REC 2014

Atas

X Jornadas sobre Sistemas Reconfiguráveis

13 de abril de 2014

Vilamoura - Algarve

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

isep Instituto Superior de Engenharia do Porto

UAlg
UNIVERSIDADE DO ALGARVE

Organizadores:
José Carlos Alves
Manuel Gericota

Edição: Comissão Organizadora da REC 2014
José Carlos Alves e Manuel Gericota

ISBN: 978-989-98875-1-0

Conteúdo

 ISBN: 978-989-98875-1-0

Prefácio

As X Jornadas sobre Sistemas Reconfiguráveis decorrem no Crowne Plaza Vilamoura Hotel, no Algarve, a 13 de abril de 2014. Esta edição vem na continuação de uma sequência de eventos que teve início também no Algarve, onde, em 2005 e sob a responsabilidade da Universidade local, se realizou o primeiro evento, com edições anuais posteriores na Faculdade de Engenharia da Universidade do Porto (2006), no Instituto Superior Técnico da Universidade Técnica de Lisboa (2007), no Departamento de Informática da Universidade do Minho (2008), na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (2009), na Universidade de Aveiro (2010), Faculdade de Engenharia da Universidade do Porto (2011) e no Instituto Superior de Engenharia de Lisboa (2012) e na Universidade de Coimbra (2013). Ao longo de todas estas edições, as Jornadas têm conseguido constituir-se como o ponto de encontro anual para a comunidade científica de língua portuguesa com reconhecida atividade de investigação e desenvolvimento na área dos sistemas eletrónicos reconfiguráveis.

O programa das X Jornadas – REC 2014 – tem uma estrutura um pouco mais curta que o das edições anteriores, decorrendo durante apenas um dia, estando, no entanto e pela primeira vez, associado a um encontro internacional, o ARC'2014, *10th International Symposium on Applied Reconfigurable Computing* que decorre no mesmo local nos três dias subsequentes.

Este ano, as Jornadas incluem uma apresentação convidada - "FP7 HARNESS: Managing Heterogeneous Resources for the Cloud" - , apresentada pelo colega José Gabriel Coutinho do Imperial College London. A apresentação versa um projeto que tem por objetivo a introdução de tecnologias heterogéneas, tais como as FPGAs, em plataformas de *cloud computing*.

Agradecemos ao colega José Gabriel a disponibilidade para partilhar com os participantes da REC 2014 as suas experiências e conhecimentos.

O programa conta ainda com a apresentação de 10 comunicações regulares nas áreas das arquiteturas de comunicação, arquiteturas multiprocessamento, linguagens de especificação e otimização, incluindo novas abordagens ao projeto de sistemas reconfiguráveis. Estas contribuições foram todas aprovadas para apresentação e publicação pelo Comité Científico. Todas as contribuições foram sujeitas a três revisões.

A organização destas Jornadas contou com o apoio de diversas pessoas e entidades, às quais gostaríamos de expressar o nosso agradecimento. Em primeiro lugar, devemos um agradecimento especial aos autores que contribuíram com os trabalhos incluídos nestas Atas, bem como aos membros do Comité Científico pelo excelente trabalho produzido, concretizado em revisões que, estamos certos, permitiram melhorar a qualidade dos trabalhos submetidos.

Igualmente os nossos agradecimentos aos colegas da Universidade do Algarve, João Lima, Rui Marcelino e José Mariano pelo imprescindível apoio logístico concedido à organização destas Jornadas, e ao colega João Bispo pela atualização e manutenção da página *web*.

Esperamos que esta edição das Jornadas constitua, uma vez mais, um espaço para divulgação e discussão dos trabalhos apresentados, bem como de convívio aberto a todos quantos partilham interesses na área dos sistemas eletrónicos reconfiguráveis, contando revê-los a todos nas jornadas do próximo ano.

José Carlos Alves, Faculdade de Engenharia da Universidade do Porto
Manuel Gericota, Instituto Superior de Engenharia do Porto

 ISBN: 978-989-98875-1-0

Comissão Organizadora

José Carlos Alves	Faculdade de Engenharia da Universidade do Porto
Manuel Gericota	Instituto Superior de Engenharia do Porto

Comité Científico

Ana Antunes	Instituto Politécnico de Setúbal
André Fidalgo	Instituto Superior de Engenharia do Porto
Anikó Costa	Universidade Nova de Lisboa – UNINOVA
António Esteves	Universidade do Minho
António Ferrari	Universidade de Aveiro – IEETA
Arnaldo Oliveira	Universidade de Aveiro – IT
Fernando Gonçalves	Instituto Superior Técnico – INESC-ID
Gabriel Falcão	Universidade de Coimbra – IT
Helena Sarmento	Instituto Superior Técnico – INESC-ID
Horácio Neto	Instituto Superior Técnico – INESC-ID
Iouliia Skliarova	Universidade de Aveiro – IEETA
João Bispo	Fac. de Engenharia da Universidade do Porto
João M. P. Cardoso	Fac. de Engenharia da Universidade do Porto – INESC Porto
João Canas Ferreira	Fac. de Engenharia da Universidade do Porto – INESC Porto
João Lima	Universidade do Algarve
Jorge Lobo	Universidade de Coimbra – ISR
José Augusto	Fac. de Ciências da Universidade de Lisboa – INESC-ID
José Carlos Alves	Fac. de Engenharia da Universidade do Porto – INESC Porto
José Carlos Metrôlho	Instituto Politécnico de Castelo Branco
José Gabriel Coutinho	Imperial College London
José Silva Matos	Fac. de Engenharia da Universidade do Porto – INESC Porto
Leonel Sousa	Instituto Superior Técnico – INESC-ID
Luis Gomes	Universidade Nova de Lisboa – UNINOVA
Luis Cruz	Universidade de Coimbra – IT
Luís Nero	Universidade de Aveiro – IT
Manuel Gericota	Instituto Superior de Engenharia do Porto
Marco Gomes	Universidade de Coimbra – IT
Mário Véstias	Instituto Superior de Engenharia do Lisboa – INESC-ID
Mário Calha	Fac. de Ciências da Universidade de Lisboa – LaSIGE

Morgado Dias	Universidade da Madeira
Nuno Roma	Instituto Superior Técnico – INESC-ID
Orlando Moreira	Ericsson
Paulo Flores	Instituto Superior Técnico – INESC-ID
Paulo Teixeira	Instituto Superior Técnico – INESC-ID
Pedro C. Diniz	University of Southern California
Ricardo Chaves	Instituto Superior Técnico – INESC-ID
Ricardo Machado	Universidade do Minho
Rui Marcelino	Instituto Superior de Engenharia da Universidade do Algarve
Valeri Skliarov	Universidade de Aveiro – IEETA

Comunicação convidada

 ISBN: 978-989-98875-1-0

FP7 HARNESS: Managing Heterogeneous Resources for the Cloud

José Gabriel de Figueiredo Coutinho

Custom Computing Research Group
Imperial College London
Londres – Reino Unido

Most cloud service offerings are based on homogeneous commodity resources to provide low-cost application hosting. However, this business model has reached a limit in satisfying performance for important classes of applications, such as geo-exploration and real-time business analytics. The HARNESS project aims to fill this gap by developing architectural principles that enable the next generation cloud platforms to introduce heterogeneous technologies such as FPGAs, programmable routers, and SSDs, and provide as a result vastly increased performance, reduced energy consumption, and lower cost profiles. In this talk we focus on three challenges for supporting heterogeneous computing resources in the context of a cloud platform, namely:

1. cross-optimisation of heterogeneous computing resources;
2. resource virtualisation, and;
3. programming heterogeneous platforms.

Short bio: José Gabriel de Figueiredo Coutinho is an associate researcher working in the Custom Computing Research Group at Imperial College London. He received his M.Eng. degree in Computer Engineering from Instituto Superior Técnico, Lisbon, Portugal in 1997. From 2000 and 2007 he received his M.Sc. and PhD in Computing Science from Imperial College London. Since 2005, he has been involved in UK and EU research projects such as Ubisense, hArtes, REFLECT and HARNESS. His main interests include mapping and optimising high-level descriptions to heterogeneous reconfigurable platforms and aspect-oriented design. He has published over 40 research papers in peer-referred journals and international conferences and has contributed to two book publications.

 ISBN: 978-989-98875-1-0

Cálculo e Reconfiguração

 ISBN: 978-989-98875-1-0

Building Reconfigurable Systems Using Open Source Components

José T. de Sousa, Carlos A. Rodrigues, Nuno Barreiro, João C. Fernandes
INESC-ID Lisboa
{*jose.desousa,carlosaarodrigues,nuno.barreiro,joaocfernandes*}*@inesc-id.pt*

Abstract

The complexity of reconfigurable systems is in great part the complexity of the hardware-software environment where they are integrated. Many different reconfigurable architectures and respective development tools have been proposed in the literature, but few reconfigurable computing startups have succeeded at creating sustainable business models. The extraordinary infrastructure needed to operate any computer in general, and a reconfigurable computer in particular, takes several years to develop and requires intensive capital investment. This problem is slowly but effectively being addressed in the open source community by making available, free of charge, high quality hardware and software components which innovative companies can use to assemble complex systems. This paper presents both a methodology for creating reconfigurable systems on chip (SoCs), using open source components, and a base SoC created using the proposed methodology. An experimental evaluation of the open source components used and of the system itself is provided.

1. Introduction

1.1. Motivation

As the world becomes more interconnected, it is time for things, in addition to people, to join the cloud. The *Internet of Things*, as it is currently coined, offers a practically infinite number of possibilities for building innovative devices. A piece of semiconductor IP, which can be quickly configured to embody a device drawn from a brilliant engineer's mind, is now, more than ever, an extremely valuable asset. Differentiation normally demands that these Internet things have aggressive specifications in terms of area, performance and power consumption.

Such piece of IP is indeed a SoC. In its 25 years of existence, the IP market evolved from supplying small components to providing complete subsystems with one or more processors and several software components. SoCs normally contain one main IP system, responsible for top-level control, and a bunch of specialized IP subsystems.

Extreme size/performance/power specifications for a programmable IP subsystem are often not achievable using conventional processors. To address this problem, reconfigurable computing techniques have, in the last 20 years, gained relevance [1]. However, despite showing incontestable advantages w.r.t. good specification trade-offs, reconfigurable machines are still difficult to program and have found little practical use.

1.2. Problem

The current situation poses a big problem to new entrants in the semiconductor IP market, especially to the ones promoting reconfigurable systems. The value proposition of a reconfigurable system may be easy to articulate as increased performance per megahertz and square millimeter, but having access to every hardware and software building block needed for a complete system is expensive and/or takes too long.

1.3. Solution

The solution may be similar to the one found for Operating Systems (OS's): free open source OS's have emerged which in many aspects are better and more robust than commercial alternatives. The same may be true for open hardware descriptions.

1.4. Objectives

This project has two objectives. The first is to investigate whether a SoC built from open source IP cores can be silicon ready and built in a time frame comparable to using commercial IP cores. The second is to use this SoC to host a reconfigurable co-processor. To accomplish these goals, we envision the following steps:

1. build a base SoC using open source components wherever possible;
2. create an automatic build and regression test environment;
3. create a development environment where software components can be easily added, removed or modified;
4. develop and add a reconfigurable co-processor.

1.5. Outline

This paper is organized as follows. In section 2, a SoC base instance implemented with open source modules is described. section 3 describes the verification methods used with the proposed SoC. The programming tools are described in section 4. In section 5, the development flow

for the proposed SoC is outlined. In section 6, the reconfigurable computing infrastructure is discussed. Finally, in section 7, the early-stage results of this research and its main conclusions are presented.

2. SoC design

In order to illustrate the design of a SoC using open source components, this research used the OpenRISC Platform SoC (ORPSoC) [2], which is a template platform for creating SoCs based on the OpenRISC processor [3]. ORPSoC is a Python program, currently under development by the OpenRISC community, whose purpose is to help assemble SoCs using pre-existing IP cores. The user describes systems and cores using text configuration files, from which ORPSoC creates a directory tree containing simulation and FPGA emulation environments.

Other open source processors have been carefully considered, namely the Leon 2 processor [4], and the LatticeMico32 [5], and others. However, decisive success factors are whether there is an active community maintaining the code, and whether the toolchain is comprehensive and open. Leon 2 is no longer maintained as the company that gave origin to it has been acquired, and LatticeMico32 still lacks crucial tools such as a good debug environment. Soft processors that are not completely open such as Microblaze and its few clones cannot even be considered. After weighing few alternatives, OpenRISC was clearly the most advantageous choice.

This paper presents Blackbird, the example SoC shown in Fig. 1. This SoC is being assembled using a modified version of the ORPSoC program developed by the authors. This design is intended as a base SoC for autonomous and low power embedded systems with DSP capabilities. According to the target application, peripherals may be added or removed, which also means the addition or removal of their respective software drivers.

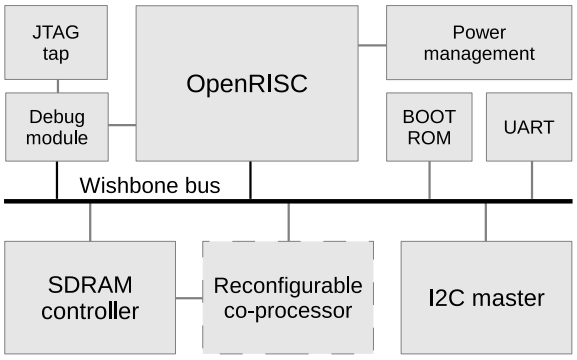


Figure 1. The Blackbird SoC.

2.1. OpenRISC and reconfigurable co-processor

The OpenRISC processor is a modern RISC processor, featuring 1 DMIPS of performance, similar to a commercial ARM9 core. OpenRISC is equipped with instruction and data caches of configurable size as well as instruction and data MMUs. It supports integer operations and optionally floating-point operations. Interrupts and exceptions of various types are also catered for. The OpenRISC is master of the system's Wishbone [6] bus, where many slave peripherals can be attached.

This research envisions the inclusion of a reconfigurable co-processor in the SoC as shown in Fig. 1. The co-processor is aimed at boosting system performance, especially in accelerating DSP like functions. The reconfigurable co-processor is a Wishbone slave but requires the crucial capability of accessing the central memory directly without intervention of the processor. For this reason, it possesses a master interface to the SDRAM controller.

2.2. Boot ROM

After reset, the processor executes code permanently stored in a Boot ROM. This code typically loads a program stored in a non-volatile memory into the central memory. The Boot ROM is a Wishbone slave.

2.3. SDRAM controller

The SRAM controller is the interface to an external central memory which is normally a single or double data rate (DDR) SDRAM. Newer generations of DDRs, such as DDR2 or DDR3, can achieve better performance and higher density in terms of stored bits per unit of area. The SDRAM controller is a Wishbone slave. However, the SDRAM controller is also a slave of the reconfigurable co-processor and must include an arbiter to choose between OpenRISC or co-processor accesses.

2.4. I2C Master

The I2C core is used to read and write to an external EEPROM, which serves as a non-volatile memory. I2C is a two-pin serial protocol to access off chip devices. It uses bidirectional clock and data pins. The I2C core is a Wishbone slave and an I2C master of the EEPROM device.

2.5. UART

The basic IO mechanism is the RS232 serial protocol implemented by a UART core. Currently, this is the only direct channel that software programs running on OpenRISC can use to communicate with the outside, sending or receiving data. In Blackbird, a setup using the UART at 115kb/s is employed.

2.6. JTAG

The interface for loading and debugging programs is the 4-pin JTAG interface. The JTAG pins are connected to the JTAG tap module in the SoC, responsible for implementing the JTAG protocol. The JTAG wires connect to an external pod, which in turn connects to a host computer using, for example, USB or Ethernet.

 ISBN: 978-989-98875-1-0

The host computer may run the program loader or debugger, or it can simply act as a proxy for another computer where these client programs may be run. A debugger program could also run locally on OpenRISC but, as the memory space for storing programs is a scarce resource, a remote debugger is preferred for embedded systems.

2.7. Debug module

The debug module sits between the JTAG tap and the Wishbone bus of which it is a master. It also has a direct interface to OpenRISC in order to monitor and control its execution. Being the second master of the Wishbone bus (OpenRISC is the first), the Debug module can read or write the central memory, which is useful for setting up soft breakpoints, access variables and other debug tasks [7].

2.8. Power management

In order to control the energy consumption of the SoC, a power management unit is needed. Note that this SoC is to be typically used in small battery operated electronic devices, for which autonomy is a crucial feature. The power management unit may, for example, contain timers that periodically wake the system from hibernation to perform tasks; during hibernation there is almost no energy consumption.

3. SoC verification

A processor-based system is a complex system where no exhaustive testing can be guaranteed. At best, the system is exercised in as many ways as possible, trying to replicate real situations and to stress critical design corners. No one-size-fits-all solution exists and a combination of techniques is often employed. The verification environment for the Blackbird SoC is illustrated in Fig. 2 and comprises three SoC verification methods: RTL (Verilog) simulation, SystemC simulation and FPGA emulation.

3.1. Test program and OpenOCD

Common to the three verification methods is the use of a C++ test bench (Test Program) and a universal on-chip debug tool called OpenOCD [8]. Using a C++ test bench provides great flexibility in generating test stimuli for the Device Under Test (DUT) and checking its responses. OpenOCD provides a unified target access mechanism with pre-built configurations supporting many commercial development platforms and boards.

The Test Program communicates with OpenOCD using a networked socket to send OpenOCD commands. A simple Test Program is, for instance, a Telnet client where some target specific commands are manually issued, including commands for loading the program, starting and halting execution, etc.

With OpenOCD, the same Test Program can seamlessly exercise our three targets: the RTL simulation, the SystemC simulation and the FPGA emulator. A GDB client can also

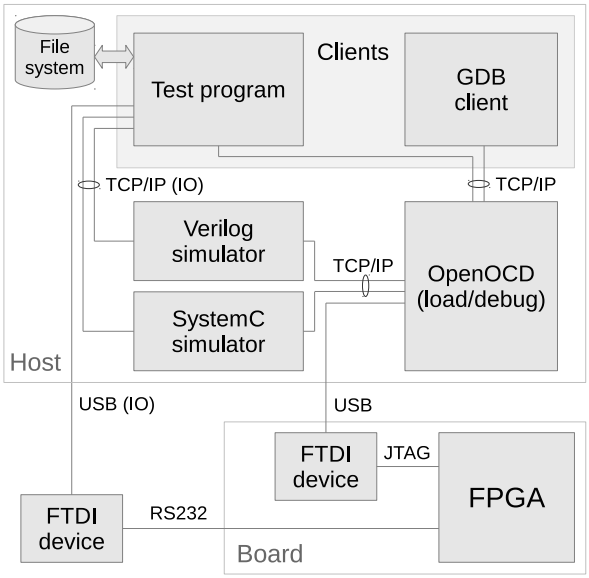


Figure 2. Verification environment.

connect to OpenOCD for program debugging: the user sets a remote target using a specific OpenOCD port; henceforth, all GDB functionalities can be used to debug remote embedded programs as if they were local (for further details see section 4.2).

3.2. RTL simulation

RTL simulation is the most commonly used method for SoCs' verification due to its accuracy, both in terms of timing and electrical behaviors. Timing accuracy is obtained via event driven simulation, where component delays dictate the instants when circuit is re-evaluated. Electrical behavior is modeled by multi-valued logic: not only 0 and 1 states can modeled but also other states such as X (0/1 conflict), U (unknown), Z (high-impedance). The most used language in the industry for RTL design and verification is the Verilog language.

RTL simulation can be slow if applied to large logic circuits such as CPUs. However, it is an indispensable tool to verify the design. Its level of detail is adequate even for the asynchronous parts of the design, namely clock domain crossings. In a Verilog design, a Verilog test bench module is needed for RTL simulation. The test bench generates stimuli to excite the design module and checks the correctness of its responses.

Simulating systems where most interactions are software driven requires a considerable amount of stimulus data. Writing such test benches in Verilog is hard. To solve this problem, a Verilog Procedural Interface (VPI) is normally employed. The VPI is a bridge between the test bench and an external C/C++ program, which actually does the verification, reducing the test bench to the role of a mere proxy.

The VPI is a compiled C/C++ library containing some functions that can be called by the C/C++ test program and some functions that can be called by the Verilog test bench.

Using a VPI does not reduce the duration of RTL simulation but greatly simplifies the writing of tests, using new or existing C code.

The Blackbird design is supplied with an RTL simulation environment, employing a VPI bridge. The Test Program exercises the design via OpenOCD, which communicates with the VPI bridge using a network socket. Given the fact that RTL simulation is slow, the Test Program selects a test suite of an adequately low complexity.

3.3. SystemC simulation

As the complexity of subsystems increases, it is not uncommon to run Verilog simulations for several hours or even a few days. In large projects, this long simulation cycles represent a critical schedule bottleneck. Hence, since the need to simulate remains crucial, designers have been forced to seek solutions other than RTL simulation.

The so called Embedded System Level (ESL) approaches have taken off in recent years to allow rapid design and verification of complex SoCs. Among such approaches, the SystemC initiative is one of the most relevant ones. SystemC is a set of C++ libraries, containing classes and methods for designing and verifying digital circuits and systems.

Blackbird is supplied with a SystemC simulation environment. The SystemC model of the design is obtained from the design's Verilog description by running a tool called Verilator. The SystemC model created by Verilator can be used as a class in a C++ program. In our design, a C++ program which instantiates the SoC's SystemC model and communicates with the OpenOCD gateway using a TCP/IP socket is currently under development.

SystemC simulation is cycle accurate and can provide full visibility of all the design signals. The Verilator tool can be configured to dump, while the model runs, the waveforms of those signals in a Value Change Dump (VCD) formatted file. These waveforms can then be visualized using any VCD viewer, including the free GTKWAVE program.

SystemC simulation is much faster than RTL simulation, but its time resolution and signal representation accuracy is considerably lower. The Verilator tool uses a fixed time step to evaluate the circuit and works with only two logic levels, 0 and 1. By contrast, RTL simulation uses a variable time step as a result of applying an event driven algorithm. Nevertheless, most problems can be identified and debugged at the Verilator level, a less onerous scheme.

3.4. FPGA emulation

SystemC simulation may still be a slow process, as the design must run for a long time before any problems manifest themselves. Ideally one would like to test the actual SoC, not yet available at the design stage. The best alternative is to use a circuit emulator, which is normally implemented with FPGA chips. Emulation provides the fastest verification method and, for circuits intended to run at low frequencies, emulation can even be real-time. As a reference, emulation speeds can be 1 to 2 orders of magnitude slower than real-time for complex SoCs implemented in the latest and densest IC technology nodes.

In spite of its performance, emulation provides poor visibility into the design's internal signals. When a bug is found, the designer first tries to diagnose the problem using software methods and/or logic analyzers to observe the interface signals. If observation of the interface signals is not sufficient, internal signals can be observed in the FPGA, with certain limitations, using an Integrated Logic Analyzer (ILA) such as Xilinx's Chipscope. The main limitations of ILAs are the maximum sampling frequency and the maximum amount of internal SRAM that can be used to store signals for observation.

Blackbird is emulated in the Terasic's DE0 Nano FPGA board, as supported by the ORPSoC program mentioned above. The board communicates with OpenOCD for programming and debugging using an USB cable. OpenOCD views this USB connection as a serial port. The USB cable connects to an USB-JTAG adapter implemented by an FTDI chip on the FPGA board. The FPGA board also communicates with the Test Program for data input/output via an USB/RS232 adapter implemented by another FTDI chip. The adapter is connected to the host computer by a second USB connection, viewed by the Test Program as a serial port, and is connected to the FPGA board using a 4-wire RS232 connection.

4. Toolchain

The OpenRISC SoC has a fully operational toolchain for barebone applications[1]. Compilation, debugging and profiling are based on the corresponding GNU tools. A proven stable version is available and a state of the art version is currently under development. An architectural simulator was developed by the OpenRISC community and constitutes the fist line of application development for the SoC.

A similar toolchain for Linux is also available, built with the libraries μClibc and BusyBox. Like the barebone toolchain, the Linux toolchain has both a stable and a development version.

4.1. Cross-compiler

The main barebone cross-compiler used for OpenRISC is GCC. Although an LLVM based version also exists, its status is still experimental and, hence, hardly suited for our purposes. There are essentially two versions endowed with an OpenRISC backend: the stable version built on GCC 4.5.1 and the development version glued to the current GCC development trunk, numbered 4.9.0. The official GCC distribution does not yet support OpenRISC and, therefore, the cross-compiler is available as a patch for the stable version and as a fork for the development version. Since some other divergent versions have emerged in the community, it is worth mentioning that the tested version is the one from the official OpenRISC repository.

[1] A barebone application runs directly on the machine, without any operating system, and must *know* the underlying hardware.

Many cross-compilers available for commercial processors are built around GCC but, without violating its GNU General Public License (GPL), include closed static libraries for many key functionalities, namely architectural simulation, IO, target loading and debugger connectivity. One might be tempted to develop clean room interface compatible processors, lured by the availability of GCC and other GNU tools. This is a mistake, as the closed components may be very hard to develop, even tough they represent a small part of the whole.

The OpenRISC cross-compiler uses Newlib and is 100% open source, allowing for customization and tweaking. This possibility makes a huge difference for the developer, who can reflect hardware changes on the cross-compiler, perhaps not trivially, but for sure without any kind of reverse engineering. Hardware changes may affect the command line options of the cross-compiler to select particular configurations of the CPU and SoC. There are options to make the compilation aware of the floating-point unit, the clock frequency for a specific board, etc.

4.2. Debugger

The GNU debugger (GDB) is also available for the OpenRISC SoC in two versions, 7.2 and 7.6.5. GDB provides an extensive range of facilities for execution tracing, halting on conditions, variable watching, etc. Usually run as a standalone application to debug local programs, GDB can also be run as a client/server application to debug a program in a remote system. The control of the target is left to a server which is accessed by the client via the `target remote` command. The client/server communication is done over TCP/IP and uses GDB's Remote Server Protocol (RSP).

The RSP server may run on the target system or on any machine that has some means of talking to the target. In our design, the RSP server runs on the same machine as the client and communicates with OpenOCD [8] which, in turn, communicates via USB/JTAG with the target and acts as a GDB proxy server. For the client this process is transparent and everything works as if the GDB server were running on the target.

4.3. Architectural simulator

The Or1ksim [9] is a highly configurable software simulator for the OpenRISC SoC. Its configuration file supports multiple parameters ranging from IO addresses to hardware modules. The main advantage of Or1ksim, when compared to hardware simulation methods, is speed. For instance, Linux runs smoothly on the Or1ksim, presenting a fast enough command console to the user. However, an architectural simulator is not cycle-accurate and can only be used to evaluate functional correctness without precise timing information. Or1ksim also includes an RSP server enabling debugging via TCP/IP with a GDB client, as shown in Fig. 3.

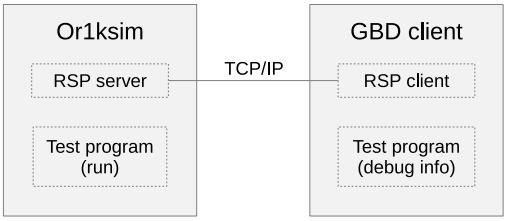


Figure 3. GDB with Or1ksim.

4.4. Profiler

The toolchain also packs the GNU profiler, GPROF, to measure and analyze the performance of programs. It records, for each function, the number of calls and the time spent in each call. It also shows information on the hierarchy of function calls. The profiler is an invaluable tool to optimize programs for performance and to debug large and/or complex code.

5. Development flow

The development flow is outlined in Fig. 4. It follows four distinct stages, each with several iterations between development and testing. Given the usual complexity, tests are seldom comprehensive: stepping back to prior stages and finding some previously uncaught issues is quite common.

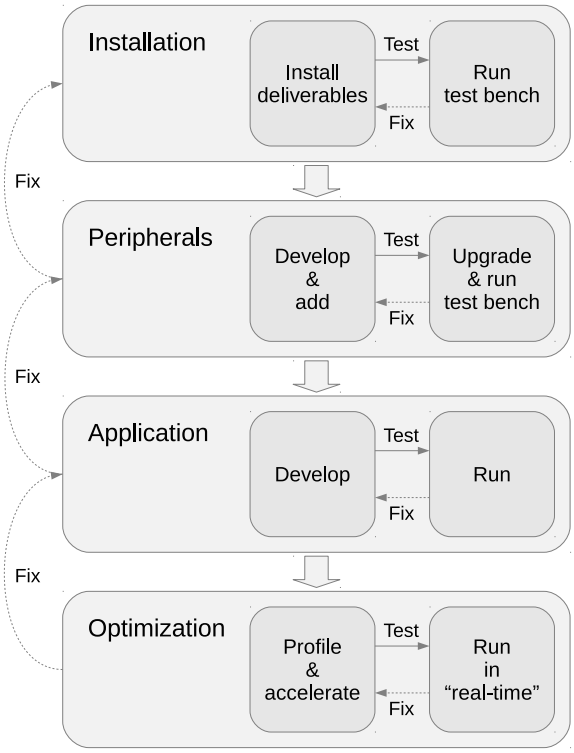


Figure 4. Development flow.

5.1. Installation

The Blackbird SoC specific deliverables are packaged in several containers: hardware (RTL Verilog, SystemC), GNU toolchain (GCC, GDB), architectural simulator (Or1ksim). But in order to have a fully operational environment, some other tools must also be installed and/or configured, namely an RTL simulator like the free Icarus program, Verilator and OpenOCD. Each package comes with a Makefile and a set of instructions to assist the installation process. Some of the packages also ship a set of tests with some indications as to where the detected problems may be occurring, but the overall procedure of obtaining a sane development system is still somewhat cumbersome.

The installation proved successful on Debian Linux (stable version Wheezy), both for 32 and 64 bits systems. Achieving a fully automated installation and testing of the whole development environment for Debian OS (at least) is one of the goals of this project.

5.2. Peripherals

Once the development environment is successfully built and tested, the hardware customization phase may begin: adding or removing modules from the OpenRISC core, configuring peripherals for the SoC, developing new peripherals suited for the aimed application. This phase includes Verilog writing and/or rewriting, but also entails the upgrading of the test bench, both for its hardware and software components.

The toolchain must be adapted to the new hardware, either via the available options mentioned in section 4.1, or via specific backend configuration files. When new peripherals are developed, the compiler backend may also need to be changed to accommodate the new functionalities.

The added functionalities may bring counterparts to the programming environment, namely by the creation of new software libraries. In that case, the test bench also needs to encompass those changes: programs using the new functions should be written and tested.

5.3. Application

The software application is developed in any favored IDE — an IDE integration is not provided. There are two straightforward ways of running and debugging the application, each with its own advantages and disadvantages: the Or1ksim architectural simulator and the FPGA SoC emulator. The other simulators, Icarus and Verilator, might be occasionally used to uncover possible hardware design issues, but they have the drawback of a lower speed factor, namely in the case of the RTL simulator.

Application testing with the Or1ksim has the advantage of running on the host computer, having no need for a board. However, the architectural simulator does not run as fast as the FPGA emulator and lacks some of the hardware features, namely the cache simulation. As such, running the application on the FPGA is often necessary for speed and timing accuracy.

In both situations, target debugging is performed using the GDB client, as described in section 4.3 for Or1ksim and in section 3.1 for the emulation on FPGA.

5.4. Optimization

Finally, once the application is built and debugged, optimization concerns arise. This stage makes extensive use of the profiler, GPROF, to search for redundant code and/or reveal more efficient implementations. The overall goal is to accelerate the application execution, using the FPGA emulation as a benchmark to measure the performance in "real-time" conditions (as mentioned in section 3.4, real-time is only achieved with the actual SoC, hence the quotes).

6. Reconfigurability

DSP tasks can be performed by fixed hardware accelerators, which is a common practice in embedded systems. However, a reconfigurable co-processor can provide better performance and energy footprint, while keeping the system programmable [10]. In fact, different tasks may use similar hardware and, as such, having multiple hardware accelerators with common subblocks constitutes a waste of silicon area and power. Reconfigurable accelerators can morph in runtime into each of the fixed accelerators, thus saving a huge amount of logic resources and energy spending. Three distinguishing features of the reconfigurable co-processors considered in this paper are: their *coarse-grain* nature [11]; their use of Direct Memory Access (DMA) [12]; their short reconfiguration time [13].

6.1. Coarse-grain arrays

For a long time, reconfigurable computing was associated with *fine-grain* arrays, i.e. FPGAs, which are almost homogeneous arrays of flexibly interconnected low input count look-up tables (LUTs), used for implementing generic logic functions. By contrast, coarse-grain arrays replace the LUTs with more sizeable structures such as ALUs, multipliers, shifters, etc.

However, FPGA reconfigurability is slow and wasteful in terms of area and power. Moreover, FPGAs often need to be combined with other chips in order to achieve a complete solution. To avoid multiple chips one could use an FPGA IP core. However, FPGA core startups have failed to succeed as the used blocks are bulky and yield expensive chips. In addition, programming FPGAs requires hardware design expertise which raises the cost of development.

Low node count coarse-grain arrays in the form of IP cores can be a good solution to this problem, but the main difficulty so far has been the fact that compiler technology for coarse-grain arrays is still in its infancy.

If we choose to incorporate a reconfigurable co-processor in Blackbird, we plan to tackle this problem in two ways: investigate methods allowing the compiler to be implemented with existing compiler frameworks such as GCC and LLVM; develop both a language and a compiler

for the co-processor.

The former appears to be a more difficult path, since the compiler depends critically on vectorization techniques which are a difficult research subject. In the latter, the language and compiler must be effective for at least the above mentioned compute kernels, and yet simple enough to make this effort sustainable.

6.2. DMA use

Reconfigurable co-processors are extremely useful for processing large amounts of data with a repetitive compute pattern. DSP algorithms such as FFTs, FIR and IIR filters are good examples of the compute patterns that reconfigurable co-processors excel at. These compute kernels operate on well defined data blocks. Most of the acceleration obtained with reconfigurable co-processors comes from the fact that the data-blocks are placed in the multiple internal RAMs of the co-processor, so that they can be accessed in parallel.

With a given configuration, the co-processor implements a computation graph with several parallel data sources, compute nodes and data sinks. If the co-processor is simply attached to the system bus, data must be accessed through the memory hierarchy. Data are accessed in small blocks, moved first to the cache and then to the co-processor. Data blocks produced by the co-processor need to be read by the processor and stored in the central memory using the cache. This process of data movement may take too long and offset the acceleration obtained in the engine itself. A DMA block can solve this problem by accessing the largest possible data blocks with a single memory transaction, efficiently moving data between the co-processor memories and the central memory.

6.3. Low reconfiguration time

In order to have a small reconfigurable co-processor, the number of reconfigurable elements should be minimized, which also means that the co-processor needs to reconfigure itself more frequently in order to maximize the use of its elements. In an extreme situation, a fixed co-processor implements all accelerators needed by the algorithm: when one accelerator is active the hardware of the others is idle, which is a common source of inefficiency.

Therefore, Run-Time Reconfiguration (RTR) is needed [14], and the faster the reconfiguration, the better the performance. In the approach we envision, using a coarse-grain array, the configuration words should be kept small enough to fit the implementation of caching schemes similar to instruction caching.

7. Conclusions and status

Assembling a SoC from pre-existing components may seem a trivial job but turns out to be an art on its own, where the whole SoC is much more valuable than the sum of its parts. However, since the different modules are reused in several projects, the components become gradually more stable and reliable, rendering the whole effort orders of magnitude smaller than building hardware and software from scratch. This is the essence of a business model consisting of building original systems from open source components.

One major barrier to entering this market is the open source licensing model of the components. There is a lack of a standard license agreement for open source hardware descriptions [15], as most open source software licenses are not generally applicable. Most open source IP cores are made available using the GNU Lesser General Public License (LGPL), but adoption by users still lacks expression.

The building of Blackbird is an ongoing process. The status of the project as of November 2013 is reported in the following subsections.

7.1. Hardware status

The status of the hardware components is summarized in Table 1. In an Altera Cyclone 4 device, Blackbird, configured with no floating-point/MAC unit, no reconfigurable co-processor and with 16kB I/D caches and MMUs, uses about 13k LEs. Tests with RTL simulation and FPGA emulation have solely been conducted with the out of the box material, accessible by using the ORPSoC program. The SystemC simulation was practically non-existing and is being developed almost from scratch. The results so far actually confirm that the open source IP cores behave well when ported to a user simulation environment.

Item	Comment
OpenRISC	OK in SystemC simulation, RTL simulation and FPGA
Reconfigurable co-processor	Under development
UART	OK in SystemC simulation and FPGA
BOOTROM	OK in SystemC simulation
I2C master	Open core not OK: being fixed
JTAG	OK in FPGA; SystemC under development, OK in RTL simulation and FPGA
SDRAM ctr.	OK in FPGA; Wishbone memory model used in SystemC simulation
Power Management	Not integrated

Table 1. Hardware status

7.2. Verification status

The status of the verification components is summarized in Table 2. It should be noted that no automatic build/test environment has yet been implemented and that these results have been obtained with a few asserted tests. FPGA emulation and RTL simulation have been run as provided by the ORPSoC program. The SystemC simulation environment is being fully developed in the scope of this

project. Being a cycle accurate simulator, the SystemC model produced by Verilator can generate code coverage metrics. It can also be coupled with GPROF to obtain preliminary profile data. Due to the Zero Wait State approximation, the profile data is optimistic compared to silicon results.

Item	Comment
Test program	Not started
OpenOCD	OK with RTL simulation and FPGA
RTL simulation	Only out of the box features tested
SystemC simulation	SystemC test bench to load and run programs from files developed and in use; integration with OpenOCD not started
FPGA emulation	Only out of the box features tested

Table 2. Verification status

7.3. Software status

The status of the software components is summarized in Table 3. These results have also been obtained with assorted tests, and no systematic build and test environment exists so far. In general, it can be said that these tools appear to be solid enough, which is a tremendous help in attaining this project's goals.

Item	Comment
GCC for C	Programs compiled correctly including a Linux kernel
GCC for C++	Simple programs compiled correctly
Or1ksim	Runs compiled programs and interacts with GDB correctly
GDB	Tested with Or1ksim, RTL and FPGA out of the box setup; tests with SystemC simulation not started
Profiler	Unknown status
Linux toolchain	Not started
Linux OS	Kernels compiled with GCC run correctly on Or1ksim and FPGA

Table 3. Software status

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under project PEst-OE/EEI/LA0021/2013.

References

[1] K. Pocek, R. Tessier, and A. DeHon. *Highlights of the First Twenty Years of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, chapter Birth and Adolescence of Reconfigurable Computing: A Survey of the First 20 Years of Field-Programmable Custom Computing Machines, pages 226–234. FCCM, April 2013.

[2] OpenRISC community. OpenRISC Reference Platform System-On-Chip. `https://github.com/openrisc/orpsoc`, 2013.

[3] OpenCores community. OpenRISC 1000 Architecture Manual. `http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf`, 2012.

[4] Zoran Stamenkovic, C. Wolf, GÃ¼nter Schoof, and Jiri Gaisler. Leon-2: General purpose processor for a wireless engine. In Matteo Sonza Reorda, Ondrej NovÃ¡k, Bernd Straube, Hana Kubatova, Zdenek KotÃ¡sek, Pavel KubalÃk, Raimund Ubar, and Jiri Bucek, editors, *DDECS*, pages 50–53. IEEE Computer Society, 2006.

[5] K.N. Horst. *Latticemico32*. Dign Press, 2012.

[6] OpenCores community. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. `http://cdn.opencores.org/downloads/wbspec_b4.pdf`, 2010.

[7] Nathan Yawn. Debugging System for OpenRisc 1000-based Systems. `http://opencores.org/websvn,filedetails?repname=adv_debug_sys&path=%2Fadv_debug_sys%2Ftrunk%2FDoc%2For1k_debug_sys_manual.pdf`, 2012.

[8] OpenOCD — Open On-Chip Debugger. `http://openocd.sourceforge.net`, 2013.

[9] Jeremy Bennett. OR1KSIM User Guide. `https://github.com/openrisc/or1ksim/tree/or32-master/doc`, 2009.

[10] Jose Teixeira De Sousa, Victor Manuel Goncalves Martins, Nuno Calado Correia Lourenco, Alexandre Miguel Dias Santos, and Nelson Goncalo Do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, 2012. US Patent 8276120 B2.

[11] E. Mirsky and A. DeHon. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 157–166, 1996.

[12] S. Rajamani and P. Viswanath. A quantitative analysis of processor-programmable logic interface. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 226–234, 1996.

[13] Melissa C. Smith and Gregory D. Peterson. Analytical modeling for high performance reconfigurable computers. In *Proceedings of the SCS international symposium on performance evaluation of computer and telecommunications systems*. Citeseer, 2002.

[14] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. De Wit. A dynamic reconfiguration run-time system. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 66–75, 1997.

[15] Eli Greenbaum. Open source semiconductor core licensing. *Harv. JL & Tech.*, 25:131, 2011.

Floating-Point Single-Precision Fused Multiplier-adder Unit on FPGA

Wilson José[⋆], Ana Rita Silva[⋆], Horácio Neto[†], Mário Véstias[‡]
[⋆]*INESC-ID,* [†]*INESC-ID/IST/UTL,* [‡]*INESC-ID/ISEL/IPL*
wilsonmaltez@inesc-id.pt, anaritasilva@inesc-id.pt, hcn@inesc-id.pt, mvestias@deetc.isel.pt

Abstract

The fused multiply-add operation improves many calculations and therefore is already available in some general-purpose processors, like the Itanium. The optimization of units dedicated to execute the multiply-add operation is therefore crucial to achieve optimal performance when running the overlying applications. In this paper, we present a single-precision floating-point fused multiply-add optimized unit implemented in FPGA and prepared to integrate a data flow processor for high-performance computing. The unit presents a numerical accuracy according to the IEEE 754-2008 standard and a performance and resource usage comparable with a state-of-the-art non-fused single-precision unit. The fused multiplier-adder was implemented targeting a Virtex-7 speed-grade -1 device and occupies 754 LUTs, 4 DSPs and achieves a maximum frequency of 361 MHz with 18 pipeline stages. A lighter low latency design of the same unit was also implemented in the same device presenting a resource usage of 845 LUTs, 2 DSPs and achieving a maximum frequency of 285 MHz.

1. Introduction

The floating-point multiplication-add operation is very important in several digital signal processing and control engineering applications, specifically applications which rely on the dot product. For that reason, the optimization of the hardware which performs the floating-point multiplication-add operation can result in significant gains in the execution of these applications. IBM recognized the importance of the multiplication-add operation and in 1990 unveiled the implementation of a floating-point fused multiply-add arithmetic execution unit on the RISC System 6000 (IBM RS/6000) chip [1], [2]. This floating-point arithmetic unit executes the equation $(A \times B) + C$ in a single instruction.

The advantages of a fused multiplier-adder (FMA) is that it not only improves the performance of an application that recursively executes multiplication followed by addition, but as well can execute a floating-point multiplication or addition by inserting fixed constants into its data path. However, this emulation of a floating-point multiplier or floating-point adder do not come without a cost, as the block's additional hardware imposes extra latency on the stand-alone instructions as compared to their original units [3]. Also, commonly the bit-widths and interconnectivities of internal blocks of the FMA more than double compared to those of floating-point adders and floating-point multipliers which complicates the process of routing the design and achieving the timing goals. Finally, the fused multiply-add unit is characterized by a heavy power consumption. Nevertheless, with the continued demand for 3D graphics, multimedia applications, and new advanced processing algorithms, the performance benefits of the fused multiply-add unit out-weights its drawbacks.

The work presented in this paper focus on another advantage of the fused multiply-add floating-point units which is increasing the numerical accuracy. By joining the multiply and add operation we can maintain the maximum precision through the intermediate results and only perform rounding after the add operation. In this context, we present the accuracy gain of our single-precision fused floating-point multiply-add design and respective area/speed trade-offs in relation to a non-fused unit. The unit presents a numerical accuracy according to the IEEE standard [4] and a performance and resource usage comparable with a state-of-the-art non-fused single-precision unit based on the Xilinx Core Generator floating-point multiplier and adder [5]. Also, the FMA unit is intended to be integrated with our reconfigurable processor which is part of multiprocessor system dedicated to accelerate a set of High-Performance Computing (HPC) applications [6]. This particular aspect was an important one to take into account when we designed the FMA unit, more on that later.

Next, we present the paper structure. In section 2, we discuss the most relevant works exploring fused floating-point multiply-add architectures with emphasis on the FPGA approaches. Section 3 describes the floating-point IEEE standard for single-precision numbers representation. In section 4 we present our FMA architecture and in section 5 we discuss the numerical accuracy results. Also, we compare our fused unit with non-fused single-precision multiply-add unit based on the Xilinx floating-point units in terms of the accuracy provided against the resources usage and maximum frequency achieved. Finally, we present our conclusions and discuss the future work in section 6.

2. Related Work

The multiply-add fused unit, was first proposed in 1990 [2]. After that, there were several works which tried to improve FMA architectures, but often target stand-alone Application-Specific Integrated Circuits (ASICs) or units integrated into general-purpose processor pipelines [7], [8],

[9]. In [3], the author gives a survey of the wide spectrum of FMA architectures developed from 1990 to 2007. The principle of fused operators has also been applied to other computations, such as fused dot products [10] and FFT [11].

The application-specific use of non-standard formats for improved numerical accuracy has been proposed for FPGAs in [12]. The use of non-standard formats to improve performance is presented in [13] for the use of a multiply-accumulate (MAC) unit. The design uses a PCS (Partial Carry Save) representation to achieve low latency at the addition stage but relies on application-specific knowledge of the input and output value ranges. Also, the authors only provide implementation results for the accumulator. Implementations of Radix 4 and 16 exponents showed improved addition speed but slower multiplication [14]. Also, it is stated that contrary to established belief, higher radix representations are useful for FPGA applications requiring IEEE 754 compliance, since they can deliver superior numerical performance while still using less FPGA resources.

The paper in [15] discusses the fundamentals of floating-point operations on FPGAs. The authors present a 270 MHz IEEE compliant double precision floating-point performance with a 9-stage adder pipeline and 14-stage multiplier pipeline mapped to a Xilinx Virtex4 FPGA (-11 speed grade). Their area requirement is approximately 500 slices for the adder and under 750 slices for the multiplier.

The paper in [16] examines existing solutions and proposes two new architectures for floating-point fused multiply-adds having heterogeneous input formats and considering the impact of different in-fabric features of recent FPGA architectures. The units are evaluated at the application level by modifying an existing high-level synthesis system to automatically insert the new units for computations on the critical path of three different convex solvers. The unit improves performance by up to 2.5× over the closest state-of-the-art competitor and also achieves a better numerical accuracy. Although, this results seem to come at the expense of a brutal increase in the resources needed (between 4x to 5x more than the Xilinx cores).

3. IEEE Floating-Point Representation

The FMA unit design follows the ANSI/IEEE-754 standard for binary floating-point numbers representation. The single-precision format is 32-bits wide and its structure is represented in figure 1.

Together, the three components form the number $x = (-1)^{sign} \times s \times 2^{(e-bias)}$. The use of a biased exponent format has virtually no effect on the speed or cost of exponent arithmetic (addition/subtraction), given the small number of bits involved. It does, however, facilitate zero detection (zero can be represented with the smallest biased exponent of 0 and an all zero significand) and magnitude comparison (we can compare normalized floating-point numbers as if they were integers) [17].

The notation "$23+1$" for the width of the significand is meant to explain the role of the hidden bit, which contributes to the precision of the number without requiring an

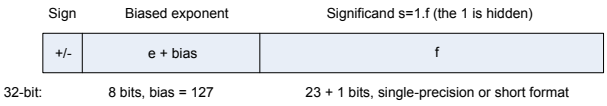


Figure 1. The ANSI/IEEE floating-point number format for single-precision.

extra bit in the representation. Denormals, or denormalized values, are defined as numbers without a hidden 1 and with the smallest possible exponent. They are provided to make the effect of underflow less abrupt. However, the standard do not require the support for denormals.

In 2008, the IEEE defined a standard for fused floating-point operations [4]. Basically, it states that the fused operation multiply-add, should compute $(A \times B) + C$ as if with unbounded range and precision, rounding only once to the destination format. The main objective of this work is to achieve the maximum accuracy on a fused multiply-add operation, following the stated IEEE guidelines.

4. Design Considerations

The FMA unit was designed with the intent to be part of the data path of the reconfigurable processor which integrates our multiprocessor architecture in [6]. This multiprocessor has a linear array based architecture in that each processor forwards data to the next processor further on the processor array. Each individual processor has a very simple instruction set and pipeline structure without the data hazards associated with general purpose processors. In the context of a matrix multiplication application and other similar applications, since none of the processors are dependent on each other results and produce independent chunks of data, as long there are enough bandwidth they can produce one result per cycle. This basically means that increasing the processor data path number of pipeline stages has a negligible impact on the application total execution time. Therefore, during the design we make a bigger effort in increasing the FMA maximum frequency and are not so concerned with the latency of the unit.

Other consideration is that the FPGAs slices have one register per each LUT, which means, pipelining in FPGA designs can come at little or no cost contrary to ASIC designs.

Since we have less restrictions in the number of pipeline stages, we focus less in parallelizing the hardware (which can be troublesome as this kind of optimization can end up in describing hardware at a very low level as is the case with implementing a leading zero counter predictor for example). Contrary to other approaches, [3], we do not try to parallelize the alignment of the adder operands with the mantissas multiplication as this can also turn more difficult using just the multiplier or the adder in a design.

In our design, like in the Xilinx floating-point units [5], we do not support denormals, as these only marginally extend the representable number range and lead to cost and speed overhead in hardware [17].

To achieve the increased precision we have proposed to,

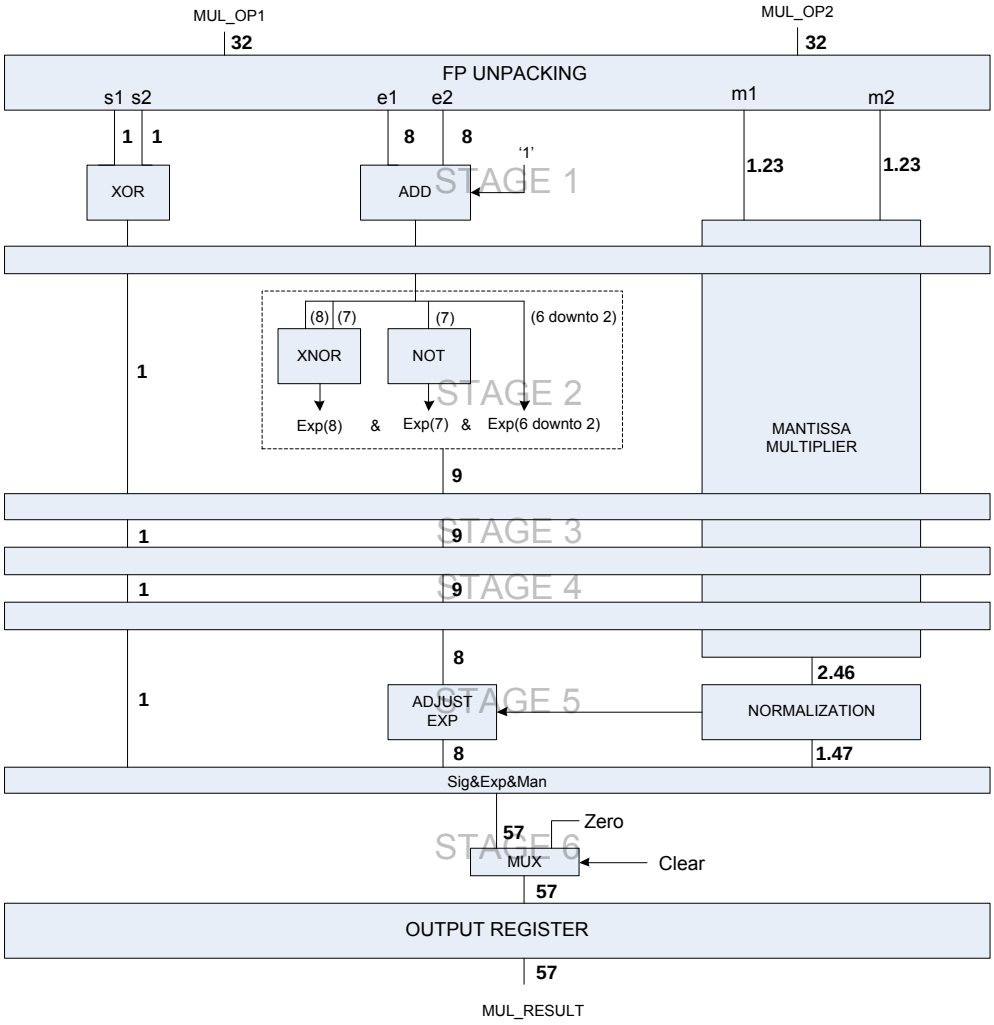


Figure 2. Floating-Point Multiplier Architecture.

we maintain the whole 48 bits from the multiplier result to the adder, contrary to a non-fused implementation. In the adder we maintain full precision in all operations till the rounding stage.

In the next section we will begin by describing the multiplier part of our design followed by the adder architecture.

5. FMA Architecture

The floating-point multiplier is divided in six stages as figure 2 shows.

When two valid 32-bits operands enter the first stage they are codified in the IEEE floating-point single-precision format, therefore these operands are unpacked into their respective components, signal, exponent and mantissa (the hidden one included). The resulting signal is immediately determined by the XOR of the two operand signals. The two exponents are also added in this stage (result exponent has 9 bits because the carry out is concatenated with the adder output). The bias subtraction is equivalent to adding -128+1 which can be done by having the carry in set to one and flipping the most significant bit of the exponent result. This last part is not that simple as we have to consider the carry out signal from the exponent adder to detect overflow/underflow situations. The correct two most significant bits of the result exponent are determined in the second pipeline stage of the multiplier as represented in the figure.

The mantissa multiplier spreads across 4 pipeline stages to achieve the maximum frequency possible. In the fifth pipeline stage the mantissa result may have to suffer a shift right to be normalized if the result is equal or bigger than 2. In this stage the exponent can be also adjusted depending if the mantissa needed to be normalized or not. The sign, exponent and mantissa are then concatenated and travel to the next stage. The last pipeline stage addresses the situation in which the result is equal to zero. When any of the operands is zero the clear signal is set to 1 and chooses a result in which both the sign, exponent and mantissa are zero. Finally, the multiplication result travels to the adder.

The floating-point adder architecture 12 pipeline stages is depicted in figure 3. The 12 pipeline architecture was achieved after an effort to maximize the maximum frequency and then reducing the area of the design. As it was referenced in beginning of the paper, we also designed a low latency version of the FMA architecture. A lighter unit may enable us to have more processors in our multiprocessor system which can be more advantageous than having less processors working at a higher frequency. This version is not represented in a figure as the modifications were not enough to justify it. A small description of those modifications is given in the end of this section.

The first step of the floating-point addition consists in the alignment of the operands. The operands need to be aligned to enable the mantissas addition. In our case the alignment of the operands is preceded by a swap step, necessary to avoid having two shifters. In the first pipeline stage we determine the operand to be right shifted, that is, the operand with smallest exponent. We subtract both exponents to calculate the shift amount and the sign of this operation lets us know which operand has the smallest exponent. Also, the mantissa from the second operand is concatenated with zeros to the right in order to have the same width of the mantissa from the multiplier result before entering the multiplexer. Finally, in this stage we verify if the two operands are equal and have different signs, being that the case we set the result exponent to zero.

The second and third stage of the floating-point adder correspond to the mantissas alignment. The shifter implemented accounts for the maximum shift necessary to achieve high precision. Figure 4 allows to develop a better understanding of the maximum shift we have to support to achieve the precision we desire. Basically, there are two different situations we have to consider.

When the second operand from the addition has a smaller exponent we must allow a maximum shift of 47 bits because of the ”rollback effect”, that is, we have to take into consideration all the bits which can ripple during the mantissas addition and change the final rounded result. The 48 most significant bits from the shifter output form the aligned mantissa of the second operand. The 23 less significant bits are used to calculate the sticky bit in the next stage. In fact, for shifts bigger than 47, we have to always take into consideration all bits shifted past the 48th bit because the discarded bits affect the sticky bit, which may be needed to determine if the result is rounded when the round bit (mantissa result's 25th bit) is one. We do this by fixing the maximum shift in 48 and always calculating the sticky bit with the 24 less significant bits of the shifter output.

On the other hand, when the multiplier result has a smaller exponent, we must allow a maximum shift of 24 bits like is represented in the right part of the figure 4. Shifting past this value is not worth doing because in that case the round bit is always zero and therefore the final result is simply truncated and not rounded. As we only use one shifter, we need a shifter capable of shifting a maximum value of 48 bits and thus the multiplier result can also be shifted up to 48 bits. This means the shifter must have a 96 bit output. The 48 most significant bits correspond to the mantissa and the 48 less significant bits to the discarded bits used to calculate the sticky bit. The shifter module is a base 4 logarithmic shifter composed by three 4:1 multiplexer stages which can shift the respective mantissa up to 63 bits. Also, the shifter is structured in pipeline, having a set of registers between the second and the third multiplexers stages.

In the fourth and fifth stage we add the mantissas. The mantissa adder is mapped on a DSP which spreads across two pipeline stages being that we only use one stage of registers in the DSP and the result is stored in a register outside the DSP. In parallel with the mantissa adding in the fourth stage, we determine the sticky bit.

The normalization step begins in the sixth stage. The block *adjust mantissa* makes the two's complement of the mantissa result in case the result from the mantissa adder is negative or right shift the mantissa by one in case the two operands have the same signal and the result has overflown. The dropped bit is accounted for the sticky bit.

When two operands are subtracted the result mantissa may be less than one and therefore need a left shift to be normalized. The next four stages involve the left shift normalization process. Across the seventh and eighth stages we have the leading zero detector block is responsible for determining the amount the mantissa must be left shifted. The left shifter is a base 4 logarithmic shifter composed by three 4:1 multiplexer stages which can shift the respective mantissa by 63 bits (since the input is 48 bits, the shift will never be bigger than 48 bits). The shifter is spread across the stage 9 and 10, having a set of registers between the second and the third multiplexers stages

The exponents have also to be updated during the normalization process. In stage 9 the exponent is updated accordingly to the normalization needed (increments 1 in the case of a right shift or is subtracted by the amount shifted in the case the mantissa was left shifted).

The tenth stage is also the beginning of the rounding process. The rounding process is defined by the round to nearest even method specified by IEEE. To determine if the result has to be rounded we consider the least significant bit of the mantissa (24th bit), the round bit (25th bit) and the sticky bit. The final sticky bit is the result of the OR of the previous sticky bit with the OR of all bits past the round bit from the normalized mantissa. The result is rounded if the round bit is 1 and the least significant bit or the sticky bit are 1.

We add the round bit in a DSP which spreads across the eleventh and twelfth stage. The first input of the adder is the exponent as the 8 significant bits and the mantissa (already without the hidden 1) 23 least significant bits. The second input is zero and the carry in is the round bit. In the case an overflow happens an exception is set to high. The definitive value of the exponent correspond to the 8 most significant bits of the adder result and the definitive value of the mantissa to the 23 least significant bits of the adder result. Note that the addition of the round bit with the mantissa can lead to an overflow and consequentially is needed the exponent and the mantissa must be normalized

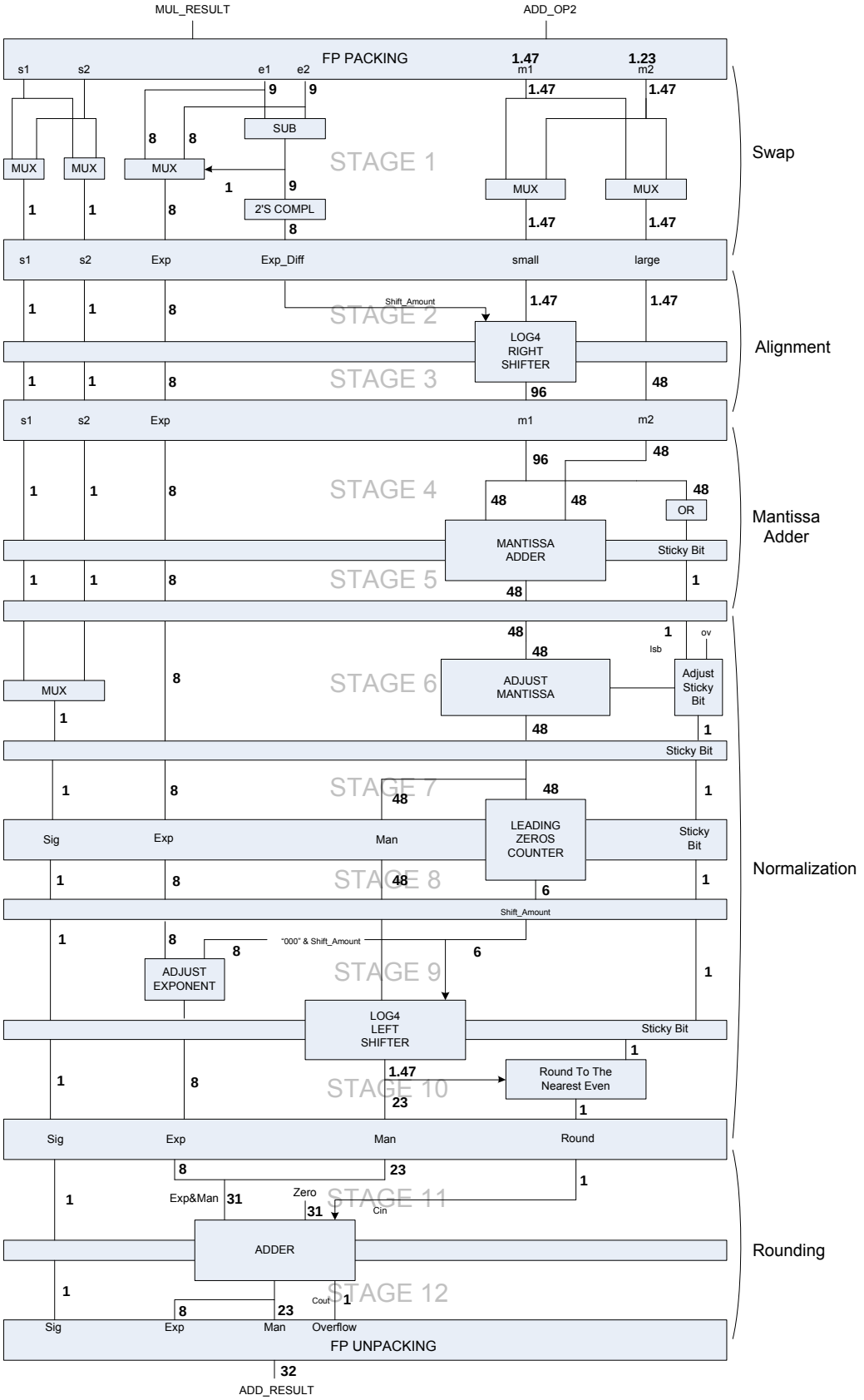


Figure 3. Floating-Point Adder Architecture.

as explained next. If an overflow occurs after adding the round bit with the mantissa, the exponent is automatically updated because it is the most significant part of the adder result. On the other hand the mantissa does not need to be

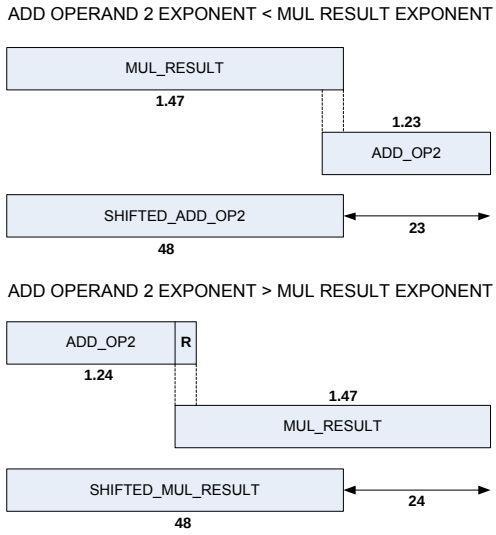


Figure 4. The alignment boundaries considering the two floating-point adder operands.

shifted. Since the mantissa entering the adder does not have the hidden bit, it means that an overflow only occurs when the mantissa is all ones, and this results in the mantissa being all zeros after the round bit addition.

Hereinafter we describe the modifications done in order to have low latency version of our floating-point adder. We designed this version with the objective of having a lighter unit which can be useful in multiprocessor systems where the FMA unit is not the performance bottleneck. Moreover, we implement the low latency unit only with LUTs which enable the possibility of the adder being mapped in previous FPGAs generations where the DSPs could not implement a 48-bit adder. The first simplification was to fuse the second and third pipeline stages, that is, the alignment is done in one clock cycle. The fifth and sixth stages were fused too and the leading zeros counter hardware was moved from the previous seventh stage to the now fifth stage. Finally the rounding adder is part of only one pipeline stage in the low latency version.

6. Results

The unit was tested with valid random generated data which follows a normal distribution. Valid data means that we apply some boundaries in the test data to better appreciate the accuracy results. Situations as overflows/underflows which rise exceptions are avoided. Also, we limit the multiplication result and the second addition operand exponents to avoid having a big difference between the two exponents because we ended adding zero to another operand except for the case in which the multiplication result exponent is bigger than the exponent of the other operand and the sticky bit can contribute to the decision of rounding the result.

We generated 10000 samples for each one of the three operands involved in the multiply-add operation which are fed to our hardware unit during the simulation. A high level C implementation was created to compute the result of each multiply-add operation emulating a non-fused single-precision unit and a double-precision unit. The results from the double-precision unit serve as a standard to which our unit and the non-fused single-precision unit compare to. Although, before comparing, the results from the double-precision unit are rounded following the IEEE round to nearest even rules. The process of calculating the 10000 samples was repeated 20 times to confirm the consistence of the results.

The results given by our FMA unit were compared to the non-fused single-precision and double-precision results. We verified that our hardware always achieves the maximum precision possible, whereas that does not always happen for the non-fused single-precision. In fact, over the 20 sets of tests made, our hardware presents an average of 1127 results in 10000 with more accuracy than the non-fused single-precision multiply-add unit. The single-precision error is in average 0.0000001192092896, approximately 2^{-23} in base 2.

Next, we discuss the implementation results of our FMA unit against a non-fused multiply-add unit composed by a state-of-the-art floating-point multiplier and floating-point adder from Xilinx[5]. Presented in table 1 are the post place and route results obtained with the Xilinx ISE tool version 14.5 targeting the device Virtex-7 (XC7V585T-1).

Table 1. Implementation results for our FMA unit and a non-fused multiply-add unit based on the Xilinx floating-foint units

XC7V585T-1				
	Ours (Low Lat.)	Xilinx (Low Lat.)	Ours (High Lat.)	Xilinx (High Lat.)
FFs	773	820	990	683
LUTs	845	1018	754	751
DSPs	2	2	4	4
Freq(MHz)	285	350	361	332
Latency	14	14	18	18

In table 1 we present results for two different versions of our FMA unit and the Xilinx based non-fused equivalents. The low latency versions uses only LUTs to implement the floating-point adder as this can possibly lead to a lower routing delay overhead and consequently to achieving higher frequencies. When comparing the two low latency units we can see that our version uses less resources than the non-fused equivalent unit. This advantage, is however, balanced by the fact that our unit has a lower maximum frequency. The frequency could be improved by putting more effort in parallelizing the hardware in some stages in order to achieve better frequencies with the same latency, even if it means spending more resources.

In relation to the high latency units, we verify that our FMA resource usage is very close to the non-fused unit resource usage, with the exception of the number of registers used. This is difficult to avoid because with more pipeline stages and having bigger buses (due the increase width of

the signals necessary to maintain more precision in the calculations), we end up spending more registers. Nevertheless, our high latency unit has the advantage of not only enabling higher accuracy in the calculations, but also can work at a higher frequency.

7. Conclusions and Future Work

We have proposed a hardware design for an FMA unit in complete compliance with the IEEE guidelines for the fused multiply-add operation (section 2.1.28 in [4]).

Comparing the resource and performance results of our unit with a state-of-the art non-fused multiply-add design, we verified that we can achieve significant better numerical accuracy results with only mild degradation of the performance or area usage. Also, the low latency design can be useful when implementing multiprocessor systems in target devices where the DSPs are the critical resource. More so, if the system is synchronous and the FMA is not the limiting factor in the system maximum frequency achieved.

In the future, we pretend to integrate our FMA unit in an application driven processor and test the impact that the increased accuracy of a fused architecture can have in continuous multiply-add operations like the dot product (in which the error can accumulate).

Finally, we pretend to extent this study to double-precision operations and see how well the hardware can scale.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2013 and PTDC/EEA-ELC/122098/2010.

References

[1] E. Hokenek R.K. Montoye and S.L. Runyon. Design of the ibm risc system/6000 floating-point execution unit. *IBM Journal of Research & Development*, 34:59–70, 1990.

[2] R. Montoye E. Hokenek and P. Cook. Second-generation risc floating point with multiply-add fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, 1990.

[3] E. Quinnell. *Floating-Point Fused Multiply-Add Architectures*. PhD thesis, The University of Texas at Austin, May 2007.

[4] IEEEStandard754-2008. Ieee standard for floating-point arithmetic. Technical report, IEEE Computer Society, 2008.

[5] Xilinx. Xilinx logicore ip floating-point operator v6.1 product specification. Technical report, Xilinx, 2012.

[6] H. Neto W. Maltez, A. R. Silva and M. Véstias. Analysis of matrix multiplication on high density virtex-7 fpga. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sep 2013.

[7] E. S. Jr E. Quinnell and C. Lemonds. *Three-path fused multiply-adder circuit*, 2011.

[8] V. Erraguntla S. Jain and S. R. Vangal. A 90mw/gflop 3.4ghz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm. In *2010 23rd Int. Conf. on VLSI Design*, pages 252–257, Jan 2010.

[9] J. Brooks and C. Olson. *Processor Pipeline which Implements Fused and Unfused Multiply-Add Instructions*, 2012.

[10] H. H. Saleh and E. E. Swartzlander. A floating-point fused dot-product unit. In *in 2008 IEEE Int. Conf. on Computer Design*, pages 427–431. IEEE, Oct 2008.

[11] E. Swartzlander and H. Saleh. Fft implementation with fused floating-point operations. *Computers, IEEE Transactions on*, 61(2):284–288, 2012.

[12] B. Pasca S. Banescu, F. de Dinechin and R. Tudoran. Multipliers for floating-point double precision and beyond on fpgas. *ACM SIGARCH Computer Architecture News*, 38(4):73–79, 2010.

[13] F. D. Dinechin and B. Pasca. An fpga-specific approach to floating-point accumulation and sum-of-products. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 33–40, Dec 2008.

[14] B. Catanzaro and B. Nelson. Higher radix floating-point representations for fpga-based arithmetic. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM05)*, pages 161–170, Apr 2005.

[15] K. S. Hemmert and K. D. Underwood. Fast, efficient floating-point adders and multipliers for fpgas. *ACM Transactions on Reconfigurable Technology and Systems*, 3(3):1–30, 2010.

[16] J. Huthmann B. Liebig and A. Koch. Architecture exploration of high-performance floating-point fused multiply-add units and their automatic use in high-level synthesis. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 134–143, May 2013.

[17] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[18] J. Detrey and F. Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *The Journal of VLSI Signal Processing Systems for Signal*, 49(1):161–175, 2007.

 ISBN: 978-989-98875-1-0

Linguagens e Otimização

 ISBN: 978-989-98875-1-0

Using SystemC to Model and Simulate a Many-Core Architecture for LU Decomposition

Ana Rita Silva*, Wilson José*, Horácio Neto†, Mário Véstias‡
*INESC-ID, †INESC-ID/IST/UTL, ‡INESC-ID/ISEL/IPL
anaritasilva@inesc-id.pt, wilsonmaltez@inesc-id.pt, hcn@inesc-id.pt, mvestias@deetc.isel.pt

Abstract

Designing efficient many-core architectures with hundreds of cores is a very challenging task due to the complexity and size of the design space. System-level simulation can help designers exploring the design space of many-core architectures. In this paper, we use SystemC to model a many-core architecture and run a parallel LU decomposition algorithm, an important linear algebra kernel that is widely used in both scientific and engineering applications. The design is parameterized permitting to adapt the model to various hardware constraints. The simulation of the model outputs the number of communications and the number of clock cycles required to complete the algorithm. Given the obtained simulation times, SystemC can be used to efficiently model many-core architectures. Also, the results show the scalability of the architecture and are according to the theoretical formulations.

1. Introduction

Matrix factorization is a key computational kernel in linear algebra. The Lower/Upper (LU) decomposition algorithm is the most used matrix factorization method. It is widely used in scientific and engineering applications to solve dense linear equations and is included in many popular linear algebra libraries such LAPACK [1].

LU decomposition can be implemented on FPGAs to accelerate algorithms in scientific computing, where latency performance of the hardware accelerator is important.

In this work we map a parallel LU decomposition algorithm on a system with p processing elements based on the column-oriented LU decomposition [2] and describe the implementation of the system model using SystemC. The design is parameterized so that it can easily adapt to the hardware constraints; the parameters include the number of cores and the local storage size.

Using SystemC we can model systems above RTL, including systems that might be implemented in software, hardware, or some combination of the two [3].

The higher level of abstraction gives the design team a fundamental understanding, early in the design process, of the behaviour of the entire system and enables better system trade offs, better and earlier verification, and overall productivity gains through reuse of early system models as executable specifications [4].

The rest of the paper is organized as follows. The next section describes previous proposals of LU decompositions architectures and Section 3 does an overview of LU decomposition. Section 4 describes the many-core architecture to be modeled, the LU decomposition algorithm to be simulated and a theoretical model for the number of communication cycles and operations. Section 5 describes the SystemC model of the architecture. Experimental results are discussed in Section 6. Finally, section 7 concludes the paper.

2. Related Work

There have been some works on FPGA-based matrix factorization. Wang et al. [5] proposed a multiprocessor system on an FPGA device, and used it for LU decomposition of sparse block-diagonal bordered matrices. Each processor on the FPGA is attached to a floating-point adder/-subtractor, a floating point multiplier and a floating-point divider.

Choi et al. [6] proposed a linear array architecture for fixed-point LU. The array consists of n processing elements (PEs) and each PE contains a multiplier, an adder/subtractor and a reciprocator.

In [7], a circular array architecture was proposed for floating-point LU decomposition. It uses n PEs and only the first PE, PE_0, contains a divider. PE_j $(1 \leq j \leq n-1)$ contains one MAC (Multiplier and ACumulator) and a storage of size n - j. However, due to the design complexity of the floating-point units, an FPGA device cannot contain n PEs when n is larger than several tens.

The authors in [8] give a detailed description of block LU decomposition algorithm and proposes an architecture to run it. The matrix is partitioned into $b \times b$ blocks, where b is determined by the number of configurable slices.

3. LU Decomposition

In LU decomposition a $n \times n$ matrix, A (a_{xy}), is decomposed into a lower triangular matrix L (l_{xy}) with 0s above the diagonal and 1s on the diagonal, and an upper triangular matrix U (u_{xy}) with 0s below the diagonal, both of size $n \times n$, where x is the row index and y is the column index (see figure 1).

Lower/Upper triangular decomposition is performed by a sequence of Gaussian eliminations to form $A = LU$. In

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

Figure 1. Matrices A, L and U.

this work, we assume that matrix A is a non-singular matrix and no pivoting is needed.

In [3], a sequential algorithm for LU decomposition is discussed. It consists of three main steps:

- Step 1: The column vector $a_{s1}(2 \leq s \leq n)$ is multiplied by the reciprocal of a_{11}. The resulting column vector is l_{s1};
- Step 2: a_{1j} is multiplied by the column vector l_{i1}. The product is subtracted from the submatrix $a_{ij}(2 \leq i, j \leq n)$;
- Step 3: Steps 1 and 2 are recursively applied to the new submatrix generated in Step 2. During the kth iteration, l_{ik} and $u_{ij}(k+1 \leq i, j \leq n)$ are generated. When $k = n-1$, the decomposition is complete.

A pseudo code for the decomposition is given in Listing 1.

Listing 1. Column Oriented LU Decomposition

```
for k = 1 to n-1
  for s = k+1 to n
    $l_{sk} = \frac{a_{sk}}{a_{kk}}$
  for j = k+1 to n
    for i = k+1 to n
      $a_{ij} = a_{ij} - l_{ik} \times
          a_{kj}$
```

4. Many-Core Architecure and Algorithm Analysis

In this section, we present the algorithm proposed to parallelize the LU Decomposition problem for a system with p cores organized as a 2-dimensional array.

The parallel architecture is organized as a 2D mesh of processing elements (cores). Each core consists mainly of a floating-point divider, a floating-point multiplier and adder/subtractor and a dual-port memory. Access to the external memory is controlled with a direct memory access

MEM	B_0	B_1	B_2	B_3	B_4	B_5
P1	L_1	B'_1	B'_2	B'_3	B'_4	B'_5
P2		L_2	B''_2	B''_3	B''_4	B''_5
MEM			B''_2	B''_3	B''_4	B''_5
P1			L_3	B'''_3	B'''_4	B'''_5
P2				L_4	B''''_4	B''''_5
MEM					B''''_4	B''''_5
P1					L_5	B'''''_5

Figure 2. LU Decomposition considering $A = 6 \times 6$ and 2 processing elements.

(DMA) module that can deliver burst transactions with one transfer per cycle. There is only a single interface to external memory, so writing to and reading from external cannot be done in parallel.

To facilitate the presentation of the algorithm, we consider a LU Decomposition of a matrix A with size 6×6 and a model with 2 processing elements.

Processing element P1 is responsible for calculating the first submatrix generated, A' (k=1). P1 receives the first column of matrix A, $B_0(a_{11}, a_{21}, a_{31}, a_{41}, a_{51}, a_{61})$, and determines the values of the first column of matrix L $(l_{21}, l_{31}, l_{41}, l_{51}, l_{61})$), according to the algorithm in Listing 1.

The operations are initiated at the moment the first value is available and the values l are stored in a local memory and sent to the external memory.

Next, P1 receives the second column, $B_1(a_{12}, a_{22}, a_{32}, a_{42}, a_{52}, a_{62})$, and calculates the new column $B'1(a'_{22}, a'_{32}, a'_{42}, a'_{52}, a'_{62})$ and so on, until we obtain the submatrix A´. The first value of each column (a´22, a´23, a´24, a´25, a´26) is stored in the external memory.

As processing element P1 determines the columns of submatrix $A'(B'_1, B'_2, B'_3, B'_4, B'_5)$, the results are sent to the processing element P2.

Processing element P2 is responsible for determining submatrix $A''(k = 2)$. Similarly to processing element P1, P2 receives the column $B'1$, obtained by P1, and determines the second column of matrix L $(l_{32}, l_{42}, l_{52}, l_{62})$, stores the values in a local memory and send it to the external memory, followed by the computation of $B2''(a''_{33}, a''_{43}, a''_{53}, a''_{63})$ from B2' and so forth.

Being P2 the last processing element in this example, it sends all the results to the external memory. Then, processing element P1 receives the values of matrix A", obtained by processing element P2 and determines matrix A"' and so forth, according to figure 2.

As mentioned above, each processing element stores the values of the matrix L in a local memory to be used in the

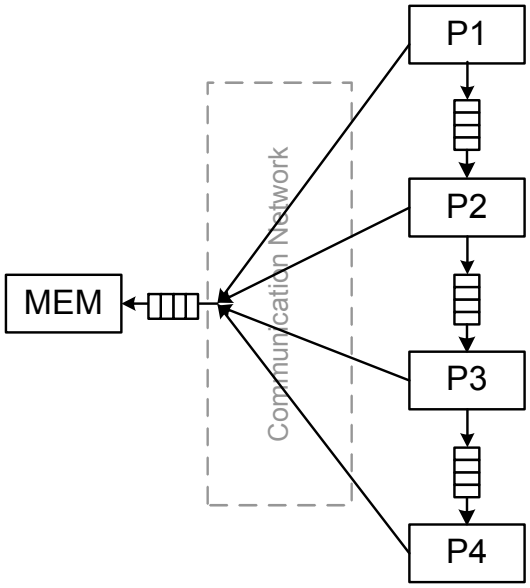


Figure 3. SystemC model of the many-core architecture with four processing elements.

calculations of new columns. Thus, each processing element P_x needs a memory location with a size equal to a column-x.

The total number of communications is given by equation 1, where the size of matrix A is $n \times n$ and p the number of processing elements.

$$\sum_{k=0}^{\frac{n}{p}-1}(n-kp)^2+\sum_{k=1}^{n-1}k+\sum_{i=0}^{\frac{n}{p}-1}(\sum_{k=1}^{p-1}(n-pi+k))+\sum_{k=1}^{\frac{n}{p}-1}(n-kp)^2 \quad (1)$$

The first parameter of the sum is the number of data sent from memory to the first processing element, P1. The other parameters correspond to the sending of data from processing elements to memory: the second corresponds to the values of matrix L, the third to the data matrix B calculated by all processing elements except for the last one and the fourth parameter correspond to the values of matrix B calculated by the last processing element.

The total number of operations is given by equation 2.

$$\sum_{k=1}^{n-1}k+2\times\sum_{k=1}^{n-1}k^2 \quad (2)$$

The first parameter is the number of divisions needed to calculate the matrix L and the second is the number of multiplications required to compute the matrix U, which is equal to the number of subtractions.

5. SystemC Model of the Architecture

This section describes the implementation of the system model using SystemC. For ease of explanation, we describe the model considering only four processing elements (see Figure 3)

The system consists of a module MEM representing the DMA and p modules P_x representing the cores. The communication between modules is performed with FIFOs.

The details of communication among modules are separated from the details of the implementation of the functional units.

There is a module Arbiter used to arbitrate competing communication requests and consists of one method process (SC_METHOD). Outstanding requests are passed to the arbiter as a vector of structures. On each cycle, module Arbiter receives all the requests and selects several for execution based on their arbitration policy. Requests are chosen according to the chronological order and attended if FIFOs are free to write.

When a module wants to write to a certain FIFO, it is created and filled a new request using the function *fill_req*. After writing all data, the request is released using the function *unlock_req* (see code in Listing 2).

Listing 2. Functions *fill_req* and *unlock_req*

```
void fill_req(int *i_req, req *v_req,
    int n, int req_out, int *x){
        v_req[*i_req].n_in=n;
        v_req[*i_req].out=req_out;
        for (int j=0; j<n; j++){
                v_req[*i_req].in[j]=x[j
                    ];
        }
        *i_req=*i_req+1;
}

void unlock_req(int n, int *x, int *
    lock_list){
        for (int j=0; j<n; j++){
                lock_list[x[j]] = 0;
        }
}
```

FIFOs are modeling using the code in Listing 3

Listing 3. systemC modelling of read and write FIFO functions

```
void write_fifo(int *f, int *full,
    double *FIFO, sc_event *ev_in,
    sc_event *ev_out, double value ,int
     size){
        if(*full==size) wait(*ev_out);
        FIFO[*f]=value;
        *f=*f+1;
        *full=*full+1;
        ev_in->notify();
        if (*f==size) *f=0;
}

double read_fifo(int *full, int *i1,
    double *FIFO, sc_event *ev_in,
    sc_event *ev_out, int size){
        double x;
        if(*full==0) wait(*ev_in);
        x=FIFO[*i1];
```

```
        *i1=*i1+1;
        *full=*full-1;
        ev_out->notify();
        if (*i1==size) *i1=0;
        return x;
}
```

In regard to modules P_x, they are basically the model of the 2D mesh array of cores. Each module P_x consists of one thread process, which is responsible for the operations and communications.

6. Simulation and Results

To implement this model with a given number of cores, we developed a program in C to generate the respective code in SystemC. Using this program, we can implement systems with several cores to simulate the LU decomposition for square matrices of different sizes.

The number of cycles is obtained by considering that each operation has a throughput of one result per cycle and the number of cycles need to write data is one cycle and FIFOs with sufficient size to obtain the fewest number of cycles.

Tables 1 to 3 show the results obtained in regard to the number of communications, the total number of cycles and the simulation time, considering square matrices of different sizes and different number of cores.

Table 1. Number of communications, execution cycles and simulation times of LU decomposition with 16 cores

Matrix Size	Communications	Exec. Cycles	Simul. Time
32	2.512	2.541	1s
64	15.200	15.234	7s
128	103.872	104.017	34s
256	763.776	765.216	3 m
512	5.848.832	5.853.972	18 m

Table 2. Number of communications, execution cycles and simulation times of LU decomposition with 32 cores

Matrix Size	Communications	Exec. Cycles	Simul. Time
64	10.144	10.177	2s
128	61.120	61.164	35s
256	416.640	416.824	2 m
512	3.059.456	3.061.743	18 m

Observing the results, we conclude that the number of communications is consistent with equation 1. Considering the example of matrices with size 32×32 and 16 processing elements the total number of communications if given by equation 3.

Table 3. Number of communications, execution cycles and simulation times of LU decomposition with 64 cores

Matrix Size	Communications	Exec. Cycles	Simul. Time
128	40.768	40.820	34s
256	245.120	245.171	3 m
512	1.668.864	1.669.070	18 m

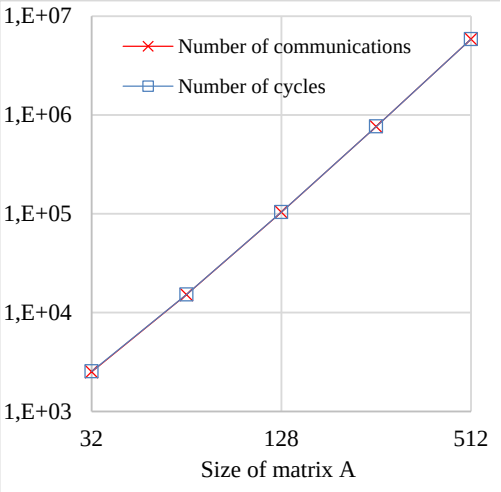


Figure 4. Model with 16 cores.

$$\sum_{k=0}^{\frac{n}{p}-1}(n-kp)^2+\sum_{k=1}^{n-1}k+\sum_{i=0}^{\frac{n}{p}-1}\left(\sum_{k=1}^{p-1}(n-pi+k)\right)+\sum_{k=1}^{\frac{n}{p}-1}(n-kp)^2=1280+496+480+256=2512 \quad (3)$$

Figures 4 and 5 compares the number of communications and the total number of cycles with the size of matrix A, considering 16 and 64 cores, respectively.

According to both figures, we verified that the number of communications and the number of cycles increases exponentially with the increasing size of matrix A.

The number of communications is practically equal to the number of cycles, concluding that the number of computations can be totally overlapped with the communications.

Looking at tables 1, 2 and 3, we found that the number of communications and the total number of cycles decreases with the increase of the number of processing elements (see figure 6).

As can be seen, the larger the size of the matrix, the more accentuated is the variation in the number of communications.

Table 4 shows the total number of operations needed to perform the algorithm.

Observing the results, we can conclude that the number of computations is consistent with equation 2. Also, the increase in the number of operations is according to the complexity of the algorithm (n^3).

 ISBN: 978-989-98875-1-0

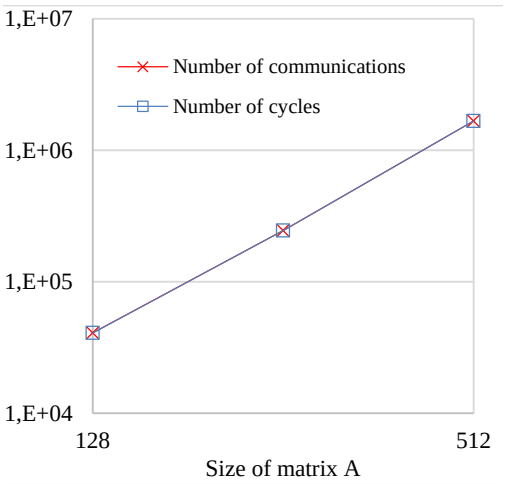


Figure 5. Model with 64 cores.

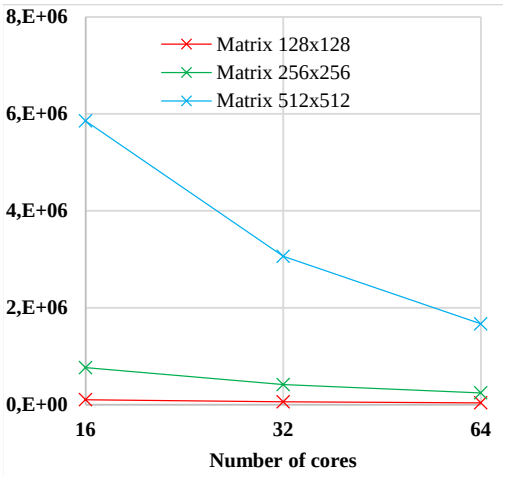


Figure 6. Number of execution cycles for 16, 32 and 64 cores for different matrix sizes.

From the results we can determine the performance efficiency (ratio between sustained performance and peak performance) of the architecture. For example, considering a matrix of size 512×512, we achieve efficiencies up to 46%.

7. Conclusions and Future Work

In this paper we have described a many-core architecture using SystemC to perform LU decomposition. We started with a brief overview of LU decomposition, describing in particular the Column-Oriented method.

The algorithm to execute on a 2-dimensional multiprocessor array was presented and analyzed theoretically in regard to the number of communications and computations. We simulated the model to evaluate number of communications and total number of clock cycles required for the complete algorithm execution.

The results show the scalability of the architecture and are according to the theoretical formulations.

Given the obtained simulation times, SystemC is a good candidate to efficiently model many-core architectures. Therefore, the modeling process with SystemC is being applied to other parallel algorithms.

Table 4. Number of Computations for different matrix sizes

Matrix Size	# Operations
32	21.328
64	172.704
128	1.389.888
256	11.152.000
512	89.347.328

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2013 and PTDC/EEA-ELC/122098/2010.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK user's guide third edition", August 1999.

[2] Gene Golub, James M. Ortega, "Scientific Computing: An Introduction with Parallel Computing ", Academic Press Professional, Inc. San Diego, CA, USA ©1993.

[3] D. Black and J. Donovan, "SystemC: from the Ground Up", Kluwer Academic Publishers, 2004.

[4] T. Grötker, S. Liao, G. Martin and S. Swan, "System Design with SystemC", Kluwer Academic Publishers, 2002.

[5] X.Wang and S. G. Ziavras, "Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Operations," in Proc. of FPT 2003, December 2003.

[6] S. Choi and V. K. Prasanna, "Time and Energy Efficient Matrix Factorization using FPGAs," in Proc. of FPL 2003, September 2003.

[7] G. Govindu, S. Choi, and V. K. Prasanna, "Efficient Floating-Point Based Block LU Decomposition on FPGAs," in Proceedings of ERSA 2004, June 2004.

[8] V. Daga, G. Govindu, S. Gangadharpalli, V. Sridhar, and V. K. Prasanna, "Efficient Floating-point based Block LU Decomposition on FPGAs," in Proc. of ERSA 2004, June 2004.

[9] Guiming Wu; Yong Dou; Junqing Sun; Peterson, G.D., "A High Performance and Memory Efficient LU Decomposer on FPGAs," Computers, IEEE Transactions on , vol.61, no.3, pp.366,378, March 2012

[10] http://www.accellera.org/home/

[11] http://www.doulos.com/knowhow/systemc/

 ISBN: 978-989-98875-1-0

Comparando a geração de hardware a partir de uma Linguagem de Domínio Específico à uma abordagem de Síntese de Alto Nível a partir de C

Cristiano B. de Oliveira[†], João M. P. Cardoso[‡], Eduardo Marques[†]
[†]*Instituto de Ciências Matemáticas e de Computação (ICMC),Universidade de São Paulo, Brasil*
[‡]*Faculdade de Engenharia da Universidade do Porto (FEUP), Universidade do Porto, Portugal*
{cbacelar, emarques}@icmc.usp.br, jmpc@acm.org *

Resumo

Field-Programmable Gate Arrays (FPGAs) permitem o desenvolvimento de aceleradores de hardware enquanto mantém a programabilidade. Contudo, as vantagens da utilização de FPGAs ainda dependem da experiência do desenvolvedor e do seu conhecimento sobre linguagens de descrição de hardware (HDLs – Hardware Description Languages). Apesar de ferramentas de Síntese de Alto Nível (HLS – High-Level Synthesis) terem sido desenvolvidas a fim de minimizar este problema, elas, geralmente, apresentam soluções consideradas ineficientes, quando comparadas às obtidas por um desenvolvedor especializado. Linguagens de Domínio Específico (DSLs – Domain-specific languages) podem ser soluções alternativas para a programação de FPGAs. Elas podem prover um nível de abstração mais alto do que as HDLs enquanto permitem ao programador ajustar as implementações sempre que as ferramentas de HLS não produzirem designs eficientes. Neste artigo comparamos uma DSL, chamada LALP (Language for Aggressive Loop Pipelining), com uma típica abordagem de HLS e mostramos os resultados experimentais usando ambas as abordagens. Os resultados mostram um maior desempenho obtido com uso de LALP do que o atingido pela ferramenta de HLS.

1. Introdução

A exploração de paralelismo é uma abordagem utilizada para melhorar a velocidade em implementações de *hardware*. A grande quantidade de recursos de *hardware* reconfigurável providos pelos FPGAs (*Field-Programmable Gate Arrays*) permite a implementação de esquemas eficientes de paralelismo, a nível de operações e *pipelining*, e a nível de tarefas. Contudo, é necessário um elevado grau de especialização a fim de tirar proveito deste paralelismo e dos recursos dos FPGAs. Abordagens de alto nível têm buscado o fornecimento de ferramentas para a compilação automática de programas diretamente para *hardware* em FPGAs, tentando diminuir os requisitos de programação necessários aos desenvolvedores de sistemas embebidos.

Linguagens de domínio específico, como LALP [1] (*Language for Aggressive Loop Pipelining*), podem ser soluções alternativas para a programação de FPGAs. Elas fornecem um maior nível de abstração do que as HDLs e, além disso, permitem ao programador controlar certos aspectos da implementação, como o paralelismo, que nem sempre é bem tratado por técnicas de compilação automáticas. Em [1] são mostrados resultados anteriores obtidos usando LALP. Nesses casos, as implementações LALP obtiverem um *speedup* médio de 5.60× sobre C2Verilog [2] para um conjunto de 10 *kernels*, e 1.14× sobre ROCCC [3] para um conjunto de 3 *kernels*.

Neste artigo são mostrados e comparados os resultados de duas abordagens diferentes para geração de aceleradores de *hardware*: o uso de LALP, uma linguagem de domínio específico, e Vivado HLS [4], uma ferramenta de síntese de alto nível a partir de C. Nossa intenção principal é fornecer uma comparação e perceber as possíveis diferenças e vantagens de ambas as abordagens. Os resultados, obtidos para um conjunto de 10 *benchmarks*, evidenciam que o uso de LALP apresenta melhorias significativas.

Este artigo está organizado da seguinte forma: na Seção 2 é apresentada uma breve descrição do LALP e algumas de suas características. Os resultados experimentais são apresentados na Seção 3. As Seções 4 e 5 mostram uma visão geral de outras linguagens relacionadas ao LALP e as conclusões, respectivamente.

2. LALP – Language for Aggressive Loop Pipelining

LALP [1] é uma linguagem de propósito específico, com uma sintaxe semelhante a C, focada no mapeamento de estruturas de repetição (*loops*) para *hardware* reconfigurável. LALP permite a exploração do paralelismo presente nos *loops* por meio de técnicas de *loop pipelining*, sendo apropriada para o desenvolvimento de aplicações altamente paralelas. LALP fornece um alto nível de abstração enquanto permite que o programador explore o paralelismo controlando a execução de cada estágio do *pipeline*. Este nível de controle pode resultar em aceleradores de *hardware* altamente eficientes.

O compilador LALP gera código VHDL a partir de código LALP, de acordo com o fluxo de compilação mostrado na Figura 1. O *framework* usa contadores para controlar o número de ciclos de *clock* atrasados ou executados para cada operação. Uma vez que em LALP todas as instru-

*Os autores agradecem à FAPESP (Fundação de Apoio à Pesquisa do Estado de São Paulo) pelo suporte financeiro recebido.

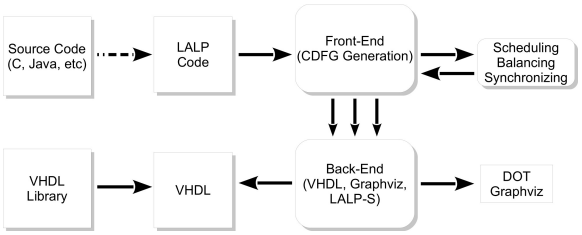


Figura 1. Fluxo de compilação LALP

ções são consideradas concorrentes, a sequência de execução de cada operação é determinada pela presença de dependências, automaticamente tratadas pelo compilador ou de acordo com os valores de *delays* definidos em código LALP com o uso do operador '@'. Este operador é usado pelo programador para permitir o controle manual do escalonamento das operações e para definir o número de ciclos necessários para habilitar as operações. Esta característica permite um alto controle do fluxo do programa e o escalonamento das operações de diferentes iterações de um *loop* em uma execução em *pipelining*.

A Figura 2 mostra um exemplo simples de código LALP. Neste exemplo, os endereços dos vetores (x e y) são definidos pelo valor do contador i (linhas 4 e 5). Os cálculos de a e b (linhas 6 e 7) ocorrem concorrentemente e, devido à dependência de dados, o cálculo de c (linha 8) é automaticamente escalonado para executar no próximo ciclo de *clock*. Além disso, a multiplicação é definida para ser executada após dois ciclos, enquanto o incremento do contador é atrasado por quatro ciclos de *clock* (linha 3).

```
1  foo(out int result, out bit done, in bit init){
2    { ... } //Variables declarations
3    counter (i=0; i<3; i++@4);
4    x.address = i;
5    y.address = i;
6    a = (x + y);
7    b = (x - y);
8    c = (a *@2 b);
9    acc += c when i.step@4;
10   result = acc;
11   done = i.done;
12 }
```

Figura 2. Exemplo de código LALP

3. Resultados Experimentais

LALP foi avaliado usando um conjunto de dez *benchmarks*, descritos na Tabela 1, os quais foram codificados em LALP a partir do código original equivalente em C. Estão incluídos os resultados de duas versões diferentes para o Vivado HLS: a) utilizando diretivas de *pipelining* (*target* II=1) para o *loop* mais interno e b) sem diretivas de *pipelining*. Ambas as versões utilizam a configuração padrão para a família Virtex 7 (*device* XC7VX330T), com meta de período de 1 *ns*. Alguns *benchmarks* possuem uma ou mais implementações LALP alternativas (veja a Tabela 1).

As versões LALP seguem o máximo possível o algoritmo e as estruturas computacionais dos códigos originais em C. Isto é importante para tornar a comparação justa.

Benchmarks	**Descrição**	**Data Size**	**Apenas LALP**
adpcm_coder	Codificador de áudio	1024	Não
adpcm_decoder	Decodificador de áudio	1024	Não
autocor	Autocorrelação entre 2 vetores	160	Não
dotprod	Produto escalar de 2 vetores	2048	Não
dotprod_pipe	*dotprod* com multiplicador com *pipeline*	2048	Sim
fdct	Transformada Rápida Discreta do Cosseno	640	Não
fdct_opt	*fdct* com diferente escalonamento de operações	640	Sim
fibonacci	Cálculo da sequência de Fibonacci	32	Não
fibonacci_alt	*fibonacci* com reuso de dados	32	Sim
max	Valor máximo de um vetor	2048	Não
sobel	Operador de filtro de Sobel	100	Não
sobel_alt	*sobel* com diferente escalonamento de operações	100	Sim
sobel_nomul	*sobel* com diferente escalonamento de operações	100	Sim
sobel_reg	*sobel* com reuso de dados	100	Sim
vecsum	Soma de vetores	2048	Não
matmul	Multiplicação de matrizes	25	Não

Tabela 1. Descrição dos *benchmarks* usados

Dentre as transformações de *loop* que podem ser aplicadas pelo Vivado HLS, apenas a de *loop pipelining* foi considerada, pois, caso contrário, seria preciso aplicar as demais transformações também em LALP para manter a comparação justa. Considere que, apesar de um menor esforço para a implementação de circuitos do que com uso de HDLs, LALP requer bem mais esforços do que utilizando uma ferramenta de HLS, especialmente porque o compilador LALP não é tão sofisticado em termos de escalonamento, sendo que, na maioria dos casos, o programador deve escalonar parcialmente os *loops*. A comparação entre essas duas abordagens, trata-se, de facto, de uma comparação entre uma abordagem intrinsecamente baseada em contadores, *delays* enfileirados e controle distribuído (LALP), e outra abordagem orientada à FSMs (*Finite State Machines*), com uma unidade de controle centralizado, normalmente obtida a partir do grafo de fluxo de controle referente ao exemplo que estiver sendo compilado (HLS).

A Figura 3 mostra os *speedups* de tempo de execução de LALP sobre a ferramenta de HLS usada. Em todos os casos LALP apresenta um melhor resultado que as versões de Vivado HLS sem *pipelining*, com média geométrica de 8.5×. LALP também mostra melhores tempos de execução na maioria dos casos em comparação ao Vivado HLS com *pipelining*, à exceção dos casos onde LALP apresenta maiores valores de latência (Figura 4(b)), como *fdct* e *sobel*. Apesar destes casos, os resultados mostram uma média geométrica de *speedup* de 2.84× sobre Vivado HLS com *pipelining*, mesmo quando um menor número recursos é usado em LALP do que nas soluções geradas pelo Vivado HLS (ver os recursos de *hardware* usados nas Figuras 4(c), 4(d) e 4(e)). Os resultados das versões alternativas também mostram que é possível um maior desempenho em LALP ao serem utilizados diferentes esquemas de escalonamento das operações (*fdct* e *sobel*), reuso de dados (*fibonacci*) e operadores com *pipeline* (*dotprod*).

A Figura 4(a) apresenta as frequência máximas atingidas por ambas as abordagens. Na média, LALP atingiu valores de frequência máxima similares aos do Vivado HLS. Contudo, em alguns casos LALP atingiu frequências significativamente menores (*autocor*, *fdct* e *matmul*). Isto ocorre devido à falta de uso de operações em *pipelining* nestes casos. O único caso onde é usado um multiplicador implementado em *pipeline* é o *dotprod_pipe*.

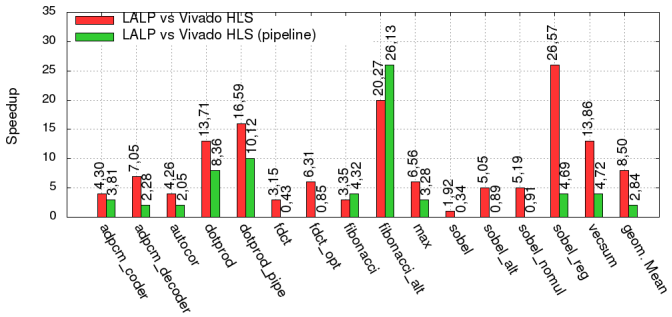


Figura 3. *Speedups* obtidos para as implementações LALP sobre as soluções geradas pelo Vivado HLS

As baixas latências produzidas pelo LALP para o *fdct* e *sobel* (Figura 4(b)) devem-se à exploração limitada das operações em *loops* nos exemplos. Para estes dois exemplos, o compilador não é capaz de escalonar completa e automaticamente as operações. A abordagem usando LALP foi feita parcialmente de forma manual (sem muito esforço). Perceba, contudo, que com um esforço manual adicional seria possível atingir um melhor desempenho.

As diferenças no número de Flip-Flops (FFs) variam conforme o exemplo (Figura 4(c)). Na maioria dos casos, LALP requer menos FFs que o Vivado HLS. Entretanto, existem casos onde LALP usou muito menos FFs (*dotprod*), enquanto em outros LALP usou um maior número de FFs (*matmult*). O uso de contadores, *shift-registers* para imposição dos *delays*, e de componentes sem *pipeline* pelo LALP justifica estes resultados. Em termos de LUTs, as soluções LALP usaram menos LUTs na maioria dos exemplos (Figura 4(d)), embora LALP tenha usado, significativamente, mais LUTs em poucos casos (*autocor* e *matmul*).

Em termos de DSPs (Figura 4(e)), LALP usou mais recursos, já que não há compartilhamento de recursos em LALP, ao contrário do que ocorre no Vivado HLS. Em termos de BRAMs, os resultados são similares, e as pequenas variações são referentes à promoção de *arrays* para ROMs

4. Trabalho Relacionado

Em adição à geração de *hardware* a partir de código C, tem havido várias abordagens para o desenvolvimento de novas linguagens para programação de *hardware* com nível de abstração maior do que típicas HDLs, como VHDL e Verilog. Aqui serão descritas aquelas que acreditamos serem mais relacionadas à linguagem LALP.

Handel-C [5] é uma linguagem de programação imperativa que expõe alguns componentes dos FPGAs ao programador, como memórias e FIFOs, assumindo um ciclo de *clock* de latência para cada *statement* de atribuição. As operações associadas ao mesmo ciclo em uma expressão são controladas pelo programador através da decomposição de expressões. Em Handel-C, a concorrência é explicitamente especificada como seções de código usando instruções *par*. SystemC [6, 7] é uma biblioteca padrão de classes C++ para desenvolvimento de sistemas e *hardware*. Ele permite a definição processos concorrentes (C++ functions) em uma hierarquia de módulos, os quais usam canais para comunicação entre módulos/processos. Após a criação dos módulos, os processos são escalonados e sincronizados durante uma etapa de simulação. O escalonador usa eventos (como escrita e leitura) para determinar a ordem de execução dos processos, incluindo aqueles que devem ser executados concorrentemente. Bluespec [8] é uma linguagem para descrição de *hardware* baseada em SystemVerilog. Bluespec permite que o ciruito seja descrito diretamente por meio de regras, onde as instruções são executadas em paralelo. Isto fornece um nível de abstração alto, além de um controle da arquitetura pelo desenvolvedor.

Em LALP, o programador não especifica o circuito em termos arquiteturais, mas em termos comportamentais. LALP diferencia-se de abordagebs anteriores no sentido em que é assume que todos os *statements* são concorrentes e o que os torna sequenciais é a dependência de dados/controle entre eles e/ou o uso explícito dos qualificadores de escalonamento ('@'). LALP é uma linguagem de propósito especial usada com o objetivo de programar aceleradores em FPGAs focados em *loops*.

5. Conclusões

Este artigo apresentou uma comparação entre *hardware* gerado usando LALP, uma linguagem de propósito específico para programação de aceleradores em FPGAs, *versus* o uso de uma ferramenta de Síntese de Alto Nível (HLS). Foi usado um conjunto de dez *benchmarks* representativos para propósitos de avaliação, e os resultados motraram uma média geométrica de *speedup* de 2.84× das implementações LALP sobre as obtidas pela ferramenta de HLS. Com poucas exceções, LALP mostrou melhores resultados que o Vivado HLS para tempo de execução e uso de recursos.

O nível de abstração fornecido pelo LALP (sintaxe similar ao C), em adição aos resultados obtidos, torna LALP uma escolha razoável para implementação de aceleradores em *hardware* quando as ferramentas de HLS não atingirem desempenho suficiente. Tais resultados motivam-nos a continuar melhorando a linguagem e o compilador LALP.

Referências

[1] R. Menotti, J. M. P. Cardoso, M. M. Fernandes, and E. Marques. LALP: A Language to Program Custom FPGA-based Acceleration Engines. *Int. Journal of Parallel Programming*, 40(3):262–289, 2012.

[2] Nadav Rotem. [online]: http://www.c-to-verilog.com/, 2010.

[3] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C With ROCCC 2.0. In *2010 18th IEEE Annual Int. Symp. on Field-Programmable Custom Computing Machines*, pages 127–134. IEEE, 2010.

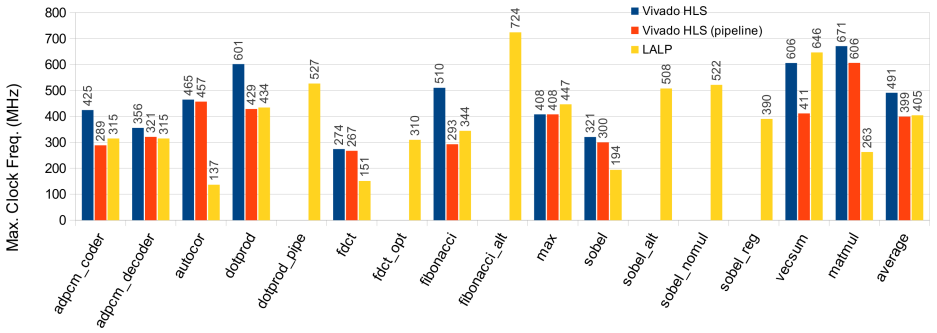
[4] Tom Feist. Vivado Design Suite. White Paper, 2012.

[5] Agility Design Solutions Inc. Handel-C Language Reference Manual. White paper, 2007.

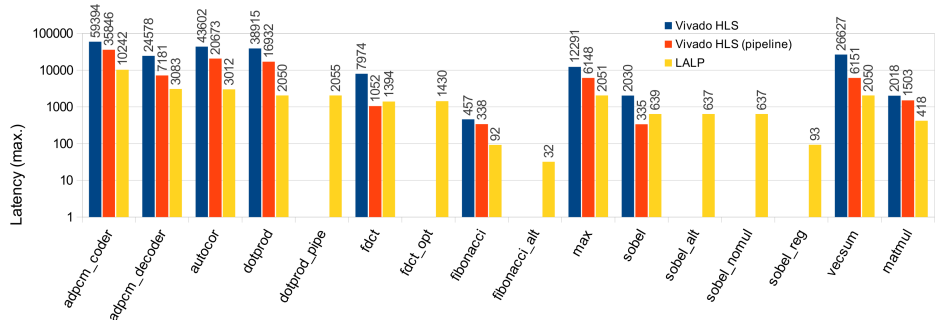
[6] IEEE Comp. Society. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666-2011, 2012.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Springer, 2002.

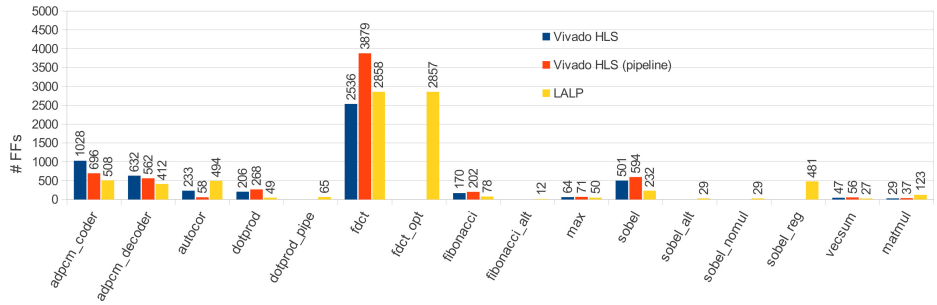
[8] Bluespec Inc. Bluespec SystemVerilog Reference Guide. White Paper, 2012.



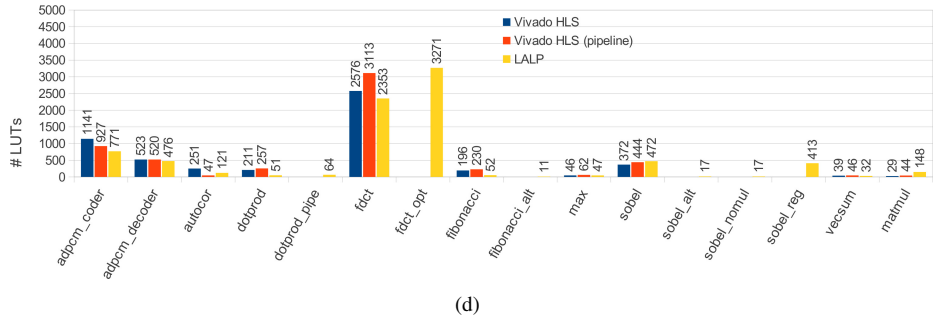
(a)



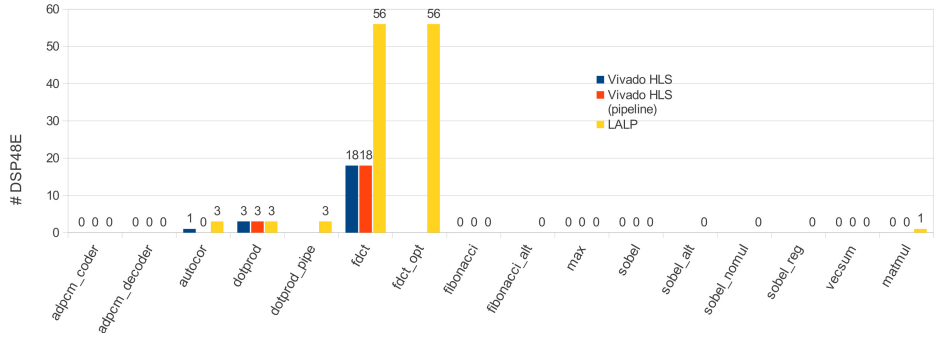
(b)



(c)



(d)



(e)

Figura 4. Resultados obtidos usando LALP e Vivado HLS: (a) Frequência máxima; (b) Latência em ciclos de *clock*; (c) Flip-flops (#FFs); (d) LUTs; (e) #DSP48E

A Design Space Exploration tool for combine code transformations and cache configuration for a open-source softcore processor

†Luiz G. A. Martins, ‡Cristiano B. de Oliveira, ‡Erinaldo Pereira, ‡Eduardo Marques
†*Faculty of Computing (FACOM), Federal University of Uberlandia (UFU), Brazil*
†‡*Inst. of Mathematical and Computer Sciences (ICMC), Univ. of Sao Paulo (USP), Brazil*
gustavo@facom.ufu.br, {lmartins, cbacelar, erinaldo, emarques}@icmc.usp.br *

Abstract

Systems that use softcore processors can perform both architecture customizations and code optimizations in order to satisfy design requirements and constraints. This paper presents a preliminary integrated compilation environment developed to search design configurations that have an attractive trade-off between execution time and cache usage efficiency. This environment is composed by a design space exploration engine based on NSGA-II Genetic Algorithm, a source-to-source compiler and a cycle accurate simulator. It automatically performs cache configuration and C code transformations, in order to improve the performance of the application. The proposed environment was evaluated by using a set of kernel codes, which resulted in performance and cache usage improvements.

1. Introduction

Many Systems on Chip (SoCs) use softcore processors in their architectures. By doing this is possible to leverage the benefits of a high-level sequential programming paradigm while allowing the developer to customize the architecture according to the application. In this scenario, hardware and software level optimizations are needed for a better use of devices resources. These optimizations allow an exploration of design possibilities directly related to the application, leading to a better system's performance.

Once both software and hardware can be optimized in order to achieve solutions for a high performance computation, an efficient Design Space Exploration (DSE) is fundamental to discover the synergy points which compose these potential configurations. Likewise, an integrated compilation process can result in designs committed to the trade-off between resource usage and performance.

Prototyping and architecture exploration are frequently used by HLS (High-level Synthesis) tools to model the functionality of different architectures and to define limits for area, delay and power requirements [1].Another possibility is to perform code transformations in order to improve the software performance, by managing, for instance, the data access and loop operations [2]. However, the number of options and the possibilities of customization for some applications can make the whole process unfeasible. In such cases, the exploration process can be cumbersome, once the design space grows significantly according to the number of architectural and code transformation parameters.

This paper presents a compilation system that performs a design space exploration focused on cache configuration and C code optimizations for a custom softcore processor. The basic infrastructure of the proposed system uses a DSE engine based on NSGA-II (a multiobjective evolutionary algorithm), a ROSE-based source-to-source C compiler and a cycle accurate simulator targeting a custom softcore processor. Such infrastructure searches the best trade-off between the number of cycles and the cache usage for a given C code. This is a preliminary version of the system proposed in [3, 4].

The remaining parts of this paper are organized as follows. In Section 2 some of the tools related to the proposed system are presented. The proposal details are described in Section 3. The Section 4 shows the experiments and the results. Finally, a conclusion is presented in Section 5.

2. Related Work

Considering the challenges in the development of reconfigurable architectures, some projects investigate solutions to assist the application design. Efforts are made to develop optimizing compilers that are able to perform an automatic design space exploration (DSE), aiming different requirements and technologies.

Many of these compilers use code transformations approaches, like in [5, 6, 2]. The optimizations performed by these compilers are, generally, focused on loops and/or data-oriented. Other tools try to tackle architectural issues through performing a multiobjective DSE for simultaneous area, delay and power optimizations [7, 8]. Another possible approach is to consider the program behavior in order to define the hardware architecture [9, 10]. These approaches handle simultaneously hardware and software parameters, aiming the customization of the system.

3. Proposed System

This paper presents a preliminary compilation environment focused on software optimization and cache configu-

*LGAM is granted by CAPES (process 0352-13-6). The other authors would like to thank FAPESP for the financial support provided.

ration for a single softcore processor system. Despite targeting a specific processor, the tool can be easily adapted for different processors. The Figure 1 shows the proposed environment.

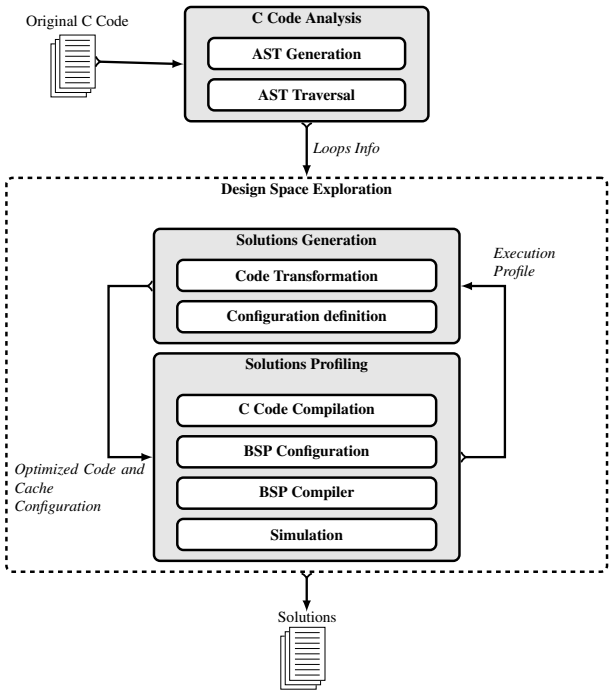


Figure 1. Proposed tool

The main task performed by this environment is the DSE, which can be seen as an iterative process with two basic orthogonal issues: 1) how to explore the design space in order to improve the diversity of the evaluated solutions and 2) how to evaluate a single design space point into a viable time. The selection of the search method determines the granularity of design covering, while the choice of the space evaluation method seeks a compromise between accuracy and cost of the evaluation model [11].

In this paper, the DSE is focused in discovering design points which determine the code transformations and the cache parameters. Then, it is expected that the final solutions present a good trade-off between execution time (number of cycles) and cache usage efficiency. The DSE process is a loop divided into two main steps: Solutions Generation and Solutions Profiling. The approach used for the DSE adopts a heuristic that guides the exploration in the search for a set of parameters used to transform the input C code and to configure the cache memory of the softcore processor. Since a multiobjective approach is used, the final result of the DSE is a set of pre-selected solutions. This set contains the best group of individuals (non-dominated solutions) considering both target metrics.

The environment also includes a source-to-source compiler that applies the code transformations defined by the DSE. This compiler is developed according to the source-to-source ROSE Compiler infrastructure (see [12]).

3.1. Code Analysis

The first task performed consists of an analysis of the input source code, written in C. This step is performed by the same ROSE-based compiler previously mentioned. After the lexical, syntatic and semantic analyses, the compiler generates an intermediary representation in form of AST (Abstract Syntax Tree). The AST is then traversed in order to generate information about the loops structures (*for* statements) inside the code. The system retrieves the number of iterations, the nest and depth levels and the position in AST for each loop present in the code. This information is used in the DSE process.

3.2. Solution Generation

The Solutions Generation runs a multiobjective evolutionary algorithm called *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) [13]. This algorithm starts with the random generation of the initial population. Each individual encoding represents one candidate configuration that consists of cache and code transformations parameters. The selected code transformations are applied to the original source code, while the cache configuration is used for configure the softcore processor that will run the optimized code. The Table 1 shows the set of parameters supported by the compilation environment, where N is the number of loop iterations. The both instruction and data cache sizes are defined by the number of cache lines $\times$ line size (word length of 4 bytes). The number of cache lines presented in Table 1 is represented as exponent of power of 2 (2^K, $4 \leq K \leq 12$). A description of the code transformations can be found in [14].

Description	Parameters defined in DSE
Loop interchange	Yes / No
Loop blocking type	None, Outermost, Innermost or All loops
Blocking size	1 to N
Loop fusion	None, Single or Multi-level
Loop fission	Yes / No
Loop splitting	Yes / No
Loop unrolling	1 to N
Optmizaion level	1 to Max depth
Data Cache size	K
Instruction Cache size	K

Table 1. Supported set of optmization parameters

3.3. Solution Profiling

The Solutions Profiling step consists in evaluate these codes through their simulation profiles. The profiles are used in the evaluation phase of the NSGA-II to classify the population. The major difference between NSGA-II and a single-objective GA [15] is how the individuals are evaluated. NSGA-II is based on the concept of dominance depth [16], which classifies the current population in sev-

eral non-dominance fronts. The fitness is obtained by two attributes: the individual non-dominance front and a density estimation, called crowding distance, which increases the population's diversity. More details about NSGA-II can be found in [13]. During this step the system retrieves the fitness of each solution by using two metrics: the number of clock cycles (N_{cycles}) required for computation and a cache usage metric (E_{cache}) computed from hit and miss rates for both data and instruction cache, according to the following equation:

$$E_{cache} = \frac{D_s}{\left(\frac{D_h}{D_h+D_m}\right)} + \frac{I_s}{\left(\frac{I_h}{I_h+I_m}\right)} \quad (1)$$

Where D_h, I_h, D_m and I_m are the number of hit and miss events from data and instruction caches, respectively; and D_{size}, I_{size} are the cache sizes defined by the DSE. The DSE aims minimize N_{cycles} and E_{cache} values.

The metrics (N_{cycles} and E_{cache}) are calculated for a specific processor. This project uses the BSP (Bluespec SoftProcessor) processor. This processor is a reconfigurable softcore processor developed by the Laboratory of Reconfigurable Computing of University of São Paulo. Its code is written in Bluespec SystemVerilog language [17]. It is a 32-bits processor with a 3-stage pipeline that implements the same Instruction Set Architecture (ISA) of the Altera's Nios II processor.

In contrast to Nios II, the BSP is an open-source processor that allows entire customization. The BSP also includes a cache monitor, which provides the cache hit and cache miss rates. Despite the number of configurations supported by the BSP, the proposed system configures only its cache memory. This is done according to the parameters defined by the DSE.

The Bluespec SystemVerilog (BSV) development environment provides a cycle accurated simulator named Bluesim. Thus, it is possible to run the code and to gather an execution profile about ten times faster than the complete synthesis for a real hardware execution. The number of cycles and the hit and miss rates for both data cache and instruction cache are directly obtained in simulation.

The simulation flow using Bluesim is presented in Figure 2. In order to perform the simulation, it is necessary to compile the optimized C code using the *nios2-elf-gcc* compiler, provided by Altera. The binary file produced is formatted into a memory file that contains all the instructions and data that will be used by the simulator. This memory file is then linked to the BSP Bluespec code that incorporates the cache parameters provided during Solutions Generation step.

After the processor is configured, it is necessary to compile the processor's BSV code using the Bluespec compiler (BSC). The BSC produces an output file that is passed to the Bluesim in order to perform the simulation.

The simulation results in a customized profile report that contains the number of cycles, the hit and miss rates and other informations as well. The evaluation is made for all valid individuals. If an invalid individual is created, bad values are attributed to its metric without requiring simulation.

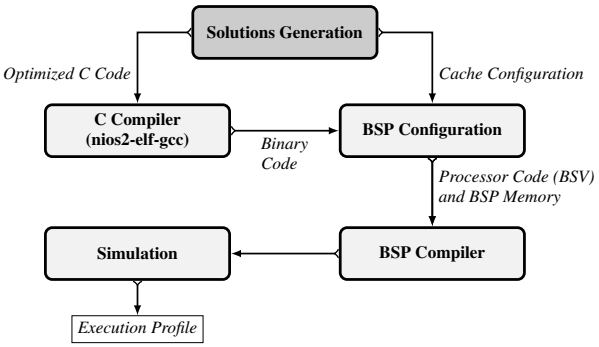


Figure 2. Execution flow steps for simulation using Bluesim

4. Experiments and Results

The system was tested with three different programs: a matrix multiplication, a sobel filter and a bubble sort algorithm. The design space size is determined by the number of parameters (see Table 1). For instance, considering a code with 2 nested loops and 10 loop iterations, the design space is composed by over $1.5x10^6$ possible configuration points.

The experiments used the same GA configuration, which were empirically chosen. The NSGA-II runs over 20 generations, each one with a population size of 100 individuals. A simple tournament selection ($Tour$ = 3) was used for parents selection [15]. A uniform crossover operator was defined with 70% of probability for applying crossover to a pair of individuals and another probability of 50% for swapping each gene's value of two individuals. It was used a mutation rate of 10%. The elitism reinsertion strategy keeps the best design configurations in population. Due to stochastic nature of GAs, each experiment was repeated 10 times.

Programs	Solutions	N_{cycles}	Speedup
BubbleSort	Reference	190679	
	Best N_{cycles}	190238	1.00
	Best E_{cache}	273029	0.70
MatMult	Reference	32342	
	Best N_{cycles}	12789	2.53
	Best E_{cache}	29658	1.09
Sobel	Reference	14174	
	Best N_{cycles}	12559	1.13
	Best E_{cache}	15730	0.90

Table 2. Average values of N_{cycles} and speedups for the final population over the original programs

The Table 2 shows the average of the best values in the 10 executions for N_{cycles} and speedup while Table 3 shows the average of the best values for E_{cache} and the sizes (in bytes) of both data and instruction cache for each benchmark. The original code and maximum cache configuration are used as reference and the showed values were obtained

Programs	Solutions	E_{cache}	D_{size}	I_{size}
BubbleSort	Reference	32848,54	16384	16384
	Best N_{cycles}	4630,38	4096	512
	Best E_{cache}	207,41	64	64
MatMult	Reference	34607,09	16384	16384
	Best N_{cycles}	2291,34	1024	1024
	Best E_{cache}	334,31	64	128
Sobel	Reference	34497,16	16384	16384
	Best N_{cycles}	2781,63	2048	512
	Best E_{cache}	246,96	64	128

Table 3. Average values of E_{cache} and sizes of Data cache (D_{size}) and Instruction cache (I_{size}) memories

for the best solutions considering each metric separately. The solutions with best N_{cycles} values corresponds to the worst E_{cache} and vice-versa for all benchmarks.

As can be noted, in the best case for N_{cycles} the *MatMult* was 2.53× faster than original program, while the original program was 30% faster in relation to the best E_{cache} solution. However, *BubbleSort* was not improved in this metric. In other hand, it presented the best solution considering E_{cache} values, with a reduction of 158× over the reference configuration and of 7× in the worst case.

The adoption of the maximum cache size strongly influences the E_{cache} metric computation. Thus, the high decreasing of this values is due to the difference of the cache size between the original and the optimized solutions. A reduction of E_{cache} results in a reduction of both instruction and data cache sizes (see Table 3).

5. Conclusion

This paper presented a compilation scheme able to optimize C code for architectures based on a open-source softcore processor. The scheme's kernel is a DSE tools that explore simultaneously the design space composed by code transformations and cache parameters, enabling the achievement of custom designs focused on a trade-offs between execution time and cache efficiency.

Although the current environment is a preliminary version of the proposed HLS-based project, its first results are considered satisfactory. However, since were used very simple benchmarks, more robust tests are required in order to determine the efficiency of the system for real-life applications.

This project aims to help the development of SoCs applications by assisting the developer in the process of optimizing both code and architecture. The current focus is on code transformations and cache configuration, but the future version intends to include other optimization parameters, like power consumption and area.

References

[1] D. S. H. Ram, M. C. Bhuvaneswari, and S. M. Logesh. A novel evolutionary technique for multi-objective power, area and delay optimization in high level synthesis of datapaths. In *2011 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI'11, pages 290–295, Washington, DC, USA, 2011. IEEE Computer Society.

[2] J.M.P. Cardoso, R. Nane, P.C. Diniz, Z. Petrov, K. Krátkỳ, K. Bertels, M. Hübner, F. Gonçalves, J.G.F. Coutinho, G. Constantinides, et al. A new approach to control and guide the mapping of computations to fpgas. In *Int. Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA'11)*, pages 231–240, 2011.

[3] C. B. Oliveira and E. Marques. Combining data and computation transformations for fine-grain reconfigurable architectures. In *22nd Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 489–490, 2012.

[4] L. G. A. Martins and E. Marques. Design space exploration based on multiobjective genetic algorithms and clustering-based high-level estimation. In *23nd Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–2, 2013.

[5] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient hardware code generation for FPGAs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–26, 2008.

[6] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C With ROCCC 2.0. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134. IEEE, 2010.

[7] G. Beltrame, L. Fossati, and D. Sciuto. Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 29:1083–1095, Jul 2010.

[8] SM Logesh and Harish Ram. A survey of high-level synthesis techniques for area, delay and power optimization. *International Journal of Computer Applications*, 32(10), 2011.

[9] C. Dubach. *Using Machine-Learning to Efficiently Explore the Architecture/Compiler Co-Design Space*. Phd thesis, University of Edinburgh, Edinburgh, 2009.

[10] Ralf Jahr, Theo Ungerer, Horia Calborean, and Lucian Vintan. Automatic multi-objective optimization of parameters for hardware and code optimizations. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 308–316. IEEE, 2011.

[11] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, Dec 2004.

[12] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi. *ROSE User Manual: A Tool for Building Source-to-Source Translators (Draft Manual)*. Lawrence Livermore National Laboratory.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Transactions on Evolutionary Computing*, 6(2):182–197, Apr 2002.

[14] J. M. P. Cardoso and P. C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer, 2008.

[15] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[16] E. Zitzler, M. Laumanns, and S. Bleuler. A tutorial on evolutionary multiobjective optimization. In X. Gandibleux et al., editor, *Metaheuristics for Multiobjective Optimisation*, Lecture Notes in Economics and Mathematical Systems. Springer, 2004.

[17] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proc. 2nd ACM and IEEE Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE'04)*, pages 69–70. IEEE, 2004.

A DSE Example of Using LARA for Identifying Compiler Optimization Sequences

Ricardo Nobre, João M. P. Cardoso and José C. Alves
Universidade do Porto, Faculdade de Engenharia (FEUP) and INESC TEC
Porto, Portugal
{*ricardo.nobre, jmpc, jca*}*@fe.up.pt*

Abstract

This paper presents a technique to devise sequences of compiler optimizations able to increase the performance of a given function when mapped to software or hardware. The techniques are being evaluated using an integrated hardware/software compiler controlled by strategies expressed in LARA, a domain-specific language developed to guide/control code instrumentation, transformations and compiler optimizations. We evaluate a Design Space Exploration (DSE) scheme based on the Simulated Annealing algorithm, which is able to explore compilation design space points generated by distinct compilation sequences. In this work, two hot-spot functions from industrial applications, filter subband *and* grid iterate*, were used to evaluate the Simulated Annealing-based DSE. The experiments resulted in maximum speedups of* 1.86 *and* 1.25 *for* filter subband*, and* 2.29 *and* 1.11 *for* grid iterate*, when targeting a MicroBlaze processor and reconfigurable hardware, respectively.*

1. Introduction

Designing high-performance hardware and software components requires the exploration of multiple design alternatives, through a process called Design Space Exploration (DSE). A complete exploration of the design space for software and hardware components, given a set of DSE parameters, can be an infeasible task, due to the large number of alternatives to be explored. This problem is even aggravated when targeting multiple platforms, e.g. different microprocessors and/or reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs).

Although it is a common practice to apply the same set of optimizations in a fixed order on each method of a program when targeting a given architecture/platform, several researchers have shown the best order of optimizations is function-specific (see [1]). Therefore, an important goal of a compiler DSE scheme is finding sequences of compiler optimizations, e.g. to increase performance [2].

In order to support the specification of compilation sequences and DSE schemes, a hardware/software toolchain and an aspect-oriented programming language called LARA [3] have been recently developed in the context of the FP7 REFLECT project [4]. LARA allows developers to express monitorization, hardware/software partitioning, synthesis actions, separately from the application code, thus providing a novel approach to hardware/software design with potential to reduce the burden that is traditionally associated with such tasks.

The work presented here is focused on describing DSE strategies to devise suitable optimization schemes for a given function/program when targeting software and reconfigurable hardware. We present in this paper the results of using a Simulated Annealing-based DSE scheme to guide the compilation of two kernels when targeting a MicroBlaze processor and reconfigurable hardware, striving to minimize the latency of the generated design.

In Section 2 we introduce the used DSE infrastructure, describing the aspect-oriented language LARA, used to control an integrated toolchain, and explaining how LARA can be used to implement DSE schemes. Section 3 presents the Simulated Annealing algorithm used in the proposed DSE approach. Section 4 describes the methodology of our experiments and the obtained results. Section 5 presents related work and concluding remarks are presented in Section 6.

2. DSE Infrastructure

Our work uses as experimental setup the REFLECT toolchain [5]. This toolchain is able to target multiple processor architectures as well as FPGAs. DSE schemes are implemented by writing LARA aspects that rely on an outer loop which controls calls to toolchain tools, as depicted in Figure 1. This outer loop can select/compute and send parameters to different tools or to calls of other LARA aspects, and uses a feedback mechanism to read information from tool reports in an integrated way.

A DSE strategy is therefore implemented using LARA aspects as follows. A top-level aspect implements the DSE algorithm relying on the LARA outer-loop mechanism, while other aspects are responsible for receiving an optimization sequence that guides the toolchain compiler, *ReflectC*, in the process of generating a solution design, and for simulation/execution of the resulting designs. The modularity of our DSE infrastructure requires only the aspect that encapsulates the execution of the toolchain compiler to be modified when using a different compiler.

This work was partially supported by Fundação para a Ciência e a Tecnologia (FCT) under grant SFRH/BD/82606/2011, and FEDER/ON2 and FCT project NORTE-07-124-FEDER-000062.

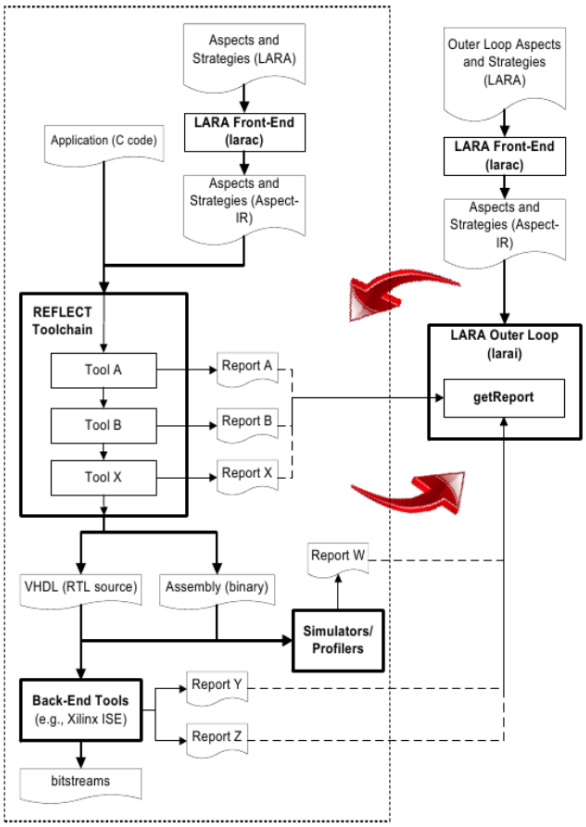


Figure 1. LARA based toolchain flow and the LARA outer loop mechanism (from [5]).

2.1. *ReflectC* compiler

The *ReflectC* compiler supports the aspect-oriented design flow controlled by LARA aspects/strategies, and relies on a set of CoSy-based [6] compiler optimization engines, engines for generating code targeting traditional processors, as well as behavioral-RTL VHDL code, using DWARV [7] as the hardware generator.

The integration of LARA with the *ReflectC* CoSy-based compiler is done by a LARA weaving mechanism, implemented in the form of a CoSy engine that is executed when executing joinpoint and attribute queries, or actions upon the CoSy intermediate representation, called CCMIR, prior to the execution of a pipeline to which all CoSy engines passable of being executed are attached to.

As an example, a LARA aspect that instructs the *ReflectC* compiler to perform analysis, scalar replacement and unroling on the innermost loops with known number of iterations smaller or equal than 20 is depicted by Figure 2. Note the *select*, *apply* and *condition* keywords, used for selecting joinpoints, applying actions, and filtering joinpoints by an expression constructed with their attributes, which evaluated to true or false at weaving time. A description of the over 20 attributes currently supported by the *ReflectC* LARA weaver engine can be consulted in [5].

```
aspectdef  optimize
  input functionName end
  select function{name==functionName}.loop{type=="for "
       } end
  apply
    optimize(kind: "loopanalysis");
    optimize(kind: "loopscalar");
    optimize(kind: "loopunroll", k: "full"); // fully
         unroll loop
  end
  condition
    $loop.is_innermost && $loop.numIterIsConstant &&
         $loop.num_iter<=20
  end
end
```

Figure 2. A LARA aspect for performing loop optimizations.

2.2. LARA outer-loop mechanism

The LARA outer-loop aspect is executed by a separated LARA interpreter tool, *larai*.

Figure 3 depicts, as an example, LARA code that is used inside a LARA loop in order to implement a DSE scheme targeting the Microblaze processor. This code instructs the toolchain to execute the *ReflectC* compiler, simulate the generated solution, and extract the latency value for the generated design. *reflectcOpts* contains arguments for the *ReflectC* compiler, including the optimization sequence.

```
run(tool: "reflectc", args: reflectcOpts, verbose:2);
run(tool:"microblazecc", args: ccargs, verbose:2);
run(tool:"microblazesim", args: simargs, verbose:2);
report("microblazeexporter", [cfile_noext]);
var latencyRef = @function[cfile_noext].latency;
```

Figure 3. LARA code used to control the *ReflectC* compiler toolchain when targeting the MicroBlaze soft-processor.

3. Simulated Annealing Algorithm

In this paper we focus on finding the optimization sequence that results in the best performance for a given function, considering a target platform (processor or reconfigurable hardware) and the set of optimization engines supported by the toolchain.

A straightforward but naive approach to DSE of the optimization sequences subspace is to test all possible combinations of k optimizations from a list of n possible optimizations to choose from. While this approach is guaranteed to find an optimal design point, as it tests all optimization combinations, it is not practical.

Simulated annealing [8] (see pseudocode in Algorithm 1) is an effective and practical algorithm for optimization problems. Variables s, e and t represent the state, energy and temperature at a given iteration of the algorithm. Variables s_0, s_{best} and e_{best} represent the initial solution configuration, the currently accepted configuration and the associated energy, respectively.

Algorithm 1 Simulated annealing pseudo-code.

```
s ← s0
e ← E(s)
s_best ← s
e_best ← e
k ← 0
while T > t_min do
    T ← temperature(k)
    s_new ← neighbour(s)
    e_new ← E(s_new)
    if P(e, e_new, T) > random() then
        s ← s_new
        e ← e_new
    end if
    if e_new < e_best then
        s_best ← s_new
        e_best ← e_new
    end if
    k ← k + 1
end while
return s_best
```

Function $E(s)$ evaluates the energy used by a given configuration s, i.e. the latency of a given solution targeting the MicroBlaze processor or reconfigurable hardware, and $temperature(k)$ calculated the next temperature T.

The perturbation rules that generate the next optimization sequence $neighbour(s)$ to be tested for latency were the following three:

1. Swap two randomly picked optimizations;
2. Replace an optimization by an optimization from the complete optimization list;
3. Replace the unroll factor of a loop by a factor from the list of allowed unroll factors.

Only one perturbation strategy is selected each iteration of the simulated annealing algorithm. Each perturbation rule has the same probability of being selected, being the last (rule *3*) only selected if the loop unrolling engine is used.

4. Experiments

Our Simulated Annealing-based DSE scheme is able to explore design points, targeting software or hardware, and runs on top of the REFLECT toolchain using the LARA aspect-oriented programming language. The DSE was performed using 34 CoSy engines, including but not limited to constant propagation, loop invariant code motion, common sub-expression elimination, and loop unrolling. Two separated executions of the DSE were performed, without and with the inclusion of the loop unrolling engine. The unroll factor was allowed to arbitrary be set to either one (i.e. no unroll) or two for each loop of the given kernel. Additional loop factors could be explored at the cost of aditional exploration time.

The kernels used as test-beds for the developed DSE scheme were given by Coreworks and Honeywell, and are respectively a *filter subband* function and a relaxation kernel called *grid iterate*. The first is part of an MPEG audio encoder, and the latter is from an avionics 3D path planning application. The *filter subband* has two input arrays of data type *float* (z and m) and an output array of the same data type (s). The *grid iterate* has a multidimensional array of fixed-point data type (represented as *int*) as input (the *obstacles* array) and an array of the same type and shape as output (the *potential* array).

Multiple executions of the Simulated Annealing DSE scheme were performed using different values for alpha, i.e. the temperature decrease multiplication factor. However, here are reported the results obtained for alpha equal to 0.98. The minimum and maximum temperatures (T_{min} and T_{max}) were experimentally chosen to be 0.01 and 100000, respectively; resulting in the exploration of 798 alternative optimization sequences. Also, distinct maximum optimization sequence length were tested to determine how it affects performance (i.e. latency) for each kernel, represented by the energy function $E(s)$. A cycle accurate MicroBlaze simulator and a RTL ModelSim simulator were used for simulating the software and hardware implementations, respectively.

Table 1 depict results for the *filter subband* and *grid iterate* kernels generated when targeting the MicroBlaze processor without loop unrolling. Maximum length of optimization sequences, resulting speedup and exploration time are respectively represented by #, S and Δt. As expected, the optimization sequences selected by the Simulated Annealing DSE scheme improve the reference latency (i.e. without optimization).

Table 1. Software optimization sequences found for *Filter Subband* and *Grid Iterate* without loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Subb.	4	1.57	17.76	loopinvariant, loopstrength, loopscalar, loopguard
	8	1.68	21.24	loopinvariant, loopscalar, dismemun, loopstrength, strength, looprev, lowerboolval, loopbcount
	16	1.71	26.39	loopinvariant, cse, copyprop, loopstrength, loopguard, loopscalar, cache, vstrength, dismemun, blockmerge, strength, looprev, loophoist
Grid.	4	1.98	30.15	dismemun, loopinvariant, cse, loopguard
	8	2.07	34.04	noreturn, loopguard, loopinvariant, chainflow, dismemun, cse, cache, strength
	16	2.07	38.53	copyprop, loopguard, cache, loopive, vprop, looprev, dismemun, loopscalar, loopinvariant, demote, promote, cse, strength, lrre-name, lowerbitfield

Table 2 depicts results obtained including the loop unrolling engine when targeting the MicroBlaze soft-core. Unrolling the loops lead to an increase in performance for the both kernels, at the cost of considerably more exploration time (around 65% and 54% for *filter subband* and *grid iterate* respectively). For *grid iterate*, the using loop unrolling results in performance improvements when using up to 8 or 16 optimizations, 7% and 11%, in relation to the reference implementations. For the *filter subband* kernel, the speedup improvements obtained with loop unrolling were 4%, 7% and 9% respectively for sequence composed of up to 4, 8 and 16 engines.

Table 3 depict results for the *filter subband* and *grid iterate* kernels without loop unrolling when targeting hardware. The DSE took longer when targeting hardware because of longer simulation time.

Table 2. Software optimization sequences found for *Filter Subband* and *Grid Iterate* with loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Subb.	4	1.63	29.92	loopinvariant, unroll, loopguard, loopscalar
	8	1.79	32.47	condassigncreate, loopinvariant, loopscalar, dismemun, loopstrength, looprev, unroll, strength
	16	1.86	45.60	loopguard, loopbcount, loopstrength, mvpostop, setrefobj, dismemun, loopinvariant, loopfuse, lrrename, loopscalar, promote, vprop, funceval, strength, looprev, unroll
Grid.	4	1.74	47.50	loopinvariant, alias, unroll, loopguard
	8	2.21	50.03	dismemun, unroll, constprop, conevun, loopinvariant, cse, lowerpfc, loopguard
	16	2.29	60.64	ifconvert, dismemun, exprprop, constprop, unroll, unroll, algebraic, loopinvariant, loopguard, markconvert, demote, tailmerge, cse, tailrec, strength, conevun

Table 3. Hardware optimization sequences found for *Filter Subband* and *Grid Iterate* without loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Subb.	4	1.13	168.60	loopscalar, cache, loopremove, loopstrength
	8	1.13	171.34	scalarreplace, loopscalar, noreturn, loopstrength, cache, exprprop, hwloopcreate, conevun
	16	1.13	175.18	tailrec, loopscalar, alias, conevun, setpurity, cache, loophoist, loopstrength, lowerpfc, funceval, lowerboolval, hwloopcreate
Grid.	4	1.11	216.31	loopstrength, loopbcount, promote, loopinvariant
	8	1.11	221.98	lowerboolval, setpurity, cse, lowerbitfield, loopstrength, vstrength, loopinvariant, loopbcount
	16	1.11	224.32	loopive, cse, globcse, loopstrength, loopinvariant, misc, conevun, loopbcount, noreturn, promote, ckfstrength, loopcanon, dismemun

Table 4 depicts the results using loop unrolling when targeting hardware for the *filter subband* kernel. Speedups over reference improve for all up to k sequence lengths in relation to experiments without loop unrolling, 5% for up to 4 and 11% for up to 8 and 16 optimizations, at the cost of an average exploration time increase around 6%.

Table 4. Hardware optimization sequences found by SA for *Filter Subband* with loop unrolling.

Kernel	#	S	Δt (min)	Optimization Sequence
Subb.	4	1.19	176.95	loopinvariant, unroll, cache, loopguard
	8	1.25	171.98	exprprop, loopstrength, loopguard, conevun, unroll, loopscalar, cache, promote
	16	1.25	197.76	ckfstrength, condassigncreate, loopguard, loopinvariant, loopstrength, tailrec, loopfuse, cse, unroll, loopstrength, lrrename, cache, loopcanon, loopremove, loopbcount, scalarreplace

More significant speedups were achieved in software than in hardware and generally using more engines resulted in performance improvements.

Additional experiments considering the exploration of 395 (alpha equal to 0.96) instead of 798 (alpha equal to 0.98) alternative optimization sequences, resulted in reducing the exploration time to half while the performance degradation was only around 7% and 5% when targeting *filter subband* and *grid iterate* for software with loop unrolling; with no performance degradation when targeting hardware.

5. Related Work

State-of-art DSE optimization phase ordering approaches (e.g. [9]) demonstrate that dynamically choosing the best ordering of optimizations has a significant impact on the execution time of applications.

The development of software in the form of complete toolchains that have optimization ordering into account has been considered by a number of authors (see, e.g. [2]). However most existent tools either rely heavily on code annotations (see, e.g. [10]), thus limiting the kind of optimizations that can be performed and the possible compilation targets, or simply do not provide an integrated hardware/software view.

This work uses an environment developed in the context of the REFLECT FP7 project [5], providing mechanisms that ease the implementation of DSE strategies through the usage of a aspect-oriented domain specific language named LARA to control the compilation flow of toolchain.

6. Conclusions

LARA aspects have the potential to guide the tool-chain in the exploration of design points targeting a single or multiple hardware/software platforms. In this context, the effective exploration of design points targeting a software and hardware platforms was successfully expressed using a single incrementally developed LARA aspect implementing a Simulated Annealing algorithm.

The choice of optimizations, and their order of application, can have an appreciable impact on performance and are platform- and application-dependent. Besides, the usage of the loop unrolling engine generally improves the speedups of loop-based kernels when targetting both software and hardware at the cost of a considerably increased exploration time when targetting software.

References

[1] L. Almagor *et al.*, "Finding effective compilation sequences," *SIGPLAN Not.*, vol. 39, pp. 231–239, Jun. 2004.

[2] G. Fursin *et al.*, "MILEPOST GCC: machine learning based research compiler," in *Proc. of the GCC Developers' Summit*, Jul. 2008.

[3] J. M. P. Cardoso *et al.*, "Lara: an aspect-oriented programming language for embedded systems," in *Proc. 11th int. conf. on Aspect-oriented Software Development (AOSD '12)*. ACM, 2012, pp. 179–190.

[4] *REFLECT Project (2013), http://www.reflect-project.eu*.

[5] J. M. P. Cardoso *et al.*, *Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach*. Springer, 2013. [Online]. Available: http://books.google.pt/books?id=PRlgLwEACAAJ

[6] *ACE CoSy Compiler Development System (2012), http://www.ace.nl/compiler/cosy.html*.

[7] R. Nane *et al.*, "Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler," in *Proc. 22nd Int. Conf. on Field- Programmable Logic and Applications (FPL' 12)*, 2012, pp. 619–622.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[9] P. A. Kulkarni *et al.*, "Practical exhaustive optimization phase order exploration and evaluation," *ACM Trans. Archit. Code Optim.*, vol. 6, pp. 1:1–1:36, Apr. 2009.

[10] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in *Proc. IEEE Int. Symp. on Parallel & Distributed Processing (IPDPS '09)*. IEEE CS, 2009, pp. 1–11.

Comunicações de dados

 ISBN: 978-989-98875-1-0

Design and Verification of a Multi-Port Networked Test and Debug Core

João C. Fernandes, José T. de Sousa
INESC-ID Lisboa
{*joaocfernandes,jose.desousa*}*@inesc-id.pt*

Abstract

Verifying reconfigurable systems or any computing system is hard. This work presents an implementation of multiple UDP ports for FaceWorks, an IP core used for hardware test and debug over the UDP/IP network protocol, which is a patented and proprietary technology of the company Coreworks SA. The objective of having multiple ports is to allow multiple software threads or processes to independently control the device under test (eg. multiple audio streams driven by multiple programs). The new FaceWorks hardware has been designed in Verilog and tested using a SystemC model and an FPGA board. The SystemC model is automatically generated and simulated using Verilator. A PC software application to exercise the two UDP ports for testing the system has been developed. This application connects to both the SystemC model or FPGA board, indistinguishably. Experiments and implementation results are reported.

1. Introduction

Nowadays, integrated circuit designers are presented with extremely short design cycles. To deliver on time, companies are required to optimize the design flow. Of the many solutions created, one of the most successful is the reuse of pre-existent and complex blocks, often called Intellectual Property Cores, or simply IP Cores.

As FPGAs present themselves as flexible and cost effective platforms to test, verify and produce IP cores, challenges arise on how to properly communicate independently with each IP core in the FPGA. IP cores need to be observed, configured, tested and debugged. Complex systems such as reconfigurable systems demand complicated and concurrent testing and debug methods, often requiring expensive equipment to be used.

To solve this problem, the company Coreworks SA created and patented a technology [1, US 2008/0288652], [2, EP 2003571/A2] that uses a simple PC and the office network to exercise multiple cores in the FPGA. The technology has been called Core Access Networks®and comprises the FaceWorks IP core, which is the gateway between the IP cores under test and a test program running on a PC.

FaceWorks implements the UDP/IP/Ethernet protocol stack in hardware, dispensing with an embedded processor and a software protocol stack. FaceWorks also implements the Coreworks Datagram Protocol (CWDP), which adds IP core test and debug functions on top of the UDP/IP stack. Other functions such as data streaming can also be performed with this technology. This solution represents an almost costless, high-speed and versatile test and debug mechanism.

The implementation of FaceWorks that existed when this work started used only a single UDP port to communicate with the FPGA board. The main problem of having a single port is that, to exercise IP modules concurrently, the test driver application becomes very difficult to write. To solve this problem, it has been decided to increase the number of UDP ports supported by FaceWorks, in order to allow multiple software processes or threads to independently and concurrently stimulate and observe multiple IP cores in the system.

This paper is organized as follows. In section 2, necessary background information on FaceWorks is provided. Section 3 presents the new FaceWorks design and its innovative verification strategy. Experimental results are presented and discussed in section 4. Finally, conclusions and directions for future work are given in section 5.

2. Background

A high-level diagram of the FaceWorks IP is shown in Figure 1. Being a network IP, its organization follows the multi-layered ISO networking model. Each layer is explained in each of the following subsections.

2.1. Physical Layer

Like most other networking cores, FaceWorks relies on a third party chip to implement the physical layer and does not include this function inside. The high-speed electronics needed to implement this functionality requires specialized technology, and can not be implemented by regular digital circuitry operating with a single voltage level.

2.2. Data Link Layer

The physical layer is interconnected with the data link layer (aka Media Access Control (MAC)[3] layer) by means of the Media Independent Interface (MII)[4], a well known industry standard. MII is used to interconnect the FaceWorks core in the FPGA and the Ethernet Physical Transceiver (PHY) chip on the same board.

The MII data stream is composed of the Interframe Gap

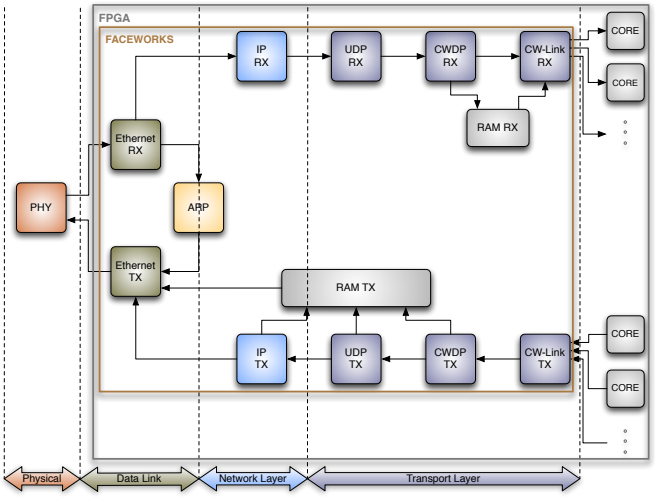


Figure 1. FaceWorks Block Diagram

(IFG), the Preamble, the Start of Frame Delimiter and the actual frame. The MAC frame is encapsulated inside the MII stream. The MAC encapsulates the payload with a 14 byte header before the data payload, followed by a 4 byte Frame Check Sequence (FCS) as show in Figure 2. The FCS algorithm is a cyclic redundancy check (CRC).

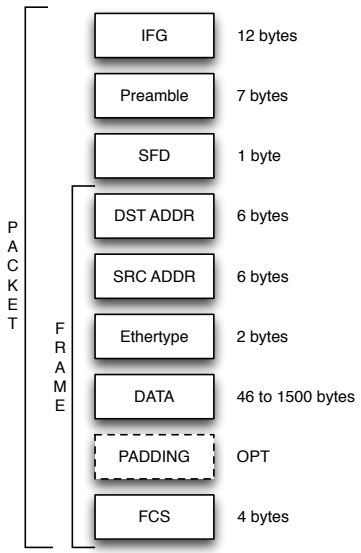


Figure 2. Ethernet packet structure with the MAC protocol

The MAC layer consists of two logic blocks, the Ethernet receiver and an Ethernet transmitter. The receiver block is responsible for removing the preamble, detecting the SFD, performing packet filtering based on the destination MAC and Ethertype, and checking the CRC to validate the data. The inverse operation is performed by the transmitter block: it adds the preamble to the payload, inserts the SFD, the Ethernet header, and calculates the trailing CRC.

2.3. Network Layer

The network layer in FaceWorks uses two blocks, the IP RX block and the IP TX block.

The IP RX block implements the IP protocol for the receive packets. It processes the IP packets that are received from the MAC layer, according to the values of their IP header version, destination IP address and protocol type. It calculates the checksum to verify the header and forwards the payload data to the next block, the UDP RX block.

For transmit packets, the IP TX block calculates the header checksum and writes the IP header and checksum in the RAM TX block, which is where the packet is being formed. Then, it requests the Ethernet TX module to send the packet.

2.4. Coreworks Datagram Protocol

Before the transport layer blocks can be presented, the Coreworks Datagram Protocol (CWDP)[5] must be explained. Figure 3 shows how a CWDP packet is structured, and its fields are briefly described below.

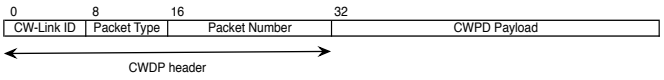


Figure 3. CWDP packet

CW-Link ID This field identifies the destination core interface for which the packet data is intended.

Packet Type This field identifies the CWDP packet type; not all available packet types are used in this work.

Packet Number This field is a sequence number for data packets. It is used to implement reliable transmission. The Packet Number is incremented for every packet acknowledged by the receiver. Only packets with the expected Packet Number are acknowledged by the receiver.

Payload Contains the data to be delivered to the cores, or parameters for FaceWorks.

The CWDP packet types that have been used in this work are presented below. Refer to [5] for a complete outline of the existing CWDP packet types.

SET_CORE_ACCESS (packet type 0x03) This packet type is used to initialize the FaceWorks core. In Figure 4 the structure of a CWDP SET_CORE_ACCESS packet is shown. When a CWDP SET_CORE_ACCESS packet is received by FaceWorks it locks to the host IP and UDP port of the origin for the purpose of replying. The ARP table is cleared and any packet present in the receive or transmit buffers are deleted. Incoming or outgoing packet counters are both reset to zero. The registered host will maintain control of FaceWorks core until a CWDP SET_MAC packet is sent by the host. When FaceWorks is locked to a

host, other incoming CWDP packets from other origins are dropped.

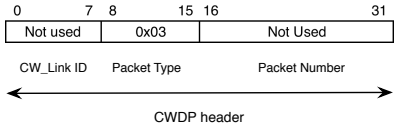


Figure 4. CWDP SET_CORE_ACCESS packet

SET_MAC (packet type 0x04) This type is now used to release control and terminate a connection with the FaceWorks core. In a complete FaceWorks implementation this could be used to change from core access mode to MAC mode. In MAC mode FaceWorks operates as a regular MAC core and CORE_DATA packets can not be transmitted. In Figure 5 the structure of the CWDP SET_MAC packet is shown. When a CWDP SET_MAC packet is received the control of the FaceWorks core is released, the registered IP and UDP port are deleted, the ARP table is cleared and packet buffers are cleared. Incoming or outgoing packet counters are both reset to zero. Control can be regained by sending a SET_CORE_ACCESS packet again.

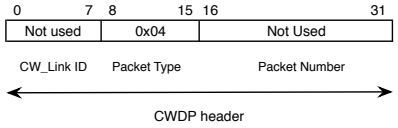


Figure 5. CWDP SET_MAC packet

CORE_DATA (packet type 0x01) This packet type is used to send data between systems and cores. In Figure 6 a CWDP CORE_DATA packet is shown. The packet payload is composed of several CW-Link words, each word is 32-bit long and each CORE_DATA packet can contain up to 360 CW-Link words.

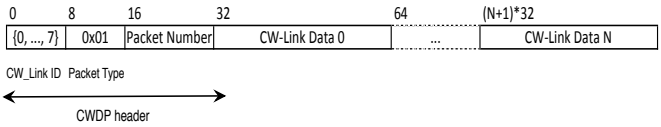


Figure 6. CWDP CORE DATA packet

ACK (packet type 0x02) This packet type is used to acknowledge received packets. In Figure 7, the structure of a CWDP ACK packet is shown. Received CWDP packets, except for the ACK packet itself, trigger a reply with an ACK packet. The packet number of the transmitted ACK matches the packet number of the received packet when a CORE DATA packet is received. If the received packet is a SET_CORE_ACCESS packet then the packet number of the ACK packet is set to zero.

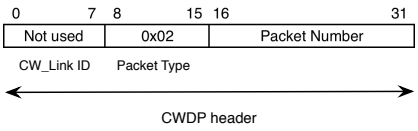


Figure 7. CWDP ACK packet

2.5. Transport Layer

The FaceWorks implementation of the transport layer is composed of the following blocks, the UDP RX and TX blocks, the CWDP RX and TX blocks and the CW-Link RX and TX blocks. An interaction diagram is shown in Figure 8.

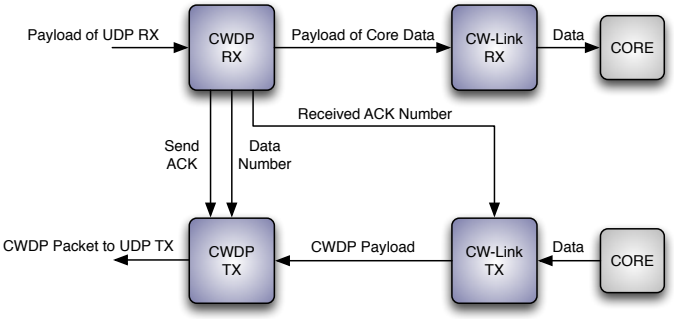


Figure 8. CWDP Layer

The UDP RX block filters out UDP messages that are not being sent to the UDP port being used for CWDP communication. For messages addressed to the UDP port being used for CWDP communication, the UDP RX block removes the UDP header and forwards the UDP payload (CWDP packet) to the CWDP RX module.

The UDP TX block inserts the UDP header in the RAM TX buffer, calculates the checksum assuming the pseudo-header is attached upfront, and forwards the transmission of the UDP packet to the IP TX module.

The CWDP RX block implements the CWDP protocol for the received packets. It decodes the type of the received packet and performs the instructed action. The payload of CORE_DATA packets is placed in the RAM RX buffer. After the reception of a CWDP packet it requests the CWDP TX to send the acknowledge packet. If the CWDP packet is a CORE DATA packet it forwards the data to CW-Link RX block in order to be delivered to the addressed IP Core.

The CWDP TX block receives data from the CW-Link TX block and sends the data to the UDP TX module by means of the RAM TX buffer. Then it waits for the ACK packet of the sent packet, which will arrive in the CWDP RX block. When the ACK packet arrives, the CWDP RX block presents the sequence number to the CWDP TX block, so it can verify that it matches the sequence number of the packet previously sent.

3. Design and Verification

In this section a two-port FaceWorks implementation is presented, as well as the methodology used to debug and verify this design.

3.1. New Hardware Implementation

The new architecture is presented in Figure 9. During the development a loopback has been used, where the CW-Link RX is directly connected to CW-Link TX. With this test case, no IP cores other than FaceWorks are needed.

Compared to the original architecture, the CW-Link modules are duplicated on both the reception and transmission sides. A signal generated from the destination UDP destination port field is used to decide which of CW-Link RX unit is active. An arbiter is added to decide which of the CW-Link TX is granted access to the medium. The RAM_RX block is also duplicated in order to receive and store two consecutive packets with different destination ports.

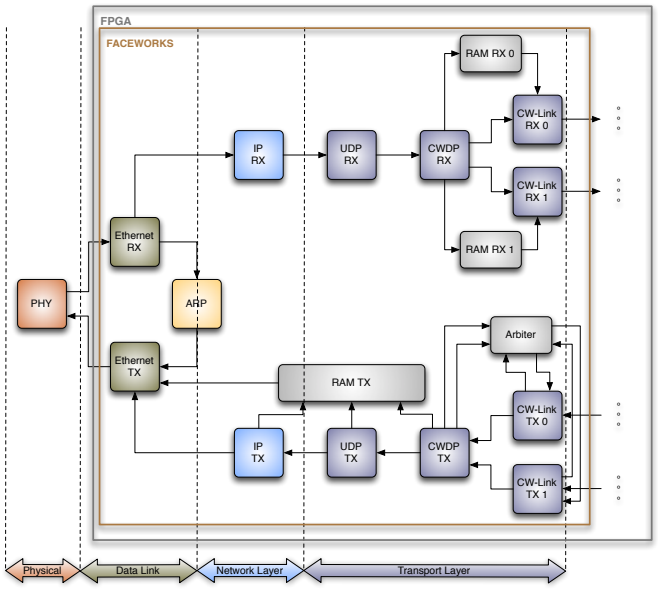


Figure 9. Two-port FaceWorks Block Diagram

In this implementation FaceWorks supports a single sender IP address and listens to two hardwired UDP ports, whose numbers differ by 1. For example, ports 1234 and 1235.

3.2. Verification

Initially, verification of the design was being performed directly on the FPGA using as main tools for debug a packet sniffer (Wireshark), an ILA (Xilinx's ChipScope Pro) and the Face-Test application to send stimuli to the FPGA board and analyze the responses. However, as the design got near completion, some intermittent bugs showed up, which were very difficult to identify using this verification method.

The main difficulty was to know when and what signal was the causing a problem, so that correct trigger conditions could be set. Using the ILA proved ineffective due to too many inconclusive trigger conditions, and a limited buffer size to store data samples for analysis, which is the result of having a limited number of BRAM in the device.

An alternative consisted in creating a testbench and running the system into a simulator such as Xilinx's iSim. However, it was difficult to recreate realistic test conditions, unless a significant amount of C and Verilog code were developed. Another way is to modify the system in order to be able to run a simpler to write testbench, but this approach is risky as important features may end up untested.

Besides, methods for the testbench to read and write data are limited to file IO since no Verilog Procedural Interface (VPI) support is offered for iSim. Reading files with data to input to the MII interface and writing files with data output from the same interface is hard and complicated to implement in a testbench, considering the sheer amount of data needed to adequately test a network interface. So, other alternatives were searched so that the design could be properly debugged and verified. Of the considered alternatives, Verilator, a free open source application, has been chosen to perform this task.

Verilator is an application that translates a synthesizable Verilog description into an optimized cycle-accurate behavioral model in C++/SystemC, called a verilated file. Verilator is a two state simulator (0,1), but it has features to deal with unknown states. The testbench is a C/C++ application that wraps the verilated model and is compiled with it using a common C++ compiler. Verilated models show high simulation speed, which is on par or higher than commercial simulators such as NC-Verilog, VCS and others. [6].

Figure 10 shows the verification environment for FaceWorks, encompassing the Face-test application and the C++ testbench.

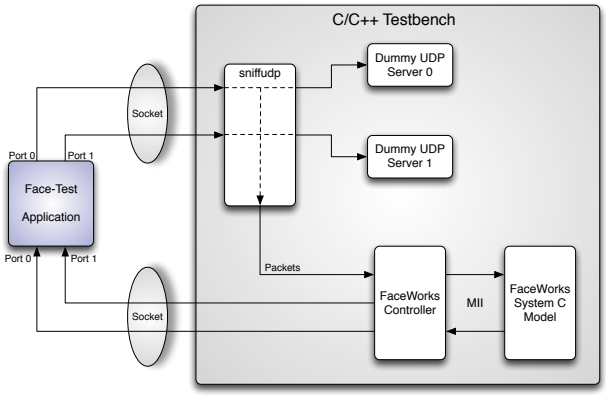


Figure 10. Verification Environment

 ISBN: 978-989-98875-1-0

3.3. Face-Test Application

The FaceWorks test application (Face-Test) is a C language application that implements the CWDP protocol. The application has been tested on the pre-existing FaceWorks architecture and on the new FaceWorks architecture.

The two basic functions used in the Face-Test application are the `CWDP_receive_packet()` and the `CWDP_send_packet()` functions. These functions create an abstraction layer which hides the CWDP details. They are used for sending / receiving packets to / from FaceWorks, and can be called for different sessions/connections inside a user process.

3.4. Testbench

Using Verilator, a FaceWorks simulator has been created which can be stimulated with the same socket-based test application used for exercising FaceWorks in an FPGA. In this way test patterns that fail in the FPGA device can be replicated and analyzed in simulation with full visibility over all design signals. A FaceWorks SystemC model is created by running Verilator on the FaceWorks Verilog code, and this object is then instantiated in the testbench application.

The testbench has five components: `Dummy UDP Server 0`, `Dummy UDP Server 1` and `sniffudp`, the FaceWorks Controller, and the FaceWorks model.

3.4.1. Dummy UDP Servers

The two threads `Dummy UDP Server 0` and `Dummy UDP Server 1` perform the simple operation of reading the UDP ports persistently, to prevent the input data coming from the application from filling the OS socket buffer.

3.4.2. sniffudp

The `sniffudp` thread launches the data capture code, which is based on the sniffex.c code example of using `libcap`, a portable C/C++ library for network traffic capture [7].

A simpler solution to capture data from the Face-Test application would be to implement a C/C++ UDP socket server in the testbench to read the incoming data from the Face-Test application and provide it to the FaceWorks SystemC model. However, since the FaceWorks connection to the outside is the MII interface, one would need to recreate the MII packet from the received UDP payload. Recreating the MII packet implies recreating the UDP packet followed by the IP packet, followed the Ethernet frame. This would be unnecessarily complicated. Using the `libpcap` library, one can capture the packets at the data link layer and avoid the process of recreating the Ethernet, IP and UDP packets.

The `sniffudp` function is set to capture data from the Linux loopback (`lo`) device, which is addressed with the two FaceWorks UDP ports. The `lo` device is a virtual network interface used to receive packets sent from the host to itself. After the setup, the `sniffudp` thread is waiting for filtered packets. When such packets arrive they are passed to the callback function `got_packet()`.

The `got_packet()` function receives almost complete Ethernet frames (Figure 2) which miss the FCS field and have a blank destination and source address, since the data is traveling through the loopback interface. The preamble, SFD, destination address, source address and Ethertype fields are inserted. The data field (containing the IP packet) is copied from the sniffed packet. The FCS field contains the CRC computed by both the `process_recv_packet` and the `got_packet` function. The source code used to calculate the CRC has been generated by a Python script denominated `pycrc`[8]. The `pycrc` script outputs a C header file (.h) and a C source code file (.c) which can compute the CRC.

Finally, the `got_packet()` function reaches a critical section of the code where it pushes the packet into a queue shared with the main thread, after which it terminates execution. When another packet is filtered the callback function is called again and the process repeats all over again.

3.4.3. FaceWorks Controller and Model

The FaceWorks Controller is responsible for receiving data from the `sniffudp` thread, formatting the data according to the MII format and deliver them to the FaceWorks MII interface. The FaceWorks Controller is also responsible for unpacking the data it receives from the FaceWorks model in order do send them to the test application via UDP sockets.

The FaceWorks model is a SystemC Model created by running Verilator on the FaceWorks Verilog code and communicates with the FaceWorks Controller.

Since the FaceWorks Controller does not implement the ARP protocol, to properly support the communication between the Face-Test application and the FaceWorks model, the FaceWorks controller starts by sending a fixed ARP reply packet as shown in Figure 11. This packet is sent even without receiving any ARP request from the FaceWorks model. The purpose is to fill the ARP cache with an IP / MAC address pair, so the FaceWorks Ethernet TX module can obtain the MAC address by consulting the ARP module. If the cache is already stuffed, no ARP query packets will be generated by the FaceWorks model.

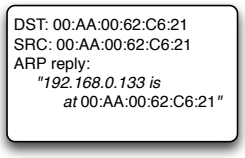


Figure 11. Injected ARP Reply Packet

After the ARP stuffing is completed, the FaceWorks controller is in a loop, trying to transfer data from the packet queue, shared with the `sniffudp` thread, to the MII RX interface of the FaceWorks model, and/or to transfer data from the MII TX interface to the socket that leads to the Face-Test application.

To transfer data from the queue to the MII RX interface the FaceWorks controller needs to get hold of the mutex that protects the shared packet queue. When access to the mutex is granted and at least one packet exists in the queue, the packet is popped from the queue and fed into the MII RX interface of the FaceWorks model, nibble by nibble.

To transfer data to the socket leading to Face-Test, the FaceWorks controller stores the nibbles received from the FaceWorks model in a buffer, which is then passed to another function, the `process_recv_packet` function.

The `process_recv_packet` function receives as argument a packet buffer from the FaceWorks controller, processes the packet and sends it over the network socket to the Face-Test application. The processing consists in checking if the Preamble, SFD and Ethernet headers are correctly constructed, by comparing their values to the expected values. The FCS field is calculated for the received packet and compared with the FCS value in the packet itself. The FCS field contains the CRC checksum computed as described in section 3.4.2. If any field does not match the expected value, the function returns with an error which indicates the malformed field. If a well formed packet is received, the IP headers and UDP headers are read to extract the destination IP address, UDP port and UDP payload offset, and the UDP packet is sent to the Face-Test Application. By verifying the data from the MII interface, the operation of the FaceWorks model is always under scrutiny, which is a valuable debug feature. Next, the Ethernet, IP and UDP headers are removed, and the data is placed into the UDP socket to be sent to the Face-Test application, which is listening to the UDP ports in use.

4. Results

In this section two kinds of results are presented: 1) FPGA implementation results and 2) bandwidth results.

4.1. FPGA Implementation Results

The system has been developed and tested on a Xilinx SP605 board, featuring a Spartan 6 XC6SLX45T device and a Marvell Alaska PHY (88E1111) chip. The Spartan 6 is a low end device, designed for larger production volumes and low price but with limited performance.

Table 1. Results for the Single-Port FaceWorks

	Used	Total	Percentage Used
Slice Logic Utilization			
Number of Registers	1527	54576	2%
Number of LUTs:	2178	27288	7%
Number used as Logic	2036	27288	7 %
Number used as Memory	16	6048	1 %
Specific Feature Utilization			
Number of BRAM (16Kb)	2	116	2%

The synthesis results presented are obtained with the Xilinx ISE 14.4 suite of tools. In Table 1, synthesis results for the original single port implementation are presented. Table 2 presents synthesis results for the two-port implementation designed in this work.

Table 2. Results for the Two-Port FaceWorks

	Used	Total	Percentage Used
Slice Logic Utilization			
Number of Registers	1917	54576	3%
Number of LUTs:	2965	27288	10%
Number used as Logic	2933	27288	10 %
Number used as Memory	32	6048	1 %
Specific Feature Utilization			
Number of BRAM (16Kb)	3	116	2%

The synthesis results show that both FaceWorks designs are very economic in terms of size. In terms of logic resources, implemented with Look Up Tables (LUTs), the two-port design is about 40% larger than the single-port design but it supports 16 CW-Link connections whereas the single-port design only supports 8 CW-Link connections. The IP cores use little internal memory, implemented with Block RAMs (BRAMs), and zero DSP blocks. The overall size can be considered adequate for a test and debug core. For a small FPGA like the Spartan 6 XC6SLX45T device, it only occupies 10 % of the logic resources in the two-port configuration.

Table 3. Period Timing Constraints

	Used	Minimum
Sys_clk	20 ns	8.341 ns
TX_CLK	40 ns	NA
RX_CLK	40 ns	NA

The constraints for the system clock (`Sys_Clk`) and the PHY receive and transmit clocks (`RX_CLK` and `TX_CLK`) are shown in Table 3 for the two-port design. The minimum `Sys_Clk` period that has been possible to meet (8.341 ns) corresponds to a frequency of about 120MHz which is a competitive frequency for a Spartan 6 device. Compared to the single port implementation, where the minimum clock period is 6.912 ns, the maximum frequency for the two-port design is 20% lower than the single-port implementation. This is mostly due to the Arbiter block design, where the operation frequency scales poorly in the number of ports. A better design would not be very difficult to implement but falls out of the scope of this work. However, for a number of ports higher than two, it is highly recommended that the Arbiter design is reviewed.

4.2. Ethernet Network Characterization

The results presented in this section have been obtained using two notebooks featuring an Intel U4100 @ 1.30GHz processor, using 3 GB of RAM and an Atheros AR8131 Ethernet adapter. The Linux kernel 3.2.0-x86-64 has been used. To interconnect the two notebooks, a TP-Link TL-WR841N 10/100 Mbps switch has been used.

Two tests have been performed to the determine the net-

work capabilities, and detect if it can limit the FaceWorks performance.

The first test was designed to determine the maximum throughput achievable with the switch. This test consisted in exchanging data in both directions, simultaneously, between two notebooks interconnected via the switch.

The throughput values presented in Table 4 represent raw values obtained using a random stream, without sending acknowledge packets, and without any data dependencies on the data sent by the other peer. Throughput values are measured for upstream, downstream and aggregate (sum of upstream and downstream). In the remainder of this work, the aggregate throughput obtained is used as a reference.

Table 4. TL-WR841N 10/100 Mbps Maximum Raw Throughput

	Downstream	Upstream	Aggregate
Throughput	94.54 Mbps	95.88 Mbps	190.42 Mbps

These results show that in practice a throughput close to the nominal network speed has been achieved. One can say the network setup introduces a 5% degradation compared to ideal conditions.

The second test consisted in measuring the effective throughput for various packet sizes. Using the `ping` application 10 packets were sent with three distinct data sizes, 54 bytes, 720 bytes and 1440 bytes. The RTT values as reported by the `ping` application and effective throughput for 3 packet sizes are presented in Table 5, where the aggregate throughput is calculated using the following expression:

$$Throughput = \frac{2 \times PacketSize \times 8}{RTT.10^6}[Mbps] \qquad (1)$$

Table 5. TL-WR841N 10/100 Mbps Latency and Throughput

Packet Size	Latency	Throughput	Utilization
54 bytes	0.593 ms	1.51 Mbps	<1%
720 bytes	0.84 ms	13.71 Mbps	7.2%
1440 bytes	1.107 ms	20.81 Mbps	10.9%

These results show that the network latency is the main factor limiting the system performance. The data is packetized and a latency penalty is incurred for each packet sent. Naturally, the penalty decreases with the packet size, but even for the maximum packet size allowed by the CWDP protocol (1440 bytes), the obtained practical upper bound for the aggregate throughput (21 Mbps) is only about 11% of the maximum possible throughput.

4.3. FaceWorks Sustained Throughput

To measure the FaceWorks sustained[1] throughput, the TP-Link TL-WR841N 10/100 Mbps switch has been used to interconnect one notebook and the FPGA board containing the FaceWorks core. In the FPGA, the CW-link RX-TX loopback was in place, so that the tests could be done using FaceWorks alone. Table 6 summarizes the results obtained.

Table 6. FaceWorks Throughput Results

	Single Port	Dual Port	Gain
Old FaceWorks	9.86 Mbps	-	-
New FaceWorks	11.02 Mbps	14.71 Mbps	33%
Verilator Model	42.27 Mbps	79.59 Mbps	88%

Both the old and new FaceWorks implementations have been tested using the Face-Test application. The new version is significantly faster than the original FaceWorks implementation in VHDL. Since the algorithms are the same, the suspicion is that the previous version may have been dropping packets, due to the presence of problems detected by the Verilator linter, which have been fixed in this new and cleaned up Verilog version.

The two-port FaceWorks core has been tested using two Face-Test applications, one for each UDP port. As can be seen, the two-port FaceWorks makes a better use of the available bandwidth when compared to the single-port implementation. Due to the independence of the two channels, this gain roughly corresponds to doubling the packet size.

Using the cycle accurate Verilator simulator, about 42 Mbps for a single-port core and 80Mbps for a two-port core have been obtained. Note that, in hardware simulation all latencies are zero (network, switch, host and Operating System (OS)), except for the FaceWorks and the MAC protocol small hardware latencies. Thus, the simulation results can be taken as a reference for when the latency is almost null. The fact that the practical results differ so much from the simulation results clearly demonstrates the dramatic effect of system latency on the overall throughput.

5. Conclusion

The design of a new two-port FaceWorks architecture, the main objective of this work, has been fully accomplished, including its verification and experimental characterization.

The UDP and CWDP hardware modules have been modified to work with two ports. The number of CW-Link interfaces has been duplicated, in order to keep the same number of debug interfaces per UDP port as before. FaceWorks has been rewritten in Verilog from a previous VHDL language version. Besides being a lot more common in the industry, the use of Verilog is mandatory for the Verilator

[1] In networking the term *sustained* is used for results obtained by averaging over long periods of time.

simulator, a key tool which made possible the completion of this project in time.

Verilator provides a cutting-edge approach to verification compared to using a standard Verilog simulator. The Verilator tool has been used to create a SystemC model of the FaceWorks design, which has been embedded in a C/C++ testbench. This solves the classical difficulty of integrating software and hardware in a simulation environment. Other advantages of using Verilator are the detailed linting it effects on the Verilog code, the exhaustive dumping of wave traces for debugging purposes, and the detailed coverage metrics it extracts from the design simulation.

The design has been synthesized and tested in a Xilinx Spartan 6 FPGA. The implementation proved small and fast enough for a debug core, as such core should not take too much space in the overall system design.

The experiments performed aimed at obtaining throughput values for the new FaceWorks implementation and comparing them with the previous single-port implementation.

First the network used for testing has been characterized in order to have practical upper bounds for bandwidth and throughput. The bandwidth was found to be close to the nominal 100 Mbps, while the thoughput varied significantly with the packet size, which illustrates the strong penalty introduced by the system latency.

Then the FaceWorks cores have been tested both in simulation and in the actual network formed by a host notebook, an Ethernet switch and the FPGA board.

The simulation results illustrate the situation when the system latency is close to zero as the simulated hardware latencies are very small compared with the host or switch latencies. About 40% network utilization has been obtained for the single-port FaceWorks and 80% network utilization has been obtained for the two-port FaceWorks. The degradation compared to the available bandwidth of 100 Mbps is due to the handshaking needed for the CWDP protocol.

When the same tests are performed on the actual network only 11% network utilization is obtained for the single-port FaceWorks and 15% network utilization has been obtained for the two-port FaceWorks. These results clearly show the dramatic effect of system latency on the available throughput.

The achievable throughput improves considerably when more ports are added, since a packet for one port can be sent without waiting for the acknowledge of the packet sent to another port. The two-port implementation results show more than 50% improvement on the throughput compared to the single port implementation.

5.1. Future work

To improve the FaceWorks throughput one obvious solution is upgrading it to use a 1 Gbps PHY chip. This would allow for a lower latency because of a higher transmission rate and a higher MTU with the use of Jumbo frames. However, implementing support for Jumbo frames would require larger amounts of on-chip RAM, and this could be a restriction depending on the target FPGA device.

An alternative solution consists in not sending acknowledge packets; if some packet is lost or received out-of-order then the retransmission of that packet only is requested. This solution is based on the assumption that most packets sent from the PC to FaceWorks and vice-versa will be delivered and arrive in order.

Another improvement that can be done in the future is to eliminate the FaceWorks core from the simulation environment, as it is only needed to connect PC and FPGA board. In fact, if the objective is to the debug the IP core, simulating FaceWorks as part of the whole system may be an unnecessary burden. This scheme is presented in Figure 12, where the FaceWorks model has been replaced with a simpler UDP/CWDP server written in C++.

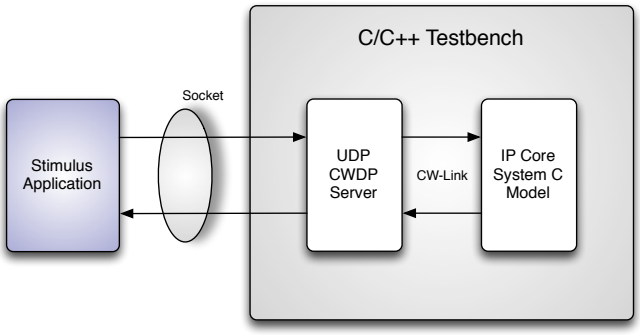


Figure 12. A UDP/CWDP cycle accurate simulator

References

[1] José de Sousa, Nuno Lourenço, Nelson Ribeiro, Victor Martins, and Ricardo Martins. Network core access architecture. http://www.google.com/patents/US8019832, May 2008.

[2] José de Sousa, Nuno Lourenço, Nelson Ribeiro, Victor Martins, and Ricardo Martins. Network core access architecture. http://www.google.com/patents/EP2003571A2, December 2008.

[3] IEEE Standards Association. *IEEE Standard for Ethernet*, chapter Section 1, Chapter 3 Media Access Control (MAC) frame and packet specifications. IEEE, August 2012.

[4] IEEE Standards Association. *IEEE Standard for Ethernet*, chapter Section 2, Chapter 22 Reconciliation Sublayer (RS) and Media Independent Interface (MII). IEEE, August 2012.

[5] Coreworks. *FaceWorks CWnet01 Datasheet, Multi-purpose Autonomous Network Interface Core*. Coreworks, January 2008.

[6] Wilson Snyder. Verilog simulator benchmarks. http://www.veripool.org/wiki/veripool/Verilog_Simulator_Benchmarks, March 2011.

[7] Tim Carstens. Sniffex, sniffer example of tcp/ip packet capture using libpcap. http://www.tcpdump.org/sniffex.c, June 2013.

[8] Thomas Pircher. Cyclic redundancy check (crc) calculator and c source code generator. http://www.tty1.net/pycrc/index_en.html, April 2013.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under project PEst-OE/EEI/LA0021/2013

Serial Peripheral Interface (SPI) Master Evaluation in FPGA for ATLAS TileCalorimeter High Voltage Control System

J. Alves[1,2], G. Evans [2,3], J. Soares Augusto[2,4], J. Silva[1], L. Gurriana[1], A. Gomes[1,2]
jalves@lip.pt, gevans@fc.ul.pt, jasa@fisica.fc.ul.pt, jmanuel@lip.pt, gurriana@lip.pt, agomes@lip.pt

1 - Laboratório de Instrumentaçãoe Física Experimental de Partículas
2 - Faculdade de Ciências da Universidade de Lisboa, Departamento de Física
3 - Centro de Física da Matéria Condensada (CFMC)
4 - Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa (INESC-ID)

Abstract

The Tile Calorimeter (TileCal) is a sub-detector of the ATLAS detector of CERN. It provides accurate measurement of the energy of jets of particles resulting from proton-proton collisions that occur in the LHC (Large Hadron Collider) experiments, measurement of Missing Transverse Energy and identification of low-pt muons. Particles that cross the TileCal produce photons in the scintillators, which are absorbed and carried by Wavelength Shifting Fibers until they feed a set of photomultipliers tubes (PMTs). The PMTs convert photons into electrical signals which drive the TileCal front-end electronics. A High Voltage (HV) distributor system is used to power the PMTs. A new version of a high voltage control board (HV_opto) is being developed. The controlling and monitoring of the high voltages' system will be implemented in a FPGA which communicates with the HV_opto through a Serial Peripheral Interface (SPI). A SPI Master module is under preliminary evaluation in order to be used in this new high voltage control board. The SPI module is the subject of this article.

1. Introduction

The LHC [1] collider, at CERN, is a circular structure with 27 km of perimeter. The main goal of LHC is to collide two beams of protons or heavy ions travelling in opposite directions. The particle beams travel inside the ring of the accelerator tube in which vacuum is established. The protons are accelerated and kept in the centre of the tubes by a magnetic field generated by powerful superconducting magnets, cooled by a suitable cryogenic system. The collisions of protons occur at key points - the locations of the LHC detectors. The protons travel in bunches that intersect in the key points with a frequency of 40 MHz. The ATLAS is one of the detectors installed at the LHC.

The HV system is one of the components of TileCal, which is being upgraded. In this new HV system, a reconfigurable device (a Kintex-7 FPGA) hosts the firmware used to access and control the new high voltage control board which provides high voltage to the PMTs. The communication between the FPGA and the high voltage control board is performed through SPI protocol. An SPI master interface in the FPGA is now being tested, with the new high voltage control board tests in sight.

This paper is structured in 5 sections. In section 2 is presented the configuration of the new TileCal HV system. Section 3 is dedicated to the description of SPI protocol. The description of the SPI master interface which is to be tested is presented in section 4. Finally, in section 5 are the final considerations and conclusions from this work.

2. The New TileCal HV System

The high voltage has strong influence in the detector performance, so a stable HV system is fundamental for the good behaviour of the PMTs. Fig. 1 shows the new configuration for supplying, controlling and monitoring the high voltage applied to the PMTs. The *HV_opto* is a new version of the high voltage control board, used to control the value of the high voltage that is supplied to each PMT. The *HV_Opto* is divided into lengthwise independent halves, but the HV input is common to them [2]. Each half serves 6 PMTs, so an *HV_opto* board has 12 HV output channels. It produces an output voltage derived from the input HV, with the output proportional to a DAC (Digital-to-Analog Converter) voltage, up to a maximum which is the value of the input HV. The output HV is monitored by an ADC (Analog-to-Digital Converter), using a voltage divider to step the voltage down to a level that can be

processed by the low voltage circuits [2]. The digital interface to the ADC and to the DAC uses an SPI slave interface.

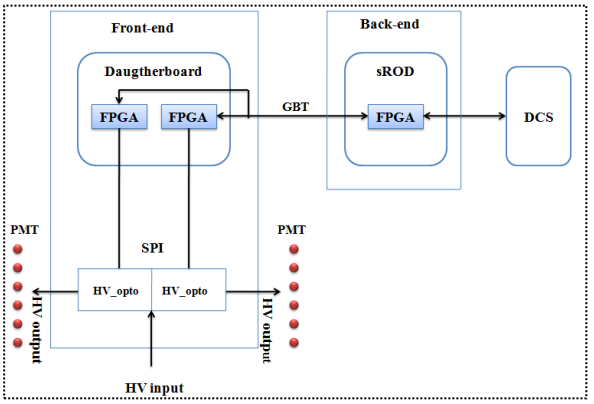


Figure 1. New configuration for TileCal HV system.

In the front-end Daughterboard FPGAs (Kintex-7 from Xilinx) will be implemented a SPI master interface (to communicate with the *HV_opto* board) and firmware to process commands from sROD (super-Read Out Drivers) in the back-end: a *Write* command to a DAC on a HV channel, to change the output high voltage for that channel; or a *Read* command to read back a digitized value of the current high voltage [2]. A SPI master interface from Xilinx, which is the subject of this paper, is under evaluation to be used in the *HV_opto* tests, and finally it could be implemented in the front-end Daughterboard FPGAs. Aside of the SPI evaluation, the firmware to process commands from the sROD and communicate (using SPI) with the *HV_opto* is being developed.

The Detector Control System (DCS) is responsible for the control and monitoring the HV system. The sROD FPGA serves as data/command router between the DCS and the Daughterboard FPGAs, and handles communications with the DCS computers.

3. Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a full duplex, synchronous, serial data link that is standard for many microprocessors, microcontrollers, and peripherals. It allows communication between microprocessors and peripherals and/or inter-processor communication [3]. The devices communicates in a master/slave configuration and during a SPI transfer, data is simultaneously transmitted (shifted out, serially) and received (shifted in, serially). A serial clock line synchronizes shifting and sampling of the information on two serial lines. A slave select line allows individual selection of a slave SPI device. The SPI bus consists of four wires (Fig. 2): *Serial Clock* (SCLK), *Master Slave In* (MOSI), *Master In Slave Out* (MISO), and *Slave Select* $\overline{SS}$, which carry information between the devices connected to the bus [4].

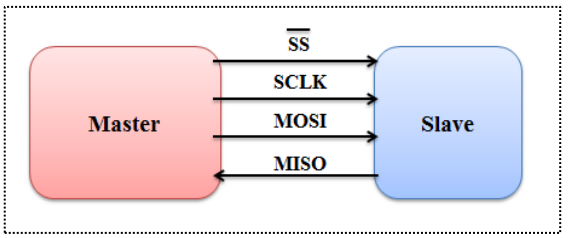


Figure 2. SPI bus.

The *MOSI* data line is the output data from the master which is shifted as the input data to the selected slave. The *MISO* data line is the output data from the selected slave to the input of the master. There may be no more than one slave transmitting data during a particular transfer [3], [4].

The *SCLK* is a serial clock which is provided by the SPI master and regulates the flow of bits. The *SCLK* line transitions once for each bit in the transmitted data. When the master initiates a transfer, a number of clock cycles equal to the number of the bits to be transmitted are automatically generated on the *SCLK* pin. *SCLK* pin is an input to the slave and synchronizes the data transfer between the master and slave devices [3], [4].

The $\overline{SS}$ control line is used to select a particular slave allowing individual selection of a slave SPI device. The slave devices not selected do not interfere with SPI bus activities, by ignoring the *SCLK* signal and keep the *MISO* output pin in a high impedance state unless its slave select pin is active low [3],[4].

The SPI specification allows to the user to select different modes of operation accordingly with the clock phase and polarity. It is possible four combinations of clock phase and polarity. The clock polarity is specified by a control signal (1 bit) normally named CPOL, which selects an active high or active low clock and has no significant effect on the transfer format. The clock phase is also specified by a control signal (1 bit) named CPHA, which selects one of two fundamentally different transfer formats. If *CPHA = '0'*, data is valid on the first *SCLK* edge (rising or falling) after $\overline{SS}$ has asserted. If *CPHA = '1'*, data is valid on the second *SCLK* edge (rising or falling) after $\overline{SS}$ has asserted. The clock phase and polarity should be identical for the master device and for the slave device [3], [4].

4. SPI Master Evaluation

We have evaluated a SPI master interface in a Virtex-6 FPGA hosted in a Xilinx ML605

 ISBN: 978-989-98875-1-0

evaluation board. We started from a provided SPI master interface from Xilinx, described in VHDL[1], and apt to be used in a Xilinx CoolRunner-II CPLD (Complex Programmable Logic Devices) [3] in order to interface microprocessors. This SPI master interface was prepared for 8-bit data transfers, so an update of the code to handle 16-bit data transfer was performed, because the *HV_opto* uses ADCs and DACs which requires 16 bit data transfers. Fig. 3 shows the block diagram [3] of the SPI master interface to be tested in FPGA, in the future, with high voltage board. In the following, it is presented a description of the functionality of each of these blocks.

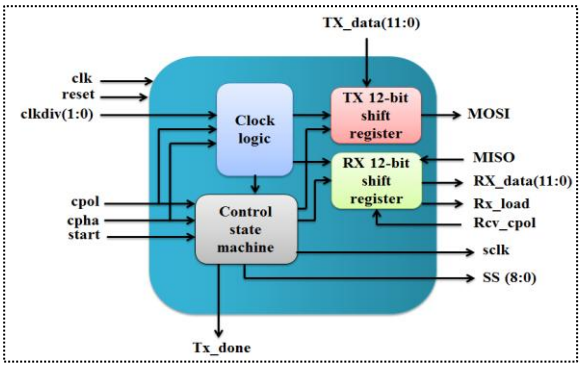


Figure 3. Block diagram SPI master interface.

Clock logic block: is responsible to generate the output serial clock *(sclk)* based on the values of the *clkdiv*, *cpha* and *cpol* inputs. The *clkdiv* input set the frequency of the serial clock, dividing down the input clock *(clk)* frequency by 4, 8, 16, or 32, as shown in table 1 [3].

clkdiv setting	Frequency of sclk
00	1/4 of clk
01	1/8 of clk
10	1/16 of clk
11	1/32 of clk

Table 1. Serial clock settings.

The *cpha* sets the phase of the serial clock in relation to the serial data. If *cpha ='1'*, data is valid on first *sclk* edge (rising or falling) after slave select has asserted, while if *cpha = '0'*, data is valid on second *slck* edge (rising or falling) after slave select has asserted. The *cpol* bit sets the clock polarity of the serial clock. If *cpol = '0'*, *sclk* is low when it is in an idle state, while if *cpol = '1'*, *sclk* is high when it is in an idle state [3].

TX 16-bit shift register: it is a 16-bit shift register in which data can be loaded and transmitted. Each bit is shifting out through *MOSI* line in each serial clock cycle. The *Tx_data* is the input to load the 16-bit data.

Control state machine: this block is responsible to control the shifting and loading process of the 16-bit transmitter shift register, and it generates the signals for slave's selection *(SS)*. It also monitors the transfer process to determine when a 16-bit data transfer is complete. In this case, it outputs a flag signal, the *Tx_done* signal, which goes high when a transfer is complete. The state machine implemented in this block remains in the IDLE state until the *start* bit is asserted. This state machine also control when the serial clock is output to the SPI bus.

RX 16-bit shift register: it is used to receive the data sent by slave devices through the *MISO* line. The *Rcv_cpol* input allows specifying which edge of the external *sclk* the incoming *MISO* data is sampled on. This allows flexibility in dealing with all types of different slave devices, as some SPI slaves clock data out on the rising edge of *sclk*, while others clock data out on the falling edge of *sclk*. When a receiving process is complete, it outputs a flag signal, *Rx_load*, which goes high when all bits sent by a slave is received; at this time the received data is available in the 16 bit output, the *RX_data* line.

The hardware resources in the Virtex-6 FPGA are presented in table 2. The percentage of resources of each one of the resources category is about 1 %. We use an output SPI clock frequency of 2.5 MHz, which is the recommended frequency for the *HV_opto* board [2]. The input clock, *clk* was used with a frequency of 10 MHz, well below the maximum frequency of 415.45 MHz allowable for this module (post-synthesis value given by the Xilinx development tool).

Device Utilization Summary		
Slice Logic Utilization:		
Number of Slice Registers:	660 out of 301,440	1%
Number used as Flip Flops:	659	
Number used as Latches:	1	
Number used as Latch-thrus:	0	
Number used as AND/OR logics:	0	
Number of Slice LUTs:	586 out of 150,720	1%
Number used as logic:	283 out of 150,720	1%
Number using O6 output only:	193	
Number using O5 output only:	53	
Number using O5 and O6:	37	
Number used as ROM:	0	
Number used as Memory:	169 out of 58,400	1%
Number used as Dual Port RAM:	0	
Number used as Single Port RAM:	0	
Number used as Shift Register:	169	
Number using O6 output only:	138	
Number using O5 output only:	2	
Number using O5 and O6:	29	
Number used exclusively as route-thrus:	134	
Number with same-slice register load:	127	
Number with same-slice carry load:	7	
Number with other load:	0	

Table 2. Device utilization on the Virtex-6 FPGA.

The next step of this work will be testing this SPI master interface with the *HV_opto* board. The configuration for the next step is presented in Fig. 4. The computer (PC) replaces the sROD/DCS system,

[1]*VHDL - VHSIC (Very High Speed Integrated Circuits) Hardware Description Language*

sending *Write/Read* commands to the control module. The control module processes these commands and transmits them to the *HV_opto* DAC/ADC using the SPI Master interface under evaluation.

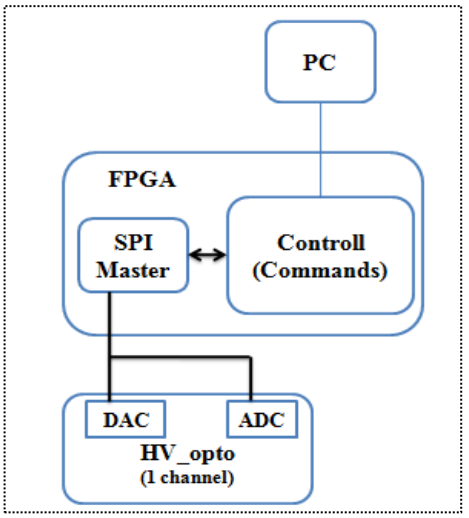


Figure 4: Configuration for the next step of this work - testing the SPI, the Control module and the *HV_opto* integrated in one system.

When integrating the SPI master interface in this system, it must to be adapted to the serial specifications of the DAC/ADC implemented in the *HV_opto*. As an example the setup time of the slave select ($\overline{SS}$) line must be checked. This SPI interface asserts the $\overline{SS}$, 1 *SCLK* clock cycle before the first *SCLK* edge, meeting the $\overline{SS}$ setup time requirement of most SPI slave devices. This timing parameter must be adapted for the target system, as this setup time requirement varies between different SPI slaves [3]. As it was apt to interface microcontrollers, it must also be adapted to interface the control module responsible to process commands from the sROD.

When all the adaptations were performed, we could test all the integrated system, checking the behaviour of the SPI master interface, the control module firmware and the *HV_opto* itself.

5. Conclusion

A SPI master interface is under preliminary evaluation in a Virtex-6 FPGA and will be tested with an ADC or a DAC slave device. We study an SPI master interface provided by Xilinx and ready to be used in one of their CPLD devices, but our target is the implementation in a FPGA, in this case a Virtex-6 and a Kintex-7 FPGA which is already implemented in the TileCal front-end electronics. We have upgraded it from an 8-bit SPI master interface to one with 16 bits which is required for the final target, the *HV_opto* board.

A simple loop-back test was performed in FPGA, putting the master working as slave at same time. We could verify a correct transmit and receive operation. But it is essential to test this design with an independent slave device, which is the next step of this work presently under planning. Whenever we have a first version of the new TileCal high voltage control board, this SPI master interface will be used and tested for the target which it was planned for, the *HV_opto* board.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2013 and PTDC/EEAELC/122098/2010.

References

[1] The Large Hadron Collider, The LHC Study Group, CERN/AC/95-05, 1995.https://cds.cern.ch/record/291782/files/cm-p00047618.pdf

[2] Gary Drake, “Specifications for the HV_Opto Control Board V1.0 for the ATLAS TileCal Demonstrator Project”, V1.2, November 11 2013, Argonne National Laboratory-High Energy Physics Division.

[3] Xilinx, CoolRunner-II Serial Peripheral Interface Master XAPP386 (v1.1) November 9, 2009.

[4] Motorola, M68HC11 Microcontrollers Reference Manual – Rev. 1, April 2002.

An FPGA-based Fine-grained Data Synchronization for Pipelining Computing Stages

Ali Azarian, João M. P. Cardoso
Faculdade de Engenharia da Universidade do Porto (FEUP)
INESC TEC, Porto, Portugal
azarian@fe.up.pt, jmpc@acm.org

Abstract

In recent years, there has been increasing interest on using task-level pipelining to accelerate the overall execution of applications mainly consisting of Producer-Consumer tasks. This paper proposes fine-grained data synchronization approaches to achieve pipelining execution of the Producer-Consumer tasks in FPGA-based multicore architectures. Our approach is able to speedup the overall execution of successive, data-dependent tasks, by using multiple cores and specific customization features provided by FPGAs. An important component of our approach is the use of different fine-grained data synchronization approaches to communicate data and to synchronize the cores associated to the Producer-Consumer tasks. All the schemes and optimizations proposed in this paper were evaluated with FPGA implementations. The experimental results show the feasibility of the approach for both in-order and out-of-order Producer-Consumer tasks. Furthermore, the results using our approach to task-level pipelining and a multicore architecture reveal noticeable performance improvements for a number of benchmarks over a single core implementation without using task-level pipelining.

1. Introduction

Recently, there has been growing interest in the use of task-level pipelining to accelerate the overall execution of the applications consisting mainly of *Producer-Consumer (P/C)* tasks. The main idea of task-level pipelining is to partially overlap the execution of data-dependent tasks (computing stages). Many applications, such as image/video processing applications, are structured as a sequence of data-dependent processing stages, use the P/C pair communication paradigm, and are thus amendable to pipelining execution [1, 2].

Task-level pipelining is an important technique for multicore-based systems, especially when dealing with applications consisting of the Producer-Consumer computing stages (see, e.g., [3]). It may provide additional speedups over the ones achieved when exploring other forms of parallelism. In the presence of multicore based systems, task-level pipelining can be achieved by mapping each task to a distinct core and by synchronizing their execution using a handshaking based data communication scheme. In a fine-grained data synchronization scheme, the producer and consumer may overlap significant parts of their execution.

The simplest implementation of task-level pipelining uses a FIFO channel between cores implementing *P/C pairs* (see, e.g., [2, 4]). The FIFO can communicate an array element or a set of array elements to establish data synchronization between the producer and consumer. The FIFO is sufficient when the sequence of producing data is the same than the sequence of consuming data (referred herein as *in-order data communication pattern* or simply in-order). In this case, the data communication between the producer and consumer can be synchronized by using a FIFO is storing one data element in each stage. However, the FIFO may not be sufficient when the sequence of producing data is different with the sequence of consuming data (herein as out-of-order communication pattern or simply *out-of-order*). The most recent studies are based on software fine-grained data communication approaches for task-level pipelining (see, e.g., [2, 4]).

In this paper, we explore different fine-grained data synchronization models implemented in customized multicore architectures for pipelining computing stages. We evaluate our approaches with FPGA implementations and measurements on a set of benchmarks running on an FPGA board. Our approaches provide the ability to pipeline the tasks of in-order and out-of-order communication patterns between P/C pairs in run-time. We compare the execution speedup obtained by our fine-grained approaches to task-level pipelining over the execution of applications in a single core and without using task-level pipelining. The results obtained noticeable performance improvements for a number of benchmarks when considering different fine-grained data communication approaches.

The remainder of this paper is organized as follows. Section 2 presents fine-grained data synchronization approaches for pipelining computing stages. Section 3 presents the experimental results. Section 4 presents an overview of related work. Finally, Section 5 concludes this paper.

2. Fine-grained Data Synchronization Model

For task-level pipelining, the applications are split in sequences of tasks that represent P/C pairs. Decreasing the overall program execution time is achieved by mapping each stage to a distinct core (processor) and by overlap-

ping the execution of computing stages. In the simple case, FIFOs are used to communicate data between the stages. Writes to output arrays are substituted by blocking writes to the FIFO, and reads from the output arrays are substituted by blocking reads from the FIFO. Instead of writing the results of the first stage in an *output* array as in the original code, the producer puts (writes) the output into the FIFO directly. Also, the consumer reads data elements from the FIFO. When considering the in-order data communication pattern between the producer and consumer, the FIFO channel with blocking reads/writes is sufficient to synchronize data communications.

In fine-grained architectures, a single data element pass through an appropriate communication structure (e.g., FIFO-like) between cores. In the context of data communication and synchronization between cores, there are several approaches to overlap some of the execution steps of computing stages (i.e., functions or loops waiting for data may start computing as soon as the required data items are produced in a previous function or by a certain iteration of a previous loop), which are presented in [5, 1].

Although the use of FIFO channels between producers and consumers is an effective solution for in-order P/C pairs, it may not be efficient or practicable for out-of-order P/C pairs. The use of FIFO channels is strictly dependent on the order of the communication pattern between producers and consumers. In this case, FIFOs might not be sufficient to synchronize data communication between producers and consumers and it is necessary to use other data communication mechanisms [4, 6].

In this section, we propose fine-grained data synchronization schemes for pipelining computing stages. In these schemes, we assume that the producer and consumer computing stages process *N* arrays.

2.1. Scheme 1: Baseline

The baseline architecture is a single core with two dependent computing stages executing sequentially. The execution time of this scheme provides a criteria to compare the performance impact of different proposed fine-grained data synchronization approaches.

2.2. Scheme 2 (a): FIFO

In order to pipeline computing stages, the producer and consumer can be implemented as shown in Figure 1. In this scheme, computing stages split into two cores: one core as a producer and the other core as a consumer. The communication component between the producer and consumer can be a simple FIFO. Reading and writing from/into the FIFO are blocked. When the FIFO is full, the producer waits to write into the FIFO. Similarly, when the FIFO is empty, the consumer waits until a data element is written to the FIFO. In this scheme, the producer sends data elements ($d_0, d_1, etc.$) into the FIFO and the consumer reads data from the FIFO as soon as it is available. This scheme is sufficient when the order of consumption is the same than the order of the produced data, i.e., when in presence of in-order data communication.

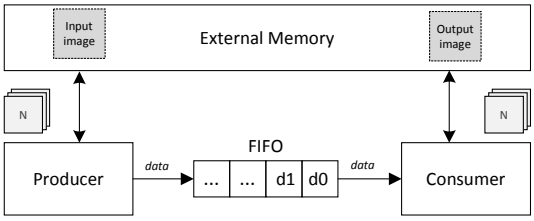


Figure 1. Fine-grained Data Synchronization scheme using a FIFO between the producer and consumer (Scheme 2).

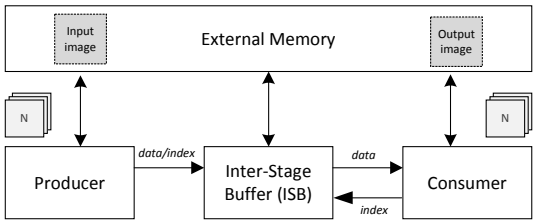


Figure 2. Fine-grained Data Synchronization scheme using an Inter-Stage Buffer (ISB) between the producer and consumer (Scheme 3).

2.3. Scheme 2 (b): FIFO w/ delay in Consumer

This scheme is similar to Scheme 2 (a) but it considers the slowdown of the consumer and this may reduce the rate of external memory accesses by the consumer which in turn may reduce the number of access conflicts.

2.4. Scheme 3: Inter-Stage Buffer (ISB)

As shown in Figure 2, in this scheme, the communication component between the producer and the consumer is an *Inter-Stage Buffer (ISB)* [6]. To provide the task-level pipelining between producers and consumers and to overcome the limitations related to inter-stage communications based on FIFOs, we explore an alternative inter-stage scheme. In this scheme, for each data element being communicated between the producer and the consumer, there is an empty/full flag. Empty/full tagged memories have been described in [7], in the context of shared memory multi-threaded/multi-processor architectures.

The producer is connected to the ISB using one channel responsible for communication between the producer and the ISB. The consumer is connected to the ISB by using two FIFO channels: *Receiving* and *Sending channel*. Our current approach uses non-blocking write over the *Sending channel* of the ISB and the block read from the ISB over the *Receiving channel*. The consumer uses the *Receiving channel* to get data from the ISB. *Sending channel* is used to transmit the requests to the ISB concurrently. Producer and consumers are both connected to an external shared memory.

In scheme 3, for each array element produced, the producer sends its index and value to the ISB (e.g., i and $A[i]$).

 ISBN: 978-989-98875-1-0

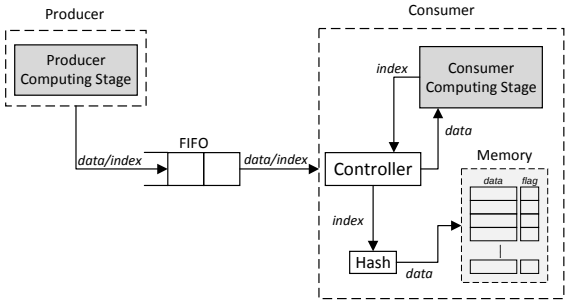


Figure 3. Fine-grained Data Synchronization scheme using a FIFO between the producer and consumer and considering the inter-stage buffer within the consumer (Scheme 4).

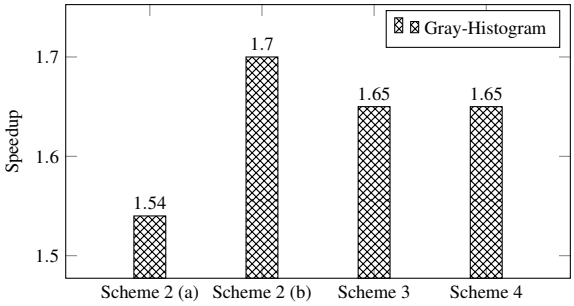


Figure 4. Speedups obtained by considering fine-grained data synchronization schemes with task-level pipelining vs. a single core architecture for in-order benchmarks and $N = 50$.

The ISB receives the index from producer side and maps it to the local memory using a simple hash function (e.g., using a number of least significant bits of the binary representation of the index). The index and value produced are then stored in the ISB local memory location defined by the address given by the hash function [6]. Related to the value stored in the ISB, there is a flag which indicates if a data element was produced and thus can be consumed by the Consumer.

2.5. Scheme 4: FIFO with ISB in Consumer

Figure 3 shows a fine-grained data synchronization scheme which uses a FIFO between the producer and the consumer and includes an inter-stage buffer within the consumer. In this scheme, the producer sends the produced index and data elements through the FIFO.

The consumer sends the requested index to the controller. The controller reads the FIFO and checks if the current read index is equal to the requested index of the consumer. If the indexes are equal, the consumer reads data from the FIFO and send it to the consumer directly. If the read index from the FIFO is not equal to the requested index, the controller maps the index into the on-chip memory (local memory) of the consumer. The local memory structure is based on empty/full flag bit synchronization model. If the read index cannot be stored in local memory, the reading from the FIFO is disabled by the controller. In a similar way, if the requested index of consumer is not equal to the read index from the FIFO and the consumer cannot load the requested index from the local memory, reading the next requested index of the consumer is disabled by the controller.

3. Experimental Results

For evaluating our task-level pipelining approaches, we use a Genesys Virtex-5 XC5LX50T FPGA Development Board. The target architecture was developed with Xilinx EDK 12.3 tools. For the cores of the architecture, we used Xilinx MicroBlaze processors (MB) [8] without caches. Each processor is capable of connecting to on-chip local memory (BRAMs) via local memory bus interfaces, which is the Local Memory Bus (LMB) in the MicroBlaze.

Table 1 presents a set of benchmarks which are used in our experiments. All benchmarks have two dependent computing stages (producer and consumer). As examples of in-order benchmarks, we consider the Gray-Histogram kernel. Also, we used Fast DCT (FDCT), Wavelet transforms, FIR-Edge and Edge-Detection as a set of out-of-order benchmarks. These benchmarks consist of two computing stages (each one with nested loops). The size of the local memory considering the ISB scheme, is defined as $1,024 \times 32$ *bit*.

Table 1. Benchmarks used in the experiments

Benchmarks	Communication Order	Pipelining Stages
Gray-Histogram	in-order	two ($S1$ and $S2$)
Fast DCT (FDCT)	out-of-order	two ($S1$ and $S2$)
Wavelet Transforms	out-of-order	two ($S1$ and $S2$)
FIR-Edge	out-of-order	two ($S1$ and $S2$)
Edge-Detection	out-of-order	two ($S1$ and $S2$)

Figure 4 shows the speedups achieved for in-order benchmark in different fine-grained data synchronization schemes. All the speedups presented are compared to the scheme 1 (baseline architecture). As shown in figure 4, the highest achieved speedup for the Gray-Histogram is $1.70\times$ reported for scheme 2 (b) when the communication component between the producer and consumer is a simple FIFO and with a slowdown of the consumer stage. This result shows that slowing down the consumer can reduce the rate of external memory accesses by the consumer which in turn may reduce the number of access conflicts to the external memory.

Figure 5 shows the speedup obtained by considering fine-grained data synchronization schemes with task-level pipelining vs. a single core architecture for out-of-order benchmarks. Since in fine-grained data synchronization schemes, FIFO channels are not applicable for out-of-order benchmarks, scheme 2 (a) and 2 (b) are not evaluated in this figure. We assume $N = 50$ in all measurements of scheme 3 and 4. As shown in Figure 5, the highest speedup is achieved when we use the ISB between the producer and consumer (scheme 3). Although including the ISB into the consumer may reduce the FPGA resources and

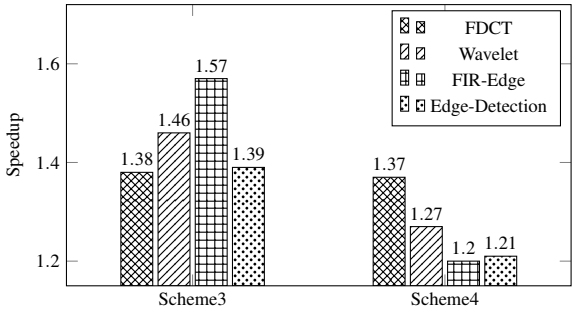


Figure 5. Speedups obtained by considering fine-grained data synchronization schemes 3 and 4 with task-level pipelining vs. a single core architecture for out-of-order benchmarks and $N = 50$.

obtain the same speedup for in-order benchmarks ($1.65\times$ for Gray-Histogram), It may not provide higher performance compared to the scheme 3. Since that the pattern of data communication is out-of-order, slowing down the consumer does not provide higher speedup.

4. Related Work

There is a number of related efforts considering fine-grained data synchronization and concurrency allow task-level pipelining. In the context of data synchronization, there are several approaches to overlap some of the execution steps of computing stages (see, e.g., [5]). In these approaches, data is communicated to subsequent stages in the code through a FIFO mechanism. Each FIFO stage stores an array element or a set of array elements. Array elements in each FIFO stage can be consumed by a different order than the one they have been produced. FIFO approach is sufficient when the order of consuming data is the same as the order of producing the data. However, the FIFO may not be efficient when the order of producing and consuming data is not the same.

In the presence of in-order P/C pairs, several attempts have been made to resolve the data communication for out-of-order tasks in compile time. For instance, Turjan et al. [4] address a task-level pipelining model maintaining the simple solution based on the FIFO between Producer and Consumer tasks and using a reordering mechanism to deal with out-of-order tasks [4, 9]. These approaches may not be feasible for all applications and they can be seen as an optimization phase of our approach. In our approach, we focused on the architectural schemes to enable task-level pipelining given in-order or out-of-order applications and without requiring code transformations.

5. Conclusions

We presented fine-grained approaches for task-level pipelining in the context of FPGA-based multicore architectures. We analyzed and compared different implementations of fine-grained data synchronization schemes for a set of in-order and out-of-order benchmarks in run-time. All the solutions proposed in this paper were implemented using an FPGA board and evaluated by measuring execution times using in-order and out-of-order benchmarks. The results show that our task-level pipelining approaches using a multicore architecture can achieve relevant speedups (close to the theoretical upper bound) over the sequential execution of computing stages in a single core. In the case of in-order benchmarks, the highest speedup achieved when the data communication component between the producer and consumer is a simple FIFO. For out-of-order benchmarks, the highest speedup obtained when the data communication component between the producer and consumer is an inter-stage buffer (ISB). The ISB can be a generic fine-grained data synchronization approach for in-order and out-of-order benchmarks.

References

[1] H. Ziegler, B. So, M. Hall, and P.C. Diniz. Coarse-grain pipelining on multiple FPGA architectures. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 77–86, 2002.

[2] Rui Rodrigues, Joao M. P. Cardoso, and Pedro C. Diniz. A data-driven approach for pipelining sequences of data-dependent loops. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '07, pages 219–228, 2007.

[3] D. Kim, K. Kim, J.Y. Kim, S. Lee, and H.J. Yoo. Memory-centric network-on-chip for power efficient execution of task-level pipeline on a multi-core processor. *Computers & Digital Techniques, IET*, 3(5):513–524, 2009.

[4] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Solving out-of-order communication in kahn process networks. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 40(1):7–18, 2005.

[5] H.E. Ziegler, M.W. Hall, and P.C. Diniz. Compiler-generated communication for pipelined fpga applications. In *Proceedings of the 40th annual Design Automation Conference*, pages 610–615, 2003.

[6] Ali Azarian, João M. P. Cardoso, Stephan Werner, and Jürgen Becker. An FPGA-based multi-core approach for pipelining computing stages. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1533–1540. ACM, 2013.

[7] B.J. Smith et al. Architecture and applications of the hep multiprocessor computer system. *Real-Time signal processing IV*, 298:241–248, 1981.

[8] Xilinx, Inc. *MicroBlaze Processor Reference Guide v12.3*, 2010.

[9] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A technique to determine inter-process communication in the polyhedral model. In *Proceedings of the 10th International Workshop on Compilers for Parallel Computers,(CPC 2003)*, pages 1–9, 2003.

[10] Benjamin James Braun. *Ehrhart Theory for Lattice Polytopes*. PhD thesis, Washington University, 2007.

Suporte de Comunicação para Controladores Distribuídos Modelados com Redes de Petri

Rogério Campos-Rebelo[1,2] *rcr@uninova.pt*
Edgar M. Silva[1,2] *ems@uninova.pt*
Filipe Moutinho[1,2] *fcm@uninova.pt*
Pedro Maló[1,2] *pmm@uninova.pt*
Anikó Costa[1,2] *akc@uninova.pt*
Luís Gomes[1,2] *lugo@fct.unl.pt*

[1] *Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia, Portugal*
[2] *UNINOVA - Centro de Tecnologia e Sistemas, Portugal*

Sumário

Neste trabalho é abordada a execução distribuída de modelos de redes de Petri IOPT que modelam redes de controladores distribuídos. Cada controlador tem associado um submodelo. Os submodelos utilizam canais de comunicação para trocar eventos entre si, permitindo a evolução global do sistema distribuído. Os canais de comunicação são implementados por nós de comunicação associados a cada controlador distribuído. Os nós de comunicação são caracterizados em termos de camadas e buffers. Um exemplo de aplicação simples é usado para demonstrar os conceitos. Neste exemplo é implementada uma rede de controladores distribuídos interligados através de uma rede com topologia em anel. São usadas placas Arduíno como plataformas de implementação para fins de prova de conceito.

1. Introdução

O desenvolvimento de várias tecnologias ao longo das últimas décadas tem levado a uma mudança nos tipos de sistemas desenvolvidos. Em *hardware*, este desenvolvimento permite a integração de múltiplos componentes em um único chip, tais como processadores, memórias e controladores dedicados, resultando assim na integração de um sistema completo. Por outro lado, a evolução das comunicações, tem permitido uma melhor interacção entre sistemas geograficamente distribuídos. Isto tem levado a uma mudança na forma como os sistemas são implementados. Estas mudanças permitem desenvolver sistemas distribuídos, tanto integrados num único circuito integrado, que normalmente é chamado de *System-on-Chip* (SoC), ou *System-on-a-Programmable-Chip* (SoPC), quando implementados num circuito reprogramável, como na implementação de sistemas com componentes fisicamente distribuídos, podendo ser implementado em plataformas heterogéneas.

Este aumento da complexidade dos sistemas faz com que as exigências de modelação sejam cada vez maiores, como tal, é natural encarar a divisão de um sistema em vários subsistemas interatuantes.

As redes de Petri *Input-Output Place-Transition* (IOPT) [1] são uma classe de Redes de Petri de baixo nível que estendem as Redes Lugar-Transição com características não autónomas, permitindo a modelação e interacção com o ambiente. Esta interacção é feita através de sinais e eventos de entrada e de saída. Estes sinais e eventos estão associados ao disparo das transições e a marcação dos lugares.

As redes de Petri IOPT [1] disponibilizam um conjunto de ferramentas que permitem o desenvolvimento de sistemas, permitindo a modelação, através de um editor [2], a verificação, através de um gerador de espaço de estados que tem associado um interface para perguntas (*queries*) [3] e ainda a implementação através de geradores automáticos de código (C ou VHDL) [4] [5]. Utilizando redes de Petri IOPT é também possível obter a especificação de sistemas distribuídos, quer seja através da decomposição de modelos, tanto por partição de um modelo único [6] em vários submodelos interatuantes, como através da modelação do sistema recorrendo a domínios temporais diferentes para cada subsistema [7] e a canais para especificar a interacção, garantindo-se a geração automática de código em qualquer das situações.

Do ponto de vista da comunicação entre estes subsistemas, o uso de ligações dedicadas é, normalmente, inviável, pois, apesar de oferecer melhor desempenho, ocupa demasiada área do ponto de vista da sua implementação em SoC ou necessitaria de cablagem específica quando se considerassem subsistemas fisicamente distribuídos, implementados, ou não, em plataformas heterogéneas. Desta forma, justifica-se a utilização de soluções de interligação dos subsistemas através de uma rede dedicada. No caso dos SoCs estas redes são designadas NoCs (*Network-on-Chip*), e um

trabalho sobre a sua implementação é apresentado em [8]. Nesse trabalho é ainda apresentada uma solução para a ligação entre NoCs usando código VHDL implementado em dispositivos FPGA e código C em PICs.

Neste trabalho pretende-se estender esses conceitos tendo em conta os sistemas fisicamente distribuídos, implementados em plataformas distintas. Para isso são estudadas várias topologias possíveis e é desenvolvido um nó de comunicação, que permite a implementação das várias topologias em dispositivos Arduíno. O código C que irá suportar a implementação de cada componente nos dispositivos *Arduino* é gerado automaticamente. Para a validação dos conceitos é usada a topologia em anel, mas o nó de comunicação é desenvolvido de forma a, com alterações mínimas, poder suportar diferentes topologias.

2. Topologias de Rede

Nesta secção é apresentada uma breve descrição de algumas topologias conhecidas (ver Figura 1) e que devem ser levadas em conta ao interligar os submodelos. As redes de Petri IOPT [6] foram estendidas para permitir a especificação de controladores distribuídos. Estes são responsáveis pela execução de cada submodelo estando associados a nós de comunicação.

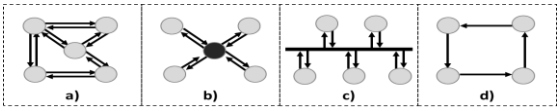


Figura 1. Topologias de Rede: a) Ponto a ponto; b) Estrela; c) Bus; d) Anel.

Ponto-a-ponto (Figura 1a)) é uma topologia onde cada nó (controlador) comunica com os outros nós através de uma ligação dedicada. Esta ligação permite a troca directa de mensagens entre eles, seguindo o caminho mais curto. Os nós são menos complexos uma vez que não precisam de descartar ou encaminhar mensagens. Por outro lado apresenta um número elevado de ligações dedicadas.

A topologia em **estrela** (Figura 1b)) é uma topologia com um nó central a que todos os outros nós se ligam e por onde todas as mensagens passam. É necessária uma ligação dedicada entre cada nó da rede e o nó central. Com esta topologia, novos nós são facilmente adicionados mas se o nó central falhar, toda a rede falha.

Na topologia em **bus** (Figura 1c)) todos os nós usam o mesmo canal (bus) para comunicar. Isto é, todos os nós escutam todas as mensagens e têm de esperar que o canal esteja livre para enviar novas mensagens. Assim sendo, os nós têm de descartar todas as mensagens de que não são destinatários.

Esta topologia apresenta uma aplicação de baixo custo mas um alto nível de complexidade no acesso ao canal de comunicação (bus).

Por último, na topologia em **anel** (Figura 1d)), cada nó (controlador) está apenas conectado a outros dois nós. Um pelo qual são recebidas as mensagens e outro para qual se envia mensagens. Estando todos conectados estes formam uma rede em anel fechado.

3. Nó de Comunicação

Este trabalho apresenta um nó de comunicação que fornece o suporte à execução distribuída dos modelos das redes de Petri IOPT, suportando a criação de redes de controladores distribuídos. Este nó de comunicação executa a transformação de eventos dos submodelos em mensagens que envia pela rede e vice-versa (Figura 2).

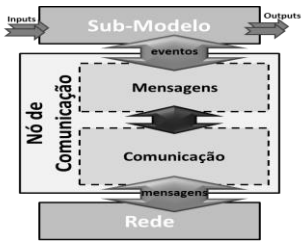


Figura 2. Arquitectura do Nó de Comunicação.

O nó de comunicação está dividido em duas camadas. A camada *Mensagens* é responsável pela transformação de eventos para mensagens e pela decomposição das mensagens para eventos. Efectua ainda a análise das mensagens verificando se estas chegaram ou não ao seu destino. É importante salientar que os tamanhos dos *buffers*, tanto no caso dos eventos como para as mensagens, podem ser definidos a priori através da análise do modelo da rede de Petri IOPT. A segunda camada, *Comunicação*, é responsável por fornecer uma abstracção em relação à interface de comunicação utilizada no envio das mensagens pela rede, i.e. do ponto de vista da camada *Mensagens*, uma mensagem é composta e enviada sem a preocupação de qual o tipo de comunicação utilizada (e.g. ethernet, série, I2C). É responsável por efectuar o registo das diferentes interfaces de comunicação, ou seja, quais as interfaces que podem ser utilizadas no envio das mensagens. A descoberta de caminhos e efectuar o reencaminhamento das mensagens são igualmente da responsabilidade da camada de *Comunicação*. Em relação à descoberta tradicional de caminhos vários protocolos podem ser utilizados, tais como o bem conhecido *Border Gateway Protocol* (BGP) [9], ou *Open Shortest Path First* (OSPF) [10], ou considerando multicaminhos com engenharia de tráfego podemos considerar o protocolo *Dynamic Topological Information Architecture* (DTIA) *Traffic Engineering* [11].

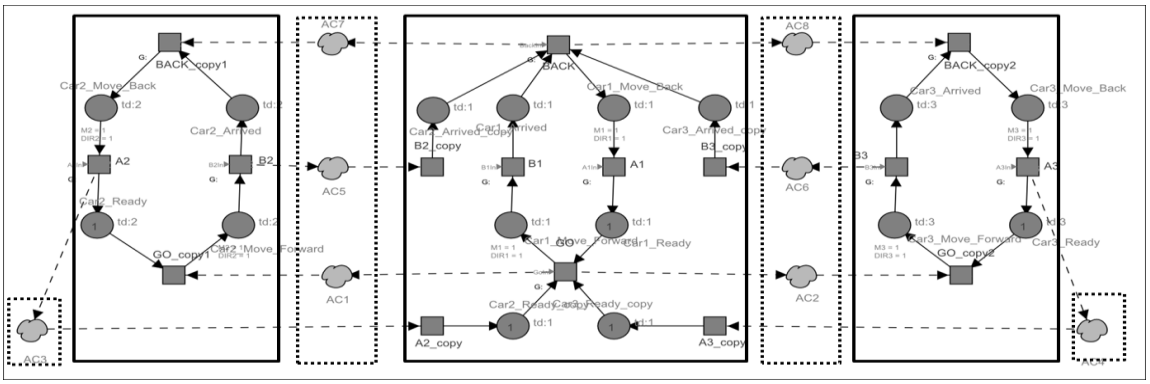


Figura 3. Modelos em redes IOPT para os 3 controladores distribuído.

4. Exemplo de aplicação

Para demonstrar a aplicabilidade da arquitectura apresentada é usado um exemplo onde se pretende implementar um sistema de controlo distribuído de três carros (Figura 3).

Os carros movem-se entre dois pontos (A e B). Apesar dos movimentos dos carros serem independentes em termos de velocidade e direcção, apenas quando os três veículos estão simultaneamente no mesmo ponto e o botão de início de movimento for premido (GO e BACK respectivamente), podem começar o seu movimento.

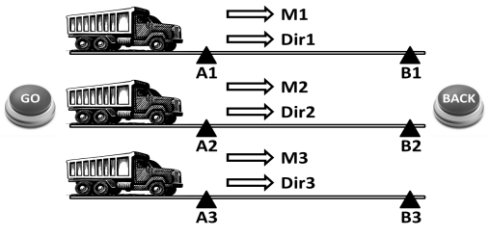


Figura 4. Exemplo de aplicação.

O sistema tem oito sinais de entrada (seis deles provenientes de sensores de passagem e dois de botões) e seis sinais de saída.

Pretende-se implementar este sistema recorrendo a um conjunto de controladores distribuídos, onde cada controlador controla um dos carros.

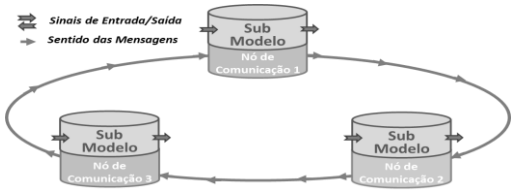


Figura 5. Execução distribuída usando uma topologia em anel.

A Figura 5 ilustra a montagem do exemplo de aplicação, uma execução distribuída do modelo em redes de Petri IOPT suportada numa rede de controladores seguindo uma topologia em anel. Cada controlador, associado a um nó de comunicação, é responsável pela execução de um submodelo IOPT. O submodelo responsável pela sincronização do início do movimento dos veículos está localizado no nó 1 (ou seja, o submodelo invoca / recebe eventos de / para os outros dois submodelos) e utiliza o canal entre o nó 1 e nó 2 para passar mensagens (ou seja, eventos) para os outros nós.

5. Resultados

O teste experimental executa o controlo de movimento de três veículos usando três controladores distribuídos ligados em anel, onde cada controlador foi implementado usando um dispositivo Arduíno (Arduíno Mega 2560). Cada submodelo (representado na Figura 3 por rectângulos com linha a cheio) foi implementado num dispositivo diferente utilizando uma comunicação série para receber mensagens e outra para o envio de mensagens, cada uma com taxa de transmissão de 19200 bits por segundo.

Para recolher as formas de onda dos sinais, os submodelos foram configurados para executarem o percurso no sentido da direita para a esquerda, isto é, as entradas B [i] ficam sempre activas, o GO é activado no início do passo de execução e desligado no passo de execução seguinte. As entradas BACK e A [i] estão desactivadas. Foi usado um osciloscópio para fazer a análise de dados com uma resolução de 2500 pontos por janela de tempo. Este foi calibrado com 25 milissegundos/divisão num total de 250 milissegundos para a janela de tempo, e com 5.0 volts/divisão no eixo vertical.

Os resultados mostram (Figura 6) que desde o sinal de entrada GO ser activo até ao momento em que é desligado (execução do passo seguinte da rede) o tempo gasto é de 6.68ms (forma de onda 1), este tempo inclui não só o tempo da execução de um passo da rede mas também o processamento dos dois eventos (um para o nó 2 e outro para o nó 3) para mensagens e o seu envio. Para a comunicação entre os nós foram utilizadas mensagens com um tamanho fixo de 6 bytes com a seguinte informação: id do submodelo de origem e de destino, id do evento de origem e de destino, tamanho dos parâmetros dos eventos e o limitador da mensagem. Cada campo

tem 1 byte. A forma de onda do sinal 2 mostra o momento em que é recebida a resposta do nó 2 com a informação que chegou ao fim do seu percurso. A forma de onda do sinal 3 representa o mesmo para o nó 3, ou seja, o momento em que chega a indicação que o carro 3 chegou ao ponto mais à direita. O tempo entre o envio das mensagens pelo nó 1 até receber a reposta do nó 2 é de 4.24ms e do nó 3 é de 8.16ms.

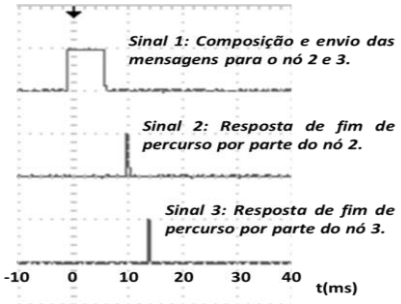


Figura 6. Formas de onda do controlador 1.

Os submodelos das redes de Petri IOPT foram transformados automaticamente usando gerador de código C da ferramenta IOPT-Tools (disponível em http://gres.uninova.pt), aos quais foram adicionados os ficheiros do nó de comunicação para suportar a comunicação série. É de salientar que o espaço ocupado pelo código do nó 1 é de 10.674 bytes (4.14%), do nó 2 de 9.752 bytes (3.78%) e do nó 3 de 9.760 bytes (3.78%), tendo cada Arduíno 258.048 bytes disponíveis.

Apesar de os resultados dos testes terem sido analisados em termos de duração, o objectivo deste trabalho não era tirar conclusões sobre o desempenho do funcionamento da rede. O propósito deste trabalho consiste na validação, como prova de conceito em como a arquitectura do nó de comunicação efectua a implementação distribuída de um modelo de rede IOPT. E assim sendo um ponto de partida para a criação de uma ferramenta a ser adicionada às IOPT-Tools para a geração automática dos ficheiros que suportem a implementação dos nós de comunicação, suportando a execução distribuída dos submodelos.

6. Conclusões e trabalho futuro

Este trabalho apresenta um nó de comunicação que suporta a execução distribuída de modelos IOPT. Este nó realiza a comunicação entre controladores distribuídos permitindo a execução descentralizada dos modelos da rede IOPT, como o uso de múltiplas interfaces de comunicação, possibilitando o uso de controladores heterogéneos.

Em termos de trabalho futuro está planeada a implementação usando outas topologias como por exemplo estrela, para se confrontar com os resultados obtidos neste trabalho.

Também, está prevista a implementação utilizando comunicação sem fios (Xbee), com o uso de diferentes tipos de plataformas (PC, FPGA, Arduíno), bem como diferentes linguagens (Java, VHDL, C).

O trabalho desenvolvido será integrado nas ferramentas IOPT-Tools (disponíveis em http://gres.uninova.pt), de modo a habilitar estas ferramenta com a capacidade de configuração e geração automática do código do nó de comunicação, para controladores distribuídos.

Agradecimentos

Este trabalho foi parcialmente financiado por Fundos Nacionais através da Fundação para a Ciência e a Tecnologia (FCT) no âmbito dos projetos PEst-OE/EEI/UI0066/2011 e PTDC/EEI-AUT/2641/2012

References

[1] L. Gomes, J. P. Barros, A. Costa, and R. Nunes, “The Input-Output Place-Transition Petri Net Class and Associated Tools,” in Industrial Informatics, 5th IEEE International Conference on, 2007, vol. 1, pp. 509–514.

[2] L. Gomes, F. Moutinho, and F. Pereira, “IOPT-tools - A Web based tool framework for embedded systems controller development using Petri nets,” Field Programmable Logic and Applications (FPL), 23rd International Conference on. p. 1, 2013.

[3] F. Pereira, F. Moutinho, L. Gomes, J. Ribeiro, and R. Campos-Rebelo, “An IOPT-net state-space generator tool,” 9th IEEE Int. Conf. Ind. Informatics, pp. 383–389, Jul. 2011.

[4] R. Campos-Rebelo, F. Pereira, F. Moutinho, and L. Gomes, “From IOPT Petri nets to C: An automatic code generator tool,” 9th IEEE Int. Conf. Ind. Informatics, pp. 390–395, Jul. 2011.

[5] F. Pereira and L. Gomes, “Automatic synthesis of VHDL Hardware Components from IOPT Petri Net models,” 2013.

[6] A. Costa and L. Gomes, “Petri net partitioning using net splitting operation,” Industrial Informatics, INDIN 2009. 7th IEEE International Conf. on. pp. 204–209.

[7] F. Moutinho and L. Gomes, “Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems,” in in Technological Innovation for Value Creation SE - 16, vol. 372, L. Camarinha-Matos, E. Shahamatnia, and G. Nunes, Eds. Springer Berlin Heidelberg, 2012, pp. 143–150.

[8] R. Ferreira, A. Costa, and L. Gomes, “Intra- and inter-circuit network for Petri nets based components,” Industrial Electronics (ISIE), IEEE International Symposium on. pp. 1529–1534, 2011.

[9] Y. Rekhter, T. Li, and S. Hares, “RFC 4271: Border gateway protocol 4.” Technical report, IBM, Cisco Systems, 2006. Last checked on May 19th, 2006.

[10] J. Moy, “Open shortest path first (ospf) version 2,” IETF Internet Eng. Taskforce RFC, vol. 2328, 1998.

[11] P. Amaral, E. Silva, L. Bernardo, and P. Pinto, “Inter-Domain Traffic Engineering Using an AS-Level Multipath Routing Architecture,”, IEEE International Conference on Communications (ICC). pp. 1–6, 2011.

REC 2014

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

isep Instituto Superior de Engenharia do Porto

UAlg
UNIVERSIDADE DO ALGARVE