# Compilação para o processador de núcleo virtual Nios II

Antonio Carlos F. da Silva[1], Renata Spolon Lobato[2] e Roberta Spolon Ulson[3]

[1]UTFPR - Universidade Tecnológica Federal do Paraná
Coordenação de informática. Cornélio Procópio, PR, Brasil
antonio@utfpr.edu.br

[2]UNESP - Universidade Estadual Paulista
Instituto de Biociências, Letras e Ciências Exatas
Dep. de Ciências de Computação e Estatística. São José do Rio Preto, SP, Brasil
renata@ibilce.unesp.br

[3]UNESP - Universidade Estadual Paulista
Faculdade de Ciências. Departamento de Computação. Bauru, SP, Brasil.
roberta@fc.unesp.br

**Resumo** Arquiteturas reconfiguráveis são a base para o paradigma de computação reconfigurável, o que nos permite ganho de desempenho e maior flexibilidade para o *design* de circuitos. Diante deste cenário, tornam-se necessárias ferramentas para o desenvolvimento ou migração de *softwares* para este novo modelo. O presente artigo demonstra os passos necessários para o desenvolvimento de módulos a serem acoplados ao *framework* Phoenix, visando o desenvolvimento de um compilador para linguagem C, que tem com alvo uma arquitetura reconfigurável formada por FPGAs Altera com suporte ao processador virtual Nios II.

**Palavras chave:** Compilação, Computação Reconfigurável, FPGA, Nios II.

## 1 Introdução

A computação reconfigurável tem como objetivo preencher a lacuna entre os modelos de computação tradicionais, que segundo Compton [8] dividem-se em execução através de ASICs (*Application Specific Integrated Circuit*) ou através de processadores de uso geral, buscando obter o desempenho de um ASIC e a flexibilidade de um processador de uso geral. Para isso ela utiliza dispositivos lógicos conhecidos como FPGAs (*Field-programmable Gate Array*).

Arquiteturas reconfiguráveis têm como grande característica a possibilidade de modificação do *hardware* durante o ciclo de vida do dispositivo. Segundo Cardoso [6], existem áreas de aplicação em que a utilização destes sistemas fornece implementações com desempenhos inalcançáveis quando comparados com sistemas computacionais tradicionais, visto que estas arquiteturas têm como principal objetivo executar o processamento mais intenso em *hardware*, acelerando assim sua execução.

Para tornar este cenário mais atrativo para programadores em geral, várias ferramentas que permitem o desenvolvimento de sistemas de *hardware* vêm sendo desenvolvidas. Estas ferramentas têm o intuito de permitir que um programador, usando linguagem de alto nível, gere um circuito tão eficiente quanto o gerado de forma manual por um especialista da área [9], além de aumentar a possibilidade de reutilização de código, sendo uma forma rápida e fácil de se criar sistemas reconfiguráveis.

Além destas ferramentas, também estão sendo desenvolvidas ferramentas para o auxílio à migração ou desenvolvimento de aplicação para arquiteturas híbridas, FPGA/UCP (Unidade Central de Processamento), e para arquiteturas baseadas somente em FPGAs. Algumas destas ferramentas, como o Trident [15] e Molen [14], são apresentadas na seção 3.

Este artigo apresenta o desenvolvimento dos módulos de otimização de código, *hardware* para manipulação de números de ponto flutuante e um novo gerador de código, em substituição ao existente no *framework* Phoenix, para um compilador que tem como objetivo gerar código para o processador Nios II. O texto está estruturado da seguinte maneira: na seção 2 são apresentados o processo de compilação para arquiteturas reconfiguráveis e as diferenças em relação a compilação para as arquiteturas tradicionais; na seção 3 são discutidos trabalhos relacionados com a compilação para arquitetura reconfigurável; na seção 4 é apresentado o *framework* utilizado como *front-end*; na seção 5 são discutidos o compilador desenvolvido e os passos seguidos no processo de compilação; na seção 6 é apresentada a conclusão deste trabalho.

## 2 Compilação para arquiteturas reconfiguráveis

Na compilação para arquiteturas reconfiguráveis o processo de compilação tem preocupações um pouco diferentes do processo tradicional, principalmente com o que diz respeito ao particionamento da aplicação e à definição do que será desenvolvido em *software* e o que será mapeado em *hardware* [14].

Mesmo no caso de compilação para uma arquitetura já definida e na qual o aplicativo não irá gerar circuito de *hardware*, deve existir uma maior preocupação com a identificação de pontos de paralelismo em nível de instrução que podem ser explorados. Esta identificação pode ser feita através da representação intermediária, que é formada por uma série de estruturas com o objetivo de permitir a exploração de paralelismo. Esta representação pode ser formada por vários grafos que armazenam o fluxo e a dependência de dados, o fluxo e a dependência de controle, informações de ordem de execução entre os blocos básicos e as estruturas de controle. Este grafos são descritos a seguir.

– Grafo de fluxo de controle (GFC): É um grafo orientado onde cada nó representa um segmento do código e cada arco representa um caminho possível para a seqüência de execução. Uma seqüência é um conjunto de instruções que devem ser executadas em ordem, a última instrução de uma seqüência é uma instrução de salto e a primeira é o destino de determinado salto.

Este grafo constitui a base para a geração dos grafos de dependências para exploração do paralelismo máximo do programa [6].
- Grafo de dependência de controle (GDC): É um grafo dirigido que engloba o conjunto de nós do grafo de fluxo de controle, onde as ligações entre os blocos representam a dependência de controle e restringem a ordem de execução dos blocos, de forma a preservar a funcionalidade do programa.
- Grafo de fluxo de dados (GFD): Este grafo demonstra o fluxo de dados entre as operações do programa, seus nós representam as operações e a ligação entre dois nós representa a existência de dependência de fluxo entre eles e todas as condições segundo as quais determinada operação é realizada, ou seja as construções condicionais.
- Grafo de dependência de dados (GDD): Para sua geração cada nó do grafo de fluxo de controle é correspondente a um nó no grafo de dependência de dados, as ligações entre os nós representam dependência de ordem de execução em termos de dependência de dados. Através deste grafo é possível explorar o paralelismo em nível de blocos básicos do programa. Sempre que dois blocos forem independentes em termos de fluxo de dados, os mesmos podem ser executados em paralelo.
- Grafo de dependência de fusão (GDF): Proposto por Cardoso [6], é um grafo direcional cujos nós são os blocos básicos gerados pelo GFC, onde os nós fontes indicam os controladores e os nós destinos indicam os nós que necessitam ser controlados. Para cada nó do GDD que represente uma dependência de dados rotulada com a mesma variável local, é inserido um ponto de seleção, que irá indicar qual definição de determinada variável atingirá o próximo ponto.
- Grafo hierárquico de tarefas (GHT): Proposto por Girkar [10], é um grafo construído através do GDD e GDF, que lida com o paralelismo funcional de forma eficiente pois permite a representação explícita de todos os fluxos de controle e dependências de dados, fluxos de controle múltiplos e os ciclos existentes no programa fonte. Este grafo é formado por três tipos de nós: nós simples que representam instruções ou blocos básicos do programa, nós compostos que representam blocos de decisão *if-then-else* e nós ciclos que representam blocos que podem ser executados várias vezes consecutivas dentro do GHT.
- Grafo de dominância: Este grafo auxilia na construção do GDC e indica os caminhos possíveis para se alcançar determinado nó.
- Grafo de pós-dominância: Grafo semelhante ao de dominância, mas demonstra a dominância ao se percorrer o grafo partindo do nó final em direção ao inicial.

## 3 Trabalhos relacionados

A computação reconfigurável mostra-se como uma alternativa para a crescente demanda de processamento em várias áreas da computação e também da robótica [8], pois permite a exploração, de forma eficiente, do paralelismo existente. Mas,

para possibilitar o uso deste tipo de arquitetura, são necessárias ferramentas para permitir a programação e compilação de aplicativos capazes de gerar código alvo que permita o uso deste paralelismo.

Estes compiladores representam uma grande evolução para a síntese de alto nível de circuitos digitais, por aproximar programadores de linguagens de alto nível desta área tão restrita a projetistas de *hardware*.

O Nenya/Galadriel [6] realiza a compilação de *bytecodes* Java, tendo como objetivo a geração eficiente de *hardware* especializado em VHDL-RTL (*Very High Speed Integrated Circuit Description Language - Register Transfer Level*), e é otimizado para arquiteturas constituídas por um FPGA e uma ou mais memórias RAM (*Random Access Memory*) [7].

O ROCCC [11] [5] é um compilador para geração de VHDL a partir de várias linguagens, como por exemplo C/C++, Java e Fortran. Ele foi construído a partir do SUIF [12] e busca otimizar o fluxo intenso de dados, para isso realiza alterações nos laços e aplica um conjunto de otimizações tradicionais na geração de código, sendo mais eficiente para aplicações com poucos desvios no fluxo de controle.

O Trident [15] é um compilador de código aberto para compilação de algoritmos em linguagem C visando a geração de VHDL. Desenvolvido a partir do LLVM (*Low Level Virtual Machine*) [13]. No Trident o programador deve fazer manualmente a partição entre *hardware* e *software*, além de escrever o código responsável pela troca de mensagens entre as partes.

O Molen [14] é um compilador desenvolvido para uma arquitetura conhecida como máquina de Molen [16], formado por um processador IBM PowerPC 405 e um FPGA Virtex II Pro. Este compilador tem como principal característica gerar código para o FPGA (*hardware*) e para o processador (*software*), de acordo com a característica de cada trecho de código.

O HThreads [4] é um compilador para a linguagem C, que utiliza o GCC (*Gnu C Compiler*) como *front-end*, e gera código VHDL para a execução de *threads* em *hardware* especializado. O *HThreads* utiliza a sintaxe da biblioteca *PThreads*, permitindo o teste do código em linguagem de alto nível em um computador executando uma distribuição Linux.

O Phoenix [9] é um *framework* para síntese de circuitos digitais, e por ser utilizado como base para este trabalho, é discutido na seção 4.

## 4 *Framework Phoenix*

O Phoenix é um *framework* para compilação que tem como objetivo permitir a síntese de circuitos digitais e também o processo de compilação para várias máquinas alvo [9].

Os objetivos deste *framework* são [9]:

- Permitir compilação eficiente em tempo e espaço de código em linguagem C;
- Permitir geração de uma representação intermediária de uso tanto para geração de código para processadores quanto para o processo de síntese de alto nível de circuitos digitais;

– Permitir expansão de recursos.

O processo de compilação ocorre em 4 fases, sendo a primeira responsável pela análise léxica, sintática e semântica, geração da árvore sintática e representação na forma de instruções de três endereços. A segunda fase é responsável pela geração da representação intermediária e a implementação de instruções de desvios como por exemplo *continue*, *break*, *goto* e *return*. A terceira fase do *framework* é a fase destinada à implementação de otimizações do código e a quarta fase é responsável pela geração de código para o alvo específico.

### 4.1 Geração da representação intermediária

Com a geração da representação intermediária deve ser possível a geração de código nativo para qualquer processador ou para o processo de síntese de alto nível de circuitos digitais. Nesta representação os fluxos de controle do programa original são encapsulados juntamente com suas instruções e dependências em uma série de grafos.

Segundo Duarte [9], os grafos que compõem a representação intermediária foram escolhidos através da análise de diversos trabalhos relativos à síntese de alto nível e podem também ser utilizados como algoritmos tradicionais para a geração de código nativo.

Os seguintes grafos são gerados como representação intermediária no Phoenix [9]: grafo de fluxo de controle, grafo de dependência de controle, grafo de fluxo de dados, grafo de dependência de dados e grafo hierárquico de tarefas.

## 5 Compilação para o Nios II

O compilador foi desenvolvido com o intuito de gerar código para o processador virtual Nios II [2], ao contrário da maior parte dos compiladores existentes na literatura atual, que buscam a geração de código VHDL. Para seu desenvolvimento foi utilizado como *front-end* o *framework* Phoenix [9], este *front-end* passou por alterações para a implementação de um processo de otimização, contando com os seguintes algoritmos:

– Eliminação de sub-expressões comuns;
– Propagação de cópias;
– Transposição para constantes;
– Movimentação de código ciclo-invariante e
– Otimizações *Peephole*.

Esta implementação aconteceu com base em [1]. Não foram realizados testes para verificação da eficiência das otimizações, visto que como citado em Aho [1], não se pode garantir que estas otimizações tragam melhorias ao código, por terem grande dependência do código de entrada. Por este motivo, em alguns padrões de código de entrada podemos ter um grande ganho e para outros padrões podemos não ter ganho significativo, ou até nenhum ganho.

Após o processo de otimização do código, o fluxo normal do *framework* é retomado, permitindo assim a geração da representação intermediária, formada pelos grafos citados anteriormente. Para gerar o código para o processador Nios II, é passado para o *back-end* do processador o GHT. Sua tradução consiste na leitura das instruções de três endereços de forma a identificar qual a operação a ser realizada e depois na separação dos operandos necessários. O compilador não permite o uso de alguns recursos da linguagem C, como por exemplo o uso de ponteiros, chamadas a funções e recursividade.

Durante o processo de compilação faz-se necessária a descrição da arquitetura do processador para definição de registradores a serem utilizados para alocação dos operandos e também forma de alocação de memória. Na especificação da arquitetura desenvolvida estão descritos os registradores do Nios II, conforme a tabela 1.

**Tabela 1.** Conjunto de registradores do Nios II [2].

| **Reg.** | **Nome** | **Função** | **Reg.** | **Nome** | **Função** |
|---|---|---|---|---|---|
| r0 | Zero | `0x0000000` | r16 | | Propósito Geral |
| r1 | at | Assembler temporário | r17 | | Propósito Geral |
| r2 | | Valor de retorno | r18 | | Propósito Geral |
| r3 | | Valor de retorno | r19 | | Propósito Geral |
| r4 | | Argumentos | r20 | | Propósito Geral |
| r5 | | Argumentos | r21 | | Propósito Geral |
| r6 | | Argumentos | r22 | | Propósito Geral |
| r7 | | Argumentos | r23 | | Propósito Geral |
| r8 | | Propósito Geral | r24 | et | Exceção temporária |
| r9 | | Propósito Geral | r25 | bt | *Breakpoint* temporário |
| r10 | | Propósito Geral | r26 | gp | Ponteiro global |
| r11 | | Propósito Geral | r27 | sp | Apontador de pilha |
| r12 | | Propósito Geral | r28 | fp | Ponteiro de quadro[1] |
| r13 | | Propósito Geral | r29 | ea | End. de retorno de exceções |
| r14 | | Propósito Geral | r30 | ba | End. de retorno de *breakpoints* |
| r15 | | Propósito Geral | r31 | ra | End. de retorno |

[1]Este registrador contém a cópia do valor de `sp` e pode ser utilizado como um registrador temporário.

Os testes do código gerado foram realizados com o uso de um simulador para instruções do Nios II, desenvolvido para reconhecer as instruções no formato *little-endian* formadas por 4 *bytes* cada uma.

O *back-end* desenvolvido foi acoplado ao *framework* Phoenix na forma de uma classe da linguagem C, permitindo sua substituição, em caso de necessidade de geração de código para outro processador.

O fluxograma, apresentado na Figura 1, exemplifica o fluxo de funcionamento do compilador.As implementações descritas neste trabalho são demonstradas nas

áreas cinzas do fluxograma. Em paralelo ao compilador foi estudado o suporte ao tipos *float* e *double*, não presentes no processador Nios II. A implementação do suporte a estes tipos é descrita a seguir.

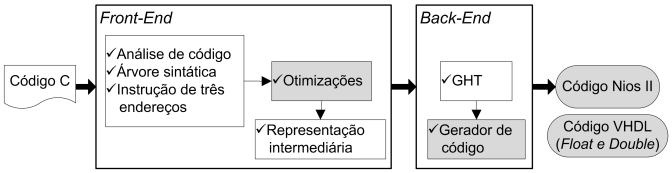


**Figura 1.** Fluxo de compilação adaptado de [9].

### 5.1 Implementação de instruções em *hardware* para os tipos *Float* e *Double*

A implementação destes tipos de dados foi realizada através de instruções me forma de *hardware*. Estas instruções são implementadas na forma de blocos lógicos que funcionam juntamente com a ULA (Unidade de Lógica e Aritmética) no processador, como pode ser visto na Figura 2.

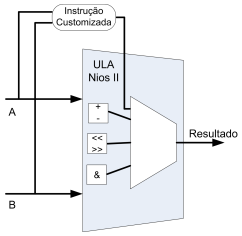


**Figura 2.** Instrução em forma de *hardware* conectada a ULA do processador Nios II [3].

A implementação de instruções em *hardware* para os tipos *float* e *double* em forma de *hardware* foi escolhida devido à sua praticidade e ao ganho de desempenho apresentado em relação à execução em *software*. O desempenho das instruções pode variar de acordo com o FPGA utilizado, pois está ligado ao número de LUTs (*Look-Up Table*) ou elementos lógicos de cada um dos FPGAs, sendo maior em FGPAs com número maior de LUTs ou elementos lógicos [3]. Na Tabela 2 são apresentados os valores de aceleração com o uso de instruções customizadas para dois modelos diferentes de FPGA.

Os testes com o compilador foram realizados utilizando um *kit* de desenvolvimento *DE2 - Altera Development and Education Board*, equipado com o

**Tabela 2.** Fator de aceleração com o uso de instruções em *hardware* para ponto flutuante [3].

| FPGA | Soma | Subtração | Multiplicação | Divisão |
|---|---|---|---|---|
| EP2S60 | 14x | 15x | 12x | 14x |
| EP1S40 | 20x | 20x | 19x | 18x |

FPGA EP2C35. Por se tratar de um *kit* desenvolvimento que conta com vários periféricos acoplados a placa, faz-se necessário configurar o que será utilizado, através de uma configuração VHDL. Pelo fato de o desenvolvimento do circuito VHDL não ser o foco principal deste trabalho, foi utilizada uma configuração padrão, disponibilizada pelo fabricante. Esta configuração padrão foi alterada de modo a dar suporte ao *hardware* responsável pela execução das operações de ponto flutuante, com o auxílio de aplicativos gráficos para compilação e geração de código VHDL (Quartus II e SOCP Builder), disponibilizados pelo fabricante do *kit* de teste.

A configuração do circuito é passada para o *kit* de testes através de qualquer computador que possua uma conexão USB e os *drivers* e aplicativos instalados. Para a visualização dos resultados é utilizado um monitor conectado á saída de vídeo do próprio *kit*, que deve ser configurada via VHDL.

Para efeito de teste foi gerado um programa utilizando o Nios II IDE, *suite* de programação para o Nios II. Neste programa um vetor de 1000 posições foi iniciado com valores gerados de forma aleatória, os quais foram submetidos às 4 operações básicas (adição, subtração, multiplicação e divisão).

Com a execução do teste foi possível a obtenção do tempo gasto na execução em *hardware* e do seu equivalente quando executado em *software*. Estes tempos foram apresentados em segundos e também em ciclos de relógio do processador.

O ganho de desempenho apresentado na implementação em *hardware* dos tipos *float* e *double* no compilador desenvolvido pode ser visto na Tabela 3.

**Tabela 3.** Resultado do teste com a implementação dos tipos *float* e *double* em *hardware*

| FPGA | Soma | Subtração | Multiplicação | Divisão |
|---|---|---|---|---|
| EP2C35 | 22x | 21x | 19x | 20x |

Após geradas as instruções, sua utilização acontece através de uma biblioteca da linguagem C (`system.h`), na qual existe toda a configuração de *hardware* do circuito. Esta biblioteca é gerada pela IDE do Nios II no processo de compilação, e pode ser utilizada em qualquer ambiente de programação da linguagem C/C++, através da diretiva `#include`.

### 5.2 Análise

Como apresentado na seção 3, existem vários compiladores para arquiteturas reconfiguráveis, mas em sua grande maioria são voltados para a geração de código VHDL para a criação de circuitos específicos para uma dada aplicação.

O Molen [14] é o único compilador que tem objetivo semelhante ao compilador descrito neste trabalho, pois gera código para um FPGA. Este compilador diferencia-se no que diz respeito à função do código gerado, no Molen o código gerado em VHDL tem como objetivo acelerar a execução das partes de código com processamento mais intenso e o gerar código para o processador IBM PowerPC para controle de reconfiguração do FPGA e manipulação de trechos de código para o qual o FPGA não é eficiente como por exemplo: *loops* de tamanho variável e controle de saltos, além de estar ligado a uma arquitetura específica. O compilador desenvolvido tem como objetivo permitir o desenvolvimento de aplicações que sejam executadas diretamente no processador Nios II, independente da arquitetura na qual o processador esta inserido.

O uso da linguagem de programação C como linguagem de entrada justifica-se pelo fato de ser essa a linguagem utilizada no *framework* e, também, pelo fato de que C é de uso comum em outras áreas além da computação, como por exemplo nas engenharias.

## 6 Conclusão

Este trabalho apresentou o desenvolvimento de módulos para um compilador que tem como objetivo servir como ferramenta para a geração e migração de aplicativos para um nova arquitetura formada por FPGAs. Estas novas arquiteturas são as bases da computação reconfigurável, que vem se desenvolvendo como uma grande alternativa para a crescente demanda por desempenho.

A contribuição deste trabalho encontra-se no fato deste ser um dos primeiros compiladores a gerar código para o Nios II, e por não se limitar à geração do código, preocupando-se também com o suporte aos tipos de dados de ponto flutuante, implementados em *hardware*.

Como trabalhos futuros, novos recursos da linguagem, como por exemplo invocação de métodos, recursividade e alocação dinâmica, podem ser implementados. Estas implementações poderão facilitar a migração de códigos existentes, para o Nios II, bem como a implementação de novos aplicativos.

**Agradecimentos:** O autor Antonio Carlos F. da Silva agradece à UTFPR e a Fundação Araucária pelo auxílio e as autoras Renata Spolon Lobato e Roberta Spolon Ulson agradecem à Fundunesp pelo auxílio financeiro.

## Referências

1. Aho, Alfred V. and Lam, Monica S. and Sethi , Ravi and Ullman, Jeffrey D., Compilers: Principles, Techniques, and tools, Addison-Wesley, Massachusetts (2007).

2. Altera, Inc., Nios II Processor Reference Handbook (2006).
3. Altera, Inc., Nios II Custom Instruction User Guide (2007).
4. Andrews, David L. and Sass, Ron and Anderson, Erik and Agron, Jason and Peck, Wesley and Stevens, Jim and Baijot, Fabrice and Komp, Ed.. Achieving Programming Model Abstractions for Reconfigurable Computing. In: IEEE Trans. VLSI Syst, Volume 16, Number 1 (2008).
5. Buyukkurt, Betul and Guo, Zhi and Najjar, Walid A.. Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. In: Lecture Notes in Computer Science, volume 3985, pp. 401–412 (2006).
6. Cardoso, João M. P. Compilação de algoritmos em java para sistemas computacionais reconfiguráveis com exploração de paralelismo ao nível das operações. Escola Técnica de Lisboa, Portugal (2000).
7. Cardoso, João M. P. and Neto, Horácio C. Compilation Increasing the Scheduling Scope for Multi-memory-FPGA-Based Custom Computing Machines. In: Lecture Notes in Computer Science, volume 2147 (2001).
8. Compton, Katherine and Scott, Hauck. Reconfigurable Computing: A Survey of Systems and Software. In: CSURV: Computing Surveys, volume 34 (2002).
9. Duarte, Flavio L.. Phoenix: um framework para trabalhos em síntese de alto nível de circuitos digitais. Universidade Federal de Uberlândia (2006).
10. Girkar, M. and Polychronopoulos, C. D.. Automatic extraction of functional parallelism from ordinary programs. In: IEEE Trans. on Parallel & Distributed Systems (1992).
11. Guo, Zhi and Buyukkurt, Betul and Najjar, Walid and Vissers, Kees. Optimized Generation of Data-Path from C Codes for FPGAs. In: Proceedings of the conference on Design, Automation and Test in Europe, pp. 112–117. IEEE Computer Society (2005).
12. Hall, Mary W. and Anderson, Jennifer M. and Amarasinghe, Saman P. and Murphy, Brian R. and Liao, Shih-Wei and Bugnion, Eduoard and Lam, Monica S.. Maximizing Multiprocessor Performance with the SUIF Compiler. In: Digital Technical Journal of Digital Equipment Corporation, volume 10, number 1 (1998).
13. Lattner, Chris and Adve, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization. Palo Alto, California (2004).
14. Panainte, Elena Moscu and Bertels, Koen and Vassiliadis, Stamatis. The Molen compiler for reconfigurable processors. In: ACM Trans. Embedded Comput. Systems, volume 6, number 1. (2007).
15. Tripp, Justin L. and Gokhale, Maya B. and Peterson, Kristopher D.. Trident: From High-Level Language to Hardware Circuitry. In: Computer, volume 40, number 3, pp. 28–37. IEEE Computer Society. Los Alamitos, CA, USA (2007).
16. Vassiliadis, Stamatis and Gaydadjiev, Georgi and Bertels, Koen and Panainte, Elena Moscu . The Molen Programming Paradigm. In: Computer Systems: Architectures, Modeling, and Simulation, Third and Fourth International Workshops, SAMOS 2004, Samos, July 19-21, 2004, Proceedings, volume 3133 (2004).