UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JEFERSON SANTIAGO DA SILVA

Architectural Exploration of Digital Systems Design for FPGAs Using C/C++/SystemC Specification Languages

Thesis presented in partial fulfillment
of the requirements for the degree of
Master in Computer Science

Prof. Dr. Sergio Bampi
Advisor

Porto Alegre, January 2015

CIP – CATALOGING-IN-PUBLICATION

da Silva, Jeferson Santiago

Architectural Exploration of Digital Systems Design for FPGAs Using C/C++/SystemC Specification Languages / Jeferson Santiago da Silva. – Porto Alegre: PPGC da UFRGS, 2015.

83 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Sergio Bampi.

1. High-level Synthesis. 2. FPGA. 3. Design Space Exploration. 4. Digital Design. 5. Optimization Techniques. I. Bampi, Sergio. II. Architectural Exploration of Digital Systems Design for FPGAs Using C/C++/SystemC Specification Languages.

“Best remain silent and be thought a fool,
than open your mouth and remove all doubt.”
— Abraham Lincoln

ACKNOWLEDGMENTS

I would like to thank all my family, specially to my wife Clara. I want to thank Prof. Bampi, my advisor in this work. My thanks also go to all staff of PPGC, including professors and administrative personnel.

ABSTRACT

The increasing demand for high computational performance and massive data processing has driven the development of systems-on-chip. One implementation target for complex digital systems are FPGA (Field-programmable Gate Array) devices, heavily used for prototyping systems or complex and fast time-to-market electronic products development. Certain inefficient aspects of FPGA devices relate to performance and power degradation with respect to custom hardware design.

In this context, this master thesis proposes a survey on FPGA optimization techniques. This work presents a literature review on methods of power and area reduction applied to FPGA designs. Techniques for performance increasing and design speedup enhancing will be presented based on classic and state-of-the-art academic works. The main focus of this work is to discuss high-level design techniques and to present the results obtained in synthesis examples we developed, comparing with hand-coded HDL (Hardware Description Language) designs.

In this work we present our methodology for fast digital design development using High-Level Synthesis (HLS) environments. Our methods include efficient high-level code partitioning for proper synthesis directives exploration in HLS tools. However, a non-guided HLS flow showed poor synthesis results when compared to hand-coded HDL designs. To fill this gap, we developed an iterative design space exploration method aiming at improving the area results. Our method is described in a high-level script language and it is compatible with the Xilinx Vivado™ HLS compiler. Our method is capable of detecting optimization checkpoints, automatic synthesis directives insertion, and check the results aiming at reducing area consumption.

Our Design Space Exploration (DSE) experimental results proved to be more efficient than non-guided HLS design flow by at least 50% for a VLIW (Very Long Instruction Word) processor and 62% for a 12^{th}-order FIR (Finite Impulse Response) filter implementation. Our area results in terms of flip-flops were up to 4X lower compared to a non-guided HLS flow, while the performance overhead was around 38%, for the VLIW processor compilation. In the FIR filter example, the flip-flops reduction were up to 3X, with no relevant LUTs and performance overhead.

Keywords: High-level Synthesis, FPGA, Design Space Exploration, Digital Design, Optimization Techniques.

Exploração Arquitetural no Projeto de Sistemas Digitais para FPGAs Utilizando Linguagens de Especificação C/C++/SystemC

RESUMO

A crescente demanda por alto desempenho computacional e massivo processamento de dados tem impulsionado o desenvolvimento de sistemas-on-chip. Um dos alvos de implementação para sistemas digitais complexos são os dispositivos FPGA (*Field-programmable Gate Array*), muito utilizados para prototipação de sistemas e rápido desenvolvimento de produtos eletrônicos complexos. Certos aspectos ineficientes relacionados aos dispositivos FPGA estão relacionadas com degradação no desempenho e na potência consumida em relação ao projeto de hardware customizado.

Neste contexto, esta dissertação de mestrado propõe um estudo sobre técnicas de otimização em FPGAs. Este trabalho apresenta uma revisão da literatura sobre os métodos de redução de potência e área aplicados ao projeto de FPGA. Técnicas para aumento de desempenho e aceleração do tempo de desenvolvimento de projetos são apresentadas com base em referencias clássicas e do estado-da-arte. O principal foco deste trabalho é discutir sobre as técnicas de alto nível e apresentar os resultados obtidos nesta área, comparando com os projetos HDL (*Hardware Description Language*) codificados a mão.

Neste trabalho, é apresentado uma metodologia para o desenvolvimento rápido projetos digitais utilizando ambientes HLS (*High-Level Synthesis*. Estes métodos incluem eficiente particionamento de código de alto nível, para a correta exploração de diretivas de síntese em ferramentas HLS. Porém, o fluxo HLS não guiado apresentou pobres resultados de síntese quando comparado com modelos HDL codificado a mão. Para preencher essa lacuna, foi desenvolvido um método iterativo para exploração de espaço de projeto com o objetivo de melhorar os resultados de área. Nosso método é descrito em uma linguagem de script de alto nível e é compatível com o Vivado™ HLS Compiler. O método proposto é capaz de detectar pontos chave para otimização, inserção automatica de diretivas síntese e verificação dos resultados com objetivo de reduzir o consumo de área.

Os resultados experimentais utlizando o método de DSE (*Design Space Exploration*) provaram ser mais eficazes que o fluxo HLS não guiado, em ao menos 50% para um processador VLIW e em 43% para um filtro FIR (*Finite Impulse Response* de 12ª ordem. Os resultados em área, em termos de flip-flops, foram até 4X menores em comparação com o fluxo HLS não guiado, enquanto redução no desempenho ficou em cerca de 38%, no caso do processador VLIW. No exemplo do filtro FIR, a redução no número flip-flops chegou a 3X, sem relevante aumento no número de LUTs e redução no desempenho.

Palavras-chave: Síntese de Alto Nível, FPGA, Exploração de Espaço de Projeto, Sistemas Digitais, Técnicas de Otimização.

LIST OF ABBREVIATIONS AND ACRONYMS

ABEL	Advanced Boolean Expression Language
ABL	A Block diagram Language
ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
APL	A Programming Language
ASIC	Application-Specific Integrated Circuit
CE	Clock Enable
CMOS	Complementary Metal-Oxide-Semiconductor
CT	Compute Time
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSE	Design Space Exploration
DDDG	Dynamic Data Dependence Graphs
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FF	Flip-flop
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GIMP	GNU Image Manipulation Program
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers

ISP	Instruction Set Processing
JPEG	Joint Photographic Experts Group
LDMC	Logic Delay Measurement Circuit
KARL	Kaiserslautern RTL
LE	Logic Element
LUT	Look-up Table
MAC	Multiply and Accumulate
MIPS	Microprocessor without Interlocked Pipeline Stages
MPEG	Moving Picture Experts Group
OpenCL	Open Computing Language
PLL	Phase-locked loop
QoR	Quality of Results
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer Level
SDK	Software Development Kit
SRAM	Static RAM
SW	Software
TCL	Tool Command Language
Vex	VLIW Example
VLIW	Very Long Instruction Word
VHDL	VHSIC HDL
VHSIC	Very-High-Speed IC

LIST OF SYMBOLS

α	Switching Rate
C	Capacitance
f	Operation Frequency
V_{dd}	Power-Supply Voltage
τ_{DN}	Wire Delay
L	Wire Length
c	Capacitance of a Wire per Unit Area
r	Sheet Resistance of a Wire ($\Omega/square$)
ρ-Vex	Reconfigurable Vex Processor

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

FPGA devices are widely used for faster implementation of digital functions in hardware and for ASIC (Application-Specific Integrated Circuit) emulation. However, even state-of-the-art FPGAs have serious limitations in terms of performance, area utilization and power consumption. With the increasing complexity of digital systems, optimization techniques must be performed to make digital projects with these devices competitive.

In this work we discuss the most common techniques available in the literature targeting FPGA design optimization. These methods include power savings such techniques as clock gating and frequency/voltage scaling. In terms of area, we will discuss the importance of resource sharing. For performance enhancing, techniques such as pipelining, parallelism and fan-out reduction will be reviewed.

The main purpose of this master thesis is to discuss and to explore high-level design optimization techniques applied to FPGA systems. The Register-Transfer Level (RTL) design is the project schedule bottleneck in digital systems design. HLS (High-Level Synthesis) methods and utilization will be discussed for fast turn-around purpose. Commercial and academic HLS tools will be presented, as well as, advanced design techniques using high-level specification languages, for example C/C++ or SystemC.

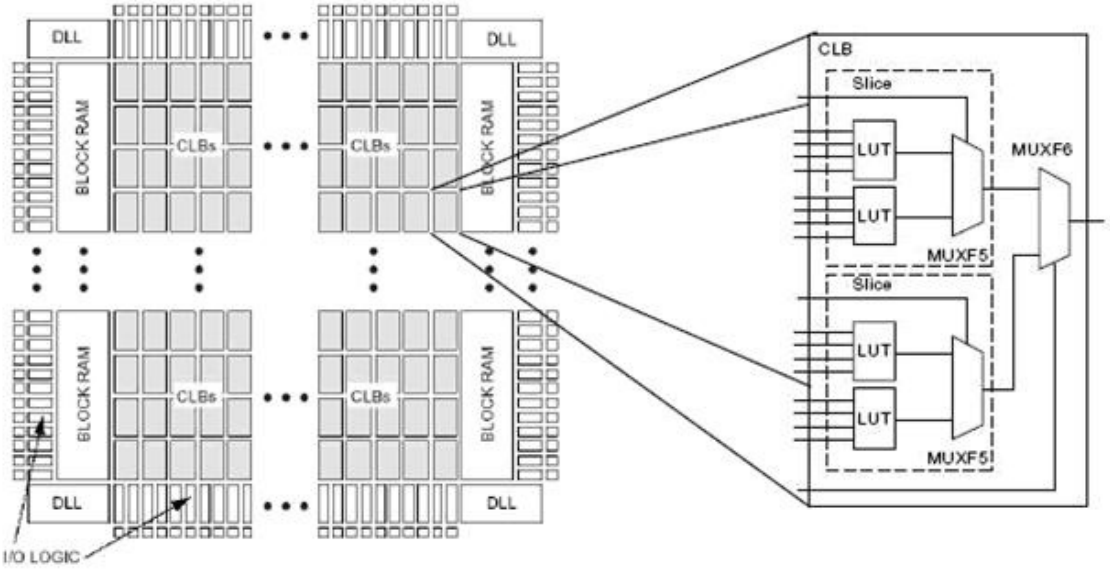
Many algorithm standards are open-source available in high-level languages, such as C. In the classic FPGA design flow, hardware designers must to translate these algorithms onto FPGA hardware performing all steps of optimization using register-transfer level languages. RTL development and verification is a complex and expensive task when compared to software implementations. Higher level abstractions allow software designers to develop FPGA hardware using HLS compilers. HLS tool providers affirm that for the certain test-cases, the productivity can be increased up to 50% using HLS methods compared to hand-coded RTL design. The addition of directives/pragmas in the HLS compilers allows the designers to guide the algorithm implementation for a specific design trade-off, for example, performance or area reduction. The verification process in a RTL based flow is very hard due to limitations of HDL languages in describing the real world circuit behaviour, while using higher abstraction languages it is possible to emulate the operation conditions. Moreover, a RTL simulation can take several hours, even days depending of circuit complexity, which normally requires powerful simulation servers.

In this context, this work presents a methodology for digital design using high-level languages through HLS design flow. We explore techniques of code partitioning and pragmas insertion targeting Quality of Results (QoR) improvement. In our work, we developed an iterative method for wide design space exploration with HLS aiming at area reduction. Our method proves to be very effective when compared to a non-guided HLS design flow, being up to 50% more efficient, using an academic VLIW processor as design benchmark.

1.1 Motivation

The FPGA is a pre-defined chip, which contains a large number of macro-cells, interconnected with each other by routing wires and routing switches. Each macro-cell has inside a programmable input multiplexer logic which is capable to implement all possible logic equations of its inputs combinations and a storage element (flip-flop). Modern FPGA devices include RAM (Random Access Memory) blocks, DSP (Digital Signal Processing) blocks, PLLs (Phase-locked loop), gigabit transceivers and even embedded hard processors. Figure 1.1 illustrates a typical FPGA diagram.

Figure 1.1: Typical FPGA Scheme.



Source: Xilinx, Inc.

According to data from the FPGA vendors, normally more than 70% of the silicon area of an FPGA is destined to interconnections, which makes these devices very inefficient in terms of area consumption. As a pre-defined silicon wafer, the power consumption is a problem, because the power leakage component is always present in the entire chip, even if its logic cells and flip-flops are not used. In terms of dynamic power, the major vendors of FPGA have made a hard effort to make these devices more efficient. Hard components - like blocks of SRAM (Static Random Access Memory) and DSPs - also consume power, but these components optimization methods and utilization will not be discussed in this work.

The performance in FPGAs is worse than in ASIC devices. While ASICs are designed targeted to a specific application, FPGAs are generic and this impacts directly the timing issues. Many logic levels between storage elements, large fan-outs, the clock tree, the wire delay - including nets and routing switches - and fixed logic element placement contribute to decreasing of maximum operation frequency.

The hardware development is high cost and takes a large fraction of design time. The traditional FPGA development is based on system level architecture, RTL description, functional simulation and at end, logical synthesis and place and routing. In this method, the hardest task is the RTL description, normally a slow and meticulous task. Advanced

HLS techniques automate task, generating a synthesizable RTL code from a high-level language. Figure 1.2 shows the classic design flow. Recent efforts in HLS research have made the high-level systems design competitive for FPGA-based systems. Regular architectures, such as those DSP algorithms, are normally present in works in the literature to evaluate the high-level implementations. Algorithm scheduling, loop/function pipelining, parallelism and loop unrolling are common techniques used by HLS tools providers aiming at results improvement.

Even with HLS tools enhancement in recent years, some designs still difficult to obtain good synthesis results. Irregular architectures and control based algorithms, for instance, have limited synthesis results comparing to the hand-coded HDL designs. In this context, it is necessary to develop architectural design exploration for these designs, providing the necessary adaptations in the design entry code and to explore efficiently the HLS compiler parameters.

1.2 Contributions

The main contribution of this master thesis is to propose high-level exploration techniques for digital systems design targeted to FPGA devices. The main contributions of this work are listed below:

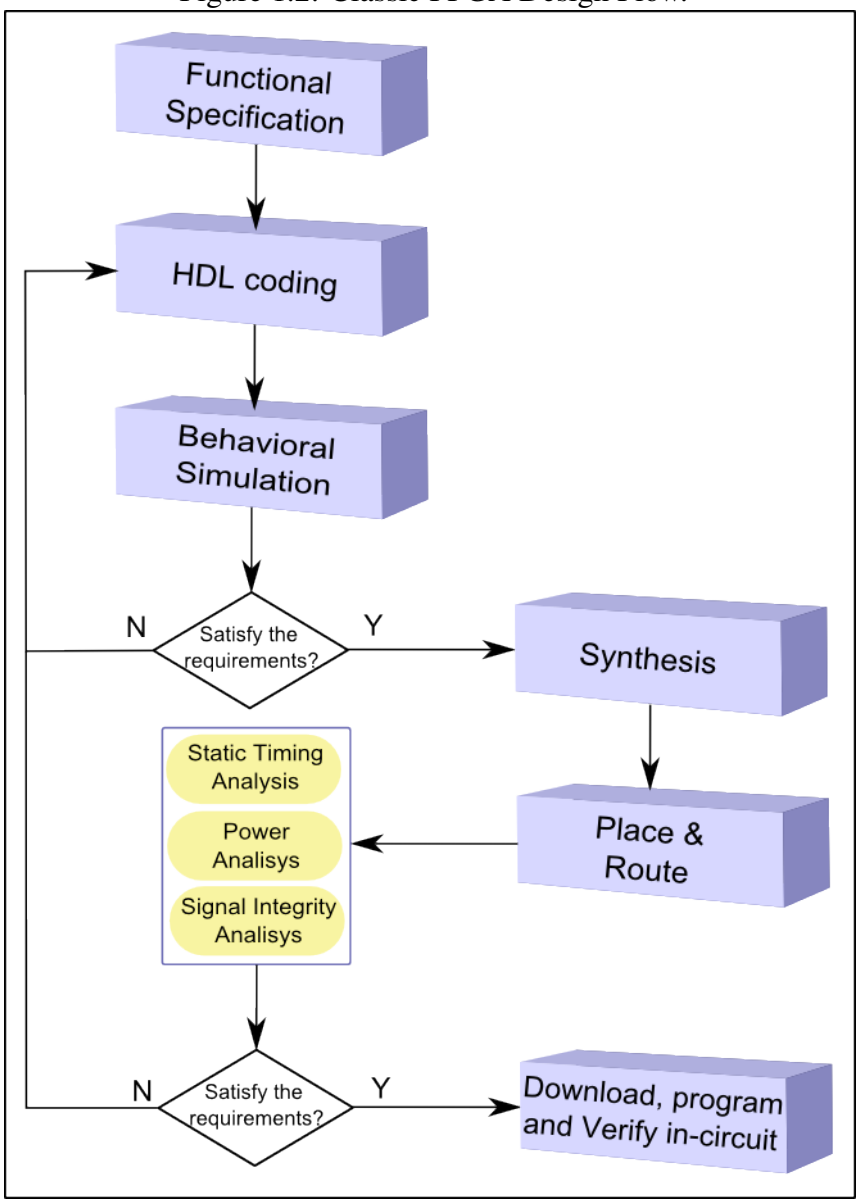
- **High-level Architectural Exploration:** to achieve good synthesis results with high-level descriptions it is necessary some code improvements. We discuss in our methodology some techniques used in HLS environments. We propose an efficient code partitioning combined with appropriate HLS compilation directives/pragmas aiming at synthesis results improvement. Our experiments addressed two HLS tools: the LegUp compiler and the Vivado™ HLS compiler.

- **Design Space Exploration Method with High-level Synthesis:** addressing higher Quality of Results in the HLS environment, we proposed an iterative method for DSE of hardware systems. Our method is multi-platform, it is written in Lua language (LUA, 2014) and it is compatible with Vivado™ HLS compiler. This method is basically composed of these steps: high-level code parsing and analysis, automatic compilation directives insertion and results evaluation.

1.3 Thesis Organization

This master thesis is organized as follows: Hardware optimization techniques and methods are reviewed in Chapter 2. In this chapter, we discuss about well-known methodologies for design optimization in terms of area, power and performance for FPGAs. HLS tools and methods are revisited in Chapter 3, presenting the evolution of HDL languages and the state-of-the-art HLS tools and the known HLS optimization methods available in the literature. Design Space Exploration in hardware systems is presented in Chapter 4, where are introduced important concepts for high-efficient DSE applied to hardware design. In the Chapter 5 we present our methodology for this work, which includes high-level code tuning and an iterative DSE method with HLS. Chapter 6 presents our experimental results, for both HLS tools comparison and DSE methodology in a HLS environment. Finally, the conclusions of this work are drawn in the Chapter 7.

Figure 1.2: Classic FPGA Design Flow.



Source: the author.

2 HARDWARE DESIGN OPTIMIZATION TECHNIQUES

FPGA devices are known for being very inefficient in terms of power consumption, useful silicon area and performance, when compared to equivalent dedicated ASIC devices. Over the last decades, researchers have addressed much effort to make this design flow more competitive compared to ASICs. Clock enable and clock gating methods have driven the power reduction techniques in recent years. Deep pipelines and hardware parallelism are widely used targeting performance increasing in FPGAs. For area purposes, a well known method is resource sharing combined with efficient scheduling mechanisms.

In this context, this chapter will discuss design techniques applied to FPGA-based systems. These methods address different design trade-offs. Area saving techniques, power reduction methods and performance optimization methods are presented according to state-of-the-art literature.

2.1 Area Saving Methods

EDA (Electronic Design Automation) tools from FPGA providers have in their design suites, many synthesis options for area saving purpose. These options include register sharing, equivalent registers removal, hierarchy destruction and FSM (Finite State Machine) extraction into RAMs.

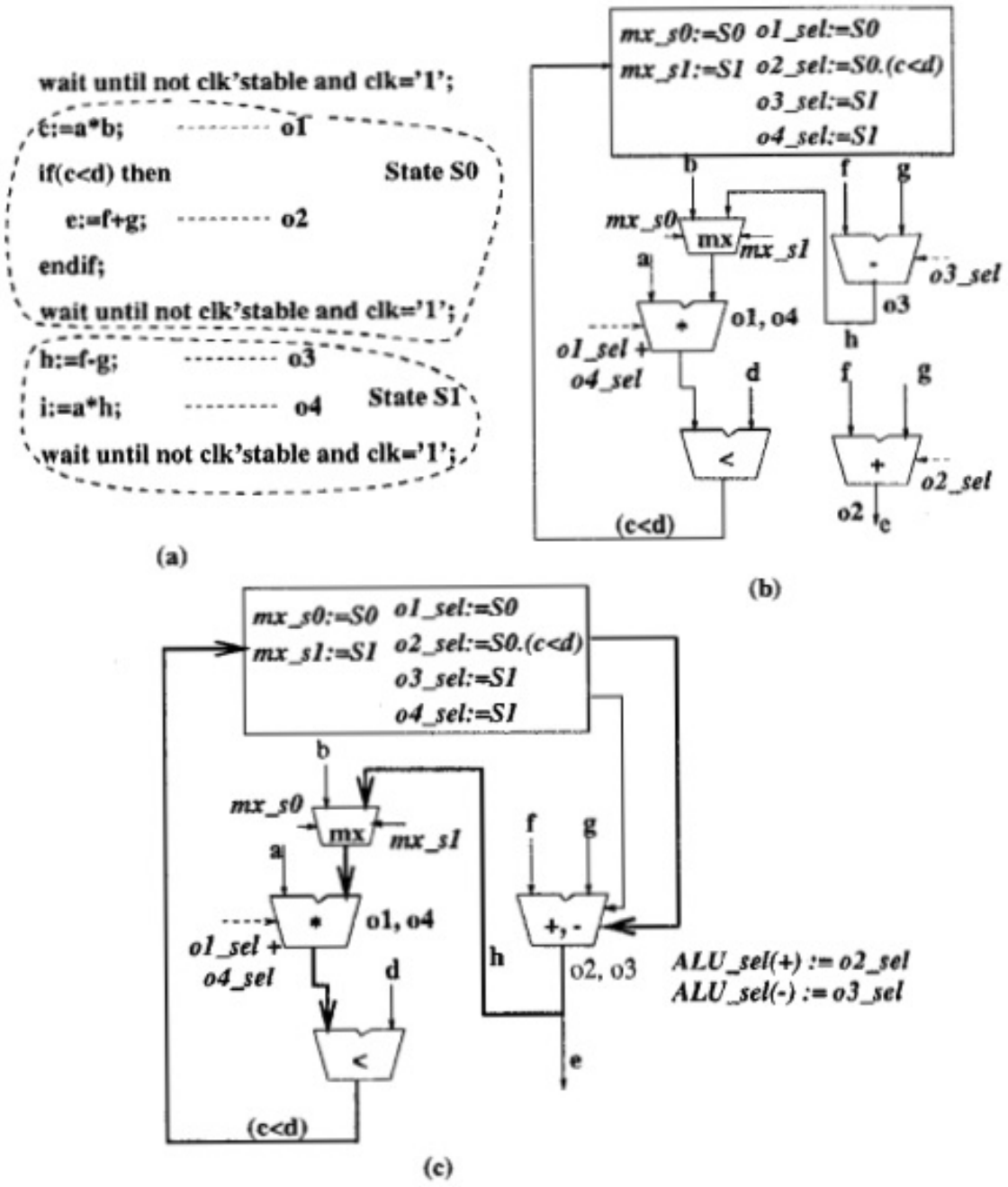
In the literature, a common area saving method is the resource sharing/re-utilization. Figure 2.1 shows an illustration of the resource sharing method proposed in (RAJE; BERGAMASCHI, 1997). This technique includes the optimal component placement and task scheduling allowing efficient resource sharing (SUN; WIRTHLIN; NEUENDORFFER, 2007).

2.2 Power Saving Methods

As in ASIC design, frequency scaling is a common power saving technique when we talk about FPGA design. Many works present in the literature illustrates the good results in terms of power reduction using frequency scaling. One method applied to synchronous system is the clock enable. All modern FPGAs have a clock enable pin in their flip-flops. This pin is used to perform the clock enable technique in FPGA devices. Figure 2.2 illustrates a typical LE (Logic Element) of an FPGA device.

When CE (Clock Enable) is asserted, the storage element in the FPGA LE works exactly like the D-type FF (Flip-flop). When the CE pin is setted to 0, the clock source is disabled in the flip-flop and the outputs do not change their logic states. According Equation 2.1, the dynamic power in a CMOS (Complementary Metal-Oxide-Semiconductor)

Figure 2.1: Resource Sharing. a) Behavioural description. b) Data and Control Path. c) Data and Control Path with Sum/Sub Operations Merged



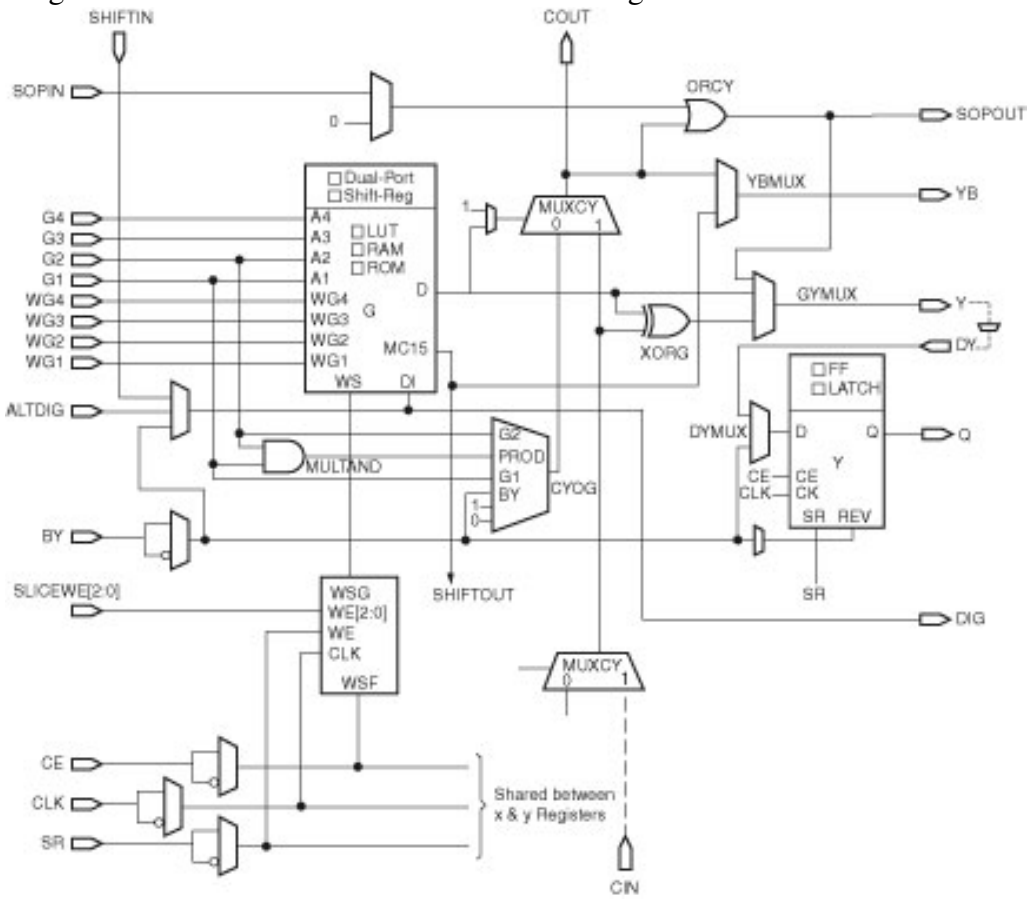
Source: (RAJE; BERGAMASCHI, 1997).

circuit is a function, between others parameters, of the switching rate:

$$P_{dyn} = CV_{dd}^2 f \alpha \tag{2.1}$$

where P_{dyn} is the dynamic power, C represents the capacitance, V_{dd} represents the supply voltage and α is the switching rate. If CE pin is not asserted the dynamic power goes to 0. However, the clock enable method reduces power only in the flip-flop and combinational

Figure 2.2: Internal Architecture of a FPGA Logic Element inside Virtex™ 5.



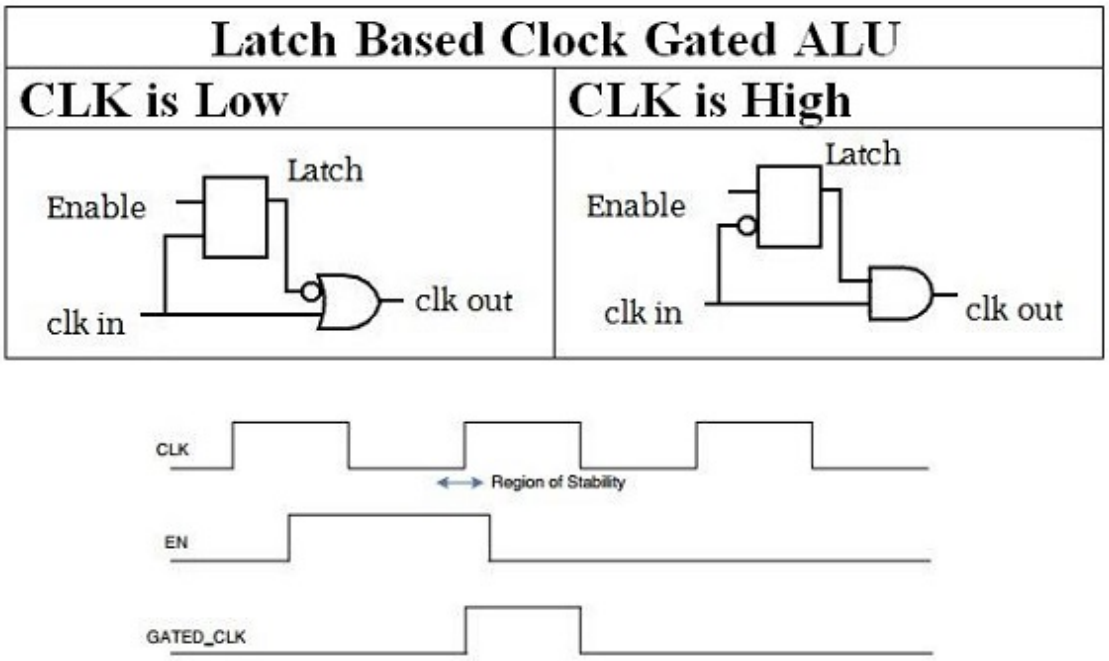
Source: Xilinx, Inc.

logic, being not effective in the clock tree.

On the other hand, the clock gating technique is independent from FPGA technology and optimize power consumption, including in the clock tree. This method is performed using simple circuits, according to clock polarity. Many works have been presented in recent years considering this topic. (ZHANG; ROIVAINEN; MAMMELA, 2006), presented in 2006 a comparative study between FPGA and ASIC clock gating. His experiments suggest good results for dynamic power reduction in FPGAs, compared to ASIC, however this power reduction is insignificant because of static FPGA power consumption. Huda presented in 2009 (HUDA; MALLICK; ANDERSON, 2009), a clock-gating technique based on efficient FPGA clock region division and a placement algorithm. Pandey proposes in (PANDEY et al., 2013), two latch-based gated clock generator, as shown in Figure 2.3. The gated clock generated by the circuits drives the registers and it controls the global reset for an ALU (Arithmetic and Logic Unit).

In 2012, Oliver presented in (OLIVER et al., 2012), a comparative study between three power saving techniques, two of them, clock-enable and clock gating, were already previously discussed. The other technique used in his work was the inputs blocking. It was made implementing a kind of latch, which is enabled with the CE pin of FPGA LE. In his results section it was clear the impact of clock tree on power consumption, because in standby-mode, the clock-enable and inputs blocking techniques had power increasing with enlarging circuits, while using the clock gating technique the power consumption

Figure 2.3: Gated Clock Circuits: a) Falling edge. b) Rising edge.



Source: (PANDEY et al., 2013).

had no changes. On the other hand, the active-mode experiment had almost the same consumption for all three cases.

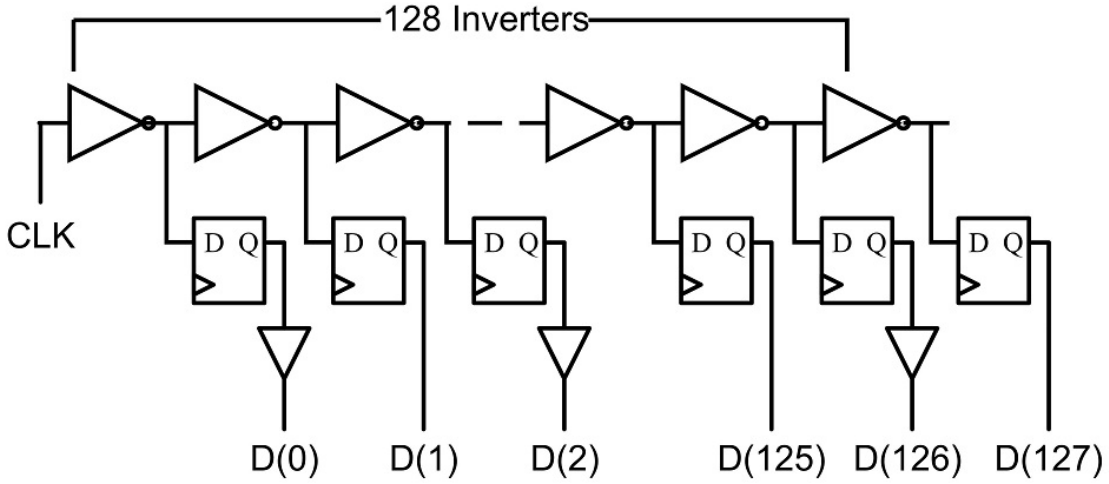
The voltage scaling - as frequency scaling was initially developed for ASIC designs - is a technique used in FPGA-based systems for power saving purpose. This technique reduces the power consumption in some areas in the chip. Remembering the Equation 2.1, the dynamic power is proportional to square of the supply voltage.

Adaptive voltage scaling systems have been developed in last years. The goal of this technique is tuning the correct supply voltage according the performance requirements. In this context, Chow (CHOW et al., 2005) proposed in 2005, dynamic voltage scaling method based on LDMC (Logic Delay Measurement Circuit) to control the supply voltage. This LDMC is a 128 components inverter chain, which analyses the logic delay to find out the critical delay with no data errors, and the tuned LDMC value is used to calibrate the external power supply controller. The LDMC scheme is illustrated in Figure 2.4.

2.3 Performance Enhancing Methods

Most of performance optimizations techniques targeted to ASIC devices are also applicable to FPGA systems. Pipelining and parallelism are main methods for performance increasing. The MIPS (Microprocessor without Interlocked Pipeline Stages) processor, the precursor of the modern RISC (Reduced Instruction Set Computer) machines, was developed in mid-1980s with pipelinig technique (PATTERSON; HENNESSY, 2007). Most of present applications' datapath use deep pipelines to increase performance. In high definition video coding, for instance, it is common architectures with more than ten stages of pipeline.

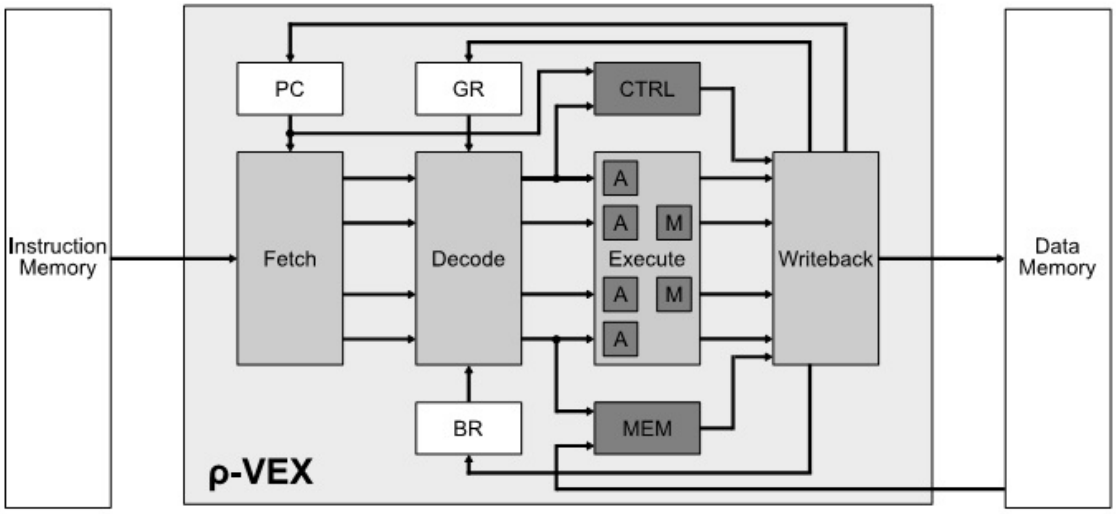
Figure 2.4: LDCM scheme proposed in (CHOW et al., 2005).



Source: (CHOW et al., 2005).

The concept of parallelism is to define the operations which could be executed at same time in parallel, using multiple hardware units. For proper operation of parallelism method, the processed data cannot have dependency with each other. Examples of parallelism application are vectorial and super-scalar processors. The characteristic of these machines are the massive datapath computing. Wong proposed in (WONG; AS; BROWN, 2008), a flexible VLIW implementation targeted to FPGA. The processor is basically a regular MIPS, where is included a flexible number of parallel ALUs. The processor scheme is illustrated in Figure 2.5.

Figure 2.5: Four issue VLIW scheme.



Source: (WONG; AS; BROWN, 2008).

The methods cited in previous paragraphs refer only to architectural optimizations for

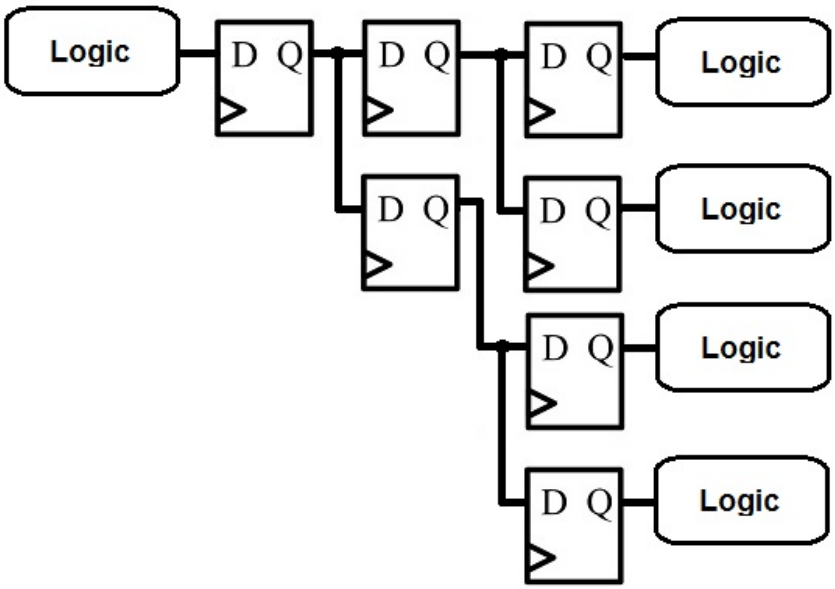
FPGA devices. Others techniques address how is possible to deal the technology characteristics, such as long wires in CMOS circuits, which could represent a bad design guide in a performance-oriented system. The Equation 2.2 (RABAEY; CHANDRAKASAN; NIKOLIC, 2003) associates the wire length and specific resistance/capacitance. The relation between net delay and wire length is quadratic, then the wire increasing means a drastic frequency reduction:

$$\tau_{DN} = \frac{rcL^2}{2} \tag{2.2}$$

where τ_{DN} represents the wire delay, r is the specific resistance per square, c is the capacitance per unit area, and L is the wire length.

The fan-out reduction technique could be useful to reduce long wires and wire capacitance. One known method of fan-out control is based on a binary tree. Its a register-oriented technique and each FF has maximum of two internal register fan-out. The number of output is 2^N with latency N. The area overhead of this fan-out reducing proposal is $\sum_{i=0}^{N-1} 2^i$. Figure 2.6 shows a typical circuit for a factor 2 fan-out reduction.

Figure 2.6: Fan-out Reduction Circuit (N = 2).



Source: the author.

2.4 Hardware Optimization Methods Summary

The methods discussed in previous sections are a few of possible techniques available on a large digital design optimization universe. In the last 50 years, several other techniques were presented in the literature and all possible techniques cannot be presented in single work.

In this chapter, we revisited the most known area saving methods, as resource sharing and re-utilization. In terms of power reduction techniques, we reviewed some works: clock gating, clock enable and voltage scaling were introduced. For performance optimization, we discussed over pipelining, parallelism and fan-out reduction.

This work focuses in architectural exploration with high-level synthesis, aiming at improving timing-to-market. Next chapters present the state-of-the-art on HLS research, work methodology and experimental results with HLS methods.

3 METHODS TO REDUCE DESIGN TIME

The complexity increase of digital systems has motivated designers worldwide to describe digital designs faster, where multi-million logic cells are normal in recent digital projects. The productivity increase is related to discovering how it is possible to accelerate the time necessary to describe and verify a hardware system. A known method is using high-level synthesis tools. HLS tools map directly an algorithmic description into a sysnthesizable RTL language, making the design cycle shorter.

This chapter will discuss how to reduce time-to-market in the design flow. Recent research and tools allow the utilization of HLS methods aiming at improving the design time. In this chapter we revisited the evolution of HDL languages over the last decades in Section 3.1. Section 3.2 reviews academic and commercial HLS tools. In Section 3.3 we present the high-level techniques applied to FPGA systems.

3.1 Evolution of Hardware Description Languages

Since the transistor invention and the establishment of the integrated circuit technology in 1947 and after 1958 respectively, designers have tried to accelerate the digital design. First know design entry was the schematic capture. Using schematics were possible to create small-to-medium complexity circuits. However, Moore's law (MOORE, 1998) was far away to be achieved using schematic design.

In 1971, C. Gordon Bell and Allen Newell, introduced in (BELL; NEWELL, 1971) the concept of RTL description (i.e. hardware description at the register transfer level). The RTL still being a major characteristic today in all synthesizable hardware description languages. In their book, they show the PDP-8 processor and present the RTL through ISP (Instruction Set Processing) language. In the end of 1970 decade, the University of Kaiserslautern developed their own register-transfer level language, called KARL (Kaiserslautern Register-Transfer Level). KARL was developed simultaneously with ABL (A Block diagram Language) language. Both together formed a common VLSI framework in Europe, in middle 1980's (HARTENSTEIN, 1993).

The 1980 decade was the peak for the HDL development. In the first half (1983) the ABEL (Advanced Boolean Expression Language) language was created. The ABEL developed framework was presented in 1985 (LEE et al., 1985). This framework was composed by the ABEL HDL language and the ABEL processor. ABEL syntax and semantics are derived from C language. It is composed by conditional and sequential statements, and logical and arithmetical operators.

The two most important HDL languages today, Verilog and VHDL (Very-high-speed integrated circuit Hardware Description Language, were developed in 1980's. Verilog was standardized by IEEE (Institute of Electrical and Electronics Engineers) 1364 Standard.

Verilog was developed by Gateway Design Automation, in 1985. Verilog is similar to C language, and this characteristic made the designers to become interested in this language. Two other derivations of C are the case-sensitive code and the preprocessor, allowing conditional compilation. The reserved words for sequential and conditional statements or logical and arithmetical operators are equal to C. Some differences among C and Verilog are the bit-width definition for variables and the clock orientation. Beside this, Verilog has four logic values for bit representation: 1 (high), 0 (low), floating and undefined. Digital circuits are represented, in Verilog, as modules, where each module has ports and parameters. Ports are defined according direction: in, out and inout. In Verilog, all statements are executed in parallel, except some ones that are placed inside **always** statement, in this case the execution is sequential. In the **always** statement is defined the sensitivity list for sequential execution. This characteristic of parallelism makes the Verilog design complicated for software designers, because the execution of programs in C is assumed to be sequential only. In the Figure 3.1 is shown a D type FF implementation in Verilog. The most recent version of Verilog is from 2005.

Figure 3.1: Verilog Implementation for a D-FF.

```
module dff(clock, reset, d);
    input clock, reset, d;
    output q;
    reg q;
    always @(reset or posedge clock)
        if (reset == 1)
            q <= 0;
        else
            q <= d;
        end
endmodule
```

Source: the author.

VHDL is defined in the IEEE 1076 Standard. VHDL was initially developed by the US Department of Defense for ASIC development purpose. VHDL syntax and semantics were based on Ada language. Similar to Verilog, VHDL has no defined bit-width for its variables. VHDL has nine logic values for each bit: uninitialized ('U'), unknown ('X'), high ('1'), low ('0'), high impedance('Z'), week signal ('W'), weak high ('H'), weak low ('L') and don't care ('-') . Another characteristic similar to Verilog is the parallelism. **Architecture**s in VHDL are executed in parallel, while the code inside the **process**es are sequential, being in the **process** statement the definition for the sensitivity list. Figure 3.2 presents a VHDL implementation for a D type flip-flop. The last version of VHDL is from 2009.

For the last 30 years, Verilog and VHDL have struggled among themselves aiming at being the standard HDL language for FPGA/ASIC design. While Verilog is most appreciated in United States, VHDL is more common in Europe and Asia. This competition and the difficulties encountered by software designers for the RTL description understanding have motivated the development of higher level languages for hardware description purpose. In the beginnings of 2000's, a set of EDA providers created the SystemC language. SystemC is a class of C++ thought for hardware modelling. SystemC is defined in the

Figure 3.2: VHDL Implementation for a D-FF.

```
library ieee;
    use ieee.std_logic_1164.all;
entity dff is
    port(
        clock, reset, d : in  std_logic;
        q               : out std_logic
    );
end dff;
architecture dff of dff is
begin
    process(clock, reset)
    begin
        if reset = '1' then
            q <= '0';
        elsif rising_edge(clock) then
            q <= d;
        end if;
    end process;
end dff;
```

Source: the author.

IEEE 1666 Standard. SystemC introduces a set of characteristics for hardware description, for instance: clocking and bit-width for variables. Similar to VHDL and Verilog, SystemC is organized in modules connected to others using ports. In SystemC, all modules are executed in parallel, while methods have sequential execution. A D-FF SystemC implementation is presented in Figure 3.3.

Figure 3.3: SystemC Implementation for a D-FF.

```
#include "systemc.h"
SC_MODULE(dff) {
    sc_in_clk clock;
    sc_in  <bool> reset;
    sc_in  <bool> d;
    sc_out <bool> q;
    void ff() {
       if (reset.read())
          q.write(0);
       else
          q.write(d.read());
    }
    SC_CTOR(dff) {
        SC_METHOD(ff);
            sensitive << clock.pos() << reset;
    }
};
```

Source: the author.

Another recent high-level HDL language is the SystemVerilog. The SystemVerilog conception was standardized in 2005 by IEEE 1800 Standard. SystemVerilog is an object-oriented language derived from Verilog. It introduces a series of higher level abstractions for verification purpose, such as classes and structures. In 2009, the most recent version of Verilog was merged onto SystemVerilog, since then SystemVerilog presents all characteristics of Verilog plus itself functionalities. SystemVerilog is widely used for verification, allowing designers building elaborated testbenches using behavioural constructions very similar to real-world applications, which is hard to make using Verilog and VHDL.

The evolution of High-Level Synthesis (HLS) tools in recent years, bring back the research on C-based languages for HW (hardware) development. The complexity increase of digital systems has also boosted this design methodology. The micro-architecture exploration is a role of the HLS compiler and the designer can direct his thoughts to a systemic exploration, because state-of-the-art HLS compiler can provide the lower level optimizations which before were made by the hand of the designer.

3.2 High-Level Synthesis Tools

The beginnings of HLS design were related with the ALERT system (FRIEDMAN; YANG, 1969), developed by IBM, in the T. J. Watson Research Center, in the 1960's. The objective of ALERT system was mapping a behavioural RTL, using APL (A Programming Language) language, to logic cells. Many other academic tools were developed until the beginning of the 1980's, for example: CMUDA (DIRECTOR et al., 1981), MIMOLA (ZIMMERMANN, 1979). After 1980's hiatus, (MCFARLAND; PARKER; CAMPOSANO, 1990) presented a detailed paper which explained the methodology used in the HLS design. According (MCFARLAND; PARKER; CAMPOSANO, 1990), a hardware system is divided in "domains" regions, and each abstraction level of the system has its own domain. These domains are composed by: Behaviour, Structure and Physical domains. Table 3.1 presents an adaptation of the hardware system domains along the abstraction levels proposed by (MCFARLAND; PARKER; CAMPOSANO, 1990).

Table 3.1: Hardware Design Domains and Abstraction Levels.

	DOMAINS		
Level	Behaviour	Structure	Physical
System	Communicating Process	Processors Memories Switches	Cabinets Cables
Algorithm	Input-Output	Memory, Ports Processors	Board Floorplan
Register-Transfer	Register Transfers	ALUs, Regs Muxes, Bus	ICs Macro Cells
Logic	Logic Equations	Gates, flip-flops	Std Cell, Layout
Circuit	Network Equations	Transistors Connections	Transistor, Layout

Source: adapted from (MCFARLAND; PARKER; CAMPOSANO, 1990).

(MCFARLAND; PARKER; CAMPOSANO, 1990) define High-Level Synthesis (HLS) being a directly mapping of the behaviour specification to a RTL level structure which im-

plements that behaviour, and this high-level specification should constrain the synthesis steps as little as possible. The HLS tools should take the behaviour requirements and produce the datapath description as a set of logical operators, multiplexers, registers, etc. In this process, it is also role of the HLS tool to specify the control part (control path) of the system, in terms of finite state machines (FSMs).

It is also defined in this work the basic concepts related to HLS design, such as: tasks definitions, design space, scheduling and datapath allocation. These concepts are briefly presented below:

- **Tasks Definitions:** the synthesis tool should organize the tasks according priority and hierarchy. The input high-level language must provide mechanisms of hierarchy definition (like functions or procedures) and a way of specifying concurrent tasks. The tasks are normally represented in terms of a graph, respecting the precedence and concurrence characteristics of the input behavioural code.

- **Design Space:** a digital system can be implemented in several ways. High performance applications require more processing capacity, while other low cost applications require lower silicon area or power consumption. All these design methods compose the design space. A design space can be represented by a curve of area versus performance, for instance, where each point in the graphic represents a given implementation of the design space.

- **Scheduling:** this step is responsible to schedule the tasks. The scheduling step must detect the priority of the tasks in order to map for an specific hardware block. The data dependency is also analysed by the scheduler in order to find out operations which could be executed in parallel.

- **Datapath Allocation:** this step is responsible to map the operations into hardware operators, such as registers, adders, multipliers, etc. This step is also responsible for performing optimizations for area reduction or performance improvement, for instance.

In a more recent work, Coussy and Morawiec (COUSSY; MORAWIEC, 2008) discuss in their book, many characteristics and methods in high-level synthesis. One of them deals about high-level languages and their relations with each others. The most acceptable languages for HLS design are ANSI (American National Standards Institute) C/C++ and SystemC. Table 3.2 shows us a comparison between them. Scheduling techniques, design space exploration in HLS environments and binding techniques are also presented in this work.

In the first years of 2000's, the research in HLS reached another level. Academic research efforts produced many HLS tools around the world. In 2003, (GUPTA et al., 2003) presented the SPARK tool. SPARK takes an ANSI-C behavioural code as input, compiles and generates a synthesizable register transfer level VHDL code. In his work, the complete SPARK design flow is introduced and the results are presented for the two case studies dealt by the authors: a MPEG-1 (Moving Picture Experts Group) and GIMP (GNU Image Manipulation Program) image processing tool. The performance were increased up to 70%, without significant increase in area. Figure 3.4 shows the SPARK flow.

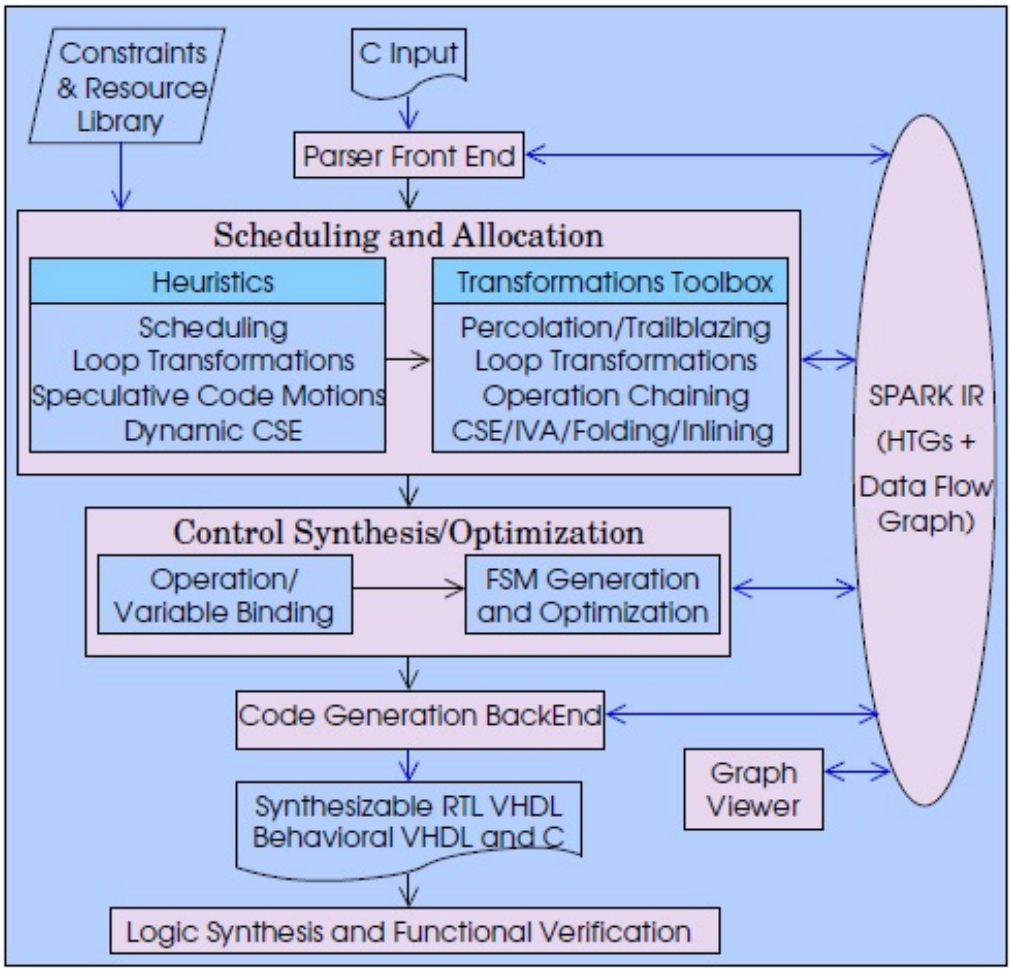
LOPASS is an academic low power HLS tool proposed by (CHEN et al., 2010) targeting FPGA-based designs. This work presents a power estimator tool, a binding register

Table 3.2: Differences between C/C++ and SystemC.

	ANSI C/C++	SystemC
Synthesizable code	Untimed C/C++	Untimed/timed SystemC
Abstraction level	Very high	High
Concurrency	Proprietary support	Standard support
Bit accuracy	Proprietary support	Standard support
Specific timing model	Very hard	Standard support
Complex interface design	Impossible	Standard support, but hard
Ease of use	Easy	Medium

Source: (COUSSY; MORAWIEC, 2008).

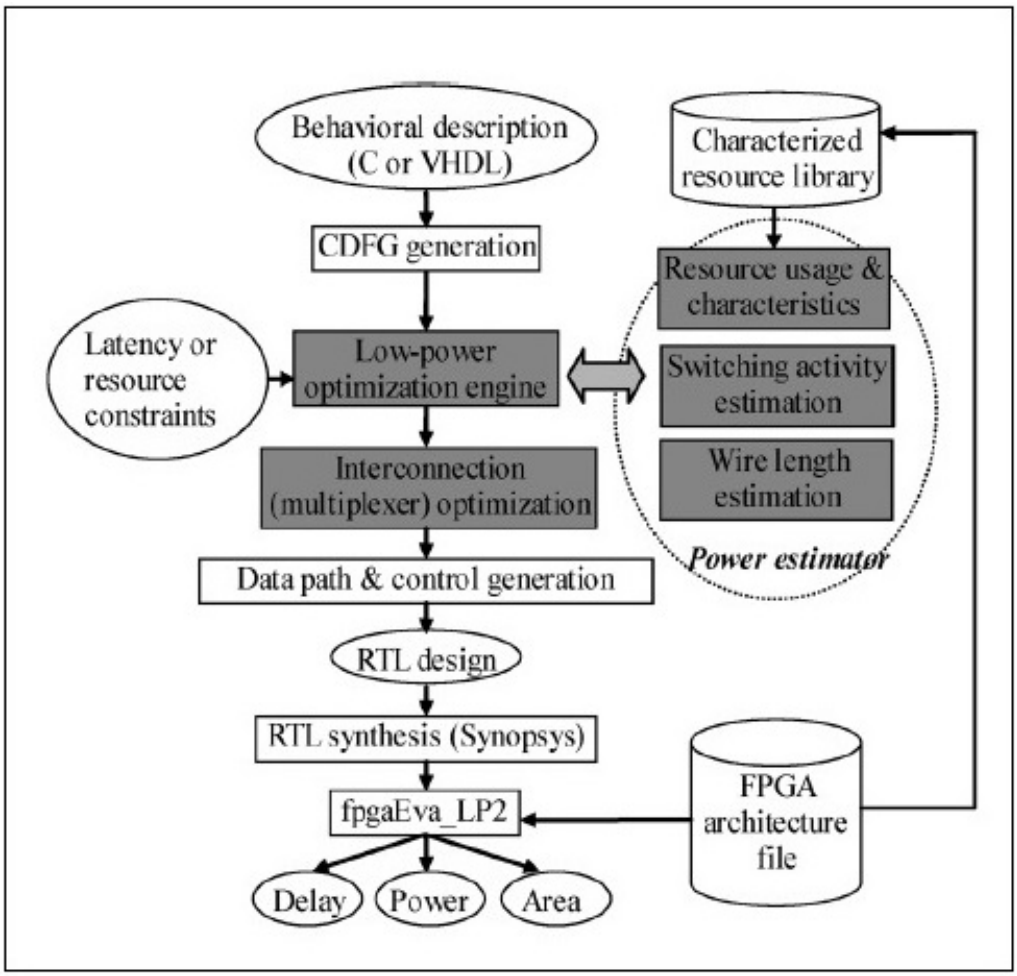
Figure 3.4: SPARK Design Flow.



Source: (GUPTA et al., 2003).

algorithm and an efficient port assignment algorithm to reduce interconnections in FPGA devices. LOPASS power results are significantly better than commercial HLS vendors, between 30% and 60% depending on the specific HLS tool considered. The area results - as measured by the FPGA LUTs and registers usage - are almost the same. The LOPASS HLS flow is shown in Figure 3.5.

Figure 3.5: LOPASS Design Flow.

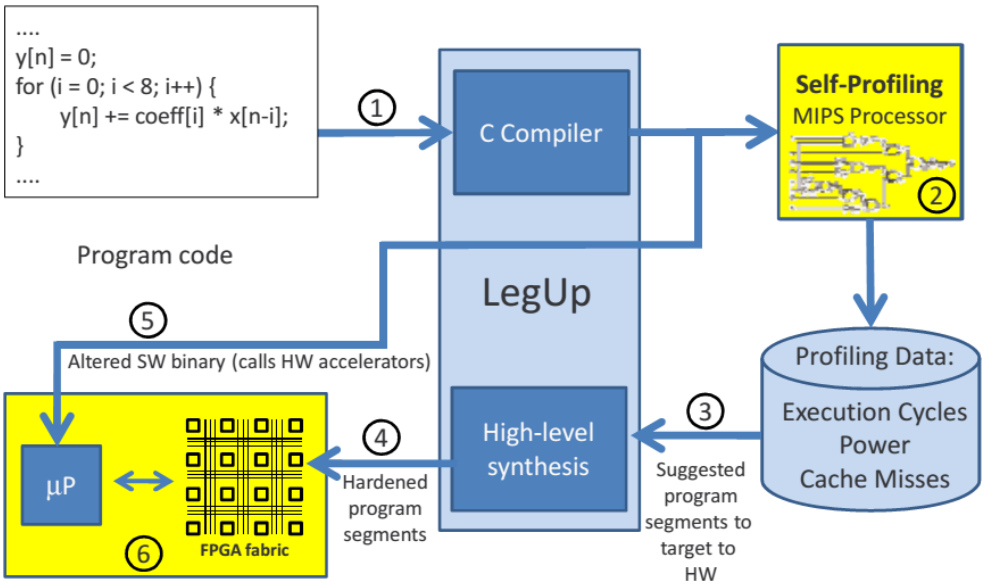


Source: (CHEN et al., 2010).

(CANIS et al., 2011) presented in 2011 a hybrid academic HLS tool called LegUp. The LegUp is an open source HLS framework, which takes a standard C algorithm as input, compiles and generates a hybrid solution based on Verilog RTL and a MIPS soft-core processor, interconnected by a bus. The results of LegUp have quality comparable with commercial HLS vendors. Figure 3.6 presents the LegUp design flow.

The major vendors of EDA tools have invested much capital in HLS tool development. Xilinx and Altera, the largest FPGA providers, have included in their packages HLS tools. In case of Xilinx, the HLS tool is the Vivado™ HLS (XILINX, 2014). The Vivado™ HLS supports C, C++ and SystemC mapped directly onto Xilinx FPGAs. The hard-IP components, like DSP macros or memories, could be used as inference directly in the high-level language. On the other hand, Altera has a similar HLS tool. The language supported by Altera SDK (Software Development Kit) is the OpenCL (Open Computing Language) (ALTERA, 2014), a standard architectural language developed for heterogeneous environments, based on concurrent parallel execution. The HLS tool of Synopsys is the Synphony C Compiler (SYNOPSYS, 2014). The supported input languages of Synphony are C and C++. According Synopsys, the speedup of development time is up to ten times compared to a hand-coded HDL.

Figure 3.6: LegUp Design Flow.



Source: (CANIS et al., 2011).

3.3 High-level Optimization Techniques

Hara (HARA et al., 2008), presents a set of benchmark codes written in C, for HLS evaluation proposal. This work shows the synthesis results for several C-based algorithms, since a MIPS processor until a JPEG (Joint Photographic Experts Group) processor. In his work, Hara was not concerned in application of any aggressive optimization technique before HLS compilation. Another contribution of his work is presenting a case study on functions transformations, as function inlining, partitioning and goto conversions, explaining pros and cons of these transformations on complexity of generated RTL codes, for instance, larger control FSMs.

Many other several works around HLS usage for FPGA-target digital design have been developed over the last ten years. Most of them address specific design trade-offs, like power, area or performance. (LHAIRECH-LEBRETON; COUSSY; MARTIN, 2010), for instance, developed a dedicated HLS flow targeted to low power FPGA design. This work proposed the reduction of clock frequency in some parts of design, reducing thus the clock tree complexity and long wires. Another power saving technique implemented in this work is the clock gating, in other words, the clock is gated not allowing switching of states in the logic blocks which are not active. The results for DSP algorithms were up to 15% of power reduction, in the multi clock domain design, compared to single clock domain implementation.

Hadjis presents in (HADJIS et al., 2012), the impact of the target FPGA device in HLS design aiming at area saving through resource sharing technique. He discussed about some code patterns to allow an efficient resource sharing. His synthesis experiments were addressed to two different FPGA devices, one of them with 4 inputs LE and other with 6 inputs. According his work, the results of 6 inputs LE FPGA were better than 4 inputs FPGA, due to ability of the synthesis tool of mapping muxes and operators at same LE, reducing LUTs under-utilization.

In 2011, (ROSADO-MUñOZ et al., 2011) presented two approaches for FPGA adaptive noise filter implementation. One of them is based on hand-coded VHDL and another using HLS flow. The quality of both implementations is basically the same, while the code complexity in the hand-coded VHDL method is higher. The area results of both solutions are almost the same, but the performance for three different FPGA devices, Xilinx Spartan™ 3, Xilinx Virtex™ 4 and Xilinx Virtex™ 5, is two times better in hand-made VHDL code.

Similar to (ROSADO-MUñOZ et al., 2011), (SANCHEZ et al., 2013) proposed a comparative study between HLS methods and HDL implementation of a grid synchronization algorithm. In this study, the hand-coded HDL implementation is superior in the area utilization. However, the usage of hard-IP blocks - DSP blocks and block RAMs - were better in the HLS implementation mode, due to the resource re-utilization heuristic of the HLS tool, in this case Vivado™ HLS, which could explain the large area consumption.

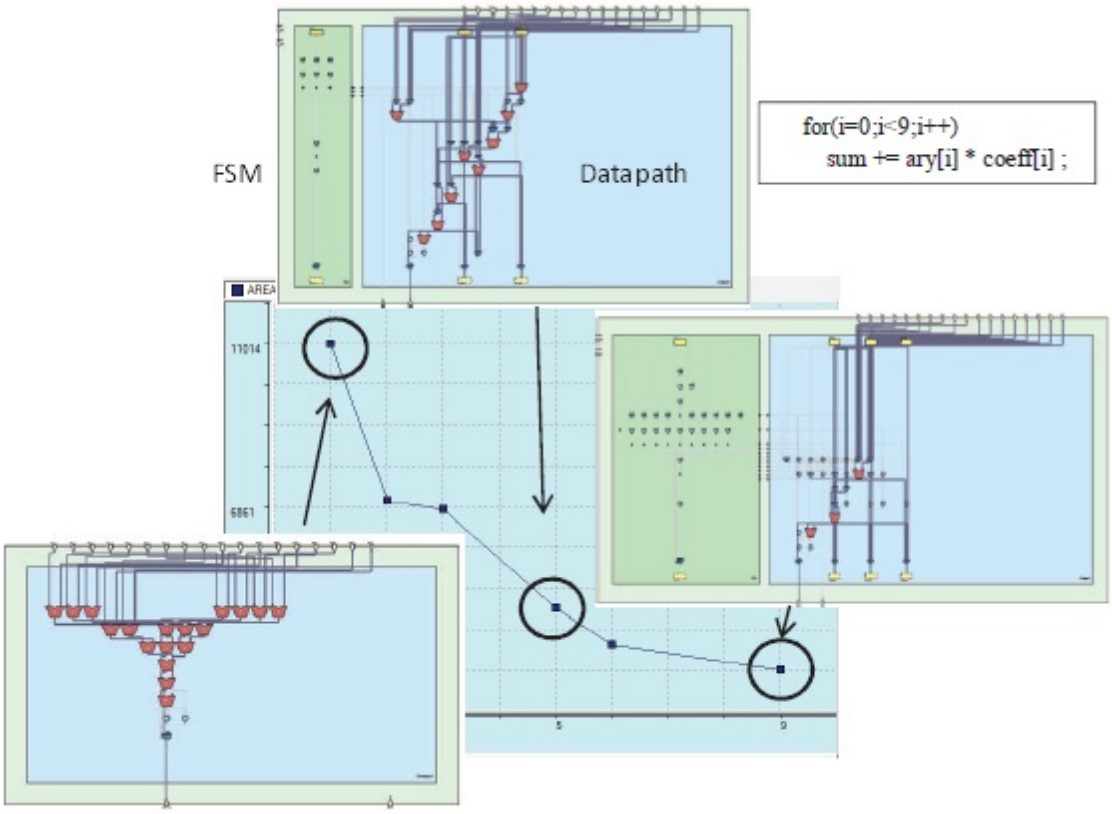
Aiming at area saving, (SCHAFER, 2014) presented in his work an optimized allocation technique for FPGA DSP-blocks. In this this paper, the proposed method is targeted to multi-process DSP algorithms. Basically, the allocation algorithm places the DSP-blocks in optimized locations for hardware re-utilization purpose and area minimization. The design space exploration of used HLS tool is presented in terms of area and performance. That is illustrated by Figure 3.7. In the figure the area-performance curve for three different space explorations are shown in the background. The results, in terms of area, were up to 12.90% of reduction compared to direct/auto DSP-block allocation.

In (GURUMANI et al., 2013) the authors proposed an implementation of multiple dependent CUDA (Compute Unified Device Architecture) Kernels in FPGA using HLS techniques. In this work a comparison between the HLS implementation and a GPU (Graphic Processing Unit) based implementation is presented. (GURUMANI et al., 2013) showed the HLS flow for CUDA kernels implementation. A step-by-step design flow and the challenges in the automatic CUDA kernel synthesis were also presented. The results with HLS implementation reduced up to sixteen times the energy consumption compared to a GPU implementation. The performance obtained for both implementations are almost the same. The HLS tool used in this work was the AutoPilot-C, the FPGA synthesis was made with Xilinx Vivado™. The FPGA device target in their experiments was the Xilinx Virtex™ 7.

Wakabayashi presents in (WAKABAYASHI; TAKENAKA; INOUE, 2014), a HLS and CPUs (Central Processing Unit) comparison in terms of compiler point of view. In his work, he showed the compiler optimization possibilities in HLS environments. Tasks scheduling, pipeline and loop unrolling - limited for CPUs - are easily performed in high-level design mapped to FPGA devices. One example of these techniques is shown in Figure 3.8.

Recent efforts have addressed the pipelining exploration in HLS environments. Most of the HLS tools provide pipelining optimizations including: modulo scheduling, memory port reduction and polyhedral analysis. Although, the HLS compilers have to insert stalls in pipelines when some operation takes mores time than average, for instance when a cache miss is detected and it is necessary to access an external memory. Tan presented in (TAN et al., 2014), an approach for efficient pipelining synthesis for data-parallel kernels. He used the concept of context data switching using a context buffer to perform out-of-order kernel processing. Using buffer context is possible to avoid stalls due to latency processing of a pipeline stage, where regular pipelining synthesis stalls all stages, causing

Figure 3.7: FIR Filter Design Space Exploration.



Source: (SCHAFER, 2014).

reduction in the data processing throughput. He also presented a proposal of buffer micro-architecture with a low cost scheduler for the storage kernels. Figure 3.9 presents the context buffer in a kernel graph. His results in terms of throughput are up to 17.5x when compared to a baseline HLS kernel synthesis.

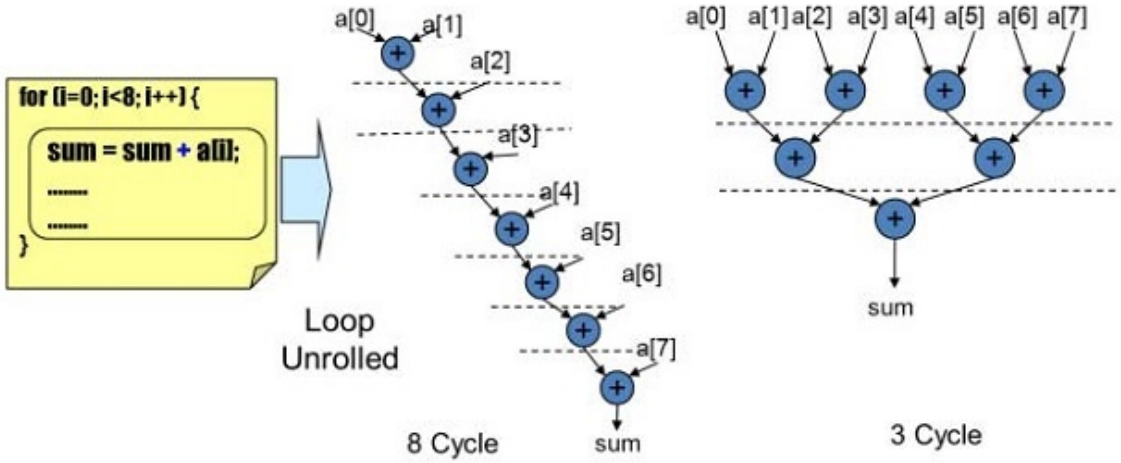
3.4 HLS tools and Methods Summary

This chapter reviewed the well-known methods for design speedup. The first section revisited the evolution of the HDL languages over the last 40 years. The two more important languages, VHDL and Verilog, are presented in more details. The high-level languages introduced in the beginnings of 2000's, SystemC and SystemVerilog, were revised.

In the second section, we revisited the academic and commercial HLS tools. We introduced some basic concepts in HLS design and we reviewed a set of academic HLS tools in details. The SPARK, LOPASS and LegUP HLS flow were presented, as well as, the state-of-the-art commercial HLS tools.

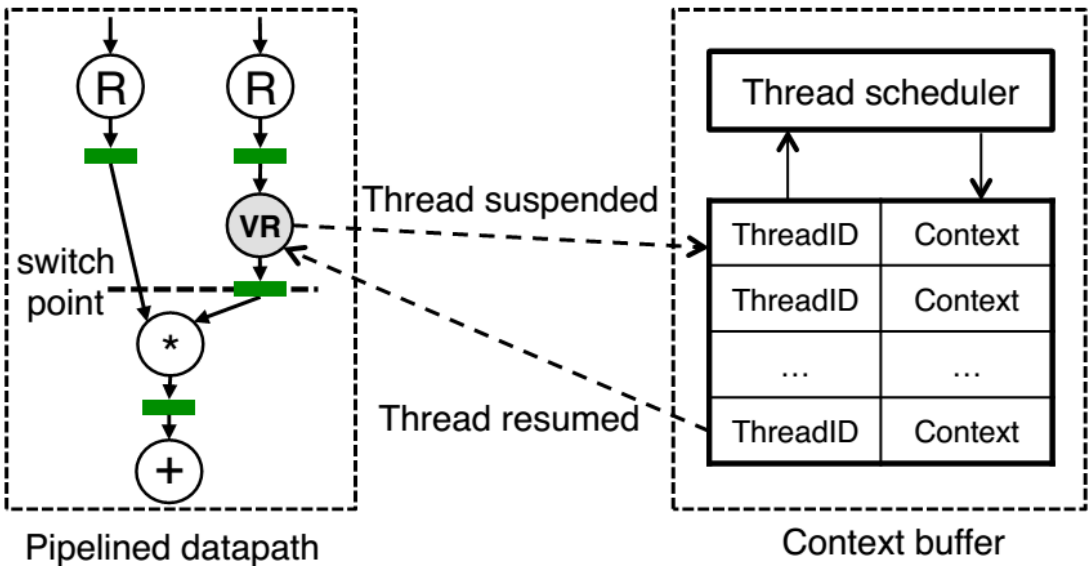
The third section presented the high-level methods with HLS. We presented a literature review for design optimizations with HLS tools. These methods include high-level code transformations, pipelining approaches and resource sharing techniques were presented.

Figure 3.8: Loop Unrolling. a) Loop Pseudo-code. b) Unrolled Data Flow. c) Tree Reduction Structure.



Source: (WAKABAYASHI; TAKENAKA; INOUE, 2014).

Figure 3.9: Kernel Graph. R represents a memory reading while VR represents an variable latency for an external memory access.



Source: (TAN et al., 2014).

4 DESIGN SPACE EXPLORATION IN HARDWARE SYSTEMS

4.1 Introduction

Design space exploration (DSE), in hardware systems, refers to design trade-offs balancing for the best implementation cost-benefit. These trade-offs for hardware systems normally include at least three variables: silicon area, power consumption and performance. Project time, or time-to-market, could be added to DSE variables for commercial applications. At end, the main purpose of DSE in digital systems design is reduce resource investment. With a smaller circuit is possible to manufacture/buy a smaller and cheaper chip. High performance applications require modern technologies to achieve high frequencies, in other words, more expansive integrated circuits. Power-hunger designs require stronger power supplies, which means spending more money. Summarizing, optimizations in any of main variables mean costs reduction.

4.2 Architectural Exploration

Palermo presented in (PALERMO; SILVANO; ZACCARIA, 2003) his definition for design space. In his work, he defined design space as a set of all possible architectural implementations in a specific platform. Any possible configuration for an architecture is defined as a geometric point in the design space. This point, in the multidimensional space, is a vector $a \in A$, being A the architectural space defined as:

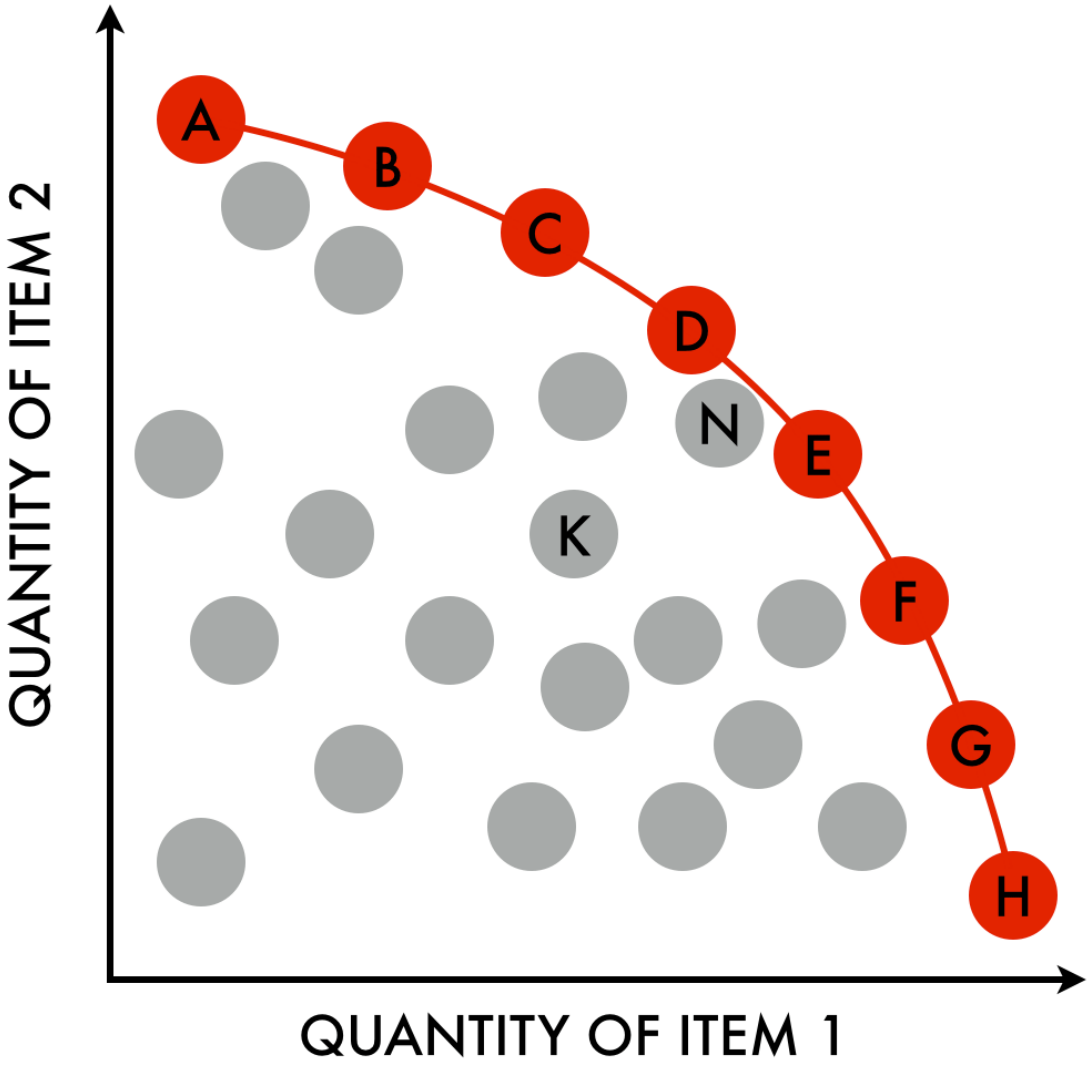
$$A = S_{p1} \times ...S_{pl}... \times S_{pn} \tag{4.1}$$

where S_{pn} is a set of possible configurations for the parameter pn. Each point has a set of metrics associated. These metrics compose a multi-dimensional space named design evaluation space. He also introduced the concept of Pareto point, which it is the point in the design space where it is impossible to make an individual objective better without making at least one individual objective worse. A set of Pareto points form the trade-off curve or Pareto curve. Figure 4.1 presents a example of trade-off curve.

Pareto curves are widely used in many different knowledge areas - for instance Economics, Engineering and Life Sciences - to represent cost-benefit relations. In HW systems, the Pareto curve represents the best cost-benefit for a specific architecture. The area of the graphic under the trade-off curve we consider the entire design space.

In other work, Palermo presented in (PALERMO; SILVANO; ZACCARIA, 2005) a multi-objective DSE applied to embedded systems. He discussed in this work about a multi-objective DSE, in this case energy and delay. He also presented a framework for

Figure 4.1: Pareto Curve.



Source: Wikipedia, the free encyclopedia.

for DSE in order to approximate the Pareto optimality. This framework presents a composition of three Pareto approximation algorithms, basically derived from *Monte Carlo* analysis.

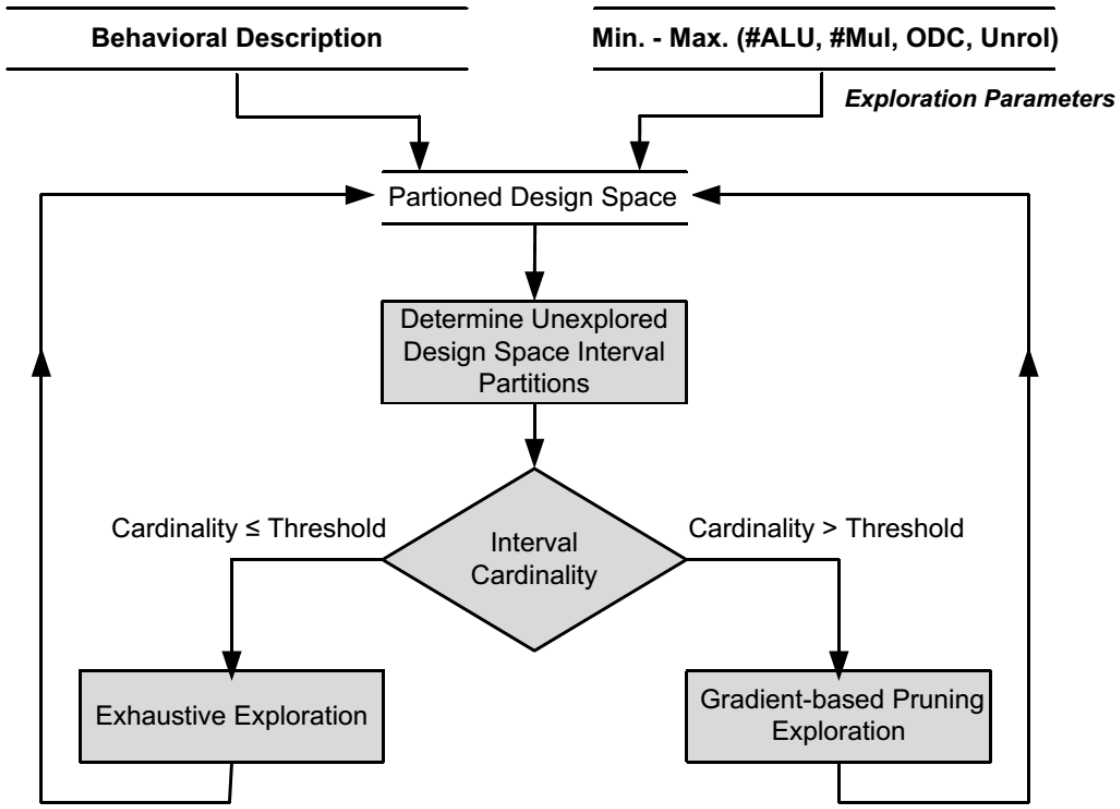
In the literature, we can find many works on DSE applied to digital systems. Most of them present methodologies for high efficient DSE, which include exploration algorithms and efficient hardware design partitioning. Sotiropoulou, for example, presented in (SOTIROPOULOU; NIKOLAIDIS, 2010) a methodology for DSE applied to FPGA-based multiprocessing systems. In her work, she presented an efficient JPEG algorithm partitioning in processors implemented in an FPGA, allowing data and task-level parallelism for each processor. She presented four different multiprocessing architectures, each one optimized for a specific design exploration. She also defined a relation called HW efficiency, presented as follows:

$$HW_{eff} = \frac{SpeedUp}{Area_{inc}} \tag{4.2}$$

where the $SpeedUp$ represents the performance improvement and $Area_{inc}$ represents the area increasing for that $SpeedUp$ result.

High-level synthesis researchers have studied how to apply DSE concepts in their designs. Xydis discussed in (XYDIS et al., 2010), a methodology for design space exploration with HLS using a combination of exhaustive exploration and gradient-based pruned searching. In his work, he proposed an iterative DSE technique using both exhaustive and heuristic methods. In Figure 4.2 is shown the DSE flow methodology of his work. In his heuristic algorithm, he defines the space boundary - as number of adders or multipliers - in order to define the Pareto curve for the implemented architecture. The Pareto curve is extracted iterating step-by-step on boundaries variables.

Figure 4.2: DSE methodology by Xydis.

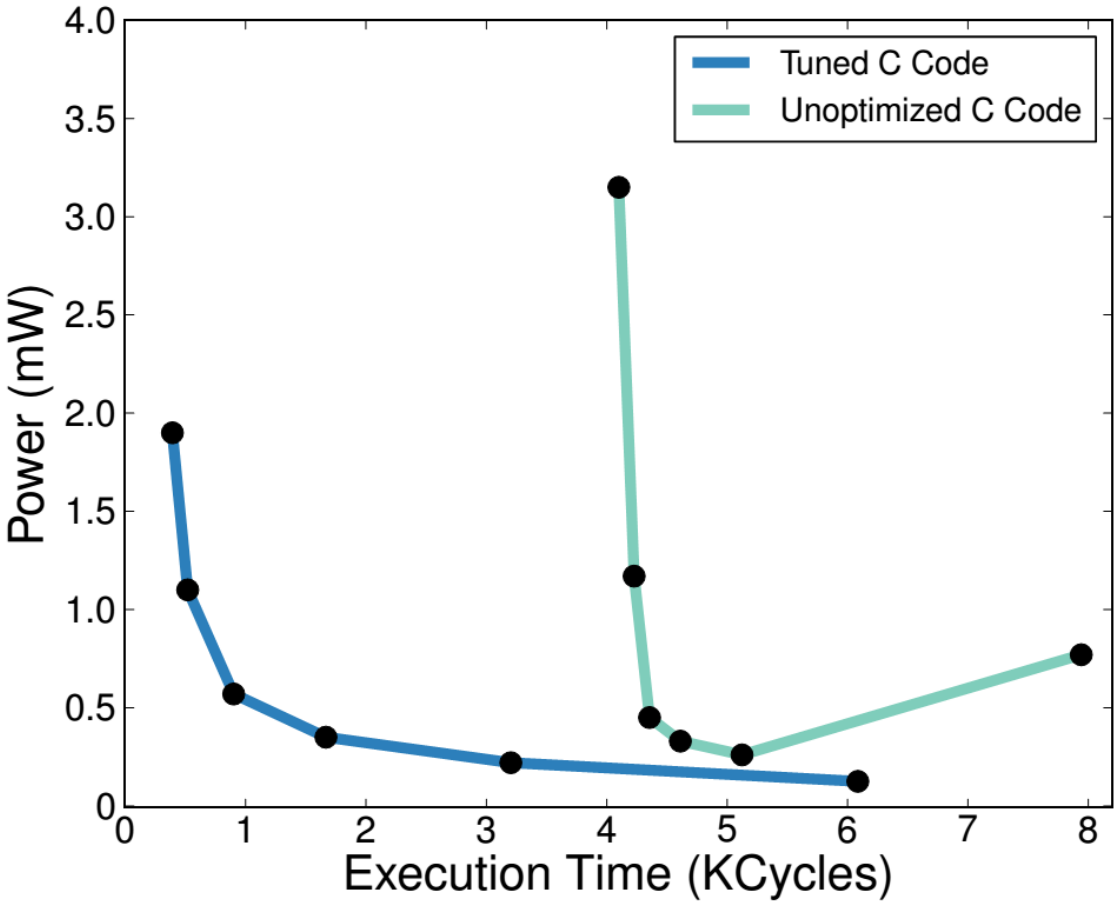


Source: (XYDIS et al., 2010).

In (SHAO et al., 2014), Shao proposed a framework for power-performance simulation allowing large DSE for custom architectures. In her work, she discusses about dynamic data dependence graphs on digital designs. Her framework is composed by two phases: optimization and realization phases. On optimization phase, the framework detects and performs the code optimizations, as DDDG (Dynamic Data Dependence Graphs) and architectures optimizations (node/loop unrolling and memory accesses). The realization phase performs the power-performance estimation based on input constraints, data/-

control dependencies, etc. She also discusses about code tuning and presented a comparison between tuned/untuned C codes, which is presented in Figure 4.3. Code tuning is a key factor in HLS design flow, allowing hardware parallelism, pipelining and false data/control dependencies removal.

Figure 4.3: Tuned and Untuned C code comparison.



Source: (SHAO et al., 2014).

4.3 Conclusions

HLS methods can help considerably the design space exploration. The basics of this exploration is introduced in this chapter. The importance of optimizing the input high-level description was emphasized, based on previous works presented in the literature.

5 METHODOLOGY PROPOSED FOR ARCHITECTURAL EXPLORATION

This chapter presents the methodology used for this work. The current work targets the high-level architectural exploration, thereby our methodology is mainly focused in two high-level methods: high-level code tuning and an iterative method for design space exploration. The first method was based on similar works (SOTIROPOULOU; NIKOLAIDIS, 2010; SHAO et al., 2014) present in the literature. The framework we developed is based on Xydis work (XYDIS et al., 2010). Next sections will present these methods in details.

5.1 High-Level Code Tuning

High-level code abstractions, normally, are not well welcome in HLS tools. However, high abstraction is common in C-based languages and this property makes the HLS development easier for fast turn-around purpose. To fill this gap, we have to adapt the source code to get the expected results. These code transformations include: efficient code partitioning and appropriate HLS directives exploration.

C-based codes are serially executed. As an FPGA designer, we have to translate this serial code onto an FPGA-targeted parallel code. A transformation we studied in this work was the efficient code partitioning (PELLERIN; THIBAULT, 2005). A well partitioned code allows the HLS tool to interpret correctly the source code for proper optimizations. Techniques as loop pipelining, loop unrolling and resource sharing are possible using this method. The technique we implemented in this work detects code snippets working as a block and isolate them in a function. Figure 5.1 shows an example of code transformation on a kernel C-based snippet code. This code transformation allows to drive the synthesis for at lest for two parameters: area reduction and performance improvement. For area reduction, the synthesis tool implements only one hardware block for the function *kernel*. Aiming at performance increase, the synthesis process implements the necessary number of *kernel* hardware modules for maximum parallelism.

Code partitioning and HLS parameters exploration are closely related. On Vivado™ HLS, the implementation parameters could be inserted directly on the source code - using pragmas - or through a constraint script. With the source code functionally divided, the HLS parameters insertion can be more assertive. For performance purpose, the more useful parameters are: pipelining, loop unrolling, array partitioning and datapath. When the target is area, common used parameters are inlining functions and resource sharing. As example of HLS parameters usage, we present the snippet code below extracted from our C-based VLIW processor implementation, based on Wong work (WONG; AS; BROWN,

Figure 5.1: Code transformation example: Original code, on left. On right, code partitioned.

```
void foo () {
    for (int i = 1; i < 10; i++)
        b[i] += b[i-1]*a[i];
    for (int i = 1; i < 10; i++)
        d[i] += d[i-1]*c[i];
    for (int i = 1; i < 10; i++)
        f[i] += f[i-1]*e[i];
}
```

```
void kernel (int *a, int *b) {
    for (int i = 1; i < 10; i++)
        b[i] += b[i-1]*a[i];
}
void foo () {
    kernel(a, b);
    kernel(c, d);
    kernel(e, f);
}
```

Source: the author.

2008):

Figure 5.2: Code Snippet Example of a VLIW Implementaion.

```
void execute(...)
{
...
    /* Perform ALU operations */
    L2: for (i = 0; i < ALU_NUM; i++) {
        alu(op_code[i], A_op[i], B_op[i], &alu_result[i]);
    }
...
}
```

Source: the author.

For area optimization, we used the snippet script below:

```
set_directive_allocation -limit 1 -type function "execute/L2" alu
```

This directive forces the HLS tool to implement just one *alu* function. This function is translated to just one RTL block, which is shared for each loop iteration. This implementation reduces area, but increases the execution time by a factor of the integer *ALU_NUM*.

For speed optimization, we used the snippet script below:

```
set_directive_allocation -limit ALU_NUM -type function "execute/L2" alu
set_directive_unroll "execute/L2"
```

Using the script above, the *alu* is implemented *ALU_NUM* times. This code has no data dependency, thus, the function *alu* is translated to *ALU_NUM* hardware blocks, which are executed in parallel. The area overhead increases by a factor *ALU_NUM*.

5.2 Iterative Design Space Exploration Method with High-level Synthesis

The methods discussed in the Section 5.1 are tedious to exploit all possible combinations and have a lower efficiency if applied manually. To support these methods, we implemented an iterative DSE framework. This method allows a quick response in terms of trade-off analysis.

In this section we present the DSE methodology adopted in our DSE framework. Our framework explores efficiently the design space aiming at achieving better area results compared to non-guided HLS flow. This framework is targeted to Xilinx FPGAs and the framework result is the best directives file for the Vivado™ HLS tool, in a TCL (Tool Command Language) format. It is important to highlight that our method does not provide any architectural transformation in the source code, it only discovers the proper set of HLS directives for efficient DSE. Hence, it is a fact that a true design space exploration in general terms requires a more complex, designer-driven exploration.

Our framework is an iterative, recursive and multi-platform method, developed in a free and high-level language, called Lua (LUA, 2014). The Lua executable is available for download at `http://www.lua.org/download.html`.

We used as a basis for this framework implementation the work proposed by Xydis in (XYDIS et al., 2010). The Xydis work was already discussed in the Chapter 4.

Figure 5.3 presents a simplified view of our framework flow for DSE, which uses the Vivado™ commercial tool at its core as an engine to be driven by our strategy for DSE. In the next sub-chapters, we will discuss the implementation of our automatic DSE method.

5.2.1 High-level Design Entry

The current version of our DSE framework supports as input languages both ANSI C and C++. Some code guidelines should be observed for proper functioning of the DSE script. These guidelines include:

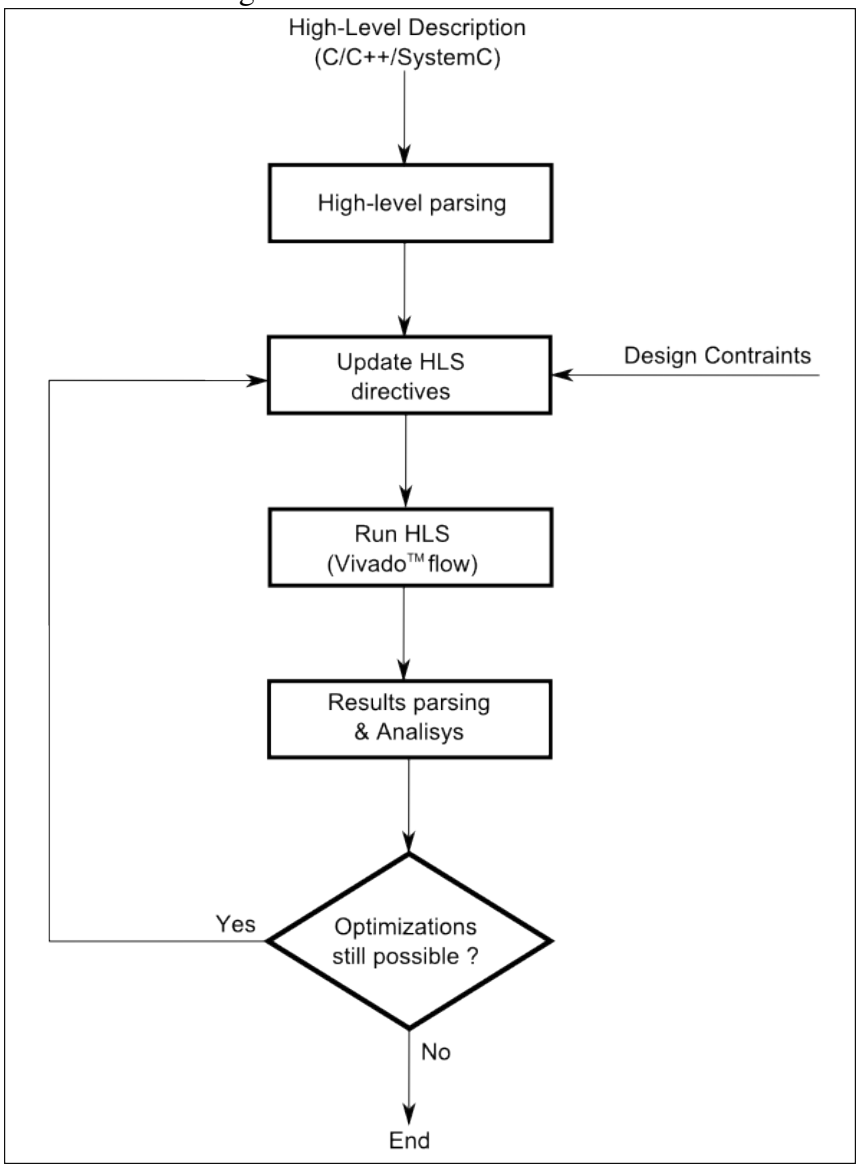
- Loops labelling, for instance: Loop1: for (int i = 0; i <10; i++);
- Sequential and conditional blocks markers, the '{' character, should be placed on a new line;
- Array declarations should be written just one per line;
- Returning function types cannot be a defined type. The input high-level description supports only language native type, such as: void, int, short int, etc.

Furthermore, for maximization of optimization results, a well partitioned high-level code is extremely necessary. The code writing guidelines are important to steer the HLS tool through the steps of synthesis. A snippet C code example is shown in Figure 5.4.

5.2.2 High-Level Parsing

High-level parsing step is responsible for detecting key points in the C/C++ code where some optimization directive can be inserted. Our DSE script is capable of detecting functions and arrays declarations, loops and multiple function execution. As a start, we have decided to detect these kind of checkpoints, due to the wide possibilities for area optimizations.

Figure 5.3: DSE Framework Flow.



Source: the author.

For this step, we had to work with regular expressions and a searching algorithm to find out the correct points for design optimization. The high-level parsing algorithm had to be able to determine when multiple blocks are executed inside others. To synthesize area-efficient hardware with HLS, it is necessary to discover the exact number of possible functions execution.

Observing snippet code shown in Figure 5.4, our framework is able to detect the checkpoints below:

- Function declarations: *r_vex_core* and *execute*;
- Loops execution: L2;
- Array declarations: *op_code*, *A_op*, *B_op* and *alu_result*;
- Multiple function execution: *ALU_NUM*execute*.

Figure 5.4: Example Code for ALU operation.

```
void execute(char *op_code, int *alu_result)
{
    int A_op[SYLLABLE_NUM];
    int B_op[SYLLABLE_NUM];
...
    /* Perform ALU operations */
    L2: for (i = 0; i < ALU_NUM; i++) {
        alu(op_code[i], A_op[i], B_op[i], &alu_result[i]);
    }
...
}

void r_vex_core ()
{
    char op_code[ALU_NUM];
    int alu_result[ALU_NUM];
...
    execute(op_code, alu_result);
...
}
```

Source: the author.

5.2.3 Design Constraints

For obtaining convergence for an optimized and acceptable synthesis in Figure 5.3, we need to define some design constraints. These constraints are the guide for our iterative method. As our trade-off is area, the DSE script uses the following design constraints:

- Minimum and maximum function/loop execution time in clock cycles;
- Minimum and maximum resource allocation at function granularity;
- Array partitioning threshold. The threshold value is the limit for a single array. Larger arrays will be partitioned into smaller arrays, where the maximum dimension is the threshold value.

We have decided not to define or set an allocation constraint for FPGA operators - adders/subtracters, comparators, multipliers - due to the high cost of large data-width multiplexers in FPGAs. This decision for not constraining the operators was based on the finding that Vivado™ HLS gets better area reduction by letting function sharing instead of operation sharing.

5.2.4 Update HLS Directives

Exploring Vivado™ HLS optimization directives (XILINX, 2014) is fundamental for this framework success. Using the checkpoints detected in Section 5.2.2, we insert the appropriate directives for each checkpoint. The relation between code checkpoints and HLS directives are explained as follows:

- Array declaration: array partitioning directive. This directive defines the partitioning and memory resource allocation for an array;

- Function declaration: code inline directive. Inline a code snippet;
- Loop detection: execution time directive. Set a defined latency for execution, removing pipelining inferring;
- Multiple function execution: resource allocation directive. Allows HW resource sharing.

5.2.5 Results Parsing and Analysis

The criterion of acceptance for our DSE framework is better area results. We assume as area elements the number of FFs and LUTs. The area consumption information is extracted from the parsing of the synthesis report file. Regular expressions were used to detect LUTs, FFs, latency and maximum frequency from report file. Being an iterative method, our stop criterion is the convergence of the DSE script, in other words, we stop when the maximum area optimization was achieved for the target design ($Area_{iteration\ n} > Area_{iteration\ n-1}$).

5.3 Methodology Summary

This chapter presented the methodology proposed in this work. This first section discuss about high-level code transformation aiming at Quality of Results (QoR) improvement. We presented a code partitioning methodology and how to explore the HLS compilation directives.

We also presented a automatic DSE method for area reduction purpose. We describe all steps and requirements for the convergence of the proposed framework. This method is described in a high-level script language and it is compatible with Vivado$_{TM}$ HLS Compiler. Out method trades-off area and performance aiming at achieving the best QoR for the given implementation. At this moment we are not concerned in the power impacts of the propose DSE method, however, reducing the area usage also contributes to power savings.

6 HIGH-LEVEL SYNTHESIS EXPERIMENTS

To support this study, this chapter presents the results on HLS design flow. The results chapter is in two parts: i) HLS tools and design methods comparison and ii) design space exploration in HLS environment using the method discussed in the Chapter 5. In the first section, we present the comparison between two different HLS tools: an academic tool called LegUp and the commercial Vivado™ HLS. In this section, we also compare the results of the HLS flow with hand-coded design for each test-case. For second section, we present the results of design exploration methods on HLS. At this time we are only able to present the DSE methods results for Vivado™ HLS flow.

6.1 High-Level Synthesis Tools and Design Methods Comparison Results

This section presents the HLS compilers comparison results for HLS design. It is also presented in this section the comparison between high-level description flow against hand-coded HDL design flow. For results evaluation we used four test cases:

- a MIPS processor;
- an integer square root algorithm;
- a VLIW processor;
- a FIR filter.

Next sub-sections describe the experiments and their results.

6.1.1 MIPS processor

This experiment aims at comparing two different implementations for a MIPS processor. As discussed previously, the MIPS architecture was introduced by Patterson (PATTERSON; HENNESSY, 2007) in the 1980's and this processor is still usual for performance evaluation purpose. A high-level MIPS processor, proposed by Hara in (HARA et al., 2008), written in C language and a VHDL implementation were compared in terms of area, CT (Compute Time), frequency and code complexity. The MIPS processor used as benchmark, for both C/C++ and VHDL, is a monocycle machine, with no cache and no external memory. The hand-code VHDL was developed by the author for this project. For this experiment we used the LegUp compiler (CANIS et al., 2011) as HLS tool,

Quartus™ II as synthesis tool and the FPGA target was a Cyclone™ IV. Both, synthesis tool and FPGA device are provided by Altera Inc. The results of this experiment are shown in Table 6.1.

For performance evaluation, we considered CT as the time necessary for a test program execution. The test program used as benchmark was a bubble-sort algorithm. The assumed code complexity metric is the sum of the lines of code of all source files used for this experiment, excluding comment lines. The area metric was measured in terms of LE, which in case of Cyclone™ IV FPGA has one four inputs LUT (Look-up Table) and one FF.

Table 6.1: MIPS Implementation Comparison.

Method	**Area (LE)**	**CT (cycles)**	**Frequency (MHz)**	**Code Complexity (lines of code)**
C/C++	4579	5472	72.22	320
Hand-coded VHDL	9070	611	44.92	1200

The results shown in the Table 6.1 suggest:

1. Area reduction in HLS method: That suggests that the HLS tool has an area oriented mapping algorithm, including resources sharing and re-utilization.

2. Better performance in hand-coded method: even with the higher frequency on HLS method, the CT is much shorter in the microprocessor obtained by hand-coded RTL design. This could be explained because of HLS tool heuristics for area reduction.

3. Lower complexity using C language: high-level constructions make the code easier and shorter compared to RTL design. RTL descriptions normally require bit-level accuracy, while in high-level it is not necessary. Some code guidelines are required on hand-coded RTL design, such latch-avoid techniques and careful FSM implementations. These techniques are automatically performed by HLS tool.

6.1.2 32 bit Integer Square Root Algorithm

Four different implementations for a 32 bits Integer Square Root Algorithm were done for RTL hand-made versus HLS comparison. The algorithm used for this experiment is shown in Figure 6.1. Both RTL and high-level implementations were provided by author for this work.

The algorithm shown in Figure 6.1 was directly used as input of HLS tool using C language. Three different VHDL implementation were proposed: a non-optimized description, an area-oriented description and a performance-oriented description.

In this experiment, we used the LegUp C compiler for HLS flow, Quartus™ II and Cyclone™ IV as FPGA device. For performance comparison we considered CT as the time necessary for a 32 bit integer square root calculation, in this case it was the worst case input interger: 0xFFFFFFFF. The code complexity metric was the same used in the Section 6.1.1. Table 6.2 shows the results obtained for these different implementations.

According Table 6.2 we can conclude:

1. Similarity between VHDL SW (Software) pipeline implementation and HLS method: That suggests the HLS tool has some datapath optimization technique. Pipeline is a very common technique in datapath oriented algorithms.

Figure 6.1: Square Root Algorithm Pseudo-code.

```
function SQRT(I)
    r ← 1
    d ← 2
    s ← 4
    while s − I ≤ 0 do
        r ← r + 1
        d ← d + 2
        s ← s + d + 1
    end while
    return r
end function
```

Source: the author.

Table 6.2: Square Root Algorithm Implementation Comparison.

Method	**Area (LE)**	**CT (cycles)**	**Frequency (MHz)**	**Code Complexity (lines of code)**
C/C++	255	46343	123.9	15
Hand-coded VHDL	180	185368	311.14	240
Hand-coded VHDL (HW re-utilization)	101	231707	308.93	200
Hand-coded VHDL (SW Pipeline)	227	46353	233.48	230

2. Worst performance in VHDL, excepts with SW pipeline implementation: even achieving higher frequency, the VHDL techniques are much worse than HLS. The area reduction is not good enough to compensate the worst performance.

3. Code complexity: all three VHDL implementations have more than 10 the times number of lines of code compared to high-level implementation.

6.1.3 VLIW Processor

In this experiment, we implemented an academic VLIW processor aiming at comparing HLS tools. The chosen VLIW processor was the ρ-Vex processor, designed by Wong in (WONG; AS; BROWN, 2008). The RTL model is open source available for download at `https://code.google.com/p/r-vex/`. We developed a high-level implementation for ρ-Vex processor using C language and this code was used as design entry for both HLS tools. The processor implemented was a four issue VLIW architecture, according to the scheme already presented in the Section 2.3. The high-level input C code is available in the Appendix B.1.

The main purpose of this experiment was the HLS tools comparison. For this, we used the LegUp C HLS compiler and the Vivado™ HLS compiler, provided by Xilinx Inc. For performance analysis, we considered CT as the worst case instruction compute time, for this case a 32 bits multiplication. The code complexity metric was the same used in the Section 6.1.1. In this experiment we considered for area consumption analysis not just logic elements, but also memory blocks and DSP blocks, due to processor complexity.

The LegUp results are shown in the Table 6.3. The synthesis tool used in this experiments was Quartus™ II and the target FPGA was a Cyclone™ IV. Each RAM block on

a Cyclone™ IV has storage capacity of 8 kb. The multipliers on this device can operate inputs with 9 bits width.

Table 6.3: ρ-Vex Synthesis Results Using LegUp Compiler.

Method	Area (LE)	DSP Blocks	RAM Blocks	CT (cycles)	Frequency (MHz)	Code Complexity (lines of code)
C/C++	7122	24	29	514	123.9	801
Hand-coded VHDL	1771	2	0	7	311.14	4359

In the Table 6.4 are presented the synthesis results obtained using Vivado™ HLS. As synthesis tool we used Vivado™ and the target FPGA was an Spartan™ 6. Both synthesis tool and FPGA device are provided by Xilinx Inc. The area metric used in this experiment was LUTs and FFs. Spartan™ 6 has 6 input LUT, and its RAM blocks have 16 kb of storage capacity and the embedded multipliers can operate 18 bits inputs.

Table 6.4: ρ-Vex Synthesis Results Using Vivado™ HLS.

Method	Area (LUT)	Area (FF)	DSP Blocks	RAM Blocks	CT (cycles)	Frequency (MHz)	Code Complexity (lines of code)
C/C++	6765	12535	5	2	300	333	801
Hand-coded VHDL	4944	1765	10	1	7	119.64	4359

Comparing Table 6.3 and Table 6.4 we can conclude about each HLS tool:

1. LUTs consumption similarity with both tools: resource allocation mechanism is similar on both LegUp and Vivado™.

2. Large RAM blocks usage with LegUp: the schedule algorithm could explain large consumption. It is necessary memories to store data context for time scheduling.

3. Large DSP blocks consumption with LegUp: hard blocks allocation technique is worse than Vivado™, although FFs and LUTs allocation mechanism are better.

4. Better performance on Vivado™: summarizing previous two topics is clear the performance results. DSP and RAM macros have a fixed latency, around 3 ns, which explain the best performance on Vivado™.

6.1.4 12th-order FIR Filter

In this experiment, we implemented a 12-tap FIR filter. The main goal of this experiment is to compare two different HLS tools: LegUp and Vivado™ HLS. For this experiment, we used a C code provided by Xilinx Inc. The source code could be downloaded at Vivado™ HLS tutorials web page. The pseudo-code for this FIR filter is shown in Figure 6.2.

We also synthesized a 12-tap FIR filter in VHDL. The implementation is described by Meyer-Baese in (MEYER-BAESE, 2001). In his book, Meyer-Baese describes a generic filter implementation, where parameters as input and output data width, tap number and multiplier pipeline stage number can be customized. The original code only supports Altera devices, we modified the VHDL code to support both Xilinx and Altera devices.

Figure 6.2: FIR Filter Pseudo-code.

```
function FIR(x, c[12])
    acc ← 0
    data ← 0
    shiftreg[12] ← 0
    i ← 11
    for i ≥ 0 do
        if i = 0 then
            shiftreg[i] ← x
            data ← x
        else
            shiftreg[i] ← shiftreg[i − 1]
            data ← shiftreg[i]
        end if
        acc ← acc + data ∗ c[i]
        i ← i − 1
    end for
    return acc
end function
```

Source: the author.

It is important emphasizing that RTL optimizations were not performed, because it is not the purpose of this implementation, it is used only for propose of comparison.

A FIR filter implementation, proposed by Mirzaei in (MIRZAEI; HOSANGADI; KASTNER, 2006), was also used as benchmark comparison. In his work, Mirzaei proposes a multiplier-less 12-bits FIR filter implementation, based on add-shift method. He presented multiples FIR filter sizes, but we will compare only the 13-tap filter, which could be considered a good approximation to our filters. Furthermore, we considered only the area results, in terms of LUTs/FFs, because the FPGA device used in this work is legacy, thereby the performance or power results cannot be considered.

In Table 6.5, we present the comparison between two implementations: Meyer-Baese RTL based and a high-level implementation with LegUp. Both implementations were targeted to Cyclone™ IV FPGA device and were synthesized with Quartus™ II. Comparison metrics were the same used in the Section 6.1.3, except CT. CT metric, in this experiment, was defined as the time necessary to complete one filtering operation for one input sample.

Table 6.5: Synthesis Results for FIR Filter Example: Target Altera Cyclone™ IV.

Method	**Area (LE)**	**DSP Blocks**	**RAM Blocks**	**CT (cycles)**	**Frequency (MHz)**	**Code Complexity (lines of code)**
C/C++	990	9	13	60	115.15	20
VHDL by Meyer-Baese	511	22	0	1*	242.48	150

* 14 cycles initial pipeline latency.

Table 6.6 shows synthesis results for Xilinx FPGAs. The target FPGA was a Spartan™ 6, synthesized with Xilinx ISE, except for Mirzaei work, which was implemented on a Virtex™ II, also provided by Xilinx Inc.

Table 6.6: Synthesis Results for FIR Filter Example: Target Xilinx Spartan™ 6 and Xilinx Virtex™ II.

Method	**Area (LUT)**	**Area (FF)**	**DSP Blocks**	**RAM Blocks**	**CT (cycles)**	**Frequency (MHz)**	**Code Complexity (lines of code)**
C/C++	730	8220	4	0	47	594	20
VHDL by Meyer-Baese	378	353	11	0	1*	128	150
VHDL by Mirzaei**	334	739	-	-	-	-	-

* 14 cycles initial pipeline latency.
** Performance and code complexity metrics not considered.

Analysing Table 6.5 and Table 6.6 we can conclude:

1. Lower code complexity using HLS tools: HLS utilization reduces in almost 10 times the code complexity in terms of lines of code. Architectural structures, for instance pipelining, are automatically performed by both used HLS tools.

2. Large area consumption with HLS method: both tools have similar area consumption (LEs or LUTs/FFs), which is worse comparing to VHDL implementations. Although, both tools optimized the design for DSP macros re-utilization.

3. Extremely high frequency using Vivado™ HLS: Vivado™ HLS implements deep pipelines, mainly in the multipliers, increasing the frequency. The large FF utilization in this method has the same cause: FFs are used in the pipeline registers.

6.2 Design Space Exploration Results

This section presents the results for both interactive and automatic design exploration methods. Section 6.2.1 shows the results for three compilation methods applied to a VLIW processor. Section 6.2.2 presents the experiments for the proposed iterative DSE method for two benchmarks: the ρ-Vex processor and a set of FIR filters.

6.2.1 Interactive DSE Results

The design space exploration is easy with Vivado™ HLS. With Vivado™ it is possible to define area and performance constraints, using a TCL script, aiming at discovering the best cost-benefit for the target design. For the VLIW processor we defined 3 different constraints:

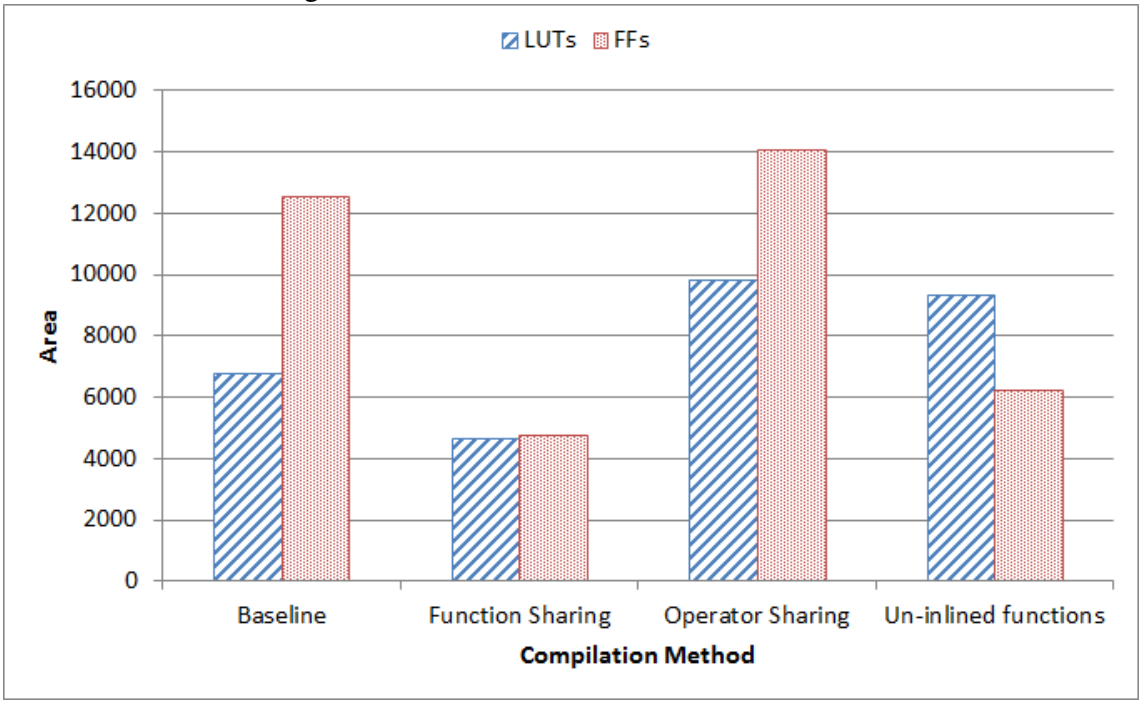
- Functions sharing: resource sharing is made at level of functions and logical blocks.

- Operators sharing: resource sharing is made at level of operators, for instance adders/subtracters, comparators, multipliers.

- Un-inlined functions: force functions being executed un-inlined, avoiding HW re-utilization/resource sharing. This method allows the tool instantiate multiple HW modules.

Figure 6.3 illustrates the area results of manual design space exploration with Vivado™. Figure 6.4 shows the performance obtained through interactive DSE using Vivado™. The processor implemented and the assumed compute time (CT) for this experiment are the same presented in 6.1.3. The baseline flow is the non-guided HLS flow, where it is not setted any design constraint or synthesis directive.

Figure 6.3: DSE Area Results with Vivado™.



Source: the author.

According graphics presented in Figure 6.3 and Figure 6.4 we can conclude:

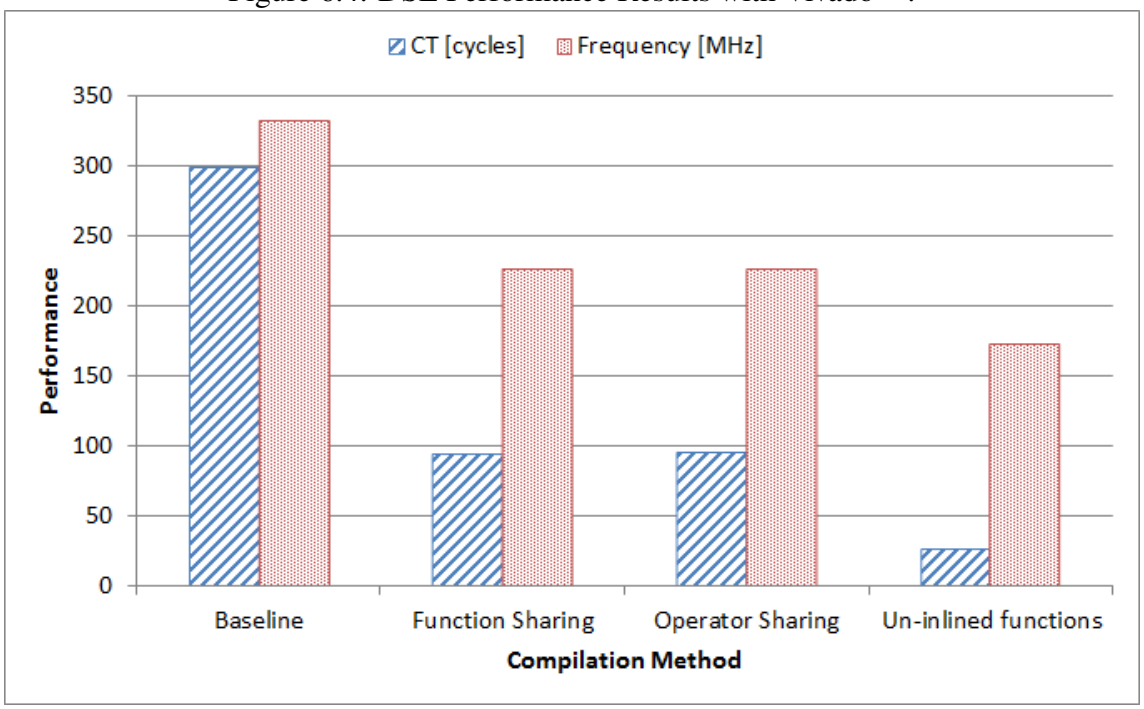
1. Smaller area usage results from using the function sharing method: allows HW resource re-utilization. This method is very effective, lower multiplexers overhead compared to total area.

2. Bad area result at operators sharing: multiplexers are very expensive in FPGAs. This technique requires a large number of multiplexers and register to select/store the operators, increasing area.

3. Better performance with un-inlined functions: this technique allows unrolling loops. Loop unrolling is a very effective method for HW parallelism.

4. Higher frequency at baseline: this method infers pipeline in most of operations, increasing frequency, although the CT is also higher. This benchmark is a sequential machine, thus it is necessary to finish an operation to start a new one. In this case, the higher frequency does not compensate the large compute time.

Figure 6.4: DSE Performance Results with Vivado™.



Source: the author.

6.2.2 Iterative DSE Method Results

This section presents the results for the iterative DSE method with HLS flow proposed in Section 5.2. We have tested our DSE framework using a Virtex™ 7 FPGA. As first benchmark we choose a four-issue academic VLIW processor, called ρ-Vex(WONG; AS; BROWN, 2008). The C code describing this processor was developed by the author. The benchmark design description in high-level has 800 lines of code.

For this benchmark we defined the following design constraints:

- Array threshold = 12;
- Maximum execution time = 8 cycles;
- Minimum execution time = 1 cycle;
- Maximum number of function instances = 4;
- Minimum number of function instances = 1.

The machine used for framework execution was an AMD Ahtlon™ 7550 Dual-core processor @ 2.5 GHz, 32 bits, 3 GB of RAM memory and Windows 7 operational system. The total framework execution time took 140 minutes, running 70 HLS compilations. The results of framework execution are shown in Table 6.7. We consider for this analysis - normalized by baseline results - area metrics, in terms of FFs and LUTs, and performance, represented by the compute time (CT) taken to run out the benchmark. We assume CT as the necessary time to execute the worst case of one ρ-Vex instruction (32 bits integer multiplication). For comparison analysis, we assume the baseline as a non-guided HLS

flow. This non-guided synthesis uses no design constraints nor sets specific directives to the HLS tool. For a more realistic analysis, we also ran one synthesis constraining the HLS tool to optimize the results for area reduction, applying only a function inline directive to all functions. This synthesis flow was called the "first pass", because it uses the results obtained in the first iteration of our DSE method. The directives file generated by DSE script is available in the Appendix B.2.

Table 6.7: DSE Framework Results: VLIW processor.

Benchmark	**Method**	**FF**	**LUT**	**CT**	**Performance**
ρ-Vex	Baseline	1.0	1.0	1.0	1.0
ρ-Vex	First Pass	0.89	1.08	0.98	1.02
ρ-Vex	DSE	0.28	0.68	1.38	0.72

Observing the results obtained in Table 6.7, the area reduction is evident. The number of FFs used by our method is almost 4x lower than a non-guided HLS flow. Our proposed method also reduces number of LUTs by 32% compared to the baseline flow. The FF and LUT reductions cause an overhead of 38% in the CT through our methodology. Our method in comparison to the manual-guided first pass flow also proves to be much better in terms of area reduction, due to similarity between first pass and baseline flow.

Analysing the cost-benefit ratio of our solution, we had to define a QoR metric. This metric was derived from speed/area relation proposed in (SOTIROPOULOU; NIKOLAIDIS, 2010). A global HW efficiency metric was defined as the quotient among normalized performance and area results:

$$HW_{eff} = \frac{Speed_{norm}}{Area_{norm}} \tag{6.1}$$

Considering the QoR metric defined by Equation 6.1, our DSE framework respects the relation in Equation 6.2. For this relation we used the medium value between LUTs and FFs to define a normalized area metric.

$$HW_{eff_{DSE}} > HW_{eff_{baseline}} \tag{6.2}$$

The QoR metric indicates HLS flow using our DSE framework is 50% more efficient than a baseline HLS flow, i.e., the normalized $HW_{eff_{DSE}} = 1.5$.

The second test-case used with our DSE method was a 12^{th}-order FIR filter. The C code was provided by Xilinx Inc. and it has around 20 lines of code. The machine used for DSE execution was the same used for first benchmark. Total runtime of our DSE script was 40 minutes, running 20 HLS compilations. Table 6.8 presents the results of our DSE method. The metrics used in this experiment are the same as in first benchmark, except for CT. In this case, CT is the time taken for one filtering operation.

Analysing the results in the Table 6.8, we can observe the area reduction using our method. The FFs consumption are reduced more than 3x, while the LUTs utilization and performance are almost the same, compared to the baseline flow. Our method also proves being around 62% more efficient for this test-case compared to the baseline flow. For this benchmark, the first pass synthesis had the same results of the baseline, i.e., only the function inline directive was not effective for this example.

Table 6.8: DSE Framework Results: FIR Filter.

Benchmark	Method	FF	LUT	CT	Performance	HW_{eff}
FIR filter	Baseline	1.0	1.0	1.0	1.0	1.0
FIR Filter	First Pass	1.0	1.0	1.0	1.0	1.0
FIR filter	DSE	0.31	1.02	0.92	1.08	1.62

To assure the scalability capability of the proposed method, we compiled a set of FIR filters with multiple orders, from 4 taps to 40 taps, with steps of 4 taps. For this we used the same generated directives script used for the 12-tap FIR filter experiment. In this experiment we used the same machine, HLS tool and FPGA device of previous automatic DSE experiments. The results of this experiment are shown in Table 6.9.

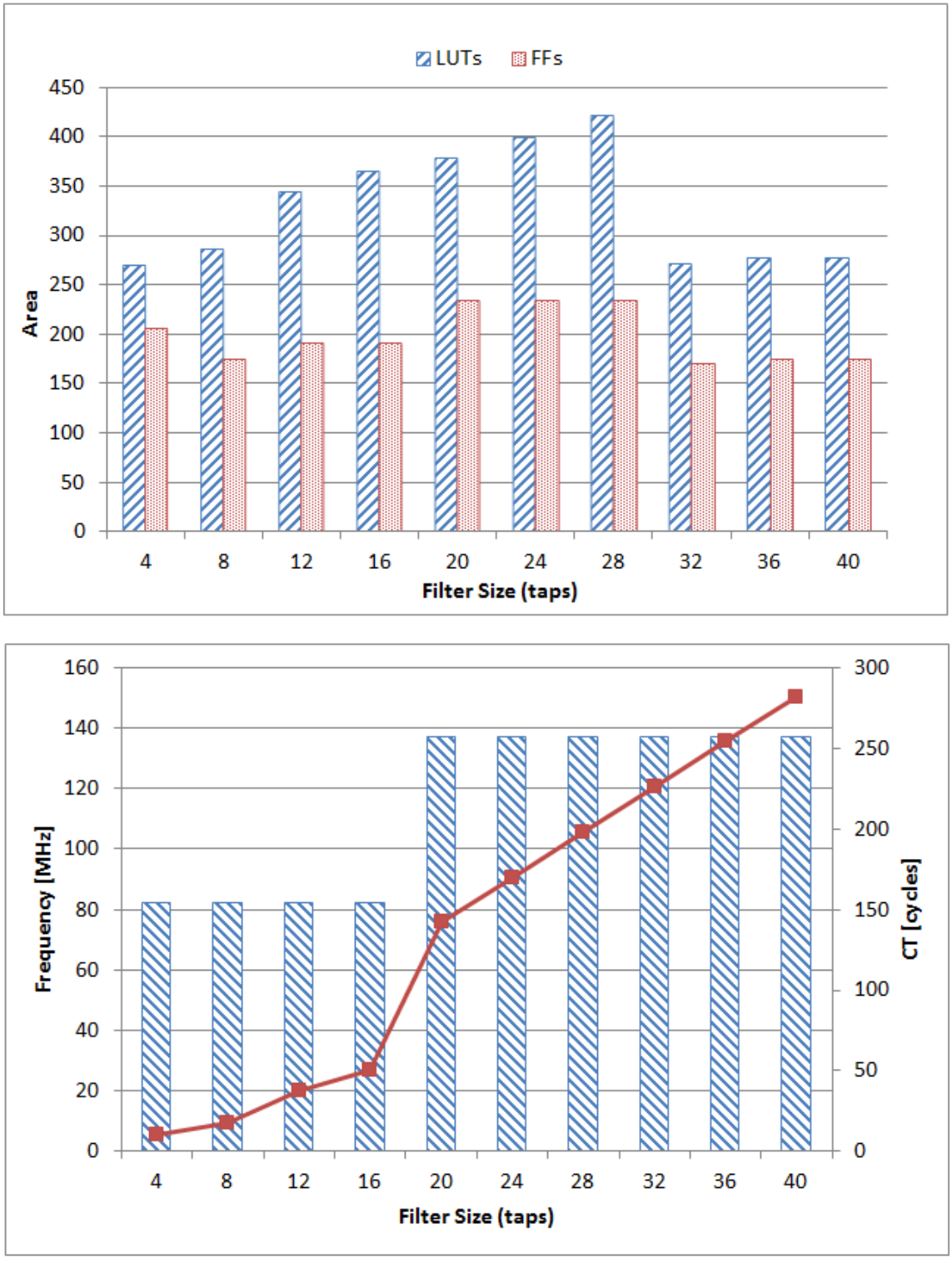
Table 6.9: Synthesis Results for Multiple Order FIR Filter Example.

Filter Size	Area (LUT)	Area (FF)	DSP Blocks	RAM Blocks	CT (cycles)	Frequency (MHz)	HW_{eff}
4-taps	270	205	2	0	10	81.96	2.85
8-taps	285	174	2	0	18	81.96	2.14
12-taps	343	190	2	0	38	81.96	1.62
16-taps	365	190	2	0	50	81.96	1.87
20-taps	378	234	2	0	142	136.80	1.37
24-taps	399	234	2	0	170	136.80	1.36
28-taps	421	234	2	0	198	136.80	1.34
32-taps	271	170	2	1	226	136.80	1.59
36-taps	277	174	2	1	254	136.80	1.57
40-taps	277	174	2	1	282	136.80	1.57

Analysing data from Table 6.9 we drew two relation graphics: area consumption versus filter size, and performance versus filter size. For area analysis we considered the usage of LUTs and FFs. For performance evaluation we considered the compute time and maximum operation frequency. These relation curves are presented in Figure 6.5.

Considering data from Table 6.9 and the curves of Figure 6.5 we can affirm that our proposed method has a regular scalability with the circuit increase. Analysing the performance curve we can see a linearity characteristic on compute time increasing, which is expected in our model, since our optimizations address only area trade-off. In the area curve, one point calls attention, the drastic area reduction with 32-taps filter. It occurs because the data samples and internal products are mapped into block RAMs, not in registers. Another interesting data from Table 6.9 is the same number DSP macros consumption by all filter sizes. It is another characteristic of our method, the MAC (Multiply and Accumulate) operations are forced to not being unrolled in the kernel loop.

Figure 6.5: Multiple Size FIR Filters. a) Area Versus Filter Size Curve. b) Performance Versus Filter Size Curve.



Source: the author.

7 CONCLUSIONS

This work presented a study about recent techniques in FPGA development flow. We discussed about well-known methods for FPGA development aiming at improving FPGA efficiency in terms of area, performance and power. We reviewed the state-of-the art of the HLS tools and HLS project methodology aiming at fast turn-around purpose. We also revisited the evolution of the HDL languages over the last decades.

Our practical experiments were focused in high-level synthesis utilization targeted to FPGA devices. In this work, we evaluate the results of two HLS tools, one of them is an academic and open source tool, the LegUp. The other tool analysed in our work was a commercial tool, the Vivado™ HLS compiler. Our practical results with using both tools show a notable code complexity and development time reduction, although, in the most cases, the complexity reduction means a performance decreasing or area overhead, when compared to a hand-coded HDL design.

Some synthesis results call attention due to the similarity between the RTL design and the HLS method, for instance, the square-root implementation using SW pipeline technique. This is explained due to the capability of the HLS tool to perform a pipeline implementation with few stalls, because we do not have data dependency in this algorithm. On the other hand, we observe a wide disparity in some other cases, as in the VLIW implementation, comparing the hand-coded HDL design with the HLS compilation for both HLS tool providers. The explanation for this is due the large number of conditional statements avoiding a efficient scheduling and pipelining technique.

To address this area/performance overhead, we had to improve the high level descriptions aiming at better synthesis results. Combining efficient code partitioning and the flexibility of synthesis parameters insertion of Vivado™ HLS compiler, we introduced an iterative method for efficient design space exploration with HLS targeting area reduction. To support method design, we revisited recent methods for DSE in hardware system, where is introduced the Pareto optimally concept, used to evaluate a cost-benefit of a solution.

Our DSE method results for two test-cases, a VLIW processor and a 12^{th}-order FIR Filter, proved to be very effective when compared to a non-guided HLS flow. Our results in terms of area are up to 4X better for the VLIW processor and 3X for the FIR Filter benchmark. Our method is also much better than the baseline flow in terms of QoR. Our iterative method is 50% and 62% more efficient than baseline, for the VLIW processor and for the digital filter, respectively.

As future work we addressed improving the proposed DSE method, making it no purely exhaustive, but using an synthesis heuristic to guide the HLS compilation, reducing then the convergence time. It is also desired to extend the iterative DSE method to other design parameters, such as power and performance. Moreover, this method can and

should be extended to others HLS tools and FPGA providers, not limited to an specific vendor.

REFERENCES

ALTERA. Altera, Inc. **http://www.altera.com**, [S.l.], 2014.

BELL, C. G.; NEWELL, A. **Computer structures**: readings and examples. New York, St. Louis, San Francisco: McGraw-Hill, 1971 c, 1971. (McGraw-Hill computer science series).

CANIS, A. et al. LegUp: high-level synthesis for fpga-based processor/accelerator systems. In: ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 19., New York, NY, USA. **Proceedings...** ACM, 2011. p.33–36. (FPGA '11).

CHEN, D. et al. LOPASS: a low-power architectural synthesis system for fpgas with interconnect estimation and optimization. **Very Large Scale Integration (VLSI) Systems, IEEE Transactions on**, [S.l.], v.18, n.4, p.564–577, April 2010.

CHOW, C. et al. Dynamic voltage scaling for commercial FPGAs. In: FIELD-PROGRAMMABLE TECHNOLOGY, 2005. PROCEEDINGS. 2005 IEEE INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2005. p.173–180.

COUSSY, P.; MORAWIEC, A. **High-Level Synthesis**: from algorithm to digital circuit. 1st.ed. [S.l.]: Springer Publishing Company, Incorporated, 2008.

DIRECTOR, S. et al. A design methodology and computer aids for digital VLSI systems. **Circuits and Systems, IEEE Transactions on**, [S.l.], v.28, n.7, p.634–645, Jul 1981.

FRIEDMAN, T.; YANG, S.-C. Methods Used in an Automatic Logic Design Generator (ALERT). **Computers, IEEE Transactions on**, [S.l.], v.C-18, n.7, p.593–614, July 1969.

GUPTA, S. et al. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In: VLSI DESIGN, 2003. PROCEEDINGS. 16TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2003. p.461–466.

GURUMANI, S. et al. High-level synthesis of multiple dependent CUDA kernels on FPGA. In: DESIGN AUTOMATION CONFERENCE (ASP-DAC), 2013 18TH ASIA AND SOUTH PACIFIC. **Anais...** [S.l.: s.n.], 2013. p.305–312.

HADJIS, S. et al. Impact of FPGA Architecture on Resource Sharing in High-level Synthesis. In: ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, New York, NY, USA. **Proceedings...** ACM, 2012. p.111–114. (FPGA '12).

HARA, Y. et al. CHStone: a benchmark program suite for practical c-based high-level synthesis. In: CIRCUITS AND SYSTEMS, 2008. ISCAS 2008. IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2008. p.1192–1195.

HARTENSTEIN, R. **The History of KARL and ABL**. [S.l.]: Universität Kaiserslautern, 1993. (Fachbereich Informatik: Interner Bericht).

HUDA, S.; MALLICK, M.; ANDERSON, J. Clock gating architectures for FPGA power reduction. In: FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 2009. FPL 2009. INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2009. p.112–118.

LEE, K. et al. A high level design language for programmable logic devices. **VLSI Design**, [S.l.], p.50–62, Jun 1985.

LHAIRECH-LEBRETON, G.; COUSSY, P.; MARTIN, E. Hierarchical and Multiple-Clock Domain High-Level Synthesis for Low-Power Design on FPGA. In: FIELD PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 2010 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. p.464–468.

LUA. Lua: the programming language. **http://www.lua.org/**, [S.l.], 2014.

MCFARLAND, M. C.; PARKER, A.; CAMPOSANO, R. The high-level synthesis of digital systems. **Proceedings of the IEEE**, [S.l.], v.78, n.2, p.301–318, Feb 1990.

MEYER-BAESE, U. **Digital Signal Processing with Field Programmable Gate Arrays**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

MIRZAEI, S.; HOSANGADI, A.; KASTNER, R. FPGA Implementation of High Speed FIR Filters Using Add and Shift Method. In: COMPUTER DESIGN, 2006. ICCD 2006. INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2006. p.308–313.

MOORE, G. Cramming More Components Onto Integrated Circuits. **Proceedings of the IEEE**, [S.l.], v.86, n.1, p.82–85, Jan 1998.

OLIVER, J. et al. Clock gating and clock enable for FPGA power reduction. In: PROGRAMMABLE LOGIC (SPL), 2012 VIII SOUTHERN CONFERENCE ON. **Anais...** [S.l.: s.n.], 2012. p.1–5.

PALERMO, G.; SILVANO, C.; ZACCARIA, V. A Flexible Framework for Fast Multi-objective Design Space Exploration of Embedded Systems. In: INTEGRATED CIRCUIT AND SYSTEM DESIGN, POWER AND TIMING MODELING, OPTIMIZATION AND SIMULATION, 13TH INTERNATIONAL WORKSHOP, PATMOS 2003, TORINO, ITALY, SEPTEMBER 10-12, 2003, PROCEEDINGS. **Anais...** [S.l.: s.n.], 2003. p.249–258.

PALERMO, G.; SILVANO, C.; ZACCARIA, V. Multi-objective Design Space Exploration of Embedded Systems. **J. Embedded Comput.**, Amsterdam, The Netherlands, The Netherlands, v.1, n.3, p.305–316, Aug. 2005.

PANDEY, B. et al. Clock Gating Aware Low Power Global Reset ALU and Implementation on 28nm FPGA. In: COMPUTATIONAL INTELLIGENCE AND COMMUNICATION NETWORKS (CICN), 2013 5TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2013. p.413–417.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design**: the hardware/software interface. 3rd.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

PELLERIN, D.; THIBAULT, S. **Practical FPGA Programming in C**. [S.l.]: Prentice Hall Professional Technical Reference, 2005. (Prentice Hall modern semiconductor design series).

RABAEY, J. M.; CHANDRAKASAN, A.; NIKOLIC, B. **Digital Integrated Circuits**: a design perspective. 2nd.ed. [S.l.]: Prentice Hall, 2003.

RAJE, S.; BERGAMASCHI, R. Generalized resource sharing. In: COMPUTER-AIDED DESIGN, 1997. DIGEST OF TECHNICAL PAPERS., 1997 IEEE/ACM INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 1997. p.326–332.

ROSADO-MUñOZ, A. et al. FPGA Implementation of an Adaptive Filter Robust to Impulsive Noise: two approaches. **Industrial Electronics, IEEE Transactions on**, [S.l.], v.58, n.3, p.860–870, March 2011.

SANCHEZ, F. et al. Comparative of HLS and HDL implementations of a grid synchronization algorithm. In: INDUSTRIAL ELECTRONICS SOCIETY, IECON 2013 - 39TH ANNUAL CONFERENCE OF THE IEEE. **Anais...** [S.l.: s.n.], 2013. p.2232–2237.

SCHAFER, B. Allocation of FPGA DSP-macros in multi-process high-level synthesis systems. In: DESIGN AUTOMATION CONFERENCE (ASP-DAC), 2014 19TH ASIA AND SOUTH PACIFIC. **Anais...** [S.l.: s.n.], 2014. p.616–621.

SHAO, Y. et al. Aladdin: a pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In: COMPUTER ARCHITECTURE (ISCA), 2014 ACM/IEEE 41ST INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2014. p.97–108.

SOTIROPOULOU, C.-L.; NIKOLAIDIS, S. Design space exploration for FPGA-based multiprocessing systems. In: ELECTRONICS, CIRCUITS, AND SYSTEMS (ICECS), 2010 17TH IEEE INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. p.1164–1167.

SUN, W.; WIRTHLIN, M.; NEUENDORFFER, S. FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.26, n.2, p.254–265, Feb 2007.

SYNOPSYS. Synopsys, Inc. **http://www.synopsys.com**, [S.l.], 2014.

TAN, M. et al. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD), 33. **Proceedings...** [S.l.: s.n.], 2014. (ICCAD'14).

WAKABAYASHI, K.; TAKENAKA, T.; INOUE, H. Mapping complex algorithm into FPGA with High Level Synthesis reconfigurable chips with High Level Synthesis compared with CPU, GPGPU. In: DESIGN AUTOMATION CONFERENCE (ASP-DAC), 2014 19TH ASIA AND SOUTH PACIFIC. **Anais...** [S.l.: s.n.], 2014. p.282–284.

WONG, S.; AS, T. van; BROWN, G. ρ-VEX: a reconfigurable and extensible softcore vliw processor. In: ICECE TECHNOLOGY, 2008. FPT 2008. INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2008. p.369–372.

XILINX. Xilinx, Inc. **http://www.xilinx.com**, [S.l.], 2014.

XYDIS, S. et al. Efficient High Level Synthesis Exploration Methodology Combining Exhaustive and Gradient-Based Pruned Searching. In: VLSI (ISVLSI), 2010 IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2010. p.104–109.

ZHANG, Y.; ROIVAINEN, J.; MAMMELA, A. Clock-Gating in FPGAs: a novel and comparative evaluation. In: DIGITAL SYSTEM DESIGN: ARCHITECTURES, METHODS AND TOOLS, 2006. DSD 2006. 9TH EUROMICRO CONFERENCE ON. **Anais...** [S.l.: s.n.], 2006. p.‘584–590.

ZIMMERMANN, G. The Mimola Design System a Computer Aided Digital Processor Design Method. In: DESIGN AUTOMATION, 1979. 16TH CONFERENCE ON. **Anais...** [S.l.: s.n.], 1979. p.53–58.

APPENDIX A RESUMO DA DISSERTAÇÃO "ARCHITECTURAL EXPLORATION OF DIGITAL SYSTEMS DESIGN FOR FPGAS USING C/C++/SYSTEMC SPECIFICATION LANGUAGES"

A.1 Introdução

Dispositivos FPGA são largamente utilizados para rápida implementação de funções digitais em *hardware* e para emulação de circuitos ASIC (*Application-Specific Integrated Circuit*). Entretanto, mesmo os FPGAs do estado-da-arte possuem sérias limitações em termos de desempenho, utilização de área e consumo de potência. Com o aumento da complexidade dos sistemas digitais, técnicas de otimização são necessárias para tornar os projetos com esses devices competitivos.

Este trabalho discute técnicas conhecidas disponíveis na literatura visando otimização de projeto em FPGAs. Estes métodos incluem técnicas de redução de potência, como *clock gating* e *voltage/frequency scaling*. Em termos de área, é discutida a importância do compartilhamento de recursos de *hardware*. Para aumento do desempenho, técnicas como *pipelining*, paralelismo e redução de *fan-out* são revisadas.

O principal objetivo desta dissertação de mestrado é discutir e explorar técnicas de otimização em alto nível aplicadas a sistemas com FPGAs. O projeto RTL (*Register-Transfer Level*) é um gargalo nos cronogramas de projetos de sistemas digitais. A utilização e métodos HLS (*High-Level Synthesis*) são discutidos visando aumento de produtividade em projetos baseados em FPGA. Ferramenta HLS acadêmicas e comerciais são apresentadas, bem como técnicas avançadas de projeto utilizando linguagens de especificação, por exemplo C/C++ ou SystemC.

Muitos algoritmos possuem seus padrões em código aberto e estão disponíveis em linguagens de alto nível, como C. No clássico fluxo de projeto para FPGA, os projetistas de *hardware* necessitam traduzir esses algoritmos para FPGAs, desempenhando todos as etapas de otimização utilizando linguagens RTL. O desenvolvimento e a verificação RTL é uma tarefa complexa e cara quando comparada a implementações em *software*. Níveis de abstração mais altos permitem que projetistas de *software* desenvolvam sistemas para FPGA utilizando compiladores HLS. Fornecedores de ferramentas HLS afirmam que em certos casos, a produtividade pode aumentar em até 50% utilizando HLS comparada ao projeto RTL codificado a mão. A adição de diretivas/pragmas nos compiladores HLS permitem que o projetista guie a implementação do algoritmo visando um *trade-off* específico, por exemplo, desempenho ou redução de área. O processo de verificação utilizando fluxo RTL é bastante complexo devido à dificuldade das linguagens HDL (*Hardware Description Languages*) em descrever comportamento dos circuitos no mundo real, enquanto

utilizando linguagens de mais alta abstração é possível emular as condições de operação. Além disso, uma simualação RTL pode levar muitas horas, e até dias, dependendo da complexidade do circuito, o que normalmente requer poderosos servidores de simulação.

Neste contexto, este trabalho apresenta um metodologia para projeto digital utilizando linguagens de alto nível através do fluxo HLS. São exploradas técnicas de particionamento de código e inserção de pragmas visando aumento no *Quality of Results* (QoR). Neste trabalho, foi desenolvido um método para exploração de espaço de projeto com HLS visando redução de área. Este método provou ser muito efetivo quando comparado ao fluxo de projeto HLS não guiado, sendo até 50% mais eficiente em termos de área utilizando um processador VLIW (*Very Long Instruction Word*) acadêmico como *benchmark*.

A.1.1 Motivação

O FPGA é um *chip* pré definido, que possui um grande número de macro células interconectadas uma com as outras através de linhas de roteamento e chaves. Cada macro célula tem em seu interior um multiplexador configurável que é capaz de implementar todas as equações lógicas das combinações de suas entradas e um elemento de armazenamento de dados (*flip-flop*). Dispositivos FPGA modernos incluem blocks de memória RAM (*Random Access Memory*), blocos de DSP (*Digital Signal Processing*), PLLs (*Phase-locked loop*), *gigabit transceivers* e até processadores.

De acordo com dados de fabricantes de FPGAs, normalmente mais de 70% da área de silício em um FPGA é destinada a interconxões, o que faz esses dispositivos muito ineficientes em termos de consumo de área. Como uma pastilha pré definida de silício, o consumo de potência é uma problema, porque o componente de potência estática está sempre presente em todo *chip*, mesmo se as células lógicas e os flip-flops não são usados.Em termos de potência dinâmica, os grandes fabricantes tem feito um grande esforço em fazer os FPGAs mais eficientes. Componentes *hard*, como blocos de SRAMs e DSPs - também consomem potência, mas a tésnicas de otimização desses componentes não serão discutidas neste trabalho

O desempenho em FPGAs é pior do que em dispositivos ASIC. Enquanto ASICs são projetados visando uma aplicação específica, FPGAs são genéricos e isto impacta diretamente no desempenho. Muitos níveis lógicos entre elementos de armazenamento, grandes *fan-outs*, a árvore de relógio, o atraso das linhas de roteamento e o posicionamento fixo dos elementos lógicos contribuem para redução da máxima frequência de operação.

O desenvolvimento de *hardware* tem um alto custo e toma grande porção do tempo de projeto. O tradicional desenvolvimento para FPGAs é baseado em arquitetura sistêmica, descrição RTL, simulação funcional, e por fim, a síntese lógica e *place and route*. Neste método, a tarefa mais difícil é a descrição RTL, normalmente um trabalho lento e meticuloso. Técnicas avançadas em HLS automatizam esta etapa, gerando um código RTL sintetizável a partir de uma linguagem de alto nível. Esforços recentes tornaram o projeto de sistemas em alto nível competitivos para sistemas baseados em FPGAs. Arquiteturas regulares, como algoritmos DSPs, frequentemente estão presentes na literatura para avaliar implementações em alto nível. Escalonamento de algoritmos, *pipelining* de laços/funções e *loop unrolling* são técnicas comuns utilizadas pelos fabricantes de ferramentas HLS visando melhoria nos resultados de síntese.

A.1.2 Contribuições

A principal contribuição desta dissertação de mestrado é propor técnicas de exploração em alto nível para sistemas digitais que tem como alvo dispositivos FPGAs. As

contribuições deste trabalho estão listadas abaixo:

- **Exploração Arquitetural em Alto Nível:** para atingir bons resultados de síntese com descrições de alto nível, melhorias no código se fazem necessárias. A metodologia adotada neste trabalho discute algumas destas técnicas utilizadas em ambientes HLS. Neste trabalho é proposto uma técnica de particionamento de código combinada com inserção de diretivas/pragmas apropriadas de compilação visando melhoria dos resultados de síntese. Estes métodos foram testados com duas ferramentas HLS: o LegUp *compiler* e o Vivado^{TM}HLS *compiler*.
- **Exploração de Espaço de Projeto com Síntese de Alto Nível:** visando melhor QoR em ambientes HLS, foi proposto um método iterativo para DSE (*Design Space Exploration*) em sistemas de *hardware*. Este método é multi plataforma escrito em linguagem e é compatível com o Vivado^{TM}HLS *compiler*. Este método é basicamente composto por: análise do código em alto nível, inserção automática de diretivas de compilação e avaliação dos resultados.

A.1.3 Organização da Dissertação

Esta dissertação de mestrado está organizada da seguinte forma: as técnicas de otimização de *hardware* são revisadas no Capítulo 2, onde são discutidas metodologias conhecidas para otimização de projetos em termos de área, potência e desempenho para FPGAs. Ferramentas e métodos HLS são revistos no Capítulo 3, sendo apresentado a eveolução das linguagens HLS, as ferramentas HLS do estado-da-arte e os métodos de otimização em HLS presentes na literatura. Eploração de espaço de projeto em sistemas de *hardware* é apresentado no Capítulo 4, onde são introduzidos importantes conceitos para um eficiente DSE aplicado ao projeto de *hardware*. No Capitulo 5 é apresentada a metodologia proposta neste trabalho, que inclui melhorias propostas no código em alto nível e um método iterativo para DSE em HLS. O Capítulo 6 apresenta os resultados experimentais, para avaliação das ferramentas HLS e a metodologia de DSE em um amibiente HLS. Finalmente, as conclusões são apresentadas no Capítulo 7.

A.2 Resumo das Contribuições da Dissertação: Exploração Arquitetural no Projeto de Sistemas Digitais para FPGAs Utilizando Linguagens de Especificação C/C++/SystemC

Códigos com altos níveis de abstração, normalmente, não são bem aceitos por ferramentas HLS. Entretanto, níveis de abstração mais altos são comuns em linguagens derivadas do C e essas propriedades facilitam o desenvolvimento HLS visando aumento na produtividade. Para preencher esta lacuna, é necessário adaptar o código fonte para obter os resultados esperados. Essas transformações no código incluem: eficiente particionamento de código e exploração adequada das diretivas HLS.

Códigos baseados na liguagem C são serialmente exeutados. O projetista de FPGA deve traduzir este código serial em um código paralelo para execução em FPGA. Uma transformação estudada neste trabalho foi o particionamento de código (PELLERIN; THIBAULT, 2005). Um código bem particionado permite que a ferramenta HLS interprete corretamente o código fonte visando otimizações apropriadas. Técnicas como *loop pipelining*, *loop unrolling* e compartilhamento de recursos de *hardware* são possíveis utilizando esse método. A técnica implementada neste trabalho detecta trechos de código que executam como um bloco e os isola em uma função.

Particionamento de código e exploração de parâmetros HLS estão intimamente relacionados. No Vivado™ HLS, os parâmetros de compilação podem ser introduzidos diratamente no código, através de pragmas, ou através de um *script* TCL (*Tool Command Language*). Com um código funcionalmente dividido, a inserção das diretivas HLS são mais assertivas. Para aumento de performance, os parâmetros mais úteis são: *pipelining*, *loop unrolling*, *array partitioning* e *datapath*. Quando o objetivo é redução de área, as diretivas mais comuns são *inlining functions* e *resource sharing*.

É difícil e tedioso explorar todas as combinações dos métodos discutidos nos parágrafos anteriores, e seria ineficiente explorá-los manualmente. Para suportar esses métodos, foi implementado um *framework* iterativo para DSE em ambientes HLS. Este método permite uma rápida resposta em termos de análise de *trade-off*.

Neste contexto é apresentado a metodologia adotada neste método de DSE. O método proposto explora eficientemente o espaço de projeto visando atingir melhores resultados de área comparado ao fluxo HLS não guiado. Este *framework* tem como alvo FPGAs Xilinx e o seu resultado é um arquivo de diretivas TCL visando o menor consumo possível de área. é importante salientar que este método não provê nenhuma alteração arquitetural no código fonte, somente descobre quais são as melhores diretivas de otimização para o código em questão.

O *framework* é um método iterativo, recursivo e multi plataforma desenvolvido em linguagem Lua. Foi utilizado como base para este método parte do *framework* proposto por Xydis em (XYDIS et al., 2010). A Figura 5.3 apresenta uma visão simplificada do *framework* DSE proposto, o qual utiliza o Vivado™ HLS como compilador HLS.

A versão atual do método de DSE suporta C/C++. O código de entrada para o *script* DSE precisa seguir algumas diretrizes para facilitar a convergência do *script* de DSE. Entre essas diretrizes estão: adicionar *labels* nos laços, posicionar o caracter de abertura de bloco '{' sempre em uma nova linha, declarar a variávies uma por linha e utilizar somente tipos nativos das linguagens C/C++. Além disso, é fundamental que o código fonte esteja particionado funcionalmente para maximização dos resultados.

A etapa de análise do código de entrada é responsável por identificar pontos chave no código onde é possível inserir alguma diretiva de otimização. O *script* é capaz de identificar declaração de funções e vetores, laços e execuções multiplas de uma mesma função. Nesta etapa, foi necessário utilizar-se de expressões regulares e algoritmos de busca para descobrir os pontos exatos para otimização.

Visando convergência do método porposto, foi necessário definir uma lista de restrições para delimitar o espaço de projeto. Essas restrições servem como guia para o script de DSE proposto. Entre essas restrições destacam-se: mínimo/máximo tempo de execução de laços e funções (em ciclos), número mínimo/máximo de recursos para alocação em granularidade de função e *threshold* para particionamento de vetores.

Utilizando os pontos chave de otimização descobertos pelas etapas anteriores, é então autilizado o arquivo TCL de restrições e então invoca-se o Xilinx Vivado™ HLS *compiler*. Depois da excução da compilação, os resultados são analisados e o script define se continua a iterar sobre o espaço de projeto ou não. O critério de aceitação é menor consumo de área em relação a iteração anterior.

A.3 Conclusões

Este trabalho apresentou um estudo sobre recentes técnicas no fluxo de desenvolvimento de projetos utilizando FPGAs. Foram discutidas métodos conhecidos para desen-

vimento com FPGAs visando melhorar a eficiência na síntese em termos de área, desempenho e pontência. Foram revisadas ferramentas HLS do estado-da-arte e a metodologia de projeto HLS visando aumento de produtividade no projeto de sistemas digitais. Este trabalho também revisitou a evolução das liguagens HLS nas últimas décadas.

O experimentos deste trabalho focaram na utilização de ferramentas de siíntese de alto nível tendo como alvo dispositivos FPGA. Este trabalho avaliou os resultados de duas ferramentas HLS, uma delas é uma ferramenta acadêmica e código aberto, o LegUp. A outra ferramenta utilizada foi uma ferramenta comercial, o Vivado™ HLS *compiler*. Os resultados práticos utilizando ambas ferramentas mostram a notória redução na complexidade dos códigos e tempo de desenvolvimento, embora, na maioria dos casos, esta redução de complexidade representou piores resultados em desempenho e área, quando comparados a projetos codificados a mão utlizando linguagens HDL.

Alguns resultados chamam atenção devido a similaridade entre o projeto RTL e os métodos HLS, como, por exemplo, a implementação do algoritmo da raiz quadrada utilizando a técnica de SW (*software*) *pipelining*. Isto é explicado devido à capacidade da ferramenta em implementar um *pipeline* com poucos *stalls*, porque não há dependência de dados neste algoritmo. Por outro lado, foi observado uma grande disparidade em outros casos, como na implementação do processador VLIW, comparando com o projeto HDL codificado a mão e a compilação HLS com ambas as ferramentas analisadas. A explicação para isto é o grande número de comandos sequenciais, impossibilitando o eficiente escalonamento de tarefas e implementação de *pipeline* pelos compiladores HLS.

Para lidar com esses resultados insatisfatórios de área/desempenho, foi necessário melhorar as descrições em alto nível visando melhores resultados de síntese. Combinando eficiente particionamento de código e a flexibilidade na inserção de diretivas de compilação do Vivado™ HLS *compiler*, foi introduzido um método iterativo para eficiente exploração de espaço de projeto em HLS visando redução de área. Para suportar o método proposto, foram revisado recentes métodos para DSE em sistemas de *hardware*, onde é introduzido o conceito de otimalidade de Pareto, utilizado para avaliar o custo-benefício de uma solução.

Os resultados do método DSE iterativo para dois exemplos, um processador VLIW e um filtro FIR (*Finite Impulse Response*), provaram ser bastante efetivos quando comparados do fluxo HLS não guiado. Os resultados em termos de área foram até 4X melhores para o processador VLIW e 3X melhores para o exemplo do filtro FIR. O método proposto também é muito melhor que o fluxo não guiado em termos de QoR, sendo 50% e 62% mais eficiente que o fluxo não guiado, para o processador VLIW e para o filtro digital, respectivamente.

Como trabalho futuro, deseja-se melhorar este método DSE, fazendo com que não seja mais puramente exaustivo, utilizando alguma heurística para guiar a compilação, reduzindo então o tempo de convergência. Pretende-se também estender o método DSE desenvolvido neste trabalho para otimizações envolvendo outros parâmetros de projeto, como potência e desempenho. Além disso, este método DSE pode e deve ser estendido a outros fabricantes de ferramentas HLS e fornecedores de FPGAs, não limitado a um fabricante específico.

APPENDIX B VLIW PROCESSOR SOURCE CODES

B.1 C-code Design Entry

B.1.1 Constants Definitions (r_vex.h)

```
/*****************************************************************/
/* Name:     r-Vex processor constant definitions                */
/* Purpose: Control and DataPath of Academic VLIW procesor       */
/* Author:  Jeferson Santiago da Silva                           */
/* Notes:    Based on r-Vex suite (https://code.google.com/p/r-vex/) */
/*****************************************************************/
#ifndef RVEX_H
#define RVEX_H

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* Misc */
#define DEBUG_LEVEL 0

/* Generic definitions */
#define DATA_MEM_SIZE       64
#define SYLLABLE_NUM        4
#define ALU_NUM             4
#define MUL_NUM             2
#define INI_MUL_SLOT        1
#define CTRL_SLOT           0
#define MEM_SLOT            3
#define REGISTER_GR_NUM     64
#define REGISTER_BR_NUM     8

/* Fixed registers */
#define LINK_REGISTER       REGISTER_GR_NUM - 1
#define STACK_POINTER       1

/* Opcode definitions */
#define OP_CODE(X)          (X >> 25)
#define IMM_OP(X)           (X >> 23) & 3

/* Operands */
#define DEST_GR(X)          (X >> 17) & 63
#define SRC1_GR(X)          (X >> 11) & 63
#define SRC2_GR(X)          (X >> 05) & 63
#define DEST_BR(X)          (X >> 02) & 7
```

```
#define SHORT_IMM_DATA(X)    (X >> 02) & 511
#define BRANCH_IMM_DATA(X)   (X >> 05) & 4095
#define LONG_LO_IMM_DATA(X)  (X >> 01) & 1023
#define LONG_HI_IMM_DATA(X)  (X >> 03) & 0x1FFFFF

/* Operations */
/* Op code as Input */
#define NOP_TYPE(X)          X == 0
#define STOP_TYPE(X)         X == 31
#define ALU_TYPE(X)          (X >> 6) == 1
#define MUL_TYPE(X)          (X >> 4) == 0 && !NOP_TYPE(X)
#define CTRL_TYPE(X)         (X >> 4) == 2
#define MEM_TYPE(X)          (X >> 4) == 1

/* Destiny operation as input */
#define BRANCH_DEST(X)       X == 0

/* Immediate switch */
/* Immediate operation as input */
#define NO_IMM               0
#define SHORT_IMM            1
#define BRANCH_IMM           2
#define LONG_IMM             3

/* ALU Operations opcodes */
#define ADD      65      // Add
#define AND      67      // Bitwise AND
#define ANDC     68      // Bitwise complement and AND
#define MAX      69      // Maximum signed
#define MAXU     70      // Maximum unsigned
#define MIN      71      // Minimum signed
#define MINU     72      // Minimum unsigned
#define OR       73      // Bitwise OR
#define ORC      74      // Bitwise complement and OR
#define SH1ADD   75      // Shift left 1 and add
#define SH2ADD   76      // Shift left 2 and add
#define SH3ADD   77      // Shift left 3 and add
#define SH4ADD   78      // Shift left 4 and add
#define SHL      79      // Shift left
#define SHR      80      // Shift right signed
#define SHRU     81      // Shift right unsigned
#define SUB      82      // Subtract
#define SXTB     83      // Sign extend byte
#define SXTH     84      // Sign extend half word
#define ZXTB     85      // Zero extend byte
#define ZXTH     86      // Zero extend half word
#define XOR      87      // Bitwise XOR
#define MOV      88      // Copy s1 to other location
#define CMPEQ    89      // Compare: equal
#define CMPGE    90      // Compare: greater equal signed
#define CMPGEU   91      // Compare: greater equal unsigned
#define CMPGT    92      // Compare: greater signed
#define CMPGTU   93      // Compare: greater unsigned
#define CMPLE    94      // Compare: less than equal signed
#define CMPLEU   95      // Compare: less than equal unsigned
#define CMPLT    96      // Compare: less than signed
#define CMPLTU   97      // Compare: less than unsigned
```

```
#define CMPNE    98       // Compare: not equal
#define NANDL    99       // Logical NAND
#define NORL     100      // Logical NOR
#define ORL      102      // Logical OR
#define MTB      103      // Move GR to BR
#define ANDL     104      // Logical AND

/* ALU Operations */
#define ALU_ADD(A, B)        A + B
#define ALU_AND(A, B)        A & B
#define ALU_ANDC(A, B)       (~A) & B
#define ALU_MAX(A, B)        (A > B) ? A : B
#define ALU_MAXU(A, B)       ((unsigned int) A > (unsigned int) B) ? A :
    B
#define ALU_MIN(A, B)        (A < B) ? A : B
#define ALU_MINU(A, B)       ((unsigned int) A < (unsigned int) B) ? A :
    B
#define ALU_OR(A, B)         A | B
#define ALU_ORC(A, B)        (~A) | B
#define ALU_SH1ADD(A, B)     (A << 1) + B
#define ALU_SH2ADD(A, B)     (A << 2) + B
#define ALU_SH3ADD(A, B)     (A << 3) + B
#define ALU_SH4ADD(A, B)     (A << 4) + B
#define ALU_SHL(A, B)        A << B
#define ALU_SHR(A, B)        A >> B
#define ALU_SHRU(A, B)       ((unsigned int) A) >> B
#define ALU_SUB(A, B)        A - B
#define ALU_XOR(A, B)        A ^ B
#define ALU_CMPEQ(A, B)      A == B
#define ALU_CMPGE(A, B)      A >= B
#define ALU_CMPGEU(A, B)     (unsigned int) A >= (unsigned int) B
#define ALU_CMPGT(A, B)      A > B
#define ALU_CMPGTU(A, B)     (unsigned int) A > (unsigned int) B
#define ALU_CMPLE(A, B)      A <= B
#define ALU_CMPLEU(A, B)     (unsigned int) A <= (unsigned int) B
#define ALU_CMPLT(A, B)      A < B
#define ALU_CMPLTU(A, B)     (unsigned int) A < (unsigned int) B
#define ALU_CMPNE(A, B)      A != B
#define ALU_NANDL(A, B)      ~(A && B)
#define ALU_NORL(A, B)       ~(A || B)
#define ALU_ORL(A, B)        A || B
#define ALU_ANDL(A, B)       A && B
#define ALU_SXTB(A)          (A << 24) >> 24
#define ALU_SXTH(A)          (A << 16) >> 16
#define ALU_ZXTB(A)          A & 0xFF
#define ALU_ZXTH(A)          A & 0xFFFF

/* CTRL Operations opcodes*/
#define GOTO     33          // Unconditional relative jump
#define IGOTO    34          // Unconditional absolute indirect jump to
    link register
#define CALL     35          // Unconditional relative call
#define ICALL    36          // Unconditional absolute indirect call to
    link register
#define BR       37          // Conditional relative branch on true
    condition
#define BRF      38          // Conditional relative branch on false
    condition
```

```
#define RETURN   39          // Pop stack frame and goto link register
#define RFI      40          // Return from interrupt
#define XNOP     41          // Multicycle NOP

/* MUL Operations opcode */
#define MPYLL    1           // Multiply signed low 16 x low 16 bits
#define MPYLLU   2           // Multiply unsigned low 16 x low 16 bits
#define MPYLH    3           // Multiply signed low 16 (s1) x high 16 (
    s2) bits
#define MPYLHU   4           // Multiply unsigned low 16 (s1) x high 16
    (s2) bits
#define MPYHH    5           // Multiply signed high 16 x high 16 bits
#define MPYHHU   6           // Multiply unsigned high 16 x high 16 bits
#define MPYL     7           // Multiply signed low 16 (s2) x 32 (s1)
    bits
#define MPYLU    8           // Multiply unsigned low 16 (s2) x 32 (s1)
    bits
#define MPYH     9           // Multiply signed high 16 (s2) x 32 (s1)
    bits
#define MPYHU    10          // Multiply unsigned high 16 (s2) x 32 (s1)
     bits
#define MPYHS    11          // Multiply signed high 16 (s2) x 32 (s1)
    bits, shift left 16

#define MUL_MPYLL(A, B)      (A & 0xFFFF) * (B & 0xFFFF)
#define MUL_MPYLLU(A, B)     ((unsigned int)(A) & 0xFFFF) * ((unsigned
    int)(B) & 0xFFFF)
#define MUL_MPYLH(A, B)      (A >> 16) * (B & 0xFFFF)
#define MUL_MPYLHU(A, B)     ((unsigned int)(A) >> 16) * ((unsigned int)
    (B) & 0xFFFF)
#define MUL_MPYHH(A, B)      (A >> 16) * (B >> 16)
#define MUL_MPYHHU(A, B)     ((unsigned int)(A) >> 16) * ((unsigned int)
    (B) >> 16)

/* MEM Operations opcodes */
#define LDW      17          // Load word
#define LDH      18          // Load halfword signed
#define LDHU     19          // Load halfword unsigned
#define LDB      20          // Load byte signed
#define LDBU     21          // Load byte unsigned
#define STW      22          // Store word
#define STH      23          // Store halfword
#define STB      24          // Store byte
#define PFT      25          // Prefetch

#endif
```

B.1.2 Instruction Memory (r_vex_imem.h)

```
/*****************************************************************/
/* Name:     Instruction memory for r-Vex processor              */
/* Purpose:  Control and DataPath of Academic VLIW procesor      */
/* Author:   Jeferson Santiago da Silva                          */
/* Notes:    Based on r-Vex suite (https://code.google.com/p/r-vex/) */
/*****************************************************************/

#ifndef RVEX_IMEM_H
```

```
#define RVEX_IMEM_H

#include <stdlib.h>
#include <stdio.h>
#include "r_vex.h"

#define INST_MEM_SIZE             5

/* Instrction memory */
unsigned int imem[SYLLABLE_NUM][INST_MEM_SIZE] =
{
    {0x82840005, 0x4B000061, 0x43000021, 0x00000001, 0x3E000001},
    {0x829400B0, 0xB2004940, 0x00000000, 0x00000000, 0000000000},
    {0xB0020000, 0x82060040, 0x82020060, 0xB0120000, 0x00000000},
    {0x829E0006, 0x82040842, 0x82924806, 0x2C027802, 0x00000002}
};

#endif
```

B.1.3 Branches and Memory Access Functions (r_vex_fun.h)

```
/*****************************************************************/
/* Name:    r-Vex processor functions                            */
/* Purpose: Control and DataPath of Academic VLIW procesor       */
/* Author:  Jeferson Santiago da Silva                           */
/* Notes:   Based on r-Vex suite (https://code.google.com/p/r-vex/) */
/*****************************************************************/

#ifndef RVEX_FUN_H
#define RVEX_FUN_H

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "r_vex_imem.h"

/* CTRL Operations opcodes*/
void ctrl_goto(int *pc_goto, int offset)
{
    *pc_goto = offset;
    return;
}

void ctrl_br(int *pc_goto, int offset, char br)
{
    if (br) *pc_goto = offset;
    return;
}
void ctrl_brf(int *pc_goto, int offset, char br)
{
    if (!br) *pc_goto = offset;
    return;
}

/* MEM Operations */
void mem_ldw(int *dmem, char address, int *mem_rdata){
    *mem_rdata = dmem[address];
```

```
    return;
}
void mem_stw(int *dmem, char address, int mem_wdata){
    dmem[address] = mem_wdata;
    return;
}

#endif
```

B.1.4 Functions Prototype (r_vex_top.h)

```
/*****************************************************************/
/* Name:    r-Vex processor Functions Prototypes                 */
/* Purpose: Control and DataPath of Academic VLIW procesor       */
/* Author:  Jeferson Santiago da Silva                           */
/* Notes:   Based on r-Vex suite (https://code.google.com/p/r-vex/) */
/*****************************************************************/

#ifndef RVEX_TOP_H
#define RVEX_TOP_H

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "r_vex_fun.h"

/*****************************************************************/
/* Functions prototype                                           */
/*****************************************************************/
void advance_pc(int *next_pc, int pc_goto);
void fetch(unsigned int imem[SYLLABLE_NUM][INST_MEM_SIZE], unsigned int
    *syllable, int pc);
void decode(unsigned int *syllable, char *op_code, char *imm_op, char *
    dest_gr, char *src1_gr, char *src2_gr, char *dest_br, int *
    short_imm_data, int *branch_imm_data, int *long_imm_data);
void execute(int *reg_gr, int dmem[DATA_MEM_SIZE], char *reg_br, char *
    op_code, char *imm_op, char *dest_gr, char *src1_gr, char *src2_gr,
     char *dest_br, int *short_imm_data, int *branch_imm_data, int *
    long_imm_data, int *pc_goto, int *alu_result, int *mult_result, int
    *mem_rdata);
void alu(char op_code, int A_op, int B_op, int *alu_result);
void mult(char op_code, int A_op, int B_op, int *mult_result);
void ctrl(int *pc_goto, char op_code, int offset, char br);
void mem(int dmem[DATA_MEM_SIZE], char op_code, char address, int
    mem_wdata, int *mem_rdata);
void writeback(int *reg_gr, char *reg_br, char *dest_gr, char *dest_br,
    char *op_code, int *alu_result, int *mult_result, int *mem_rdata);
void r_vex_core(int *done, int *cycles);

#endif
```

B.1.5 ρ-Vex Processor Core (r_vex_top.c)

```
/*****************************************************************/
/* Name:    r-Vex processor                                      */
/* Purpose: Control and DataPath of Academic VLIW procesor       */
```

```
/* Author:  Jeferson Santiago da Silva                                */
/* Notes:   Based on r-Vex suite (https://code.google.com/p/r-vex/)   */
/**********************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "r_vex_top.h"

/**********************************************************************/
/* PC calculation routine                                             */
/**********************************************************************/
void advance_pc(int *next_pc, int pc_goto)
{
    *next_pc = pc_goto;
    return;
}

/**********************************************************************/
/* Fetch intruction stage routine                                     */
/**********************************************************************/

void fetch(unsigned int imem[SYLLABLE_NUM][INST_MEM_SIZE], unsigned int
    *syllable, int pc)
{
    int i;
    L1: for (i = 0; i < SYLLABLE_NUM; i++) {
        syllable[i] = imem[i][pc];
    }
    return;
}

/**********************************************************************/
/* Decode intruction stage routine                                    */
/**********************************************************************/
void decode(unsigned int *syllable, char *op_code, char *imm_op, char *
   dest_gr, char *src1_gr, char *src2_gr, char *dest_br, int *
   short_imm_data, int *branch_imm_data, int *long_imm_data)
{
    int i = 0;

    /* Get operations and Operands */
    L1: for (i = 0; i < SYLLABLE_NUM; i++) {
        /* Get operations */
        op_code[i] = OP_CODE(syllable[i]);
        imm_op[i] = IMM_OP(syllable[i]);

        /* Get Operands */
        dest_gr[i] = DEST_GR(syllable[i]);
        src1_gr[i] = SRC1_GR(syllable[i]);
        src2_gr[i] = SRC2_GR(syllable[i]);
        dest_br[i] = DEST_BR(syllable[i]);

        short_imm_data[i] = SHORT_IMM_DATA(syllable[i]);
        branch_imm_data[i] = BRANCH_IMM_DATA(syllable[i]);
        long_imm_data[i] =  (LONG_HI_IMM_DATA(syllable[i]) << 11) +
           LONG_LO_IMM_DATA(syllable[i]);
    }
```

```
    return;
}

/*****************************************************************/
/* Execute stage routines                                        */
/*****************************************************************/
void execute(int *reg_gr, int *dmem, char *reg_br, char *op_code, char
    *imm_op, char *dest_gr, char *src1_gr, char *src2_gr, char *dest_br
    , int *short_imm_data, int *branch_imm_data, int *long_imm_data,
    int *pc_goto, int *alu_result, int *mult_result, int *mem_rdata)
{
    int i = 0;
    int A_op[SYLLABLE_NUM];
    int B_op[SYLLABLE_NUM];
    int mem_wdata = reg_gr[dest_gr[MEM_SLOT]];
    char br = reg_br[dest_br[MEM_SLOT]];

    // Get operands
    L1: for (i = 0; i < SYLLABLE_NUM; i++) {
        A_op[i] = reg_gr[src1_gr[i]];
        switch (imm_op[i])
        {
            // Registers operation
            case NO_IMM:
                B_op[i] = reg_gr[src2_gr[i]];
                break;
            // Short Immediate operation
            case SHORT_IMM:
                B_op[i] = short_imm_data[i];
                break;
            // Branch Immediate operation
            case BRANCH_IMM:
                B_op[i] = branch_imm_data[i];
                break;
            // Long immediate operation
            default:
                B_op[i] = long_imm_data[i];
                break;
        }
    }

    /* Control operations */
    ctrl(pc_goto, op_code[CTRL_SLOT], branch_imm_data[CTRL_SLOT], (char
        ) br);

    /* Perform ALU operations */
    L2: for (i = 0; i < ALU_NUM; i++) {
        alu(op_code[i], A_op[i], B_op[i], &alu_result[i]);
    }

    /* Perform MUL operations */
    L3: for (i = 0; i < MUL_NUM; i++) {
        mult(op_code[INI_MUL_SLOT + i], A_op[INI_MUL_SLOT + i], B_op[
            INI_MUL_SLOT + i], &mult_result[i]);
    }

    /* Perform memory operations */
```

```
    mem(dmem, op_code[MEM_SLOT], (char)A_op[MEM_SLOT], mem_wdata,
        mem_rdata);

    return;
}

/*****************************************************************/
/* ALU routines                                                  */
/*****************************************************************/
void alu(char op_code, int A_op, int B_op, int *alu_result)
{
    // Operation selection
    switch (op_code) {
        // Add
        case ADD:
            *alu_result = ALU_ADD(A_op, B_op);
            break;
        // Bitwise AND
        case AND:
            *alu_result = ALU_AND(A_op, B_op);
            break;
        // Bitwise complement and AND
        case ANDC:
            *alu_result = ALU_ANDC(A_op, B_op);
            break;
        // Maximum signed
        case MAX:
            *alu_result = ALU_MAX(A_op, B_op);
            break;
        // Maximum unsigned
        case MAXU:
            *alu_result = ALU_MAXU(A_op, B_op);
            break;
        // Minimum signed
        case MIN:
            *alu_result = ALU_MIN(A_op, B_op);
            break;
        // Minimum unsigned
        case MINU:
            *alu_result = ALU_MINU(A_op, B_op);
            break;
        // Bitwise OR
        case OR:
            *alu_result  = ALU_OR(A_op, B_op);
            break;
        // Bitwise complement and OR
        case ORC:
            *alu_result  = ALU_ORC(A_op, B_op);
            break;
        // Shift left 1 and add
        case SH1ADD:
            *alu_result = ALU_SH1ADD(A_op, B_op);
            break;
        // Shift left 2 and add
        case SH2ADD:
            *alu_result  = ALU_SH2ADD(A_op, B_op);
            break;
        // Shift left 3 and add
```

```
case SH3ADD:
    *alu_result = ALU_SH3ADD(A_op, B_op);
    break;
// Shift left 4 and add
case SH4ADD:
    *alu_result = ALU_SH4ADD(A_op, B_op);
    break;
// Shift left
case SHL:
    *alu_result = ALU_SHL(A_op, B_op);
    break;
// Shift right signed
case SHR:
    *alu_result = ALU_SHR(A_op, B_op);
    break;
// Shift right unsigned
case SHRU:
    *alu_result = ALU_SHRU(A_op, B_op);
    break;
// Bitwise XOR
case XOR:
    *alu_result = ALU_XOR(A_op, B_op);
    break;
// Compare: equal
case CMPEQ:
    *alu_result = ALU_CMPEQ(A_op, B_op);
    break;
// Compare: greater equal signed
case CMPGE:
    *alu_result = ALU_CMPGE(A_op, B_op);
    break;
// Compare: greater equal unsigned
case CMPGEU:
    *alu_result = ALU_CMPGEU(A_op, B_op);
    break;
// Compare: greater signed
case CMPGT:
    *alu_result = ALU_CMPGT(A_op, B_op);
    break;
// Compare: greater unsigned
case CMPGTU:
    *alu_result = ALU_CMPGTU(A_op, B_op);
    break;
// Compare: less than equal signed
case CMPLE:
    *alu_result = ALU_CMPLE(A_op, B_op);
    break;
// Compare: less than equal unsigned
case CMPLEU:
    *alu_result = ALU_CMPLEU(A_op, B_op);
    break;
// Compare: less than signed
case CMPLT:
    *alu_result = ALU_CMPLT(A_op, B_op);
    break;
// Compare: less than unsigned
case CMPLTU:
    *alu_result = ALU_CMPLTU(A_op, B_op);
```

```
            break;
        // Compare: not equal
        case CMPNE:
            *alu_result  = ALU_CMPNE(A_op, B_op);
            break;
        // Logical NAND
        case NANDL:
            *alu_result  = ALU_NANDL(A_op, B_op);
            break;
        // Logical NOR
        case NORL:
            *alu_result  = ALU_NORL(A_op, B_op);
            break;
        // Logical OR
        case ORL:
            *alu_result  = ALU_ORL(A_op, B_op);
            break;
        // Move GR to BR
        case MTB:
            *alu_result  = A_op & 1;
            break;
        // Subtract
        case SUB:
            *alu_result  = ALU_SUB(A_op, B_op);
            break;
        // Sign extend byte
        case SXTB:
            *alu_result  = ALU_SXTB(A_op);
            break;
        // Sign extend half word
        case SXTH:
            *alu_result  = ALU_SXTH(A_op);
            break;
        // Zero extend byte
        case ZXTB:
            *alu_result  = ALU_ZXTB(A_op);
            break;
        // Zero extend half word
        case ZXTH:
            *alu_result  = ALU_ZXTH(A_op);
            break;
        // Copy s1 to other location
        case MOV:
            *alu_result  = A_op;
            break;
        // Default - Decode Error
        default:
            break;
    }
    return;
}

/*****************************************************************/
/* Multiplication routines */
/*****************************************************************/
void mult(char op_code, int A_op, int B_op, int *mult_result)
{
    // Operation selection
```

```
    switch (op_code) {
        // Multiply signed low 16 x low 16 bits
        case MPYLL:
            *mult_result = MUL_MPYLL(A_op, B_op);
            break;
        // Multiply unsigned low 16 x low 16 bits
        case MPYLLU:
            *mult_result = MUL_MPYLLU(A_op, B_op);
            break;
        // Multiply signed low 16 (s1) x high 16 (s2) bits
        case MPYLH:
            *mult_result = MUL_MPYLH(A_op, B_op);
            break;
        // Multiply unsigned low 16 (s1) x high 16 (s2) bits
        case MPYLHU:
            *mult_result = MUL_MPYLHU(A_op, B_op);
            break;
        // Multiply signed high 16 x high 16 bits
        case MPYHH:
            *mult_result= MUL_MPYHH(A_op, B_op);
            break;
        // Multiply unsigned high 16 x high 16 bits
        case MPYHHU:
            *mult_result = MUL_MPYHHU(A_op, B_op);
            break;
        // Default - Decode Error
        default:
            break;
    }

    return;
}

/*****************************************************************/
/* Control routines                                              */
/*****************************************************************/
void ctrl(int *pc_goto, char op_code, int offset, char br)
{
    // Operation selection
    switch (op_code)
    {
        // Unconditional relative jump
        case GOTO:
            ctrl_goto(pc_goto, offset);
            break;
        // Conditional relative branch on true condition
        case BR:
            ctrl_br(pc_goto, offset, br);
            break;
        // Conditional relative branch on false condition
        case BRF:
            ctrl_brf(pc_goto, offset, br);
            break;
        default:
            break;
    }
    return;
}
```

```
/*****************************************************************/
/* Memory access routines                                        */
/*****************************************************************/
void mem(int *dmem, char op_code, char address, int mem_wdata, int *
    mem_rdata)
{
    // Operation selection
    switch (op_code) {
        // Load word
        case LDW:
            mem_ldw(dmem, address, mem_rdata);
            break;
        // Store word
        case STW:
            mem_stw(dmem, address, mem_wdata);
            break;
        // Default - Decode Error
        default:
            break;
    }
    return;
}

/*****************************************************************/
/* Writeback routine                                             */
/*****************************************************************/
void writeback(int *reg_gr, char *reg_br, char *dest_gr, char *dest_br,
     char *op_code, int *alu_result, int *mult_result, int *mem_rdata)
{
    int i;
    L1: for (i = 0; i < SYLLABLE_NUM; i++) {

        if (ALU_TYPE(op_code[i])) {
            if (dest_gr[i] == 0) {
                reg_br[dest_br[i]] = alu_result[i];
            }
            else {
                reg_gr[dest_gr[i]] = alu_result[i];
            }
        }

        switch (i) {
            case CTRL_SLOT:
                if (CTRL_TYPE(op_code[CTRL_SLOT])) {
                }
                break;
            case 1:
                if (MUL_TYPE(op_code[i])) {
                    reg_gr[dest_gr[i]] = mult_result[0];
                }
                break;
            case 2:
                if (MUL_TYPE(op_code[i])) {
                    reg_gr[dest_gr[i]] = mult_result[1];
                }
                break;
            case MEM_SLOT:
```

```
                if (MEM_TYPE(op_code[MEM_SLOT])) {
                    reg_gr[dest_gr[i]] = *mem_rdata;
                }
                break;
            }

        }

}

/*****************************************************************/
/* r-Vex core                                                    */
/*****************************************************************/
void r_vex_core(int *done, int *cycles)
{
    int i, j = 0;

    /* Flow control */
    int next_pc = 0;
    int pc_goto = 0;

    /* Registers: general purpose and branch */
    int reg_gr[REGISTER_GR_NUM] = {0};
    char reg_br[REGISTER_BR_NUM] = {0};

    /* Fecth */
    unsigned int syllable[SYLLABLE_NUM];

    /* Decode variables */
    char op_code[SYLLABLE_NUM];
    char imm_op[SYLLABLE_NUM];

    char dest_gr[SYLLABLE_NUM];
    char src1_gr[SYLLABLE_NUM];
    char src2_gr[SYLLABLE_NUM];
    char dest_br[SYLLABLE_NUM];

    int short_imm_data[SYLLABLE_NUM];
    int branch_imm_data[SYLLABLE_NUM];
    int long_imm_data[SYLLABLE_NUM];
    char c;

    int dmem[DATA_MEM_SIZE];

    int alu_result[ALU_NUM];
    int mult_result[MUL_NUM];
    int mem_rdata;

    /* Main Loop */
    r_vex_core_label0: while (1) {

        /* PC calculation */
        advance_pc(&next_pc, pc_goto);

        /* Fix PC offset [nPC = nPC + 1]*/
        pc_goto = next_pc + 1;
        *cycles = *cycles + 1;
```

```
        /* Fetch intruction stage routine */
        fetch(imem, syllable, next_pc);

        /* Decode intruction stage routine */
        decode(syllable, op_code, imm_op, dest_gr, src1_gr, src2_gr,
            dest_br, short_imm_data, branch_imm_data, long_imm_data);

        /* Detects end of program */
        L1: for (i = 0; i < SYLLABLE_NUM; i++) {
             if (STOP_TYPE(op_code[i])) {
                 *done = 1;
                 return;
             }
        }

        /* Execution Stage */
        execute(reg_gr, dmem, reg_br,  op_code, imm_op, dest_gr,
            src1_gr, src2_gr, dest_br, short_imm_data, branch_imm_data,
             long_imm_data, &pc_goto, alu_result, mult_result, &
            mem_rdata);

        /* Writeback stage */
        writeback(reg_gr, reg_br, dest_gr, dest_br, op_code, alu_result
            , mult_result, &mem_rdata);

    }
}
```

B.1.6 ρ-Vex Processor Testbench (r_vex_tb.h)

```
/*****************************************************************/
/* Name:     r-Vex processor Testbench                           */
/* Purpose: Testbench Control and Datapath of VLIW processor     */
/* Author:   Jeferson Santiago da Silva                          */
/* Notes:    Based on r-Vex suite (https://code.google.com/p/r-vex/) */
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "r_vex.h"

int main()
{
    int i = 0, j = 0;
    int program_done = 0;
    int cycles = 0;

    /* r-Vex core */
    r_vex_core(&program_done, &cycles);

    printf("-----------------------------------\n");
    printf("Final execution\n");
    printf("Program done:          %d\n", program_done);
    printf("Number of instructions: %d\n", cycles);
    printf("Result:                ");
```

```
    if (cycles != 94) {
        printf("Error\n");
    }
    else {
        printf("Passed\n");
    }
    printf("--------------------------------------\n");

    return 0;

}
```

B.2 Synthesis Directives Generated by DSE Script (directives.tcl)

```
set_directive_inline "advance_pc"
set_directive_inline "decode"
set_directive_inline "execute"
set_directive_inline "ctrl"
set_directive_inline "mem"
set_directive_inline "writeback"
set_directive_inline "r_vex_core"
set_directive_inline -off "alu"
set_directive_allocation -limit 1 -type function "execute/L2" alu
set_directive_inline -off "mult"
set_directive_allocation -limit 1 -type function "execute/L3" mult
set_directive_pipeline -off "decode/L1"
set_directive_pipeline -off "execute/L1"
set_directive_pipeline -off "execute/L2"
set_directive_pipeline -off "execute/L3"
set_directive_pipeline -off "writeback/L1"
set_directive_pipeline -off "r_vex_core/L1"
set_directive_pipeline -off "advance_pc"
set_directive_pipeline -off "decode"
set_directive_pipeline -off "execute"
set_directive_pipeline -off "alu"
set_directive_pipeline -off "mult"
set_directive_pipeline -off "ctrl"
set_directive_pipeline -off "mem"
set_directive_pipeline -off "writeback"
set_directive_pipeline -off "r_vex_core"
set_directive_latency -min 1 -max  1 "decode/L1"
set_directive_latency -min 1 -max  1 "execute/L1"
set_directive_latency -min 1 -max  1 "execute/L2"
set_directive_latency -min 1 -max  1 "execute/L3"
set_directive_latency -min 1 -max  1 "writeback/L1"
set_directive_latency -min 1 -max  1 "r_vex_core/L1"
set_directive_latency -min 1 -max  1 "advance_pc"
set_directive_latency -min 1 -max  1 "decode"
set_directive_latency -min 1 -max  1 "execute"
set_directive_latency -min 1 -max  1 "alu"
set_directive_latency -min 1 -max  1 "mult"
set_directive_latency -min 1 -max  1 "ctrl"
set_directive_latency -min 1 -max  1 "mem"
set_directive_latency -min 1 -max  1 "writeback"
set_directive_latency -min 1 -max  1 "r_vex_core"
```