João Paulo Pizani Flor

Síntese comportamental de componentes de um Sistema Operacional em Hardware

21 de Julho de 2011

João Paulo Pizani Flor

Síntese comportamental de componentes de um Sistema Operacional em Hardware

Apresentado como requisito à obtenção do grau de bacharel em Ciência da Computação

Orientador:
Prof. Antônio Augusto Medeiros Fröhlich

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

21 de Julho de 2011

Sumário

[1] Embedded Parallel Operating System

Lista de abreviaturas e siglas

BLP	Bit-Level Paralelism,	p. 7
ILP	Instruction-Level Paralelism,	p. 7
FPGA	Field-Programmable Gate Array,	p. 7
EDA	Electronic Design Automation,	p. 7
VHSIC	Very High-Speed Integrated Circuit,	p. 7
VHDL	VHSIC Hardware Description Language,	p. 7
EPOS	Embedded Parallel Operating System,	p. 8
HLS	High-Level Synthesis,	p. 8
ASIC	Application-Specific Integrated Circuit,	p. 10
GPP	General Purpose Processor,	p. 10
RDPA	Reconfigurable Data-Path Array,	p. 11
ULA	Unidade Lógico-Aritmética,	p. 11
HDL	Hardware Description Language,	p. 12
TLM	Transaction-Level Modelling,	p. 17
NoC	Network-on-Chip,	p. 17
DSL	Domain-Specific Language,	p. 17
RTL	Register Transfer Level,	p. 17
GPU	Graphics Processing Unit,	p. 20
ELF	Executable and Linking Format,	p. 20

Resumo

Atualmente, com os avanços na tecnologia de lógica programável e a crescente necessidade de explorar o paralelismo inerentes às aplicações, é cada vez maior a utilização de FPGAs[1] para a aceleração de algoritmos. Porém, a descrição de modelos de hardware para a execução de algoritmos ainda é feita em um baixo nível de abstração (se comparada à programação de software convencional). O presente trabalho almeja realizar a descrição de um componente de hardware em nível comportamental, e a síntese de tal descrição para um dispositivo FPGA.

[1] Field-Programmable Gate Arrays

1 Introdução

Alan Turing, em 1936, criou uma sólida base para a Ciência da Computação como conhecemos nos dias atuais com seu modelo de uma máquina de computação [1], definida no célebre artigo *"On computable numbers with an application to the Entscheidungsproblem"* (TURING, 1937). Simultaneamente Alonzo Church dava uma definição bastante diferente do conceito de *função computável*, através do Cálculo Lambda. A noção mais frequente e difundida que temos de algoritmo advém da máquina de Turing. Nesse contexto um algoritmo é uma *sequência* de operações, de fato uma sequência de *estados* da máquina de Turing, a qual ao chegar em seu estado final terá produzido a saída da função.

O modelo de computação definido pela máquina de Turing foi tão frutífero que já os primeiros computadores programáveis (como, por exemplo, o EDVAC(NEUMANN; GODFREY, 1993)) eram uma implementação física, uma cópia quase exata, do funcionamento de uma máquina universal de Turing. Eles representam um algoritmo como um *fluxo* de instruções a ser executado, manipulando dados presentes em uma memória global.

Não só as arquiteturas computacionais foram definidas fortemente pela máquina de Turing. Também a grande maioria das linguagens de programação, desde os primórdios, buscou emular esse paradigma. Com a exceção notável das linguagens funcionais, lógicas e da família declarativa em geral, as mais variadas linguagens de programação (mesmo as que dizem oferecer um "alto nível de abstração") obrigam o programador a codificar seu algoritmo como uma sequência de operações, atribuições, desvios e laços explícitos.

Essa, porém, está longe de ser a única maneira de se descrever um algoritmo. É também controverso afirmar que ela é a *melhor* ou *mais intuitiva* maneira de realizar tal descrição. John Backus, o inventor da linguagem FORTRAN, já a criticava no

[1] do inglês "Computing machine"

seu discurso de aceitação do prêmio Turing, recebido em 1977(BACKUS, 2007). Um outro modelo de computação com o mesmo poder de expressão do que a máquina de Turing são os circuitos booleanos. Nesse modelo, um algoritmo é representado por um grafo onde os nodos são portas lógicas fundamentais (and, or e not) e as arestas representam as conexões entre tais portas.

Este último modelo também é extremamente frutífero, e é uma das "vertentes" mais promissoras atualmente para o estudo da teoria de complexidade algorítmica, inclusive para a resolução da questão P vs. NP. Apesar de já ser sabido há um bom tempo que todo e qualquer algoritmo pode ser implementado por um circuito, questões essencialmente tecnológicas impediram por muito tempo a exploração prática dessa possibilidade.

O problema fundamental no uso de circuitos booleanos como uma maneira direta de se implementar um algoritmo é a falta de *flexibilidade*. Até recentemente, uma vez que um circuito fosse fabricado, ele não poderia mais ser alterado ou reconfigurado para exercer tarefa diferente daquela para qual foi projetado. A forma de se obter uma máquina *universal*, capaz de resolver qualquer problema computável, foi projetando-se um circuito capaz de interpretar instruções codificadas em binário e realizar diferentes operações de acordo com o tipo de instrução. Essa arquitetura, uma implementação da máquina universal de Turing, é a tão conhecida arquitetura de von Neumann(NEUMANN; GODFREY, 1993), e ainda hoje é a base para quase todos os processadores.

Apesar de algumas desvantagens inerentes à arquitetura de von Neumann, a hegemonia por ela conquistada no mercado até os dias atuais é simples de se explicar. Os avanços na tecnologia de fabricação de transistores vêm permitindo que o poder de computação dos processadores aumente sem grandes remodelagens arquiteturais. De fato, o cofundador da Intel®, Gordon E. Moore, previu essa "tendência" da indústria de microprocessadores com um artigo(MOORE et al., 1998) publicado em 1965. Tal tendência foi verificada quase exatamente por décadas, porém vem perdendo validade gradativamente nos dias atuais.

Levando em conta essa perda de momento no aumento das frequências dos processadores, a comunidade profissional e acadêmica volta-se para modelos de programação paralela. Cresce também cada vez mais o interesse pela implementação de algoritmos diretamente em hardware, dado o seu paralelismo inerente e os recentes avanços na tecnologia de lógica programável. Um fator que dificulta a ampla

implementação de algoritmos em hardware é a escassez de técnicas e ferramentas que permitem descrições dos algoritmos em alto nível de abstração, e a síntese automática de tais descrições. Investigar o estado da arte na área de síntese de alto nível, assim como realizar um estudo de caso implementando um algoritmo de uso prático, são as metas deste trabalho.

1.1 Justificativa

Grande parte das desvantagens da arquitetura de von Neumann são causadas, de certa forma, pela "uniformização" na maneira com que os algoritmos são executados. Para que o processador não seja um circuito demasiadamente grande e complexo, há um número limitado de instruções, um conjunto pequeno e fixo de unidades funcionais para realizar todo e qualquer programa possível. Essas restrições, apesar de manter a generalidade da máquina, fazem com que possíveis otimizações inerentes ao problema em questão não possam ser aplicadas.

Um exemplo que ilustra muito bem uma desvantagem de tal "uniformização" é a grande dificuldade atual para aproveitar a capacidade de processamento dos *multicores*. Após notar que a lei de Moore estava chegando ao seu fim e que a frequência dos processadores já não aumentaria num passo tão rápido, a indústria de microprocessadores investiu na inclusão de vários processadores independentes num mesmo chip, na esperança de que os programadores reescrevessem seus algoritmos aproveitando a possível execução paralela de vários fluxos de instruções.

Agora, porém, os incrementos na performance são muito mais difíceis de serem alcançados; há muito mais a ser feito do que esperar o aumento "natural" das frequências de relógio. Avanços significativos têm sido feitos em vários níveis de abstração, desde as técnicas de BLP[2] e ILP[3] até o aparecimento de novas bibliotecas e linguagens de programação voltadas a modelar o paralelismo. Esses avanços têm dois grandes objetivos: tirar das mãos do programador a responsabilidade pela paralelização de seu programa, deixando esse trabalho para o compilador e hardware; e facilitar o trabalho do programador para expressar tal paralelismo.

Em uma vertente paralela, uma boa parte dos fatores que impediam a implementação de algoritmos diretamente em circuitos lógicos vem se revertendo. A tecnologia de

[2] Bit-Level Parallellism
[3] Instruction-Level Parallelism

lógica programável deu um grande salto a partir de 1985 e durante toda a década de 90, com a introdução do FPGA[4]. Tais dispositivos são flexíveis, e permitem a implementação de todo e qualquer circuito booleano. Alguns FPGAs mais recentes permitem até mesmo reconfiguração dinâmica e parcial, ou seja, partes da malha lógica podem ser reconfiguradas enquanto o restante do dispositivo está ativo executando um algoritmo.

Também a indústria de EDA[5] vem trabalhando para aproximar o processo de projetar um circuito integrado do processo de desenvolvimento de software. Esse esforço teve um início na década de 80, quando foi criada a linguagem VHDL[6]; uma linguagem para descrição de hardware com foco arquitetural, mas que possui construções típicas de linguagens de programação (por exemplo, laços e condicionais). O aparecimento das linguagens de descrição de hardware contribuiu para o encurtamento da distância entre o mundo dos projetistas de hardware e o dos programadores.

Caso programadores pudessem descrever algoritmos de uma maneira não muito distante da que o fazem atualmente – ou seja, de maneira *comportamental* – e mesmo assim implementá-los em hardware, as vantagens seriam muitas. Talvez a maior vantagem de todas seria a exploração do paralelismo inerente à aplicação desenvolvida. Bons compiladores poderiam explorar esse paralelismo nos vários níveis (bit, instrução, dados, tarefa), de acordo com a flexibilidade da arquitetura subjacente. Algumas alternativas nesse sentido já existem atualmente, e são implementadas tanto como novas linguagens (e novos compiladores), ou como linguagens embutidas (na forma de bibliotecas e *frameworks*). Alguns exemplos são, por exemplo, a linguagem Single-assignment C(GRELCK; SCHOLZ, 2007) e a biblioteca C++ Accelerator(BOND et al., 2010). Esses avanços são discutidos com mais detalhes no capítulo de trabalhos correlatos.

1.2 Objetivos

Atualmente a indústria de EDA, conjuntamente com pesquisadores da área, vem buscando desenvolvimentos em HLS[7]. Novas linguagens foram criadas com esse propósito, além de metodologias e ferramentas que visam obter um sistema completo,

[4] Field-Programmable Gate Array
[5] Eletronic Design Automation
[6] VHSIC Hardware Description Language
[7] High-Level Synthesis

com todos os seus componentes de software e hardware, a partir de uma modelo algorítmico em alto nível de abstração.

O objetivo deste trabalho se insere nesse cenário como um estudo de caso. Será realizada a modelagem em hardware de um componente de sistema operacional comumente implementado em software, um *escalonador*. O modelo será descrito em uma linguagem de alto nível, e utilizando ferramentas que permitam a síntese direta para hardware, com o mínimo possível de intervenção humana.

O componente sintetizado em um FPGA será integrado aos componentes restantes do EPOS[8], os quais executarão em um soft-processor no mesmo FPGA. A verificação de corretude do modelo se dará através de simulações funcionais e a nível de portas lógicas. O componente será validado pela execução de um aplicativo (*multi-threaded* e com requisitos rígidos de tempo real) sobre o sistema EPOS com escalonador em hardware.

Parâmetros da síntese de hardware – tais como área ocupada e atraso máximo – serão coletados e comparados com outras implementações do mesmo componente; já os parâmetros de execução do algoritmo serão analisados em comparação com execuções onde o escalonador estava implementado em software.

[8]Embedded Parallel Operating System

2 Fundamentos

Neste capítulo são detalhados os conceitos teóricos e tecnologias mais importantes considerados como base para o desenvolvimento deste trabalho.

2.1 Computação Reconfigurável

Até recentemente, havia majoritariamente duas maneiras de se implementar um algoritmo(COMPTON; HAUCK, 2002): A primeira é projetar um circuito lógico digital que realiza a computação necessária; tipicamente um ASIC[1] ou então (mais raro) a integração de componentes eletrônicos discretos em uma placa. Essa abordagem é extremamente inflexível, pois após a fabricação de um circuito para um algoritmo específico ele não poderá ser alterado, e executará o mesmo algoritmo por toda sua vida útil.

A segunda forma de se implementar um algoritmo é executá-lo em um processador. Um processador é capaz – essencialmente – de interpretar uma sequência de instruções (dentre um conjunto pré-definido). Escolhendo-se adequadamente tal conjunto de instruções pode-se fazer com que um processador seja universal, ou seja, capaz de resolver todo e qualquer problema decidível.

Dado esse contexto, pode-se dizer que a *Computação Reconfigurável* é uma forma alternativa de se realizar algoritmos, um *paradigma de implementação*. A figura 2.1 situa a computação reconfigurável com relação aos ASICs e aos GPPs[2]. Considerando-se uma escala de flexibilidade pode-se dizer que os ASICs ocupam o extremo inferior, enquanto os GPPs ocupam o extremo superior. A região intermediária em tal escala é ocupada por dispositivos de computação reconfigurável, sendo que a posição exata de um dispositivo dentro dessa região depende de alguns parâmetros discutidos mais

[1] Application-Specific Integrated Circuit
[2] General Purpose Processors

adiante.

Figura 2.1: Escala de flexibilidade/performance comparando GPPs, ASICs e Computação Reconfigurável

O paradigma de computação reconfigurável consiste, praticamente, na implementação de algoritmos em dispositivos de hardware reconfigurável. Tais dispositivos podem ter suas funções lógicas digitais, assim como malhas internas de comunicação, redefinidas após a fabricação. Alguns dispositivos permitem a reconfiguração até mesmo *durante* a execução de um algoritmo.

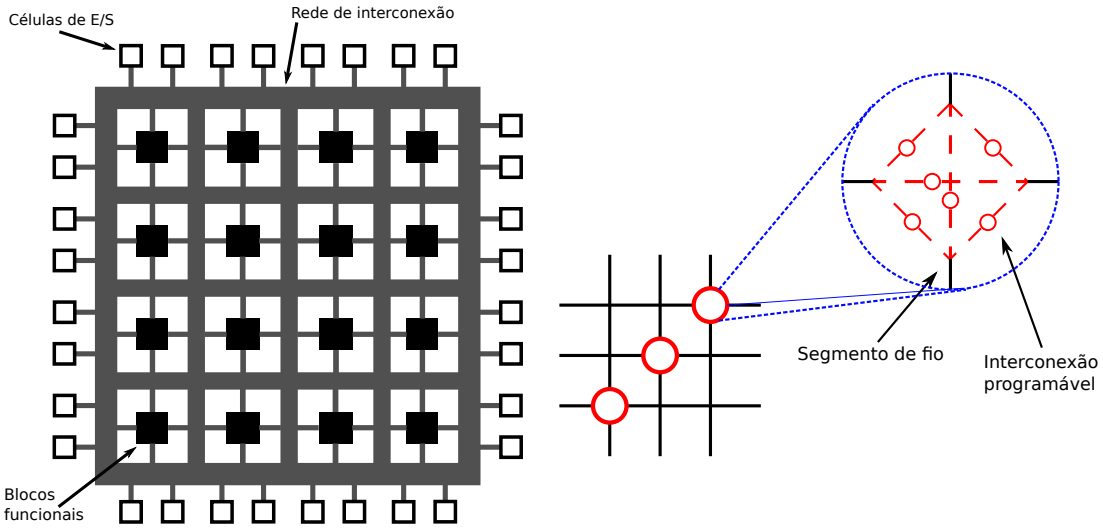
A utilização de dispositivos reconfiguráveis para acelerar a execução de algoritmos encontra-se em crescimento, e pode ser observada em diversas áreas de aplicação com marcantes características de paralelismo. Alguns exemplos citados na frequentemente na literatura(COMPTON; HAUCK, 2002) são:

- Criptografia e quebra de cifras
- Reconhecimento de padrões
- Algoritmos genéticos
- Computação científica

2.1.1 Famílias de dispositivos reconfiguráveis

Existem muitas arquiteturas de dispositivos reconfiguráveis atualmente implementadas, e uma medida crítica para classificar essas arquiteturas é a *granularidade* de configuração. Pode-se dividir os dispositivos, de acordo com essa granularidade, em dois grandes grupos:

FPGA Em um FPGA, a customização dos blocos funcionais e das interconexões entre blocos pode ser feita a nível de bit. Cada bloco funcional de um FPGA tipicamente implementa uma expressão lógica definida de forma tabular (uma memória armazena a tabela-verdade da função). Além disso, cada fio de interconexão entre blocos é um caminho que pode ser aberto ou fechado na programação. A figura 2.2 mostra a arquitetura de um FPGA.



(a) Estrutura global de um FPGA (b) Uma interconexão programável

Figura 2.2: Arquitetura de um FPGA

RDPA[3] Já em um RDPA, a unidade básica de reconfiguração é normalmente uma ULA[4], sendo que alguns dispositivos também trazem unidades funcionais específicas para operações comuns em processamento de sinais. Os caminhos de conexão entre as unidades funcionais são barramentos ao invés de simples fios, e portanto o *quantum* de informação que circula entre os blocos de um RDPA é em geral uma palavra, e não um bit.

2.2 Linguagens de descrição de hardware

Linguagens de descrição de hardware (HDLs[5]) são linguagens que visam modelar o *comportamento* e a *estrutura* de um circuito digital que implementa um algoritmo, e sua utilização iniciou-se em meados da década de 80 com a criação da linguagem Verilog. As HDLs têm bastante em comum com linguagens convencionais de programação de software; sua sintaxe de comandos de controle de fluxo de execução, expressões lógico-aritméticas, etc. são bastante semelhantes. Porém, ao contrário de linguagens de programação de software, as HDLs possuem construções específicas para a modelagem de aspectos temporais.

Quando de sua criação, as HDLs foram utilizadas principalmente para a modela-

[4]Unidade Lógico-Aritmética
[5]Hardware Description Languages

gem e simulação de circuitos; somente com o desenvolvimento tecnológico tornou-se então possível realizar a síntese de ASICs a partir de código-fonte escrito em uma HDL. Um grande passo para o aumento massivo na utilização das HDLs veio com surgimento dos FPGAs a partir do início da década de 90. As cadeias de ferramentas dos fabricantes de FPGAs suportam atualmente código-fonte VHDL e/ou Verilog como a *maneira natural* de se descrever um circuito.

As linguagens de descrição de hardware são bastante versáteis, e permitem modelar um sistema digital em vários níveis de abstração. Em geral, porém, as ferramentas de síntese dos fabricantes de FPGA suportam subconjuntos de VHDL e Verilog. O padrão IEEE1076.6 (IEEE..., 2004) descreve o subconjunto sintetizável da linguagem VHDL, enquanto o padrão IEEE1364.1(IEEE..., 2002) é o correspondente para a linguagem Verilog.

Em geral, quando se realiza a descrição *sintetizável* de um circuito em VHDL ou Verilog, tal descrição é feita a nível de transferência de registradores. O projetista modela seu circuito como um conjunto de registradores e unidades funcionais (funções lógico/aritméticas). As etapas de computação ocorrem quando da transferência de um dado entre registradores, passando através de uma unidade funcional. Esse tipo de descrição está longe da forma como algoritmos são normalmente descritos, e exige um conhecimento que a maioria dos programadores não tem.

A figura 2.3 demonstra essas diferenças entre uma linguagem de descrição de hardware e uma linguagem de programação. Um função *soma* foi escrita em VHDL e em C.

```c
int add(int a, int b) {
  return a + b;
}
```

(a) Função soma em C

```
entity adder is port (
  a, b   : in unsigned(7 downto 0);
  sum    : out unsigned(7 downto 0);
  carry  : out std_logic );
end entity adder;

architecture behavioral of adder is
  signal temp : unsigned(7 downto 0);
begin
  temp  <= ("0" & a) + ("0" & b);
  sum   <= temp(7 downto 0);
  carry <= temp(8);
end architecture behavioral;
```

(b) Função soma em VHDL

Figura 2.3: Comparação entre o código C e VHDL de uma função soma

A descrição em C é consideravelmente mais concisa que em VHDL, e isso é explicado pela necessidade que há de se descrever, no modelo VHDL, detalhes relativos à implementação da função em hardware. Por exemplo, no código C o tipo dos parâmetros para a função *add* é *int*, um tipo cujo tamanho é definido pela arquitetura subjacente. Já em VHDL o projetista precisa realizar uma análise das situações onde o circuito será usado e definir explicitamente tamanhos para os tipos.

Outro ponto a se considerar é o tratamento da situação de *carry*. Em nenhum ponto da descrição em C aparece qualquer referência a esse tratamento, já que ele também é realizado pelo processador de maneira transparente ao programador. Em VHDL, por outro lado, o número de bits presentes no resultado da soma foi também definido explicitamente (para não perder o caso de *carry*), e a função possui de fato 2 sinais de saída, um deles sendo o indicador de carry.

2.3 O sistema operacional EPOS

O sistema operacional EPOS é um sistema operacional voltado para aplicações embarcadas. Ele foi programado em C++ utilizando-se de técnicas como orientação a aspectos e meta-programação estática para alcançar um alto grau de configurabilidade e mesmo assim atender aos requisitos estritos de sistemas embarcados (como performance e tamanho de código).

O EPOS utiliza a metodologia de desenvolvimento de sistemas embarcados orientados à aplicação[6], guiando o desenvolvimento conjunto de componentes de software e hardware adaptáveis aos requisitos particulares de cada aplicação. De fato, pode-se dizer que o EPOS é um repositório de componentes de software e hardware, juntamente com ferramentas capazes de integrar tais componentes e produzir um ambiente de suporte à execução. O sistema assim gerado possui somente os componentes julgados necessários para a aplicação.

Conceitos fundamentais de sistemas operacionais (por exemplo o conceito de *escalonador*, de *thread*, de *semáforo*) são descritos como famílias de abstrações. Técnicas de programação orientada a aspectos são utilizadas para adaptar essas abstrações às peculiaridades do ambiente em que o sistema irá executar (convenção de chamadas, número de registradores, ordem de bytes, entre outras). Mais detalhes sobre o sistema EPOS e a metodologia que lhe serve como base podem ser encon-

[6] do inglês “Application-Driven Embedded System Design

trados em Fröhlich (2001).

Neste trabalho pretendemos descrever um componente de hardware do sistema EPOS em um alto nível de abstração, sintetizá-lo, e integrá-lo ao restante do SO rodando em software (sobre um processador convencional).

3 Trabalhos Correlatos

O esforço de se realizar síntese de hardware a partir de descrições de alto nível ainda é bastante imaturo, e várias metodologias, linguagens e ferramentas foram propostas desde o início dos anos 2000 sem grande consolidação. Inicialmente, ainda em meados dos anos 90, já existiam ferramentas para síntese de circuitos a partir de uma descrição comportamental. Porém tais ferramentas geralmente utilizavam a linguagem VHDL (mais precisamente seu subconjunto comportamental) como entrada, e trabalhavam com um modelo de computação parcialmente temporal. Tanto esse modelo quanto a linguagem se mostraram inadequados para uma descrição de circuitos em alto nível.

Somente a partir dos anos 2000 que ferramentas de síntese de alto nível passaram a ter uma maior aceitação por parte da indústria. Isso pois começaram a ser utilizadas como linguagens de entrada C, C++ e SystemC. Circuitos descritos em alto nível usando C, C++ e SystemC se assemelham muito mais a software. Além disso, essas são linguagens já amplamente conhecidas por programadores ao redor do mundo.

Atualmente também estão em andamento alguns projetos na área de sistemas operacionais para computação reconfigurável. Esses projetos buscam trazer conceitos e modelos de sistemas operacionais para a computação em FPGAs. Nesse âmbito estão inclusos tanto a aplicação de técnicas de sistemas operacionais em computação reconfigurável (escalonamento de tarefas, troca de contexto, comunicação inter-processos), como esforços de integração entre sistemas operacionais convencionais (executando em software) e aplicações implementadas em hardware.

Nesse capítulo fazemos uma revisão de algumas das metodologias e ferramentas para síntese de alto nível com maior relevância atualmente, descrevendo suas particularidades. Tal revisão teve o objetivo de ajudar-nos a fazer uma *escolha informada* quanto à plataforma utilizada na implementação do trabalho. Levando em conta tal objetivo, procuramos obter na revisão uma amostra ampla de soluções, buscando va-

riados fabricantes, linguagens de entrada e paradigmas de programação. Também é feita uma revisão de algumas propostas de integração de componentes de sistemas operacionais rodando em software e hardware.

3.1 Síntese de alto nível

Em uma analogia simplificada entre o projeto de hardware e o desenvolvimento de software podemos dizer que atualmente os projetistas de hardware "programam" seus sistemas em linguagem de montagem. As linguagens e ferramentas para síntese de alto nível têm como objetivo transportar a descrição de um algoritmo em hardware para um nível de abstração equivalente à programação de software em linguagens como C++, Java, etc.

Uma das vertentes mais promissoras para atingir esse objetivo é a utilização do *framework* SystemC(INITIATIVE, 2005). O SystemC é um grupo padronizado de classes C++ que permite a modelagem e simulação de componentes de hardware. Em SystemC é possível descrever o comportamento de componentes de hardware usando a sintaxe C++, e utilizando-se de várias abstrações para a comunicação inter-componentes.

A OSCI[1](THE...,), organização responsável pela manutenção e avanço do framework SystemC, disponibiliza também sob licença livre uma biblioteca para simulação de modelos SystemC. O projetista compila seu projeto descrito em SystemC com a cadeia usual de ferramentas (*g++*, *ld*, *as*), e como resultado da compilação é gerado um arquivo executável. Ao ser executado, ele *simula* o circuito descrito e pode produzir vários tipos de saída para depuração (incluindo arquivos com formas de onda). Não há ainda, porém, nenhum projeto de software livre para a síntese de hardware a partir de SystemC.

3.1.1 Abordagens baseadas em SystemC

Já dentre os vários produtos comerciais para HLS, um dos que utiliza modelos em systemc como entrada é o *Cynthesizer*®, da Forte Design Systems. os modelos são escritos em SystemC de nível TLM[2] e a ferramenta faz a tradução para componentes

[1] Open SystemC Initiative
[2] Transaction-Level Modelling

RTL[3], os quais podem então ser fabricados como ASICs, ou sintetizados em FPGAs (utilizando as ferramentas dos fornecedores).

Muitos dos detalhes necessários à implementação do modelo em hardware são omitidos na descrição TLM. por isso, a ferramenta *Cynthesizer* ®incorpora também um módulo para exploração de espaço de projeto, permitindo que o projetista defina restrições e métricas sobre o circuito e buscando otimizar tais métricas.

Outra cadeia de ferramentas de grande destaque para síntese de alto nível é o *Catapult-C*®, do fabricante mentor graphics. uma peculiaridade do Catapult-C é que ele aceita como linguagens de entrada no projeto de um sistema tanto ansi c++ como systemc. enquanto a entrada em c++ modela basicamente o *comportamento* do sistema sendo projetado, a entrada em systemc permite a especificação de mais detalhes arquiteturais, tais como a configuração de barramentos e NoC[4].

Em um processo de refinamento interativo, o Catapult-C oferece ao projetista comparativos entres as várias opções de parâmetros deixados livres pela descrição em alto nível, e através de estatísticas e gráficos o projetista faz uma decisão informada levando em consideração as prioridades do projeto em questão. o Catapult-C também é capaz de realizar a síntese automática de interfaces entre componentes, e possui um módulo que faz a otimização do consumo de energia nos componentes produzidos, levando em conta as características próprias dos dispositivos de vários fabricantes.

3.1.2 Haskell ForSyDe

Uma abordagem bastante diferente de todas as mencionadas anteriormente chama-se ForSyDe (Formal System Design)(LU; SANDER; JANTSCH, 2002). ForSyDe é uma linguagem de domínio específico (DSL[5]) embutida na linguagem Haskell na forma de biblioteca, sendo capaz de simular circuitos ou gerar código VHDL sintetizável a partir de uma descrição em alto nível. Um fato interessante é que o código VHDL gerado pelo ForSyDe é bem estruturado e legível, o que torna bastante fácil realizar alterações no VHDL, caso o projetista deseje. Na figura 3.1 temos o exemplo de um componente *multiply-accumulate*, que implementa a seguinte função:

$$a \leftarrow a + (b \times c)$$

[3]Register Transfer Level
[4]Network-on-Chip
[5]Domain-Specific Language

O componente multiplica o valor de seus dois sinais de entrada a cada ciclo do relógio, e acumula o resultado da multiplicação em um registrador. A operação *multiply-accumulate* é uma operação básica e importantíssima que compõe uma vasta quantidade de operações em álgebra linear e processamento digital de sinais. Na figura 3.2 está o código VHDL gerado a partir do modelo descrito em Haskell.

```
{-# LANGUAGE TemplateHaskell #-}
module DotProductOnline where

import ForSyDe
import Data.Int (Int16)
type Element = Int16

times = zipWithSY "times"
  $(newProcFun [d|
    f :: Element -> Element -> Element
    f x y = x * y  |])

accum = scanlSY "accum"
  $(newProcFun [d|
    f :: Element -> Element -> Element
    f x y = x + y  |]) 0

dotp = newSysDef (\v w -> accum (times v w)) "dotp" ["v1", "v2"] ["res"]
```

Figura 3.1: Descrição ForSyDe do componente *multiply-accumulate*

Talvez a característica que mais chame a atenção na descrição em ForSyDe é o tamanho reduzido. Enquanto a descrição em ForSyDe do circuito tem cerca de 20 linhas, o código VHDL gerado tem mais de 200 linhas (a figura 3.2 mostra somente o componente raiz da hierarquia VHDL).

Os conceitos fundamentais na descrição de um circuito em ForSyDe são o *processo* e o *sinal*.

Sinal Um sinal é um tipo de dados isomórfico à uma lista, e onde cada elemento da lista é o valor do sinal em um determinado ciclo de relógio (o ForSyDe utiliza um modelo de computação síncrono). Um sinal representa um canal de comunicação entre processos.

Processo É a unidade básica de modelagem em ForSyDe. Um processo é uma função que transforma sinais de entrada em sinais de saída. Existem vários *construtores de processos*, tanto para processos combinacionais quanto sequenciais. Um exemplo de um simples processo combinacional (mapSY) é dado

na figura 3.3. Ele transforma um sinal de entrada em um sinal de saída, aplicando a cada ciclo de relógio uma função sobre o valor atual da entrada, produzindo a saída correspondente.

Através de um processo de combinação recursiva de processos é que se dá a construção de um modelo ForSyDe. Dois ou mais processos podem ser combinados utilizando-se, por exemplo, de *composição*. Assim, os sinais de entrada e saída do processo de mais alto nível nessa árvore definem a própria interface do componente sendo modelado.

3.1.3 Microsoft Accelerator

O Microsoft®Accelerator é uma abordagem para a expressão de algoritmos paralelos em um alto nível de abstração, implementada na forma de uma biblioteca de classes C++. Utilizando-se de classes e funções da biblioteca Accelerator o usuário é capaz de expressar algoritmos no paradigma de *paralelismo aninhado de dados*[6]. Esses algoritmos podem ser executados em variadas plataformas-alvo (GPUs[7], *multi-cores* e FPGAs).

Através dos objetos da classe *ParallelArray* – vetores sobre os quais operações são realizadas em paralelo – e de diversos operadores sobrecarregados entre objetos desse tipo, a bilbioteca Accelerator produz uma árvore de expressão representando o algoritmo. Na figura 3.4 temos um exemplo de código utilizando Accelerator e a respectiva árvore de expressão.

Tal código toma dois vetores de números em ponto flutuante como entrada e realiza a adição desses dois vetores, elemento-a-elemento. Após a execução de todas as operações desejadas pode-se obter o resultado da execução do algoritmo

3.2 Sistemas operacionais para Computação Reconfigurável

Como já exposto no capítulo de conceitos, a computação reconfigurável é um novo paradigma para a execução de algoritmos. A implementação efetiva desse paradigma

[6] do inglês “Nested Data Parallelism”
[7] Graphics Processing Units

ainda está dando seus primeiros passos, e a comodidade de desenvolvimento de sistemas reconfiguráveis é incomparavelmente maior do que a do desenvolvimento de software convencional (para a arquitetura de *von Neumann*).

Uma das razões que retardam a adoção ampla de sistemas de computação reconfigurável é a falta de um sistema operacional para esse paradigma. Um sistema operacional realiza escalonamento de recursos (processador, periféricos, etc.), gerencia entrada e saída de dados – entre outras tarefas – abstraindo assim vários detalhes arquiteturais da máquina subjacente e simplificando a interface de programação.

Um sistema operacional para sistemas reconfiguráveis poderia ter as seguintes responsabilidades(SO; BRODERSEN, 2008), análogas às responsabilidades de um sistema operacional convencional:

- Estabelecer uma noção de *componente de hardware* (análoga à noção de processo)
- Fornecer uma interface unificada para interação entre componentes de hardware e entre componentes de hardware e componentes de software
- Realizar o escalonamento do dispositivo FPGA (tanto temporal quanto espacial) para acomodar os componentes de hardware atualmente em execução
- Controlar a configuração do dispositivo FPGA, realizando a reconfiguração das áreas que serão ocupadas por novos componentes

Em So e Brodersen (2008), os autores descrevem a implementação de um sistema operacional para computadores reconfiguráveis bastante interessante: o sistema BORPH[8]. O BORPH é uma extensão de um sistema Linux padrão, fornecendo suporte de execução a aplicativos em FPGA.

Nessa extensão do Linux, o conceito de processo é estendido também para aplicativos executando em hardware, e tais processos têm (como os processos em software) acesso normal a periféricos e ao sistema de arquivos. Processos em hardware podem comunicar-se entre si, e também com processos em software, através de arquivos *pipe*, o mecanismo tradicional Unix de comunicação inter-processos. A operação de “carga” de um processo em hardware se dá pela configuração de uma área do

[8]Berkeley Operating system for ReProgrammable Hardware

FPGA que irá executá-lo, e o código binário para tal configuração é armazenado em um formato de arquivo inspirado no ELF[9].

[9] Executable and Linking Format

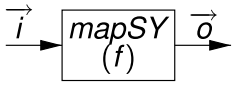
```
entity \dotp\ is
     port (resetn : in std_logic;
           clock : in std_logic;
           \v1sig\ : in int16;
           \v2sig\ : in int16;
           \result\ : out int16);
end entity \dotp\;

architecture synthesizable of \dotp\ is
     signal \times_out\ : int16;
     signal \accum_out1\ : int16;
begin
     \times\ : block
          port (\times_in1\ : in int16;
                \times_in2\ : in int16;
                \times_out\ : out int16);
          port map (\times_in1\ => \v1sig\,
                    \times_in2\ => \v2sig\,
                    \times_out\ => \times_out\);
          function \f\ (\x_0\ : int16;
                        \y_0\ : int16)
                       return int16 is
          begin
               return \x_0\ * \y_0\;
          end;
     begin
          \times_out\ <= \f\(\x_0\ => \v1sig\, \y_0\ => \v2sig\);
     end block \times\;

     \accum\ : entity work.\scanlSY_accum\
                    port map (resetn => resetn,
                              clock => clock,
                              \in1\ => \times_out\,
                              \out1\ => \accum_out1\);

     \result\ <= \accum_out1\;
end architecture synthesizable;
```

Figura 3.2: Código VHDL gerado para o componente *multiply-accumulate*



$$mapSY(f)(\ll v_0, v_1, v_2, \cdots \gg) = \ll f(v_0), f(v_1), f(v_2), \cdots \gg$$

Figura 3.3: O construtor de processo mapSY

Figura 3.4: Produto elemento-a-elemento de dois vetores e a árvore de expressão gerada

Referências Bibliográficas

BACKUS, J. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. In: ACM. *ACM Turing award lectures*. [S.l.], 2007. p. 1977.

BOND, B. et al. Fpga circuit synthesis of accelerator data-parallel programs. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 167–170, 2010.

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

FRÖHLICH, A. Application-oriented operating systems. *Sankt Augustin: GMD-Forschungszentrum Informationstechnik*, Citeseer, v. 1, 2001.

GRELCK, C.; SCHOLZ, S.-B. Sac: off-the-shelf support for data-parallelism on multicores. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. New York, NY, USA: ACM, 2007. (DAMP '07), p. 25–33. ISBN 978-1-59593-690-5. Disponível em: <http://doi.acm.org/10.1145/1248648.1248654>.

IEEE Standard for Verilog Register Transfer Level Synthesis. *IEEE Std 1364.1-2002*, p. 1–100, 2002.

IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)*, p. 1–112, 2004.

INITIATIVE, O. *IEEE 1666: SystemC Language Reference Manual, 2005*. 2005.

LU, Z.; SANDER, I.; JANTSCH, A. A case study of hardware and software synthesis in forsyde. In: *System Synthesis, 2002. 15th International Symposium on*. [S.l.: s.n.], 2002. p. 86 – 91.

MOORE, G. et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, [New York, NY]: Institute of Electrical and Electronics Engineers,[1963-, v. 86, n. 1, p. 82–85, 1998. ISSN 0018-9219.

NEUMANN, J. V.; GODFREY, M. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, Institute of Electrical and Electronics Engineers, Inc, 445 Hoes Ln, Piscataway, NJ, 08854-1331, USA,, v. 15, n. 4, p. 27–75, 1993.

SO, H. K.-H.; BRODERSEN, R. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 7, n. 2, p. 1–28, 2008. ISSN 1539-9087.

THE Open SystemC Initiative. http://www.systemc.org.

TURING, A. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Oxford University Press, v. 2, n. 1, p. 230, 1937.