Universidade de São Paulo
Instituto de Física de São Carlos

JOSÉ TEIXEIRA DA SILVA JUNIOR

***PIFLOW - projeto, simulação e implementação
de um protótipo dataflow em FPGA***

São Carlos

2016

JOSÉ TEIXEIRA DA SILVA JUNIOR

***PIFLOW - projeto, simulação e implementação
de um protótipo dataflow em FPGA***

Dissertação apresentada ao Programa de Pós-Graduação em Física do Instituto de Física de São Carlos da Universidade de São Paulo, para obtenção do título de Mestre em Ciências.

Área de Concentração: Física Aplicada

Opção: Física Computacional

Orientador: Prof. Dr. Carlos Antônio Ruggiero

Versão Corrigida

(Versão original disponível na Unidade que aloja o Programa)

São Carlos

2016

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica revisada pelo Serviço de Biblioteca e Informação do IFSC,
com os dados fornecidos pelo(a) autor(a)

Silva Junior, José Teixeira da
PIFLOW - projeto, simulação e implementação de um
protótipo dataflow em FPGA / José Teixeira da Silva
Junior; orientador Carlos Antônio Ruggiero - versão
corrigida -- São Carlos, 2016.
148 p.

Dissertação (Mestrado - Programa de Pós-Graduação em
Física Computacional) -- Instituto de Física de São
Carlos, Universidade de São Paulo, 2016.

1. PIFLOW. 2. Dataflow. 3. Protótipo. 4. FPGA. 5.
Bluespec. I. Ruggiero, Carlos Antônio, orient. II.
Título.

AGRADECIMENTOS

O primeiro agradecimento dessa dissertação é ao orientador, professor e grande amigo Carlos Antonio Ruggiero, quem sempre mostrou os caminhos corretos e orientou magistralmente a realização do presente trabalho. Não menos importante nesse sentido, gostaria de agradecer ao Paulo Matias, que igualmente sempre orientou com bastante destreza, indicando técnicas e ensinando as melhores práticas para o desenvolvimento do projeto, a partir da linguagem de programação escolhida para o desenvolvimento. Um grande agradecimento ao Mateus José Martins, quem sempre proporcionou ótimas discussões acerca de *Dataflow*, e ofereceu bastante suporte técnico para a realização do trabalho. O Jecel Mattos Assumpção Jr., que introduziu de maneira extremamente intuitiva e didática, os princípios fundamentais, para o uso de FPGAs, muito obrigado por toda a paciência e dedicação dessas pessoas.

Agradeço imensamente ao Grupo CIERMag (Centro de Imagens e Espectroscopia *in vivo* por Ressonância Magnética), onde atuei profissionalmente durante o período deste projeto de mestrado, em especial ao professor Dr. Alberto Tannús, que sempre me ofereceu imenso suporte para o desenvolvimento desse trabalho. Agradeço ainda a todo o suporte técnico oferecido por esse grupo, que nos ofereceu todas as ferramentas e recursos necessários para este desenvolvimento.

Gostaria de agradecer imensamente à minha esposa e grande companheira Lhaís Visentin, quem sempre me deu grande apoio e suporte durante a realização de toda essa dissertação, não seria correto atribuir apenas a mim, a conquista da realização desse trabalho. Muito obrigado por todos os momentos que me ajudou a dedicar a esse desenvolvimento.

Gostaria de mencionar os nomes de alguns amigos especiais, que sempre ofereceram ótimas discussões, incríveis idéias, e transformaram esse trabalho no que ele se tornou, como Yole Vaz, Rodrigo Barreto, Danilo Mendes, Edson Vidoto, Guilherme Freire, Gustavo Vilaça, Felipe Coelho, André Freitas, Thereza Cury, Everton Oliveira, Pedro de Souza, Dennis Ortega, Eliane Teixeira e Helmuth Barbosa. Um agradecimento muito grande a todas essas pessoas, pois esse trabalho é parte de muito aprimoramento acadêmico, que certamente não seria possível sem o auxílio de todos eles.

Gostaria de agradecer a minha família, em particular meu pai José Teixeira, que deu bastante suporte ao longo de minha vida, permitindo que eu chegasse a mais essa etapa. Mas, em especial, gostaria de registrar nessas páginas toda a gratidão que possa ser possível

oferecer, à minha mãe, Ediene Santos da Silva, que sempre foi a melhor pessoa que eu tive o imenso prazer de compartilhar a minha vida. Uma pessoa que, apesar de seu acesso limitado à informação, sempre fez de tudo o que era possível ser feito, para que eu pudesse desenvolver certa capacidade intelectual, me permitindo chegar até aqui. Portanto, conseqüentemente, uma das partes mais fundamentais para a realização desse trabalho e de qualquer outro que eu possa vir a realizar ao longo de minha carreira. MUITÍSSIMO obrigado por tudo o que você fez por mim, sempre serei eternamente grato a tudo.

E para finalizar, gostaria de agradecer a todo o corpo docente e os funcionários do Instituto de Física de São Carlos, afinal de contas, sem todo o imenso esforço empregado por todos eles, ao longo de tantos anos, nada disso teria valor algum. Um grande agradecimento a todos.

A gratidão é a memória do coração.

Antistene

RESUMO

SILVA JUNIOR, J. T. *PIFLOW - projeto, simulação e implementação de um protótipo dataflow em FPGA*. 2016. 148 p. Dissertação (Mestrado em Ciências) – Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2016.

Esse trabalho tem por objetivo descrever o desenvolvimento e os atuais resultados do protótipo *dataflow* PIFLOW, um processador baseado no modelo de *dataflow* dinâmico, inspirado na Máquina *Dataflow* de Manchester e desenvolvido no Instituto de Física de São Carlos, da Universidade de São Paulo. Esse protótipo foi capaz de oferecer *speedups* muito próximos do ideal, a partir de programas que possuem grande grau de paralelismo, apresentando o grande potencial deste modelo - já bastante estudado no decorrer das últimas décadas - em oferecer desempenho superior ao de processadores sequenciais comerciais modernos.

Palavras-chave: PIFLOW. Dataflow. Protótipo. FPGA. Bluespec.

ABSTRACT

SILVA JUNIOR, J. T. *PIFLOW - project, simulation and implementation of a dataflow prototype in FPGA*. 2016. 148 p. Dissertação (Mestrado em Ciências) – Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2016.

The aim of this work is to describe the development and the current results of the PIFLOW dataflow prototype, a processor based on the dynamic dataflow model of execution, inspired by the Manchester Dataflow Machine and developed in the Instituto de Física de São Carlos, Universidade de São Paulo. This prototype shows speedups very close to the ideal case, when executing programs with a high degree of parallelism, showing the dataflow model potential, which has been extensively analysed in the last decades, to offer higher performance when compared to commercial modern sequential processors.

Keywords: PIFLOW. Dataflow. Prototype. FPGA. Bluespec.

LISTA DE FIGURAS

Figura 2.1 - Pequeno trecho de um Desenho Técnico	27
Figura 2.2 - Ilustração de um Wire Wrap	28
Figura 2.3 - Exemplo de circuito descrito em <i>Spice Netlist</i>	29
Figura 2.4 - Figura ilustrativa de uma PLA	31
Figura 2.5 - Programa de exemplo em PALASM	32
Figura 2.6 - Exemplo de um Mapa de Fusíveis, resultado da compilação de um PALASM	32
Figura 2.7 - Fundamentos da Tecnologia FPGA	34
Figura 2.8 - Complexidade do desenvolvimento no tempo	37
Figura 3.1 - Elaboração Estática	41
Figura 3.2 - Diagrama representativo de um módulo	42
Figura 3.3 - Composição de regras no BSV	44
Figura 3.4 - Escalonamento de regras	45
Figura 3.5 - Estação de trabalho do BSV	46
Figura 4.1 - Diagrama ilustrativo de uma instrução	48
Figura 4.2 - Representação ilustrativa de um programa Dataflow	51
Figura 4.3 - Programa Sisal	52
Figura 4.4 - Programa IF1	53
Figura 4.5 - Programa em Linguagem Montadora	53
Figura 4.6 - Diagrama ilustrativo da MDFM	55
Figura 4.7 - Diagrama ilustrativo da Fila de Fichas	56
Figura 4.8 - Diagrama ilustrativo da Unidade de Emparelhamento	57
Figura 4.9 - Diagrama ilustrativo da Unidade de Armazenamento de Instruções	58
Figura 4.10 -Diagrama ilustrativo da Unidade de Processamento	61

Figura 5.1 - Representação ilustrativa da arquitetura	64
Figura 5.2 - Representação ilustrativa da Fila de Fichas	65
Figura 5.3 - Representação ilustrativa da Unidade de Emparelhamento	68
Figura 5.4 - Representação ilustrativa da Unidade de Armazenamento de Instruções	71
Figura 5.5 - Representação ilustrativa da primeira versão da Unidade de Armazena- mento	73
Figura 5.6 - Representação ilustrativa da Unidade de Processamento	74
Figura 5.7 - Representação ilustrativa das Unidades Funcionais	78
Figura 6.1 - Representação ilustrativa da segunda fase na implementação da arqui- tutura.	85
Figura 7.1 - Speedup da primeira versão do projeto	99
Figura 7.2 - Speedup x Atraso	101
Figura 7.3 - Speedup da segunda versão do projeto para o programa do fatorial modificado	103
Figura 7.4 - Speedup da segunda versão do projeto para o programa de Integração Binária	105
Figura 7.5 - <i>Speedup</i> da segunda versão do projeto, para diferentes tamanhos de programas.	107
Figura 7.6 - Síntese da solução em um kit de FPGA da Terasic DE1-SoC, imple- mentado com o programa do somatorial de 120.	109
Figura A.1 - Comparação entre Linguagem Montadora e Verilog VMEM	125
Figura B.1 - Registrador de Histórico Efêmero	129
Figura B.2 - Implementação de uma FIFO	130
Figura C.1 - Distribuição de pacotes entre FIFOs	133
Figura C.2 - Implementação de um módulo para a distribuição de dados entre FIFOs.	134
Figura C.3 - Ciclo zero de execução do módulo para distribuição de dados entre FIFOs.	136
Figura C.4 - Ciclo um de execução do módulo para distribuição de dados entre FIFOs.	137
Figura C.5 - Ciclo dois de execução do módulo para distribuição de dados entre FIFOs.	138

Figura C.6 - Ciclo três de execução do módulo para distribuição de dados entre FIFOs.	139
Figura C.7 - Ciclo quatro de execução do módulo para distribuição de dados entre FIFOs.	140
Figura C.8 - Ciclo cinco de execução do módulo para distribuição de dados entre FIFOs.	141
Figura C.9 - Ciclo seis de execução do módulo para distribuição de dados entre FIFOs.	142
Figura D.1 - Exemplo de um programa <i>Dataflow</i> para a computação do cálculo numérico para a integração da curva sobre a função $y = x^2$	144
Figura D.2 - Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.	145
Figura D.3 - Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.	146
Figura D.4 - Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.	147
Figura D.5 - Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.	148

LISTA DE TABELAS

Tabela 7.1 - Resultados obtidos para o programa do fatorial modificado, durante a primeira fase da implementação.	100
Tabela 7.2 - Resultados obtidos para o programa do fatorial modificado, durante a segunda fase da implementação.	104
Tabela 7.3 - Resultados obtidos para o programa de integração binária, durante a segunda fase da implementação.	106

LISTA DE ACRÔNIMOS

AN - *Activation Name*

ASIC - *Application-Specific Integrated Circuits*

BSV - *Bluespec SystemVerilog*

BY - *Bypass*

CPLD - *Complex Programmable Logic Devices*

EEPROM - *Electrically Erasable Programmable Read-Only Memory*

EW - *Extract and Wait*

FPGA - *Field Programmable Gate Array*

GAL - *Generic Array Logic*

HDL - *Hardware Description Language*

IEEE - *Institute of Electrical and Eletronic Engineers*

IP - *Intelectual Property*

LUT - *Look-Up Tables*

MDFM - *Manchester Dataflow Machine*

MIMD - *Multiple Instructions and Multiple Data*

OTP - *One-Time Programmable*

PAL - *Programmable Array Logic*

PALASM - *PAL Assembler*

PLD - *Programmable Logic Devices*

PROM - *Programmable Read-Only Memory*

RAM - *Random Access Memory*

RISC - *Reduced Instruction Set Computer*

RTL - *Register Transfer Language*

TRS - *Term Rewriting System*

VHDL - *VHSIC Hardware Description Language*

VHSIC - *Very High Speed Integrated Circuits*

SUMÁRIO

1	Introdução	23
1.1	Motivação	23
1.2	Objetivos	24
1.3	Introdução Teórica	24
2	FPGA / HDL	27
2.1	Wire Wrap	27
2.2	Netlist	29
2.3	PAL	30
2.4	CPLD / FPGA	33
2.5	VHDL	35
3	Bluespec SystemVerilog	39
3.1	Sintaxe	40
3.2	Compilador	43
3.3	Escalonamento	43
4	Máquina Dataflow de Manchester (MDFM)	47
4.1	Aspectos Fundamentais	47
4.2	Programas	50
4.2.1	Sisal	52
4.2.2	IF1	52
4.2.3	Linguagem Montadora	53
4.3	Visão Geral	54
4.3.1	Fila de Fichas	54

4.3.2	Unidade de Emparelhamento	55
4.3.3	Unidade de Sobrecarga	57
4.3.4	Unidade de Armazenamento de Instruções	58
4.3.5	Unidade de Processamento	59
5	PIFLOW - Primeira Versão	63
5.1	Visão Geral	65
5.2	Fila de Fichas	65
5.3	Unidade de Emparelhamento	67
5.4	Unidade de Armazenamento de Instruções	71
5.5	Unidade de Processamento	74
5.6	Unidades Funcionais	78
6	PIFLOW - Segunda Versão	81
6.1	Motivação	81
6.1.1	Fila de Fichas	81
6.1.2	Unidades de Emparelhamento e de Armazenamento de Instruções	83
6.2	Objetivo	83
6.3	Implementação	86
6.3.1	Fila de Fichas	86
6.3.2	Unidade de Emparelhamento	88
6.3.3	Unidade de Armazenamento de Instruções / <i>Parser</i>	89
6.3.4	Unidade de Processamento	91
6.3.5	Unidades Funcionais	92
7	Resultados	95
7.1	PIFLOW - Primeira versão	95
7.2	PIFLOW - Segunda versão	102

7.3	Síntese	106
8	Conclusão	111
	REFERÊNCIAS	113
	Apêndice A – Parser	119
A.1	Especificações da Ferramenta	120
A.2	Definição das Características	122
	Apêndice B – Registradores de Histórico Efêmero	127
	Apêndice C – Exemplo de Bluespec	133
	Apêndice D – Exemplo de execução na MDFM	143

CAPÍTULO 1

Introdução

Toda a nossa ciência, comparada com a realidade, é primitiva e infantil – e, no entanto, é a coisa mais preciosa que temos. Albert Einstein

1.1 Motivação

Ao longo dos anos, o ganho em desempenho de sistemas computacionais tem sido motivo de grande discussão. Em meados dos anos 60, essa discussão já pairava sobre duas vertentes principais. Por um lado, imaginava-se que o aumento na frequência dos relógios das máquinas seria sempre acompanhado pelo avanço da tecnologia, como foi previsto pela Lei de Moore (1). Por outro lado, imaginava-se que esse aumento na frequência pudesse ser limitado, devido a limitações físicas como a velocidade da luz, o princípio da incerteza (2) ou a dissipação da potência em calor, por exemplo. Com isso, acreditava-se que o ganho em desempenho só seria possível através da execução concorrente de instruções, já que o potencial de exploração de paralelismo era perdido quando da execução puramente sequencial de programas.

Ambas as teorias perduraram até o final do século passado, quando foi possível observar que realmente existe um limite superior para o aumento na frequência do relógio em *chips* cada vez menores (1). Isso fortaleceu imensamente a outra vertente da discussão, de forma que a computação concorrente se tornou a principal fonte para o ganho de desempenho na computação atual (3–4). Esse será o principal foco do presente estudo.

Diversas formas de computação paralela foram propostas ao longo dos anos (5–6), dentre as quais destaca-se, por sua simplicidade, a computação dirigida por dados (7–10). Essa abordagem nos fornece uma imensa fonte de recursos na literatura e trata-se de um modelo que já foi profundamente explorado. Entretanto, ainda nos dias atuais, essa abordagem vem ganhando bastante notoriedade no mundo científico pela sua capacidade de explorar todo

o paralelismo disponível em uma aplicação (11–16). Nesse cenário específico de computação concorrente, podemos destacar um protótipo capaz de oferecer grande simplicidade para implementação e desempenho superior ao dos principais processadores de arquitetura convencionais de sua época, que é a Máquina *Dataflow* de Manchester (**MDFM**) (11).

1.2 Objetivos

Como toda máquina baseada em fluxo de dados, a **MDFM** apresenta uma arquitetura MIMD (Múltiplas Instruções e Múltiplos Dados) e possui caráter bastante modular, além de ser baseada em fichas dinâmicas, o que permite a reutilização de instruções durante a computação dos dados (17–18). Sendo assim, a partir dessa estrutura, o presente projeto pretende avaliar a visão geral da **MDFM** aplicada a tecnologias modernas, utilizando FPGAs, através de linguagens de descrição de hardware de alto nível, a fim de ganhar maior agilidade na implementação de diferentes abordagens, a partir de simulações e síntese nesses dispositivos. O principal objetivo do projeto é explorar o máximo em desempenho dessa arquitetura bastante consolidada no mundo acadêmico, buscando alternativas viáveis de implementação, a partir de um ponto de partida já muito bem estabelecido.

1.3 Introdução Teórica

O fluxo de dados é uma maneira de representar programas que apresentem paralelismo. Segundo essa abordagem, os programas são representados na forma de um grafo bidimensional: instruções que podem ser executadas paralelamente são dispostas uma ao lado da outra e instruções que dependem dos resultados anteriores, ou seja, instruções puramente sequenciais, são dispostas uma abaixo da outra. Nessa representação, a dependência de dados entre instruções, é evidenciada a partir de arestas direcionadas. Nesse modelo, as instruções não fazem referências a memórias, como pode ser visto em arquiteturas convencionais, uma vez que a dependência dos dados é capaz de transmitir as informações às instruções subsequentes, a partir das instruções que são executadas. Sendo assim, sistemas baseados em fluxo de dados, tem o objetivo de implementar esse modelo gráfico abstrato de computação (17).

Em uma arquitetura dirigida por dados, as informações transitam entre as instruções que devem ser executadas segundo a lógica descrita pelo programa. Estes dados são transpor-

tados entre as diversas unidades que compõem a estrutura da arquitetura por uma entidade denominada ficha. Essas entidades possuem todas as informações que um determinado dado necessita para a execução da instrução para a qual ele é destinado. A ficha carrega o dado, seu tipo, o endereço da instrução que essa ficha deve executar, e um rótulo, que é o identificador responsável pelo caráter dinâmico apresentado pela **MDFM** (10).

Um modelo dataflow dinâmico é caracterizado pela sua capacidade para a reutilização de instruções, no decorrer da execução de programas. Portanto, essa característica possibilita a realização de laços e recursividade, por exemplo. Essa propriedade é obtida a partir da marcação das fichas, indicando a qual nível de iteração ela se refere. Em um modelo dataflow estático, cada instrução pode ser referenciada uma única vez, pois dado o caráter assíncrono que uma execução pode apresentar, segundo essa abordagem, a reutilização de fichas não marcadas, poderia apresentar conflitos para a execução de diferentes níveis de iteração, durante esse tipo de operação. Dessa forma, modelos que apresentam essa característica estática, devem executar programas muito grandes, além de não ser possível a realização de operações recursivas, que não possuam a determinação explícita da quantidade de operações necessárias (19).

A estrutura proposta pela **MDFM** consiste basicamente em cinco unidades principais, que serão descritas brevemente a seguir, porém explorada com muito mais riqueza de detalhes nos próximos capítulos. Cada uma dessas unidades são conectadas por *buffers* de comunicação entre elas e são dispostas conforme listado a seguir:

- **Fila de Fichas:** trata-se de uma fila que atua como um *buffer* de entrada para a estrutura da máquina;
- **Unidade de Emparelhamento:** essa unidade é composta por uma memória de acesso pseudoaleatório limitada, mapeada por uma função *hash* característica. Ela garante o emparelhamento de fichas destinadas à mesma instrução, caso seja necessário;
- **Unidade de Sobrecarga:** uma unidade auxiliar à unidade anterior, caso o espaço de memória, correspondente a ficha corrente, estiver ocupado por uma outra ficha na função *hash* característica, essa ficha será encaminhada à **Unidade de Sobrecarga** para emparelhamento futuro;
- **Armazenamento de Instruções:** composta por uma memória de acesso aleatório que armazena o código de máquina, do programa que está sendo executado. Essa unidade fornece o *opcode* da instrução que essas fichas necessitam para serem executadas;

- **Unidade de Processamento:** essa unidade recebe as fichas emparelhadas, juntamente com a instrução a ser executada, e faz a execução dessa instrução no instante que qualquer **Unidade Funcional** estiver disponível. Essa unidade gera novas fichas que serão enviadas de volta a **Fila de Fichas** (17, 20).

Para o desenvolvimento desse projeto foi promovida a busca por uma linguagem de descrição de hardware de alto nível, mais abstrata, e que fosse capaz de fornecer grande versatilidade na criação do protótipo, de forma que, alterações substanciais, com relação ao projeto inicial, pudessem nos fornecer informações cruciais do desempenho da máquina, principalmente em nível de simulação. Buscamos essas características para que fosse possível desenvolver um protótipo que, embora totalmente inspirado na **MDFM**, permitisse imprimir novas características em um curto período de tempo.

Dentre tantas opções de linguagens de descrição de hardware com as características mencionadas, como o SystemC, ou o SystemVerilog por exemplo, optou-se pela utilização do **Bluespec SystemVerilog** (BSV). O BSV é uma linguagem de descrição de hardware funcional proposta pelo Prof. Arvind do MIT em 2003, que é essencialmente uma extensão do Haskell, destinada para o desenvolvimento de chips e automação eletrônica em geral (21–23). A principal característica do BSV é a garantia de um código-fonte curto, mais abstrato e fortemente tipado, de forma que a **Bluespec Inc.** garante 50% de ganho no desenvolvimento de hardware, com relação às linguagens RTL convencionais * (24), além da garantia de síntese na aplicação de qualquer solução, e ainda contando com um poderoso simulador intitulado como Bluesim (25), que foi uma das principais ferramentas utilizadas para o desenvolvimento do presente projeto.

* como o VHDL e o Verilog, por exemplo

ferrocianeto de potássio. Quando expostas à luz, as áreas do papel sensibilizado que não estivessem obscurecidas pelo desenho, sofriam duas reações químicas capazes de formar a cor azul característica de desenhos técnicos (28). Todo esse processo era necessário para garantir que o desenho não perderia seu contraste devido aos efeitos naturais do tempo.

Nesse momento, com o desenho técnico devidamente constituído, as placas de circuitos eletrônicos eram criadas através do método conhecido como *Wire Wrap* (29), para o desenvolvimento de placas em "larga escala". Esse método constitui-se em conectar os componentes eletrônicos, ou os caminhos do circuito das placas, através de fios isolados, enrolados entre seus terminais. Pode-se ver na figura 2.2 que esse método feito de forma manual é muito laborioso, sensível a erros e que torna bastante difícil a depuração, além de que, simulações do circuito que se deseja projetar, não são possíveis.

Figura 2.2 – Ilustração de um Wire Wrap

Fonte: HARTMANN (30)

É possível notar que esse método torna inviável o desenvolvimento de placas de circuitos maiores, devido a diversos fatores, tais como sua impossibilidade de simulação, a grande dificuldade para depuração de erros, além da inviabilidade para se analisar o circuito proposto a partir apenas de seu desenho técnico. Grandes circuitos possuem desenhos técnicos muito complexos, divididos em diversas páginas de *blueprint*, de forma que a documentação do hardware acaba ficando comprometida. Embora exista a possibilidade de utilizar esse nível de

abstração para o desenvolvimento de hardware, nota-se imediatamente as grandes dificuldades encontradas para fazê-lo.

2.2 Netlist

Devido a essa carência de documentação e mesmo para fins de formalismo matemático, passaram a ser utilizados *Netlists* para descrição de *Wire Wrap*. Pode-se ver na figura 2.3 um exemplo de código *Spice* para esse objetivo. Note a presença de definição da estrutura, além da definição do comportamento do circuito proposto. A figura apresenta um circuito muito simples descrito através de uma "linguagem para descrição de hardware" em nível de circuitos, que garante uma análise mais objetiva em trechos específicos, além de possibilitar depuração e mesmo simulação desse hardware, a partir da compilação de programas *Spice* por exemplo.

Figura 2.3 – Exemplo de circuito descrito em *Spice Netlist*

Fonte: MEDIA (31)

Dessa forma chega-se a definição de Linguagem de Descrição de Hardware, ou HDL (do inglês *Hardware Description Language*), que são linguagens de computadores, especializadas em gerar estruturas, desenhos e operações, em circuitos eletrônicos, mais especificamente circuitos de lógica digital (26). Esse tipo de linguagem garante uma descrição formal do hardware proposto, além de possibilitar a análise, testes de simulação e compilação desse código em um nível de abstração mais restrito, a fim de garantir a síntese desse código em dispositivos de componentes eletrônicos físicos.

Em nível de circuito, é possível realizar operações muito interessantes, como por exemplo utilizar um capacitor para gerar um atraso intencional, ou mesmo adicionar uma bobina para gerar um campo magnético a fim de se realizar uma operação bem mais complexa, ou ainda criar portas lógicas através da composição de um pequeno conjunto de elementos específicos. Porém, é possível perceber que esse nível de abstração ainda desfavorece a criação de circuitos muito complexos. Tome como exemplo o circuito proposto na Figura 2.1 e imagine um programa escrito na linguagem *Spice* para descrevê-lo. Trata-se de um programa extremamente complexo e difícil de ser analisado.

2.3 PAL

O problema para a descrição mais compacta era recorrente. A necessidade de criação de circuitos cada vez mais complexos, obrigaram o aumento do nível de abstração para descrição. Foram introduzidos, em meados dos anos 70, dispositivos capazes de conectar portas lógicas, definidas explicitamente no hardware: os Arranjos de Lógicas Programáveis, ou PALs (do inglês *Programmable Array Logic*) (29), que foram os primeiros dispositivos programáveis comercialmente viáveis. Uma PAL é constituída por vetores de fusíveis, arranjados em um plano chamado como OR-fixos e AND-programáveis, usados para implementar equações lógicas binárias de "somas de produtos" para cada uma de suas saídas, em termos de suas entradas definidas. O único problema, é que o dispositivo foi proposto com uma memória composta por uma Memória Programável Apenas de Leitura, as PROMs (do inglês *Programmable Read-Only Memory*). Tratava-se de um OTP (do inglês *One-Time Programmable*), ou seja, só podia ser escrito uma única vez.

A figura 2.4 mostra que é possível escrever equações matemáticas em termos das portas lógicas que compõe a PAL, a partir das informações vindas das portas de entrada do dispositivo. Observe portanto que, ao invés de se criar as portas lógicas em nível de circuito, como anteriormente, já existem tais portas definidas por padrão no dispositivo, realizando as operações desejadas a partir delas. Note que a criação de uma única porta lógica seguindo o método do *Wire Wrap* consome bastante trabalho. Já a PAL remove grande parte do esforço para a implementação, deixando apenas a escolha das operações desejadas a cargo do desenvolvedor do circuito, havendo portanto um ganho considerável em tempo para a implementação de circuitos, que antes pareciam inviáveis de serem desenvolvidos.

Juntamente com o surgimento desses dispositivos, foram propostas algumas Linguagens

Figura 2.4 – Figura ilustrativa de uma PLA

Fonte: WASHINGTON (32)

para Descrição de Hardware capazes de representar através de um processo de "compilação", as operações previstas para eles. Um exemplo é o PALASM (*PAL Assembler*), proposta pelo criador da PAL, no início dos anos 80. O PALASM é usado para traduzir funções booleanas e tabelas de transição de estados, em um **Mapa de Fusíveis**. A figura 2.5 mostra um programa escrito em PALASM. Note que os programas nessa linguagem já são divididos em três sessões distintas: Estrutura, Comportamento e Tabela de Funções, esta última, tendo por objetivo, realizar um teste no comportamento proposto, podendo ser interpretada como um teste de caso (ou um *Testbench*), para o hardware proposto. A estrutura do hardware é definida pela definição da placa utilizada, seguida pelos sinais que estarão disponíveis em suas portas de entrada. O comportamento é definido pelas linhas subsequentes à definição das portas, determinando a operação matemática desejada através de composições entre os sinais de entrada.

A compilação do PALASM gerará um Mapa de Fusíveis, que é a entidade mais importante desse processo e por analogia, poderá facilitar bastante o entendimento dos dispositivos que sucederam a PAL. A figura 2.6 apresenta um Mapa de Fusíveis, obtido através da compilação de um programa PALASM, onde cada ZERO ou UM, presente nesse arquivo, representa fusível queimado ou ativo, respectivamente. Quando "sintetizado" em uma PAL esse arquivo irá "mapear" a PROM presente na PAL, queimando os fusíveis representados pelos zeros desse mapa. Dessa maneira, ao final do processo, é possível obter um circuito perfeitamente definido

```

PAL16R4 PAL          PAL DESIGN SPECIFICATION
CNT4SC
4 bit counter with synchronous clear
Michael Holley and Dave Pellerin
Clk Clear NC NC NC NC NC NC NC GND
OE NC      NC /Q3 /Q2 /Q1 /Q0 NC NC VCC

Q3 := Clear
    + /Q3 * /Q2 * /Q1 * /Q0
    + Q3 * Q0
    + Q3 * Q1
    + Q3 * Q2

Q2 := Clear
    + /Q2 * /Q1 * /Q0
    + Q2 * Q0
    + Q2 * Q1

Q1 := Clear
    + /Q1 * /Q0
    + Q1 * Q0

Q0 := Clear
    + /Q0

FUNCTION TABLE
OE Clear Clk  /Q0 /Q1 /Q2 /Q3
-----
L  H  C      L  L  L  L
L  L  C      L  L  L  L
L  L  C      L  H  L  L
L  L  C      H  H  L  L
L  L  C      L  L  H  L
L  H  C      L  L  L  L

```

Figura 2.5 – Programa de exemplo em PALASM

Fonte: HOLLEY (33)

e dadas as definições das portas de entradas é possível obter as saídas esperadas pelo circuito.

```

L0000 01111111111111111111111111111111*
L0032 11111111111111111111111111111101*
L0064 00000000000000000000000000000000*
L0096 00000000000000000000000000000000*
L0128 00000000000000000000000000000000*
L0160 00000000000000000000000000000000*
L0192 00000000000000000000000000000000*
L0224 00000000000000000000000000000000*
L0256 11111111111111111111111111111101*
L0288 11101111111111111111111111111111*
L0320 11111111111111111111111111011111*
L0352 11111111111111111111111111110111*

```

Figura 2.6 – Exemplo de um Mapa de Fusíveis, resultado da compilação de um PALASM

Fonte: HOLLEY (33)

Portanto é possível verificar um avanço substancial para a criação de circuitos eletrônicos com o surgimento das PALs, já que grande parte do trabalho que existia para a criação de novos circuitos foram substituídos apenas pela definição de operações lógicas específicas, que dispostas de maneira coerente, poderiam realizar uma quantidade muito grande de operações. Porém, é importante observar que embora exista um grande avanço no processo, imagine-se a criação do circuito proposto na Figura 2.1, a partir de PALs. É possível verificar imediatamente que o espaço de memória presente em uma PROM, não permite a criação de circuitos muito complexos. Além disso, nem todas as operações previstas para esse circuito poderiam ser

transportadas para essa nova abordagem, sobrando ainda o fato que a utilização de múltiplas PALs em cascata aumenta consideravelmente a capacidade para visualização do código e consequente dificuldade para manutenção do circuito desejado, inviabilizando a utilização dessa possibilidade.

Tendo em vista as limitações existentes nas PALs, ocorreu uma evolução natural em dispositivos programáveis. Em meados dos anos 80 surge o que foi chamado de Lógica de Arranjos Genéricos, GAL (do inglês *Generic Array Logic*), onde a PROM característica da PAL foi substituída por uma Memória Programável, apenas para Leitura, Apagada Eletricamente, a EEPROM (do inglês *Electrically Erasable Programmable Read-Only Memory*), mas em geral acrescentou-se pouco às principais características das PALs em nível de desempenho nos resultados e evolução dos dispositivos.

2.4 CPLD / FPGA

Uma evolução real com relação às PALs foram os Dispositivos de Lógicas Programáveis, as PLDs (do inglês *Programmable Logic Devices*), que em sua concepção tinham a capacidade de conectar 4 PALs através de interconexões programáveis, facilitando a comunicação entre PALs, que era uma limitação recorrente desses dispositivos. Eles sofreram diversas modificações até atingir números próximos a centenas de milhares de PALs em um único chip, permitindo a criação de circuitos muito mais complexos a partir desses dispositivos, dando origem aos Dispositivos de Lógicas Programáveis Complexas, ou CPLDs (do inglês *Complex Programmable Logic Devices*). Na verdade, uma CPLD é um dispositivo perfeitamente intermediário entre uma PAL e uma FPGA (do inglês *Field Programmable Gate Array*), ou um Arranjo de Portas Programáveis de Campo (34). A principal diferença entre uma CPLD e uma FPGA é que essa última possui suas interconexões compostas por blocos lógicos complexos, com ampla capacidade de roteamento, enquanto que a CPLD é basicamente composta pelas funções combinacionais de operações lógicas básicas, como a soma de produtos característica das PALs por exemplo.

A FPGA teve sua criação paralelamente à evolução das PALs descritas anteriormente, ainda na década de 80. Bem como a ideia original da PAL, a FPGA combina os melhores recursos dos Circuitos Integrados Construídos para Tarefas Específicas, os famosos ASICs (do inglês *Application-Specific Integrated Circuits*) e sistemas baseados em processadores. As pastilhas de silício reprogramáveis possuem a mesma flexibilidade de software, mas não são

limitadas pela quantidade de núcleos de processamento. Diferentemente de processadores convencionais, FPGAs são verdadeiramente paralelos, de forma que diferentes operações que exigem processamento, não precisam competir pelos mesmos recursos. Cada tarefa específica é enviada para uma seção dedicada do *chip* e pode funcionar de modo autônomo, sem nenhuma influência de outros blocos lógicos. Como resultado, o desempenho de uma parte da aplicação não é afetada ao se adicionar mais tarefas.

Figura 2.7 – Fundamentos da Tecnologia FPGA

Fonte: NATIONAL INSTRUMENTS (35)

Toda FPGA é constituída de um número finito de recursos de hardware predefinidos, com interconexões programáveis por software e implementadas em hardware, além de possuir diversos blocos de entrada e saída, que permitem que o circuito se comunique com o mundo externo. As especificações de uma FPGA geralmente incluem o número de blocos lógicos configuráveis que compreendem o *chip*, o número de blocos lógicos e funções fixas, como multiplicadores e blocos de Memória de Acesso Aleatório, RAM (do inglês *Read-Only Memory*), que dentre todos os diversos componentes que compõem uma FPGA são as mais importantes para uma aplicação (35). Os blocos lógicos configuráveis são as unidades lógicas básicas da FPGA, compostos essencialmente de dois componentes principais:

- **Flip-Flops:** Circuitos digitais utilizados para sincronizar lógicas e registrar os estados lógicos entre os ciclos do relógio; e
- **Look-Up Tables:** Toda lógica combinatória * são implementadas dentro de tabelas ver-

*ANDs, ORs, NANDs, XORs e assim por diante

dades na memória LUT (do inglês *Look-Up Tables*), ou seja, dentro de listas predefinidas de saídas para cada combinação de entradas específicas.

2.5 VHDL

Observe por essa breve descrição histórica que, embora os dispositivos programáveis tenham sofrido diversas modificações e avanços ao longos dos anos, as linguagens de descrição de hardware para esses dispositivos ficaram um tanto congeladas em evolução, ao longo do tempo. Ao final da década de 70, o Departamento de Defesa dos EUA estava finalizando a criação e implementação da linguagem de programação ADA, que tinha a intenção de substituir e unificar as centenas de linguagens de programação de software utilizadas em todos os projetos liderados por esse departamento. A linguagem foi implementada com imenso sucesso, de forma que todos os projetos estavam sendo documentados seguindo as definições dessa linguagem.

De maneira equivalente, os projetos ASICs liderados por esse departamento, carecia muito de especificações e documentação, afinal não haviam maneiras objetivas e formais para fazê-lo. Havia *Blueprints* com a especificação de programas baseados em Netlist's, ou mesmo compatíveis com PALs ou CPLDs, além de muitos outros formatos que já haviam sido criados até esse momento (que não foram apresentadas nesse capítulo). Dessa forma ocorreu uma grande mobilização para a criação de normas e diretrizes para a descrição e documentação de todos os ASICs liderados pelo Departamento de Defesa dos EUA (36).

Essa mobilização deu origem ao VHDL (do inglês *VHSIC Hardware Description Language*), ou Linguagem para Descrição de Hardware VHSIC (do inglês *Very High Speed Integrated Circuit*), Circuitos Integrados de Alta Velocidade.

O VHDL é uma linguagem com nível de abstração baseada em registradores, também referenciada como RTL (do inglês *Register Transfer Language*), nível bem mais alto que aquele baseado em portas lógicas, existentes anteriormente. Registradores são implementados através de *Flip-Flops* que podem ser criados a partir de portas lógicas, que por sua vez são criadas a partir de circuitos eletrônicos físicos. Essa evolução de nível de abstração da linguagem, se deu pela necessidade inicial do VHDL, que tinha o objetivo apenas de descrever de maneira sucinta, porém precisa, as operações previstas pelos ASICs propostos. Note que o nível de abstração da linguagem HDL se dá no sentido mais estrito da palavra, já que se utiliza a abstração de determinado componente físico, de forma que, embora a descrição desses

componentes estejam presentes no circuito final, eles não precisam ser escritos explicitamente pela linguagem. Por exemplo, o nível RTL faz o uso de registradores, componentes obtidos a partir da combinação de *Flip-Flops*, composto por portas lógicas. Portanto, as portas lógicas estão presentes no nível RTL, mas não precisam ser escritas explicitamente pelo desenvolvedor do circuito.

A ideia de simular ASICs a partir dessas informações de documentação era obviamente atrativa aos olhos do Departamento de Defesa (36), de forma que foram desenvolvidos diversos compiladores, para simulação de circuitos lógicos, a partir da leitura desses arquivos VHDL, o que foi seguido pela criação de ferramentas para a síntese desses códigos nos mais diversos dispositivos de lógicas programáveis existentes naquele momento, como PALs, GALs, CPLDs, ou FPGAs. Com isso, a VHDL passou a ser a linguagem padrão para descrição de hardware de dispositivos programáveis. É interessante interpretar o ato de síntese nesses dispositivos FPGAs (conforme vem-se destacando em diversos instantes dessa descrição) da mesma maneira como a gravação das PALs, através do mapa de fusíveis. A compilação da linguagem de descrição de hardware fornece um código análogo a um *Assembler*, e esse código tem por objetivo definir todas as rotas de comunicação entre os blocos lógicos desses dispositivos, além de definir quais ações específicas esses blocos irão realizar, através das LUTs. Note portanto a equivalência com os mapas de fusíveis, embora tratando-se de uma tarefa muito mais complicada e específica.

Com o sucesso do VHDL, a sua definição foi colocada como domínio público, o que a levou a ser padronizada pelo IEEE (do inglês *Institute of Electrical and Eletronic Engineers*). O fato de ser uma linguagem de domínio público ampliou ainda mais a sua utilização, tendo sido propostas diversas alterações. Como é natural num processo de aprimoramento, a linguagem vem recebendo diversas revisões desde então. Além da evolução da própria linguagem, ela motivou a criação de muitas outras linguagens para descrição de hardware, dentre as quais destaca-se por ter igualmente se tornado um padrão IEEE, a Verilog. Por possuir nível de abstração RTL, a linguagem Verilog peca em alguns pontos cruciais para síntese de hardware, quando comparada ao VHDL, principalmente no que diz respeito a sincronismo entre módulos (37). Embora o Verilog careça de rigorosidade em alguns pontos, ela se tornou igualmente importante nesse nível de abstração e paralelamente ao VHDL, constituem os principais padrões em linguagens para descrição de hardware atualmente.

É muito importante observar a evolução constante em nível de abstração para descrição de hardware. Os principais motivos para essa evolução são: o aumento natural na complexidade de ASICs; o tempo gasto para o desenvolvimento; além do tempo levado para compilação, que aliado ao processo de simulação, inviabiliza a utilização de linguagens em níveis de abstração mais baixos. O desenvolvimento em nível de abstração menor, oferece maior controle sobre o

hardware que se deseja implementar, como por exemplo a definição explícita de um registrador, ou mesmo de um circuito lógico específico. Porém é importante lembrar que houve uma evolução natural até a chegada a esse nível: da mesma forma que o nível RTL oferece maior controle sobre um registrador, o nível de circuitos oferece maior controle sobre circuitos, bem como o nível de portas lógicas oferece maior controle sobre operações lógicas. Dessa forma, o único argumento coerente para a utilização de uma ou outra linguagem é a facilidade para implementação de determinadas características fundamentais, que variam de acordo com as necessidades do circuito integrado que se deseja desenvolver. Deve-se ainda considerar a quantidade de linhas de código necessárias para a implementação dessas características, já que é possível desenvolver qualquer circuito com a utilização de *Wire Wrap* por exemplo; devemos apenas avaliar a viabilidade de realizar essa tarefa.

Figura 2.8 – Complexidade do desenvolvimento no tempo

Fonte: GRAAF (38)

Nos dias atuais, pode-se perceber que o uso apenas de linguagens RTL para o desenvolvimento de ASICs tem se tornado inviável, mesmo para o desenvolvimento de soluções corriqueiras, já que a construção desses modelos exige grande esforço, consumindo longos períodos de tempo. Também, a simulação de grandes construções nessas linguagens apresenta desempenho bastante lento, pois a principal proposta delas é trazer maior comodidade para síntese, não havendo grande esforço para implementação de módulos específicos para simulação. Dessa forma, a principal solução encontrada ao longo dos anos foi a criação de linguagens com nível superior ao do RTL, que têm por objetivo sanar essas deficiências. As principais

soluções nesse contexto inclusive, são ferramentas geradoras de códigos RTL, mesmo pelo fato de os maiores produtores de FPGAs da atualidade não fornecerem ferramentas para síntese de códigos diferentes desses.

Diversas linguagens já foram implementadas para esses propósitos, e dentre tantas opções foi escolhido para o presente trabalho o **Bluespec SystemVerilog**, uma linguagem HDL que apresenta sintaxe semelhante a do SystemVerilog[†]. Todo código desenvolvido nessa linguagem possui garantia de ser 100% sintetizável, além de possuir excelente qualidade de síntese (25).

[†]Outra linguagem HDL de alto nível, já largamente explorada.

CAPÍTULO 3

Bluespec SystemVerilog

O **Bluespec SystemVerilog** (BSV) (24) é direcionado para desenvolvedores de hardware que procuram uma ferramenta que seja compatível com as principais linguagens de descrição de hardware padrões existentes (Verilog, VHDL ou SystemVerilog, por exemplo), para o desenvolvimento de ASICs ou síntese em FPGAs. O BSV é baseado no subconjunto sintetizável do SystemVerilog, que inclui Tipos de Dados, Módulos, Instância de Módulos, Interfaces, Instância de Interfaces, Parametrização, Elaboração Estática e Geração de Elaboração. É uma linguagem para síntese de hardware fortemente tipada, exigindo que todo contexto e comunicação sejam equivalentes (21). Essa característica garante a correção de uma das principais deficiências do Verilog, que não exige total coerência nos tipos de dados utilizados. Além disso, a linguagem baseia-se no Sistema de Reescrita de Termos, o TRS (do inglês Term Rewriting System), um sistema de substituição abstrato, usado para criar cadeias lógicas a partir de determinadas regras. No BSV, o TRS é usado para descrever a sintaxe da linguagem como uma série de mudanças atômicas de estados.

Uma das principais inovações do BSV com relação a outras linguagens de nível mais alto, é a possibilidade de expressar um comportamento totalmente sintetizável a partir das chamadas Regras, ao invés de trechos síncronos, comuns nas linguagens HDL, como por exemplo o *always* do Verilog, ou o *process* do VHDL. Nessas linguagens, esses blocos oferecem um ambiente para definição comportamental do hardware, garantindo concorrência de operações, porém deixando a resolução de eventuais conflitos entre seus elementos a cargo do desenvolvedor. As regras no BSV introduzem um conceito poderoso para alcançar um paralelismo coerente, eliminando as condições que apresentam conflitos. Nesse sentido, cada regra pode ser vista como uma declaração assertiva, que expressa uma transição **atômica** no estado do circuito.

Embora as regras possam ser expressadas através do estilo modular característico, uma regra pode conter múltiplos módulos, ou seja, pode testar e afetar o estado do hardware em múltiplos módulos, o que permite a criação de hardware reconfigurável de maneira bastante simplificada. Pelo fato de serem **atômicas**, as regras podem concorrer com elementos de estado comuns entre elas, como registradores por exemplo. O compilador do BSV produz um código Verilog (RTL), capaz de gerenciar potenciais interações entre as regras, adicionando

determinadas arbitrariedades e lógicas de escalonamento entre elas. Pode-se notar portanto, que o caráter atômico das regras oferece uma maneira escalável para lidar com concorrências, mesmo que inesperadas a priori, em hardwares muito complexos.

Como sua primeira versão foi desenvolvida com base em Haskell (23), o BSV possui muitas características funcionais. A programação funcional é um paradigma que trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis (39). Ela enfatiza a aplicação de funções, em contraste à programação imperativa, que enfatiza mudanças no estado do programa. Suas características funcionais, aliadas à sintaxe de linguagens padrões HDL - além do inovador padrão para desenvolvimento comportamental do hardware - são os principais fundamentos que dão ao BSV um grande aumento na produtividade. Como exemplo, em BSV é possível instanciar diversos módulos iguais com interfaces de comunicação específicas e valores padrões definidos, em uma única linha de programação, operação que levaria uma quantidade muito maior de linhas e lógica de programação de hardware em linguagens RTL convencionais.

3.1 Sintaxe

Uma das características mais importantes do BSV é a restrição que ele impõe no desenvolvimento, que busca a geração de circuitos puramente combinacionais. As variáveis devem ser interpretadas como a identificação de uma ligação apenas, e não como costuma-se interpretá-las em linguagens de software convencionais (equívoco comum entre iniciantes em desenvolvimento de hardware), ou seja, como elementos de armazenamento ou recipientes que irão conter determinado valor. No BSV, as variáveis e expressões de atribuição apenas descrevem o fluxo dos dados em um circuito combinacional. É importante interpretar essas expressões na forma como elas estarão dispostas no circuito que será construído. Note na figura 3.1 o caráter da elaboração estática que a linguagem impõe sobre essas expressões, e observe ainda, a que se referem as definições de variáveis nesse contexto, identificando as conexões entre as entidades definidas. A figura 3.1 apresenta um fluxograma para a computação do seguinte trecho de código:

```
b = 10;  
bsq = b*b;  
discr = bsq - 4*a*c;
```

Figura 3.1 – Elaboração Estática

Fonte: NIKHIL (22)

Muitas linguagens HDL buscam aumentar o nível de abstração em sua sintaxe, para tornar a experiência do desenvolvimento mais convencional. Com isso, elas acabam apresentando semânticas de linguagens de software convencionais, que são puramente sequenciais, como C ou C++ por exemplo. Essa característica acarreta a perda do controle do hardware que se deseja projetar, além de perder o caráter fundamental de um hardware que é a sua capacidade em explorar um ambiente estritamente paralelo. Isso não acontece com o BSV, pois ele foi construído com base na semântica tradicional de linguagens HDL, muito embora a linguagem consiga explorar conceitos muito avançados de software - o que pode ser visto em evidência a partir de seu caráter funcional - que são utilizados apenas para verificação e elaboração estática da estrutura e comportamento do hardware.

O BSV utiliza os modelos padrões e familiares em estruturas de hardware, compatíveis com VHDL ou Verilog. São os Módulos, as Instâncias de Módulos e as Hierarquias de Módulos. Essas entidades caracterizam as estruturas capazes de garantir a elaboração estática, além de determinar a estrutura da arquitetura que se deseja implementar. De maneira equivalente, o BSV incorporou o modelo de Interfaces e Instância de Interfaces do SystemVerilog para comunicação entre os Módulos, determinando parte das características comportamentais do hardware. Além disso, o sistema de Tipos de Dados padrão da linguagem, também é baseado inteiramente no sistema equivalente do SystemVerilog. Porém, toda essa estrutura é extremamente aprimorada com algumas novas ideias de organização, a fim de obter o aumento na produtividade proposto pela **Bluespec Inc.**, garantindo com isso, um desenvolvimento mais rápido, seguido naturalmente por uma verificação mais rápida (a partir de sua ferramenta

única para simulação), além de maior facilidade para manutenção.

Figura 3.2 – Diagrama representativo de um módulo

Fonte: DAVE (40)

O comportamento geral dos módulos são implementados através do conjunto de regras mencionadas anteriormente, bem como a composição da interface dos módulos - a maneira através da qual é possível fazer comunicação entre módulos - que é definida por um conjunto de fragmentos de regras e são chamadas de Métodos. Ambas as entidades, Métodos de Interfaces e Regras são constituídas pela tupla $(executar, ação)$, o identificador *executar* é uma função do tipo booleano que especifica quando uma regra pode ser disparada, ou no caso de um método, quando ele pode ser invocado. O identificador *ação* determina o que essas entidades irão executar no estado atual da estrutura dos módulos. Uma regra dentro de um módulo pode invocar métodos em interfaces de outros módulos, que por sua vez ainda podem invocar métodos de novos módulos. Note porém que, embora essas regras possam estender múltiplos módulos, elas serão compostas modularmente, isto é, com condições e ações encapsuladas em cada módulo.

É importante enfatizar que mesmo quando uma regra se estende por múltiplos módulos, existe a garantia de uma propriedade atômica global, de forma que o comportamento da estrutura de todo o sistema pode ser analisado como uma composição sequencial de um conjunto de regras, que podem ser disparadas concorrentemente dentro de um ciclo específico do relógio. Esse é o tipo de encapsulamento que é totalmente garantido pelo escalonador do compilador do BSV. Imagine a quantidade de trabalho que deveria ser empregado para garantir uma estrutura consistente e sincronizada com essas características em uma linguagem RTL convencional. Esse portanto é o nível de abstração que o BSV é capaz de oferecer, de

maneira bastante coerente. É importante ainda recordar-se que Regras e Métodos de Interfaces possuem as mesmas características; então, quaisquer referências à regras mencionadas, estendem-se diretamente para os métodos.

3.2 Compilador

O compilador BSV tem o objetivo de traduzir a descrição da linguagem em duas principais ferramentas, um código RTL pronto para síntese, descrito em Verilog e um código em C, destinado para simulação e totalmente baseado no primeiro. Como foi desenvolvida com base nesses preceitos, a linguagem consegue garantir que ambas as soluções sejam perfeitamente equivalentes, ou seja, a simulação específica da linguagem, será composta exatamente como o hardware proposto. Essa característica é imposta a partir do *Parser* da linguagem, em uma descrição TRS das regras e consequentes estados. Baseado nessa descrição TRS, o compilador faz o escalonamento das ações e transições de estados em uma descrição de hardware segmentada. Essa tarefa determina os instantes em que regras podem ser disparadas com e/ou sem concorrência, adicionando lógicas capazes de lidar com a troca de estados dos elementos alterados por essas regras e por fim, aplicando otimizações booleanas a fim de simplificar o design do circuito.

3.3 Escalonamento

Toda regra é candidata para execução em todo ciclo de máquina. Além disso, todas as ações propostas dentro de uma regra serão executadas no mesmo ciclo de máquina, ou seja, concorrentemente. Dessa forma, ao final de um determinado ciclo, todo estado proposto por essa regra terá afetado o estado do hardware para o próximo ciclo. Uma regra não será disparada somente se uma das três considerações seguintes forem satisfeitas:

1. Uma condição explícita, ou predicado, for verdadeiro. Baseado no estado atual da máquina, identificado a partir de registradores gerais, que são entidades comuns entre regras;
2. Um método que dispute recursos com essa regra específica foi invocado. Lembre-se que os métodos são responsáveis pela comunicação entre os módulos; portanto não é

possível identificar a priori quando eles serão disparados, eles possuem nível de prioridade superior ao das regras;

3. A execução de uma outra regra concorrente, que possua nível de prioridade superior ao da regra analisada. É possível determinar esses níveis de prioridades a partir de diretivas explícitas presentes na linguagem. Caso essas diretivas não sejam apresentadas, o conflito será descrito em detalhes no processo de compilação, identificando que em algum momento da execução, poderão haver conflitos entre as regras determinadas.

Note portanto, que o escalonador do BSV tem a função de realizar a operação que deveria ser analisada em todo o comportamento do hardware explicitamente pelo desenvolvedor da arquitetura, em uma linguagem RTL convencional. Em projetos muito complexos, a descrição TRS tende a ser muito mais consistente em comparação com uma verificação manual.

Figura 3.3 – Composição de regras no BSV

Fonte: NIKHIL (22)

O escalonador faz a verificação a partir da manipulação explícita de dois sinais específicos em todo ciclo de máquina. Se as considerações 1 e 2 descritas anteriormente, não forem satisfeitas, então essa regra específica será marcada através do sinal booleano *CAN_FIRE*. A partir desse instante já existe um filtro de regras que podem ser disparadas nesse ciclo. Nesse momento, são verificadas individualmente todas as regras que possuem esse sinal habilitado. Para cada regra selecionada, são analisadas todas as regras que possuem o sinal *WILL_FIRE* definidos como verdadeiro. Caso não existam regras de prioridade superior com esse outro sinal habilitado, essa regra específica terá esse sinal (*WILL_FIRE*) definido como verdadeiro. Essa verificação segue uma série de condições que irão garantir no hardware uma execução coerente e determinística. O correto entendimento desse ciclo foi fundamental para a realização de determinadas verificações manuais para o escalonamento de certas regras no desenvolvimento do presente trabalho, com o auxílio da ferramenta explicada em detalhes no Apêndice B. Essas verificações manuais que foram necessárias serão apontadas explicitamente no decorrer da descrição, quando conveniente.

Figura 3.4 – Escalonamento de regras

Fonte: NIKHIL (22)

Os métodos no BSV tem o objetivo de fazer a comunicação entre os módulos. É possível identificar os parâmetros de entrada dos métodos como as portas de entrada de um módulo e o retorno desses métodos (se houver), como as portas de saída deles. Como visto anteriormente, esses métodos possuem características semelhantes às de regras e portanto, devem ser planejadas como tal, porém não são todos os tipos de métodos capazes de entrar em conflito com as regras. Essa característica é definida a partir de qual tipo de método deve ser utilizado, eles são definidos conforme a descrição a seguir:

- **Método Value:** São puramente combinacionais. Esse tipo de método não afeta o estado do hardware, ele tem por objetivo apenas retornar o valor do estado de um componente do módulo. Tome como exemplo o método *first()* de uma FIFO, ele não afeta o estado da FIFO, apenas retorna seu valor.
- **Método Action:** Esse tipo de método afeta o estado do hardware e portanto pode apresentar conflitos com regras ou outros métodos, caso essas entidades disputem pelo estado do elemento causador desse conflito. Métodos desse tipo tem o objetivo apenas de alterar o estado desses elementos, não podendo retornar quaisquer resultados. Pode-se colocar como exemplo o método *enq()* de uma FIFO. Esse método remove o elemento inicial desse componente. Imagine se uma regra buscar acessar essa FIFO, isso implicaria em um conflito que não pode ser tratado por causa das características atômicas das entidades envolvidas nesse processo. Por isso o escalonador deve bloquear a execução da regra conflitante, até o próximo ciclo, onde o estado desse elemento já tenha sido alterado.
- **Método ActionValue:** Esse método altera o estado do hardware e ainda retorna o estado resultante de quaisquer elementos envolvidos na operação, para a entidade que o invocou. Tome como exemplo o método *pop()* de uma estrutura de pilha. Ele remove o primeiro elemento dessa pilha e retorna-a.

Figura 3.5 – Estação de trabalho do BSV

Fonte: NIKHIL (22)

Observe que o BSV oferece uma imensa gama de ferramentas capazes de garantir a criação de Propriedades Intelectuais, ou IPs (do inglês Intellectual Property) de muitas formas distintas, seja através da criação de simulações de alto nível (que acima de tudo possui garantia de síntese), ou através da criação de ASICs, ou de síntese em FPGAs. Um ponto muito forte da linguagem é sua ferramenta própria para simulação, que é gerada a partir do hardware proposto, de forma que ele consegue obter especificamente todas as características desse e com isso permite simular o hardware da mesma forma como ele será executado quando implementado, ciclo a ciclo. Tudo isso é garantido através de uma linguagem funcional de alto nível, que embora possua características comportamentais diferentes com relação às linguagens HDL padrões, segue no mesmo embasamento teórico dessas linguagens, de forma que embora exista uma curva de aprendizagem grande, para trabalhar com essa ferramenta, essa curva certamente será compensada no ganho em tempo de desenvolvimento.

O Apêndice C apresenta um exemplo bastante ilustrativo, que permite um melhor entendimento do processo de escalonamento do compilador BSV, a partir de um problema com descrição bem específica. Esse exemplo ainda é capaz de mostrar a simplicidade que o BSV é capaz de oferecer para o desenvolvimento de circuitos específicos.

CAPÍTULO 4

Máquina Dataflow de Manchester (MDFM)

4.1 Aspectos Fundamentais

A Máquina Dataflow de Manchester é organizada na forma de um anel. Dessa maneira, fichas, ou pacotes, são transferidos de uma unidade a outra, a fim de coletar todas as informações exigidas pelas Unidades Funcionais para a execução de uma instrução, operação que gerará novas fichas, reiniciando o ciclo. Nessa arquitetura particular do modelo de fluxo de dados, cada instrução exige no máximo dois dados para sua execução, e são capazes de fornecer no máximo dois dados em sua saída. Note que essa característica é própria da arquitetura proposta por Manchester, já que o modelo não é restrito a esse número (17–18, 41).

Essa implementação é baseada no paradigma de fluxo de dados dinâmico, que permite a reutilização de instruções no decorrer da execução de programas. Essa característica permite a realização de laços, recursividade, ou execução de funções, por exemplo. Para alcançar essa propriedade dinâmica a máquina utiliza um identificador chamado Nome de Ativação, ou AN (do inglês *Activation Name*), que é um rótulo numérico, sequencial, presente em todas as fichas e que deve ser gerado diante de uma solicitação explícita no código do programa corrente em execução. Dessa forma, o processador deve ser capaz de gerar quantos ANs forem necessários para a execução de programas. Ainda existem os identificadores Nível de Iteração e Índice, que em conjunto com o AN são capazes de garantir essa propriedade dinâmica em diversas situações.

Uma ficha deve conter todas as informações necessárias para a execução de uma instrução. A principal entidade que uma ficha possui é o dado que ela carrega, bem como o tipo desse dado. As fichas podem ser marcadas como *bypass* ou *non-bypass*. Fichas do tipo *bypass* não precisam ser emparelhadas com fichas parceiras antes de buscar a instrução a que se destinam. Estas instruções exigem apenas um dado para sua execução, portanto não consomem recursos da memória que compõe a Unidade de Emparelhamento. Em contrapartida, fichas do tipo

non-bypass deverão ser emparelhadas e ficarão armazenadas na memória interna da Unidade de Emparelhamento, enquanto seu dado correspondente não estiver disponível. Essa característica é marcada na ficha, no momento de sua criação.

A ficha também contém o endereço da instrução a qual ela se destina no programa e em qual lado da instrução essa ficha deve ser emparelhada (se for necessário). Essa última informação é carregada para conferência e tratamento de eventuais equívocos no processo de emparelhamento, que podem ter sido gerados pelo compilador da solução. Todo erro é reportado a partir da criação de fichas de erros que são enviadas ao canal de saída da máquina.

Após a efetivação do emparelhamento de fichas, ou identificação da ficha como *bypass*, são gerados pacotes que contém todas as informações presentes nas fichas, porém com ambos os dados (e seus respectivos tipos) destinados a mesma instrução. Na saída da Unidade de Armazenamento de Instruções um novo tipo de pacote é gerado; ele possui as mesmas características do anterior, porém com todas as informações da instrução a ser executada, o *opcode*, a saída lógica a ser considerada e o endereço das instruções subsequentes à atual. Todas essas informações em conjunto são então encaminhadas à Unidade de Processamento para a execução da instrução (42). Uma ficha possui 96-bits de largura, e transita entre a Unidade de Processamento, a Fila de Fichas e a Unidade de Emparelhamento, possuindo o seguinte formato:

$$\langle \text{dado}(37\text{-bits}), \text{rótulo}(36\text{-bits}), \text{destino}(22\text{-bits}), \text{marcador}(1\text{-bit}) \rangle \quad (4.1)$$

Figura 4.1 – Diagrama ilustrativo de uma instrução

Fonte: Elaborada pelo autor.

A Figura 4.1 mostra a representação ilustrativa de uma instrução característica da MDFM.

Nessa representação, o quadrado representa a instrução que deve ser executada, onde o primeiro caractere indica qual a saída lógica que essa instrução espera, e os demais caracteres representam o *opcode* dessa instrução, no conjunto de instruções da arquitetura, conforme será descrito com mais detalhes a seguir. Cada uma das linhas presentes acima da instrução, representa as entradas de fichas nessa instrução. No exemplo, a entrada esquerda receberá uma ficha com o valor inteiro um e nome de ativação zero *. Da mesma forma, a entrada direita da instrução receberá o valor booleano falso, com o mesmo nome de ativação zero. As saídas de fichas dessa instrução seguem exatamente as mesmas características e encontram-se abaixo do quadrado que representa esta instrução.

É importante observar na figura, que as instruções nessa arquitetura podem fornecer até duas saídas lógicas distintas, como mencionado anteriormente. No entanto, em determinadas condições, apenas uma dessas saídas pode ser necessária para sua execução no programa. Por exemplo, em uma instrução de soma, será gerado apenas o resultado dessa operação (dado presente na ficha esquerda somado ao dado presente na ficha direita) na saída lógica esquerda da instrução. Já a operação de divisão por exemplo, irá gerar na saída lógica esquerda o resultado da operação de divisão e na saída da direita o resto da divisão entre esses números (43). Isso faz com que o conjunto de instruções possa ser mais enxuto e ainda seja capaz de garantir um nível de generalização que garante a computação de qualquer problema proposto. A característica que determina a saída lógica a ser considerada é representada pelo primeiro caractere mostrado na figura 4.1. Ele pode assumir 4 valores distintos, conforme descrito a seguir:

- **L**: Somente a saída lógica esquerda será considerada para ambas as saídas da instrução;
- **R**: Somente a saída lógica direita será considerada para ambas as saídas da instrução;
- **N**: A instrução em questão possui apenas uma saída lógica [†]. Portanto essa será a saída lógica considerada em ambas as saídas da instrução;
- **D**: Ambas as saídas lógicas serão consideradas em ambas as saídas da instrução [‡].

A Figura 4.1 mostra uma instrução de *Branch*, do conjunto de instruções da MDFM. Na figura é interessante observar que o AN das fichas de entrada de cada instrução devem ser iguais. A entrada esquerda da instrução de *Branch* recebe o dado a ser passado para

*indicado dentro dos caracteres < e >.

[†]como a instrução de soma citada como exemplo.

[‡]como a instrução de divisão, que poderá utilizar essa opção.

suas saídas. A entrada direita recebe um valor booleano. Se o valor da entrada direita dessa instrução for **verdadeiro**, a saída lógica esquerda da instrução será nula. Se o valor da entrada direita for **falso**, então o valor da entrada esquerda da instrução será copiado para ambas as saídas físicas da instrução.

Pode-se ver pela figura, que somente a saída lógica esquerda deverá ser considerada pela instrução, pela presença do identificador L, que antecede o nome da instrução na representação. Por esse motivo, ambas as saídas são idênticas, fichas com o valor um e nome de ativação zero, ou seja, uma cópia exata da ficha presente na entrada esquerda desta instrução. Observe que, dessa forma, se o valor da entrada direita da instrução for **verdadeiro** então ambas as saídas lógicas serão nulas, não sendo geradas novas fichas a partir dessa instrução.

4.2 Programas

O modelo do fluxo de dados tem uma implicação direta na representação de programas. Esse modelo apresenta os programas no formato de um grafo bidimensional, onde instruções que podem ser executadas paralelamente são dispostas uma ao lado da outra e instruções que dependem dos resultados anteriores [§], são dispostas uma abaixo da outra. Nesse grafo bidimensional, as instruções são representadas pelos nós do grafo e as arestas determinam os caminhos lógicos por onde as fichas deverão percorrer o grafo, para a correta execução dos programas.

A Figura 4.2 nos mostra um programa extremamente simples, apresentado somente para fins didáticos, a fim de exemplificar a execução de programas na máquina. É importante observar que ele utiliza somente um AN, pois não reutiliza nenhuma informação ao longo de sua execução, seja através de funções, laços ou recursividade. Na figura, as arestas de cor vermelha representam fichas do tipo *bypass*, e as arestas de cor preta, representam fichas *non-bypass*. Os nós circulares, representam dados adicionados ao programa de forma imediata, ou seja, adicionado explicitamente no código em linguagem montadora. Já os nós quadrados, representam as instruções presentes no conjunto de instruções da arquitetura, exatamente como a instrução apresentada na figura 4.1. É importante observar, para acompanhar a descrição, que o primeiro caractere visto nos nós, refere-se ao endereço da instrução, na memória do programa, que estará gravado na Unidade de Armazenamento de instruções, conforme será descrito em detalhes nos próximos sub-capítulos.

[§]instruções puramente sequenciais

Figura 4.2 – Representação ilustrativa de um programa Dataflow

Fonte: Elaborada pelo autor.

É importante ressaltar que a figura 4.2 trata apenas de uma representação ilustrativa. Através dessa representação, é possível analisar o fluxo de dados durante a execução de programas nesse modelo. Deve-se analisar esse fluxo, a partir da distribuição das novas fichas geradas pela execução de cada instrução, para as instruções subsequentes, segundo a lógica aplicada pelo compilador da solução.

A instrução 0 simplesmente cria um AN válido para o programa. Ela envia sua saída lógica da esquerda às instruções 1 e 2 como *bypass*, pois as instruções subsequentes não necessitam emparelhamento de dados para suas execuções. Em seguida, essas instruções atribuem o AN gerado, aos dados de entrada do programa, que são definidos explicitamente como os valores inteiros 7 e 2, respectivamente. A instrução 1, envia sua ficha de saída à instrução 4; essa ficha ficará aguardando emparelhamento com a ficha equivalente, que virá da instrução 3, como pode ser visto na figura. A instrução 2 envia sua ficha à instrução 3 como *bypass*, que simplesmente irá duplicar essa ficha para suas duas saídas. Entretanto, como somente a saída esquerda possui ligação no código, somente esta saída será gerada. Dessa forma, a ficha

presente na saída esquerda da instrução 3 fará o emparelhamento com a ficha que a estava aguardando, vinda da instrução 1. Com isso, os valores 2 e 7 serão multiplicados na instrução 4. A ficha à esquerda da instrução 4 será encaminhada à entrada esquerda da instrução 5, que irá apenas enviar as fichas de saída do programa, presente em sua entrada esquerda.

4.2.1 Sisal

```
EXPORT MAIN
FUNCTION MAIN(returns integer)
  X := 7;
  Y := 2;

  X*Y
END FUNCTION
```

Figura 4.3 – Programa Sisal

Fonte: Elaborada pelo autor.

Na Figura 4.3 pode ser visto o nível mais alto de abstração como podem ser escritos os programas para a MDFM. A linguagem em alto nível apresentada chama-se SISAL. Sua propriedade de atribuição única determina que cada variável pode ser utilizada apenas uma única vez no programa. Essa propriedade oferece uma semântica muito mais simples que a usual [¶] para o compilador explorar seu paralelismo. Obviamente esse paralelismo pode ser igualmente explorado em linguagens padrões, mas o processo de extração pode ser complexo e alguns princípios importantes podem se tornar obscuros algumas vezes, que são completamente aparentes em SISAL (44–46).

4.2.2 IF1

O processo de compilação do programa SISAL inicialmente faz a tradução do código fonte em uma linguagem intermediária chamada IF1. Trata-se de uma linguagem equivalente a uma linguagem macro *assembler* tradicional. O programa IF1 apresentado na figura 4.4 não se trata do mesmo programa gerado automaticamente no processo de compilação. Apenas por

[¶]Com relação à linguagens de programação convencionais, como a linguagem de programação C por exemplo.

```
(\I "TASS" "MULTIPLY")
!   multiplicacao de dois inteiros

!   inicializacao das variaveis
x   = (Data "I 7")
y   = (Data "I 2")

!   multiplicacao final
out = (MUL x y)

!   saida do valor inteiro final
      (OPT out "G 0")
```

Figura 4.4 – Programa IF1

Fonte: Elaborada pelo autor.

simplicidade são identificadas as variáveis presentes no programa SISAL, da figura 4.3. No processo de compilação são geradas variáveis genéricas, além de diversas linhas de códigos *assembler* redundantes. Observe que as variáveis X e Y são equivalentes às do programa SISAL visto anteriormente. Além disso, é importante identificar a presença das instruções MUL e OPT, nativas no conjunto de instruções da MDFM.

4.2.3 Linguagem Montadora

```
0 0NKILO 0LBY
0 1ROPT0 0LBYG0
1 0LSWA1 1LBY1 2LBY
1 1RSYN1 4LEWI7
1 2RSYN1 3LBYI2
1 3NDUP1 4REW
1 4NMULO 1LBY
```

Figura 4.5 – Programa em Linguagem Montadora

Fonte: Elaborada pelo autor.

Finalmente, a partir da abstração, de nível intermediário vista na figura 4.4, o compilador da MDFM faz a geração do arquivo em linguagem montadora padrão da máquina, conforme pode ser visto na figura 4.5. As instruções são formadas por até três seções distintas. Na primeira seção são determinados o endereço, seguido pelo *opcode* da instrução. Na segunda seção encontra-se o endereço da instrução esquerda, subsequente à instrução referenciada nessa linha, além da função de emparelhamento dessa ligação. No exemplo pode-se ver dois tipos de funções de emparelhamento, que são:

- **BY**: indica uma função que não necessita emparelhamento. São as instruções do tipo

bypass, mencionadas anteriormente; e

- **EW**: do inglês *Extract and Wait*. Indica uma função que necessita emparelhamento, determina que, se sua ficha parceira já estiver disponível na Unidade Emparelhamento, essa instrução deverá extrair essa ficha e formar o pacote de saída dessa unidade. Caso contrário, essa ficha deverá ficar aguardando emparelhamento.

A segunda seção pode conter a entrada de dados literais, que são dados definidos explicitamente no código em linguagem montadora, como pode ser vistos nas instruções de endereços 1 e 2 (no segmento 1 do programa), no programa de exemplo. Essa informação é precedida pelo tipo de dado referenciado (no exemplo, o tipo de dados é um inteiro, portanto definido pela letra I), seguida pelo dado especificado.

O Apêndice D apresenta um exemplo de execução de programa Dataflow, que realiza a integração analítica de uma função específica, mostrando o caráter paralelo das instruções representadas lado a lado, segundo o contexto geral dessa arquitetura.

4.3 Visão Geral

Uma representação ilustrativa da MDFM pode ser vista em detalhes na figura 4.6. A máquina é composta essencialmente por cinco unidades principais, que serão descritas em detalhes a seguir.

4.3.1 Fila de Fichas

Uma representação ilustrativa dessa unidade é mostrada na figura 4.7. A Fila de Fichas é composta por uma *FIFO* circular de armazenamento de 32K palavras / fichas. A *FIFO* de armazenamento da unidade possui 96-bits de largura, a fim de acomodar as fichas descritas pelo formato 4.1 que são geradas pela Unidade de Processamento, ou definida como ficha de início de programas. Dessa forma, a principal função da Fila de Fichas é atuar como um *buffer* para toda a estrutura da máquina, com a capacidade de fornecer uma ficha a cada ciclo do relógio, para a Unidade de Emparelhamento.

Figura 4.6 – Diagrama ilustrativo da MDFM

Fonte: GURD (17)

4.3.2 Unidade de Emparelhamento

A Unidade de Emparelhamento (figura 4.8) é composta por uma memória pseudo-associativa de 1.25M palavras de capacidade de armazenamento. Essa memória é utilizada para acomodar fichas que exigem pareamento, enquanto aguardam por suas fichas parceiras, mutuamente dependentes para a execução de suas instruções. O pareamento é realizado a partir de uma função *hash* característica, inicialmente baseada no rótulo (Nome de Ativação, Nível de Iteração e Índice) e no endereço da instrução a que se destinam as fichas. Os 22-bits que compõe o campo destino de uma ficha 4.1 é formado de acordo com a especificação a seguir:

Figura 4.7 – Diagrama ilustrativo da Fila de Fichas

Fonte: GURD (17)

$$\langle \text{endereço da instrução}(18\text{-bits}), \text{entrada esquerda/direita}(1\text{-bit}), \text{função de emparelhamento}(3\text{-bits}) \rangle \quad (4.2)$$

Instruções de entrada única recebem fichas do tipo *bypass*. Elas não precisam aguardar por fichas parceiras, e portanto não consomem os recursos da memória dessa unidade, passando diretamente da entrada para a saída dela. Quando ambas as fichas forem pareadas segundo sua função *hash* específica, a unidade ainda realiza uma última verificação para atestar que o compilador fez a marcação correta das fichas. Após realizar o emparelhamento, a unidade verifica se cada ficha emparelhada encontra-se no lado correto de destino. Essa verificação é feita a partir do identificador "entrada esquerda/direita", no campo de 22-bits de destino, da ficha. A mesma verificação é realizada na ficha da direita, se essa característica não for verificada, uma ficha de erro é gerada para a reportagem desse equívoco. Caso seja atestada a coerência do pacote de saída, com ambas as fichas localizadas e devidamente pareadas, elas serão disponibilizadas na forma de um pacote com 133-bits de largura, conforme a especificação a seguir:

Figura 4.8 – Diagrama ilustrativo da Unidade de Emparelhamento

Fonte: GURD (17)

$$\begin{aligned}
 &< \text{dado}(37\text{-bits}), \text{dado}(37\text{-bits}), \text{rótulo}(36\text{-bits}), \text{destino}(22\text{-bits}), \\
 &\qquad\qquad\qquad \text{marcador}(1\text{-bit}) > \quad (4.3)
 \end{aligned}$$

4.3.3 Unidade de Sobrecarga

Pelo fato de a Unidade de Emparelhamento ser composta por uma memória pseudo-associativa existe a possibilidade de fichas não parceiras serem destinadas ao mesmo endereço, na memória dessa unidade. Dessa forma, caso fichas não parceiras conflitem no mesmo endereço de memória especificado, a ficha que gerou o conflito, ao invés de ser alocada nesse

endereço, será encaminhada a Unidade de Sobrecarga. Observe que, para que as fichas sejam definidas como parceiras, ambas devem possuir exatamente o mesmo endereço de destino, além de possuir o mesmo rótulo, isso garante tratar-se de fichas destinadas à mesma instrução. A Unidade de Sobrecarga armazena essas fichas não pareadas em listas encadeadas, e verifica em períodos definidos, quando o endereço de seu destino já não está mais ocupado, isto é, quando a ficha que ocupava o endereço no momento do conflito, já foi pareada e deixou o módulo definitivamente. Com isso, as fichas destinadas a esse determinado endereço pode finalmente ocupar a memória da Unidade de Emparelhamento para aguardar por sua ficha parceira.

4.3.4 Unidade de Armazenamento de Instruções

Figura 4.9 – Diagrama ilustrativo da Unidade de Armazenamento de Instruções

Fonte: GURD (17)

A Unidade de Armazenamento de Instruções (figura 4.9) é composta por uma memória de acesso aleatório, que contém o código em linguagem montadora do programa que está sendo executado na máquina. Os pacotes que chegam a essa unidade possuem o endereço da instrução a que se destinam. Conforme especificado no formato 4.1, esse atributo indica o endereço da instrução dentro dessa memória de acesso aleatório, a qual armazena todas as

informações pertinentes dessa instrução. As instruções podem conter os seguintes formatos:

$$< opcode(10\text{-bits}), destino\ esquerdo(22\text{-bits}) > \quad (4.4)$$

$$< opcode(10\text{-bits}), destino\ esquerdo(22\text{-bits}), destino\ direito(22\text{-bits}) > \quad (4.5)$$

$$< opcode(10\text{-bits}), destino\ esquerdo(22\text{-bits}), dado\ literal(37\text{-bits}) > \quad (4.6)$$

Observe que as instruções possuem os formatos especificados pelo código em linguagem montadora do programa. A principal entidade presente nessas especificações é o *opcode*, que determina a instrução que o pacote deve executar. Essa informação é composta por um caractere que determina a saída lógica definida para essa instrução, seguida por três caracteres que definem a instrução que deve ser executada (como por exemplo: adição, multiplicação, duplicação, desvio condicional, definidas pelos identificadores ADD, MUL, DUP, BRR (47), respectivamente). As informações de destino, definidas nos padrões acima, determinam os endereços das fichas subsequentes à ficha que está requisitando a operação, como pode ser visto claramente através da caracterização padrão dos programas (formato de grafo exibido anteriormente). Dessa forma, a partir do resgate dessas informações na memória dessa unidade, é composto um novo pacote, com 165-bits de largura que possui o seguinte formato:

$$\begin{aligned} < dado(37\text{-bits}), dado(37\text{-bits}), opcode(10\text{-bits}), rotulo(36\text{-bits}), \\ &dest.\ esquerdo(22\text{-bits}), dest.\ direito(22\text{-bits} [opc]), marcador(1\text{-bit}) > \quad (4.7) \end{aligned}$$

4.3.5 Unidade de Processamento

Os pacotes com ambos os dados emparelhados, instrução e destinos, são enviados à Unidade de Processamento (figura 4.10). Algumas instruções são processadas em um pre-processador existente nessa unidade, mas a grande maioria dos pacotes são enviados a primeira Unidade Funcional disponível para execução, através de um barramento de distribuição. As unidades funcionais são verificadas em sequência, e diante da ocorrência de disponibilidade para execução, a unidade específica recebe o pacote recém chegado. Caso não exista nenhuma unidade funcional disponível, esse pacote aguardará para nova verificação no barramento de distribuição, no próximo ciclo do relógio (48). As instruções são executadas independentes

umas das outras, ou seja, não existe nenhuma interação entre essas unidades. Por esse motivo, cada ficha possui todas as informações necessárias para a execução da instrução que ela representa. Instruções que exigem a criação de um novo AN, fazem com que a Unidade Funcional realize uma requisição explícita para a Unidade de Processamento, responsável por organizar essa informação, já que os ANs devem ser únicos e disponibilizados em uma sequência numérica definida. As saídas das Unidades Funcionais compartilham o mesmo barramento que é ligado ao módulo de *Switch*, responsável pela entrada e saída da máquina. Caso a ficha resultante não seja uma ficha de saída da máquina (fichas de erros ou fichas de resultados), esse *Switch* a enviará de volta à Fila de Fichas, reiniciando o ciclo novamente.

Figura 4.10 – Diagrama ilustrativo da Unidade de Processamento

Fonte: GURD (17)

CAPÍTULO 5

PIFLOW - Primeira Versão

O principal objetivo nessa etapa do desenvolvimento foi a realização de uma implementação fiel da MDFM com a utilização de tecnologias modernas. Na máquina de origem, a capacidade de armazenamento nas unidades de memória, como a Unidade de Emparelhamento, era um grande gargalo (49–50). Esse é o principal motivo que obrigou a criação da Unidade de Sobrecarga nessa máquina. Como capacidade de armazenamento não oferece um grande problema nos dias atuais, essa unidade foi deixada de lado nessa primeira versão. A implementação das outras unidades foi realizada da forma como descrita a MDFM no capítulo 4.

A Fila de Fichas é composta por uma FIFO com capacidade para 4K-fichas. A Unidade de Emparelhamento possui uma memória RAM e FIFOs de entrada e saída conectadas entre si para fichas *Bypass*. A Unidade de Armazenamento de Instruções possui outra memória RAM que armazena o código gerado pelo *Parser*, descrito em detalhes no apêndice A. A Unidade de Processamento possui um conjunto de Unidades Funcionais, além de um circuito específico, destinado para a obtenção de um melhor aproveitamento dessas unidades, que foram implementadas em forma de um *pipeline* com cinco estágios, para as principais instruções.

A comunicação entre cada unidade é realizada através de duas interfaces de comunicação nativas do BSV. Uma delas chama-se *Get*. Esse tipo de interface tem por objetivo fornecer uma *FIFO* de comunicação para o outro tipo de interface utilizado. O outro tipo de interface é chamada *Put*, que tem por objetivo resgatar as *FIFOs* presentes na outra interface (51). Essas entidades conectam as *FIFOs* nativas da linguagem e garantem que as informações serão enviadas de um módulo ao outro imediatamente, diante da ocorrência de dados na *FIFO* do tipo *Get*.

Inicialmente, foram utilizadas as *FIFOs* de tamanho definido padrões do BSV. Porém, pôde-se observar, que isso gerava certos conflitos durante a execução de determinados comportamentos da máquina, devido ao controle de acesso para leitura e escrita de dados nessas entidades. Esses conflitos criaram certo atraso para a comunicação entre as unidades. Portanto, passou-se a utilizar *FIFOs* do tipo *Bypass* - presentes na biblioteca *SpecialFIFOs* do BSV - que permitem operações de escrita e leitura nessas entidades dentro do mesmo ciclo do

relógio. Foi definido como padrão, que as *FIFOs* de comunicação entre as unidades possuam comprimento $4 \times n$, onde n é a quantidade de Unidades Funcionais presentes na Unidade de Processamento.

Figura 5.1 – Representação ilustrativa da arquitetura

Fonte: Elaborada pelo autor.

5.1 Visão Geral

Em linhas gerais, toda a estrutura básica da MDFM foi utilizada para a construção da PIFLOW, como pode ser visto na figura 5.1. Nessa primeira fase do desenvolvimento, a PIFLOW utilizou duas abordagens distintas para a implementação da MDFM. A necessidade de uma releitura do projeto inicial se deu pelo fato que as Unidades Funcionais foram desenvolvidas sob o mesmo domínio de *clock* que o restante da máquina. Dessa forma, o tempo para a execução de uma instrução passou a ser equivalente ao tempo para as fichas percorrerem todas as unidades da arquitetura, inviabilizando essa abordagem. A figura 5.1 apresenta a visão geral da primeira etapa de desenvolvimento implementada e será descrita em detalhes a seguir. A segunda etapa do projeto será apresentada no capítulo 6.

5.2 Fila de Fichas

Figura 5.2 – Representação ilustrativa da Fila de Fichas

Fonte: Elaborada pelo autor.

A Fila de Fichas (figura 5.2) é composta por uma *FIFO* de tamanho definido; ela possui uma capacidade de 4K-fichas posições válidas para armazenamento. Diferentemente da MDFM, foi realizada uma ligação direta entre as *FIFOs* de entrada dessa unidade com sua *FIFO* de saída, evitando a passagem das fichas pelo seu armazenamento interno, quando possível. Isso garante certo ganho em desempenho, principalmente no início e final dos programas, trechos onde não existe uma quantidade significativa de paralelismo, ou mesmo para programas que se apresentam fundamentalmente sequenciais.

Observe, portanto, que foram exploradas as principais alternativas para ganho em desempenho na passagem das fichas através das unidades, evitando a realização de uma quantidade excessiva de operações complexas, a fim de garantir melhor aproveitamento do percurso ao longo do anel. Essa característica tem o objetivo de permitir o aumento da quantidade de fichas para execução, em menor espaço de tempo. Na PIFLOW, as fichas - entidades que transitam por essa unidade - são definidas da seguinte maneira.

$$< dado(37\text{-bits}), \text{rótulo}(12\text{-bits}), destino(20\text{-bits}), marcador(1\text{-bit}) > \quad (5.1)$$

Observe que nessa etapa do desenvolvimento, apenas o AN foi utilizado para fornecer o caráter dinâmico da arquitetura, com relação a MDFM, ou seja, não foram utilizadas as informações de índice e nível de iteração, de forma que o rótulo é composto apenas pelo Nome de Ativação. Uma outra diferença com relação a MDFM é a largura do endereço, que na arquitetura PIFLOW possui apenas 16-bits.

Na configuração do módulo, além das interfaces de comunicação que interligam essa unidade com a Unidade de Processamento e a Unidade de Emparelhamento, ainda existe um método do tipo *Action*, que tem por objetivo a adição manual de uma ficha na unidade. Esse método é usado, por exemplo, para adição das fichas iniciais para execução de programas, que devem ser inseridas explicitamente na máquina. Ele apenas adiciona a informação vinda como o parâmetro de sua definição, no armazenamento interno da unidade. É importante observar que esse atributo é utilizado para comunicação com o *Switch*, característico da MDFM, que no presente projeto é representado pelo *Top Level* da aplicação.

Pode-se ver pela figura 5.2 que o módulo é bastante simples, contendo apenas as *FIFOs* de entrada e a de saída, além da *FIFO* de armazenamento interno que caracteriza a memória da unidade. Existem regras distintas para o controle de cada *FIFO* de entrada do módulo; elas são executadas de maneira intercalada em cada ciclo do relógio, uma vez que disputam acesso pelo buffer de saída da unidade. O controle dessas regras é realizado a partir de um

módulo EHR (descrito em detalhes no apêndice B) do tipo booleano, onde cada uma dessas regras concede a execução da outra, diante da ocorrência de dados em sua FIFO de controle particular. Caso não haja novas fichas na outra ficha de entrada, é mantida a execução da regra que controla o *buffer* corrente. O módulo EHR garante a análise de toda sua estrutura antes do início de um ciclo do relógio, o que permite a execução de uma ou outra regra a partir de seus predicados, baseados nessa estrutura. As fichas são armazenadas na memória interna da unidade somente quando a FIFO de saída encontra-se cheia. Dessa forma, fichas armazenadas no compartimento interno da unidade são enviadas ao *buffer* de saída somente quando não existem novas fichas nos buffers de entrada.

É interessante observar o controle que a arquitetura fornece para a implementação. Caso haja uma ficha fundamental para execução, que esteja armazenada no compartimento interno dessa unidade, a execução do programa ficará prejudicada e cada vez menos fichas são geradas e enviadas à Fila de Fichas. Isso garante o sucesso desse tipo de implementação particular, uma vez que apenas as fichas que tiverem menor prioridade para execução imediata irão permanecer por mais tempo armazenadas na memória dessa unidade, como por exemplo fichas presentes no arco dinâmico da máquina (4). Essa característica é interessante de ser implementada, pois é preferível manter a memória dessa unidade mais ocupada do que a memória da Unidade de Emparelhamento, essencialmente mais cara.

5.3 Unidade de Emparelhamento

A Unidade de Emparelhamento (figura 5.3), tem o objetivo de reunir fichas parceiras advindas da Fila de Fichas, montá-las em um pacote característico e encaminhá-lo à Unidade de Armazenamento de Instruções. Sua estrutura é formada pelos *buffers* de entrada e saída da unidade; a memória RAM de armazenamento interno, destinada a fichas ainda não pareadas, e um registrador de estado, que ativa a regra para leitura de um endereço da memória, ou a regra de retorno da leitura. Em linhas gerais, trata-se de uma unidade bastante simplificada com a relação àquela existente na MDFM, uma vez que existiam diversas outras operações de emparelhamento implementadas, não previstas nesse protótipo inicial. O objetivo principal do presente projeto é apenas oferecer uma porta de entrada já tão bem estabelecida como a da máquina de referência.

A memória de armazenamento interna da unidade possui endereço com largura de 20-bits e largura para armazenamento de 71-bits, ou seja, a largura da estrutura de uma ficha mais 1-

Figura 5.3 – Representação ilustrativa da Unidade de Emparelhamento

Fonte: Elaborada pelo autor.

bit, utilizado para marcar se existem dados válidos em um endereço particular da memória, ou não. Note que os 20-bits de largura de endereçamento são necessários para a caracterização da função *hash* utilizada nesse protótipo, obtido a partir da concatenação do AN com o Endereço de destino da ficha. Essa largura proporciona $2^{20} = 1048576$ posições válidas para endereçamento*.

Fichas do tipo *Bypass*, são marcadas com essa característica na composição de suas fichas, nas informações presentes no campo destino de uma ficha, especificado no formato 5.1, o que

*É importante observar que a capacidade de armazenamento da Unidade de Emparelhamento é impraticável com a utilização apenas de FPGAs comerciais. A implementação proposta foi utilizada fundamentalmente para fins de simulações e o desenvolvimento prevê a utilização de memórias RAM comerciais, em conjunto com FPGAs, para atingir essa grande capacidade de armazenamento.

torna possível que essa verificação seja realizada pelo predicado da regra responsável por esse tipo de operação. Essa é outra característica interessante que a arquitetura proporciona para implementação a partir do BSV. Caso as fichas não fossem marcadas com essa opção, deveria ser utilizado um outro registrador de estado para essa verificação, já que existe concorrência entre as regras para cada tipo de opção de emparelhamento, sob o *buffer* de saída da unidade. Uma vez que a regra destinada para fichas *Bypass* é ativada, ela somente resgata a informação presente na *FIFO* de entrada da unidade, monta o pacote de saída característico e envia esse pacote para a saída do módulo. Os pacotes característicos que saem dessa unidade tem o seguinte formato:

$$\langle \text{dado esq}(37\text{-bits}), \text{dado dir}(37\text{-bits}), \text{rótulo}(12\text{-bits}), \text{destino}(20\text{-bits}), \text{marcador}(1\text{-bit}) \rangle > \quad (5.2)$$

Note que a única diferença desse pacote com relação a uma ficha, é a presença de ambos os dados necessários para execução da instrução especificada no campo *destino*. Se a ficha for do tipo *Bypass*, será adicionado um dado com valor nulo no campo destinado ao dado direito, pois o pacote de saída dessa unidade deve possuir exatamente o formato indicado acima, mesmo pela característica da tipagem existente no BSV.

Fichas com o tipo de emparelhamento *Extract and Wait* - mencionadas no capítulo anterior - devem analisar o endereço da memória RAM composto pela concatenação do AN com o endereço de destino truncado da ficha corrente. O truncamento do endereço foi necessário para completar os 20-bits de endereçamento da memória (já muito grande), uma vez que o AN possui 12-bits e o endereço da instrução possui 16-bits; como os programas utilizados para validação do protótipo são bastante simples, não houve perda alguma nesse truncamento. As operações para fichas desse tipo são desdobradas em duas regras distintas. Da mesma forma que para fichas do tipo *Bypass*, o predicado da regra verifica se a ficha atual é do tipo *Extract and Wait*. Além disso é verificado se o registrador de estado permite acesso a essa regra. Foi necessária certa precaução na execução dessas regras distintas, pois não existe a possibilidade de ler e escrever na memória RAM característica do BSV (chamada BRAM) simultaneamente. Essa impossibilidade existe pois a porta de entrada do módulo que constitui essa memória possui uma *FIFO* de tamanho definido, nativa da linguagem, e portanto, não existe a possibilidade de ler e escrever neste elemento no mesmo ciclo do relógio. Portanto essa primeira regra apenas faz a concatenação necessária, seguida pela requisição da informação contida no endereço particular da memória, além de alterar o estado do registrador que ativa a

regra seguinte. Para o resgate das informações necessárias para a concatenação, a ficha não é removida do *buffer* de entrada da unidade, pois será utilizada na próxima regra. Portanto, esse ciclo do relógio é utilizado exclusivamente para a realização da leitura da memória; somente ao final do próximo ciclo a próxima ficha poderá ser lida.

Com a leitura do conteúdo presente no endereço especificado devidamente realizada, no próximo ciclo do relógio a regra seguinte é ativada. Nessa regra, o conteúdo lido no endereço é então adicionado a uma "variável" do tipo *Maybe* - própria do BSV (esse tipo de dados é acompanhado por uma *flag* que tem o objetivo de informar se o dado contido no registrador de sua composição é válido ou não). Por esse motivo, a largura dos dados para a memória RAM dessa unidade é do tamanho de uma ficha mais 1-bit, bit este responsável pelo armazenamento dessa *flag* de estado. Após a leitura da ficha de entrada da unidade, ela finalmente é removida do *buffer*, permitindo a leitura de novas fichas no próximo ciclo do relógio. Com o dado lido da memória, é verificada se a informação é válida; em caso positivo, a informação será desempacotada na estrutura de uma ficha e será montado o pacote de saída da unidade, com ambos os dados emparelhados. No endereço lido da memória é adicionada uma ficha não válida, para permitir a adição de novas fichas nesse endereço, para emparelhamento de outras fichas. Caso a informação lida da memória não seja válida, a ficha que busca emparelhamento será adicionada a uma variável do tipo *Maybe* com sua *flag* válida e gravada na memória da unidade nesse endereço particular, para aguardar por sua ficha parceira.

Observe que o ato de manter o *buffer* de entrada ocupado tem o objetivo apenas de controlar o fluxo de informações que transitam dentro da unidade. Se não houvesse o controle através do registrador de estado mencionado, ou ainda se as informações da ficha de entrada fossem removidas imediatamente do *buffer* de entrada, ao chegarem duas fichas do tipo *Extract and Wait* subsequentes, no instante que a primeira ficha estivesse resgatando a informação contida na memória, a segunda ficha estaria requisitando a leitura de um novo endereço, o que geraria um conflito desnecessário. Por outro lado, se a segunda ficha mencionada fosse do tipo *Bypass*, ainda assim poderia haver um conflito, pois a regra destinada ao resgate da memória poderia ter encontrado uma ficha para emparelhamento e com isso deveria enviar o pacote resultante ao *buffer* de saída da unidade, concorrendo com a ficha *Bypass* para essa operação. Portanto, a melhor solução encontrada foi manter a entrada da unidade em espera, até que tenha sido resolvida toda solução de emparelhamento.

5.4 Unidade de Armazenamento de Instruções

Figura 5.4 – Representação ilustrativa da Unidade de Armazenamento de Instruções

Fonte: Elaborada pelo autor.

A Unidade de Armazenamento de Instruções (figura 5.4) tem o objetivo de armazenar o código em linguagem montadora interpretado pelo *Parser*, em uma memória de acesso aleatório, de forma que os pacotes vindos da Unidade de Emparelhamento tenham a possibilidade de requisitar qualquer endereço dessa memória para o resgate de todas as informações da instrução especificada. Sua estrutura é bem semelhante a da Unidade de Emparelhamento: ela possui os *buffers* de comunicação da unidade, que são ligados à Unidade de Emparelhamento e à Unidade de Processamento; sua memória RAM com 16-bits de endereçamento [†] e um registrador de estados para o controle de regras.

As ações previstas pela unidade são bem simples e quaisquer pacotes que chegam a ela levam um período fixo de 2 ciclos do relógio para resgatar todas as informações necessárias. Bem como a Unidade de Emparelhamento, a Unidade de Armazenamento de Instruções possui uma quantidade de memória impraticável com a utilização apenas de FPGAs comerciais, sendo usada nessa etapa apenas para fins de simulação. Porém, os programas utilizados para

[†]Ou seja, $2^{16} = 65536$ endereços válidos

a validação da máquina apresentam poucas linhas de código em linguagem montadora, de forma que grande parte dessa memória não foi utilizada, mesmo nas simulações.

Existem apenas duas regras para descrever todas as ações dessa unidade e o registrador de estado é o elemento responsável por ordenar as atividades. A primeira regra apenas resgata o pacote presente no *buffer* de entrada da unidade (sem removê-lo dessa *FIFO*). Esse pacote carrega consigo o endereço da instrução que precisa resgatar na memória, presente no campo *destino*, descrito no pacote. Com essa informação, é solicitada a leitura dos dados presentes no endereço especificado. Vale notar que o campo *destino* mencionado, contém 3 informações distintas: o endereço, o lado da ficha que realizou o emparelhamento, e a função de emparelhamento das fichas. A informação com o lado da ficha que realizou o emparelhamento é utilizada para verificar se a instrução corrente possui um dado literal em sua composição. O pacote de saída será montado de acordo com essa especificação, ou seja, a ficha que irá compor o pacote de saída será alocada na posição especificada por esse campo e o dado literal no outro lado. Com a solicitação de leitura realizada, a regra altera o estado do registrador de controle para que a próxima regra possa ser executada.

A segunda regra do módulo remove a ficha presente no *buffer* de entrada da unidade e resgata as informações presentes no endereço especificado, onde verifica inicialmente a saída lógica que a instrução deve considerar. Com essa informação, será possível identificar fichas que possuam dados literais presentes em sua composição, de acordo com a especificação adicionada pelo *Parser*, descrito no Apêndice A. Dessa forma, se essa informação for maior que 3, então haverá um dado literal para o pacote de saída da unidade. Neste caso, os dados de saída serão ordenados de acordo com a informação presente no pacote de entrada, no campo que determina o lado da ficha que realizou o emparelhamento, já que a informação vinda nessa ficha deverá ocupar esse lado específico determinado. Por fim, será montado o pacote de saída da unidade. Se a saída lógica for menor que 3, então os dados serão mantidos em suas posições de chegada e será montado o pacote de saída da unidade imediatamente. Após a realização dessa verificação, existirá um pacote montado que deverá ser enviado ao *buffer* de saída. Este pacote possui o seguinte formato:

$$\begin{aligned} < \text{dado esq}(37\text{-bits}), \text{dado dir}(37\text{-bits}), \text{opcode}(10\text{-bits}), \text{rótulo}(12\text{-bits}), \\ &\quad \text{destino esq}(20\text{-bits}), \text{destino dir}(21\text{-bits}), \text{marcador}(1\text{-bit}) > \quad (5.3) \end{aligned}$$

Observe que existe 1-bit a mais no destino direito. Esse bit tem o objetivo de informar imediatamente à Unidade Funcional que irá executar esse pacote, se existe um destino direito

no presente pacote. Esse bit de controle foi necessário pois não existe uma identificação imediata dessa característica no pacote padrão da MDFM.

Ao final da regra de envio do pacote de saída, o registrador de controle retorna ao estado inicial, o que irá permitir que a primeira regra seja executada novamente, reiniciando o ciclo. A memória BRAM utilizada - nativa do BSV - permite que a leitura seja realizada dentro do mesmo ciclo de relógio, o que garante que essa unidade irá consumir necessariamente 2 ciclos para a saída de novos pacotes, o que se mostrou como um gargalo do protótipo, uma vez que o comportamento ideal é o consumo de apenas 1 ciclo do relógio para saída de informações nos *buffers* de saída de cada unidade. Inicialmente foi promovida a utilização de duas portas de entrada para a leitura na memória RAM da Unidade de Armazenamento de Instruções. Porém essa característica aumentou consideravelmente a complexidade da unidade, aumentando portanto o caminho crítico do circuito, fato que motivou a simplificação de acordo com a descrição acima. Esse gargalo foi uma das motivações para o desenvolvimento da próxima versão do protótipo que será descrita em detalhes no capítulo 6. A arquitetura proposta para utilização de duas portas de leitura da memória BRAM pode ser vista na figura 5.5:

Figura 5.5 – Representação ilustrativa da primeira versão da Unidade de Armazenamento

Fonte: Elaborada pelo autor.

5.5 Unidade de Processamento

Figura 5.6 – Representação ilustrativa da Unidade de Processamento

Fonte: Elaborada pelo autor.

De acordo com a descrição até aqui, pode-se observar que todas as informações necessárias para a execução de instruções estão presentes nos pacotes que chegam à Unidade de Processamento. Essa unidade tem por objetivo receber os pacotes vindos da Unidade de Armazenamento de Instruções, adicionar esses pacotes à primeira Unidade Funcional livre para execução, e resgatar as fichas geradas por essas Unidades para enviá-las à Fila de Fichas. Uma característica marcante dessa primeira versão da PIFLOW com relação a MDFM, foi o aumento da largura de banda entre essa unidade e a Fila de Fichas. Foram criadas duas interfaces de comunicação entre essas unidades, cada qual relacionada ao lado do anel que deve conter cada uma das duas fichas possíveis de serem geradas pelas Unidades Funcionais.

Como pode ser transmitida apenas uma ficha em cada ciclo do relógio - pelo fato de serem conectadas por *FIFOs* e essas entidades não permitirem a escrita de mais de uma informação dentro do mesmo ciclo - foi explorado esse caráter "dual" promovido pela arquitetura da MDFM. A implementação dessa característica apresentou resultados bastante positivos e teve impacto decisivo para a evolução da PIFLOW.

Esse módulo é composto pelas *FIFOs* de comunicação entre essa unidade, a Unidade de Armazenamento de Instruções e a Fila de Fichas. O módulo ainda possui diversos outros registradores de controle para organizar as operações que devem ser realizadas. Um registrador de estado é utilizado para determinar o primeiro ciclo do relógio, a fim de definir uma identificação única para cada Unidade Funcional. Um módulo EHR armazena o Nome de Ativação, que, por definição, trata-se de um registrador numérico, sequencial e único, características que podem ser garantidas por esse tipo de registrador especial. Um registrador booleano, utilizado para verificar quando a primeira ficha do programa chega a unidade, é usado para registrar o primeiro Nome de Ativação válido para esse programa. O módulo ainda contém a instância de todas as Unidades Funcionais, além de um par de *FIFOs* referentes a cada uma dessas instâncias, que tem o objetivo de armazenar as fichas de saída de cada unidade, para serem conectadas aos *buffers* de saída. As Unidades Funcionais e *FIFOs* de saída, são definidas dentro de vetores característicos do BSV, que permitem o tratamento de cada um desses elementos através de um índice específico. O tamanho desse vetor é definido por um valor constante, o que permite a alteração da quantidade de unidades funcionais em cada processo de compilação. Os *buffers* de saída das Unidades Funcionais tem o objetivo de armazenar todas as fichas resultantes de cada uma dessas unidades, resgatando as informações imediatamente diante da ocorrência de novas fichas nas suas saídas, liberando-as para receber novas instruções para execução.

Todas as regras dessa unidade passaram por grandes adaptações ao longo do desenvolvimento. Serão descritos a seguir os principais aspectos de cada regra, na sequência que estão dispostas no código BSV do projeto. Lembre-se que as regras são atômicas nessa linguagem, sendo executadas em todo ciclo que suas condições de execução forem atendidas. As saídas esquerda e direita de cada Unidade Funcional é conectada a sua *FIFO* específica de saída por um conjunto de regras idênticas. Essas regras não possuem nenhum predicado para execução e são definidas em um laço padrão do BSV. Cada iteração desse laço define a regra que irá controlar a *FIFO* e a Unidade Funcional de seu nível de iteração. Por não possuírem predicados, essas regras **não** serão executadas somente no caso onde as *FIFOs* que elas controlam estiverem cheias, ocasionando o bloqueio para execução de novas instruções da Unidade Funcional que abastece esse *buffer*.

A quantidade fixa n de *FIFOs* presentes nas saídas das Unidades Funcionais devem estar

conectadas às duas *FIFOs* (esquerda e direita) da Unidade de Processamento. Foi necessária a implementação de um circuito de prioridades para a realização dessa operação em uma regra. Um registrador numérico do tipo EHR armazena a última Unidade Funcional que teve suas fichas enviadas à Fila de Fichas, e a partir do índice dessa unidade específica é verificado se existem fichas válidas em cada uma das unidades subsequentes, de maneira circular. Se, por exemplo, existirem 5 Unidades Funcionais e o registrador de posição armazenar o índice 2 (como a unidade verificada no último ciclo) serão verificadas todas as outras unidades na seguinte sequência: 3 - 4 - 5 - 0 - 1 - 2. Na ocorrência de fichas em qualquer destas *FIFOs* na sequência indicada, a verificação é parada e a Unidade Funcional desse índice determinado será escolhida para o resgate de suas fichas. Essa informação fica armazenada no mesmo registrador de posição EHR verificado, que irá determinar o predicado das regras que fazem a comunicação efetiva entre as *FIFOs* de saída das Unidades Funcionais e as *FIFOs* de saída da Unidade de Processamento. Essas regras são definidas no mesmo laço que define as regras para a comunicação entre as Unidades Funcionais e suas *FIFOs* equivalentes. A condição verifica se o registrador de posição possui em seu valor o índice equivalente à iteração desse laço. Dessa forma, apenas um par dessas regras serão executadas em cada ciclo do relógio, bem como será executada a regra que implementa o circuito de prioridades que irá determinar a Unidade Funcional que será analisada no próximo ciclo.

Existem duas regras particulares que são utilizadas para o controle da unidade. Uma delas é executada apenas no primeiro ciclo do relógio, quando a máquina é iniciada. Essa regra tem por objetivo definir uma identificação única para cada Unidade Funcional. Essa característica foi utilizada para o resgate de informações de unidades específicas durante a validação do protótipo. A outra regra com esse caráter é executada no instante que é identificada a ocorrência da primeira ficha que chega na unidade. Essa regra tem o objetivo de registrar o primeiro Nome de Ativação válido na máquina, que será definido pela primeira ficha adicionada explicitamente para o programa corrente. Isso permite que o programador tenha possibilidade de testar valores de Nomes de Ativação diferentes, para validar o comportamento definido pelo compilador da linguagem, por exemplo.

A última regra definida neste módulo tem o objetivo de adicionar qualquer pacote presente no *buffer* de entrada da unidade e enviá-lo a primeira Unidade Funcional livre para execução. Inicialmente é verificado o *Opcode* da instrução que esse pacote carrega: se a instrução for destinada a manipulação do Nome de Ativação, essa operação será realizada nesse momento, diretamente na estrutura do pacote. Por exemplo, considere a instrução GAN que faz a geração de um novo Nome de Ativação (informação presente em um registrador do tipo EHR dessa unidade) (47). Na ocorrência de um pacote que deve executar essa instrução, o próximo Nome

de Ativação será adicionado no campo de dados do pacote corrente, antes dele ser atribuído a uma Unidade Funcional específica. Após a realização desse pré-processamento, é verificada a disponibilidade de cada Unidade Funcional de maneira sequencial - em um laço característico do BSV - implementada da mesma forma que o circuito para saída das fichas da unidade, descrito anteriormente. Um registrador armazena o índice da Unidade Funcional que foi alimentada no último ciclo do relógio, de forma que o circuito verifica a disponibilidade das próximas Unidades Funcionais de maneira sequencial, a partir do valor contido nesse registrador. Se uma determinada unidade estiver pronta para execução, a verificação é parada e o pacote é atribuído a essa unidade a partir do método de entrada desta. Se não for encontrada nenhuma Unidade Funcional livre após a execução desse laço, então o pacote será mantido no *buffer* de entrada da Unidade de Processamento, que realizará essa mesma operação no próximo ciclo do relógio.

5.6 Unidades Funcionais

Figura 5.7 – Representação ilustrativa das Unidades Funcionais

Fonte: Elaborada pelo autor.

As Unidades Funcionais (figura 5.7) apresentam uma estrutura de pipeline de quatro estágios, na maioria das instruções[‡]. No instante que ela recebe um pacote, ele é atribuído a um registrador principal. Em seguida, esse pacote fornecerá o *Opcode* da operação que será executada. É interessante observar que essa arquitetura possui caráter estritamente RISC, com operações extremamente simples por instrução, de forma que foi possível garantir facilmente um único ciclo do relógio para a execução desse estágio. Após a execução da instrução, a unidade irá analisar quais saídas lógicas a instrução exige para a geração das fichas subsequentes – já que todas as fichas são marcadas com essa informação. Com isso, temos uma ou duas fichas

[‡]Não foi implementada a operação em um *pipeline* explicitamente. A estrutura encontra-se dividida em estágios, porém é necessário que seja finalizada a execução de uma instrução inteiramente, até que a unidade funcional esteja pronta para receber novos pacotes.

prontas para retornar ao *anel*. Nesse estágio a Unidade Funcional ficará aguardando até que a Unidade de Processamento resgate essas fichas, para que seu estado possa ser alterado para, "pronta para execução", novamente. É importante observar que somente algumas instruções do Manual Básico de Programação da MDFM (47) foram implementadas, pois os programas utilizados para validação não exigiram as operações mais diversas propostas pela arquitetura de origem.

Na estrutura do módulo existe um registrador de estado, para controle das regras. Ambas as saídas (esquerda e direita) ficam armazenadas em *FIFOs*, necessárias para a utilização dos métodos de comunicação nativos do BSV, utilizados em todas as outras unidades. Um registrador para armazenamento da identificação da Unidade Funcional, informação mais utilizada no processo de simulação para geração de relatórios, porém, pode ser utilizada para controle de prioridades, por exemplo. Um registrador do tipo booleano identifica se a Unidade Funcional está pronta para execução, analisada através do método de inserção de fichas na unidade. Existem diversos registradores que armazenam os dados e tipos de dados de cada saída lógica da unidade. Eles são disponibilizados nesse tipo de entidade para que possam ser gerenciados por diferentes regras, nos diversos estágios do *pipeline*. Além disso, existe um outro registrador que armazena o Nome de Ativação para as fichas que serão geradas, pois essa informação pode ser manipulada através de instruções específicas, portanto utilizadas em regras distintas. E por fim, existe um registrador que armazena o pacote corrente da unidade; ele não é modificado em nenhum estágio e apenas fornece qualquer informação que uma instrução, ou processo necessite para execução de suas atividades.

Existem três métodos de comunicação no módulo, para realizar as operações previstas pela Unidade de Processamento. A primeira delas refere-se à definição da identificação da Unidade Funcional, e é um método do tipo *Action*: ele recebe a identificação específica como parâmetro e atribui esse valor ao registrador que armazena essa informação. Um método do tipo *value* retorna para a Unidade de Processamento se a Unidade Funcional específica está ocupada, ou seja, se ela está executando alguma instrução no instante verificado. E o último método é do tipo *ActionValue*: ele recebe um pacote como parâmetro e retorna um valor booleano. Esse método verifica se a Unidade Funcional está pronta para execução, e em caso positivo, atribui o pacote vindo como parâmetro ao registrador que carrega o pacote corrente, altera o estado da unidade informando não estar pronta para execução, altera o registrador de controle das regras e retorna Verdadeiro. Caso a unidade não esteja pronta para execução, ele apenas retorna Falso para a entidade que chamou esse método.

Para a realização das operações previstas para o módulo, existe um conjunto de regras que são ativadas por seus predicados, através da verificação do registrador de controle das

regras e da análise das informações contidas no registrador que carrega o pacote corrente. O estágio de execução da instrução foi dividido em diversas regras, cada qual responsável para a execução de uma instrução particular. Essa característica foi utilizada, para evitar um aumento muito grande no caminho crítico de toda a solução, permitindo que todo esse estágio pudesse ser executado em um único ciclo do relógio. Instruções mais complexas e que demandam maior esforço computacional, foram divididas em mais de um ciclo do relógio, utilizando as informações presentes nos próprios pacotes, juntamente com o registrador de controle das regras, para a realização de suas operações. Os outros estágios mencionados anteriormente foram implementados em regras específicas para cada ação. Note portanto, que algumas instruções levam mais do que quatro ciclos do relógio para gerarem novas fichas.

CAPÍTULO 6

PIFLOW - Segunda Versão

Tendo em vista as limitações encontradas nos testes e simulações realizadas na primeira fase do desenvolvimento, surgiu a necessidade de uma reformulação na abordagem para o desenvolvimento do protótipo. A seguir serão destacadas as principais deficiências observadas durante os testes realizados, em seguida serão descritos os detalhes da implementação dessa nova abordagem proposta.

6.1 Motivação

6.1.1 Fila de Fichas

Na Fila de Fichas foram observadas algumas incoerências relevantes, como a manutenção de fichas na memória de armazenamento do módulo por longos períodos, uma vez que serão liberadas apenas quando não houverem fichas transitando entre a entrada e a saída do módulo. Além disso, existe o afunilamento provocado por essa unidade que pode vir a causar sobrecarga entre ela e a Unidade de Processamento. Lembre-se que existe uma quantidade definida de Unidades Funcionais e cada uma dessas unidades são capazes de gerar até duas fichas concorrentemente. Imagine um cenário onde ao menos 4 Unidades Funcionais geraram simultaneamente duas fichas de saída cada. Nesse caso particular é importante lembrar que esses pares de fichas serão enviados a Fila de Fichas em cada ciclo do relógio, uma a uma. Além disso, para cada par de fichas que chegam a essa unidade, apenas 1 delas serão enviadas ao buffer de saída (caso ele não estiver cheio). Nesse período, novas fichas estarão sendo criadas por essas mesmas 4 Unidades Funcionais analisadas anteriormente, que gerarão ainda mais fichas. Em um determinado momento é possível que a FIFO de saída de cada uma dessas Unidades Funcionais fique cheia, impedindo que essas unidades possam executar novas instruções. Esse impedimento sobrecarregará as outras Unidades Funcionais que podem vir a encher suas FIFOs de saída também. Portanto, em um determinado momento, pode acontecer de todas as FIFOs de todas as Unidades Funcionais existentes estarem cheias, impedindo a saída

de fichas e consequente entrada de novas fichas a cada novo ciclo do relógio, de forma que, se o restante da estrutura não for capaz de solucionar todas as operações previstas, para a liberação das saídas das Unidades Funcionais, a arquitetura inteira congelará indefinidamente.

É importante lembrar que essa é realmente a principal função da Fila de Fichas, pois seu principal objetivo é atuar como um *buffer* para toda a máquina, na ausência dessa unidade, diante desse tipo de cenário descrito anteriormente, haveria sobrecarga de toda a estrutura imediatamente. Sem a Fila de Fichas, a Unidade de Processamento deveria estar ligada diretamente a Unidade de Emparelhamento, que não possui capacidade para absorver um número muito grande de fichas, pois sua memória é destinada apenas para armazenamento de fichas não pareadas. Como mencionado anteriormente, a quantidade de fichas geradas por unidade de tempo pode ser muito grande, gerando sobrecarga entre essas duas unidades particulares. A partir dessa sobrecarga, não haveria possibilidade da Unidade de Armazenamento de Instruções enviar seus pacotes à Unidade de Processamento, pois o *buffer* de comunicação entre essas unidades também estariam sobrecarregados, pois as Unidades Funcionais ficariam sobrecarregadas e consequentemente o *buffer* de entrada da Unidade de Processamento também mostraria essa característica. Dessa forma, a Fila de Fichas é capaz de garantir que sempre haverá espaço em sua memória de armazenamento, garantindo que as Unidades Funcionais (ou ao menos uma parcela das unidades) possam sempre se manter prontas para execução de novas instruções.

Entretanto, a ligação direta da entrada e a saída dessa unidade, gerou esse tipo de incoerência, podendo manter fichas dentro do armazenamento interno da Fila de Fichas por grandes períodos de tempo. Embora seja esperado que apenas fichas com menos prioridade para execução imediata, fiquem alocadas nessa memória, pode acontecer que fichas com maior prioridade acabe ficando armazenada nela em determinado instante. Esse cenário implica em um grande atraso em toda a estrutura do anel, até que essa ficha particular seja liberada. A ligação dos *buffers* de entrada e saída ofereceram resultados muito bons no início e final da execução dos programas, porém, seria interessante que essa característica não interferisse na estrutura de toda a arquitetura, sob nenhuma situação específica como essa.

6.1.2 Unidades de Emparelhamento e de Armazenamento de Instruções

Analisando a Unidade de Emparelhamento, podemos destacar o atraso que as regras destinadas a emparelhamento são capazes de gerar. Conforme destacado no capítulo 5 pôde-se verificar que são consumidos pelos menos 2 ciclos do relógio para que uma ficha possa resgatar sua ficha parceira. Da mesma forma são consumidos 3 ciclos até que uma ficha que não encontrou sua ficha parceira possa ser gravada na memória interna da unidade (sendo 1 ciclo destinado ao processo de gravação nessa memória). Esse é um período muito longo para a realização de apenas uma operação, pois durante esse processo não é possível a entrada ou saída de novas fichas na unidade. Isso pode ser considerado como um grande gargalo da arquitetura, uma vez que é esperado que em todo ciclo do relógio ao menos uma ficha ou pacote possa entrar, e da mesma forma uma outra ficha ou pacote possa deixar cada unidade, de preferência concorrentemente. De maneira perfeitamente equivalente pode-se destacar exatamente o mesmo problema na Unidade de Armazenamento de Instruções, que toma o período fixo de 2 ciclos do relógio em qualquer operação que ela deve realizar. Sendo um ciclo destinado para a requisição da leitura de um endereço específico da memória RAM e o outro ciclo destinado a leitura do endereço lido, e criação do pacote de saída da unidade.

6.2 Objetivo

É importante destacar a importância da geração de ao menos uma ficha ou pacote de saída (e consequentemente de entrada) em cada unidade, em cada ciclo do relógio. Pelo mesmo motivo destacado na Fila de Fichas (descrito em detalhes anteriormente), a permanência de uma ficha ou pacote em uma unidade, para a realização de suas operações, acaba por gerar um atraso em todas as outras unidades, pois todas as operações previstas em cada uma delas são mutuamente importantes para a execução de cada instrução. A permanência dos pacotes dentro da Unidade de Emparelhamento por exemplo, pode gerar uma sobrecarga no *buffer* de entrada dessa unidade. Se essa sobrecarga for suficiente para preencher todo esse *buffer*, esse atraso será propagado para a Fila de Fichas, que deverá aguardar mais de 1 ciclo do relógio para liberar novas fichas. Mas pôde-se observar pela descrição anterior que a Fila de Fichas por si só, já é capaz de gerar certo atraso na arquitetura, além do fato que a ideia dessa unidade é justamente liberar novas fichas a cada ciclo do relógio. Dessa forma esse tipo de

sobrecarga é contrário a própria definição da arquitetura.

Para lidar com esse problema foi tomada como inspiração a solução da largura de banda implementada entre a Fila de Fichas e a Unidade de Processamento, descrita no capítulo 5. Embora essa característica tenha induzido um novo gargalo na implementação (problema do afunilamento provocado na Fila de Fichas), ela mostrou-se grande fonte para ganho em desempenho, uma vez que aumentou bastante a taxa de transferência das fichas entre essas unidades, conforme estimado no momento de sua concepção. Dessa forma, a ideia fundamental para a reformulação proposta, foi propagar essa característica para todas as unidades que formam a arquitetura.

A falta de homogeneidade para a criação de fichas no modelo *dataflow*, para as mais diversas e variadas instruções, não necessariamente correlacionadas a priori, garantem o objetivo principal mencionado anteriormente. Como não existe uma forte correlação entre as fichas que transitam entre as unidades, a separação delas em dutos distintos, pode ser capaz de garantir que ao menos uma ficha ou pacote seja capaz de entrar e sair dessas unidades a cada ciclo do relógio, em dutos distintos. Pode-se destacar como exemplo um caso particular da Unidade de Emparelhamento, suponha que enquanto um desses dutos estiver realizando uma leitura na memória dessa unidade (no duto que lhe for pertinente) em busca de uma ficha parceira, em um outro duto uma ficha pode estar resgatando sua parceira, formando o pacote de saída e enviando-o ao seu *buffer* de saída particular. Nesse mesmo instante um outro duto pode ainda estar realizando uma operação sobre uma ficha do tipo *Bypass*, que já fornece um pacote de saída no mesmo ciclo do relógio. Note portanto a auspiciosidade dessa implementação para o determinado objetivo esperado, uma vez que é possível um ganho superior ao necessário. O mesmo exemplo pode ainda ser aplicado à Unidade de Armazenamento de Instruções, que enquanto faz a requisição de uma instrução em sua memória, em um duto particular, pode estar gerando um pacote de saída em diversos outros dutos.

Figura 6.1 – Representação ilustrativa da segunda fase na implementação da arquitetura.

Fonte: Elaborada pelo autor.

6.3 Implementação

Vale ressaltar nesse ponto os benefícios garantidos pelo BSV para essa implementação. A grande maioria da estrutura criada na primeira fase do projeto foi utilizada integralmente nessa segunda abordagem. Algumas modificações necessitaram de alguns ajustes, mas em geral, nenhuma ideia foi totalmente perdida e em alguns pontos foram apenas realocadas para as novas necessidades. Em linhas gerais, os *buffers* de comunicação entre as unidades, alguns registradores de estado e os campos de memórias foram expandidas através do uso dos vetores característicos da linguagem, que apenas permitem suas identificações a partir de um índice determinado. As regras passaram a ser compostas dentro de laços, a fim de determinar o comportamento destas, para as instâncias de cada elemento desses vetores. Além disso, as regras e métodos que necessitavam de controle a partir das diretivas padrões do BSV (que permitem a organização explícita das regras ou métodos que apresentam conflitos, conforme mencionado no capítulo 3) passaram a utilizar registradores do tipo EHR em seus predicados (módulo este, descrito em detalhes no apêndice B). É interessante observar, que a definição de grande parte dos elementos que compõe o protótipo em vetores, foi aplicado de maneira direta, sem adição de novas linhas de código, apenas explorando o caráter funcional que a linguagem é capaz de oferecer. As dimensões desses vetores e a divisão dos campos de memória foram definidos a partir de um valor constante que determina a quantidade de dutos que irão compor a largura de banda da arquitetura, sendo definidos portanto em tempo de compilação.

6.3.1 Fila de Fichas

Essa unidade manteve-se praticamente intacta, com relação a primeira fase do desenvolvimento. Os *buffers* de comunicação foram divididos em vetores, sendo cada *buffer* desse vetor, responsável pela comunicação de um duto específico. A memória de armazenamento interno permaneceu com o mesmo tamanho anterior de 4K-fichas, que foram divididas igualmente entre cada duto de largura de banda. É importante observar, que esse valor considera apenas a parcela inteira da divisão, de forma que o valor total de memória pode ser um pouco inferior a essa quantidade, dependendo da quantidade de dutos definidos. Essa característica foi avaliada e não houveram perdas significativas pois a quantidade de dutos analisados não precisa ser superior a 20 unidades. Quando comparado às 4K-fichas de capacidade total de memória, o valor perdido no resto da divisão torna-se desprezível.

Um vetor de registradores EHR do tipo booleano foi utilizado para determinar a prioridade do conjunto de regras que definem o comportamento dos *buffers* esquerdos e direitos, além das regras que resgatam as informações das memórias da unidade. Isso foi necessário pois as regras pertencentes a cada duto concorrem acesso ao buffer de saída e à memória interna da unidade. Lembrando que o módulo EHR tem a capacidade de transmitir a ocorrência de um evento dentro do mesmo ciclo do relógio, conforme descrito em detalhes no apêndice B. Dessa forma, diante da ocorrência de uma ficha na entrada de qualquer dos dutos, essa informação será passada às regras que competem o acesso ao *buffer* de saída desse duto particular, garantindo o bloqueio delas. Por fim, um outro registrador do tipo EHR, verifica quando o método de entrada de dados da unidade foi disparado. Esse registrador tem o objetivo de bloquear todas as classes de regras, afinal todas elas concorrem acesso à memória de armazenamento de seu duto particular.

O primeiro predicado de cada regra analisa o vetor de EHRs destinados a prioridade. O primeiro índice desse vetor determina o duto de largura de banda a que se destina essa regra (índice do vetor) e o segundo índice determina a prioridade do elemento dentro do módulo EHR (índice do EHR). As regras de maior prioridade em cada duto, são aquelas destinadas ao controle das fichas esquerdas que chegam em cada um destes dutos. A primeira ação prevista para todas as regras dessa unidade é a alteração do valor desse registrador EHR - específico para o duto - para **falso**. Com isso, caso uma regra de menor prioridade verificar seu estado no instante que a regra de maior prioridade alterou seu estado, então essa outra regra não será executada. Da mesma forma, caso os outros predicados (idênticos as condições definidas na primeira fase) não forem atendidas, então as regras de menores prioridades serão capazes de realizar suas verificações.

Com todas essas análises de prioridades devidamente estabelecidas e as regras devidamente escalonadas dentro de cada ciclo, o comportamento delas manteve-se intacto com relação à fase anterior. Nesse formato, cada duto atua exatamente igual à Fila de Fichas da fase anterior. A ficha esquerda de cada duto é verificada inicialmente e caso o *buffer* de saída desse duto não estiver cheio, essa ficha será enviada diretamente a ele. Caso contrário, se o *buffer* de saída desse duto particular estiver cheio, a ficha recém chegada será adicionada a memória desse duto. Se houver ficha na direita de algum duto então a regra particular dele será ativada para o próximo ciclo (desativando a regra destinada às fichas da esquerda). E caso não hajam fichas de entrada (esquerda ou direita) em algum duto, então a regra que resgata fichas armazenadas na memória interna do duto e a envia ao *buffer* de saída, será ativada. Observe portanto que não houve alterações no corpo das regras e existe uma perfeita equivalência dos dutos com a unidade definida na primeira fase do projeto.

6.3.2 Unidade de Emparelhamento

Da mesma forma que a Fila de Fichas, a Unidade de Emparelhamento não sofreu muitas alterações em seu formato, com relação a implementação inicial do desenvolvimento. Em linhas gerais os *buffers* de comunicação da unidade foram adicionados em vetores, para garantir o acesso a essas informações a partir de índices estabelecidos. Um registrador do tipo EHR foi adicionado para lidar com as mesmas verificações propostas na Fila de Fichas. Esse registrador determina a prioridade das regras que concorrem acesso pelos *buffers* de saída da unidade. Das regras que concorrem por esse acesso uma delas é responsável pela definição do comportamento das fichas do tipo *Bypass* e a outra pelo comportamento das fichas do tipo *Non-bypass* que já fizeram a leitura da memória interna da unidade e podem vir a enviar pacotes montados para a Unidade de Armazenamento de Instruções. Além disso, dois novos vetores de registradores foram adicionados para organizarem as ações de fichas do tipo *Non-bypass* em cada duto do módulo. Um desses registradores armazenam a ficha de entrada da unidade, o que permite a remoção dessa ficha do *buffer* de entrada no ciclo que é realizada a solicitação de leitura da memória interna do módulo. Essa característica garante que o *buffer* possa receber novas fichas no caso extremo onde ele encontra-se todo cheio. O outro registrador, armazena o endereço que deverá ser lido na memória interna do módulo, não exigindo a replicação do mesmo circuito para a geração da regra de solicitação de leitura e da regra de recebimento da informação lida na memória. Como existem diversas cópias da mesma regra (para atender cada duto da largura), toda precaução foi avaliada para evitar a replicação de circuitos idênticos em cada duto, nesse caso, evitando que as regras devam montar o endereço de leitura da memória na regra de requisição e na regra de resgate das informações. * Por fim, a memória interna do módulo manteve-se com a mesma capacidade de armazenamento que o projeto original de 2^{20} posições para armazenamento, que aqui foi dividido igualmente entre os dutos da largura de banda. Apenas deve ser reforçado que esse é um valor impraticável para utilização apenas de FPGAs comerciais. O valor exato foi obtido aplicando-se o logaritmo sobre a divisão entre o exponencial da quantidade de bits para endereçamento e a quantidade de dutos de largura de banda.

Exatamente como na Fila de Fichas, a partir da verificação minuciosa nos predicados das regras, a fim de realizar o escalonamento manual delas em cada duto, seus comportamentos mantiveram-se sem grandes alterações. Para as fichas do tipo *Bypass*, apenas é realizada a montagem do pacote de saída característico da unidade e enviado ao *buffer* de saída do

* Lembrando que o endereço para as fichas nessa memória é obtido pela composição do Nome de Ativação com o endereço da instrução na Unidade de Armazenamento de Instruções.

duto particular. Para as fichas do tipo *Non-bypass* foram alteradas apenas as características relacionadas ao armazenamento das fichas em registradores próprios, a fim de evitar replicação de circuitos idênticos, para a formação do endereço de destino, além da adição da ficha de entrada da unidade em um registrador próprio para essa finalidade, conforme mencionado anteriormente.

Novamente de acordo com o projeto inicial, aqui existe um conjunto de regras que verificam se a informação lida na memória refere-se a uma ficha válida. Em caso positivo, o pacote de saída é gerado e enviado ao *buffer* de saída do duto. Se não for encontrada uma ficha válida nesse endereço, então a ficha atual será adicionada a esse endereço de memória para aguardar por sua ficha parceira. É importante reforçar o grande aumento na taxa de transferência obtida nessa unidade, uma vez que, agora a arquitetura permite que todos os dutos de largura de banda possam enviar pacotes simultaneamente, no melhor caso possível. Essa unidade podia ser tomada como um grande gargalo da arquitetura no projeto inicial, já que podia levar até 3 ciclos inteiros para nem ao menos gerar um novo pacote de saída. Lembrando que esse pior caso era obtido para a gravação de uma nova ficha na memória interna da unidade. Essa característica não é mais verificada na nova arquitetura e a capacidade de pacotes paralelos que podem deixar a unidade é definida pelo valor constante que determina a largura de banda de toda a nova estrutura.

6.3.3 Unidade de Armazenamento de Instruções / *Parser*

As alterações previstas para essa unidade influenciaram diretamente a composição do *Parser* (descrito no apêndice A). A memória da Unidade de Armazenamento de Instruções foi dividida igualmente entre cada duto da largura de banda. As instruções foram disponibilizadas nos dutos de maneira sequencial, ou seja, a primeira instrução foi alocada na memória do primeiro duto, a segunda instrução no segundo duto e assim por diante. Após a ocorrência do $n - \text{ésimo}$ duto, a instrução subsequente foi alocada no primeiro duto novamente. Esse balanceamento foi proposto a fim de disponibilizar as instruções em dutos "distantes" - por assim dizer - uma vez que sua definição é que irá determinar o duto por onde as fichas / pacotes vão percorrer toda a estrutura ao longo do anel, desde a Fila de Fichas até a Unidade de Processamento. Após a saída das novas fichas geradas pelas Unidades Funcionais, a Unidade de Processamento se encarrega de organizar as fichas em seus dutos equivalentes, baseados no endereço da instrução presente na ficha, informação determinada no código gerador e definida

pelo *Parser*. Observe portanto que o *Parser* é que realiza todo o escalonamento das instruções definidas no código gerador em arquivos distintos, que serão incorporados nas memórias de cada duto da Unidade de Armazenamento de Instruções.

Ademais, a estrutura da unidade não sofreu grandes alterações. Os *buffers* de comunicação da unidade foram igualmente adicionada a vetores, com tamanho igual a quantidade de dutos de largura de banda. Um vetor de registradores foi adicionado para guardar o pacote do *buffer* de entrada de cada duto, liberando esse *buffer* no caso crítico onde ele se encontra completamente cheio, exatamente como a solução empregada na Unidade de Emparelhamento, a fim de garantir que fichas ou pacotes possam transitar entre as unidades sem a ocorrência de bloqueios por capacidade de armazenamento. Além disso, um vetor com as estruturas de configurações do módulo *BRAM* foi criado para realizar a definição da memória interna da unidade (que também foi dividida em um vetor no formato definido pelo *Parser*, conforme mencionado anteriormente). Em um laço, foram definidos os nomes dos arquivos de origem que devem estar presentes em cada memória. Após a definição desse laço, foi aplicada a função *mapM()* do BSV, que instancia cada uma das memórias dentro do vetor, "mapeando" essas estruturas de configuração na memória de cada duto particular.

Da mesma forma que nas outras unidades, as regras que compunham o projeto inicial foram adicionadas em laços característicos do BSV, definindo as ações particulares de cada duto. Conforme a primeira fase do projeto existem apenas duas regras que determinam as ações dessa unidade. A primeira regra resgata a informação presente no *buffer* de entrada do duto. Diferentemente do projeto inicial, essa informação é adicionada a um registrador característico, para utilizar essa informação na próxima regra, para a formação do pacote de saída da unidade. Com isso, é possível fazer a remoção do pacote recém chegado do *buffer* de entrada desse duto. Além disso, é realizada a solicitação de leitura da informação presente no endereço de memória indicada nesse pacote. A regra seguinte resgata a informação presente no endereço de memória solicitado e faz a criação do pacote de saída que é adicionado ao *buffer* de saída do duto particular. Da mesma forma como ocorreu na Unidade de Emparelhamento, o aumento na taxa de transferência dessa unidade cresce de acordo com o tamanho da largura de banda, garantindo incrível ganho em desempenho, quando comparado ao projeto inicial. Observe que no melhor caso, existe a possibilidade de serem geradas a quantidade de pacotes de saída, igual o número de dutos de largura de banda da arquitetura. Lembrando que essa unidade apresentava a característica de gerar pacotes somente a cada 2 ciclos do relógio, sendo considerada um outro grande gargalo da solução.

6.3.4 Unidade de Processamento

Diferente das outras unidades descritas até aqui, essa unidade apresentou algumas dificuldades para implementação. Todos os *buffers* de comunicação e a grande maioria dos registradores presentes na unidade (descritos em detalhes na seção 5) foram adicionados em vetores. Os registradores de registro foram mantidos sem alterações, responsáveis por identificar o primeiro ciclo do relógio que a máquina foi executada, para definir a identificação de cada Unidade Funcional, bem como o registrador que identifica a primeira ficha que entra na unidade, que tem o objetivo de registrar o primeiro Nome de Ativação válido. Dois vetores de EHR, têm o objetivo de determinar a ordem com que as Unidades Funcionais serão exploradas por cada duto de largura de banda. Além disso, quatro outros vetores de EHR foram adicionados para determinar as prioridades das fichas de saída da unidade, uma vez que todas as fichas possuem um destino muito bem definido dentro da arquitetura. Lembre-se que cada Unidade Funcional possui duas portas de comunicação, cada uma responsável por uma das saídas possíveis das unidades.

As regras que determinam as operações de registros mantiveram-se exatamente da mesma forma que o projeto inicial, como a regra responsável pela determinação da identificação de cada Unidade Funcional e a regra para registro do primeiro Nome de Ativação. Da mesma forma que no projeto inicial, existe uma regra responsável por analisar a próxima Unidade Funcional livre, baseado na informação presente no registrador que armazena o índice da última Unidade Funcional que recebeu um pacote. A diferença é que nessa nova abordagem, são verificadas todas as Unidades Funcionais sequencialmente, de maneira rotativa (após a $n - \text{ésima}$ unidade é verificada a unidade zero novamente). Além disso, o registrador de controle armazena o último índice que iniciou a sequência, ao invés do índice da última Unidade Funcional utilizada. Dessa forma esse novo circuito faz o mapeamento dos dutos de largura de banda nas Unidades Funcionais livres. Porém, a quantidade de Unidades Funcionais pode ser maior que a quantidade de dutos, o que exigiu uma verificação mais restrita para o escalonamento das unidades. Um conjunto de regras foi adicionada em dois laços aninhados, o registrador de EHR que realizou o mapeamento previamente, é verificado no predicado dessas regras, verificando se determinado duto particular é o escolhido para atender a Unidade Funcional em questão. Se essa condição é atendida, ainda é verificado um novo registrador de EHR, que analisa se essa Unidade Funcional particular realmente não foi utilizada. Essa última verificação garante o caso onde a quantidade de Unidades Funcionais é maior que a quantidade de dutos de largura de banda. Com essas análises devidamente realizadas, foi possível obter o escalonamento das regras mapeadas anteriormente, com a alocação de cada Unidade

Funcional livre para receber os pacotes de cada duto específico, garantindo o abastecimento de múltiplas unidades, paralelamente. Cada uma dessas regras realizam as mesmas operações previstas no projeto inicial, realizando o resgate do pacote presente no *buffer* de entrada do duto e a alocação desse pacote na Unidade Funcional equivalente a ele.

Os *buffers* de saída das Unidades Funcionais (esquerdo e direito) são organizados de maneira bastante semelhante a configuração utilizada em suas entradas. Em todo ciclo são verificadas as saídas de todas as unidades, isso é feito a partir de um laço que determina um conjunto de regras para essa finalidade. No escopo da regra responsável por uma Unidade Funcional particular, a ficha de saída dessa unidade é resgatada. A partir dessa ficha é obtido o endereço de origem dela, note que o resto da divisão entre esse endereço e a quantidade total de dutos de largura de banda, determina qual duto específico, essa ficha deve percorrer o anel, para encontrar o endereço de seu destino corretamente. O valor equivalente ao duto que essa ficha deve percorrer o anel é então adicionado a um vetor de EHRs, onde seu índice refere-se à Unidade Funcional que gerou essa ficha. É importante observar que toda essa operação é realizada em duas regras distintas, cada qual responsável pelo resgate das fichas esquerda ou direita dessas unidades. Em um laço aninhado que representa cada duto de largura de banda, existe um novo conjunto de regras que verifica o valor desse vetor de EHRs em seus predicados, confirmando se o valor contido no seu índice particular refere-se ao duto a que se destina a ficha. Note que por se tratar de um EHR, as Unidades Funcionais de menor índice possuem maior prioridade sobre essa operação, pois realizam o acesso a essa entidade primeiramente. Essa característica não pode ser vista como uma deficiência, por causa da distribuição de carga proposta pela arquitetura, implementada no *Parser*, conforme descrito anteriormente.

6.3.5 Unidades Funcionais

As Unidades Funcionais não foram afetadas pelo aumento da largura de banda, porém foi proposta uma alternativa para a entrada dos pacotes nessas unidades. Foi adicionado um *buffer* de entrada em cada unidade, a fim de garantir a disponibilidade integral delas. No projeto inicial, a ficha de entrada era adicionada em um registrador que mantinha esse pacote em todas os estágios do *pipeline*. Ao final do último estágio, as informações eram todas resetadas, inclusive o registrador responsável por determinar a disponibilidade dessa unidade. Isso quer dizer que ao término de todas as operações previstas pela unidade, ainda era perdido um ciclo para a indicação de disponibilidade dela. Com a utilização do *buffer*

de entrada, as Unidades Funcionais não receberam novas fichas somente no caso onde esse elemento estiver cheio. O caráter rotativo para distribuição dos pacotes entre as Unidades Funcionais é a característica responsável pela correta distribuição da carga entre elas. Nesse novo cenário não existe qualquer perda para o aproveitamento das unidades, uma vez que o primeiro estágio do *pipeline* passou a ser realizado na ação de limpeza das informações, que já realiza o resgate da informação contida no *buffer* de entrada. Com essa nova característica adicionada e a distribuição de carga descrita anteriormente, o aproveitamento das Unidades Funcionais passou a ser muito mais efetivo.

CAPÍTULO 7

Resultados

7.1 PIFLOW - Primeira versão

Na primeira versão do protótipo, foi utilizada exatamente a mesma configuração da MDFM para a criação da PIFLOW, conforme a descrição detalhada do capítulo 4. Com exceção da Unidade de Sobrecarga, toda a estrutura foi mantida exatamente igual a proposta da arquitetura de origem. A comunicação entre os módulos foi realizada a partir de FIFOs de tamanho definido, padrões do BSV, a fim de garantir a liberação dos recursos das unidades o quanto antes possível.

Para validar o correto funcionamento da estrutura, de acordo com a MDFM, alguns atributos foram adicionados nas unidades, a fim de registrar alguns comportamentos do protótipo, em nível de simulação. Na Unidade de Processamento, foi adicionado um registrador responsável pela contagem dos ciclos utilizados para a execução dos programas. Além disso, foi adicionado um vetor de registradores, onde cada índice desse vetor representa uma Unidade Funcional específica. Esse vetor tem a finalidade de armazenar a porcentagem de utilização de cada Unidade Funcional, durante a execução de programas. Na Fila de Fichas e na Unidade de Emparelhamento foram adicionados registradores que armazenam a máxima utilização de suas memórias internas. Diante da ocorrência da instrução de saída do conjunto de instruções da MDFM, a Unidade Funcional faz a geração de uma ficha específica, que ao passar por uma unidade é reconhecida através do bit identificado como *marcador*, conforme descrito no capítulo 5. Diante da ocorrência desse tipo de ficha, cada unidade faz a impressão das informações contidas nos registradores citados acima *.

Para a validação do correto funcionamento do protótipo, foram utilizados dois *micro benchmarks* principais, gerados a partir do compilador original da MDFM. O primeiro programa faz o cálculo do fatorial de um número definido estaticamente no programa, implementado de maneira duplamente recursiva, no cálculo da multiplicação presente nessa operação. Esse programa foi adaptado diretamente no código em linguagem montadora, realizando uma operação de soma, ao invés da multiplicação do fatorial. Essa alteração se fez necessária para

*Em nível de simulação.

a execução de programas maiores, para maior exploração de paralelismo, uma vez que o fatorial de números muito grandes exigiria algumas adaptações na representação numérica da arquitetura e esse não era o objetivo principal do projeto.

Além desse programa, foi utilizado um programa que realiza a integração numérica binária de uma função especificada, através do método do trapézio, também implementado de maneira duplamente recursiva. Esse programa recebe como valores de entrada os limites de integração da função, além da profundidade da árvore formada pela operação.

Ambos os programas possuem imensa capacidade de paralelismo devido aos seus caracteres duplamente recursivos. Entretanto, é importante observar, que para execuções de valores pequenos, ou seja, com uma exploração pequena da capacidade de recursividade apresentada pelos programas, pouco paralelismo é obtido deles, induzindo a uma execução praticamente sequencial. Outros programas muito pequenos e puramente metodológicos foram desenvolvidos diretamente em linguagem montadora, para atestar o correto funcionamento da implementação do conjunto de instruções da PIFLOW.

Para verificar se o comportamento do protótipo estava de acordo com os resultados que seriam obtidos na MDFM, foi utilizado o emulador desse protótipo, desenvolvido pela equipe criadora dessa arquitetura (52). Esse emulador foi desenvolvido em Pascal, e tem o objetivo de emular detalhadamente todo o comportamento da MDFM. A utilização desse simulador foi muito importante nessa fase do desenvolvimento, pois o objetivo principal era obter o mesmo comportamento da máquina de origem. A partir deste emulador foi possível encontrar alguns comportamentos incompatíveis entre a PIFLOW e a MDFM, como por exemplo o máximo armazenamento nas memórias internas da Fila de Fichas e da Unidade de Emparelhamento e a porcentagem de utilização das Unidades Funcionais.

Uma característica que pôde ser observada a partir dessas informações foi uma relativa sobrecarga no armazenamento da Fila de Fichas. Esse fator se dá pela ausência do *Throttle* na PIFLOW. Esse módulo tinha o objetivo de controlar o paralelismo da arquitetura, evitando a sobrecarga de fichas na Unidade de Emparelhamento (49) e consequentemente na Fila de Fichas. O *Throttle* foi omitido até o presente momento, pois foi considerado que a quantidade de memória disponível na Unidade de Emparelhamento seria capaz de atender a toda a demanda, fato que inclusive motivou a omissão da Unidade de Sobrecarga, descrita no capítulo 4.

As memórias utilizadas na Unidade de Emparelhamento e na Unidade de Armazenamento de Instrução, são implementadas a partir de um módulo nativo do BSV, intitulado BRAM. Esse módulo tem a finalidade de utilizar os campos de memória nativos das FPGAs, ou em

nível de simulação, onde ele reserva um espaço da memória RAM do computador onde está sendo realizada essa simulação. O módulo BRAM é capaz de realizar a leitura de informações em apenas um ciclo do relógio, de forma que são exigidos um total de dois ciclos, desde a solicitação da informação, até o retorno do valor presente no endereço solicitado, nas portas de saída do módulo.

Inicialmente, foi motivada a implementação das duas portas de comunicação presentes nesse módulo. Essa prática tinha o objetivo de reduzir a pequena latência apresentada pelo módulo BRAM, de dois ciclos, para o resgate dos dados. Porém, a implementação dessa característica acabou por aumentar consideravelmente a complexidade da Unidade de Emparelhamento e da Unidade de Armazenamento de Instruções, conforme foi mostrado em detalhes na figura 5.5. O controle das duas portas aumentou consideravelmente o caminho crítico de toda a arquitetura, o que não apresenta um cenário desejado, de forma que passou a ser utilizada apenas uma das porta para a comunicação com os módulos BRAM em ambas as unidades que o utilizam. No entanto, foi promovido o ganho máximo em desempenho das Unidades Funcionais, fator que deveria se capaz de compensar a latência promovida pelo módulo citado, o que não foi verificado conforme será apresentado a seguir.

Uma métrica comum para avaliação do desempenho de processadores paralelos, é o *speedup*. Essa medida identifica o desempenho do processador com uma quantidade de unidades de processamento operando paralelamente, com relação ao desempenho da mesma computação, no mesmo processador, operando com apenas uma unidade de processamento. O ideal de *speedup* para processamento paralelo é que o ganho em desempenho seja proporcional a quantidade de unidades de processamento. Por exemplo, a execução de um programa com duas unidades paralelas deve ser realizada duas vezes mais rapidamente que o mesmo programa executado por apenas uma unidade. Para a realização dessa medida foi adicionado um registrador responsável pela contagem de ciclos na Unidade de Processamento, conforme mencionado anteriormente. Uma regra se encarrega de incrementar esse registrador a cada ciclo do relógio.

Todos os resultados apresentados nesse capítulo, com as curvas de *speedup* obtidas, foram tomados através do simulador nativo do BSV, o **Bluesim**, mencionado anteriormente. Uma das características mais interessantes dessa linguagem de descrição de *hardware*, é sua imensa capacidade de oferecer uma ferramenta nativa e totalmente confiável, para simulação. O **Bluesim** é gerado a partir do código **Verilog** utilizado para síntese da solução em FPGA, a partir do qual são gerados arquivos descritos em linguagem de programação C, perfeitamente coerentes com o material disposto para a realização desta síntese. Sendo assim, de acordo com a Bluespec Inc., é possível assegurar de que todo o comportamento obtido a partir

dessa ferramenta de simulação, será o comportamento obtido pelo *hardware* proposto pelo desenvolvedor. Por esse motivo que grande parte do desenvolvimento do presente projeto foi realizado em nível de simulação, sendo atestado que em nenhum momento houve perda de generalidade por essa prática.

Com o processador configurado de acordo com as características mencionadas acima, foi possível observar um grande gargalo na arquitetura. Na PIFLOW, as Unidades Funcionais foram implementadas em formato de *pipeline*. Dessa forma, após 4 ciclos do relógio, as fichas resultantes já estão disponíveis para deixar essas unidades. Entretanto, para que fichas do tipo *bypass* possam percorrer todas as unidades que formam a arquitetura, em busca das informações que lhe são destinadas, é decorrido um período máximo de 6 ciclos [†]. Note portanto, que dessa forma, no máximo três Unidades Funcionais podem ser aproveitadas a partir dessa abordagem, pois durante o percurso de uma ficha entre as unidades da arquitetura, uma Unidade Funcional já foi capaz de finalizar a execução da ficha corrente.

Essa característica é ainda atenuada pelo atraso existente na Unidade de Armazenamento de Instruções, que promove o atraso de um ciclo para o envio de novas fichas à Unidade de Processamento, por causa da latência característica do módulo BRAM mencionado anteriormente. Dessa forma, mesmo que exista um acúmulo de fichas na entrada dessa unidade, ao final do resgate das informações de 3 fichas, já decorreram o período de 6 ciclos do relógio, período o qual, uma Unidade Funcional já foi capaz de realizar a execução de uma instrução, gerar novas fichas e estar novamente pronta para o recebimento de novas fichas. Sendo assim, pode-se observar esse limite sistemático para utilização das Unidades Funcionais da arquitetura.

A figura 7.1 a seguir apresenta a curva de *speedup* dessa primeira etapa do desenvolvimento.

Na tabela apresentada 7.1, estão disponibilizadas todas as informações coletadas no protótipo, em nível de simulação. A primeira coluna da tabela indica a quantidade de Unidades Funcionais ativas durante a execução. A coluna MTQ[‡] apresenta a máxima utilização da memória interna que compõe a Fila de Fichas. A coluna MMU[§] indica a máxima utilização na memória interna da Unidade de Emparelhamento. A coluna Ciclos, apresenta a quantidade de ciclos necessários para a execução do programa. A coluna Speedup apresenta o valor

[†]Desde a saída da nova ficha na Unidade de Processamento, até seu retorno a essa mesma unidade, para sua execução.

[‡]Máxima utilização da *Token Queue*

[§]Máxima utilização da *Matching Unit*

Figura 7.1 – Speedup da primeira versão do projeto

Fonte: Elaborada pelo autor.

desse atributo, em cada medida obtida. E finalmente, a coluna %PU[¶] indica a porcentagem de utilização das Unidades Funcionais presentes na execução. Essas medidas foram bastante utilizadas em conjunto com os resultados obtidos a partir do emulador da MDFM, indicado anteriormente, a fim de conseguir deixar a arquitetura da PIFLOW com resultados os mais próximos possíveis com relação a arquitetura de origem.

Conforme pode ser visto no gráfico da figura 7.1, não existe ganho substancial em desempenho, a partir da adição de mais do que três Unidades Funcionais. Na tabela 7.1 é possível observar que embora não exista ganho efetivo em desempenho, a utilização das Unidades Funcionais possuem valores acima do esperado. Nessa primeira fase do projeto, as Unidades Funcionais recebiam novos pacotes a partir de um simples registrador. Esse registrador mantinha o pacote de entrada durante toda a execução de sua instrução. Enquanto as fichas de saída não fossem geradas, a Unidade Funcional ficava em um estado bloqueado para o recebimento de novos pacotes, estado este que determinava quando essas unidades estavam em operação, para a contagem na porcentagem de utilização delas. Além disso, o circuito utilizado para alimentação das Unidades Funcionais, implementado em formato circular, conforme descrito em detalhes no capítulo 5, garantia a atividade de todas as unidades durante uma execução. Essas características promoveram a alta utilização das diversas Unidades Funcionais presentes

[¶]Porcentagem de utilização da *Processing Unit*

Tabela 7.1 – Resultados obtidos para o programa do fatorial modificado, durante a primeira fase da implementação.

PUs	MTQ	MMU	Ciclos	Speedup	%PU
1	1849	2251	207028	1.000	99%
2	1849	2251	104289	1.985	96%
3	1830	2203	89058	2.325	91%
4	1663	1994	88702	2.334	90%
5	1499	1746	88472	2.340	89%
6	1456	1766	88494	2.339	89%
7	1211	1494	88449	2.341	91%
8	963	1199	88437	2.340	84%
9	978	1278	88469	2.340	87%
10	982	1280	88474	2.334	87%

Fonte: Elaborada pelo autor

na arquitetura, embora na prática, essa utilização seguia o baixo aproveitamento das unidades, conforme pode ser visto pela curva apresentada no gráfico acima, devido à falha sistemática verificada na arquitetura, atenuada pelo atraso verificado na Unidade de Armazenamento de Instruções, conforme discutido anteriormente.

Durante a execução de um programa convencional diversas fichas são geradas: fichas do tipo *bypass* e fichas do tipo *non-bypass*. Além disso, um dos principais atributos do compilador da MDFM era seu processo de otimização, capaz de garantir que a grande maioria das instruções estivessem conectadas a outras duas instruções subsequentes. Essa característica, faz com que para a grande maioria das instruções, duas novas fichas subsequentes fossem geradas. Essas fichas são então reordenadas sequencialmente na Fila de Fichas, ficando defasadas por um ciclo do relógio dentro de cada Unidade que compõe o anel. Por causa do gargalo encontrado, devido ao desempenho das Unidades Funcionais, superior que o restante da estrutura da máquina, pôde-se verificar na figura anterior, que o *speedup* atinge um limite a partir de três Unidades Funcionais.

Como o compilador da MDFM procura fornecer duas saídas subsequentes, para a maioria das instruções, foi motivada a criação de uma nova porta de comunicação entre a Unidade de Processamento e a Fila de Fichas. Dessa forma, as fichas que deixavam as Unidades Funcionais passaram a realizar essa operação concorrentemente, chegando juntas na Fila de Fichas. Note que essa característica promove um aumento considerável no *throughput*, entre essas unidades. O *throughput* determina a quantidade de fichas que transitam de um módulo ao outro, por unidade de tempo, nesse caso, por ciclo do relógio. Com essa simples modificação, foi possível observar uma redução substancial no tempo total para a execução dos programas, porém a curva de *speedup* permaneceu com o mesmo padrão observado na figura 7.1.

Para validar o argumento de que o baixo *speedup* estava diretamente relacionado ao baixo *throughput* entre as unidades que formam o anel, foi adicionado explicitamente um atraso intencional entre as etapas de execução de cada Unidade Funcional, impedindo a geração de novas fichas por um período determinado. Esse atraso adicionado possui o mesmo período para todas as Unidades Funcionais. Essa característica deveria ser capaz de mostrar que, quanto maior fosse o valor do atraso, maior seria o *speedup* obtido pela máquina, já que, a falha sistemática observada, refere-se ao tempo que cada Unidade Funcional leva para retornar ao estado de Pronta para Execução, com relação ao tempo que as fichas levam para percorrer todo o anel. Portanto, o atraso adicionado é capaz de atestar que, quanto maior o período levado pelas Unidades Funcionais para gerar novas fichas, maior seria o aproveitamento da quantidade de unidades presentes na máquina, com relação ao tempo que as fichas levam para percorrer todo o anel. Esse atributo foi adicionado a partir de um valor constante do BSV, o que permitiu a alteração desse valor dinamicamente, a cada processo de compilação. Para isso, foi adicionado um laço padrão do BSV, antes das regras destinadas para o envio das fichas resultantes à saída da unidade. A partir dessa modificação, foi possível obter a curva de *speedup* da figura 7.2.

Figura 7.2 – Speedup x Atraso

Fonte: Elaborada pelo autor.

A figura 7.2 mostra um ganho substancial em *speedup*, atingindo um limite próximo a quantidade de Unidades Funcionais presentes na execução, conforme esperado. É importante

observar que o gráfico apresenta o valor de *speedup* como função do atraso adicionado intencionalmente nas Unidades Funcionais. É interessante observar que o perfil da curva é equivalente àquela da figura 7.1, mostrando que existe uma deficiência sistemática para o ganho em *speedup* na estrutura da MDFM, bem como já era observado na máquina de origem. Note que o limite para o aumento no *speedup*, para 10 e 20 Unidades Funcionais estão próximos ao dobro do atraso intencional, mostrando que embora a arquitetura possua a deficiência sistemática observada, existe um limite onde ela ainda é capaz de apresentar um grande potencial para processamento paralelo. Para isso, era necessário que o desempenho das estrutura que compõem o anel tivesse o dobro de desempenho, do que o que era apresentado nessa primeira fase do desenvolvimento.

7.2 PIFLOW - Segunda versão

O resultado mostrado pela figura 7.2, em conjunto com a alteração da arquitetura original através do aumento na comunicação adicionado entre a Unidade de Processamento e a Fila de Fichas, motivou a implementação do aumento na largura de banda descrita em detalhes no capítulo 6. O aumento no *throughput* de toda a estrutura do processador, deveria ser capaz de fornecer muito mais fichas às Unidades Funcionais por unidade de tempo. Para a realização dessa alteração na estrutura da arquitetura de origem, foi adicionada uma constante fixada, que tem o objetivo de representar a largura de banda de toda a estrutura, definida em tempo de compilação da solução.

O grande desafio para implementação dessa característica se deu na necessidade de sincronismo e tratamento de concorrência entre as diferentes unidades da arquitetura, a fim de obter o paralelismo desejado entre elas. A principal fonte de conflitos nessa etapa, ocorreu no circuito utilizado para a escolha da Unidade Funcional destinada para a execução, em cada duto da largura de banda, além do circuito definido para o envio das fichas geradas em uma Unidade Funcional para seu duto equivalente. É importante recordar que é nessa última fase que se define o duto que irá transportar as fichas e pacotes ao longo de todo o anel. Essa característica é retirada do endereço da instrução para a qual uma ficha específica é destinada. A principal dificuldade para essa implementação foi conseguir uma maneira genérica para fazer todas as conexões necessárias para realizar a distribuição dos pacotes entre as Unidades Funcionais, uma vez que a escolha para a quantidade dessas unidades também é realizada de forma dinâmica. Além disso, a escolha do duto de destino de novas fichas é realizada a partir das informações presentes nessa ficha, portanto possuindo caráter ainda mais dinâmico. Com

isso, foi necessário realizar essas operações de forma atômica, excluindo quaisquer eventuais conflitos que poderiam surgir para a realização das operações propostas. A partir dessa nova abordagem o protótipo foi capaz de oferecer o *speedup* apresentado na figura 7.3.

Figura 7.3 – Speedup da segunda versão do projeto para o programa do fatorial modificado

Fonte: Elaborada pelo autor.

É importante observar que não existe uma quantidade ótima para a escolha de uma determinada quantidade de dutos de largura de banda, a fim de se obter o melhor resultado a partir dessa característica. Essa escolha foi realizada a partir da observação dos melhores resultados verificados. Esse fator é observado porque a distribuição das instruções são feitas dinamicamente, através do *Parser*, que aloca as instruções nos dutos de largura de banda de forma sequencial, conforme descrito em detalhes no capítulo 6. Dessa forma, para cada definição de largura de banda, instruções mais "importantes" para a execução do programa corrente, podem ser alocadas no mesmo duto de largura de banda. Essa ocorrência pode implicar na sobrecarga desses dutos particulares no decorrer de uma execução.

Por exemplo, se uma instrução for muito requisitada em instantes de tempo próximos, as Unidades Funcionais tentarão acessar o mesmo duto de largura de banda em suas saídas, concorrentemente. Dessa forma, apenas uma dessas unidades poderão enviar suas fichas de saída a esse duto, em cada ciclo do relógio. Enquanto isso, as outras unidades ficarão aguardando até poderem enviar suas fichas para esse mesmo duto. Além disso, esse duto particular terá muitas fichas para serem reordenadas na Fila de Fichas, já que cada duto

Tabela 7.2 – Resultados obtidos para o programa do fatorial modificado, durante a segunda fase da implementação.

PU _s	BW	MMU	Ciclos	Speedup	%PU
1	2	1167	43553	1.000	99%
2	2	1173	21918	1.987	99%
3	7	1339	14713	2.960	98%
4	7	1270	11323	3.846	96%
5	14	1274	9174	4.747	95%
6	14	1072	7878	5.528	92%
7	14	1058	7043	6.184	88%
8	14	1064	6399	6.806	85%
9	18	1116	6025	7.228	80%
10	18	1211	5714	7.622	76%

Fonte: Elaborada pelo autor

possui duas portas de comunicação entre a Unidade de Processamento e esta outra unidade. Isso faz com que o *buffer* de entrada da Fila de Fichas fique sobrecarregado, impedindo o acesso a essa unidade a partir desse duto específico. Esse fator leva ao bloqueio das Unidades Funcionais temporariamente, até que a Fila de Fichas consiga ordenar todas as fichas desse duto sobrecarregado. Dessa forma, existem algumas configurações para a distribuição das instruções - de acordo com a definição de determinados valores para largura de banda - que garantem melhor alocação de recursos, como é possível verificar na tabela 7.2. Deve-se observar que embora uma determinada instrução possa não ser central para o programa corrente, ela pode ser responsável por gerar fichas para uma instrução com essa característica, fato que evidencia que todas as instruções são importantes nessa arquitetura.

Observe a ausência da medida da máxima utilização na memória da Fila de Fichas, na tabela acima. Para a realização dessa medida, era necessária a adição de muita complexidade nessa unidade, pois os registradores responsáveis pelo armazenamento dessa informação, sofriam acesso simultâneo na entrada e na saída da unidade. Essa característica passou a impedir o envio de fichas em determinados momentos, o que não deveria ser observado, por um simples contador. Dessa forma, ele foi omitido nessa etapa do desenvolvimento, para evitar perda de desempenho.

Observe também, que nessa etapa do desenvolvimento, foram adicionados *buffers* de comunicação, para a entrada de novas fichas nas Unidades Funcionais, de forma que passou a ser mais preciso o cálculo para a verificação da utilização efetiva dessas unidades. A porcentagem de utilização representa o período que as unidades estavam efetivamente realizando a execução das instruções destinadas a elas. O único bloqueio possível para a contagem desse valor, é no caso descrito acima, onde uma determinada sobrecarga na Fila de Fichas pode vir

a bloquear as fichas de saída das Unidades Funcionais. Porém, essa sobrecarga não ocorre de forma sistemática e eventuais ocorrências não seriam representativas de forma a prejudicar a medida desse valor.

Figura 7.4 – Speedup da segunda versão do projeto para o programa de Integração Binária

Fonte: Elaborada pelo autor.

A figura 7.4, mostra a curva de *speedup* relativa a execução do programa para o cálculo de integração, implementado de forma duplamente recursiva. É possível observar praticamente o mesmo padrão de curva, que o observado na figura 7.3. Isso já era totalmente esperado, porque ambos os programas possuem características muito semelhantes de implementação, em nível de paralelismo, uma vez que eles foram igualmente implementados de maneira duplamente recursiva, segundo as especificações de cada programa. Dessa forma, pode-se verificar na figura 7.4, praticamente o mesmo padrão de curva que o do programa para o cálculo do fatorial, apresentado anteriormente.

A figura 7.5 mostra o *speedup* da máquina para a execução do mesmo programa de somatorial, porém para o cálculo de valores menores, no mesmo gráfico. Observe que quanto menor o paralelismo, mais o perfil da curva se aproxima do estado da primeira fase do desenvolvimento. Isso se dá porque a execução torna-se praticamente sequencial. Por outro lado, pode-se ver que quanto maior o paralelismo fornecido pelo programa em execução, melhor será o desempenho da execução nessa arquitetura, que é o nosso principal objetivo. Dessa forma, é possível observar que a utilização de uma arquitetura paralela, não é a solução mais

Tabela 7.3 – Resultados obtidos para o programa de integração binária, durante a segunda fase da implementação.

PU _s	BW	MMU	Ciclos	Speedup	%PU
1	2	508	18176	1.000	99%
2	6	572	9193	1.977	98%
3	10	544	6223	2.921	97%
4	10	623	4770	3.810	95%
5	10	481	3921	4.636	92%
6	10	507	3380	5.378	89%
7	14	567	3008	6.043	86%
8	19	542	2705	6.719	84%
9	21	459	2523	7.204	80%
10	21	472	2356	7.715	77%

Fonte: Elaborada pelo autor

indicada para a execução de programas puramente sequenciais. Entretanto, isso não significa que a arquitetura paralela levará mais tempo para a execução, mas ela não conseguirá ser muito superior, como para programas com exploração de grande quantidade de paralelismo.

7.3 Síntese

É muito importante reforçar que os resultados mostrados até o presente momento foram retirados do simulador padrão do BSV, o **Bluesim**. Conforme mencionado no capítulo 3, uma das características mais marcantes dessa linguagem é sua plena capacidade para a geração de códigos perfeitamente sintetizáveis. Além disso, essa linguagem se destaca bastante por sua plena capacidade de simulação. Uma observação muito interessante realizada durante o desenvolvimento desse projeto, é o determinismo que o código gerado pelo **Bluesim** é capaz de alcançar. Muito embora a arquitetura desenvolvida seja totalmente baseada em fluxo de dados e tenha sido implementada com bastante enfoque no dinamismo de determinados aspectos - em busca de conseguir variar diversas características da arquitetura em nível de compilação - o BSV sempre apresentou um resultado bastante coerente. Ao realizar a depuração das fichas geradas pela arquitetura é possível observar um perfeito determinismo na ordem de geração dessas fichas e consequentemente, no resultado final da execução dos programas, o que mostra a perfeita coerência do simulador de todo o circuito proposto. Essa característica é alcançada pela linguagem, por ser baseada em elaboração estática, para a geração de código *Verilog*, além de todo o tratamento de conflitos, realizado a partir da descrição TRS que ela é baseada.

Figura 7.5 – *Speedup* da segunda versão do projeto, para diferentes tamanhos de programas.

Fonte: Elaborada pelo autor.

Para atestar o correto funcionamento da arquitetura sintetizada em FPGA, foi utilizado um módulo extraordinário ao projeto. Esse módulo encontra-se com uma licença de código livre no *GitHub* e foi proposto pelo aluno Paulo Matias do Instituto de Física de São Carlos. O *AltSourceProbe*, nome atribuído a esse módulo particular, tem o objetivo de criar uma comunicação persistente entre a entrada *JTag* de uma placa de FPGA e a porta *USB* do computador onde essa placa estiver conectada (53). Para a transmissão de pequenos pacotes de dados, esse módulo possui grande capacidade para a realização dessa comunicação. Como as necessidades do presente projeto era simplesmente a impressão do resultado da computação do programa corrente, esse módulo atendeu perfeitamente todas as expectativas. Através do *AltSourceProbe*, um programa escrito em *Python*, fica verificando a todo instante se existem novas informações de saída da FPGA, em caso positivo, esse programa imprime o resultado na linha de comando onde ele estiver sendo executado. Além disso, foi utilizado o *display* de sete segmentos do kit utilizado para implementação, para apresentação do resultado independente da comunicação USB. Neste *display* é apresentado o resultado dos programas de *micro benchmark* utilizados para validação do protótipo, em formato hexadecimal.

Para a realização da síntese, foi utilizada uma placa de FPGA Altera Cyclone V 5CSEMA5-F31C6, em um kit Terasic DE1-SoC. Infelizmente não foi possível a realização da síntese a uma frequência de relógio superior a 71MHz, limitado à quantidade de duas Unidades Funcionais,

utilizando dois dutos de largura de banda ^{||}, o que pode ser considerado um grave problema para esse tipo de implementação. O presente trabalho sempre teve o objetivo principal de atestar o correto funcionamento da arquitetura proposta pela MDFM, utilizando tecnologias modernas, a partir de FPGAs. Para esse propósito, foi motivada a implementação de todas as unidades que compreendem essa arquitetura, a fim de coletar todas as informações da implementação com bastante agilidade, mesmo em nível de simulação. Essa característica implicou em uma grande sobrecarga de elementos lógicos combinacionais, principalmente na Unidade de Processamento, em particular, nos circuitos destinados para a distribuição de pacotes entre as Unidades Funcionais, bem como o circuito para o resgate das fichas que deixam essas unidades. Essa sobrecarga aumentou consideravelmente o caminho crítico da solução, implicando na limitação da frequência do relógio, para síntese na placa utilizada. Essa característica se deu principalmente pela necessidade de circuitos que fossem gerados dinamicamente, aliado a necessidade de entrada e saída simultânea de pacotes e fichas, nas Unidades Funcionais. Estudos futuros podem analisar valores fixos na relação entre dutos de largura de banda e unidades funcionais, a fim de conseguir limitar a generalidade com a qual foi desenvolvido esse primeiro protótipo, conseguindo explorar frequências de relógio muito maiores.

Ainda foi possível observar que a porcentagem de utilização dos elementos lógicos da Altera Cyclone V utilizada, não são suficientes para a síntese da solução, com valores superiores a cinco Unidades Funcionais, com dez dutos de largura de banda ^{**}. A partir desses valores, mais de 100% dos blocos lógicos dessa placa são necessários para a realização da síntese. Entretanto, a simplificação dos circuitos sugerida acima, já seria suficiente para reduzir também essa porcentagem de utilização da placa. Além disso, o modelo de FPGA utilizado é bastante simples, mais utilizados para fins didáticos apenas, de forma que esse tipo de limitação não figura um problema muito sério, uma vez que a porcentagem de utilização em placas de FPGAs de mais alto desempenho poderiam oferecer suporte para valores muito maiores dessas variáveis. Foi sugerida a utilização de um modelo de FPGA mais simples, para atestar a imensa capacidade da arquitetura, que pode apresentar ótimos resultados, mesmo em ambientes menos complexos.

A figura 7.6, apresente o kit de FPGA utilizado para a síntese da solução. Na figura, pode-se ver o resultado impresso no *display* de sete segmentos do kit. Foi implementada a estrutura da máquina com dois dutos de largura de banda e duas unidades, com o programa para o cálculo do somatorial de 120, que possui como resultado 7260, que em representação

^{||}Que é o segundo ponto das curvas de *speedup* apresentadas nesse capítulo.

^{**}Que é o quinto ponto, nas curvas de *speedup* apresentadas nesse capítulo

Figura 7.6 – Síntese da solução em um kit de FPGA da Terasic DE1-SoC, implementado com o programa do somatorial de 120.

Fonte: Elaborada pelo autor.

hexadecimal é igual a $0x1C5C$, como pode ser visto no *display* de sete segmentos da figura.

Embora a solução não tenha permitido a síntese para frequências superiores a 71MHz, a simulação nos apresentou resultados bastante promissores. Realizando alguns testes preliminares, muito pouco rigorosos, em processadores comerciais convencionais, foi recriado em C++ os micro *benchmarks* utilizados para a validação do protótipo. É possível observar que o tempo levado para a execução desses programas é ligeiramente mais lento, com relação a execução deste programa no protótipo apresentado nesse trabalho, se for possível operá-lo a uma frequência de pelo menos 100MHz, que é uma frequência muito inferior do que as frequências utilizadas nesses processadores analisados. Embora não tenha sido possível sintetizar em FPGA a solução, nessa frequência, esse resultado é extremamente motivador para seguir explorando as possibilidades que essa arquitetura ainda consegue oferecer. É importante enfatizar que os testes mencionados foram realizados de forma muito pouco rigorosa; há de se considerar muitas questões que não foram exploradas no presente trabalho.

CAPÍTULO 8

Conclusão

Um dos pontos cruciais do desenvolvimento, desse trabalho se deu na escolha da linguagem de descrição de hardware utilizada para a criação do protótipo. O **Bluespec SystemVerilog** foi capaz de fornecer agilidade para a aplicação das mais variadas e complexas modificações propostas. Mesmo para alterações significativas, como por exemplo a escolha do número de **Unidades Funcionais** que serão utilizadas no anel em uma determinada execução, definido a partir de uma constante, o BSV permitiu grande versatilidade. Claro que em alguns pontos a linguagem exige um pouco mais de trabalho em determinadas escolhas, em momentos que se exige total controle sobre o hardware que será disposto, como por exemplo, se precisarmos de maior controle na composição de uma *FIFO*. Entretanto, pode-se concluir que o desenvolvimento do projeto através do **Bluespec SystemVerilog**, forneceu uma grande autonomia sobre o hardware, garantindo bastante agilidade para o desenvolvimento.

Pôde-se verificar no capítulo 7, que a arquitetura proposta é capaz de oferecer um desempenho muito bom, com alterações relativamente pequenas, com relação ao protótipo de inspiração, a **MDFM**. Essa arquitetura foi proposta no final dos anos 70 e até os dias atuais, continua sendo capaz de oferecer um paralelismo muito difícil de ser alcançado nas máquinas de propósito geral mais recentes, mas que ainda utilizam arquiteturas puramente sequenciais. Os resultados mostrados na Figura 7.5 apresentam um *speedup*, que realmente nos motiva muito a seguir explorando toda a capacidade que essa arquitetura é capaz de oferecer.

A primeira fase do desenvolvimento, descrita em detalhes no capítulo 5, foi capaz de mostrar que a estrutura exata da MDFM, aplicada a tecnologias modernas, através do uso de FPGAs e linguagens de descrição de hardware de alto desempenho, não é a melhor abordagem possível. Na MDFM, as Unidades Funcionais operavam a uma frequência de relógio muito inferior com relação ao restante da máquina. Dessa forma, era possível a exploração massiva dessas unidades, a partir da estrutura proposta. Nos dispositivos mais modernos utilizados, é possível alocar as Unidades Funcionais, na mesma estrutura onde será alocado todo o restante da arquitetura, uma vez que a quantidade de blocos lógicos presentes em uma FPGA é muito grande. Com isso, pôde-se verificar um grande gargalo em todo o caminho que as fichas e pacotes devem percorrer, até chegar as Unidades Funcionais, que operando na mesma

frequência de relógio que o restante da arquitetura, passou a ser capaz de gerar novas fichas tão rápido, quanto a estrutura é capaz de fornecer pacotes a elas. Entretanto, mesmo com esse grande gargalo, já era possível verificar que a quantidade de ciclos gastos para a execução de programas, era bastante inferior àqueles obtidos na arquitetura de origem.

Já na segunda fase do desenvolvimento, descrita em detalhes no capítulo 6, foi possível obter alguns resultados realmente significativos. Adicionando a característica de dutos de largura de banda, o *throughput* de toda a máquina aumentou consideravelmente. Unidades que levavam mais de 2 ciclos para a geração de novos pacotes de saída, passaram a fornecer quantidades máximas iguais a quantidade de dutos de largura de banda presentes na estrutura, definidas em tempo de compilação. Com isso, foi possível obter curvas de *speedup*, muito próximas do ideal. Entretanto, a implementação dessa característica foi realizada de maneira muito custosa para a síntese do hardware proposto. Mesmo em nível de simulação, o processo de compilação era bastante demorado para quantidades superiores a 20 dutos de largura de banda, mostrando um indicativo bastante desagradável que foi confirmado no processo de síntese da solução. O circuito que realiza a ligação entre os dutos de largura de banda e as unidades funcionais ficou extremamente custoso, uma vez que havia a necessidade que a geração desse circuito fosse implementada de forma dinâmica. Essa característica particular, aumentou consideravelmente o caminho crítico do circuito lógico da solução, permitindo que a síntese da solução para quantidades de dutos de largura de banda superiores a dois dutos, e quantidades superiores a duas unidades funcionais fosse realizada somente a uma frequência inferior a 18MHz.

A síntese do protótipo ficou comprometida por causa da generalidade com que a implementação foi realizada. Embora a implementação não tenha atingido plenamente a todas as expectativas, foi possível atestar todos os objetivos iniciais do projeto. Foi verificada a viabilidade da implementação da arquitetura da MDFM em uma linguagem de descrição de hardware de alto nível, e os resultados obtidos a partir do simulador padrão do BSV, foram capazes de mostrar resultados muito bons.

Nesse momento, a máquina é capaz de fornecer curvas de *speedup* muito próximas do ideal. O PIFLOW mostra que o modelo de fluxo de dados dinâmico, implementado com tecnologias modernas, pode ser uma alternativa bastante atraente aos processadores sequenciais, com o potencial de apresentar um desempenho muito maior do que eles, em aplicações com alto grau de paralelismo.

REFERÊNCIAS

- 1 KISH, L. B. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, v. 305, n. 3-4, p. 144–149, 2002.
- 2 MARKOV, I. L. Limits on fundamental limits to computation. *Nature*, v. 512, p. 147–157, 2014.
- 3 SHARP, J. A. *Dataflow computing: theory and practice*. New York: Ablex Publishing Company, 1992.
- 4 GURD, J. R.; WATSON, I. Data driven system for high speed parallel computing - Part 2: Hardware design. *Computer Design*, v. 19, n. 6, p. 97–106, 1980.
- 5 EVRIPIDOU, P.; KYRIACOU, C. Data-flow vs control-flow for extreme level computing. In: DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, DFM, 2013, Edinburgh, Scotland, UK. *Proceedings...* Edinburgh: IEEE Computer Society, 2013. p. 9–13. doi: 10.1109/DFM.2013.17.
- 6 DIAVASTOS, A.; TRANCOSO, P.; LUJÂN, M.; WATSON, I. Integrating transactions into the data-driven multi-threading model using the TFlux platform. In: DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, DFM, 2011, Galveston, Texas USA. *Proceedings...* Galveston, Texas: IEEE Computer Society, 2011. p. 19–27. doi: 10.1109/DFM.2011.14.
- 7 ARVIND; AGERWALA, T. Data flow systems. *IEEE Computer Magazine*, v. 15, n. 2, 1982.
- 8 DENNIS, J. B. Data flow supercomputers. *IEEE Computer*, v. 13, n. 11, p. 48–56, 1980.
- 9 GLAUERT, J. R. W.; GURD, J.; KIRKHAM, C. C.; WATSON, I. The data flow approach, In: CHAMBERS, F. B.; DUCE, D. A.; JONES, G. P. (Eds.). *Distributed computing*, New York: Academic Press. 1984, p. 3-53.
- 10 GURD, J. R. The Manchester dataflow machine. *Computer Physics Communications*, v. 37, n. 1, p. 49–62, 1985.
- 11 GURD, J. R.; KIRKHAM, C. C. Dataflow: achievements and prospects. *Information Processing*, v 86, p. 61–68, 1986.
- 12 SRINI, V. P. An architecture comparison of dataflow systems. *IEEE Computer*, v. 19, n. 3, p. 68–87, 1986.

- 13 GOODMAN, D.; KHAN, B.; LUJÁN, M.; WATSON, I. Improved dataflow executions with user assisted scheduling. In: DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, DFM, 2013, Edinburgh, Scotland, UK. *Proceedings...* Edinburgh: IEEE Computer Society, 2013. p. 14–21. doi: 10.1109/DFM.2013.10.
- 14 VERDOSCIA, L.; VACCARO, R. Position paper: validity of the static dataflow approach for exascale computing challenges. In: DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, DFM, 2013, Edinburgh, Scotland, UK. *Proceedings...* Edinburgh: IEEE Computer Society, 2013. p. 38–41. doi: 10.1109/DFM.2013.15.
- 15 CICOTTI, P.; BADEN, S. B. Latency hiding and performance tuning with graph-based execution. In: DATA-FLOW EXECUTION MODELS FOR EXTREME SCALE COMPUTING, DFM, 2011, Galveston, Texas USA. *Proceedings...* Galveston, Texas: IEEE Computer Society, 2011. p. 28–37. doi: 10.1109/DFM.2011.15.
- 16 GIORGI, R. et al. TERAFLUX: Harnessing dataflow in next generation teradevices. *Journal of Microprocessors and Microsystems*, v. 38, n. 8, p. 976–990, 2014.
- 17 GURD, J. R.; KIRKHAM, C. C.; WATSON, I. The Manchester prototype dataflow computer. *Communications of the ACM*, v. 28, n. 1, p. 34–52, 1985.
- 18 WATSON, I.; GURD, J. R. A prototype dataflow computer with token labelling. In: INTERNATIONAL WORKSHOP ON MANAGING REQUIREMENTS KNOWLEDGE, AFIPS, 1979, New York, NY, USA. *Proceedings...* New York, 1979. p. 623–628. doi: 10.1109/A-FIPS.1979.14.
- 19 VEEN, A. H. Dataflow machine architecture. *ACM Computing Surveys*, v. 18, n. 4, p. 365–396, 1986. doi: 10.1145/27633.28055.
- 20 GURD, J. R.; KIRKHAM, C. C. Dataflow: achievements and prospects. *Information Processing*, v. 86, p. 61–68, 1986.
- 21 NIKHIL, R. Bluespec SystemVerilog: efficient, correct RTL from high level specifications. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS AND MODELS FOR CO-DESIGN. MEMOCODE '04, 2004. *Proceedings...* Waltham: IEEE Computer Society, 2004. doi: 10.1109/MEMCOD.2004.1459818.
- 22 NIKHIL, R. *Bluespec SystemVerilog training*. 2012. Disponível em: <http://www.demosondemand.com/dod/proddemos/vendors/pd_bluespec.aspx>. Acesso em: 30 junho 2015.
- 23 WIKIPEDIA. *Bluespec, Inc.* 2015. Disponível em: <https://en.wikipedia.org/wiki/Bluespec,_Inc.>. Acesso em: 02 janeiro 2016.

- 24 BLUESPEC INC. *High-level synthesis tools*. 2013. Disponível em: <<http://www.bluespec.com/high-level-synthesis-tools.html>>. Acesso em: 29 junho 2015.
- 25 NIKHIL, R. S.; ARVIND. What is Bluespec? *SIGDA Newsletter*, v. 39, n. 1, p. 1–1, 2009. doi: 10.1145/1862876.1862877.
- 26 HILL, F. J.; PETERSON, G. R. *Digital systems: hardware organization and design*. 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 1978.
- 27 BARRY ALLEY. *Astrocade add-under blueprints - astro-vision*. 1982. Disponível em: <http://www.ballyalley.com/documentation/misc_hardware_docs/misc_hardware_docs.html>. Acesso em: 25 junho 2015.
- 28 WIKIPEDIA. *Blueprint*. 2015. Disponível em: <<https://en.wikipedia.org/wiki/Blueprint>>. Acesso em: 03 janeiro 2016.
- 29 HOROWITZ, P.; HILL, W. *The art of electronics*. 3rd ed. Cambridge: Cambridge University Press, 1989.
- 30 RALF-UDO HARTMAN. *Laser show control history*. 1980. Disponível em: <http://drhart.ucoz.com/index/laser_show_control_history/0-136>. Acesso em: 25 junho 2015.
- 31 EETECH. *All about circuits*. 2003. Disponível em: <<http://www.allaboutcircuits.com/textbook/reference/chpt-7/example-circuits-and-netlists/>>. Acesso em: 25 junho 2015.
- 32 UNIVERSITY OF WASHINGTON. *Programmable logic devices*. 1999. Disponível em: <<https://courses.cs.washington.edu/courses/cse370/99au/lectures/au99-05/au99-05.pdf>>. Acesso em: 25 junho 2015.
- 33 HOLLEY, M.; PELLERIN, D. *PAL design specification*. 1991. Disponível em: <https://upload.wikimedia.org/wikipedia/commons/a/a0/PALASM_Design.jpg>. Acesso em: 29 junho 2015.
- 34 ZEIDMAN, B. *Designing with FPGAs and CPLDs*. Lawrence, Kansas, USA: CRC Press, 2002.
- 35 NATIONAL INSTRUMENTS. *Fundamentos da tecnologia FPGA*. 2013. Disponível em: <<http://www.ni.com/white-paper/6983/pt/>>. Acesso em: 29 junho 2015.
- 36 WIKIPEDIA. *VHDL*. 2015. Disponível em: <<https://en.wikipedia.org/wiki/VHDL>>. Acesso em: 02 janeiro 2016.

- 37 SMITH, D. VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C. In: CONFERENCE ON DESIGN AUTOMATION PROCEEDINGS, 33RD, 1996, Las Vegas, NV, USA. *Proceedings...* Las Vegas, NV: IEEE Computer Society, 1996. p. 771–776. doi: 10.1109/DAC.1996.545676.
- 38 GRAAF, A. *SystemC: an overview*. 2010. Disponível em: <<http://ens.ewi.tudelft.nl/Education/courses/et4351/SystemC-14v1.pdf>>. Acesso em: 29 junho 2015.
- 39 WADLER, P. The essence of functional programming. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 19TH, POPL '92, 1992. *Proceedings...* Albuquerque, New Mexico, ACM, 1992. p. 1–14. doi: 10.1145/143165.143169.
- 40 DAVE, N. H. *Designing a processor in Bluespec*. 2005. Master Thesis - Massachusetts Institute of Technology, Massachusetts, 2005.
- 41 GURD, J. R.; WATSON, I.; GLAUERT, H. *A multi-layered dataflow computer architecture*. 1980. Disponível em: <https://www.researchgate.net/publication/242410939_A_multilayered_data_flow_computer_architecture>. Acesso em: 07 janeiro 2016.
- 42 GURD, J.; KIRKHAM, C. C.; BÖHM, W. The Manchester dataflow computing system, In: DONGARRA, J. J. (Ed.). *Experimental parallel computing architectures*. Amsterdam. North Holland. 1987.
- 43 WATSON, I.; GURD, J. R. A practical dataflow computer. *IEEE Computer*, v. 15, n. 2, p. 51–58, 1982.
- 44 BÖHM, A. P. W.; SARGEANT, J. *Efficient code generation for SISAL*. Manchester: University of Manchester, 1985. Technical Report UMCS-85-10-2.
- 45 BUSH, V. I.; GURD, J. R. Transforming recursive programs for execution on parallel machines. *Lecture Notes in Computer Science*, v. 201, p. 350–367, 1985.
- 46 BÖHM, A. P. W.; SARGEANT, J. Code optimisation for tagged-token dataflow machines. *IEEE Transactions on Computers*, v. 38, n. 1, p. 4–14, 1989.
- 47 KIRKHAM, C. C. *The Manchester prototype dataflow system: basic programming*, 6th ed. Manchester, UK: UMC-DF-BPM, 1987.
- 48 BARAHONA, P. M. C. C.; GURD, J. R. Processor allocation in a multi-ring dataflow machine. *Journal of Parallel and Distributed Computing*, v. 3, p. 305–327, 1986.
- 49 SARGEANT, J.; RUGGIERO, C. A. Control of parallelism in the Manchester dataflow machine. *Lecture Notes in Computer Science*, v. 274, p. 1–15, 1987.

- 50 KAWAKAMI, K.; GURD, J. R. A scalable dataflow structure store. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA '86, 1986, New York, NY, USA. *Proceedings...* New York: ACM Digital Library, 1986. p. 243–250. doi: 10.1145/17356.17385.
- 51 BLUESPEC INC. *Bluespec SystemVerilog: reference guide*. 2012. Disponível em: http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/reference-guide.pdf. Acesso em: 03 março 2014.
- 52 JINKS, P. J. *A dataflow system simulator*. 1977. Master Thesis - Manchester University, Manchester, 1977.
- 53 MATIAS, P.; DUTRA, I. *AltSourceProbe*. 2014. Disponível em: <https://github.com/thotypous/altsourceprobe>. Acesso em: 21 dezembro 2015.
- 54 ROSENBAND, D. L. The ephemeral history register: flexible scheduling for rule-base designs. In: INTERNATIONAL CONFERENCE OF FORMAL METHODS AND MODELS FOR CODESIGN, MEMOCODE'2004, 4., 2004, San Diego, California, USA. *Proceedings...* San Diego: IEEE Computer Society, 2004. p. 189–198.

APÊNDICE A

Parser

A partir dos mesmos requisitos verificados para a escolha da linguagem de descrição de hardware utilizada para o desenvolvimento do presente trabalho, foi realizada uma breve análise sobre a linguagem de programação que seria utilizada para a criação do Parser - responsável pela tradução do código de linguagem montadora da MDFM, em uma codificação legível ao BSV - e quaisquer outras ferramentas de suporte necessárias para acompanhar o desenvolvimento do projeto. Dentre tantas linguagens de programação disponíveis, foram analisadas as que seriam capazes de oferecer grande facilidade para implementação dessa ferramenta, que fosse capaz de oferecer recursos poderosos para a manipulação de *Strings* principalmente. A linguagem mais próxima das necessidades propostas foi o Python. Essa linguagem foi concebida com a filosofia de enfatizar o esforço do programador, sobre o esforço computacional. Priorizando a legibilidade do código sobre a velocidade ou expressividade. Ela é capaz de oferecer uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão, ou através de um grande número de módulos e *frameworks* desenvolvidos por terceiros, em sua comunidade *Open Source*.

O Python é uma linguagem de propósito geral, de alto nível, multi paradigma, que suporta no mesmo ambiente: orientação a objetos; programação imperativa; programação funcional e/ou procedural. Possui tipagem dinâmica e uma de suas principais características é permitir grande legibilidade no código e exigir poucas linhas de código fonte, se comparado ao mesmo programa em outras linguagens como C ou C++, por exemplo. Devido as suas características, ela é principalmente utilizada para processamento de textos, dados científicos e criação de CGIs para páginas dinâmicas da web. Ou seja, a linguagem possui todos os recursos e atributos desejados para o desenvolvimento de ferramentas para suporte no presente projeto. Portanto foi escolhida, fundamentalmente para a criação do Parser da linguagem montadora da MDFM na PIFLOW.

A.1 Especificações da Ferramenta

O único objetivo do Parser é interpretar cada atributo presente no código em linguagem montadora gerado pelo compilador da MDFM e convertê-lo para um formato que o BSV seja capaz de identificar. A primeira etapa do programa consiste em analisar as linhas que contém código efetivamente, removendo-se comentários e linhas em branco. O código em linguagem montadora da MDFM possui suporte para adição de comentários em linha (não existe a possibilidade para comentários em blocos). De acordo com sua definição, toda informação contida após um caractere ";", refere-se a comentários e portanto não devem ser consideradas pelo Parser. (47)

Lembrando-se da sintaxe da MDFM, pode-se observar que o primeiro caractere de todas as linhas - onde cada linha possui uma única instrução - refere-se ao segmento da instrução, na arquitetura da máquina. O protótipo PIFLOW não fez a utilização de segmentos como a MDFM propõe, dessa forma o Parser dispõe as instruções de maneira sequencial. A partir do segmento 0 as instruções são organizadas em ordem crescente, baseado no endereço que ele foi proposto para a Unidade de Armazenamento de Instruções no processo de compilação. Ao término da análise desse segmento, o último endereço identificado passa a ser utilizado como um *Offset* para as instruções dos segmentos subsequentes. Dessa forma, ao final do processo de Parser, os endereços estarão organizados em ordem crescente, com todas as instruções contidas no "mesmo segmento", por assim dizer, pois conforme mencionado, não foram utilizados segmentos nesse projeto.

Para toda linha de código efetivo do programa em linguagem montadora, identificado pelo *Parser*, é aplicada uma expressão regular adicionando-se cada seção do formato do código em uma lista característica do Python. Portanto, serão mantidos apenas os atributos efetivos do código nesse elemento. A adição dessa informação em uma lista, é realizada para simplificar a identificação de cada informação individualmente nas próximas etapas. Cada uma dessas listas são então adicionadas a uma outra lista principal, que irá armazenar todas essas sub-listas. Dessa forma, todo código do programa proposto, estará inserido em uma lista de listas. Nesse instante é realizada a ordenação da lista principal em ordem crescente, a partir dos itens 0 e 1 de cada lista que compõe essa entidade principal. Ao final dessa operação todo o código estará ordenado por seu segmento e cada segmento estará ordenado por seu endereço de instrução.

Nesse ponto, é interessante reforçar os principais atributos da linguagem Python, enfatizados anteriormente, para o desenvolvimento dessa ferramenta. Toda operação descrita até esse ponto - considerando-se bastante cautela na separação das informações, através de condições

compostas - foi realizada em apenas 9 linhas de código Python, desde a identificação de código em linguagem montadora válido, à ordenação do conjunto de listas. Esse número mostra-se surpreendente, diante das linguagens mais convencionais, ainda mais quando consideradas para manipulação de *Strings*. Pode-se apresentar como exemplo da simplicidade que a linguagem oferece, a operação de ordenação da lista principal a partir das informações contidas em sua sub-lista, realizada a partir do código a seguir:

$$lines = sorted(lines, key = itemgetter(0,1))$$

Onde a variável *lines*, refere-se a lista principal que armazena cada linha do código em linguagem montadora, lembrando que ela possui uma outra lista em cada um de seus elementos.

Com essa lista de listas devidamente ordenada, é realizada a contagem da quantidade de instruções em cada segmento da MDFM, essa informação é armazenada em uma nova lista, onde cada índice refere-se ao segmento particular e seu valor contém o total de instruções desse segmento específico. Essa informação acrescida ao offset mencionado anteriormente é capaz de fornecer o endereço exato esperado para a PIFLOW. Esse endereço é fundamental no processo de Parser, pois ele será utilizado para identificar a instrução específica, além de ser referenciado pelas outras instruções para suas ligações com as instruções subsequentes.

Nesse instante, o programa passa a formatar cada linha do código em um formato hexadecimal reconhecido pelo BSV, chamado Verilog VMEM. Esse é o formato hexadecimal, próprio para carregamento dentro de simulações Verilog, usado através da função *\$readmemh* dessa linguagem. O arquivo de texto a ser lido deve conter apenas espaços em branco, comentários (ambos tipos de comentários C++ são suportados) e números hexadecimais. Durante a leitura do arquivo Verilog VMEM, cada número encontrado é marcado como um elemento de palavra sucessiva da memória. O endereçamento é controlado através do início de uma linha, ou através do término da palavra iniciada pelo identificador "@". Abaixo pode-se ver um exemplo de arquivo Verilog VMEM, contendo o valor "Hello Word[rq]", à ser carregado no endereço 0x1000:

```
@00000400 48656C6C 6F2C2057 6F726C64 0AFFFFFF
```

A.2 Definição das Características

O Parser da PIFLOW foi definido com palavras para a memória de 16 números hexadecimais, ou seja, 64 bits em cada endereço. Foi utilizado como padrão uma definição sequencial das instruções da PIFLOW baseadas no conjunto de instruções da MDFM (47). Ao final desse capítulo será mostrada uma imagem A.1, com o código em linguagem montadora da MDFM e o código interpretado pelo Parser lado a lado, a fim de ilustrar todo o processo descrito em detalhes a seguir:

- Os primeiros 4 números hexadecimais do formato, são utilizados para fazer o endereçamento da memória. Esse endereço é antecedido pelo identificador "@" (padrão do formato Verilog VMEM) e seguido por um espaço;
- Após a identificação do endereço, o primeiro número hexadecimal da linha, refere-se aos identificadores que determinam a saída lógica a ser considerada para uma instrução particular, conforme definido pela arquitetura da MDFM e descrito no capítulo 4;
- Os números 2 e 3 em conjunto, compõem o *Opcode* da instrução, eles são organizados conforme o manual de programação da MDFM e segmentados da seguinte maneira:
 - As sequências iniciadas pelos números 0, 1, 2, ou 3 referem-se a Funções Usuais, como por exemplo a instrução AND, que realizará essa operação lógica, na PIFLOW, é identificada pelos números hexadecimais 01. A instrução ADI, responsável pela operação de soma de números inteiros, na PIFLOW é identificada pelos números hexadecimais 0B. A instrução SBI, responsável pela instrução de subtração de números inteiros é identificada pelos números 13. Todas as outras operações dessa classe são definidas por esse formato, de maneira sequencial;
 - As sequências iniciadas pelo número 4, referem-se a instruções relacionadas a Fluxo de Controle. Nesse segmento pode-se observar como exemplo a instrução DUP, responsável pela duplicação de uma determinada ficha em ambas saídas lógicas da instrução, na PIFLOW ela é identificada pela combinação de números hexadecimais 40. Ou a instrução BRR, responsável pela realização de uma condição, baseado na entrada direita da instrução que deve ser do tipo booleano, ela será identificada na PIFLOW pela combinação 46;
 - As sequências iniciadas pelo número 5 referem-se a instruções relacionadas a Operações Ordinárias na arquitetura. Como por exemplo, a instrução GAN, que tem

por objetivo a geração explícita de um novo Nome de Ativação que será carregado no campo de dados em uma ficha, essa instrução é referenciada pelos números 53 na PIFLOW;

- As sequências iniciadas pelo número 6 referem-se a instruções relacionadas a operações de Contextos. Como por exemplo, a instrução PRP, que tem o objetivo de preparar uma ficha especial capaz de fazer emparelhamento com uma ficha que não está ligada explicitamente à instrução que fez sua geração. Essas fichas são usadas pelo compilador para reunir os resultados advindos de funções recursivas por exemplo, essa instrução será referenciada pela combinação 63;
 - As sequências iniciadas pelo número 7 referem-se a instruções de Mudança de Tags, como a instrução SAN, que a partir da instrução GAN, tem o objetivo de setar o novo Nome de Ativação para as instruções subsequentes, resultando em novas fichas com o novo Nome de Ativação gerado pela GAN, em sua composição, essa instrução é referenciada pela combinação 73 na PIFLOW;
 - As sequências iniciadas pelo número 8 referem-se a instruções relacionadas ao Arco Dinâmico, como a instrução SCD por exemplo, que tem por objetivo setar instruções vindas da instrução PRP, deixando-as prontas para o emparelhamento com suas fichas dinâmicas parceiras. Essa instrução é referenciada pelos números hexadecimais 81;
 - As sequências iniciadas pelo número hexadecimal C referem-se a instruções de Entrada e Saída da máquina. Como a instrução OPT por exemplo, que tem o objetivo de gerar fichas de saída para a máquina. Essas fichas passarão para o Switch a fim de imprimir o resultado do programa em um terminal. *
- A sequência de números do caractere 4 ao 7, são destinados à especificação do endereço da instrução esquerda subsequente à instrução definida na linha atual;
 - O número definido pelo caractere 8, determina se a ficha resultante da instrução definida na linha atual, entrará à esquerda, ou a direita da instrução esquerda subsequente a essa. Esse número será 0 caso seja destinado ao lado esquerdo da instrução esquerda subsequente, ou 1 caso seja destinado ao lado direito da instrução esquerda subsequente;
 - O número definido pelo caractere 9 pode assumir até 8 valores distintos, ele determina qual a função de emparelhamento da instrução esquerda subsequente. Dentre os valores

*Para a validação da PIFLOW, apenas um subconjunto do conjunto de instruções da MDFM foi implementado, porém o Parser foi projetado com a finalidade de ser capaz de acomodar todas as instruções propostas para a MDFM. Por esse motivo ele foi segmentado através das categorias descritas acima. O conjunto de instruções completo pode ser verificado no manual de referência da MDFM referenciado anteriormente.

previstos pelo conjunto de instruções da MDFM, vale destacar por sua importância no presente trabalho, os valores a seguir:

- **BY**: Determina que a ficha esquerda subsequente será do tipo *Bypass*, portanto não precisa ser emparelhada na Unidade de Emparelhamento, pois não possui uma ficha parceira, dessa forma essa ficha passará diretamente à saída dessa unidade no seu formato especificado;
- **EW**: Determina que a ficha subsequente será do tipo *Extract and Wait*, ao chegar na Unidade de Emparelhamento deve verificar se sua ficha parceira já encontra-se na unidade e está aguardando-a, nesse caso essa ficha será resgatada e conjuntamente com a ficha recém chegada formará o pacote de saída da unidade. Se a ficha parceira não estiver na unidade, a ficha recém chegada deverá ficar armazenada para aguardar emparelhamento;
- Os números definidos a partir do caractere 10 pode representar diferentes operações, já que o formato do código em linguagem montadora da MDFM permite até 3 formatos de instruções, conforme mencionado no capítulo 4. Lembrando que cada endereço de memória deverá possuir exatamente 16 números hexadecimais, deve-se observar que caso a informação seguinte não possuir 7 números, então ela será completada com F's para atingir os 16 números previstos. Os caracteres seguintes podem conter os seguintes formatos:
 - Se a instrução da linha não possui nenhuma outra referência além da instrução esquerda subsequente descrita anteriormente, nada deve ser adicionado nos próximos números, portanto eles serão completados com F's;
 - Se a instrução da linha atual refere-se ao endereço da instrução direita subsequente a instrução atual, então os próximos números seguirão exatamente o mesmo comportamento dos caracteres definidos dos números 4 ao 9 descritos anteriormente, sendo os números de 10 à 13, responsáveis por identificar o endereço da instrução direita subsequente e assim por diante;
 - Se a instrução da linha atual possuir um dado literal definido, ele será determinado a partir desse número. O caractere 10 irá representar o tipo de dado que refere-se, podendo assumir 16 valores distintos na PIFLOW, que são alguns dos tipos de dados suportados pela MDFM. Os próximos números referem-se ao dado literal especificado [†].

[†]Se a instrução corrente possuir um dado literal definido, o Parser irá somar 4 no valor da saída lógica esquerda

0 0NKI0 0LBY	@0000 041000000FFFFFFF
0 1ROPT0 0LBYG0	@0001 6C1000000B000000
1 0LSWA1 1LBY1 2LBY	@0002 17C0003000000400F
1 1RSYN1 4LEWI7	@0003 64C0006010000007
1 2RSYN1 3LBYI2	@0004 64C0005000000002
1 3NDUP1 4REW	@0005 040000611FFFFFFF
1 4NMUL0 1LBY	@0006 0000100FFFFFFF

Figura A.1 – Comparação entre Linguagem Montadora e Verilog VMEM

Fonte: Elaborada pelo autor.

Na Figura A.1 pode-se verificar o código apresentado no capítulo 4, disposto lado a lado com o código gerado através do Parser. Observe a instrução de endereço @0002 do código interpretado pelo Parser, por exemplo. Note que ele refere-se a instrução de endereço 0 do segmento 1 no código da MDFM, lembre-se que na PIFLOW não foram utilizados os segmentos característicos, previsto pela MDFM. Essa instrução - do endereço @0002 - deve considerar como saída lógica, apenas a instrução gerada em sua saída esquerda, identificada como 1 no código interpretado. Deverá ser executada a instrução com *Opcode* definido como 7C, equivalente à instrução SWA, que tem por objetivo atribuir um AN válido a ficha corrente. Essa instrução será ligada a instrução de endereço @0003 em sua saída esquerda subsequente (equivalente à instrução de endereço 1, do segmento 1, no código em linguagem montadora). Ele deve entrar no lado esquerdo dessa instrução (L) e não deve ser emparelhada (BY), informação identificada como 00 pelo Parser. A instrução corrente também deverá estar conectada a instrução de endereço @0004 (que é a instrução de endereço 2, no segmento 1 do código). A informação também estará ligada no lado esquerdo dessa instrução (L) e não deve ser emparelhada (BY), bem como a saída esquerda da presente instrução.

subsequente. Essa característica foi necessária para garantir a possibilidade de diferenciar os casos entre dado literal e instrução direita subsequente, dando controle ao hardware para analisar tal característica e realizar a operação correta necessária para cada caso. Dessa forma toda instrução que possuir um dado literal em sua composição terá sua saída lógica esquerda maior que 3, garantindo o reconhecimento desse caso particular.

APÊNDICE B

Registradores de Histórico Efêmero

Pôde-se perceber, no capítulo 3, a árdua tarefa empregada pelo escalonador do BSV para o controle de regras. De fato, qualquer linguagem para descrição de hardware baseada em regras, possui essa característica. Essa dificuldade se dá pela ocorrência de diversos conflitos para acesso à registradores convencionais. Conforme descrito em detalhes no capítulo 3, as regras são atômicas. Além disso, todas essas regras são escolhidas para serem executadas dentro de um mesmo ciclo, a menos que condições implícitas (controladas pela escalonador do compilador da linguagem HDL) e/ou explícitas (controladas pelo programador) impeçam-nas de fazê-las. Desse modo, grande esforço é empregado nas operações realizadas pelo escalonador, a fim de se obter o máximo em concorrência dessas regras, dentro de um único ciclo de *clock*. Entretanto, os resultados obtidos a partir desses esforços, acabam por fornecer códigos RTL (como VHDL, ou Verilog) muito semelhantes àqueles desenvolvidos diretamente nessas linguagens. Grandes projetos acabam por apresentar resultados imprevisíveis, ou mesmo desempenho inferior ao esperado, devido a métodos de escalonamento ineficientes. Muitas vezes, a única maneira de se obter o escalonamento desejado, é por meio da configuração manual das características esperadas para garanti-lo. Porém, um equívoco qualquer nesse processo, pode levar a *designs* com baixo desempenho, ou ainda com erros na execução.

A partir desse problema observado, foi introduzido um algoritmo de escalonamento que se baseia em um novo elemento de estado primitivo, chamado Registrador de Histórico Efêmero EHR, do inglês Ephemeral History Register (54). Além de garantir aumento de desempenho, o EHR é capaz de remover a possibilidade de falhas na execução. De forma que, se o desenvolvedor se equivocar ao introduzir configurações de escalonamento, embora o desempenho resultante possa não ser satisfatório, ao menos o comportamento do hardware resultante poderá ser explicado a partir da sequência lógica de regras que será proposta pelo escalonador.

A ideia geral em torno do EHR, é a possibilidade de transferir informações entre regras distintas, conseguindo ainda manter a premissa da linguagem, a atomicidade das ações. Como dito anteriormente, trata-se de um elemento de estado primitivo, como um registrador convencional. Se uma regra estiver escrevendo em um registrador e outra regra estiver lendo esse mesmo registrador, o EHR é capaz de garantir que o valor lido, será o mesmo valor escrito

pela outra regra, dentro do mesmo ciclo. Em um registrador primitivo convencional, não são permitidos múltiplos acessos concorrentemente. O escalonador da linguagem se encarrega de garantir que isso não aconteça sobre qualquer circunstância. No entanto, o EHR possui um histórico de todos os estados que esse elemento obteve dentro de um mesmo ciclo de *clock*, permitindo que sejam acessados todos esses valores concomitantemente. Essa característica fornece ao desenvolvedor maior autonomia para reordenar o escalonamento de acordo com suas necessidades, sem a necessidade de aumentar consideravelmente o caminho crítico do seu *design*. Em contraste com a abordagem padrão de registradores em uma linguagem HDL baseado em regras, o EHR fornece uma ferramenta capaz de oferecer um escalonamento dinâmico. Através de algumas diretivas limitadas, o BSV oferece a possibilidade de pouca intervenção no escalonamento. Muitas vezes, um aspecto dinâmico pode ser necessário para o desenvolvimento, pois o hardware desejado pode apresentar diferentes características específicas, como:

- Grande quantidade de dados que precisam transitar em caminhos condicionais dependentes, cada qual com suas definições de tempo e recursos únicos;
- Pode possuir subsistemas com variáveis e latências imprevisíveis a priori;
- Possuem eventos de entrada que não podem ser definidos estaticamente;

Observe que, em essência, é como se cada ciclo estivesse dividido em subciclos e cada regra fosse atribuída a um subciclo específico. Dessa forma, os valores que são escritos nos registradores de um subciclo, podem ser lidos pelas regras presentes nos subciclos posteriores. Por exemplo, imagine a implementação de uma FIFO básica, seu método *enqueue* poderia ser capaz de verificar quando o registrador que marca seu estado como **cheio**, for alterado pelo método *dequeue*. Dessa forma, se essa FIFO estiver inicialmente cheia, diante da operação de *dequeue*, o método *enqueue*, que aguardava para escrever nessa entidade, poderia ser capaz de fazer essa operação no mesmo ciclo de *clock* onde o *dequeue* foi efetuado. Isso seria capaz de oferecer um ganho em desempenho substancial, dependendo da sua aplicação, sem grande aumento do caminho crítico para a realização da operação.

A seguir você poderá ver um diagrama ilustrativo do circuito do módulo EHR, na figura B.1. Identifique o índice super-escrito de cada método, como a instância deste, por exemplo, *write*² é a instância 2 do método *write*. Observe que cada método *write* possui dois sinais associados a ele. O sinal *x* identifica a entrada de dados desse método. O sinal *en* identifica o controle de entrada dele, de forma que um valor não pode ser escrito no método a menos que esse sinal esteja ativado.

Figura B.1 – Registrador de Histórico Efêmero

Fonte: ROSEBAND (54)

Algumas notas são importantes de serem observadas:

- $read^0$ retorna o valor corrente do registrador;
- Se $write^i$ não estiver ativo para qualquer i , então o valor do registrador não é alterado;
- $write^i$ prevalece sobre $write^j$, para todo $i > j$;
- $read^i$ retorna o valor presente em $write^j$, onde j é o maior valor, menor que i , que possuir um $write$ ativo. Retornando o valor do **registrador** caso não houver nenhum $write$ ativo.

Qualquer quantidade de $read$ e $write$ é permitida para a instância de um EHR. Por exemplo, um EHR com apenas um $read$ e apenas um $write$ funciona exatamente igual a um registrador convencional, essa característica evidencia o caráter de estado primitivo atribuído à essa implementação. Recordando do exemplo da implementação de uma FIFO, imagine se ao invés de registradores convencionais para o armazenamento do dado e para o sinal de controle que verifica se a FIFO está cheia, módulos EHR os substituírem. A seguir, será apresentada uma implementação de exemplo de uma FIFO, capaz de realizar remoções e escritas dentro de um mesmo ciclo:

Note que todos os métodos estão interligados pelo módulo EHR, através do sinal que verifica a disponibilidade de escrita na FIFO. Essa ligação pode ser observada pelos predicados

```

enqueue x =
    data.write1 x;
    full.write2 1;
when (full.read2 == 0)

dequeue =
    full.write1 0;
when (full.read1 == 1)

first =
    return data.read0;
when (full.read0 == 1)

bypass =
    return data.read2;
when (full.read3 == 1)

```

Figura B.2 – Implementação de uma FIFO

Fonte: ROSEBAND (54)

dispostos em cada método. Essas verificações permitem que todos esses blocos possam ser executados dentro de um mesmo ciclo, seguindo um fluxo lógico determinado pelo programador. Observe a sequência que pode ser propagada para cada subciclo determinado pelo módulo EHR. O método *first* utiliza a instância 0 do sinal que indica a disponibilidade da FIFO, de forma que se esse sinal for **Verdadeiro**, retornará a instância 0 presente no EHR que guarda o valor corrente nessa entidade. O método *dequeue* verifica se a FIFO está cheia, na instância 1 do EHR. Em caso positivo, o sinal que indica a disponibilidade da entidade recebe o valor **Falso**, liberando a FIFO para receber um novo valor. É importante observar que essas operações poderão ser realizadas dentro do mesmo ciclo de *clock*, então logo após o resgate da informação presente no topo da fila, ela já estará pronta para receber novos dados. Na instância 2 do EHR, que indica a disponibilidade da FIFO, o método *enqueue* verifica se a entidade está cheia, se essa informação for **Falsa**, o método escreverá o parâmetro de entrada dele, na instância 1 do módulo EHR que armazena o dado e escreverá **Verdadeiro** na instância 2 do módulo EHR que indica a disponibilidade. Por fim, o método *bypass* verifica se existe dado na FIFO, a partir da instância 3 do EHR, que indica a disponibilidade da FIFO e em caso positivo retornará o dado presente na instância 2 do EHR, que armazena o dado.

Observe portanto, que é possível resgatar 2 informações dessa FIFO, dentro do mesmo ciclo de *clock*. Em um registrador convencional essa operação poderia ser realizada pelo mínimo de 4 ciclos. Note que o método *bypass* apenas retorna o dado presente no módulo

EHR que armazena a informação corrente da FIFO. Dessa forma, o método *dequeue* deverá ser utilizado no próximo ciclo, caso o valor lido pelo método *bypass* tenha sido efetivamente utilizado. É importante se recordar que caso não tenha sido escrito um novo valor na FIFO, no ciclo corrente, mesmo que o método *bypass* utilize as últimas instâncias previstas para ambos os módulos EHRs presentes nessa entidade, ele ainda retornará o valor corrente da FIFO, pois esse valor se propaga por todas as instâncias desse módulo, conforme pode ser visto na figura de sua arquitetura B.1.

Esse exemplo apresenta apenas a estrutura de uma FIFO capaz de realizar as operações previstas por esse tipo de entidade, dentro de um mesmo ciclo de *clock*. A partir dessa estrutura é possível escrever qualquer *design*, capaz de lidar com essa sequência de operações implementadas por esse módulo de FIFO particular. A sequência implementada segue uma arquitetura não convencional e pode ser alterada de acordo com as necessidades do programador. Nesse exemplo, foi mostrado apenas as possibilidades que o módulo EHR oferece para lidar com as operações atômicas previstas por uma linguagem HDL baseada em regras, que registradores convencionais não são capazes de garantir.

No presente projeto, o módulo EHR foi utilizado para lidar com as características dinâmicas da arquitetura proposta. A primeira fase do desenvolvimento, descrita em detalhes no capítulo 5, forneceu uma estrutura concisa para a implementação da MDFM descrita em BSV. Para a implementação da segunda fase do desenvolvimento (capítulo 6) foi definido um valor constante que representa a largura de banda da nova arquitetura. Para utilizar todos os componentes presentes na primeira fase do desenvolvimento, toda a estrutura de regras foi replicada a partir de laços, baseados no valor que representa a largura de banda. Mas, essa operação gerava diversos conflitos para acesso a FIFOs e registradores em diferentes regras. As diretivas nativas do BSV, que foram largamente exploradas para o desenvolvimento da primeira fase do projeto, não eram capazes de lidar com todas as instâncias de regras que a segunda fase do desenvolvimento gerou. Para lidar com todos esses conflitos o módulo EHR foi utilizado, a fim de realocar o escalonamento das regras a partir de seus predicados.

A última versão estável do **Bluespec SystemVerilog** (2014.07.A) incorporou o módulo EHR em suas bibliotecas padrões.

APÊNDICE C

Exemplo de Bluespec

Conforme pôde ser visto no capítulo 3, o **Bluespec SystemVerilog** é uma linguagem para descrição de hardware muito peculiar. Sua maneira de tratar o aspecto comportamental do hardware através de regras atômicas, é o que mais a difere das outras linguagens HDL mais convencionais. Porém, a correta utilização dessa característica, é o que garante o grande ganho em produtividade que essa linguagem é capaz de oferecer. Como acontece para qualquer linguagem de programação, independente do propósito, seja para o desenvolvimento de software ou hardware, é natural que o entendimento seja mais conciso a partir de exemplos de aplicações práticas. Para isso, considere o seguinte problema abstrato. O exemplo apresentado nesse apêndice foi retirado do conjunto de documentos para o Treinamento de Bluespec, desenvolvido pela Bluespec Inc. (22).

Figura C.1 – Distribuição de pacotes entre FIFOs

Fonte: NIKHIL (22)

Pacotes chegam em duas FIFOs de entrada e precisam ser alternados para outras duas FIFOs de saída, de acordo com a informação presente no primeiro *bit* do pacote de entrada. Além disso, alguns pacotes de interesse devem ser contabilizados, como disposto na figura C.1. Conforme descrito a seguir, algumas outras especificações também devem ser consideradas:

- As FIFOs de entradas podem estar vazias;
- As FIFOs de saída podem estar cheias;
- Podem haver conflitos de recursos compartilhados nas FIFOs de saída, se:
 - Existirem pacotes disponíveis em ambas as FIFOs de entrada;
 - Ambas as FIFOs de entrada possuírem mesmo destino; e
 - A FIFO de destino não estiver cheia.
- Podem haver conflito de recursos compartilhados no contador, se:
 - Houverem pacotes disponíveis em ambas as FIFOs de entrada;
 - Cada FIFO de entrada possuir destino diferente;
 - Ambas as FIFOs de saída não estiverem cheias; e
 - Ambos os pacotes precisarem ser contabilizados.
- Em caso de conflitos a primeira FIFO de entrada deve possuir prioridade mais alta;
- A solução deve possuir *throughput* máximo, ou seja, um pacote deve sair imediatamente se ele puder, obedecendo-se todas as regras especificadas acima.

```

module mkSmallSwitch (SmallSwitch)
  Reg#(int)  c  <- mkReg(0);
  FIFO#(Dat) i1 <- mkSizedFIFO(4);
  FIFO#(Dat) i2 <- mkSizedFIFO(4);
  FIFO#(Dat) o1 <- mkSizedFIFO(4);
  FIFO#(Dat) o2 <- mkSizedFIFO(4);

  (* descending urgency = "r1, r2" *)
  rule r1; // for packets from FIFO i1
    let x = i1.first;
    let out = (x[0] == 0) ? o1 : o2;
    i1.deq(); out.enq(x);
    if(count(x)) c <= c + 1;
  endrule: r1

  rule r2; // for packets from FIFO i2
    let x = i2.first;
    let out = (x[0] == 0) ? o1 : o2;
    i2.deq(); out.enq(x);
    if(count(x)) c <= c + 1;
  endrule: r2
  ...

```

```

Typedef Bit#(8) Dat;
function bool count(Dat a);
  ...
endfunction: count

```

Figura C.2 – Implementação de um módulo para a distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

A figura C.2, apresenta uma possível implementação do problema proposto, a partir do BSV. Observe a grande simplificação que a linguagem é capaz de oferecer para a resolução de um problema bastante abstrato e com um grande conjunto de especificações. É importante lembrar que será mostrado o funcionamento das regras do BSV, que determina os aspectos comportamentais da linguagem. Dessa forma, não estão dispostos os métodos que irão compor as entradas e saídas dessa implementação. Ao contrário, o exemplo irá focar no funcionamento das regras. Imagine apenas que existem outros módulos responsáveis por gerar novos pacotes na entrada desse módulo e outro responsável por consumir os pacotes processados aqui. Observe que a maneira como são gerenciados esses módulos não importa, de alguma forma, segundo o gerenciamento de suas próprias regras, existe um controle para a geração e consumo desses pacotes. Um aspecto interessante que deve-se notar, é a ausência de predicados na definição das regras (as condições explícitas, definidas pelo programador), que estariam logo após a determinação do nome da regra caso fossem necessárias. Na verdade, todas as especificações propostas, são garantidas a partir das condições implícitas, verificadas automaticamente pela linguagem.

As operações *enq* e *deq* nativas nas FIFOs padrões do BSV, tem o objetivo de alterar as informações presentes nos registradores que compõem essa entidade. Com isso, o BSV identifica a ocorrência de conflitos entre essas regras e seu escalonador se encarrega de agendar a execução de cada uma delas em ciclos distintos. Observe que isso garante de maneira definitiva a correção de conflitos nas FIFOs de saída. Da mesma forma, o registrador identificado como o registrador *c*, que armazena a contagem dos pacotes de interesse, é atribuído com novos valores em ambas as regras. O escalonador do BSV também identifica esse conflito e impede a execução deles no mesmo ciclo, garantindo a ausência de conflitos no contador. Além disso, existe a diretiva padrão do BSV identificada como *descending_urgency*, definida antes das definições das regras. Essa diretiva determina a prioridade das regras em caso de conflitos. No exemplo, caso exista qualquer conflito entre os recursos dessas regras, a regra *r1* deverá ser executada antes da regra *r2*. Isso garante que a primeira FIFO de entrada possua prioridade mais alta que a segunda. Observe que a regra *r1* é destinada para o resgate das informações contidas na primeira FIFO, característica determinada pelas operações: *i1.deq()* e *i1.first()*.

Além da análise de conflitos nesses recursos compartilhados, o BSV ainda identifica a utilização da operação *enq* em uma regra e verifica se a FIFO onde é aplicada essa operação não está cheia. Caso essa afirmação seja verdadeira, o escalonador se encarrega de não executar essa regra, até que a FIFO não apresente mais essa característica. De forma equivalente, o escalonador faz a mesma verificação sobre a operação *deq*, se certificando que a entidade onde essa operação é executada não encontra-se vazia. Com isso, o BSV se encarrega de garantir

outras duas especificações, verificando automaticamente se as FIFOs estão cheias ou vazias. Em caso de conflitos, a primeira regra será executada, enviando seus pacotes a sua FIFO de saída correspondente. Porém, no ciclo onde não houver qualquer conflito, ambas as regras poderão executar paralelamente. Isso garante a última especificação, que exige *throughput* máximo, de acordo com as regras determinadas.

Note que o identificador *Dat*, refere-se a um tipo de dados extraordinário à linguagem, definindo um tipo de dado de 8-bits onde ele for utilizado, como por exemplo, nas FIFOs que compõem o módulo, definidas antes da caracterização das regras. Além disso, a função *count* tem o objetivo de identificar os pacotes de interesse para a realização da contagem e foi omitida, pois não é o caso de interesse para a explicação do módulo proposto. A seguir será explicado o funcionamento de cada ação do módulo nos primeiros ciclos de sua execução.

Figura C.3 – Ciclo zero de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

Baseado na figura C.3, suponha que já no ciclo zero da execução haja um impedimento para o consumo dos pacotes da FIFO de saída um. Além disso, no início desse ciclo não houve a ocorrência de pacotes destinados à FIFO de entrada dois, de forma que a regra responsável por seu gerenciamento não será executada, uma vez que o método *i2.first* não poderá ser executado. O pacote presente na FIFO de entrada um, possui o primeiro *bit* igual a um, de forma que ele será encaminhado à FIFO de saída dois, de acordo com as especificações. Suponha que a função que realiza a verificação da contagem de pacotes tenha identificado que este pacote não deve ser contabilizado, portanto não é necessário incrementar o contador.

Figura C.4 – Ciclo um de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

Para o próximo ciclo, apresentado na figura C.4, ambas as FIFOs de entrada receberam novos pacotes para serem analisados pelo módulo, dessa forma, ambas as regras são candidatas a execução. O pacote inserido a partir da primeira FIFO possui o primeiro bit igual a zero, portanto ele será encaminhado para a FIFO de saída um. Além disso, a função que verifica a contagem de pacotes retornou verdadeiro para esse pacote, portanto ele deve incrementar o registrador de contagem de pacotes. O pacote vindo da FIFO de entrada dois possui o primeiro bit igual a um, então esse pacote deverá ser encaminhado a FIFO de saída dois. Ademais, ela não precisa ser contabilizada de acordo com a verificação de contagem. Note portanto, que não existe nenhum conflito de recursos a partir dos pacotes de entrada. Dessa forma, ambas as regras podem ser executadas paralelamente, obtendo o máximo desempenho do módulo, nesse ciclo particular.

No ciclo dois de execução (figura C.5) novamente, ambas as FIFOs de entrada receberam novos pacotes. Dessa forma, ambas as regras são candidatas a serem executadas. Porém, nesse ciclo, existe um conflito de recursos. Ambas as fichas de entrada possuem o primeiro bit iguais a zero, de forma que ambas as regras deveriam acessar a FIFO de saída um. O BSV verifica essa condição e busca uma solução de escalonamento para lidar com a situação. Como foi definido pelo usuário que em caso de conflitos, a primeira regra deveria ter prioridade, então assim será realizado. A segunda regra será inibida de realizar as suas operações e será agendada para ser executada no próximo ciclo, de forma que, se não houver novo conflito, o módulo

Figura C.5 – Ciclo dois de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

irá consumir esse pacote de entrada. Além disso, a verificação de contagem verificou que o pacote de entrada da primeira FIFO deve ser contabilizado, então o registrador responsável será incrementado. Observe que ainda existe um bloqueio para o consumo dos pacotes na FIFO de saída um, de forma que haverá um acúmulo de fichas nessa FIFO, a partir do próximo ciclo. Observe também que foram consumidos todos os pacotes que foram destinados a FIFO de saída dois, até o presente momento.

Para o próximo ciclo, figura C.6, chegaram novos pacotes em ambas as FIFOs de entrada. Note que ocorreu um acúmulo de pacotes na FIFO de entrada dois, uma vez que o pacote dessa entidade não pôde ser consumido no último ciclo. Nesse ciclo, ocorre exatamente como no ciclo um, onde cada pacote é destinado para uma FIFO de saída específica. Porém, nesse caso, nenhum dos pacotes precisam ser contabilizados. Dessa forma, não existe qualquer conflito de recursos, então ambas as regras serão executadas paralelamente.

Observe que nenhum pacote chegou a nenhuma das FIFOs de entrada nesse ciclo, da figura C.7. Dessa forma, somente a regra responsável pela distribuição da FIFO de entrada dois será executada. Isso porque houve um acúmulo de pacotes nessa FIFO, devido ao conflito de recursos encontrado no ciclo dois. Assim sendo, a ficha possui o primeiro bit igual a zero e a função de verificação para a contagem de pacotes retornou verdadeiro, de forma que o registrador de contagem será incrementado. A FIFO de saída um continua bloqueada para o

Figura C.6 – Ciclo três de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

consumo de pacotes, de forma que no próximo ciclo, essa FIFO estará cheia, com isso, não admitindo novos pacotes, conforme poderá ser visto a seguir.

No ciclo da figura C.8, um novo pacote chegou apenas a partir da FIFO de entrada um, de forma que a segunda regra não será executada, pois o predicado intrínseco de FIFO vazia do BSV, irá inibir a execução desta. Mas, aqui pode-se observar um novo fator, o pacote de entrada da FIFO um, possui o primeiro bit igual a zero. Dessa forma, ele deveria ser encaminhado a FIFO de saída um. Entretanto, note que essa FIFO de destino do pacote encontra-se cheia, de forma que a condição implícita do BSV, que verifica a disponibilidade para a execução do método *enq* nas FIFOs controladas pela regra, irá inibir a execução dessa regra também. Sendo assim, nenhuma das duas regras serão executadas nesse ciclo.

Finalmente, no ciclo seis de execução desse módulo, visto na figura C.9, o método ligado a FIFO de saída um, passou a consumir os pacotes dessa FIFO. Além disso, existem pacotes em ambas as FIFOs de entrada, cada uma destinada a uma das FIFOs de saída. Entretanto, ambas precisam ser contabilizadas, precisando acessar o registrador que realiza essa contagem. Como a primeira regra possui maior prioridade no caso de conflitos, então somente essa regra será executada. O pacote presente na outra FIFO de entrada, necessariamente será consumida no próximo ciclo, uma vez que não existe novo pacote chegando na FIFO um, de forma que não existem meios de ocorrerem conflitos de recursos no próximo ciclo. Ademais, caso não

Figura C.7 – Ciclo quatro de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

exista novo bloqueio no método que resgata os pacotes da FIFO de saída um, todos os pacotes presentes nela serão consumidos a cada novo ciclo seguinte.

Você pôde verificar nesse apêndice a grande facilidade para implementação e mesmo para leitura e acompanhamento de um código escrito em BSV. Essa facilidade para leitura permite a análise minuciosa do comportamento de seu *hardware*, o que garante uma depuração mental em cada módulo específico do resultado. Note que, sempre é possível manter um *hardware* tão pequeno quanto se queira, como no módulo do exemplo proposto. A partir do qual poderá conectar nele outros módulos mais complexos, construindo assim o projeto final que pode possuir o tamanho que for, mas constituído de pequenos blocos com tarefas simples. A fácil leitura e entendimento de cada módulo, conforme mostrado por esse exemplo, mostra a imensa capacidade de documentação de um código BSV. A partir do momento que se entende como funcionam as condições implícitas das regras, que tem o objetivo apenas de facilitar o desenvolvimento, toda a criação do código fica muito simplificado. Imagine a implementação do módulo proposto escrito em uma linguagem de descrição de hardware convencional, como o Verilog, ou o VHDL. Necessariamente toda a verificação de conflitos deveria ser implementada explicitamente, por exemplo. Somente o controle dos possíveis conflitos descritos pelas especificações do problema, já deixaria o código HDL muito mais complexo e que em BSV podem ser omitidos, sem perda de generalidade. Esses foram uns dos principais argumentos que motivaram a escolha dessa linguagem para o desenvolvimento

Figura C.8 – Ciclo cinco de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

do presente projeto.

Figura C.9 – Ciclo seis de execução do módulo para distribuição de dados entre FIFOs.

Fonte: NIKHIL (22)

APÊNDICE D

Exemplo de execução na MDFM

No capítulo 4, foi mostrado o exemplo da execução de um programa bastante simples, utilizado somente para métodos didáticos, pois ele não tem o objetivo de realizar nenhuma computação de dados complexa. Ele buscava simplesmente a realização da multiplicação entre dois números definidos estaticamente. Para fins didáticos, ele funciona muito bem. Entretanto, nesse apêndice, será apresentado o funcionamento passo-a-passo de um programa que tem o objetivo de realizar o cálculo numérico de uma integração. É importante ressaltar que o programa proposto nesse apêndice, não refere-se ao mesmo programa de micro *benchmark* utilizado nos testes de validação da PIFLOW. O programa que será mostrado a seguir foi retirado do artigo (17), bem como parte da explicação utilizada aqui. Ele será retratado no presente trabalho apenas para complemento da base de conhecimento utilizada para o desenvolvimento da PIFLOW. Para maiores informações do exemplo, verificar o trabalho citado.

Bem como referido no apêndice anterior C, foi inferido durante o desenvolvimento do presente trabalho, que a melhor forma para tentar entender de maneira mais efetiva o funcionamento da MDFM, seria através de um exemplo de aplicação prática. Na figura a seguir D.1 será apresentada a representação gráfica de um programa que tem o objetivo de realizar o cálculo numérico da integração sobre a curva $y = x^2$, entre os limites $x = 0.0$ e $x = 1.0$, através do método do trapézio, com intervalos de 0.02.

Essa é a representação mais comum para programas escritos para *dataflow*. Como mencionado anteriormente, esse paradigma busca a representação de programas computacionais no formato de um grafo. Dessa forma, instruções que devem ser executadas paralelamente são dispostas uma ao lado da outra e instruções que devem ser executadas sequencialmente, são dispostas uma abaixo da outra. Sendo assim, a figura D.1 possui o formato de um grafo bidimensional. Os nós representados nesse grafo, caracterizam as instruções que devem ser executadas. As arestas desse grafo, representam os caminhos por onde os dados devem seguir, entre as instruções.

Uma característica importante que deve ser reforçada é o caráter dinâmico dessa arquitetura. O cálculo da integração desse programa de exemplo, necessita de diferentes níveis de iteração, onde cada nível representa um intervalo que divide a curva a ser calculada, a partir do

Figura D.1 – Exemplo de um programa *Dataflow* para a computação do cálculo numérico para a integração da curva sobre a função $y = x^2$.

Fonte: GURD (17)

método proposto. Dessa forma, é interessante observar que a instrução determinada como ADL no programa da figura D.1, tem o objetivo de incrementar o identificador que representa cada um desses níveis de iteração. Esse tipo de instrução deve existir para permitir que qualquer instrução presente no grafo (os nós do grafo), possam ser reutilizadas durante a execução. Caso não houvesse essa característica, poderia ocorrer de fichas destinadas a níveis de iteração distintos, chegarem simultaneamente para a execução, na mesma instrução, fato que poderia gerar um grande equívoco no cálculo proposto.

A seguir será apresentado um conjunto de imagens que tem o objetivo de representar graficamente, uma possível sequência para a execução do programa apresentado na figura D.1. É importante lembrar que a execução de um programa *dataflow* é totalmente assíncrono. O sincronismo necessário para a comunicação entre as instruções é obtido a partir do atraso de cada instrução que depende de outros dados para realizar sua execução. Esse atraso é realizado pela operação de emparelhamento, descrito em detalhes nos capítulos do presente

Figura D.3 – Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.

Fonte: GURD (17)

Note que o valor "verdadeiro" entra nos outros dois BRRs presentes no programa D.4. Uma característica interessante de se ressaltar nesse ponto é que essas instruções particulares são as responsáveis por pararem a execução do programa, caso o resultado da instrução CGR seja "falso". Em seguida, o valor 0.02 é duplicado, para a realização da operação de potenciação, na função de integração, e ele também é somado ao intervalo que compõe o método do trapézio utilizado de 0.02. Esse novo valor será encaminhado para o início da função, para realizar o processo novamente.

Aqui D.5, o programa segue o seu fluxo e se encaminha para o final da função de integração. Os novos valores calculados, tomarão os lugares que outrora foram ocupados pelos valores iniciais do programa. A execução seguirá o mesmo curso descrito, até que o dado de saída da instrução ADL à direita, seja maior que 1. Nesse caso particular, a instrução CGR retornará "falso", de forma que as instruções de BRR presentes no início da função de integração, não gerarão novas fichas. Já a instrução de BRR à esquerda irá encaminhar o resultado de todo o cálculo - vindo da instrução ADL no final do grafo - para a saída da máquina.

Figura D.4 – Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.

Fonte: GURD (17)

Dessa forma, essa é a maneira que a MDFM executava seus programas, a partir de toda a estrutura explicada em detalhes no capítulo 4. E da mesma forma, é a maneira como a PIFLOW realiza a execução, seguindo os mesmos critérios estabelecidos pela arquitetura de origem. É importante ressaltar que o exemplo apresentado nesse apêndice recia o mesmo exemplo utilizado no artigo referenciado anteriormente (17), muito utilizado como referência, para o desenvolvimento do presente projeto.

Figura D.5 – Exemplo de execução de um programa Dataflow para o cálculo numérico de integração.

Fonte: GURD (17)