# FlowPGA: DataFlow de Aplicações em FPGA

Leandro A. J. Marzulo , Fabio H. Flesch, Alexandre S. Nery, Felipe M. G. França, Edil S. T. Fernandes
Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação, COPPE
Rio de Janeiro - Brazil
{lmarzulo, fabioflesch, solon, felipe, edil}@cos.ufrj.br

## Resumo

*A arquitetura* WaveScalar *é a primeira arquitetura* dataflow *a apresentar uma interface de memória que mantém a semântica de acessos requerida pelas linguagens imperativas. Um protótipo da arquitetura, em desenvolvimento, permitiria passar de experimentação por simulação para um cenário mais real, com o processador desenvolvido em FPGA. No entanto, este protótipo não é acessível (financeiramente) para qualquer instituição que também queira produzí-lo. Neste trabalho é apresentada a* FlowPGA*, uma versão reduzida desta arquitetura para ser utilizada com* FPGAs *com pequeno número de células lógicas. Uma* FPGA *com 1,5 milhões de gates foi utilizada para implementação. A corretude da implementação foi avaliada com a execução de um programa de multiplicação entre dois números positivos usando sucessivas somas. Os resultados mostram que a arquitetura* FlowPGA *tem desempenho equivalente ao* WaveScalar*. Ainda, para avaliar a versatilidade do projeto, a* FlowPGA *foi modificada para utilizar um sistema de numeração* RNS*, com esforço de implementação de aproximadamente 20 horas.*

## 1. Introdução

Com a popularização de máquinas multi-processadas (e *multi-cores*), extrair paralelismo de aplicações vem se tornando uma preocupação cada vez mais importante para atingir alto desempenho. O uso do modelo de Von Newmann, com sua natureza sequencial e baseada no fluxo de controle, dificulta esta tarefa, motivando importantes pesquisas sobre modelos alternativos, como TRIPS [5, 8] e Raw [14]. Outra alternativa a esse cenário é o modelo *Dataflow*, onde as *instruções* são executadas assim que tenham disponíveis os *operandos* de entrada necessários, ao invés de seguir a ordem do programa. Como é possível ter diversas *instruções* no *pipeline* provenientes de diversos contextos, os problemas causados por dependências de dados e/ou de controle podem ser evitados [9]. Infelizmente as máquinas realmente *Dataflow*, nunca foram populares, principalmente porque elas não suportavam a semântica de memória requerida pelas linguagens imperativas, exigindo uma nova arquitetura e linguagem de programação para migrar para este modelo.

A arquitetura *WaveScalar* proposta por Swanson é uma releitura dos conceitos *Dataflow*, para endereçar este problema, provendo uma interface de memória que realiza os acessos respeitando a ordem do programa [10, 11, 12]. A idéia chave é que a computação é dividida em ondas (*waves*), tal que cada onda tem a execução guiada pelo fluxo de dados, mas o sequenciamento de ondas garante a tradicional ordem de acesso à memória. Com isto, foi possível executar programas imperativos em uma máquina *Dataflow*, e ainda obter *speedups* significativos. Para aplicações *single-threaded* o *WaveScalar* apresenta desempenho equivalente a um superescalar ou a um *CMP*, com economia de 30% em área de silício. Para aplicações *multi-threaded* o *WaveScalar* apresenta *speedups* de 2 a 11, em relação a *CMPs* [10].

O projeto *RAMP* [1] propõe a criação de uma plataforma de emulação baseada em FPGA (Field-Programable Gate Array) para acelerar a pesquisa em multiprocessadores. A intenção é que os grupos de pesquisa em computação paralela adotem a prototipagem para avaliar suas idéias ao invés de usarem apenas simuladores. O projeto *WaveScalar* é citado no *RAMP* como um dos possíveis grupos a sererm beneficiados pelo uso desta plataforma.

No protótipo do *WaveScalar* [6], em desenvolvimento, estão sendo usadas 16 placas de circuito, cada qual com 4 FPGAs Virtex II Pro (Xilinx®). Em cada FPGA é representado um *Domain*, com oito *Processing Elements*, além de porções do *switch* e *StoreBuffer*. É visível que este tipo de projeto não está acessível a qualquer centro de pesquisa. A principal motivação deste trabalho é criação de uma uma versão simplificada da arquitetura *WaveScalar*, que possa ser mapeada em FPGAs com baixo número de

células reconfiguráveis. O uso da linguagem Handel-C [7] permite descrever o projeto em um nível mais alto, possiblitando que adaptações para adequá-lo a diferentes necessidas sejam realizadas, de forma simples e rápida.

A Seção 2 descreve o conjunto de instruções e a arquitetura do *WaveScalar*. A Seção 3 apresenta a arquitetura *FlowPGA* e discute alguns detalhes de implementação, bem como as ferramentas desenvolvidas para facilitar a criação de programas para a *FlowPGA*. A Seção 4 apresenta os experimentos realizados para verificar a corretude do projeto, avaliar a funcionalidade do conjunto de ferramentas e também a facilidade de alterar o projeto para adaptá-lo a outras necessidades (neste caso o uso do sistema RNS (Residue Number System) de numeração, aplicado frequentemente em processamento de sinais). A Seção 5 apresenta as conclusões e trabalhos futuros.

## 2. WaveScalar

O *WaveScalar* [10, 11, 12] foi a arquitetura *dataflow* usada como base na implementação da *FlowPGA*, apresentada neste trabalho. Esta Seção provê uma breve descrição do conjunto de instruções e arquitetura deste processador.

### 2.1. O Conjunto de Instruções

Um grafo de fluxo de dados é utilizado para descrever um programa no modelo *DataFlow*. Os nós no grafo são as *instruções*, que são inteligentes, no sentido de que estão associadas a uma unidade funcional. As arestas representam *operandos* trocados entre instruções. O conjunto de instruções do *WaveScalar* é derivado do conjunto de instruções da máquina *Alpha* [3]. A principal diferença é que os desvios devem ser transformados em um mecanismo que seleciona os consumidores dos valores (operandos): *(i)* a instrução *Select* ($\phi$) recebe dois valores $v_1$ e $v_2$ e um booleano $s$, que seleciona um dos valores; a instrução *(ii) Steer* ($\rho$) recebe um valor $v$ e um booleano $b$ que define um de dois possíveis destinos para o envio de $v$.

#### 2.1.1 Ondas

As ondas (*waves*) são fragmentos acíclicos e conexos do grafo de fluxo de controle com uma única entrada, estendendo os hiper-blocos, pois também podem possuir junções. As diferentes iterações em um laço podem executar em paralelo no modelo *dataflow*, caso não haja dependências entre suas instruções. Dada uma instrução em um laço e um operando enviado para a mesma, a instância da instrução de destino é indicada por um rótulo no operando, que marca o número da onda (ou iteração). A instrução `Wave-Advance` (`WA`) incrementa este número para cada operando de entrada de uma onda. A Figura 1 mostra o grafo *dataflow* (a) associado a um trecho de programa (b). As ondas estão envolvidas por linhas pontilhadas.

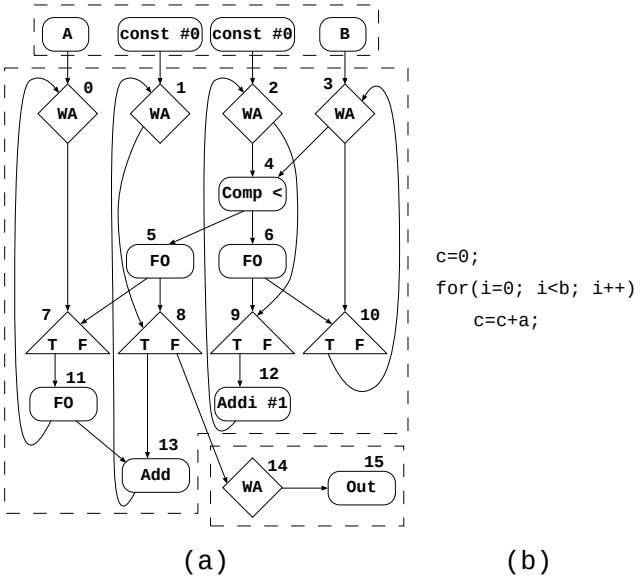


**Figura 1. Grafo dataflow do WaveScalar.**

#### 2.1.2 Wave-ordering annotations

Durante o processo de compilação todas as operações de memória recebem uma chave $< P, C, S >$ (Predecessor, Corrente e Sucessor), permitindo que o subsistema de memória estabeleça uma cadeia conectando-as em uma onda. Toda requisição de acesso à memória é enviada para a interface de memória, inserida em um *buffer* e só poderá ser atendida se todas as requisições anteriores na cadeia e todas as ondas anteriores já tiverem sido executadas.

### 2.2. A Arquitetura WaveScalar

A arquitetura *WaveScalar* é composta por um conjunto de elementos de processamento (*PEs*) idênticos, o *hardware* da *Wave-ordered memory* e uma rede hierárquica para suportar a comunicação. O bloco de construção da *WaveCache*, é o *Cluster*, que possui uma *cache* L1, o *StoreBuffer* que faz a interface com a *Wave-ordered memory*, e um *Switch* que prove comunicação inter e intra-*Cluster*. Cada *Cluster* tem quatro *Domains*, que por sua vez possuem oito elementos de processamento agrupados em *Pods* de dois *PEs* cada. Os *Clusters* são replicados no *die*, formando uma matriz que é conectada à *cache* L2 em suas bordas. A Figura 2 (reproduzida com

permissão do autor, de [10]) mostra uma visão geral da *WaveCache*.

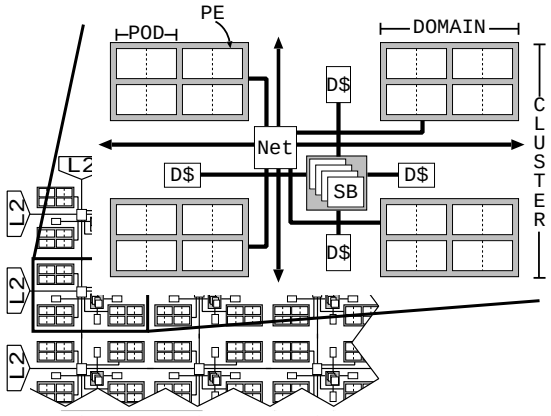


**Figura 2. A arquitetura WaveCache**

Os *PEs* implementam a regra de disparo *Dataflow* e a execução de instruções. Cada *PE* tem a sua *ALU*, estruturas de memória para armazenar operandos, lógica de controle de execução e comunicação e um *buffer* de instruções. Quando um programa é executado no *WaveScalar*, múltiplas instruções são mapeadas em um mesmo *PE*, segundo um algoritmo de *placement*. Conforme a evolução da execução do programa, algumas instruções tornam-se desnecessárias e são substituídas por outras. A regra de disparo garante que uma instrução é executada quando todos os seus operandos de entrada estão disponíveis. Cada *PE* possui também uma *Matching Table* que armazena operandos destinados à instruções mapeadas naquele *PE*, até que as mesmas estejam prontas para serem disparadas. Quando isto ocorre, tais operandos são consumidos, produzindo resultados que serão enviados para outras instruções (em *PEs* remotos ou no *PE* local). O *PE* possui um *pipeline* de cinco estágios, com redes de *bypass* que permitem a execução de instruções dependentes no mesmo *PE*. A Figura 3 mostra a arquitetura de um *PE*, destacando os estágios de seu *pipeline*, descritos a seguir:

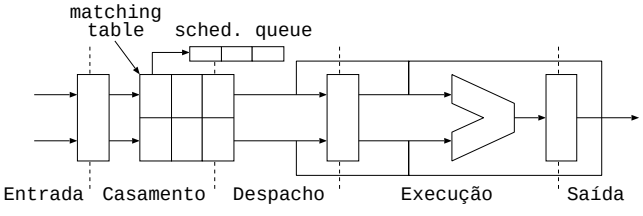


**Figura 3. Um Processing Element**

**Entrada**: Chegada de operandos no *PE*, enviados por outro *PE* ou pelo próprio através da rede de *bypass*.

**Casamento de *tag* dos operandos (*Match*)**: Os operandos são colocados em uma tabela chamada *matching table*, onde suas *tags* são verificadas em busca de um casamento de operandos para uma mesma instrução e onda. Quando uma instrução já possui todos os seus operandos disponíveis é movida para uma fila de instruções prontas para execução, chamada *scheduling queue*. O casamento também pode ocorrer especulativamente, quando o *PE* supõe que alguma instrução que está executando localmente produzirá operandos para uma outra, também local.

**Despacho**: O *PE* seleciona uma instrução da *scheduling queue*, lê seus operandos da *matching table* e os envia para o estágio de execução.

**Execução**: Executa a instrução e envia os resultados para o estágio de saída, exceto quando: *(i)* a mesma foi disparada especulativamente e ainda não possui todos os operandos de que necessita; *(ii)* o *buffer* de saída está cheio. No primeiro caso, a instrução é eliminada da *scheduling queue* e no segundo ocorre um *stall* no estágio de execução até que haja espaço disponível.

**Saída**: Os resultados da instrução são enviados pelo barramento de saída para o próprio *PE* ou outro remoto. É feita a difusão da informação pelo barramento que conecta os *PEs* em um *Domain*, usando um protocolo de transmissão ACK/NACK.

## 3. A FlowPGA

Nesta seção é descrita a arquitetura *FlowPGA* e seu conjunto de instruções, os detalhes da implementação em FPGA, a linguagem de descrição de grafos dataflow, bem como o conjunto de ferramentas (tradutor e visualizador de mensagens).

### 3.1. Conjunto de Instruções e Arquitetura

A arquitetura *FlowPGA* é baseada, com algumas simplificações, na arquitetura *WaveScalar*, permitindo o mapeamento em FPGAs com baixo número de células reconfiguráveis e viabilizando o estudo de arquiteturas *dataflow*, em um ambiente real de execução. Nesta versão ainda não está contemplada a interface de acesso à memória (*Wave-ordered memory*) e um mecanismo de *placement* dinâmico que permita carregar programas com mais instruções estáticas do que o suportado pelas listas de instruções dos *PEs*. Esse mecanismo faria o *swap* de instruções da lista, conforme a evolução do programa em execução. O *swapping* de operandos da *Matching Table* e


**Figura 4. Formato das instruções**

**Tabela 1. Significado dos campos de posição**

| | |
|---|---|
| 0 | Campo destino correspondente não é utilizado |
| 1 | Operando direito |
| 2 | Operando esquerdo |
| 3 | Único operando |

listas de espera também não é contemplado nesta versão. Sendo assim, o sistema não está preparado para execução de programas com alto grau de paralelismo, que esgotariam o espaço de armazenamento de operandos.

As instruções da *FlowPGA* possuem um tamanho fixo e regular, para facilitar a decodificação. Cada instrução contém um número identificador (ID), o *opcode* que indica a operação a ser realizada na ALU, um imediato, além dos IDs das instruções de destino. Nesta versão do *FlowPGA* cada instrução pode receber 1 ou 2 operandos e enviar resultados para até 2 instruções. Caso seja necessário enviar o resultado para mais de dois destinos, é usada a instrução `FANOUT` que recebe um operando de entrada e encaminha para dois destinos.

Dependendo da instrução, a posição dos operandos de entrada é relevante para a produção do resultado (em uma divisão, por exemplo, é necessário saber quem são o divisor e o dividendo). Sendo assim, para cada um dos 2 destinos de uma instrução é informada também a posição. A Figura 4 exibe o formato de uma instrução da *FlowPGA*, com todos os seus campos, e as Tabelas 1 e 2 relacionam os valores para os campos posição e opcode, com seus respectivos significados.

**Tabela 2. Significado do campo opcode**

| | | | |
|---|---|---|---|
| 0 | Soma | 13 | FanOut |
| 1 | Subtração | 14 | Saída de Resultado (USB) |
| 2 | Multiplicação | 16 | Soma com imediato |
| 3 | Divisão | 17 | Subtração com imediato |
| 4 | Mod | 18 | Multiplicação com imediato |
| 5 | E lógico | 19 | Divisão com imediato |
| 6 | Ou lógico | 20 | Resto com imediato |
| 7 | Negação lógica | 21 | E lógico com imediato |
| 8 | Comparação $>$ | 22 | Ou lógico com imediato |
| 9 | Comparação $<$ | 24 | Comparação $>$ Imediato |
| 10 | Comparação $=$ | 25 | Comparação $<$ Imediato |
| 11 | Steer | 26 | Comparação $=$ Imediato |
| 12 | Wave-Avance | | |

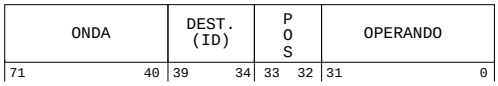


**Figura 5. Mensagens entre instruções**

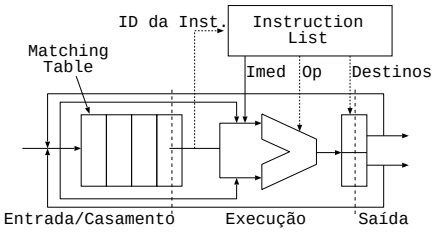


**Figura 6. Um PE na FlowPGA**

A execução de uma instrução gera operandos de saída que podem ser enviados para outras instruções, mapeadas em qualquer *PE*. Os operandos são enviados em forma de mensagens contendo o número da onda, ID da instrução de destino, posição do operando na instrução destino e valor do operando. O formato da mensagem é exibido na Figura 5.

Cada *Processing Element* da *FlowPGA* foi implementado como um *pipeline* assíncrono de 3 estágios. A Figura 6 exibe o *PE*, cujos estágios são descritos a seguir:

**Entrada e Casamento**: Chegada de no máximo 1 operando por ciclo no *PE*, enviados por outro *PE* ou pelo próprio através da rede de *bypass*. A mensagem de entrada é: *(i)* enviada para execução, se o campo *Posição* da mensagem possui o valor 3; *(ii)* enviada para execução juntamente com outro operando da *Matching Table*, se ocorrer um casamento; *(iii)* enviada para uma fila de espera, caso a *Matching Table* esteja cheia; *(iv)* inserida na *Matching Table*, caso o campo posição seja 1 ou 2, e não tenha ocorrido casamento.

**Execução**: Acessa a instrução indicada na *Instruction List*, realiza a execução e envia os resultados para o estágio de saída.

**Saída**: Os resultados da instrução são inseridos em um *buffer de saída* e enviados pelo barramento de saída para o próprio *PE* (pela rede de *bypass*) ou outro remoto (por um *switch Butterfly* [2]). Uma mensagem é produzida para cada destino. O *PE* pode seguir com a execução de instruções, pois cada mensagem só será apagada do *buffer* quando tiver sido recebida pelo switch, que garantirá a sua entrega.

Percebe-se que não há necessidade da *scheduling queue*, pois não há execução especulativa e, além disso, como só

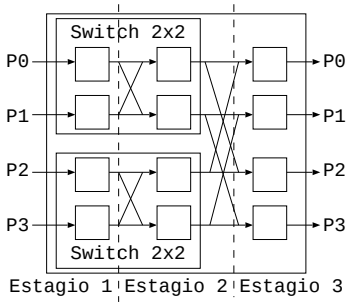


**Figura 7. Switch Butterfly 4x4**

uma mensagem é recebida por ciclo, todo o casamento gera uma execução imediata. Pelo mesmo motivo, só é necessário armazenar, no máximo, um operando por instrução na *Matching Table*, ficando a mesma reduzida a linhas contendo apenas o número da onda, número da instrução, operando e posição. Como a *Matching Table* é percorrida em paralelo no estágio de casamento, ela deve ter um número pequeno de linhas para que esta tarefa não seja cara. Daí a existência da lista de espera. Também não é necessário o estágio de Despacho, pois não há *scheduling queue* e a rede de *bypass* envia operandos apenas para o estágio de Entrada/Casamento.

A comunicação entre os *PEs* é feita por intermédio de um *Switch Butterfly*, descrito na Figura 7. Este é implementado com um *pipeline* sincrono de 3 estágios, sendo cada um responsável pela execução de um *hop* até a entrega da mensagem ao *Destino*. É importanete salientar que no *WaveScalar*, a comunicação entre *PEs* de um mesmo *Domain* é feita ponto-a-ponto. Embora isto possibilite um maior paralelismo, na presente arquitetura, optou-se pela alternativa de alterar o *switch* para simplificar o projeto e facilitar a escalabilidade do mesmo (inserir novos *PEs*). O *Switch Butterfly* 4x4 é construído recursivamente a partir de *switches* 2x2. Para ampliá-lo, basta seguir o mesmo princípio. Para saber qual processador deve receber uma determinada mensagem, o *switch* consulta uma tabela de roteamento que guarda uma lista dos *PEs*, associados a todas as instruções mapeadas.

### 3.2. Implementação em FPGA

A arquitetura *FlowPGA*, descrita na Seção 3, foi implementada utilizando a linguagem Handel-C, tendo como alvo a placa RC10, ambas desenvolvidas pela Celoxica®. A RC10 utiliza uma FPGA Xilinx®Spartan-3E, que pode operar com um clock entre 2 e 300MHz. Ela possui 1,5 milhões de células lógicas e disponibiliza uma interface de comunicação via USB, além de uma série de dispositivos de E/S (VGA, teclado, mouse, *display* de sete segmentos, leds, entre outros). A implementação desta arquitetura se deu por meio do kit de desenvolvimento DK5, também licenciado pela Celoxica®, em conjunto com as ferramentas do ISE WebPack da Xilinx®.

A comunicação entre *PEs* e *Switch* é feita através de canais com *FIFOs* para armazenar até sete mensagens em cada. As estruturas *FIFO* atuam como os *reject buffers* do *WaveScalar*. Para a comunicação entre os estágio do pipeline assíncrono de cada *PE* foram utilizados canais sem *FIFO*. Uma interface de recebimento de mensagens externas através da USB foi elaborada para fazer a carga de operandos de inicialização da computação. A instrução `OUT`, descrita na Seção 3.1 gera uma mensagem de saída contendo o operando, que é encaminhada para a interface USB e pode ser coletada pelo PC hospedeiro. A Figura 8 mostra uma visão geral da arquitetura *FlowPGA* (envolvida pela linha pontilhada) e sua comunicação com o PC.

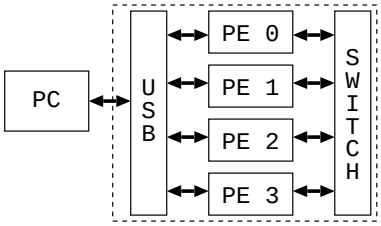


**Figura 8. Visão geral da FlowPGA**

A implementação de *ALUs* com capacidade para executar todos os tipos de instrução descritos na Tabela 2 só seria possível, para a placa RC10, em alternativas de *design* menos eficientes. Neste trabalho, houve uma preocupação maior em criar uma versão da *FlowPGA* com desempenho mais próximo possível da arquitetura *WaveScalar*. Desta forma, o projeto foi otimizado ao máximo, dentro dos limites da placa. As operações de divisão e multiplicação são as que ocupam a maior área no projeto e também as que demandam o ciclo mais longo para execução. Estas operações não foram incluídas nesta implementação, resultando em um projeto que ocupa aproximadamente quinhentos e cinqüenta mil células lógicas e pode executar em freqüências de até 48MHz. Vale lembrar que esta é uma das possíveis alternativas de *design*. A descrição da arquitetura em uma linguagem de alto nível, como o Handel-C, facilita a adequação do projeto para outras necessidades. Na Seção 4.2 é feita a descrição de uma versão do *FlowPGA* para usar o sistema *RNS* [4, 13].

### 3.3. Descrição do grafo dataflow

Para facilitar a escrita de programas para a *FlowPGA*, foi desenvolvida uma linguagem para descrever o grafo

*dataflow* e definir o *placement* das instruções nos *PEs*. Nesta, informa-se a quantidade de *PEs*, as instruções a serem executadas por cada *PE* e os dados de entrada do programa. A Figura 9 mostra o programa *FlowPGA* associado ao grafo dataflow da Figura 1.

```
pe=4;
pe(0){
    wa(2) -> (5);
    addi#1(5) -> (6)<l>, (11)<l>;
    compless(6) -> (7), (8);
    fo(8) -> (11)<r>, (12)<r>;
    steer(11) -> (2);
}
pe(1){
    wa(0) -> (9)<l>,(4)<l>;
    steer(9) -> (0);
}
pe(2){
    wa(1) -> (4)<r>;
    add(4) -> (10)<l>;
    fo(7) -> (9)<r>, (10)<r>;
    steer(10) -> (1), (13);
    wa(13) -> (14);
    out(14);
}
pe(3){
    wa(3) -> (6)<r>,(12)<l>;
    steer(12) -> (3);
}
init{
    a = 3;
    b = 2;
    a->wa(0);
    0->wa(1);
    0->wa(2);
    b->wa(3);
}
```

**Figura 9. Um programa FlowPGA**

Primeiramente, a tabela de roteamento associa cada instrução estática a um *PE*. Um vetor armazena uma lista de números de *PEs* indexada pelo identificador da instrução. Dentro de cada bloco *pe(id)*, uma instrução corresponde a uma linha que obedece ao formato:

$$opcode\#im(id) \rightarrow (id) < side >, (id) < side >;$$

Neste formato, o resultado da operação indicada à esquerda da seta será enviado para as instruções de destino indicadas à direita da mesma. Além disso, este resultado poderá ser inserido no operando direito ou esquerdo destas instruções, conforme indicado pelo campo $< side >$. Caso nada seja informado a respeito da posição de inserção, o tradutor assumirá que a instrução de destino admite apenas um operando. Algumas instruções também permitem operandos imediatos, indicados no campo $\#im$.

O último bloco, denominado *init*, inicializa a computação enviando mensagens para cada um dos *PEs* descritos nos blocos anteriores. As mensagens podem conter constantes de inicialização ou variáveis, cujos valores devem ser atribuídos antes do envio.

### 3.4. Ferramentas

Uma ferramenta de tradução foi criada para a especificação de um programa *FlowPGA*. Ela tem como entrada um grafo dataflow, descrito de acordo com a linguagem especificada na Seção 3.3, e como saída o código binário correspondente. Um visualizador de mensagens também foi criado para separar em campos e exibir, de forma inteligível, as mensagens enviadas pela *FlowPGA* e coletadas pelo PC (via USB).

## 4. Experimentos e resultados

A Seção 4.1 provê uma avaliação da arquitetura *FlowPGA* e sua semelhança comportamental em relação ao *WaveScalar*. Para avaliar a versatilidade da *FlowPGA*, a solução original foi modificada, para que a arquitetura possa adotar um sistema de numeração *RNS*, resultando na *FlowPGA-RNS*. A Seção 4.2 descreve este sistema de numeração e a Seção 4.3 detalha os experimentos realizados para esta alternativa de *design*.

### 4.1. Avaliação da FlowPGA

A linguagem Handel-C permite implementar rapidamente uma solução em FPGA, mas como a descrição é feita em alto nível, o produto final pode não ser o mais otimizado. Uma solução em Handel-C é mais genérica que uma solução em uma linguagem de descrição de hardware (como VHDL ou Verilog). Além disso, no Handel-C, cada atribuição a uma variável leva um ciclo de *clock*. Sendo assim, cada estágio do *pipeline* de um *PE* pode levar mais de um ciclo, embora o trabalho realizado em cada ciclo seja inferior ao de uma implementação descrita em HDL.

Com o objetivo de avaliar o grau de equivalência entre as duas implementações e validar a solução construída, o algoritmo da Figura 1 (que realiza uma multiplicação usando sucessivas somas) foi descrito como um grafo dataflow (de acordo com o modelo exposto na Seção 3). O tradutor, descrito na Seção 3.4, foi usado na elaboração do binário para a arquitetura *FlowPGA*. O programa foi mapeado manualmente nas listas de instruções dos processadores e, em seguida, o projeto foi compilado e gravado na FPGA. O algoritmo foi executado na *FlowPGA* e no simulador *WaveScalar*, variando o multiplicador (valor de $B$) entre cinqüenta e cem mil. O simulador foi configurado com as mesmas características da implementação da *FlowPGA* (1 *Cluster* com 1 *Domain* de 4 *PEs*, com até 16 instruções cada).

A Figura 10 mostra os resultados deste experimento, observando-se que as implementações são equivalentes. Cada estágio do *pipe* assíncrono da *FlowPGA* leva entre 2 e 4 ciclos. Sendo assim, uma distância quase constante,

de aproximadamente quatro vezes, é verificada no número de ciclos da execução entre as arquiteturas. É importante lembrar que, em uma solução descrita em HDL, cada estágio pode ser projetado para durar apenas um ciclo.

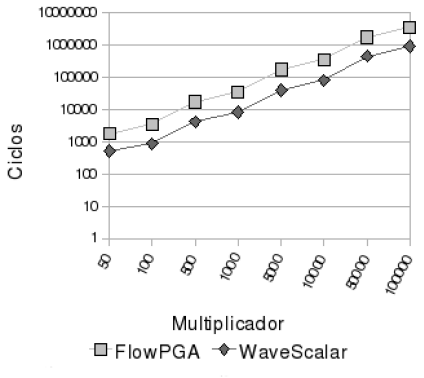


**Figura 10. FlowPGA vs. WaveScalar**

## 4.2. RNS

O RNS (Residue Number System) é um sistema numérico de inteiros cujas operações de adição, subtração e multiplicação são decompostas em sub-operações equivalentes que podem ser executadas em paralelo. Conseqüentemente, o RNS é conveniente para aplicações que utilizam tais operações com alta freqüência, como em processamento digital de sinais. Por outro lado, as demais operações são mais complexas e lentas.

Um sistema baseado em resíduos é construído a partir de um conjunto de *N* números primos relativos entre si, chamados de *módulos*. Ou seja, para todos os pares de módulos $(P_i, P_j)$, tal que $i \neq j$, tem-se que o máximo divisor entre eles é 1.

Seja um conjunto $P = \{P_0, P_1, ..., P_n\}$ uma base RNS, conforme a definição acima. A faixa dinâmica M, isto é, o maior número inteiro que pode ser representado em RNS, é determinada pelo produto dos *N* módulos que compõem a base. Logo, para o conjunto P, tem-se:

$$M = P_0 \cdot P_1 \cdot ... \cdot P_n$$

Então, dois números decimais A e B positivos terão como representação em RNS as *N*-tuplas $A_{RNS} = (A_0, A_1, ..., A_n)$ e $B_{RNS} = (B_0, B_1, ..., B_n)$, dadas por $A_i = |A|_{P_i}$ e $B_i = |B|_{P_i}$. A notação $|A|_{P_i}$ equivale a $A$ mod $P_i$. Para números negativos é necessário fazer uma operação de complemento (mais detalhes podem ser obtidos em [4, 13]).

Operações de adição, subtração e multiplicação entre $A_{RNS}$ e $B_{RNS}$, têm como resultado uma *N*-tupla $R = (R_0, R_1, ..., R_n)$, dada por $R_i = |A_i \; op \; B_i|_{P_i}$, lembrando que o resultado $A_{RNS}$ op $B_{RNS} = R$ deve estar dentro da faixa dinâmica estabelecida por $M$.

Uma vez que os pares de módulos $(P_i, P_j)$ são primos relativos entre si, pode-se usar o Teorema Chinês do Resto [4, 13] a fim de converter um número do sistema de resíduos para o sistema numérico convencional. Para a base $P$, cuja faixa dinâmica corresponde a $M$, deseja-se obter o valor decimal de $R$. Pelo Teorema Chinês do Resto:

$$R = \left| \sum_{i=0}^{3} \left( |P'_i \cdot R_i|_{P_i} \cdot \left( \frac{M}{P_i} \right) \right) \right|_M$$

Os coeficientes $P'_i$ são constantes que podem ser computadas uma única vez e armazenadas. Tais constantes são calculadas encontrando-se o valor de $x_i$ para o qual:

$$\left| \left( \frac{M}{P_i} \right) \cdot x_i \right|_{P_i} = 1$$

## 4.3. Avaliação da FlowPGA-RNS

A corretude da *FlowPGA-RNS* foi avaliada também com o programa da Figura 1. Os multiplicadores foram variados entre dois e mil, e a execução foi realizada para ambas arquiteturas (*FlowPGA* e *FlowPGA-RNS*). O conjunto de co-primos utilizado foi $P = \{32, 29, 27, 23\}$, resultando em uma faixa dinâmica $M = 576288$. Como o maior valor para um componente RNS deste sistema é 31, a solução foi preparada para trabalhar com operandos de 5 bits. As *ALUs* também foram alteradas para realizar apenas operações de soma, subtração e multiplicação (com e sem imediato), no formato *RNS*. Foram necessárias aproximadamente vinte horas para modificar a solução original e gerar a *FlowPGA-RNS*, o que confirma a versatilidade da arquitetura e da linguagem Handel-C.

A Figura 11 mostra a comparação de desempenho entre estas alternativas de *design*. Observa-se que para multiplicadores pequenos a *FlowPGA* sobrepuja a *FlowPGA-RNS*. A medida que os multiplicadores aumentam, os números de ciclos de execução se aproximam e se cruzam para valores de $B$ (multiplicador, no algoritmo) entre quinze e cinqüenta. Como qualquer número em *RNS* tem valor máximo igual a trinta e um, para suas componentes, este será o maior número de iterações executadas, no cálculo da multiplicação (por sucessivas somas). Desta forma, o número de ciclos para o *FlowPGA-RNS* se mantém na mesma faixa, enquanto que para a *FlowPGA* ocorre um aumento proporcional ao tamanho de $B$. Obviamente não esta sendo considerado o tempo de conversão decimal-RNS, pois esta tarefa é realizada antes de enviar os operandos de entrada para a FPGA.

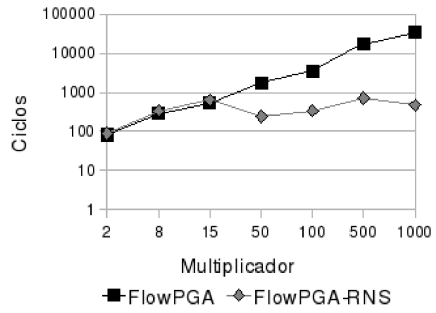


**Figura 11. FlowPGA vs. FlowPGA-RNS**

Um outro dado a ser observado é a redução de 9% dos recursos utilizados na *FPGA*, diante da adoção do *RNS*. Isto ocorre pois a *FlowPGA-RNS* utiliza um menor número de bits para seus operandos, simplificando também as unidades funcionais. Foi possível atingir tal redução, mesmo com a modificação destas unidades para realizar uma operação de resto da divisão, associada a cada operação aritmética já existente.

## 5. Conclusões e Trabalhos Futuros

Neste Trabalho foi apresentada a *FlowPGA*, uma arquitetura *dataflow* em *FPGA*, para o estudo de aplicações e outras alternativas de *design* para este modelo. Em experimentos realizados, foi verificada a proximidade com a arquitetura *WaveScalar* (usada como base). Tais experimentos mostram que a *FlowPGA* traduz corretamente o comportamento do *WaveScalar*.

A implementação em Handel-C facilita a alteração do projeto para outros propósitos. Como prova de conceito, foi contruída uma segunda versão da arquitetura, que utiliza *RNS* como sistema de numeração. A implementação da *FlowPGA-RNS* levou aproximadamente vinte horas, comprovando a versatilidade da implementação original.

Alguns trabalhos futuros decorrem deste trabalho, entre eles: *(i)* a criação de um conjunto de aplicações para avaliar a implementação da versão com *RNS*, mais detalhadamente; *(ii)* a inclusão de um mecanismo que permita mapear aplicações dinâmicamente, sem a necessidade de recompilar o projeto; *(iii)* a inclusão de uma interface de acesso à memória; e *(iv)* a experimentação de uma implementação em uma HDL, permitindo o uso da *FlowPGA* com outras *FPGAs*.

## Referências

[1] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. Ramp: Research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. Technical Report UCB/CSD-05-1412, EECS Department, University of California, Berkeley, Sep 2005.

[2] B. A. Computers. Quarterly technical report no. 3 april l & july 15. 1984 development of a butterfly multiprocessor test bed: The butterfly switch. Technical report, BBN Advanced Computers, Cambridge, Massachusetts, 1985.

[3] R. Desikan, D. C. Burger, S. W. Keckler, and T. Austin. Sim-alpha: A validated, execution-driven alpha 21264 simulator. Technical Report TR-01-23, UT-Austin Computer Sciences, 2001.

[4] I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[5] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A design space evaluation of grid processor architectures. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 40–51, 1-5 Dec. 2001.

[6] A. Putnam, S. Swanson, K. Michelson, M. Mercaldi, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers. The microarchitecture of apipelined wavescalar processor: An rtl-based study. Technical Report TR-2005-11-02, Computer Science and Engineering, University of Washington, Nov 2005.

[7] RG. *Handel-C Language Reference Manual*. Celoxica Limited, http://babbage.cs.qc.edu/courses/cs345/Manuals/HandelC.pdf, 2005.

[8] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[9] J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. Technical report, 1997.

[10] S. Swanson. *The WaveScalar Architecture*. PhD thesis, University of Washington, 2006.

[11] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 291–302, 2003.

[12] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers. Area-performance trade-offs in tiled dataflow architectures. In *Computer Architecture, 2006. 33rd International Symposium on*, pages 314–326, 17-21 June 2006.

[13] N. S. Szabó and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill series in information processing and computers. 1967.

[14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sept. 1997.