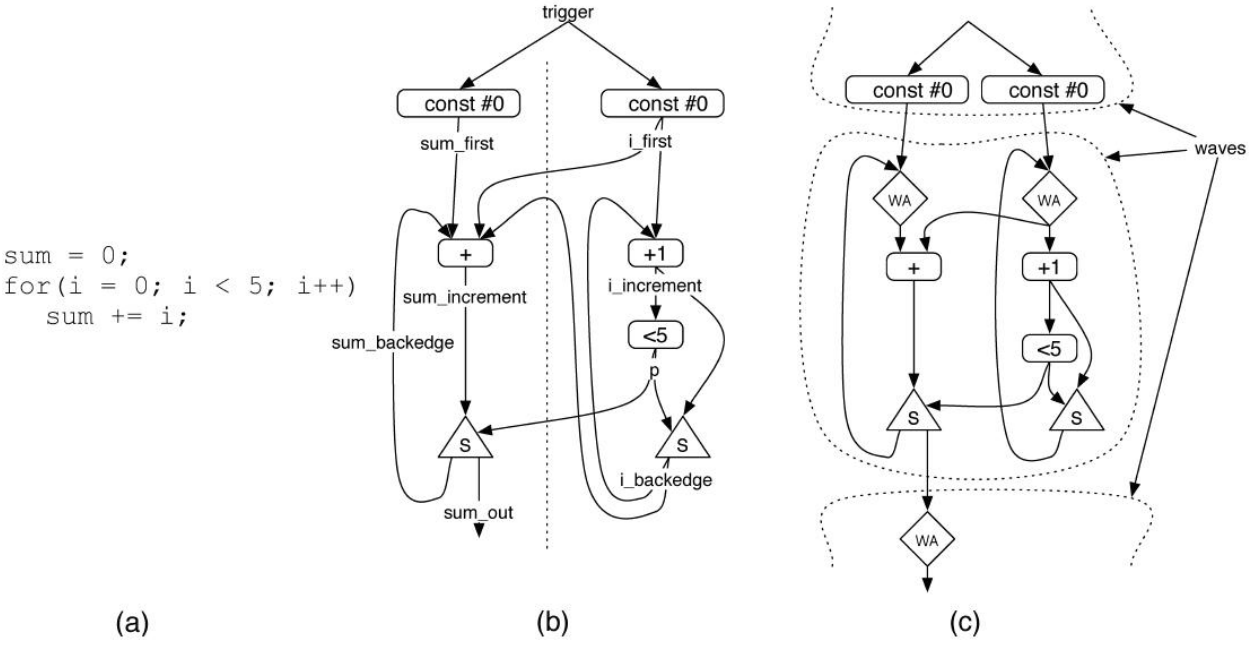


Figura 3.9: *Loops* no *WaveScalar*: (a) um *loop* simples; (b) uma implementação a fluxo de dados; e (c) implementação *WaveScalar*.

No topo de cada *wave* está um grupo de instruções chamadas *WAVE-ADVANCE*, cada qual incrementa o valor *wave number* que passa por ele.

Assumindo que o código antes do *loop* é *wave number* 0. Quando o código executa, as duas instruções *CONST* irão produzir 0.0 (*wave number* 0, valor 0). A instrução *WAVE-ADVANCE* irá receber ele como entrada e produzirá saída 1.0 e assim sucessivamente.

Dentro de cada *wave*, o compilador anota o acesso das instruções na memória para codificar a ordem entre elas.

As arquiteturas baseadas no modelo a fluxo de dados não possuem mecanismos que assegurem que as instruções armazenadas na memória sejam executadas na ordem presente nos programas.

[5]O *hyperblock* corresponde a uma estrutura com uma única entrada e múltiplas saídas laterais

A Figura 3.10 mostra a diferença que pode acontecer entre a execução de um programa em uma linguagem imperativa e um programa em grafo a fluxo de dados. No grafo, a instrução *load* deve executar depois da instrução *store*. O grafo a fluxo de dados não expressa esta implícita dependência entre as duas instruções. Como já mostrado anteriormente, o *WaveScalar* fornece um mecanismo para codificar esta implícita dependência para suportar linguagens imperativas. *Wave-ordered memory* resolve o problema de ordenação de memória em grafos a fluxo de dados usando *waves*. A linha pontilhada na Figura 3.10 representa a esta implícita depedência entre as instruções. No interior de cada *wave*, o compilador anota o acesso das instruções à memória para codificar a ordem entre elas. Visto que o número de *waves* aumentam com a execução do programa, eles fornecem a ordem da execução dos *waves*.

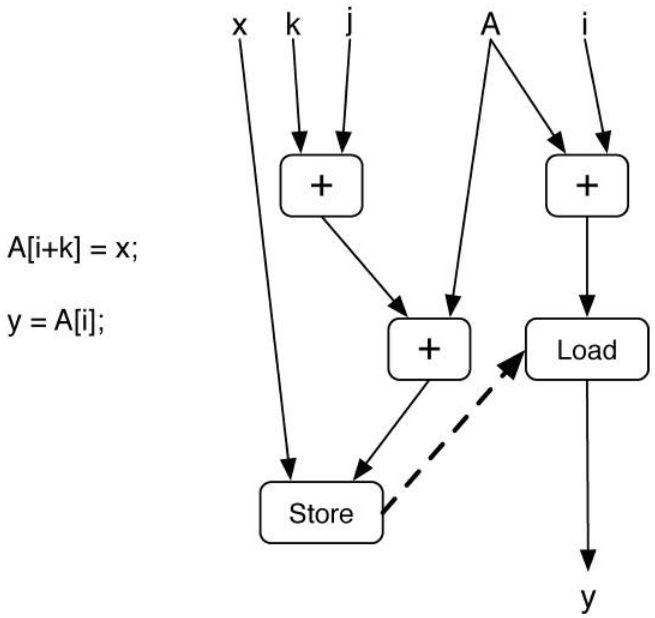


Figura 3.10: Problema relacionado a falta de dependência em um grafo a fluxo de dados.

A arquitetura do *WaveScalar* permite: 1) *Wave-ordered memory* permite ao *WaveScalar* fornecer a semântica que linguagens imperativas requerem e para expressar o paralelismo entre operações carregadas; 2) expressar explicitamente o paralelismo a nível de instruções, enquanto ainda mantém a semântica convencional da memória; 3) o modelo de execução *WaveScalar* é distribuído, ou seja, só instruções que tem que trocar dados com outras se comunicam.

O *WaveCache* inclui tudo, exceto memória principal, necessária para executar um programa *WaveScalar*. Ele contém uma rede escalonada de elementos de processamentos a fluxo de dados

idênticos que são organizados hierarquicamente para reduzir os custos de comunicação. Cada nível de hierarquia usa uma estrutura de comunicação separada.

A Figura 3.11 mostra a organização hierárquica da micro-arquitetura do *WaveCache*. Ao invés de projetar um *core* monolítico que englobe o programa inteiro, um *tiled* processador cobre o programa com centenas ou milhares de *tiles* idênticos, cada qual é uma unidade de processamento simples, porém completa. Visto que eles são menos complexo que um *core* monolítico, os *tiles* tem o custo e a verificação do projeto amortizados.

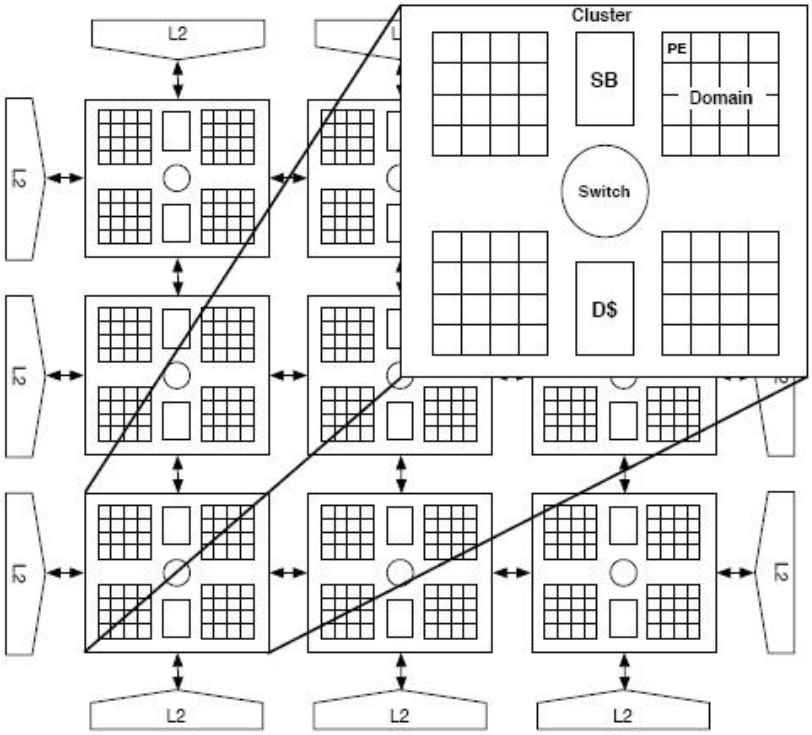


Figura 3.11: Organização hierárquica da micro-arquitetura do *WaveCache*.

No *WaveCache*, cada *tile* é chamado de *cluster*. Um *cluster* contém quatro *domains* idênticos, cada qual com oito elementos de processamento idênticos. Os *domains* da Figura 3.11 possuem 16 *PEs* cada.

Na Figura 3.12 é representado a execução do *loop* da Figura 3.9(c) mapeada sobre dois *domains*. Cada quadrado grande representa um elemento de processamento. Em adição, cada *cluster* possui quatro bancos de cachê de dados, interface em *hardware* para *wave-ordered memory* e uma rede de troca para comunicação com *clusters* adjacentes. Múltiplas instruções são dinamicamente colocadas em um número fixo de elementos de processamento, cada qual pode calcular 64 in-

struções. Os *PEs* acessam a memória enviando requisições para a interface de memória em seu *cluster* local.

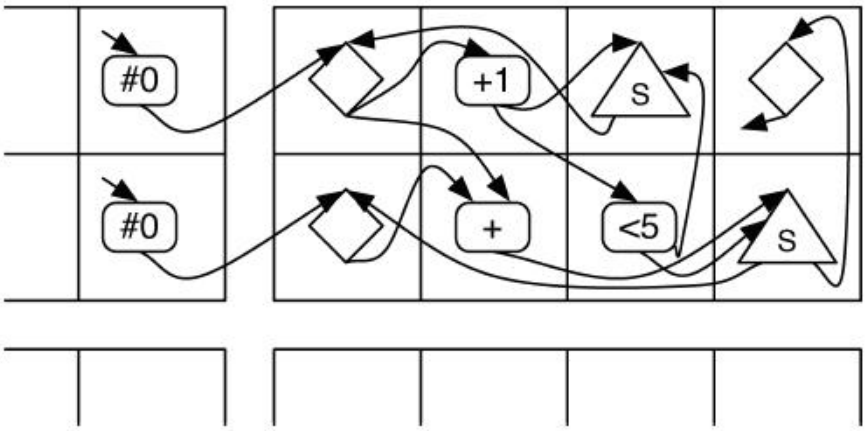


Figura 3.12: *Loop* da Figura 3.9(c) mapeado sobre dois *domains WaveCache*.

As instruções são mapeadas e colocadas nos *PEs* dinamicamente conforme a execução do programa.

As responsabilidades de um elemento de processamento são implementar a regra de disparo do modelo a fluxo de dados e executar as instruções.

Cada *PE* contém uma unidade funcional, memórias especializadas para guardar operandos e lógica para controlar a execução das instruções e comunicação. Ele também contém *buffers* e armazenamento para várias instruções estáticas diferentes. Um *PE* possui cinco estágios de *pipeline*, com redes de desvio, que permitem a execução de instruções dependentes no mesmo *PE*.

A Figura 3.13 mostra o elemento de processamento do *WaveCache* que foi implementado em cinco estágios de pipeline:

- Entrada (*input*): Mensagens de operandos chegam ao *PE* por meio de outro *PE* ou do próprio.
- Combinação (*match*): Os operandos entram na tabela de *matching*, que determina quais instruções estão prontas para disparar.
- Envio (*dispatch*): O *PE* escolhe uma instrução da fila de execução, lê seus operandos pela tabela de *matching*, e os envio para o estágio "execução".

- Execução (*execute*): Executa a instrução e envia os resultados para a fila de saída e/ou para a rede de comunicação local.

- Saída (*output*): Uma saída de instrução é enviada para as instruções consumidoras por meio da rede de comunicação interna. Consumidores podem ser outros *PEs* ou o próprio. Para reduzir os custos de comunicação com a rede, os *PEs* são organizados hierarquicamente junto com suas infra-estruturas de comunicação. Um aspecto principal do projeto garante que ele evite o grande, centralizado armazenamento do *tag-matching* encontrado em algumas máquinas a fluxo de dados prévias.

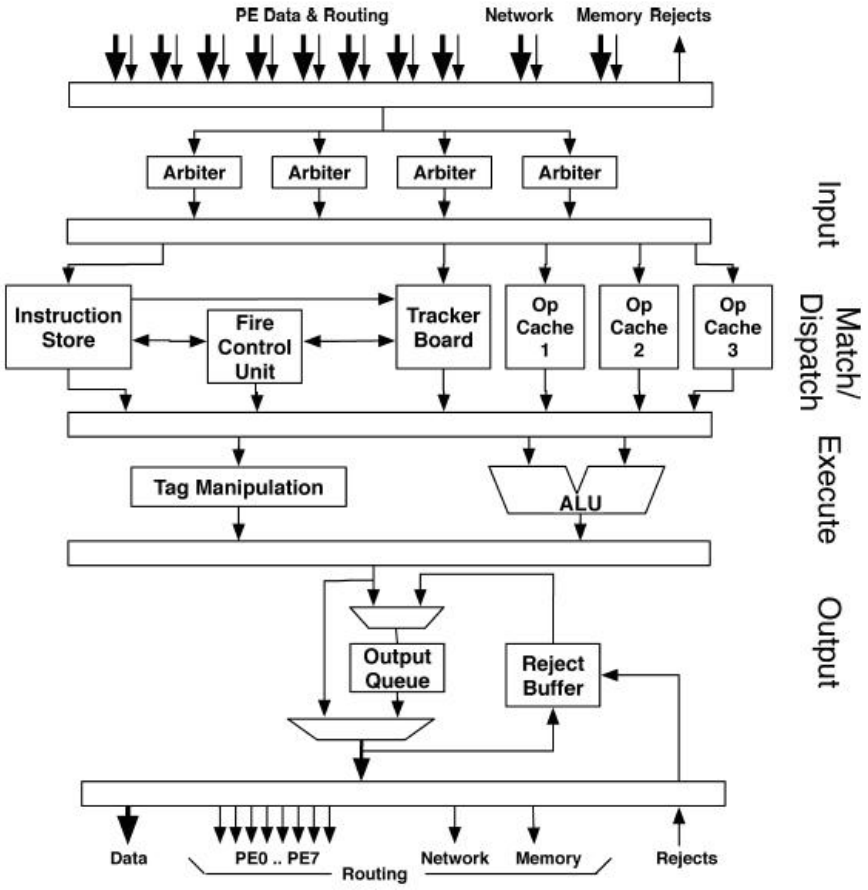


Figura 3.13: Diagrama de bloco de um *PE*.

A um alto nível, a estrutura do *pipeline* do *PE* lembra uma estrutura convencional de *pipeline*. A grande diferença entre os dois é que a execução do *PE* é inteiramente dirigida por dados ao invés da execução de instruções provida por um contador de programas, como as encontradas nas máquinas *von-Neumann*.

Novos *tokens* são armazenados em um cachê *matching*. Se um *token* residir no *matching* por um tempo longo, outro *token* pode sucedê-lo. Neste caso o *token* antigo é enviado para a tabela *matching* na memória.

A tabela *matching* é separada em três colunas, uma para cada entrada de instrução (certas instruções *WaveScalar* podem ter três entradas).

O esquema de alocação de instruções no *WaveScalar* tem um compilador e um componente em tempo de execução. O compilador é responsável por agrupar as instruções em seguimentos, enquanto o componente em tempo de execução aloca o seguimento inteiro de instruções em um mesmo *PE*. Por causa disso, o compilador tenta agrupar as instruções em um mesmo seguimento se não for provável que essas instruções executem simultaneamente, mas compartilham operandos, então podem utilizar a rede local rápida disponível dentro de cada *PE*.

Em tempo de execução, o *WaveCache* carrega um seguimento de instruções quando uma instrução que não foi mapeada dentro dele necessita ser executada. Como mostrado acima, um seguimento inteiro é mapeado em um único *PE*. Por causa da ordenação que o compilador usa para gerar o seguimento, eles irão ser dependentes uns dos outros. Como resultado, eles não irão competir para a execução de recursos, mas ao invés disso, eles irão executar em ciclos consecutivos. O algoritmo preenche todos os *PEs* em um *domain* e então todos os *domains* em um *cluster*, antes de ir para o próximo *cluster*. Esse esquema de alocação faz o trabalho de escalonamento para minimizar a contenção de recursos e a latência de comunicação.

As instruções do *WaveScalar* descritas até aqui replicam a funcionalidade de um processador *von-Neumann*. Essa capacidade é essencial se o *WaveScalar* for utilizado como uma alternativa a arquiteturas *von-Neumann*, mas isso não limita o que o *WaveScalar* pode fazer.

WaveScalar possui uma segunda interface chamada *unordered memory*, construída para expressar o paralelismo em memória.

A interface *wave-ordered memory* é necessária para executar programas convencionais, mas pode apenas expressar paralelismo limitado.

A interface *unordered memory* do *WaveScalar* não fornece a ordenação seqüencial eficiente que os programas convencionais necessitam, mas expressa o paralelismo porque elimina a restrição de ordenação desnecessária (Swanson et al., 2007). Assim, ela permite aos programadores e compi-

ladores explorar esse fato para expressar o paralelismo entre operações na memória, que podem ser executadas fora de ordem de forma segura. Para assegurar, por exemplo, que uma instrução *load* execute depois de uma operação de *store*, elas devem ter uma dependência de dados entre elas.

O processador *TRIPS / GPA* (Nagarajan et al., 2001; Sankaralingam et al., 2003) e *WaveScalar* estão investindo no mesmo desafio tecnológico e tendem a usar a mesma terminologia para descrever aspectos de seus projetos (Swanson et al., 2003).

TRIPS é uma arquitetura polimórfica, hibrida *von-Neumann / modelo a fluxo de dados*, a qual pode ser configurada para diferentes granularidades e tipos de paralelismo. Em (Swanson et al., 2003) o processador *TRIPS* é descrito como uma arquitetura híbrida *VLIW* a fluxo de dados.

TRIPS contém mecanismos que habilitam os cores de processamento e sistema em memória *on-chip* que pode ser configurado e combinado em diferentes nós para paralelismo ao nível de instruções, dados ou *threads*.

Para explorar o paralelismo da aplicação e prover um grande uso dos recursos disponíveis, *TRIPS* usa três nós de execuções diferentes: *D-morph*, que procura o paralelismo em nível de instruções; *T-morph* que trabalha ao nível de *thread*, mapeando múltiplos *threads* em um único *core* do *TRIPS* e *S-morph* que tem como objetivo aplicações como fluxo de mídia (*streaming media*), com alto nível de paralelismo ao nível de dados.

Segundo (Swanson et al., 2003), a única característica arquitetural que *TRIPS* e *WaveScalar* compartilham é o uso de ligações entre instruções no mesmo *hyperblock* (no *TRIPS*) e grafos acíclicos dirigidos ou *wave* (no *WaveScalar*).

No interior dos *hyperblocks*, as instruções disparam de acordo com a regra de disparo do modelo a fluxo de dados, enquanto a comunicação entre *hyperblocks* ocorre por meio de arquivo de registrador (Swanson et al., 2003).

No *core TRIPS* há resquícios da arquitetura *von-Neumann*, porque o contador de programa para guiar a execução, mas ao invés de selecionar instrução por instrução na memória, o contador de programas no sistema *TRIPS* seleciona *frames* (semelhantes a *hyperblocks*) de instruções de execução em um arranjo de dezesseis elementos de processamento que constituem o processador *TRIPS* (Swanson et al., 2007).

TRIPS utiliza a estratégia de construção de *chips* heterogêneos, os quais contêm múltiplos cores de processamento, cada qual designado a executar uma classe distinta de carga de trabalho. O polimorfismo da arquitetura *TRIPS* deve-se a capacidade de configurar o *hardware* para executar de forma eficiente, grandes classes de aplicações.

A Figura 3.14 mostra algumas arquiteturas de granularidades diferentes. A arquitetura de granularidade fina, mostrado na figura, pode oferecer um alto desempenho em aplicações com paralelismo de grão fino (paralelismo ao nível de dados), mas terá dificuldade de alcançar bom desempenho em aplicações de propósito geral e aplicações seriais. No outro extremo da figura, pode ser visto uma arquitetura de granularidade grossa, que por sua vez, não tem a capacidade de usar o *hardware* interno para mostrar alto desempenho em aplicações de granularidade fina, ou seja, aplicações com alto paralelismo.

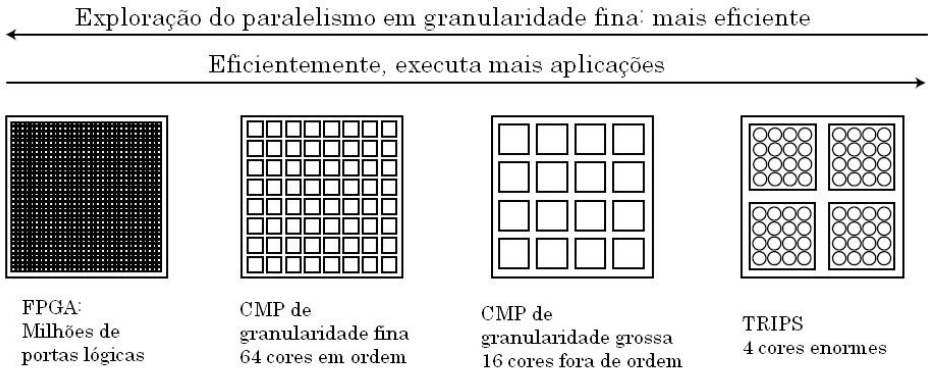


Figura 3.14: Granularidade de elementos de processamento paralelos em um *chip*.

O polimorfismo pode atravessar essa dicotomia com qualquer uma das duas abordagens. Uma aproximação de síntese usa a granularidade fina para explorar aplicações com grão fino e paralelismo regular; paralelismo de grão grosso sintetiza múltiplos elementos de processamento em um processador *lógicos* maior (Sankaralingam et al., 2003).

Uma aproximação de divisão implementa uma granularidade grossa em *hardware* e logicamente divide esse grande processador para explorar o paralelismo em grão fino, quando ele existe.

A arquitetura *TRIPS* usa uma abordagem de separação, combinando uma rede de *cores* de processadores polimórficos de granularidade grossa com um sistema polimórfico adaptativo de memória em *chip*. Assim, a arquitetura prove o máximo desempenho para aplicações *single-thread*, enquanto o restante do divisor explora o paralelismo de grão-fino.

A Figura 3.15 ilustra a arquitetura do *TRIPS*. Como visto na figura, a arquitetura de *TRIPS* utiliza grandes *cores* de processamento de granularidade grossa para alcançar alto desempenho em aplicações *single-thread* com alto *ILP* (*Instruction-Level Parallelism*). A arquitetura é dividida para evitar grandes estruturas centralizadas e longos fios de execução. Esta divisão computacional e elementos de memória são conectados por canais ponto a ponto que são expostos por escalonadores de *software* para otimização. Segundo (Sankaralingam et al., 2003) o desafio chave em definir as características do polimorfismo está em balancear suas granularidades apropriadas para que cargas de trabalho envolvendo diferentes níveis de *ILP*, *TLP* (*Thread Level Parallelism*) e *DLP* (*Data Level Parallelsim*) possam maximizar seus usos de recursos disponíveis e ao mesmo tempo evitar estruturas complexas em ascensão e não escalonáveis. O sistema *TRIPS* emprega características polimórficas de granularidade grossa ao nível de bancos de memória e armazenamento de instruções, para minimizar a complexidade de *software* e *hardware* e *overhead* de configuração.

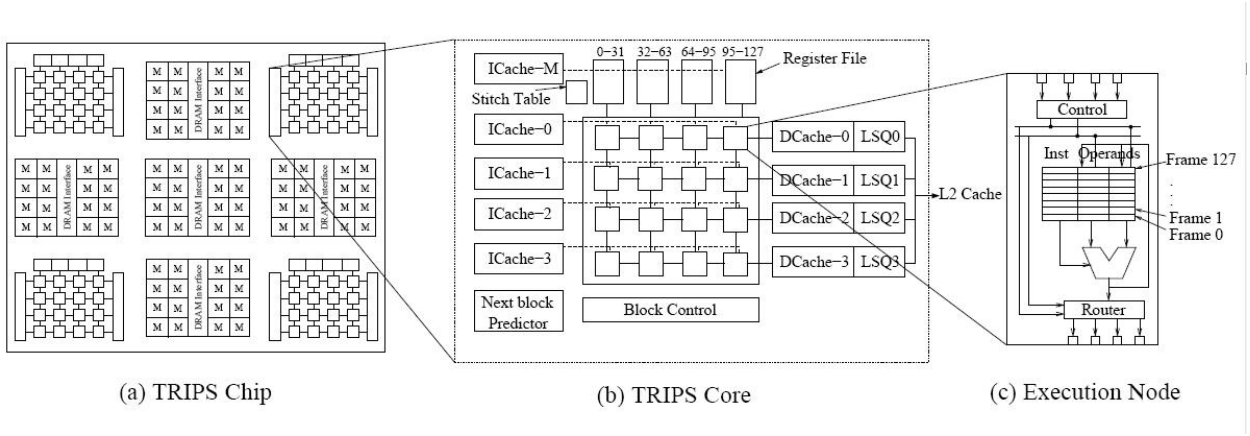


Figura 3.15: Arquitetura *TRIPS*.

A Figura 3.15 (a) mostra o diagrama da arquitetura *TRIPS* que irá ser implementada em um protótipo em *chip*. O protótipo consiste de quatro *cores* polimórficos, com dezesseis nós de execução em um arranjo de 32*Kb* de *tiles* de memória conectados por uma rede de roteamento e um grupo de controladores de memória distribuídos com canais para memória externa.

A Figura 3.15 (b) mostra uma visão expandida de um *core* do sistema *TRIPS* e o sistema primário de memória. O *core* do sistema *TRIPS* é um exemplo de rede de processadores, os quais são tipicamente compostos por um arranjo nós de execução homogêneos, cada qual contém uma *ULA* inteira, unidades de ponto flutuante, um grupo de estações de reserva e conectores de roteamento nas entradas e saídas. Cada estação de reserva tem armazenamentos para uma instrução

e dois operandos de origem. Quando uma estação de reserva contém uma instrução válida e um par de operandos validos, o nó pode selecionar a instrução para execução. Depois da execução, o nó pode transferir o resultado para qualquer *slot* de operando localmente ou estações de reserva remotas dentro de arranjos da *ULA*. Os nós são diretamente conectados com seus vizinhos próximos, mas a rede de roteamento pode enviar os resultados a qualquer nó no arranjo. Detalhes sobre *I-Cache*, *D-Cache* e registradores utilizados no *TRIPS* pode ser visto em (Nagarajan et al., 2001; Sankaralingam et al., 2003).

A Figura 3.15 (c) mostra um nó de execução do sistema *TRIPS*. Cada nó de execução contém um grupo de estações de reservas. Estações de reserva com o mesmo índice atravessam todos os nós para formar um *frame*. Por exemplo: ao combinar-se o primeiro *slot* de todos os nós da rede, forma-se o *frame* 0. Uma coleção de frames é um recurso polimórfico no sistema *TRIPS*, como ele é administrado diferentemente por diferentes nós, ele suportar de forma eficiente a execução de alternadas formas de paralelismo.

A arquitetura *TRIPS* é fundamentalmente orientada a blocos. Em todos os nós de operações, os programas compilados para *TRIPS* são divididos em grandes blocos de instruções com um único ponto de entrada, nenhum laço de repetição e múltiplos pontos de saídas possíveis como encontrados nos *hyperblocks*.

Para todos os nós de execução, o compilador é responsável por escalonar estaticamente cada bloco de instruções em mecanismos computacionais tais que as dependências entre instruções são explicitas.

Em tempo de execução, o fluxo básico operacional do processador inclui recuperar um bloco da memória, abri-lo dentro do mecanismo computacional, executá-lo por completo, entregar seus resultados para o estado freqüente se necessário e então proceder para o próximo bloco.

Segundo (Swanson et al., 2007), apesar do alto nível de semelhanças entre *waves* (*WaveScalar*), *frames* (*TRIPS*), os projetos *WaveScalar* e *PEs* do sistema *TRIPS*, as duas arquiteturas são totalmente diferentes.

Devido ao uso de contador de programas para selecionar *frames* de instruções para execução, *TRIPS* deve especular agressivamente. Ao mapear um *frame* de instrução em um arranjo de *PEs* leva vários ciclos, então o processador *TRIPS* de forma especulativa organiza os frames nos *PEs*

antes do tempo. *WaveScalar* por sua vez, não tem esse problema porque a execução do modelo na fluxo de dados dinâmico permite que as instruções continuem na rede para muitas execuções, livrando-se assim da necessidade de especulação (Swanson et al., 2007).

A desvantagem da abordagem *WaveScalar* é a necessidade de um complexo *tag-matching* em *hardware* para suportar a execução a fluxo de dados dinâmica.

Outras diferenças e semelhanças entre *WaveScalar* e *TRIPS* podem ser encontrados em (Swanson et al., 2007).

CAPÍTULO

4

Projeto ChipCflow

O *ChipCflow* é o projeto de uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em *hardware* reconfigurável. Tem como principal objetivo acelerar programas de aplicação escritos em C. Em particular, essa aceleração vai acontecer por meio da execução direta em *hardware*, aproveitando ao máximo o paralelismo considerado natural do modelo a fluxo de dados.

A Figura 4.1 descreve a seqüência de atividades para que uma aplicação seja utilizada na ferramenta *ChipCflow*.

A partir do programa de aplicação escrito originalmente na linguagem C, se extrai os grafos a fluxo de dados tendo como referência para os operadores, uma base de grafos a fluxo de dados a ser gerada especificamente para essa ferramenta (Costa, 2008). O grafo a fluxo de dados gerado já estará otimizado. O grafo então é convertido em *VHDL*, tendo como base todo o conjunto de operadores propostos para o *ChipCflow*, previamente implementados em *VHDL*. O código *VHDL* é então sintetizado e simulado em ferramenta *EDA* comercial, em particular trabalharemos com *ISE* da *Xilinx*, e o *bitstream* é então gerado para serem executados direto no *hardware*.

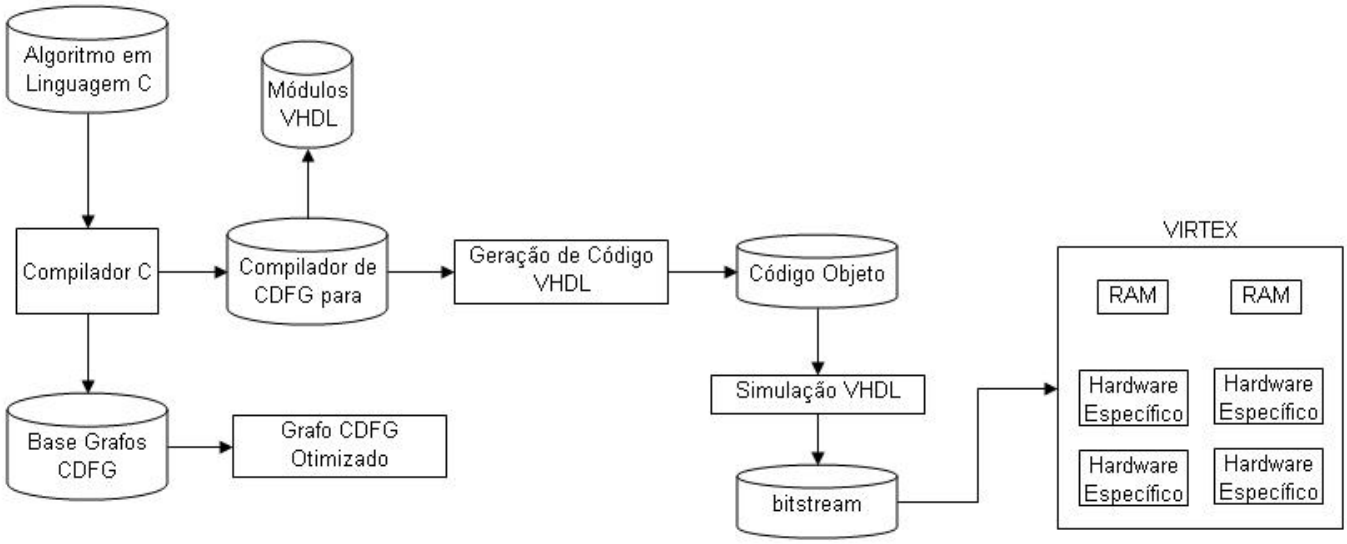


Figura 4.1: Diagrama de Fluxo da Ferramenta *ChipCflow*.

4.1 Estrutura do Modelo a Fluxo de Dados

O modelo de programação a fluxo de dados proposto aqui, segue a linha proposta por (Veen, 1986) que está baseada em uma linguagem gráfica contendo operadores e arcos interligando esses operadores.

Como a arquitetura a fluxo de dados escolhida é o modelo dinâmico, que permite a existência de mais que um dado em cada arco, a implementação do modelo é tal que novas instâncias dos operadores sejam geradas para cada dado que chegue em um arco. Assim, várias instâncias de um operador podem ser geradas, esperando por um dado parceiro.

4.1.1 O Modelo de Instâncias

O projeto *ChipCflow* é baseado em instâncias dos diferentes operadores, utilizando *tagged-tokens* associados a cada dado, permitindo que diferentes instâncias de um mesmo operador possam ser executadas simultaneamente (Astolfi, 2007; Sanches, 2007; Junior, 2008). Cada vez que um operador receber um novo dado em um de seus arcos e o dado parceiro ainda não estiver presente, uma nova instância do operador deverá ser criada esperando a chegada do dado parceiro. Esse modelo será implementado para que dados que possuam o mesmo *tagged-token* façam parte da mesma instâncias de um operador.

Na Figura 4.2 estão descritos várias instâncias de um operador "x"em um grafo a fluxo de dados dinâmico.

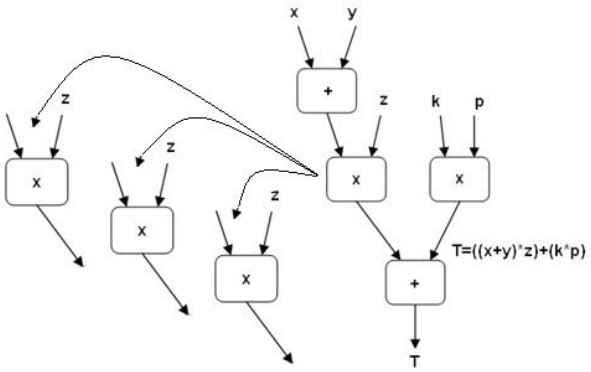


Figura 4.2: Instâncias diferentes para o operador "x".

Para implementar a criação dinâmica de instâncias dos operadores em um grafo a fluxo de dados, foi necessário criar um processo para inserir e remover sub-grafos. Na Figura 4.3 é descrito um grafo T e um sub-grafo F que será inserido no grafo original T.

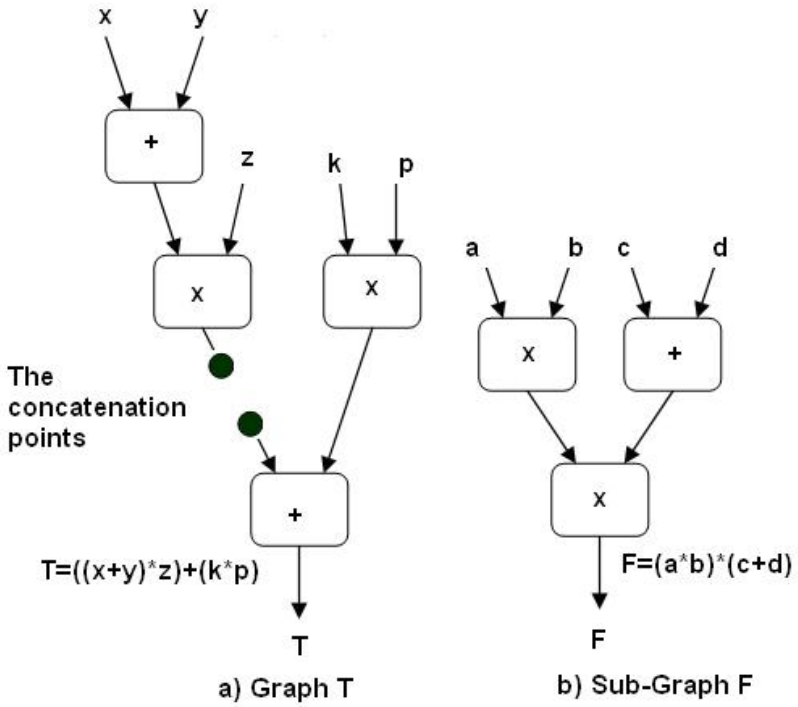


Figura 4.3: Sub-grafo F a ser incluído no grafo original T.

O modelo para inserir e retirar sub-grafos foi baseado em uma estrutura de concatenação. Na Figura 4.3 parte (a), o grafo original T tem dois pontos de concatenação entre dois operadores onde o sub-grafo F parte (b) será concatenado.

Em termos de grafo, precisamos apenas ligar as linhas existentes entre os pontos do grafo T e os arcos do sub-grafo F, mas em termos do modelo, é necessário implementar um protocolo de comunicação que informa "se"e de "onde"um dado estará vindo e "se"e para "onde"um dado estará indo. O resultado da concatenação é descrito na Figura 4.4.

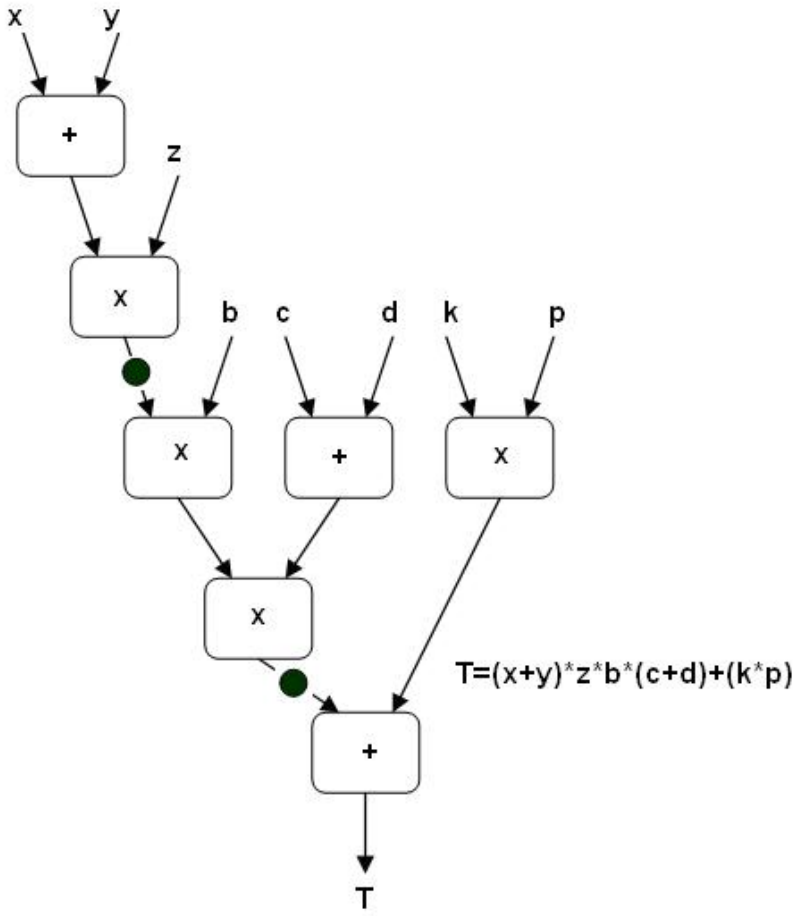


Figura 4.4: Resultado da concatenação do sub-grafo F ao grafo T.

Para remover sub-grafos do grafo T conforme mostra a Figura 4.5, definimos pontos onde a concatenação deverá ocorrer e o mesmo processo acima deverá ser executado, mas agora concatenando os pontos onde o sub-grafo foi removido. O grafo resultante é mostrado na Figura 4.6.

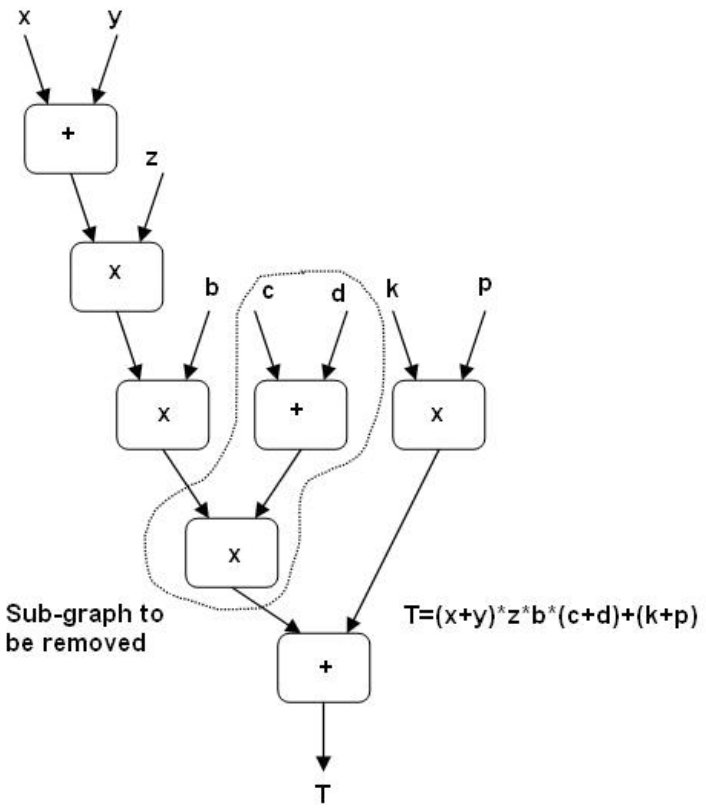


Figura 4.5: Sub-grafo a ser removido do grafo original T.

Assim, quando for gerada uma instância de um operador, um sub-grafo é criado tendo como pontos de concatenação os mesmos pontos de entrada e saída de dados do operador. Da mesma forma, quando uma instância de um operador for disparado para execução, um dado de saída é gerado e aquela instância (sub-grafo do operador) é removida, restando apenas o operador e suas instâncias esperando por dados parceiros.

4.1.2 Operadores utilizados no Modelo

Os operadores a serem utilizado no modelo são: *Decider*, *Non Deterministic Merge*, *Deterministic Merge*, *Branch*, *Copy* e *Operator*, sendo esses os principais objetivos da dissertação aqui proposta, acrescidos de suas execuções em *hardware* através da implementação de grafos a fluxo de dados. O operadores estão descritos na Figura 4.7

Como mostra a Figura 4.8, há dois tipos de enlace sobre um grafo a fluxo de dados, um representado por linhas contínuas, que transporta dados ou estruturas de dados (*data link*), e outro que

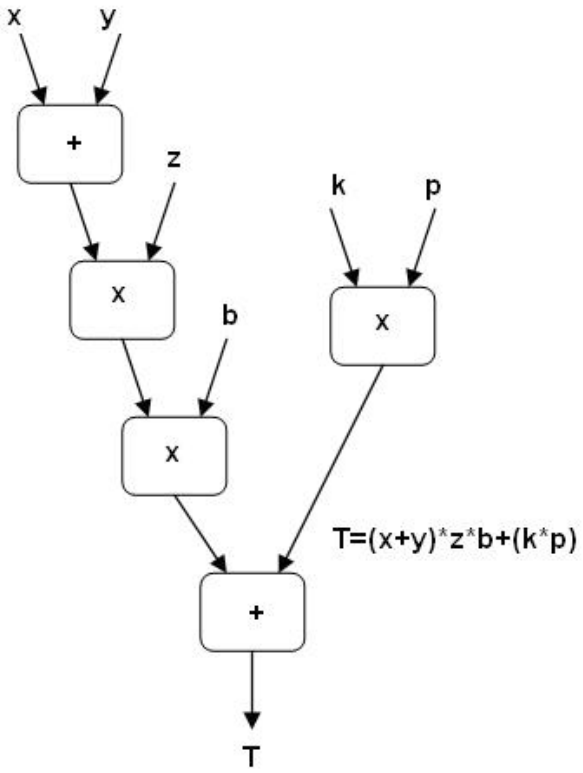


Figura 4.6: Grafo Resultante da remoção de sub-grafo.

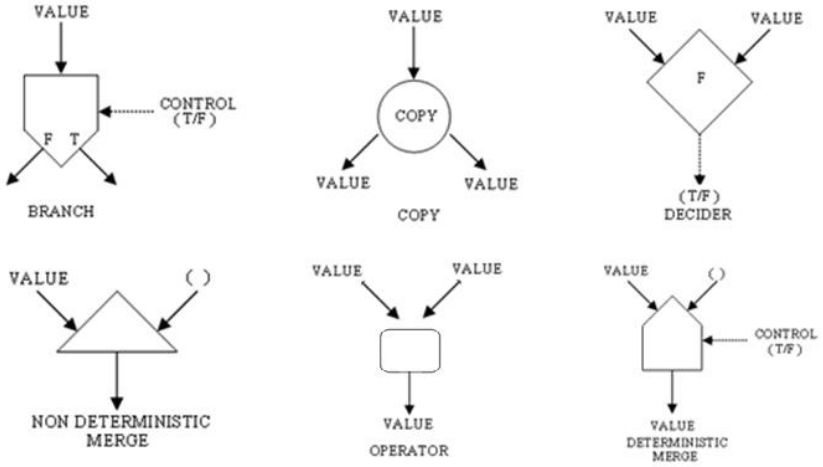


Figura 4.7: Operadores do Modelo ChipCflow.

transporta valores booleanos, que são utilizados com finalidade de controle (*logical link*), representado por linhas tracejadas.

Utilizando os operadores descritos na Figura 4.7 podemos construir grafos como o apresentado na Figura 4.9 que expressa alguns comando em C.

Os operadores de ramificação (*branch*), junção (*merge*) e decisão (*decider*), são usados para representar computações iterativas ou condicionais no grafo. O nó *decider*, como mostra a Figura 4.7,

Figura 4.8: Tipos de enlace dos grafos a fluxo de dados.

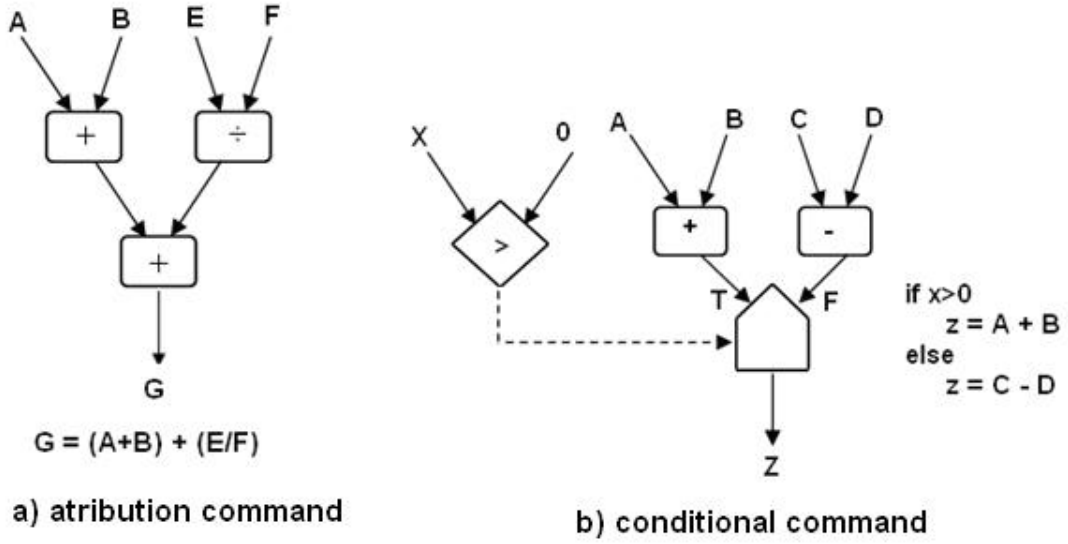


Figura 4.9: Grafo representando comandos em C.

gera um pacote (ou *token*) de controle com valor booleano, dependendo de seus valores de entrada e da função de teste T. Em um nó de ramificação (*branch*), ilustrado pela Figura 4.7, uma cópia do *token* absorvido pela porta de entrada é colocada no arco de saída verdadeiro (T) ou falso (F), dependendo do valor booleano do *token* de controle.

Nos grafos a fluxo de dados, pacotes de dados podem ser dirigidos por meio dos nós de junção (*merge*). Em uma junção determinística (*deterministic merge*), um *token* de controle, que pode assumir valor verdadeiro ou falso, determina de qual porta de entrada um *token* é absorvido. Uma cópia do *token* absorvido é enviado para o arco de saída. O *token* não escolhido é descartado. Na junção não-determinística (*nondeterministic merge*), não existe uma regra de habilitação estrita, isto é, o nó é habilitado tão logo uma de suas portas de entrada contenha um *token*. Quando ele dispara, uma cópia de seu *token* de entrada é enviada para seu nó subseqüente.

Como já visto, um *token* de dado é produzido por um operador (*operator*) como resultado de alguma operação aritmética ou lógica. Finalmente, copiador (*copy*) é um operador que duplica *tokens* de entrada.

4.1.3 Construções Iterativas no Modelo

As construções iterativas são estruturas que necessitam um sincronismo toda vez que um novo dado circula por qualquer uma dessas construções. Para isso foram definidos *tags* que serão utilizados para sincronizar os dados do grafo. O formato dos *tags* proposto no projeto *ChipCflow* está descrito na Figura 4.10.

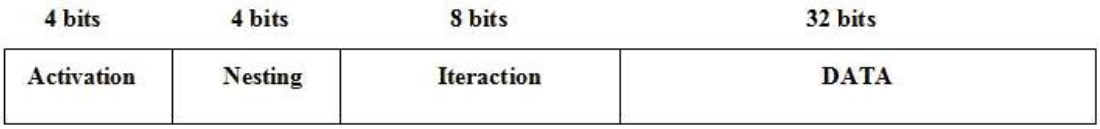


Figura 4.10: Formato dos dados contendo Tags.

Para cada dado sendo executado em um programa, função ou procedimento, é gerado um *tag*, representando uma ativação (campo *activation* na Figura 4.10). No caso de existirem *loops* implementados com operações iterativas (*while, repeat* e *for*), a entrada em cada uma dessas operações irá alterar o *tag* (campo *iteration* da Figura 4.10), especificando nova iteração.

Como as operações iterativas podem ser aninhadas, é necessário que o *tag* contenha também o nível de aninhamento (campo *nesting* na Figura 4.10).

Finalmente, ao final de cada programa, função ou procedimento, o *tag* será modificado informando que aquela ativação terminou.

Os operadores específicos para o controle dos construtores iterativos são: *new tag manager* (NTM); *new iteration generation* (NIG) e *new tag destructors* (NTD).

O operador NTM é responsável pela alocação de um novo *tag*. O operador NIG modifica o *tag* gerando um novo valor para o campo *iteration*. O operador NTD modifica os *tags* no campo *activation*, quando esse deixa um programa, função ou procedimento.

O exemplo de um programa utilizando construções iterativas é descrito na Figura 4.11.

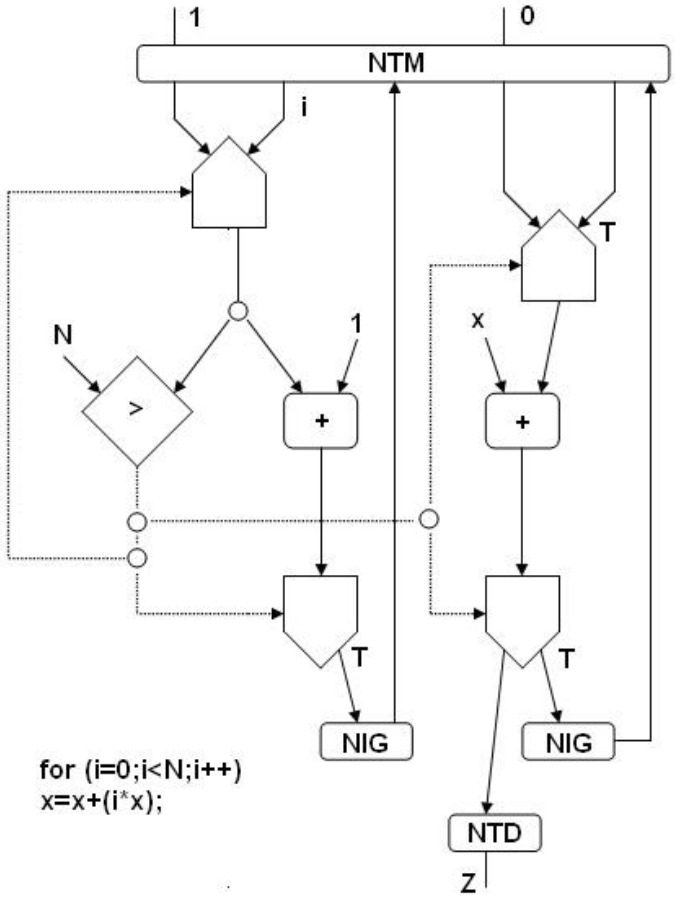


Figura 4.11: Exemplo de um programa com construtores iterativos.

4.2 Estrutura "Matching"de Dados

Como muitas instâncias de um mesmo operador podem existir ao mesmo tempo, foi necessário desenvolver um sistema que permitisse um processo de *matching* de dados de forma rápida com o objetivo de disparar a execução das instâncias cujos dados são parceiros (Lopes, 2008).

Como a identificação de dados parceiros dependem das informações contidas nas *tags*, um circuito de *matching* foi inserido em cada instância de um operador e a busca é feita simultaneamente em cada uma delas. Se uma instância possui um dado com o mesmo *tag*, esse novo dado será inserido naquela instância e se a mesma já possuir todos os dados para execução, ela então é executada e seus dados são retirados daquela instância. Caso contrário, se não houver nenhuma instância com aquele dado, uma nova instância será ocupado com esse dado e ficará aguardando a chegada de dados parceiros.

Quando um dado é encontrado em alguma das instâncias, uma variável é "setada"indicando que existe uma instância com dado parceiro e portanto não será necessária utilizar uma nova instância para aquele dado. Caso contrário, se não ocorrer o *matching* daquele dado, essa mesma variável será "resetada", informando que uma nova instância deve ser utilizada. O circuito *Matching* é descrito na Figura 4.12.

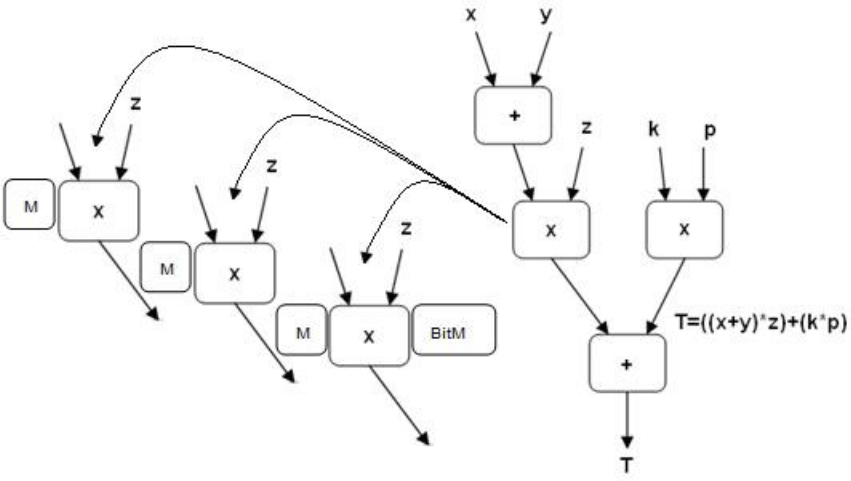


Figura 4.12: Instâncias com circuito de *matching* e variável comum.

Na Figura 4.12, *"M"* representa o circuito de *matching* para cada instância, e a variável *"BitM"* é a variável que informa se ocorreu ou não o *matching* dos dados parceiros.

CAPÍTULO

5

Implementação e Resultados

Neste capítulo o objetivo foi descrever os principais aspectos de implementação dos operadores propostos pelo projeto *ChipCflow*, bem como suas execuções em *hardware*. Para a execução em *hardware* dos operadores implementados por este trabalho de mestrado, serão construídos manualmente grafos a fluxo de dados estáticos e implementados na ferramenta *ISE 8.2i* da *Xilinx* para em seguida serem executados em um dispositivo *FPGA*.

5.1 Operadores do modelo a Fluxo de Dados

Os operadores implementados neste projeto foram os propostos no Capítulo 4.1.2. Estes foram implementados por meio de máquina finita de estados (MFS) e em seguida convertidos para a linguagem *VHDL*.

O estado inicial para cada operador, é responsável em receber os dados de entrada do operador. Uma vez que todos os dados estejam presentes nas entradas de um operador, a MFS faz com que

o operador seja executado. Depois disso, a MFS é responsável por enviar os dados resultantes da execução do operador. Os dados que circulam pelos operadores são dados com 16 *bits*.

Todos operadores possuem sinais de *strobe* (*str*) associados aos sinais de saída de dados. Eles indicam que os sinais de saída de dados de um determinado operador estão disponíveis para serem consumidos pelo operador de destino em seus sinais de entrada de dados. Os operadores também possuem sinais de reconhecimento (*ack*) associados ao sinais de entrada de dados, cuja função é confirmar o recebimento de um dado ao operador que o produziu. Todos operadores possuem um sinal de *clock*. Cabe observar que este *clock* é utilizado somente para sincronismo interno do operador, mas do ponto de vista macro, os operadores se comunicam de forma assíncrona, pois trocam dados por protocolo de comunicação descrito acima.

O operador *Branch* possui um arco de entrada e dois arcos de saída. O dado contido no arco de entrada é copiado para o arco de saída verdadeiro (T) ou falso (F), dependendo do valor booleano contido no sinal de controle. O bloco esquemático do operador *Branch* e sua respectiva representação gráfica são apresentados na Figura 5.1.

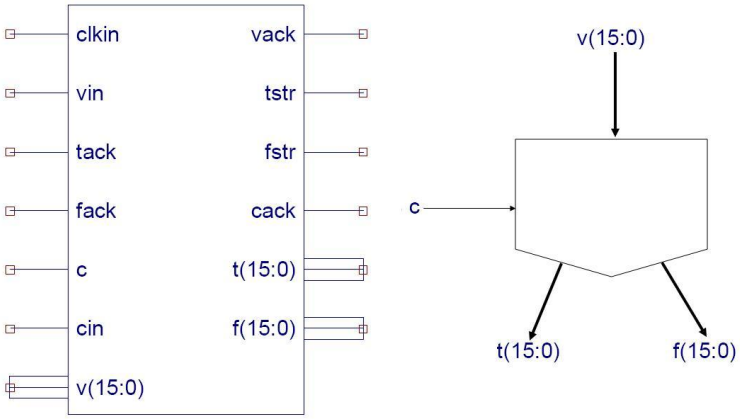


Figura 5.1: Bloco esquemático do operador Branch e sua representação gráfica.

Os sinais *v*, *vin*, *c*, *cin*, *fack*, *tack*, *clkin* são sinais de entrada e os sinais *t*, *f*, *cack*, *fstr*, *tstr* e *vack* são sinais de saída. O sinal *v* é o sinal de entrada de dados; *vin* e *vack* são os sinais de *strobe* e de reconhecimento para o sinal de entrada *v*. O sinal *c* é o sinal de controle do operador; *cin* e *cack* são os sinais de *strobe* e reconhecimento para o sinal de controle. Os sinais *t* e *f* são os sinais de saída de dados do operador; *tstr* e *fstr* são sinais de *strobe* para as saídas de dados *t* e *f*; *tack* e *fack* são os sinais de reconhecimento dos sinais de saída de dados. A Figura 5.2 apresenta o diagrama de estados do operador *Branch*.

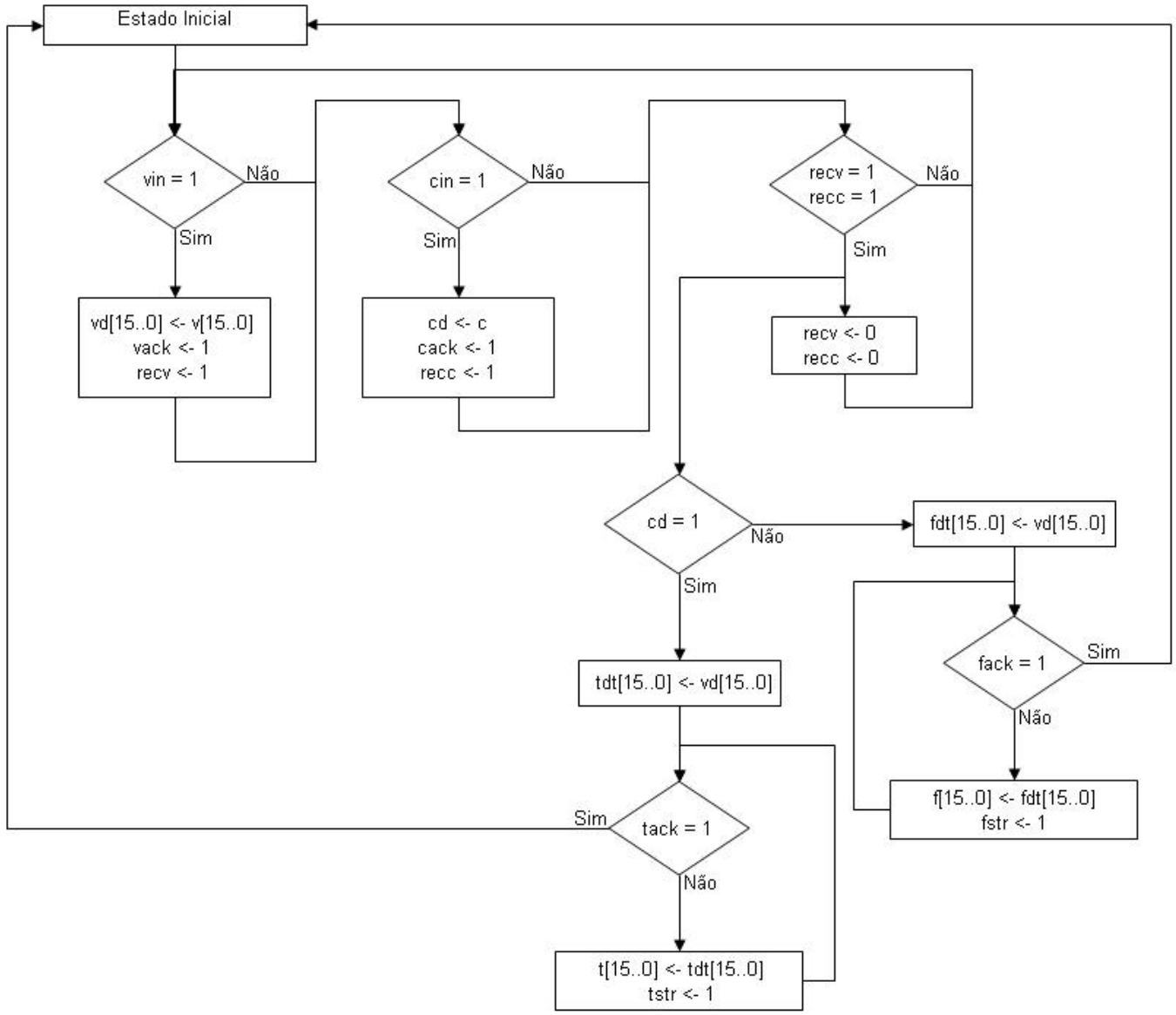


Figura 5.2: ASM chart do operador Branch.

O operador *Copy* simplesmente copia o dado contido no seu arco de entrada para os seus dois arcos de saída. O bloco esquemático do operador *Copy* e sua respectiva representação gráfica são apresentados na Figura 5.3.

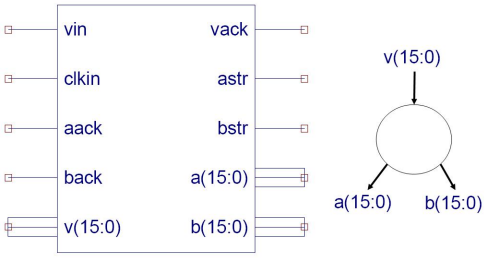


Figura 5.3: Bloco esquemático do operador Copy e sua representação gráfica.

Os sinais *v*, *back*, *aack*, *clkin*, *vin* são sinais de entrada, já os sinais *a*, *b*, *astr*, *bstr*, *vack* são de saída. O sinal *v* é o sinal de entrada de dados; *vin* e *vack* são os sinais de *strobe* e de reconhecimento para o sinal de entrada de dados *v*. Os sinais *a* e *b* são os sinais de saída de dados do operador; *astr* e *bstr* são os sinais de *strobe* para os sinais de saídas de dados; *aack* e *back* são os sinais de reconhecimento para os sinais de saída de dados do operador. A Figura 5.4 apresenta o diagrama de estados do operador *Copy*

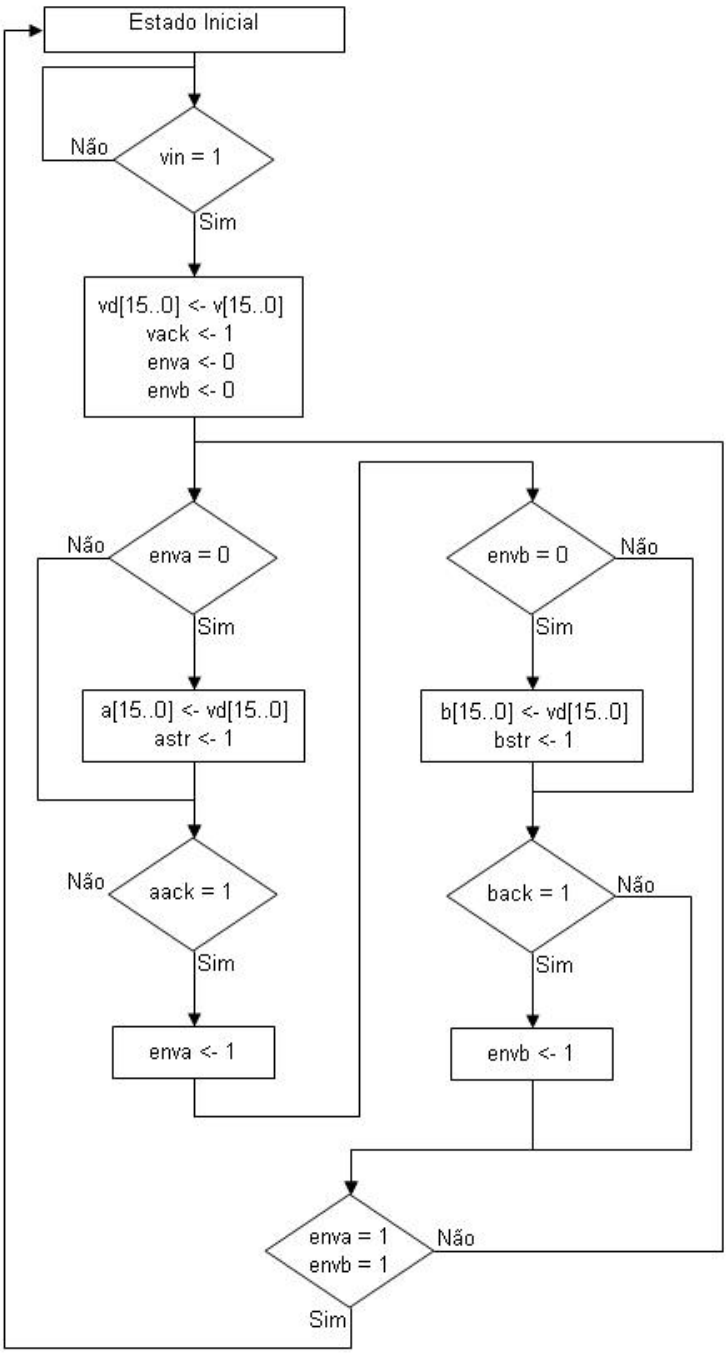


Figura 5.4: ASM chart do operador Copy.

O operador *Decider* aplica uma operação lógica no dado de entrada e gera um dado como resultado contendo um valor booleano. O bloco esquemático do operador *Decider* e sua respectiva representação gráfica são apresentados na Figura 5.5.

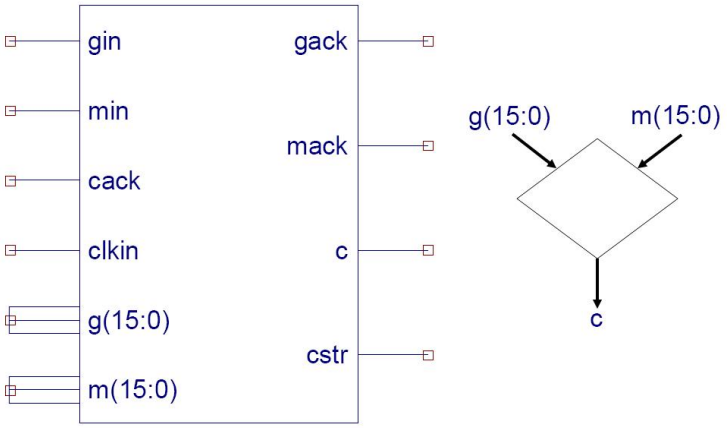


Figura 5.5: Bloco esquemático do operador Decider e sua representação gráfica.

Os sinais *m*, *g*, *clkin*, *cack*, *min*, *gin* são sinais de entrada do operador e os sinais *gack*, *mack*, *c*, *cstr* são sinais de saída. Os sinais *m* e *g* são os sinais de entrada de dados; *min* e *gin* são os sinais de *strobe* para os sinais de entrada de dados; *mack* e *gack* são os sinais de reconhecimento para os sinais de entrada de dados. O sinal de saída de dados *c* contém o resultado de uma decisão lógica aplicado aos dados contidos nos sinais de entrada; *cstr* e *cack* são os sinais de *strobe* e reconhecimento para o sinal de saída de dados. A Figura 5.6 apresenta o diagrama de estados do operador *Decider*.

O operador *Non Deterministic Merge* não possui uma regra de habilitação específica, isto é, nem todos os seus arcos de entrada precisam conter um dado para que o operador possa executar, ele simplesmente copia o dado que recebeu em um dos seus arcos de entrada para o seu arco de saída. A Figura 5.7 apresenta o bloco esquemático do operador *Non Deterministic Merge* e sua respectiva representação gráfica.

Os sinais *a*, *b*, *ain*, *bin*, *vack*, *clkin* são sinais de entrada e os sinais *v*, *vstr*, *aack*, *back* são sinais de saída. Os sinais *a* e *b* são os sinais de entrada de dados; *ain* e *bin* são os sinais de *strobe* para os sinais de entrada de dados; *aack* e *back* são os sinais de reconhecimento para os sinais de entradas de dados. O sinal de sáida de dados *v* contém o dado de umas das entradas *a* ou *b*; *vstr* e *vack*

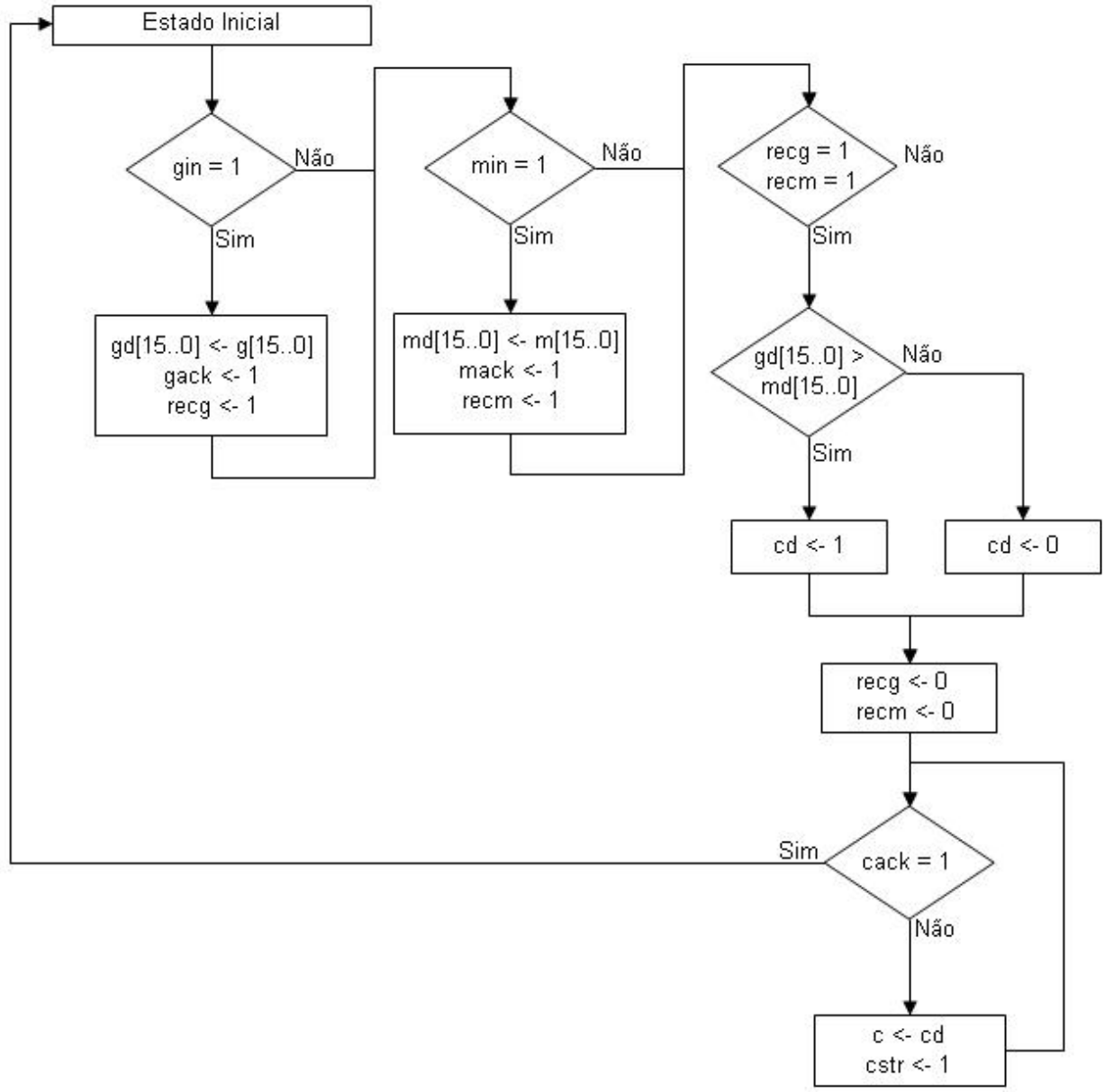


Figura 5.6: ASM chart do operador Decider.

são os sinais de *strobe* e reconhecimento para o sinal de saída de dados. A Figura 5.8 apresenta o diagrama de estados do operador *Non Deterministic Merge*.

Como a implementação das operações aritméticas realidas pelo operador *Operator* são similares, somente o operador *OPADD* que realiza a operação aritmética soma será descrito com maiores detalhes. O bloco esquemático do operador *Operator* responsável pela operação soma e sua representação gráfica são apresentados na Figura 5.9.

Os sinais *a*, *b*, *astr*, *bstr*, *zack*, *clkin* são sinais de entrada e os sinais *z*, *zstr*, *aack* e *back* são sinais de saída. Os sinais *a* e *b* são sinais de entrada de dados; *astr* e *bstr* são os sinais de *strobe*

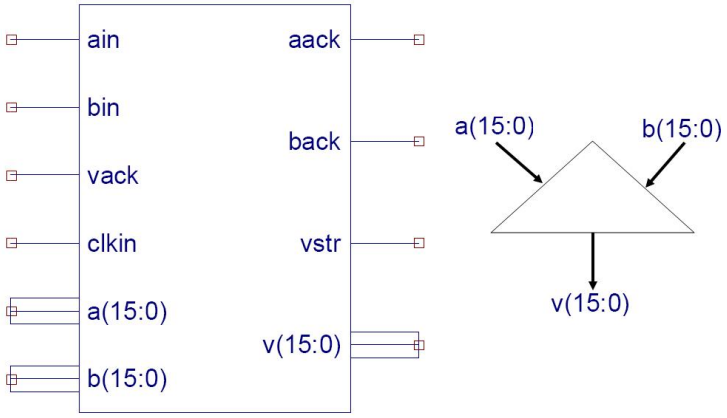


Figura 5.7: Bloco esquemático do operador Non Deterministic Merge e sua representação gráfica.

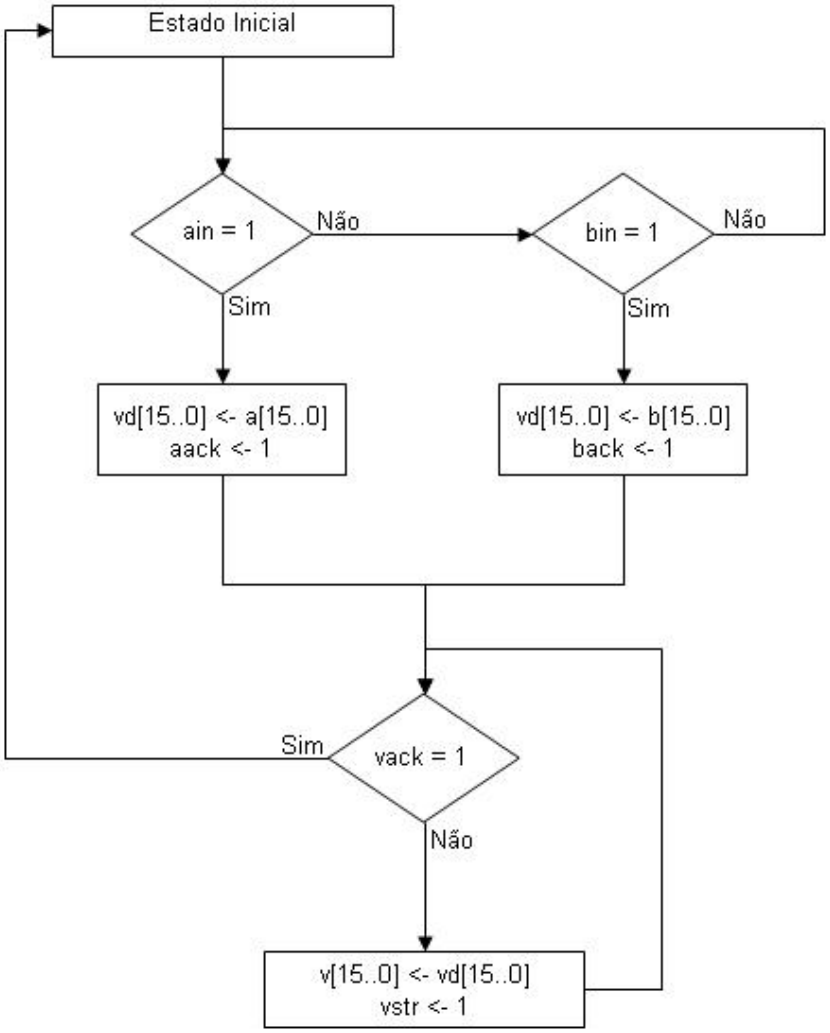


Figura 5.8: ASM chart do operador Non Deterministic Merge.

para os sinais de entradas *a* e *b* respectivamente; *aack* e *back* são os sinais de reconhecimento para os sinais de entrada. O sinal de saída de dados do operador *z* contém a soma dos dados contidos

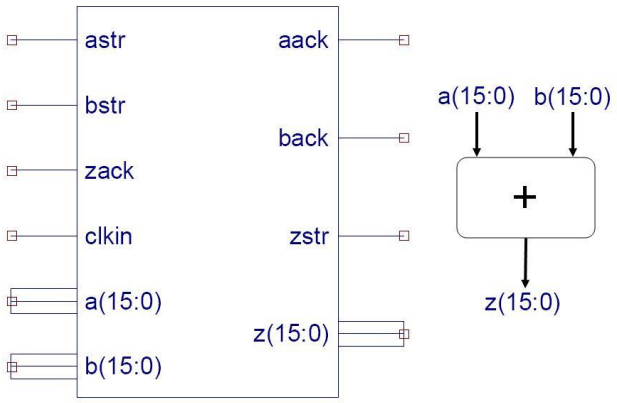


Figura 5.9: Bloco esquemático do operador Operator responsável pela operação soma e sua representação gráfica.

nos sinais de entrada *a* e *b*; *zstr* e *zack* são sinais de *strobe* e reconhecimento para o sinal de saída de dados. A Figura 5.10 apresenta o diagrama de estados do operador *Operator* responsável pela operação soma.

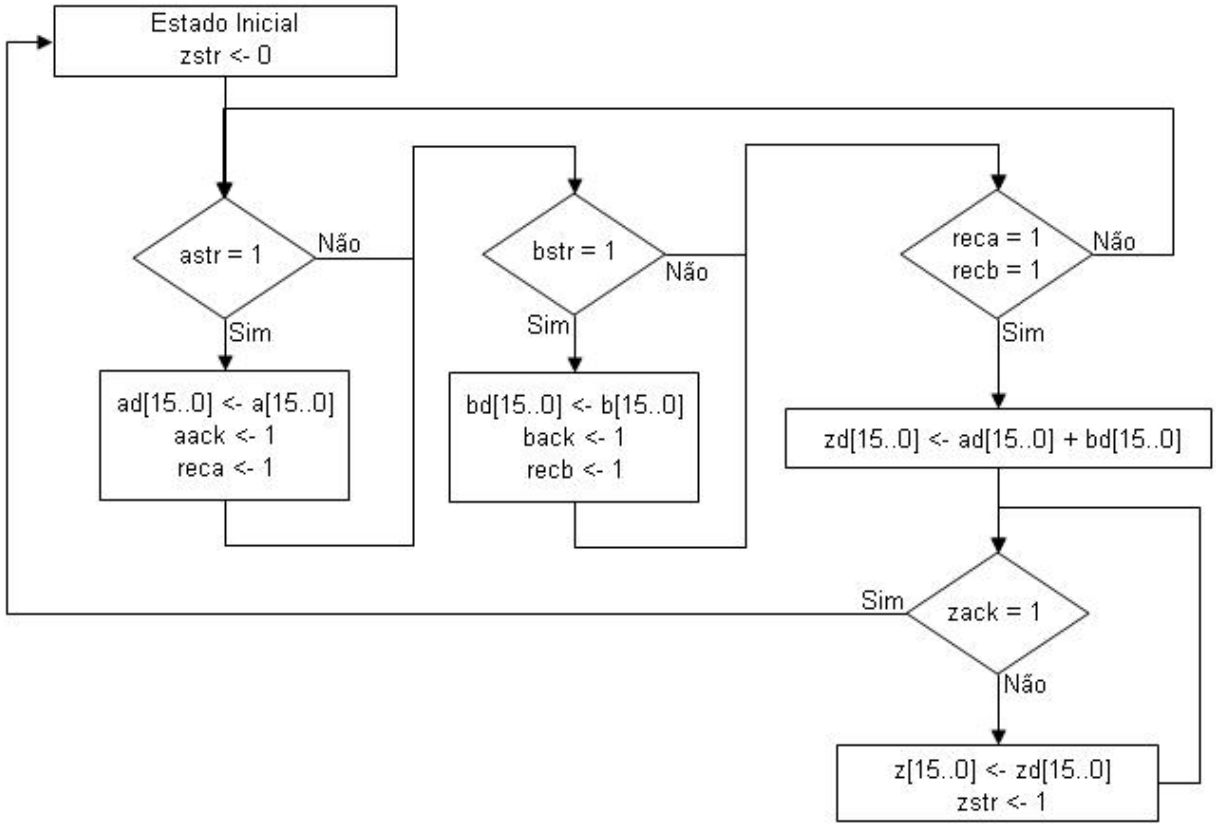


Figura 5.10: ASM chart do operador Operator responsável pela operação soma.

O operador *Deteministic Merge* precisa que todos os dados de entrada estejam disponívies para que ele possa executar, quando isso acontece um dos dados de entrada é copiado para o arco de

saída de acordo com o valor booleano contido no sinal de controle. O bloco esquemático do operador *Deterministic Merge* e sua respectiva representação gráfica são apresentados na Figura 5.11.

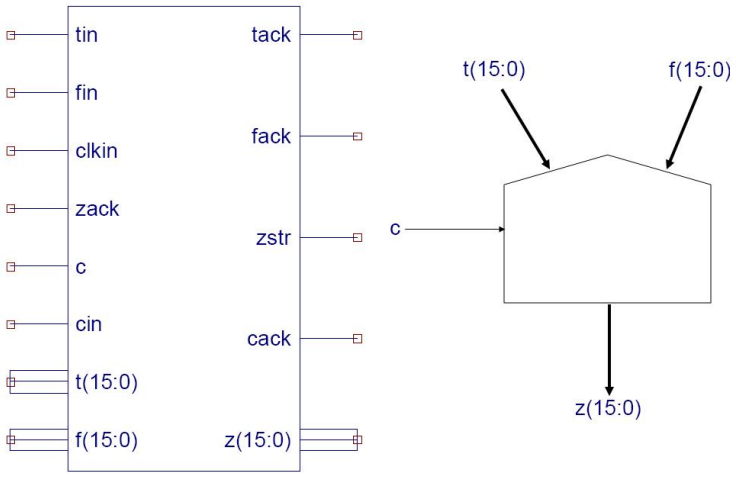


Figura 5.11: Bloco esquemático do operador Deterministic Merge e sua representação gráfica.

Os sinais *t*, *f*, *tin*, *fin*, *c*, *cin*, *zack*, *clkin* são sinais de entrada e os sinais *z*, *zstr*, *tack*, *fack* e *cack* são sinais de saída. Os sinais *t*, *f* são os sinais de entradas de dados; *tin* e *fin* são os sinais de *strobe* para os sinais de entrada de dados; *tack* e *fack* são os sinais de reconhecimento para os sinais de entrada de dados. O sinal *c* é o sinal de controle do operador; *cin* e *cack* são os sinais de *strobe* e reconhecimento para o sinal de controle. O sinal de saída de dados *z* contém um dos dados contidos nos sinais de entrada *t* ou *f* de acordo com o sinal de controle *c*; *zstr* e *zack* são os sinais de *strobe* e reconhecimento para o sinal de saída de dados *z*. A Figura 5.12 apresenta o diagrama de estados do operador *Deterministic Merge*.

5.2 Implementação de Grafos a Fluxo de Dados

Para a validação dos operadores implementados, eles foram executados em *hardware*. Para isso foram construídos manualmente grafos a fluxo de dados. Estes foram implementados utilizando a ferramenta *ISE 8.2i* da *Xilinx*. As simulações dos grafos a fluxo de dados foram realizadas utilizando o *ISE Simulator* da *Xilinx*.

Para implementar os grafos os grafos a fluxo de dados percebeu-se a necessidade de criar operadores extras, sendo eles *INDATA*, *AOUT* e *ITMAN*.

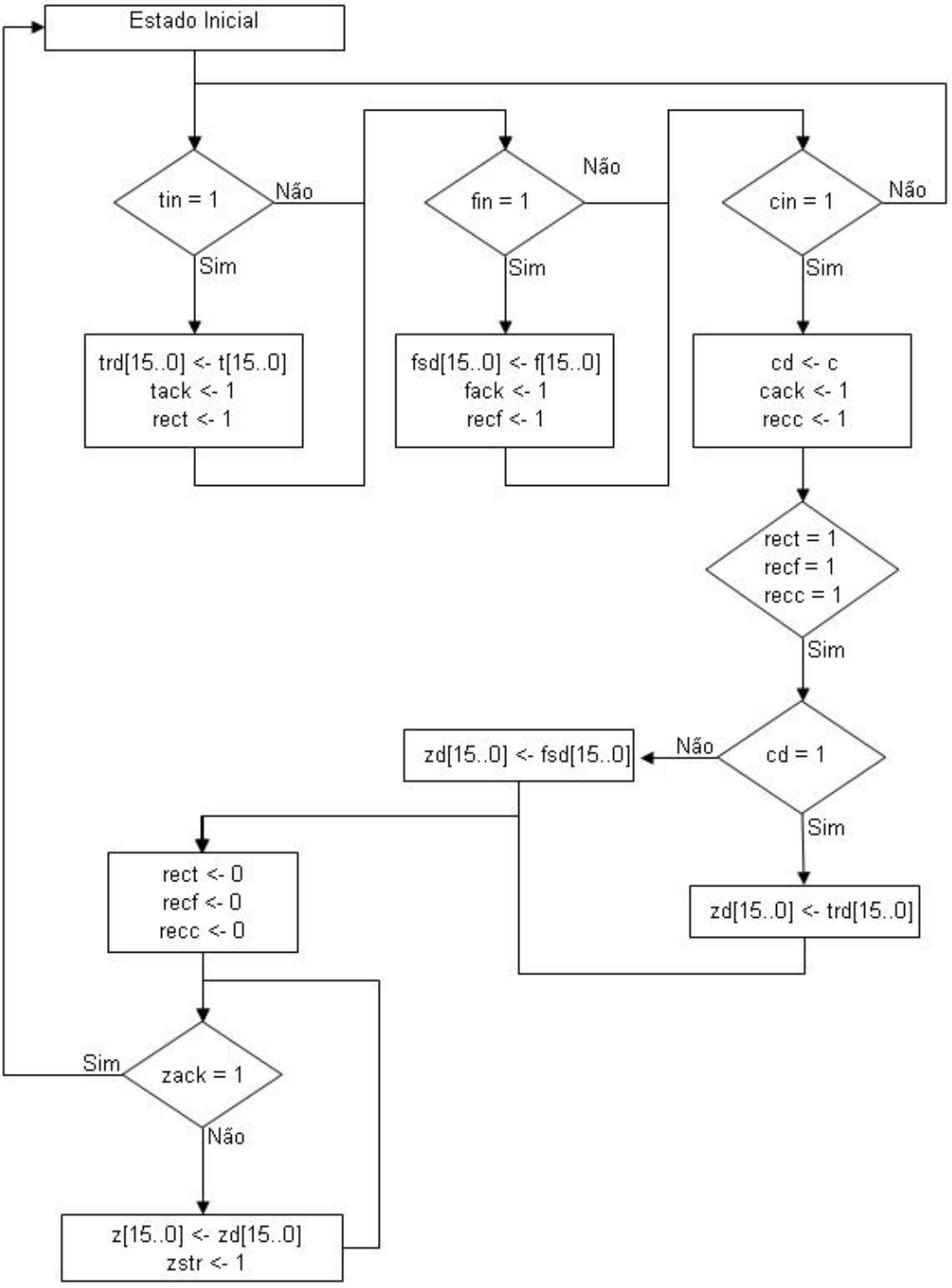


Figura 5.12: ASM chart do operador Deterministic Merge.

O operador *INDATA* é responsável em sincronizar os dados inseridos dentro do grafo. O bloco esquemático do operador *INDATA* é apresentado na Figura 5.13.

Os sinais *a*, *aack*, *ain* e *clkin* são os sinais de entrada e os sinais *aout* e *astr* são os sinais de saída. O sinal *a* é o sinal de entrada de dados; *ain* é o sinal de *strobe* para o sinal de entrada de

Figura 5.13: Bloco esquemático do operador INDATA.

dados. O sinal *aout* é o sinal de saída de dados; *astr* e *aack* são os sinais de *strobe* e reconhecimento para o sinal de saída de dados. A Figura 5.14 apresenta o diagrama de estados do operador *INDATA*.

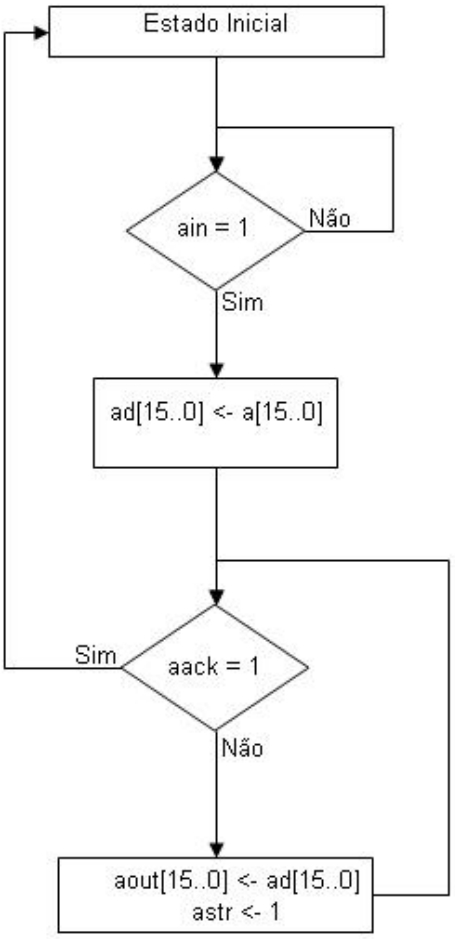


Figura 5.14: ASM chart do operador INDATA.

Similarmente ao operador *INDATA*, o operador *AOUT* é responsável em sincronizar os dados de saída do grafo. O bloco esquemático do operador *AOUT* é apresentado na Figura 5.15.

Os sinais *a*, *astr* e *clkin* são os sinais de entrada e os sinais *z*, *zstr* e *aack* são os sinais de saída. O sinal *a* é o sinal de entrada de dados; *astr* e *aack* são respectivamente os sinais de *strobe*

Figura 5.15: Bloco esquemático do operador AOUT.

e reconhecimento para o sinal de entrada de dados. O sinal *z* é o sinal de saída de dados; *zstr* é o sinal de *strobe* para o sinal de saída de dados. A Figura 5.16 apresenta o diagrama de estados do operador *AOUT*.

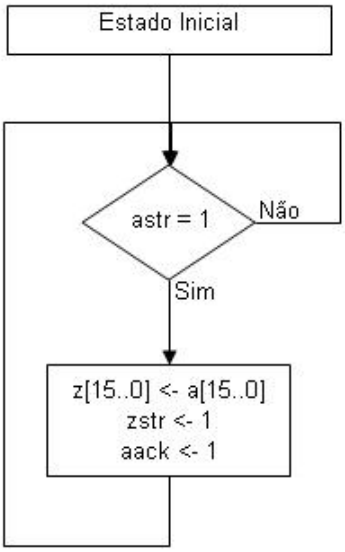


Figura 5.16: ASM chart do operador AOUT.

No caso de grafos que utilizam *loops*, o operador *ITMAN* é inserido para garantir que o operador *Merge Deterministic* possua todos os dados de entrada necessários para a primeira execução. O bloco esquemático do operador *ITMAN* é apresentado na Figura 5.17.

Os sinais *v1*, *v2*, *v1in*, *v2in*, *intsgnack*, *aack*, *back* e *clkin* são os sinais de entrada de dados e os sinais *a*, *b*, *astr*, *bstr*, *v1ack*, *v2ack*, *intsgnstr* e *intsgn* são os sinais de saída. Os sinais *v1* e *v2* são os sinais de entrada de dados; *v1in* e *v2in* são os sinais de *strobe* para os sinais de entrada de dados; *v1ack* e *v2ack* são os sinais de reconhecimento para o sinal de entrada de dados. Os sinais *a* e *b* são os sinais de saída de dados; *astr* e *bstr* são os sinais de *strobe* para o sinal de saída de dados; *aack* e *back* são os sinais de reconhecimento para os sinais de saída de dados. O sinal *intsgn* é o sinal de controle do operador; *intsgnstr* e *intsgnack* são respectivamente os sinais de

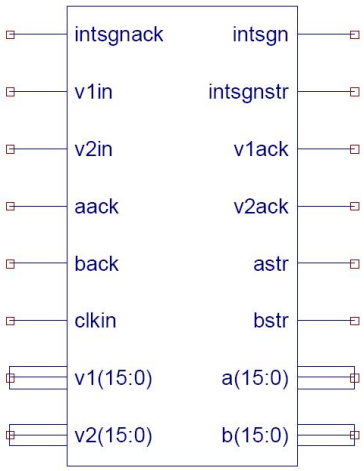


Figura 5.17: Bloco esquemático do operador ITMAN.

strobe e reconhecimento para o sinal de controle. A Figura 5.18 apresenta o diagrama de estados do operador *ITMAN*.

Enquanto todos operadores foram implementados utilizando a linguagem *VHDL*, os grafos a fluxo de dados foram implementados utilizando diagramas esquemáticos por permitirem uma representação visual do grafo.

Os grafos implementados expressam decisões e iterações (*loops*).

Para expressar decisões os grafos a fluxo de dados implementados representam os comandos *If-Else* e *Switch*. Na Figura 5.19 é descrito o grafo para o comando *If-Else* e sua implementação é apresentada na Figura 5.20, onde é possível verificar a utilização dos operadores *INDATA* e *OUTDATA* para sincronizar os dados de entrada e saída do grafo.

A Figura 5.21 apresenta as simulações realizadas para o grafo a fluxo de dados do comando *If-Else*. Na Tabela 5.1 são descritos os conjuntos de dados de entrada e saída apresentados na Figura 5.21.

Tabela 5.1: Dados utilizados na simulação apresentada na Figura 5.21.

Entradas					Saída
a	b	c	d	x	z
2	3	5	1	2	5
3	4	9	9	0	0

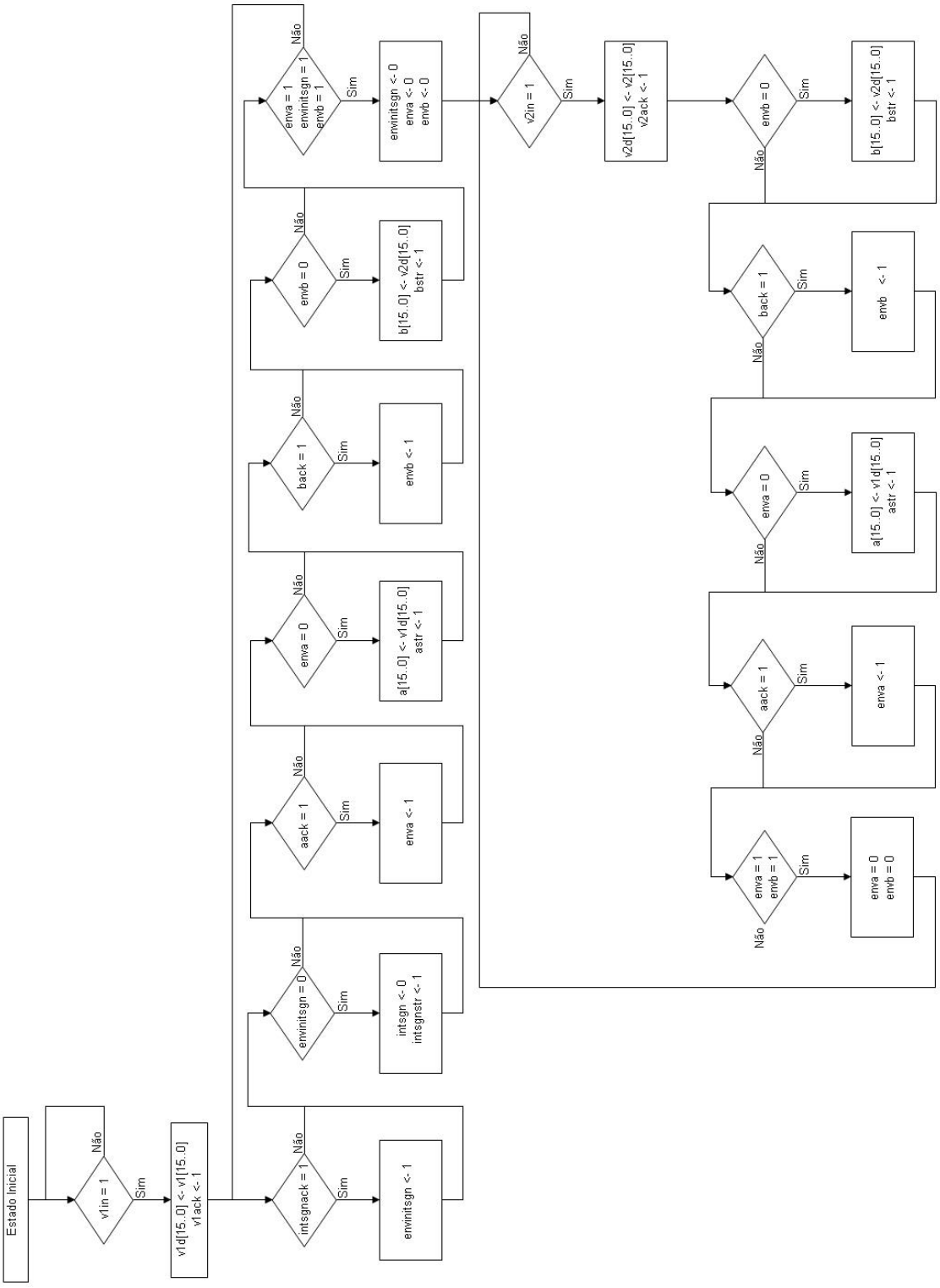


Figura 5.18: ASM chart do operador ITMAN.

Podemos verificar na Tabela 5.1 que no primeiro conjunto de dados, para x igual a 2 o resultado obtido por z é 5 (soma dos dados de entrada a e b). No segundo conjunto de dados, para x igual a 0 o resultado obtido por z é 0 (subtração dos dados de entrada c e d).

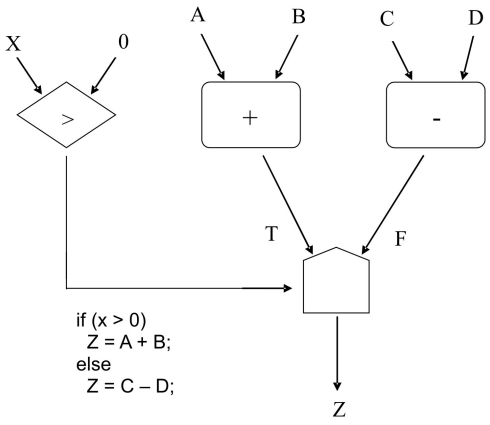


Figura 5.19: Grafo a fluxo de dados para o comando If-Else.

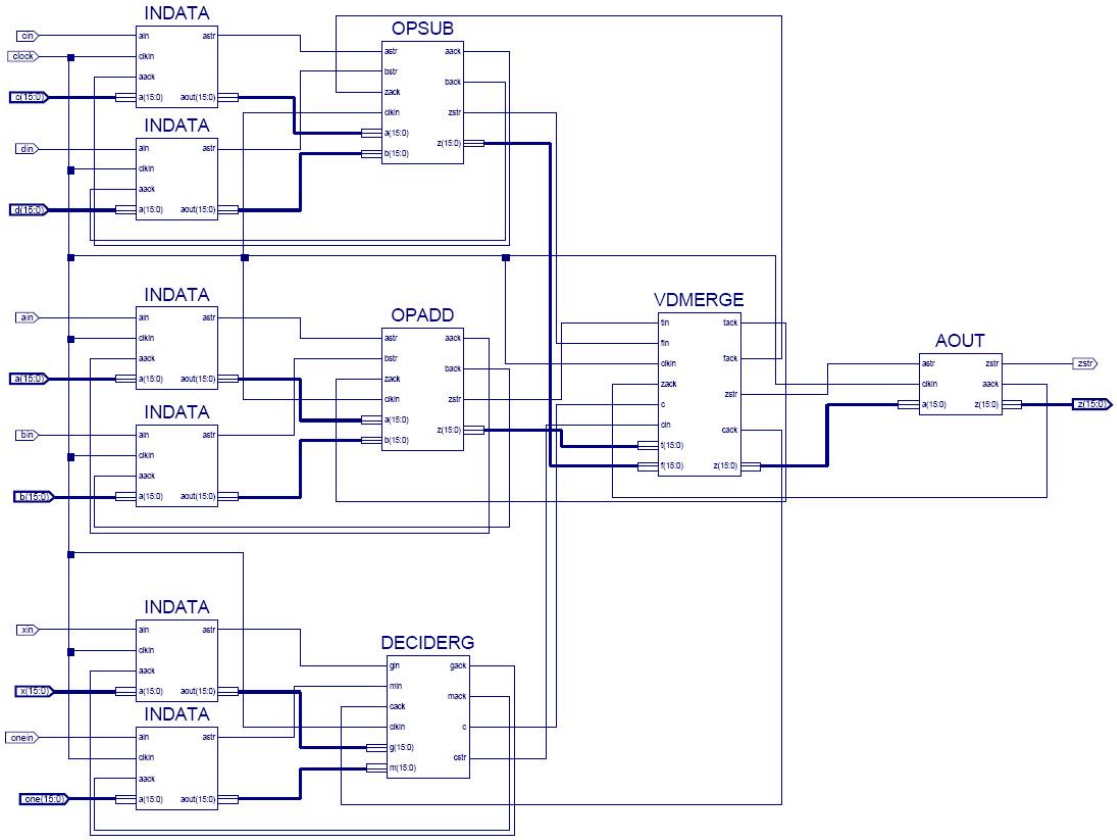


Figura 5.20: Implementação do grafo a fluxo de dados para o comando If-Else.

A Figura 5.22 descreve o grafo a fluxo de dados do comando *Switch*, onde uma variável é testada em relação a diversos valores pré-estabelecidos. A implementação do grafo para o comando *Switch* é apresentada no Apêndice A.

A Figura 5.23 apresenta a simulação do grafo a fluxo de dados implementado para o comando *Switch*. Na Tabela 5.2 são descritos os conjuntos de dados de entrada e saída apresentados na Figura 5.21.

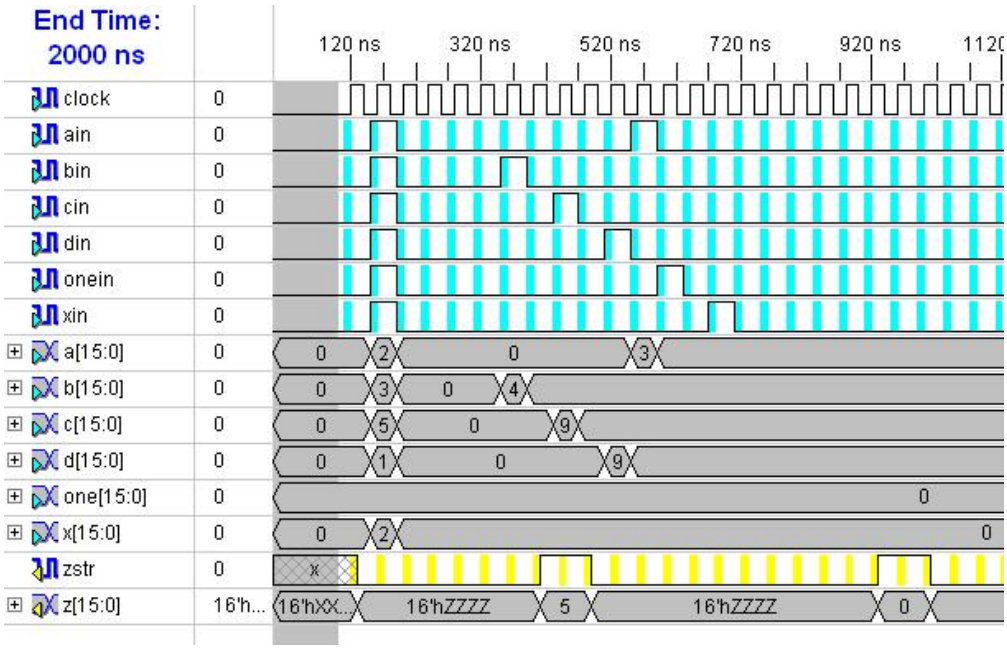


Figura 5.21: Simulação do grafo a fluxo de dados implementado para o comando If-Else.

Tabela 5.2: Dados utilizados na simulação apresentada na Figura 5.23.

Entradas											Saída
brch0	brch1	brch2	brch3	case0	case1	case2	case3	defaultbrch	z	x	z
5	3	4	6	0	1	2	3	5	5	0	5
5	3	4	6	0	1	2	3	5	5	1	3
5	3	4	6	0	1	2	3	5	5	2	4
5	3	4	6	0	1	2	3	5	5	4	10

Podemos verificar na Tabela 5.2 que o valor inicial de z é 5, a entrada x assume os seguintes valores 0, 1, 2, e 4. O resultado da saída z é igual a 5, 3, 4 e 10 respectivamente.

O comando *While* é usado para repetir a execução de um bloco de comandos enquanto a condição de teste seja verdadeira. Como a condição de teste é feita no começo da iteração, o bloco pode ocasionalmente não executar. A Figura 5.24 apresenta o grafo a fluxo de dados para o comando *While*.

A implementação do grafo para o comando *While* é apresentada no Apêndice A.

O grafo a fluxo de dados do comando *For* é uma generalização do grafo do comando *While*. A Figura 5.25 mostra o grafo a fluxo de dados do comando *For*. Similarmente ao grafo a fluxo de dados do comando *While*, a condição do teste é executada no começo da iteração. Por esta razão a estrutura geral de ambos os grafos são praticamente as mesmas. A diferença entre eles consiste no

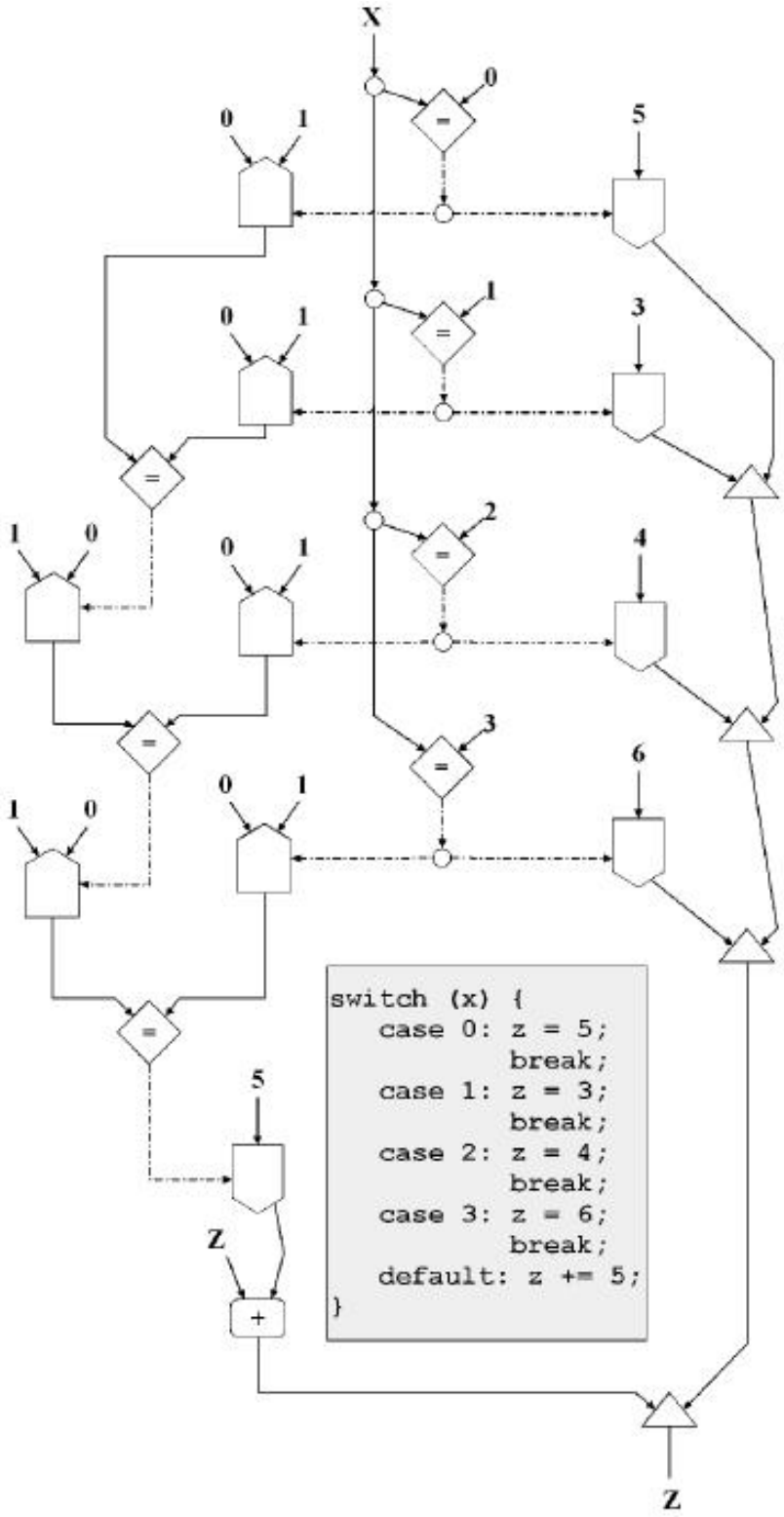


Figura 5.22: Grafo a fluxo de dados para o comando Switch.

incremento do contador (i++) do comando *For*, que é a última operação a ser executada dentro do laço. A implementação do grafo para o comando *For* é apresentada no Apêndice A.

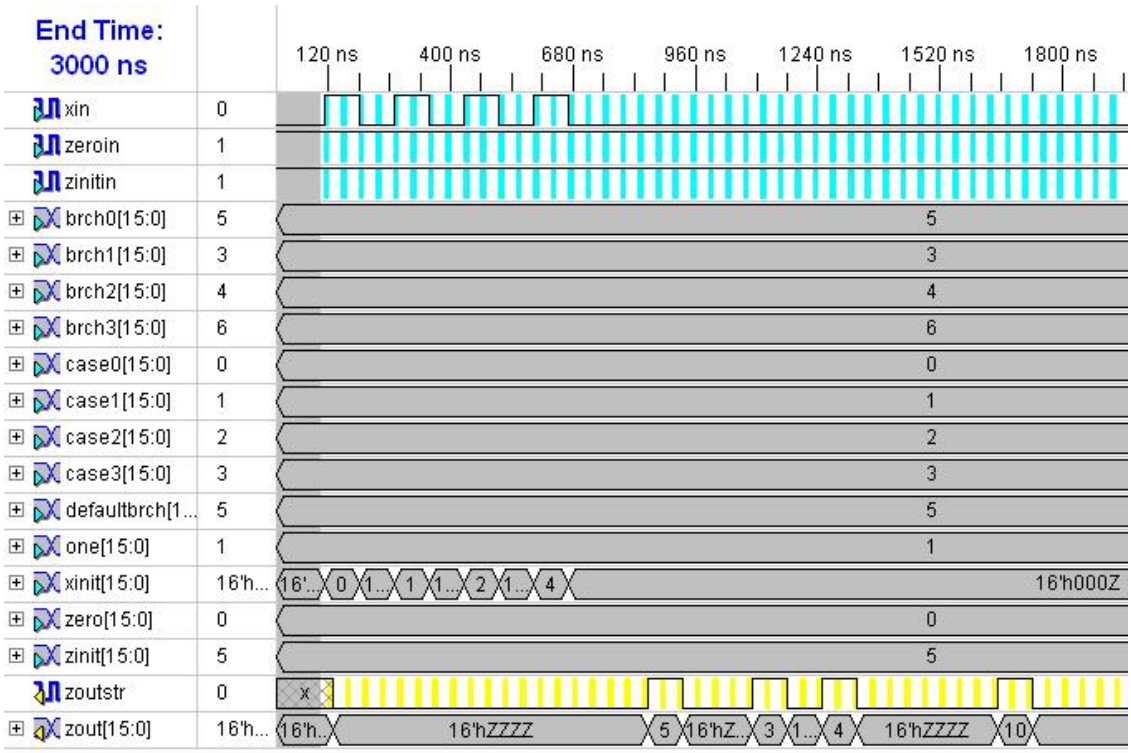


Figura 5.23: Simulação do grafo a fluxo de dados implementado para o comando Switch.

Na Figura 5.26 é apresentado o grafo a fluxo de dados do comando *Do-While*, onde a condição do teste é executado no final da iteração. Por este motivo, pelo menos uma vez os blocos de comandos dentro do laço são executados. Isto faz o grafo do comando *Do-While* ligeiramente diferente dos grafos dos comandos *While* e *For*. A implementação do grafo a fluxo de dados para o comando *Do-While* é apresentada no Apêndice A.

As Figuras 5.27, 5.28 e 5.29 apresentam os resultados obtidos com as simulações dos grafos implementados que expressam os comandos *While*, *For* e *Do-While*. A Tabela 5.3 apresenta os conjuntos de dados utilizados nas simulações. O valor inicial de z é 2 (*zinit*); o valor de n é 3; o valor para o contador i é 0 (*iinit*) e o valor de incremento do contador é 1 (*incr*). O resultado final obtido por z é 8, 5 e 8 respectivamente para os grafos *While*, *For* e *Do-While*.

Tabela 5.3: Dados utilizados nas simulações apresentadas nas Figuras 5.27, 5.28 e 5.29.

	Entradas				Saída
	iinit	incr	zinit	n	z
While	0	1	2	3	8
For	0	1	2	3	5
Do-While	0	1	2	3	8

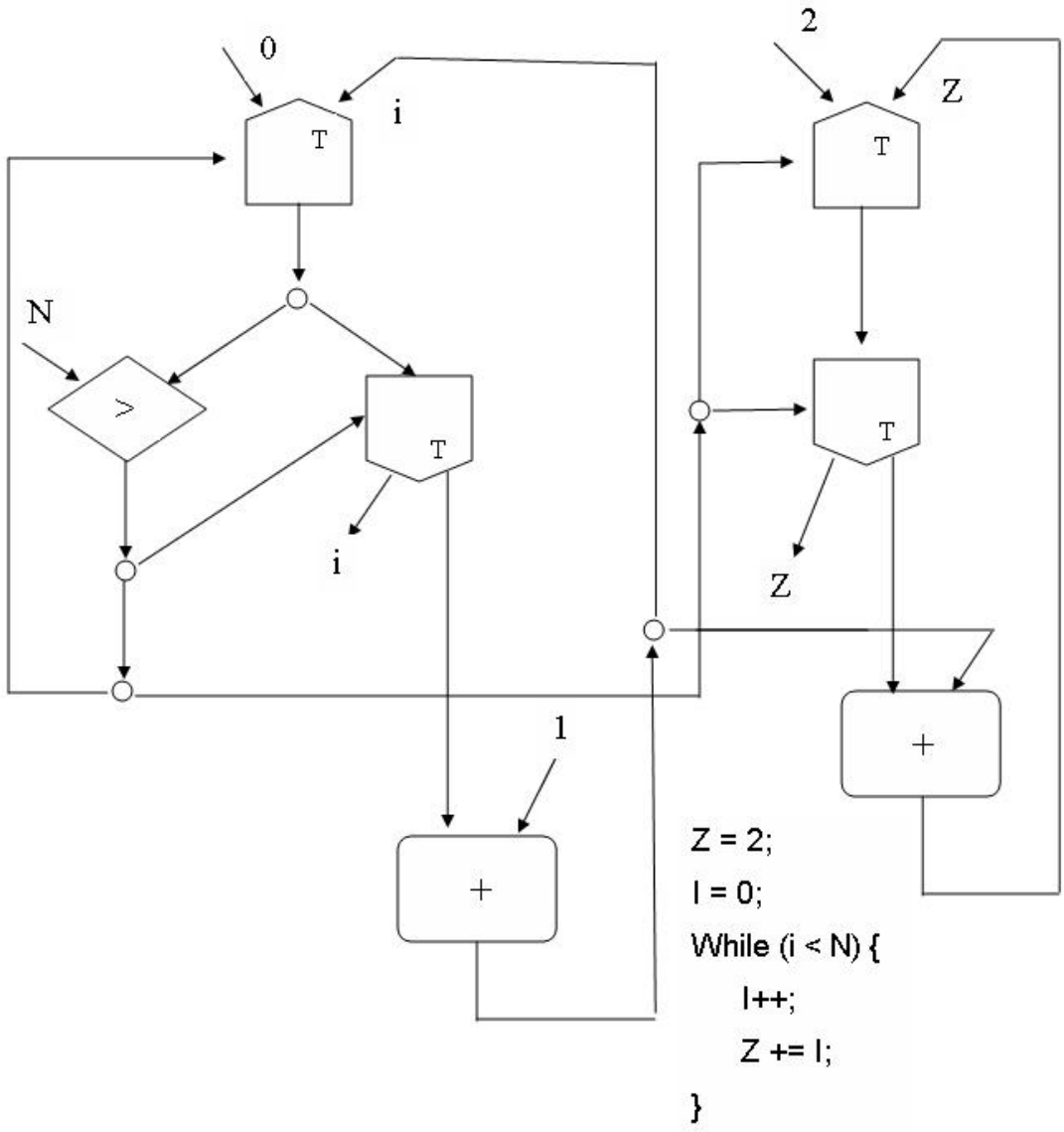


Figura 5.24: Grafo a fluxo de dados para o comando While.

A Figura 5.30 apresenta o resultado da execução dos programas escritos em linguagem C dos quais os grafos a fluxo de dados dos comandos *While*, *For* e *Do-While* foram convertidos.

5.2.1 Implementação da Sequência de Fibonacci

A sequência de *FIbonacci* é uma construção iterativa, onde cada elemento é formado pela soma dos dois elementos anteriores, com execeção dos dois primeiros elementos que são respectivamente 0 e 1.

A Figura 5.31 apresenta o grafo a fluxo de dados para a Sequência de *Fibonacci*. A implementação do grafo é apresentada no Apêndice A.

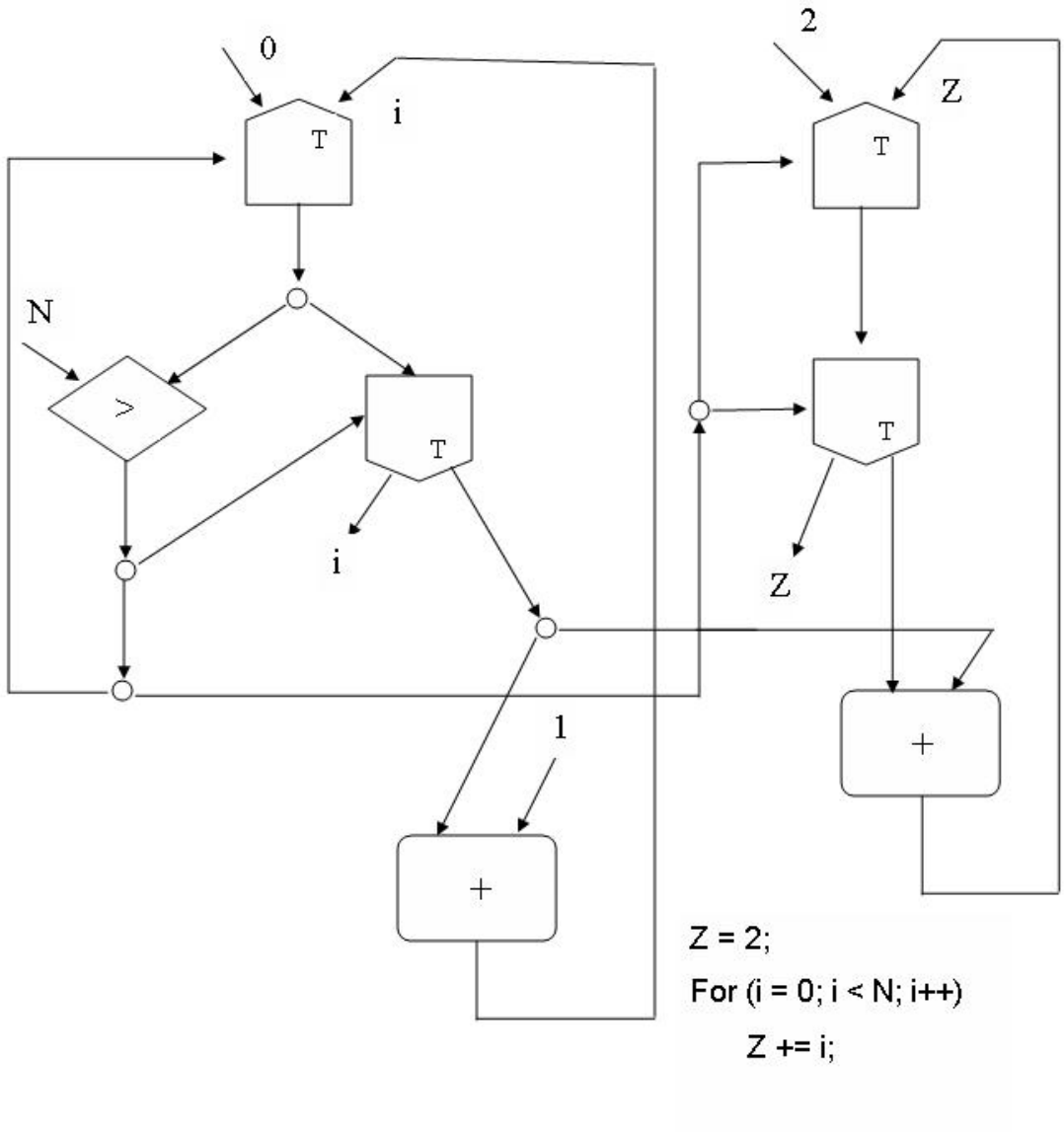


Figura 5.25: Grafo a fluxo de dados para o comando For.

A Figura 5.32 apresenta a simulação do grafo a fluxo de dados implementado para a sequência de *Fibonacci* e a Tabela 5.4 é descrito o conjunto de dados utilizado na simulação, onde o valor dos dois primeiros elementos são 0 (*a*) e 1 (*b*); o valor de *n* é 3; o valor para o contador *i* é 0 (*iinit*) e o valor de incremento do contador é 1 (*incr*). O resultado do cálculo da sequência de *Fibonacci* é 987, que corresponde ao décimo sexto número da sequência.

Tabela 5.4: Dados utilizados na simulação apresentada na Figura 5.32.

Entradas					Saída
a	b	iinit	incr	n	fibonacci
0	1	0	1	16	987

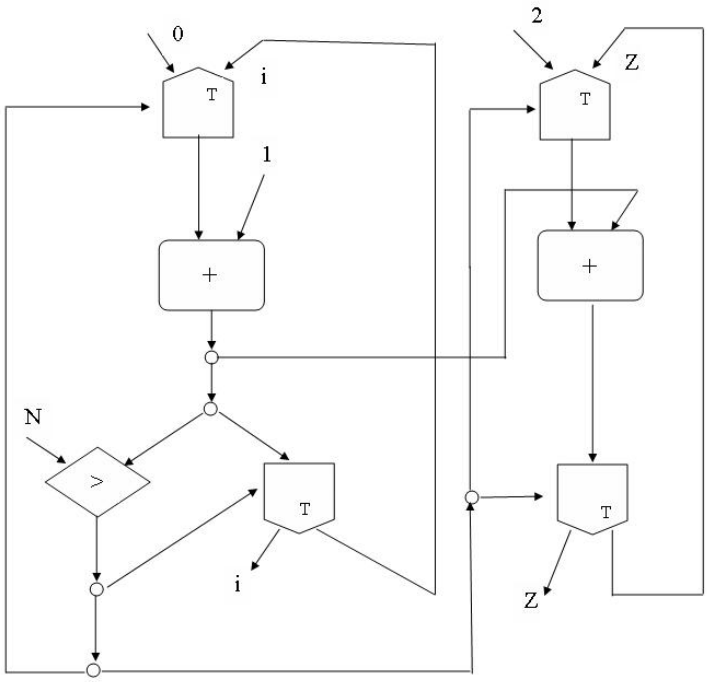


Figura 5.26: Grafo a fluxo de dados para o comando Do-While.

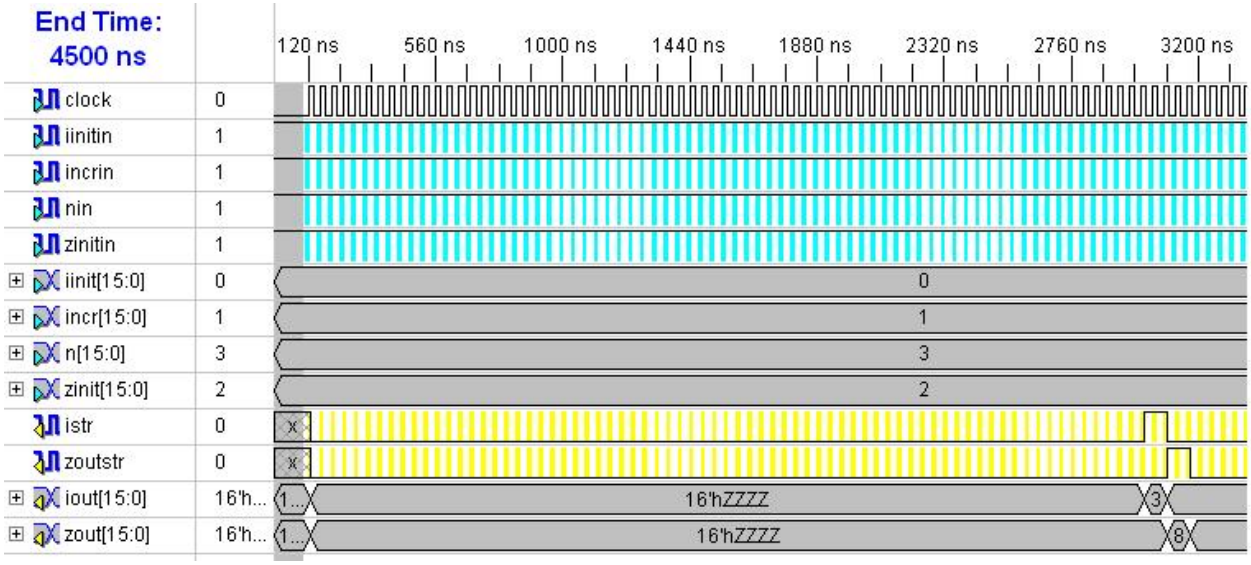


Figura 5.27: Simulação do grafo a fluxo de dados implementado para o comando While.

A Figura 5.33 apresenta o resultado da execução do programa escrito em C do qual o grafo a fluxo de dados para a sequência de *Fibonacci* foi convertido.

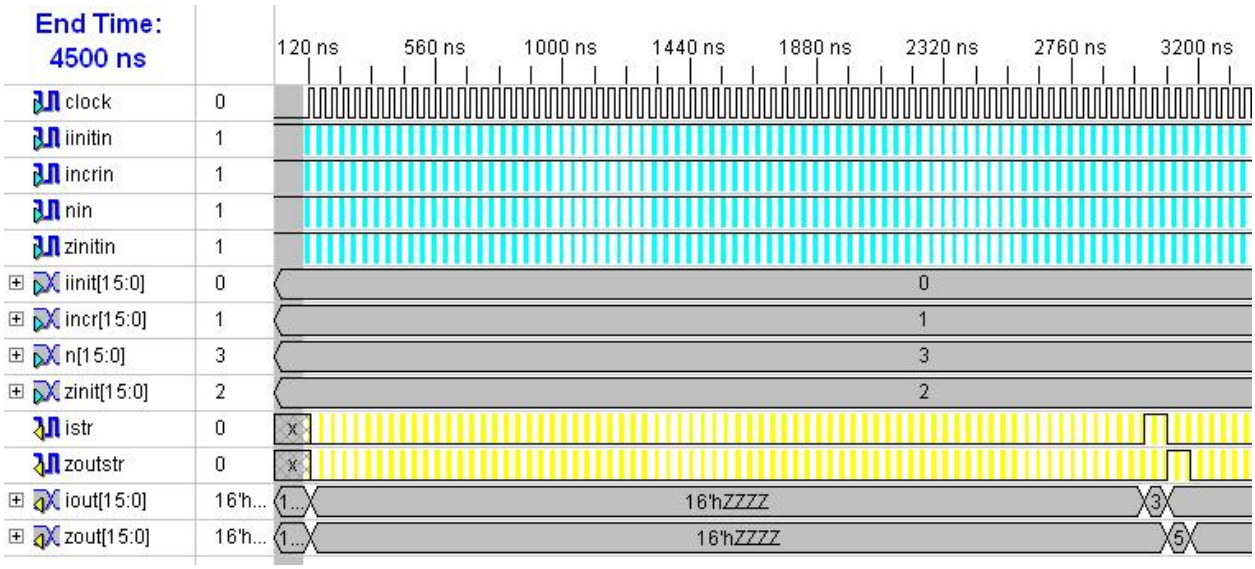


Figura 5.28: Simulação do grafo a fluxo de dados implementado para o comando For.

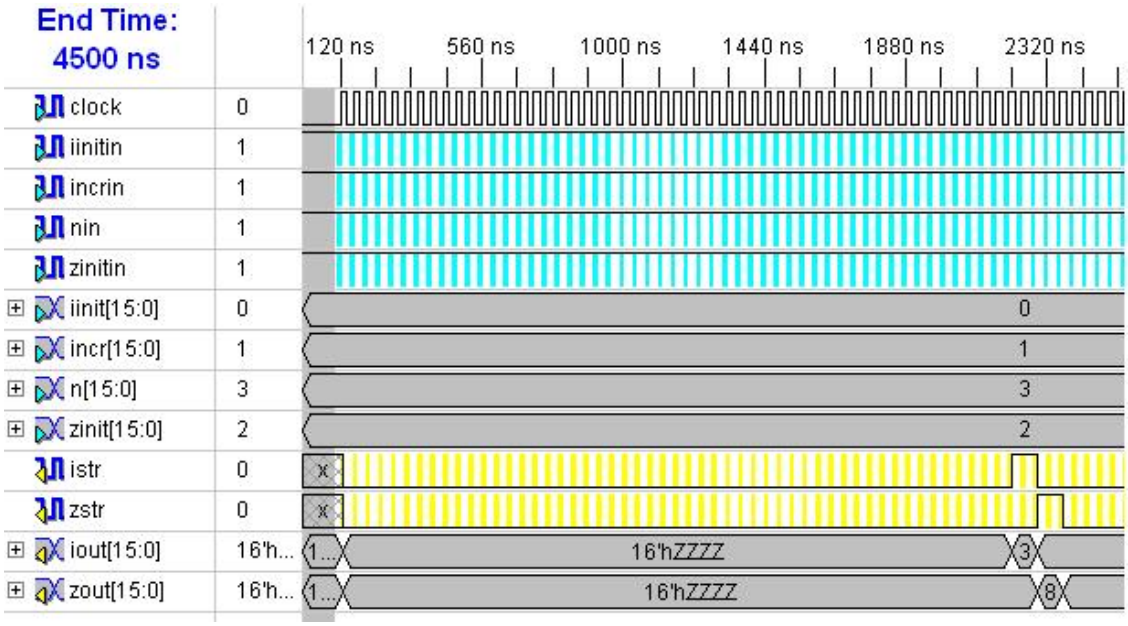


Figura 5.29: Simulação do grafo a fluxo de dados implementado para o comando Do-While.

Figura 5.30: Resultado da execução de programas em C dos comandos While, For e Do-While.

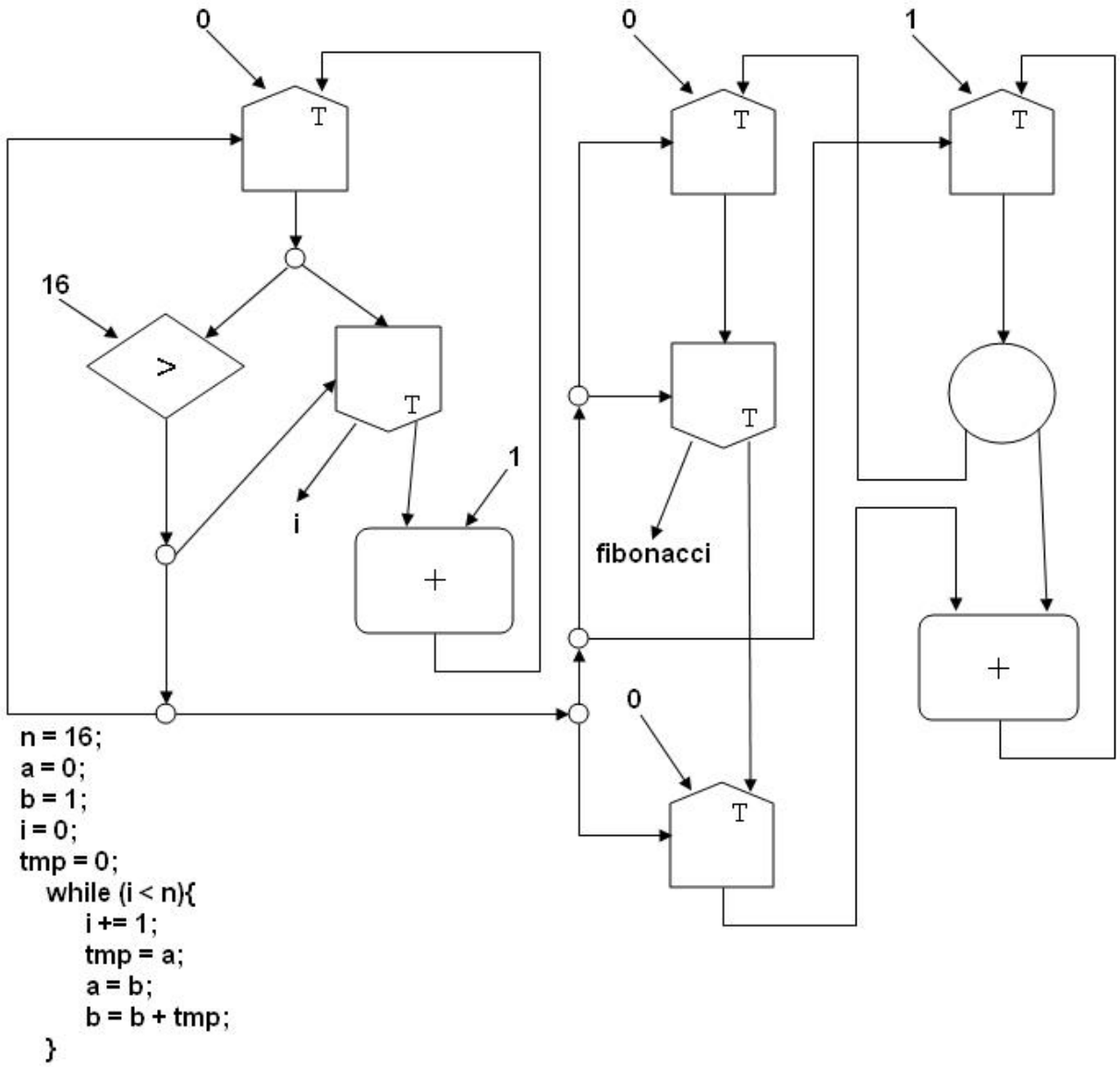


Figura 5.31: Grafo a fluxo de dados para a Sequência de Fibonacci.

A Tabela 5.5 mostram os recursos de *hardware* gastos para implementar o grafo a fluxo de dados para a sequência de *Fibonacci*, no FPGA Virtex II-pro xc2vp2-5-fg256.

Tabela 5.5: Recursos gastos para implementação do grafo para a Sequência de Fibonacci.

Elementos do dispositivo	Quantidade	Porcentagem de recurso utilizada
Número de slices	701 de 1408	49
Número de slice flip flops	1172 de 2816	41
Número de Luts de 4 entradas	936 de 2816	33
Número bounded IOs	120 de 140	85

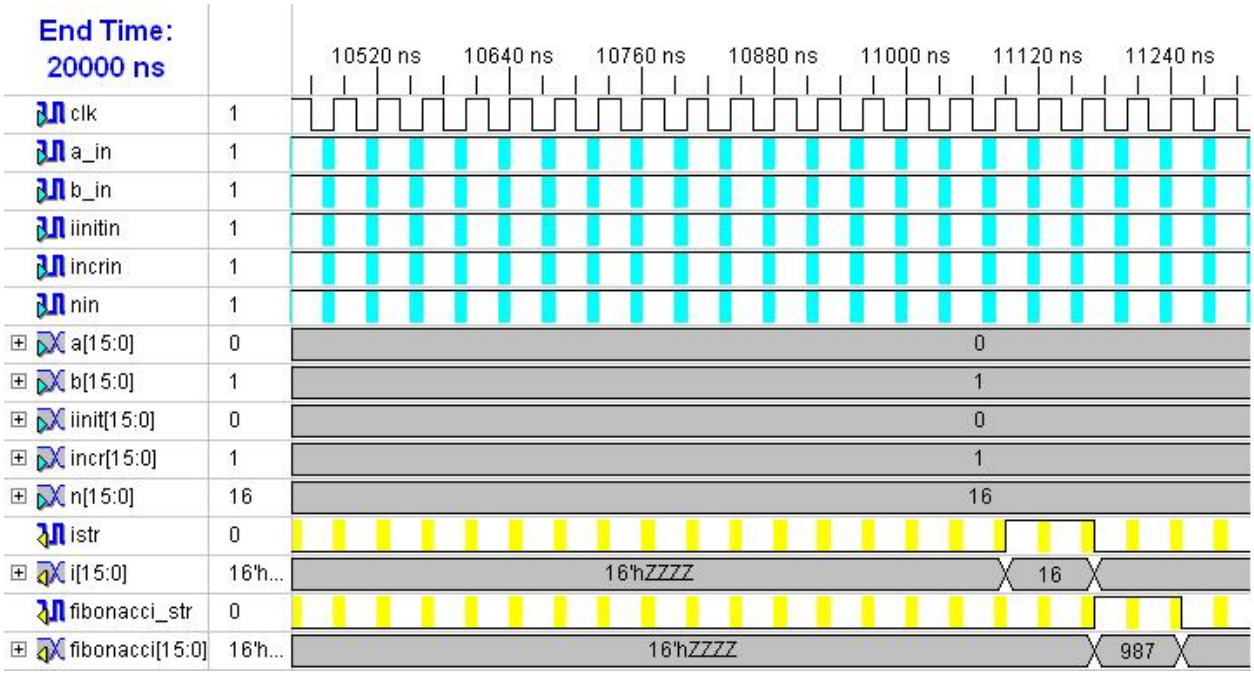


Figura 5.32: Simulação do grafo a fluxo de dados implementado para a Sequência de Fibonacci.

Figura 5.33: Resultado da execução do programa em C para a Sequência de Fibonacci.

CAPÍTULO

6

Conclusão

A partir dos objetivos deste trabalho de mestrado, que eram basicamente estudar e implementar os operadores propostos no projeto *ChipCflow*; implementar manualmente grafos a fluxo de dados estáticos com o objetivo de validar os operadores implementados, dominar os conceitos de programação a fluxo de dados e na conversão de algoritmos em linguagens de alto nível para grafos que foram executados diretamente em *hardware*, são descritos a seguir os resultados obtidos com o desenvolvimento do projeto.

Inicialmente no projeto foi realizada uma descrição sobre computação reconfigurável, apresentando a evolução dos circuitos digitais, as características de reconfigurabilidade dos *FPGAs*, dando destaque aos seus modelos de programação, plataforma e arquitetura.

Em seguida, foram apresentadas as características das máquinas a fluxo de dados, a linguagem original para programas a fluxo de dados, a máquina que foi proposta para execução da linguagem, sua arquitetura, considerando máquinas a fluxo de dados estáticas e máquinas a fluxo de dados dinâmicas, foram apresentadas também as arquiteturas de algumas máquinas a fluxo de dados existentes.

A partir dessa descrição, foi apresentado o projeto *ChipCflow* em detalhes, apresentando os diferentes módulos em desenvolvimento e em particular a contribuição deste trabalho dentro do projeto *ChipCflow*.

Inicialmente foram apresentados a implementação de cada operador, com a sua representação em esquemático e em diagrama de estados. Em seguida, diferentes comandos em linguagem de alto nível foram apresentados em forma de grafo a fluxo de dados e suas respectivas implementações, validando assim diferentes comandos em alto nível que, permitem implementar diferentes programas escritos em alto nível, mas sendo executados direto em *hardware*. Finalmente foi descrita a sequência de *Fibonacci*, com uma implementação originalmente escrita em linguagem de alto nível convertida para grafo a fluxo de dados e executada diretamente em hardware.

Como o objetivo deste trabalho foi a prova de conceitos dos operadores a fluxo de dados, nenhuma consideração foi feita em relação a desempenho dos algoritmos a fluxo de dados implementados, que será tratado em outra parte do projeto

A síntese de arquiteturas a fluxo de dados é uma alternativa poderosa para a síntese de circuitos digitais de alto desempenho, pois a computação depende do processamento dos dados e não do fluxo de controle existente no modelo *von Neumann*.

Este trabalho é uma primeira contribuição para o projeto *ChipCflow* fornecendo os recursos necessários para programação a fluxo de dados para os demais trabalhos de pesquisa que se encontram em desenvolvimento.

Como continuidade do projeto, está a implementação de grafos a fluxo de dados dinâmicos, desenvolvendo um protocolo de comunicação entre os operadores, onde cada novo dado que circula pelo grafo contenha informações (*tags*) que o identifiquem, para que um circuito de *matching* de dados possa identificar os dados que são "parceiros"e consequentemente disparar a execução do operador. Outro ponto que pode ser explorado é a avaliação de desempenho de grafos a fluxo de dados em relação ao seu respectivo programa em linguagem de alto nível.

Referências Bibliográficas

ARVIND Dataflow: Passing the token. In: *ISCA Keynote*, 2005.

ARVIND; CULLER, D. E. Dataflow architectures. *Annual review of computer science*, v. vol. 1, p. 225–253, 1986.

ARVIND; KATHAIL, V. A multiple processor data flow machine that supports generalized procedures. In: *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, p. 291–302.

ASTOLFI, V. F. *Chipcflow - em hardware dinâmicamente reconfiguráve*. Dissertação de Mestrado, Qualificação apresentada ao ICMC/USP, 2007.

BOBDA, C. *Introduction to reconfigurable computing - architectures, algorithms,and applications*. Hardcover, 362 p., 2007.

BUDIU, M.; ARTIGAS, P.; GOLDSTEIN, S. Dataflow: A complement to superscalar. *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, p. 177 – 186, 2005.

BUDIU, M.; GOLDSTEIN, S. C. Compiling application-specific hardware. In: *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, London, UK: Springer-Verlag, 2002, p. 853–863.

CAPPELLI, A.; LODI, A.; MUCCI, C.; TOMA, M.; CAMPI, F. A dataflow control unit for c-to-configurable pipelines compilation flow. In: *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, p. 332–333.

CARDOSO, J. M. P. *Compilação de algoritmos em java para sistemas computacionais reconfiguráveis com exploração do paralelismo ao nível das operações*. Tese de Doutoramento, Universidade Técnica de Lisboa, 2000.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, v. 34, n. 2, p. 171–210, 2002.

COSTA, K. A. P. *Chipcflow - uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável - módulo de conversão c em grafo a fluxo de dados*. Tese de Doutoramento, Qualificação apresentada a EESC/USP, 2008.

CRAVEN, S. D. *Structured approach to dynamic computing application development*. Tese de Doutoramento, Bradley Department of Electrical and Computer Engineering, 2008.

DAVID, P.; SCOTT, T. *Practical fpga programming in c*. Prentice Hall PTR, 464 p., 2005.

DAVIS, A. L. The architecture and system method of ddm1: A recursively structured data driven machine. In: *ISCA '78: Proceedings of the 5th annual symposium on Computer architecture*, New York, NY, USA: ACM Press, 1978, p. 210–215.

DEHON, A. Dpga utilization and application. In: *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, New York, NY, USA: ACM, 1996, p. 115–121.

DENNIS, J. B.; MISUNAS, D. P. A preliminary architecture for a basic dataflow processor. In: *in Proceedings of the 2nd Annual Sysmposium on Computer Architecture*, 1974, p. 126–132.

GRAFE, V. G.; DAVIDSON, G. S.; HOCH, J. E.; HOLMES, V. P. The epsilon dataflow processor. In: *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, New York, NY, USA: ACM Press, 1989, p. 36–45.

GURD, J. R.; KIRKHAM, C. C.; WATSON, I. The manchester prototype dataflow computer. *Commun. ACM*, v. 28, n. 1, p. 34–52, 1985.

JUNIOR, F. S. *Chipcflow - validação e implementação do modelo de partição e protocolo de comunicação no grafo a fluxo de dados dinâmico*. Dissertação de Mestrado, Qualificação apresentada ao ICMC/USP, 2008.

KISHI, M.; YASUHARA, H.; KAWAMURA, Y. Dddp-a distributed data driven processor. In: *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1983, p. 236–242.

LOPES, J. J. *Chipcflow - uma ferramenta para execução de algoritmos utilizando o modelo a fluxo de dados dinâmico em hardware reconfigurável*. Tese de Doutoramento, Qualificação apresentada ao ICMC/USP, 2008.

MONTMINY, D. P.; BALDWIN, R. O.; WILLIAMS, P. D.; MULLINS, B. E. Using relocatable bitstreams for fault tolerance. In: *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, Washington, DC, USA: IEEE Computer Society, 2007, p. 701–708.

NAGARAJAN, R.; SANKARALINGAM, K.; BURGER, D.; KECKLER, S. A design space evaluation of grid processor architectures. *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, p. 40 – 51, 2001.

PAPADOPOULOS, G. M.; CULLER, D. E. Monsoon: an explicit token-store architecture. In: *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, New York, NY, USA: ACM Press, 1990, p. 82–91.

RIBEIRO, A. A. L. *Reconfigurabilidade dinâmica e remota de fpgas*. Dissertação de Mestrado, Universisdade de São Paulo, 2002.

SANCHES, L. B. *Chipcflow - partição e protocolo de comunicação no grafo a fluxo de dados dinâmico*. Dissertação de Mestrado, Qualificação apresentada ao ICMC/USP, 2007.

SANKARALINGAM, K.; NAGARAJAN, R.; LIU, H.; KIM, C.; HUH, J.; BURGER, D.; KECKLER, S. W.; MOORE, C. R. 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.

SATO, M.; KODAMA, Y.; SAKAI, S.; YAMAGUCHI, Y.; KOUMURA, Y. Thread-based programming for the em-4 hybrid dataflow machine. In: *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, New York, NY, USA: ACM Press, 1992, p. 146–155.

SHIMADA, T.; HIRAKI, K.; NISHIDA, K.; SEKIGUCHI, S. Evaluation of a prototype data flow processor of the sigma-1 for scientific computations. In: *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, p. 226–234.

SILVA, J. L. *Processamento a fluxo de dados tolerante a falhas em um computador paralelo*. Tese apresentada no programa de pós-graduação em engenharia elétrica, FEEC - Universidade de Campinas, 1992.

SOUZA, A. A. C. O. *Uma arquitetura sistólica para solução de sistemas lineares implementada com circuitos fpgas*. Dissertação de Mestrado, Universidade de Sâo Paulo, 1998.

SWANSON, S.; MICHELSON, K.; SCHWERIN, A.; OSKIN, M. Wavescalar. In: *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA: IEEE Computer Society, 2003, p. 291.

SWANSON, S.; SCHWERIN, A.; MERCALDI, M.; PETERSEN, A.; PUTNAM, A.; MICHELSON, K.; OSKIN, M.; EGGERS, S. J. The wavescalar architecture. *ACM Trans. Comput. Syst.*, v. 25, n. 2, p. 4, 2007.

TADIGOTLA, V.; SLIGER, L.; COMMURI, S. Fpga implementation of dynamic run-time behavior reconfiguration in robots. *Intelligent Control, 2006. IEEE International Symposium on*, p. 1220 – 1225, 2006.

TRELEAVEN, P. C.; BROWNBRIDGE, D. R.; HOPKINS, R. P. Data-driven and demand-driven computer architecture. *ACM Comput. Surv.*, v. 14, n. 1, p. 93–143, 1982.

UPEGUI, A.; SANCHEZ, E. Evolving hardware by dynamically reconfiguring xilinx fpgas. *International conference on evolvable systems (ICES)*, v. 3637, p. 56–65, 2005.

VEEN, A. H. Dataflow machine architecture. In: *ACM Computing Surveys*, 1986, p. 365–396.

ZHENG, W. H.; MARZWELL, N.; CHAU, S. In-system partial run-time reconfiguration for fault recovery applications on spacecrafts. *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, v. 4, p. 3952 – 3957, 2005.

APÊNDICE

A

Implementação de grafos a fluxo de dados

Listing A.1: Código VHDL para o operador Branch.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity BRANCH is
    Port ( clkin : in  STD_LOGIC;
                        v : in  STD_LOGIC_VECTOR (15 downto 0);
                        vin : in  STD_LOGIC;
                        vack : out  STD_LOGIC;
           t : out  STD_LOGIC_VECTOR (15 downto 0);
           tstr : out  STD_LOGIC;
           tack : in  STD_LOGIC;
```

```
            f : out  STD_LOGIC_VECTOR (15 downto 0);
            fstr : out  STD_LOGIC;
            fack : in  STD_LOGIC;
            c : in  STD_LOGIC;
            cin : in  STD_LOGIC;
            cack : out  STD_LOGIC);
end BRANCH;

architecture Behavioral of BRANCH is
type estados is(ramificar, enviot, enviof);
begin

        dados: process(clkin, vin, tack, fack, cin)
                variable estado: estados := ramificar;
                variable recv: STD_LOGIC := '0';
                variable recc: STD_LOGIC := '0';
                variable vd: STD_LOGIC_VECTOR(15 downto 0);
                variable tdt: STD_LOGIC_VECTOR(15 downto 0);
                variable fdt: STD_LOGIC_VECTOR(15 downto 0);
                variable cd: STD_LOGIC;
        begin
                if clkin'event and clkin = '1' then
                        t <= "ZZZZZZZZZZZZZZZZ";
                        f <= "ZZZZZZZZZZZZZZZZ";
                        tstr <= '0';
                        fstr <= '0';
                        vack <= '0';
                        cack <= '0';
                        case estado is
                        when ramificar =>
                                if vin = '1' then
                                        vd := v;
                                        vack <= '1';
                                        recv := '1';
                                end if;
```

```
                                if cin = '1' then
                                        cd := c;
                                        cack <= '1';
                                        recc := '1';
                                end if;
                                if recv = '1' and recc = '1' then
                                        if cd = '1' then
                                                tdt := vd;
                                                estado := enviot;
                                        else
                                                fdt := vd;
                                                estado := enviof;
                                        end if;
                                        recv := '0';
                                        recc := '0';
                                end if;
                        when enviot =>
                                if tack = '1' then
                                        estado := ramificar;
                                else
                                        t <= tdt;
                                        tstr <= '1';
                                end if;
                        when enviof =>
                                if fack = '1' then
                                        estado := ramificar;
                                else
                                        f <= fdt;
                                        fstr <= '1';
                                end if;
                        end case;
                end if;
        end process dados;
end Behavioral;
```

Listing A.2: Código VHDL para o operador Copy.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity COPY is
    Port ( v : in  STD_LOGIC_VECTOR (15 downto 0);
           vin : in  STD_LOGIC;
           vack : out  STD_LOGIC;
                       clkin : in  STD_LOGIC;
           a : out  STD_LOGIC_VECTOR (15 downto 0);
           astr : out  STD_LOGIC;
           aack : in  STD_LOGIC;
           b : out  STD_LOGIC_VECTOR (15 downto 0);
           bstr : out  STD_LOGIC;
           back : in  STD_LOGIC);
end COPY;

architecture Behavioral of COPY is
type estados is(inicio, envio);
begin
        dados: process(clkin, vin, aack, back)
                variable estado: estados := inicio;
                variable enva: STD_LOGIC := '0';
                variable envb: STD_LOGIC := '0';
                variable vd: STD_LOGIC_VECTOR(15 downto 0);
        begin
                if clkin'event and clkin = '1' then
                        a <= "ZZZZZZZZZZZZZZZZ";
                        b <= "ZZZZZZZZZZZZZZZZ";
                        astr <= '0';
                        bstr <= '0';
                        vack <= '0';
                        case estado is
```

```
                        when inicio =>
                                if vin = '1' then
                                        vd := v;
                                        vack <= '1';
                                        enva := '0';
                                        envb := '0';
                                        estado := envio;
                                end if;
                        when envio =>
                                if aack = '1' then
                                        enva := '1';
                                end if;
                                if enva = '0' then
                                        a <= vd;
                                        astr <= '1';
                                end if;
                                if back = '1' then
                                        envb := '1';
                                end if;
                                if envb = '0' then
                                        b <= vd;
                                        bstr <= '1';
                                end if;
                                if enva = '1' and envb = '1' then
                                        estado := inicio;
                                end if;
                        end case;
                end if;
        end process dados;
end Behavioral;
```

Listing A.3: Código VHDL para o operador Decider.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DECIDERG is
    Port ( g : in  STD_LOGIC_VECTOR (15 downto 0);
           gin : in  STD_LOGIC;
           gack : out  STD_LOGIC;
           m : in  STD_LOGIC_VECTOR (15 downto 0);
           min : in  STD_LOGIC;
           mack : out  STD_LOGIC;
           c : out  STD_LOGIC;
           cstr : out  STD_LOGIC;
           cack : in  STD_LOGIC;
                        clkin: in  STD_LOGIC);
end DECIDERG;

architecture Behavioral of DECIDERG is
type estados is(comp, envio);
begin
        dados: process(clkin, gin, min, cack)
                variable estado: estados := comp;
                variable recg: STD_LOGIC := '0';
                variable recm: STD_LOGIC := '0';
                variable gd: STD_LOGIC_VECTOR(15 downto 0);
                variable md: STD_LOGIC_VECTOR(15 downto 0);
                variable cd: STD_LOGIC;
        begin
                if clkin'event and clkin = '1' then
                        c <= '0';
                        cstr <= '0';
                        gack <= '0';
                        mack <= '0';
                        case estado is
                        when comp =>
                                if gin = '1' then
                                        gd := g;
```

```
                                        recg := '1';
                                        gack <= '1';
                                end if;
                                if min = '1' then
                                        md := m;
                                        recm := '1';
                                        mack <= '1';
                                end if;
                                if recg = '1' and recm = '1' then
                                        if gd > md then
                                                cd := '1';
                                        else
                                                cd := '0';
                                        end if;
                                        recg := '0';
                                        recm := '0';
                                        estado := envio;
                                end if;
                        when envio =>
                                if cack = '1' then
                                        estado := comp;
                                else
                                        c <= cd;
                                        cstr <= '1';
                                end if;
                        end case;
                end if;
        end process dados;
end Behavioral;
```

Listing A.4: Código VHDL para o operador Deterministic Merge.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity VDMERGE is
    Port ( t : in  STD_LOGIC_VECTOR (15 downto 0);
           tin : in  STD_LOGIC;
           f : in  STD_LOGIC_VECTOR (15 downto 0);
           fin : in  STD_LOGIC;
           tack : out  STD_LOGIC;
           fack : out  STD_LOGIC;
           clkin : in  STD_LOGIC;
           z : out  STD_LOGIC_VECTOR (15 downto 0);
           zstr : out  STD_LOGIC;
           zack : in  STD_LOGIC;
           c : in  STD_LOGIC;
           cin : in  STD_LOGIC;
           cack : out  STD_LOGIC);
end VDMERGE;

architecture Behavioral of VDMERGE is
type estados is(juncao, envio);
begin
        dados: process(clkin, tin, fin, zack, cin)
                variable estado: estados := juncao;
                variable rect: STD_LOGIC := '0';
                variable recf: STD_LOGIC := '0';
                variable recc: STD_LOGIC := '0';
                variable trd: STD_LOGIC_VECTOR(15 downto 0);
                variable fsd: STD_LOGIC_VECTOR(15 downto 0);
                variable zd: STD_LOGIC_VECTOR(15 downto 0);
                variable cd: STD_LOGIC;
        begin
                if clkin'event and clkin = '1' then
                        z <= "ZZZZZZZZZZZZZZZZ";
                        zstr <= '0';
                        tack <= '0';
                        fack <= '0';
```

```
cack <= '0';
case estado is
when juncao =>
        if tin = '1' then
                trd := t;
                tack <= '1';
                rect := '1';
        end if;
        if fin = '1' then
                fsd := f;
                fack <= '1';
                recf := '1';
        end if;
        if cin = '1' then
                cd := c;
                cack <= '1';
                recc := '1';
        end if;
        if rect = '1' and recf = '1' and recc = '1' then
                if cd = '1' then
                        zd := trd;
                else
                        zd := fsd;
                end if;
                rect := '0';
                recf := '0';
                recc := '0';
                estado := envio;
        end if;
when envio =>
        if zack = '1' then
                estado := juncao;
        else
                z <= zd;
                zstr <= '1';
```

```
                                end if;
                            end case;
                    end if;
            end process dados;
end Behavioral;
```

Listing A.5: Código VHDL para o operador Operator.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity OPADD is
    Port ( a : in  STD_LOGIC_VECTOR (15 downto 0);
           astr : in  STD_LOGIC;
           b : in  STD_LOGIC_VECTOR (15 downto 0);
           bstr : in  STD_LOGIC;
           zack : in  STD_LOGIC;
                      clkin: in  STD_LOGIC;
           aack : out  STD_LOGIC;
           back : out  STD_LOGIC;
           z : out  STD_LOGIC_VECTOR (15 downto 0);
           zstr : out  STD_LOGIC);
end OPADD;

architecture Behavioral of OPADD is
type estados is(soma, envio);
begin
        dados: process(clkin, astr, bstr, zack)
                variable estado: estados := soma;
                variable reca: STD_LOGIC := '0';
                variable recb: STD_LOGIC := '0';
                variable ad: STD_LOGIC_VECTOR(15 downto 0);
                variable bd: STD_LOGIC_VECTOR(15 downto 0);
                variable zd: STD_LOGIC_VECTOR(15 downto 0);
```

```
        begin
                if clkin'event and clkin = '1' then
                        z <= "ZZZZZZZZZZZZZZZ";
                        zstr <= '0';
                        aack <= '0';
                        back <= '0';
                        case estado is
                        when soma =>
                                if astr = '1' then
                                        ad := a;
                                        aack <= '1';
                                        reca := '1';
                                end if;
                                if bstr = '1' then
                                        bd := b;
                                        back <= '1';
                                        recb := '1';
                                end if;
                                if reca = '1' and recb = '1' then
                                        zd := ad + bd;
                                        reca := '0';
                                        recb := '0';
                                        estado := envio;
                                end if;
                        when envio =>
                                if zack = '1' then
                                        estado := soma;
                                else
                                        z <= zd;
                                        zstr <= '1';
                                end if;
                        end case;
                end if;
        end process dados;
end Behavioral;
```

Listing A.6: Código VHDL para o operador INDATA.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity INDATA is
    Port ( a : in  STD_LOGIC_VECTOR (15 downto 0);
           ain : in  STD_LOGIC;
           clkin : in  STD_LOGIC;
           aack : in  STD_LOGIC;
           astr : out  STD_LOGIC;
           aout : out  STD_LOGIC_VECTOR (15 downto 0));
end INDATA;

architecture Behavioral of INDATA is
type estados is(inicio, envio);
begin
        dados: process(clkin, ain, aack)
                variable estado: estados := inicio;
                variable ad: STD_LOGIC_VECTOR(15 downto 0);
        begin
                if clkin'event and clkin = '1' then
                        astr <= '0';
                        aout <= "ZZZZZZZZZZZZZZZZ";
                        case estado is
                        when inicio =>
                                if ain = '1' then
                                        ad := a;
                                        estado := envio;
                                end if;
                        when envio =>
                                if aack = '1' then
                                        estado := inicio;
                                else
```

```
                                        astr <= '1';
                                        aout <= ad;
                                end if;
                        end case;
                end if;
        end process dados;
end Behavioral;
```

Listing A.7: Código VHDL para o operador AOUT.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity AOUT is
    Port ( a : in  STD_LOGIC_VECTOR (15 downto 0);
           astr : in  STD_LOGIC;
           clkin : in  STD_LOGIC;
           z : out  STD_LOGIC_VECTOR (15 downto 0);
           zstr : out  STD_LOGIC;
           aack : out  STD_LOGIC);
end AOUT;

architecture Behavioral of AOUT is
begin
        dados: process(clkin, astr)
        begin
                if clkin'event and clkin = '1' then
                        z <= "ZZZZZZZZZZZZZZZZ";
                        zstr <= '0';
                        aack <= '0';
                        if astr = '1' then
                                z <= a;
                                zstr <= '1';
                                aack <= '1';
```

```
                        end if;
                end if;
        end process dados;
end Behavioral;
```

Listing A.8: Código VHDL para o operador ITMAN.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ITMAN is
    Port ( intsgn : out  STD_LOGIC;
           intsgnstr : out  STD_LOGIC;
           intsgnack : in  STD_LOGIC;
           v1 : in  STD_LOGIC_VECTOR (15 downto 0);
           v1in : in  STD_LOGIC;
           v1ack : out  STD_LOGIC;
           v2 : in  STD_LOGIC_VECTOR (15 downto 0);
           v2in : in  STD_LOGIC;
           v2ack : out  STD_LOGIC;
           a : out  STD_LOGIC_VECTOR (15 downto 0);
           astr : out  STD_LOGIC;
           aack : in  STD_LOGIC;
           b : out  STD_LOGIC_VECTOR (15 downto 0);
           bstr : out  STD_LOGIC;
           back : in  STD_LOGIC;
           clkin : in  STD_LOGIC);
end ITMAN;

architecture Behavioral of ITMAN is
type estados is(inicio, fst_iter, wait_v2, nth_iter);
begin
        dados: process(clkin, intsgnack, v1in, v2in, aack, back)
                variable estado: estados := inicio;
```

```
        variable v1d: STD_LOGIC_VECTOR (15 downto 0);
        variable v2d: STD_LOGIC_VECTOR (15 downto 0);
        variable envinitsgn: STD_LOGIC := '0';
        variable enva: STD_LOGIC := '0';
        variable envb: STD_LOGIC := '0';
begin
        if clkin'event and clkin = '1' then
                intsgn <= '0';
                intsgnstr <= '0';
                v1ack <= '0';
                v2ack <= '0';
                a <= "ZZZZZZZZZZZZZZZ";
                astr <= '0';
                b <= "ZZZZZZZZZZZZZZZ";
                bstr <= '0';
                case estado is
                when inicio =>
                        v2d := "0000000000000000"; -- um valor qualquer para v2
                        if v1in = '1' then
                                v1d := v1;
                                v1ack <= '1';
                                estado := fst_iter;
                        end if;
                when fst_iter =>
                        if intsgnack = '1' then
                                envinitsgn := '1';
                        end if;
                        if envinitsgn = '0' then
                                intsgn <= '0';
                                intsgnstr <= '1';
                        end if;
                        if aack = '1' then
                                enva := '1';
                        end if;
                        if enva = '0' then
```

```
                a <= v1d;
                astr <= '1';
        end if;
        if back = '1' then
                envb := '1';
        end if;
        if envb = '0' then
                b <= v2d;
                bstr <= '1';
        end if;
        if envinitsgn = '1' and enva = '1' and envb = '1' then
                envinitsgn := '0';
                enva := '0';
                envb := '0';
                estado := wait_v2;
        end if;
when nth_iter =>
        if aack = '1' then
                enva := '1';
        end if;
        if enva = '0' then
                a <= v1d;
                astr <= '1';
        end if;
        if back = '1' then
                envb := '1';
        end if;
        if envb = '0' then
                b <= v2d;
                bstr <= '1';
        end if;
        if enva = '1' and envb = '1' then
                enva := '0';
                envb := '0';
                estado := wait_v2;
```

```
                                end if;
                        when wait_v2 =>
                                if v2in = '1' then
                                        v2d := v2;
                                        v2ack <= '1';
                                        estado := nth_iter;
                                end if;
                        end case;
                end if;
        end process dados;
end Behavioral;
```

A implementação do grafo a fluxo de dados para o comando *Switch* pode ser observada na Figura A.1.

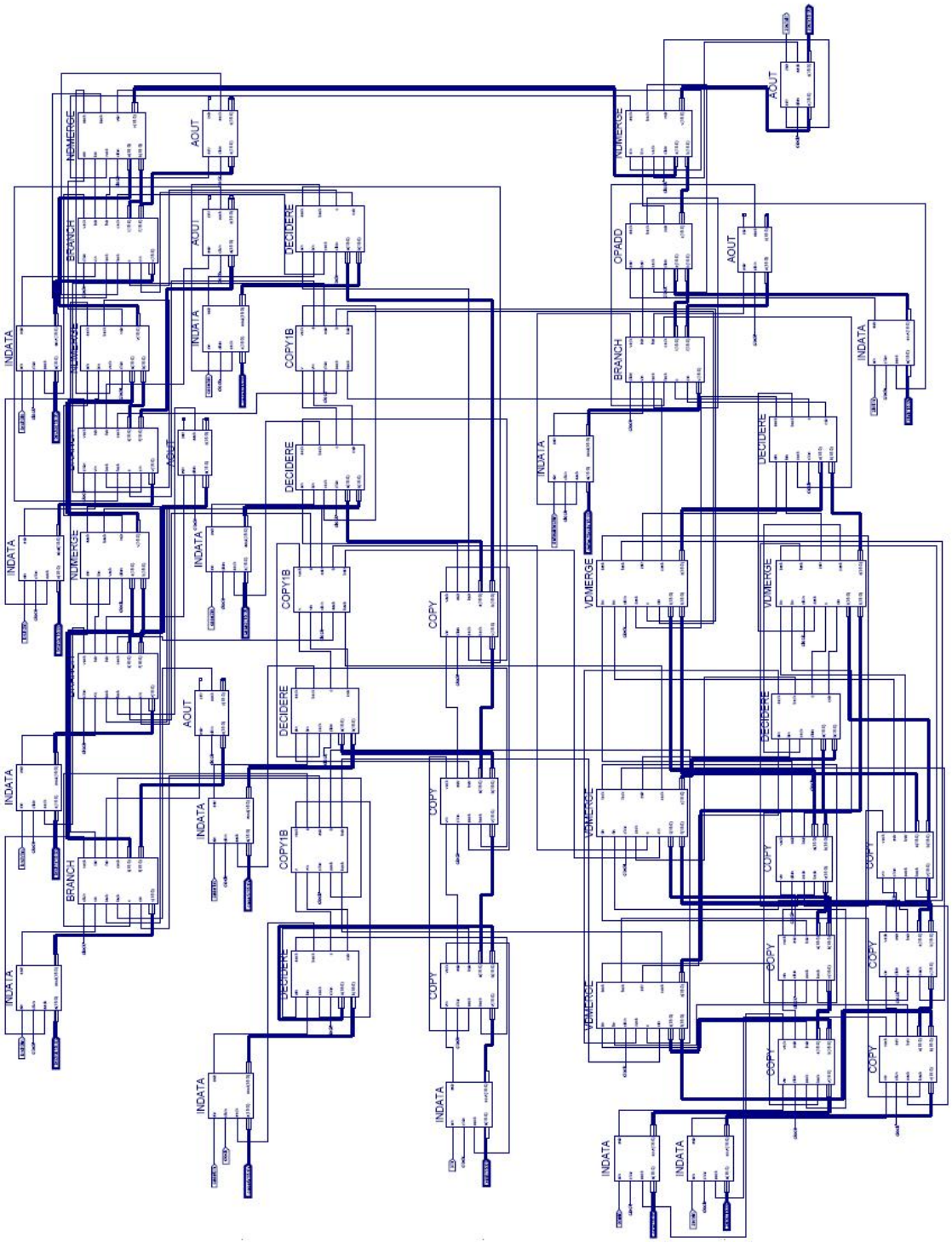


Figura A.1: Implementação do grafo a fluxo de dados para o comando Switch.

A implementação do grafo a fluxo de dados para o comando *While* é apresentada na Figura A.2.

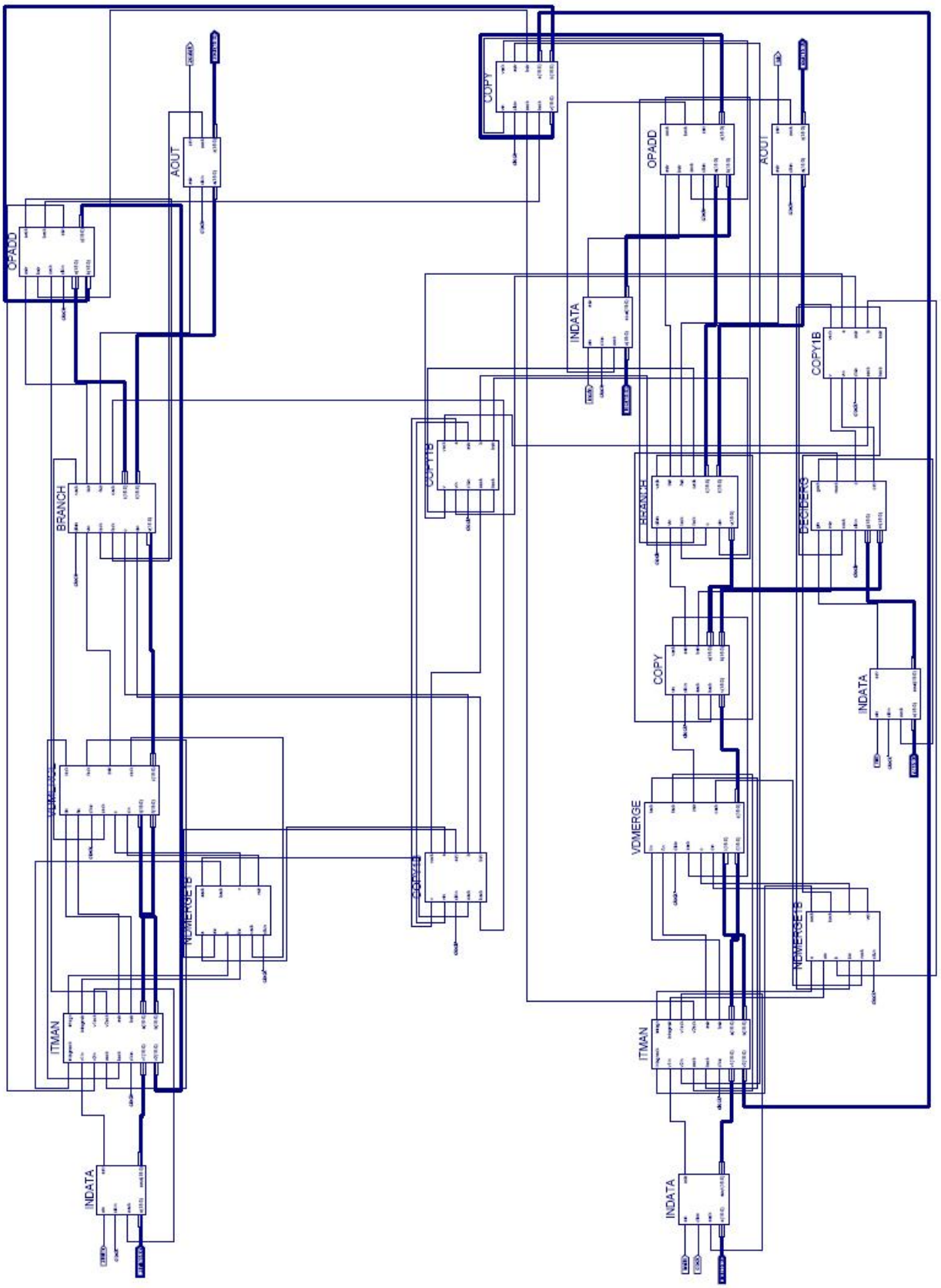


Figura A.2: Implementação do grafo a fluxo de dados para o comando While.

A implementação do grafo a fluxo de dados para o comando *For* é apresentada na Figura A.3.

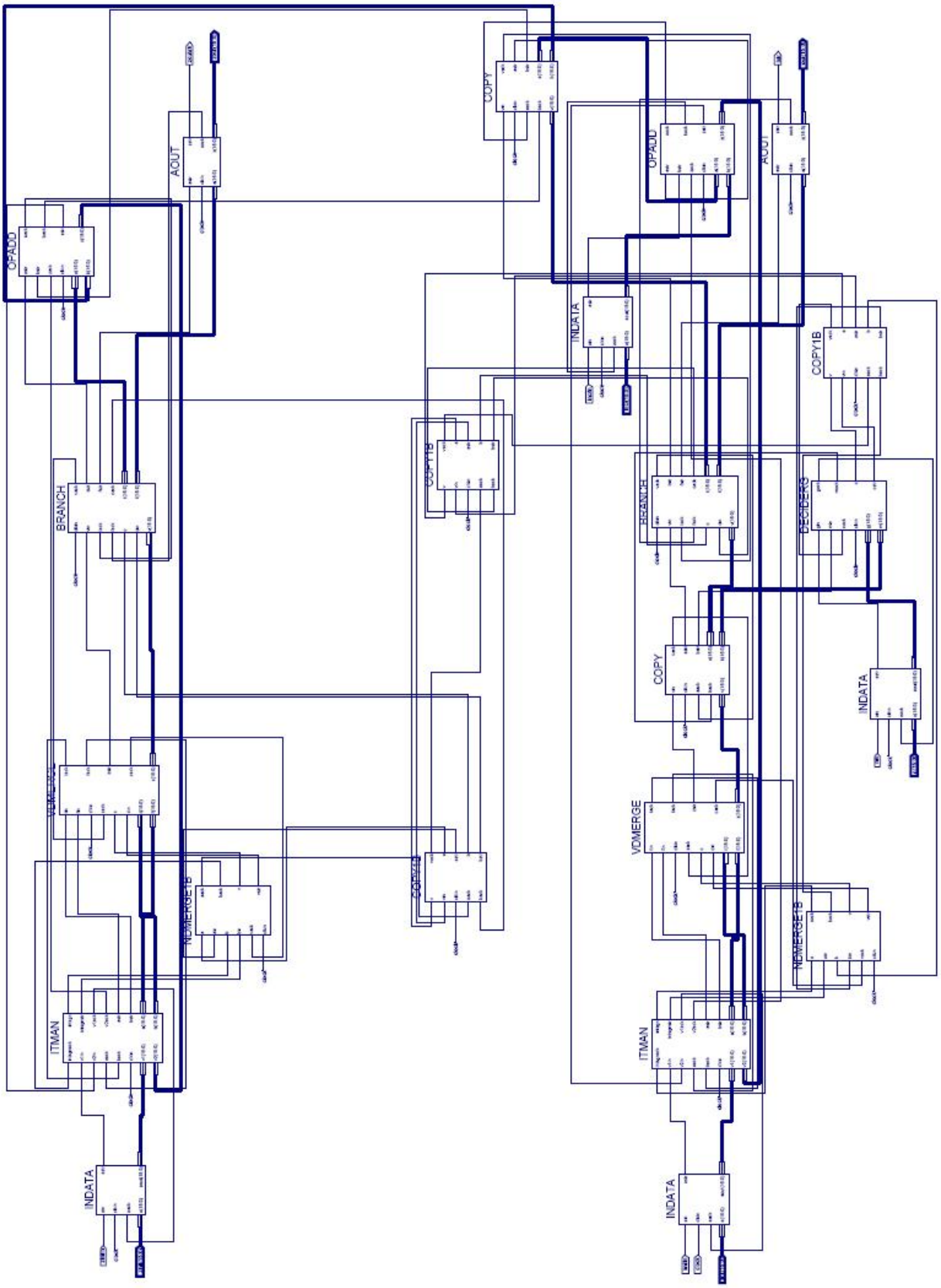


Figura A.3: Implementação do grafo a fluxo de dados do comando For.

A implementação do grafo a fluxo de dados para o comando *Do-While* é apresentada na Figura A.4.

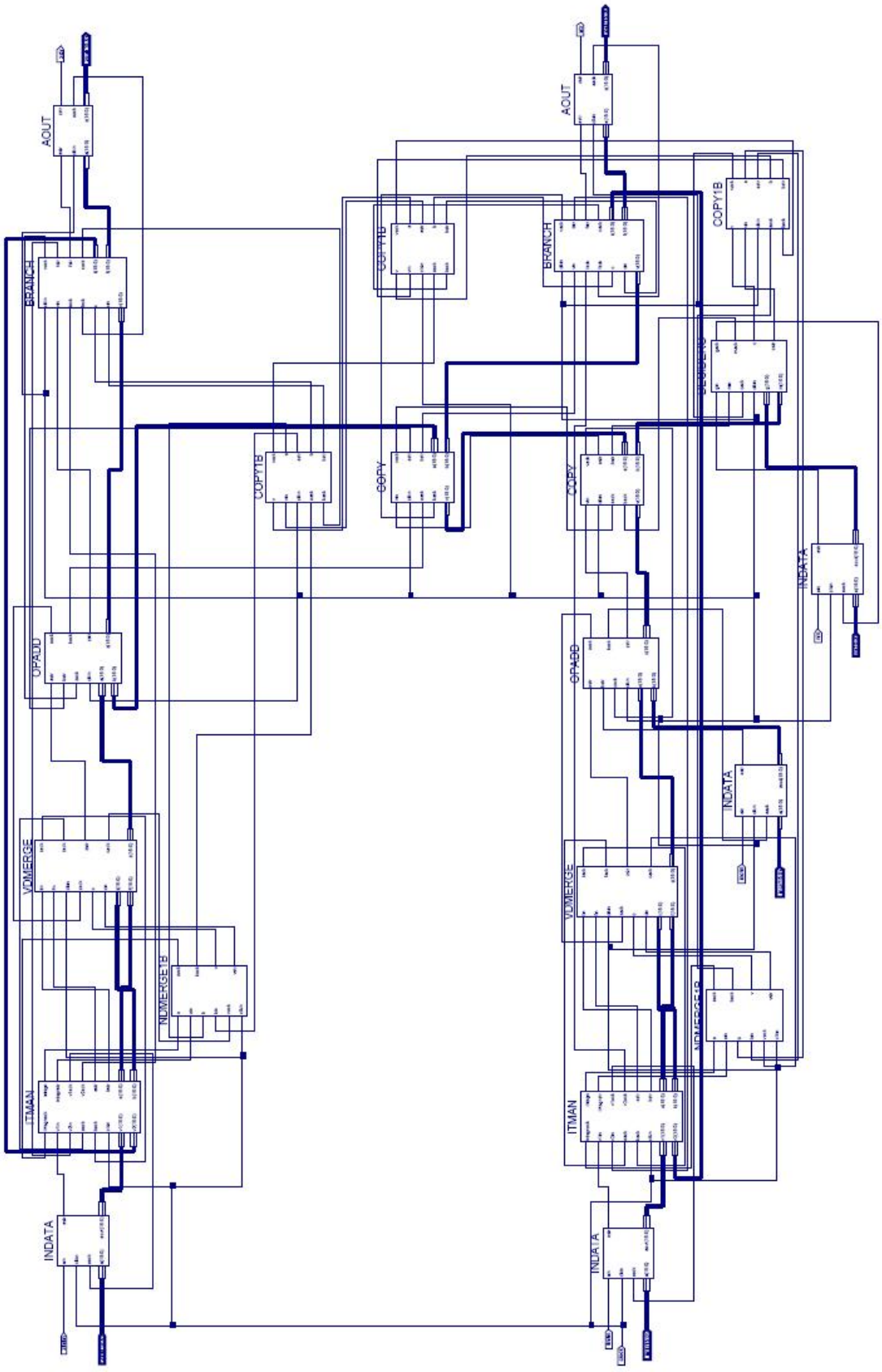


Figura A.4: Implementação do grafo a fluxo de dados para o comando Do-While.

A implementação do grafo a fluxo de dados para a sequência de *Fibonacci* está disponível na Figura A.5.

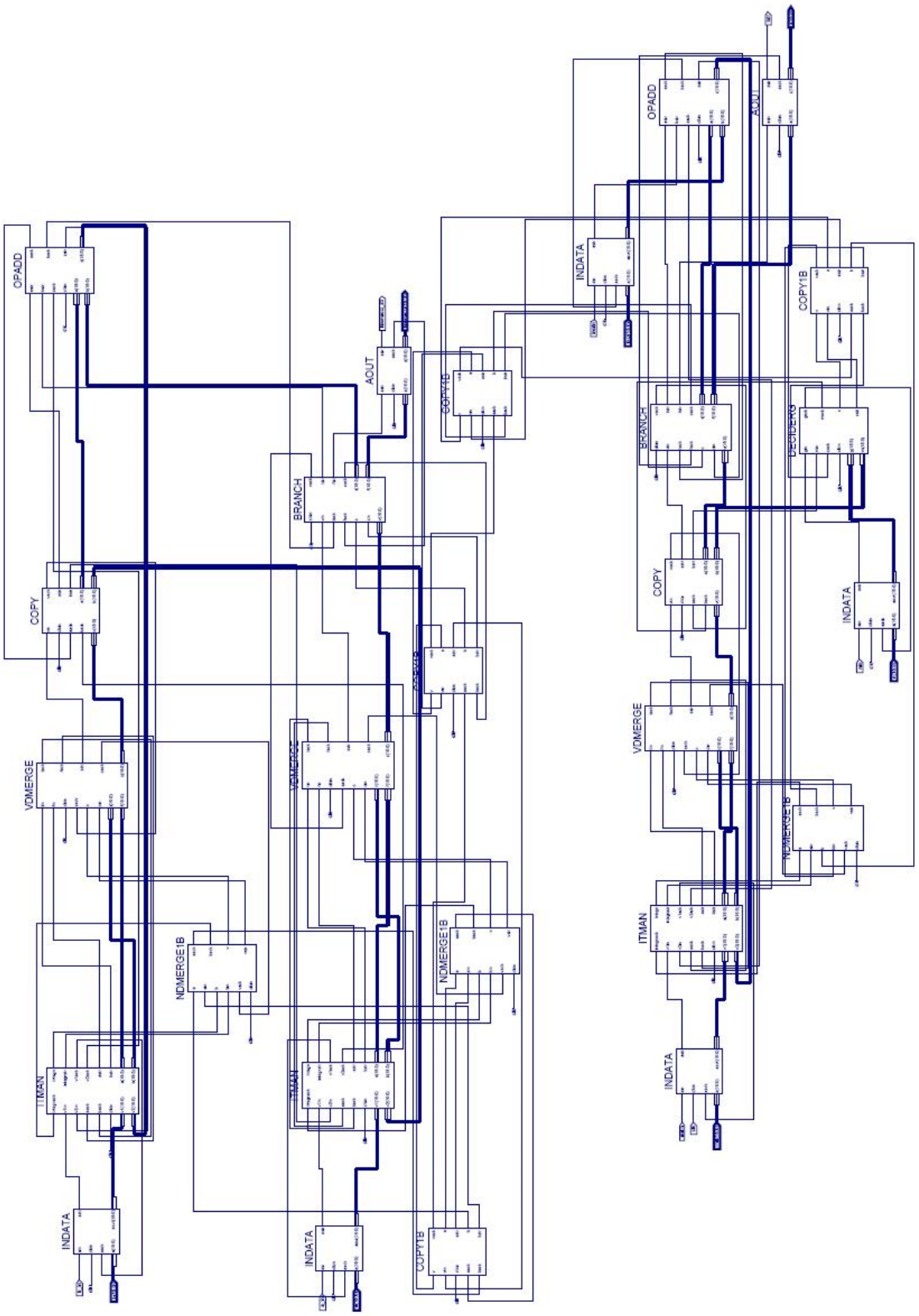


Figura A.5: Implementação do grafo a fluxo de dados para a Sequência de Fibonacci.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo