João Paulo Pizani Flor

Síntese comportamental de componentes de um Sistema Operacional em Hardware

15 de Junho 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. Dr. Antônio Augusto Medeiros Fröhlich

Co-orientador:
Tiago Rogério Mück, B.Sc

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

15 de Junho de 2011

Monografia sob o título “Síntese comportamental de componentes de um Sistema Operacional em Hardware”, defendida em 15 de Junho de 2011 por João Paulo Pizani Flor, como parte dos requisitos à obtenção do grau de Bacharel em Ciência da Computação, e aprovada pela seguinte banca examinadora:

Prof. Dr. Antônio Augusto M. Fröhlich
Orientador

Prof. Dr. José Luís Almada Güntzel

Prof. Dr. Olinto José Varela Furtado

And now for something *completely* different...

Monty Python

Abstract

With the recent advances on semiconductor fabrication technology (in particular, the evolution of programmable logic devices) and the growing need to harness the inherent parallelism in many applications, it is an increasingly popular trend to implement in FPGAs algorithms that were previously implemented in software.

On the other hand, the description of the hardware models realizing these algorithms is still done in a low level of abstraction (when compared to conventional software programming). Also, most attempts at high-level synthesis focus on very specific application areas. This work aims to describe an Operating System component in a high level of abstraction, allowing this component to be synthesized targeting several FPGA devices. The chosen application area (Operating Systems) and the methodology adopted characterize this work as a feasibility study of automatic hardware implementation for general-purpose algorithms.

Resumo

Com os avanços na fabricação de semicondutores (especialmente na tecnologia de lógica programável) e a crescente necessidade de se explorar o paralelismo inerente a várias aplicações, é cada vez mais comum implementar em FPGAs algoritmos antes implementados em software.

Porém, a descrição de modelos de hardware para realizar tais algoritmos ainda é feita em um baixo nível de abstração (se comparada à programação de software convencional). Além disso, a maioria dos trabalhos que visam a síntese de alto nível tem foco em áreas específicas de aplicação. Este trabalho visa descrever em alto nível de abstração um componente de Sistema Operacional a ser implementado em hardware. O componente deve poder ser sintetizado para diversos dispositivos FPGA. A área de aplicação escolhida (Sistemas Operacionais) bem como a metodologia adotada visam fazer deste trabalho um estudo de viabilidade da implementação automática em hardware de algoritmos de propósito geral.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

1 Introdução p. 10

1.1 Justificativa . . . . p. 11

1.2 Objetivos . . . . p. 13

2 Fundamentos p. 15

2.1 Computação Reconfigurável . . . . p. 15

2.1.1 Famílias de dispositivos reconfiguráveis . . . . p. 16

2.2 Linguagens de descrição de hardware . . . . p. 17

2.3 Síntese de alto nível . . . . p. 19

2.3.1 Etapas da síntese de alto nível . . . . p. 20

2.4 O sistema operacional EPOS . . . . p. 22

3 Trabalhos Correlatos p. 25

3.1 Síntese de alto nível . . . . p. 26

3.1.1 Abordagens baseadas em SystemC . . . . p. 27

3.1.2 Síntese de ANSI C++ . . . . p. 27

3.1.3 Haskell ForSyDe . . . . p. 28

3.1.4 Microsoft Accelerator . . . . p. 30

3.2 Sistemas operacionais para Computação Reconfigurável . . . . p. 32
3.2.1 BORPH . . . . p. 32
3.2.2 HybridThreads . . . . p. 33

4 Desenvolvimento p. 35
4.1 O escalonador do EPOS em software . . . . p. 36
4.2 Estruturas de dados utilizadas . . . . p. 37
4.2.1 Lista duplamente encadeada . . . . p. 39
4.2.2 Lista ordenada . . . . p. 40
4.2.3 Fila de escalonamento . . . . p. 40
4.3 Implementação RTL de referência . . . . p. 41
4.4 Nossa proposta de escalonador em hardware . . . . p. 42
4.4.1 O tipo Maybe<T> . . . . p. 43
4.4.2 Gerenciamento de alocação . . . . p. 44
4.4.3 Wrapper da interface raiz . . . . p. 46

5 Ambiente de validação p. 48
5.1 Barramento AMBA AXI4Lite . . . . p. 49
5.2 O processador Plasma MIPS . . . . p. 50
5.2.1 Adaptador mlite_cpu – AXI4Lite . . . . p. 52

6 Resultados p. 54
6.1 Resultados da verificação funcional . . . . p. 54
6.2 Cenários de síntese . . . . p. 55
6.3 Resultados de síntese . . . . p. 57

7 Conclusões p. 59
7.1 Principais contribuições . . . . p. 60

7.2 Trabalhos futuros . . . . p. 61

Referências Bibliográficas p. 62

Apêndice A – Lista duplamente encadeada em hardware p. 64

A.1 Código-fonte C++ – Template . . . . p. 64

A.2 Código-fonte C++ – Wrapper de instanciação . . . . p. 71

A.2.1 Cabeçalho . . . . p. 71

A.2.2 Definições . . . . p. 75

A.3 Testbench VHDL . . . . p. 75

A.4 Diretivas da síntese de alto nível . . . . p. 81

Apêndice B – Escalonador implementado em hardware p. 83

B.1 Gereciador de alocação . . . . p. 83

B.2 Wrapper de chamadas . . . . p. 85

B.3 Diretivas de síntese . . . . p. 85

B.3.1 Cenário 1 . . . . p. 85

B.3.2 Cenário 2 . . . . p. 87

B.3.3 Cenário 3 . . . . p. 88

B.4 Testbench VHDL . . . . p. 90

Apêndice C – Testes do adaptador mlite_cpu – AXI4Lite p. 99

C.1 Programa C de teste de escrita/leitura . . . . p. 99

C.2 Testbench VHDL . . . . p. 100

Lista de Figuras

2.1 Flexibilidade vs. performance de processadores . . . . p. 16

2.2 Arquitetura de um FPGA . . . . p. 17

2.3 Código C e VHDL de uma soma . . . . p. 19

2.4 Fluxo de projeto da síntese de alto nível . . . . p. 20

2.5 Geração de uma imagem de sistema EPOS . . . . p. 23

2.6 Relações entre as entidades do sistema EPOS . . . . p. 24

3.1 Descrição ForSyDe do componente *multiply-accumulate* . . . . p. 29

3.2 VHDL gerado para o componente *multiply-accumulate* . . . . p. 30

3.3 O construtor de processo mapSY . . . . p. 31

3.4 Árvore de expressão de um objeto ParallelArray . . . . p. 31

3.5 Arquitetura geral de um SoC utilizando HThreads . . . . p. 34

4.1 Diagrama de herança do escalonador do EPOS . . . . p. 36

4.2 Hierarquia das estruturas de dados do escalonador . . . . p. 38

4.3 Diagrama de classe da lista duplamente encadeada . . . . p. 39

4.4 Diagramas de classe das classes Scheduling_Queue e Thread . . . . p. 41

4.5 Implementação RTL de referência do escalonador . . . . p. 42

4.6 Diagrama de blocos do gerenciador de alocação . . . . p. 45

4.7 Busca por posição livre no bitmap de armazenamento . . . . p. 46

4.8 Diagrama do bloco raiz do escalonador . . . . p. 47

5.1 Arquitetura do SoC usado para validação . . . . p. 48

5.2 Diagrama de blocos do *mlite_cpu* . . . . p. 51

5.3 Máquina de estados finitos da interface mlite_cpu – AXI4Lite . . . . . p. 53

Lista de Tabelas

4.1 Operações oferecidas pela implementação de referência . . . . p. 42

5.1 Sinais de um barramento AXI4Lite . . . . p. 50

6.1 Casos de teste de verificação funcional do escalonador proposto . . . p. 55

6.2 Dados pré-síntese RTL de área e vazão do componente proposto . . p. 57

6.3 Dados pós-síntese RTL de área e atraso do componente proposto . . p. 58

6.4 Comparação de área e atraso entre nossa proposta e a referência . . p. 58

Lista de abreviaturas e siglas

EPOS	Embedded Parallel Operating System,	p. 10
HLS	High-Level Synthesis,	p. 10
BLP	Bit-Level Paralelism,	p. 10
FPGA	Field-Programmable Gate Array,	p. 10
EDA	Electronic Design Automation,	p. 10
VHSIC	Very High-Speed Integrated Circuit,	p. 10
VHDL	VHSIC Hardware Description Language,	p. 10
ASIC	Application-Specific Integrated Circuit,	p. 15
GPP	General Purpose Processor,	p. 15
RDPA	Reconfigurable Data-Path Array,	p. 15
ULA	Unidade Lógico-Aritmética,	p. 15
HDL	Hardware Description Language,	p. 15
ESL	Electronic System Level,	p. 15
HLS	High-Level Synthesis,	p. 15
DFG	Data Flow Graph,	p. 15
SoC	System-on-Chip,	p. 15
OSCI	Open SystemC Initiative,	p. 25
TLM	Transaction-Level Modelling,	p. 25
NoC	Network-on-Chip,	p. 25
DSL	Domain-Specific Language,	p. 25
RTL	Register Transfer Level,	p. 25
GPU	Graphics Processing Unit,	p. 25
SMP	Symmetric Multiprocessing,	p. 25
BORPH	Berkeley Operating system for ReProgrammable Hardware,	p. 25
BRAM	Block RAM,	p. 35
AMBA	Advanced Microcontroller Bus Architecture,	p. 35
AXI	Advanced eXtensible Interface,	p. 35
EDF	Earliest Deadline First,	p. 35
LUT	Look-Up Table,	p. 54

1 Introdução

> “E então a totalidade da aritmética se situava, repentinamente, dentro das possibilidades da mecânica”
>
> Charles Babbage - 1864

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.

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 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.

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 o uso de circuitos booleanos como um modelo para a implementação de algoritmos em larga escala. O problema fundamental no uso de circuitos booleanos

[1] do inglês “Computing machine”

era 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 física 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 é simples de se explicar. Os avanços na tecnologia de fabricação de transistores vinham permitindo que o poder de computação dos processadores aumentasse sem grandes remodelagens arquiteturais. Durante muito tempo, a frequência dos processadores cresceu seguindo uma função exponencial.

De fato, o cofundador da Intel®, Gordon E. Moore, previu essa “tendência” da indústria de microprocessadores com um célebre artigo(MOORE et al., 1965). Tal tendência foi verificada quase exatamente por décadas, porém vem perdendo validade gradativamente nos dias atuais. Embora a miniaturização dos transistores continue, o *aumento na frequência* dos processadores perde força devido à grande quantidade de potência dissipada em forma de calor(KURODA, 2001).

Levando em conta essa perda de velocidade no aumento da frequência 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 permitam 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 reduzido 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 frequência dos processadores já não aumentaria num passo tão rápido, a indústria 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 mais difíceis de serem alcançados; há muito mais a ser feito do que esperar o aumento "natural" das frequências de clock. Avanços significativos têm sido feitos em vários níveis de abstração, desde as técnicas de BLP e ILP 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* o paralelismo presente nas aplicações.

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 lógica programável deu um grande salto a partir de 1985 e durante toda a década de 90, com a introdução dos FPGAs. Tais dispositivos são flexíveis, e permitem a implementação de todo e qualquer circuito booleano. Alguns FPGA 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).

Além disso, a indústria de EDA vem trabalhando para aproximar o processo de projetar um circuito integrado do processo de desenvolver software. Esse esforço teve um início na década de 80, quando da criação das linguagens VHDL e Verilog; voltadas para a descrição de hardware com enfoque arquitetural, mas com construções típicas de linguagens de programação (por exemplo, laços e condicionais). O aparecimento das linguagens de descrição de hardware foi um passo inicial para encurtar a 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 no desenvolvimento de software – 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), aproveitando 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), a biblioteca C++ Accelerator(BOND et al., 2010) e a ferramenta Catapult-C(MENTOR, 2011). 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. Novas linguagens foram criadas com esse propósito, além de metodologias e ferramentas que visam obter um sistema completo, 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 viabilidade. 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 implementado utilizando uma ferramenta que permite a síntese direta para hardware, com o mínimo possível de intervenção humana.

Um escalonador foi escolhido como algoritmo a ser implementado em nosso estudo de caso por situar-se em uma área de aplicação (Sistemas Operacionais) suficientemente diferente das encontradas na bibliografia relacionada e por ter implementação em hardware suficientemente complexa. Um escalonador, dentre as várias famílias de abstrações em um Sistema Operacional, é a única que demonstra comportamento tanto *assíncrono* como *autônomo*(MARCONDES; FRÖHLICH, 2009), tornando um desafio a sua implementação em hardware. Certas implementações em hardware de um escalonador dão ao sistema em que se inserem a característica de possuir tempo de execução totalmente determinístico, o que pode ser crucial no suporte à execução de

aplicações *hard real-time*.

Várias possibilidade de microarquitetura para o mesmo algoritmo serão exploradas, e o componente sintetizado em um FPGA será integrado aos componentes restantes do EPOS, que executarão em um soft-processor no mesmo FPGA.

A verificação de corretude do modelo se dará através de simulações funcionais com a elaboração de um *testbench* auto-verificável, englobando casos de teste das situações de contorno para cada operação do modelo. Resultados da síntese de hardware – tais como área ocupada e atraso máximo – serão coletados e comparados com os mesmos dados obtidos de uma implementação RTL do mesmo componente.

2 Fundamentos

Neste capítulo são detalhados os conceitos teóricos e tecnologias mais importantes nas quais este trabalho se baseia.

2.1 Computação Reconfigurável

Até recentemente, havia majoritariamente duas maneiras de se implementar um algoritmo(COMPTON; HAUCK, 2002): A primeira consiste em projetar um circuito lógico digital que realiza a computação necessária; tipicamente um ASIC 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 ASIC e aos GPP. Considerando-se uma escala de flexibilidade pode-se dizer que os ASIC ocupam o extremo inferior, enquanto os GPP 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 parâmetros discutidos mais adiante.

O paradigma de computação reconfigurável consiste, praticamente, na implementação de algoritmos em dispositivos de hardware reconfigurável. Tais dispositivos podem ter

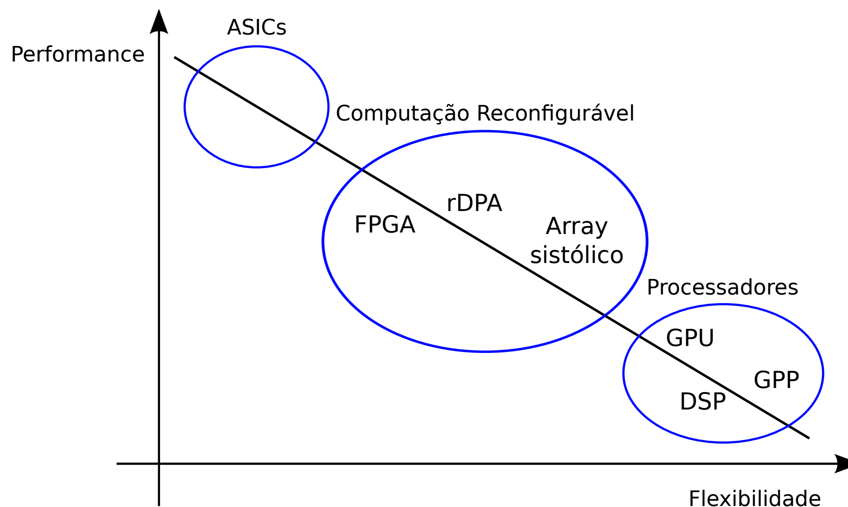


Figura 2.1: Escala de flexibilidade/performance comparando GPP, ASIC e Computação Reconfigurável

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.

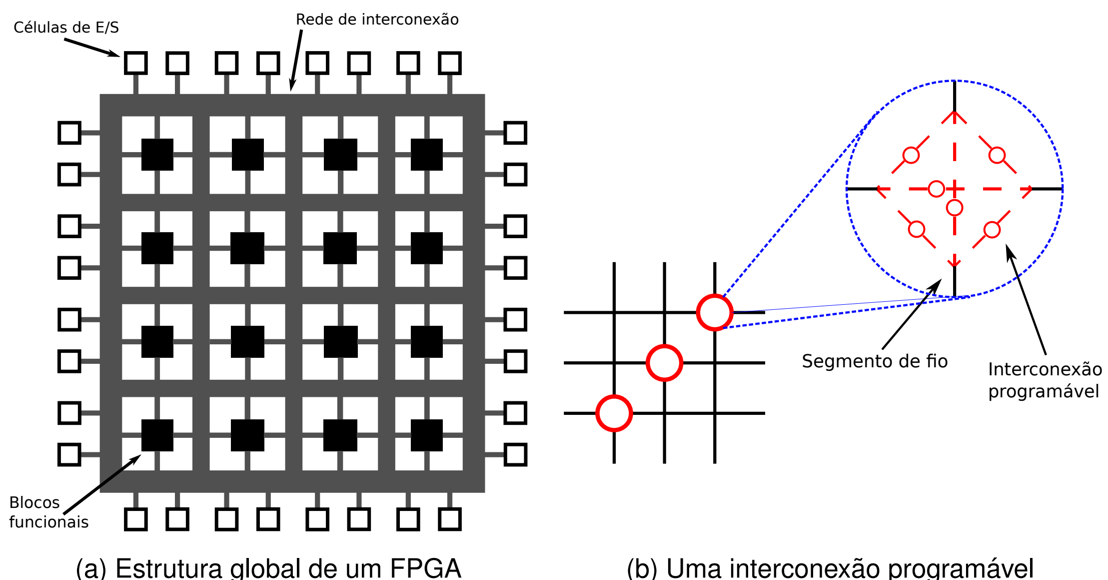
A utilização de dispositivos reconfiguráveis para acelerar a execução de algoritmos encontra-se em crescimento, e pode ser observada em diversas áreas de aplicação com marcantes características de paralelismo. Alguns exemplos citados na frequentemente na literatura(COMPTON; HAUCK, 2002) são:

- Criptografia e quebra de cifras
- Reconhecimento de padrões
- Algoritmos genéticos
- Computação científica
- Processamento Digital de Sinais (a área com maior utilização)

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 Já em um RDPA, a unidade básica de reconfiguração é normalmente uma ULA, 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 (HDL) 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 HDL 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 HDL possuem construções específicas para a modelagem de aspectos temporais.

Quando de sua criação, as HDL foram utilizadas principalmente para a modelagem e simulação de circuitos; somente com o desenvolvimento tecnológico tornou-se então possível realizar a síntese de ASIC a partir de código-fonte escrito em uma HDL. Um grande passo para o aumento massivo na utilização das HDL veio com surgimento dos FPGA a partir do início da década de 90. As cadeias de ferramentas dos fabricantes de FPGA aceitam código-fonte VHDL e/ou Verilog como a entrada "natural" para o processo de síntese.

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.

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.

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

(a) Função soma em C

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

architecture behavioral of adder is
   signal temp: signed(8 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

Outro ponto a se considerar é o tratamento da situação de *carry* [1]. 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 definido explicitamente (para não perder o caso de *carry*), e o bloco possui de fato 2 sinais de saída, um deles sendo o indicador de carry.

2.3 Síntese de alto nível

O processo de síntese de *alto nível* (HLS) consiste na geração de blocos de hardware a partir de uma descrição *comportamental* de um algoritmo(COUSSY; MORAWIEC, 2008) [2]. Esta descrição comportamental se opõe à descrição *arquitetural* realizada quando se utiliza o subconjunto sintetizável das linguagens VHDL ou Verilog.

A entrada para a síntese de alto nível normalmente é código-fonte escrito em uma linguagem de programação convencional. De fato, um dos princípios da síntese de alto nível é que o algoritmo deve ser primeiramente testado em software e só então rodar, com pouca ou nenhuma alteração, como um bloco de hardware. Certamente as linguagens mais populares utilizadas como entrada no processo de síntese de alto nível são C e C++, devido à sua ubiquidade no domínio de sistemas embarcados e à

[1] A soma binária de dois operandos de tamanho n pode resultar em um valor de tamanho n+1. Quando essa situação ocorre, diz-se que o somador produziu um bit de *carry*

[2] O processo de síntese de alto nível também é conhecido pelos nomes de síntese algorítmica, síntese no nível ESL, dentre outros

grande comunidade de desenvolvedores e engenheiros que as dominam.

Comumente, o resultado final do processo é uma descrição arquitetural em nível RTL do algoritmo modelado, em código VHDL ou Verilog, apesar de algumas ferramentas poderem gerar também *netlists*[3]. Tal resultado pode então ser processado por ferramentas dos fabricantes de FPGAs. A figura 2.4 resume o *fluxo de projeto* da síntese de alto nível.

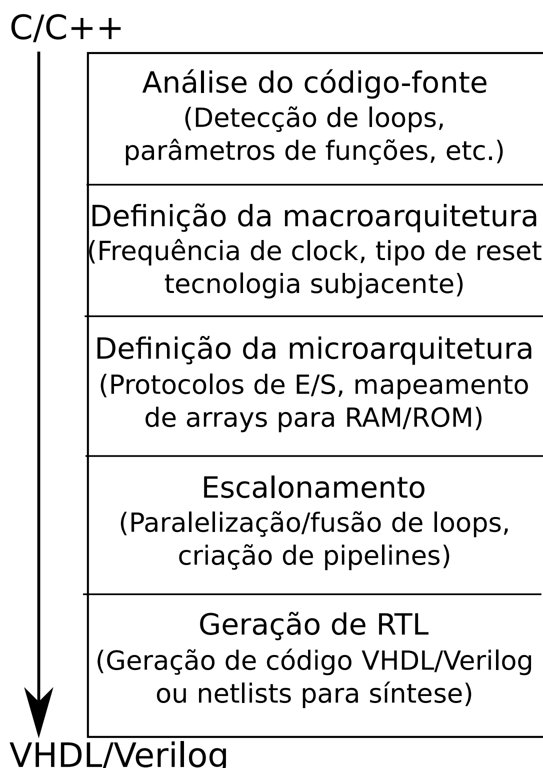


Figura 2.4: Fluxo de projeto da síntese de alto nível

2.3.1 Etapas da síntese de alto nível

Durante o processo de síntese de um algoritmo descrito em C ou C++ para RTL, vários parâmetros do sistema digital devem ser especificados, pois não podem ser inferidos automaticamente a partir da descrição na linguagem fonte. Esse ciclo: escolha dos parâmetros arquiteturais; síntese do sistema; verificação da adequação das características à especificação do projeto e escolha de novos parâmetros (caso seja necessário), é chamado de *exploração do espaço de projeto*.

[3]uma netlist é uma representação do circuito como um grafo onde os vértices são portas lógicas básicas e as arestas são fios. Existem formatos padronizados de representação de netlists, como por exemplo o EDIF.

De fato, a exploração do espaço de projeto na síntese de um sistema digital é um problema típico de otimização, bastante sujeito à automatização. As ferramentas de síntese de alto nível buscam facilitar esse processo através de uso de scripts para a definição de parâmetros, de maneira a tornar possível a interação com ferramentas que busquem a otimização automática desses parâmetros.

Os parâmetros cuja definição é necessária em cada etapa da síntese, assim como a influência de cada parâmetro nas etapas posteriores, são os seguintes(FINGEROFF, 2010):

Análise do código-fonte Nessa etapa são coletados os limites de loops, assim como informações sobre o uso dos parâmetros de funções; também é construído um grafo de dependências de dados (DFG). Loops ilimitados irão causar dificuldades em posteriores otimizações, e parâmetros que são ao mesmo tempo de entrada e saída exigirão lógica adicional na geração de RTL.

Definição da macroarquitetura Frequência de clock, tecnologia subjacente e limites gerais para o mapeamento de estruturas de dados em memórias são definidos nessa fase. A frequência de clock irá influenciar o escalonamento, definindo se operações podem ser realizadas em um único ciclo ou devem ocupar vários. A família de tecnologia subjacente também influencia o escalonamento, pois o atraso de um componente define se ele “cabe” em um ciclo de clock.

Definição da microarquitetura Nessa fase do processo o projetista já pensa em detalhes de implementação do bloco de hardware: quais serão os protocolos de comunicação pelos quais os parâmetros serão recebidos e enviados, a escolha precisa do tipo de armazenamento para cada array (registradores, RAM, ROM), *pipelines*, paralelização e fusão de loops.

Escalonamento Escalonar um algoritmo em um circuito digital consiste em definir quais operadores serão ativados em quais ciclos de clock, e com quais entradas. O escalonamento toma como entrada o DFG e as características de *timing* dos operadores (dependentes de tecnologia). Nessa etapa possivelmente são gerados pipelines, inserindo-se registradores e lógica de controle conforme necessário.

Geração de código RTL O passo final na síntese de alto nível. Nessa fase a representação intermediária do circuito, já escalonada, é convertida para um (ou mais) arquivo(s) VHDL/Verilog ou então para uma *netlist*. Tipicamente, as ferramentas

de HLS possuem uma biblioteca de componentes em VHDL e/ou Verilog, os quais são instanciados e interligados convenientemente em um grande arquivo que contém o bloco projetado pelo usuário e todas as suas dependências. Esse grande arquivo RTL pode então servir de entrada para as ferramentas das fabricantes de FPGAs.

2.4 O sistema operacional EPOS

O sistema operacional EPOS(FRÖHLICH, 2001) é um sistema operacional voltado para aplicações embarcadas. Ele foi programado em C++ utilizando-se de técnicas como orientação a aspectos e metaprogramação estática para alcançar um alto grau de configurabilidade e mesmo assim atender aos requisitos estritos de sistemas embarcados (como performance e tamanho de código). Várias aplicações interessantes e inovadoras demonstram a aplicabilidade do EPOS, por exemplo as aplicações em redes de sensores sem fio(FRÖHLICH; WANNER, 2008), envolvendo a plataforma EPOSMote®

O EPOS utiliza a metodologia de desenvolvimento de sistemas embarcados orientados à aplicação[4], 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. A figura 2.5(FRÖHLICH, 2011) detalha os vários processos de integração e configuração dos componentes e aspectos que podem fazer parte de uma imagem de sistema EPOS. Também se destaca o fato de que uma aplicação que utiliza o EPOS pode ser implementada como um SoC, utilizando-se de componentes implementados em hardware e em software.

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). A entidade

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

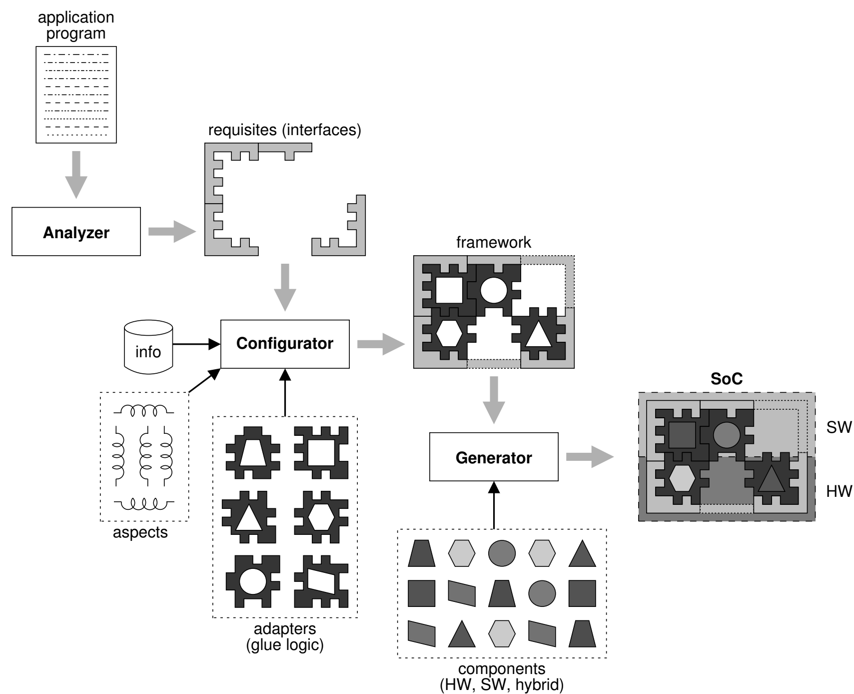


Figura 2.5: Processo de integração e adaptação dos componentes para produzir uma imagem de sistema EPOS

responsável por realizar essa adaptação da abstração ao cenário de execução é convenientemente chamada de *Adaptador de cenário*. A figura 2.6(FRÖHLICH, 2011) demonstra a relação entre adaptador de cenário, cenário, aspectos, componentes do sistema operacional e a aplicação.

Um dos ideais do EPOS é que os componentes do sistema operacional possam ser implementados tanto em software (executando em um GPP) como diretamente em hardware. De fato, o conceito de *componente híbrido* já foi bastante pesquisado, e em Marcondes e Fröhlich (2009) é dada a definição de uma arquitetura para a comunicação transparente entre componentes em hardware e em software.

Dois grandes problemas ainda restam, porém: 1) As descrições em VHDL/Verilog são pouco suscetíveis à aplicação de aspectos (a aplicação de aspectos sobre código-fonte dessas linguagens não é comum). 2) O mesmo componente deve ser descrito em duas linguagens diferentes (C++ e VHDL/Verilog), com grandes diferenças no estilo e na semântica das descrições, aumentando a chance de serem introduzidos *bugs* no sistema;

Uma proposta de solução para o primeiro problema mencionado está sendo pesquisada (MÜCK, 2011) ativamente. Ela consiste em realizar a descrição RTL dos

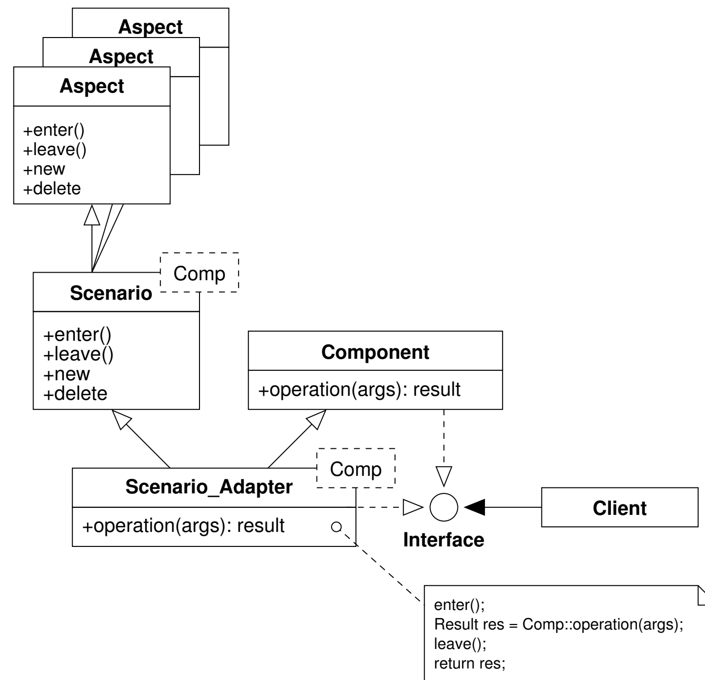


Figura 2.6: diagrama UML identificando as relações entre os principais conceitos do sistema EPOS

componentes a serem implementados em hardware utilizando o *framework* SystemC, em seu subconjunto sintetizável. Desse modo, todas as técnicas já conhecidas de aplicação de aspectos sobre C++ podem ser utilizadas, ao mesmo tempo em que a descrição pode ser sintetizada automaticamente (ferramentas de síntese de SystemC RTL já são relativamente comuns).

Já o segundo problema mencionado é bem mais complexo: idealmente, deveria haver apenas uma versão de código-fonte C++ para um componente híbrido, e a versão desse componente implementada em hardware deveria ser sintetizada automaticamente, sem requerer uma tradução manual para RTL. O presente Trabalho de Conclusão de Curso busca contribuir com a solução dessa questão através de um estudo de caso, tentando demonstrar a plausibilidade da utilização de HLS para a descrição de componentes híbridos de SO.

3 Trabalhos Correlatos

"Se vi mais longe foi por me apoiar
sobre ombros de gigantes"

Isaac Newton

O esforço de se realizar síntese de hardware a partir de descrições comportamentais ainda é bastante imaturo, e várias metodologias, linguagens e ferramentas foram propostas desde meados dos anos 80. Essas propostas eram voltadas, porém, para aplicações específicas, como por exemplo o célebre "compilador de silício"[1] *Cathedral*(MAN et al., 1986), desenvolvido no IMEC (Bélgica), que focava a área de processamento digital de sinais.

Durante a década de 90, o processo de síntese a partir de VHDL/Verilog em nível RTL foi se aprimorando e até mesmo foram feitas algumas propostas de utilização de VHDL em nível comportamental, resultando em um modelo parcialmente temporal. Porém, 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 de *propósito geral* passaram a ter uma maior penetração na 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 SO para a computação em FPGA. Nesse âmbito estão inclusos tanto a aplicação de técnicas de SO em computação reconfigurável (escalonamento de tarefas, troca de contexto, comunicação inter-processos), como esforços de integração

[1] do original em inglês *silicon compiler*

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 variados 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(OSCI, 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, 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

Dentre os produtos comerciais para síntese de alto nível, 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 e a ferramenta faz a tradução para componentes RTL, os quais podem então ser fabricados como ASIC, ou sintetizados em FPGA (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, buscando otimizar tais métricas.

3.1.2 Síntese de ANSI C++

Além do SystemC, existem também abordagens que buscam realizar a síntese a partir de descrições em ANSI C++. Algumas ferramentas trabalham com C++ padrão ANSI, e permitem utilização de construções avançadas da linguagem, tais como ponteiros, classes e structs, *arrays* multidimensionais e metaprogramação via *templates*. Já outras ferramentas trabalham com subconjuntos mais restritos da linguagem, e usam extensões que tornam as descrições de hardware incompatíveis com compiladores tradicionais de C++ para software.

Algumas construções de C++, porém, são invariavelmente inadequadas à uma implementação em hardware e, portanto, não são suportadas por nenhuma das ferramentas de síntese pesquisadas. Em geral a inadequação dessas construções advém da necessidade de conhecer todos os detalhes do algoritmos em tempo de síntese. Tal necessidade proíbe:

Alocação dinâmica de memória A quantidade de memória a ser utilizada pelo algoritmo precisa ser um valor conhecido em tempo de síntese. Alocar memória dinamicamente em um FPGA significaria reconfigurar partes de um FPGA dinamicamente, uma tarefa atualmente ainda complexa e ineficiente.

Recursões não limitadas Funções recursivas sem limite explícito de profundidade de chamadas recaem no uso de alocação dinâmica e são, portanto, inadequadas à implementação em hardware. Isso pois, na chamada de uma nova instância

de função recursiva, novo espaço teria de ser alocado para as variáveis no escopo local *daquela* instância. Algumas ferramentas dão suporte à *recursão com profundidade limitada*, implementada utilizando-se de metaprogramação estática via *templates*.

Uma cadeia de ferramentas de grande destaque em HLS 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 C++ (ANSI) 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.

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.

3.1.3 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) 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:

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

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.

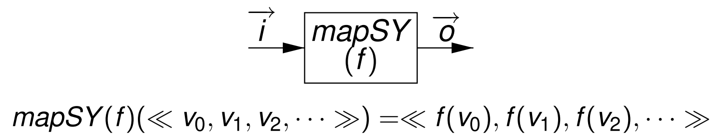
```
entity dotp is
    port (resetn : in std_logic;
          clock  : in std_logic;
          v1sig  : in int16;
          v2sig  : in int16;
          result : out int16);
end entity dotp;

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: VHDL gerado para o componente *multiply-accumulate*

Através da 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.

3.1.4 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 clas-



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

Figura 3.3: O construtor de processo mapSY

ses C++. Utilizando-se de classes e funções da biblioteca Accelerator o usuário é capaz de expressar algoritmos no paradigma de *paralelismo aninhado de dados*[2] (BLELLOCH, 1996). Esses algoritmos podem ser executados em variadas plataformas-alvo (GPU, SMP, *clusters* e FPGA).

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 biblioteca Accelerator produz uma árvore de expressão representando o algoritmo. Na figura 3.4 temos um exemplo de árvore de expressão Accelerator.

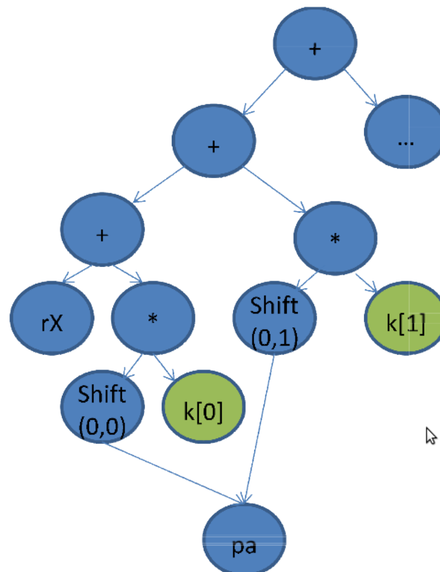


Figura 3.4: Árvore de expressão gerada por meio de operações sobre um objeto do tipo ParallelArray

[2]do inglês "Nested Data Parallelism"

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 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

3.2.1 BORPH

Em So e Brodersen (2008), os autores descrevem a implementação de um sistema operacional para computadores reconfiguráveis bastante interessante: o sistema BORPH. 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 FPGA que irá executá-lo, e o código binário para tal configuração é armazenado em um formato de arquivo inspirado no ELF.

O BORPH não trata, porém, da questão de implementação de *componentes do SO* em hardware. Como este SO segue o modelo de implementação do Linux, o *kernel* do sistema é monolítico. Desse modo, componentes como escalonador, timer e sistemas de arquivos não podem ser implementados em hardware, pois não são processos.

3.2.2 HybridThreads

O projeto *HybridThreads*(ANDREWS et al., 2004) (HThreads), da Universidade de Arkansas, consiste de um SO que suporta a implementação de *threads* tanto em hardware quanto em software, e a utilização dessas *threads* de forma transparente para o usuário. Um *wrapper* que disponibiliza a interface com todas as operações típicas realizadas sobre uma *thread* foi implementado como um Soft IP, fazendo com que a interface seja uniforme entre *threads* em hardware e em software seja uniforme.

Uma das características distintivas do HThread é o fato de que, como pode ser observado na figura 3.5, alguns componentes do SO são implementados em hardware, como por exemplo o próprio escalonador, visando a redução de *jitter*[3] e melhora no determinismo do tempo de execução dos aplicativos usando o sistema. O mesmo objetivo (diminuição de jitter) também é almejado com a nossa implementação do escalonador do EPOS em hardware.

Tanto as *threads* do usuário, quanto os componentes do sistema HThreads que rodam em hardware devem ser, porém, descritos em VHDL ou Verilog em nível RTL. Faz parte do projeto *HybridThreads* um esforço para fazer síntese automática a partir de código-fonte C, mas esse esforço ainda é bastante imaturo. Talvez a própria modelagem do sistema, considerando uma *thread* como componente básico a ter implementação em hardware, seja inadequada.

[3]termo normalmente usado para variação de atraso

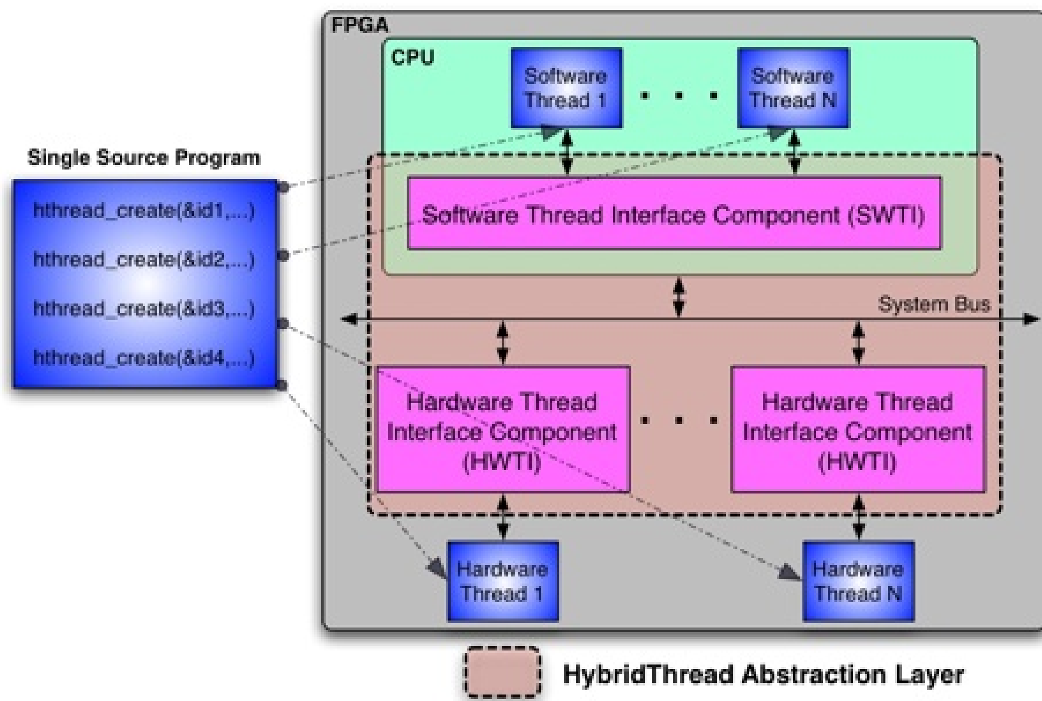


Figura 3.5: Arquitetura geral de um SoC utilizando HThreads

4 Desenvolvimento

> "A elegância não é luxo supérfluo, mas uma qualidade *decisiva* entre o sucesso e o fracasso"
>
> Edsger Wybe Dijkstra

A solução desenvolvida para alcançar os objetivos previstos, com todos seus detalhes de implementação, é apresentada neste capítulo. São também justificadas as escolhas de projeto feitas durante o desenvolvimento.

Nossa implementação de um escalonador em hardware para o EPOS iniciou-se com uma análise da atual implementação (em software) do escalonador do sistema. Propositadamente, tentamos nos aproximar o máximo possível da implementação em software, realizando na modelagem e no código somente as alterações impostas pela ferramenta de síntese de alto nível.

A ferramenta de síntese de alto nível utilizada foi o Catapult-C®, da fabricante Mentor Graphics. O uso desta ferramenta em particular se justifica, principalmente, pelo seu suporte abrangente às várias funcionalidades da linguagem C++ (linguagem em que a maior parte do EPOS está implementada). Sem o suporte do Catapult-C à *templates*, em especial, seria praticamente impossível sintetizar código EPOS.

Apesar de não termos testado outras ferramentas, cremos que as limitações com as quais nos deparamos durante o trabalho (detalhadas nas seções seguintes) são gerais, referindo-se ao próprio cenário de implementação (hardware) e não à ferramenta que escolhemos.

Nenhuma otimização ou alteração foi feita no código C++ visando especificamente a implementação em hardware. Dessa maneira, podemos situar o presente trabalho como um "extremo oposto" em nível de abstração, se comparado à modelagem RTL.

Nas seções seguintes é mostrado o escalonador atual do sistema EPOS (sua

implementação em software) e é feita uma análise de suas estruturas de dados. Logo após, damos uma visão geral da implementação de referência em hardware (nível RT) com a qual nossa modelagem será comparada. Por fim, descrevemos em detalhes nossa proposta de implementação.

4.1 O escalonador do EPOS em software

O componente escalonador atual do sistema EPOS está modelado no *template* de classe System::Scheduler<T>, e portanto é capaz de escalonar recursos de um tipo arbitrário, e com um critério de escalonamento também arbitrário. Os requisitos a serem seguidos pelo parâmetro T são os mesmos impostos pelo template System::Scheduling_Queue<T>, e tais requisitos são detalhados na seção 4.2. De fato, a classe Scheduler *herda* seus métodos da classe Scheduling_Queue, adicionando alguns outros para exportar uma interface mais conveniente. A figura 4.1 mostra as relações de herança em que a classe Scheduler se envolve.

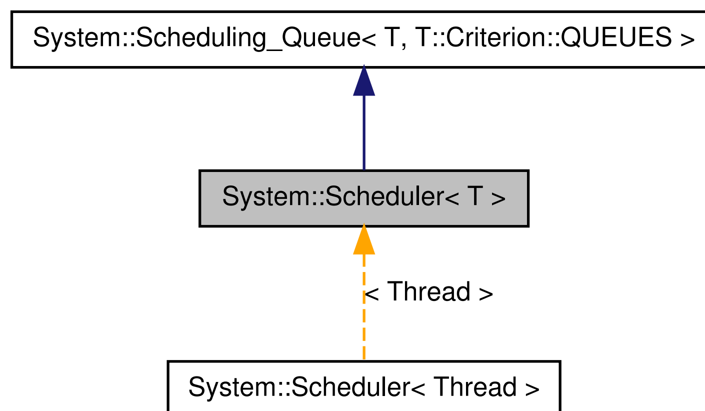


Figura 4.1: Diagrama mostrando as relações de herança e especialização do escalonador atual do EPOS (implementação em software). A linha pontilhada indica uma especialização de template.

Em vários sistemas operacionais o escalonador é o componente de software responsável pelo gerenciamento do recurso CPU, a ordenação das threads que concorrem pelo seu uso e a troca de contexto no momento da escolha de uma nova thread para ocupar a CPU. Já no EPOS o componente escalonador tem a *única responsabilidade* de gerenciar uma (ou várias) filas de entidades que concorrem ao recurso (CPU ou outros). As responsabilidades de gerar interrupções periódicas para *sensibilizar* o

escalonador e de realizar a troca de contexto propriamente dita ficam por conta dos componentes *Alarm* e *CPU*, respectivamente.

Essa boa separação de responsabilidades(HÜRSCH; LOPES, 1995) no projeto do EPOS contribui para uma maior facilidade na implementação dos componentes-chave do sistema operacional em hardware. Tal separação se manifesta de maneira bastante clara nas estruturas de dados do sistema operacional, que têm aspectos de encadeamento e armazenamento fatorados e integrados em tempo de compilação através de metaprogramação estática(FRÖHLICH, 2001).

4.2 Estruturas de dados utilizadas

O diagrama 4.2 mostra a hierarquia de herança entre as classes de estruturas de dados utilizadas pelo escalonador atual do EPOS (em software).

A estrutura de dados fundamental do escalonador é uma fila de escalonamento, objeto da classe System::Scheduling_Queue<T,Q>, com os parâmetros:

T: O tipo de objeto a ser escalonado. Como a fila é genérica, não apenas *Threads* são passíveis de escalonamento. Qualquer tipo pode ser utilizado como T, devendo satisfazer apenas aos seguintes requisitos:

- Declarar um tipo chamado “Criterion”, que será usado como critério de ordenamento da fila.
- Declarar um método “link”, para satisfazer à interface de elemento de lista.

Q: parâmetro de tipo inteiro sem sinal, indica *quantas* filas de escalonamento devem ser utilizadas.

Sobre a hierarquia de listas e filas do EPOS cabem ainda alguns comentários adicionais:

List<T, El> É um template de lista duplamente encadeada, onde o parâmetro El encapsula o encadeamento.

- El deve implementar *getters* e *setters* para os atributos “next” e “prev”.
- El é um parâmetro **opcional** do template, e seu valor padrão é List_Elements::Doubly_Linked<T>

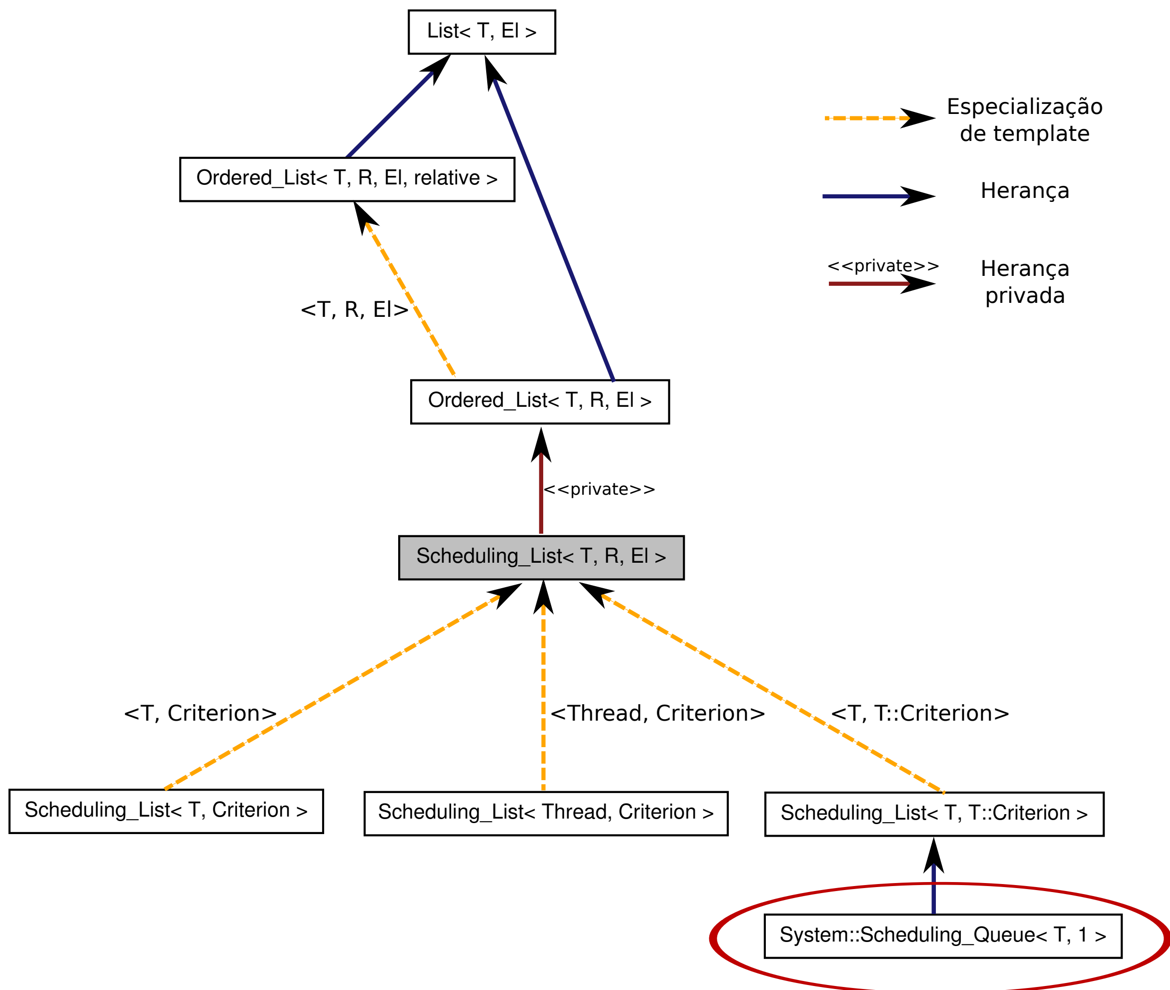


Figura 4.2: Diagrama das relações de herança e especialização entre as estruturas de dados da implementação do escalonador, destacando-se a fila de escalonamento

Ordered_List<T, R, El, relative> Subclasse de List<T, El>, este é um template de lista ordenada, que recebe dois parâmetros adicionais:

R: um tipo que encapsula o conceito de *rank* de um elemento. Qualquer tipo de dados *isomórfico* aos números inteiros pode ser usado como R, devendo implementar o método de *cast* para int. Este parâmetro é **opcional** e seu valor padrão é List_Element_Rank.

relative: um parâmetro booleano indicando se a lista é *relativa*. Diz-se que uma lista ordenada é relativa quando cada elemento armazena apenas a diferença entre seu rank e o de seu antecessor. Este parâmetro é **opcional** e seu valor padrão é false.

4.2.1 Lista duplamente encadeada

A raiz na hierarquia de classes a serem implementadas em hardware, e portanto a estrutura de dados fundamental para o escalonador, é uma lista duplamente encadeada. O diagrama UML da figura 4.3 mostra as operações implementadas pela lista.

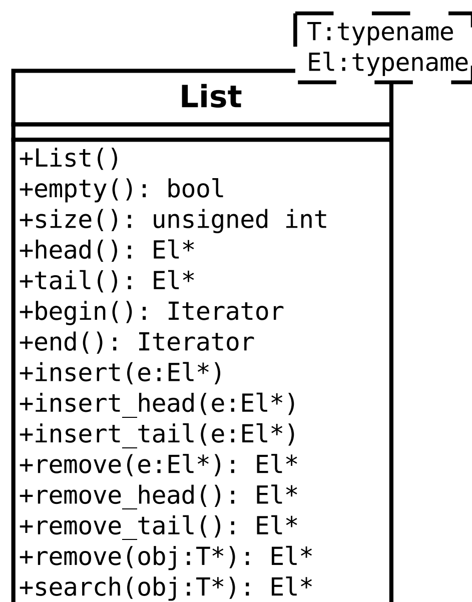


Figura 4.3: Diagrama de classe UML da lista duplamente encadeada usada no trabalho

Dada a importância dessa estrutura nós decidimos implementá-la antes de partir para a implementação do escalonador em si, como maneira de explorar a ferramenta

de síntese e conhecer mais a fundo suas limitações. O código C++ da lista, do seu gerenciador de alocação, assim como do testbench de verificação funcional VHDL e o script com as diretivas da síntese se encontram no apêndice A.

A estratégia de verificação por *asserções* foi utilizada, com o uso das palavras-chave *assert* e *report* da linguagem VHDL. O testbench consiste de vários casos de teste autoverificáveis, que são executados em sequência. Cada caso de teste compara o valor de uma saída produzida pelo bloco sob teste com uma saída esperada. Caso haja discrepância entre os valores, tanto o valor esperado quanto o produzido são relatados.

4.2.2 Lista ordenada

A lista ordenada do EPOS (classe Ordered_List<T, R, El>) estende a lista duplamente encadeada, e possui a mesma interface (mesmos métodos públicos) que aquela. Duas diferenças são notáveis, porém:

- Há um parâmetro extra (R) para o template: esse parâmetro é o tipo de *rank* usado na ordenação dos elementos. Um tipo T qualquer, para poder servir de rank, deve realizar a sobrecarga do operador de cast para tipo inteiro.
- A classe Ordered_List<T, R, El>reimplementa os métodos insert e remove da sua superclasse, para realização a inserção sempre *em ordem* (de acordo com o rank), bem como a atualização dos ranks vizinhos na remoção (no caso de rank relativo).

4.2.3 Fila de escalonamento

Seguindo a filosofia de implementação do EPOS, o comportamento de filas de escalonamento está isolado dos seus detalhes de implementação (como alocação e armazenamento) e implementado na classe *Scheduling_Queue<T, Q>*. O parâmetro T é o tipo de entidade a ser escalonado, enquanto o parâmetro Q define o número de filas a serem usadas no escalonamento.

Qualquer tipo T de entidade pode ser escalonado, desde que satisfaça ao seguinte requisito:

- T deve declarar um tipo “Criterion”. O tipo T::Criterion será utilizado como critério de ordenação das filas subjacentes ao escalonador, e define qual é o algoritmo de escalonamento a ser utilizado. Alguns critérios já implementados no EPOS são Round-Robin, Rate monotonic e EDF.

O diagrama 4.4 detalha os métodos públicos da classe Scheduling_Queue, assim como um exemplo de entidade escalonável, uma *Thread*

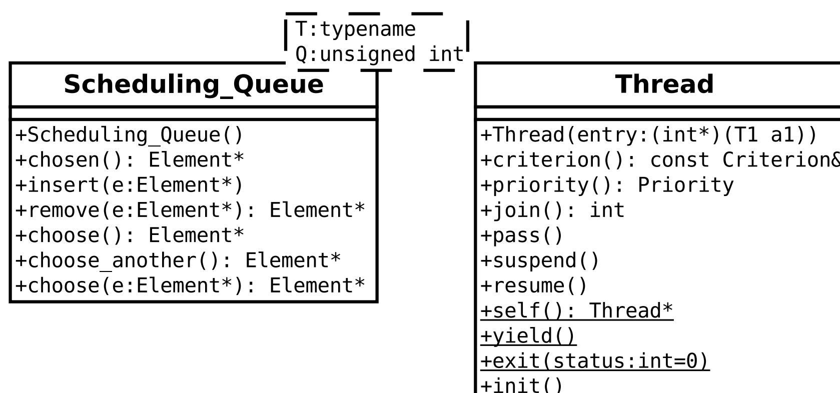


Figura 4.4: Diagrama de classe UML da fila de escalonamento utilizada no trabalho e da classe Thread, um exemplo de entidade escalonável

4.3 Implementação RTL de referência

Para avaliar a eficácia do processo de síntese de alto nível nós realizamos uma análise comparativa (em termos de área ocupada e atraso) com uma implementação de referência; os detalhes dessa análise encontram-se no capítulo 6.

Nossa referência é a implementação de um escalonador em nível RT (escrito na linguagem VHDL), descrito em (MARCONDES; FRÖHLICH, 2009). A figura 4.5 mostra a arquitetura geral desse escalonador.

O bloco escalonador contém vários pequenos sub-blocos, cada um encapsulando um dos serviços fornecidos. Além desses blocos algorítmicos, ainda compõem o escalonador uma memória que armazena a fila de escalonamento, um bloco responsável pelo gerenciamento de recursos e uma interface que faz a interpretação dos comandos recebidos do mundo externo e ativa o bloco algorítmico correspondente.

Para proporcionar uma comparação válida com a referência nossa proposta de escalonador aceita em sua interface o mesmo conjunto de comandos. A tabela 4.1 deta-

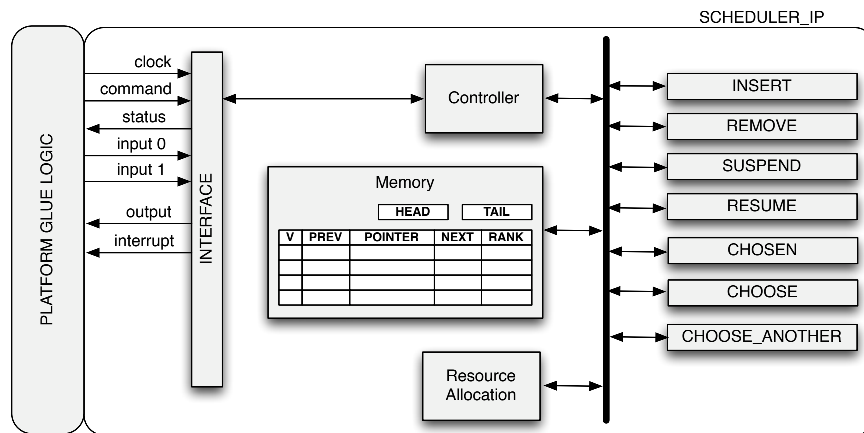


Figura 4.5: Diagrama de blocos da implementação de referência do escalonador (descrita em VHDL nível RT)

lha cada uma das operações oferecidas pela implementação de referência (e também na implementação proposta, por consequência).

Nome	Código	Entradas	Saídas
ENABLE	0x08	nenhuma	nenhuma
DISABLE	0x09	nenhuma	nenhuma
CHOSEN	0x12	nenhuma	índice de thread
CREATE	0x01	thread_id, priority	índice de thread
INSERT	0x03	índice de thread	nenhuma
DESTROY	0x02	índice de thread	nenhuma
REMOVE	0x04	índice de thread	nenhuma
REMOVE_HEAD	0x05	nenhuma	índice de thread
SIZE	0x13	nenhuma	tamanho da fila
GET_ID	0x11	índice de thread	thread_id

Tabela 4.1: Nome, código, entradas e saídas de cada uma das operações oferecidas pela implementação de referência.

4.4 Nossa proposta de escalonador em hardware

Como já foi mencionado no início do capítulo, nossa implementação de um escalonador a ser implementado em hardware foi guiada pelo desejo de se realizar o menor número possível de modificações em relação ao escalonador que executa em software. Em particular, quanto às alterações que tiveram de ser realizadas, dois princípios foram seguidos:

- As alterações devem permitir que o componente possa *continuar* a executar em software sem alteração de semântica, mesmo que com penalidade de performance.
- Deve-se, sempre que possível, encapsular as alterações em classes, evitando alterar diretamente o código-fonte do componente a ser sintetizado.

As alterações realizadas foram ditadas pelas restrições impostas pela ferramenta de síntese (Catapult-C, seção 3.1.2). A seguir é detalhado cada um dos tipos de alterações realizados sobre o código C++ original do EPOS, justificando-se a necessidade de tal alteração e o raciocínio por trás da solução adotada.

4.4.1 O tipo Maybe<T>

Durante o fluxo de síntese de alto nível, estruturas do tipo array[1] presentes no código são mapeadas para bancos de registradores ou memórias, e as variáveis do tipo ponteiro são mapeadas para índices dos arrays aos quais se referem.

Em um código C++ sintetizável, todo ponteiro deve se referir a uma estrutura de dados definida no código. Uma análise estática do código é realizada pela ferramenta de síntese para garantir que em nenhum dos possíveis fluxos de execução um ponteiro inválido seja desreferenciado. Em particular, são proibidas atribuições de literais inteiras a variáveis do tipo ponteiro.

As interfaces das estruturas de dados do EPOS em geral, e a da classe Scheduling_Queue, em particular, trabalham com ponteiros, e em várias situações ponteiros nulos (inválidos), são retornados para sinalizar situações de falha. Para contornar essa inadequação aos requisitos de síntese introduzimos o *tipo opção*[2] Maybe<T>.

O tipo Maybe<T>pode ser conceitualmente definido da seguinte maneira:

```
Maybe T = Nothing | Just T
```

Ou seja, há dois construtores possíveis para obter-se um valor do tipo Maybe<T>: Um construtor vazio (chamado de *Nothing*) e um construtor que toma um parâmetro (chamado de *Just*).

[1]uma sequência de elementos de dados residentes em uma faixa contínua do espaço de endereçamento

[2]do inglês "Option Type". http://en.wikipedia.org/wiki/Option_type

O conceito de tipo opção advém de linguagens de programação funcionais, e é uma maneira popular de se modelar operações que podem falhar. Em nosso caso, por exemplo, o método de remoção de um elemento da fila de escalonamento fica com a seguinte assinatura:

```
Maybe<Element*> remove(Element* e)
```

A operação de remoção pode falhar (no caso de o elemento procurado não estar presente na fila), e essa falha é então representada pelo retorno de um valor Nothing.

O código completo da classe Maybe<T>implementada neste trabalho segue:

```
template<typename T> class Maybe {
public:
    Maybe(): _exists(false), _thing(T()) {}
    Maybe(T thing, bool exists = true): _exists(exists), _thing(thing) {}

    bool exists() const { return _exists; }
    T get(T rogue = T()) const { return _exists ? _thing : rogue; }

    bool operator==(const Maybe<T>& other) const {
        return (!_exists && !other._exists) ||
            (_exists && other._exists && _thing == other._thing);
    }

    bool operator!=(const Maybe<T>& other) const { return !(*this == other); }

private:
    T _thing;
    bool _exists;
};
```

4.4.2 Gerenciamento de alocação

Uma das limitações da implementação de algoritmos em hardware, como já discutido no capítulo de fundamentos, é a impossibilidade de se utilizar alocação dinâmica de memória.

Por outro lado, todas as estruturas de dados do EPOS são independentes da forma com que seus elementos estão alocados. Isso permitiu que não fizéssemos qualquer

alteração no código das estruturas nesse sentido. Devido, porém, justamente a essa independência, tivemos que implementar separadamente um gerenciador de alocação para as filas de escalonamento.

A classe Scheduler_PreAlloc implementa esse gerenciador. Um objeto da classe Scheduler_PreAlloc é um *invólucro* para um objeto da classe Scheduling_Queue, e exporta publicamente uma interface bastante semelhante à do objeto envolvido. Duas grandes diferenças podem ser notadas: as operações possuem semântica de passagem *por valor* e nas inserções e remoções são feitas alocações e liberações de espaço no armazenamento subjacente. A figura 4.6 mostra um diagrama de blocos do gerenciador, enquanto seu código se encontra no apêndice B.1.

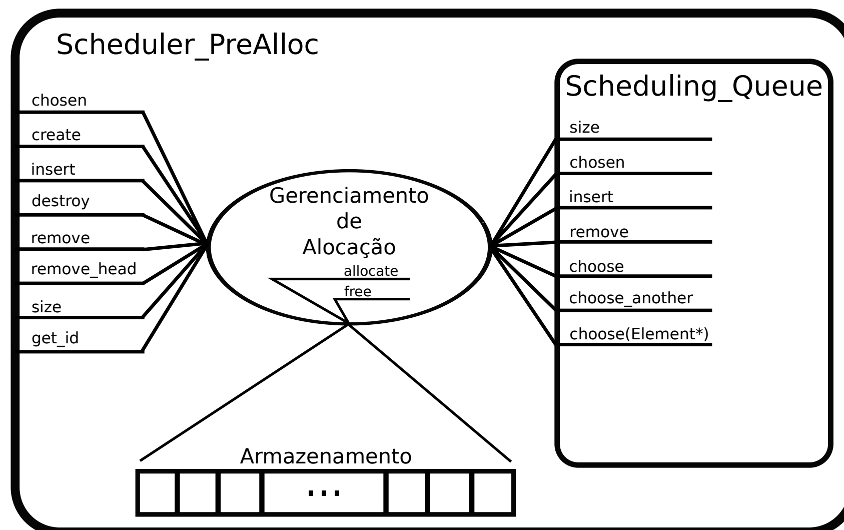


Figura 4.6: Diagrama de blocos do gerenciador de alocação para a fila de escalonamento Scheduling_Queue

O armazenamento para os elementos das filas de escalonamento é realizado em uma estrutura de dados do tipo array. A figura 4.7 mostra o código da busca por uma posição livre no array, realizada quando há a necessidade de alocar espaço para um novo elemento.

Este código faz uma busca linear sobre um bitmap de alocação para o array. O laço que percorre o bitmap pode ser desenrolado, e essa é, de fato, uma das otimizações discutidas no capítulo de resultados.

```
alloc_search:
for(Idx i = Idx(); i < Max; ++i) {
    if(alloc_bitmap[i] == false) {
        alloc_bitmap[i] = true;
        return i;
    }
}
```

Figura 4.7: Busca por uma posição livre no bitmap de armazenamento da classe Scheduler_PreAlloc

4.4.3 Wrapper da interface raiz

Uma das restrições mais importantes da ferramenta de síntese de alto nível utilizada se refere à forma como é feita a inferência das portas de entrada e saída do bloco de hardware sendo modelado. O projetista deve designar *uma única* função em todo o seu código como a função "raiz" do bloco. Isso é feito colocando-se o *pragma*[3] "hls_design top" sobre a assinatura da função.

A ferramenta de síntese irá então inferir, a partir dos parâmetros presentes na assinatura e da forma como eles são utilizados, as portas de entrada e saída do bloco de hardware sendo modelado, sendo que os tipos dos parâmetros definirão os tamanhos das portas.

Nosso escalonador é um objeto, e sua interface possui um método para cada operação oferecida, com parâmetros diferentes para cada método. A figura 4.8 mostra a solução adotada para adaptar a interface da classe *Scheduler_PreAlloc* à função que define a interface do bloco em hardware.

Basicamente, a união de todos os possíveis parâmetros dos métodos da classe Scheduler_PreAlloc é manifestada na assinatura da função *call* (que é anotada com o pragma "hls_design top"). Além disso, a função call possui um parâmetro extra do tipo *MethodId*. O tipo MethodId é um tipo enumerado e um valor desse tipo identifica uma operação realizável pelo bloco de hardware.

Dois parâmetros de saída estão presentes na função call: *status* e *return_val*. A ferramenta de síntese não impede que a função raiz possua valor de retorno, porém a utilização de parâmetros passados por referência como forma de saída de dados foi escolhida por a considerarmos uma maneira mais uniforme de expressar as saídas.

[3]anotação não-padronizada feita no código que dá ao compilador informações extras sobre o software, não presentes na linguagem fonte

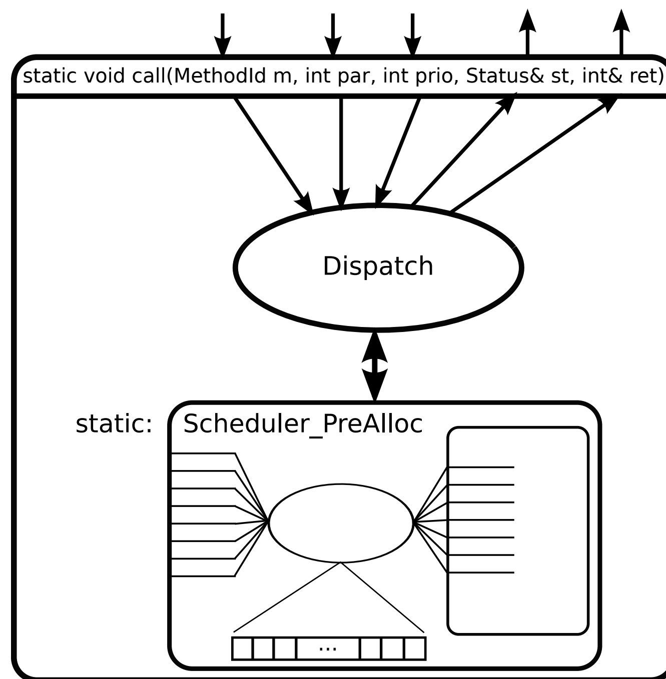


Figura 4.8: Diagrama do bloco raiz do escalonador sintetizado, enfatizando as portas de entrada e saída e a lógica de *dispatch* das operações

5 Ambiente de validação

Para efeitos de validação da implementação desenvolvida para o escalonador, o SoC com a arquitetura na figura 5.1 foi desenvolvido. Nesse cenário, todos os componentes do EPOS executam em software sobre um *soft-processor*[1], excetuando-se unicamente o escalonador.

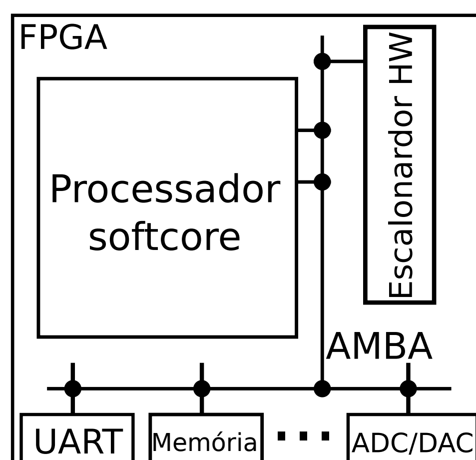


Figura 5.1: Arquitetura do SoC utilizado na validação da implementação desenvolvida

Todo o sistema foi sintetizado para um kit de desenvolvimento Virtex®6, modelo ML605, da fabricante Xilinx®. O uso da Virtex®6, em detrimento de outros FPGA que temos disponíveis, se deu principalmente pela maior quantidade de células lógicas que ele possui.

Na implementação de referência (VHDL) do escalonador(MARCONDES; FRÖHLICH, 2009) havia pouca área disponível no FPGA, e por isso alguns cenários de implementação com maior nível de paralelismo não puderam ser avaliados. Desejamos agora poder avaliar tais cenários.

Algumas especificações do kit ML605 seguem:

[1] um processador implementado na lógica programável de um FPGA

FPGA Família Virtex®6, modelo XC6VLX240T, encapsulamento FF1156

Memórias DDR3 SODIMM RAM, 16MB Platform Flash (configuração), 32MB BPI Flash

Entrada/Saída CompactFlash, USB JTAG, VGA, Gigabit Ethernet, PCI Express, UART via USB

A memória utilizada pelo processador *softcore* que roda O EPOS é uma BRAM de 1MB, interna ao próprio FPGA. Tal escolha foi tomada em parte por comodidade, pois evita a necessidade de se utilizar um controlador DDR3 e definir no arquivo *top-level* todos os pinos necessários à comunicação com a memória externa ao FPGA. Além disso, a aplicação a ser executada possui baixa utilização de memória, e caso seja necessária mais capacidade para futuras aplicações, a comunicação com a memória externa ao FPGA pode ser implementada de maneira relativamente simples.

Como barramento de sistema, conectando todos os *cores*, foi utilizado o AXI4Lite, parte da família de protocolos de barramento AMBA, da ARM®. Também a escolha do AXI4Lite se deu em boa parte pela sua simplicidade, e também pelo crescente uso de barramentos da família AMBA, tendo sido inclusive escolhido pela Xilinx®como barramento padrão para as futuras versões da sua cadeia de ferramentas.

A arquitetura do barramento e as modificações necessárias para que o processador e os outros blocos do sistema se adaptassem à ele são descritas com mais detalhes na seção 5.1.

5.1 Barramento AMBA AXI4Lite

O protocolo de barramento utilizado foi o AMBA AXI4(ARM, 2010), mais especificamente em sua variante simplificada, o AXI4Lite. O AXI4Lite é um subconjunto (totalmente interoperável) do protocolo AXI4 voltado para a comunicação com registradores de controle mapeados em memória. Comparado ao AXI4, o AXI4Lite tem as seguintes limitações:

- Todas as transferências tem tamanho de 1 palavra
- Todos os acessos a dados são de mesmo tamanho que a largura do barramento
- A largura do barramento pode ser somente 32 ou 64 bits

- Funcionalidades de *cache* e acesso exclusivo não estão implementadas

O AXI4 (e por extensão, o AXI4Lite) é um protocolo bastante ortogonal, e os canais de transmissão para operações de leitura são separados dos de escrita, assim como para transmissão de endereço e de dados. A tabela 5.1 lista todos os sinais do protocolo, com o canal ao qual cada um pertence.

Globais	**Endereço Escrita**	**Dados Escrita**	**Resposta Escrita**	**Endereço Leitura**	**Dados Leitura**
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESET	AWREADY	WREADY	BREADY	ARREADY	RREADY
–	AWADDR	WDATA	BRESP	ARADDR	RDATA
–	AWPROT	WSTRB	–	ARPROT	RRESP

Tabela 5.1: Canais de transmissão do protocolo AXI4Lite

5.2 O processador Plasma MIPS

Como processador *softcore* para execução do sistema em nosso cenário de validação foi escolhido o Plasma MIPS(PLASMA, 2001), disponível no repositório de código aberto OpenCores[2]. O Plasma MIPS é um processador que oferece compatibilidade binária (a nível de ABI) com processadores da arquitetura MIPS I.

Ele é implementado em VHDL, e é sintetizável tanto em FPGAs Xilinx®quanto Altera®. Ele possui um pipeline configurável de dois ou três estágios, e vem acompanhado de controladores SRAM, DDR, UART e Ethernet.

Em nossa implementação, utilizamos apenas o núcleo principal do Plasma (mlite_cpu), por entender que já possuíamos os outros controladores em nosso próprio repositório de IPs[3], e que seria mais elegante e robusto integrar o núcleo aos controladores por meio de um barramento AMBA. O cenário de configuração do mlite_cpu que utilizamos é o seguinte:

Clock 50 MHz

[2] `http://www.opencores.org`

[3] "IP", é um termo normalmente utilizado para se referir a blocos de hardware descritos em uma HDL, significando "Semiconductor Intellectual Property Core"

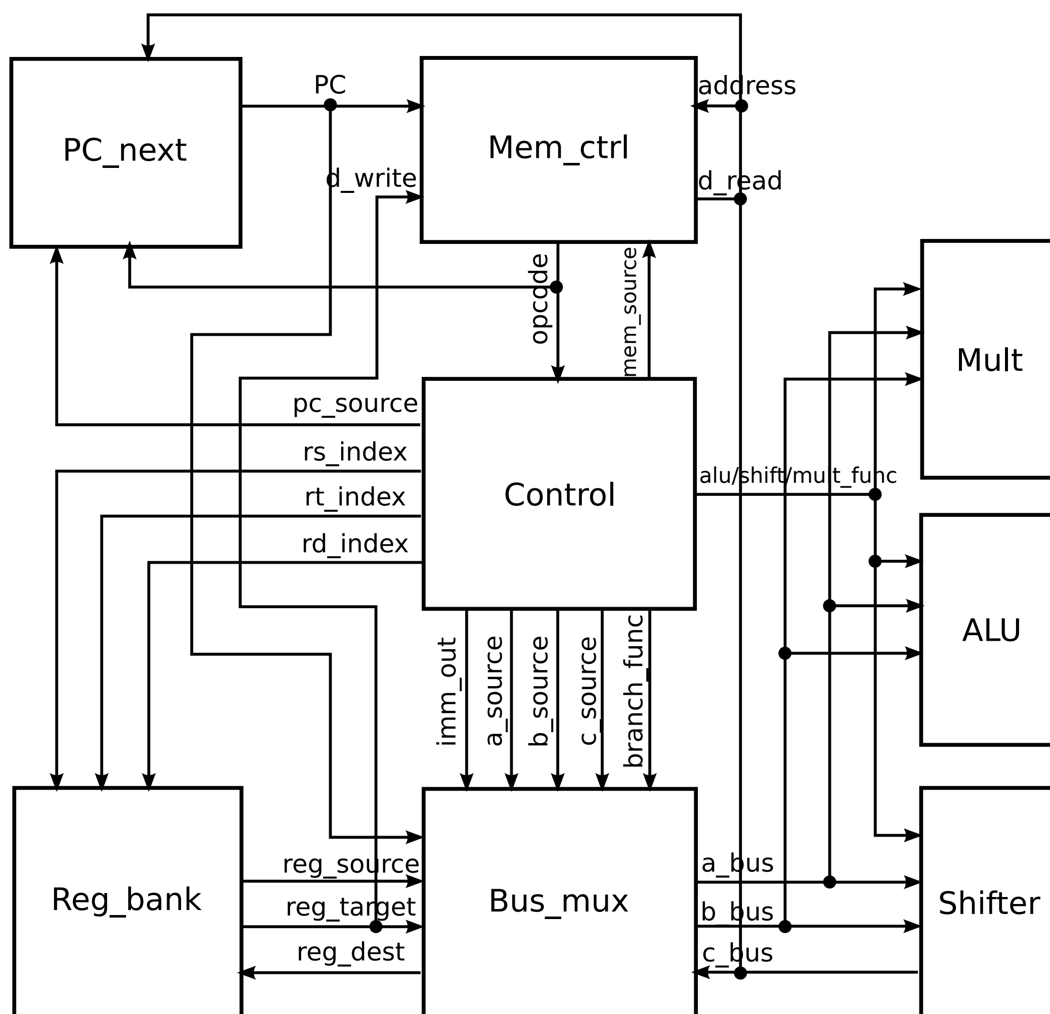


Figura 5.2: Diagrama de blocos do núcleo do IP Plasma, o mlite_cpu

Pipeline Dois estágios de *pipeline*. No primeiro ocorrem as operações de *fetch* e *decode*, já no segundo estágio as etapas de *execute* e *writeback*

ALU ALU com multiplicadores dedicados (utilizando blocos DSP do FPGA quando possível)

A figura 5.2, obtida em (PLASMA, 2001), é o diagrama de blocos do núcleo do Plasma, o *mlite_cpu*. No diagrama, pode-se notar a presença do controlador de memória, assim como do multiplicador dedicado. Também está presente um banco de registradores, com 32 registradores de 32 bits cada, implementados nas BRAMs do FPGA.

A escolha do Plasma como processador para integrar nosso cenário de validação foi motivada em grande parte pelo fato de já existir um porte razoavelmente estável do sistema operacional EPOS para a arquitetura MIPS. Também procuramos utilizar

um IP de código aberto e, de fato, precisamos dessa liberdade para implementar as modificações já discutidas.

Todos os componentes do EPOS têm sua implementação em software, armazenados em um bloco de memória com interface AXI4 e executando no mlite_cpu, com a única exceção do escalonador, que é um *slave* AXI4 por si próprio

5.2.1 Adaptador mlite_cpu – AXI4Lite

Para integrar o núcleo mlite_cpu aos componentes AXI4 restantes foi necessário o desenvolvimento de um bloco adaptador, que fosse responsável pela tradução bidirecional entre o protocolo nativo do mlite_cpu e o AMBA AXI4Lite. Envolvido por esse adaptador, o processador mlite_cpu torna-se um mestre no barramento AXI4. O diagrama 5.3 ilustra a máquina de estados finitos que realiza a tradução entre os protocolos.

Como pode se observar no diagrama da máquina de estados finitos, os sinais de dados e endereço em geral foram ligados diretamente (paralelamente) aos sinais correspondentes na interface AXI4Lite. Fazem parte das saídas da máquina de Moore apenas os sinais de *handshaking*.

A verificação funcional do adaptador foi realizada através de simulação em nível RTL, no simulador ModelSim®. Um simples programa de teste gerava um número de escritas na memória e em seguida tentava ler dos mesmos endereços onde escreveu. Este programa foi carregado num bloco de memória e um *testbench* em VHDL foi escrito para: instanciar processador e memória; conectá-los por meio do barramento e verificar se as escritas e leituras ocorriam corretamente.

O código C do programa de teste, assim como o *testbench* VHDL e o diagrama de forma de onda correspondentes à verificação do adaptador estão no apêndice C.

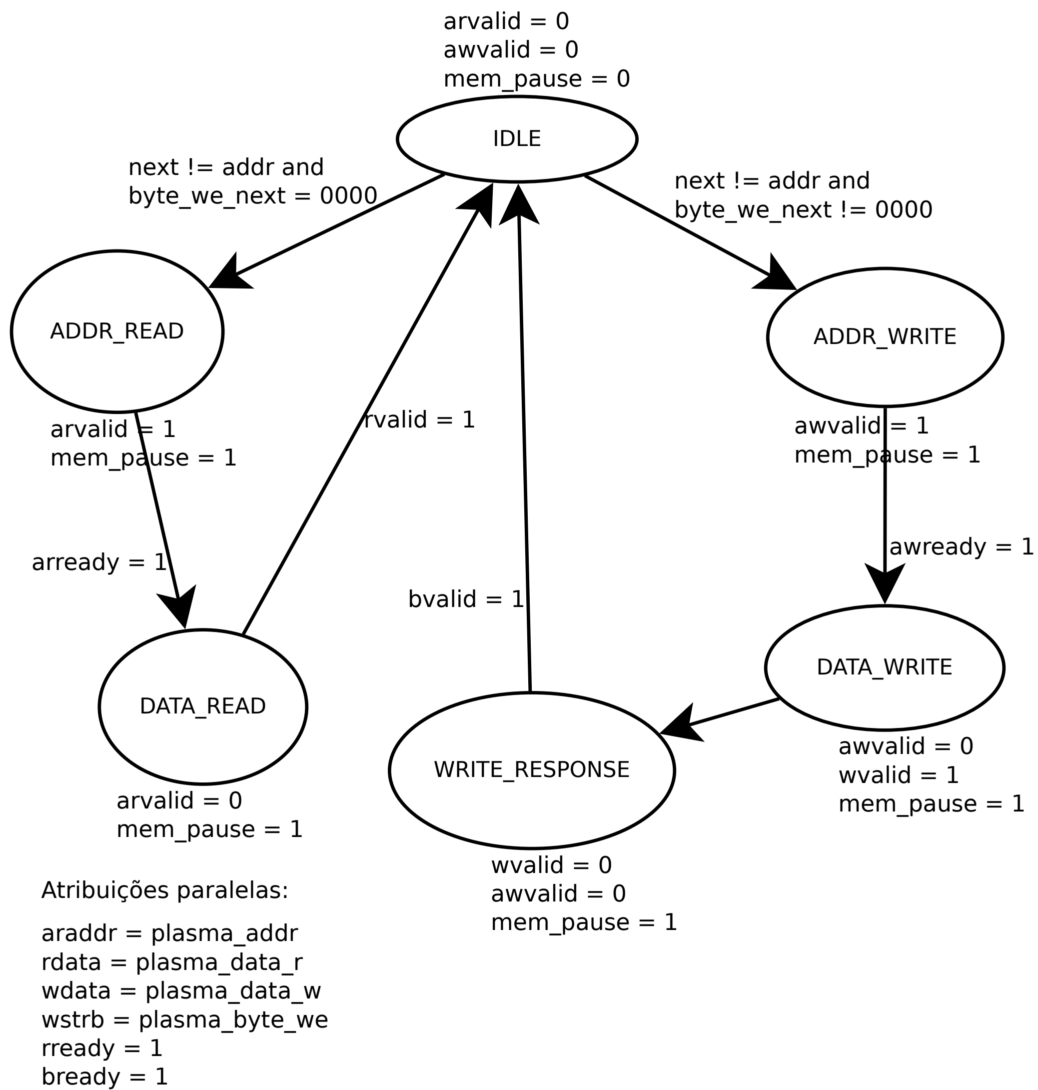


Figura 5.3: Máquina de Estados Finitos (Máquina de Moore) responsável pela tradução entre os protocolos nativo mlite_cpu e AXI4Lite

6 Resultados

> "Testes demonstram somente a *presença* de erros em um programa, jamais sua ausência."
>
> Edsger Wybe Dijkstra

O escalonador escrito em C++ foi sintetizado com várias combinações de diretivas e otimizações, buscando demonstrar uma importante vantagem da síntese de alto nível: a possibilidade de se obter várias microarquiteturas a partir de uma única descrição comportamental. A verificação funcional dos circuitos gerados foi feita com a execução de um testbench VHDL sobre os mesmos.

Nas próximas seções detalhamos o testbench de verificação e todos os cenários de síntese. São também exibidas e analisadas as métricas resultantes da síntese (tanto pré como pós-RTL). A avaliação de nossa implementação é feita em comparação com uma implementação de referência do escalonador em nível RT.

Em todos os cenários, o dispositivo-alvo para a síntese pós-RTL foi um FPGA da família Virtex6®, fabricado pela Xilinx®, conforme já detalhado na seção 5.

6.1 Resultados da verificação funcional

Todos os cenários de síntese escolhidos geram um bloco de hardware com a mesma interface e mesmo comportamento esperado. Assim, a execução de um único testbench sobre todos os blocos resultantes é suficiente para verificar sua corretude e equivalência.

O testbench desenvolvido executa uma séria de operações em sequência. A sequência de operações realizadas, com as respectivas entradas fornecidas e saídas esperadas, está detalhada na tabela 6.1

Operação	input	priority	status	return
SIZE	X	X	STAT_OK	0
CREATE	77	NORMAL	STAT_OK	0
INSERT	0	X	STAT_OK	X
SIZE	X	X	STAT_OK	1
CREATE	666	HIGH	STAT_OK	1
INSERT	1	X	STAT_OK	X
SIZE	X	X	STAT_OK	2
GET_ID	666	X	STAT_OK	1
CHOSEN	X	X	STAT_OK	1
GET_ID	77	X	STAT_OK	0
REMOVE	0	X	STAT_OK	X
SIZE	X	X	STAT_OK	1
DESTROY	0	X	STAT_OK	X
CREATE	42	NORMAL	STAT_OK	0

Tabela 6.1: Sequência dos casos de teste para a verificação funcional do escalonador proposto. Entradas com valor X são desconsideradas pelo circuito e saídas com valor X são irrelevantes (seu valor não importa para a corretude).

O código VHDL completo do testbench encontra-se no apêndice B.4

6.2 Cenários de síntese

Em geral, a escolha dos cenários de síntese para nossa proposta foi guiada pelo interesse em avaliar microarquiteturas que se situassem em extremos opostos do espectro de área/latência, ou seja, optamos por realizar a síntese de microarquiteturas que tivessem grande área e baixa latência e vice-versa. Uma descrição mais detalhada de cada um dos cenários de síntese segue:

Cenário 1 Neste cenário os arrays usados para armazenar os elementos da fila de escalonamento foram todos mapeados para memórias (RAMs) dedicadas no FPGA. Nenhuma paralelização de loops foi realizada. O script com as diretivas do cenário 1 constam do apêndice B.3.1

Cenário 2 Propositadamente, neste cenário de execução não houve qualquer intervenção do projetista no processo. Todas as diretivas de síntese foram mantidas em seus valores *default*, visando avaliar uma espécie de caso “otimista”, em que a ferramenta de síntese tem independência total para inferir a microarquitetura do bloco sendo sintetizado. Neste cenário os arrays acabaram sendo mapeados

para registradores e os loops também foram mantidos intactos (sem nenhuma paralelização). As diretivas de síntese para este cenário estão no apêndice B.3.2

Cenário 3 Cenário com paralelismo total. Todos os loops foram paralelizados, o que diminuiu drasticamente o número de ciclos necessários (no pior caso) para realizar uma operação do escalonador. Naturalmente, tal paralelização acarretou um aumento correspondente na área consumida e também no atraso máximo do bloco. As diretivas do cenário 3 estão no apêndice B.3.3

É importante ainda citar que em todos os cenários a síntese pós-RTL foi realizada objetivando-se uma frequência de clock de 50 MHz, portanto o limite máximo de atraso era de 20 ns. Caso algum cenário tivesse uma atraso que ultrapassasse tal limite, sua implementação seria inviável.

Algumas das mais importantes diretivas de síntese, comuns a todos os cenários, merecem comentários:

START_FLAG e DONE_FLAG incluem no bloco um sinal de entrada que indica que todas as entradas já foram fornecidas e a computação pode começar, bem como um sinal de saída, o qual quando ativado pelo bloco significa que suas saídas estão estáveis. Tais sinais facilitam bastante o processo de verificação funcional.

CLOCKS Essa diretiva especifica todos os parâmetros relativos ao clock do sistema, incluindo período, *duty cycle* e tolerância. Além disso essa diretivas também define o tipo (síncrono/assíncrono) e a polaridade do sinal de reset.

TECHLIBS Através dessa diretiva é especificada a *biblioteca* de componentes a ser usada no design. As informações de atraso e área dependentes de tecnologia são importantes na determinação de um escalonamento adequado para o design. Existem bibliotecas para os mais diversos modelos de FPGAs, de vários fabricantes.

DESIGN_GOAL Essa diretiva tem dois valores possíveis: *área* ou *latency*, e define a principal grandeza a ser minimizada pelos algoritmos de otimização nas várias fases da síntese.

6.3 Resultados de síntese

Dados de cada um dos três cenários definidos foram coletados em duas etapas do fluxo de projeto. A primeira delas, pré-RTL, corresponde ao resultado da síntese de alto nível, um arquivo VHDL que contém a descrição do bloco em nível de transferência de registradores. Já a segunda etapa, pós-RTL, corresponde ao resultado do mapeamento do modelo RTL para os blocos funcionais dispositivo FPGA utilizado. A tabela 6.2 mostra os resultados na etapa pré-RTL.

Cenário	Área ocupada	Ciclos por operação (pior caso)
Cenário 1	4301.698	79
Cenário 2	6496.576	43
Cenário 3	9847.783	4

Tabela 6.2: Dados de área e vazão pré-síntese RTL de nossa proposta, em alguns cenários de parâmetros. Tais dados foram reportados após a síntese de alto nível, sobre o modelo RTL gerado

A coluna “Área ocupada” se refere à pontuação de área dada pela ferramenta de síntese, sendo que tal pontuação leva em conta a quantidade de operadores de cada tipo usados no blocos, assim como o peso de cada um dos tipos de operadores.

É notável a diferença entre a quantidade de ciclos necessários por operação nos cenários 1 e 2. Essa diferença pode ser explicada pelo uso de uma memória para o armazenamento, resultando em um gargalo no sistema e impedindo a leitura paralela de dados que não possuem dependências entre si. Já no cenário 3 o consumo de área aumenta em mais de 2 vezes, porém o número de ciclos necessários por operação diminui em proporção bem maior. De fato, os 4 ciclos são provenientes sobretudo do bloco de controle do sistema (máquina de estados finitos).

A tabela 6.3, a seguir, mostra a utilização de recursos e os atrasos máximos para cada cenário na etapa de síntese pós-RTL.

Salienta-se a utilização de blocos de memória RAM da FPGA no cenário 1. Com a utilização de memórias, o número de LUTs[1] ocupadas caiu significativamente.

Outro ponto a ser destacado é que a diferença entre o número de LUTs ocupadas nos cenários 2 e 3 chega a quase 100%. No entanto, quando se compara as porcen-

[1]LUT: Look-Up Table, o bloco funcional mais comum (numeroso) em FPGAs

Cenário	LUTs	LUTs (% do FPGA)	RAMs	Atraso máximo
Cenário 1	1654	1.10%	26	6.672 ns
Cenário 2	3392	2.25%	0	7.341 ns
Cenário 3	5121	3.40%	0	10.597 ns

Tabela 6.3: Dados de área (consumo de recursos) e atraso máximo de nossa proposta. Os dados foram obtidos após a síntese RTL do escalonador, com a netlist já mapeada para o FPGA utilizado

tagens de utilização do FPGA, a diferença já não é tão significativa (2.25% vs. 3.40%). Isso nos leva a crer que em um futuro ainda próximo, à medida que os FPGAs ganharem mais blocos funcionais, a síntese de alto nível se tornará cada vez mais viável em termos de área.

Como última observação sobre a etapa de síntese pós-RTL, gostaríamos de chamar a atenção para os atrasos máximos obtidos. Mesmo no cenário 3 o atraso máximo previsto ainda é bem menor do que nosso limite superior (20 ns), e portanto a frequência de clock do bloco poderia ser maior, caso fosse necessário.

A comparação com a implementação RTL de referência foi realizada (como já mencionado) somente na etapa de síntese pós-RTL. O mesmo dispositivo-alvo e os mesmos níveis de otimização foram utilizados em todos os cenários. A tabela 6.4 demonstra essa comparação.

Cenário	LUTs	LUTs (% do FPGA)	Atraso máximo
RTL	1250	0.83%	5.598 ns
Cenário 1	1654	1.10%	6.672 ns
Cenário 2	3392	2.25%	7.341 ns
Cenário 3	5121	3.40%	10.597 ns

Tabela 6.4: Comparação entre dados de área (consumo de recursos) e atraso máximo entre nossa proposta e a implementação RTL de referência

7 Conclusões

"Otimização prematura é a raiz de todo o mal"

Donald Knuth, criador do LaTeX, sistema usado para formatar este relatório

O objetivo principal deste trabalho foi avaliar a viabilidade de se utilizar síntese de alto nível para a implementação de algoritmos de propósito geral em hardware reconfigurável. Tendo em vista os resultados alcançados podemos dizer que, com uma escolha razoável de configurações, uma ferramenta de síntese de alto nível pode chegar a resultados bastante próximos dos alcançados por um experiente projetista de hardware.

Apesar de, em termos absolutos, a solução sintetizada automaticamente ter ocupado uma área bastante maior (cerca de 30% de acréscimo), em termos de porcentagem de ocupação do dispositivo FPGA a diferença não foi significativa. Cremos que essa é uma tendência, e que com o aumento do poder computacional dos FPGAs a síntese de alto nível se tornará de fato cada vez mais viável.

Nosso estudo de caso, um escalonador de recursos, normalmente seria considerado um mau candidato a beneficiar-se de síntese de alto nível, pois é relativamente pobre em computação e possui um estado interno significativo. Mesmo assim conseguimos demonstrar que, com uma boa engenharia de domínio e separação de responsabilidades entre os componentes de um sistema, é possível se ter código genérico suficiente a ponto de executar com poucas modificações tanto em software quanto em hardware.

O principal argumento que esperamos ter ajudado a sustentar neste trabalho é de que as técnicas e ferramentas de síntese de alto nível são um passo largo na direção de *separar* o *comportamento* de um algoritmo dos aspectos relativos ao seu *cenário*

de implementação (hardware ou software). Não procuramos alegar, de maneira alguma, que o conhecimento de arquitetura de hardware fica desnecessário com o uso de ferramentas de síntese de alto nível. Usando tais ferramentas, porém, o projetista evita que preocupações relativas a desempenho venham afetar a sua *modelagem* do algoritmo sendo implementado. Com a síntese de alto nível, o projetista consegue mais facilmente se afastar do abismo já profetizado por Donald Knuth em sua célebre frase:

> "Otimização prematura é a raiz de todo o mal"

Nas seções seguintes procuramos destacar as principais contribuições técnico-científicas do trabalho e enumerar alguns dos pontos que poderiam ser explorados em trabalhos futuros.

7.1 Principais contribuições

Compreendemos que a modelagem do problema estudado e a utilidade dos artefatos desenvolvidos não se restringe ao algoritmo que desenvolvemos (um escalonador). Em particular, ressaltamos as seguintes principais contribuições deste trabalho:

Camadas de adaptação de um objeto para implementação em hardware Em nosso desenvolvimento do escalonador proposto, fomos guiados sobretudo pelas restrições da ferramenta de síntese de alto nível. Essas restrições nos levaram a modelar uma arquitetura *em camadas*, que encapsula um objeto, responsabilizando-se pelo gerenciamento de recursos necessários à sua execução e pela comunicação com o mundo externo. Os componentes descritos nas seções 4.4.1, 4.4.2 e 4.4.3 são genéricos, e podem encapsular outras classes de objetos além do escalonador.

Adaptador Plasma / AMBA Como ambiente de execução do escalonador desenvolvido, escolhemos o processador *softcore* Plasma MIPS. O principal motivo dessa escolha foi o fato de que o Plasma MIPS é um processador de código aberto, livremente modificável. Já nossa escolha de barramento para o SoC onde o escalonador executaria recaiu sobre a família AMBA®, da fabricante ARM®. Como o Plasma MIPS não possuía interface AMBA®, nós desenvolvemos um

componente adaptador, o qual torna um processador Plasma mestre em um barramento AXI4Lite (barramento AMBA®simplificado). Este adaptador foi desenvolvido em linguagem VHDL e seu código-fonte, documentação e testes foram publicados no repositório de hardware livre *OpenCores*[1].

7.2 Trabalhos futuros

Como principais trabalhos futuros que poderiam explorar avanços na mesma temática deste trabalho ressaltamos:

- Fazer do escalonador proposto um protótipo de componente de SO efetivamente híbrido, segundo a visão de longo prazo do EPOS, como descrita em Marcondes e Fröhlich (2009). Isto é, através de alguns avanços no sistema de compilação do EPOS toda a camada de adaptação para hardware deveria ser acrescentada a um componente, sempre que fosse escolhido pelo usuário sua implementação em hardware.
- Modelar outras estruturas de dados e algoritmos em hardware, utilizando-se também de síntese de alto nível. Seria particularmente interessante uma pesquisa sobre estruturas de dados paralelas, sua descrição em alto nível de abstração e implementação em hardware.
- Partir das contribuições deste trabalho em direção a um protocolo de comunicação entre objetos independentes de implementação. Objetos em hardware teriam um protocolo padronizado para troca de mensagens entre si, assim como com objetos em software. De certa forma, essa linha de trabalho está relacionada à temática de invocação remota e objetos distribuídos.

[1] `http://www.opencores.org`

Referências Bibliográficas

ANDREWS, D. et al. Programming models for hybrid FPGA-CPU computational components: a missing link. *IEEE MICRO*, Citeseer, v. 24, n. 4, p. 42–53, 2004.

ARM. *AMBA AXI Protocol Specification v2.0*. 2010. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022c/index.html.

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.

BLELLOCH, G. Programming parallel algorithms. *Communications of the ACM*, ACM, v. 39, n. 3, p. 85–97, 1996.

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.

COUSSY, P.; MORAWIEC, A. *High-Level Synthesis: from Algorithm to Digital Circuit*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008. ISBN 1402085877, 9781402085871.

FINGEROFF, M. *High-Level Synthesis Blue Book*. [S.l.]: Xlibris Corporation, 2010. ISBN 1450097243.

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

FRÖHLICH, A. A comprehensive approach to power management in embedded systems. 2011.

FRÖHLICH, A.; WANNER, L. Operating system support for wireless sensor networks. *Journal of Computer Science*, Citeseer, v. 4, n. 4, p. 272–281, 2008.

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>.

HÜRSCH, W.; LOPES, C. Separation of concerns. Citeseer, 1995.

IEEE. Ieee standard for verilog register transfer level synthesis. *IEEE Std 1364.1-2002*, p. 1–100, 2002.

IEEE. 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.

KURODA, T. Cmos design challenges to power wall. In: IEEE. *Microprocesses and Nanotechnology Conference, 2001 International*. [S.l.], 2001. p. 6–7.

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.

MAN, H. et al. Cathedral-ii: A silicon compiler for digital signal processing. *IEEE Des. Test*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 3, p. 13–25, November 1986. ISSN 0740-7475. Disponível em: <http://portal.acm.org/citation-.cfm?id=1304057.1304461>.

MARCONDES, H.; FRÖHLICH, A. A. A hybrid hardware and software component architecture for embedded system design. *Analysis, Architectures and Modelling of Embedded Systems*, Springer, p. 259–270, 2009.

MENTOR. *Catapult-C*. 2011. http://www.mentor.com/esl/catapult/overview.

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, 1965. ISSN 0018-9219.

MÜCK, T. R. An aop-based approach for embedded system design. 2011.

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.

OSCI. *IEEE 1666: SystemC Language Reference Manual, 2005*. 2005.

PLASMA. *Plasma - most MIPS I opcodes*. 2001. http://opencores.org/project,plasma.

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.

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.

APÊNDICE A – Lista duplamente encadeada em hardware

A.1 Código-fonte C++ – Template

```
#ifndef __list_h
#define __list_h

#include "../../maybe.h"

// List Element Rank (for ordered lists)
class List_Element_Rank
{
public:
    List_Element_Rank(int r = 0): _rank(r) {}

    operator int() const { return _rank; }

protected:
    int _rank;
};

// List Elements
namespace List_Elements
{
    typedef List_Element_Rank Rank;

    // Vector Element
    template <typename T>
    class Pointer
    {
    public:
        typedef T Object_Type;
        typedef Pointer Element;
```

```
public:
    Pointer(const T * o): _object(o) {}

    T * object() const { return const_cast<T *>(_object); }

private:
    const T * _object;
};

// List Element
template <typename T>
class Doubly_Linked
{
public:
    typedef T Object_Type;
    typedef Doubly_Linked Element;

public:
    Doubly_Linked(const T * o): _object(o), _prev(0), _next(0) {}

    T * object() const { return const_cast<T *>(_object); }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }

private:
    const T * _object;
    Element * _prev;
    Element * _next;
};

// Ordered List Element
template <typename T, typename R = Rank>
class Doubly_Linked_Ordered
{
public:
    typedef T Object_Type;
    typedef Rank Rank_Type;
    typedef Doubly_Linked_Ordered Element;
```

```
public:
    Doubly_Linked_Ordered(const T * o,  const R & r = 0):
        _object(o), _rank(r), _prev(0), _next(0) {}

    T * object() const { return const_cast<T *>(_object); }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }

    const R & rank() const { return _rank; }
    void rank(const R & r) { _rank = r; }
    int promote(const R & n = 1) { _rank -= n; return _rank; }
    int demote(const R & n = 1) { _rank += n; return _rank; }

private:
    const T * _object;
     R _rank;
    Element * _prev;
    Element * _next;
};

// Scheduling List Element
template <typename T, typename R = Rank>
class Doubly_Linked_Scheduling
{
public:
    typedef T Object_Type;
    typedef Rank Rank_Type;
    typedef Doubly_Linked_Scheduling Element;

public:
    Doubly_Linked_Scheduling(const T * o,  const R & r = 0):
        _object(o), _rank(r), _prev(0), _next(0) {}

    T * object() const { return const_cast<T *>(_object); }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }
```

```
        const R & rank() const { return _rank; }
        void rank(const R & r) { _rank = r; }
        int promote(const R & n = 1) { _rank -= n; return _rank; }
        int demote(const R & n = 1) { _rank += n; return _rank; }

    private:
        const T * _object;
         R _rank;
        Element * _prev;
        Element * _next;
    };
};


// List Iterators
namespace List_Iterators
{
    // Bidireacional Iterator (for doubly linked lists)
    template<typename El>
    class Bidirecional
    {
    private:
        typedef Bidirecional<El> Iterator;

    public:
        typedef El Element;

    public:
        Bidirecional(): _current(0) {}
        Bidirecional(Element * e): _current(e) {}

        operator Element *() const { return _current; }

        Element & operator*() const { return *_current; }
        Element * operator->() const { return _current; }

        Iterator & operator++() {
            _current = _current->next(); return *this;
        }
        Iterator operator++(int) {
            Iterator tmp = *this; ++*this; return tmp;
        }
```

```
        Iterator & operator--() {
            _current = _current->prev(); return *this;
        }
        Iterator operator--(int) {
            Iterator tmp = *this; --*this; return tmp;
        }

        bool operator==(const Iterator & i) const {
            return _current == i.current;
        }
        bool operator!=(const Iterator & i) const {
            return _current != i._current;
        }

    private:
        Element * _current;
    };
}


// Doubly-Linked List
template <typename T,
          typename El = List_Elements::Doubly_Linked<T> >
class List
{
public:
    typedef T Object_Type;
    typedef El Element;
    typedef List_Iterators::Bidirecional<El> Iterator;

public:
    List(): _size(0), _head(0), _tail(0) {}

    bool empty() const { return (_size == 0); }
    unsigned int size() const { return _size; }

    Maybe<Element*> head() { return Maybe<Element*>(_head, _head != 0); }

    Maybe<Element*> tail() { return Maybe<Element*>(_tail, _tail != 0); }

    Iterator begin() { return Iterator(_head); }
    Iterator end() { return Iterator(_tail->next()); }
```

```
void insert(Element * e) { insert_tail(e); }

void insert_head(Element * e) {
    if(empty())
        insert_first(e);
    else {
        e->prev(0);
        e->next(_head);
        _head->prev(e);
        _head = e;
        _size++;
    }
}

void insert_tail(Element * e) {
    if(empty())
        insert_first(e);
    else {
        _tail->next(e);
        e->prev(_tail);
        e->next(0);
        _tail = e;
        _size++;
    }
}

Element * remove() { return remove_head(); }

Element * remove(Element * e) {
    if(last())
        remove_last();
    else if(!e->prev())
        remove_head();
    else if(!e->next())
        remove_tail();
    else {
        e->prev()->next(e->next());
        e->next()->prev(e->prev());
        _size--;
    }

    return e;
}
```

```
    Element * remove_head() {
        if(empty())
            return 0;
        if(last())
            return remove_last();
        Element * e = _head;
        _head = _head->next();
        _head->prev(0);
        _size--;

        return e;
    }

    Element * remove_tail() {
        if(empty())
            return 0;
        if(last())
            return remove_last();
        Element * e = _tail;
        _tail = _tail->prev();
        _tail->next(0);
        _size--;

        return e;
    }

    Element * remove(const Object_Type& obj) {
        Element * e = search(obj);
        if(e)
            return remove(e);
        return 0;
    }

    Element * search(const Object_Type& obj) {
        Element * e = _head;
        for(; e && (e->object() != obj); e = e->next());
        return e;
    }

protected:
    bool last() const { return (_size == 1); }
```

```
    void insert(Element * e, Element * p,  Element * n) {
        p->next(e);
        n->prev(e);
        e->prev(p);
        e->next(n);
        _size++;
    }

    void insert_first(Element * e) {
        e->prev(0);
        e->next(0);
        _head = e;
        _tail = e;
        _size++;
    }

    Element * remove_last() {
        Element * e = _head;
        _head = 0;
        _tail = 0;
        _size--;

        return e;
    }

private:
    unsigned int _size;
    Element * _head;
    Element * _tail;
};

#endif
```

A.2 Código-fonte C++ – Wrapper de instanciação

A.2.1 Cabeçalho

```
#ifndef __LIST_INT_HW__
#define __LIST_INT_HW__

#include "list.h"
```

```
#define LISTSIZE 10

template <unsigned int Q> class List_int_HW;

template <typename T>
class Doubly_Linked_Value
{
    template <unsigned int Q>
    friend class List_int_HW;
public:
    typedef T Object_Type;
    typedef Doubly_Linked_Value Element;

public:
    Doubly_Linked_Value():
        _object(T()), _prev(0), _next(0), _used(false) {}

    Doubly_Linked_Value(T o):
        _object(o), _prev(0), _next(0), _used(true) {}

    T object() const { return _object; }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }

private:
    T _object;
    Element * _prev;
    Element * _next;

    bool _used;
};



template <unsigned int Q>
class List_int_HW {
public:
    typedef Doubly_Linked_Value<int> Element;
    typedef List<int, Element> WrappedType;
```

```
    enum MethodId {
        EMPTY = 0,          SIZE = 1,
        HEAD = 2,           TAIL = 3,
        INSERT_HEAD = 4, INSERT_TAIL = 5,
        REMOVE_HEAD = 6, REMOVE = 7,
        REMOVE_TAIL = 8
    };

private:
    typedef Maybe<int> M;
    static Element _storage[Q];

public:
    #pragma hls_design top
    static M call(MethodId m, M input) {
        Element* pointer = (Element*) 0;

        M return_value;
        Maybe<Element*> p;

        static WrappedType _l;

        switch(m) {
        case EMPTY:
            return_value = Maybe<int>(_l.empty()); break;
        case SIZE:
            return_value = Maybe<int>(_l.size()); break;
        case HEAD:
            p = _l.head();
            return_value = p.exists() ?
                p.get(_storage)->object() : Maybe<int>();
            break;
        case TAIL:
            p = _l.tail();
            return_value = p.exists() ?
                p.get(_storage)->object() : Maybe<int>();
            break;
        case INSERT_HEAD:
            pointer = allocate();
            if(pointer != 0) {
                *pointer = Element(input.get());
                _l.insert_head(pointer);
            }
```

```
            break;
        case INSERT_TAIL:
            pointer = allocate();
            if(pointer != 0) {
                *pointer = Element(input.get());
                _l.insert_tail(pointer);
            }
            break;
        case REMOVE_HEAD:
            pointer = _l.remove_head();
            if(pointer != 0) {
                free(pointer);
                return_value = Maybe<int>(pointer->object());
            } else {
                return_value = Maybe<int>();
            }
            break;
        case REMOVE:
            pointer = _l.remove(input.get());
            if(pointer != 0) {
                free(pointer);
                return_value = pointer->object();
            } else {
                return_value = Maybe<int>();
            }
            break;
        case REMOVE_TAIL:
            pointer = _l.remove_tail();
            if(pointer != 0) {
                free(pointer);
                return_value = Maybe<int>(pointer->object());
            } else {
                return_value = Maybe<int>();
            }
            break;
        default:
            return_value = Maybe<int>(); break;
        }

        return return_value;
    }

private:
```

```
    static void free(Element* p) {
        p->_used = false;
    }

    static Element* allocate() {
        Element* position = 0;

        alloc_search: for(int i = 0; i < Q; ++i)
            if(!_storage[i]._used) {
                _storage[i]._used = true;
                position = _storage + i;
                break;
            }

        return position;
    }

};

#endif /* __LIST_INT_HW__ */
```

A.2.2 Definições

```
#include "List_int_HW.h"

template<unsigned int Q>
typename List_int_HW<Q>::Element List_int_HW<Q>::_storage[Q];

template class List_int_HW<LISTSIZE>;
```

A.3 Testbench VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.conv_std_logic_vector;
use ieee.numeric_std.all;
library std;
use std.env.all;

entity epos_linked_list_testbench is
end epos_linked_list_testbench;

architecture Behavioral of epos_linked_list_testbench is
```

```
-- clock period
constant CLKPERIOD : time := 10 ns;

-- Component declarations
component List_int_HW_10U_call is
    PORT(
        start : IN STD_LOGIC;
        done : OUT STD_LOGIC;
        m_rsc_z : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        input_thing_rsc_z : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        input_exists_rsc_z : IN STD_LOGIC;
        List_int_HW_10U_call_return_thing_rsc_z :
            OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        List_int_HW_10U_call_return_exists_rsc_z :
            OUT STD_LOGIC;
        clk : IN STD_LOGIC;
        arst_n : IN STD_LOGIC
    );
end component;

-- signal declarations
signal sig_clk   : std_logic;
signal sig_reset : std_logic;

signal sig_start              : std_logic;
signal sig_done               : std_logic;
signal sig_m_rsc_z            : std_logic_vector(31 downto 0);
signal sig_input_thing_rsc_z  : std_logic_vector(31 downto 0);
signal sig_input_exists_rsc_z : std_logic;
signal sig_return_thing_rsc_z : std_logic_vector(31 downto 0);
signal sig_return_exists_rsc_z : std_logic;

constant OP_EMPTY       : std_logic_vector(31 downto 0)
    := conv_std_logic_vector(0, 32);
constant OP_SIZE        : std_logic_vector(31 downto 0)
    := conv_std_logic_vector(1, 32);
constant OP_HEAD        : std_logic_vector(31 downto 0)
    := conv_std_logic_vector(2, 32);
constant OP_TAIL        : std_logic_vector(31 downto 0)
    := conv_std_logic_vector(3, 32);
constant OP_INSERT_HEAD : std_logic_vector(31 downto 0)
    := conv_std_logic_vector(4, 32);
```

```
    constant OP_INSERT_TAIL  : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(5, 32);
    constant OP_REMOVE_HEAD  : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(6, 32);
    constant OP_REMOVE       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(7, 32);
    constant OP_REMOVE_TAIL  : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(8, 32);

begin

    clock: process
    begin
        sig_clk <= '1';
        wait for (CLKPERIOD/2);
        sig_clk <= '0';
        wait for (CLKPERIOD/2);
    end process;

    -- component instantiations
    list: List_int_HW_10U_call
        port map (
            start                                   => sig_start,
            done                                    => sig_done,
            m_rsc_z                                 => sig_m_rsc_z,
            input_thing_rsc_z                       => sig_input_thing_rsc_z,
            input_exists_rsc_z                      => sig_input_exists_rsc_z,
            List_int_HW_10U_call_return_thing_rsc_z  => sig_return_thing_rsc_z,
            List_int_HW_10U_call_return_exists_rsc_z => sig_return_exists_rsc_z,
            clk                                     => sig_clk,
            arst_n                                  => sig_reset);


    testbench: process
        -- test procedures, one for each test case
        procedure testEmptyHead is
        begin
            sig_start              <= '1';
            sig_m_rsc_z            <= OP_HEAD;
            sig_input_exists_rsc_z <= '0';

            wait until rising_edge(sig_done);
            sig_start <= '0';
```

```
    wait for CLKPERIOD;

    assert (sig_return_exists_rsc_z = '0')
    report
        "testEmptyHead" & LF &
        "expected:␣"    & integer'image( 0 ) & LF &
        "actual:␣"      & std_logic'image( sig_return_exists_rsc_z ) & LF
    severity FAILURE;
end procedure testEmptyHead;

procedure testFirstInsertion is
begin
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_INSERT_HEAD;
    sig_input_exists_rsc_z <= '1';
    sig_input_thing_rsc_z  <= conv_std_logic_vector(77, 32);

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
end procedure testFirstInsertion;

procedure testGetHead is
begin
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_HEAD;
    sig_input_exists_rsc_z <= '0';

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;

    assert (to_integer(signed(sig_return_thing_rsc_z)) = 77)
    report
        "testGetHead" & LF &
        "expected:␣" & integer'image( 77 ) & LF &
        "actual:␣"   &
        integer'image( to_integer(signed(sig_return_thing_rsc_z)) ) & LF
    severity FAILURE;
end procedure testGetHead;

procedure testSecondInsertion is
begin
```

```
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_INSERT_TAIL;
    sig_input_exists_rsc_z <= '1';
    sig_input_thing_rsc_z  <= conv_std_logic_vector(42, 32);

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
end procedure testSecondInsertion;

procedure testThirdInsertion is
begin
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_INSERT_TAIL;
    sig_input_exists_rsc_z <= '1';
    sig_input_thing_rsc_z  <= conv_std_logic_vector(31, 32);

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
end procedure testThirdInsertion;

procedure testGetTail is
begin
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_TAIL;
    sig_input_exists_rsc_z <= '0';

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;

    assert (to_integer(signed(sig_return_thing_rsc_z)) = 31)
    report
        "testGetTail" & LF &
        "expected:␣"    & integer'image( 31 ) & LF &
        "actual:␣"      &
        integer'image( to_integer(signed(sig_return_thing_rsc_z)) ) & LF
    severity FAILURE;
end procedure testGetTail;

procedure testFirstRemoval is
begin
```

```
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_REMOVE_HEAD;
    sig_input_exists_rsc_z <= '0';

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;

    assert (to_integer(signed(sig_return_thing_rsc_z)) = 77)
    report
        "testFirstRemoval" & LF &
        "expected: "       & integer'image( 77 ) & LF &
        "actual: "         &
        integer'image( to_integer(signed(sig_return_thing_rsc_z)) ) & LF
    severity FAILURE;
end procedure testFirstRemoval;

procedure testSecondRemoval is
begin
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_REMOVE_HEAD;
    sig_input_exists_rsc_z <= '0';

    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;

    assert (to_integer(signed(sig_return_thing_rsc_z)) = 42)
    report
        "testSecondRemoval" & LF &
        "expected: "        & integer'image( 42 ) & LF &
        "actual: "          &
        integer'image( to_integer(signed(sig_return_thing_rsc_z)) ) & LF
    severity FAILURE;
end procedure testSecondRemoval;

procedure testThirdRemoval is
begin
    sig_start              <= '1';
    sig_m_rsc_z            <= OP_REMOVE_HEAD;
    sig_input_exists_rsc_z <= '0';

    wait until rising_edge(sig_done);
```

```
            sig_start <= '0';
            wait for CLKPERIOD;

            assert (to_integer(signed(sig_return_thing_rsc_z)) = 31)
            report
                "testThirdRemoval" & LF &
                "expected:␣"       & integer'image( 31 ) & LF &
                "actual:␣"         &
                integer'image( to_integer(signed(sig_return_thing_rsc_z)) ) & LF
            severity FAILURE;
        end procedure testThirdRemoval;

    begin
        -- ACTIVE-HIGH reset
        sig_reset <= '1';
        wait for 60 ns;
        sig_reset <= '0';

        -- call the test procedures here
        testEmptyHead; wait for CLKPERIOD*3;
        testFirstInsertion; wait for CLKPERIOD*3;
        testGetHead; wait for CLKPERIOD*3;
        testSecondInsertion; wait for CLKPERIOD*3;
        testThirdInsertion; wait for CLKPERIOD*3;
        testGetTail; wait for CLKPERIOD*3;
        testFirstRemoval; wait for CLKPERIOD*3;
        testSecondRemoval; wait for CLKPERIOD*3;
        testThirdRemoval; wait for CLKPERIOD*10;

        finish(0);
    end process;

end Behavioral;
```

A.4 Diretivas da síntese de alto nível

```
project new
project set -name EPOSMaybeList

solution file add ../src/List_int_HW.cpp -type C++

directive set -REGISTER_IDLE_SIGNAL false
directive set -IDLE_SIGNAL {}
```

```
directive set -TRANSACTION_DONE_SIGNAL false
directive set -DONE_FLAG done
directive set -START_FLAG start
directive set -FSM_ENCODING none
directive set -REG_MAX_FANOUT 0
directive set -NO_X_ASSIGNMENTS true
directive set -SAFE_FSM false
directive set -RESET_CLEARS_ALL_REGS true
directive set -ASSIGN_OVERHEAD 0
directive set -DESIGN_GOAL area
directive set -OLD_SCHED false
directive set -PIPELINE_RAMP_UP true
directive set -COMPGRADE fast
directive set -SPECULATE true
directive set -MERGEABLE true
directive set -REGISTER_THRESHOLD 256
directive set -MEM_MAP_THRESHOLD 32
directive set -UNROLL no
directive set -CLOCK_OVERHEAD 20.000000
directive set -OPT_CONST_MULTS -1
go analyze

directive set -CLOCK_NAME clk
directive set -CLOCKS {clk {-CLOCK_PERIOD 10.0 -CLOCK_EDGE rising
        -CLOCK_UNCERTAINTY 0.0 -CLOCK_HIGH_TIME 5.0 -RESET_SYNC_NAME rst
        -RESET_ASYNC_NAME arst_n -RESET_KIND async -RESET_SYNC_ACTIVE high
        -RESET_ASYNC_ACTIVE high -ENABLE_NAME {} -ENABLE_ACTIVE high}}
directive set -TECHLIBS {{Xilinx_accel_VIRTEX-6-1.lib Xilinx_accel_VIRTEX-6-1}
    {mgc_Xilinx-VIRTEX-6-1_beh_psr.lib
    {{mgc_Xilinx-VIRTEX-6-1_beh_psr part 6VLX75TFF484}}}
    {ram_Xilinx-VIRTEX-6-1_RAMSB.lib ram_Xilinx-VIRTEX-6-1_RAMSB}}

go compile
go architect
go extract
```

APÊNDICE B – Escalonador implementado em hardware

B.1 Gereciador de alocação

```
#ifndef __SCHEDULER_PREALLOC__
#define __SCHEDULER_PREALLOC__

#include "scheduler.h"
#include "value_based_elements.h"
#include "maybe.h"

template<typename Obj_Type, typename Idx, Idx Max, typename Status = char>
class Scheduler_PreAlloc {
private:
    typedef typename Obj_Type::Criterion Criterion;
    typedef Doubly_Linked_Scheduling_Value<Obj_Type, Criterion> Elem;
    typedef Scheduling_Queue<Obj_Type, 1, Elem> Queue;

public:
    enum {
        STAT_OK    = 0x01,
        STAT_ERROR = 0x02
    };

public:
    Scheduler_PreAlloc()
        : elements(), rogue(&elements[Max]), alloc_bitmap(), queue() {}
    // NO BOUNDS CHECKING IN THE STORAGE

    Obj_Type chosen(Status& status) {
        Maybe<Elem*> e = queue.chosen();
        if(e.exists()) status = STAT_OK;
        else status = STAT_ERROR;
```

```
    return e.get(rogue)->object();
}

Idx create(Obj_Type obj, Criterion criterion, Status& status) {
    Idx index = allocate();
    if(index < Max) {
        status = STAT_OK;
        elements[index] = Elem(obj, criterion);
    } else
        status = STAT_ERROR;
    return index;
}

void insert(Idx index, Status& status) {
    status = STAT_OK;
    queue.insert(&elements[index]);
}

void destroy(Idx index, Status& status) {
    status = STAT_OK;
    free(index);
}

void remove(Idx index, Status& status) {
    Maybe<Elem*> e = queue.remove(&elements[index]);
    if(e.exists()) status = STAT_OK;
    else status = STAT_ERROR;
}

unsigned short size(Status& status) {
    status = STAT_OK;
    return queue.size();
}

Idx get_id(Obj_Type obj, Status& status) {
    get_id_search:
    for(Idx i = Idx(); i < Max; ++i)
        if(elements[i].object() == obj) {
            status = STAT_OK;
            return i;
        }
    status = STAT_ERROR;
    return Max;
```

```
    }

private:
    inline Idx allocate() {
        alloc_search:
        for(Idx i = Idx(); i < Max; ++i)
            if(alloc_bitmap[i] == false) {
                alloc_bitmap[i] = true;
                return i;
            }
        return Max;
    }

    inline void free(Idx i) {
        alloc_bitmap[i] = false;
    }

private:
    Elem elements[Max+1];
    Elem* rogue;
    bool alloc_bitmap[Max];

    Queue queue;
};

#endif /* __SCHEDULER_PREALLOC__ */
```

B.2 Wrapper de chamadas

B.3 Diretivas de síntese

B.3.1 Cenário 1

```
project new
project set -name EPOSScheduler

solution file add ../../../src/scheduler_hw.cpp -type C++

directive set -REGISTER_IDLE_SIGNAL false
directive set -IDLE_SIGNAL {}
directive set -TRANSACTION_DONE_SIGNAL false
directive set -DONE_FLAG done
```

```
directive set -START_FLAG start
directive set -FSM_ENCODING none
directive set -REG_MAX_FANOUT 0
directive set -NO_X_ASSIGNMENTS true
directive set -SAFE_FSM false
directive set -RESET_CLEARS_ALL_REGS true
directive set -ASSIGN_OVERHEAD 0
directive set -DESIGN_GOAL area
directive set -OLD_SCHED false
directive set -PIPELINE_RAMP_UP true
directive set -COMPGRADE fast
directive set -SPECULATE true
directive set -MERGEABLE true
directive set -REGISTER_THRESHOLD 256
directive set -MEM_MAP_THRESHOLD 32
directive set -UNROLL no
directive set -CLOCK_OVERHEAD 20.000000
directive set -OPT_CONST_MULTS -1
go analyze

directive set -CLOCK_NAME clk

directive set -CLOCKS {clk {-CLOCK_PERIOD 20.0 -CLOCK_EDGE rising
        -CLOCK_UNCERTAINTY 0.0 -CLOCK_HIGH_TIME 10.0 -RESET_SYNC_NAME rst
        -RESET_ASYNC_NAME arst_n -RESET_KIND async
        -RESET_SYNC_ACTIVE high -RESET_ASYNC_ACTIVE high -ENABLE_NAME {}
        -ENABLE_ACTIVE high}}

directive set -TECHLIBS {{Xilinx_accel_VIRTEX-6-1.lib Xilinx_accel_VIRTEX-6-1}
    {mgc_Xilinx-VIRTEX-6-1_beh_psr.lib {{mgc_Xilinx-VIRTEX-6-1_beh_psr part
    6VLX75TFF484}}} {ram_Xilinx-VIRTEX-6-1_RAMSB.lib ram_Xilinx-VIRTEX-6-1_RAMSB}}

go compile

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call
    /core/insert_search} -ITERATIONS 10

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call}
    -EFFORT_LEVEL high


directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call/
    core/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
```

```
    sched.elements._object._id:rsc} -MAP_TO_MODULE
    ram_Xilinx-VIRTEX-6-1_RAMSB.singleport
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call/
    core/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    sched.elements._rank._priority:rsc} -MAP_TO_MODULE
    ram_Xilinx-VIRTEX-6-1_RAMSB.singleport
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    sched.elements._prev:rsc} -MAP_TO_MODULE ram_Xilinx-VIRTEX-6-1_RAMSB.singleport
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call/
    core/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    sched.elements._prev.idx:rsc} -MAP_TO_MODULE ram_Xilinx-VIRTEX-6-1_RAMSB.singlepo
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    sched.elements._next:rsc} -MAP_TO_MODULE ram_Xilinx-VIRTEX-6-1_RAMSB.singleport
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    sched.elements._next.idx:rsc} -MAP_TO_MODULE ram_Xilinx-VIRTEX-6-1_RAMSB.singlepo

go architect

go extract
```

B.3.2 Cenário 2

```
project new
project set -name EPOSScheduler

solution file add ../../../src/scheduler_hw.cpp -type C++

directive set -REGISTER_IDLE_SIGNAL false
directive set -IDLE_SIGNAL {}
directive set -TRANSACTION_DONE_SIGNAL false
directive set -DONE_FLAG done
directive set -START_FLAG start
directive set -FSM_ENCODING none
directive set -REG_MAX_FANOUT 0
directive set -NO_X_ASSIGNMENTS true
directive set -SAFE_FSM false
directive set -RESET_CLEARS_ALL_REGS true
directive set -ASSIGN_OVERHEAD 0
directive set -DESIGN_GOAL area
directive set -OLD_SCHED false
```

```
directive set -PIPELINE_RAMP_UP true
directive set -COMPGRADE fast
directive set -SPECULATE true
directive set -MERGEABLE true
directive set -REGISTER_THRESHOLD 256
directive set -MEM_MAP_THRESHOLD 32
directive set -UNROLL no
directive set -CLOCK_OVERHEAD 20.000000
directive set -OPT_CONST_MULTS -1
go analyze

directive set -CLOCK_NAME clk

directive set -CLOCKS {clk {-CLOCK_PERIOD 20.0 -CLOCK_EDGE rising
        -CLOCK_UNCERTAINTY 0.0 -CLOCK_HIGH_TIME 10.0 -RESET_SYNC_NAME
        rst -RESET_ASYNC_NAME arst_n -RESET_KIND async -RESET_SYNC_ACTIVE
        high -RESET_ASYNC_ACTIVE high -ENABLE_NAME {} -ENABLE_ACTIVE high}}

directive set -TECHLIBS {{Xilinx_accel_VIRTEX-6-1.lib Xilinx_accel_VIRTEX-6-1}
    {mgc_Xilinx-VIRTEX-6-1_beh_psr.lib {{mgc_Xilinx-VIRTEX-6-1_beh_psr part
     6VLX75TFF484}}} {ram_Xilinx-VIRTEX-6-1_RAMSB.lib ram_Xilinx-VIRTEX-6-1_RAMSB}}

go compile

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/insert_search} -ITERATIONS 10

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call}
-EFFORT_LEVEL high

go architect

go extract
```

B.3.3 Cenário 3

```
project new
project set -name EPOSScheduler

solution file add ../../../src/scheduler_hw.cpp -type C++

directive set -REGISTER_IDLE_SIGNAL false
directive set -IDLE_SIGNAL {}
```

```
directive set -TRANSACTION_DONE_SIGNAL false
directive set -DONE_FLAG done
directive set -START_FLAG start
directive set -FSM_ENCODING none
directive set -REG_MAX_FANOUT 0
directive set -NO_X_ASSIGNMENTS true
directive set -SAFE_FSM false
directive set -RESET_CLEARS_ALL_REGS true
directive set -ASSIGN_OVERHEAD 0
directive set -DESIGN_GOAL area
directive set -OLD_SCHED false
directive set -PIPELINE_RAMP_UP true
directive set -COMPGRADE fast
directive set -SPECULATE true
directive set -MERGEABLE true
directive set -REGISTER_THRESHOLD 256
directive set -MEM_MAP_THRESHOLD 32
directive set -UNROLL no
directive set -CLOCK_OVERHEAD 20.000000
directive set -OPT_CONST_MULTS -1
go analyze

directive set -CLOCK_NAME clk

directive set -CLOCKS {clk {-CLOCK_PERIOD 20.0 -CLOCK_EDGE rising
        -CLOCK_UNCERTAINTY 0.0 -CLOCK_HIGH_TIME 10.0 -RESET_SYNC_NAME rst
        -RESET_ASYNC_NAME arst_n -RESET_KIND async -RESET_SYNC_ACTIVE
        high -RESET_ASYNC_ACTIVE high -ENABLE_NAME {} -ENABLE_ACTIVE high}}

directive set -TECHLIBS {{Xilinx_accel_VIRTEX-6-1.lib Xilinx_accel_VIRTEX-6-1}
    {mgc_Xilinx-VIRTEX-6-1_beh_psr.lib {{mgc_Xilinx-VIRTEX-6-1_beh_psr part
     6VLX75TFF484}}} {ram_Xilinx-VIRTEX-6-1_RAMSB.lib ram_Xilinx-VIRTEX-6-1_RAMSB}}

go compile

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/insert_search} -ITERATIONS 10

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::call}
-EFFORT_LEVEL high

directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/main/alloc_search} -UNROLL yes
```

```
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/main/insert_search} -UNROLL yes
directive set {/Scheduler_HW<Thread_Stub,Index,(Index)'\n',Status>::
    call/core/main/get_id_search} -UNROLL yes

go architect

go extract
```

B.4 Testbench VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.conv_std_logic_vector;
use ieee.numeric_std.all;
library std;
use std.env.all;

entity scheduler_testbench is
end scheduler_testbench;

architecture Behavioral of scheduler_testbench is

    constant CLKPERIOD : time := 10 ns;

    component Scheduler_HW_Thread_Stub_Index_Index_n_Status_call is
        Port(
            start : IN STD_LOGIC;
            done : OUT STD_LOGIC;
            m_rsc_z : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            param_rsc_z : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            priority_rsc_z : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            status_rsc_z : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
            return_val_rsc_z : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
            clk : IN STD_LOGIC;
            arst_n : IN STD_LOGIC);
    end component;

    signal sig_clk   : std_logic;
    signal sig_reset : std_logic;
    signal sig_start          : std_logic;
    signal sig_done           : std_logic;
    signal sig_m_rsc_z        : std_logic_vector(31 downto 0);
```

```
    signal sig_param_rsc_z       : std_logic_vector(31 downto 0);
    signal sig_priority_rsc_z    : std_logic_vector(31 downto 0);
    signal sig_status_rsc_z      : std_logic_vector(7 downto 0);
    signal sig_return_val_rsc_z  : std_logic_vector(31 downto 0);

    constant COMM_ENABLE       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(8, 32);
    constant COMM_DISABLE      : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(9, 32);
    constant COMM_CHOSEN       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(12, 32);
    constant COMM_CREATE       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(1, 32);
    constant COMM_INSERT       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(3, 32);
    constant COMM_DESTROY      : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(2, 32);
    constant COMM_REMOVE       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(4, 32);
    constant COMM_REMOVE_HEAD  : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(5, 32);
    constant COMM_SIZE         : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(13, 32);
    constant COMM_GET_ID       : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(11, 32);

    constant STAT_OK    : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(1, 32);
    constant STAT_ERROR : std_logic_vector(31 downto 0)
        := conv_std_logic_vector(2, 32);

begin

    clock: process
    begin
        sig_clk <= '1';
        wait for (CLKPERIOD/2);
        sig_clk <= '0';
        wait for (CLKPERIOD/2);
    end process;

    scheduler: Scheduler_HW_Thread_Stub_Index_Index_n_Status_call
        port map(
```

```
            start             => sig_start,
            done              => sig_done,
            m_rsc_z           => sig_m_rsc_z,
            param_rsc_z       => sig_param_rsc_z,
            priority_rsc_z    => sig_priority_rsc_z,
            status_rsc_z      => sig_status_rsc_z,
            return_val_rsc_z  => sig_return_val_rsc_z,
            clk               => sig_clk,
            arst_n            => sig_reset);

testbench: process
    procedure testInitiallySizeIsZero is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_SIZE;
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_return_val_rsc_z = conv_std_logic_vector(0, 32))
            report
                "initiallySizeIsZero" & LF &
                "expected = " & integer'image(0) & LF &
                "actual =   " &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
    end procedure testInitiallySizeIsZero;

    procedure indexReturnedFromFirstCreation is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_CREATE;
        sig_param_rsc_z <= conv_std_logic_vector(77, 32);
        sig_priority_rsc_z <= conv_std_logic_vector(1, 32);
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_return_val_rsc_z = conv_std_logic_vector(0, 32))
            report
                "indexReturnedFromFirstCreation" & LF &
                "expected = " & integer'image(0) & LF &
                "actual =   " &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
```

```
end procedure indexReturnedFromFirstCreation;

procedure statusAfterFirstInsertion is
begin
    sig_start   <= '1';
    sig_m_rsc_z <= COMM_INSERT;
    sig_param_rsc_z <= conv_std_logic_vector(0, 32);
    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
    assert (signed(sig_status_rsc_z) = signed(STAT_OK))
        report
            "statusAfterFirstInsertion" & LF &
            "expected␣=␣" & integer'image(to_integer(signed(STAT_OK))) & LF &
            "actual␣=␣␣␣" &
            integer'image(to_integer(signed(sig_status_rsc_z))) & LF
        severity failure;
end procedure statusAfterFirstInsertion;

procedure sizeIsOneAfterFirstInsertion is
begin
    sig_start   <= '1';
    sig_m_rsc_z <= COMM_SIZE;
    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
    assert (sig_return_val_rsc_z = conv_std_logic_vector(1, 32))
        report
            "sizeIsOneAfterFirstInsertion" & LF &
            "expected␣=␣" & integer'image(1) & LF &
            "actual␣=␣␣␣" &
            integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
        severity failure;
end procedure sizeIsOneAfterFirstInsertion;

procedure indexReturnedFromSecondCreation is
begin
    sig_start   <= '1';
    sig_m_rsc_z <= COMM_CREATE;
    sig_param_rsc_z <= conv_std_logic_vector(666, 32);
    sig_priority_rsc_z <= conv_std_logic_vector(10, 32);
    wait until rising_edge(sig_done);
    sig_start <= '0';
```

```
        wait for CLKPERIOD;
        assert (sig_return_val_rsc_z = conv_std_logic_vector(0, 32))
            report
                "indexReturnedFromSecondCreation" & LF &
                "expected␣=␣" & integer'image(0) & LF &
                "actual␣=␣␣␣" &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
    end procedure indexReturnedFromSecondCreation;

    procedure statusAfterSecondInsertion is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_INSERT;
        sig_param_rsc_z <= conv_std_logic_vector(1, 32);
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_status_rsc_z = signed(STAT_OK))
            report
                "statusAfterSecondInsertion" & LF &
                "expected␣=␣" & integer'image(signed(STAT_OK)) & LF &
                "actual␣=␣␣␣" &
                integer'image(to_integer(signed(sig_status_rsc_z))) & LF
            severity failure;
    end procedure statusAfterSecondInsertion;

    procedure sizeIsTwoAfterSecondInsertion is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_SIZE;
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_return_val_rsc_z = conv_std_logic_vector(2, 32))
            report
                "sizeIsTwoAfterSecondInsertion" & LF &
                "expected␣=␣" & integer'image(2) & LF &
                "actual␣=␣␣␣" &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
    end procedure sizeIsTwoAfterSecondInsertion;
```

```
procedure getIdOfThread666 is
begin
    sig_start   <= '1';
    sig_m_rsc_z <= COMM_GET_ID;
    sig_param_rsc_z <= conv_std_logic_vector(666, 32);
    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
    assert (sig_return_val_rsc_z = conv_std_logic_vector(1, 32))
        report
            "getIdOfThread666" & LF &
            "expected␣=␣" & integer'image(1) & LF &
            "actual␣=␣␣␣" &
            integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
        severity failure;
end procedure getIdOfThread666;

procedure thread666IsChosen is
begin
    sig_start   <= '1';
    sig_m_rsc_z <= COMM_CHOSEN;
    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
    assert (sig_return_val_rsc_z = conv_std_logic_vector(1, 32))
        report
            "thread666IsChosen" & LF &
            "expected␣=␣" & integer'image(1) & LF &
            "actual␣=␣␣␣" &
            integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
        severity failure;
end procedure thread666IsChosen;

procedure getIdOfThread77 is
begin
    sig_start   <= '1';
    sig_m_rsc_z <= COMM_GET_ID;
    sig_param_rsc_z <= conv_std_logic_vector(77, 32);
    wait until rising_edge(sig_done);
    sig_start <= '0';
    wait for CLKPERIOD;
    assert (sig_return_val_rsc_z = conv_std_logic_vector(0, 32))
        report
```

```
                "getIdOfThread77" & LF &
                "expected␣=␣" & integer'image(0) & LF &
                "actual␣=␣␣␣" &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
    end procedure getIdOfThread77;

    procedure statusAfterRemovalIndexZero is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_REMOVE;
        sig_param_rsc_z <= conv_std_logic_vector(0, 32);
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_status_rsc_z = signed(STAT_OK))
            report
                "statusAfterRemovalIndexZero" & LF &
                "expected␣=␣" & integer'image(signed(STAT_OK)) & LF &
                "actual␣=␣␣␣" &
                integer'image(to_integer(signed(sig_status_rsc_z))) & LF
            severity failure;
    end procedure statusAfterRemovalIndexZero;

    procedure sizeIsOneAfterFirstRemoval is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_SIZE;
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_return_val_rsc_z = conv_std_logic_vector(1, 32))
            report
                "sizeIsOneAfterFirstRemoval" & LF &
                "expected␣=␣" & integer'image(1) & LF &
                "actual␣=␣␣␣" &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
    end procedure sizeIsOneAfterFirstRemoval;

    procedure statusAfterDestructionIndexZero is
    begin
        sig_start   <= '1';
```

```
        sig_m_rsc_z <= COMM_DESTROY;
        sig_param_rsc_z <= conv_std_logic_vector(0, 32);
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_status_rsc_z = signed(STAT_OK))
            report
                "statusAfterDestructionIndexZero" & LF &
                "expected = " & integer'image(signed(STAT_OK)) & LF &
                "actual =   " &
                integer'image(to_integer(signed(sig_status_rsc_z))) & LF
            severity failure;
    end procedure statusAfterDestructionIndexZero;

    procedure indexReturnedFromSecondCreationIsZero is
    begin
        sig_start   <= '1';
        sig_m_rsc_z <= COMM_CREATE;
        sig_param_rsc_z <= conv_std_logic_vector(42, 32);
        sig_priority_rsc_z <= conv_std_logic_vector(1, 32);
        wait until rising_edge(sig_done);
        sig_start <= '0';
        wait for CLKPERIOD;
        assert (sig_return_val_rsc_z = conv_std_logic_vector(0, 32))
            report
                "indexReturnedFromSecondCreationIsZero" & LF &
                "expected = " & integer'image(0) & LF &
                "actual =   " &
                integer'image(to_integer(signed(sig_return_val_rsc_z))) & LF
            severity failure;
    end procedure indexReturnedFromSecondCreationIsZero;

begin
    sig_reset <= '1';
    wait for CLKPERIOD*3;
    sig_reset <= '0';

    testInitiallySizeIsZero; wait for CLKPERIOD*7;
    indexReturnedFromFirstCreation; wait for CLKPERIOD*7;
    statusAfterFirstInsertion; wait for CLKPERIOD*7;
    sizeIsOneAfterFirstInsertion; wait for CLKPERIOD*7;
    indexReturnedFromSecondCreation; wait for CLKPERIOD*7;
    statusAfterSecondInsertion; wait for CLKPERIOD*7;
```

```
        sizeIsTwoAfterSecondInsertion; wait for CLKPERIOD*7;
        getIdOfThread666; wait for CLKPERIOD*7;
        thread666IsChosen; wait for CLKPERIOD*7;
        getIdOfThread77; wait for CLKPERIOD*7;
        statusAfterRemovalIndexZero; wait for CLKPERIOD*7;
        sizeIsOneAfterFirstRemoval; wait for CLKPERIOD*7;
        statusAfterDestructionIndexZero; wait for CLKPERIOD*7;
        indexReturnedFromSecondCreationIsZero; wait for CLKPERIOD*7;

        finish(0);
    end process;

end Behavioral;
```

APÊNDICE C – Testes do adaptador mlite_cpu – AXI4Lite

C.1 Programa C de teste de escrita/leitura

```
#include "plasma.h"

#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)


int main()
{
    unsigned int i = 0;
    for(i = EXT_RAM_BASE; i < EXT_RAM_TOP; i += 4){
        MemoryWrite(i, i);
    }
    for(i = EXT_RAM_BASE; i < EXT_RAM_TOP; i += 4){
        if(MemoryRead(i) != i){
            while(1);
        }
    }

    return 0;
}
```

C.2 Testbench VHDL

```
library ieee;
use ieee.std_logic_1164.all;
library std;
use std.env.all;

entity plasma_axi4lite_testbench is
end plasma_axi4lite_testbench;

architecture Behavioral of plasma_axi4lite_testbench is
    component plasma_axi4lite_master is
        generic(
            memory_type     : string  := "XILINX_16X";
            mult_type       : string  := "DEFAULT";
            shifter_type    : string  := "DEFAULT";
            alu_type        : string  := "DEFAULT";
            pipeline_stages : natural := 2); --2 or 3
        port(
            aclk        : in std_logic;
            areset      : in std_logic;
            -- write address channel
            awvalid     : out std_logic;
            awready     : in std_logic;
            awaddr      : out std_logic_vector(31 downto 0);
            awprot      : out std_logic_vector(2 downto 0);
            -- write data channel
            wvalid      : out std_logic;
            wready      : in std_logic;
            wdata       : out std_logic_vector(31 downto 0);
            wstrb       : out std_logic_vector(3 downto 0);
            -- write response channel
            bvalid      : in std_logic;
            bready      : out std_logic;
            bresp       : in std_logic_vector(1 downto 0);
            -- read address channel
            arvalid     : out std_logic;
            arready     : in std_logic;
            araddr      : out std_logic_vector(31 downto 0);
            arprot      : out std_logic_vector(2 downto 0);
            -- read data channel
            rvalid      : in std_logic;
            rready      : out std_logic;
```

```
        rdata      : in std_logic_vector(31 downto 0);
        rresp      : in std_logic_vector(1 downto 0);

        -- plasma cpu interrupt, externalized
        intr       : in std_logic);
end component;

component ram_amba_128k is
    port(
        s_aclk          : in std_logic;
        s_aresetn       : in std_logic;
        s_axi_awaddr    : in std_logic_vector(31 downto 0);
        s_axi_awvalid   : in std_logic;
        s_axi_awready   : out std_logic;
        s_axi_wdata     : in std_logic_vector(31 downto 0);
        s_axi_wstrb     : in std_logic_vector(3 downto 0);
        s_axi_wvalid    : in std_logic;
        s_axi_wready    : out std_logic;
        s_axi_bresp     : out std_logic_vector(1 downto 0);
        s_axi_bvalid    : out std_logic;
        s_axi_bready    : in std_logic;
        s_axi_araddr    : in std_logic_vector(31 downto 0);
        s_axi_arvalid   : in std_logic;
        s_axi_arready   : out std_logic;
        s_axi_rdata     : out std_logic_vector(31 downto 0);
        s_axi_rresp     : out std_logic_vector(1 downto 0);
        s_axi_rvalid    : out std_logic;
        s_axi_rready    : in std_logic);
end component;

signal clk_50MHz     : std_logic;
signal sig_reset     : std_logic;
signal sig_intr      : std_logic;
signal sig_awvalid   : std_logic;
signal sig_awready   : std_logic;
signal sig_awaddr    : std_logic_vector(31 downto 0);
signal sig_awprot    : std_logic_vector(2 downto 0);
signal sig_wvalid    : std_logic;
signal sig_wready    : std_logic;
signal sig_wdata     : std_logic_vector(31 downto 0);
signal sig_wstrb     : std_logic_vector(3 downto 0);
signal sig_bvalid    : std_logic;
signal sig_bready    : std_logic;
```

```
    signal sig_bresp   : std_logic_vector(1 downto 0);
    signal sig_arvalid : std_logic;
    signal sig_arready : std_logic;
    signal sig_araddr  : std_logic_vector(31 downto 0);
    signal sig_arprot  : std_logic_vector(2 downto 0);
    signal sig_rvalid  : std_logic;
    signal sig_rready  : std_logic;
    signal sig_rdata   : std_logic_vector(31 downto 0);
    signal sig_rresp   : std_logic_vector(1 downto 0);

begin

    plasma_amba: plasma_axi4lite_master
        generic map(
            memory_type     => "XILINX_16X",
            mult_type       => "DEFAULT",
            shifter_type    => "DEFAULT",
            alu_type        => "DEFAULT",
            pipeline_stages => 2)
        port map(
            aclk        => clk_50MHz,
            areset      => sig_reset,
            -- write address channel
            awvalid     => sig_awvalid,
            awready     => sig_awready,
            awaddr      => sig_awaddr,
            awprot      => sig_awprot,
            -- write data channel
            wvalid      => sig_wvalid,
            wready      => sig_wready,
            wdata       => sig_wdata,
            wstrb       => sig_wstrb,
            -- write response channel
            bvalid      => sig_bvalid,
            bready      => sig_bready,
            bresp       => sig_bresp,
            -- read address channel
            arvalid     => sig_arvalid,
            arready     => sig_arready,
            araddr      => sig_araddr,
            arprot      => sig_arprot,
            -- read data channel
            rvalid      => sig_rvalid,
```

```
        rready      => sig_rready,
        rdata       => sig_rdata,
        rresp       => sig_rresp,

        intr        => sig_intr);

ram_amba: ram_amba_128k
    port map(
        s_aclk          => clk_50MHz,
        s_aresetn       => sig_reset,
        s_axi_awvalid => sig_awvalid,
        s_axi_awready => sig_awready,
        s_axi_awaddr  => sig_awaddr,
        s_axi_wvalid  => sig_wvalid,
        s_axi_wready  => sig_wready,
        s_axi_wdata   => sig_wdata,
        s_axi_wstrb   => sig_wstrb,
        s_axi_bvalid  => sig_bvalid,
        s_axi_bready  => sig_bready,
        s_axi_bresp   => sig_bresp,
        s_axi_arvalid => sig_arvalid,
        s_axi_arready => sig_arready,
        s_axi_araddr  => sig_araddr,
        s_axi_rvalid  => sig_rvalid,
        s_axi_rready  => sig_rready,
        s_axi_rdata   => sig_rdata,
        s_axi_rresp   => sig_rresp);

clk_process: process
begin
    clk_50MHz <= '1';
    wait for 10 ns;
    clk_50MHz <= '0';
    wait for 10 ns;
end process;

-- do not interrupt the CPU
sig_intr <= '0';

tb : process
begin
    sig_reset <= '0';
    wait for 60 ns;
```

```
        sig_reset <= '1';

        wait for 100000 ns;

        finish(0);
    end process;

end Behavioral;
```