REC 2013

Actas
IX Jornadas sobre Sistemas Reconfiguráveis

7-8 de Fevereiro de 2013

Instituto de Sistemas e Robótica
Universidade de Coimbra

Organizadores
Jorge Lobo
Manuel Gericota

INSTITUTO DE SISTEMAS E ROBÓTICA
UNIVERSIDADE DE COIMBRA

isep Instituto Superior de Engenharia do Porto

REC 2013

Actas

IX Jornadas sobre Sistemas Reconfiguráveis

7 e 8 de fevereiro de 2013

Instituto de Sistemas e Robótica - Universidade de Coimbra

Organizadores:
Jorge Lobo
Manuel Gericota

Edição: Comissão Organizadora da REC 2013
Jorge Lobo e Manuel Gericota

ISBN: 978-972-8822-27-9

Conteúdo

Arquiteturas multiprocessamento

Arquiteturas para processamento de alto débito

Reconfiguração dinâmica

Prefácio

As IX Jornadas sobre Sistemas Reconfiguráveis decorrem em Coimbra, no Instituto de Sistemas e Robótica - Universidade de Coimbra (ISR-UC), nos dias 7 e 8 de Fevereiro de 2012. Esta edição vem na continuação de uma sequência de eventos que teve início na Universidade do Algarve, em 2005, 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). 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 IX Jornadas – REC 2013 – tem uma estrutura semelhante às edições anteriores, decorrendo durante dia e meio. Este ano, as Jornadas incluem três apresentações convidadas focando diferentes domínios de utilização das FPGAs, tanto ao nível da investigação como do desenvolvimento industrial. Dado o trabalho do ISR-UC na área da visão por computador e robótica, pretendeu-se incluir um tópico mais focado nesta área. Assim, na primeira apresentação contamos com Julien Dubois, Maître de conférences da Université de Bourgogne, em Dijon, França, e membro do LE2I, Laboratoire d'Electronique, Informatique et Image, que abordará a utilização de hardware configurável em câmaras inteligentes para aplicações em ambientes industriais. A segunda apresentação está a cargo de Ricardo Chaves, investigador do grupo capa - Computer Arquitectures & high Performance Algorithms - do INESC-ID e professor auxiliar do Instituto Superior Técnico da Universidade de Lisboa, que abordará o tema da segurança em sistemas reconfiguráveis e implementação de sistemas de segurança com sistemas reconfiguráveis. A terceira apresentação será proferida por António Costa e por Miguel Falcão, gestores de I&D da Synopsis Portugal, que abordarão o tema da utilização de plataformas reconfiguráveis para prototipagem de controladores HDMI e MIPI. A todos agradecemos a disponibilidade para partilharem com os participantes da REC 2013 as suas experiências e conhecimentos.

O programa conta ainda com a apresentação de 15 comunicações regulares nas áreas da reconfiguração dinâmica, arquiteturas para processamento de alto débito, arquiteturas multiprocessamento, interfaces de comunicação, plataformas experimentais e educativas, bem como algumas 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, num total de 45 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 Actas, 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 ao Instituto de Sistemas e Robótica – Universidade de Coimbra, pelo imprescindível apoio concedido à organização destas Jornadas através da disponibilização de meios logísticos e administrativos.

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, e contamos vê-los a todos nas jornadas do próximo ano.

Jorge Lobo, ISR – Universidade de Coimbra

Manuel Gericota, Instituto Superior de Engenharia do Porto

Comissão Organizadora

Jorge Lobo	Universidade de Coimbra – ISR
Manuel Gericota	Instituto Superior de Engenharia do Porto

Comité Científico

Ana Antunes	Instituto Politécnico de Setúbal
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 Lima	Universidade do Algarve
João Canas Ferreira	Fac. de Engenharia da Universidade do Porto – INESC Porto
João M. P. Cardoso	Fac. de Engenharia da Universidade do Porto – INESC Porto
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 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 Porto – INESC-ID
Mário Calha	Fac. de Ciências da Universidade de Lisboa – LaSIGE
Morgado Dias	Universidade da Madeira
Nuno Roma	Instituto Superior Técnico – INESC-ID
Paulo Flores	Instituto Superior Técnico – INESC-ID

Paulo Teixeira	Instituto Superior Técnico – INESC-ID
Pedro C. Diniz	University of Southern California
Ricardo Machado	Universidade do Minho
Valeri Skliarov	Universidade de Aveiro – IEETA

Comunicações convidadas

 ISBN: 978-972-8822-27-9

FPGA-Based Smart Camera for industrial applications

Julien Dubois
LE2I (Laboratoire d'Electronique, Informatique et Image) – CNRS
Université de Bourgogne
Dijon - França

For the last two decades, smart cameras have been offering innovative solutions for industrial vision applications. This kind of system associates a flexible image acquisition with high-speed processing possibilities. Many smart camera designs are based on FPGA components to obtain these two features. Indeed, the FPGA enables the CMOS sensor to be controlled and therefore to propose a configurable acquisition according to the application constraints (i.e. dynamic windowing). The configurable structure of an FPGA represents a key advantage for modifying the embedded processing (even on-the-fly using dynamic reconfiguration). Additionally, FPGA components offer a large number of hardware resources, such as multipliers or embedded memory blocks, which enable complex image processing to be implemented and to be performed in real-time. Designers can even consider increasing the spatial image resolution and/or the frame-rate due to the FPGA technology improvements. The new solutions on the prototyping tools as well as the modelling languages available for FPGA design should be considered. Indeed, design methods based on High-Level Synthesis (HLS) enable the time to market to be significantly reduced. Moreover, these improvements enable gains on the smart camera design to be obtained, as for instance quick HW/SW implementations or quick communication interface integrations.

After a general presentation of the smart camera structure, the Le2i laboratory's experience on smart camera designs will be used to highlight these gains. The high processing capacities of an FPGA component at high frame rates, with high resolution images, will be demonstrated The presentation of the impact of co-processing on the smart camera performances, followed by a description of a new data-flow formalism, which enables quick prototyping of HW/SW implementations including communication interfaces to be automatically obtained, will be proposed. Finally, a configurable system supporting automatic video compression adaptation in function of event detection will be presented.

Keywords : smart camera, configurable systems, co-processing, High-Level Synthesis, communication interfaces

 ISBN: 978-972-8822-27-9

Secure computation using reconfigurable systems

Ricardo Chaves
INESC-ID / IST
TULisbon

With the vast expansion of electronic systems and digital information usage, an increasing demand from applications and users for secure and reliable computing environments arises. To deal with this demand, several mechanisms are being developed at different levels in computing and communications systems, namely at protocol, architectural, and algorithm levels.

An emerging new requirement to security systems is flexibility in terms of fast adaptation to new protocols, algorithms, and newly developed attacks. To provide this flexibility and adaptability while maintaining an adequate performance, the usage of reconfigurable devices, such as FPGAs, are of key importance.

Recent FGPAs have the additional advantage of enabling dynamically partial reconfiguration. However, this raises another security issue since it must be assure that whatever is being loaded into the reconfigurable device is in fact being loaded into the intended location and that once loaded it will behave as expected.

This presentation will discuss how the computation of security protocols and algorithms can be improved with the use of reconfigurable devices and how these reconfigurable devices can be securely used.

 ISBN: 978-972-8822-27-9

Synopsys HDMI and MIPI Prototyping Platforms

António Costa, Miguel Falcão
R&D Managers
Synopsys
Maia

One of the key components of Synopsys products are microelectronics sub-systems that are acquired by most of the semiconductors top vendors for integration in their SOC. These subsystems are commonly called IPs (for Intellectual Property) and Synopsys is the worldwide lead vendor of interface IP that includes popular standards such as USB, PCIe, DDR, SATA, HDMI, MIPI, etc...

HDMI and MIPI Controllers and PHY IPs are developed at Synopsys Portugal - Porto site. The quality of Synopsys IPs is worldwide recognized and such success is due to the quality of the design, verification and test flows.

The test flow requires prototyping these protocols in Synopsys laboratories where protocol controllers are prototyped in FPGA and connected to Synopsys PHYs implemented in real foundry silicon testchip. Prototyping is the ultimate verification and proof of the quality and robustness of the IP. It assures that our customer will receive a fully functional product when integrating it in their chips The importance of FPGA-based HAPS Prototyping activities is huge due to the impact it has in the business.

FPGA-based HAPS Prototyping platforms for HDMI and MIPI IP, its usages (both technical and business-oriented), past and future challenges and solutions will be presented.

http://www.synopsys.com

 ISBN: 978-972-8822-27-9

Plataformas experimentais e educativas

 ISBN: 978-972-8822-27-9

Using FPGAs to create a reconfigurable IEEE1451.0-compliant weblab infrastructure

Ricardo Costa[12], Gustavo Alves[1] and Mário Zenha-Rela[2]
ISEP/CIETI/LABORIS[1], FCTUC/CISUC[2]
rjc@isep.ipp.pt, gca@isep.ipp.pt, mzrela@dei.uc.pt

Abstract

The reconfiguration capability provided by Field Programmable Gate Arrays (FPGA) and the current limitations of weblab infrastructures, opened a new research window. This paper focus on describing the way weblabs can be reconfigured with different Instruments & Modules (I&M) required to conduct remote experiments, without changing the entire infrastructure. For this purpose, the paper emphasizes the advantage of using FPGAs to create reconfigurable weblab infrastructures using the IEEE1451.0 Std. as a basis to develop, access and bind embedded I&Ms to an IEEE1451.0-Module.

1. Introduction

In the electronic domain reconfiguration is becoming a familiar word since the appearance of FPGAs. These provide the ability of redefining an architecture based on a set of internal modules that can be interconnected according to a set of rules described by standard Hardware Description Languages (HDL). This means reconfiguring the device, and therefore the way it runs, without replacing its main hardware. This flexibility provided by FPGAs can be viewed not only as a thematic of study in engineering courses, but also as devices able to create the so-called weblab infrastructures, by the implementation of sensors/actuators that can be the I&Ms required to use in a remote experiment [1].

Weblabs allow the remote conduction of laboratorial experiments, providing a way for teachers and students to access real equipment, provided by an infrastructure, using a simple device connected to the Internet. Since the 90's that weblabs are proliferating in education, especially in engineering and science disciplines [2][3][4][5] where laboratorial work is fundamental [6][7]. This is justified essentially by the flexibility they provide on accessing, without time and place constrains, the equipment commonly available in a laboratory, which comprehends a set of I&Ms connected to an Experiment Under Test (EUT). Noticeably, the implementation of weblabs in different institutions can be increased if improving their infrastructures, namely by: i) enabling their reconfiguration (only setting up connections of predefined I&Ms is currently allowed [8]) and, ii) adopting a standard solution for their implementation and access. Despite these two problems are being debated in the GOLC technical committee [9], currently there is not yet a solution to solve them. While the standard access to a weblab infrastructure can be overcome by the use of a common API, infrastructural and reconfiguration aspects are still unsolved. It is precisely in this scenario that the reconfigurable nature of FPGAs and the use of a standard approach, can contribute to overcome the two referred problems.

Adopting FPGAs as the main device of a weblab infrastructure allow reconfiguring, in its core, a set of embedded I&Ms that, if described through standard HDL files, can be shared by the entire educational community. At the same time, if these same I&Ms follow a specific standard, they will be easily shared, integrated and accessed, promoting more collaboration among institutions in the development and dissemination of weblabs.

Therefore, for promoting a high widespread of weblabs in education, this paper proposes joining the capabilities provided by the reconfigurable nature of FPGAs, to the large focus provided by the IEEE1451.0 Std., that allows defining and network-interfacing transducers, which can be the referred I&Ms. The paper describes the implementation of a generic and synthesizable IEEE1451-Module for FPGA devices, and a methodology to develop, access and bind I&Ms compatible with this module.

Section 2 provides an overview about the IEEE1451.0 Std., and presents the weblab infrastructure implemented at our laboratory. Section 3, presents the IEEE1451.0 Std. and the IEEE1451.0-Module, this entirely described in the standard Verilog HDL. Section 4, describes the process of creating and binding I&Ms to that IEEE1451-Module, so they can be used by the weblab infrastructure to conduct experiments. Section 5 explains the reconfiguration process of the weblab infrastructure, and section 6 concludes this paper and presents ongoing work.

2. Weblab infrastructure overview

The IEEE1451.0 Std. [10] aims to network-interface transducers through an architecture based on two modules: the Transducer Interface Module (TIM), that controls Transducer Channels (TC), and the Network Capable Application Processor (NCAP), that provides network access to the TIM and to those TCs. The behaviour and features of a TIM and TCs are described by Transducer Electronic Data Sheets (TEDS) monitored by a status register and controlled by a set of commands that may be accessed by an IEEE1451.0 HTTP API.

As illustrate in figure 1, the implemented infrastructure uses the NCAP implemented in a micro webserver, connected by a serial RS232 interface to the TIM. This is implemented in a FPGA-based board that provides a set of interfaces (digital I/O, DAQs, etc.) to access a specific Experiment Under Test (EUT). Internally, the adopted FPGA is reconfigured by a generic IEEE1451.0-Module that, by decoding a set of commands received from the NCAP, controls TCs and therefore, the embedded I&Ms bound to it. To run remote experiments, users remotely access these I&Ms, that are connected to the EUT, using the IEEE1451.0 HTTP API implemented in the NCAP.

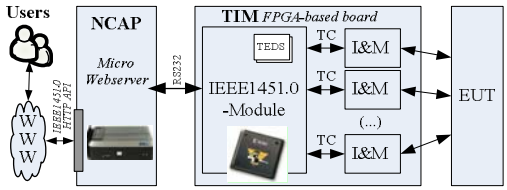


Fig. 1. Implemented weblab infrastructure.

While standardization is guaranteed by the adoption of the IEEE1451.0 Std., the use of an FPGA for implementing the TIM is fundamental, since it can be reconfigured with I&Ms required for a specific experiment, and these can run independently and in parallel, like in a traditional laboratory.

Therefore, seeking for a flexible and reconfigurable solution, the TIM was entirely described in Verilog HDL, which guarantees its portability towards any kind of FPGA. Internally the IEEE1451.0-Module implements all features described by the standard, controlling the TCs used to access the embedded I&Ms. The adoption of this architecture required the TIM description supported in two fundamental aspects: i) the IEEE1451.0-Module is able to be redefined according to the adopted I&Ms and, ii) each I&M is described through a set of files following a specific methodology.

3. IEEE1451.0-Module

Entirely described in Verilog HDL, as illustrated in figure 2, the IEEE1451.0-Module internally comprehends 4 other modules:

1- Decoder/Controller Module (DCM) - is the Central Processing Unit (CPU) that controls all the other modules, by decoding commands received from an Universal Asynchronous Receiver / Transmitter Module (UART-M) or by the reception of event signals generated by I&Ms.

2- TEDS Module (TEDS-M) - comprehends an internal controller able to access TEDSs.

3- Status/State Module (SSM) - manages the operating states and the status registers of each TC and TIM.

4- UART Module (UART-M) - interfaces the NCAP and the TIM through a RS232 interface using receiver/transmitter modules (Rx/Tx).

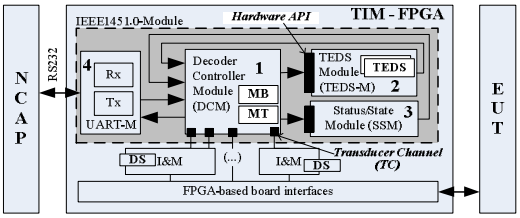


Fig. 2. Internal modules of the IEEE1451.0-Module.

The DCM controls the entire IEEE1451.0-Module by implementing the following features: i) provides IEEE1451.0 commands defined through a set of command-tasks, ii) implements error detection mechanisms, iii) controls both the SSM and the TEDS-M by reading, writing or updating their internal memories using a set of commands provided by dedicated hardware APIs, iv) controls the UART-M used to establish the NCAP-TIM interface, and iv) controls a set of embedded TC-tasks that manage the TCs, running as actuators, sensors or event sensors. The DCM provides a set buses that interfaces the TEDS-M, SSM and I&Ms, the UART-M to receive/transmit commands from/to the NCAP, and two external memories that support the operations of the DCM, named Memory Buffer (MB) and Map Table (MT). The MB gathers temporary TEDS' fields before they can be written into a TEDS's memory provided within the TEDS-M. It also acts as a data-bridge to Data Sets (DS), which are available in each I&M to hold internal data sent or received by IEEE1451.0 commands. The MT implements a table to associate each TEDS, defined in the TEDS-M, to a particular TC or TIM, according to a specific Identification Field (ID). Defined during a reconfiguration process described in section 5, it is based on this association that the DCM may understand which TEDS should be accessed after a reception of command.

 ISBN: 978-972-8822-27-9

The TEDS-M integrates all TEDSs adopted by the infrastructure, including those associated to a particular I&M, to the TIM and/or to TCs. This module comprehends an internal controller that provides particular commands to write, read or update each TEDS. To facilitate the access to those commands, the TEDS-M provides a hardware API, that can be used by the DCM, namely by command-tasks that implement IEEE1451.0 commands, and by TC-tasks that manage the interface between the I&Ms and the DCM.

The SSM provides access to two independent memories whose contents specify the operation states and the status of the TC/TIM. During the DCM operation, those memories will be accessed by command/TC-tasks to update the state and the status of each TC/TIM. The access to those memories is made using a set of commands provided by an internal controller, whose access can also be made by a hardware API.

The UART-M is controlled by the DCM using a handshake protocol that manage a set of signals to access two internal buffers and to control all data flow during transmissions. Structured in internal modules, the UART-M also implements a mechanism for validating and creating data structures according to the IEEE1451.0 Std..

In order to fulfill the reconfigurable requirements of the weblab infrastructure, besides using FPGA technology, the IEEE1451.0-Module was described through a set of Verilog HDL files some of them redefined according to the I&M adopted for a particular experiment. Moreover, its automatic redefinition, that is a part of the reconfiguration process, required the use of a specific architecture for developing and binding I&Ms, so they can be compatible with the IEEE1451.0-Module and therefore, able to be accessed according to the IEEE1451.0 Std..

4. Compatible Instruments & Modules

To bind I&Ms to the IEEE1451.0-Module, these should be designed in different parts. These parts include one or more modules bound through TC lines to a set of TC-tasks, which are described in Verilog HDL and embedded in the DCM. As illustrated in figure 3, these tasks allow the access to the other modules and the interface between the IEEE1451.0-Module and each I&M, enabling their control according to TEDSs' contents that should also be defined by the developer. The number of TCs depends on the I&M's architecture and the parameters to be controlled.

Therefore, the design of an I&M compatible with the IEEE1451.0-Module comprehends an architecture divided in 3 distinct parts: i) HDL modules describing the I&M itself, ii) TC-tasks to control and interface those same modules with the DCM; and iii) TEDSs to define the behaviour of the entire IEEE1451.0-Module and of each TC. An I&M is accessed by one or more TCs controlled by TC-tasks managed according to the data available within TEDSs and status/state memories. Since I&Ms' developers need to define both the TC-tasks and the HDL modules, they can adopt any type of handshake protocol to exchange data between the DCM and the I&Ms. Some TC-tasks are optional others mandatory, and they are responsible to automatically access the TEDS-M, the SSM, in some situations the UART, and the MB, when the IEEE1451.0-Module receives event signals or IEEE1451.0 commands.

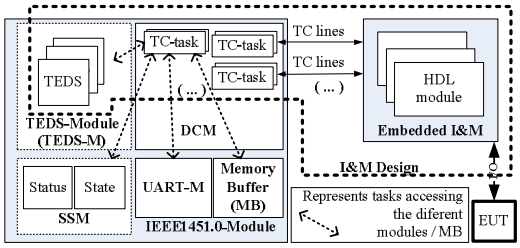


Fig. 3. Parts required for defining I&Ms compatible with the IEEE1451.0-Module.

To simplify the design of an I&M, each TC-task accesses those modules using the hardware APIs, facilitating this way their description and independence toward the specificities of the DCM implementation. They should be defined according to the adopted TC, so the DCM may automatically use them to handle received commands or events generated by I&Ms. The number of adopted TCs depends on developers' options that should take into consideration the parameters to control in an I&M, the TEDS's definitions, and the resources available in the FPGA. Therefore, the development of an I&M compatible with the IEEE1451.0-Module should follow the sequence presented in figure 4.

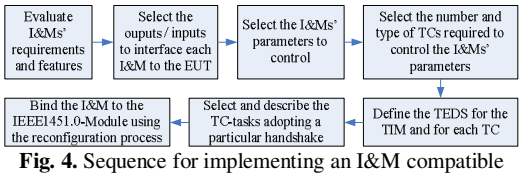


Fig. 4. Sequence for implementing an I&M compatible with the IEEE1451.0-Module.

Developers should start by evaluating the requirements and features of the I&M they want to develop, estimating its complexity to understand what modules should be described. For that purpose, the outputs and inputs connected to the EUT should be selected, namely the associated signals, which are managed by I&Ms' parameters controlled by TCs. After selecting the inputs/outputs and the parameters to be controlled, developers should define the number of TCs. This definition should be made according to the type of parameters to control and

the requirements posed to the FPGA device, since the use of several TCs may require many FPGA resources. Once selected the TCs used to access the I&M, developers should define the TEDSs to describe the TIM architecture and the TCs' behaviour that, among other definitions, specifies if a TC acts as an actuator, a sensor or as an event sensor. Current solution suggests that at least the TC-TEDS should be defined for each TC, but developers may define others TEDSs, as described by the IEEE1451.0 Std.. The way those TCs are controlled is made by a set of predefined TC-tasks described by the developer, so they can provide the interface to the other modules within the IEEE1451.0-Module. To simplify developments, the hardware APIs provided by the TEDS-M and the SSM should be used with the protocol adopted to control the data transmission/reception of the UART-M. After all these definitions, a specific I&M is available to bind to the IEEE1451.0-Module using a reconfiguration process.

5. Reconfiguration

After describing the I&Ms, these can be bound to the IEEE1451.0-Module so they can be used in a specific experiment. For this purpose, the infrastructure, namely the TIM, should be reconfigured, which means changing the internal connections of the FPGA. This reconfiguration process involves a set of steps described in figure 5, currently supported by a specific web reconfiguration tool already detailed in [11].

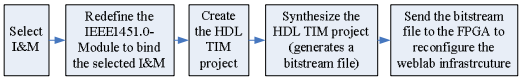


Fig. 5. Weblab infrastructure reconfiguration sequence.

This tool is available in a remote machine named Labserver that runs the entire reconfiguration process. Internally this machine integrates a set of software modules and, in particular, the IEEE1451.0-Module that will be redefined according to the selected I&Ms and to some configuration rules. For this purpose, users should start selecting two groups of files. The first group describing each I&M, and the second group describing all changes to be made in the TIM and in the IEEE1451.0-Module, so it may bind the selected I&Ms. The TIM, and in particular the IEEE1451.0-Module, is then redefined according to the rules defined in a configuration file, and a new HDL project will be created and synthesized to the selected FPGA using the tool associated to its manufacturer. A *bitstream* file is then created and sent to the FPGA, reconfiguring the weblab infrastructure to run the specified experiment.

6. Conclusions and ongoing work

The use of FPGAs is a promising solution for developing reconfigurable weblab infrastructures. This document emphasized this aspect, presenting current weblabs problems, and the way these can be solved by joining the IEEE1451.0 Std. basis with FPGA technology. The development of a reconfigurable, flexible and universal solution at low prices, is the main objective of the described work. In the next months a prototype experiment based on step-motors will be validated by some specialist in the area. The goal is to get feedback about the implemented infrastructure and the methodology for reconfiguring the weblab infrastructure. In the future, the intention is to enlarge the offer of compatible I&Ms, so other experiments can be designed. For further details, readers are invited to visit the webpage: www.dee.isep.ipp.pt/~rjc/phd.

References

[1] Antonio de la Piedra, An Braeken and Abdellah Touhafi, 'Sensor Systems Based on FPGAs and Their Applications A Survey', *Sensors*, vol. 12, no. 9, pp. 12235–12264, Sep. 2012.

[2] Javier García Zubía and Gustavo R. Alves, *Using Remote Labs in Education - Two Little Ducks in Remote Experimentation* -. Deusto Digital, 2011.

[3] Abul K.M. Azad, Michael E. Auer and V. Judson Harward, Ed., *Internet Accessible Remote Laboratories - Scalable Elearning Tools for Engineering and Science Disciplines*. Engineering Science Reference - IGI Global, 2011.

[4] Luís Gomes, 'Current Trends in Remote Laboratories', *IEEE Transactions on industrial electronics*, vol. 56, no. 12, p. 4744, Dec. 2009.

[5] G.R. Alves et al., 'Using VISIR in a large undergraduate course: Preliminary assessment results', *2011 IEEE Global Engineering Education Conference (EDUCON)*, pp. 1125 –1132, Apr. 2011.

[6] Lyle D. Feisel and George D. Peterson, 'A Colloquy on Learning Objectives For Engineering Education Laboratories', *Proceedings of the American Society for Engineering Education*, p. 12, 2002.

[7] Doru Popescu and Barry Odbert, 'The Advantages Of Remote Labs In Eng. Education', *Educator's Corner - Agilent Tech. - application note*, p. 11, Apr. 2011.

[8] Mohamed Tawfik et al., 'Virtual Instrument Systems in Reality (VISIR) for Remote Wiring and Measurement of Electronic Circuits on Breadboard', *IEEE Transactions on Learning Technologies*, vol. PP, no. 99, p. 1, 2012.

[9] 'GOLC - The Global Online Laboratory Consortium', 2012. [Online]. Available: http://www.online-lab.org/. [Accessed: 12-Nov-2012].

[10] IEEE Std. 1451.0™, 'IEEE Standard for a Smart Transducer Interface for Sensors and Actuators', *The Institute of Electrical and Electronics Engineers, Inc.*, p. 335, Sep. 2007.

[11] Ricardo Costa, Gustavo Alves and Mário Zenha-Rela, 'Reconfigurable IEEE1451-FPGA based weblab infrastructure', *9th Int. Conf. on Remote Eng. and Virtual Instrumentation (REV)*, pp. 1 –9, Jul. 2012.

A Remote Demonstrator for Dynamic FPGA Reconfiguration

Hugo Marques
Faculdade de Engenharia
Universidade do Porto
ee06273@fe.up.pt

João Canas Ferreira
INESC TEC and Faculdade de Engenharia
Universidade do Porto
jcf@fe.up.pt

Abstract

This paper presents a demonstrator for partial reconfiguration of FPGAs applied to image processing tasks. The main goal of the project is to develop an environment which allows users to assess some of the advantages of using dynamic reconfiguration. The demonstration platform is built around a Xilinx Virtex-5 FPGA, which is used to implement a chain of four reconfigurable filters for processing images. Using a graphical interface, the user can choose which filter goes into which reconfigurable slot, submit images for processing and visualize the outcome of the whole process.

1. Introduction

Partial dynamic reconfiguration consists in adapting generic hardware in order to accelerate algorithms or portions of algorithms. It is supported by a General-Purpose Processor (GPP) and reconfigurable hardware logic [1]. The processor manages the tasks running in hardware and the reconfiguration of sections of the FPGA. In addition, it also handles the communication with external devices.

It is expected that this technology will be more and more present in everyday devices. Therefore, it is important to introduce reconfigurable computing to electronic and computer engineering students. The goal of the project is to build a tool to support teaching the fundamentals of dynamic reconfiguration, as a starting point for students to experiment and be motivated to begin their on research on this field. To emphasize the dynamic reconfiguration operations, a system capable of processing images was implemented. In this way the user can easily visualize the results of the whole process.

The basic hardware infrastructure contains a processing chain with four filters working as reconfigurable partitions of the system. In regular operation, images flow through the four filters in sequence. To select the filters and to visualize the results of the selected operation, a remote graphical user interface was implemented in the Java programming language. This interface allows the user to send reconfiguration orders to the board in which the system is implemented. The user is able to visualize the original submitted image and the result after the process. As this application emphasizes the advantage of using dynamic reconfiguration, the remote user interface also shows some performance indicators. The system can be used in two different ways: a basic use and an advanced use. In the first one, the user only uses the graphical interface and the available filter library. An advanced user can expand the system by developing and implementing new filters following a guide that describes the design flow and the rules for a successful implementation.

The board chosen for the implementation of the system was the Xilinx ML505, which has a Virtex-5 FPGA [2]. The on-board software to control the reconfiguration process and the image processing tasks runs on a soft-core processor (MicroBlaze) inside the FPGA. This software was developed using the Embedded Development Kit (EDK) tool by Xilinx. The graphical user interface (GUI) runs in any environment that supports the Java language.

This document describes the approach used and the results obtained in the development of this project, which is mainly focused on a teaching context.The next section talks about work that has been developed in the scope dynamic reconfiguration on demand and dynamic reconfiguration systems with teaching purposes. The other sections of this article describe an overall view of the system (Sect. 3), the approach of hardware that was developed (Sect. 4), the strategies that were used to implement image filters (Sect. 5), the software that was developed for the whole system (Sect. 6), and the results (Sect. 7). Section 8 presents some conclusions.

2. Related Work

The use of dynamic reconfiguration is directly connected with its ability to speed up computing. It has already been proved that this kind of technology is capable of producing significant performance improvements in numerous applications. Also, the use of dynamic reconfiguration enables the reduction of the number of application-specific processors inside a single device, opening the way to weight and power consumption reductions in many areas, like for example a car or a cell phone.

In the scope of reconfiguration-on-demand, a project was developed in 2004, whose main objective was to incorporate many car functionalities in a single reconfigurable device [3]. As there was no need to have all the functionalities working at the same time, the main approach was to have them multiplexed to save energy efficiently. Algorithms were developed to do the exchange between tasks according to the run-time needs and energy efficiency con-

siderations.

This project concluded that systems of this kind are viable for the non-critical functionalities of a car, like air conditioning, radio, and lights control. It was possible to have four tasks working simultaneously in the FPGA and do exchanges with others as needed, without compromising the purpose of any activity and saving power consumption and space.

In the scope of demonstrating dynamic reconfiguration of FPGAs for education proposes, a system was recently developed whose main goal is to help students learn the fundamentals of this technology, allowing them to do experiments with a base system applied to video processing [4]. This system consists of a video stream that runs from a computer through a Virtex ML506 and back to the computer. The bitstreams are loaded at start-up to the board's internal memory. There are two bitstreams available corresponding to two possible implementations for transcoding the stream: up-scaling the stream to 256×256 pixels or broadcasting the input video stream to four lower quality receivers.

The downloading of the bitstreams into FPGA memory is done through the ICAP. In this specific project, the student experience has two phases: elaborating and implementing a C application to run on the MicroBlaze with the goal of transferring the bitstreams for the board, and receiving and executing reconfiguration orders. The second phase is about the user controlling the function, that is being executed in the board, by sending reconfiguration requests through a serial connection.

In the system that we developed, the user has a graphical interface in which he can choose and send reconfiguration requests to the board and visualize the results before and after processing.

3. Base System Overview

The implementation of the whole system has three parts: software developed for the remote interface, software developed for the soft-core processor and the hardware infrastructure compsed of the reconfigurable areas and the support hardware (CPU, DMA controller, ICAP, memory interfaces). Figure 1 represents the information flow and the modules used in the system. The reconfigurable hardware module is called `Img_process`. The communication between the interface and the board Virtex ML505 is done using TCP/IP sockets.

The soft-core processor is the center and brain of the system. It receives information and commands through the Ethernet module and acts accordingly. Before being processed, the image that the user sends, is stored on DDR2 RAM memory and then, when the image has been completely received, it is transferred to the processing module `Img_process`. This transfer is done with Direct Memory Access (DMA) [5] module which, when the information is processed, does the same operation in the reverse direction as well.

The `Img_process` module is pipelined and has the four partitions connected in series. This means that four

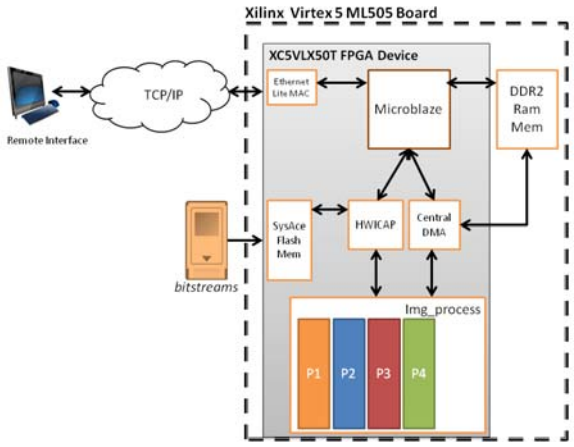


Figure 1. System overview.

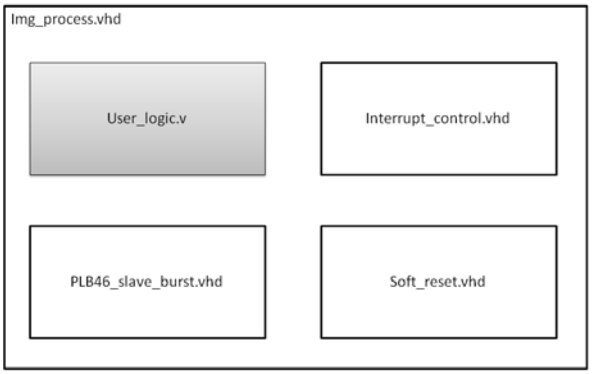


Figure 2. Top module `Img_process.v`

slots, each one with the correspondent filter selected by the user, are working simultaneously over successive bands of the image. When the DMA transfers a portion of the image, the module immediately starts processing and puts all processed bytes in a output memory for the DMA to transfer back again do DDR2 memory. This procedure is repeated until the image has been totally processed.

The partial bitstreams to reconfigure `Img_process` are stored in a flash memory card. This card is read using the SysAce module and the bitstream is sent to the Internal Access Configuration Point (ICAP) [6] by the control program running on the MicroBlaze processor. The ICAP is responsible for writing to the FPGA memory reserved to partial reconfiguration. All these modules are connected by the Processor Local Bus (PLB) [7].

4. Image Processing Hardware

As figure 2 shows, the module `Img_process` has four sub-modules. These are all generated by Xilinx Platform Studio when building the base system. They are responsible for the reset of the system, and the communication with the PLB bus. The interrupt control is not used in this version of the project, but it was maintained for future developments.

The `user_logic.v` module is where all the process-

 ISBN: 978-972-8822-27-9

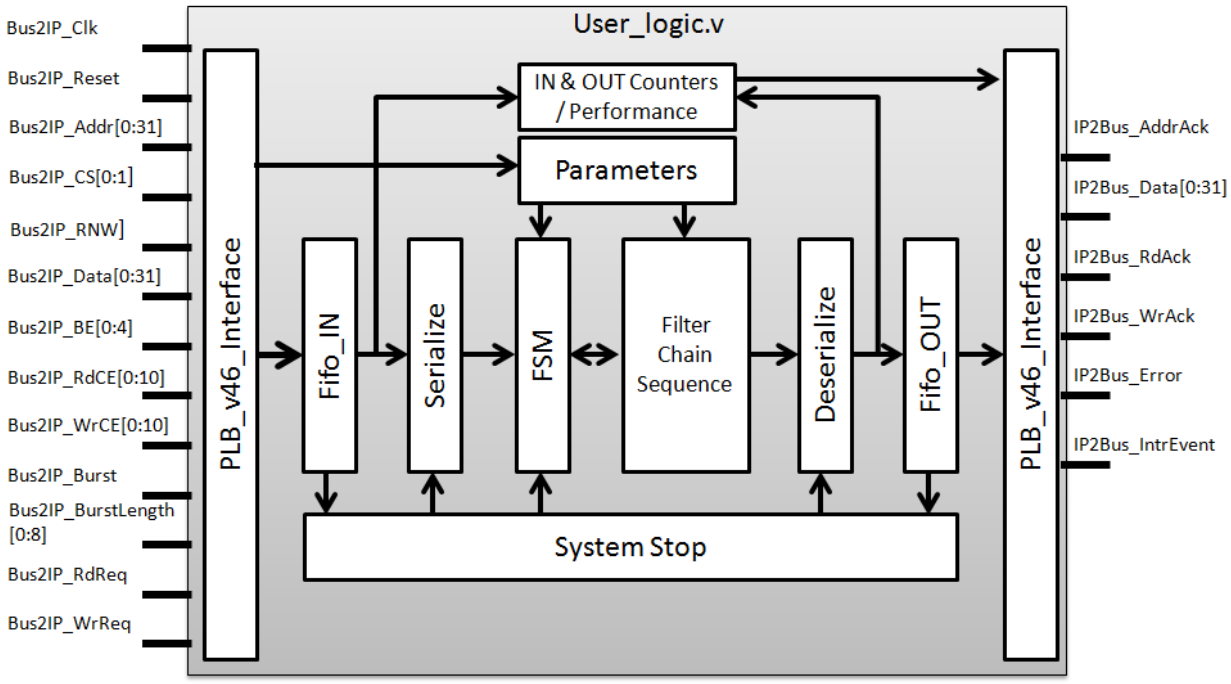


Figure 3. Implemented user logic block digram view.

ing logic is implemented. It is here that the four reconfigurable partitions are instantiated as black-boxes. This module receives collections of bytes with a rate which depends on the performance of the DMA and MicroBlaze. The transfer is done via PLB bus, hence with an input and output of four bytes in parallel in every clock cycle.

The chain of filters receives one data byte at a time. Therefore, it is necessary to convert the four bytes received in parallel to a byte sequence (parallel-to-serial converter) . There is also the need to aggregate the result information in a 4-byte word to be sent to the DDR2 memory. Hence, serialization and de-serialization modules were placed before and after the filters chain respectively. Figure 3 shows a more detailed view of the implemented `user_logic.v`.

In addition to the modules already mentioned, the `img_process` module contains performance counters, parameters, a stop process system, two FIFO memories, a state machine and the sequence of reconfigurable filters. The performance counters measure how many clock cycles are needed to process the whole image. The parameters block represents the registers used to store values introduced by the user through the graphical interface. These parameters are inputs of the user-designed filters and can be used in many ways. For example, the input parameter of one filter can be used as the threshold value. The two FIFO memories are used to store received and processed data. The finite state machine controls the whole operation: it has as inputs all the states of the other modules and acts according to them. The stop module prevents the system from losing data: when the `Fifo_OUT` memory is full, the system stops, or, when the `Fifo_IN` is empty and the last byte has not yet been received, the system stops. The system resumes processing when `Fifo_OUT` memory has space available for storing new results and `Fifo_IN` memory has new bytes to be processed.

The filter chain consists in four filters connected in sequence by FIFO memory structures. In overall there are eight FIFO memories, two behind each filter, as Figure 4 shows. In a given filter, the most current line of the image, is passed directly to the next filter and, at the same time, is updating the middle FIFO memory. Also, the bytes that were previously in the middle FIFO are entering the upper FIFO memory. In this way, every time a line is processed, the upper FIFO memory has the older line (N-2), the middle FIFO has the second to last line (N-1) and the most recent line (N) enters the filter directly. This arrangement ensures that the filter block receives successive columns of a three-line memory band. This means, that point operations based on a 3×3 neighborhood can be easily implemented in each filter block.

5. Partial Reconfiguration

5.1. Filter Development

As Fig. 4 shows, the system devised for this work allows the user to include new image filters based on point and neighbourhood (3×3) operations. Every filter developed for this infrastructure should have its inputs stored in registers inside the filters. This will decrease the possibility of timing violations on place and route phase. Since the filter has three inputs for bytes from three different image lines, in order to perform local operations the filters should register the incoming image byte, so that the three previous bytes, from each line at a given moment, are stored.

Data must pass through the filters at the same rate as

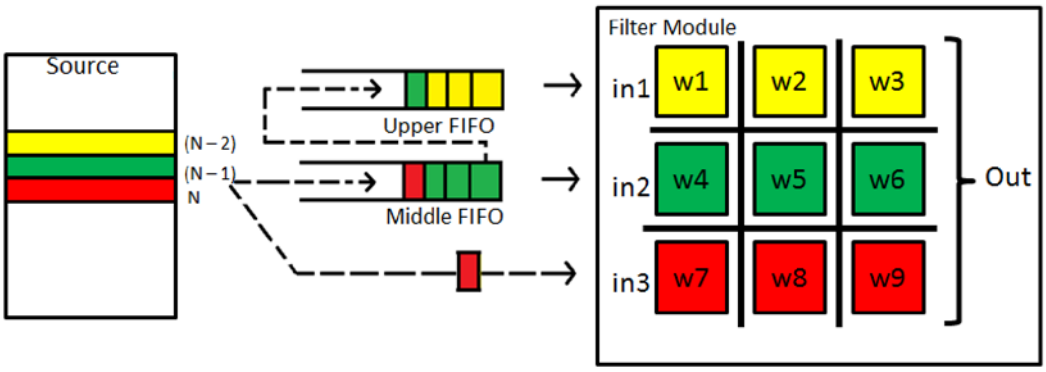


Figure 4. Single filter image process mechanism.

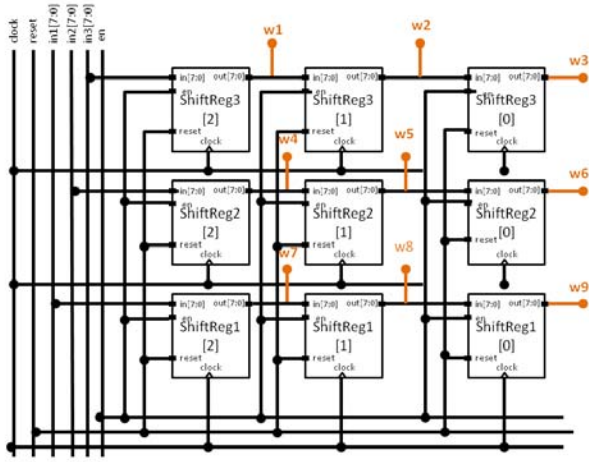


Figure 5. RTL schematic of a 3x3 operation window.

they enter the pipeline. Therefore, output calculations on a 3×3 window must also be done in a pipelined fashion, so as to not stall the filter pipeline: in each cycle a new output result must be produced. Hence, the operation is divided in sequential parts with registers between them. This will cause an initial delay at the output, but will not affect the overall frequency of the system. Instead, it will enable the utilization of higher frequencies and decrease timing violations such as hold and setup times.

To implement a 3×3 window like the one shown in Fig. 4, it is necessary to have nine bytes available at a given clock cycle. Hence, nine registers grouped in three shift chains are needed. Figure 5 shows how this arrangement is translated into an RTL schematic. At each cycle, the data for calculations related to the middle pixel (`w5`) are available. If these operations take longer than the cycle time, they too must be pipelined.

5.2. Floorplanning

The floorplanning [8] [9] phase is done using the Xilinx Planahed tool. Here, the netlists of the filters developed are added to the partitions defined as reconfigurable and then a placement and routing of all the system logic is performed. Finally, the partial and global bitstreams of the system are generated.

6. Communication and Control Software

To allow the user to control the operations done on the board, a remote graphical interface was developed. The communication between the board and the interface is done with the help of a TCP/IP-based protocol implemented on both sides. The protocol implemented starts out with the interface sending a configuration frame. This frame has all the information about the selected filters, the image size, the parameters for each filter and if it's a hard or soft processing request.

If the user chooses to run the process in hardware, the processing is done by the `Img_process` module. If the user chooses the software version, the image processing is done in the soft-core with the implemented software filters. After user approval, the graphical interface sends the whole image and then waits while the board is processing. When the board finishes, the image is sent back to the graphical interface and a new window pops out showing the outcome image.

On the board side, the protocol was implemented using MicroBlaze soft-core and the `lwip`library provided by Xilinx. When the image is received, a state machine starts sending and retrieving portions of the image to/from the `Img_process` module. In the case of software processing, the soft-core itself begins processing the image using filters implemented in the C language and compiled with `gcc` (-O2 optimization level).

Using the graphical user interface it is possible for the user to configure the connection settings, send an image to process, choose the filters he wants to use and see the image that results from the whole process. Besides the visual result, the user is also able to see the measured time that was needed to process the image. In this way, it is possible to do comparisons between software and hardware runs.

The user is allowed to perform any number of execution runs. Every time the user selects a different set of filters, the filter slots that change are reconfigured with the bitstream corresponding to the user's choice. Unused slots are configured with a pass-through ("empty") filter that the user chooses in the graphical interface. Figure 6 shows the

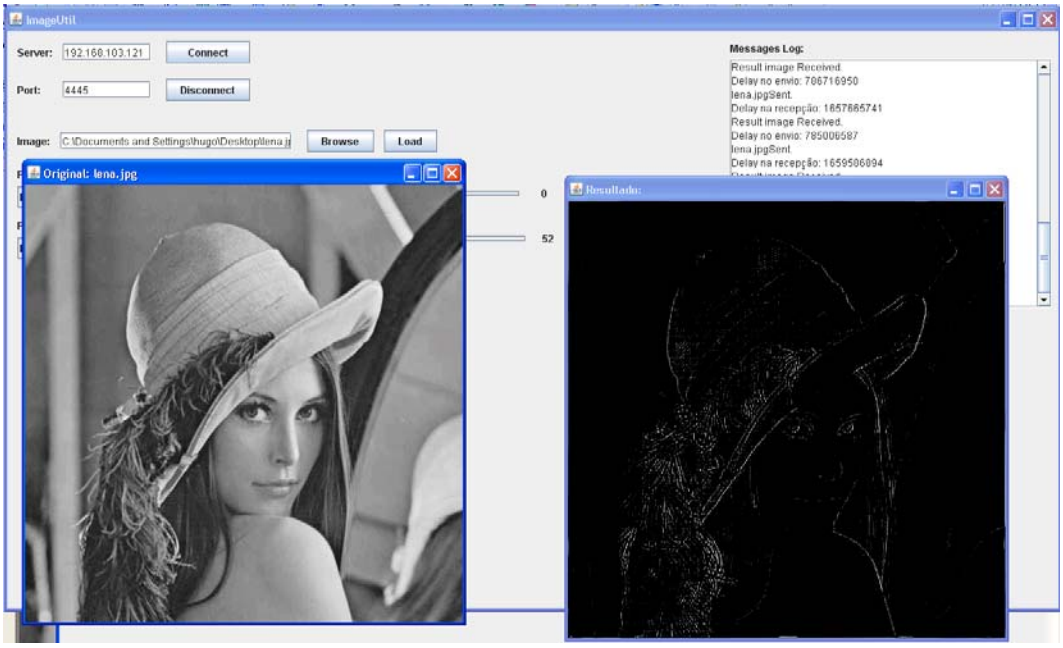


Figure 6. GUI with original and processed image.

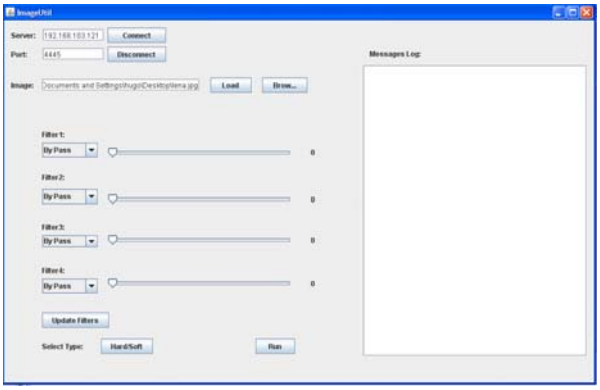


Figure 7. Graphical user interface.

interface with one image and the resulting image after processing, and Fig. 7 shows a snapshot of the interface developed;

7. Results

The soft-core (MicroBlaze) and the `Img_process` core are running at 125 MHz. To measure the performance of the implemented system, some operations with different image sizes were performed and the results were compared to the same chain of processing in Matlab, running on a Intel(R) Core(TM)2 Duo CPU E4500 at 2.2 GHz. Table 1 shows the processing time per pixel for different images.

There are two indicators for the hardware operation (`Img_process`"): total time and effective time. Total time is time taken by the whole operation, including the time that DMA spends transferring portions of the image. Effective time accounts only for the time when `Img_process` is actually processing image data.

Table 1. Performance indicators.

	`Img_process` time		Matlab
Image Size	Total	Effective	Total Time
512 x 512	124.86 ns	8.13 ns	54.55 ns
875 x 700	127.80 ns	8.10 ns	102.69 ns
1024 x 1024	129.03 ns	8.06 ns	189.40 ns

Examining Tab. 1 it is possible to conclude that as the image size grows, the processing time per pixel in Matlab increases significantly, but stays approximately constant for the hardware implementation. This can be explained by the pipelined architecture, which processes the image with the four filters working simultaneously. It is also possible to conclude that, as the image size grows, the effective time tends to 8 ns, which is the period that corresponds to the 125 MHz clock frequency. This is also a consequence of the pipelined architecture, which introduces some initial delays, but does not affect the overall frequency. So, as the image gets bigger these delays become more insignificant. The use of other type of filters only affects the initial delay of the pipeline (depending on the hardware complexity of the filter), but does not affect the performance of the whole operation.

Knowing that the system was implemented in a Xilinx XC5VLX50T FPGA device, the resources that were occupied were: 33% of registers, 32% of LUTs, 63% of slices, 48% of IOBs and 66% of BlockRAMs. This means that there is room for future developments and improvements.

8. Conclusion

The implementation meets the initial objectives and is completely functional. The user is capable of submitting an image, choosing the filters and parameters he wants to use, run the process and visualize the processed image and performance indicators.

It is possible to enlarge the window for local image operation. For that to be accomplished it is necessary to add some more memory FIFOs before each filter. Given the number of BRAM blocks available after implementation, it would be feasible to use an 8×8 window for image processing. Adding a new filter to the chain will not degrade the overall performance of the system. However, FPGA routing will be more congested and could lead to a reduction in operating frequency.

In order to enhance and improve the demonstration of the FPGA dynamic reconfiguration capabilities, we aim for future developments in the graphical interface, such as adding animations which can provide a quick understanding of what's happening in the FPGA during the reconfiguration process. Also, to simplify the development and addition of user-customized filters, a mechanism, provided by the graphical interface as well, could be developed to automatically do the integration, in the design, of a Verilog description filter done by the user; a filter would be then immediately available for further usage. As a result, we would have an automated process able to read a Verilog module description of a filter, synthesize it, implement it in the design's floorplan and generate a partial bitstream of it. In this way the user would have an easier understanding of the reconfiguration and would be able to have available his own filters to use in the implemented design just by writing a Verilog description of it.

Acknowledgments This work was partially funded by the European Regional Development Fund through the COMPETE Programme (Operational Programme for Competitiveness) and by national funds from the FCT–Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-022701.

References

[1] Pao-Ann Hsiung, Marco D. Santambrogio, and Chun-Hsian Huang. *Reconfigurable System Design and Verification*. CRC Press, February 2009.

[2] Xilinx Inc. *Virtex-5 Family Overview*, October 2011.

[3] Michael Ullmann, Michael Hubner, Bjorn Grimm, and Jurgen Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[4] Pierre Leray, Amor Nafkha, and Christophe Moy. Implementation scenario for teaching partial reconfiguration of FPGA. In *Proc. 6th International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC)*, Montpellier, France, June 2011.

[5] Xilinx Inc. *LogiCORE IP XPS Central DMA Controller (v2.03a)*, December 2010.

[6] Xilinx Inc. *LogiCORE IP XPS HWICAP (v5.01a)*, July 2011.

[7] Xilinx Inc. *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a)*, September 2010.

[8] Xilinx. *PLanahead Software Tutorial, Design Analysis And Floorplanning for Performance*. Xilinx Inc, September 2010.

[9] P. Banerjee, S. Sur-Kolay, and A. Bishnu. Floorplanning in modern FPGAs. In *20th International Conference on VLSI Design, 2007.*, pages 893 –898, January 2007.

Modular SDR Platform for Educational and Research Purposes

André Faceira, Arnaldo Oliveira, Nuno Borges Carvalho
Universidade de Aveiro - DETI / Instituto de Telecomunicações
Email: andrefaceira@ua.pt, arnaldo.oliveira@ua.pt, nbcarvalho@ua.pt

***Abstract*—Software Defined Radio (SDR) is a large field of research and its capabilities and applications are growing fast due to the large growth of wireless systems and the flexibility requirements of modern communication protocols. There are a few affordable SDR platforms in the market for educational purposes and research entry point. However, most of these development kits are very limited in terms of computational capacity, external interfaces, flexibility and/or observability of the internal signals. The most well known SDR kit is the USRP from Ettus Research LLC, that includes a motherboard with an FPGA for IF signal processing and that can be connected to RF frontends with different characteristics (frequency, bandwidth, output power, etc). The digital processing performed in the USRP FPGA is restricted to digital up/down frequency conversion and filtering. USRP relies on a PC for baseband processing using the GNURadio or Matlab/Simulink based packages leading to performance bottlenecks. On the other hand, traditional FPGA kits used for reconfigurable digital systems teaching and research do not include the AD/DA converters with the required performance for SDR systems. This paper introduces a modular SDR platform, that consists in a designed AD/DA board that interfaces directly with COTS FPGA kits and RF frontends, allowing to build flexible and cost effective platforms for SDR teaching, research and development. Different FPGA kits with the required capacity and interfaces, as well as frontends for several RF frequency bands can be used in a very flexible way, allowing an easy customization or upgrade of the system accordingly to the application requirements.**

I. Introduction

A radio is any kind of device that is able to transmit or receive wireless signals. Traditional radios were implemented in hardware using analog design approaches. Such radios were designed for a particular communication protocol and are very inflexible since they can only be modified or upgraded through physical intervention. With the emergence of new protocols, those radios become obsolete because they can not be adapted to new protocols, which leads to higher design, production and maintenance costs [1]. As the design is dominated by hardware, upgrading to a new protocol, usually means abandoning the old design and start a new one [2].

The concept of Sofware Defined Radio (SDR) or Software Radio was introduced by J. Mitola in 1992 [3][4]. The Wireless Innovation Forum in collaboration with an IEEE group has defined SDR as:

> *"A radio in which some or all of the physical layer functions are software defined" [5]*

A SDR has some kind of digital processor that is capable of implementing some part of the signal processing traditionally done in special-purpose hardware. Moreover, with SDR the physical layer (or part of it) can be controlled by software, thus the problems of the hardware radio are tackled. The same piece of hardware can perform different actions such as communicate using more than one protocol, turning this radio into a multi-functional and reconfigurable one. If there is the need to modify the radio behaviour to adapt to new protocols, it is not necessary to change the hardware, but only a piece of software or firmware.

The ideal SDR was defined by Mitola [3] (see figure 1) as consisting of a Low Noise Amplifier (LNA) and an Analog to Digital Converter (ADC) in the receiver chain, and a Digital to Analog Converter (DAC) and a Power Amplifier (PA) in the transmitter side. With this architecture all the signal processing is done in the digital domain by a baseband programmable processor.

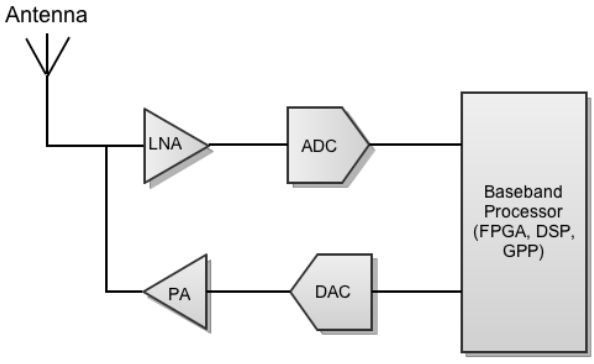


Figure 1. Ideal SDR, idealized by Mitola [4]

The ideal SDR would cover most of the radio frequency (RF) spectrum used. However, such high bandwidth exceeds the limitations of the technology. The data converters would need a very high sampling rate to support wide bandwidths. An operating bandwidth of several GHz to support the conversion over a high range of frequencies and other characteristics (such as dynamic range) are not currently achievable.

Hence, a practical SDR needs some hardware to accommodate the signal feed to the DAC or received from the ADC, and a baseband processor to implement the rest of the signal processing in software as shown on figure 2. SDR front ends are similar to those used in most transceivers, they have mixers, filters, amplifiers, voltage-controlled oscillators (VCO) and phase-locked loops (PLL). However, front end

characteristics such as frequency of operation, bandwidth and output power can be changed and controlled by software.

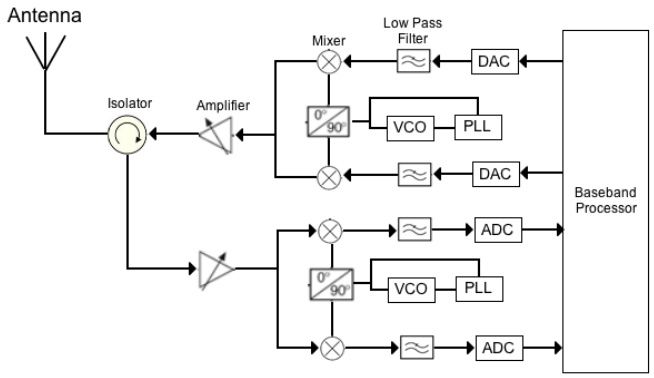


Figure 2. Practical SDR

Mitola also proposed in 1999 [6] an extension to SDR, the Cognitive Radio (CR). CR is a radio that has artificial intelligence and is able to sense its external environment and change the internal state to adapt and improve its performance [5]. CR is an environment aware radio that autonomously observes the spectrum usage, identifies unused radio spectrum bands and uses it in an intelligent way. It relies on the reconfigurability of SDR to change its radio operational parameters and machine learning algorithms to monitor its performance and adjust its behaviour dynamically.

SDR architectures are dominated by software and therefore the baseband processor plays a crucial role in the system. There are three main categories of digital hardware that can be used: Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) and General Purpose Processors (GPPs). It is also possible to use a hybrid solution with more than one kind of processor.

Each digital hardware category has its advantages and disadvantages. When choosing the digital hardware it should be considered some criteria, such as:

- Flexibility: the ability to handle a variety of air-interfaces and protocols;
- Level of integration: the ability to integrate several functions into a single device;
- Development cycle: the time needed to develop, implement and test the design;
- Performance: the throughput or processing time to perform a set of actions;
- Power: the power needed to perform some action.

FPGAs are used for designing and prototyping most systems. They are more suitable than the other types of digital processing platforms in the following situations [2]:

- Systems with high sampling rate;
- Systems involving variable word length: unlike DSP and ASIC, on the FPGA the word length can be set to the required length, improving their performance;
- Systems with high levels of parallelism, such as high order FIR filters (in the FPGA the algorithm can be implemented in parallel, decreasing the time required for processing);
- Systems that require custom datapaths, memories and/or functional units, such as fast correlators, FFTs and (de)coders.

These caractheristics are very useful to implement a SDR, therefore FPGAs are a commonly used digital processing platform for such purposes.

II. Commercial Platforms

There are many available SDR commercial platforms suitable for educational and research purposes, each one has different characteristics as shown in table I. As we can see, FPGA is the digital processor used in most platforms.

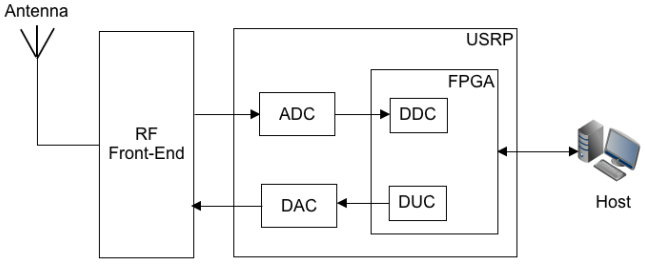


Figure 3. USRP architecture

The most well known platform is USRP. The USRP implements the front end functionality and AD/DA conversion, but assumes that the physical layer baseband processing is performed on a general purpose computer. Its architecture is illustrated in figure 3.

Platform	Digital Processor	Frequency Range	ADC Rate	DAC Rate	Cost
USRP1 [7]	FPGA	Up to 4 Ghz	64 MHz	128 MHz	$700
USRP N210 [7]	FPGA	Up to 4Ghz	100 MHz	400 MHz	$1500
QS1R [8]	FPGA	10 kHz - 62.5 MHz	130 MHz	-	$900
SDR-IP [9]	FPGA	0.1 kHz - 34 MHz	80 MHz	200 MHz	$2999
Perseus [10]	FPGA	10 kHz 40 MHz	80 MHz	-	$1199
AR 2300 [11]	FPGA	40 kHz 3.15 GHz	65 MHz	-	$3299
WinRaio WR-G31DCC [12]	DSP	0.09 - 50 MHz	100 MHz	-	$950
Flex-3000 [13]	DSP	10 kHz 65 MHz	96 kHz	96 kHz	$1700
Matchstiq [14]	DSP + GPP	300 MHz 3.8 GHz	40 MHz	40 MHz	$4500

Table I
Commercial Platforms

 ISBN: 978-972-8822-27-9

USRP N210 consists in a motherboard with two 100 MHz 14 bit ADCs, two 400 MHz 16 bit DACs, an Altera FPGA that provides two Digital Up Converters (DUC) and two Digital Down Converters (DDC) (the rest of the signal processing is by default done on a computer) and a connection to a RF daugtherboard. There are various cost effective daugtherboards provided by Ettus (table II), making USRP very flexible in terms of frequency of operation.

Daughterboard	Frequency Range	Type (Rx/Tx)
BasicTX	1 - 250 MHz	Transmitter
BasicRX	1 - 250 MHz	Receiver
LFTX	0 - 30 MHz	Transmitter
LFRX	0 - 30 MHz	Receiver
TVRX2	50 - 860 MHz	Receiver
DBSRX2	800 - 2300 MHz	Receiver
WBX	50 - 2200 MHz	Transceiver
SBX	400 - 4400 MHz	Transceiver
XCVR2450	2.4 - 2.5 GHz, 4.9 - 5.9 GHz	Transceiver
RFX900	750 - 1050 MHz	Transceiver
RFX1800	1.5 - 2.1 GHz	Transceiver
RFX2400	2.3 - 2.9 GHz	Transceiver

Table II
RF FRONT ENDS AVAILABLE ON ETTUS RESEARCH, LLC

Almost all platforms require a computer to control its operation and transfer the data for signal processing. There are many software platforms available and generally, each company that develops SDR platforms have its own. Some examples are: GNU Radio [15], Winrad [16], PowerSDR [17], SpectraVue [18] and Quisk [19]. GNU Radio is an open source platform for SDR that has many implemented blocks for signal processing such as equalizers, modulators, demodulators, filters, scramblers, among others. It can be used with the USRP hardware and, since it is open source, it allows customizing the blocks or design new ones. The other software platforms are not so flexible and customizable as they are limited and do not allow an easy implementation of new features.

III. MOTIVATION

As described before, most SDR systems need three basic blocks: a RF front end, a baseband processor and an interface to convert between the analog and the digital domains. In the platforms available on the market these three blocks are often physically integrated, making it impossible to change or upgrade the block individually. On the other hand, modular SDR platforms are very expensive and include high end FPGAs, DSPs or both, as well as high speed AD/DA components. To the best of our knowledge Ettus Research LLC, is the only company that develops affordable platforms that are independent from the front end and thus, it is possible to operate in different frequency ranges changing only the front end board. However, the FPGA used in the Ettus USRP kit has very limited capacity and it is relatively difficult to change its functionality.

On the other hand, universities and research institutes with FPGA-based design courses and projects already have Commercial Off-The-Shelf (COTS) FPGA kits with a rich set of interfaces that can be used in a very flexible way for baseband processing in SDR systems if adequate AD/DA and front end modules are provided. However, in the market place there is no affordable RF + AD/DA platform that is independent from the digital processor making them more expensive and inflexible to accommodate developments of the digital technology.

The motivations for developing the modular architecture presented in this paper are the following:

- to foster the research of SDR and its enabling technologies with a modular platform where every block is independent from each other, enabling an easy modification of each one (the FPGA module, the AD/DA stage and the RF front end are completely independent with a well defined interface among them);
- to reuse a large set of affordable FPGA kits normally used in reconfigurable digital system classes and research projects;
- to reuse a large set of RF front ends available for the USRP that cover a vast frequency range of the radio spectrum.

Since this work is based on COTS FPGA and front end kits, the main contribution is on the AD/DA module and supporting IP core components that would allow using the entire set in the design of SDR systems for educational and research purposes.

IV. PROPOSED PLATFORM

The proposed platform is composed by three boards (figure 6), each SDR basic block corresponds to a different board.

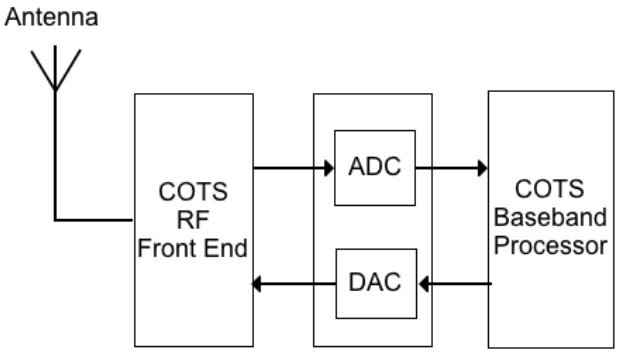


Figure 6. Conceptual kit architecture.

The baseband processor can be any COTS FPGA kit that includes a VHDC connector to transmit or receive a signal. To work as a transceiver (Rx/Tx) it is needed a FPGA kit with two VHDC connectors or two kits each one with one connector. The FPGA used to implement this platform is the Genesys Virtex-5 from Digilent [20], but the cheaper Nexys 3 or Atlys kits can also be used. Such kits are the most used platforms for teaching reconfigurable digital systems. The Genesys kit has two VHDC connectors, gigabit ethernet and has a clock generator up to 400 MHz, among a large set of peripherals and add-on modules.

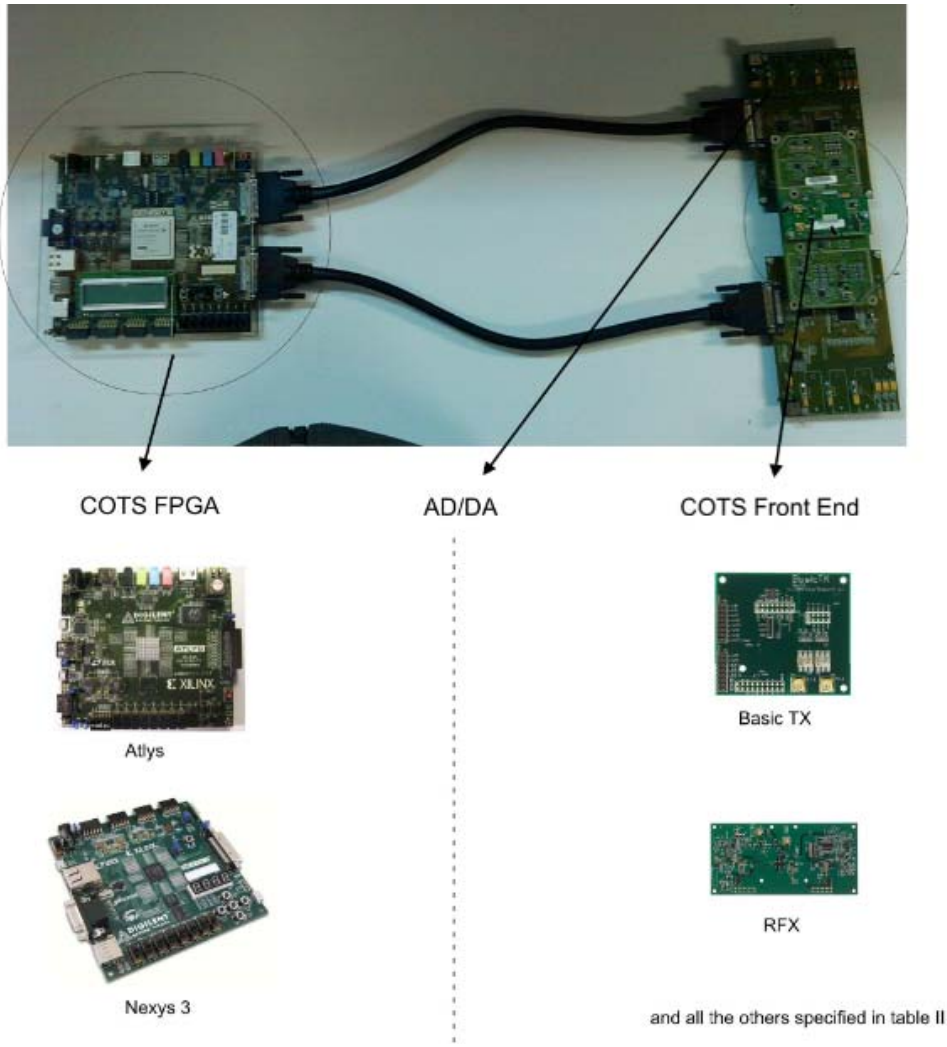


Figure 4. Overall structure of the developed setup with different combinations of FPGA and RF modules

The RF front end has to accommodate the signal to a suitable range to be converted by the DAC or the ADC. In the transmitter path the front end has to shift the signal from an intermediate frequency (IF) to RF. In the receiver path it has to shift the signal from RF to IF or baseband. The AD/DA converter board was designed to support any front end from Ettus Research (table II), but this platform can also be used to test an experimental front end. The architecture of the front ends from Ettus is similar to the architecture shown in figure 2 with amplifiers, mixers, filters, VCOs and PLLs. The VCO can be externally controlled to change the carrier frequency.

An overview of some of the components that can be used are show in figure 4.

The AD/DA converter is responsible for the interface between the front end and the FPGA kit. It is constituted by two boards, one for transmission and one for reception. The layout of these boards are depicted in figure 7 and 8 respectively.

The transmitter board includes two 105 MHz DAC with 16 bits a clock manager that receives the clock from the FPGA and accomodate it to the DAC and to the front end. It has an auxiliary ADC and an auxiliary DAC to do analog control in the front end and a CPLD. It also has its own power supply and provides necessary power to the front end.

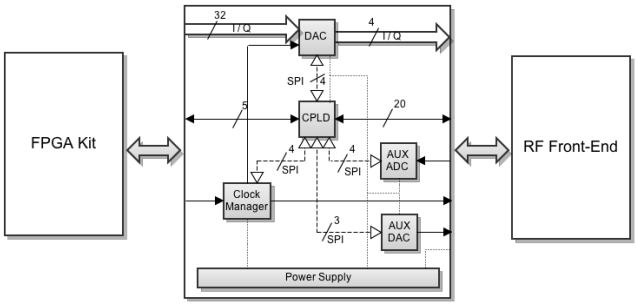


Figure 7. Transmitter board layout

The receptor board is similar to the transmitter board. It

 ISBN: 978-972-8822-27-9

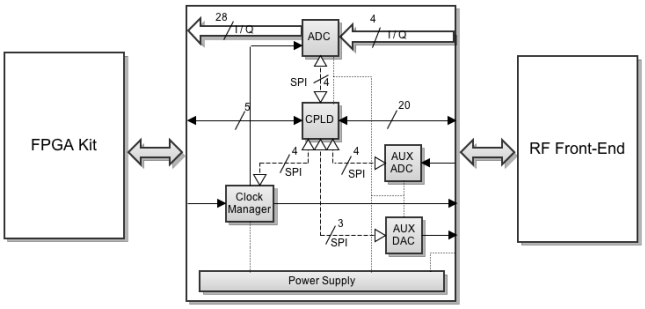


Figure 8. Receiver board layout

has the same components, but instead of a DAC, it has an 400 MHz ADC with 14 bits.

The VHDC connection has 20 differential pairs (40 bits), however the AD/DA board needs many more signals to work correctly. In the case of the trasmitter board, it needs 32 bits to feed the signal to the DAC, 2 bits for the clock, 15 bits to implement the Serial Peripheral Interface (SPI) protocol to the various components, and 20 bits to control and configuration of the front end. The trasnsmitter board would need a 69 bits connection with the FPGA, thus to be able to connect with only one VHDC connection a CPLD was included in the design of the board. The roles of CPLD are:

- comunicate with the FPGA kit;
- implement a SPI Master to control all components on the board by SPI protocol;
- provide the front end 20 input/output bits.

With the CPLD in the board, the transmitter only needs a 39 bits connection with the FPGA being possible to use the VHDC connection.

The purpose of the auxiliary ADC/DAC is to be able to do some analog control in the front end. Using the WBX front end from Ettus Research, they are used to control the quadrature modulator.

The clock manager is used to convert and accomodate the clock to the ADC/DAC and front end. The input is a differential clock and the output to the front end is CMOS/LVDS, to the ADC CMOS and to the DAC LVPECL.

The components used in the board are specified in table III .

Component	Reference
DAC	AD9777 [21]
ADC	LTC2284 [22]
CPLD	XC2C128 [23]
Clock Manager	AD9512 [24]
Auxiliary ADC	AD7922 [25]
Auxiliary DAC	AD5623 [26]

Table III
COMPONENTS USED

These boards have characteristics similar to the commercial available platforms, with an ADC capable of operate at a rate of 100 MHz and a DAC operating at 400 MHz. Using Ettus front ends, these platform have a frequency range of operation up to 4 GHz.

The platform will provide all the front end and AD/DA board functionality with open source intellectual property (IP) cores. This way, the software will be reusable and the implemented hardware will be independent from the FPGA used.

V. SOFTWARE TOOLS

The platform needs two types of software, the processing chain to be implemented in the FPGA and the digital signal

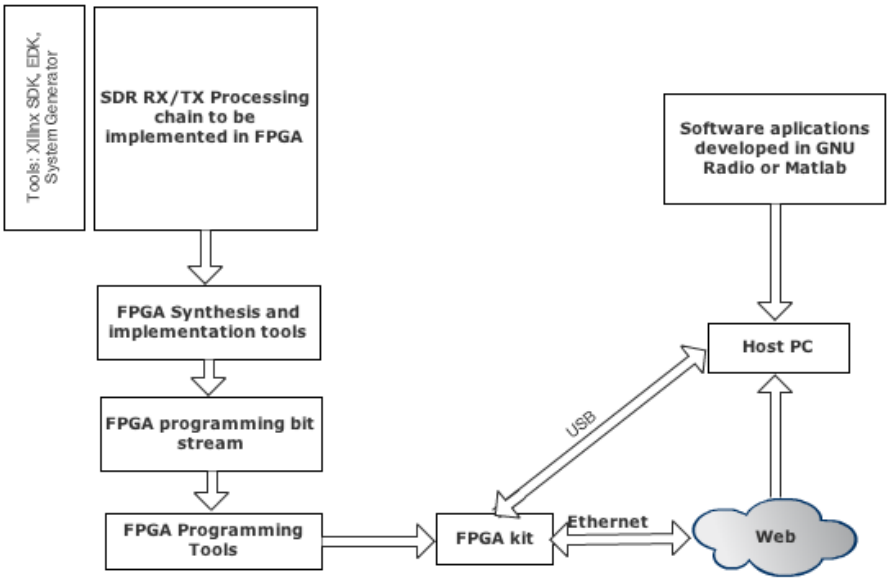


Figure 5. Programming Flow

processing to be implemented in the host PC.

Software for the FPGA kit can be developed using traditional FPGA tools, such as the Xilinx ISE [27] (a package that contains all the programs needed for the entire FPGA development flow, it enables device programming, design entry, synthesis, place & route and simulation), the Xilinx EDK [28] (a package that includes tools for designing embedded processing systems supported by Xilinx) and the System Generator [29] (a interconnection with Matlab/Simulink that allows the implementation and debugging of digital system for the FPGA in Matlab).

With the generated programming bitstream, vendor specific tools (e.g. Digilend Adept, Xilinx Impact, etc.) can be used to download the bitstream to the FPGA.

The FPGA kit interfaces with a host PC via USB or ethernet. The host PC can process the data with software applications developed in GNU Radio or Matlab.

An overview of the programming flow is depicted in figure 5.

VI. Application Example

An SDR can work as a remote radio or a Web SDR, converting part of the radio spectrum in a remote location and transmitting it to the Web or fetching data from the Web and converting it to RF. This would enable to configure and control the SDR remotely and every kind of device with internet connection could access the data being received by the SDR or send data to be processed and transmitted. The conceptual idea is shown in figure 9.

To achieve this the SDR must have an ethernet connection. The vast majority of the COTS FPGA kits has an on-board ethernet connection and they can connect to the Internet and act as a stand alone web server and therefore this application is costless and useful.

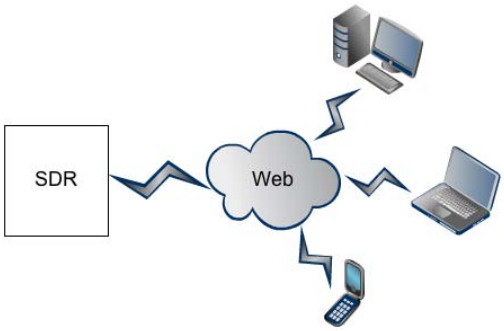


Figure 9. SDR Web

This application can be very helpful to implement and test new algorithms for CR as we can connect various SDR to an host that coordinates their mode of operation to use the radio spectrum inteligently.

VII. Conclusion and future work

In this paper, we presented an overview of SDR and its enabling technologies and some commercial available platforms. We also presented a new modular architecture for SDR platforms to offer more flexibility to accomodate new developments in the technology associated. A prototype platform using this architecture is in production and we hope that it can be used by universities or research centers to develop a broader community of users and share new applications.

This platform could benefit from further developments, such as:

- IP cores to support the implemented hardware;
- signal processing blocks for the FPGA, such as modulator, demodulators, etc;
- portability to the computer, to be able to test new radio designs in programs such as GNU Radio, before implementing in the FPGA.

References

[1] Wireless Innovation Forum, http://www.wirelessinnovation.org/introduction_to_sdr
[2] Jeffrey H. Reed, "Software Radio: A Modern Approach to Radio Engineering", Prentice Hall, 2002
[3] J. Mitola, The software radio architecture, IEEE Commun. Mag., vol. 33, no. 5, pp. 2638, May 1995.
[4] J. Mitola, "Software Radios: Survey, Critical Evaluation and Future Directions", National Telesystems Conference, 1992.
[5] SDR Forum, http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf
[6] J. Mitola and G. Q. Maguire, Cognitive radio: Making software radios more personal, IEEE Pers. Commun., vol. 6, no. 4, pp. 1318, Aug. 1999.
[7] Ettus Research, http://www.ettus.com/products
[8] Software Radio Laboratory, LLC, http://www.srl-llc.com/
[9] RF Space, http://www.rfspace.com/RFSPACE/SDR-IP.html
[10] Microtelecom, http://www.microtelecom.it/perseus/
[11] AOR, http://www.aorusa.com/receivers/ar2300.html
[12] WinRadio, http://www.winradio.com/home/g31ddc.htm
[13] Flex Radio, http://www.flexradio.com/Products.aspx?topic=SDR_Feature_Matrix
[14] Epiq Solutions, http://www.epiqsolutions.com/matchstiq/
[15] GNU Radio, http://gnuradio.org/redmine/projects/gnuradio/wiki
[16] Winrad, http://www.winrad.org/
[17] PowerSDR, http://www.flex-radio.com/Products.aspx?topic=powersdr1x
[18] SpectraVue, http://www.moetronix.com/spectravue.htm
[19] Quisk, http://james.ahlstrom.name/quisk/
[20] Digilent, http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,819&Prod=GENESYS
[21] AD9777, http://www.analog.com/static/imported-files/data_sheets/AD9777.pdf
[22] LTC2284, http://cds.linear.com/docs/Datasheet/2284fa.pdf
[23] XC2C128, http://www.xilinx.com/support/documentation/data_sheets/ds093.pdf
[24] AD9512, http://www.analog.com/static/imported-files/data_sheets/AD9512.pdf
[25] AD7922, http://www.analog.com/static/imported-files/data_sheets/AD7912_7922.pdf
[26] AD5623 http://www.analog.com/static/imported-files/data_sheets/AD5623R_5643R_5663R.pdf
[27] Xilinx ISE, http://www.xilinx.com/products/design-tools/ise-design-suite/logic-edition.htm
[28] Xilinx EDK, http://www.xilinx.com/tools/platform.htm
[29] Xilinx System Generator, http://www.xilinx.com/tools/sysgen.htm

Interfaces de comunicação

 ISBN: 978-972-8822-27-9

Dimensionamento de buffers para redes ponto a ponto de sistemas GALS especificados através de redes de Petri

Filipe Moutinho, José Pimenta, Luís Gomes
Universidade Nova de Lisboa – Faculdade de Ciências e Tecnologia, Portugal
UNINOVA – CTS, Portugal
fcm@uninova.pt, jep19206@campus.fct.unl.pt, lugo@uninova.pt

Sumário

Partindo da especificação de sistemas distribuídos globalmente assíncronos localmente síncronos (GALS) utilizando redes de Petri, neste artigo apresenta-se uma forma de dimensionar as suas redes de comunicação com topologia ponto a ponto. Tendo em conta que a interação entre componentes se faz através da troca de eventos, quando um componente gera eventos, estes têm de ser armazenados (em buffers) até serem enviados, e os componentes de destino têm de armazenar (em buffers) os eventos recebidos até estes serem consumidos. Este artigo apresenta uma forma de dimensionar estes buffers, integrada numa abordagem de desenvolvimento de sistemas baseada em modelos, suportada por ferramentas de validação e de geração automática de código para dispositivos reconfiguráveis.

1. Introdução

Abordagens de desenvolvimento baseadas em modelos têm contribuído para o desenvolvimento de sistemas embutidos [1][2][3][4]. A possibilidade de simular e verificar a especificação utilizando ferramentas computacionais, aliadas à geração automática de código, permite desenvolver sistemas embutidos em menos tempo e com menos erros de desenvolvimento [4].

A utilização de abordagens de desenvolvimento de sistemas embutidos baseadas em modelos, complementadas pela utilização de dispositivos reconfiguráveis para a sua implementação, permite a criação de sistemas embutidos mais fáceis de alterar, tanto durante a fase de desenvolvimento, como durante a vida útil do sistema, facilitando a sua manutenção. A facilidade de alteração resulta da possibilidade de utilizar ferramentas de desenvolvimento que permitem simular, validar e gerar automaticamente o código de implementação, possibilitando a reconfiguração dos dispositivos de implementação, evitando (frequentemente) a sua substituição.

Tal como para os sistemas embutidos, o desenvolvimento de sistemas embutidos distribuídos (SEDs) também beneficia do uso de abordagens de desenvolvimento baseadas em modelos e do uso de dispositivos reconfiguráveis.

Este trabalho (ainda em desenvolvimento) contribui para o uso de abordagens de desenvolvimento para SEDs baseadas em redes de Petri (RdP), propondo uma forma de dimensionar os *buffers* necessários para a infraestrutura de comunicação (de sistemas embutidos síncronos) com topologia ponto a ponto. Este trabalho contribui assim para o desenvolvimento de SEDs onde cada componente do sistema é síncrono com um sinal de relógio, tornando o SED num sistema globalmente assíncrono localmente síncrono (GALS) [5].

Num trabalho anterior [6] foi proposto o dimensionamento de *wrappers* assíncronos [7] para a interligação de componentes de sistemas GALS previamente especificados através de redes de Petri. Este artigo complementa o referido trabalho, permitindo o dimensionamento de *buffers* para nós de uma rede com topologia ponto a ponto, em que os *wrappers* assíncronos foram substituídos por nós com comunicação série assíncrona.

2. Especificação de sistemas GALS

A classe de redes de Petri Input-Output Place-Transition (RdP-IOPT), que foi proposta em [8] para especificar sistemas embutidos e sistemas de automação, foi estendida em [9] para permitir a especificação de sistemas distribuídos GALS. Neste trabalho utilizam-se RdP-IOPT para especificar sistemas embutidos distribuídos globalmente assíncronos localmente síncronos (SED-GALS).

Na figura 1 apresenta-se um exemplo de um sistema GALS especificado através de RdP-IOPT,

onde a interação entre os componentes síncronos é especificada através de canais assíncronos (ACs) (representados por nuvens ligadas a transições de componentes distintos através de arcos a tracejado). A identificação do componente a que pertence cada nó da RdP é feita indicando o domínio temporal (*td*).

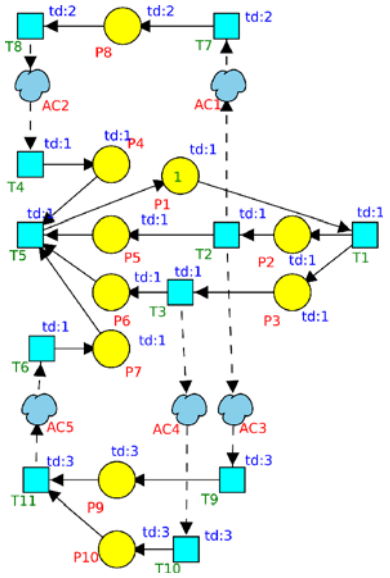


Fig. 1. RdP com a especificação de um sistema embutido distribuído GALS composto por 3 componentes.

3. Implementação de sistemas GALS utilizando redes ponto a ponto

Para implementar um sistema GALS é necessário implementar cada um dos seus componentes e a infraestrutura de comunicação, que neste trabalho será uma rede com topologia ponto a ponto.

A partir da especificação do sistema GALS utilizando RdP-IOPT é possível gerar automaticamente código VHDL de cada componente utilizando a ferramenta PNML2VHDL [10] (ou uma nova ferramenta de geração de código ainda em fase de desenvolvimento). Para isso é necessário retirar os ACs da figura 1 e inserir eventos de saída nas transições que ligavam aos ACs e eventos de entrada nas transições onde os ACs ligavam, obtendo-se assim os três sub-modelos (dos componentes) a partir dos quais se gera o código.

Na figura 2 é apresentada a arquitetura de ligação dos componentes utilizando uma rede com topologia ponto a ponto. Os componentes utilizam nós de envio e de receção para comunicar. As entradas e as saídas dos componentes permitem a interação entre o controlador e o sistema a controlar.

A comunicação entre dois componentes (e com um sentido especifico) que era especificada na figura 1 através de canais assíncronos com o mesmo componente de origem e com o mesmo componente de destino (por exemplo os dois canais assíncronos *AC3* e *AC4*), é feita através de um nó de envio e um nó de receção (ver figura 2).

Na figura 3 é proposta uma arquitetura para o nó de envio e uma arquitetura para o nó de receção.

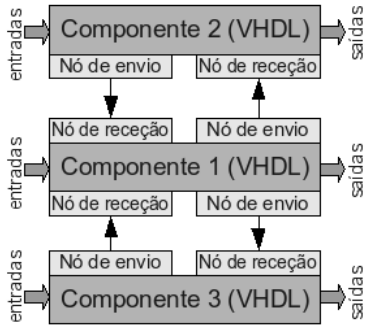


Fig. 2. Arquitetura de ligação entre os 3 componentes da figura 1 utilizando uma rede ponto a ponto.

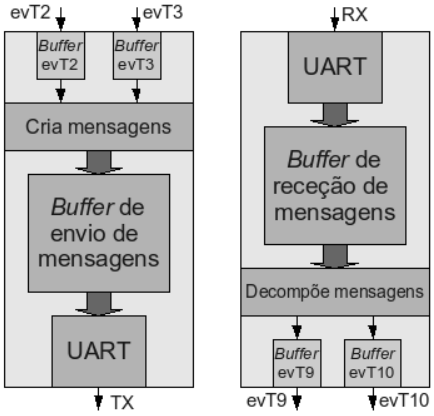


Fig. 3. Arquitetura do nó de rede para envio de eventos (à esquerda), e a arquitetura do nó de rede para receção de eventos (à direita).

No nó de envio (parte esquerda da figura 3), existe: um conjunto de *buffers* (*Buffer evT2*, *Buffer evT3*) para guardar o número de ocorrências de cada evento (*evT2*, *evT3*) criado pela respetiva transição; e um *buffer FIFO* para armazenar as mensagens que serão enviadas (uma mensagem por cada ocorrência de um evento específico).

No nó de receção (parte direita da figura 3), existe um *buffer FIFO* para armazenar as mensagens recebidas e um conjunto de *buffers* (*Buffer evT9*, *Buffer evT10*) para armazenar o número de ocorrências de cada evento (*evT2*, *evT3*).

No sentido de garantir uma implementação robusta de todo o sistema, antes de implementar os nós de comunicação apresentados na figura 3 é necessário fazer o dimensionamento dos seus *buffers*.

 ISBN: 978-972-8822-27-9

4. Dimensionamento de *buffers* para redes com topologia ponto a ponto

Para dimensionar os *buffers* dos nós de comunicação é necessário saber quantos eventos podem estar a ser enviados (ou ainda em transito) num determinado momento. Em particular é necessário saber: (1) para cada um dos eventos gerados pelos componentes (*evT2*, *evT3*, *evT11*, etc.), quantos podem estar em simultâneo entre dois pontos da rede; (2) qual o número máximo de eventos (tendo em conta todos os eventos) em simultâneo entre dois pontos da rede.

O número de eventos gerados por uma transição (por exemplo *evT2*) que podem estar a circular em simultâneo na rede, é a dimensão do *buffer* respetivo (por exemplo *Buffer evT2 e Buffer evT9*).

Tendo em conta que o espaço de estados (gerado a partir de modelos IOPT-GALS com a ferramenta [11]) inclui em cada nó o número de eventos a circular em cada canal assíncrono, a dimensão do B*uffer evTx* (db_x) pode ser obtida através da análise do espaço de estados, e é dada pela máxima marcação observada (*bound*) (ver [8]) nos canais assíncronos respetivos (equação 1).

$$db_x = max\left(\forall_{m\in[0..n]}\left(M_m(AC_x)\right)\right) \quad (1)$$

Onde m é a ordem do nó do espaço de estados associado, representando o estado de marcação global; $n+1$ é o número de nós do espaço de estados; M_m é a marcação do nó m; e x é a referência do canal assíncrono envolvido na interligação dos dois componentes. Assim $M_m(AC_x)$ é a marcação do canal x no nó m. Na tabela 1 é apresentado o *bound* de cada um dos canais e a dimensão dos *buffers respetivos.*

	Bound	Dim. do *buffer* (db_x)
Canal assínc. AC1	1	1
Canal assínc. AC2	1	1
Canal assínc. AC3	1	1
Canal assínc. AC4	1	1
Canal assínc. AC5	1	1

Tabela 1. *Bound* dos canais assíncronos da RdP da figura 1, e dimensão dos *buffers* respetivos.

O número máximo de eventos que podem estar a circular entre dois pontos da rede é igual à dimensão dos *buffers* de envio (dbe_{xy}) e de receção (dbr_{xy}) da figura 3, e é igual ao máximo da soma das marcações dos canais assíncronos (que interligam o ponto x ao ponto y) em cada estado do espaço de estados (equações 2 e 3).

$$dbe_{xy} = dbr_{xy} \quad (2)$$

$$dbe_{xy} = max\left(\forall_{m\in[0..n]}\left(\sum_{i=0}^{k_{xy}} M_m(AC_{w_i})\right)\right) \quad (3)$$

Onde m é a ordem do nó do espaço de estados associado; $n+1$ é o número de nós do espaço de estados; k_{xy} é o número de canais assíncronos que interligam os dois componentes (com um sentido específico); M_m é a marcação do nó m; e w_i é a referência do canal assíncrono envolvido na interligação dos dois componentes. Assim $M_m(AC_{w_i})$ é a marcação do canal w_i no nó m.

A tabela 2 apresenta os canais assíncronos que existem entre os vários componentes do sistema GALS, e a dimensão dos *buffers* de envio e receção.

Origem (*x*) -> Destino (*y*)	**Canais assíncronos**	$dbe_{xy} = dbr_{xy}$
Comp. 1 -> Comp. 2	*AC1*	*1*
Comp. 2 -> Comp. 1	*AC2*	*1*
Comp. 1 -> Comp. 3	*AC3*, *AC4*	2
Comp. 3 -> Comp. 1	*AC5*	*1*
Comp. 2 -> Comp. 3	-	-
Comp. 3 -> Comp. 2	-	-

Tabela 2. Canais assíncronos entre os vários componentes do sistema GALS especificado na figura 1, e a dimensão dos *buffers* de envio (dbe_{xy}) e receção (dbr_{xy}).

Utilizando a ferramenta de verificação das RdP-IOPT estendidas para sistemas GALS [11] foi gerado o espaço de estados (ver figura 4) do modelo da figura 1. O espaço de estados tem 95 estados, não tem bloqueios nem conflitos. A análise do espaço de estados permitiu dimensionar os *buffers* de acordo com a informação apresentada nas tabelas 1 e 2.

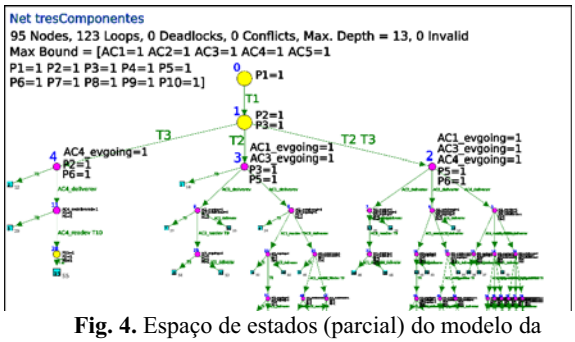


Fig. 4. Espaço de estados (parcial) do modelo da figura 1.

O sistema especificado na figura 1 foi implementado em FPGAs, utilizando comunicação série assíncrona, satisfazendo do ponto de vista lógico o protocolo de comunicação RS-232.

Nas tabelas 3 e 4 apresentam-se os recursos ocupados na *Spartan-3 Starter Kit Board* da *Xilinx,* pelo nó de envio e pelo nó de receção apresentados na figura 3, que implementam os canais *AC3* e *AC4.*

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	16	3,840	1%
Number of 4 input LUTs	28	3,840	1%
Logic Distribution			
Number of occupied Slices	18	1,920	1%
Number of Slices containing only related logic	18	18	100%
Number of Slices containing unrelated logic	0	18	0%
Total Number of 4 input LUTs	28	3,840	1%

Tabela 3. Recursos da *Spartan-3* ocupados pelo nó de envio para o dimensionamento obtido para *AC3* e *AC4*.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	70	3,840	1%
Number of 4 input LUTs	99	3,840	2%
Logic Distribution			
Number of occupied Slices	64	1,920	3%
Number of Slices containing only related logic	64	64	100%
Number of Slices containing unrelated logic	0	64	0%
Total Number of 4 input LUTs	99	3,840	2%

Tabela 4. Recursos da *Spartan-3* ocupados pelo nó de receção para o dimensionamento obtido para *AC3* e *AC4*.

4. Conclusões e trabalhos futuros

O método apresentado neste artigo suporta a implementação de nós de comunicação para sistemas GALS utilizando redes de comunicação com topologia ponto a ponto. Este método está incluído numa abordagem de desenvolvimento baseada em modelos, que utiliza redes de Petri como formalismo de modelação.

O método permite dimensionar os *buffers* necessários à implementação de nós de comunicação, independentemente do protocolo de comunicação. O sistema GALS apresentado neste artigo foi implementado em dispositivos reconfiguráveis (FPGAs), utilizando comunicação série assíncrona.

Ainda no decorrer deste trabalho será proposto um algoritmo para obter todos os canais assíncronos entre dois componentes num determinado sentido, e para determinar qual o estado do espaço de estados em que a soma das marcações de todos os canais entre dois componentes é máximo. O algoritmo será então implementado na ferramenta de verificação das IOPT-tools [12] (disponível online em http://gres.uninova.pt/). Será ainda estudado o dimensionamento de redes de comunicação com mensagens de confirmação e reenvio de mensagens.

Com trabalhos futuros será estudada a forma de dimensionar redes com outras topologias de rede, nomeadamente a topologia em anel e a topologia em barramento (bus).

Agradecimentos

Este trabalho foi parcialmente financiado por Fundos Nacionais através da Fundação para a Ciência e a Tecnologia (FCT) no âmbito do projeto PEst-OE/EEI/UI0066/2011.

O trabalho do primeiro autor foi suportado por uma bolsa da Fundação para a Ciência e a Tecnologia (FCT), referência SFRH/BD/62171/2009.

Referencias

[1] B. Schatz, A. Pretschner, F. Huber & J. Philipps, "Model-based development of embedded systems", Advances in Object-Oriented Information Systems Workshops, Springer LNCS, Montpellier, France.

[2] D. de Niz, G. Bhatia, and R. Rajkumar, "Model-Based Development of Embedded Systems: The SysWeaver Approach," in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium. Washington, DC, USA, 2006.

[3] C. Bunse, H.-G. Gross, and C. Peper, "Applying a Model-based Approach for Embedded System Development," in the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications. Washington, DC, USA, 2007.

[4] L. Gomes, J. P. Barros, A. Costa, R. Pais, and F. Moutinho, "Towards usage of formal methods within embedded systems co-design," in 10th IEEE Conference on Emerging Technologies and Factory Automation, Univ. Catania, Italy, September 2005.

[5] Chapiro, DM 1984, 'Globally-Asynchronous Locally-Synchronous Systems', PhD thesis, Stanford University.

[6] F. Moutinho, J. Pimenta, L. Gomes, "Dimensionamento da infraestrutura de comunicação em sistemas GALS especificados através de redes de Petri", VIII Jornadas sobre Sistemas Reconfiguráveis, Instituto Superior de Engenharia de Lisboa, 2012.

[7] H. Ferreira, 2010, "Petri Nets Based Components Within Globally Asynchronous Locally Synchronous systems", Master's thesis, Universidade Nova de Lisboa.

[8] L. Gomes, J. P. Barros, A. Costa & R. Nunes, 2007, "The Input-Output Place-Transition Petri Net Class and Associated Tools", Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN'07), Vienna, Austria.

[9] F. Moutinho & L. Gomes, 2012, "Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems", Proceedings of the 3th Doctoral Conference on Computing, Electrical and Industrial Systems (DoCEIS'12), Costa da Caparica, Portugal.

[10] L. Gomes, A. Costa, J. Barros & P. Lima, "From Petri net models to VHDL implementation of digital controllers", in the 33rd Annual Conference of the IEEE Industrial Electronics Society, Taiwan, 2007

[11] F. Moutinho, & L. Gomes, 'State Space Generation Algorithm for GALS Systems Modeled by IOPT Petri Nets', in the 37th Annual Conference of the IEEE Industrial Electronics Society, Australia, 2011

[12] F. Pereira, F. Moutinho, and L. Gomes, "Model-checking framework for embedded systems controllers development using iopt petri nets," in 2012 IEEE International Symposium on Industrial Electronics (ISIE), May 2012, pp. 1399 –1404.

Accelerating user-space applications with FPGA cores: profiling and evaluation of the PCIe interface

Adrian Matoga, Ricardo Chaves, Pedro Tomás, Nuno Roma
INESC-ID, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{*Adrian.Matoga,Ricardo.Chaves,Pedro.Tomas,Nuno.Roma*}*@inesc-id.pt*

Abstract

This paper describes the process of implementing a custom accelerator in a reconfigurable device with the PCI Express interface, and exposing its functions to user space applications. The process includes attaching the computing unit(s) to the DMA controller and the PCIe interface IPs, followed by writing appropriate kernel driver and a user space library. A case study is presented, in which an AES encryption IP core is used to accelerate the EncFS encrypted file system on Linux on a hardware platform consisting of an Intel Atom CPU and an Altera Arria II FPGA, connected with 1-lane PCIe interface. The implementation of the interface between the accelerator and software applications is capable of data transfers of over 190 MB/s, and the overall speedup of the encryption procedure compared to an equivalent software implementation from the OpenSSL library is 3. Further optimizations are possible and are also discussed in the paper.

1. Introduction

Custom accelerators implemented in reconfigurable devices are often developed for embedded systems, which are dedicated to small numbers of fixed tasks, and therefore use only minimal Real-Time kernels or no operating systems at all. In such systems, engineers usually have the freedom to design the interface between hardware and software, and the software usually has direct access to the device registers.

More complex systems usually connect multiple devices through one or more industry standard interfaces eventually providing virtual memory to isolate multiple processes running on them. Taking advantage of these features requires more effort, as the protocol specifications and the programming interface defined by the operating system must be strictly followed. Such systems have already been very common for many years, and the protocols are becoming increasingly complex with their subsequent revisions. Therefore the task of implementing and integrating a custom accelerator may be challenging for a researcher who has little experience with this particular set of technology and just wants to get the things done.

In this paper, a complete implementation of a PCI Express-based accelerator for the Advanced Encryption Standard (AES) is described, including the evaluation of its performance and discussion of the results. Although the PCIe interface has been an industry standard for about 9 years, it still seems to have found relatively little interest in the scientific community of the embedded systems area. Initially, it might be caused by the complexity of the specification, making the development of the PCIe-enabled devices from scratch difficult and costly. However, the situation has changed. Nowadays, top FPGA vendors have multiple devices available in their portfolio that provide easily configurable standard components implementing the PCIe standard (see for example [1] and [2]). Furthermore, although the architecture of the bus is fundamentally different than the older parallel PCI standard, the software protocol remains compatible with other PCI variants, which greatly facilitates the development of software, especially device drivers, for PCIe devices.

With this observation in mind, a large part of the paper focuses on the discussion of the issues related to the development of the Linux device driver and with its integration with the accelerating core. Instead of crafting a specialized driver, a generic PCI driver was used, which made it possible to develop the custom drivers in the user space with significantly less effort. It is also worth mentioning that although the Altera components and tools are referred to throughout the paper, equivalent components and tools can be also found for Xilinx devices.

The remaining part of this paper is organized as follows. The next section gives an overview of the considered case study by briefly introducing the software application that was accelerated and by characterizing the IP core implementing the equivalent algorithm. Section 3 discusses the architecture of the entire hardware design, i.e. the configuration of all the components that turns the IP core into a useful accelerator. Section 4 focuses on the layered implementation of the device driver, which makes the accelerator functions available to user applications. Section 5 shows the results of the performance evaluation of the entire system. Section 6 briefly discusses alternative design options and their preliminary evaluation. Finally, Section 7 concludes and gives directions for the future work.

2. Case study: EncFS

2.1. Software application

The accelerated application is EncFS (Encrypted Filesystem) [3]. It is based on FUSE (Filesystem in

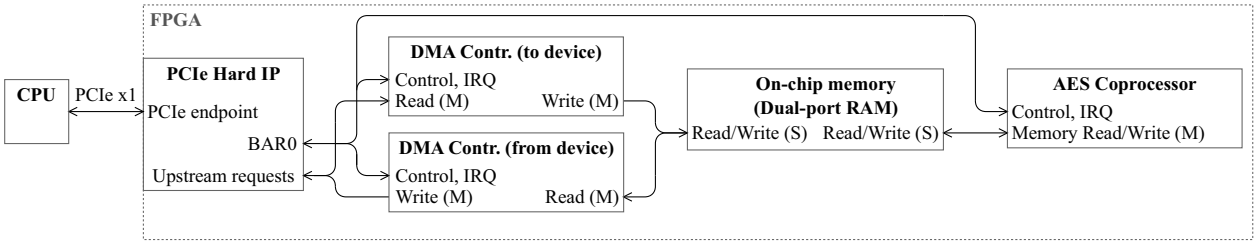


Figure 1. Architecture of the memory-based accelerator. M represents an Avalon MM Master port (one that initiates transactions), whereas S represents a Slave port (one that responds to Master's requests).

USErspace) and works by providing an unencrypted view of each encrypted file from its underlying storage directory under the mount point. EncFS is written in C++ and its architecture permits adding ciphers, by implementing the necessary methods for key setup and data encoding and decoding. It originally uses the OpenSSL library to provide encryption using the AES cipher in the Cipher-Block Chaining (CBC) mode of operation with a 128-, 192- or 256 bit key.

2.2. The AES IP core

The cryptographic core used in this work is the AES coprocessor proposed in [4], which implements the AES cipher, first proposed in [5]. Encryption and decryption with 128-, 192- and 256-bit keys, as well as the Electronic Codebook (ECB) and CBC cipher modes of operation are implemented. The core is able to execute one round of the AES algorithm per clock cycle, thus requiring 10, 12 or 14 cycles per 16-byte block, depending on the key length. With the operating frequency of 125 MHz, it provides a throughput of 190 MB/s, 159 MB/s and 136 MB/s for key lengths of 128, 192 and 256 bits, respectively.

3. Hardware architecture

The architecture of the developed accelerator is presented in Figure 1. It was designed using the Altera Qsys system design tool and uses the Avalon Memory Mapped (Avalon MM) interface [6] to interconnect the several components in the FPGA.

The AES core was equipped with the appropriate logic to read the keys and the input data from the on-chip memory and store the results therein. Likewise, a set of control registers for the configuration of the buffer addresses and the operation mode, as well as an interrupt signal source to inform the CPU when the encryption is done were provided.

The use of a dual port on-chip RAM simplified the interface between the AES core and the memory, for it was not necessary to implement any control logic to negotiate the access of multiple masters to the memory.

Since the transfers of large amounts of data are done more efficiently with the Direct Memory Access mechanism, the design includes two Scatter/Gather DMA controller cores, one per each direction. The Scatter/Gather capability is needed to compensate for the non-contiguous mapping of the virtual memory pages in the physical memory. Since the DMA cores available in the Qsys component library have the bus management logic already implemented, they can share the second port of the on-chip memory. The system also includes two additional instances of on-chip RAM (one per DMA controller), used to store the descriptor lists. For clarity purposes, they are omitted from the figure, as logically they belong to the relevant controllers.

The configuration of the PCI Express Hard IP core [1], serving as the bridge between the PCIe interface and the Avalon MM interface, needs slightly more attention. Its functions include: *a)* the PCI configuration space, containing the information required by the operating system to identify the device and map its I/O ports and/or memory into the CPU physical address space; *b)* a simple interrupt controller, allowing multiple Avalon interrupt sources to share a single PCIe interrupt request signal; *c)* a configurable address translation table, allowing the whole 32- or 64-bit PCIe bus address space to be addressed using the much narrower address space of the Avalon MM slave port. Further discussion of the mapping between the different address spaces is given in Section 4.1.

The design built in Qsys requires some additional support in the top-level module, whose details are covered in [1].

4. Software

4.1. DMA and address translation

The major difficulty when implementing an efficient PCI device driver is not related to the bus protocol itself, for which the Linux kernel offers a relatively simple API (Application Programming Interface), described in detail in [7]. What increases the conceptual complexity (and sometimes also causes performance loss), is the existence of multiple different address spaces in the system. Different APIs are used to access the memory and registers located in different address domains. Mapping ranges of addresses between different domains sometimes requires building complex data structures. If the mapping cannot be done by hardware, data must be copied. This section briefly discusses the most important issues related to address translation and the solution commonly implemented in order to efficiently exchange the data between the user process and the hardware.

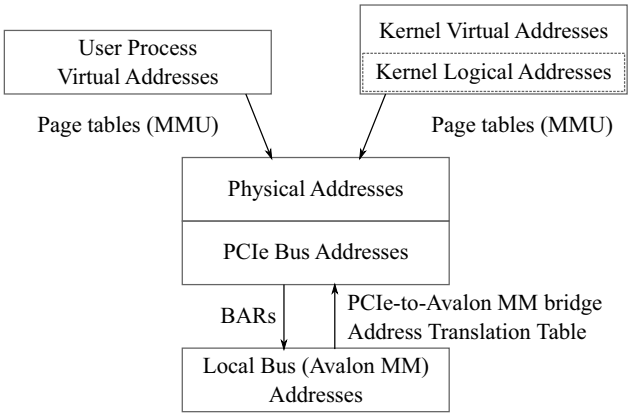


Figure 2. Address domains and translations between them.

The exact structure of the relations between these address domains depends on the system architecture. Figure 2 illustrates the set of the address domains in the system where the accelerator was evaluated. This structure would be even more complex on an architecture with an IOMMU (Input/Output Memory Management Unit), translating the addresses between the physical address space and the bus address space (here, they are the same addresses), or if the I/O memory could not be mapped into virtual address space. A portable driver should be aware of these issues and use the proper API for every type of address [7].

The kernel logical addresses (see Figure 2) are a part of the kernel virtual memory, where the physical memory (or a part of it) is linearly mapped, possibly with some offset. This region of memory is used whenever a contiguous buffer needs to be allocated. Such buffers generally make a DMA transaction easier to configure (with the simplest DMA controllers they are the only possible option), but require either a modification of the application, so that it requests the allocation of a buffer within the kernel space and `mmap`s it to the process space, or a copy of the data to be made between the user memory and the kernel buffer.

In many cases, however, the programmer does not have the opportunity to change the way the buffer is allocated, because the memory may be allocated by a different software module. This is the case in the EncFS, where the encryption function only receives the pointer to the buffer and the data length. Making the cipher implementation able to preallocate the buffer would require major changes in the design of the whole application. The solution here is to map the user memory buffer into the bus address space. However, the virtual memory pages are almost certainly scattered all over the physical memory, and are usually small (4 KB in x86-based systems). To avoid the need to transfer each page separately, a Scatter/Gather DMA controller may be used. The SG-DMA controller reads a singly linked list of descriptors, containing the addresses and sizes of subsequent blocks, and transfers the data from or to all of them in a single transaction, without further CPU intervention.

4.2. Linux device driver

Device drivers are usually written with some specific, fixed hardware in mind, and therefore their interface offers a well-defined and closed set of functions. However, implementation devices based on FPGAs offer a high level of customizability, and so it is desirable for the driver to be more flexible.

The "flexibility" here means the possibility to adjust the functions of the driver to the modified device configuration, without replacing the driver module in the kernel. It also means that the driver interface should not be obscured due to the limitations of standard device driver interfaces of the operating system, such as the need to use the `ioctl()` calls to implement arbitrary operations outside the usual semantics of `read()` and `write()`.

In a monolithic kernel, such as Linux, such flexibility can be achieved by implementing a generic PCI device driver, which does not focus on any particular function of the device, but makes the device registers available to user space processes, so that they can program them directly. A very well-known example of such universal driver is the commercial Jungo WinDriver [8]. Simpler and open-source drivers can also be found, such as the generic PCI driver from the MPRACE framework [9], which was chosen as the basis for the user-space drivers described in the next section.

The driver is composed of two parts: the actual kernel driver, exposing its functions through a character device offering a number of `ioctl`s to control the driver, and a C++ library, providing a convenient programming interface for the user applications. The main functions of the driver include: *a)* device detection and initialization; *b)* creation of the device nodes (/dev/fpga*n*); *c)* mapping the memory regions defined by the Base Address Registers (BARs) into user process address space; *d)* allocating contiguous buffers in the kernel memory and mapping them as DMA buffers available to the device and to the user process; and *e)* pinning user-allocated virtual memory buffers in the physical memory and mapping them to the PCIe bus addressing space.

Although the first prototype of the AES accelerator was evaluated with a "traditional" kernel-mode driver, the migration to the generic driver architecture made it possible to try out more different configurations in a shorter time. Most likely, it was because the memory mapping intricacies were hidden behind the simple C++ interface, which had already been tested and debugged.

The original driver required a slight change to make it recognize the Altera PCIe devices and a more significant modification of the interrupt handling subsystem. This latter part is difficult to be implemented generically in the user space, because interrupt acknowledgment must be performed from within the interrupt service routine, which cannot execute outside of the kernel space.

The Altera PCIe Hard IP supports up to 16 distinct Avalon MM interrupt sources. They can be enabled separately, even from the user space, by setting the relevant bits of the interrupt control register. After the specific interrupt

is enabled in the PCIe-to-Avalon MM bridge and the appropriate core is configured to generate interrupts, the application should request the driver to wait for an interrupt. The approach that was implemented in the original driver for some boards was to expect the application to specify the interrupt source to wait for. In the Altera variant, however, the driver catches any interrupt that arrives first, disables all interrupts on the PCIe bridge, and returns the mask of all interrupt signals that were asserted when the interrupt service routine started. It is then up to the user space program to properly handle them, end re-enable them on the bridge if necessary.

Although this semantic difference may seem confusing, it is not against the original authors' assumption, which states that the interrupt handling code would be specific to particular devices. It also makes the kernel part of the driver shorter and usable for virtually all devices using the standard library component for the PCIe bridge.

It is worth to remind that with great power comes great responsibility, and providing the access to device registers to the user-space application makes it able to set up, for example, a DMA transaction that will overwrite the kernel memory with arbitrary data. This may result in all kinds of unwanted effects, from system instability to data loss to major security compromises. Therefore, care must be taken to properly test and secure the application using such driver before giving it to the end user. The user-space drivers described in the next section help to address this issue by ensuring that all buffer mappings are valid for the entire time of the DMA transaction.

Another important note is that the operation of the generic driver relies on the assumption that the PCI I/O regions can be mapped into the CPU memory addressing space. Although this is true for the x86 platform, it is not fully portable, and the access to the device registers in future versions of the driver should instead be provided by wrapping the appropriate kernel API.

4.3. User space drivers

A set of classes was specifically developed to support the Altera components (the PCIe-to-Avalon MM bridge and the SG-DMA controller) and the cryptographic core. They use the C++ API of the generic PCI driver to access the registers of all cores in the FPGA, request the memory mappings and configure and handle the interrupts. The RAII (Resource Acquisition Is Initialization) idiom and smart pointers are used throughout the code, in order to minimize the likelihood of resource leakages and memory corruption errors.

4.3.1. PCIe-to-Avalon MM bridge

The programming interface of the device drivers for the Altera IP cores is summarized in Figure 3. The `AltPCIe` class encapsulates the access to the PCIe-to-Avalon MM bridge. The constructor accepts three parameters: the minor number of the char device node, the BAR where the control registers of the bridge are located, and the base address of the control register block within that BAR. This way, the location where the software expects the registers can be easily adjusted to the actual configuration of the system implemented in the FPGA. The `AltPCIe` class provides wrappers for the memory mapping calls of the `PciDevice` class of the generic driver, which keep track of all allocated resources. It also gives the access to the Avalon MM to PCIe address translation table and the interrupt controller.

```
class AltPCIe {
  AltPCIe(unsigned dev, unsigned bar,
    unsigned long offset);
  void* getMappedBar(unsigned bar);
  UserMemPtr mapUserMemory(void* buf,
    unsigned size, dma_direction_t dir);
  AddrTable& getAddrTable();
  IrqPtr requestIrq(unsigned irq);
};
class AddrTable {
  PageMapPtr createPageMap();
};
class PageMap {
  unsigned map(unsigned addr,
    unsigned size, unsigned* mapped);
};
class Irq {
  void enable();
  void disable();
  void wait();
};
```

Figure 3. Public API of the user space driver for the Altera PCIe-to-Avalon MM bridge.

The address translation table is managed by the `AddrTable` object, to which a reference can be obtained using the `getAddrTable()` method of the `AltPCIe` object. Entries serving a common purpose (such as mapping a pinned user-space DMA buffer) are managed by a single `PageMap` object, which can be obtained from the `AddrTable::createPageMap()` function. Subsequent portions of the buffer are then mapped by simply calling `PageMap::map()`, providing the PCIe bus address of the buffer and its size. The returned value is the equivalent address on the Avalon MM slave interface, which can be used to set the address for a DMA controller. Entries in the global table are reused if more than one buffer slice falls within the same Avalon MM page, even if they belong to different `PageMap` objects. Each entry has a reference count associated with it, so that it is moved to the free list when the last `PageMap` object using it is deleted.

An interrupt line can be allocated using the `AltPCIe::requestIrq()` function, which returns a reference-counted pointer to an `Irq` object, encapsulating all necessary operations, which include enabling or disabling the interrupt line and waiting until an interrupt occurs on that line.

```
class AltSGDMA {
  AltSGDMA(AltPCIe& altpcie, ...);
  void copyToDevice(const void *srcBuf,
    unsigned destAddr, unsigned size);
  void copyFromDevice(void *destBuf,
    unsigned srcAddr, unsigned size);
  void waitForCompletion();
};
class AltSGDMAPair {
  AltSGDMAPair(AltSGDMA& todev,
    AltSGDMA& fromdev);
  void stream(void *src, void *dest,
    unsigned len);
  void waitForCompletion();
};
```

Figure 4. Public API of the user space driver for the Altera Scatter/Gather DMA controller core.

4.3.2. SG-DMA controller

The API used to control the Altera SG-DMA controller core is presented in Figure 4. The `AltSGDMA` class encapsulates the functionality of the Altera Scatter/Gather DMA controller core. It is configured by specifying the `AltPCIe` object associated with the device, the location (BAR and offset) of the DMA core registers and the descriptor memory, interrupt line and the policy for the `waitForCompletion()` method (polling the device registers in a loop or putting the process to sleep and waiting for an interrupt). The sibling functions `copyToDevice()` and `copyFromDevice()` pin the pages of the virtual memory where the local buffer is allocated to the physical address space, map them into the Avalon MM address space, build the list of descriptors and set the controller registers to start transferring data. Since configuring the above data structures is quite costly, they are kept configured and reused in subsequent calls, provided that the specified buffer lies within the previously mapped area.

The `AltSGDMAPair` class is a convenience wrapper which joins two controllers in a pair that effectively handles the processing in a stream-based architecture. This topic is further covered in Section 6.

Figure 5 demonstrates the use of the just described API, showing that copying data between a user-process memory buffer and the accelerator memory is only a matter of creating an `AltSGDMA` object, configured according to its configuration in the FPGA design, and invoking the functions to start the transfer and wait until it is finished.

The communication diagram in Figure 6 presents the detailed control flow of the implementation of these functions, showing how the SG-DMA driver uses the `AltPCIe` API to map the memory, build the descriptor list and make the controller perform the actual transfer.

4.3.3. The cryptographic IP core

The API of the cryptographic core driver is presented in Figure 7. The `setKey()` function performs the key expansion (in software) and writes the expanded keys into the device memory at a specified address. The `processData()` function initiates the encryption according to the specified parameters. The `waitForCompletion()` function puts the process to sleep until the encryption is complete, which is signaled to the CPU by an interrupt.

```
AltPCIe altpcie(DEV, BAR, CRA_BASE);
AltSGDMA dmaToDev(altpcie, ...
  /* configuration of the SG-DMA core:
     address of the core and descriptor
     memory, Avalon IRQ # */)
void *buf = malloc(BUF_SIZE);
 ... fill buf ...
dmaToDev.copyToDevice(buf, DEV_BUF_ADDR,
  BUF_SIZE);
dmaToDev.waitForCompletion();
```

Figure 5. Example usage of the SG-DMA user-space driver.

As all components have their interfaces split into the "initiation" and "completion" steps, it is easy to implement a pipeline, where at each time instant one portion of data is transferred to the device, another one is encrypted, and yet another one is received from the device. However, such method is only practicable if the application is aware of that possibility. Many applications request the processing of the data in a synchronous manner, which would require dividing the buffer in smaller chunks to take the advantage of pipeline processing.

5. Evaluation

5.1. Evaluation platform

The conceived accelerator was implemented on a Kontron MSMST board incorporating an Intel Stellarton processor, which includes an Intel Atom CPU at 1.3 GHz and an Altera Arria II GX FPGA, connected with two PCIe v1.1 1× links [10]. In the prototype developed in this work, only one bidirectional PCIe link is used, which implies the maximum raw throughput of 250 MB/s (500 MB/s in duplex mode). All cores implemented in the FPGA are driven by the 125 MHz application clock from the PCIe Hard IP core.

5.2. SG-DMA controller performance

To estimate the data transfer throughput limitations of the accelerator, the performance of the SG-DMA controllers was evaluated first. The results are shown in Figure 8. The dashed lines show the throughput with the memory mapping and descriptor list configured before each DMA transaction. The solid lines show the throughput with these steps omitted, as when the buffer is allocated and mapped once and then reused for subsequent data transfers.

The overhead of mapping the memory and creating the descriptor list can be as high as hundreds of microseconds. It is mostly caused by copying the list of page addresses

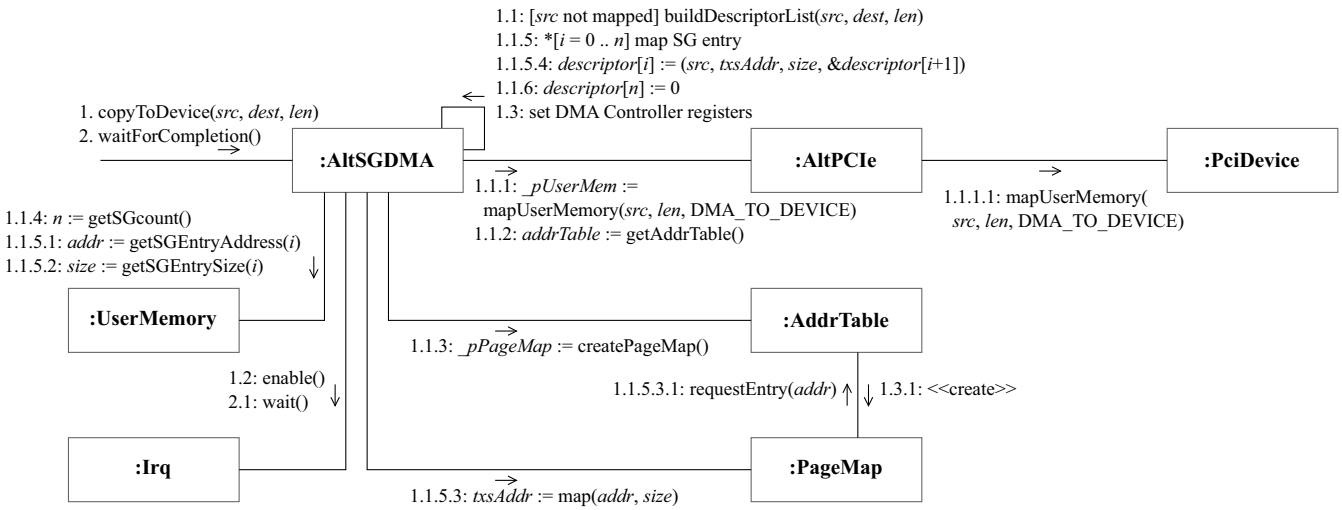


Figure 6. A communication diagram presenting the interaction between the SG-DMA controller driver and the objects of the PCIe-to-Avalon MM bridge driver during a single DMA transfer from the primary memory to the accelerator memory.

```
class AESCore {
  AESCore(AltPCIe& pcie, unsigned bar,
    unsigned long offset);
  enum Mode { MODE_ECB, MODE_CBC };
  void setKey(const void* key,
    unsigned keyAddr, unsigned rounds);
  void processData(unsigned srcAddr,
    unsigned destAddr, unsigned nbytes,
    unsigned keyAddr, unsigned rounds,
    Mode mode, bool decrypt);
  void waitForCompletion();
};
```

Figure 7. Summary of the public API of the AES encryption core user-space driver.

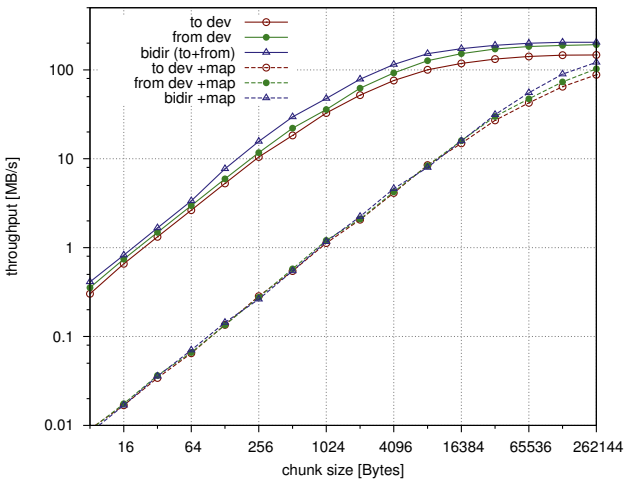


Figure 8. Throughput of the SG-DMA controllers in memory-to-memory transfer mode.

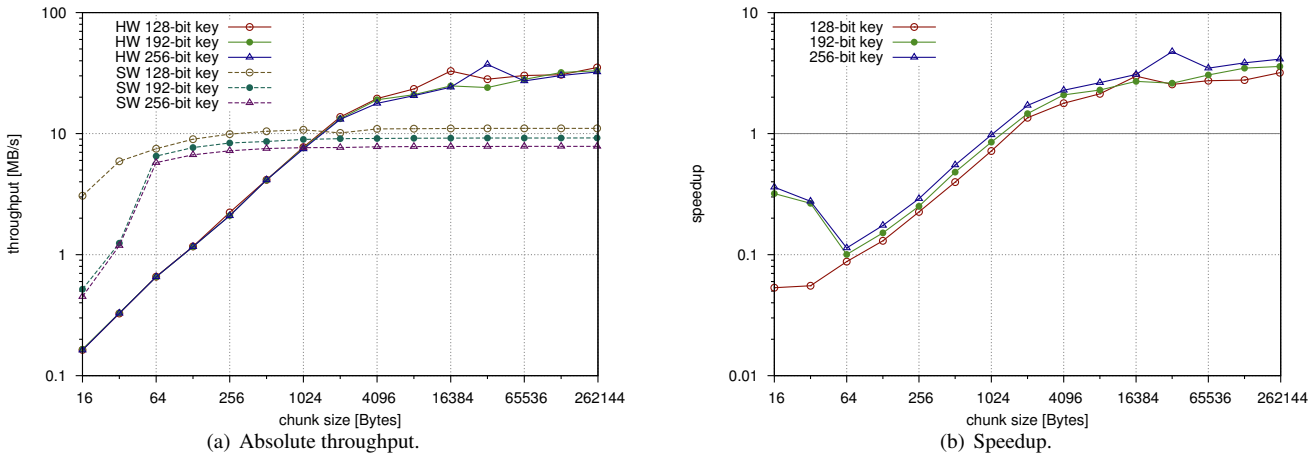
and sizes from kernel to user space, and writing considerable amount of data to device registers and memory using programmed I/O (i.e. separate transaction for each word of data). Since heap memory allocation and deallocation is also considered a costly operation in purely software applications, it is generally avoided, which makes the need for creating a new mapping rare. Therefore only the case where the buffer mapping is reused will be considered.

As it can be seen, the overhead of the transaction initialization and completion, amounting to about 20 μs per transaction, seriously harms the performance for transfers of up to tens of kilobytes. This suggests that, whenever possible, multiple requests to process small amounts of data should be merged into a single, bigger transaction. Otherwise, the additional cost of transferring the data may outweigh the benefit of using the accelerated implementation. As a consequence, it may be better to still use its software equivalent for smaller chunks.

While the data rate of the transfers from the device (reaching 191 MB/s), seems to be mainly limited by the maximum raw throughput of the PCIe bus, the transfers in the opposite direction are significantly slower, with a maximum of about 147 MB/s. This difference may result from the fact that a transfer to the device is conducted as a series of read transactions initiated by the PCIe endpoint, and each of them is composed of a request and a response. The round-trip time of such transaction is naturally higher than the time of a purely unidirectional write transaction. Likewise, the attempt to simultaneously transfer the data in both directions results in a total throughput which is only slightly larger than the transfer in only one direction. The read requests not only have their own latency, but they also "steal" some bandwidth from the reverse link.

5.3. Comparison with OpenSSL

The overall performance of the accelerator was evaluated by measuring the total time required to transfer the data to the accelerator memory, process them, and then re-



(a) Absolute throughput. (b) Speedup.

Figure 9. Throughput comparison between the cryptographic core (HW) and software AES implementation from OpenSSL (SW) for different key lengths.

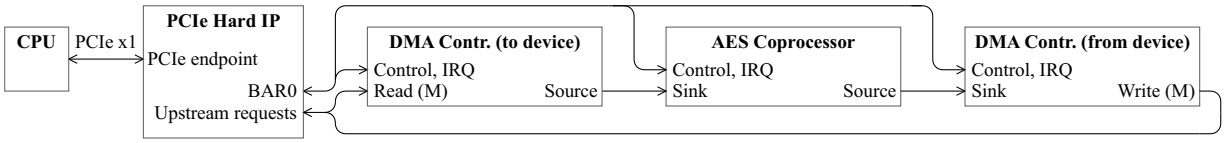


Figure 10. Architecture of a stream-based accelerator.

ceive the encrypted data. The keys were assumed to have been set previously. The throughput measured this way was compared with the overall throughput of the equivalent software implementation from the OpenSSL library (version 1.0.0e), and the results are shown in Figure 9(a). Both implementations run in the CBC mode. The smallest chunk size is 16 bytes, which is the size of the block encrypted at once by the AES algorithm. The largest chunk size is 256 KB, which is the limit imposed by the size of the on-chip RAM.

As it can be seen from the graph, the throughput of the software implementation is almost constant for chunks larger than 64 bytes, reaching about 11, 9.2 and 7.8 MB/s for 128-, 192- and 256-bit key, respectively.

Since the DMA transfers take a major part of the total time, the throughput difference of the AES core in what concerns the key length becomes insignificant. The overall throughput in the useful region, where the accelerator is faster that the software implementation, is from 14 MB/s for 2-KB chunks to 28 MB/s for 256-KB chunks, thus giving a speedup of about 3 over the software implementation (see Figure 9(b)).

To evaluate the actual speedup that a real application would get from such accelerator, an EncFS file system was mounted using the software AES implementation with 4096-byte blocks and a 256-bit encryption key. The Linux 2.6.35 source code (34 MB of data, distributed among several files and directories of different sizes) was used for benchmarking. The files were copied into the file system and the time spent in the encryption was measured. The whole operation took 22 seconds, out of which the encryption only took 6.8 s. The encryption of chunks of 2 KB and larger (i.e. those, for which the use of the accelerator makes sense) took 3.43 s (15.6% of the total time). Assuming the $2\times$ speedup for such chunks, the overall speedup is about 1.08.

6. Stream-based accelerator

Greater performance increases may still be obtained by replacing the memory-based architecture with a stream-based one, as shown in Figure 10. The key difference is that the DMA controllers do not copy the data from one memory to another, but convert the sequence of words read from memory into a stream and vice versa. The main advantages of such approach are that it does not require any additional memory blocks in the FPGA (except for relatively small FIFOs), and the latency is much shorter, because the data no longer needs to be copied.

The AES core has a few important features that make such an architecture the right choice for it. First and foremost, the nature of the algorithm, transforming an arbitrarily long stream of plain-text data into an encrypted stream of the same length (or vice versa) just fits naturally into such architecture. Second, the latency of 10 to 14 cycles between the output and the input is very low, when compared with the other sources of overhead in the system.

Figure 11 predicts the upper performance limit of a stream-based accelerator implemented on the evaluation platform. It was obtained by measuring the buffer-to-buffer throughput with the *Source* port of the "to device" DMA controller connected directly to the *Sink* port of the "from device" DMA controller. Since the AES core is still significantly faster than those controllers, the graph shows a realistic estimate of the hypothetical performance of a stream-based AES accelerator. This not only further multiplies the

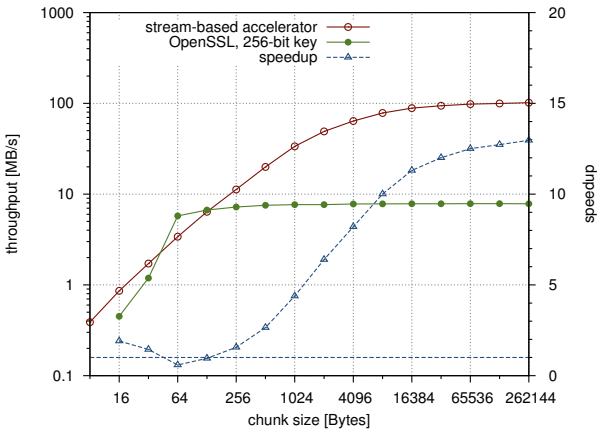


Figure 11. Maximum achievable buffer-to-buffer throughput of a stream-based accelerator, compared to the software implementation of AES from OpenSSL.

maximum throughput by a factor of 4, but it also makes the accelerator useful for almost every size of data. In the benchmark from the previous section, encryption of 16-byte chunks corresponded to about 3.1 seconds, i.e. another 14% of the total time. Thus, the overall speedup obtained from the use the stream-based accelerator in that benchmark is estimated to be over 1.25.

7. Conclusions and future work

The design, implementation and evaluation of an FPGA-based accelerator with the PCIe interface have been described in this paper. It has been shown that standard library IP cores can be used to implement an efficient interface for a custom accelerator core. A generic PCI device driver has been used to make the accelerator functions available for software, and the user-space driver approach has proven to be a useful method of device driver development. The developed drivers have a clean and compact programming interface.

The evaluation of the provided performance of the conceived accelerator has shown that the overhead associated with initiating and completing the execution of the accelerator functions has significant impact on the effective throughput for small data sizes (up to tens of kilobytes). Although the implemented AES accelerator achieved the speedup of up to 3 over the software implementation from OpenSSL, it performed worse than OpenSSL when the considered file system requested the encryption of small chunks.

A preliminary evaluation of a stream-based accelerator architecture has indicated that using such an architecture potentially results in significantly higher performance, because of the reduction of the latency and the setup/completion overhead. Thus, future work will include further studies involving stream-based accelerators, including their chaining and multiplexing.

In order to conform to the terms of the GNU General Public License, as well as to promote the continuation and use of the work presented here by other researchers and end users, the source code of the modified generic PCI driver and the user-space drivers for the IP cores used in the accelerator will be made available at `http://github.com/epi/altpcie`.

8. Acknowledgments

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project "HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis" (reference number PTDC/EEA-ELC/113999/2009), project "Threads: Multitask System Framework with Transparent Hardware Reconfiguration" (reference number PTDC/EEA-ELC/117329/2010) and project PEst-OE/EEI/LA0021/2011.

References

[1] IP Compiler for PCI Express User Guide. Altera, May 2011.

[2] LogiCORE™ IP Endpoint Block Plus v1.14 for PCI Express® User Guide. Xilinx, April 2010.

[3] EncFS Encrypted Filesystem. http://www.arg0.net/encfs.

[4] Ricardo Chaves, Georgi Kuzmanov, Stamatis Vassiliadis, and Leonel Sousa. Reconfigurable memory based aes co-processor. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 192–192, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[6] Avalon interface specifications. Altera, May 2011.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.

[8] Jungo WinDriver. http://www.jungo.com/st/windriver_usb_pci_driver_development_software.html.

[9] G. Marcus, Wenxue Gao, A. Kugel, and R. Manner. The MPRACE framework: An open source stack for communication with custom FPGA-based accelerators. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 155 –160, april 2011.

[10] MSMST. http://us.kontron.com/products/boards+and+mezzanines/pc104+sbc+and+peripherals/microspace+pc104+cpus/msmst.html.

Communication Interfaces for a New Tester of ATLAS Tile Calorimeter Front-end Electronics

José Domingos Alves [1, 2] José Silva [1] Guiomar Evans [2, 3] José Soares Augusto [2, 4]
jalves@lip.pt *jmanuel@lip.pt* *gevans@fc.ul.pt* *jaaugusto@fc.ul.pt*

1 - Laboratório de Instrumentação e 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 (INESC-ID, Lisboa)

Abstract

The Tile Calorimeter (TileCal) is a subdetector of the ATLAS detector of CERN. It provides accurate measurement of the energy of particle jets resulting from proton-proton collisions that occur in LHC (Large Hadron Collider) experiments, measurement of Missing Transverse Energy and identification of low-pt muons. The TileCal has a fast electronics system (front-end electronics) responsible for acquisition, processing, conditioning and digitalization of signals from the calorimeter, and sends data for the back-end electronics. A tester, called Mobile Drawer Integrity Checking System (MobiDICK) is used to check the correct behaviour of the front-end electronics of the Tile Calorimeter. The new version of this tester communicates with a user's computer via an Ethernet interface implemented in a FPGA (Field Programmable Gate Array). Validation and performance tests of the Ethernet interfaces available to implement in FPGAs were required to determine the most suitable for the tester. The data coming from the front-end electronics via optical connections (using the S-Link protocol) feeds an interface called G-Link, which includes a CRC (Cyclic Redundancy Check) module to check the integrity of received data.

In this paper those Ethernet experiments and the CRC development are presented, as well as their FPGA implementation.

1. Introduction

The LHC 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 occur at key points, exactly at 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 in the LHC.

The MobiDICK is a mobile tester used to check the front-end electronics of the hadronic calorimeter TileCal of the ATLAS detector. The electronic systems that MobiDICK will test are diverse, and so versatility and ease of maintenance must be ensured. These specifications led to the use of reconfigurable components as a base of this tester. The new tester uses a ML507 board from Xilinx equipped with a Virtex-5 FPGA. It communicates with the user's computer (operator) via an Ethernet interface, using the TCP/IP protocol. At the development time of the MobiDICK tester, no absolute requirements were imposed regarding the area of implementation or link speed for the Ethernet module. It was needed an Ethernet module completely functional and as fast as possible. Two Ethernet modules were available in the Xilinx's tools, and this fact led us to evaluate both and check their functionality. One of the functions of MobiDICK is to test the readout from the front-end electronics, by checking incoming data. An algorithm based on Cyclic Redundancy Check is already implemented in the TileCal back-end to check the integrity of data sent by the front-end, but in test mode the front-end electronics will communicate with the MobiDICK, and so the same module should be implemented also in MobiDICK.

This paper is structured in 7 sections. In section 2 is presented the TileCal front-end electronics. Section 3 is dedicated to the description of the tester MobiDICK 4. The description of the Ethernet interfaces is in section 4. Section 5 is dedicated to the CRC module. In section 6 the results of the validation of the Ethernet interfaces and the CRC module are presented. Finally, in section 7 there are the final considerations and conclusions about this work.

2. The TileCal Front-end Electronics

The TileCal is a sampling calorimeter with steel as absorber medium and scintillating tiles as active material. In the LHC experiments, ionizing particles that cross the TileCal produce photons in the scintillators, and the intensity of these photons is proportional to the energy deposited by the particles. These photons are absorbed and carried by Wavelength Shifting Fibers (WLSFs) and then feed a set of photomultipliers (PMTs). The photomultipliers convert light signals into electrical impulses, which serve as input signals of the *3in1* cards in the front-end electronics. These cards perform the conditioning and processing of those analog signals, and then they send them to the digitizer system where they are digitized and organized in data packets before being fed to the back-end electronics [1]. The TileCal analog signals are converted to digital signals with a sampling frequency of 40 MHz.

The front-end electronics of the TileCal is organized into compact structures called *Drawers* (Figure 1). A combination of 2 Drawers forms a *Super-drawer*. Each Super-drawer is associated with a module of the TileCal, and is able to collect information from 32 or 45 channels of the detector.

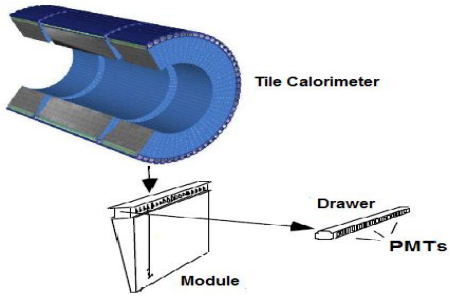


Figure 1: Configuration of the front-end electronics system of the Tile Calorimeter.

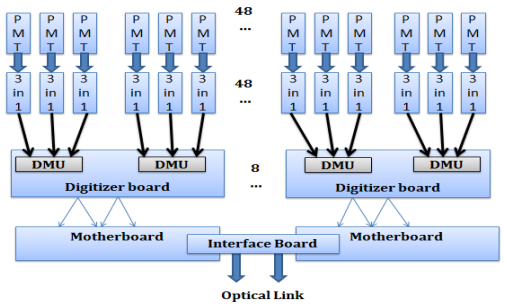


Figure 2. Front-end electronics.

The key component in a Super-drawer is the *motherboard*. It provides low voltage power and digital control signals to four digitizer boards, one Interface Board and to the circuits needed for trigger summation and distribution. One Super-drawer has two motherboards, which include all the electronics used for reading a full TileCal module. An Interface Board collects the sampled data from all the digitizers, serializes and transmits them to the back-end electronics using optical links [1]. This process is illustrated in Figure 2.

The integrity of the data received by the back-end is checked with a CRC algorithm. The DMU (Data Management Unit) is responsible for organizing the digitized samples in packets of data. There are 8 Digitizer boards in a Super-drawer, and each one has 2 DMU devices, so there are 16 DMU devices per Super-drawer. Each DMU builds its packet of data, computes two CRC values and appends them to the packet before it is sent to the Interface Board. The Interface Board computes again the CRCs over each packet of data of each DMU, checking their integrity. If there is no error, the Interface Board builds a packet containing all the DMU packets of data and computes a global CRC over all the data before sending it to the back-end. The back-end electronics computes again the CRC of each DMU data packet and of the global CRC, and compares them with the CRC values received. If the CRC computed by the back-end is equal to that sent by the front-end, there are no transmission errors.

3. The MobiDICK 4 Tester

The new version (4) of the MobiDICK is based on reconfigurable systems. Figure 3 shows the architecture of this tester, which is composed by the following components [1]:

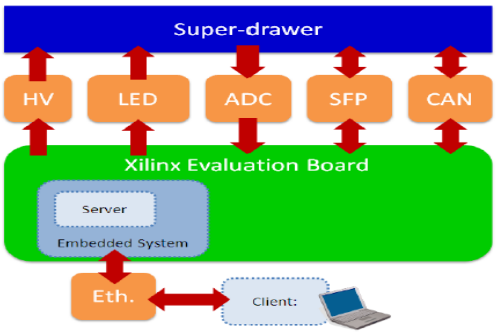


Figure 3. MobiDICK 4 [1].

Development Board: it is a Xilinx ML507 board equipped with a Virtex-5 FPGA. This platform's resources are a PowerPC 440 RISC microprocessor, some 4.25 Gbps GTX transceivers and a 10/100/1000 Ethernet MAC. The ML507 also includes 256 MB of DDR2 RAM and configuration storage, such as platform flash devices or a Compact Flash card-based system configuration controller (System ACE Controller).

High Voltage Board (HV): provides high voltage to the PMTs of the Super-drawers during tests.

LED Board: provides the 20 V pulses necessary for calibrating the *Super-drawer* readout channels.

 ISBN: 978-972-8822-27-9

SFP: optical connector that provides optical fibers' connection to the Interface Board of the *Super-drawer*.

CAN: two commercial adapters and a custom cable convert the RS232 output ports of the development board to the CAN bus interface of the *Super-drawers*.

Power Distribution Board: a commercial AC/DC converter and a custom PCB populated with many DC/DC converters provide several different voltages to the other boards of the MobiDICK.

In the Virtex-5 FPGA, available on the development board, is implemented an embedded system, which runs server software responsible for performing electronic tests to the *Super-drawer*. The client (running on a laptop) sends commands to the server software, requesting MobiDICK to perform those tests. The server handles the requests, performs the tests and sends back the results. The Server and the Client communicate via an Ethernet interface implemented in that embedded system, using the TCP/IP protocol.

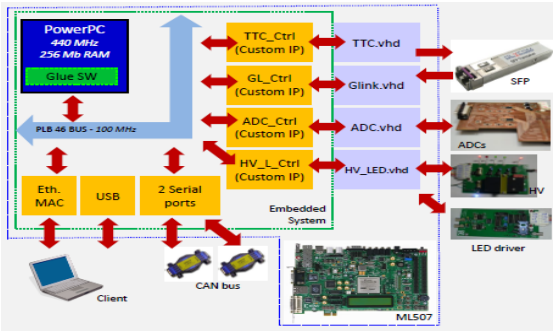


Figure 4. Embedded system in the MobiDICK 4 [1].

Figure 4 shows the embedded system implemented in the MobiDICK 4. The main core is a PowerPC microprocessor running at 440 MHz. An embedded Linux with kernel 2.6.39 has been chosen as Operating System for the PowerPC, mainly because there is an important community supporting the microprocessor and there are available drivers for Xilinx IP cores. An automatic boot of the whole system (bitstream + kernel + root file system) is performed from the Compact Flash using the System ACE Controller. The ELDK 4.2 (Embedded Linux Development Kit) cross compiler tools are used for building the Linux OS image and for developing the applications [1]. The processor is connected to its peripherals using a 100 MHz Processor Local Bus 4.6 (PLB). These peripherals are IP cores: either commercial and provided by Xilinx, or custom IP cores developed to fulfil specific needs. The custom boards of the test bench are interfaced to the embedded system by VHDL firmware modules in the FPGA side, and by application libraries that run on the server on the embedded side [1].

This embedded system receives test requests (sent by the client), runs the server software to handle these requests, executes electronic tests and sends back the results via Ethernet. Two Ethernet interfaces were tested and validated in the scope of this work. The *Glink* block in this embedded system is the interface used to receive packets of data sent by the front-end electronics in the readout test. The SFP module is an optical connector which connects through optical fibers to the Interface Board of a *Super-drawer*. This communication is supported by the S-Link protocol, which uses G-Link devices as physical layer. In the back-end is implemented an ASIC G-Link receiver, the HDMP-1024, which is used to receive and de-serialize the data sent by a G-Link transmitter, the HDMP-1032 in the front-end electronics. The *Glink* block of the MobiDICK is an emulator of the HDMP-1024 receiver described in VHDL. This work also describes the implementation of a CRC module that checks the integrity of the data sent by the front-end electronics.

4. Ethernet Interfaces

The MobiDICK communicates with the user's computer using Ethernet and the TCP/IP protocol. To perform this communication, a Media Access Controller (MAC) core has to be implemented in the embedded system of the MobiDICK. Two MAC interfaces were available as Xilinx tools' IP Cores: the *Tri-Mode Ethernet Media Access Controller (TMAC)* interface and the *Ethernet Lite Media Access Controller (ELM)* interface.

The TMAC is an IP core which supports link speeds of 10, 100 and 1000 Mbps, and half-duplex and full-duplex modes. The TMAC conforms to IEEE 802.3-2008 specifications [2]. The ELM supports link speeds of 10 and 100 Mbps, provides minimal functionality and conforms to the IEEE 802.3 Media Independent Interface [3]. To test these two Ethernet interfaces, an embedded system developed around the softcore microprocessor MicroBlaze was deployed in a Xilinx Virtex-6 FPGA on a ML605 board, as shown in Figure 5. The Ethernet interface selected after the tests was finally implemented in the MobiDICK 4 Motherboard (note that the tests were performed in a Virtex-6, while the deployment in the MobiDICK 4 is done in a Virtex-5; however, this discrepancy didn't cause any migration issues).

The embedded system was developed using Xilinx's Embedded Development Kit (EDK). The main core is the MicroBlaze, a Reduced Instruction Set Computer (RISC) optimized for Xilinx's FPGAs [4]. The ML605 board has a Marvel M88E1111 EPHY device as the physical layer for Ethernet communications and supports communications at

10/100/1000 Mbps. The connection with the outside world is implemented using a HFJ11-1G01E RJ-45 connector. The connection between the MAC interface and the physical layer was implemented using the MII/GMII interface available in the ML605 board.

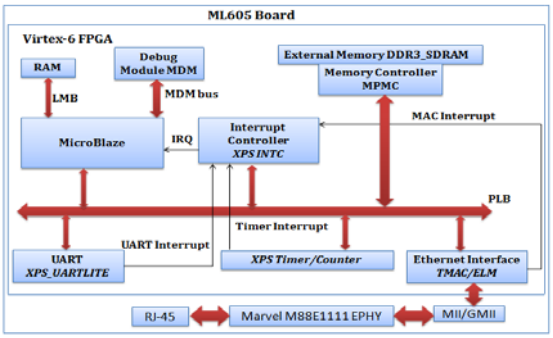


Figure 5. System implemented in a Virtex-6 FPGA to test the Ethernet interfaces.

Ethernet-TCP/IP communication tests between a computer and the ML605/Virtex-6 with the embedded system were performed. The TCP/IP protocol is usually implemented as a service in the OS. In embedded systems it is usually used a lightweight implementation, called lightweight Internet Protocol (lwIP), which does not require an operating system; however, it can be implemented on top of an operating system.

Module	IP Core	Version
Microprocessor	microblaze	8.00.b
Bus	plb_46	1.05.a
Local Memory Bus	lmb_v10	1.00.a
Local Memory	bram_block	1.00.a
Local Memory Controller	lmb_bram_if_cntlr	2.10.b
UART	xps_uartlite	1.01.a
Ethernet Interface	xps_ethernetlite	4.00.a
	xps_ll_tmac	2.02.a
Timer	xps_timer	1.02.a
External Memory Controller	mpmc	6.02.a
Debug Module	mdm	2.00.a
Interrupt Controller	xps_intc	2.01.a
Clock	clock_generator	4.01.a
Reset	proc_sys_reset	3.00.a

Table 1: IP Cores used for the implementation of the embedded system in the ML605.

Table 1 shows all the IP Cores used to implement the embedded system used to test and benchmark the two Ethernet interfaces.

5. The CRC Module

During the test of the readout system, the MobiDICK connects to the Interface board of the front-end electronics of the TileCal via optical fibers. In these tests, packets of data are sent by the front-end electronics to the G-Link interface of MobiDICK 4. The integrity of the data coming from the front-end is checked. A CRC algorithm for error detection, described in VHDL, is already implemented in the back-end, in Altera FPGAs, for checking data integrity. So, this module had to be implemented in the MobiDICK, but in a different FPGA, the Xilinx Virtex-5.

CRC is widely used in communication systems. This algorithm allows the receiver of a message sent through a noisy channel to check if it was corrupted by noise. For this to be possible, the transmitter of the message computes the called CRC or Checksum of the message, and appends it to the message packet before sending. When the receiver receives the message, it computes again the CRC and compares it with the CRC appended to the message. If they are equal, there is no error in the transmission and the message is accepted, otherwise the message is discarded [6], [7].

The front-end electronics of the TileCal performs three types of CRC computations: two for each DMU; and one global CRC over the data of all the DMU devices.

Width	16 bit
Poly	1021 This is the divisor polynome
Init	FFFF This is the initial value of the register
Refin	CRC output is not reflected
Refout	CRC output checksum is reflected
Xorout	No XOR is performed to the CRC output
Check	ascii string "123456789" checksum is 29B1

Table 2. Specifications of the global CRC algorithm.

In table 2 are presented the specifications of the global CRC algorithm as it is implemented in the TileCal, and consequently as it is implemented in MobiDICK 4. The specifications of the algorithm for the two CRCs of DMUs are presented in table 3.

Width	**16 bit**
Poly	**8005 This is the divisor polynome**
Init	**0000 This is the initial value of the register**
Refin	**Yes Reflect the input e.g. bit 0 first**
Refout	**Yes Reflect the checksum output**
Xorout	**0000 Do not xor the checksum with anything**
check	**BB3D This is the checksum for the ascii string "123456789"**

Table 3. Specifications of the algorithm for the CRC of the DMUs.

The CRC module implemented in the MobiDICK 4 (Figure 6) checks data integrity by computing the two CRCs for each DMU and the Global CRC of each global packet sent by the front-end. The CRC module of the back-end only checks the integrity of data, but the MobiDICK 4 also counts the errors using a set of 5 counters.

The structure of the CRC module implemented at the back-end in Altera FPGAs is presented in figure 7. In this work, the goal was to

 ISBN: 978-972-8822-27-9

add the functionalities of error counting and evaluate this module in the motherboard of MobiDICK, the Xilinx ML507 with a Virtex-5.

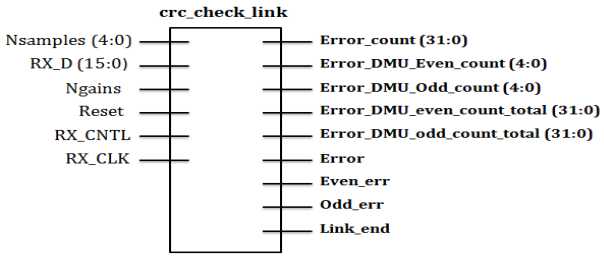


Figure 6. CRC module to check the integrity of data in MobiDICK 4.

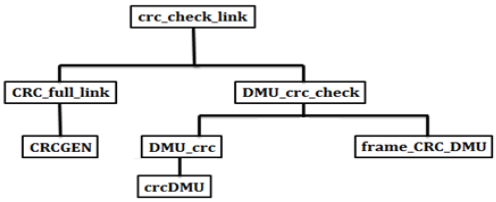


Figure 7. Structure of the CRC Module.

The top level *crc_check_link* is composed of two sub-components:

CRC_full_link: computes and checks the global CRC. It is composed by the *CRCGEN* module, which computes the global CRC, and by a state machine to detect the type of digital word (header, samples, CRC, etc.) sent by the front-end.

DMU_crc_check: computes and checks CRCs of DMUs. It is composed by two internal components: *DMU_crc* and *frame_CRC_DMU*. The *crcDMU* accepts words of 32 bits to compute the CRC of DMUs, so the *DMU_crc* component organizes the 16 bits words sent by the front-end in the 32-bits words required by the *crcDMU* component to compute the CRC of DMUs. The *frame_CRC_DMU* implements a state machine to detect the type of words (header, samples, CRC, etc.) sent by the front-end electronics.

Inputs

The input signal *Nsamples* conveys the number of digital samples sent per DMU device. These samples are 32-bit words sent by each DMU to the Interface Board of the front-end. The signal *Ngains* is a signal related with *Nsamples*; if *Ngains* = '0', the number of samples is 7, and if *Ngains* = '1' the number of samples per DMU can take values up to 16. The 16-bit *RX_D* terminal is the input of the words sent by the front-end and the CRC computation is performed on them. The signal *Reset* resets this module. *RX_CNTL* is a control signal, which is activated when the G-Link interface receives the first word sent by the front-end. When activated, the CRC computation starts. The clock signal *RX_CLK* was used in the validation tests with a frequency of 40 MHz.

Outputs

There are 5 counters implemented in this module. The output signal *Error_count* is a 31-bit counter providing the number of global CRC errors found for all packets of data sent by the front-end.

The front-end electronics performs two CRC computations for each DMU. It performs separately one CRC computation over the even bits of the words, and another CRC computation over the odd bits. So we have two 31 bit counters, one for the errors in the even bits and another for the odd bits: *Error_DMU_even_count_total* and *Error_DMU_odd_count_total*. These two counters provide the number of errors for all packets of data received. There are another two 4-bit counters: *Error_DMU_Even_count*, which provides the number of errors in each packet of data associated with even bits and *Error_DMU_Odd_count*, which gives the number of errors in each packet of data associated with odd bits.

The signal *Error* indicates a global CRC error occurrence; it takes a value '1' if it detects an error and '0' otherwise. The signals *Even_err* and *Odd_err* indicate errors associated with the packets of data of DMUs, for even and odd bits, respectively. They are '1' if there is a CRC DMU error and '0' otherwise. The signal *Link_end* indicates the end of a packet; it takes the value '1' when it detects the last digital word of a packet of data, and '0' otherwise.

6. Results

6.1. Performance Tests of the Ethernet Interfaces

In the performance tests was used the application *Iperf*, which evaluates the performance of the TCP/IP protocol in a network. This application was installed in the PC in order to communicate with the embedded system in the ML605 board. In all these tests it was used the Transmission Control Protocol (TCP) for the Transport Layer.

Transmit test: to determine the rate of transmitted data, a *lwIP* Client application (developed in C) is executed in the embedded microprocessor which connects to the *Iperf* server in the computer. The client sends packets of data continuously to the computer. Then *Iperf* determines the rate at which data are arriving. The maximum transmission rate achieved with TMAC was 101.93 Mbps, while with ELM it was 16.11 Mbps.

Receive test: to determine the rate of received data, a *lwIP* server runs in the embedded microprocessor

and accepts connection requests sent by the *Iperf*, which now works as a client. The *Iperf* client transmits packets of data via Ethernet and determines the rate at which they are sent. The maximum rate of received data with the TMAC was 114.92 Mbps, while with the ELM core it was 12.28 Mbps.

MAC	Link Speed	API	Rate (Mbps)	
			Transmit	Receive
TMAC	1 Gbps	Raw	101.93	114.92
		Socket	35.56	20.91
	100 Mbps	Raw	53.6	52.67
		Socket	36.55	22.40
	10 Mbps	Raw	9.00	9.00
		Socket	8.47	9.43

Table 4: Results of performance tests for TMAC.

MAC	Link Speed	API	Rate (Mbps)	
			Transmit	Receive
ELM	100 Mbps	Raw	10.52	12.28
		Socket	16.11	0.69
	10 Mbps	Raw	8.17	8.30
		Socket	3.15	0.53

Table 5: Results of performance tests for ELM.

Tables 4 and 5 show some results of performance tests of the two Ethernet interfaces. We conclude that the TMAC IP Core provides better communication performance than the ELM IP Core. So we recommended the implementation of the TMAC IP Core in the MobiDICK 4. We validated these two Ethernet interfaces in the ML605/Virtex-6 system, but the MobiDICK 4 uses a ML507/Virtex-5. However there is no incompatibility of versions because the two Ethernet interfaces are both available in Xilinx's EDK as IP Cores, ready to be implemented in any Xilinx's board. The ML507 and the ML605 use the same device as physical layer for Ethernet and we use the same bus (PLB), so we expect that the performance results presented in this work are (relatively) similar to those obtained with the ML507 board of the MobiDICK 4.

6.2. PING Tests of the Ethernet Interfaces

The accessibility test of the Ethernet interfaces was performed with PING *(Packet INternet Groper)* tests. This test is based in the *Internet Control Message Procotol* (ICMP) [5]. In this test the computer sends packets to the board, called *echo requests*, and waits for an *echo reply*. We use the *lwIP echo server* running on the embedded microprocessor in the ML605 to answer to the *echo requests* sent by the computer. PING tests provide statistics of packet losses in the network, so we can have an estimate of the accessibility of the two Ethernet Interfaces [5]. In all the PING tests performed, there were no losses of packets in the network for both Ethernet interfaces.

6.3. Validation Tests of the CRC Module

The CRC module was integrated in the *Glink* block of the MobiDICK 4 and was validated in a real situation of data acquisition from a *Super-drawer* located in a laboratory at CERN.

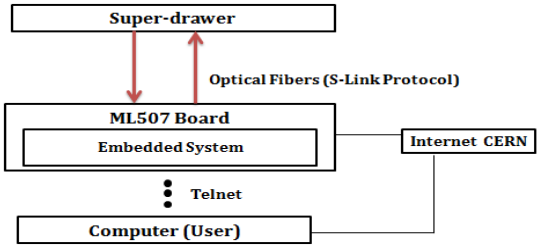


Figure 8. System implemented at a CERN laboratory to validate the CRC module.

Figure 8 illustrates the system that tests the *Glink* block and the CRC module. The ML507 board, loaded with the embedded system of MobiDICK 4, is connected to a *Super-drawer* via optical links. The user's computer and the MobiDICK 4 were connected to the Internet network of CERN, which allows interacting with the MobiDICK 4 via Telnet. A C++ application running in the embedded system sends a command, requesting the *Super-drawer* to send data. The *Glink* block of the MobiDICK 4, with the CRC module, checks the integrity of the data and counts the errors, saving the counter's values in the respective registers. This application orders the microprocessor to read these registers and to send the results to the user's computer Telnet terminal. Figure 9 shows registers being read to show the statistics of errors in the packets sent by the *Super-drawer*.

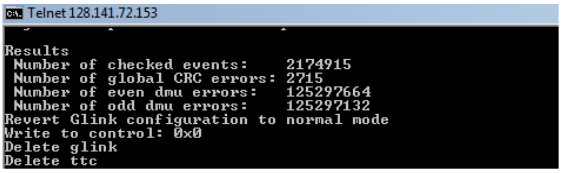


Figure 9. CRC results when the counter registers are read.

We also use the *Chipscope Analyzer* from Xilinx to monitor the signals of interest inside the FPGA, and to verify if the CRC module is able to compute properly the global CRC and the two CRCs for each DMU. In figure 10 is presented an example of the validation of global CRC computations. In this test we acquire a packet of data without error. The global CRC sent by the front-end was 0xBDAB (in the red box), while the computed global CRC is 0xD5BD. The CRC module inverts the order of bits of the computed CRC before comparing it with the CRC sent by the front-end. If the order of the bits in 0xD5BD are reversed (starting with the least significant bit and ending with the most significant bit) we get the value 0xBDAB, the same value sent

by the front-end electronics. This shows that the CRC module computes the correct global CRC, as expected.

We also performed a test to check if the CRC module was able to compute properly the CRCs of the DMUs. The CRC module, as it is implemented in the back-end, did not behave as expected. Figure 10 shows a result of this incorrect behaviour. The CRC values sent by the *Super-drawer* are 0x1A39 (even bits) and 0xE4C3 (odd bits), in the red box. The computed CRC values are: 0x7C74 (even bits) and 0x54B1 (odd bits), in the black box, which are different from those sent by the *Super-drawer*. The Interface Board of the front-end sends 32-bit words divided in the middle, forming digital words of 16 bits. Internally the CRC module has to reorganize these 16-bit words into 32-bit words before starting the CRC computations for the packets of data incoming from the DMUs. In this validation test we found that such organization was not being done properly, because it doesn't replicate the current implementation of CRC in the back-end electronics, in Altera FPGAs. A new version of the MobiDICK's CRC module was developed to overcome this incorrect behaviour and is waiting to be tested in a real scenario, as soon as possible.

In these tests we use a clock with a frequency of 40 MHz in the CRC module, well below the maximum frequency of 376.40 MHz allowable for this module (post-synthesis value given by the Xilinx development tool.) The hardware resources are presented in table 6. The most used resources are IOBs (22 %).

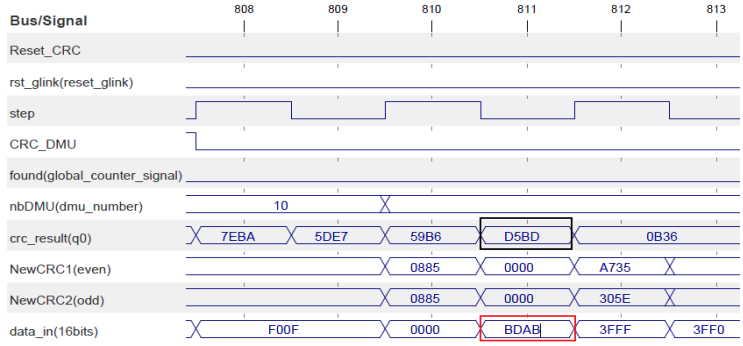


Figure 10. Results obtained in the test and validation of the computation of the global CRC.

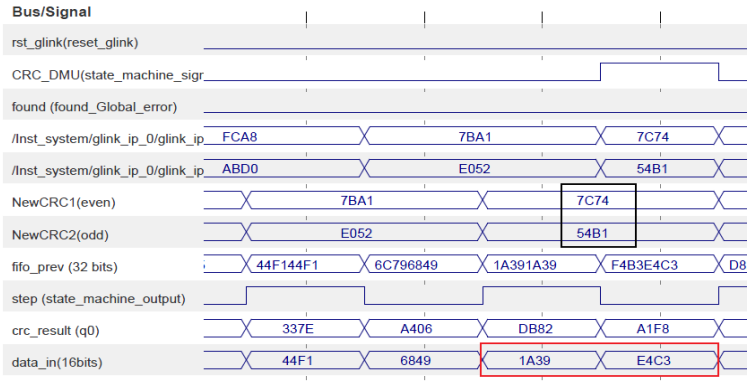


Figure 11. Results obtained in the test and validation of the computation of the CRCs of DMUs**.**

Device Utilization Summary			
Number of BUFGs	2	out of 32	6%
Number of External IOBs	50	out of 220	22%
Number of LOCed IOBs	0	out of 50	0%
Number of OLOGICs	5	out of 400	1%
Number of Slices	168	out of 4800	3%
Number of Slice Registers	385	out of 19200	1%
Number used as Flip Flops	321		
Number used as Latches	64		
Number used as LatchThrus	0		
Number of Slice LUTS	362	out of 19200	1%
Number of Slice LUT-Flip Flop pairs	523	out of 19200	2%

Table 6. Resources used in the implementation of the CRC module in a Virtex-5 FPGA**.**

7. Conclusion

This paper describes two contributions to the development of communication modules for a new tester of the ATLAS Tile calorimeter front-end electronics. These contributions consist in the implementation and test of Ethernet interfaces deployed in a FPGA, and in a VHDL module able to check data integrity. This new tester, called MobiDICK 4, is implemented around a reconfigurable device, a Virtex–5 in a Xilinx ML507 board. The MobiDICK 4 includes a server and a client. The client runs in a laptop and sends commands to the server, requesting electronics tests, while the server runs in an embedded system implemented in the motherboard and is responsible for handling the requests, performing the electronics tests and sending back the results to the client. Client and server communicate via Ethernet.

From the results of the validation of Ethernet interfaces, we conclude that the TMAC IP core is more suitable to be implemented in the MobiDICK 4 than the ELM IP core. We found, as expected, that TMAC provides better performance than the ELM. However, the ELM uses less FPGA resources than the TMAC, but our recommendation elects speed link performance as the main key for selecting the interface.

The CRC module implemented in the back-end electronics of the TileCal was migrated to the MobiDICK 4 and validated in a real situation of data acquisition from a *Super-drawer* in a CERN laboratory. The results have shown that it was working properly, but partially. A new version has been developed to overcome the incorrect behaviour found in the CRC module and to mimic its present implementation in the back-end electronics, in Altera FPGAs; at present, the new module waits to be tested as soon as possible.

Acknowledgment

This work was partially supported by national funds through FCT (Fundação para a Ciência e Tecnologia), under project *PEst-OE/EEI/LA0021/2011*. While developing this work, José Domingos Alves was the recipient of a grant from the Project *CERN/FP/123595/2011*, also with funds from FCT.

References

[1] P. Moreno, J. Alves, D. Calvet, F. Carrió, M. Crouau, K. H. Yeun, I. Minashvili, S. Nemecek, G. Qin, V. Schettino, C. Solans, G. Usai and A. Valero, “A new portable test bench for the ATLAS Tile calorimeter front-end electronics”, TWEPP 2012 Topical Workshop on Electronics for Particle Physics (Oxford), 2012.

[2] Xilinx, “LogiCORE IP Tri-Mode Ethernet MAC v4.5, User Guide, UG138”, 2011.

[3] Xilinx, “LogiCORE IP XPS Ethernet Lite Media Access Controller, Product Specification”, 2010.

[4] Xilinx, “MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 13.4, UG081”, 2012.

[5] Martin P. Clark, “Data Networks, IP and the Internet”, John Wiley & Sons Ltd, 2003.

[6] Ross N. Williams, “A Painless Guide to CRC Error Detection Algorithms”, Rocksoft Pty Ltd, Australia, 1993.

[7] Rajesh G. Nair, Gery Ryan and Farivar Farzaneh, “Symbol Based Algorithm for Hardware Implementation of Cyclic Redundancy Check”, Nortel Networks Ltd, United States Patent Nº US 6295626B1, Filled in 1998, Date of Patent: 2001.

Novas abordagens ao projeto de sistemas reconfiguráveis

Automatic Generation of Cellular Automata on FPGA

André Costa Lima
Faculdade de Engenharia
Universidade do Porto
ee06223@fe.up.pt

João Canas Ferreira
INESC TEC and Faculdade de Engenharia
Universidade do Porto
jcf@fe.up.pt

Abstract

Cellular automata (CA) have been used to study a great range of fields, through the means of simulation, owing to its computational power and inherent characteristics. Furthermore, CAs can perform task-specific processing. Spacial parallelism, locality and discrete nature are the main features that enable mapping of CA onto the regular architecture of an FPGA; such a hardware solution significantly accelerates the simulation process when compared to software. In this paper, we report on the implementation of a system to automatically generate custom CA architectures for FPGA devices based on a reference design. The FPGA interfaces with a host computer, which initializes the system, downloads the initial CA state information, and controls the CA's operation. The user interface is are provided by a user-friendly graphical desktop application written in Java.

1. Introduction

Cellular automata have already been studied for over half a century. The concept was introduced by John von Neumann in the late forties when he was trying to model autonomous self-replicating systems [1]. Since then, CA-based architectures have been extensively explored by the scientific community for studying the characteristics, parameters and behaviour through simulation of dynamic complex systems, natural and artificial phenomena, and for computational processing [2]. Specific-application areas are, e.g., urban planning, traffic simulation, pseudo-RNG, complex networks, biology, heart modeling, image processing, sound waves propagation and fluid dynamics. Furthermore, CA is also a possible candidate for future alternative computer architectures [3].

Nowadays, the need for creating simulation environments of a high degree of complexity to describe rigorously the complete behaviour of a system, and for developing algorithms to process data and perform thousands of calculations per second, typically incurs a high computational cost and leads to long simulations. Taking advantage of multi-core architectures, with parallel programming, to assign multiple task execution to different threads, enables significant software optimizations and accelerated execution. However, parallel programming in software is a complex task, as several aspects such as concurrency must be taken into account, and may be time-consuming. CA offer simplicity when it comes to define each cell's functionality and owing to its inherent massive spacial parallelism, locality and discrete nature, such systems are naturally mapped onto the regular architecture of an FPGA, organized in reconfigurable logic blocks; it is possible to perform complex operations with efficiency, robustness and, most importantly, decrease drastically the simulation time with high-speed parallel processing. Thus, as a reconfigurable hardware device, FPGA platforms are a very good candidate for CA architecture implementations on hardware: each cell functionality can be implemented in look-up tables, its state stored in flip-flops or block-RAM, and it is possible to update the state of a set of cells in parallel every clock cycle.

In this paper, we report on the implementation of a system to automatically generate CA architectures on a target FPGA technology, a Spartan6, the characteristics and rules of which are specified by the user through a software application, that also allows controlling the operation, initializing and reading the state of the CA. The implementation results show that it is possible to achieve a speed-up of 168, when comparing to software simulations, for a lattice size of 56×56 cells for the Game of Life [4] [5]. Furthermore, we achieved lattice dimensions of 40×40 and 72×72 cells for the implementation of a simplified version of the Greenberg-Hastings [6] and lattice gases automata [7], respectively.

The remainder of the paper is organized as follows. Section 2 introduces the key characteristics and properties of CA. Section 3 presents some of the existing work related to implementations of CA in FPGA platforms. Section 4 describes the overall system architecture, its features and specification. In section 5 the implementation results are summarized and discussed. Finally, section 6 concludes this paper and future developments are proposed.

2. Cellular automata

Cellular automata are mathematical models of dynamic systems that evolve autonomously [8]. They consist in a set of large number of cells with identical functionality with local and uniform connectivity, and organized in a regular lattice with finite dimensions [9], for e.g., an array a matrix

or even a cube for the multidimensional scenario; boundary conditions, that can be fixed null or periodic (the lattice is wrapped around), define the neighbourhood of the cells located at the limits of the lattice. For square-shaped cells, the neighbourhood is typically considered to be of Moore [10] or von Neumann [11] types which are applicable to 2D and 3D lattices; considering a single cell, the local neighbourhood is defined by the adjacent cells surrounding it, i.e. the first order or unit radius neighbourhood. Thus, for 2D lattices, the Moore-type neighbourhood includes all the 8 possible in a square shape and the von Neumann-type neighbourhood includes 4 cells in a cross shape. Each cell has a state that can be discrete or real-valued and the update occurs in a parallel fashion in discrete time steps, i.e. every cell in the lattice update its state simultaneously; at the time instant t_i, the cell states in the local neighbourhood of a cell c are evaluated, and the next state for c, at t_{i+1}, is determined according to its deterministic state transition rule that is a function combining arithmetic or logical operations. However, this rule can be probabilistic if it depends on a random nature variable; in this case, the CA is considered to be heterogeneous as the cells has no longer a homogeneous functionality.

3. Related work

Vlassopoulos *et al.* [6] presented a FPGA design to implement a stochastic Greenberg-Hastings CA (GHCA), which is a model that mimics the propagation of reaction-diffusion waves in active media. The architecture is organized in group, block and cell partitions in a hierarchical way across the FPGA. Each group contains a set of blocks and is processed in parallel as a top-level module; within a group, blocks are processed sequentially. Each block contains a set of cells that are distributed around BRAM-type memory resources, to maximize its usage, and control logic. Each cell contains a random event generator circuit (LFSR), which is a requirement of the GHCA model, state control logic and registers. Additional BRAMs are used to handle boundary cells data, which are shared with a subset of groups' blocks. The FPGA environment interfaces with an external host machine to initialize the CA, collect results and control its operation. A Xilinx Virtex 4 (XC4VLX100-10FF1513) FPGA device was used operating at a frequency of 100 MHz; with a lattice of maximal dimensions of 512×512 cells, 26 % of the total slides available and 136 out of 240 were required for implementation. As for the benchmark platform, an Intel Core 2 Quad CPU Q9550 machine running at 2.83 GHz was used; with a lattice size of 256 256 cells and a simulation period of 10^4 time steps, it was obtained a speed-up of approximately 1650. However, as the effective parallelism of the solution is 128 cells per cycle it means that it takes 512 clock cycles to perform a single iteration of the CA with 4 bits per cell. Considering that a several number of memory write and read operations are required per iteration, especially with regards to reading and storing boundary data, this has a negative impact on the total simulation time.

Shaw *et al.* [7] presented a FPGA design to implement a lattice gas CA (LGCA) to model and study sound waves propagation. LGCA are models that allow to emulate fluid or gas dynamics, i.e. particle collisions, and are conveniently implemented with digital systems; thus, the real-world physics are described in discrete interactions. The authors describe the behaviour of each cell with a simple set of collision rules. For simplicity, they consider unit velocity and equal mass for every particle and four distinct directions, i.e. a typical von Neumann neighbourhood. The current state of each cell indicates the direction in which existing particles are traveling according to the collision logic implemented by them; thus, 4 bits are required to represent the outgoing momentum for each direction with the 9 possible combinations: a pair (v_x, v_y) of integer values from $(-1,-1)$ to $(1,1)$. The collision logic function that evaluates a particle stream and outputs those that carry on in the next cycle is defined as follows. A particle travels to the east if a) a particle arrives from the west except if there is an east-west head-on collision and b) if there is a north-south head-on collision. An head-on collision involves only 2 particles traveling in opposing directions. This definition is analogous to the remaining directions.

The design was implemented in a SPACE (Scalable Parallel Architecture for Concurrency Experiments) machine [12] which is an array of Algotronix CAL FPGAs whose technology is outdated by now. Results showed that it was possible to reach 3×10^7 cell updates per second, 2.2 times lower than two CAM-8 modules. CAM (CA-machine) [13] [14] is a dedicated machine whose architecture is optimized for high-scale CA simulation; it was developed by Tommaso Toffoli and Normal Margolus and it received a lot of attention.

4. System architecture

The overall architecture of the system described in this work consists of a digital system design for a FPGA device and software desktop application, which is a graphical tool for the system user that provides an interface to control the hardware operation, a representation of the CA to initialize and read its state map, and the generation of bitstream files each one equivalent to a single custom CA specification. Thus, our approach allows the system user to parameterize the hardware architecture according to its needs. Prebuilt templates, that describe in Verilog the circuits of separate modules, are used as a reference to integrate complete CA specifications in the design. The CA characteristics are template configuration parameters, i.e. lattice dimensions, neighbourhood type and number of bits per cell, and the state evolution rule is the body of the cell modules that determine its functionality.

Currently, the architecture of the hardware system is flexible enough to support the following features: multidimensional lattices (1D and 2D), Moore and von Neumann neighbourhoods, periodic or null lattice boundary conditions and unconstrained number of state resolution bits per cell; the lattice width is, however, constrained to multiples of 8. The state evolution rule can be specified with logical and arithmetic operations at the bit level. Furthermore, it is

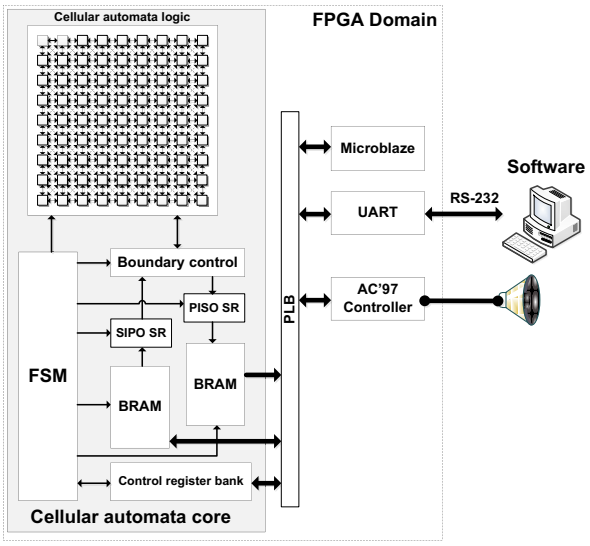


Figure 1. System architecture

also possible to provide different rules to different rows of cells in the lattice, which is an advantage to simulate multiple 1D CAs in parallel. Two operation modes are supported: run-through for n iterations or step-by-step.

4.1. Hardware specification

4.1.1. Global architecture

The overall hardware architecture is shown in figure 1. The FPGA domain it is further divided in two distinct subsystems: the CA core and the soft-core processor MicroBlaze. The MicroBlaze is used to transfer via RS-232 the cells state data, by accessing data stored in BRAM in the CA core, from and to the host software application and implements a simple protocol to do so. Moreover, the audio codec AC'97 is used for demonstration purposes to generate melodies according to the state map of the CA. The melody generator function receives the state data of a row of cells and determines the bit density, i.e. the number of bits equal to 1 over the total number of cell bits in the row. The function output is the note frequency to be played generating a square wave. This frequency is as high as the bit density; on a typical piano setup (88 keys), 100 % bit density is equivalent to approximately 4 kHz.

Our development platform, the Atlys board from Digilent, has, in addition to the Spartan-6 FPGA, numerous peripherals, including the audio codec chip that is used as a sound output. The hardware controllers for the audio codec and the UART are provided by Digilent and Xilinx, respectively; both already implement an interface to the PLB (bus). For simplicity, the corresponding available software drivers were used with the MicroBlaze CPU.

4.1.2. CA core architecture

The CA core is our custom hardware and contains the necessary modules to control the CA operation, initialize, read and hold its current state. The CA logic module contains all the generated cells for a custom specification and it is not run-time configurable, i.e., it not possible to change the CA characteristics, structure or rules. Thus, every single specification is final until a newer one is provided and its corresponding bitstream generated.

Every single row of cells is generated independently as each one is constructed from a different Verilog module. Even though the top-level description and reference logic of every single cell is identical, for each row of cells the functionality is not necessarily the same, as mentioned in the architecture features. However, all the cells in the lattice are interconnected with each other and the number of interconnects of each cell depends on the neighbourhood type: 4 for von Neumann and 8 for Moore types. The interconnects are buses whose width is defined by the number of bits per cell. Therefore, a single cell can have 4 or 8 inputs and always one output to pass its state to its neighbours. In case of 1D CA, each cell always has 4 inputs. The cell architecture is described in detail in section 4.1.3.

To initialize and read the state of the CA, as well as to pass cell data from one boundary to the opposite boundary, input (IB) and output (OB) external buses, connected to the cells on the edge of the lattice, exist. Each bus has a different width, regardless of the number of bits per cell, and it may only depend on the neighbourhood type and lattice structure. Each bus is indexed b by b bits, from IB and OB, forming a neighbour boundary interconnect that correspond to the state data of different cells along it. For example, in a 2D scenario with a von Neumann neighbourhood each boundary cell has one boundary interconnect, except the corner cells that require two. On the other hand, for a Moore neighbourhood each boundary cell has three boundary interconnects but the corner cells require five. Both IB and OB are connected to each other, in case of a periodic boundary, and are redirected by the boundary control module whenever needed. For a null boundary, IB is connected to the ground. This is explained with more detail in the operation modes below.

Figure 2 shows a more detailed view of the internals of the CA core with focus on the modules interconnects, bus widths and operation modes.

The cell state data is written and read to and from two dual-port BRAMs, one used as an input memory and another as an output memory. A 32-bit port is used to interface with the PLB (bus) and a 8-bit port is used internally by the core. Two shift-register (SR) modules are used in order to parallelize (SIPO) chunks of $c \times b$ bits and serialize (PISO) chunks of 8 bits of cell state data to be read from and written to memory, respectively, where c is the number of lattice columns and b the number of bits per cell. The SIPO SR has an 8-bit input to read data from the input BRAM and a output of $c \times b$ bits, which is the total width in state bits of a row of lattice cells. Each stage of the SR has a 8-bit depth and such amount of data should be shifted whenever a read from memory and a write in the SR are performed. The PISO SR has an input of $c \times b$ bits and a 8-bit output to write data to the output BRAM; the functionality and structure are identical to prior SR. The length of both SRs depends on the lattice dimensions and it

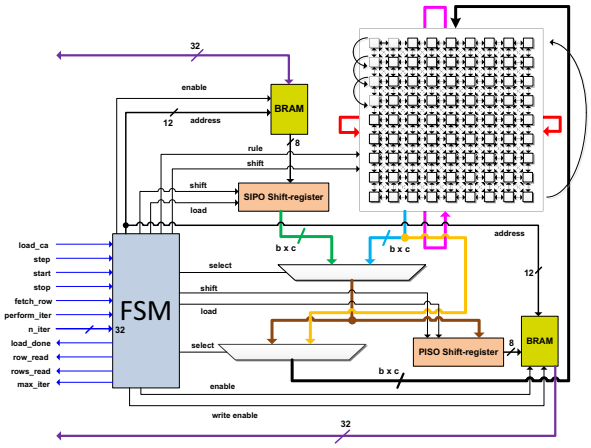


Figure 2. Detailed view of the cellular automata core

is given by $(c \times b)/8$. Thus, the number of columns of the lattice is constrained to multiples of 8. The reason for this is to make it easier to manipulate data transfers and organization for this memory configuration and processing.

The boundary control module consists of several multiplexers to redirect boundary data buses, the IB and the OB, depending on the CA operation mode, i.e., reading, initializing or iterating. In the first two modes, the lattice rows are shifted in a circular fashion, from top to bottom, as state data is read from or written to the cell's memory and, at the same time, to the I/O memories, accordingly, via SRs. After the reading operation, the state map of the CA remains as it was before. In the latter mode, the multiplexers redirect the OB to the IB wrapping around the lattice and, every clock cycle, a new state for every cell in the lattice is computed. In the iteration mode, the next state is always computed until the simulation time reaches n time steps; if the step-by-step mode is active, the simulation halts after every iteration until further order is given to continue, however it is limited as well to n time steps. In the step-by-step mode, the state map is read every iteration overwriting the previous data that was stored in memory; in the run-through mode, the state map is only read at the end of the simulation. A finite state machine (FSM) is responsible for controlling all the core modules, register operation responses and listen to user command signals from a control register bank (CRB). The bank is used to synchronize operations between the Microblaze and the FSM as shown in Figure 1.

4.1.3. Cell architecture

The module that describes the cell architecture is unique but it is replicated $c \times r$ times, where c and r are the number of columns and rows of the lattice, respectively. It consists of a sequential circuit to hold the cell's state and a combinatorial circuit that implements the state transition rule. Again, an input bus and an output bus exist to receive the neighbour cells state data and pass the next state to them, respectively. The input bus has a width of $4 \times b$ or $8 \times b$ bits for a Moore and von Neumann neighbourhood, respectively. Each b bits are the state bits of the cells adjacent to a central one. For controlling, two additional 1-bit signals are provided and are common to every single cell. One of them enables the state transition rule (iteration mode) and the other controls the direct loading of the state data from the cell located north (reading and initializing modes). Both signals are fed to two multiplexers; when neither of them is active, the cell is idle and its state remains as it is.

4.2. Software specification

The software application was written in Java, provides a simple and friendly GUI and acts as the front-end to the user of the hardware system with a high-level of abstraction of technical details. This tool is capable of running a typical FPGA Design Flow to produce a bitstream file equivalent to the custom CA specification. This process is automatic and transparent to user as it occurs in background by executing existing scripts. Each bitstream file is saved in a separate directory; an existing bitstream can be programmed any time. However, the current version of the tool, besides providing the features mentioned in section 4, can only generate bitstream files for a Spartan6 as the reference design was built for that FPGA device in specific. However, the number of generated bitstreams is unlimited.

The GUI of the software tool is shown in figure 3. The characteristics of the CA can be specified by filling in the text fields with the desired values and check box marks. On the other hand, the state transition rule must be described textually with Verilog HDL and two text areas are used for this purpose. The main one is used to explicitly indicate the state register update rule of a cell, and the other one is provided for additional optional supporting logic. To avoid knowing the input and output buses names from the Verilog module, the tool provides a simple set of tags that allow to reference quickly b bits of any neighbour interconnect. Upon saving the specification, each tag string that is found is replaced by its interconnect reference in the module description, and the specification is integrated in a copy of the pre-built cell template module file located in the working directory. This copy is then included in the directory

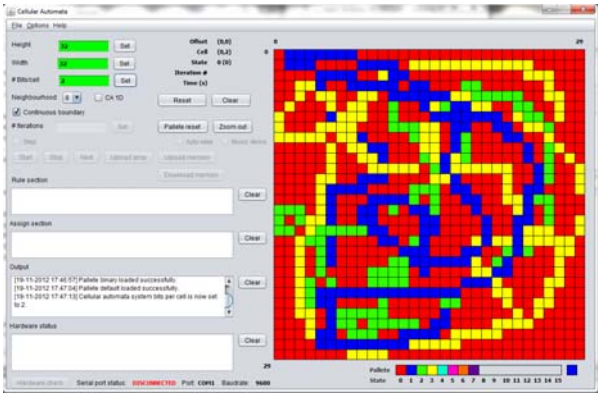


Figure 3. Graphical user interface of the software tool

where all the system module files are kept; each one of these files describe the functionality of a row of lattice cells. As for the CA characteristics, these are saved directly in the header files that are included in the core modules.

The serial communication parameters that are configurable are the COM port and the baudrate. However, the baudrate is fixed in the design at 460800 bps which is limited by the UART controller. A set of buttons in the GUI allow to interact with the FPGA once a connection is established. These buttons call functions that implement the operations modes and the protocol defined, that the Microblaze implements as well, to send commands, transmit and receive cell's state data.

To initialize and represent the state map of the CA, a graphical editor that maps cell states to colours, according to a custom palette, is provided. This utility is capable of representing a lattice with a dimension up to 200×200 cells, which is the maximum supported by the zooming function. Given that the representation area was chosen to be limited, a dragging capability was added. In addition to that, the editor supports a fast paint mode that allows switch a cell state quickly by just dragging the mouse over the selected area instead of clicking. Nevertheless, to change a cell state with the fast paint mode disabled requires a click. For easier navigation when dragging the lattice, the coordinates of the cell pointed by the cursor are indicated aside. Finally, a custom palette builder enabled the representation of any given state with a RBG-specified colour; colour palettes can be saved for future use.

5. Results

5.1. Implementation and benchmarking

The architecture was implemented on a Xilinx Spartan-6 (XC6SLX45) FPGA running at 66 MHz. For initial benchmarking, we used the Game of Life algorithm which is a bidimensional CA model to simulate life. Each cell is either *dead* or *alive*, therefore a single bit is required to represent the state; a cell that is alive remains so if either two or three neighbours are alive as well; a dead cell with exactly three neighbours alive is born; otherwise, the cell state remains unchanged. A Moore neighbourhood with a periodic boundary condition is used.

Tables 1 and 2 show the running times on several lattices for two different numbers of iteration. Iteration characteristics are summarized in tables 3 and 4. The hardware simulation time is given by $t_{HW} = n/f_{CLK}$, where n is the number of iterations and f_{CLK} the clock frequency, which is the equal to the *effective* computation time, t_c.

The implementation results refer to the Post-PAR implementation which means that the presented values are not an estimation. The percentage of the resources occupied include not only our custom CA core, but the Microblaze and the UART controller as well; the audio codec controller is not included in these implementation results as it is an extra feature. The computer has an Intel Core 2 Quad Q9400 running at 2.66 GHz and the software application is XLife [15]. Results show that the simulation time on hardware does not depend on the lattice size, but only on clock frequency (which depends on cell complexity).

We have also evaluated implementations of simplified versions of the Greenberg-Hastings CA [6] (without random event generator) and of the lattice gas model CA [7]. The results are presented in tables 5 and6, and tables 7 and8], respectively. Both rules require 4 bits per cell, however the neighbourhood types are different; the GHCA uses a Moore one and the LGCA uses a von Neumann one.

From the results shown in tables 1 and 2, we can see that both exhibit almost constant speed-ups for different array sizes; although not shown here, the same behaviour was observed for a lower n. Thus, we obtain a linear relationship between speed-up and array size. Also note that the hardware simulation time is constant regardless of the dimensions, which results from using just of a single clock cycle per iteration.

5.2. Data transfer time analysis

The results of subsection 5.1 related to hardware simulation time only take into account the CA computation time (when the CA is iterating). In order to provide a more accurate and fair comparison as far as benchmark goes, in this subsection we present a detailed analysis on how data transfer times impact the obtained speed-up figures. We are now going to consider not only the computation time (t_c),

Benchmark — $n = 10^8$ iterations			
Lattice	HW (s)	SW (s)	speed-up
32×32	1.5	59.2	**39**
40×40	1.5	110.8	**74**
56×56	1.5	251.8	**168**

Table 1. Benchmark results for Game of Life performing 10^8 iterations.

Benchmark — $n = 10^9$ iterations			
Lattice	HW (s)	SW (s)	speed-up
32×32	15.0	590.7	**39**
40×40	15.0	1103.5	**74**
56×56	15.0	2490.9	**166**

Table 2. Benchmark results for Game of Life performing 10^9 iterations.

Implementation		
Lattice	FF $(27,288)$	LUT $(54,576)$
32×32	$3,083(5.6\%)$	$10,114(37.1\%)$
40×40	$3,685(6.8\%)$	$18,449(67.6\%)$
56×56	$5,243(9.6\%)$	$24,818(90.9\%)$

Table 3. Implementation results for Game of Life — FPGA resources occupied

Implementation	
Lattice	Frequency (MHz)
32×32	85.7
40×40	72.8
56×56	69.3

Table 4. Implementation results for Game of Life — maximum frequency

Implementation		
Lattice	FF $(27,288)$	LUT $(54,576)$
24×24	$4,503(8.3\%)$	$10,558(38.7\%)$
32×32	$6,366(11.7\%)$	$17,071(62.6\%)$
40×40	$8,717(16.0\%)$	$23,854(87.6\%)$

Table 5. Implementation results for Greenberg-Hastings CA — FPGA resources occupied

Implementation	
Lattice	Frequency (MHz)
24×24	71.4
32×32	68.7
40×40	69.2

Table 6. Implementation results for Greenberg-Hastings CA — maximum frequency

Implementation		
Lattice	FF $(27,288)$	LUT $(54,576)$
56×56	$14,989(27.5\%)$	$15,394(56.4\%)$
64×64	$18,894(34.6\%)$	$19,295(70.8\%)$
72×72	$23,311(42.7\%)$	$23,870(87.5\%)$

Table 7. Implementation results for lattice gases CA — FPGA resources occupied

Implementation	
Lattice	Frequency (MHz)
56×56	76.7
64×64	69.4
72×72	77.2

Table 8. Implementation results for lattice gases CA — maximum frequency

 ISBN: 978-972-8822-27-9

$r \times c$	T_B (ms)
24×24	1.25
32×32	2.22
40×40	3.47
48×48	5.00
56×56	6.81
64×64	8.89
72×72	11.25

Table 9. Time required to transfer cell data for $b = 1$

but also a) the time it takes to write (C_{M_w}) and read (C_{M_r}) data from the CA array of flip-flops to and from the memory, and b) the time it takes to transfer the initialization data from the PC to the FPGA (t_{s_i}) and read the results back to the PC (t_{s_o}). Other existing overheads are also exposed.

It was mentioned in section 4.1 that data is transferred via RS-232 protocol and serial interface, with a baudrate of $B = 460800$ bps limited by the provided controller. The amount of data to transfer, regardless of initializing or reading results back, is always the same given a certain CA specification. Given that each cell contains b bits of data distributed in a lattice of c columns and r rows, the time it takes to transfer the data is given by

$$T_B = t_{s_i} = t_{s_o} = \frac{r \times c \times b}{B} \tag{1}$$

Table 9 summarizes the time length to transfer data for some lattice dimensions. All the data that is written to and read from memories is held, for a certain amount of time, in the shift-registers (SR) whose structure and functionality was described in section 4.1. Shifting cells data is required whenever a new state is to be loaded or the current state read from the array; such operations take time which is worse for larger lattice dimensions and increasing number of bits of cells state data.

The time it takes to load and read the CA array, C_{M_w} and C_{M_r}, respectively, are not equal for two reasons. The first is that when reading data from memory there is always one clock cycle of latency; the second is that the clock cycle to load data to the first stage of the SIPO SR is the same when a new row is loaded to the CA array, which means both operations occur in parallel saving up one clock cycle; in fact, more than one clock cycle is saved up, as it is explained below, which determines that effectively $C_{M_w} \leq C_{M_r}$. To determine the expression of C_{M_w}, given in clock cycles, we consider two different situations: when the SIPO SR is fully loaded for the first time, whenever a initialization operation begins, and then the subsequent loads. Then,

$$C_{M_w} = 1 + \frac{c \times b}{8} + r + r \cdot \left(\frac{c \times b}{8} - 1 \right) \tag{2}$$

The first two terms refer to the first load: a clock cycle of latency only taken into account once and then filling and shifting $(c \times b)/8$ stages of data of SIPO SR. The remaining two terms refer to the subsequent SR loads: the first stage is loaded r times as well as r rows are loaded into the CA array, which saves up r clock cycles; the last term accounts for the loading of the remaining stages also occurring for r times. Simplifying the expression, we obtain

$$C_{M_w} = 1 + \frac{c \times b}{8} \cdot (r + 1) \tag{3}$$

The expression of C_{M_r} is simpler to determine. When reading the CA array, it is shifted for an amount equal to the number of lattice rows, r, as well as to load a rows to the PISO SR. Then, the PISO SR shifts data to be written to the memory, a number of times equal to the number of stages available, again, r times. As a downside, for this case, it is required that all the data present in the SR is shifted out before loading up another row, in order to avoid loss of data, which consumes more time. So,

$$C_{M_r} = r \cdot \left(1 + \frac{c \times b}{8} \right) \tag{4}$$

To obtain the effective time t_{M_w} and t_{M_r}, given in time units and not clock cycles, we simply divide expressions 3 and 4 by f_{CLK}, the clock frequency.

Table 10 summarizes the time length to load and read the CA array for some lattice dimensions. We can observe that as the lattice dimensions become larger, the time to load the CA array *tends* to approximate to the time required to read it. This is obvious if we think that the gain of r clock cycles mentioned above becomes irrelevant as $r \times c$ grows. In fact, when $r, c \to \infty$ we obtain from expressions 4 and 3

$$C_M = C_{M_w} = C_{M_r} \approx \frac{r \times c \times b}{8}. \tag{5}$$

As we are interested in generating the largest lattices possible, we can use expression 5 to the determine a good approximation to the effective speed-up. From the results presented on table 10, for a squared lattice, we can also conclude that the time grows quadratically with $(b/8) \cdot x^2$, where x is the side of the lattice. For a non-squared lattice, the time grows linearly with the rows or columns. The quadric behaviour shows whenever $r \to c$ or vice-versa.

As a final remark to this matter, we point out that there are still two existing overheads on the side of the hardware and software that were *not* measured accurately. From the hardware side, a small amount of time is required to read data from memory and write them to the serial buffer; this is about the same time it takes to read data from the serial buffer and write them to the memory. From the software side, it is necessary to consider the time to read the data from the graphical display buffer and write it to the serial buffer and vice-versa. Even though data is represented graphically as it is received, we believe that the software overhead is longer. However, we are able to safely conclude that the data transfer over the serial interface is the bottleneck.

The *effective* speed-up is determined solely by the lattice dimensions and cell state data bits; it does not depend on the present rule. Adding the computation time t_c to the expressions 1 and 5 we obtain the expression for the *effective* hardware simulation time for a single experiment, i.e.,

$r \times c$	C_{M_w}	C_{M_r}	t_{M_w} (µs)	t_{M_r} (µs)	C_{M_w}/C_{M_r}
24×24	76	96	1.14	1.44	79.17%
32×32	133	160	2.00	2.40	83.13%
40×40	206	240	3.09	3.60	85.83%
48×48	295	336	4.43	5.04	87.80%
56×56	400	448	6.00	6.72	89.29%
64×64	521	576	7.82	8.64	90.45%
72×72	658	720	9.87	10.8	91.39%

Table 10. Time required to load and read the CA array, reading and writing data to memory, with $b = 1$

initialize the array, iterate n times and read the results back to the PC (hence the multiplication by 2):

$$t_{HW_{eff}} \approx \frac{n}{f_{CLK}} + 2 \cdot (r \times c \times b) \cdot \left(\frac{1}{B} + \frac{1}{8 \cdot f_{CLK}} \right) \qquad (6)$$

where n is the number of iterations. Using expression 6 and the results from tables 1 and 2, we can now determine the percent reduction of the speed-up, p_s. Considering a lattice size of 56×56, for 10^8 and 10^9 iterations we obtain a reduction of about 0.9% and 0.09%, respectively. For a lattice size of 40×40 we obtain a reduction of about 0.46% and 0.046%. However, as n decreases, p_s increases drastically, e.g., for 10^5 iterations the percent reduction is 90.1% and 82.3%. This is expected because the time required to transfer data has a greater impact on a shorter computation time. Thus, the linear relationship observed in section 5.1 between speed-up and dimensionality only occurs for large values of n, where p_s is not significant.

6. Conclusions and Future Work

The implementation described in this work performs the evaluation of the whole CA array in parallel, and constitutes a fully functional and flexible system. However, some improvements can be envisioned. The synthesis and implementation processes run by the associated tools are automatic, which means that there is no control of the mapping of the CA to the FPGA resources. It is not guaranteed that CA cells are uniformly distributed across the FPGA, which leads to the degradation of performance and a lower degree of compactness—there is a waste of resources as the cells are spread irregularly and their neighbourhood interconnections have different lengths.

The major advantages of CA architectures are its local connectivity and spacial regularity, which allow exceptional performance on hardware. In future, solutions to optimize distribution and allocation of FPGA resources need to be investigated, for example, at the floorplanning level. Moving on to a multi-FPGA scenario with greater capacity, based, for instance, on Virtex-6 devices, would enable the support of larger lattices.

Acknowledgments This work was partially funded by the European Regional Development Fund through the COMPETE Programme (Operational Programme for Competitiveness) and by national funds from the FCT–Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-022701.

References

[1] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[2] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[3] V. Zhirnov, R. Cavin, G. Leeming, and K. Galatsis. An Assessment of Integrated Digital Cellular Automata Architectures. *Computer*, 41(1):38 –44, jan. 2008.

[4] Eric W. Weisstein. Game of life. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GameofLife.html.

[5] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

[6] Nikolaos Vlassopoulos, Nazim Fates, Hugues Berry, and Bernard Girau. An FPGA design for the stochastic Greenberg-Hastings cellular automata. In *2010 International Conference on High Performance Computing and Simulation (HPCS)*, pages 565–574, June 28 2010-July 2 2010 2010.

[7] Paul Shaw, Paul Cockshott, and Peter Barrie. Implementation of lattice gases using FPGAs. *The Journal of VLSI Signal Processing*, 12:51–66, 1996.

[8] Stephen Wolfram. *Cellular Automata And Complexity: Collected Papers*. Westview Press, 1994.

[9] Burton H. Voorhees. *Computational Analysis of One-Dimensional Cellular Automata*. World Scientific Series on Nonlinear Science. Series A, Vol 15. World Scientific, 1996.

[10] Eric W. Weisstein. Moore Neighborhood. From MathWorld–A Wolfram Web Resource `http://mathworld.wolfram.com/MooreNeighborhood.html`.

[11] Eric W. Weisstein. von Neumann Neighborhood. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/vonNeumannNeighborhood.html`.

[12] Paul Shaw and George Milne. A highly parallel fpga-based machine and its formal verification. In Herbert Grünbacher and Reiner W. Hartenstein, editors, *Field-Programmable Gate Arrays: Architecture and Tools for Rapid Prototyping*, volume 705 of *Lecture Notes in Computer Science*, pages 162–173. Springer Berlin Heidelberg, 1993.

[13] Tommaso Toffoli and Norman Margolus. *Cellular automata machines: a new environment for modeling*. MIT Press, Cambridge, MA, USA, 1987.

[14] Tommaso Toffoli and Norman Margolus. Cellular automata machines. *Complex Systems*, 1(5):967–993, 1987.

[15] Jon Bennett, Chuck Silvers, Paul Callahan, Eric S. Raymond, and Vladimir Lidovski (2011-2012). XLife. `http://litwr2.atspace.eu/xlife.php`.

Computational Block Templates Using Functional Programming Models

Paulo Ferreira[†], João Canas Ferreira[‡], José Carlos Alves[‡]
[†]*Instituto Superior de Engenharia do Porto,* [‡]*Universidade do Porto,Faculdade de Engenharia*
pdf@isep.ipp.pt, jcf@fe.up.pt, jca@fe.up.pt

Abstract

The elaboration of computational blocks using declarative models, typical of functional languages, allows the use of a parameterized template for the hardware design. This paper shows how such a template can be created, for the hardware implementation of computational blocks based on a declarative model, and how it can be used to provide design space exploration alternatives and hardware acceleration for Erlang based embedded systems. The template uses a flexible TCL preprocessor for the HDL generation and control of the design space alternatives.

1. Introduction

The use of custom hardware to implement functional or declarative languages has been studied by many. This research was usually oriented to the complete implementation of functional languages in hardware, like the elaboration of a LISP based microprocessor [1] or the the creation of machines such as SKIM [2] and the parallel execution of functional programs [3]. This area of research is a consequence of study of High Level Computer Architectures in the 1970-1980 decades [4].

Different functional languages have been applied to hardware description tasks, such as Lava [5], Shard [6] or BlueSpec [7], but the majority of work that tries to apply high-level languages to hardware description uses imperative (and object-oriented) languages or derivatives such as C, C++, and Java.

The "absence of state" in functional languages [8], is usually a shock for programmers fluent only in imperative programming languages. However, in parallel architectures the existence of state, means that the state must be shared among different execution threads (or similar abstractions), originating many difficult problems [9].

A very different parallel computing model is used on the Erlang functional language [10] where a system is built from independent processes, not sharing any state and communicating only by message passing. This has a strong resemblance to hardware design, where one can build a system from different hardware blocks, without a global physical memory, using point to point connections, crossbars or networks-on-chip for communication.

The heterogeneous architecture of an Erlang system based has two interesting characteristics:

- It has a strong similarity to hardware.
- It has no strong coupling between the different processes.

This might mean that a Erlang based system, could be mapped entirely into heterogeneous hardware computational blocks. That would be very interesting. However, a more achievable task is the mapping into hardware of only some of those processes, selected considering the requirements of the full system.

As the coupling between the different processes is done by the messages, as long as the same messages arrive in the same format, with the same data, and a "process" does the same work, the "process" can be "migrated" from software to hardware without changes on the rest of the system.

In this paper some different implementation alternatives for those computational blocks, are presented together with a set of tools that facilitate their creation, use and test. This will allow the design space exploration of FPGA implemented coprocessors for Erlang based embedded systems.

2. The general template

The translation into a state machine template of a general function expressed in a declarative form (usually tail recursive) is described in [11]. In the remainder of this article the Erlang language is used for the source code examples. On listing 1 there is a small Erlang code fragment that implements the greatest common divider algorithm, with line numbers added for reference purposes.

```
1: gcd(A, B) when A<B -> gcd(B, A);
2: gcd(A, 0) -> A;
3: gcd(A, B) -> gcd(A-B, B).
```

Listing 1 – A sample Erlang greatest common divider (GCD) function

An Erlang function is composed by a series of clauses (lines of code) and each clause has:

- A condition, that may be explicit or implicit.
- An action, that specifies the steps to be executed if the associated condition is true.

The conditions are tested sequentially, and the first one found to be true triggers the execution of the associated action. Translated into common english, the above code fragment can be written as:

1. If the first argument of the function (`A`) is smaller than the second (`B`), swap them and call again the same function.

2. If the second argument (`B`) is zero, then the result of the function is the first argument (`A`) and the function execution ends here.

3. Call again the function but replacing the first argument (`A`) with the difference between both arguments (`A-B`).

On the first clause, there is an explicit condition (`A<B`). On the second clause the condition is implicit (`B==0`), and on the last clause one can assume also the presence of an implicit (true) condition.

As an example the execution of the function call `gcd(15,25)` can be traced as done in table 1.

Table 1. Tracing the execution of `gcd(15,25)`

Call	Clause	Matches	Result
`gcd(15,25)`	1	Yes	`gcd(25,15)`
`gcd(25,15)`	1	No	See next clause
`gcd(25,15)`	2	No	See next clause
`gcd(25,15)`	3	Yes	`gcd(25-15,15)`
`gcd(10,15)`	1	Yes	`gcd(15,10)`
`gcd(15,10)`	1	No	See next clause
`gcd(15,10)`	2	No	See next clause
`gcd(15,10)`	3	Yes	`gcd(15-10,10)`
`gcd(5,10)`	1	Yes	`gcd(10,5)`
`gcd(10,5)`	1	No	See next clause
`gcd(10,5)`	2	No	See next clause
`gcd(10,5)`	3	Yes	`gcd(10-5,5)`
`gcd(5,5)`	1	No	See next clause
`gcd(5,5)`	2	No	See next clause
`gcd(5,5)`	3	Yes	`gcd(5-5,5)`
`gcd(0,5)`	1	Yes	`gcd(5,0)`
`gcd(5,0)`	1	No	See next clause
`gcd(5,0)`	2	Yes	Result: 5

The corresponding flowchart that triggers each action, when a condition is true, is shown in Figure 1.

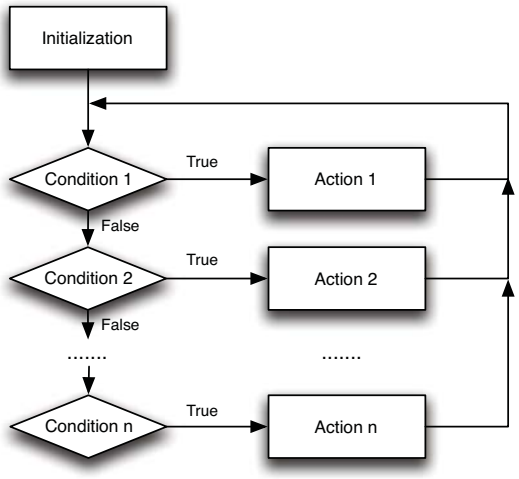


Figure 1. General execution flowchart

The absence of state in a functional language means that a variable does not change it's value after being assigned. That is the origin of the "single assignment" designation of those languages. So, if a variable does not change it's value and all the variables used in the conditions are known when a function is called, on figure 1 all the conditions can be tested in parallel (for speed) providing only the action corresponding the first condition found to be true is executed.

3. The template implementation

3.1. Tools used

For testing the different alternatives of implementation there was the need of generating the required Verilog source code, in an automatic manner. To create automatically Verilog (or VHDL) code, when the language's preprocessor and/or the generate statement is not enough, many authors propose (and use) many different alternatives.

One of the common alternatives is to write a custom program in a compiled language (such as C or C++) to write the required HDL files [12] [13]. Other options involve the use of interpreted languages [14] [15], chosen by their interactivity.

The pre-processor used was chosen, taking into account the following requirements:

- The pre-processor should not be Verilog specific
- The language used when programming the preprocessor should also be useful for other common FPGA design tasks

When all the previous requirements were taken into account, the G2 preprocessor [16] was chosen. It is a small open source tool that supports mixing TCL with the desired target language. TCL is an Open Source language [17], built in a wide range of EDA tools and is useful for many other tasks, from creating FPGA synthesis projects to controlling digital simulations.

It supports controlling the source code generation with the TCL decision constructs, and the use inside Verilog of TCL variables.

At the time of writing, the computational template automatically generates the desired Verilog code for the the full implementation of the required Verilog modules from a very simple TCL specification, giving alternatives for the parallel or sequential evaluation of the conditions.

As an example the required text for the Erlang GCD example (Listing 1) is shown on listing 2, with the corresponding Erlang code in comments (the lines starting with #).

 ISBN: 978-972-8822-27-9

```
set INPUT_BITS  64
set OUTPUT_BITS 32

set bits(OPR_A) 32
set bits(OPR_B) 32

set OUTPUT_SOURCE "OPR_A"

set INIT_STATEMENT  "OPR_A<=din\[63:32\];
                     OPR_B<=din\[31:0\];"

#  1:  gcd(A, B) when A<B -> gcd(B, A);
set condition(0)    "(OPR_A<OPR_B)"
set action(0)       "OPR_A<=OPR_B;
                     OPR_B<=OPR_A;"

# 2:  gcd(A, 0)->A;   % see OUTPUT_SOURCE
set condition(1)    "OPR_B==0"
set action(1)       $END_OF_CALC

# 3:  gcd(A, B)->gcd(A-B, B).
set condition(2)    "1"
set action(2)       "OPR_A<= OPR_A-OPR_B;"
```

Listing 2 – GCD circuit specification

3.2. Circuit Architecture

The general block architecture of the circuit can be found on figure 2. The next values of the data register are determined by the conditions that trigger one of the actions.

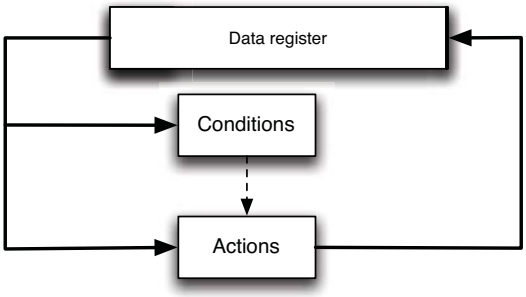


Figure 2. General Hardware Template

The evaluation of the conditions can be done sequentially, evaluating one condition at a time. If a condition is found to be true, the corresponding action is triggered, and after the execution of the action, the evaluation of the conditions starts again on the first condition. This also means that when a condition is found to be true, the following conditions are not tested, even if they are also true. The order of the conditions is used to specify their relative priority.

Other alternative is the evaluation in parallel of all the conditions, with the highest priority condition having precedence over the others. As the conditions are all evaluated in parallel, the respective circuit incorporates a priority encoder, triggering only the action corresponding to the highest priority.

The block corresponding to the actions can be built from independent blocks, one for each action, or could be implemented in a single (ALU-like) module for resource sharing.

From a circuit specification such as the one shown on Listing 2, and defining the desired type of condition evaluation (sequential/parallel) giving the variable `PARALLEL` the value 0 or 1, the tool generates the complete Verilog code of a module to implement the desired function, using a Finite State Machine (FSM) based template.

The generated Verilog module has the following characteristics:

- Input and output ports with handshake signals
- Sizing of the data registers
- Sizing of the FSM control registers on the sequential implementation
- Sizing of the condition checking signals on the parallel implementation
- Construction of the FSM logic on both implementations
- Hierarchical connection of similar modules

As Erlang has no notion of variable size (integers grow in size automatically), the size of the variables is (for now) defined only in the specification file, but in the future could be extracted from special pragmas, placed in the Erlang source code.

The implemented registers that do not depend on the data variables, such as those needed for the FSM control, are automatically sized according to the number of states needed.

The input and output handshake signals (two on the input: `start` and `busy`; and two on the output: `start_out` and `busy_in`) are handled by the implicit FSM control of each module, without user intervention. Besides the obvious inclusion of the created modules on other circuits, they allow the cascading of different circuits, in order to obtain the composition of different computational functions.

A more interesting feature is the automatic instantiation of a different computational block in an action of a clause. Supposing the required action is too complex to be executed by combinatorial logic in a single clock cycle, having a block that implements the required function, that block can be included and interconnected specifying it as in listing 3.

```
set module(1)   "div"
set aux_input_bits(1)   64
set aux_output_bits(1)  32

set din_aux_source(1)  "
{(`OPR_D*(`OPR_A-`OPR_C+1'b1)),`OPR_C}"

set dout_aux_source(1)  "`OPR_A<= `OPR_A;
                         `OPR_B<= `OPR_B;
                         `OPR_C<= `OPR_C+1;
                         `OPR_D<=dout_aux_1;"
```

Listing 3 – Sub-Module Specification Example

The different lines specify, the name of the existing module, the desired bit widths for input and output, the data placed on the input of the auxiliary module, and what to do when the auxiliary module has the output ready.

Only the total data input size is specified, because the size of each variable can be defined as a function of the input size, simplifying the module's connection. All the handshake lines are connected to adequate control logic on the main module.

```
// ## data register
reg [63:0] datareg;
//###  extract operands:
`define  OPR_A  datareg[31:0]
`define  OPR_B  datareg[63:32]
...
always @(posedge clock)
begin
 if ( reset )
 begin
   state_calc    <= WAIT_DATA;
   end_calc      <= 1'b1;
 end
 else
 begin
   case( state_calc )
  WAIT_DATA:
      if ( start_calc )
      begin
     // initialization of the variables
             state_calc <= CALC;
             end_calc <= 1'b0;
             `OPR_A<=din[63:32];
             `OPR_B<=din[31:0];
         end
     CALC:
        // calculation
         ...
endcase
 end
...
```

Listing 4 – Excerpts of the generated Verilog code

```
...
wire  [2:0] cond;
assign  cond[0]  = (`OPR_A<`OPR_B) ;
assign  cond[1]  = `OPR_B==0 ;
assign  cond[2]  = 1 ;
...
CALC:
  casex  ( cond )  // calculation
      3'bxx1:
       begin
         `OPR_A<=`OPR_B;
         `OPR_B<=`OPR_A;
       end
      3'bx1x:
       begin
         end_calc <= 1'b1;
         state_calc <= 0;
       end
      3'b1xx:
       begin
         `OPR_A<= `OPR_A-`OPR_B;
       end
  endcase  // casex
...
```

Listing 5 – Excerpts of the generated Verilog code (parallel version)

On Listing 4 there are some excerpts of the generated Verilog code common to both versions, where one can see the definitions of the data register, of the individual variables (as segments of the data register) and also the simple FSM that controls the calculations.

On the parallel condition test version of the architecture, the individual conditions are packed into a vector, and a `casex` statement is used to create a priority encoder, corresponding to the calculation, as can be seen on Listing 5.

```
...
CALC:
case  ( present_cond )  // calculation
     0:  begin
           if ( (`OPR_A<`OPR_B) )
              begin
               `OPR_A<=`OPR_B;
           `OPR_B<=`OPR_A;
     present_cond<=0;
      end
            else
     present_cond<=present_cond+1;
       end
      1:  begin
            if ( `OPR_B==0 )
            begin
              end_calc <= 1'b1;
              state_calc <= 0;
      present_cond<=0;
end
            else
           present_cond<=present_cond+1;
           end
      2:  begin
            if ( 1 )
            begin
                `OPR_A<= `OPR_A-`OPR_B;
                 present_cond<=0;
             end
            else
       present_cond<=present_cond+1;
           end
      default: present_cond<=0;
 endcase
...
```

Listing 6 – Excerpts of the generated Verilog code (sequential version)

The sequential test of the conditions uses an additional register (`present_cond`) to track the condition being tested and tests one condition per state as can be seen on Listing 6.

4. Results

As a test suite, several simple algorithms (taken from [18]) were coded, simulated with Icarus Verilog, and synthesized with ISE 12.4 from Xilinx, having as target a Spartan 3E FPGA. The results appear on table 2. An interesting result is that for simple algorithms, the parallel evaluation of the conditions, besides being faster, uses less resources. This happens because when evaluating the clauses sequentially, some resources are needed to track the current condition being tested.

On this type of computational architectures, the highest performance (lower execution time) version is also the version that uses the minimum of resources, maximizing two of the most pressing constraints for an hardware designer.

The execution times are given for a system with a Microblaze processor at a clock frequency of 50 Mhz, on a Spartan 3E FPGA. They represent in order, the time needed for the implemented hardware core to calculate the desired results from the input data, the time needed for the running Erlang system to make the same task using the implemented hardware, and the time needed using a software approach coded entirely in Erlang.

Table 2. Data for sample modules

			Times (μs):		
Algorithm	Vers.	Slices	HW	HW+SW	SW Only
Binomial	Serial	325	22,08	772	68500
Binomial	Parallel	321	8,56	758	68500
Collatz	Serial	325	2,24	752	13000
Collatz	Parallel	321	5,7	756	1300
Fibonacci	Serial	110	1,68	752	2530
Fibonacci	Parallel	103	0,86	751	2530
GCD	Serial	166	0,44	751	1500
GCD	Parallel	125	0,22	750	1500

An additional resource sharing alternative was tested, coding in the same hardware module two different computational blocks, however no resources sharing or execution time benefits were found.

5. Conclusions

There are two subjects of this paper: the tools used and the proposed architecture. The tools used (templates built using a TCL based preprocessor) allowed the prototyping and the reutilization of templates for different computational tasks. The templates also support the automatic instantiation of sub-modules, in the cases where an action is too complex to be executed in a single clock cycle. The transparency of the preprocessor (the user has full control over all the code expansion phases) also means that any bugs were easily located and corrected.

The preprocessor was also used for the creation of the testbenches, and the TCL know-how gained was applied in the automatization of common Xilinx synthesis workflows.

The flexible design architecture of the computational blocks allows:

1. An architecture suitable for the future automatic translation from Erlang into hardware.

2. The use of different design alternatives for different sub-blocks.

In the future, the used templates will be integrated in a framework for automatically exploring design alternatives, reusing some of this work, providing hardware acceleration on embedded Erlang based systems, with demonstrated advantages.

References

[1] Guy L. Steele Jr. and Gerald Jay Sussman. Design of a LISP-based microprocessor. *Commun. ACM*, 23(11):628–645, 1980.

[2] William Stoye. *The implementation of functional languages using custom hardware*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, December 1985.

[3] Philip C. Treleaven and Geoffrey F. Mole. A multiprocessor reduction machine for user-defined reduction languages. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 121–130, La Baule, United States, 1980. ACM.

[4] Yaohoan Chu. Concepts of high-level language computer architecture. In Yaohan Chu, editor, *High-Level Language Computer Architecture*, chapter 1, pages 1–14. Academic Press, New York, 1975.

[5] Satnam Singh. System level specification in Lava. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 370–375, Munich, 2003.

[6] Xavier Saint-Mleux, Marc Feeley, and Jean-Pierre David. SHard: a Scheme to hardware compiler. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, Portland, Oregon, 2006. University of Chicago.

[7] Rishiyur S. Nikhil and Kathy R. Czeck. *BSV by Example*. Bluespec Inc., Framingham, Massachusetts, 2010.

[8] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, 1999.

[9] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[10] Joe Armstrong. *Programming Erlang*. Pragmatic Bookshelf, Raleigh, NC, 2007.

[11] Paulo Ferreira, João Canas Ferreira, and José Carlos Alves. Erlang inspired hardware. In *FPL 2010 - International Conference on Programmable Logic and Applications*, pages 244–246, Milano, 2010. IEEE.

[12] David B. Thomas and Wayne Luk. FPGA-Optimised uniform random number generators using LUTs and shift registers. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 77–82, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Wei Zhang, Vaughn Betz, and Jonathan Rose. Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Trans. Reconfigurable Technol. Syst.*, 5(1):6:1–6:26, March 2012.

[14] Gary Spivey. EP3: An extensible Perl preprocessor. In *International Verilog HDL Conference and VHDL International Users Forum, 1998. IVC/VIUF 1998. Proceedings.*, pages 106–113, March 1998.

[15] Adrian Lewis. *Prepro: Embedded Perl/Python preprocessor*. Corner Case Research, 2007.

[16] Koen Van Damme. *The G2 Preprocessor*, 2002.

[17] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, Reading, Massachusetts, 1994.

[18] Mikaël Rémond. *Erlang programmation*. Eyrolles, Paris, 2003.

 ISBN: 978-972-8822-27-9

Arquiteturas multiprocessamento

 ISBN: 978-972-8822-27-9

Using SystemC to Model and Simulate Many-Core Architectures

Ana Rita Silva[1], Wilson M. José[1], Horácio C. Neto[2], Mário P. Véstias[3]
INESC-ID[1]; INESC-ID/IST/UTL[2]; INESC-ID/ISEL/IPL[3]
anaritasilva@esda.inesc-id.pt, wilson@esda.inesc-id.pt, mvestias@deetc.isel.pt, hcn@inesc-id.pt

Abstract

Transistor density has made possible the design of massively parallel architectures with hundreds of cores on a single chip. Designing efficient architectures with such high number of cores is a very challenging task. Simulation of many-core architectures can help designers to explore the design space.

This paper addresses the applicability of SystemC to simulate many-core architectures. We demonstrate the use of SystemC to model a system of P processors executing matrix multiplications. The simulation of the model allows analyzing the results regarding the number of transfers and the number of clock cycles required to complete each transaction.

1. Introduction

During the last decade massively parallel systems have been proposed as high-performance computing architectures delivering very high computing speeds which make them particularly attractive for scientific applications.

Whether in Application Specific Integrated Circuits (ASIC) or Field Programmable Gate Array (FPGA), these are very complex systems whose simulations must be done at system level since Register-Transfer Level (RTL) simulations are two or three orders of magnitude slower. Even so, only a few existing simulators are able to sustain many-core architectures with acceptable simulation times. The COTSon team at HP labs proposed a trace-driven simulator fed with thread instruction streams computed by a single-core full system simulator [1]. The approach only considers an idealized architecture with a perfect memory hierarchy, without any interconnect, caches or distribution of memory banks.

Most of the other recent approaches parallelized discrete-event simulators with varying levels of detail. SlackSim [2] is a cycle-level simulator allowing individual cores to progress at different paces in a controlled manner. In [3], the authors proposed a system level architectural modeling methodology for large designs based on SystemC. The proposed methodology uses normal function calls and SystemC method processes to implement a very fast but accurate model to evaluate a large design at system level. The platform was used to model a NoC-based SoC. The experimental results show improvement up to 98% in elaboration time and up to 90% in simulation time for small size NoCs.

SystemC is a system design language that has evolved in response to a pervasive need for improving the overall productivity for designers of electronic systems [4].

One of the primary goals of SystemC is to enable system-level modeling – that is, modeling of systems above RTL, including systems that might be implemented in software, hardware, or some combination of the two [5].

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

SystemC is based on the C++ programming language, which is an extensible object oriented modeling language. It extends the C++ data types with additional types useful for modeling hardware that support all the common operations and provide special methods to access bits and bit ranges.

SystemC adds a class library to C++ to extend its capabilities, effectively adding important concepts such as concurrency, timed events and data types. This class library is not a modification of C++, but a library of functions, data types and other language constructs that are legal C++ code [6].

This work presents a SystemC design specification and model implementation of a 2D-array multiprocessor system executing matrix multiplications using *P* processors.

The rest of the paper is organized as follows. The next section describes an overview of SystemC. Section 3 describes the parallel dense matrix multiplication algorithm proposed. Section 4 describes the system-level model of the architecture. Experimental results are discussed in Section 5. Finally, Section 6 concludes the paper.

2. Overview of SystemC

Modules are the basic building blocks for partitioning a design in SystemC. They allow designers to break complex systems into smaller, more manageable pieces, and to hide internal data representation and algorithms from other modules [5].

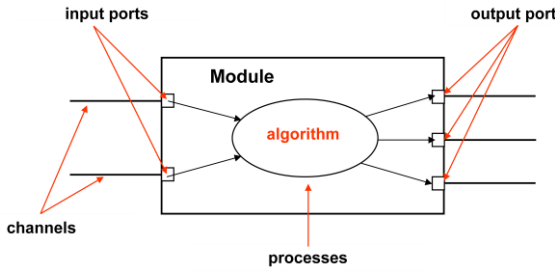


Fig. 1. Module structure.

A typical module contains processes that describe the functionality of the module, ports through which the module communicates with the environment, internal data and channels for maintenance of model state and communication among the module´s processes, among other modules (figure 1 sketches a module structure).

A SystemC module is simply a C++ class definition and is described with the *SC_MODULE* macro (figure 2).

```
// File example.h //
#include <iostream>
#include <systemc.h>

SC_MODULE (example)
{
    sc_in <bool> clk;
    sc_in <bool> port_in;
    sc_out <bool> port_out;

    SC_CTOR (example) {
            SC_METHOD (example_method);
            sensitive_pos << clk;
    }

    void example_method();
};


// File example.cpp //
#include "example.h"

void example:: example_method (){
    port_out=port_in;
}
```

Fig. 2. Code example.

Within the class definition, ports are usually the first thing declared because they represent the interface to the module. Local channels and sub-module instances come after that.

Next, we place the class constructor. Each module requires a constructor block (*SC_CTOR*), which maps designated member functions to processes and declares event sensitivities; its argument must be the name of the module being declared. The header file finishes out with the declarations of processes, helper functions and other private data.

In the example of figure 2, the definitions of the process reside in a separate file from the module declaration. The traditional template places all the instance creation and constructor definitions in header (.h) files. Implementation of processes and helper functions are deferred to the compiled (.cpp) file.

In SystemC the basic unit of functionality is called a process. A process must be contained in a module – it is defined as a member function of the module and declared to be a SystemC process in the module´s constructor [5]. The processes within a module are concurrent.

An event is an object, represented by class *sc_event*, that determines whether and when a process´s execution should be triggered or resumed; it has no value and no duration. An event is used to represent a condition that may occur during the course of simulation and to control the triggering of processes accordingly [5]. We can perform only two actions with a *sc_event*: wait for it or cause it to occur.

The owner of the event is responsible for reporting the change to the event object. The act of reporting the change to the event is called *notification*. The event object, in turn, is responsible for keeping a list of processes that are sensitive to it. Thus, when notified, the event object will inform the scheduler of which processes to trigger [5].

SystemC has two kinds of processes: method processes, defined with the macro SC_METHOD, and thread processes, defined with the macro SC_THREAD.

SC_THREAD processes are started once and only once by the simulator. Once a thread starts to execute, it is in complete control of the simulation until it chooses to return control to the simulator. Variables allocated in SC_THREAD processes are persistent.

SystemC offers two ways to pass control back to the simulator. One way is to simply exit (e.g., return), which has the effect of terminating the thread for the rest of the simulation. When an SC_THREAD process exits, it is gone forever, therefore SC_THREADs typically contain an infinite loop containing at least one wait. The other way to return control to the simulator is to invoke the module wait method.

 ISBN: 978-972-8822-27-9

The *wait* (dynamic sensitivity list) suspends the SC_THREAD process [4].

When *wait* executes, the state of the current thread is saved, the simulation kernel is put in control and proceeds to activate another ready process. When the suspended process is reactivated, the scheduler restores the calling context of the original thread, and the process resumes execution at the statement after the *wait* [4].

During simulation a thread process may suspend itself and designate a specific event *event_name* as the current event on which the process wishes to wait. Then, only the notification of *event_name* will cause the thread process to be resumed.

SC_METHOD processes never suspend internally. Instead, SC_METHOD processes run completely and return. The simulation engine calls them repeatedly based on the dynamic or static sensitivity list.

In terms of dynamic sensitivity list, SC_METHOD processes may not call the *wait* method, because they are prohibited from suspending internally. Instead of calling *wait()*, a method process calls *next_trigger()* to specify the event that must occur for it to be triggered next time.

Until the event occurs, the static sensitivity list is temporarily disabled. Unlike *wait()*, however, calling *next_trigger()* does not suspend the current method process. Instead, execution of the method process continues to the end, and next time the method process will be invoked only when the event specified by *next_triggered()* occurs [5]

SystemC provides another type of sensitivity for processes called static sensitivity. Static sensitivity establishes the parameters for resuming before simulation begins.

Like method processes, a thread process may have a sensitivity list describing the set of events to which it should normally react. As mentioned, when we encounter a *wait()* statement, the execution of a thread process is suspended. When any of the events in the sensitivity list occurs, the scheduler will resume the execution of the process from the point of suspension [5].

Interfaces, ports and channels are the way through which SystemC implements synchronization and communication at system level.

A SystemC interface is an abstract class that inherits from *sc_interface* and provides only pure virtual declarations of methods referenced by SystemC channels and ports. No implementations or data are provided in a systemC interface.

A SystemC channel is a class that implements one or more SystemC interface classes and inherits from either *sc_channel* or *sc_prim_channel*. SystemC has two types of channels: primitive and hierarchical.

SystemC´s primitive channels are known as primitive because they contain no hierarchy, no processes, and are designed to be very fast due to their simplicity. All primitive channels inherit from the base class *sc_prim_channel* [4]. The simplest channels are *sc*_signal, sc_*mutex*, *sc_semaphore*, and *sc_fifo*.

In this work will use *sc_fifo* channels to model the communication between processors and memories. *sc_fifo* is probably the most popular channel for modeling at the architectural level. First-in first-out queues are a common data structure used to manage data flow; by default, an *sc_fifo*<> has a depth of 16. The data type of the elements also needs to be specified. An *sc_fifo* may contain any data type including large and complex structures.

Two interfaces, *sc_fifo_in_if*<>, and *sc_fifo_out_if*<>, are provided for *sc_fifo*<>. Together these interfaces provide all of the methods implemented by *sc_fifo*<>.

A SystemC port is a class template with and inheriting from a SystemC interface. The port base class is called *sc_port*. A port is an object through which a module, and hence its processes, can access a channel´s interface. A channel cannot be connected to a port if it doesn´t implement the port´s interface. There are three types of ports: in, out or inout. Each port has a data type, passed on between the angle brackets (<>, template class).

At last, the SystemC library provides its own definition of *main()*, which in turns calls *sc*_main. Within *sc_main()*, code executes in three distinct major phases, which are elaboration, simulation, and post-processing.

Elaboration establishes hierarchy and initializes the data structures. Elaboration consists of creating instances of clocks, design modules, and channels that interconnect designs. At the end of elaboration, *sc_start()* invokes the simulation phase. During simulation, code representing the behavior of the model executes.

Finally, after returning from *sc_start()*, the post-processing phase begins. Post-processing is mostly optional. During post-processing, code may read data created during simulation and format reports or otherwise handle the results of simulation. Post-processing finishes with the return of an exit status from *sc_main()* [4].

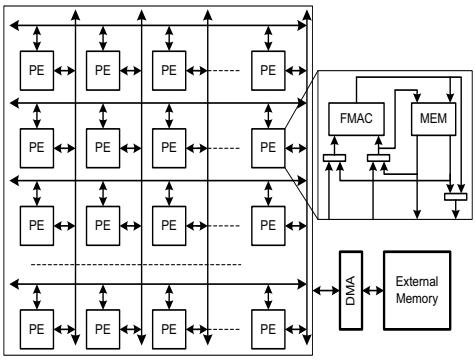


Fig. 3. High-performance many-core architecture.

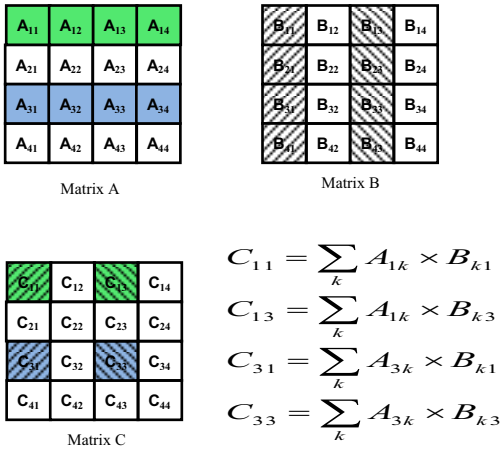


Fig. 4. Matrix Multiplication.

3. Matrix Multiplication Algorithm

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

The parallel architecture is organized as a 2D mesh of execution (see figure 3). Each core unit consists mainly of a floating-point multiply and accumulate unit (FPMAC) and a dual-port memory. Access to the external memory is controlled with a direct memory access (DMA) module that can deliver burst transactions with one transfer per cycle.

To facilitate the presentation of the algorithm, we consider that the mesh array is square and define matrix C to be the result of the product between two square matrices, A and B. The results can be easily generalized to non-square meshes. We also consider that all matrices involved are square and have the same size $n \times n$, and that its dimensions are multiple of the partitioned blocks dimensions. Again, this consideration does not limit in any way the generality of the results.

Each of the $p = q \times q$ processors, where $q = \sqrt{p}$, is responsible for calculating one block of matrix C with size $\frac{n}{q} \times \frac{n}{q}$. Each one of these blocks is partitioned, according to the memory limitations of the processor, in sub blocks C_{ij} with size $y \times x$.

To generate a C_{ij} block, the processor must multiply a $y \times n$ block of matrix A with a $n \times x$ block of matrix B. This multiplication is segmented as a sequence of $k_0 = \frac{n}{z}$ block multiplications as specified in equation (1).

$$C_{ij} = \sum_{k=1}^{k_0} A_{ik} \times B_{kj} \qquad \textbf{(1)}$$

Therefore, each partial block multiplication consists of the multiplication of a $y \times z$ sub block A_{ik} with a $z \times x$ sub block B_{kj} resulting in a partial sub block C_{ij} of size $y \times x$. The final C_{ij} result is obtained after accumulating the k_0 partial block multiplications.

Figure 4 illustrates how the system performs the product between the matrices A and B considering four processors and 16 sub-blocks (4 per processor).

The processor P1 is responsible for calculating blocks C_{11}, C_{12}, C_{21} and C_{22}, processor P2 for blocks C_{13}, C_{14}, C_{23} and C_{24}, processor P3 for calculating blocks C_{31}, C_{32}, C_{41} and C_{42}, and processor P4 for blocks C_{33}, C_{34}, C_{43} and C_{44}. The order in which the blocks of matrix C are calculated corresponds to the equations defined in figure 4 (for the first "iteration").

To calculate the four blocks, C_{11}, C_{13}, C_{31}, C_{33}, it is only necessary to transfer the data from two rows of blocks of A and two columns of blocks of B. In fact, all the processors in the same row require the same data from matrix A, while all the processors in the same column require the same data from matrix B. Therefore, each sub block fetched from memory is broadcast to a row (column) of $\sqrt{p}$, processors.

The total number of communications is given by equation (2).

$$N_{comm} = \frac{n^3}{\sqrt{p}}\left(\frac{1}{x} + \frac{1}{y}\right) + n^2 \qquad \textbf{(2)}$$

The number of communications does not depend on the dimension z of the sub blocks from matrix A and matrix B, thus z can be simply made equal to 1 in order to minimize the local memory required.

The partial block multiplication is implemented such that, each processor receives the sub block A_{ik} with y elements and stores it. Then it receives the elements of the sub block B_{kj} which are multiplied the corresponding locally stored elements of A_{ik}, resulting in a partial sub block C_{ij}. This process repeats n times, after which the final sub block C_{ij} is obtained.

The total number of computation cycles, assuming a processor throughput of one accumulation/cycle, is given by

$$N_{compcycles} = \frac{n^3}{p} \quad (3)$$

where n^3 is the total number of multiply-add operations to be performed by the p processors.

The minimal execution time is achieved when all the communications, except the initial and final overhead, can be totally overlapped with the computations, that is, when the number of communication cycles required is lower than the number of computation cycles, $N_{comp} > N_{comm}$.

If there is full overlap, the total execution time is given by

$$N_{execcycles} = \frac{n^3}{p} + Ovhd_{INI} + Ovhd_{END} \approx \frac{n^3}{p} \quad (4)$$

The initial overhead, $Ovhd_{INI}$, corresponds to the number of cycles it takes until all data becomes available for the last processor(s) to initiate the computations (that is, when the first element of the last required block of B arrives). The final overhead, $Ovhd_{END}$, corresponds to the additional number of cycles needed to write back the very last blocks of C.

These initial and final communication overheads are negligible (for large matrices) and, therefore, are not detailed herein.

4. Architecture model in SystemC

This section describes the implementation of the system model using SystemC. For ease of explanation, we describe the approach considering only four processors.

The model consists of three modules, where modules *Matrix* and *Result* represent the DMA and module *Processors* represents the 2D-Array (figure 5).

The communication between modules is performed by FIFOs of type *sc_fifo*. Each FIFO has only one direction, so it is necessary one FIFO to communicate with module *Result* and, for each processor, one FIFO to communicate with module *Matrix*.

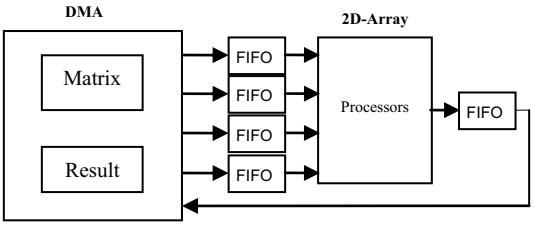


Fig. 5. SystemC model of the many-core architecture.

Module *Matrix* stores the values of the matrices *A* and *B* and consists of two thread processes (*SC_THREAD)* that are responsible for sending, in the correct order, the data to the processors through FIFOs (figure 6).

```
(…)
SC_MODULE(matrix) {
  sc_fifo_out <int> out_matrix1;
  sc_fifo_out <int> out_matrix2;
  sc_fifo_out <int> out_matrix3;
  sc_fifo_out <int> out_matrix4;

  int*A, *B;
  sc_event  MA_event, MB_event;
  void matrixA_thread();
  void matrixB_thread();

  SC_CTOR(matrix)
  : out_matrix1("out_matrix1"),
  out_matrix2("out_matrix2"),
  out_matrix3("out_matrix3"),
  out_matrix4("out_matrix4")
  {
      SC_THREAD(matrixA_thread);
      SC_THREAD(matrixB_thread);
      if ( (A = (int *) malloc((total*total) * sizeof(int)))
          == NULL ) {
          printf("Out of memory\n");
      }
      if( (B = (int *) malloc((total*total)  * sizeof(int)))
          == NULL ) {
          printf("Out of memory\n");
      }
  }
};
```

Fig. 6. File Matrix.h.

The first process is responsible for sending data from matrix *A* and the second for sending data from matrix *B*. The modules send the data words one at a time, starting by sending one block *A* followed by one block *B* and so on.

The module *Matrix* stops and waits, whenever the module *Processors* wants to send data to module *Result*. The synchronization of the two processes is maintained by two events (*sc_event)* triggered by the respective threads.

The module *Processors* consists of p thread processes (*SC_THREAD)*, being p the number of processors (figure 7). Each thread process receives and stores the two blocks and performs the operations. The operations are initiated at the moment the first value of the second block is available.

```
(…)
SC_MODULE(Processors) {

  sc_fifo_in <int> in_matrix1;
  sc_fifo_in <int> in_matrix2;
  sc_fifo_in <int> in_matrix3;
  sc_fifo_in <int> in_matrix4;
  sc_fifo_out <int> out_result;
  (…)

  void B1_thread();
  void B2_thread();
  void B3_thread();
  void B4_thread();

  SC_CTOR(Processors)
  : in_matrix1("in_matrix1"),in_matrix2("in_matrix2"),
  in_matrix3("in_matrix3"), in_matrix4("in_matrix4"),
  out_resul("out_resul")
  {
          SC_THREAD(B1_thread);
          SC_THREAD(B2_thread);
          SC_THREAD(B3_thread);
          SC_THREAD(B4_thread);
          (…)
   }
};
```

Fig. 7. File Processor.h.

The product between the two blocks corresponds to partial results of block *C*, which are stored in the processor. After obtaining the final results, they are sent to the *Result* module through a FIFO. The processors send the data to the FIFO one word at a time.

```
(…)
SC_MODULE(Result) {

    sc_fifo_in <int> in_result;

    void RES_thread();

    SC_CTOR(Result)
    : in_result("R")
    {
          SC_THREAD(RES_thread);
     }
};
```

Fig. 8. File Result.h.

Module *Result* consists of one thread process (*SC_THREAD)*, which has the function to read the final results obtained by the processors (figure 8).

5. Simulation and Results

The presented model was implemented and simulated for matrices of different sizes and different blocks, considering four processors.

Tables 1, 2 and 3 show the results obtained with regard to the number of data transfers, the total number of clock cycles and the execution time, considering z=1.

Block A	Block B	Transfers	Cycles	Time (s)
4x1	1x2	802.816	802.818	55
4x1	1x4	540.672	540.678	45
4x1	**1x8**	**409.600**	**528.486**	**32**
4x1	1x16	344.064	528.582	32
8x1	1x2	671.744	671.750	50
8x1	1x4	409.600	528.490	31
8x1	1x8	278.528	528.586	31
8x1	1x16	212.992	528.778	28
16x1	1x4	344.064	528.594	31
16x1	1x8	212.992	528.786	28
16x1	1x16	147.456	529.170	26

Table 1. Matrix with size 128x128 and z=1.

Block A	Block B	Transfers	Cycles	Time (min)
4x1	1x2	6.356.992	6.356.994	9
4x1	1x4	4.259.840	4.259.846	6
4x1	1x8	3.211.264	4.210.790	4
4x1	1x16	2.686.976	4.210.886	4
8x1	1x2	5.308.416	5.308.422	7
8x1	1x4	3.211.264	4.210.794	4
8x1	1x8	2.162.688	4.210.890	4
16x1	1x4	2.686.976	4.210.898	4
16x1	1x8	1.638.400	4.211.090	4
16x1	1x16	1.114.112	4.211.474	3

Table 2. Matrix with size 256x256 and z=1.

Block A	Block B	Transfers	Cycles	Time
4x1	1x4	269.484.032	269.484.038	Hours
4x1	**1x8**	**202.375.168**	**268.697.702**	**Hours**
8x1	1x4	202.375.168	268.697.706	Hours
8x1	1x8	135.266.304	268.697.802	Hours

Table 3. Matrix with size 1024x1024 and z=1.

The number of cycles is obtained by considering that one cycle is required to write to the FIFO and that the multiplication-addition unit has a throughput of one result per cycle. The clock cycles are described in the processes and expressed by the function *wait()*.

Observing tables 1, 2 and 3, we can conclude that the number of transfers and the number of clock cycles are consistent with the equations (2) and (3), respectively, as the difference between the simulated values and the theoretical approximation is less than 1%.

As expected, the number of transfer cycles required decreases with the size of the C blocks

($x \times y$). The minimal number of cycles is achieved when all the communications, except the initial overhead, can be totally overlapped with the computations. These are shown as best results and marked in bold.

The (to be selected) size of the C block should be sufficient for the computation cycles to dominate, that is, for the transfers to be able to fully overlap with the computations, but not larger. As long as there is full overlap, there is no need to further increase the size of the C blocks (and therefore the local memory of the processors). In fact and as shown, above the ideal values the total number of clock cycles slightly increases with x and y. This is because the initial and final overheads are proportional to the sizes of the matrix blocks.

Block A	Block B	Transfers	Cycles	Time (s)
4x2	2x4	540.672	540.682	48
4x2	2x8	409.600	528.490	30
4x2	2x16	344.064	528.586	32
8x2	2x4	409.600	528.498	32
16x2	2x4	344.064	528.610	30
4x8	8x8	409.600	528.546	33
4x16	16x8	409.600	528.673	34
4x64	64x8	409.600	529.441	37

Table 4. Matrix with size 128x128.

Block A	Block B	Transfers	Cycles	Time (min)
4x2	2x4	4.259.840	4.259.850	6
4x2	2x8	3.211.264	4.210.794	4
4x2	2x16	2.686.976	4.210.890	4
8x2	2x4	3.211.264	4.210.802	4
4x32	32x8	3.211.264	4.211.233	4
4x64	64x8	3.211.264	4.211.745	4
4x128	128x8	3.211.264	4.212.769	5

Table 5. Matrix with size 256x256.

Block A	Block B	Transfers	Cycles	Time (min)
4x16	16x8	202.375.168	268.69.889	Hours
4x128	128x8	202.375.168	268.69.681	Hours
4x256	256x8	202.375.168	268.70.729	Hours

Table 6. Matrix with size 1024x1024.

Tables 4, 5 and 6 show the results with regard to the number of data transfers and cycles and the execution time, considering different values of z. These results confirm the theoretical conclusion, and equation (2), that the number of transfers does not change when the value z is increased. Again and as expected, the number of clock cycles slightly increases with z, because the overheads depend on the size of the blocks used.

6. Conclusions

In this paper we have described an approach to simulate many-core architectures using SystemC. We started with a brief overview of the language, describing some of the important concepts needed to create a model of a system.

A parallel matrix multiplication algorithm to execute on a 2-dimensional multiprocessor array was presented and analyzed theoretically.

An architecture was developed and implemented in SystemC in order to model the multiprocessor system design.

We simulated the model to evaluate number of transfers and number of clock cycles required for the complete algorithm execution. The simulated results fully confirmed the theoretical analysis (the differences are less than 1%).

The proposed SystemC model is now being generalized to be able to simulate any number of processors, so that massively parallel architectures may be evaluated.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2011 and PTDC/EEA-ELC/122098/2010.

References

[1] M. Monchiero, J. Ho Ahn, A. Falcon, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. Technical Report HPL-2008-190, Hewlett Packard Laboratories, November 9, 2008.

[2] J. Chen, M. Annavaram, and M. Dubois, "Exploiting simulation slack to improve parallel simulation speed", in International Conference on Parallel Processing, pp.371–378, 2009.

[3] M. Hosseinabady, J.L. Nunez-Yanez, "Effective modelling of large NoCs using SystemC,", Proceedings of IEEE International Symposium on Circuits and Systems, pp.161-164, 2010.

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

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

[6] J. Bhasker, "A SystemC Primer", Star Galaxy Publishing, 2002.

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

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

Projecto de uma Arquitectura Massivamente Paralela para a Multiplicação de Matrizes

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

Resumo

A densidade de transístores tornou possível o projecto de arquitecturas massivamente paralelas com centenas de processadores num único integrado. No entanto, o projecto de arquitecturas com um número tão elevado de processadores com um eficiente rácio desempenho/área ou desempenho/energia é um grande desafio. Neste artigo, adoptámos uma abordagem diferente ao projecto de uma arquitectura de muitos núcleos. Inicialmente, efectuamos uma análise formal aos algoritmos considerando aspectos arquitecturais, e só a seguir é tomada uma decisão relativa à estrutura da arquitectura de muitos núcleos. O algoritmo de multiplicação de matrizes densas é utilizado como ponto de partida. No trabalho descrito, implementámos a arquitectura resultante da análise do modelo teórico do algoritmo de multiplicação de matrizes e simulámos o sistema em SystemC para confirmar os resultados. Os resultados indicam que a arquitectura de muitos núcleos/algoritmo propostos para a multiplicação de matrizes conseguem um desempenho de 527 GFLOP/s em precisão simples e 192 GFLOP/s com precisão dupla.

Palavras Chave—Multiplicação de Matrizes, Massivamente Paralelo, Alto-desempenho, FPGA

1. Introdução

Durante a última década têm sido propostos diversos sistemas massivamente paralelos como arquitecturas de alto desempenho com elevada capacidade de computação, o que os torna particularmente atractivos para aplicações científicas.

Como representantes de chips comerciais com grande capacidade de processamento tem-se o IBM Cell com nove elementos de processamento SIMD com capacidade para executar operações de vírgula-flutuante de 32-bits a 3 GHz [1], o processador 80-tile teraflops da Intel com capacidade para operações de vírgula-flutuante de 32-bits organizado como uma malha 2D operando a frequências até 5 GHz [2], o processador de vírgula-flutuante CSX700 [3] que se destina a computações científicas e incorpora estruturas direccionadas à computação algébrica e as mais recentes arquitecturas GPU (Graphical Processing Unit) de uso-geral com um grande número de processadores SIMD num único chip.

Os chips referidos atingem desempenhos de pico de aproximadamente um TeraFlop para precisão simples. No entanto, uma análise mais detalhada destes processadores revela que, ao executarem algoritmos específicos, o desempenho sustentado está tipicamente longe do desempenho de pico disponível. Em particular, considerando a multiplicação de matrizes densas, o processador Teraflop atinge apenas cerca de 40% do seu desempenho de pico, enquanto os GPGPUs atingem cerca de 60%. Já o acelerador CSX700 atinge um melhor desempenho, (quase) 80% do seu desempenho de pico, o qual foi especificamente projectado para operações de computação científica. Finalmente, o IBM Cell atinge quase o seu desempenho de pico.

Outras aplicações revelam piores resultados no que diz respeito ao desempenho. Por exemplo, o Teraflop apenas atinge uns escassos 2,73% de desempenho de pico quando executa a FFT 2D. Se analisarmos o rácio desempenho/área surgem ineficiências relativas piores. O processador 80-tile chega até aos 2.6 GFLOPs/mm^2, um GPU atinge 0.9 GFLOPs/mm^2, o CSX700 atinge apenas 0.2 GFLOPs/mm^2, e o processador IBM cell atinge os 2 GFLOPs/mm^2. Estas medidas de desempenho/área têm influência directa no custo e no consumo de energia do chip do processador.

Em vez de propor outra arquitectura de muitos núcleos e aplicar um conjunto específico de algoritmos para determinar o seu desempenho, a nossa abordagem consiste em efectuar uma análise formal dos algoritmos considerando os aspectos arquitecturais, como o número de processadores, a memória local disponível para cada processador, e a largura de banda entre o processador e a memória externa, e só depois decidir a estrutura da arquitectura do processador de muitos núcleos.

Neste trabalho seguiu-se essa abordagem para projectar um sistema de processamento massivamente paralelo para computação científica, cujo projecto é orientado pelos próprios algoritmos, em particular por uma multiplicação de matrizes (*GEneral Matrix-matrix Multiplication* - GEMM) [4]. Começou-se por analisar o algoritmo e em seguida usou-se os resultados dessa análise para orientar o projecto do processador de muitos núcleos. Um modelo ao nível do sistema da arquitectura foi implementado e simulado em SystemC [5] para confirmar os resultados teóricos obtidos.

O uso do SystemC facilitou a modelação ao nível do sistema da arquitectura hardware-software e permitiu obter resultados precisos em termos do número de ciclos de execução que confirmaram a análise teórica do algoritmo.

O artigo encontra-se organizado da seguinte forma. Na secção 2 são descritos outros trabalhos envolvendo a multiplicação de matrizes densas. Na secção 3 é descrito o algoritmo paralelo de multiplicação de matrizes. A secção 4 descreve o processador massivamente paralelo de alto-desempenho proposto. A secção 5 relata os resultados respeitantes à simulação e implementação. Por fim, o artigo é concluído na secção 6.

2. Estado de Arte

Existem diversos processadores comerciais de muitos núcleos para computação de alto-desempenho, incluindo os GPUs, o IBM Cell, o processador 80-tile teraflops da Intel e o processador CSX700. Todos estes processadores foram testados com a multiplicação de matrizes atingindo altos desempenhos com diferentes eficiências em termos de desempenho/área. Processadores de uso-geral foram também sujeitos a um estudo intensivo relativamente a computações matriciais de alto-desempenho [6]. No entanto, o *overhead* resultante da descodificação de instruções, entre outras complexidades, degrada o desempenho relativo da arquitectura.

Em [7] é apresentada uma análise extensa à multiplicação de matrizes. Os autores apresentam um modelo teórico com o objectivo de estudar a GEMM em várias arquitecturas. O modelo permite conhecer os diferentes compromissos resultantes da personalização dos parâmetros inerentes às arquitecturas, e indica como atingir o máximo desempenho com a máxima eficiência em termos de área e energia. Os autores propõem o seu próprio processador baseado numa malha 2D com 16 núcleos de processamento. Estes estimam que uma arquitectura de 240 núcleos consiga atingir 600 GFLOPS ao executar uma aplicação GEMM de dupla precisão na tecnologia padrão de 45nm, mostrando melhor desempenho e eficiência a nível de área e energia face a outros trabalhos (e.g. [2]).

Arquitecturas *hardware* dedicadas foram também exploradas nas FPGAs (Field Programmable Gate Arrays). Em [8] os autores apresentam um projecto de um multiplicador de matrizes em hardware de vírgula-flutuante com dupla precisão optimizado para implementação em FPGA. Neste trabalho são propostos modelos teóricos para estimar a dimensão dos sub blocos matriciais que minimize o número de comunicações com a memória externa, e para avaliar a largura de banda de pico necessária para atingir o máximo desempenho dado o número de processadores e a dimensão da memória local. Os resultados mostram que um sistema com 39 elementos de processamento utilizando um total de 1600 KB de memória interna e executando a 200 MHz (numa Virtex II Pro) consegue alcançar um desempenho de 15.6 GFLOPS, dada uma largura de banda de 400 MB/s para acesso à memória externa.

Em [9] é proposta uma implementação de diversas operações matriciais optimizada para FPGA. Os autores apresentam uma análise teórica baseada em parâmetros de projecto existentes. No trabalho, afirmam que com unidades de vírgula-flutuante mais pequenas e mais rápidas conseguem atingir 19.5 GFLOPS para a multiplicação de matrizes.

Em [10], é proposto um modelo teórico para prever o desempenho da multiplicação de matrizes esparsas e densas em sistemas baseados em FPGAs. O modelo apresentado é baseado na multiplicação de matrizes por blocos e conclui-se que os blocos ideias são quadrados o que, segundo os autores, é também verificado em trabalhos anteriores.

Tal como em alguns trabalhos referidos em FPGAs, o trabalho que apresentamos propõe também um algoritmo e uma análise teórica que guia e sustenta as opções de projecto de uma arquitectura de alto desempenho, detalhada nas secções seguintes.

3. Algoritmo Paralelo de Multiplicação de Matrizes Densas

Nesta secção, é proposto um novo algoritmo para paralelizar a multiplicação de matrizes dado um sistema com p elementos de processamento (PEs) organizado como uma matriz 2D. De forma a facilitar a exposição do algoritmo consideramos que a malha 2D de processadores é quadrada. Os resultados podem ser facilmente generalizados para malhas não quadradas.

A matriz C é definida como o resultado entre o produto de duas matrizes, A e B. O algoritmo proposto está representado graficamente na figura 1. As matrizes são também consideradas quadradas e apresentam as mesmas dimensões $n \times n$, sendo que as suas dimensões são múltiplas das dimensões dos sub-blocos. Note-se uma vez mais que esta consideração não limita de qualquer forma a generalidade dos resultados.

Como demonstrado, cada um dos $p = q \times q$ processadores, onde $q = \sqrt{p}$, é responsável por calcular um bloco da matriz C de dimensão $\frac{n}{q} \times \frac{n}{q}$. Cada um destes blocos é por sua vez repartido, de acordo com as limitações de memória do processador, em sub blocos C_{ij} de dimensão $y \times x$.

Para gerar um bloco C_{ij}, o processador deve multiplicar um bloco $y \times n$ da matrix A por um bloco $n \times x$ da matriz B. A multiplicação é implementada como uma sequência de $k_0 = \frac{n}{z}$ multiplicações parciais de blocos como

$$C_{ij} = \sum_{k=1}^{k_0} A_{ik} \times B_{kj} \tag{1}$$

Cada multiplicação parcial de blocos consiste na multiplicação de um sub bloco A_{ik} de dimensão $y \times z$ com um sub bloco B_{kj} de dimensão $z \times x$, resultando num sub bloco parcial C_{ij_k} de dimensão $y \times x$. O resultado final C_{ij} é obtido depois de acumular k_0 resultados parciais.

As multiplicações parciais de blocos são implementadas tal que, primeiro, cada processador recebe um sub-bloco A_{ik} e armazena-o. Em seguida, recebe os elementos do sub-bloco B_{kj} que são imediatamente multiplicados pelos correspondentes elementos de A_{ik} armazenados localmente, de forma a produzir os elementos resultantes do bloco parcial C_{ij_k}.

A memória local de cada PE deve conseguir armazenar um bloco A_{ik} e um bloco C_{ij}. Os elementos de B_{kj} são

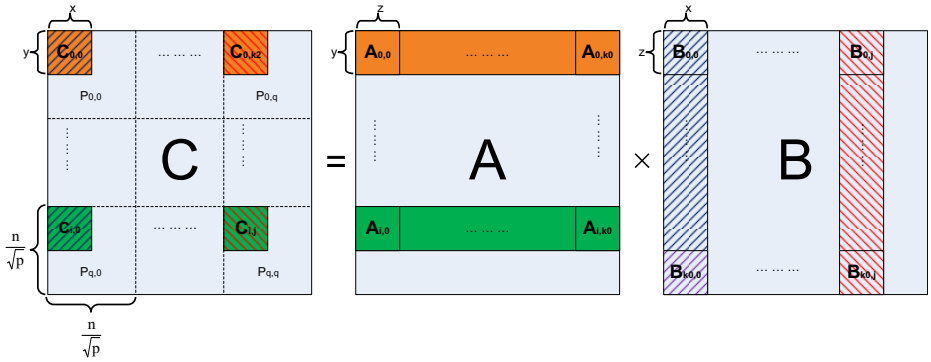


Figura 1. Algoritmo de multiplicação de matrizes por blocos.

processados assim que estão disponíveis e portanto não necessitam de ser armazenados localmente no PE.

Olhando para a figura 1, reparamos que todos os processadores na mesma linha requerem os mesmos dados da matriz A, enquanto que todos os processadores na mesma coluna requerem os mesmos dados da matriz B. Portanto, cada sub-bloco transferido da memória é difundido por uma linha (coluna) de $\sqrt{p}$ processadores.

O número total de comunicações da/para a memória externa é dado por

$$N_{comm} = \frac{n^3}{\sqrt{p}}\left(\frac{1}{x}+\frac{1}{y}\right)+n^2 \tag{2}$$

em que o primeiro termo corresponde à (repetida) leitura de elementos de A e B e o segundo termo, n^2, corresponde a escrever de volta os elementos finais de C.

O número de comunicações não depende da dimensão z dos sub blocos da matriz A e matriz B, portanto podemos simplesmente igualar a 1 de forma a minimizar a memória local necessária.

A memória necessária para armazenar um sub bloco A_{ik} (dimensão $y \times 1$) é duplicada para permitir que o processador armazene a próxima coluna necessária enquanto realiza as computações.

Deste modo, a memória local necessária para cada PE é:

$$L = 2y + xy \tag{3}$$

A partir das equações (2) e (3) determinámos as dimensões dos sub-blocos C_{ij} que minimizam o número de comunicações, como função da memória local disponível L:

$$x = \sqrt{L}\,;\; y = \frac{L}{2+\sqrt{L}} \approx \sqrt{L} \tag{4}$$

O número total de ciclos de computação, N_{PC}, supondo uma acumulação por ciclo, é dado por

$$N_{PC} = \frac{n^3}{p} \tag{5}$$

O número total de ciclos de comunicação, N_{CC}, dada uma largura de banda de b palavras por ciclo, é

$$N_{CC} = \frac{N_{comm}}{b} \approx \frac{n^3}{\sqrt{p}}\left(\frac{1}{x}+\frac{1}{y}\right)\frac{1}{b} \tag{6}$$

considerando que, para matrizes grandes, o termo n^2 é desprezável (face a n^3).

O tempo de execução mínimo é alcançado quando todas as comunicações, excepto o *overhead* inicial, conseguem ser totalmente sobrepostas às computações. A condição seguinte deve ser respeitada de forma a atingir a sobreposiçao máxima:

$$N_{CC} \leq N_{PC} \tag{7}$$

$$\frac{n^3}{\sqrt{p}}\left(\frac{1}{x}+\frac{1}{y}\right)\frac{1}{b} \leq \frac{n^3}{p} \tag{8}$$

$$\left(\frac{1}{x}+\frac{1}{y}\right) \leq \frac{b}{\sqrt{p}} \tag{9}$$

Para blocos quadrados C_{ij}, tais que $x = y = \sqrt{L}$ como indicado na equação (4), a largura de banda mínima necessária para suportar um dado conjunto de p processadores, com uma memória local de L palavras cada, é dada por

$$b_{min} = 2\sqrt{\frac{p}{L}} \tag{10}$$

O tempo total de execução, t_{exec}, depende do factor limitativo, computações ou comunicações, tendo em conta as restrições do sistema (número de processadores, largura de banda, tamanho da memória, frequência). Este pode ser estimado por

$$t_{exec} \approx max\left[\frac{n^3}{p}\,;\,\frac{2n^3}{b\sqrt{pL}}\right] \tag{11}$$

não considerando o (negligenciável para matrizes grandes) custo inicial das comunicações.

4. Arquitectura Massivamente Paralela de Alto-Desempenho

A arquitectura paralela é organizada como uma malha 2D de elementos de processamento (ver figura 2). A execução dos processadores é ditada pelo fluxo de dados, isto é, começam a executar assim que os dados de entrada necessários estão disponíveis.

Nesta versão da arquitectura paralela estamos a usar núcleos de processamento homogéneos. Cada unidade

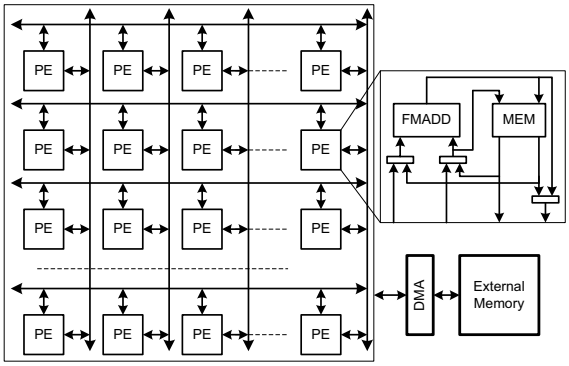


Figura 2. Arquitectura massivamente paralela de alto-desempenho.

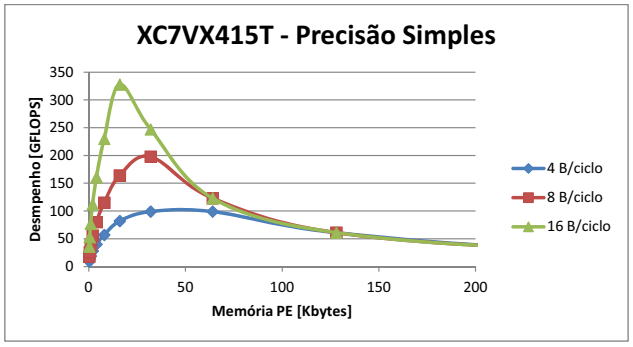


Figura 3. Desempenho (GFLOPs) da arquitectura paralela considerando precisão simples e FPGA XC7VX415T

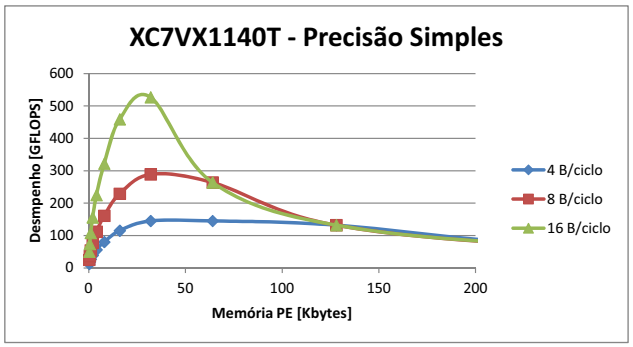


Figura 4. Desempenho (GFLOPs) da arquitectura paralela considerando precisão simples e FPGA XC7VX1140T

de processamento consiste basicamente numa unidade de vírgula-flutuante de multiplicação-soma (FMADD) e uma memória de duplo porto. A cada ciclo de relógio, a FMADD é capaz de fornecer um resultado multiplicação-adição sustentado (2 FLOPs). O acesso à memória externa é controlado por um módulo de acesso directo à memória (DMA) que consegue providenciar transacções de rajada com um ritmo igual a uma transferência por ciclo. A arquitectura suporta difusão de dados na horizontal e na vertical.

Um modelo a nível do sistema da arquitectura foi desenvolvido em SystemC para avaliar a implementação proposta do algoritmo de multiplicação de matrizes. As comunicações entre a memória e os elementos de processamento são modeladas por FIFOs (do tipo sc_fifo). O módulo *ProcessorArray* consiste em p processos *thread* (sc_thread), um para cada. Cada processo *thread* recebe os blocos matriciais e executa as operações. A multiplicação de matrizes por blocos é iniciada no momento em que o primeiro valor do bloco B é lido (ver secção 3). O produto entre dois blocos corresponde aos resultados parciais do bloco C, os quais são armazenados na memória do processador. Depois de obter os resultados finais, estes são enviados para a memória da matriz, também através de uma FIFO.

Este modelo da arquitectura foi simulado para matrizes de diferentes dimensões e sub-matrizes com blocos de diferentes dimensões. Os resultados obtidos referentes ao número de transferências de dados e ao número total de ciclos de relógio foram comparados com a análise teórica (resumida na secção 3) e confirmam em pleno a sua validade.

5. Resultados de Implementação

Um protótipo do núcleo de processamento foi projectado e sintetizado para FPGAs da família Virtex-7. A unidade de vírgula-flutuante de dupla precisão foi implementada com base nas bibliotecas da Xilinx. O módulo FMADD foi implementado de modo a suportar uma frequência de operação de 500 MHz para precisão simples e 400 MHz para precisão dupla. Assim, ao ritmo de produção máximo, o módulo atinge um desempenho de pico de 800 MFLOP/s (quando realiza as operações de multiplicação-soma) para precisão dupla e 1 GFLOP/s para precisão simples. O bloco de DMA funciona a 200 MHz. A tabela 1 apresenta os resultados de implementação do núcleo.

Tabela 1. Resultados de implementação

Precisão	Módulo	LUTs	DSP
Simples	FMADD	750	5
Dupla	FMADD	1700	14

Considerando esta implementação, determinou-se a relação entre largura de banda com a memória externa, memória local e desempenho para dois dispositivos da família Virtex-7 considerando precisão simples (ver resultados nas figuras 3, 4) e precisão dupla (ver figuras 5 e 6).

Considerando precisão simples, através dos gráficos observamos os pontos de desempenho máximo. As curvas podem ser explicadas considerando a relação entre computação e comunicação. Para uma determinada largura de banda, o aumento da memória local reduz o número de ciclos de comunicação, mas o número de processadores também reduz e consequentemente o número de ciclos de computação aumenta. Assim, para memórias locais mais pequenas, o tempo de execução é dominado pela comunicação, enquanto que para memórias locais maio-

 ISBN: 978-972-8822-27-9

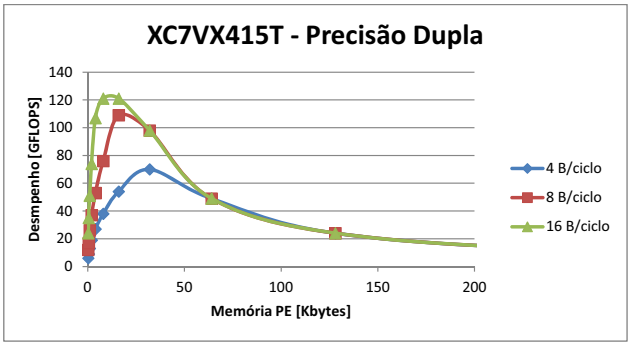


Figura 5. Desempenho (GFLOPs) da arquitectura paralela considerando precisão dupla e FPGA XC7VX415T

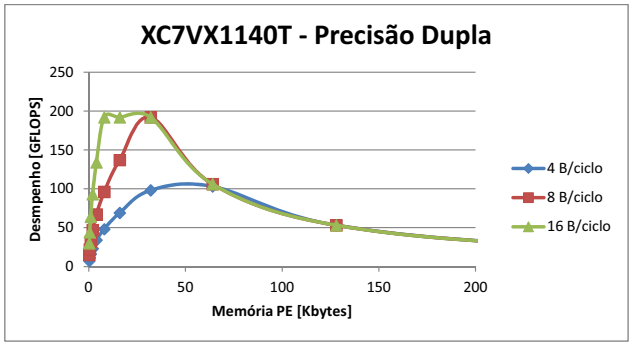


Figura 6. Desempenho (GFLOPs) da arquitectura paralela considerando precisão dupla e FPGA XC7VX1140T

res o tempo de execução é dominado pela computação. Isto significa que existe um tempo de execução óptimo em que a comunicação fica escondida pela computação que corresponde ao ponto máximo que se pode visualizar nos gráficos.

Se aumentarmos a largura de banda, o número de ciclos de comunicação decresce e em consequência o ponto óptimo é atingido para uma memória local menor. Esta alteração também aumenta o desempenho, uma vez que o ponto óptimo tem um menor número de ciclos de comunicação e de ciclos de computação (ver tabela 2).

Tabela 2. Resultados para precisão simples e matrizes de 1024×1024

XC7VX415T					
BW	Mem	#cores	Ciclos	Exec(μs)	GFLOPs
4B/ciclo	32 KB	247	4.351.287	21,71	99
8B/ciclo	32 KB	247	4.349.210	10,85	198
16B/ciclo	16 KB	343	3.131.300	6,55	328
XC7VX1140T					
BW	Mem	#cores	Ciclos	Exec(μs)	GFLOPs
4B/ciclo	32 KB	528	2.972.185	14,86	145
8B/ciclo	32 KB	528	2.036.609	7,43	289
16B/ciclo	32 KB	528	1.035.106	4,08	527

Na tabela observamos o aumento do desempenho com o aumento da largura de banda. Note-se, no entanto, que o aumento do número de núcleos e de ciclos não é proporcional ao aumento da largura e ao aumento do desempenho. O que se passa, é que o número de ciclos corresponde ao máximo entre o número de ciclos de comunicação e o de computação. Por exemplo, na segunda linha da tabela, o número de ciclos é dominado pela computação, pelo que o tempo de execução reduz para metade, comparado com o caso da primeira linha (não esquecer que frequência de computação é o dobro da de comunicação).

A análise relativa à arquitectura com núcleos de dupla precisão é similar à realizada para precisão simples. Como era de esperar existe uma redução do desempenho uma vez que temos de comunicar com palavras de 64 bits e o módulo de cálculo funciona a uma frequência menor (ver tabela 3).

Tabela 3. Resultados para precisão dupla e matrizes de 1024×1024

XC7VX415T					
BW	Mem	#cores	Ciclos	Exec(μs)	GFLOPs
4B/ciclo	32 KB	123	8.732.578	30,74	70
8B/ciclo	16 KB	151	7.112.025	19,74	109
16B/ciclo	8 KB	151	7.111.283	17,78	121
XC7VX1140T					
BW	Mem	#cores	Ciclos	Exec(μs)	GFLOPs
4B/ciclo	64 KB	132	8.138.775	20,90	103
8B/ciclo	32 KB	240	4.475.972	11,20	192
16B/ciclo	8 KB	240	4.474.437	11,19	192

Comparámos a nossa arquitectura com o resultado de [8], tendo em conta que a família de FPGA é diferente (Virtex II Pro). Ajustámos a nossa arquitectura para as mesmas frequências e largura de banda (ver tabela 4).

Tabela 4. Comparação com o estado da arte em FPGA

Arquitectura	PEs	Freq.	BW (MB/s)	GFLOPS/s
VirtexIIP [8]	39	200 MHz	400	15,6
Nosso (Equiv.)	39	200 MHz	400	15,6
Nosso (415T)	89	200 MHz	400	35,1
Nosso (1140T)	130	200 MHz	400	51,4

No caso de um dispositivo de complexidade equivalente ao de [8] (39 processadores e 1600 KB de memória interna) verifica-se que as duas arquitecturas têm um desempenho equivalente. Verifica-se ainda que, para os dispositivos de nova geração, é possível aumentar significativamente o número de processadores e, consequentemente, o desempenho.

6. Conclusões e Trabalho Futuro

O artigo descreve uma abordagem algorítmica no projecto de arquitecturas massivamente paralelas. Analisámos os compromissos entre a largura de banda referente à memória externa, dimensão de memória local e número

de processadores da arquitectura. A abordagem foi testada com o algoritmo de multiplicação de matrizes. Os resultados indicam que um projecto optimizado da arquitectura tem grandes benefícios de desempenho.

No futuro planeamos considerar mais algoritmos (*level-3 BLAS*), assim como outras operações importantes para computação científica (e.g., *Fast Fourier Transform*), de forma a generalizar a arquitectura proposta e ver como esta influencia o desempenho da arquitectura na execução da multiplicação de matrizes e o quão eficiente pode ser o projecto da arquitectura de forma a suportar estes algoritmos e operações.

Agradecimentos

Este trabalho foi financiado com fundos nacionais através da FCT, Fundação para a Ciência e Tecnologia, pelos projectos PEst-OE/EEI/LA0021/2011 e PTDC/EEA-ELC/122098/2010.

Referências

[1] Hofstee, H. P. (2005b). Power efficient processor architecture and the cell processor. In HPCA 05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, Washington, DC, USA. IEEE Computer Society.

[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S Borkar, "An 80-tile sub-100w teraflops Processor in 65-nm CMOS", IEEE Journal of Solid-State Circuits, 43(1):29-41, 2008.

[3] CSX700 Floating Point Processor. Datasheet 06-PD-1425 Rev 1, ClearSpeed Technology Ltd, 2011.

[4] K. Goto and R. Geijn, "Anatomy of a High-Performance Matrix Multiplication", ACM Transactions Math. Soft., 34(3):12, May 2008.

[5] Thorsten Grötker, Stan Liao, Grant Martin and Stuart Swan, "System Design with SystemC", Kluwer Academic Publishers, 2002.

[6] K. Goto and R. Geijn, "High Performance Implementation of the level-3 BLAS", ACM Transactions Math. Soft., 35(1):1-14, May 2008.

[7] A. Pedram, R. van de Geijn, A. Gerstlauer, "Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures,"in IEEE Transactions on Computers, vol.61, no.12, pp.1724-1736, Dec. 2012.

[8] Y. Dou, S. Vassiliadis, G. Kuzmanov, G. Gaydadjiev, "64-bit Floating-Point FPGA Matrix Multiplication", in ACM/SIGMA 13th International Symposium on Field-Programmable Gate Arrays, 2005, pp. 86-95.

[9] L. Zhuo and Viktor K. Prasanna, "High-Performance Designs for Linear Algebra Operations on Reconfigurable Hardware", in IEEE Transactions on Computers, 57(8):1057-1071, Aug. 2008.

[10] C. Lin, H. So, and P. Leong, "A model for matrix multiplication performance on FPGAs", in International Conference on Field Programmable Logic and Applications, 2011, pp.305-310.

[11] David C. Black and Jack Donovan, "SystemC: from the Ground Up", Kluwer Academic Publishers, 2004.

Arquiteturas para processamento de alto débito

Hardware Accelerator for Biological Sequence Alignment using Coreworks® Processing Engine

José Cabrita, Gilberto Rodrigues, Paulo Flores
INESC-ID / IST, Technical University of Lisbon
jpmcabrita@gmail.com, gilberto.rodrigues@ist.utl.pt, paulo.flores@inesc-id.pt

Abstract

Several algorithms exist for biological sequence alignment. The Smith-Waterman (S-W) algorithm is an exact algorithm that uses dynamic programming for local sequence alignment. Some implementations in software for General Purpose Processors (GPP) as well in hardware (using Field Programmable Gate Array (FPGA)) exist. In this paper it is proposed an implementation of the S-W algorithm for DNA, RNA and amino acids sequence alignment that uses the Coreworks® processing engine. The processor FireWorks™ will be used to control a hardware accelerator named SideWorks™ both developed by Coreworks®. In this paper is proposed an architecture based on Process Elements (PE) to be implemented in SideWorks™ accelerator template with the purpose of accelerating the S-W algorithm.
The developed application is able to read sequences from a file, align them with a library of sequences and present the results for the best local alignments using the Coreworks® processing engine.

Keywords— *DNA, Bioinformatics, Sequence Alignment, Smith-Waterman algorithm, Field Programmable Gate Array (FPGA), Cell Updates Per Second (CUPS), Platform Design, SideWorks™, FireWorks™*

1. Introduction

Sequence alignment is one of the most widely used operations in computational biology. The need for speeding up this operation comes from the exponential growth of biological sequences databases.

The sequence alignment operation consists of finding similarities between a certain test sequence and all the sequences of a database. This operation allows biologists to point out sequences sharing common subsequences. From a biological point of view, this operation leads to identifying similar functionality.

The S-W algorithm is a well-known dynamic programming algorithm for performing local sequence alignment to determine similar regions between two DNA, RNA, proteins or amino acids sequences.

There are two stages in the S-W algorithm. These are the similarity matrix (H matrix) fill and the trace back. In the first stage a matrix is filled with a similarity score for each element of the sequences. The second stage finds the maximum score of the matrix and performs the trace back to find the best local alignment. The first stage of the algorithm will consume the largest part of the total computation time.

One approach used to get high quality results in a short processing time is to use parallel processing on a reconfigurable system (FPGA) to accelerate the H matrix fill stage of the S-W algorithm. The maximum score of the matrix is then transferred to a GPP and the trace back is performed to get the optimal alignment.

2. Smith-Waterman algorithm

The Smith-Waterman algorithm is an optimal local sequence alignment algorithm that uses dynamic programming. Several alignment models can be used by the S-W algorithm. A simple model of the algorithm is the *Linear Gap Penalty* (LGP) model. In this model there is a score penalty (α) for a gap in the alignment of the sequences, the value of the score penalty is linear and defined by the user of the algorithm.

The algorithm uses a substitution matrix (Sbt matrix) that represents the similarity between elements. The matrix positions have a value of -1 if the elements are different and 2 if the elements are equal.

Using two sequences of size N and M the H matrix can be computed using the following expression:

$$H(i,j) = max\begin{cases} 0 \\ H(i-1,j) - \alpha \\ H(i,j-1) - \alpha \\ H(i-1,j-1) + Sbt(S1i, S2j) \end{cases}, \quad (1)$$

for $1 \le i \le N$, $1 \le j \le M$.

$$H(i,0) = H(0,j) = 0 \quad \text{for } 0 \le i \le N,\ 0 \le j \le M,$$

where *i* and *j* represent the element position of the sequences under evaluation. More information on S-W algorithm can be found in [1][2].

The regular computation requires an initialization of the first column and the first line filled with zero value, as presented in Fig. 1, where each cell is computed with equation (1).

Sequence 1: ATGCTGAC
Sequence 2: CGATCGAT

		A	T	G	C	T	G	A	C
	0	0	0	0	0	0	0	0	0
C	0	0	0	0	2	1	0	0	2
G	0	0	0	2	1	1	3	2	1
A	0	2	1	1	1	0	2	5	4
T	0	1	4	3	2	3	2	4	4
C	0	0	3	3	5	4	3	3	6
G	0	0	2	5	4	4	6	5	5
A	0	2	1	4	4	3	5	8	7
T	0	1	4	3	3	6	5	7	7

Fig. 1 – H matrix for sequence 1 and sequence 2.

Since the biological sequences to be aligned may be too long to be processed in fully paralleled hardware the proposed architecture will be adapted to include the possibility to divide the computation of the H matrix. This division uses the initialization values of the matrix as is show on the following example.

When splitting the computation of the matrix using, for example 4 partitions, the regular computation is repeated 4 times as presented in Fig. 2.

1

		A	T	G	C
	0	0	0	0	0
C	0	0	0	0	2
G	0	0	0	2	1
A	0	2	1	1	1
T	0	1	4	3	2

3

		T	G	A	C
	0	0	0	0	0
C	2	1	0	0	2
G	1	1	3	2	1
A	1	0	2	5	4
T	2	3	2	4	4

2

		A	T	G	C
	0	1	4	3	2
C	0	0	3	3	5
G	0	0	2	5	4
A	0	2	1	4	4
T	0	1	4	3	3

4

		T	G	A	C
	2	3	2	4	4
C	5	4	3	3	6
G	4	4	6	5	5
A	4	3	5	8	7
T	3	6	5	7	7

Fig. 2 – Divided computation of H matrix for sequence 1 and sequence 2.

Each computation inherits the line and column of previous computations as its own initialization line and column. Using this implementation is possible to obtain the exact same score result of H matrix. More information on H matrix partition computation can be found in [3].

For this application it will be used a simple trace back function [4]. This function finds the maximum score position in the H matrix and recalculates expression (1) for that position, this time evaluating from which cell the result derivate from. As show in expression (1), each cell result can only come from 3 cells, the up neighbor cell, the left neighbor cell or the up-left neighbor cell. With this information the traceback function will then move to the cell that generated the result and perform again the same operation. This will continue until the score from the cell that generated the result is zero.

The example in Fig. 3 illustrates the trace back function working for sequence 1 and sequence 2.

		A	T	G	C	T	G	A	C
	0	0	0	0	0	0	0	0	0
C	0	0	0	0	2	1	0	0	2
G	0	0	0	2	1	1	3	2	1
A	0	2	1	1	1	0	2	5	4
T	0	1	4	3	2	3	2	4	4
C	0	0	3	3	5	4	3	3	6
G	0	0	2	5	4	4	6	5	5
A	0	2	1	4	4	3	5	8	7
T	0	1	4	3	3	6	5	7	7

Fig. 3 – Trace back for sequence 1 and sequence 2.

From the trace back in Fig. 3 results that the best local alignment with a score of 8 is:

ATGCTGA
AT- C -GA

To parallelize the H matrix fill in the S-W algorithm it is necessary to respect the data dependency. Through expression (1) is possible to realize that iteration (i,j) cannot be executed until iterations (i-1,j), (i,j-1) and (i-1,j-1) are executed first due to data dependencies. However if the elements are calculated on different time cycles it is possible to execute several calculus in the same time cycle as show in Fig. 4.

		A	T	G	C	T	G	A	C
	0	0	0	0	0	0	0	0	0
C	0	PE1, C1	PE2, C2	PE3, C3	PE4, C4	PE5, C5	PE6, C6	PE7, C7	PE8, C8
G	0	PE1, C2	PE2, C3	PE3, C4	PE4, C5	PE5, C6	PE6, C7	PE7, C8	PE8, C9
A	0	PE1, C3	PE2, C4	PE3, C5	PE4, C6	PE5, C7	PE6, C8	PE7, C9	PE8, C10
T	0	PE1, C4	PE2, C5	PE3, C6	PE4, C7	PE5, C8	PE6, C9	PE7, C10	PE8, C11
C	0	PE1, C5	PE2, C6	PE3, C7	PE4, C8	PE5, C9	PE6, C10	PE7, C11	PE8, C12
G	0	PE1, C6	PE2, C7	PE3, C8	PE4, C9	PE5, C10	PE6, C11	PE7, C12	PE8, C13
A	0	PE1, C7	PE2, C8	PE3, C9	PE4, C10	PE5, C11	PE6, C12	PE7, C13	PE8, C14
T	0	PE1, C8	PE2, C9	PE3, C10	PE4, C11	PE5, C12	PE6, C13	PE7, C14	PE8, C15

Fig. 4 – H matrix example with indication on which PE and cycle the cell score is computed.

As is shown in Fig. 4 it is possible that all elements in the anti-diagonal can be computed in the same cycle (e.g. cycle 8). This parallel execution is called dataflow implementation [2], as all the computations are executed when their data dependencies are available.

This dataflow allows that the computation of the H matrix can be achieved using a chain of PEs. In each PE it will be computed a column of the H matrix and each cell computation will be streamed to the next PE.

3. Coreworks® Processing Engine

The objective of using Coreworks® processing engine is to accelerate, using hardware, the most compute intensive parts of the algorithm. In this case it will be the computation of the H matrix to determine the maximum score value of the alignment.

The Coreworks® processing engine has two major processing elements: the FireWorks™, a Harvard RISC (Reduced Instruction Set Computer) architecture 32-bit processor, and the SideWorks™, a reconfigurable hardware accelerator architecture.

The FireWorks™ is used to control the accelerator configurations and data transfer from/into the GPP and from/into the hardware accelerator.

The SideWorks™ is a reconfigurable architecture for hardware acceleration, which will also be implemented in the FPGA. This architecture uses Functional Units (FUs) to build datapaths as shown on the generic SideWorks™ template presented on Fig. 5. These FUs can be as simple as adders or registers to some more complex FUs. In this project one PE will be used as a FU. The reconfigurable possibilities of this accelerator allow more than one datapath to be defined in the FPGA. Therefore the user can select which datapath to use for each set of computations by control of FireWorks™. On this project only one datapath was be developed for the sequence alignment purpose.

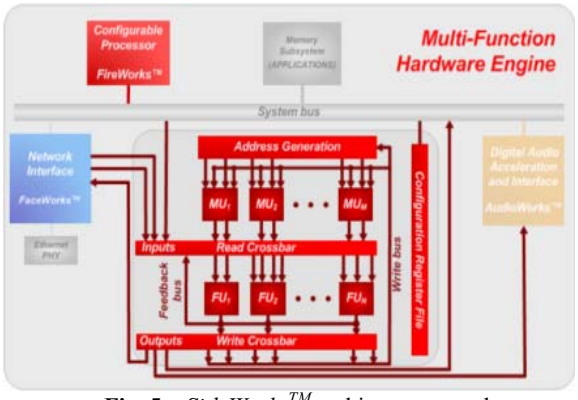


Fig. 5 – *SideWorks™* architecture template.

4. Application Overview

Our main application will run mostly on the GPP. The engine control and the hardware acceleration initialization will run on FireWorks™, but the H matrix computation will run on SideWorks™ hardware accelerator. The application runs according to the flowchart presented on Fig. 6.

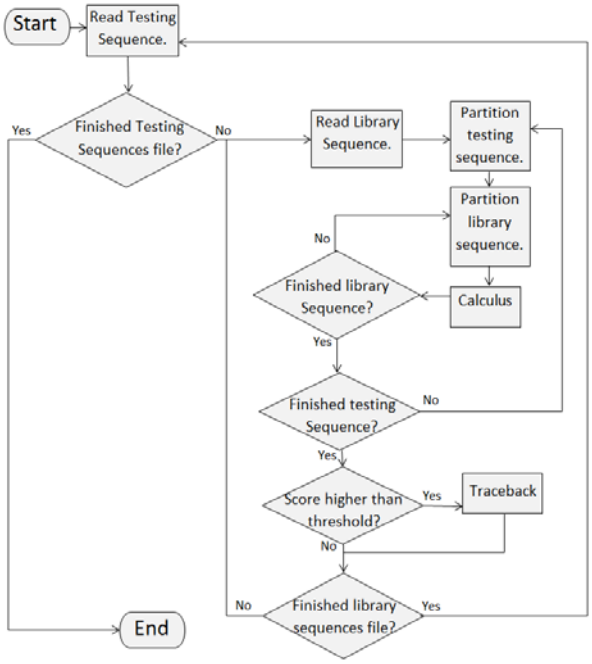


Fig 6 - Application flow chart.

The application begins by reading one sequence from the testing sequences file and one from the library sequences file. These sequences are then partitioned according to the limitations imposed that will be described in section 7. Each partition of the testing sequence will be compared to all the partitions of the library sequence before going on the next partition of the testing sequence. The computation part will end when all the partitions have ended. The result from the hardware accelerator will be the maximum score of the H matrix. As is show on Fig. 6, this value will be compared to a user defined threshold, and the trace back will only be executed if the score value is higher than the given threshold. Note that this threshold is defined in order to trace back only the sequences with high similarity values, because this increases the application efficiency.

Considering that the data transfer of the complete H matrix would take too long versus the processing time of the calculus, the traceback function rebuilds the H matrix until the computed score is equal to the score returned by the hardware and then starts the trace back itself. This option allows that, most of the times, the H matrix is not completely recalculated in software. Once the location of the maximum score is found a trace back is performed and the sequences local alignment is saved in a results file. Therefore the results file will have, for each comparison, the sequences being tested, the maximum score and the best local alignment.

The application ends after each of the sequences in the testing sequences file is compared to all the sequences in the library file.

5. Datapath Implementation on the SideWorks™

As mentioned before, the datapath for SideWorks™ is built using FUs. Two new FUs where developed for this application, the PE FU and the Trigger FU. All other FUs used were already developed and available on the Coreworks® development platform.

The PE FU is based on existing Process Elements [1][2][3], but modified to support partition computation and to be implemented on the SideWorks™ hardware accelerator. The PE FU will be described in more detail in section 6. The trigger FU is used to generate specific control signals used on the PEs. The Fig. 7 represents the datapath of our application for the SideWorks™.

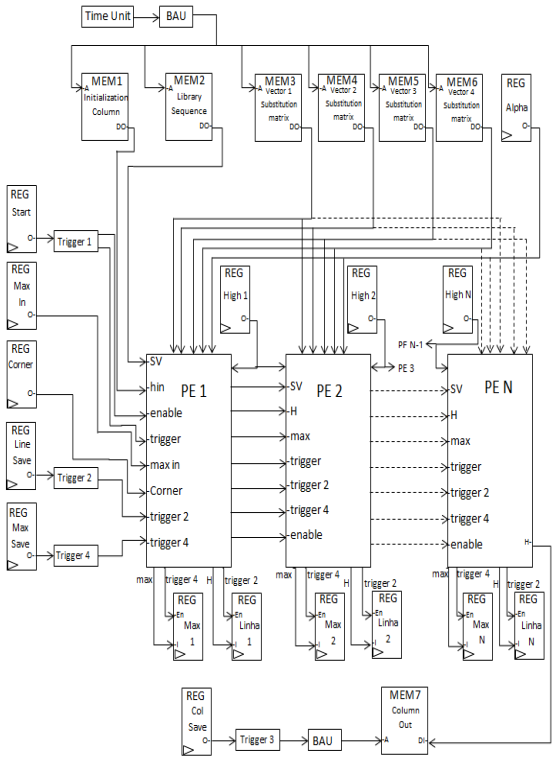


Fig 7 - SideWorks™ datapath.

The datapath is composed by a set of FUs being the most important the chain of PEs with the required FUs for input and output data. There are 6 input memory blocks. Four memory blocks are related to the data from the substitution matrix, this data is already coded according to the element from the testing sequence that is assigned to each PE. Another memory block contains the initialization column to be used only by PE1. Finally the last memory block contains the sequence from the library which will be streamed through the PEs. For each PE there is a register that contains the element from the initialization line and another register with the value of α, which can be configured by the user. For the first PE there are two additional input registers. One contains the initialization of the maximum value, in the first computation contains zero, on all other situations (e.g. computing a partition) contains the maximum score inherited by previous partitions. The other register contains the score of the top-left neighbour initialization value of a partition.

On the outputs there are two registers per PE and a memory block for the last PE. The registers will store the values of maximum score for that PE and the values from the last line that will be inherited by the next computation. The memory block will store the information for the values on the last column that will be inherited by the next computation.

The Trigger FUs are used to generate a trigger signal from an enable signal. The enable signal starts at zero and changes to one at a certain cycle, this signal is used to start the computation. At the clock cycle that enable changes to one, the trigger signal will also change to one and stays with this value only for one clock cycle. This signal is used to store some values internally in the PE. Both signals are propagated through the PEs chain.

6. Process Element Functional Unit

As previously mentioned, this FU is based on the PE described in [1], however some changes have been made to adapt the PE. Fig. 8 illustrates the resulting PE after the changes.

The main changes are related with the substitution matrix storing method and additional components to accommodate the computation of the H matrix using partitions.

There are five registers used to propagate signals to the next PE, of these, four are used for the enable and trigger signals and the remaining one is used to store and propagate the elements of the library sequence. As shown in Fig. 8 the output that results from the maximum computation and the output of H(i,j) will also be propagated to the next PE.

Each PE stores only the column of the substitution matrix related to the element of the testing sequence that has been assigned to. Therefore, each PE only needs to store 4 elements of the substitution matrix (for DNA) instead of the 16 elements that compose the full substitution matrix. The elements of the library sequence are coded using 2 bits (for DNA) and will be used to address the corresponding substitution value.

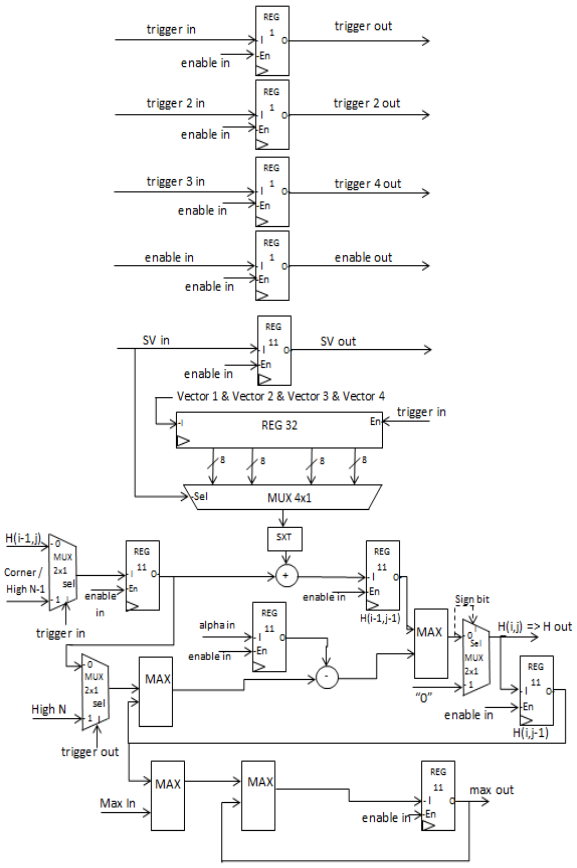


Fig 8 - Process Element Functional Unit.

The two multiplexers on the left of Fig. 8 are used for the partition of the computation of H matrix. The inputs for the multiplexer on the top left change according to the PE position on the PEs chain. In PE1 this multiplexer is used to select between the top-left neighbour initialization element and the input of the initialization column, for all other PEs this multiplexer selects between the initialization line input of a partition or the result from previous PE. The bottom left multiplexer introduces the initialization line top element.

The last multiplexer on the right is used to deal with negative values on the outcome computation of H(i,j). In these cases the output of the PE needs to be zero as shown in equation (1). Finally one register has been added to store a configurable α value.

7. Limitations

During the test of the application some limitations have been detected. The most important limitation is related to the maximum number of FUs that we are able to use in the datapath. The maximum number of PEs that has been synthesized successfully was 30, even when there is space available on the target FPGA. This number limits the partition size for testing sequences to 30 elements per partition. Another limitation is related to the vector size that can be transferred to *FireWorks*^{TM} and *SideWorks*^{TM}. It is possible to transfer vectors up to 128 elements, which limits the partition for the library sequence. For testing purposes the size of the partition used was 30 for the testing sequence and 120 for the library sequence.

The size of the registers used introduces another limitation on the maximum computable score without having overflow. All registers in the datapath have 11 bits, since the datapath uses signed calculation this limits the maximum computed score to 1023.

The results are presented with these limitations, although, there are solutions under study to improve the application.

8. Area Results

The project was implemented in a Spartan 3 XC3S5000 FPGA. Table 1 presents area results of implementations with different numbers of PEs in the Coreworks® processing engine platform.

PEs	Number of occupied slices	Total Number of 4 input LUTs
1	10 788	17 193
10	12 025	19 037
20	13 546	21 198
30	14 805	23 300

Table 1 – Areas of different implementations.

The *SideWorks*^{TM} and *FireWorks*^{TM} templates occupy a considerable amount of area (slices). Adding FUs to the *SideWorks*^{TM} template does not increase the total number of occupied slices too much. One PE alone (without *SideWorks*^{TM} overhead) occupies about 99 slices on this FPGA. However, from Table 1 is possible to average the number of occupied slices for each PE to be 134 slices. These overhead results from the extra FUs required on the data path by the *SideWorks*^{TM} platform. Therefore, for the referred FPGA, we should be able to accommodate 168 PEs on the *SideWorks*^{TM} hardware accelerator if no practical limitations exist.

9. Analysis of Application Performance

After the circuit has been synthesized and implemented using the proposed hardware on the FPGA, the minimum clock cycle attained was 27.7ns, resulting in a maximum frequency of 36MHz. Table 2 presents average load time of a configuration and different types of data transfers for 128 element vectors.

	Average (in cycles)
Configuration loading time	172
Loading a vector to memory bank in the FPGA(*Fireworks*^{TM} -> *Sideworks*^{TM})	367
Loading a vector from memory banks in FPGA(*Sideworks*^{TM} -> *Fireworks*^{TM})	244
Loading a register in the FPGA (*Fireworks*^{TM} -> *Sideworks*^{TM})	20,26
Loading a register from FPGA (*Sideworks*^{TM} -> *Fireworks*^{TM})	26
Loading a vector to the board memory banks (PC-> *Fireworks*^{TM})	13279
Loading a vector from the board memory banks (*Fireworks*^{TM} ->PC)	1004

Table 2 – Configuration and data transfer times in clock cycles.

From these results is possible to understand the impact of data transfer times on the application performance, especially with small partitions, because smaller partitions require more computations and more data transfers.

In Table 3 is presented performance of the application for different partition sizes measured in *Cell Updates Per Second* (CUPS), the number of cells from the H matrix processed per second.

Testing sequence X Library sequence	N. of partitions	Cycles per partition	Maximum computation (CUPS)	Total alignment time (µs)	Total alignment computation (CUPS)
1X1	1	72	0,5 M	15	67 K
30X30	1	104	320 M	15	60 M
60X60	2	134	498 M	30	120 M
60X120	4	192	347 M	38	189 M
120X30	1	192	694 M	15	240 M
360X120	12	196	680 M	93	464 M
510X510	85	196	655 M	470	553 M

Table 3 – Processing times in CUPS for different sequences.

From Table 3 we can calculate that the application performance increases with the size of the sequences being processed. This occurs until the sequences being tested have to be partitioned and each partition is computed by the *SideWorks*^{TM} accelerator separately. However, the performance of the total alignment computation increases with the sequences sizes.

According to [5] an optimized application (software only) has typically around 52 MCUPS average performance. Comparing this result with our results presented in Table 3 is possible to verify acceleration up to 13 times.

Other FPGA implementations of S-W algorithm achieve performances for LGP in the order of 9.2 GCUPS [1], but these results are achieved with a chain of 168 PEs and without partitions on the H matrix computation. If the limitations described for our application are solved, it is possible to achieve performances in the order of GCUPS as well.

10. Conclusions

In this work we have implemented the S-W sequence alignment algorithm using a hardware accelerator platform. The selected platform was the Coreworks processing engine, which has a RISC processor (*FireWorks*^{TM}), and a specific hardware accelerator (*SideWorks*^{TM}).

Although some practical limitations were encountered on the selected platform, we were able to implement a complete alignment application using the S-W algorithm with partitions.

The results showed that a considerable speed up was achieved even when partitions have to be used and some additional overhead is introduced by data transfer.

As future work we plan to overcome the platform limitations and implement the trace back and other parts of the algorithms in the *FireWorks*^{TM} processor.

Acknowledgements

This work was partially supported by national funds through Fundação para a Ciência e Tecnologia (FCT), under project HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis (reference number PTDC/EEA-ELC/113999/2009), the QREN Project 3487 – Sideworks, and project PEst-OE/EEI/LA0021/2011.

References

[1] T. Oliver, B. Schmidt, D. Maskell, "Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs", Proceedings ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays, 2005.

[2] P. Zhang, G. Tan, G.R. Gao, "Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform", Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications, September 2007.

[3] N. Sebastião, N. Roma, P. Flores, "Integrated Hardware Architecture for Efficient Computation of the n-Best Bio-Sequence Local Alignments in Embedded Platforms", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, July 2012.

[4] Z. Nawaz, M. Nadeem, H. van Someren, K. Bertels, "A parallel FPGA design of the Smith-Waterman traceback", International Conference on Field-Programmable Technology (FPT), December 2010.

[5] L. Hasan, Z. Al-Ars, S. Vassiliadis, "Hardware Acceleration of Sequence Alignment Algorithms – An Overview", International Conference on Design & Technology of Integrated Systems in Nanoscale Era, September 2007.

FPGA Based Synchronous Multi-Port SRAM Architecture for Motion Estimation

Purnachand Nalluri[1,2], Luis Nero Alves[1,2], Antonio Navarro[1,2]

[1]Institudo de Telecomunicações,
Pólo-Aveiro, Campus Universitário de Santiago,
3810-193 Aveiro, Portugal.
[2]Departamento de Electrónica, Telecomunicações e Informática,
Universidade de Aveiro
Campus Universitário de Santiago
3810-193 Aveiro, Portugal.
nalluri@av.it.pt, nero@ua.pt, navarro@ua.pt.

Abstract

Very often in signal and video processing applications, there is a strong demand for accessing the same memory location through multiple read ports. For video processing applications like Motion Estimation (ME), the same pixel, as part of the search window, is used in many calculations of SAD (Sum of Absolute Differences). In a design for such applications, there is a trade-off between number of effective gates used and the maximum operating frequency. Particularly, in FPGAs, the existing block RAMs do not support multiple port access and the replication of DRAM (Distributed RAM) leads to significant increase in the number of used CLBs (Configurable Logic Blocks). The present paper analyses different approaches that were previously used to solve this problem (same location reading) and proposes an effective solution based on the use of efficient combinational logic schemes to synchronously and simultaneously read the video pixel memory data through multiple read-ports.

1. Introduction

With the increased demand for multi-tasking and parallel processing, the modern applications for FPGA and ASIC based memory architectures cannot rely just on single port or dual port memories. Possible solutions for this problem are to increase the bus bandwidth and implement multiple port memories. Especially, when considering synchronous memories, implementing multiple read operation is required [1-2]. There are many applications where multiple read operations are highly required. For example, in robotic vision system, the object recognition system has to search many samples of live video frames and output one object that has minimum error. In reconfigurable vision systems, a shared memory is necessary to access the video content through multiple resources

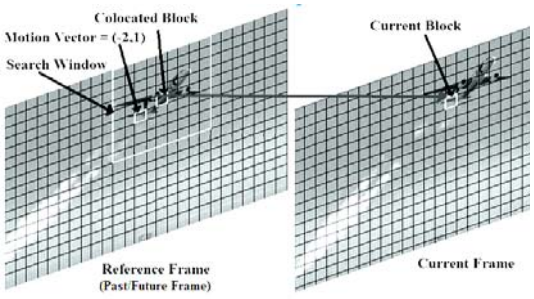


Fig. 1(a). Illustration of ME Process

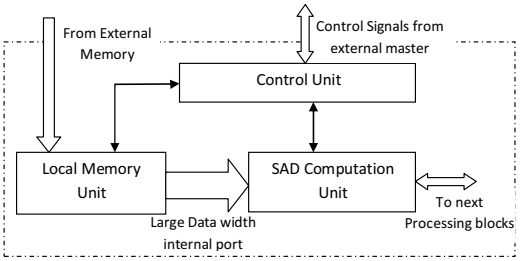


Fig. 1(b). Architecture for real time processing of Motion-Estimation

[1-2]. Similarly, in video compression systems, a shared memory is required to process many samples of video frame blocks in one clock cycle. The present paper focuses on the key role of multi-port SRAM (Static RAM) in motion estimation application [3]. Section 2 explains the memory architecture requirements for motion estimation. Section 3 discusses performance issues of some previously reported solutions. Section 4 proposes a new architecture suited for FPGAs. Section 5 details the experimental results, followed by Section 6 with the concluding remarks.

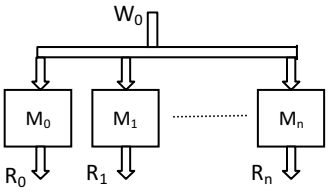


Fig. 2(a). Replicated Multiport Memory Architecture

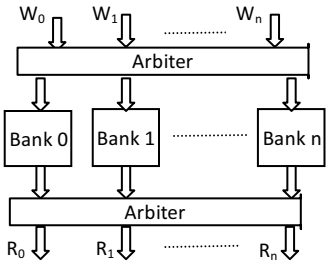


Fig. 2(b). Banked Multiport Memory Architecture

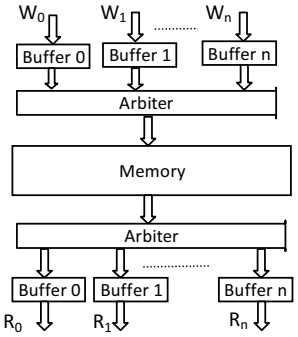


Fig. 2(c). Streaming Multiport Memory Architecture

2. Multi-Read Port Memory in Video Processing

For compressing video signals, a video encoder eliminates the temporal and spatial redundant information and encodes the error/residual information. For doing this, a video encoder typically uses predictive encoding techniques, so that residual information is further reduced. Motion estimation and motion compensation are the typical tools in a block-based video encoder that predicts and generates the video frames temporally. In a block based video encoder, each frame is divided into rectangular or square blocks of pixels, and hence prediction and other tools can be applied on each block. Especially when motion estimation is considered, the problem is very challenging since it is the most computationally intensive tool amongst all the tools in a video encoder.

Motion estimation is the process of searching the best matched block of a video frame in the past/future frame's ROI (Region of Interest, technically termed as search window) as shown in Fig.1(a) [3]. In order to find this best match, the ME algorithm uses matching criteria like SAD (Sum of Absolute Difference) or MSE (Mean Square Error) among others. For real-time applications, it is often required to apply parallel computation schemes. Instead of computing the SAD block by block sequentially, computing a set of blocks in parallel makes the ME task more efficient. Hence in real-time, it is required to access many blocks of pixels data in parallel from local memory, and send them to the SAD estimator as shown in Fig.1 (b). It is here, where the major design challenges occur, on how to implement the local memory without impairing overall performance.

3. Approaches for Implementing Multi-Read Port Memory

For writing the pixel block to local memory unit, the external bus is constrained to either 32 or 64 bits. Hence, there is no need to optimize the writing procedure. Instead, a design will have a great necessity in implementing multiple reading procedures in this scenario. For implementing the read ports there are many possible approaches [4-6] as described in the following sub-sections.

3.1 Memory Replication

Memory replication is the simplest way to implement multiple read ports. For reading the same memory location through 'n' multiple read ports, the memory is replicated 'n' times and the data is written to each of the write ports in parallel as shown in Fig.2(a). In case of FPGA BRAMs, for an application where one BRAM is required for one read port, the design has to replicate 'n' BRAMs for n read ports. The main disadvantage in this approach is the increase of system area and power.

3.2 Memory Banking

Memory banking technique divides memory capacity across smaller banks of physical memory. The total read and write port widths are also divided equally among each bank. Multiple requests to the same memory banks are usually handled by an arbiter. This is somewhat similar to memory replication except that in each memory bank, the total memory is divided instead of replicated, as shown in Fig.2 (b). However, the memory banking technique gives a limited support to multiple read operations, since in a given cycle only one read operation is possible from each memory bank. When the same memory bank is accessed by multiple sources, then each has to wait until its turn in arbiter is initiated. When all the banks are provided with a read instruction for each, the data can be read in parallel.

3.3 Streaming Multiport Memory

In streaming multiport (also called stream-buffered multiport or multi-pumping) memory, the entire memory is provided with only one read port (and one write port), and each read requester is provided an internal register to hold the requested data as shown in Fig. 2(c). Multiple requests are

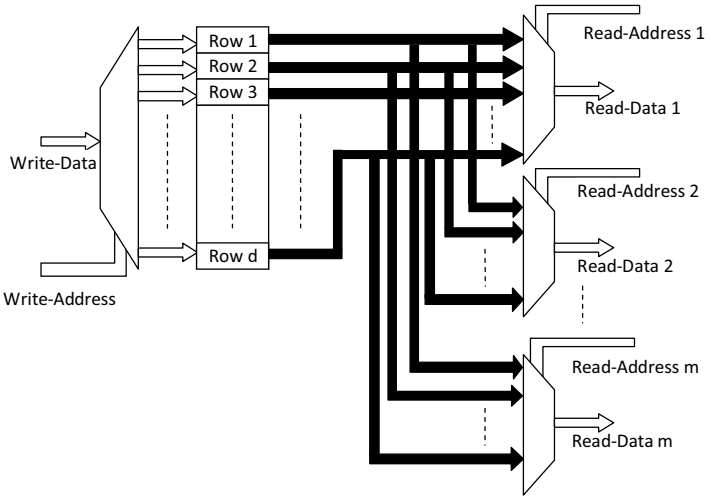


Fig. 3. Multiplexed Read Port (MRP) memory architecture for a single column of memory with 'd' rows and 'm' number of read ports

arbitrated using an arbiter or a multiplexer, with a clock frequency that is multiple of external clock frequency. Hence, this scheme decreases the maximum external clock frequency and degrades system's performance as shown in Section 5.

3.4 Other Multiport Memory

Besides the aforementioned architectures, there are other possible multi-porting techniques like bank-switching multiport memory, cached multi-port memory, etc. In bank switching, an external latch is added to the memory controller, to select or switch between memory banks. In cache based multi-port memory, a local cache is provided for each read-port, and a cache coherence protocol is used to synchronize cache contents. But in cache based porting, the read request may experience delay (variable delay) depending on the request sequence pattern and cache coherence protocol.

4. Proposed Technique

To handle multiple read transactions within the same clock cycle and without latching, each memory location, one possibility is to output data to all the read ports through a multiplexer circuit as shown in Fig.3. The select line for this multiplexer will be nothing but the read address port. Through this way the memory can have any number of read ports, independent of number of write ports.

As shown in Fig.3, the data is written to DRAM (Distributed RAM) column through one write port. Data is read through 'm' read-ports via multiplexer and each select line is the read address for that particular read operation. The advantage in this circuit is that all the address lines can be selected in the same clock cycle and hence any number of memory locations can be read synchronously in one clock cycle. The second advantage is that, the same memory location can be read through multiple read ports and hence data can be shared with multiple ports without any extra cycles.

The memory architecture shown in Fig.3 is for one column of memory (or one column of block of pixels in a video frame), however, the proposed MRP (Multiplexed Read-Port) memory method can also be applied to an entire block of pixels in a video frame. Typically, the motion estimation block requires only luminance information, where each pixel is 8 bits wide for the luminance component. The write port is drawn from external memory which is typically constrained to 64 bits wide in modern FPGAs [7]. Hence each row could be 64 bits wide storing 8 pixels of information. Thus in one clock cycle, 8 pixels of memory are written.

Fig.4 shows the application of the proposed concept. N represents number of pixel columns in a block of video frame. The internal architecture of each pixel column memory is shown in Fig.3. Usually, for H.264/AVC video standard, the maximum block size is 16x16 pixels and hence N will be equal to 2 (=16/8) in this case. For the latest video coding standard HEVC, the maximum block size is 64x64 and hence N will be equal to 8 (=64/8) for this standard. Similarly, the memory depth'd' is equal to 16 and 64 for H.264/AVC and HEVC, respectively. The variable 'M' is the number of parallel read ports in each pixel column memory. Data is written through one write port, and the corresponding memory column is selected using data and address select lines. At the output, the data is read through M read ports from each column. From all the NxM read ports, any number of read ports can be selected through the switch box. This is similar to memory banking, but the main difference is that here the designer can customize M, and can read multiple ports from each bank/column. While in memory banking, the read request is possible for only one memory location, in single cycle from each bank. Furthermore, in the design shown in Fig.4, M

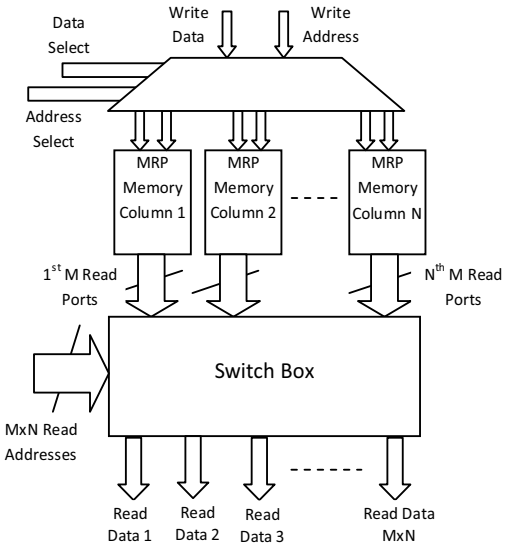


Fig. 4. Proposed Multiplexed Read Port (MRP) memory architecture..

need not be the same in each bank/column, and the designer has full flexibility to choose M, depending on the algorithm used.

5. FPGA Synthesis Results

The proposed MRP method was implemented in verilog and synthesised using Virtex-5 FPGA [7]. The row width for each memory location is chosen as 64, since the external bus is also configured to 64, and thus able to write 8 pixels of data in one clock cycle. The memory depth 'd' is configured as 16, and the memory columns 'N' is chosen as 2, which is suitable for a 16x16 block size pixels. In the proposed architecture, the value 'M' need not be a power of 2 ($M=2^{integer}$), it can be any odd number also. Without losing generality, 'M' is chosen as 5 implying that 5 concurrent read operations are possible from each memory column. Hence number of replication memories in memory replication architecture and number of read ports in stream-buffered memory architecture are also chosen as 5.

The proposed method is compared with the other methods – memory replication method and streamed memory buffer method. Memory banking is not considered for comparison since it is not able to support multiple read operations. Table 1 shows the comparison results. The results show that MRP method has the highest maximum operating frequency, with slight increase in number of CLBs (LUT-FF pairs) used in FPGA. The memory replication method has the highest resource usage, due to increase in the number of slice registers. The stream buffered method has the lowest maximum operating frequency due to the combinational logic of arbiter switching used.

Table 1. Virtex-5 FPGA synthesis results for different memory architectures.

	Memory Replication	Stream Buffer	MRP
#Slice LUTs	4573	2427	4543
LUT Utilization (#Total LUTs : 69120)	6%	3%	6%
#Slice Registers	12672	5260	4480
Slice Regs Utilization (#Total Slice Regs: 69120)	18%	7%	6%
#LUT-FF pairs	16121	6301	7899
Total LUT-FFs Utilization (#Total LUT-FFs: 69120)	23%	9%	11%
Max.Frequency (MHz)	329.96	273.09	335.21

6. Conclusions

This paper presented a multiple port read memory architecture. The propose architecture was synthesized using VIRTEX-5 FPGAs. The synthesis results show that the proposed MRP (Multiplexed Read Port) memory architecture maintain a good balance between FPGA resources used and maximum operation frequency. This architecture can be used for video processing applications like motion estimation, intra prediction or in other applications where simultaneous read operation from a common shared memory is required. Performance evaluation for motion estimation methods in video processing applications, using the aforementioned architecture will be investigated in future contributions on this line of research.

References

[1] J.M. Perez, P. Sanchez, M. Martinez, "High memory throughput FPGA architecture for high-definition Belief-Propagation stereo matching", IEEE Int. Conf on Signals Cir & Sys, pp. 1-6, Nov. 2009.

[2] S. Chang, B.S. Kim, L.S. Kim, "A Programmable 3.2-GOPS Merged DRAM Logic for Video Signal Processing", IEEE Trans. on Circuits and Systems for Video Technology, vol.10, No.6, Sept. 2000.

[3] N. Purnachand, L.N. Alves, A. Navarro, "Fast Motion Estimation Algorithm for HEVC", IEEE ICCE-Berlin Conf 2012, Berlin, pp. 34-37, Sept 2012.

[4] H. Zhao, H. Sang,T. Zhang, Y. Fan, "GEMI: A High Performance and High Flexibility Memory Interface Architecture for Complex Embedded SOC", IEEE CSSE Conf 2008, pp. 62-65, Dec 2008.

[5] W. Ji, F. Shi, B. Qiao, H. Song, "Multi-port Memory Design Methodology Based on Block Read and Write", IEEE ICCA Conf 2007, May 2007.

[6] M. Saghir, R. Naous, "A Configurable Multi-ported Register File Architecture for Soft Processor Cores", Int. Workshop on Appl. Reconfig. Computing, Springer-Verlag - pp. 14-27, March 2007.

[7] Xilinx Virtex-5 User Guide ver. 5.4, March 2012. "http://www.xilinx.com/support/documentation/user_guides/ug190.pdf",

Evaluation and integration of a DCT core with a PCI Express interface using an Avalon interconnection

Sérgio Paiágua, Adrian Matoga, Pedro Tomás, Ricardo Chaves, Nuno Roma
INESC-ID Lisbon
sergio.paiagua@ist.utl.pt, {Adrian.Matoga, Pedro.Tomas, Ricardo.Chaves, Nuno.Roma}@inesc-id.pt

Abstract

The Discrete Cosine Transform (DCT) plays an essential role in today's media-driven world, as it is at the heart of the most widely used compression algorithms for both images and videos, such as JPEG or the MPEG family codecs. Given its ubiquity and high computational requirements, it is common to find it implemented as an accelerator core within more complex systems, such as SoCs, in order to keep power consumption low and enable higher processing throughputs. Although several hardware implementations exist, and are already widely discussed in the literature, their integration with a generic computing framework is seldom discussed. In this paper, a co-processor architecture for the computation of the DCT on a generic processing platform is presented. A publicly available DCT core is interconnected to a DMA engine through an Avalon Stream interface, which performs data transactions with a general purpose processor, by using a standard PCI Express interface. Implementation results on an Altera FPGA, including resource utilization and maximum achievable throughput, are presented and discussed.

1. Introduction

Despite the recent advances in both storage and transmission of digital data, the need for efficient compression mechanisms for media applications not only has been maintained but has even seen an increase due to the most recent trends in consumer electronics. In fact, even though high throughput fiber-based broadband connections are becoming the norm among conventional computers, there is a new class of devices that often deal with mobile internet connections, operating at a fraction of the bandwidth available on their fixed counterparts. Many of these compression mechanisms make use of the 2D Discrete Cosine Transform (DCT) to encode image and video data. In fact, this transform is at the heart of some of the most commonly used video and image codecs, such as the MPEG-1/2/4 and JPEG standards, and due to its computationally intensive nature, it constitutes a good target for optimization.

One of the most common approaches to tackle the problem of decoding high-quality video in real-time has been to offload the necessary computations to dedicated hardware. However, the development and integration of these co-processors with the rest of the system is not trivial, and special attention must be given to the communication between the accelerator, the processor or the system memory, as these can easily become the bottleneck, hence limiting the performance gain that could potentially be achieved with the dedicated unit.

In this paper, a co-processor architecture to perform the 2D discrete cosine transform (2D DCT) is presented. The proposed architecture makes use of a DCT core, developed by Unicore Systems [1], connected to a DMA engine through an Avalon Stream interface. The DMA engine is, in turn, driven by a PCI Express interface which acts as an abstraction of the upper layers of the system. The resulting design was implemented and tested using an Arria II FPGA by Altera.

The rest of the paper is organized as follows. Section 2 provides an overview of the 2D DCT and how it fits within the JPEG and MPEG standards. Some common implementation techniques are briefly discussed. In section 3, the structure of the PCI Express interface and the associated DMA engine is presented. Section 4 characterizes the DCT core with emphasis on the latency, throughput, and customization options, and describes the wrapper entity that enables it to be accessed through an Avalon Stream interface. Section 5 presents and discusses the obtained results for the target device, including maximum operating frequency and resource usage of the various components of the architecture, as well as the overall system performance, discussing possible bottlenecks. Section 6 concludes this paper by summarizing the characteristics and performance of the architecture and proposes future work directions.

2. DCT Overview

A digital image, which may represent a frame in a movie sequence or be just a picture, is a collection of pixels arranged in a matrix-like fashion. Each of these picture elements may have associated one or more intensity values, depending if it is a colour image or, instead, a grayscale one. In practice, chroma sub-sampling is usually employed in a colour image to reduce the amount of data used in the two colour channels, usually referred to as *chrominances*, thus reducing the total storage needs of the image [2]. Even with this technique, the total amount of data of a motion picture would make it impractical to store and transmit any content with adequate resolution and quality, let alone high-definition video.

It is then clear that some form of efficient compres-

sion must be employed to make this type of image content manageable by all sorts of devices. Image compression techniques achieve reductions in the total content size by exploiting two main concepts: irrelevancy and spatial redundancy between neighbouring pixels, which relies on the knowledge that the Human Visual System (HVS) is not equally sensitive to all features in an image. While the first concept does not imply any loss of data, the latter forcibly results in data loss, as it is not possible to recover elements that have been discarded. However, since the exploitation of spatial redundancy is generally achieved through the use of transforms, which are usually implemented with a finite arithmetic precision, this processing element also leads to some data loss. Hence, the goal of any *lossy* compression algorithm, i.e, one whose operation does not produce a mathematical equivalent of the input image, is to maximize the perceived quality of the compressed image while reducing its size as much as possible. Conversely, *lossless* algorithms cannot exploit irrelevance and, as such, must only rely on the reduction of statistical redundancy, using techniques that are widely used for data compression in general, such as entropy coding, in order to achieve the desired size reduction.

In most image and video *codecs*, such as JPEG or the family of MPEG-1/2/4 codecs, the DCT is extensively used to reduce the correlation between neighbouring pixels. By applying this transform to an (N×N) pixels block (with N = 4, 8 or 16), a map of frequency domain components is obtained. Within this map, the first entry corresponds to the average of the pixel values, the DC component, while the rest of the entries represent the AC components, with increasing frequency content as they move towards the lower-right corner of the matrix. The 1D DCT can be formulated as defined in [3]:

$$X(m) = \sqrt{\frac{N}{2}}\xi(m)\sum_{i=0}^{N-1}\cos\Big(\frac{m(i+\frac{1}{2})\pi}{N}\Big)x(i)$$

In the previous expression, $x(i)$ represents the original input sequence, whereas $X(m)$ corresponds to its transformation. Thus, the index i denotes the coordinate in the spatial (pixel) domain, whereas m represents the coordinate in the transform-domain. Additionally, $\xi(m)$ is defined as:

$$\xi(m) = \begin{cases} \sqrt{\frac{1}{2}} & \text{for m = 0 or m = N} \\ 1 & \text{for m = 1,2,..., N-1} \end{cases}$$

In practice, the DCT computation is usually performed by using precomputed values, the *basis functions*, meaning that the output of the DCT operation will be a set of coefficients that are obtained from each of these basic elements. These elements form a a (8×8) kernel matrix $\mathbf{T}$, which is defined by [3]:

$$[\mathbf{T}(m,i)] = \sqrt{\frac{N}{2}}\xi(m)\cos\Big(\frac{m(i+\frac{1}{2})\pi}{N}\Big)$$

A representation of the first 64 DCT *basis functions* is presented in figure 1.

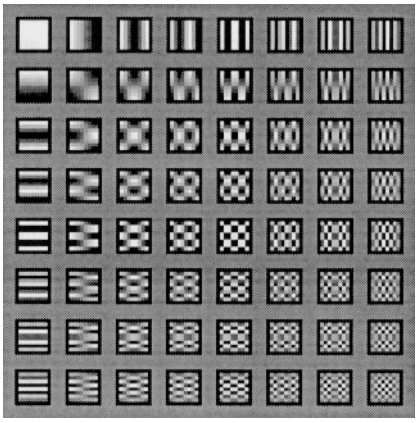


Figure 1. 64 *basis functions* of the 8×8 DCT, where the top-left entry corresponds to the DC component and all the others to AC components [4]

.

Naturally, given that the DCT *basis functions* are independent of the image to be transformed, the obtained results cannot be optimal, as not every image possesses the same frequency components and, as such, the representation of the image could still be achieved with less coefficients if these functions were specifically tailored for each image. This is the principle behind of the Karhunen-Loeve Transform (KLT), which achieves the highest degree of decorrelation between transmitted coefficients by first determining the optimal set of *basis functions* for a given image, through the computation of the eigenvectors of its covariance matrix[5]. Implementing the KLT, however, requires a great computational effort and the fastest-algorithms that are available for its computation are not as good as for other transforms. In addition, its superior compacting capabilities are not necessarily reflected in the resulting perceived quality of the image [5]. As such, the DCT is usually preferred as it closely approaches the statistically optimal KLT for highly correlated signals [6], and still provides better energy compaction than other popular transforms, such as the Fast Fourier Transform (FFT), making it more appropriate for image processing [4].

Most conventional approaches to the computation of the 2D DCT for an N×N block usually adopt a row-column decomposition, followed by the application of two 1D DCTs with N points [3]. However, this row-column method requires a matrix transposition architecture which increases the computational complexity as well as the total resource usage. On the other hand, alternative polynomial approaches reduce the order of computation, as well as the number of required adders and multipliers [7]. Due to its popularity and practical interest, several fast implementations of the discrete cosine transform exist, such as the ones proposed by Lee, Hou and Cho [8][9][10]. Among them, the most efficient is the Nam Ik Cho (Chan and Ho) algorithm [6].

When the DCT is integrated within a codec, the computation of the coefficients is followed by a quantization module, which exploits irrelevancy and, as such, it may in-

troduce some error. In this operation, each DCT coefficient is divided by a *Quantization Coefficient* and then rounded to an integer value. The entries of the associated *Quantization Matrix* are adjusted so that a desired perceptual quality is achieved, which usually involves setting different quantization steps for each coefficient, as the HVS is not equally sensitive to all frequency components.

3. PCI Express and DMA engine

The PCI Express framework on which this architecture is based was proposed in [11] and consists of a general-purpose-processor connected to a PCI Express controller, which accesses a custom peripheral on a reconfigurable device through the use of a memory-mapped interface defined in the Altera's family of Avalon interconnections. The transfer of data to and from the on-chip memory is accomplished by two DMA controllers, which operate simultaneously, thus exploiting the full-duplex capabilities of the PCI Express Link.

The architecture herein presented proposes a modification to this PCI Express framework, which trades the memory-mapped interfaces for a simpler and faster streaming interconnection, the Avalon Stream. In addition, since now the accelerator directly receives the data, there is no need for an on-chip memory, apart from those used by each DMA controller to store the corresponding descriptors. As in the original version, two DMA controllers are used to allow the independent and simultaneous transfer of data to and from the attached peripheral. A block-diagram representation of the modified framework is shown in figure 2, where the memory elements for the storage of the DMA descriptors was omitted for clarity purposes.

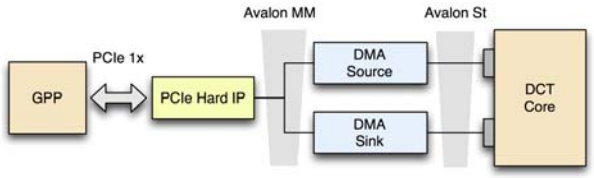


Figure 2. Architecture of the modified PCI Express framework.

A Kontron MSMST board [12] was used for the implementation of the system. The key features of this board are an Intel Atom E665C microprocessor, clocked at 1.3 GHz, and an Altera Arria II GX FPGA interconnected through two PCIe 1x links. Only one of these hard PCI-Express controllers is used to communicate with the reconfigurable fabric, which is driven by a 125 MHz clock signal provided by the controller.

4. DCT Core with Avalon Stream Interface

Although the DCT operates on fixed-size blocks, its operation can be implemented in a streaming fashion, provided that the blocks are inserted sequentially, as a stream of data. This approach yields higher processing throughput when compared to a standard memory-mapped interface, which is inherently more complex and introduces additional overhead on the data transactions. The following sections describe the implementation of a stream-based DCT core.

4.1. DCT Core and Output Buffer

The computation of the 2D DCT in the herein presented architecture is performed by a publicly available soft-core developed, by Unicore Systems, a start-up company based in Kiev, Ukraine [1]. The selection of this particular implementation of the DCT resulted from a preliminary evaluation of three different soft-cores available at OpenCores [1], an open source community dedicated to the development and sharing of IP cores. The main aspects that were taken into consideration during the selection process were the resource occupation and maximum achievable operating frequency on the targeted FPGA device, the processing throughput and initial latency and, finally, the customization options and available documentation for the core.

The selected DCT core performs an 8×8 2D DCT and is fully pipelined, meaning that blocks can be input in sucession, with no waiting period between them. Given its deeply pipelined structure, an initial latency of 132 clock cycles is imposed before the first result is obtained, after which the core stabilizes at a processing rate of 64 pixels (a full block) in 64 clock cycles, i.e, one pixel per clock cycle. The main features of the core are summarized below:

- 1 pixel per clock cycle throughput
- 11-bit input coefficients
- 12-bit output results
- 132 clock cycles of latency
- Signed or unsigned input data
- Scaled or unscaled output data

The 2D DCT is computed through a row-column decomposition, by first transforming the columns and then the rows. The number of operations that are needed to compute each 8×8 block are significantly reduced by employing the Arai, Agui and Nakajima 8-Point DCT algorithm, which results in only 13 multiplications [13]. Due to the wide use of resource sharing, only 4 hardware multipliers are required. Moreover, if the scaled output option is selected, the number of multiplications is reduced to 5, requiring only 2 hardware multipliers. In such configuration, however, the proper scaling must be performed at a later stage, before the entropy encoding.

The corresponding signal flow graph of the architecture that computes each of the 1D DCTs that are used for computing the full 2D DCT is depicted in figure 3. In this graphical representation, multiplications are represented by a box containing the value of the multiplication coefficient,

[1] OpenCores - www.opencores.org

while additions are simply represented by the intersection of two segments. This representation makes it easy to understand the difference in the number of multiplications between the scaled and non-scaled output options of the core. In the case of the former, the 8 multiplications, labeled from S_0 to S_7, are not performed, as they are relegated to the next stage of the coding algorithm, thus reducing the total number of multiplications to 5. When compared to the 64 multiplications that are needed when using a direct matrix multiplication approach, the performance and area gains resulting from the use this algorithm are evident.

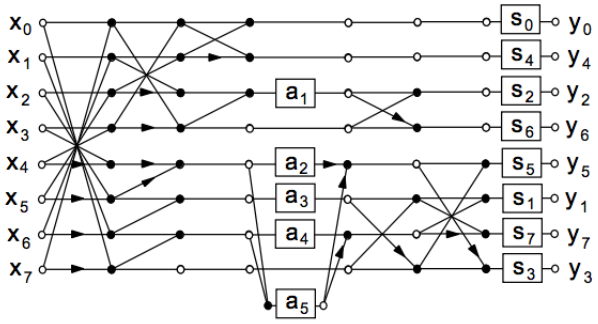


Figure 3. Signal flow graph for the scaled 1D DCT.

To further reduce the resource usage, the intermediate results are stored and transposed by using appropriate FIFO buffers based on SRL16 elements, implemented with LUTs, which means that no Block RAMs are needed.

Hence, the core interface consists of two data buses for input and output of pixels and DCT coefficients, a clock signal, and four control signals, as described in table 1.

Signal	Direction	Description
RST	IN	Synchronous reset signal for the core
EN	IN	Clock enable
START	IN	Triggers start of the computation
RDY	OUT	Signals the first valid DCT coefficient

Table 1. Control interface of the DCT Core.

After the *START* signal is asserted, a new input value is input every clock cycle, while the *EN* signal is kept *HIGH*. Similarly, after the *RDY* signal is activated, a new DCT coefficient is outputted on every rising edge of the clock. This means that it is not possible to input new pixel values without outputting the same number of coefficients and vice-versa. While this is not a problem if the core is always used in a streaming fashion, i.e, fed with a continuous stream of input data and sampled at the same rate, when the reading or writing operations need to be independently interrupted, the result will be loss of data (in the case that a write is performed without a read), or block corruption (if a read was done with no accompanying write, thus pushing a non-valid input into the core and affecting the computation of a full 8×8 block).

Since the Avalon Stream interface does not, by itself, guarantee uninterrupted stream operation, it was necessary to add a FIFO element to the output of the core. In this way, the DCT core will push an output coefficient to the FIFO whenever a new pixel value is input and the FIFO is not full, thus avoiding any loss of data. Block corruption is also avoided, as the DCT coefficients will be only read from the FIFO until it is empty, at which point the read requests are ignored until the FIFO is non-empty again, which is achieved by inputting new values into the core.

4.2. Avalon Stream Interface

The existing family of Avalon interfaces was developed by Altera to provide an easy way to connect components within their FPGAs. The standard includes interfaces appropriate for streaming high-speed data, reading and writing registers and memory, and controlling off-chip devices.

Considering that the architecture described in the previous subsection does not define addresses of any kind, nor does it directly interface with a memory component, the most appropriate interface to use in this architecture is the Avalon Stream, which provides an unidirectional flow of data in a point-to-point fashion. Since both read and write operations must be performed in the core, two instances of this type of interconnection must be used. When writing, the core will act as a *sink* and the DMA controller acts as a *source*. When reading the roles are reversed.

In addition to the *clock*, *reset* and *data* bus signals, each instance of the Avalon Stream interface can also include a *valid* and *ready* signal. These two signals can be used to control the flow of data from the *source* to the *sink*, which corresponds to *backpressure*, in the Avalon interface terminology. In this case, the *source* will assert the *valid* signal whenever it has new data to send, but should only move to the next data element after the *sink* has acknowledged that it will receive that given element. Figure 4 depicts such a situation.

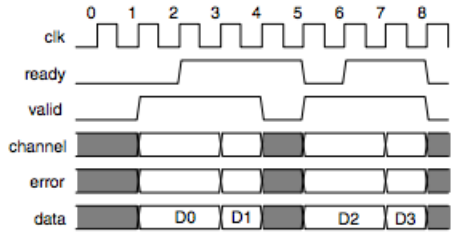


Figure 4. Data transaction from *source* to *sink* with backpressure enabled [14].

The width of the *data* bus can be configured to take any value between 1 and 4096 bits. Regardless of this size, the data transaction will take a single clock cycle. As such, this design parameter can and should be adjusted to maximize the communication efficiency between the co-processor and the DMA-engine. To allow these adjustments to be made at any time in the design process, the Avalon Stream wrapper developed for the DCT core does not rely on a specific data width value. Instead, it uses a generic parameter to define this value, which in turn configures an *Input Buffer* that is placed right before the input

of the core. This buffer assures that the *Source* does not write more pixel values than the DCT core can handle. As such, if the data width is defined to be 32 bits, corresponding to four 8-bit values, the *Input Buffer* will only assert the *valid* signal again after the four pixel values have been input into the core. At the output of the FIFO, however, no buffering is employed, as each of the 12-bit output values is sent within a 32-bit word. Therefore, while a full 32-bit value is transferred to the upstream DMA controller, only the 12 least-significant bits contain any information.

Figure 5 presents a block diagram of the full accelerator structure.

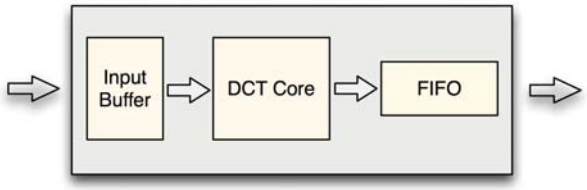


Figure 5. Block diagram of the accelerator architecture.

5. Implementation Results

The full system was assembled using Altera's Qsys implementation tools, while resource usage and frequency figures were obtained in Quartus for the Arria II GX EP2AGXE6XX FPGA device. A summary of the hardware resource requirements for both the full framework and the DCT accelerator unit are presented in tables 2 and 3, respectively.

Resource	Utilization	Available
LC Registers	8 945 (17.68 %)	50 600
LC Combinationals	6 917 (13.67 %)	50 600
Block memory bits	665 459 (12.39 %)	5 246 k
DSP 18-bit element	7 (2.24 %)	312

Table 2. Resource usage of the full system.

Resource	Utilization	Available
LC Registers	3 110 (6.15 %)	50 600
LC Combinationals	1 488 (2.94 %)	50 600
Block memory bits	768 (0.01 %)	5 246 k
DSP 18-bit element	7 (2.24 %)	312
Multiplicator 12-bit	2	
Multiplicator 18-bit	2	

Table 3. Resource usage of the DCT core.

By taking into account the Logic Cell (LC) instantiation reports, it can be concluded that the DCT core has a significant influence on the hardware resources of the architecture. This observation is supported by noting that a hard-silicon PCIe link controller is being used to provide the PCIe connectivity of the architecture, instead of a resource-heavy soft solution implemented on the reconfigurable fabric. Thus, this element is not included in the hardware resource usage results. In fact, from a resource usage point of view, the system is composed of only the DCT accelerator core and two DMA controllers, each with their own descriptor buffer. These buffers account for nearly all of the block memory usage, as the DCT core only requires 768 bits for the 12x64 output FIFO.

Hence, it is interesting to note that while the core was originally targeted for Xilinx FPGA devices and the storage to accommodate the intermediate results was described in such a way as to utilize the SRL16 elements instead of block RAM memory, the Quartus synthesis tool was also able to avoid the use of these elements without any additional modifications to the core VHDL description.

The DSP allocation, on the other hand, did not reveal the same degree of optimization. In fact, although the synthesis tool reported a total of 4 multipliers, two of them 12-bit wide and the other 18-bit wide, as expected from the core specifications, 7 DSP 18-bit element were also used. This seemingly odd result can be understood by examining the structure of the DSP block for the targeted FPGA, depicted in figure 6.

These blocks are designed to accelerate the most typical digital signal processing operations, by including a combination of dedicated elements that perform multiplication, addition, accumulation and dynamic shift operations [15]. Each of these entities consists of two half-DSPs, which share most control signals but are, for the most part, independent. Each half-DSP contains four 18x18 bit signed multiplicators. However, since the output is limited to 72-bit, only two multiplications can be performed. Unfortunately, due to the target applications of these elements, it is not possible to perform two independent multiplications within the same half-DSP, as their output is combined, as seen in figure 7. As such, when creating an independent multiplier, the synthesis tool will use two 18-bit elements. Similarly, to implement independent 12-bit multiplications, the tool will also use two 18-bit elements.

Thus, the total number of 18-bit elements should be 8, but by carefully inspecting the core's VHDL description, it can be seen that one of the 12-bit multiplications is followed by an addition. Therefore, the synthesis tool is able to fully exploit the internal architecture of an half-DSP by implementing these two operations with only one of these elements. In particular, one of the two 18-bit multipliers is used to perform the 12-bit multiplication itself, while the other is used to *pass* the second operand of the addition to the adder within the half-DSP structure, as depicted in figure 7. Since only one of the 18-bit elements is used for an actual multiplication, the tool reports the usage of only one DSP 18-bit element for this case, which results in a total usage of 7 of these elements for the full core, as all the other multipliers are of an independent nature.

To determine the maximum operating frequency of the

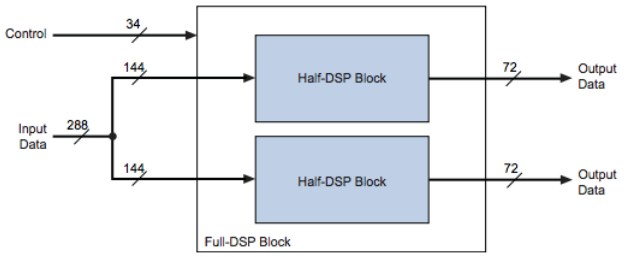


Figure 6. Block-diagram of a full DSP block in Arria II devices [15].

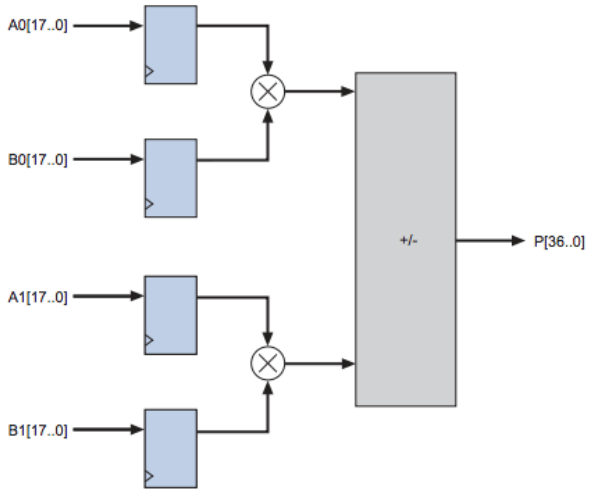


Figure 7. Basic *two-multiplier adder* building block in Arria II devices [15].

DCT core and, consequently, its maximum throughput, a timing analysis was performed using the TimeQuest Timing Analyzer. The result was a maximum operating frequency of 245.88 MHz, which is significantly higher than the 125 MHz clock, obtained from the PCIe link, which is used to drive the accelerator in the architecture herein described. As such, the maximum throughput of the core is limited by the PCIe interface clock and is given by:

$$Throughput = \frac{12 * f_{clk}}{DMA_{width}/8}$$

After the steady state has been reached, on every clock cycle the DCT Core outputs one 12-bit DCT coefficient for every 8-bit input. Thus, for a DMA data bus width of 32-bit (DMA_{width}) and a clock frequency (f_{clkc}) of 125 MHz, the resulting throughput is 375 MBit/s. However, from the DMA perspective, the core outputs 32-bit data values, although only 12 of those bits are significant. In these circumstances, the actual transaction throughput, i.e, based on 32-bit word transfers, increases to 1000 MBit/s

To characterize the proposed platform in terms of the maximum achievable throughput, a simple FIFO was inserted in the place reserved for the DCT accelerator, in order to simulate a situation where the latency of this element is negligible when compared to the operation of the DMA controllers and PCIe interface. The results showed that, for a big enough chunk size, i.e, from 16384 bytes upwards, the throughput saturated at 800 Mbit/s, as shown on figure 8. This allows us to conclude that, since the effective throughput of the DCT Core is still higher than this value, the performance of the system will exhibit the behaviour shown in figure 8.

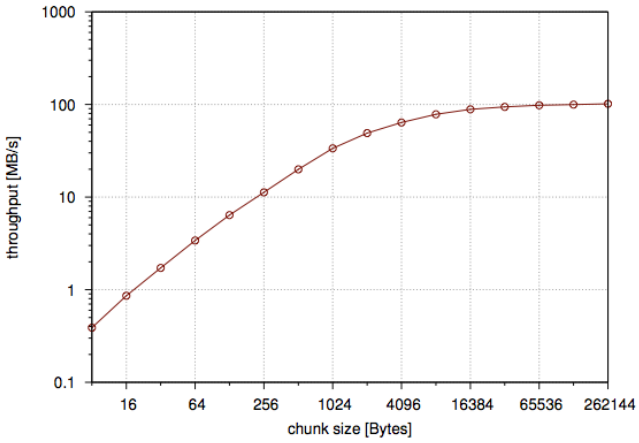


Figure 8. Evolution of the data transfer throughput for an ideal stream-based accelerator for different chunk sizes.

6. Conclusion and Future Work

The presented manuscript described and discussed, the integration of a 2D DCT core with a generic PCI Express platform for stream-based dedicated accelerators. The interface between the PCIe link and the core was accomplished by employing a standard data stream interconnection, as defined in the Avalon interfaces family. The handling of data transactions to and from the peripheral is assured by two DMA controllers, which enable the simultaneous upstream and downstream of data.

The implementation results on a Kontron MSMST board featuring an Intel Atom CPU clocked at 1.3 GHz and an Altera Arria II GX FPGA revealed that the design makes an efficient use of hardware resources. However, the attained results also suggest that the DCT core can still benefit from a revision in order to optimize it for Altera devices. Specifically, in what concerns the particular characteristics of its DSP blocks.

Finally, this paper shows that the bottleneck in the proposed platform does not lay on the accelerator itself, but instead on the remaining system, namely on the operation of the DMA controllers and the PCIe link. Future work may consider a different and independent clock domain for the DCT core, at least two times faster than the PCIe clock.

7. Acknowledgements

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project "HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis" (reference number PTDC/EEA-ELC/113999/2009) and

project "Threads: Multitask System Framework with Transparent Hardware Reconfiguration" (reference number PTDC/EEA-ELC/117329/2010) and project PEst-OE/EEI/LA0021/2011.

References

[1] OpenCores. Pipelined dct/idct. `http://opencores.org/project,dct_idct`, October 2012.

[2] Charles Poynton. *Digital video and HDTV algorithms and interfaces*. Morgan Kaufmann, 2003.

[3] Nuno Roma and Leonel Sousa. A tutorial overview on the properties of the discrete cosine transform for encoded image and video processing. *Signal Processing*, 91, November 2011.

[4] Dave Marshall. The discrete cosine transform. `http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html`, April 2001.

[5] Ed. K. R. Rao and P.C. Yip. *The Transform and Data Compression Handbook*. CRC Press LLC, 2001.

[6] Nam Ik Cho. Fast algorithm and implementation of 2-d discrete cosine transform. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS*, 38(3), 1991.

[7] R.Uma. FPGA implementation of 2D DCT for JPEG image compression. *(IJAEST) INTERNATIONAL JOURNAL OF ADVANCED ENGINEERING SCIENCES AND TECHNOLOGIES*, 7(1), 2011.

[8] N.I Cho and S.U.Lee. DCT algorithms for vlsi parallel implementation. *IEEE Trans. Acoust., Speech, Signal Processing*, 38(3), 1990.

[9] B.G. Lee. A new algoritm to compute the discrete cosine transform. *IEEE Trans. Acoust., Speech, Signal Processing*, 32(3), 1984.

[10] H.S Hou. A fast recursive algorithms for computing the discrete cosine transform. *IEEE Trans. Acoust., Speech, Signal Processing*, 35(3), 1987.

[11] A. Matoga, R. Chaves, P. Tomas, and N. Roma. An FPGA based accelerator for encrypted file systems. *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), HiPEAC*, July 2012.

[12] Kontron. Kontron MSMST. `http://us.kontron.com`, 2012.

[13] Unicore Systems Ltd. Discrete cosine transform (DCT) ip core user manual. Technical report, Unicore Systems Ltd, 2010.

[14] Altera. Avalon interface specifications. Technical Report MNL-AVABUSREF-2.0, Altera, 2011.

[15] Altera. Arria II device handbook volume 1: Device interfaces and integration. Technical Report AIIGX51001-4.4, Altera, 2010.

 ISBN: 978-972-8822-27-9

Reconfiguração dinâmica

 ISBN: 978-972-8822-27-9

Reconfiguração Dinâmica Parcial de FPGA em Sistemas de Controlo

José Luís Nunes[1], João Carlos Cunha[1],
Raul Barbosa[2], Mário Zenha-Rela[2]
[1]*Instituto Politécnico de Coimbra / CISUC,*
[2]*Universidade de Coimbra / CISUC*
jnunes@isec.pt, jcunha@isec.pt,
rbarbosa@dei.uc.pt, mzrela@dei.uc.pt

Resumo

As capacidades de processamento paralelo e de reconfiguração, aliadas ao baixo custo de desenvolvimento, têm conduzido à crescente utilização de dispositivos eletrónicos baseados em FPGA, em detrimento de ASIC, em sistemas de monitorização e controlo de processos. A sua capacidade de reconfiguração dinâmica para atualização, correção de falhas de desenvolvimento e alteração funcional dos sistemas tornou estes dispositivos particularmente adequados para instalação em localizações remotas, inacessíveis ou de difícil acesso (e.g. off-shore, desertos, montanhas, espaço, fundo dos oceanos), ou em ambientes extremos. Neste artigo caracterizamos o perfil dinâmico das aplicações de controlo que podem tirar partido da capacidade de reconfiguração dinâmica presente nos mais recentes dispositivos FPGA. Os resultados de validação experimental descritos neste artigo demonstram que processos físicos com constantes de tempo acima dos 100ms podem extrair todos os benefícios da reconfiguração dinâmica, já após a sua instalação no terreno, sem risco de perda de controlo da aplicação física.

1. Introdução

Os dispositivos FPGA (*Field Programmable Gate Arrays*) têm vindo a tornar-se cada vez mais importantes no domínio dos sistemas embebidos, já que oferecem a possibilidade de modificação dos sistemas já após a sua instalação, a um custo mais baixo do que a substituição de hardware estático (ex: ASIC – *Application Specific Integrated Circuit,* ou sistemas baseados em microprocessador). Esta possibilidade é interessante não só durante a fase de prototipagem, na qual é importante poder rapidamente analisar e testar desenhos diferentes, mas também na fase de produção, onde tornam possível introduzir atualizações no sistema ou repor um estado anterior quando este já se encontra instalado a desempenhar a sua função.

Avanços recentes desta tecnologia permitem reduzir significativamente o tempo necessário de modificação, recorrendo à reconfiguração de apenas uma parcela do sistema (reconfiguração parcial). Apesar desta possibilidade ser útil na fase de desenho e teste, é particularmente relevante quando o sistema já está em produção, uma vez que torna possível reduzir a indisponibilidade do sistema durante as atualizações. Além da reconfiguração parcial, a Xilinx introduziu recentemente a possibilidade de manter em execução a parcela do sistema que não esteja a ser reconfigurada durante o decorrer deste processo [1].

Esta funcionalidade de reconfiguração dinâmica parcial poderá ser de especial interesse para os sistemas de controlo. A reconfiguração total de um dispositivo FPGA impõe uma indisponibilidade significativa ao sistema durante a reconfiguração, ao passo que a reconfiguração parcial é efetuada numa fração desse tempo. Caso seja possível reduzir significativamente esse tempo de reconfiguração, poderá ser viável desenhar um sistema de controlo que permita atualizações dinâmicas parciais, i.e. sem ser necessário parar a execução.

Contudo, esta capacidade de atualizar ou refrescar um sistema de controlo baseado em tecnologia FPGA sem parar a sua execução, enfrenta diversos desafios, como a introdução de atrasos no sistema. O desafio principal consiste portanto em conseguir reduzir esse atraso ao mínimo indispensável, por forma a que o sistema controlado se mantenha num estado seguro, i.e. dentro do envelope de segurança das variáveis controladas.

Este artigo tem como objetivo investigar se é possível, com a tecnologia atual, reconfigurar um

sistema em execução num dispositivo FPGA sem impor uma indisponibilidade que coloque em causa a segurança do sistema controlado [2]. Por outras palavras, o objetivo desta investigação é compreender se a reconfiguração dinâmica parcial de dispositivos FPGA permite efetuar atualizações em sistemas de controlo sem requerer a sua paragem. Para tal, o presente artigo analisa os detalhes envolvidos na reconfiguração dinâmica parcial de dispositivos FPGA, compara os resultados, em termos de indisponibilidade, com a reconfiguração total (que representa a abordagem mais comum), e verifica o impacto que estes procedimentos poderão ter nos sistemas controlados.

O artigo está organizado da seguinte forma. Na Secção 2 descreve-se em detalhe o estado da arte na reconfiguração de dispositivos FPGA. Na Secção 3 apresentam-se as arquiteturas dos sistemas com os dois tipos de reconfiguração em estudo enquanto que na Secção 4 caracterizam-se os sistemas de controlo contínuo. Na secção 5 apresentam-se os testes efectuados e os resultados obtidos. O artigo encerra com uma síntese das principais conclusões e linhas de trabalho futuro.

2. Reconfiguração de Dispositivos FPGA

Um dispositivo FPGA baseado em SRAM é composto por uma matriz de blocos lógicos configuráveis, interligados através de elementos de encaminhamento, com possibilidades de armazenamento interno e mecanismos de entrada e saída. Cada dispositivo contém uma memória de configuração que consiste em memória SRAM contendo a lógica do circuito e o encaminhamento necessário. Sempre que o dispositivo é inicializado, a memória de configuração é programada através de um *bitstream*, uma sequência de bits com os dados de configuração do FPGA, habitualmente gerado por uma ferramenta de desenho de hardware e armazenado num dispositivo *flash*.

Na sua forma mais simples, a programação de um FPGA é levada a cabo através de uma reconfiguração total do circuito, i.e., uma escrita da totalidade da memória de configuração a partir do conteúdo da memória *flash*. No entanto, os fabricantes de dispositivos FPGA desenvolveram a possibilidade de reconfiguração parcial dos circuitos, permitindo reduzir significativamente o tempo necessário para a reprogramação, sempre que seja necessário atualizar apenas uma parte da memória de configuração.

A esta funcionalidade foi acrescentada a possibilidade de reconfigurar parcialmente um dispositivo FPGA *sem parar a sua execução*. Esta funcionalidade, denominada *reconfiguração parcial dinâmica*, permite reprogramar parcialmente a memória de configuração, enquanto outros módulos prosseguem a sua execução dentro do mesmo dispositivo FPGA [3], [4], [5]. A reconfiguração parcial dinâmica pode ser efetuada através de um porto externo bem como através de um porto interno (o ICAP – *Internal Configuration Access Port*) que possibilita a auto-reconfiguração de um circuito.

Ao permitir alterar a funcionalidade de um dispositivo FPGA durante a execução do sistema, a reconfiguração parcial dinâmica abre a possibilidade de atualizar bem como refrescar a lógica de um sistema de controlo sem que este pare a execução. Adicionalmente, torna-se possível reduzir o espaço ocupado no dispositivo FPGA nos casos em que nem todos os módulos tenham de executar em simultâneo, multiplexando no tempo o hardware, i.e. mantendo na sua memória apenas os módulos que estão em funcionamento [6].

Os desenvolvimentos que permitiram não só reconfigurar uma fração da memória de configuração, mas também fazê-lo *sem parar* a execução do sistema, criaram assim a possibilidade de atualizar ou refrescar um sistema de controlo durante a execução. No entanto, é necessário avaliar se os tempos envolvidos são suficientemente curtos para manter o sistema controlado num estado seguro e se é possível retomar a execução corretamente após a reconfiguração. Esta avaliação é o foco principal deste artigo.

3. Arquitetura do Sistema

Assumindo como objetivo principal a reconfiguração de um sistema de controlo baseado em FPGA, foram utilizadas duas abordagens consoante os meios utilizados para o efeito: *reconfiguração total* e *reconfiguração dinâmica parcial* [7]. Nesta secção descreve-se a arquitetura geral do sistema de controlo reconfigurável, com base num FPGA Virtex-5, da Xilinx [8].

Nas duas abordagens assume-se que o *bitstream* utilizado na reconfiguração se encontra armazenado num cartão de memória do tipo *Compact Flash* (CF).

3.1. Reconfiguração Total

A reconfiguração total do FPGA é realizada através de portos externos do dispositivo, normalmente por recurso a hardware adicional que possibilita a leitura do *bitstream* de uma memória *flash* e a reconfiguração do dispositivo. O processo é idêntico ao utilizado durante a fase de inicialização de dispositivos FPGA, baseados em SRAM.

Na **Fig. 1** é apresentada a arquitetura do sistema de reconfiguração total, onde podem observar-se os dispositivos externos, nomeadamente o controlador *System ACE CF Controller* e a memória *flash*, no formato CF.

O controlador *System ACE CF Controller* é responsável pela reconfiguração do sistema, i.e., leitura do *bitstream* da memória *flash*, e pela reconfiguração da totalidade da memória de configuração do FPGA, através do porto externo JTAG [9]. Apesar do *bitstream* do sistema de controlo ocupar uma pequena parte da memória de configuração, o tamanho do ficheiro gerado pelas ferramentas da Xilinx situa-se nos 3,8 MBytes, correspondente a toda a área do FPGA.

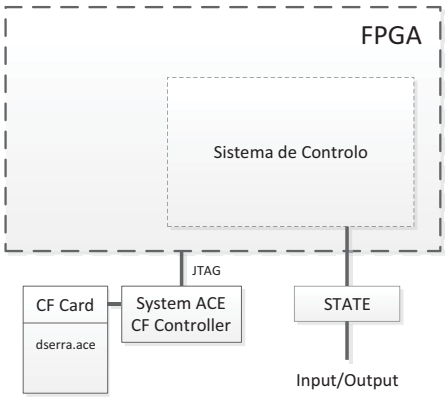


Fig. 1. Arquitetura do sistema de reconfiguração total com os dispositivos externos necessários.

Houve ainda necessidade de adicionar um módulo STATE, externo ao FPGA, responsável pela manutenção do estado do sistema de controlo durante a sua reconfiguração. Este módulo, contendo um registo de memória, retém o estado do controlador para que, quando este for reiniciado, possa recuperar a partir do estado que estava em execução.

3.2. Reconfiguração Dinâmica Parcial

Ao contrário da abordagem anterior, onde é obrigatória a utilização de dispositivos externos ao FPGA, na reconfiguração dinâmica parcial é possível colocar os módulos de reconfiguração e de manutenção do estado do sistema de controlo dentro do FPGA, através da utilização de componentes estáticos.

O sistema utilizado neste estudo é baseado num microprocessador (µP) *Microblaze* [10], [11] que comunica com um conjunto de módulos através de um bus local ao processador (PLB) – ver **Fig. 2**.

O µP utiliza 64Kbytes de memória RAM para dados e instruções, implementada através de um conjunto de blocos de RAM (BRAM) do FPGA, e acessível através de um *bus* de acesso a memória local (LMB). Em execução no µP encontra-se um programa monitor, controlado através da porta série (RS232), que permite a reconfiguração do sistema de controlo com um *bitstream* carregado do CF, o controlo do estado interno do módulo de controlo, e ainda o acesso ao estado das entradas e saídas do módulo de controlo.

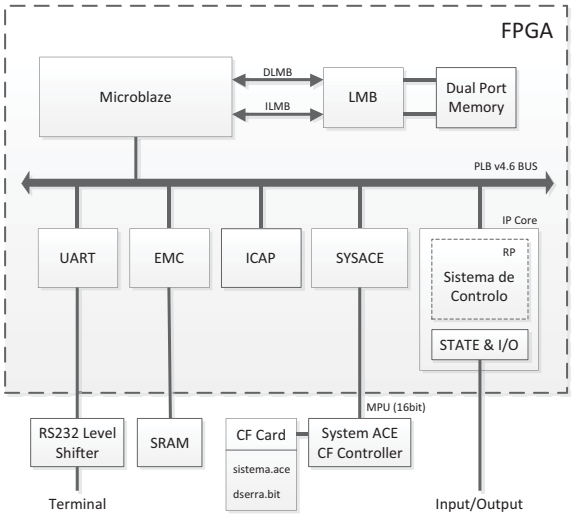


Fig. 2. Arquitetura do sistema de reconfiguração dinâmica parcial, interno ao FPGA.

Foram também utilizados alguns módulos proprietários (*IP Cores – Intellectual Property Cores*), com interface PLB, fornecidos pela Xilinx [12], para facilitar a depuração e o acesso a dispositivos internos e externos necessários à reconfiguração, tais como: um módulo universal de comunicação assíncrona (UART) para interface entre o utilizador e a aplicação que se encontra em execução no µP (através de um terminal); um módulo de acesso a ficheiros em cartões de memória CF (SYSACE) para leitura do *bitstream* do sistema de controlo; um controlador de memórias externas (EMC) para acesso a um bloco de SRAM utilizado como *cache* para o *bitstream* parcial; e um módulo ICAP.

Ao sistema base foi então adicionado um módulo IP proprietário, com interface PLB, que inclui o sistema de controlo e lógica adicional que possibilita a manutenção do estado durante a reconfiguração. O sistema de controlo encontra-se numa partição reconfigurável (RP) ao passo que o seu estado e o controlo das entradas e saídas [13], [14] estão numa zona estática (STATE & I/O).

De entre os vários dispositivos externos acoplados ao FPGA, apenas o dispositivo de memória externa *flash* é essencial ao correto funcionamento dos FPGA baseadas em SRAM, devido à necessidade de configuração inicial do dispositivo.

A utilização de um dispositivo de SRAM externo, como cache para o *bitstream* parcial,

permite reduzir os tempos de reconfiguração em uma ordem de grandeza, uma vez que o acesso ao dispositivo *Compact Flash* é bastante mais lento. Durante o arranque do sistema, o *Microblaze* é responsável pela leitura dos *bitstreams* parciais do CF e seu armazenamento na SRAM. Desta forma, a duração da reconfiguração parcial é substancialmente reduzida, uma vez que o *bitstream* a ser escrito na memória de configuração do FPGA é lido da SRAM, com tempos de acesso mais rápidos que os dos dispositivos baseados em memória flash (CF).

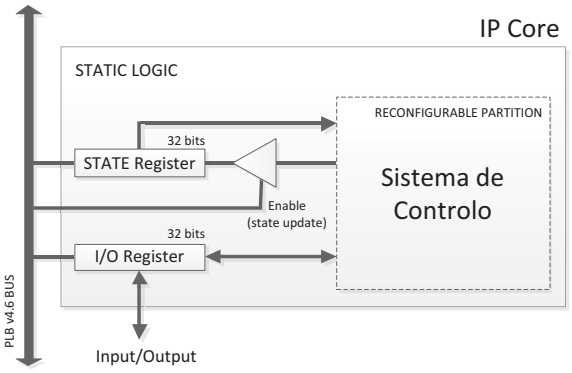


Fig. 3. Estrutura interna do módulo IP desenvolvido.

O módulo IP Core desenvolvido (ver **Fig. 3**) está divido em duas partes: uma componente estática (STATIC LOGIC) e uma partição reconfigurável (RP – *Reconfigurable Partition*) [15], [16]. Na partição reconfigurável encontra-se o sistema de controlo, que efetua a leitura periódica dos sensores que adquirem o estado atual do sistema físico, calcula o estado seguinte do controlador e as ações de controlo, e aciona as saídas ligadas aos atuadores. Para preservar o estado do sistema de controlo durante o período de reconfiguração, este deve encontrar-se localizado externamente à partição reconfigurável, sendo a sua atualização inibida durante esse tempo. Assim, na componente estática do módulo encontram-se os registos de estado (STATE *Register*) e de acesso às entradas e saídas (I/O *Register*), que permitem despoletar a reconfiguração através de um sinal externo e o interface com o sistema a controlar.

A configuração apresentada permite ao µP aceder ao estado do sistema de controlo, através do mapeamento do registo de estado no seu espaço de endereçamento. É também possível aceder, da mesma forma, ao valor das entradas e saídas do sistema de controlo, para efeitos de monitorização, através do registo de entrada e saída.

A reconfiguração da partição reconfigurável pode ser despoletada de duas formas: através de um sinal externo (designado por Input/Output na **Fig. 2**) ou por iniciativa do µP. Em qualquer das situações, é o µP que faz a leitura do *bitstream* parcial, armazenado no CF, e envia a informação ao módulo ICAP, responsável pela reconfiguração dinâmica parcial do FPGA.

Na reconfiguração parcial são necessários dois ficheiros com *bitstreams*: um contendo apenas os dados da partição reconfigurável e outro que inclui os dados de todo o FPGA, i.e., todo o sistema embutido descrito em cima, incluindo a partição reconfigurável [17], [18]. Aquando do arranque do sistema, o controlador *System ACE CF Controller* lê o *bitstream* total do CF e configura todo o FPGA. Quando já se encontra em execução e há necessidade de atualizar ou repor um determinado sistema de controlo, o *bitstream* parcial é lido do CF pelo µP, através do módulo SYSACE e do controlador *System ACE CF Controller*, e é escrito na partição reconfigurável do IP Core proprietário, através do módulo ICAP.

3.3. Análise Preliminar

Na reconfiguração dinâmica parcial, como apenas se reconfigura parte da memória de configuração do dispositivo, torna-se possível reduzir o tempo de reconfiguração, não só pelo facto do tamanho do *bitstream* ser menor, mas porque utilizamos o porto de reconfiguração interno, ICAP, que funciona ao dobro da velocidade do porto externo mais rápido.

Outra das vantagens da reconfiguração dinâmica parcial é a possibilidade de utilização de componentes estáticos, não reconfiguráveis, que se mantêm permanentemente em execução no dispositivo. Isto possibilita a construção de SoC (*System on a Chip*), onde componentes essenciais como sejam os módulos de reconfiguração, de armazenamento do estado e controlo das entradas e saídas, são internos ao FPGA e permanecem em execução durante o período de reconfiguração. A utilização de módulos externos, como foi o caso do módulo STATE criado para a reconfiguração total, é assim evitada.

4. Sistemas de Controlo Contínuo

Os sistemas de controlo contínuo são normalmente constituídos por um controlador (um computador digital) que monitoriza periodicamente o processo controlado (o sistema cuja atividade se pretende regular), compara as suas saídas com valores de referência, calcula as ações de controlo adequadas e atua sobre o processo de modo a que este cumpra as suas funções da forma esperada.

A monitorização do processo controlado é realizada periodicamente devido ao facto de, tratando-se de processos contínuos, a discretização

do estado do processo para cálculo das ações de controlo ficar desatualizada com a passagem do tempo, sendo por isso necessário refrescá-las dentro de um determinado período de tempo. A duração deste período de amostragem deve ser tal que a amostra do estado do processo adquirida pelo controlador não fique demasiado desatualizada. É fácil assim compreender que tal depende da dinâmica dos processos controlados. O cálculo do período de amostragem é usualmente efetuado de uma forma conservadora (*over-sampling*), usando regras empíricas (é comum adotarem-se valores 10 vezes inferiores à constante de tempo mais rápida do processo [19], [20]). Valores típicos para períodos de amostragem situam-se entre alguns milissegundos e algumas centenas de milissegundos.

Dado que a frequência de amostragem assume valores mais elevados do que seria estritamente necessário, a perda esporádica de uma amostra é impercetível no comportamento do processo controlado, sendo implicitamente tratada como qualquer perturbação externa que possa afetar o próprio sistema físico. Com efeito, uma vez que estes sistemas executam dentro de um ciclo fechado no qual o algoritmo de controlo é realimentado pelas saídas dos processos controlados, perturbações que afetem o processo controlado, tais como atritos, deslocações de ar, ou alterações da temperatura são consideradas como normais e tidas em conta pelo algoritmo de controlo. Falhas esporádicas nas ações de controlo são, de igual forma, toleradas pelo sistema. Tal foi observado em projetos realizados no passado com processos reais [21], [22].

Pelo acima exposto, nas aplicações práticas é assumido que a inatividade do controlador durante *um* período de amostragem, não tem qualquer efeito no sistema controlado, isto é, na qualidade do serviço fornecido pelo controlador. No presente estudo utilizamos esta observação como majorante do tempo máximo de reconfiguração do FPGA do controlador. De assinalar que a *state-of-practice* aceita valores bem mais elevados, de duas, três ou mais omissões de controlo, desde que esporádicas, i.e. que ocorram depois do último *glitch* de controlo ter sido completamente absorvido pelo sistema.

5. Testes e Resultados

Para analisar a viabilidade da reconfiguração dinâmica de um sistema de controlo baseado num dispositivo FPGA é necessário analisar dois fatores: o tempo durante o qual o sistema controlado se encontra sem controlo, e o correto retomar da execução pelo controlador. De salientar que durante a reconfiguração parcial do sistema de controlo a saída do controlador mantém o último valor gerado. Desta forma, não haverá lugar a atrasos na ação de controlo (extensão do período de amostragem), mas sim omissão em caso de demora na reconfiguração do controlador (a saída do controlador será atualizada num instante de amostragem posterior).

Com este intuito, ao invés de utilizar um algoritmo de controlo de alguma forma condicionado a uma aplicação específica utilizámos um sistema onde o tempo de inatividade fosse medido com uma elevada precisão e a correta continuidade da operação fosse facilmente percetível: um gerador de sinal em dente de serra.

5.1. Descrição do Dente de Serra

O gerador de dente de serra executa a uma frequência de 100MHz e resolução de 32 bits (ver **Fig. 4**), gerando assim 2^32 diferentes tensões de saída a intervalos de 10 nanossegundos. Desta forma é possível medir experimentalmente o tempo necessário à atualização do sistema de controlo pelo desfasamento dos sinais gerados, antes e depois da reconfiguração.

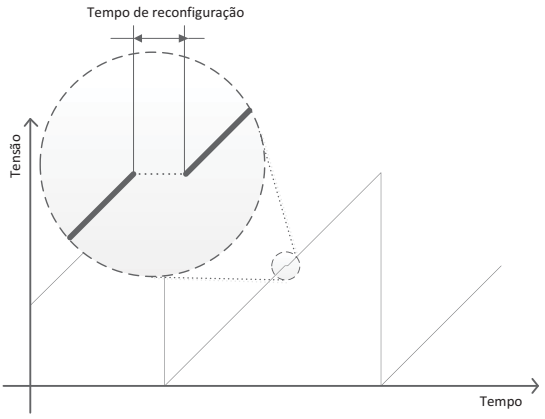


Fig. 4. Pormenor do sinal de saída do sistema de controlo – gerador de dente de serra, durante a reconfiguração dinâmica.

5.2. Descrição das experiências

As experiências foram levadas a cabo numa placa de desenvolvimento Digilent XUPV5, com um FPGA Xilinx Virtex-5 XC5VLX110T [23]. Foram utilizados dispositivos de entrada e saída (leds e interruptores, como entradas e saídas do sistema de controlo), um conversor UART de RS232 para TTL (para interface com o programa monitor em execução no μP), um dispositivo de memória CF (para armazenamento dos *bitstreams*) e uma memória SRAM (como dispositivo de cache para optimização dos acessos ao CF).

Para as medições foi utilizado um analisador lógico Hewlett Packard 1651B, com precisão máxima de 10ns, correspondente ao período de

relógio do microprocessador com frequência de 100MHz.

Nos testes ao sistema de reconfiguração total foram medidos os tempos desde a asserção do sinal de SYSACE RST do controlador *System ACE CF Controller* (que provoca a reconfiguração total da memória de configuração do FPGA com o *bitstream* do sistema de controlo, contido no cartão CF), até à deteção do início do dente de serra nas saídas do sistema de controlo.

Nos testes ao sistema de reconfiguração dinâmica parcial, com o microprocessador e *bus* PLB a funcionarem à frequência de 100MHz, foram medidos os períodos de tempo em que a saída do sistema de controlo ficou estacionária. Este tempo corresponde ao tempo máximo necessário à reconfiguração do sistema de controlo e inclui: a desativação da atualização do estado pelo sistema de controlo, a reconfiguração da partição reconfigurável e a ativação da atualização do estado.

Com o intuito de perceber a correlação entre a dimensão do *bitstream* e o tempo de reconfiguração, foram construídos mais 3 sistemas de controlo com a mesma funcionalidade, mas com dimensões da partição reconfigurável 5, 10 e 20 vezes superiores às do sistema original, gerando os *bitstream* descritos na secção seguinte (ver **Tabela 1**).

5.3. Resultados

Na **Tabela 1** são apresentados os resultados das reconfigurações total (RT) e dinâmica parcial (RDP) de um sistema de controlo com diferentes necessidades em termos de recursos lógicos do FPGA, e portanto na dimensão do ficheiro *bitstream* associado.

Exp.	Tamanho	RT	RDP	
	bitstream	*Flash*	*Flash*	SRAM
1	30.256 bytes	1.380ms	74,4ms	9,6ms
2	72.768 bytes	---	168ms	24ms
3	119.000 bytes	---	270ms	40ms
4	178.224 bytes	---	408ms	60ms

Tabela 1. Resultados das experiências de Reconfiguração Total (RT) e de Reconfiguração Dinâmica Parcial (RDP) de um sistema de controlo num FPGA, em função do tamanho da Partição Reconfigurável (PR).

Estes resultados mostram que, para a primeira experiência, a reconfiguração total demorou 1380ms, enquanto a reconfiguração parcial foi realizada em 74,4ms tendo o *bitstream* sido carregado a partir de *flash*, e 9,6ms quando carregado a partir de SRAM. Por um lado, verifica-se que existe um *overhead* no acesso ao cartão CF, externo ao FPGA, que não acontece na reconfiguração parcial a partir de SRAM. Por outro lado, na reconfiguração total o *bitstream* corresponde a dados de reconfiguração de todo o FPGA, e não apenas da componente reconfigurável.

Esta é a razão pela qual, na coluna associada à reconfiguração total, apenas se apresenta o resultado para o *bitstream* original, dado que as ferramentas da Xilinx utilizadas geram *bitstreams* com a mesma dimensão, qualquer que seja o tamanho do código VHDL associado ao sistema de controlo.

Para a reconfiguração dinâmica parcial obtiveram-se tamanhos diferentes do ficheiro *bitstream*, correspondentes a diferentes tamanhos da partição reconfigurável.

Numa primeira análise aos tempos medidos, com recurso a um analisador lógico, podemos verificar, grosso modo, que o tempo necessário para a reconfiguração dinâmica parcial do FPGA a partir do *bitstream* em SRAM, em milissegundos, equivale a cerca de 1/3 do seu tamanho, em Kbytes. Esta correlação deverá ser analisada com mais detalhe, a partir de dados de mais experiências.

Podemos, contudo, verificar que existe uma correspondência entre o tamanho da partição reconfigurável, diretamente relacionado com a lógica necessária à construção do sistema de controlo, e o tempo de reconfiguração necessário.

6. Conclusões

Neste artigo apresentámos os resultados experimentais da utilização de reconfiguração parcial dinâmica em FPGA no controlo de aplicações físicas. O objetivo foi avaliar a possibilidade de reconfigurar o dispositivo FPGA dinamicamente, em *run-time*, sem perder o controlo do processo controlado, o que ficou comprovado para cenários muito realistas. Para efeitos de precisão na medição do sinal de controlo analógico das saídas, utilizámos um gerador de onda em dente de serra, pois qualquer desvio é facilmente observável e mensurável com um analisador lógico. Comparámos dois sistemas, um utilizando reconfiguração total do FPGA e outro utilizando reconfiguração parcial apenas da zona do FPGA onde estava a lógica de geração dos outputs. Utilizámos ainda diferentes tamanhos para a partição reconfigurável, e localização do *bitstream* para reconfiguração do FPGA, em *flash* e pré-carregado em SRAM.

Uma primeira análise permitiu confirmar uma diferença de 2 ordens de grandeza entre os tempos de reconfiguração parcial e total do FPGA, pelo simples facto da área a reconfigurar poder ser substancialmente inferior. Por outro lado, pudemos verificar que existe uma grande vantagem em carregar previamente o *bitstream* para SRAM antes de efetuar a reconfiguração parcial, eliminando o

overhead no acesso ao cartão CF, diminuindo o tempo total em cerca de 7 vezes.

A partir destes resultados experimentais, verificámos ainda que o tempo de reconfiguração parcial do FPGA a partir de um *bitstream* situado em SRAM, em milissegundos, corresponde a cerca de 1/3 do tamanho do *bitstream* em KBytes. Assim é possível, a partir da constante de tempo do processo controlado, calcular a área máxima que é possível disponibilizar dentro da partição reconfigurável do FPGA para a lógica do controlador. Por exemplo, um sistema que tenha uma constante de tempo de 100ms (a esmagadora maioria dos processos industriais tem constantes de tempo bastante superiores) tolera um tempo de reconfiguração de 10ms, possibilitando assim que a lógica do sistema de controlo ocupe um *bitstream* de cerca de 30Kbytes.

Com o intuito de aplicar esta tecnologia a um sistema real, encontra-se em desenvolvimento um controlador para um sistema de *cruise control* com possibilidade de atualização em funcionamento. Este sistema é composto por um motor DC, um *encoder* óptico de elevada resolução e um controlador PID reconfigurável. Atendendo ao facto de grande parte dos sistemas de controlo serem modulares, também se irá explorar a possibilidade de reconfiguração por módulos, permitindo manter os tempos de reconfiguração em valores aceitáveis para controladores mais complexos.

No seguimento deste trabalho e com o intuito de alargar a sua aplicação a um maior número de sistemas de controlo, com requisitos temporais mais estritos, optar-se-á por acrescentar um sistema redundante em espera ativa (passando a utilizar duas partições reconfiguráveis) com o objetivo não só de assegurar uma rápida transferência para o novo sistema de controlo, mas também para validar *à priori* o seu correto funcionamento.

Referências

[1] Donthi, S., Haggard, R.L.: A survey of dynamic reconfigurable FPGA devices. Proceedings of the 35th Southeastern Symposium on System Theory, pp. 422–426 (2003).

[2] Lanuza, M., Zicari P., Frustaci F., Perri S., Corsonello P.: Exploiting self-reconfiguration Capability to Improve SRAM-based FPGA robustness in space and avionics Applications. ACM Transactions on Reconfigurable Technology and Systems, Vol. 4, No. 1, Article 8 (2010).

[3] Horta, E., Lockwood, J. W., Parlour, D.: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. Proceedings of the 39th Design Automation Conference, pp. 343–348 (2002).

[4] Hubner M., Becker, J.: Exploiting dynamic and partial reconfiguration for FPGAs - toolflow, architecture and system integration. Proceedings of the 19th annual Symposium on Integrated Circuits and Systems Design, pp. 1–4 (2006).

[5] Bolchini, C., Quarta, D., Santambrogio, M.D.: SEU mitigation for SRAM-based FPGAs through dynamic partial reconfiguration. Proceedings of the 17th ACM Great Lakes symposium on VLSI, pp. 1–6 (2007).

[6] Upegui, A., Sanchez, E.: Evolving hardware by dynamically reconfiguring Xilinx FPGAs. Evolvable Systems: From Biology to Hardware, vol. 3637/2005, pp. 56–65 (2005).

[7] Xilinx: Virtex-5 FPGA Configuration User Guide v3.10 (UG191). User guide, Xilinx Inc. (2011).

[8] Xilinx: Virtex-5 Family Overview (DS100). Datasheet, Xilinx Inc. (2009).

[9] Xilinx: System ACE CompactFlash Solution (DS080). Datasheet, Xilinx Inc. (2008).

[10] Xilinx: LogiCORE IP MicroBlaze Micro Controller System v1.0 Product Specification (DS865). Datasheet, Xilinx Inc. (2012).

[11] Xilinx: MicroBlaze Processor Reference Guide – Embedded Development Kit (UG081). User guide, Xilinx Inc. (2012).

[12] Xilinx: EDK Concepts, Tools, and Techniques – A Hands-On Guide to Effective Embedded System Design. User guide, Xilinx Inc. (2012).

[13] Xilinx: PlanAhead Software Tutorial – I/O Pin Planning (UG674). User guide, Xilinx Inc. (2010).

[14] Xilinx: Virtex-5 FPGA Packaging and Pinout Specification v4.8 (UG195). User guide, Xilinx Inc. (2010).

[15] Xilinx: Xilinx Partial Reconfiguration Flow. Presentation manual, Xilinx University Program.

[16] Dye, D.: Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite v1.1 (WP374), White paper, Xilinx Inc. (2011).

[17] Xilinx: Partial Reconfiguration of a Processor Peripheral Tutorial – PlanAhead Design Tool (UG744). User guide, Xilinx Inc. (2012).

[18] Xilinx: PlanAhead Software Tutorial – Overview of the Partial Reconfiguration Flow (UG743), User guide, Xilinx Inc. (2010).

[19] Bennett, S.: Real-Time Computer Control: An Introduction, Prentice Hall international series in systems and control engineering, 2 ed. Prentice Hall, Nova Iorque, EUA (1994).

[20] Levine, W.S.: The Control Handbook. CRC Press, New York, ISBN 978-0-8493-8570-4 (1996).

[21] Cunha, J.C., Maia, R., Rela, M.Z., Silva, J.G.: A Study on Failure Models in Feedback Control Systems. In: International Conference on Dependable Systems and Networks, Goteborg, Sweden (2001).

[22] Cunha, J.C., Rela, M.Z., Silva, J.G.: On the Use of Disaster Prediction for Failure-Tolerance in Feedback Control Systems. In: International Conference on Dependable Systems and Networks, pp. 123–132. IEEE Computer Society Press, Washington, D.C., USA (2002).

[23] Xilinx: Virtex-5 FPGA User Guide (UG190). User guide, Xilinx Inc. (2010).

 ISBN: 978-972-8822-27-9

FPGA Implementation of Autonomous Navigation Algorithm with Dynamic Adaptation of Quality of Service

José Carlos Sá
Faculdade de Engenharia
Universidade do Porto
ee06028@fe.up.pt

João Canas Ferreira
INESC TEC and
Faculdade de Engenharia
Universidade do Porto
jcf@fe.up.pt

José Carlos Alves
INESC TEC and
Faculdade de Engenharia
Universidade do Porto
jca@fe.up.pt

Abstract

The main goal of this work is to build an hardware-aided autonomous navigation system based on real-time stereo images and to study Partial Reconfiguration aspects applied to the system. The system is built on an reconfigurable embedded development platform consisting of an IBM PowerPC 440 processor embedded in a Xilinx Virtex-5 FPGA to accelerate the most critical task. Three Reconfigurable Units were incorporated in the designed system architecture. The dynamic adjustment of system's quality of service was achieved by using different reconfiguration strategies to match vehicle speed. A speedup of 2 times for the critical task was obtained, compared with a software-only version. For larger images, the same implementation would achieve an estimated speedup of 2.5 times.

1. Introduction

The performance expected from complex real-time embedded systems has been increasing more and more. The application presented in this work is a good example of such a system. It uses a complex autonomous navigation algorithm for guiding a small robot using information obtained from a pair of cameras. Since the computational effort is significant for an embedded system, a previous software/hardware partitioning step identified the critical task and proposed its implementation in hardware.

Modern platform FPGAs like the Virtex-5 (from Xilinx) can combine the flexibility of software running on an embedded processor (a PowerPC processor in this case) with the performance of dedicated hardware support. The main objectives of the work described here are to evaluate the application of dynamic partial reconfiguration (DPR) [1] by designing and building and assessing a prototype. This subject is one of the case studies proposed by the European consortium REFLECT project, which also provided the original application software written in C language.

The paper is organized as follows: Section 2 briefly provides some background information, while Sect. 3 gives an overview on the application under study. Section 4 describes the structure of the critical task to be accelerated. The system and strategies used for the evaluation of DPR are described in Sect. 5, while Sect. 6 discusses the results. Section 7 concludes the paper.

2. Background

Reconfigurable systems combine two main key concepts in embedded technology: Parallelism, which is the biggest advantage of hardware computing, as provided by Application-Specific Integrated Circuits (ASICs), and flexibility, the main reason for the success of software applications run by General Propose Processors (GPPs).

The concept of reconfigurable system is breaking the barrier between these two types of devices. This allowed the combination of the particularities of both and elevate the concept of embedded system to a new level, creating new types of compromise between the software and the hardware infrastructure. A reconfigurable embedded system allows portions of the system's hardware to be modified, thereby changing system's hardware-accelerated features depending on the need of the application executed by the GPP. This feature offers a new degree of flexibility to the embedded system, where the execution of more tasks in hardware provides a multi-level acceleration that is difficult to obtain by a fixed-functionality circuit.

The state of the art reconfiguration technology, the Dynamic Partial Reconfiguration, allows FPGA regions to be changed at run-time, without interrupting the GPP, or rest of the hardware execution. Fig. 1 shows an example of a partial reconfiguration system, where the hardware part of the system is composed of four modules that execute HW-accelerated tasks. In this example, two of them are remain fixed, executing functions that requires permanent availability, whereas the remaining blocks are reconfigurable modules. For the example, function C can the exchanged with function D, since the have been designed to use the same reconfigurable region; the same applies to functions E and F, but for a different reconfigurable region.

The compliance with this technology has some costs. For instance, the design flow for reconfigurable systems followed by the Xilinx EDK[1] [2] requires some effort to design compatible interfaces between reconfigurable modules for a particular region.

The major drawback of dynamic reconfiguration is the

[1] EDK - Embedded Development Kit.

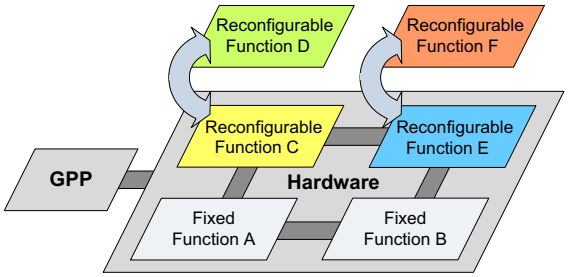


Figure 1. Dynamic Partial Reconfiguration System.

reconfiguration time, which directly depends on partial bitstream[2] data transfer time. Since the bitstream size is proportional to the FPGA area to be configured, it is important to keep it as small as possible for minimum impact on the system's performance. Other aspects like task scheduling or reconfigurable strategies have to be taken into consideration when using DPR.

An analysis of the use cases applicable to assembly of a reconfigurable system for implementation of a Software Defined Radio is made by [3], where several reconfiguration strategies are evaluated. The study shows that total flexibility of reconfiguration is achieved with a reasonable implementation complexity.

In [4] dynamic reconfiguration is applied to two different types of navigation systems, and an analysis of the use cases is also made. Several techniques dealing with dynamic reconfiguration of software for robots are discussed and implemented. The results shows that the system efficiency is very intertwined with the techniques of interaction between processing objects.

3. Embedded Application

The Stereo Navigation application used in this work supports localization mechanisms like Global Navigation Satellite Systems (GNSS) in vehicles where this service is temporarily unavailable. Two cameras pointed in the same direction capture the scenery ahead of the vehicle at the same instant, forming a stereo image. From the processing of stereo images in consecutive instants, the embedded system can determine vehicle rotation and translation matrices.

The main loop of the Stereo Navigation algorithm comprises the following steps:

Image rectification Eliminates image distortions caused by the lens surface.

Feature extraction Detects characteristics of a given image using the Harris Corner Detector algorithm [5].

Feature matching Searches correspondences between the features of the stereo image belonging to consecutive instants.

[2]Bitstream - Stream of data containing information of FPGA internal logic cells end routing configuration.

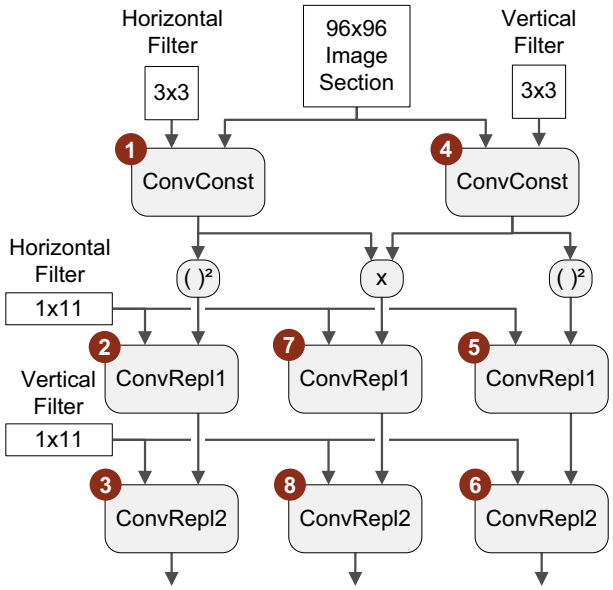


Figure 2. Operation Flow for Harris Corner Detector Algorithm

3D Reprojection Analytically calculates the three-dimensional coordinates of a point from two-dimensional images.

Robust pose estimation - Runs the RANSAC algorithm [6] to separate relevant inlier image features from outliers. In each loop iteration a Singular Vector Decomposition (SVD) algorithm computes a vehicle rotation matrix and a translation vector.

4. Critical Task: Feature Extraction

From the analysis of the run-time behavior of the application, combined with the information provided in previous works [7, 8], it follows that feature extraction is the most time consuming processing task. This task is responsible for detecting relevant image features using the Harris Corner Detector algorithm and is executed once for each 96×96 pixel block of both left and right images. Twelve blocks are processed per image, 24 for the stereo image.

4.1. Computation Flow

As shown in Fig. 2, each execution of the algorithm comprises eight 2-D convolution operations: $2 \times$ ConvConst, $3 \times$ ConvRepl1 and $3 \times$ ConvRepl2.

All the convolution operations are based on Eq. 1, where the result of multiply-accumulate (MAC) operations between matrix filter h (size 3×3) and each input element present in array u (of size 96×96) is stored in the output matrix y (also of size 96×96).

$$acc_n = u[i] \times h[j] + acc_{n-1} \quad (1)$$

 ISBN: 978-972-8822-27-9

ConvConst This procedure convolves the frame with a 3×3 horizontal (Eq. 2) and a vertical (Eq. 3) Prewitt filter. This function receives and produces only integer values.

$$H = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad (2)$$

$$V = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad (3)$$

ConvRepl1 This procedure computes a convolution between the result of ConvConst and a 1×11 horizontal Gaussian filter. This function receives only integer values, but produces single-precision floating-point values.

ConvRepl2 The procedure computes a convolution between the result from ConvRepl1 and a 11×1 vertical Gaussian filter. This function receives and produces only floating-point values.

4.2. Hardware Implementation

To quickly implement each critical subtask in hardware, we used the academic version of the high-level synthesis tool Catapult C, which synthesizes C code to RTL-level descriptions (in Verilog). However, this tool is unable to generate hardware blocks for floating-point arithmetic (only fixed-point versions can be synthesized, but no data range analysis is performed).

To solve this issue, two possible solutions were considered: synthesizing SoftFloat routines [9], or usinh a fixed-point equivalent version. Both alternatives employ 32-bit data types. The first one is a faithful software implementation of the floating-point data-type defined by the IEEE-754 standard [10]. Its hardware implementation running at 100 MHz resulted in an execution time improvement for the *ConvRepl*1 and *ConvRepl*2 functions of just 8 % when compared to the original versions in software (running on the embedded PowerPC with a Floating-Point Unit (FPU)).

The second solution is a specially crafted design, which combines hardware and software fixed-point adjustments to minimize precision loss. Data validation of this approach was carried out for the worst case scenario. Running at the same frequency of 100 MHz, this solution resulted in a execution time improvement by a factor of 4 for each of the three functions, as shown on Fig. 3. This solution was adopted for the final implementation.

The hardware implementation process comprised several software tool flows and methodologies. After using Catapult C to create the Verilog files, these were synthesized using XST (from Xilinx). The first step involves the synthesis of the system's static part, which includes user peripherals with black-box modules that match the interface of the reconfigurable modules (RM). (This step is carried out with Xilinx XPS). The other step involves the the individual synthesis of the RMs directly with Xilinx ISE.

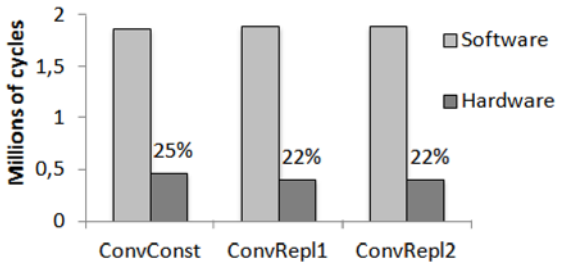


Figure 3. Critical task hardware acceleration (hand-crafted solution)

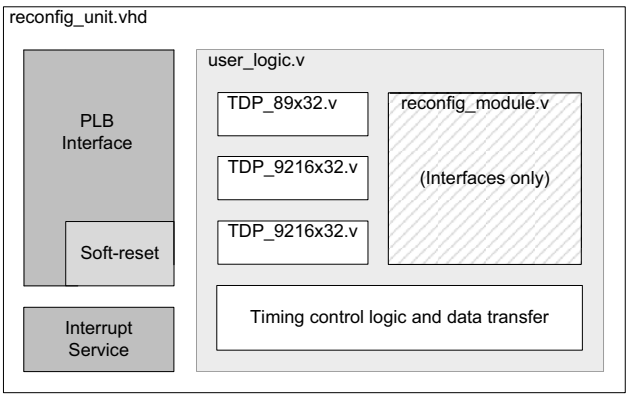


Figure 4. Reconfigurable unit peripheral

The synthesis steps produce gate-level netlists in NGC format (a Xilinx native format) for the static part of the design and each reconfigurable module, respectively. Finally the hardware implementation is completed with Xilinx PlanAhead tool, which is used for the physical definition of the reconfigurable areas and to manage the physical synthesis process. Each hardware accelerator must be undergo physical synthesis for each reconfigurable area where it may be used.

5. Reconfigurable System Design

5.1. Reconfigurable Unit

The Reconfigurable Unit is the peripheral designed to provide to the system the ability to change hardware functionality at run-time. It was created to host any of the modules created by Catapult C. The Reconfigurable Unit (RU) shown in Fig. 4 includes Block RAMs memory for U, H and Y matrix storage, processor local bus (PLB) connectivity, support for services such as system interrupt and burst data transfer, and the area for one Reconfigurable Module (RM).

5.2. System Architecture

A reconfigurable embedded system was designed with the architecture shown in Fig. 5. In addition to the processor and system RAM memory, it also contains: FPU for software acceleration; SysAce controller to load bitstreams

Table 1. Resource usage for reconfigurable modules

Function	LUTs (%)	CLBs (%)	FF-D (%)	DSP48E (%)
ConvConst	2.77	2.77	2.58	2.34
ConvRepl1	1.77	1.95	1.94	3.12
ConvRepl2	2.59	2.60	2.17	3.12

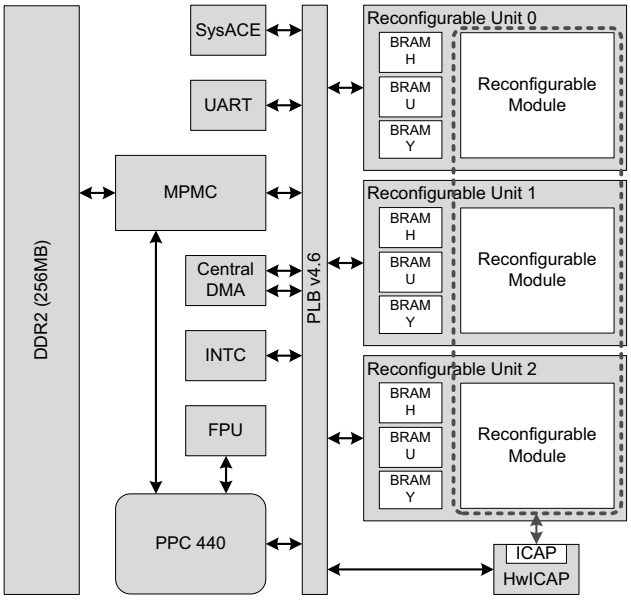


Figure 5. Implemented System Architecture.

from CompactFlash card; UART controller for terminal access; Central DMA to handle data transfer tasks; System Interrupt controller for event notification; HWICAP controller for partial bitstream download; three reconfigurable units, each with one RM. Each RM area uses 13 Virtex-5 frames of a single clock region. The total bistream size is about 74 KB. The resource usage for one RM is presented in Tab. 1.

5.3. Reconfiguration Strategies

Given the number of RUs in the system, three types of reconfiguration strategies where considered, each using one, two or three RUs, and associated with different speed requirement.

One RU For this strategy, the system uses just one RU. Each RM has to be sequentially configured with one of the accelerators as shown in Fig. 6. White blocks correspond to the execution of each convolution task, while the darker blocks correspond to the reconfiguration operations. The arrows indicate data flow dependencies.

Two RUs In this case, the system uses two RUs in a ping-pong fashion, as presented inn Fig. 7. When an accelerator is executing in one of the RUs, the other RU is simultaneously being configured with another accelerator.

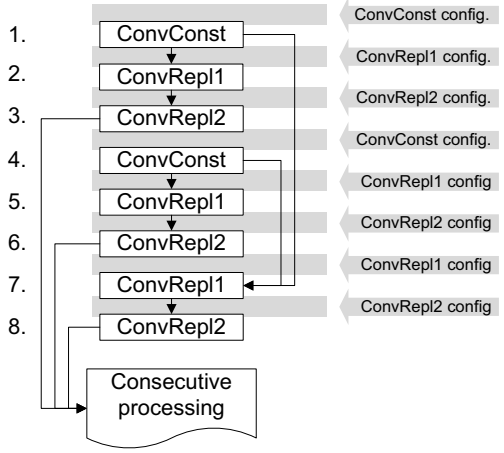


Figure 6. Single RU strategy.

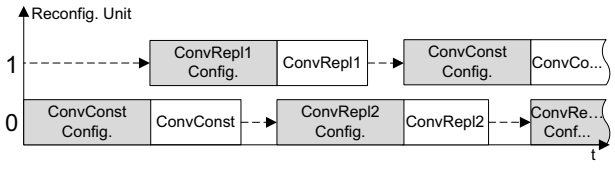


Figure 7. Ping-pong strategy (two RUs)

Three RUs Employing this strategy, each subtask (accelerator) is configured once in each individual RU, then executed in a sequential fixed fashion. This strategy has the ability to dramatically reduce the number of reconfigurations, as shown on Fig. 8. After feature extraction is complete, the RUs can be reused for other purposes, if necessary.

Three RUs – Pipelined Execution This is a pipelined version of High Speed strategy. Fig. 9 shows that this technique is able to reduce the number of steps required for task execution from 8 to 5. Note that the convolutions results data flow is the same as in the original version shown in Fig. 2.

6. Discussion and Analysis

The collected measurements show that each HW reconfiguration operation takes 5.37 ms, 4.5 times longer than any executed function module. Due to this fact, only strategies using three RUs could produce overall time execution benefits, as shown on Fig. 10. This figure presents the rela-

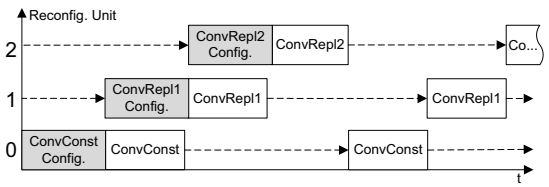


Figure 8. Scheduling with three RUs.

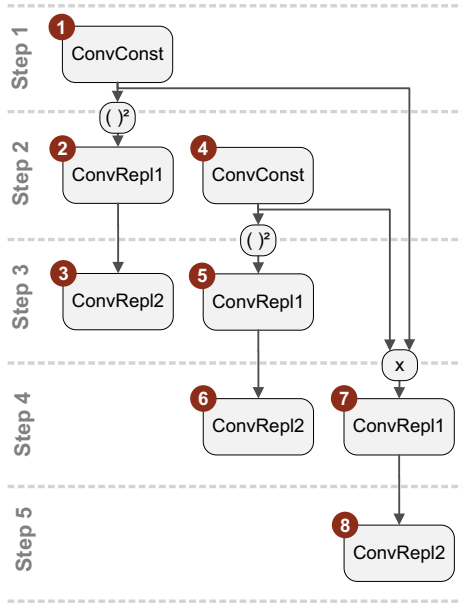


Figure 9. Task-level pipelining with three RUs

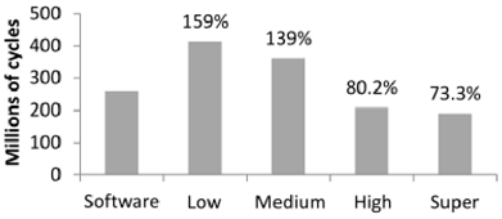


Figure 10. Number of clock cycles (relative to software execution)

tive execution time of Feature Extraction task for each implemented strategy as compared to the original software execution time. Time measurements were obtained by calculating the difference between time stamps created right before and after running the feature extraction task. The time stamps are provided by an internal timer/counter (T/C) device, which is incremented on every processor clock tick (2.5 ns).

Each read/write operation of 9216 32-bit words from/to the U/Y BRAMs takes 0.52 ms. For the fastest configuration, a maximum speed-up of 2 times was measured for the feature extraction task.

Fig. 11 shows the global frame rate improvements considering the full processing time of each pair of stereo images. Not all the strategies can execute faster than the original software. In fact, only strategies that use a fixed version of each hardware module can accelerate the application execution. The same figure shows that the software version can process 0.5 stereo frames per second (SFPS), i.e., one stereo image per two seconds. For each DPR strategy, the table also shows the SFPS rate and the relative improvement versus the original software version. The best DPR strategy results show that the image rate can be increased to 0.6 SFPS.

Table 2 shows in more detail the best obtained results

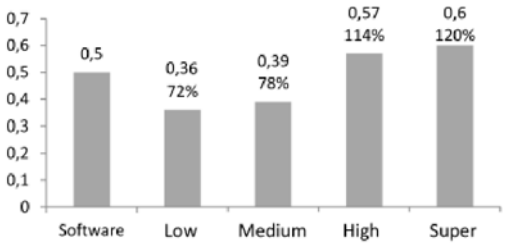


Figure 11. Rating of Stereo-FPS (SFPS).

Table 2. Original Software version vs. Best Hardware Strategy (SFPS: stereo frames per second)

Analysed Section	**Original ver. (SW exec.)**	**Super Speed ver. (3 HW RUs)**
Feature Extraction Execution Task	~650 ms	~475 ms (-26.7%)
Stereo Image Process Time	~2018 ms	~1680 ms (-16.7%)
Stereo Image Rate	~0.50 SFPS	~0.60 SFPS (+20%)

using hardware DPR compared with original software-only execution times. In software, feature extraction takes 650 ms; the best version with hardware support runs in 475 ms, which correspond to a reduction of 26.7 % (speedup of 1.37). This task reduction fraction translates to a global execution time decrease for the whole aplication of 16.7 % from 2018 ms to 1680 ms (a speedup of 1.2). This time corresponds to the stereo image processing rate of Fig. 11.

The speedup obtained when operating over bigger images was also estimated. The original embedded application operated only over images with 320×240 pixels. For this resolution, the duration of every reconfiguration operation, data transfer and computation involved in the feature extraction task was measured. These values were used together with data from the desktop version of the application to estimate the speedup that would be obtained by the embedded version for images of size 640×480 was estimated. The desktop version takes 4.54 times longer on the larger images than on the smaller ones. Combining this information, we estimated the speedup of the embedded DPR system on larger images to be 2.5 (for the feature extraction task).

7. Conclusions

The design and implementation of a DPR embedded system has been successfully completed. The implementation of an FPGA-autonomous navigation algorithm which dynamically adapts to system requirements was successfully achieved. Using DPR techniques, an effective execution time improvement of the Stereo Navigation application was achieved. The experience acquired in modifying

the application for dynamic reconfiguration will be used to devise automatic methods for carrying them out using the LARA toolchain [11].

The tradeoff between bitstream size and number of reconfiguration operations is the critical aspect of reconfigurable real-time embedded systems. Lowering both can offer great acceleration solutions, but this might not always be possible. The study of the application of DPR strategies to the feature extraction task shows that the nature of this particular task is not the most appropriate for reconfiguration, because each hardware-accelerated subtask has a significant reconfiguration time (proportional to the bitstream size). In addition, the number of sub-task executions per image is high, leading to a high number of reconfigurations when using one or two RUs. Enhancing the reconfiguration process, particularly the HwICAP communication interface, together with a more optimized synthesis of the reconfigurable modules to obtain resulting smaller bitstreams, has the potential to offer even better system performance.

Acknowledgments This work was partially funded by the European Regional Development Fund through the COMPETE Programme (Operational Programme for Competitiveness) and by national funds from the FCT–Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-022701. The authors are thankful for support from the European Community's Framework Programme 7 under contract No. 248976.

References

[1] Pao-Ann Hsiung, Marco D. Santambrogio, and Chun-Hsian Huang. *Reconfigurable System Design and Verification*. CRC Press, February 2009.

[2] Xilinx. *PlanAhead Software Tutorial, Design Analysis And Floorplanning for Performance*. Xilinx Inc, September 2010.

[3] J.P. Delahaye, C. Moy, P. Leray, and J. Palicot. Managing Dynamic Partial Reconfiguration on Heterogeneous SDR Platforms. In *SDR Forum Technical Conference*, volume 5, 2005.

[4] Z. Yu, I. Warren, and B. MacDonald. Dynamic Reconfiguration for Robot Software. In *2006 IEEE International Conference on Automation Science and Engineering. CASE'06.*, pages 292–297. IEEE, 2006.

[5] C. Harris and M. Stephens. A Combined Corner and Edge detector. In *Alvey vision conference*, volume 15, page 50, 1988.

[6] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[7] João Teixeira. Acceleration of a Stereo Navigation Application for Autonomous Vehicles. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2011.

[8] REFLECT consortium. Deliverable 1.2 - Technical report of applications delivered by Honeywell. Technical report, REFLECT Project, October 2009.

[9] John Hauser. SoftFloat, June 2010. `http://www.jhauser.us/arithmetic/SoftFloat.html`.

[10] IEEE Standard for Floating-Point Arithmetic, 2008.

[11] J.M.P. Cardoso, T. Carvalho, J.G Coutinho, W. Luk, R. Nobre, P.C. Diniz, and Z. Petrov. LARA: An aspect-oriented programming language for embedded systems. In *Proc. Int. Conf. on Aspect-Oriented Software Development (AOSD'12)*, pages 179–190, March 2012.

 ISBN: 978-972-8822-27-9

IX Jornadas sobre Sistemas Reconfiguráveis
ISR-UC, 7-8 de Fevereiro de 2013

Organização

ISBN: 978-972-8822-27-9